

Grundlagen der Informatik – Teil 4

Objektorientierung



There is water on Mars

Zwischenstand

Block IV: Algorithmen & Datenstrukturen

Abstrakte
Datentypen

Algorithmen,
Rekursion

Suchen &
Sortieren

Dynamische
Daten

Block V: Objektorientiertheit

Objekt,
Klasse

Konstruktor

Überladen,
Überschreiben

Vererbung

Imperative Programmierung

- Wir möchten ein Problem lösen
- Schrittweise wird dieses Problem gelöst
 - ◆ Erster Schritt
 - ◆ Nächster Schritt N – Schritte
 - ◆ Letzter Schritt
- Problem:
 - Sicht auf die Daten, Datenkapselung nur bedingt vorhanden.
 - Modellierung geschieht bei komplexen Problemen außerhalb (vor) der Programmierung.

Vision: Realitätsnahe Modellierung

- Der Algorithmus selbst steht nicht mehr vordergründig im Mittelpunkt, ...
- ... sondern eine „Realitätsnahe“ Modellierung
- Die Abstraktion beim Programmieren ist näher am menschlichen Denken. **Damit:**
 - ◆ Der Programmierer kann sich mehr auf die Lösung des Problems konzentrieren,
 - weil er weniger programmieren muss
 - weil er sich weniger Gedanken um Syntax-Konstrukte etc.

JETZT!



Objekt-Orientierung ... pimp my Objects



(Achtung! Autovergleich)

- Das sind Autos...
- Verschiedene Autos...?
- *Gleiche Autoart in verschiedenen Farben/Typen?*
- *Autos = Klasse*



- Das sind Fahrzeug...



- Verschiedene Fahrzeuge...?

- *Fahrzeuge = Klasse*

- *Autos sind Fahrzeuge!*



- Das ist ein Chevrolet Imapla '67
- Ein besonderes Auto...?
- *Imapla67*
= *Objekt* ?



- *ist ein spezielles Objekt der Klasse Autos (oder der Klasse Fahrzeuge), das in der Realität ein schwarzes Auto vom Typ Imapla '67 darstellt*
- *... damit haben wir einen Ansatz anders zu programmieren
... und anders zu denken*

Klassen und Objekte

- Daten und nur die auf diesen Daten zulässigen Funktionalitäten bilden jeweils eine Einheit, die Klasse
- Die Klasse ist die Schablone, der Stempel – Das Objekt ist das, was "mit" bzw. "durch" die Klasse erzeugt wird

Kapselung

- Bei der Anlage von Daten und Funktionalitäten kann sehr fein modelliert werden, welche Daten nach außen sichtbar und welche Daten sogar modifizierbar sind

Vererbung

- Ähnliche Funktionalitäten müssen nicht erneut programmiert – sondern können voneinander abgeleitet werden

- Bei der imperativen Programmierung kann (erstmal) jedes Unterprogramm auf jede sichtbare Variable zugreifen, ...
- ... und sie sogar verändern
- → **Der Zugriff auf Variablen(-inhalte) sollte "besser" kontrollierbar sein**

- Bei der imperativen Programmierung muss ähnliche Funktionalität sehr häufig erneut implementiert werden
- → **Wiederverwendbarkeit von Quellcode muss noch über Unterprogrammkonzepte (oder noch schlimmer: Copy & Paste) hinausgehen**

Schnittstellen

- Daten und Funktionalität kommunizieren ausschließlich über Schnittstellen, Methoden und freigegebene (Daten-)Felder mit der Außenwelt
- Die Außenwelt (ein verwendendes Programm) kennt nur die Schnittstellen, keine Implementierungsdetails
- Dadurch **eigentlich** keine Seiteneffekte möglich, ...
- ... aber viele objekt-orientierte Programmiersprachen sind Fortentwicklungen von imperativen Sprachen ☹
 - ◆ Gegenbeispiel: Smalltalk

- Ok, erlernen wir das Programmier-Denken neu
- Was ist eigentlich ein Objekt?

- Duden:
(lat.) (s.) Ziel, Gegenstand
- Brockhaus:
Sache, Angelegenheit, Gegenstand
abgewandelte Bedeutung in der Grammatik und
Philosophie

- Dabei sind uns Klassen und Objekte nicht unbekannt
- vgl. Kapitel Abstrakte Datentypen (ADT)

- Intuitive Vorstellung von Objekten:
 - ◆ nicht nur Gegenstände
 - ◆ nicht nur anfassbares
 - ◆ sondern alles zueinander ähnliches
- **Beispiele:** Alle Autos, alle Kunden, alle Lieferanten, alle Aufträge, alle Vorlesungen, alle Prüfungen, etc.
- Wir müssen beschreiben, welche Daten und Funktionalitäten jeweils **alle xxx** haben soll
 - ◆ Dies erfolgt mittels der Klasse – sozusagen der Schablone
 - ◆ Jedes einzelne Objekt wird von dieser Klasse abgeleitet, instanziiert

Wiederholung:

- Die Klassenbeschreibung entspricht unserer .class-Datei
- Das instanzierte Objekt entspricht dem mit new Operator bzw. dem Konstruktor erzeugten Objekt
- Erinnerung: 1 Klasse == 1 Datei

Wichtig:

- Wie bei jeder Modellierung gilt auch hier:
- Die Klassenbeschreibung erhebt in Bezug auf die Realität niemals Anspruch auf Vollständigkeit, ...
- ... sondern bildet nur (mindestens) so viel ab, wie zur Lösung des Problems notwendig ist

Ein Objekt besteht aus drei Bestandteilen:

- **Typ des Objektes** – der Datentyp
- **Ausprägung, Wert, Zustand** – die Werte der Felder
 - ◆ Zusätzlich sind in der Klassenbeschreibung die auf diesem Objekt zulässigen Operationen definiert, ...
 - ◆ ... diese werden also **NICHT** im Objekt gespeichert!
- **Identität** – eine Objekt-ID, die die Unterscheidung Objekte gleichen Typs und gleicher Ausprägung trotzdem eindeutig ermöglicht
 - ◆ Die Referenz bzw. die Speicheradresse unterscheidet solche Objekte
 - ◆ Der Bezeichner **NICHT** – vgl. den Abschnitt über das Kopieren von Objekten: Zwei Bezeichner können ein und dasselbe Objekt referenzieren

- Wie modelliert man ein Fahrzeug?
- Zustand: (Attribute, Felder)
 - ◆ Anzahl Räder (Zahl)
 - ◆ Typ: (Auto, Laster, Zug ...)
 - ◆ Antriebsart (Otto/Diesel/Sonst)
 - ◆ Beschleunigung (einfach: Gleitkomma Zahl)
 - ◆ Bremswirkung (einfach: Gleitkomma Zahl)
- Funktionalität: (Methoden)
 - ◆ beschleunigen()
 - ◆ bremsen()
- Die Methoden sollen als parameterlose Prozeduren implementiert werden, die als Seiteneffekt eine Ausgabe auf der Konsole erzeugen

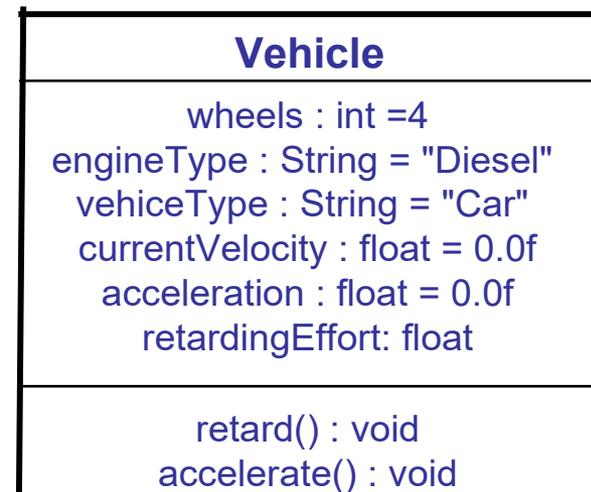
```

public class Vehicle {
    public int wheels=4;
    public String engineType = "Diesel", vehicleType ="Car";
    public double currentVelocity = 0.0;
    public float acceleration = 0.0f, retardingEffort;

    public void retard() {
        currentVelocity -= retardingEffort;
    }
    public void accelerate() {
        currentVelocity += acceleration;
    }
}
    
```

**Felder/
Attribute**

Methoden



Instanzierung

```
public class MeinTest {  
    public static void main(String[] args) {  
        Vehicle car = new Vehicle();  
        Vehicle truck = new Vehicle();
```

Feldzugriff

```
        truck.acceleration= 1.0f;  
        truck.vehicleType= "Truck";  
        truck.accelerate();
```

```
        car.vehicleType = "Car";  
        car.acceleration= 2.0f;  
        car.accelerate();
```

```
    }  
}
```

Methodenaufruf

- Instanziert wird ein Objekt durch die Benutzung des new Operators und dem Namen der Klasse
- Die Instanzierung ähnelt dem Aufruf einer Methode
- Häufig wird man – selbst bei vorgegebenen Defaultwerten – nach der Instanzierung dem Objekt neue Feldwerte zuweisen wollen
- Dann sollte dies doch einfach schon bei der Instanzierung möglich sein
- Hierzu dient der **Konstruktor** – uns auch schon bekannt
- Funktion (!), die heißt wie die Klasse, ohne Angabe des Rückgabetyps (= Objekt dieser Klasse)
- Übergabe von Parametern ist erlaubt

Kennen wir Schon: ADT

```
...  
public Vehicle() {  
    System.out.println("Entstehung eines neuen Fahrzeugs");  
}
```

**Aufruf
unverändert**

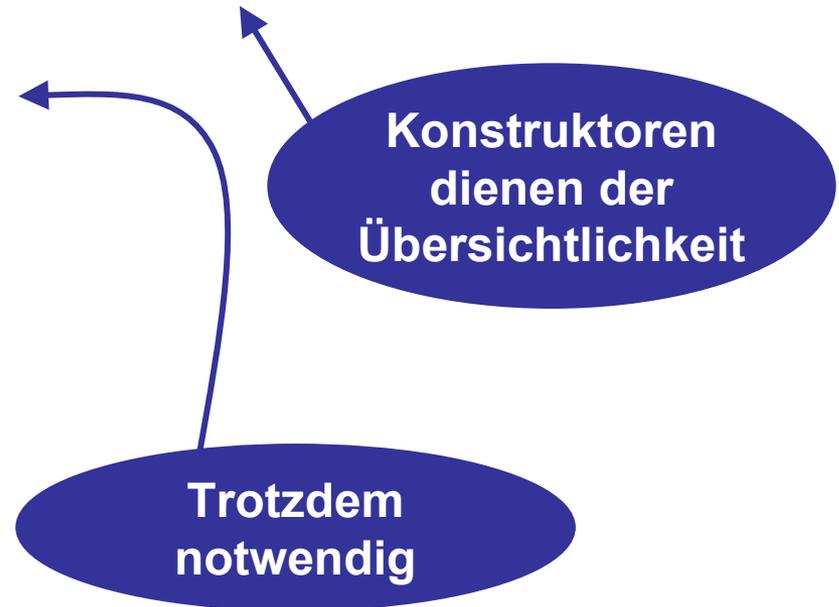
■ Oder mit Parameter(n):

```
...  
public Vehicle(String aVehicleType, float aAcceleration) {  
    vehicleType = aVehicleType;  
    acceleration = aAcceleration;  
}
```

**Aufruf
anpassen!**

- Ausgehend davon, wir hätten den zweiten Konstruktor implementiert:

```
public class MeinTest {  
    public static void main(String[] args) {  
        Vehicle car = new Vehicle( "Car",2.0d);  
        Vehicle truck = new Vehicle( "Truck",1.0d);  
  
        truck.retardingEffort= 1.0f;  
        car.retardingEffort= 1.5f;  
  
        truck.accelerate();  
        car.accelerate();  
    }  
}
```



- Nochmal **Übersichtlichkeit** – Hätten wir nicht, um u.a. Schreibarbeit zu sparen, besser folgendes implementiert?
(Man beachte die Bezeichner der formalen Parameter)

```
public class Vehicle {  
    public String vehicleType;  
    ...  
}
```

```
public Vehicle(String vehicleType) {  
    vehicleType = vehicleType;  
}  
}
```

Funktioniert nicht !!!

**Erinnerung:
Sichtbarkeitsregeln**



- Intuitiv ist klar, was gewünscht war – semantisch ist es eine (sinnlose) Selbstzuweisung
- Das Objekt ist zwar anonym –
 - ◆ es ist weder unter `car` noch `truck` ansprechbar, da diese Bezeichner nur in `main()` sichtbar sind, ...
 - ◆ ... trotzdem muss ja bereits eine Referenz existieren, da ansonsten ja nicht Feldern Werte zugewiesen werden könnten
- Ganz so anonym ist das Objekt nicht – es existiert ein Operator zur Ermittlung dieser Referenz: **this**

- Also doch übersichtlich möglich:

```
public class Vehicle {  
    public String vehicleType = "Vehicle";  
    ...  
  
    public Vehicle(String vehicleType) {  
        this.vehicleType = vehicleType;  
    }  
}
```

- this verhindert hier den Namenskonflikt
- implizit war aber auch vorher schon immer this gemeint
- wir durften es nur weglassen, weil der Bezug eindeutig war

- this ist ein Java-Schlüsselwort
- Die konsequente Angabe von this gilt als besserer Stil
- this ist in der gesamten Klasse sichtbar
- this ist nicht auf die Verwendung zum Zugriff auf Felder begrenzt – mit this können auch Methoden (dieses Objektes) aufgerufen werden
- Damit ist beispielsweise auch der folgende Konstruktor denkbar:

```
public Vehicle() {  
    System.out.println("Ein neues Fahrzeug");  
    this.accelerate();  
}
```



Methodenaufruf

- Bei dem Beispiel `car` mussten wir trotz Konstruktor noch die `retardingEffort` nachträglich setzen
- Bequemer wäre doch, man hätte die "Auswahl" zwischen verschiedenen Konstruktoren je nach "vorhandenen" Werten ...
- ... bzw. allgemeiner: nach unterschiedlichen Vorgaben, da im Konstruktor ja (offensichtlich) Methodenaufrufe möglich sind
- Aber wie bezeichnen wir dann die unterschiedlichen Konstruktoren?
- Richtig: Die Definition einer Methode besteht aus Bezeichner **und** Parameterliste, der Signatur

- Bei der Parameterliste sind allerdings nur die Datentypen und deren Reihenfolge relevant ...
- ... **NICHT** die Bezeichner
- Damit sind mehrere Konstruktoren mit dem Bezeichner Fahrzeug möglich, sofern sie
 - ◆ eine unterschiedliche Anzahl Parameter besitzen bzw. bei gleicher Parameteranzahl
 - ◆ sich in der geordneten Parameterliste die Datentypen im paarweisen Vergleich mindestens an einer Stelle unterscheiden
- Dadurch weiß dann der Compiler, welcher Konstruktor aufgerufen werden soll

- Eine Verletzung dieser "Eindeutigkeitsregeln" wird ebenfalls vom Compiler erkannt und mit einem Syntaxfehler quittiert

Beispiele:

- Vehicle() und Vehicle(String x)
 - ◆ ok, unterschiedliche Signaturen
- Vehicle(String x) und Vehicle(int x)
 - ◆ ok, unterschiedliche Signaturen
- Vehicle(String x, int y) und Vehicle(int y, String x)
 - ◆ ok, unterschiedliche Signaturen (Reihenfolge!)
- Vehicle(String x, int y) und Vehicle(String y, int x)
 - ◆ **Syntaxfehler**, trotz unterschiedlicher Bezeichner identische Signaturen

Mehrere Konstruktoren – 4

```
public class Vehicle {  
    ... // Felder  
  
    public Vehicle(String vehicleType) {  
        this.vehicleType = vehicleType;  
    }  
  
    public Vehicle(String vehicleType,  
                    float acceleration, float retardEffort) {  
        this.vehicleType = vehicleType;  
        this.acceleration= acceleration;  
        this.retardEffort = retardEffort;  
    }  
  
    ... // Methoden  
}
```

Das haben wir doch schon getippt!

Mehrere Konstruktoren – 5

Erste Idee:

...

```
public Vehicle(String vehicleType,  
                float acceleration, float retardEffort) {  
    this.Vehicle(vehicleType)  
    this.acceleration = acceleration;  
    this.retardEffort = retardEffort;  
}
```

**Funktioniert
nicht !!!**



Zweite Idee:

...

```
public Vehicle(String vehicleType,  
                float acceleration, float retardEffort) {  
    this(vehicleType);  
    this.acceleration = acceleration;  
    this.retardEffort = retardEffort;  
}
```

**Konstruktoren sind
besondere Methoden**

...

- Damit haben wir drei "Verwendungsmöglichkeiten" für this kennen gelernt:
 - this ("ohne alles") liefert eine Referenz auf das bisher anonyme Objekt (da der zur Instanziierung gewählte Bezeichner in der Klasse nicht sichtbar ist)
 - this.bezeichner für den Zugriff auf Felder und (mit einer entsprechenden Parameterliste, geklammert) für den Zugriff auf Methoden
 - this() (mit geklammerter Parameterliste) für den Zugriff auf einen Konstruktor

Vorsicht *FALLE*



- Für die Verwendung von `this()` – Aufruf eines Konstruktors – gelten die folgenden Regeln:
 - `this()`-Aufrufe sind nur innerhalb von Konstruktoren erlaubt
 - Mit `this()`-Aufrufen sind nur andere Konstruktoren innerhalb derselben Klasse erreichbar
 - Der `this()`-Aufruf muss die erste Anweisung in einem Konstruktor sein
 - Innerhalb eines Konstruktor-Rumpfes ist nur ein `this()`-Aufruf zulässig

Merke!



Ziehen wir ein Zwischenfazit:

- Modellierung von Klasse/Objekt ist realitätsnäher als die "normale" imperative Programmierung!
- Bindung von Funktionalität an die Daten
- Umso größer das Software-Projekt, umso schwerer wiegen die Vorteile

Auch die "klassische" Programmierung profitiert hiervon:

- Betrachten wir die Implementierung von (grafischen) Benutzeroberflächen

Merke!



- Imperativ: Menü darstellen, in einer "Endlosschleife" abfragen, welche Bedienelemente ausgelöst wurden, um die entsprechenden Funktionalitäten auszulösen
- Objekt-orientiert: Menü darstellen, die Bedienelemente werden in ein bestehendes Objekt zur Behandlung der Bedienoberfläche eingebettet
 - ◆ **Großer Vorteil 👍: Der Programmierer muss sich weder um die Erstellung noch um die Verwaltung der "Endlosschleife" kümmern**
 - ◆ Er legt die Ausprägung (z.B. Größe und die Beschriftung bei Knöpfen) der Bedienelemente fest
 - ◆ Er implementiert für jedes Bedienelement die Funktionalität (z.B. bei einem Knopf was geschehen soll, wenn er gedrückt wird – fertig!)

- Was tun, wenn man nicht nur maximal beschleunigen (accelerate) möchte sondern zusätzlich auch noch um eine gewisse Beschleunigung beschleunigen möchte?
- Erste Idee: acceleration merken; acceleration auf seine Wert setzen, accelerate() aufrufen dann acceleration auf alten Wert (im Hauptprogramm).
Nicht wirklich Objektorientiert. Sehr unsauber!!
- Zweite Idee: Implementierung von
public void accelerate2(float acceleration)
 - ◆ Lösung ist in Ordnung, aber tipp-intensiv
 - ◆ Geht es nicht *genauso* wie bei den Konstruktoren?

- Es war möglich, je nach Einsatzzweck (bei Konstruktoren: unterschiedliche Vorgaben bei der Instanziierung des Objektes) unterschiedliche Konstruktoren aufzurufen
- Damit der Compiler weiß, welche Methode er im "Zweifelsfall" aufzurufen hat, müssen die Signaturen von Methoden unterschiedlich sein
- Die Signatur besteht aus Bezeichner **UND** Datentypen der Liste der formalen Parameter gemäß ihrer Anordnung
- Das gilt allgemein für Methoden. Damit sind
`public void accelerate()` und
`public void accelerate(float acceleration)`
für den Compiler unterschiedliche Methoden!

- Diesen Mechanismus bezeichnet man als Überladen (von Methoden)
- Es gelten die gleichen Regeln für die Signatur wie bereits im Abschnitt mehrere Konstruktoren angegeben
- Überladen ist fehleranfällig!
 - ◆ Erweiterung von Methoden durch zusätzliche Parameter führt ggf. zu "doppelt vorhandenem" und damit doppelt zu pflegendem Code
 - ◆ Durch die Parameterliste sollte intuitiv die Funktion dieser Methode und die Abgrenzung zu anderen Methoden mit gleichem Bezeichner klargestellt sein
- Damit ergibt sich unsere Fahrzeugklasse zu:

...

```
public void accelerate() {  
    velocity += acceleration  
}
```

```
public void accelerate(float acceleration) {  
    if (this.acceleration=>acceleration)  
        velocity += acceleration  
    else  
        System.out.print("Hups! Motorschaden");  
}
```

...

Vorsicht *FALLE*

- Vorsicht bei impliziter Datentypumwandlung!
- Gegeben eine Klasse MyExample mit den Methoden
public void myMethod(int x) und
public void myMethod(double x)
- Welche Methode wird jeweils aufgerufen, bei:
MyExample some = new MyExample();
 - ◆ Aufruf some.myMethod(8) ?
→ public void myMethod(int x)
 - ◆ Aufruf some.myMethod(8.0) ?
→ public void myMethod(double x)
 - ◆ Aufruf some.myMethod(Math.pow(2,3)) ?
→ public void myMethod(double x)

- Wir kennen bereits überladene Methoden in Java:
 - ◆ `System.out.println()`
 - ◆ `System.out.print()`
- Wir kennen sogar einen überladenen Operator in Java
 - ◆ das Pluszeichen (+)
 - ◆ "normalerweise" die arithmetische Addition
 - ◆ ist aber wenigstens einer der beiden Parameter eine Zeichenkette, so ändert die Operation ihre Funktionalität zur Verkettung von Strings
- Viele objekt-orientierte Programmiersprachen unterstützen das Überladen von Operatoren
 - ◆ u.a. auch der "Vorgänger" C++
 - ◆ **Java nicht!**

- Betrachten wir abschließend noch Mal unsere Notation der Klasse Vehicle bis hierher:

Vehicle

wheels : int = 4
vehicleType : String = "Vehicle"
currentVelocity : float = 0.0f
acceleration : float = 0.0f
retardingEffort: float

Vehicle(String vehicleType) : Vehicle
Vehicle(String vehicleType, float acceleration, retardingEffort): Vehicle

retard() : void
accelerate() : void
accelerate(float acceleration) : void

Schützen von Eigenschaften

- Die Geschwindigkeit des Trucks könnte sich mit der Zeit ändern..
- ... der Bremswirkung ggf. auch.
- ... aber auf keinen Fall wird sich die Anzahl der Räder eines einmal instanziierten Fahrzeugs ändern (ohne massiven Umbau, oder Unfall ...)

Idee:

- Schön wäre ein Mechanismus, der den Zugriff auf die Felder besser steuert ...
- ... und damit Einfluss auf die Sichtbarkeit nimmt

- Dieser Mechanismus wird durch Modifikatoren geregelt
- Wir kennen bereits einen Modifikator: **public**
- Modifikatoren wirken auf
 - ◆ Felder und Methoden (betrachten wir jetzt)
 - ◆ aber auch auf Klassen! (*später ...*)
- **public** bedeutet die Sichtbarkeit (und damit lesenden und schreibenden Zugriff für Felder) in allen verwendenden Klassen
- Ein weiterer Modifikator ist **private**:
 - ◆ Mit **private** deklarierte Felder und Methoden sind nur innerhalb der eigenen Klasse sichtbar
 - ◆ Weder schreibender noch lesender Zugriff
 - ◆ Felder werden damit vor dem Zugriff aus anderen Klassen geschützt
 - ◆ Methoden, die ausschließlich innerhalb der eigenen Klasse verwendet werden, sollten ebenfalls als **private** deklariert werden – Warum?

- Schützen wir das Feld engineType:

```
public class Vehicle {  
    private int wheels;  
    private String engineType = "Diesel";  
    private String vehicleType = "Vehicle";  
    private double currentVelocity = 0.0;  
    public float acceleration = 0.0f, retardingEffort ;  
  
    public void retard() {  
        currentVelocity -= retardingEffort;  
    }  
    public void accelerate() {  
        currentVelocity += acceleration;  
    }  
}
```

- Betrachten wir jetzt wieder die Test-Umgebung:

```
public class MeinTest {  
    public static void main(String[] args) {  
        Vehicle car = new Vehicle(  
                                "Car",2.0f,1.0f);  
        Vehicle truck = new Vehicle(  
                                "Truck",2.0f,1.0f);  
  
        truck.retardingEffort=2.0f;  
        // so weit, so gut, ABER:  
        car.vehicleType = "Ich bin ein Zug";  
    }  
}
```

indirekter Zugriff
weiterhin möglich!

Syntaxfehler !!!

- "indirekter Zugriff" = schreibender bzw. lesender Zugriff auf private Felder mit public Methoden außerhalb der eigenen Klasse
- Dadurch lässt sich der Zugriff auf die Felder eines Objektes sehr fein modellieren
- Man könnte z.B. ALLE Felder als private deklarieren, ...
 - ◆ ... für den lesenden Zugriff werden deklariert:
public String getVelocity()
public String getWheels()
public String getEngineType()
 - ◆ ... und für den schreibenden Zugriff:
public void setEngineType(String name)
public void accelerate() [**schon bekannt**]

Vorteil 👍:

- Sowohl der lesende aber umso mehr der schreibende Zugriff kann sehr fein modelliert werden in Abhängigkeit von Gegebenheiten (wie z.B. anderen Feldwerten) ...
- ... und so den Zugriff durchführen, abweisen oder sinnvoll modifizieren
- Einige "reine" objekt-orientierten Programmiersprachen verbieten von vornherein den direkten Zugriff auf die Felder eines Objektes von außerhalb der Klasse!
- Man spricht auch von dem **Botschaftenkonzept**:
 - ◆ Alles – auch das Hauptprogramm – ist ein Objekt
 - ◆ Objekte kommunizieren ausschließlich über Methoden (den Botschaften) miteinander

- Wir kennen schon weitere Modifikatoren:
 - ◆ **final**: als final deklarierte Variablen können nach der Initialisierung nicht mehr verändert werden, es sind Konstanten
 - ◆ **static**: als static deklarierte Methoden und Variablen können unabhängig von der Existenz eines Objektes der Klasse benutzt werden
- Die (Start-) Methode main() ist als static deklariert, wir mussten kein Objekt erzeugen
- Beim Modifikator final zu erkennen: Modifikatoren können bei Bedarf kombiniert werden:
`static final int MAX_ANZAHL = 100;`

- public und private widersprechen sich in ihrer Bedeutung
→ können also nicht miteinander kombiniert werden!
- private erlaubt unbegrenzte Schreibzugriffe, aber eben nur aus der eigenen Klasse heraus
- Die konsequente Verwendung von ausschließlich private deklarierten Feldern und den entsprechend notwendigen getBezeichner() bzw. setBezeichner() Methoden gilt als besserer Programmierstil
- Der Unterschied von private und public hat Auswirkungen auf unsere Notation

- Öffentliche Felder und Methoden (also mit public deklariert) wird ein + vorangestellt
- Privaten Feldern und Methoden (also mit private deklariert) wird ein – vorangestellt
- für MotorType, anzahlRäder und Geschwindigkeit benötigen wir Methoden für den lesenden Zugriff, da diese nicht sichtbar sind.

Vehicle	
-	wheels : int = 4
-	vehicleType : String = "Vehicle"
-	currentVelocity : float = 0.0f
+	acceleration : float = 0.0f
+	retardingEffort: float
+	Vehicle(String vehicleType): Vehicle
+	Vehicle(String vehicleType, float acceleration, retardingEffort): Vehicle
+	retard() : void
+	accelerate() : void
+	accelerate(float acceleration) : void

- Im letzten Beispiel haben wir *currentVelocity* als *private* markiert.
- Vorteil: so kann nur noch über definierte Methoden (*accelerate* und *retard*) zugegriffen werden.
- Nachteil: wir können den Wert der Geschwindigkeit außerhalb der Klasse nicht mehr auslesen
- Lösung: Wir schreiben eine eigene Funktion (*public getVelocity*), welche die aktuelle Geschwindigkeit zurückliefert:

```
public float getVelocity() {  
    return this.currentVelocity;  
}
```

- Genauso können wir eine Methode zum setzen der Geschwindigkeit schreiben:

```
public void setVelocity(float velocity) {  
    if (velocity<200) {  
        this.currentVelocity=velocity;  
    }  
}
```

- Vorteil: so kann man zusätzlich Prüfungen und Programmlogik erzwingen. `currentVelocity` kann ja nicht (da *private*) direkt angesprochen werden.
- Eigentlich sind *accelerate* und *retard* vom Verhalten her auch *setter-Methoden*.
- In Java schreibt man für Methoden die einen Attribut direkt setzen: `setAttributname` für lesen `getAttributname`

- Betrachten wir den Modifikator `static` noch einmal genauer
- `static` deklarierte Felder und Methoden existieren bereits ohne Instanzierung – wie unser Hauptprogramm
- Ohne Instanzierung bedeutet aber auch, dass sie nur ein Mal für die Klasse existieren
- Bedeutet weiterhin – für alle Objekte dieser Klasse existieren die `static` deklarierten Felder oder Methoden ebenfalls nur ein Mal
- Zum Verständnis betrachten wir das folgende Beispiel:

Problem:

- Unsere Klasse Vehicle soll dahingehend erweitert werden, dass die Anzahl bisher erzeugter Fahrzeuge mitgezählt werden soll

Erste Idee:

```
public class Vehicle {  
    public int count = 0;  
    // weitere Felder hier ...  
  
    public void Vehicle(String vehicleType) {  
        this.vehicleType = vehicleType;  
  
        count++;  
    }  
  
    // weitere Konstruktoren und die Methoden hier ...  
}
```

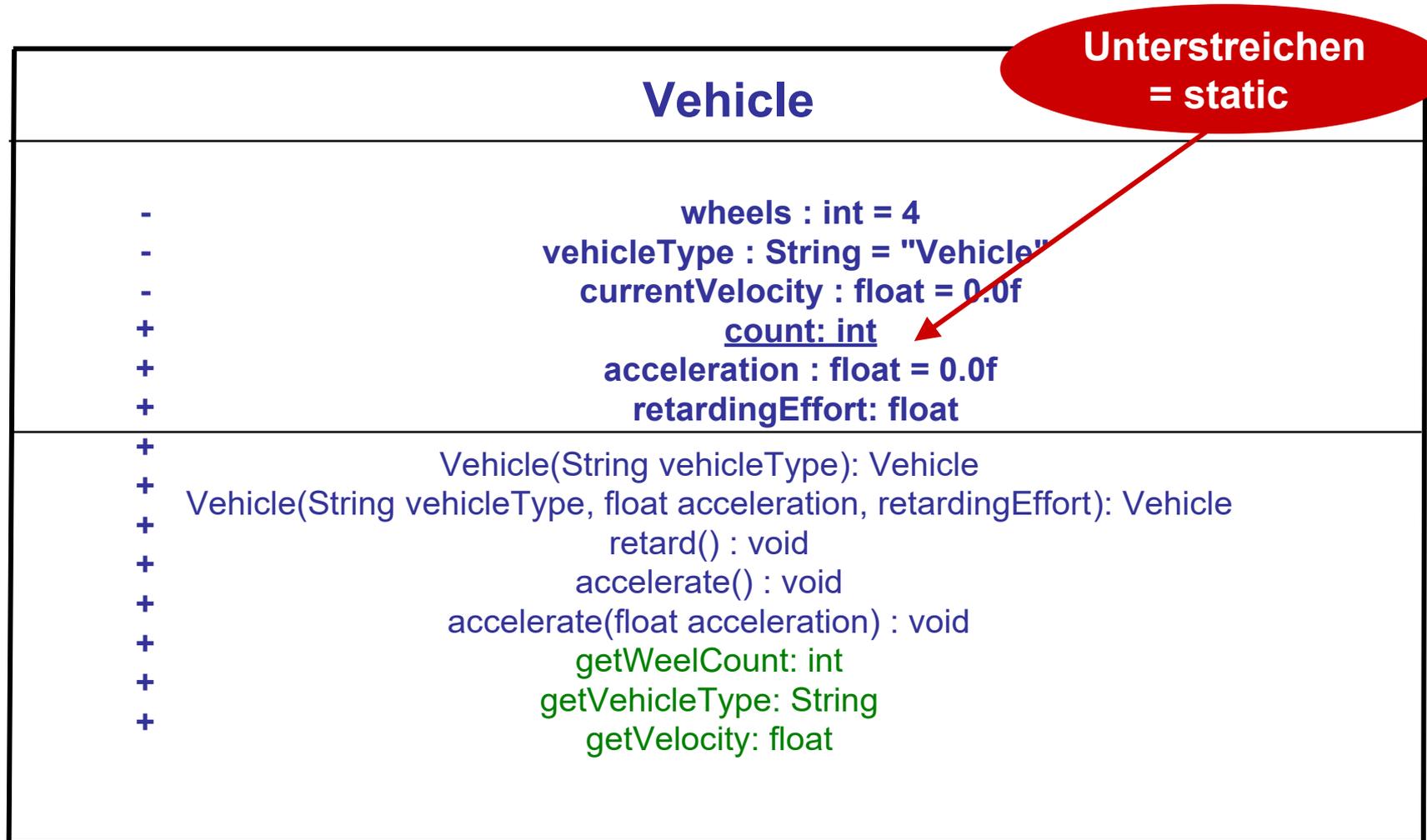


Funktioniert nicht !!!

- anzahl hat für jedes Fahrzeug den Wert 1
- Wir brauchen die Anzahl genau ein Mal – deklarieren wir sie doch als static – **zweite Idee:**

```
public class Vehicle {  
    public static int count = 0;  
    // weitere Felder hier ...  
  
    public Vehicle(String vehicleType) {  
        this.vehicleType = vehicleType;  
  
        count++;  
    }  
  
    // weitere Konstruktoren und die Methoden hier ...  
}
```

static – im Klassendiagramm



- Nur ein Mal vorhanden – daher spricht man bei static deklarierten ...
 - ◆ ... Feldern von **Klassenvariablen**
 - ◆ ... Methoden von **Klassenmethoden**

- Ein Klassenmethode hat nur Zugriff (neben ihren lokalen Variablen) auf Klassenvariablen und andere Klassenmethoden
- Die Verwendung von this ist innerhalb einer Klassenmethode ebenfalls nicht erlaubt

- Zur Abgrenzung – nicht-static deklarierte ...
 - ◆ ... Felder bezeichnet man auch als **Instanzvariablen**
 - ◆ ... Methoden bezeichnet man auch als **Instanzmethoden**

- Zur Diskussion:
Welchen Sinn macht die Verwendung von Klassenmethoden?

- Der Zugriff auf static deklarierte Felder innerhalb der eigenen Klasse erfolgt (ohne this) durch Angabe des Bezeichners
- dito für static deklarierte Methoden
- Aber wie sieht der Zugriff von außerhalb der Klasse aus?
 - ◆ Aufgrund der Sichtbarkeitsregeln kann nicht ein Feld mit gleichem Bezeichner sowohl static und nicht-static deklariert sein
 - ◆ Also es kann nicht gleichzeitig eine Klassenvariable und Instanzvariable mit gleichem Bezeichner geben
 - ◆ Aufruf innerhalb class MeinTest

```
...  
x = meinFahrzeug.count;
```



Führt zu einer
Warnung

- Die Warnung empfiehlt eine Klassenvariable auch "als solche" aufzurufen
- Was ist z.B. wenn gar keine Instanz der Klasse existiert – und damit über diesen "Umweg" auch nicht ansprechbar wäre?
- Klassenvariablen gehören zu der **Klasse** – andere Idee:

```
class MeinTest {  
    Vehicle car = new Vehicle("Auto");  
    Vehicle train = new Vehicle("Zug");  
  
    train.acceleration = 2.0f;  
    System.out.println("Anzahl bisher instanzierter Fahrzeuge: " +  
        Vehicle.count);  
}
```

Aufruf über den
Namen der Klasse

- Klassenmethoden werden außerhalb ihrer Klasse genauso über den Bezeichner der Klasse aufgerufen ...
- ... sofern sie nicht zusätzlich als `private` deklariert wurden
- Aber das würde ja genauso gelten für eine mit `private` deklarierte Klassenvariable – sinnvolle Kombination:
 - ◆ Klassenvariable `private`
 - ◆ lesender/schreibender Zugriff auf Klassenvariablen mittels `public` deklariierter Klassenmethoden
- **Zur Diskussion:**

Was passiert mit unserem Zähler `anzahl`, wenn ein Fahrzeug in einem Unterprogramm in der Klasse `MeinTest` als lokale Variable erzeugt und am Ende des Unterprogramms wieder vergessen wird?

- Aus semantischer Sicht wäre die Ausgabe der Anzahl der bisher instanziierten Fahrzeuge ein guter Kandidat für eine Klassenmethode

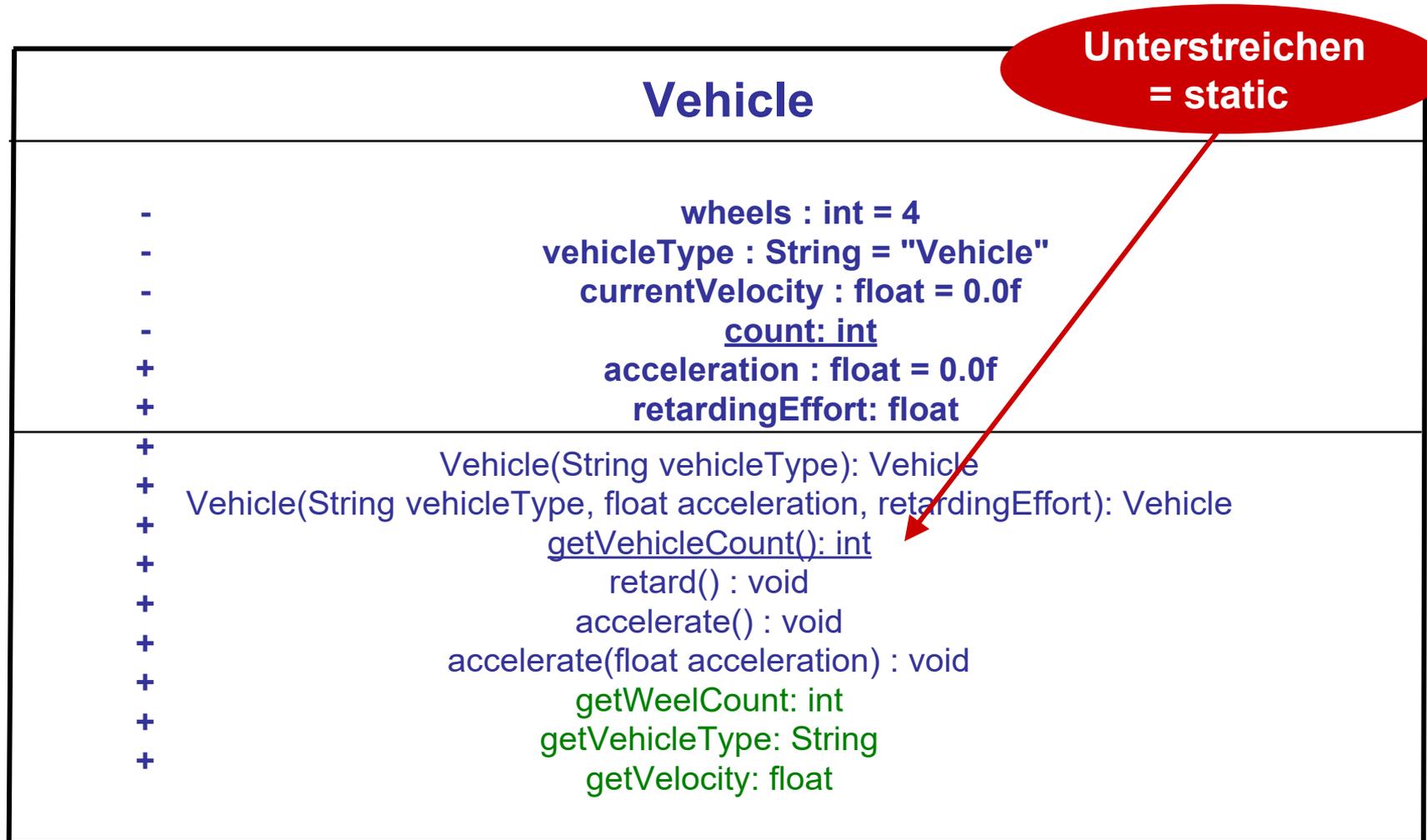
Beispiel:

```
public static int getVehicleCount() {  
    System.out.print("Anzahl bisher instanzierter Fahrzeuge: "  
+ anzahl);  
    return count;  
}
```

Aufruf:

- über die Instanzvariable:
 meinAuto.getVehicleCount();
- oder (besser:) über den Klassenbezeichner:
 Vehicle.getVehicleCount();

static – im Klassendiagramm



- Offensichtlich zählt der Zähler `anzahl` nur hoch – aber nicht wieder runter
- Schöner wäre es, wenn `anzahl` nur die Anzahl "zur Zeit existenter" Fahrzeuge wiedergeben würde
- Wann wäre der geeignete "Punkt", die Anzahl wieder zu verringern?
 - ◆ Am Ende der Sichtbarkeit
 - ◆ Dazu benötigen wir ein Gegenstück zum Konstruktor – den **Destruktor**
- Der Destruktor wird in der Objekt-Orientierung
 - ◆ neben der Information über das Erlöschen der Sichtbarkeit einer Objektinstanz an die Klasse
 - ◆ zum (Speicher-)Aufräumen für das Objekt am Ende seiner Sichtbarkeit benutzt

- (Speicher-)Aufräumen ist in Java ja nicht nötig
- Daher **kennt Java auch keinen Destruktor** im Sinne der Objekt-Orientierung!
- In der Literatur wird (**fälschlicherweise**) häufig als Destruktor ein Mechanismus angegeben, der aber erst am Ende der Lebensdauer des Objektes aufgerufen wird
- Um den zu verstehen, müssen wir vorher noch ein anderes objekt-orientiertes Konzept studieren
- Auf jeden Fall gilt für Java:
 - ◆ Ende der Lebensdauer = Zeitpunkt der Zerstörung des Objektes durch den Garbage Collector (GC) in Java damit
 - ◆ Ende der Lebensdauer \neq Ende der Sichtbarkeit – bzw.: Der GC wird vielleicht erst am Ende des Programms ausgelöst!
- Unser eigentliches Problem wäre damit immer noch nicht gelöst – da evtl. nicht mehr zugreifbare Fahrzeug-Objekte trotzdem noch nicht vom GC freigegeben wurden



- *Das ist ein Fahrzeug*
- *Außerdem ist es eine **Motorrad***
- *Das **Motorrad** ist ein Fahrzeug (**Vehicle**), und hat viele "Eigenschaften" mit **Vehicle** gemein...*
- *... aber in anderen "Eigenschaften" unterscheidet sie sich von anderen Fahrzeugen.*

- Auch eine Harley hat einen Motor, eine Farbe, kann beschleunigen und bremsen...
- ... die Harley ist außerdem geringfügig "anders" als ein Vehicle. (z.B. nur zwei Räder!)
- Trotzdem wäre es doch schön wenn wir nicht alles neu programmieren müssten
- Dazu kennt die Objekt-Orientierung ein Konzept, um eine Klasse aus einer anderen, bestehenden Klasse abzuleiten – die Vererbung
- Dabei überträgt die so genannte Basisklasse (in unserem Beispiel die Klasse Vehicle) *erstmal* alle Felder und Methoden an die so genannte Spezialisierung (in unserem Beispiel die Klasse Motorrad)
- Mit einer Einschränkung: private deklarierte Felder und Methoden! *später ...*

- Gegeben sei die folgende Vehicle-Klasse:

```
public class Vehicle {  
  
    // Felder  
    public String vehicleType  
    public static int anzahl = 0;  
    // .... weitere Felder  
    // Konstruktoren  
    public Vehicle() {  
        anzahl++;  
    }  
    public Vehicle(String vehicleType) {  
        this.vehicleType = vehicleType;  
  
        anzahl++;  
    }  
}
```

```
public Vehicle(String vehicleType, boolean honk) {  
    this(vehicleType);  
    if (honk) {  
        this.hupen();  
    }  
}
```

// Methoden

```
public void hupen() {  
    System.out.println("Tutut");  
}
```

```
public void bremsen(float toVelocity) {  
  
    while (toVelocity>velocity) {  
        velocity=velocity-retardEffort;  
    }  
    System.out.println("Speed:"+velocity);  
}  
}
```

- Weiterhin wichtig: Konstruktoren werden nicht vererbt!
- Sie müssen also bei Bedarf für jede Klasse neu implementiert werden
- Weiter im Beispiel – Ein Motorrad sei ein Fahrzeug, der aber
 - zusätzlich eine Farbe besitzt und diese mit printFarbe() Ausgeben kann
 - sich durch eine Beschleunigung von 0.5 auszeichnet

- Lösen wir das Problem schrittweise:

```
public class Motorbike extends Vehicle {  
    String color="";  
    int wheels=2;  
    vehicleType= "Motorrad";  
    // Konstruktor  
    public Motorbike(String engineType) {  
        this.acceleration = 0.5f;  
        this.engineType= engineType;  
        this.hupen();  
    }  
    // Methode  
    public void printColor() {  
        System.out.println(this.color);  
    }  
}
```

Angabe der
Basisklasse



nicht in dieser
Klasse deklariert



Aufruf von hupen()
der Klasse Vehicle



- Fügen wir zusätzlich eine andere Methode für „hupen“ ein

```
■  
  
// Methoden  
public void hupen() {  
    System.out.println("Moep Moep");  
}  
}
```

- ... und zum Testen:

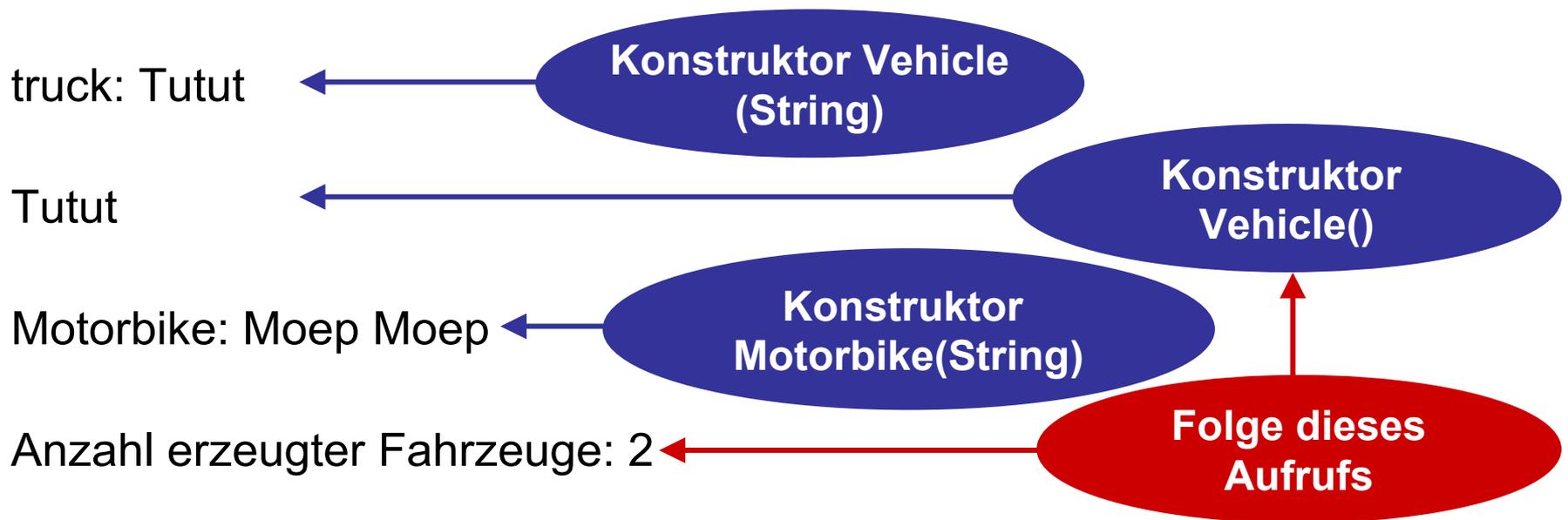
```
public class MeinTest {  
  
public static void main(String[] args) {
```

```
    Vehicle truck = new Vehicle("Diesel", 1.0f, 2.0f);  
    Motorbike HarleyRK = new Motorbike("gasoline engine");
```

```
    System.out.println("Anzahl erzeugte Fahrzeuge: " + Vehicle.count);  
}
```

- Welche Ausgabe wird produziert?





- Offensichtlich führt der Aufruf des Konstruktors für die Klasse Motorbike zum Aufruf des parameterlosen Konstruktors Vehicle
- Die Konstruktoren werden von "oben nach unten" ausgeführt, d.h. es wird zuerst der Konstruktor der Basisklasse ausgeführt und dann der Konstruktor der Subklasse
- Damit sind auch alle geerbten Felder richtig initialisiert!

- Damit stehen jedem Objekt vom Typ Motorbike die Bezeichner für die
 - ◆ Felder `acceleration` oder `color`
 - Methoden `hupen()`, `bremsen(float toVelocity)`
 - ◆ und neu: Methode `printColor()` zur Verfügung
- Lösen wir jetzt das zweite Problem: die Anzahl der Räder eines Motorrads
 - ◆ Erste Möglichkeit: Zuweisung `wheels = 2`; im Konstruktor der Klasse `Motorrad`
 - ◆ Zweite Möglichkeit: Deklaration des Feldes `int wheels = 2`; in der Klasse `Motorrad`

**Beides
funktioniert !**

- Die Funktionsweise der ersten Möglichkeit ist klar – während die zweite eine Neuerung birgt
- Offensichtlich wird ein neues Feld **wheels** oder eine Methode **hupen()** in der Klasse *Motorbike* deklariert, dass die in der Klasse *Vehicle* deklarierten Attribute/Methoden **überschreibt, verdeckt**
- Dabei hätten neue Attribute sogar einen anderen Datentyp erhalten dürfen
- Wichtig: Überschreiben \neq Überladen

```
public class Motorbike extends Vehicle {
```

```
    int wheels = 2;
```

überschriebenes
Feld



```
    // Konstruktor
```

```
    public Motorbike(String vehicleType) {
```

```
        ...
```

```
        this.hupen();
```

```
    }
```

Tutut
oder
MoepMoep?



```
    public void hupen() {
```

```
        System.out.println("Moep Moep");
```

```
    }
```

überschriebene
Methode



```
}
```

- Unsere Testklasse liefert jetzt folgende Ausgabe:
Truck: Tutut
Tutut
Motorbike: MoepMoep
Anzahl erzeugter Fahrzeuge: 2
- Also bereits im Konstruktor ist die in der Klasse Motorbike überschriebene Methode bekannt ...
- ... genauso wie in jeder Methode der Klasse Motorbike
- Zur Diskussion:
Der Begriff Überschreiben bedeutet dann wohl, dass "das Überschriebene" aus der Sicht eines Motorbike nicht mehr erreichbar ist, oder?

- Das Schlüsselwort **super** hat eine gewisse Ähnlichkeit mit dem Schlüsselwort **this**
- So können mit **super** z.B. überschriebene Methoden angesprochen werden, d.h.
 - ◆ **super** ist – wie **this** – eine Referenz auf die aktuelle Instanz ...
 - ◆ ... wobei mit **super** genau ein Element der Instanz referenziert wird, das vom Typ der unmittelbar übergeordneten Basisklasse ist
 - ◆ In unserem Beispiel würde eine Methode der Klasse **Motorbike**, die auf **super** zugreift eine Referenz auf dieses Objekt darstellen – allerdings vom Datentyp **Vehicle**!

- Angenommen wir wollen die Instanzen der Klasse „normal“ hupen lassen
- ... und dieses „normale“ Hupen klingt schon eher wie das eines "Fahrzeugs"

```
public void normalHupen() {  
    super.hupen();  
}
```

- Liefert die gewünschte Ausgabe **Motorbike: Tutut**
- Aber die Verwendung von super ist nicht nur auf überschriebene Methoden beschränkt, genauso können
 - ◆ überschriebene Attribute und
 - ◆ Konstruktoren der Basisklasse angesprochen werden

- Aber auch verdeckte Attribute können mittels `super` wieder zugreifbar gemacht werden
- In unserem Beispiel hat die Subklasse `Motorbike` das Attribut `engineType` der Klasse `Vehicle` überschrieben
- Semantisch wenig sinnvoll, aber dennoch möglich wäre die folgende Instanzmethode:

```
public String parentVehicleType() {  
    return( super.vehicleType );  
}
```



Zugriff auf das
Feld in der Basisklasse

- Der Zugriff auf verdeckte Felder erfolgt in der Praxis weitaus seltener als der Zugriff auf verdeckte Methoden
- Das liegt z.T. daran, dass man normalerweise durch das Verdecken einer Methode sie verfeinern will während durch das Überdecken eines Feldes eine komplett andere Semantik gesetzt werden soll

Beispiel:

```
Motorbike motorad = new Motorbike(); // Standardkonstruktor
```

```
...
```

```
System.out.println(motorad.vehicleType);
```

→ Zugriff auf `vehicleType` in Klasse **Motorbike**

→ Ausgabe: "Motorbike"

```
System.out.println(motorad.parentVehicleType());
```

→ Zugriff auf `vehicleType` in Klasse **Vehicle**

→ Ausgabe: "**Vehicle**"

- Die Methoden der Basisklasse kann man mit `super.Methodenbezeichner(Parameterliste);` aufgerufen werden
- In unserem Motorrad-Beispiel könnten wir uns doch so Tipparbeit in den Konstruktoren der Klasse Motorrad sparen

■ Erste Idee:

Im Konstruktor der Klasse Motobike rufen wir den Konstruktor der Klasse Motobike so auf:

```
public class Motobike {  
    // Felder ...  
    public Motobike(String color) {  
        super.Vehicle("Motobike", 2.0f, 1.5f);  
    }  
    ...  
}
```

Syntaxfehler !!!

- Zur Erinnerung: Konstruktoren sind besondere Methoden, klappte so ja auch bei this nicht
- Dann versuchen wir es (genau) so:

```
public class Motobike {  
    // Felder ...  
    public Motobike(String farbe) {  
        super(“Motorbike“, 2.0f, 1.5f);  
    }  
    ...  
}
```
- Das funktioniert!
- Also analog zu this erfolgt der Aufruf des Konstruktors ohne die Angabe des Bezeichners – also nur durch `super(Parameterliste)`;

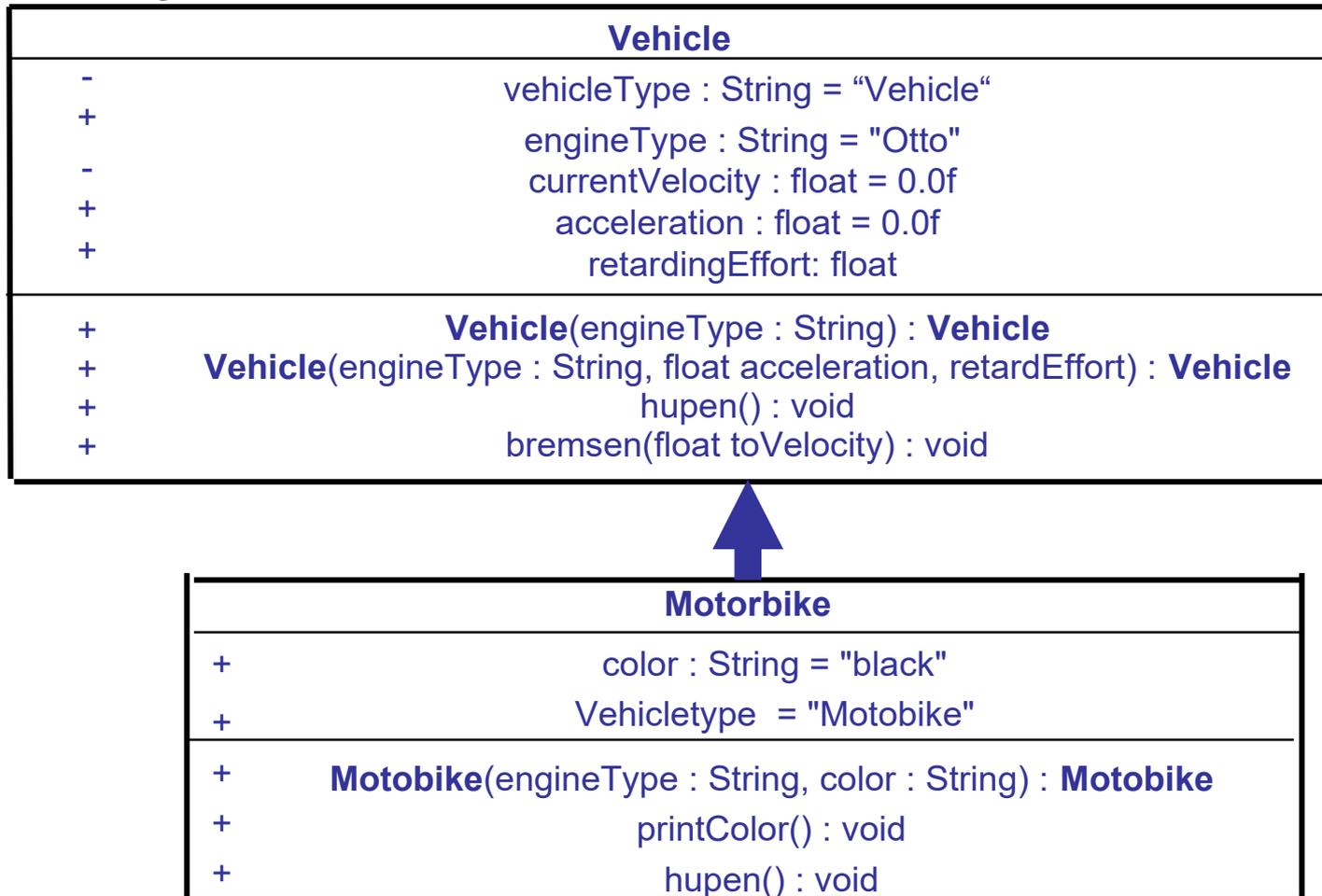
Vorsicht *FALLE!*



- Genau wie bei `this()` gibt es Regeln für die Verwendung von `super()`
 - `super()` darf nur in einem Konstruktor aufgerufen werden
 - `super()` muss die erste Anweisung im Rumpf sein – fehlt ein explizites `super()`, so wird implizit ein parameterloses `super()` aufgerufen
 - `super()` kann nur die Konstruktoren der direkt übergeordneten Basisklasse aufrufen
 - `this()` und `super()` dürfen nicht in einem Konstruktor stehen – ein `this()` verhindert sogar den impliziten `super()`-Aufruf

Notation Vererbung

- Vererbungsbeziehungen lassen sich auch grafisch darstellen
- Ein Pfeil zeigt von der Subklasse zur Basisklasse



- Wir hatten vor dem Abschnitt über die Vererbung das Feld *currentVelocity* als `private` deklariert, ...
- ... um so jegliche Zugriffe auf dieses Feld von außerhalb der Klasse auf das Attribut zu verhindern
- `private` **verhindert** aber auch die Vererbung des Feldes in Klasse `Motorbike`
- Offensichtlich gibt es einen semantischen Unterschied zwischen einer Klasse die eine andere Klasse "nur verwendet" durch Instanzieren ...
- ... und Klassen die andere Klassen verwendet durch Ableitung, also durch Vererbung

- Daher braucht man einen Modifikator, der
 - ◆ zwar bei der Instanziierung den Zugriff verhindert,
 - ◆ aber bei Vererbung diese Eigenschaft der Subklasse trotzdem zur Verfügung stellt
- Dieser Modifikator heißt **protected**

- Modifizieren wir noch ein Mal unsere Klasse Vehicle:

```
public class Vehicle {  
    public String engineType = "Diesel";  
    protected currentVelocity;  
    // Weitere Felder ...  
}
```

- Die Klasse **Motorbike** erbt durch das Schlüsselwort `extends` jetzt (trotzdem) das Feld **currentVelocity**
 - ◆ Als sei das Feld **currentVelocity** in der Klasse **Vehicle** `public` deklariert
- Die Klasse **MeinTest** instanziert lediglich Objekte der Klassen **Vehicle** und **Motorbike** ...
 - ◆ ... und hat damit keinen Zugriff auf das Feld **currentVelocity**
 - ◆ Als sei das Feld **currentVelocity** in der Klasse **Vehicle** `private` deklariert
 - ◆ Ggf. sind jetzt in der Klasse **Motorbike** Methoden für den Zugriff auf das Feld **currentVelocity** zu implementieren

- Die gleichzeitige Verwendung von public, private und protected als Modifikator für Methoden und Felder ...
 - ◆ ... macht keinen Sinn
 - ◆ ... und ist daher auch nicht erlaubt
- Zur Notation: Kennzeichnung von Feldern und Methoden
 - ◆ Modifikator public: +
 - ◆ Modifikator private: –
 - ◆ Modifikator protected: #

- Vererbung bedeutet mehr als "die Ersparnis von Tipparbeit"
- Bei der Vererbung gilt folgender Grundsatz:
Jede Instanz der Subklasse ist auch eine Instanz der Basisklasse
- Das birgt sehr mächtige Modellierungsmöglichkeiten
- Betrachten wir unsere Klasse Vehicle und die davon abgeleitete Klasse Motorrad
- Das bedeutet:
Jedes Objekt vom Typ Motorbike ist gleichzeitig auch ein Objekt vom Typ Vehicle

Dynamische Bindung – 2

```
public class MeinTest {  
    public static void main(String[] args) {  
        Vehicle truck = new Vehicle();  
        Vehicle meinMotorrad= new Motorbike();  
        Vehicle irgendeinFahrzeug;  
  
        irgendeinFahrzeug= truck;  
        irgendeinFahrzeug.hupen();  
  
        irgendeinFahrzeug = meinMotorrad;  
        irgendeinFahrzeug.hupen();  
    }  
}
```

**Ausgabe:
Tutut**

**Ausgabe:
Moep-Moep**

Dynamische Bindung – 3

```
public class MeinTest {  
    public static void main(String[] args) {  
        Vehicle truck = new Vehicle();  
        Motorbike meineHarley = new Motorbike();  
        Vehicle irgendeinFahrzeug;  
        int i = 0;  
  
        i = ... // Integerwert von der Tastatur einlesen  
        if (i == 3)  
            irgendeinFahrzeug = meineHarley;  
        else  
            irgendeinFahrzeug = truck;  
  
        irgendeinFahrzeug.hupen();  
    }  
}
```

Woher weiß der
Compiler das?

Tutut oder
Moep-Moep

- Je nach zugewiesenem Objekt wird entweder "Tutut" oder "Moep-Moep" ausgegeben
- Zur Übersetzungszeit kann der Compiler dies **NICHT** wissen:
 - ◆ Es hängt ja von der Benutzereingabe zur Laufzeit ab
 - ◆ Der Compiler kann also den Aufruf von `irgendeinFahreug.hupen()`; nicht statisch in das Programm binden (wie es bei "rein prozeduralen" Programmiersprachen üblich ist)
 - ◆ Erst zur Laufzeit erfolgt die (jeweilige) Bindung des Aufrufes an die (entsprechende) Methode – daher spricht man von **dynamischer Bindung**

- Das Objekt irgendeinFahrzeug ist zwar mit dem Datentyp **Vehicle** deklariert, kann aber so auch *irgendwie* den Datentyp **Motorbike** annehmen
- Man spricht hier auch von Vielgestaltigkeit oder von (gr.) **Polymorphie**
- Polymorphie ist ein weiteres, sehr wichtiges Konzept objekt-orientierter Programmiersprachen
- Sinnvolle Einsatzmöglichkeiten?
 - ◆ Ein Array von Vehicle, in dem friedlich nebeneinander Truck und Motorbike gespeichert werden
 - ◆ Eine Basisklasse GeometrischeForm mit abgeleiteten Klassen Quadrat, Dreieck, Kreis – und ebenfalls gleichzeitiger Speicherung von Objekten dieser Subklassen in einem Array – z.B. für ein CAD-Programm

Vorsicht *FALLE!*



- Das Prinzip der dynamischen Bindung funktioniert in Java nur für (verdeckte) Methoden ...
- ... nicht für (verdeckte) Felder!

```
public class MeinTest {
```

```
    Vehicle truck = new Vehicle();
```

```
        Motobike meineMotorrad = new Motobike();
```

```
    Vehicle irgendeinFahrzeug;
```

```
    irgendeinFahrzeug = meineMotorrad; // Motobike!
```

```
        System.out.println(irgendeinFahrzeug.vehicleType);
```

```
    ...
```

Liefert die Ausgabe
„Motorrad“ !!!

Merke!



- Der Prozess der Vererbung kann beliebig oft wiederholt werden
- Es können – neben der Klasse Motorbike – weitere Klassen von der Klasse Vehicle erben
- Ebenso können jetzt weitere Klassen von Motorbike erben
- Auf diese Weise entstehen – genau wie in der Genealogie – ganze Vererbungshierarchien
- Eine Mehrfachvererbung, also eine Subklasse aus zwei Basisklassen abzuleiten ist – so – nicht möglich
- Mit super kann immer nur auf die direkt übergeordnete Basisklasse – nicht auf deren Basisklassen – zugegriffen werden

Merke!



- Die Klasse Object steht als "Superklasse" an der Spitze der gesamten Klassenhierarchie
- Auch ohne die Angabe des Schlüsselworts extends ist jede selbst erstellte Klasse eine Subklasse der Klasse Object
- Damit sind alle Instanzen dieser Klasse auch Instanzen der Klasse Object
- Alle Klassen erben die Methoden von Object – die auch aufgerufen werden können
- Einige haben wir schon kennengelernt:
 - ◆ toString()
 - ◆ equals() etc.

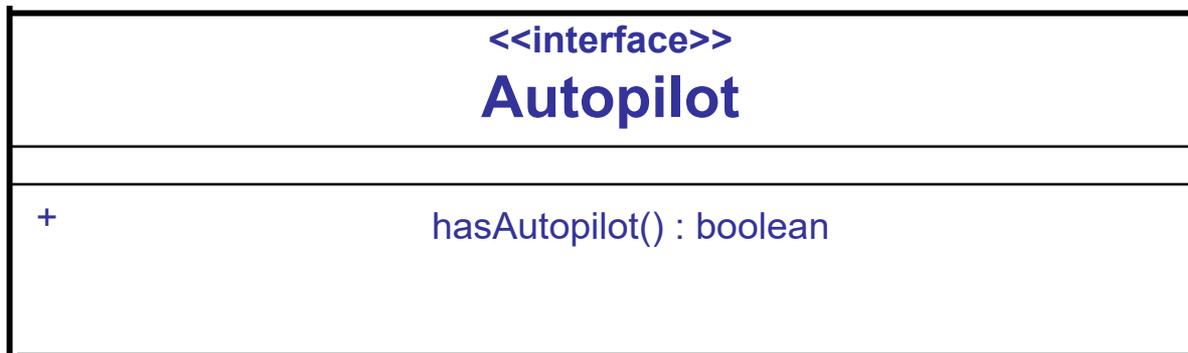
??? Fragen



*Welche Fragen
gibt es?*

- Grundsätzlich wäre doch auch eine Verknüpfung von verschiedenen Klassen denkbar. Bspw. ein PickUp (Auto + Truck) oder ein Wasserflugzeug (Vehicle + Boat).
- Viele Sprachen unterstützen diese „Mehrfachvererbung), z.B. Perl, C++, Python
- Da dies unter Umständen zu unübersichtlichem Design führt, unterstützt Java nur die Einfachvererbung
- Aber mit dem „Schnittstellenkonzept“, kann eine Klasse beliebig viele Schnittstellen – man könnte sagen „Fähigkeiten“ - erben.

```
interface Autopilot {  
    boolean hasAutopilot();  
}
```

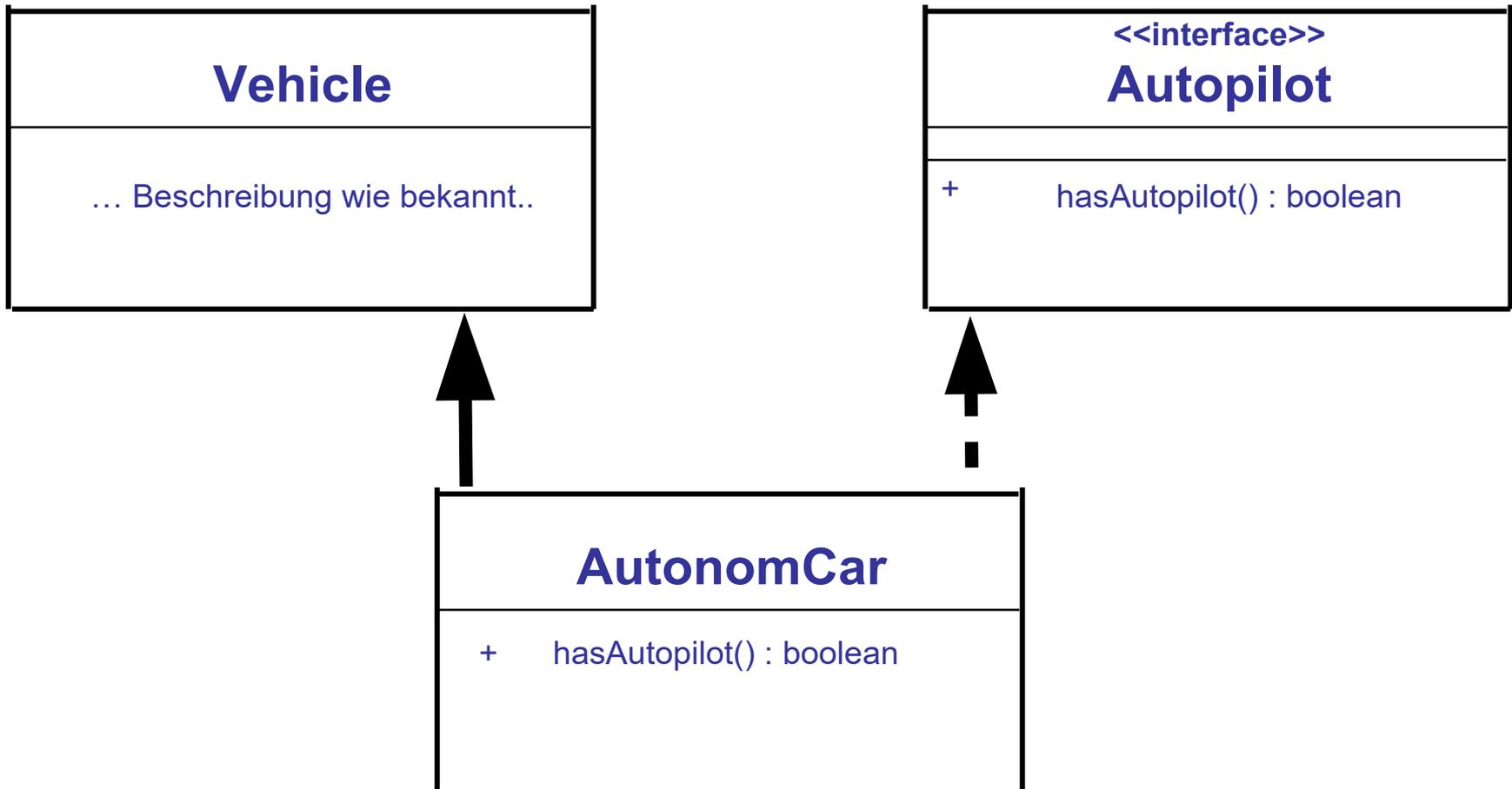


```
public class AutonomCar extends Vehicle
    implements Autopilot {

    @Override public boolean hasAutopilot() {
        return true;
    }

    // Override ist eine Annotation

}
```



??? **Fragen**



***Welche Fragen
gibt es?***

- *Das ist ein Fahrzeug*
- *Das ist ein Motorrad!?*
- **fast Richtig – das ist ein Holz Motorrad (WoodBike)**
- *Das **WoodBike** ist ein Fahrzeug (**Vehicle**), der eines **Motorads** in ein paar "Eigenschaften" nicht unähnlich ist*
- ...
- *ok, zum Verständnis: das **HolzMotorrad** ist ein Motorrad, die aus einer Vererbung entstanden ist.*
- *Das wollen wir aber verhindern*



Modifikatoren für Klassen – 1

- Um ein Objekt vom Datentyp Motorrad instanzieren zu können, benötigt man lediglich die Datei **Motorrad.class**
- Mit dieser Datei ist es (bisher) auch möglich eine eigene Klasse von der Klasse Motorrad abzuleiten
- Damit könnte jemand eine Klasse WoodBike implementieren, die von Motorrad erbt und zusätzlich die Methode hupen() überschreibt:

```
public void hupen() {  
    System.out.println("*knarz*");  
}
```
- Auch hier liegt ein Unterschied in der Verwendung vor – zwischen "reinem Instanzieren von der Klasse" und "Verwendung mittels Ableitung der Klasse"
- Auch hier hilft ein Modifikator weiter ...

- Bei Variablen hatten wir **final** als Modifikator kennen gelernt, der ein Überschreiben des Wertes nach der Initialisierung verhinderte
- **final** als Modifikator für eine Klasse verhindert ein "weiteres" Ableiten dieser Klasse ...
- ... zur Verhinderung der Klasse **WoodBike** (als Ableitung der Klasse **Motorbike**) wäre folgende Implementierung notwendig gewesen:

```
public final class Motorbike {  
    // Felder ...  
    // Konstruktoren ...  
    // Methoden ...
```



Merke!



- Wenn **final** bei Variablen und Klassen funktioniert, dann muss es doch auch eine sinnvolle Bedeutung von **final** für ...
- ... Methoden geben!
- Richtig: Mit **final** deklarierte Methoden einer Basisklasse können in den Subklassen dieser Basisklasse nicht überschrieben werden

- Es gibt sozusagen ein Gegenstück zu final: den Modifikator **abstract**
- Während final eine weitere Ableitung einer Klasse unterbindet, erzwingt abstract eine weitere Ableitung ...
- ... bevor ein Objekt dieser (dann abgeleiteten) Klasse instanziiert werden darf
- Oder mit anderen Worten:
Von einer mit abstract deklarierten Klasse selbst darf kein Object instanziiert werden

- Zur Diskussion: Sinn?

- Von der abstrakten Klasse Auto werden nun die Klassen Cabrio, Limousine, Kombi usw. abgeleitet und
 - ◆ weitere Fähigkeiten (Felder, Methoden) ergänzt oder
 - ◆ bestehende Fähigkeiten überladen oder
 - ◆ bestehende Fähigkeiten überschrieben
- Damit stellen die Subklassen notwendige Spezialisierungen, Verfeinerungen der Ober-Klasse dar.
- Da jedes Auto ihre ureigenen Fähigkeiten besitzt, beim Cabrio etwa der "verdeckOeffnen()", ist erst die Instanzierung dieser Subklassen sinnvoll

- Man könnte z.B. eine Ober-Klasse implementieren, die sozusagen eine Grundfunktionalität festlegt
- Ebenfalls als Bezeichnung üblich: **Meta-Klasse** (Klasse zur Erzeugung von Klassen)
- Aber es ist nicht sinnvoll wirklich ein Objekt dieser Klasse zu instanzieren – z.B. weil vorher noch weitere Eigenschaften, Funktionen des Objektes spezifiziert werden müssen
- In unserem Beispiel wäre es sinnvoll, eine abstrakte Klasse Vehicles zu deklarieren – aber niemand würde von seinem Auto nur als Fahrzeug sprechen, jedes Auto hat aber bspw. Typ (Audi, BMW, Opel, VW,)

Spezielle Fähigkeit?

- Von der abstrakten Klasse Auto werden nun die Klassen Cabrio, Limousine, Kombi usw. abgeleitet und
 - ◆ weitere Fähigkeiten (Felder, Methoden) ergänzt oder
 - ◆ bestehende Fähigkeiten überladen oder
 - ◆ bestehende Fähigkeiten überschrieben
- Damit stellen die Subklassen notwendige Spezialisierungen, Verfeinerungen der Ober-Klasse dar.
- Da jedes Auto ihre ureigenen Fähigkeiten besitzt, beim Cabrio etwa der "verdeckOeffnen()", ist erst die Instanzierung dieser Subklassen sinnvoll

??? **Fragen**



***Welche Fragen
gibt es?***

- Nach dem Abschnitt Konstruktoren hatten wir auch Destruktoren besprochen
- Java besitzt einen Mechanismus der dem Destruktor ähnlich ist – **ABER:**
 - ◆ Der Aufruf erfolgt NICHT am Ende der Sichtbarkeit, sondern am Ende der Lebensdauer,
 - ◆ damit der Speicherfreigabe durch den Garbage Collector!
- Jede Klasse darf eine Methode **protected void finalize() { ... }** implementieren

- Die Methode `finalize()` wird für jedes Objekt vom Garbage Collector aufgerufen, ...
- ... wenn dieses aus dem Speicher entfernt wird
- Sinn ist die Implementierung von Funktionen die beim Vernichten des Objektes aufgerufen werden sollen
 - ◆ Bei anderen Programmiersprachen sind dies z.B. explizite Freigaben von allozierten Speicherbereichen
 - ◆ Bei Java erfolgt dies automatisch und so verlor der Destruktor als solcher erheblich an Bedeutung
- Der Aufruf von `finalize()` erfolgt also u.U. erst zum Zeitpunkt der Terminierung des Programmes
- Durch die zwingende Deklaration mit dem Modifikator `protected` ist auch der explizite Aufruf aus einer "nur instanzierenden" Klasse nicht erlaubt

Bereinigen nach Ende der Sichtbarkeits– 1

- Deshalb ist `finalize` seit Java 9 „deprecated“* , da der Zeitpunkt des Aufrufs nicht gut vorhersehbar/problematisch ist.

*Deprecated: überholt, nicht mehr empfohlen

- Der „gute“ Weg Aufgaben ähnlich dem Destructor zu erledigen ist nun die Schnittstelle `AutoClosable` zu verwenden
- Oder um Aufgaben, die am genau Ende der Sichtbarkeit stattfinden, ist `java.lang.ref.Cleaner`
Dieser ist für uns noch relativ kompliziert, der Vollständigkeit wegen folgt ein Beispiel

(<https://docs.oracle.com/javase/9/docs/api/java/lang/ref/Cleaner.html>)

```
class Resource implements AutoCloseable {
    public Resource() {
        System.out.println("Resource geöffnet.");
    }

    @Override
    public void close() {
        System.out.println("Resource geschlossen.");
    }
}

public class Main {
    public static void main(String[] args) {
        try (Resource res = new Resource()) {
            System.out.println("Benutze die Resource.");
        }
    }
}
```

```
import java.lang.ref.Cleaner;

public class BeispielRessourcenCleanUp implements AutoCloseable{
    private static final Cleaner cleaner=Cleaner.create();
    private final Cleaner.Cleanable cleanable;
    // *Spezielle Klasse State für den Status der Ressource/ Aufräumaktion – nächste Folie
    private final State state;

    public BeispielRessourcenCleanUp() {
        this.state = new State( .... ); // Punkte sind exemplarisch
        this.cleanable = cleaner.register(this, this.state );
    }

    @Override
    public void close() {
        cleanable.clean();
    }
}
```

*

```
static class State implements Runnable {  
  
    State(...) {  
        // Initalisierung von State für Aufräumaktion  
    }  
  
    public void run() {  
        // Aufräumaktion , wird min. einmal ausgeführt    }  
    }  
}
```

??? Fragen



*Welche Fragen
gibt es?*