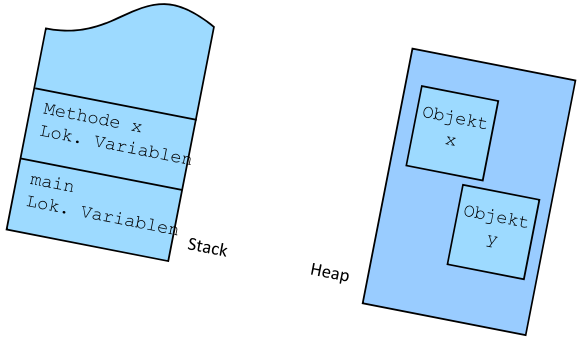
DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

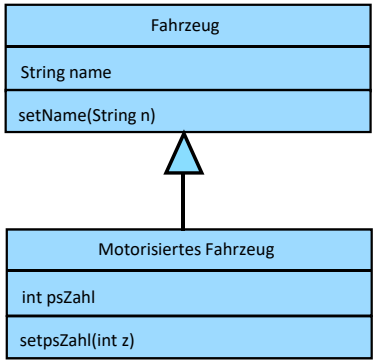
Vorlesung Programmieren
Thema 6: Objektorientierung (I)

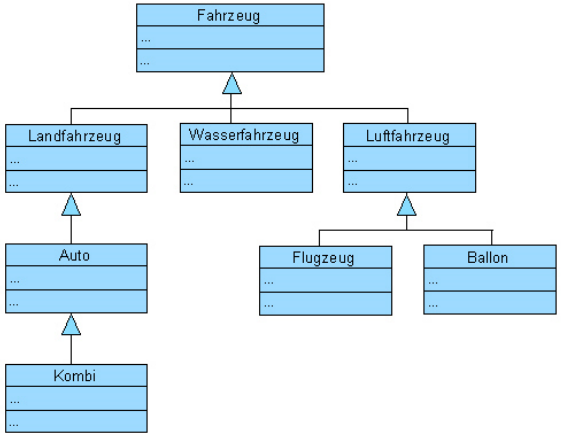
Olaf Herden

Fakultät Technik

Studiengang Informatik







Stand: 11/2023

Gliederung

- Klassen und Objekte
- Vererbung
- Klassen String und Object

Darstellungsebenen

- Folgende Betrachtungen auf zwei Ebenen:
  - Modellierung
    - Analysephase zuzurechnen
    - Notation UML (Unified Modeling Language)
  - Programmierung
    - Codierungsphase zuzurechnen
    - Notation Java-Code
- Warum Unterscheidung?
  - Im Softwareentwicklungsprozess von abstrakteren zu konkreteren Beschreibungsformen
  - Natürliche Sprache als Beschreibungsmittel nicht eindeutig/präzise genug
  - Notation in UML auch zur Verständigung mit Nicht-Programmierern
  - Aufbauend auf Modellierung Einsatz unterschiedlicher Programmiersprachen

Klassen und Objekte

- Bsp. Reihe von Fortbewegungsmitteln





- Objekt: Einzelnes Fortbewegungsmittel
- Klasse: Zusammenfassung ähnlicher Objekte
- Also: Klasse Schablone, zur Laufzeit Erzeugung von Objekten
- Beschreibung Klassen:
  - Attribute (statische Eigenschaften)
  - Methoden (dynamische Aspekte/Verhalten)

Methoden (I)

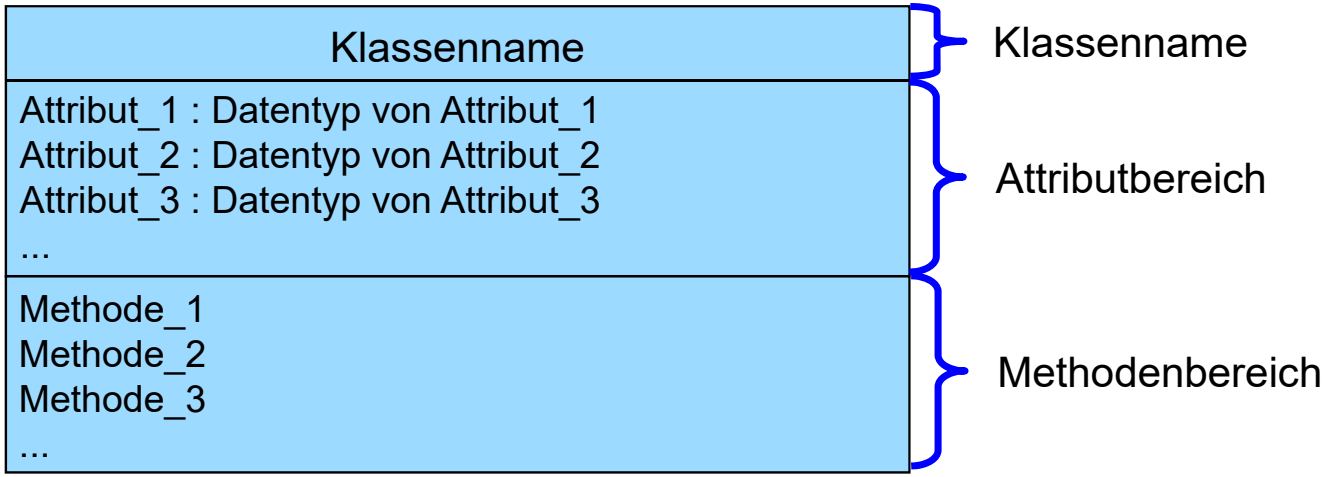
- Folge von Java-Anweisungen, von anderer Stelle im Programm aufrufbar, lösen geschlossene Teilaufgabe
- Bei Aufruf Übergabe von 0,1 oder mehr Werten (Argumente, Parameter)
- Wichtige Bezeichnungen:
  - Signatur: Besteht aus Datentypen der Parameter (z.B. `(int, int, float)`)
  - Parameterliste: Besteht aus Datentypen der Parameter und ihren Namen (z.B. `(int i, int j, float x)`)
- Rückgabe:
  - Rückgabe Wert eines definierten Datentyps oder keine Rückgabe
  - Rückgabe erfolgt mit Anweisung `return <Ausdruck>;`
  - Typkompatibilität `<Ausdruck>` mit angegebenem Rückgabedatentyp
  - Methode ohne Rückgabewert $\Rightarrow$ Kein Ausdruck hinter `return`

Methoden (II)

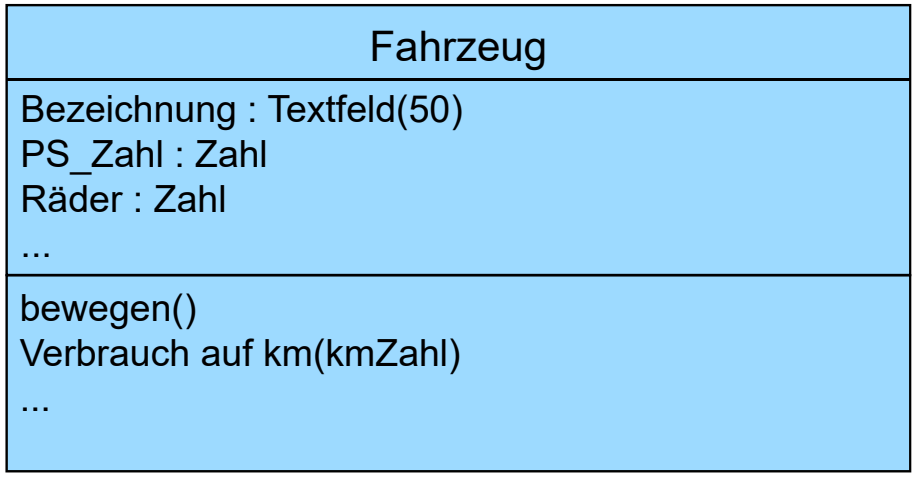
- Benennung in nicht-objektorientierten Sprachen:
  - Prozeduren: Methoden ohne Rückgabewert
  - Funktionen: Prozeduren mit Rückgabewert
- In Java:
  - Geschachtelte Methoden nicht zugelassen, d.h. in Methode darf keine weitere deklariert werden
  - Andere Programmiersprachen erlauben dies
- Aufruf von Methoden:
  - Nennung des Namens, evtl. Angabe des Objektes
  - Methoden mit Rückgabewert müssen in Ausdruck eingebunden werden
  - Nach Aufruf wird Abarbeitung des Codes im Methodenrumpf gestartet
  - Abarbeitung bis Ende des Methodenrumpfes bzw. bis `return`-Anweisung

Klassen: UML-Notation

- Aufbau:



- Beispiel:



Klassen: Java-Notation (I)

- Syntax:

```
class <Name>{
   [Attributdeklaration]
   [Methodendeklaration]
}
```

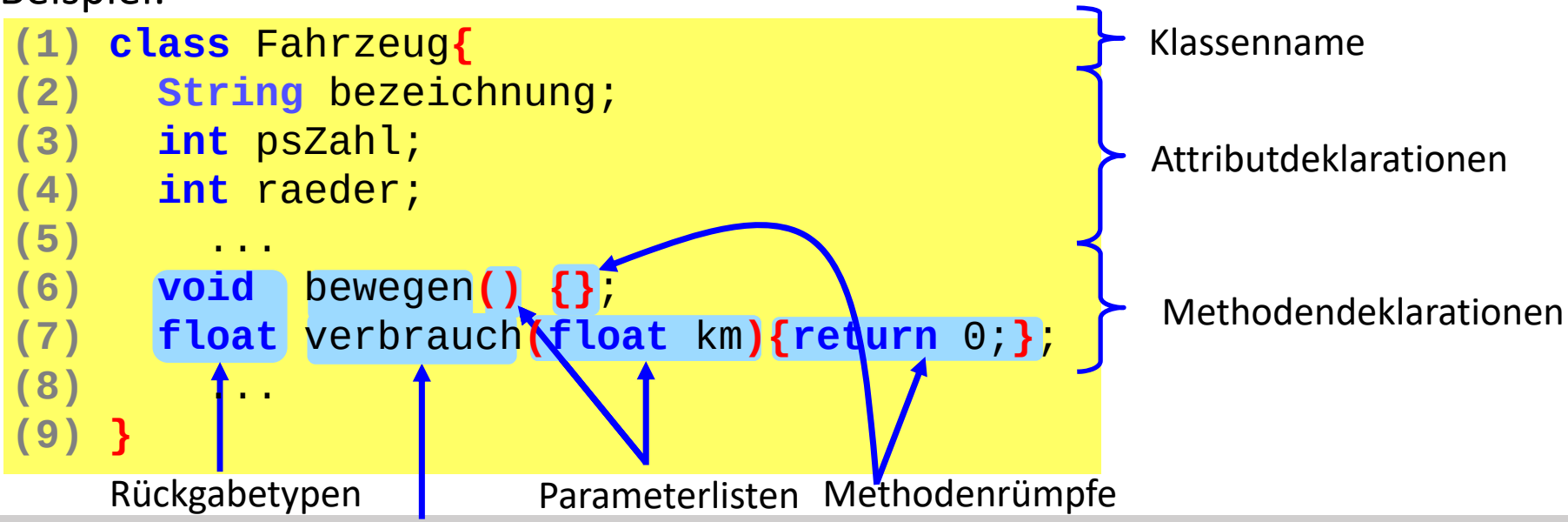
- Attributdeklaration: Angabe Datentyp und Attributname `<Datentyp> <Attributname>`
- Beispiele: `int i; String name;`
- Methodendeklaration (Methodenkopf und –rumpf):
  - Methodenkopf:
    - Typ des Wertes, den Methode zurückgibt
    - Methodennamen
    - Typ und Name der Übergabeparameter (Parameterliste)

Klassen: Java-Notation (II)

- Methodenrumpf:
  - Beliebige Anzahl Anweisungen, eingeschlossen in geschweiften Klammern:

  `[Rückgabetyp] <Methodenname> ({<Datentyp> <Parametername>})`
  `{<Rumpf>};`
- Methoden, die keinen Wert zurückgeben, haben Rückgabetyp `void`

- Beispiel:



Java: `main`-Methode

- Sonderstellung
- Konventionen:
  - Benötigt als Parameterliste ein String-Feld
  - Rückgabetyp muss `void` sein
  - Methode muss als `static` deklariert sein (Anm.: Später in VL mehr)
- Beispiel:

```
static void main(String[] args){
    ...
}
```

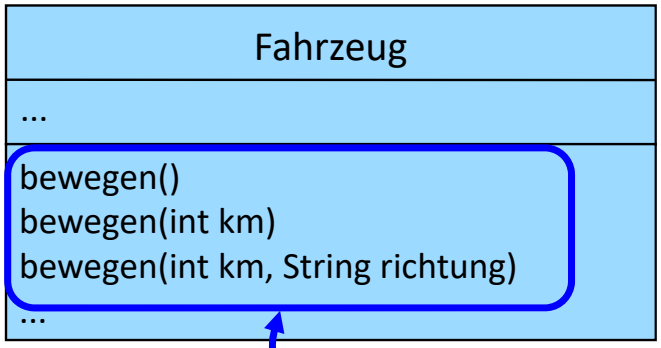
- Bedeutung:
  - Ausführung `main`-Methode bei Aufruf einer Klasse (wie in bisherigen Beispielen)

Namenskonventionen in Java

- Zwingend: Klassennamen …
  - dürfen prinzipiell unbeschränkte Länge haben (bis auf technische Einschränkungen durch System)
  - müssen (wie alle anderen Java-Bezeichner) mit Buchstaben oder „_“ oder „$“ beginnen, alle weiteren Zeichen Buchstaben, Zahlen und ein paar Sonderzeichen
  - werden (wie alle anderen Bezeichner in Java) nach Groß-und Kleinschreibung unterschieden
  - dürfen (wie andere Bezeichner) nicht einem Java-Schlüsselwort entsprechen
- Üblich: Klassennamen …
  - sollten nicht mit Paketnamen identisch sein
  - sollten möglichst aussagekräftig sein („Sprechende Bezeichner“)
  - sollten mit Großbuchstaben beginnen, dann sollten Kleinbuchstaben folgen
  - bestehend aus mehreren Wörtern sollten zusammen geschrieben werden, neues Wort mit Großbuchstaben beginnen

Überladen von Methoden (I)

- Klassen können mehrere Methoden mit gleichem Namen besitzen
- Jedoch: Unterscheidung in Signatur
- Statische Polymorphie
- Notation in UML und Java: Angabe mehrerer Methodendeklarationen
- Beispiel:




```
class Fahrzeug{
  ...
  void bewegen(){};
  void bewegen(int km){};
  void bewegen(int km,
               String richtung){};
  ...
}
```

Überladene Methode bewegen

Überladen von Methoden (II)

- Unterscheidung in Signatur, Unterscheidung in Parameterliste nicht ausreichend:

```
class Fahrzeug{
  ...
  void bewegen(int km, int x){...};
  void bewegen(float km, int x){...};
 }
```

funktioniert, aber

```
class Fahrzeug{
  ...
  void bewegen(int km, int x){...};
  void bewegen(int a , int b){...};
 }
```

scheitert: `bewegen(int,int) is already defined in Fahrzeug`

Überladen von Methoden (III)

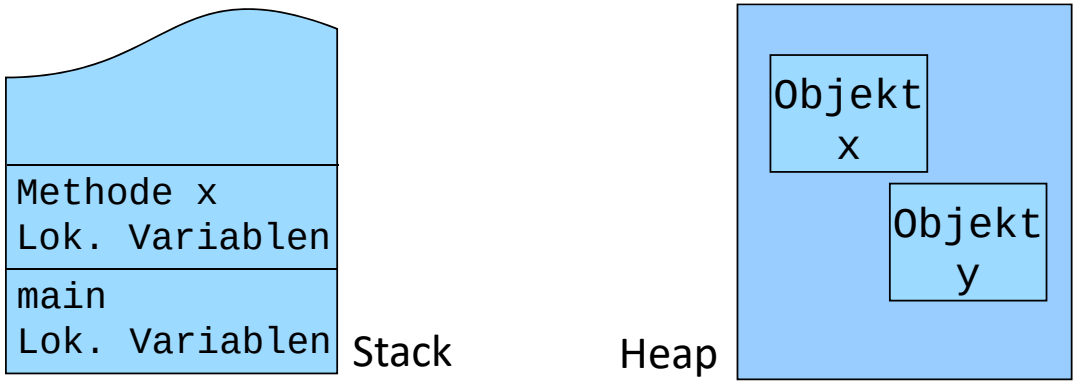
- Ebenso unzureichend: Unterscheidung Methoden nur im Rückgabetyp:

```
class Fahrzeug{
  ...
  void bewegen(int km, int x){...};
  int  bewegen(int a , int b){...};
 }
```

scheitert: `bewegen(int,int) is already defined in Fahrzeug`

Java-Speichermodell

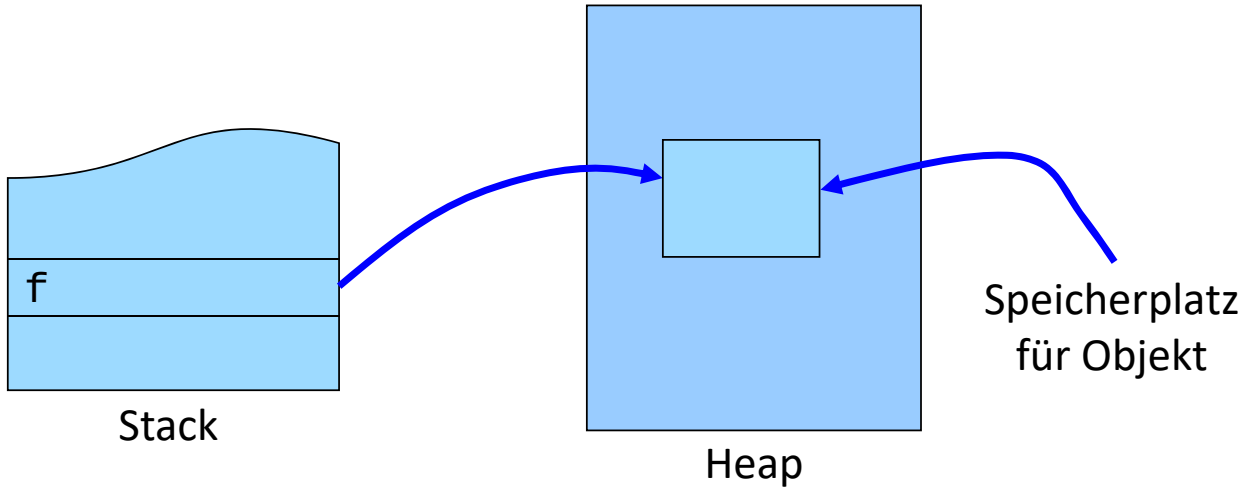
- Stack:
  - Verwaltet für jede Methode Rahmen (Aufruf- und Rückgabeparameter, lokale Variablen)
  - Arbeit nach FIFO-Prinzip (First In – First Out)
  - Zuletzt aufgerufene Methode wird als erste wieder verlassen
- Heap:
  - Speicherung von Objekten
  - Erzeugen Objekt ⇒ Allokation passender Speicherplatz „irgendwo“ im Heap



- Anm.: In Java insgesamt fünf Speicherbereiche

Konstruktoren (I)

- Spezielle Methoden zum Anlegen von Objekten
- Name enstpricht Klassennamen
- Standard-Konstruktor: leere Signatur, wird automatisch angelegt
- Anlegen von Objekten mit Operator `new`
- Syntax: `<Klassenname> <Objektname> = new <Klassenname>();`
- Beispiel: `Fahrzeug f = new Fahrzeug();`
- Im Speicher:
  - Anlegen Variable `f` auf Stack
  - Reservierung entsprechender Speicherplatz für Objekt auf Heap
  - `f` verweist auf diesen Speicherplatz



Konstruktoren (II)

- Pro Klasse beliebige Anzahl Konstruktoren (Mechanismus des Überladens)
- Beispiel:

```
Fahrzeug(){;}
Fahrzeug(String n){bezeichnung=n;}
Fahrzeug(String n,int p){bezeichnung=n;psZahl = p;}
```

- Anlegen Objekte:

```
Fahrzeug f = new Fahrzeug();
Fahrzeug g = new Fahrzeug(„Opel Astra");
Fahrzeug h = new Fahrzeug(„VW Golf",75);
```

Zugriff auf Komponenten einer Klasse

- Innerhalb Klasse: Zugriff auf Komponenten (Attribute und Methoden) durch Angabe
- Bsp.:

```
Fahrzeug(String n,int p){
   bezeichnung=n;
   psZahl = p;
}
```

Direkter Zugriff innerhalb Klasse

- Methodenaufruf oder Attributzugriff einer Variable ⇒ Punktnotation, z.B.:

```
class Fahrzeug{
  ...
  public static void main(String args[]){
     Fahrzeug f = new Fahrzeug("Astra");
    f.psZahl = 90;
    f.bewegen();
  }
 }
```

Variablenzugriff über Punktnotation

Das gleiche und dasselbe (I)

- Programm

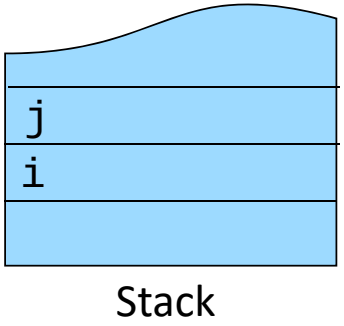
```
int i = 3;
int j = 3;
if (i == j){
   System.out.println("Gleich");
}
else{
   System.out.println("Ungleich");
}
Fahrzeug f = new Fahrzeug("Astra");
Fahrzeug g = new Fahrzeug("Astra");
if (f == g){
   System.out.println("Gleich");
}
else{
   System.out.println("Ungleich");
}
```

führt zur Ausgabe

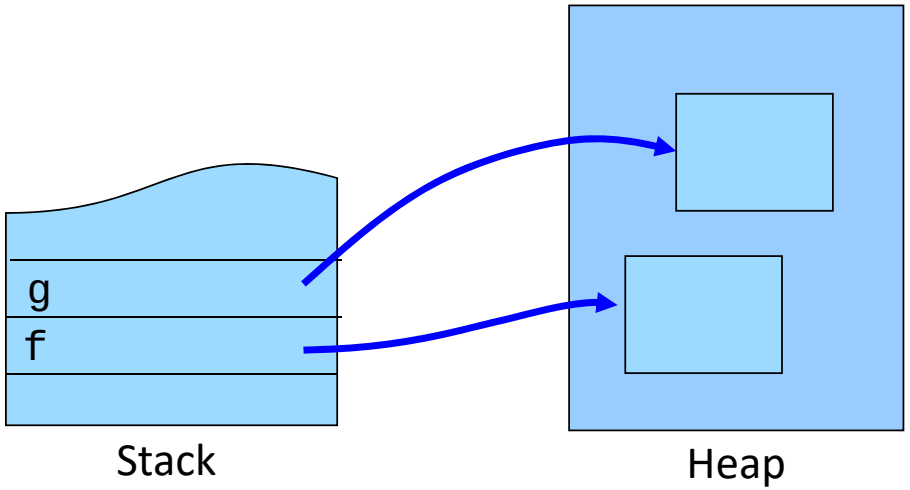
```
Gleich
Ungleich
```

Das gleiche und dasselbe (II)

- Variablen:
  - Ablage von i und j auf Stack
  - Vergleich der Werte

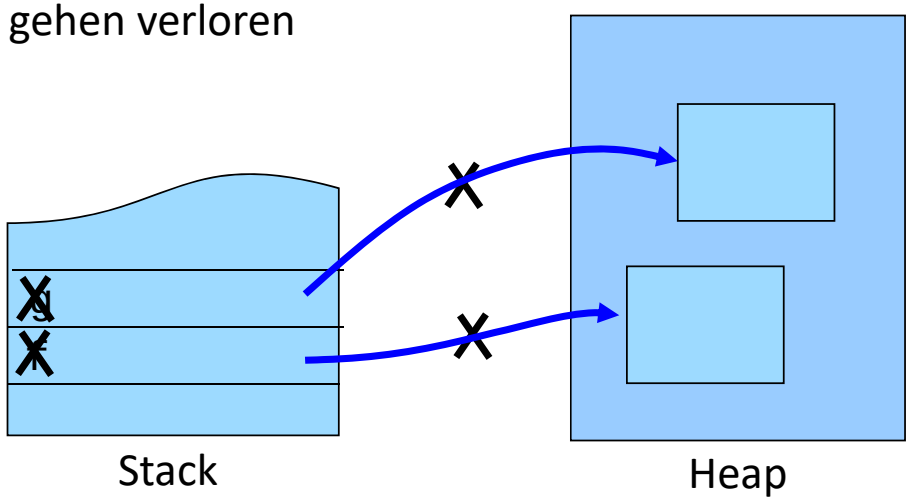


- Objekte:
  - Ablage von f und g auf Stack
  - Aber: Verweis auf Heap
  - Unabhängig von Werten verweisen f und g auf unterschiedliche Adressen im Heap



Löschen von Objekten

- Keine explizites Löschen von Objekten in Java
- Ende der Lebensdauer (Methoden- bzw. Blockende):
  - Entnehmen Objektbezeichner vom Stack
  - Verweise auf Heap gehen verloren



- Garbage Collector:
  - Komponente in JVM
  - Bereinigt in „gewissen Zeitabständen“ Heap (Freigabe Speicher)

Methode `finalize`

- Manchmal znmittelbar vor Objektlöschung noch gewisse Aktionen notwendig, z.B.
  - Schließen geöffneter Datei
  - Freigeben von Verbindungen zu einer Datenbank
- Hierzu: Methode finalize(), standardmäßig leere Implementierung hat
- Bsp.: Zählen der Instanzen

```
public class Fahrzeug{
   static int anzObjekte = 0;
   public Fahrzeug(){
      ++anzObjekte;
      <Weitere Anweisungen>
   }
   public void finalize(){
      --anzObjekte;
      <Weitere Anweisungen>
   }
   ...
}
```

Variable Anzahl Argumente (I)

- Bisher: Anzahl Übergabeparameter fest
- Manchmal variable Parameteranzahl sinnvoll
- Beispiel: Addition einer beliebigen Anzahl von ganzen Zahlen
- Syntax: `<Rückgabetyp> <Methodenname>(<Datentyp>... <Var_Name>)`
- Beispiel:

```
int summiere(int... intListe){
    ...
}
```

- Abarbeitung mit vereinfachter `for`-Schleife:

```
for (int j : intListe){
     summe += j;
}
```

Variable Anzahl Argumente (II)

- Zusammenfassend:

```
public static int summiere(int... intListe){
    int summe = 0;
    for (int i : intListe){
        summe += i;
    }
    return summe;
}
```

- Aufrufe in `main`-Methode

```
System.out.println(summiere(1,2,3));
System.out.println(summiere(1,2,3,4,5));
System.out.println(summiere(1,2,3,4,5,6,7,8,9,10));
```

führen zur Ausgabe

```
6
15
55
```

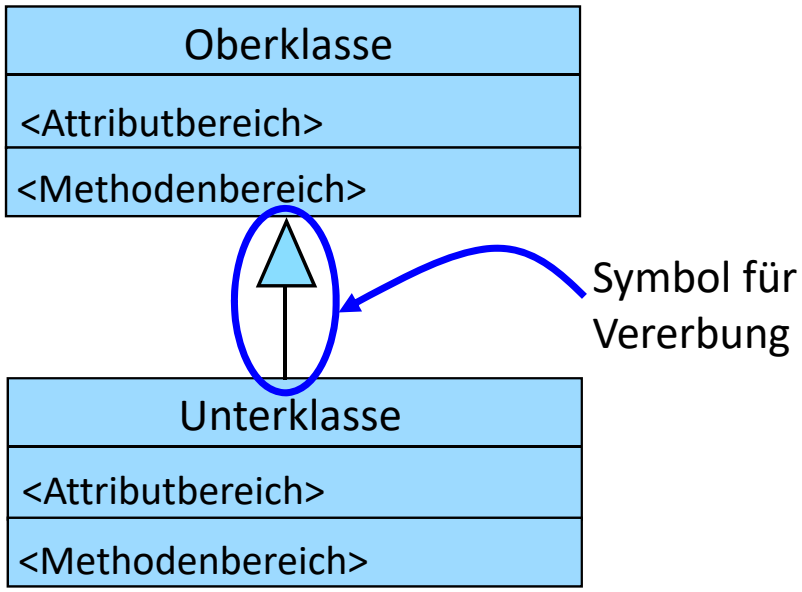
Gliederung

- Klassen und Objekte
- Vererbung
- Klassen String und Object

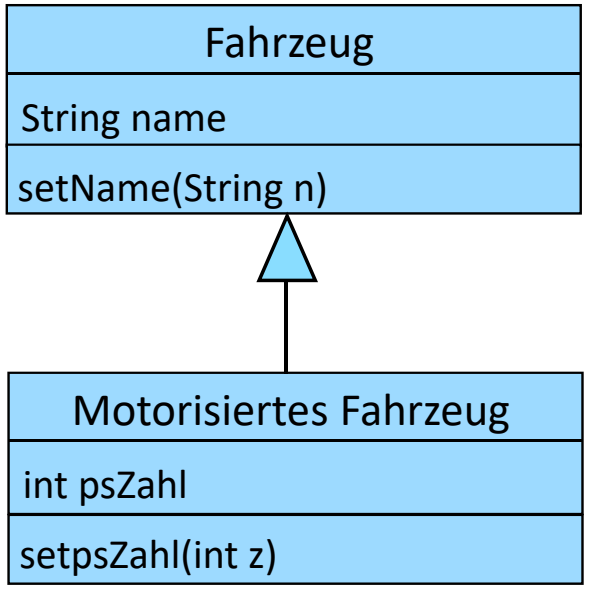
Idee

- Beobachtung:
  - Alle Objekte im Beispiel Fahrzeuge
  - Teilweise gemeinsame Attribute (z. B. Bezeichnung), andere verschieden (z. B. anzahlRaeder)
  - Gleiches gilt für Methoden
- Daher: Klassen besser/feiner strukturieren durch Zusammenfassen gemeinsamer Eigenschaften
- In Objektorientierung: Vererbung:
  - Alle Attribute und Methoden von Oberklasse (Superklasse, vererbende Klasse) werden an Unterklasse (Subklasse, erbende Klasse) übertragen (vererbt)
  - Ergänzung Unterklassen um weitere Attribute und Methoden
  - Synonyme: Generalisierung bzw. Spezialisierung

UML-Notation



Beispiel:



- Anmerkung:
  - Objekt der Subklasse (im Bsp. Motorisiertes Fahrzeug) auch Objekt der Superklasse (im Bsp. Fahrzeug)
  - Damit: Attribute und Methoden der Superklasse in Subklasse vorhanden

Java-Notation

- Vererbung wird in Java durch das Schlüsselwort `extends` festgelegt
- Definition der Oberklasse:

```
class Oberklasse{
  <Attributbereich>
  <Methodenbereich>
}
```

- Definition der Unterklasse:

```
 class Unterklasse extends Oberklasse{
  <Attributbereich>
  <Methodenbereich>
}
```

Java: Beispiel

- Definition der Superklasse:

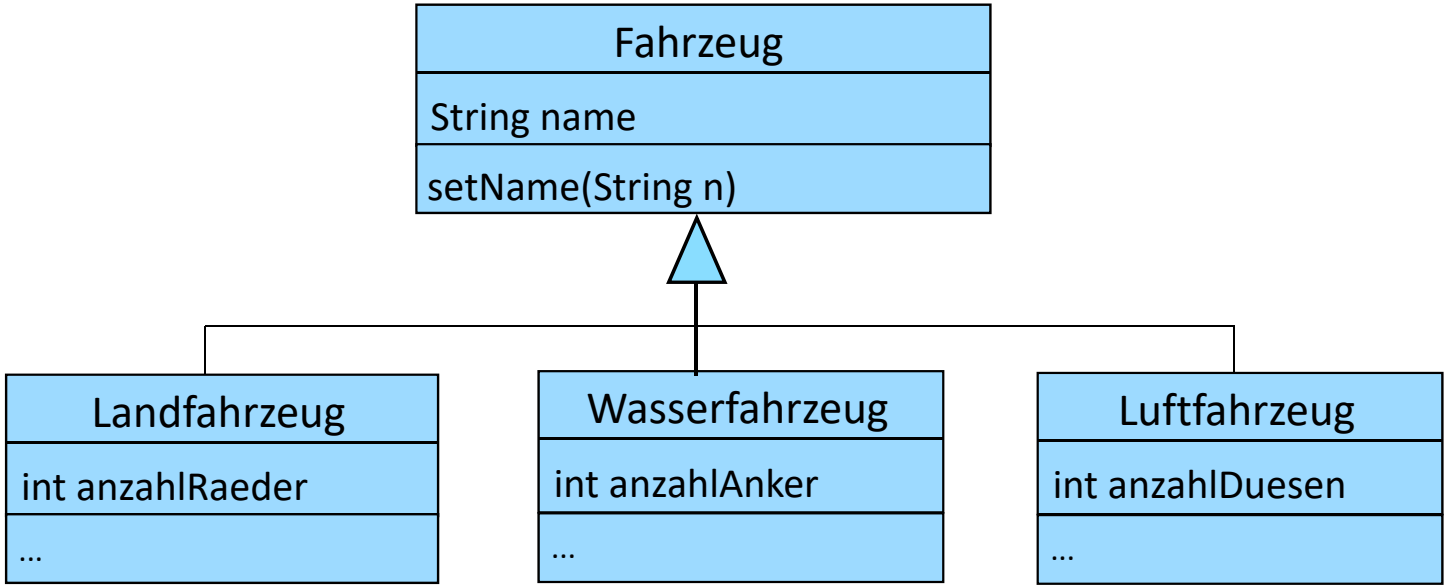
```
class Fahrzeug{
  String name;
  void setName(String n){name=n;};
}
```

- Definition der Subklasse:

```
 class MotorisiertesFahrzeug extends Fahrzeug{
  int psZahl;
  void setpsZahl(int p){psZahl=p;};
  public static void main(String[] args){
    MotorisiertesFahrzeug m = new MotorisiertesFahrzeug();
    m.setpsZahl(75);
    m.setName("Astra");
     //Ausgabe
   System.out.println(m.name);
       System.out.println(m.psZahl);
 }
```

Mehrere Subklassen (I): UML

- Vererbung einer Superklasse an mehrere Subklassen
- Beispiel UML-Notation:



Mehrere Subklassen (II): Java

- Superklasse:

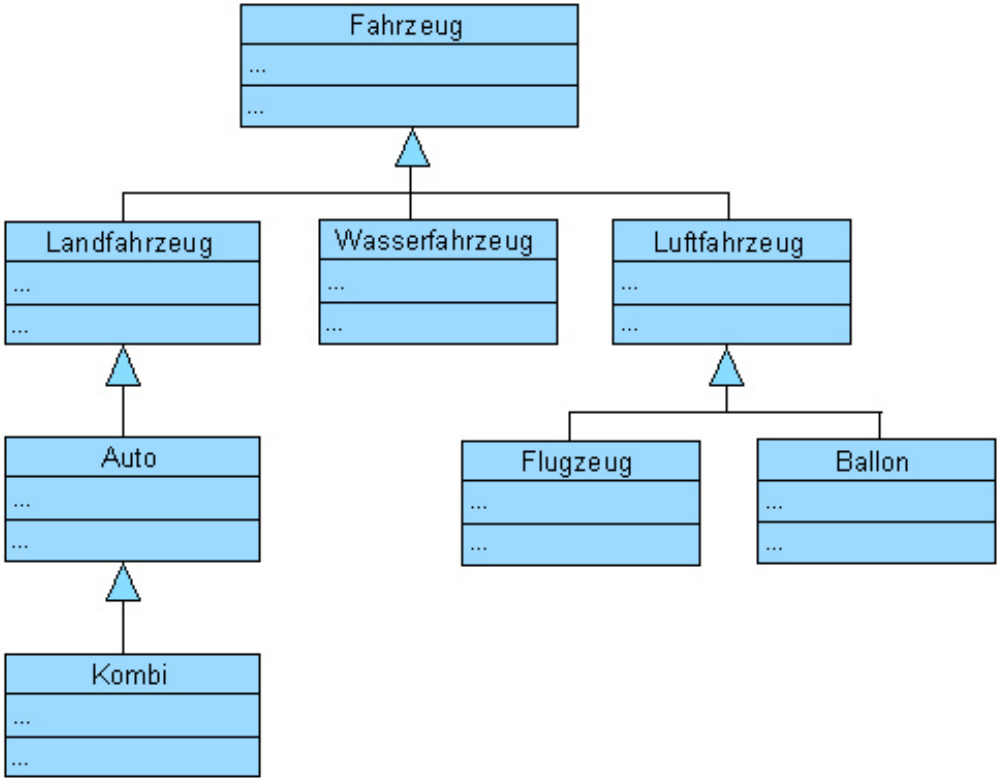
```
class Fahrzeug{
  String name;
  void setName(String n){name=n;};
}
```

- Subklassen:

```
class Landfahrzeug extends Fahrzeug{
  int anzahlRaeder;
  ...
}
class Wasserfahrzeug extends Fahrzeug{
  int anzahlAnker;
  ...
}
class Luftfahrzeug extends Fahrzeug{
  int anzahlDuesen;
  ...
}
```

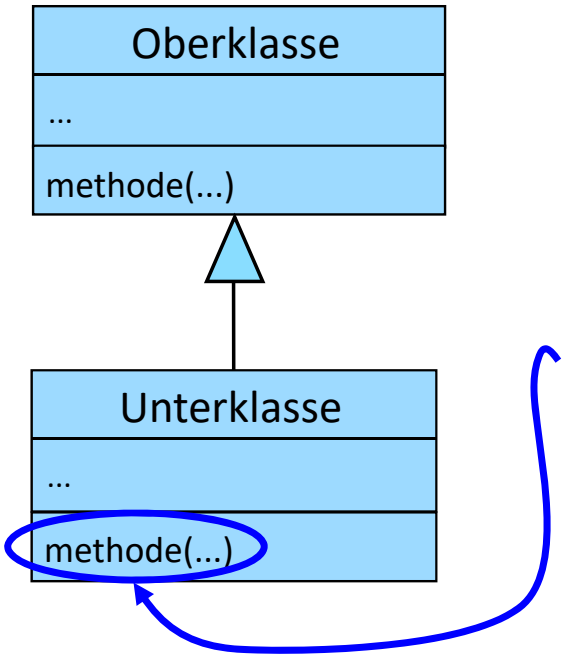
Klassenhierarchie

- Vererbungen über mehrere Ebenen
- Bezeichnung: Klassen- bzw. Vererbungshierarchie



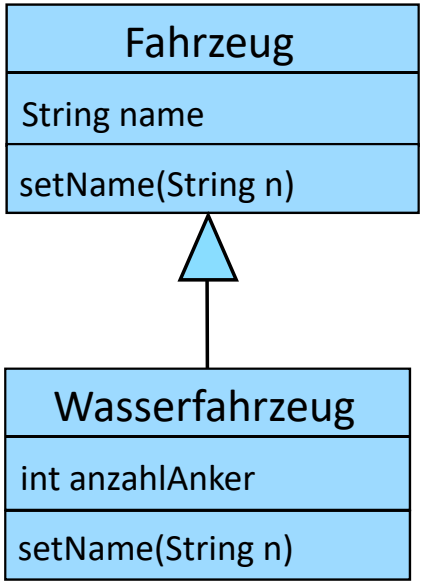
Überschreiben (I): UML

- Möglich: Methoden in Superklasse definiert, in Subklasse nicht passend
- Überschreiben: Methode in Subklasse redefinieren



Erneutes Aufführen überschriebener Methode in Subklasse

Beispiel:



Überschreiben (II): Java

- Methode in Subklasse nochmals definieren
- Superklasse:

```
class Fahrzeug{
  String name;
  void setName(String n){name=n;}
}
```

- Subklasse mit überschriebener Methode `setName`:

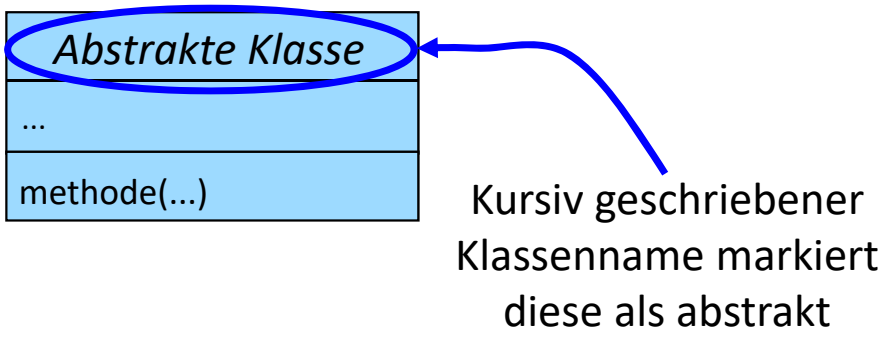
```
class Wasserfahrzeug extends Fahrzeug{
  int anzahlRaeder;
  ...
  void setName(String n){name= "Schiff "+n;}
  ...
}
```

Dynamische Polymorphie

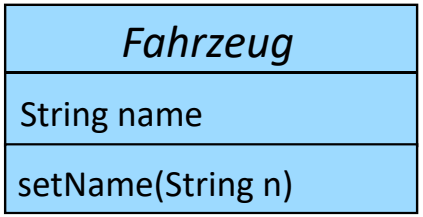
- In OO-Sprachen Methoden mit gleichem Namen
- Anwendung auf verschiedene Objekte
- Objekte unterschiedlicher Klassen werden bei Anwendung der gleichen Methode unterschiedlich reagieren
- Beispiel:
  - Superklasse Fahrzeug mit Methode bewegen
  - Subklassen Auto und Flugzeug überschreiben Methode bewegen
  - Bei Aufruf werden unterschiedliche Methoden aufgerufen

Abstrakte Klassen (I): UML

- Bei Aufbau von Klassenhierarchien Klassen, zu denen keine sinnvollen Objekte angelegt werden können
- Klassen dennoch sinnvoll zur Strukturierung
- Bsp.: Klasse Fahrzeug, das nur durch seinen Namen gekennzeichnet ist
- Abstrakte Klasse: Klasse, von der keine Objekte angelegt werden dürfen



Beispiel:



Abstrakte Klassen (II): Java

- Voranstellen Schlüsselworts `abstract`
- Beispiel:

```
abstract class Fahrzeug{
  String name;
  void setName(String n){name=n;};
}
```

- Versuch

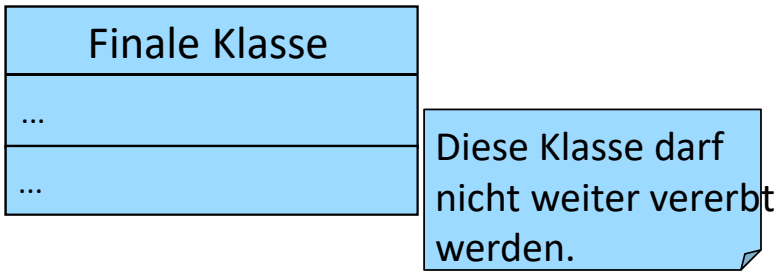
```
Fahrzeug f = new Fahrzeug();
```

Objekt anzulegen, führt zur Fehlermeldung:

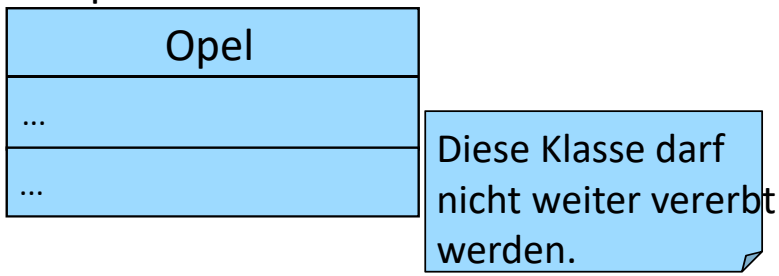
```
Fahrzeug is abstract; cannot be instantiated
```

Finale Klassen (I): UML

- Klassen, die nicht weiter vererbt werden sollen
- Anwendung z.B. Festlegung von Standards
- Kein besonderes Symbol in UML
- Abhilfe durch Annotation



Beispiel:



Finale Klassen (II): Java

- Voranstellen Schlüsselwort `final`
- Beispiel:

```
final class Opel{
  String name;
  void setName(String n){name=n;}
}
```

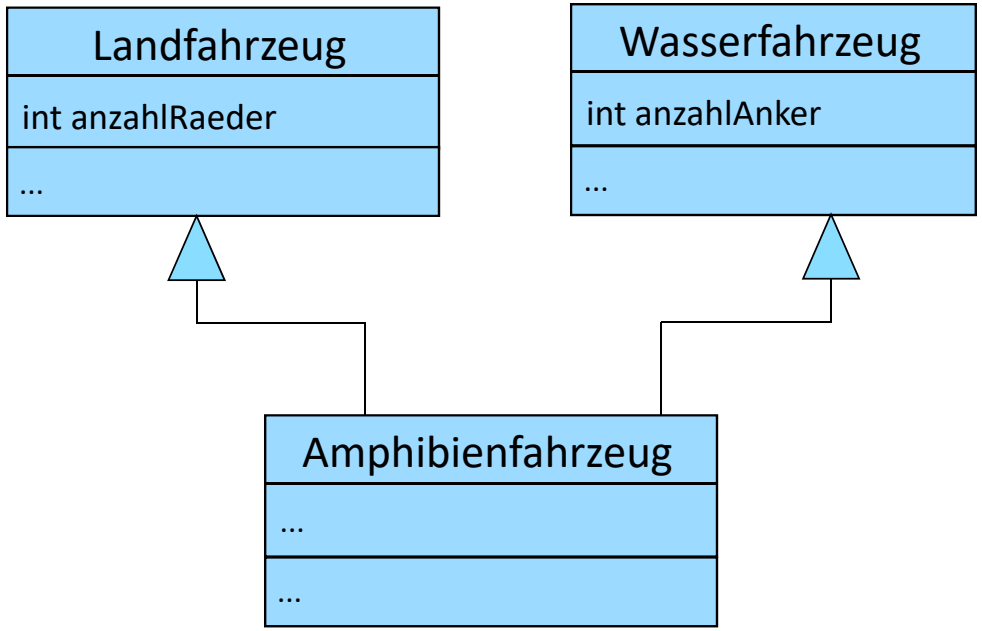
- Versuch

```
class Astra extends Opel{
  ...
}
```

von dieser Klasse abzuleiten, führt zur Fehlermeldung:
`cannot inherit from final Opel`

Mehrfachvererbung (I)

- Vererbung von mehreren Superklassen an eine Subklasse
- Bsp: Amphibienfahrzeug sowohl Land- als auch Wasserfahrzeug
- UML-Notation:



Mehrfachvererbung (II)

- Einerseits: Mehrfachvererbung gutes Strukturierungsmittel
- Andererseits: Problematisch, welche Methode beim Erben aus zwei Superklassen gültig ist
- Unterschiedliche Strategien:
  - Gleich benannte Methoden in Superklassen mit unterschiedlicher Semantik verbieten
  - Methode muss in Subklasse zwingend überschrieben werden
  - Genaue Benennung, welche der Methoden gemeint ist
- Jede Strategie bringt Probleme mit sich
- Daher: Keine Mehrfachvererbung in Java

Gliederung

- Klassen und Objekte
- Vererbung
- Klassen String und Object

Klasse `String` (I)

- Neben selbst geschriebenen Klassen liefert Java riesige Klassenbibliothek
- Klasse String für Zeichenketten
- Anlegen:

```
String s = new String("Horb");
String s = "Horb";
```

- Wichtige Methoden:
  - `length()` liefert Länge der Zeichenkette
  - Beispiel:

```
String s = new String("Horb");
String t = "Programmierung";
System.out.println(s.length());
System.out.println(t.length());
```

führt zur Ausgabe

```
4
14
```

Klasse `String` (II)

- `equals(String s)` liefert `true` im Falle der Gleichheit, ansonsten `false`
- Beispiel:

```
String s = new String("Horb");
String t = new String("Horb");
if (s.equals(t)){
   System.out.println("Gleich");
}
else{
   System.out.println("Ungleich");
}
```

führt zur Ausgabe

```
Gleich
```

Klasse `String` (III)

- `equalsIgnoreCase(String s)` wie `equals`, allerdings Vernachlässigung von Groß-/Kleinschreibung, d.h. auch nach

```
String s = new String("Horb");
String t = new String("hoRb");
```

sind Zeichenketten gleich

- `charAt(int i)` gibt Zeichen an i-ter Position zurück, z.B.

```
String s = new String("Horb");
System.out.println(s.charAt(2));
```

führt zur Ausgabe

```
r
```

Klasse `String` (IV)

- `compareTo(String s)`:
  - Liefert < 0, wenn aufrufendes String-Objekt lexikographisch vor s kommt
  - Liefert = 0, wenn aufrufendes String-Objekt gleich s ist
  - Liefert > 0, wenn aufrufendes String-Objekt lexikographisch nach s kommt
- Beispiel:

```
String s = new String("Horb");
String t = new String("Stuttgart");
String u = new String("Balingen");
System.out.println(s.compareTo(t));
System.out.println(s.compareTo(s));
System.out.println(s.compareTo(u));
```

führt zur Ausgabe

```
-11
  0
  6
```

Klasse `String` (V): Konvertierung

- Methode `valueOf(<Typ>)` verwandelt angegebenen Typ in String, z.B.

```
String s = String.valueOf(7) ;
String t = String.valueOf(3<4);
```

führt zu: `s=„7“` bzw. `t=„true“`

- Zu jedem einfachen Typ gibt es Klasse, z.B. Klasse `Integer` zu `int`
- Bieten Methoden zur Konvertierung an, z.B.

```
String s = Integer.toString(int i)
```

verwandelt `i` in Zeichenkette

```
int i = Integer.parseInt(s)
```

verwandelt Zeichenkette `s` in `int`-Wert

Klasse `Object` (I)

- Oberste aller Klassen, d.h. jede andere Klasse von `Object` abgeleitet, d.h. jedes Objekt ist Instanz von `Object`
- Wichtige Methoden: `getClass()`, `toString()`, `equalsTo()`
- `getClass()` liefert Namen der Klasse des Objekts, z.B.

```
Fahrzeug f = new Fahrzeug();
System.out.println(f.getClass());
```

führt zur Ausgabe

```
class Fahrzeug
```

bzw.

```
Fahrzeug f = new Fahrzeug();
System.out.println(f.getClass().getName());
```

führt zur Ausgabe

```
Fahrzeug
```

Klasse `Object` (II)

- `toString()` gibt Inhalt des Objekts aus
- Automatischer Aufruf bei `print` oder `println` mit Angabe eines Objekts
- Beispiel:

```
Fahrzeug f = new Fahrzeug();
System.out.println(f);
```

führt zur kryptischen Ausgabe

`Fahrzeug@3e86d0`

- Zusammensetzung: Klassenname + „@“ + Hexadezimaler Hashcode
- Existiert in Klasse Methode `toString`, z.B.:

```
public String toString(){return bezeichnung;}
```

führt

```
Fahrzeug f = new Fahrzeug(“Astra“);
System.out.println(f);
```

zur Ausgabe `Astra`

Klasse `Object` (III)

- `boolean equals(Object o)` prüft auf Gleichheit, z.B.

```
public boolean equals(Fahrzeug f){
   if (   (bezeichung == f.bezeichnung)
        &&(psZahl     == f.psZahl)
        &&(raeder     == f.raeder)
           <Alle weiteren Attribute auf Gleichheit prüfen>
      )
       return true;
   else return false;
}
```

dann führt

```
Fahrzeug f = new Fahrzeug("Astra");
Fahrzeug g = new Fahrzeug("Astra");
if (f.equals(g)){
   System.out.println("Gleich.");
}
```

zur Ausgabe `Gleich.`

Zusammenfassung (I)

- Klassen und Objekte:
  - Klassen und Objekte
  - Attribute und Methoden
  - Klassen in UML und Java
  - main-Methode
  - Überladen, Statische Polymorphie
  - Speichermodell (Stack und Heap)
  - Konstruktoren
  - Zugriff auf Komponenten
  - Gleichheit
  - Löschen von Objekten
  - Methode `finalize`
  - Variable Anzahl Argumente

Zusammenfassung (II)

- Vererbung:
  - Idee
  - Mehrere Subklassen
  - Klassenhierarchie
  - Überschreiben
  - Dynamische Polymorphie
  - Abstrakte Klasse
  - Finale Klasse
  - Mehrfachvererbung

- Klassen `String` und `Object`:
  - Bedeutung
  - Wichtige Methoden

Aufgaben (I)

- **Aufgabe 1**

  Im Moodle-Kurs finden Sie eine Reihe von .java-Dateien. Bitte kopieren Sie sich diese in Ihr eigenes Verzeichnis.

  a) Stellen Sie diese als UML-Klassendiagramm dar!

  b) Erweitern Sie das Klassendiagramm um eine Klasse Leitender Angestellter, der sich von einem Angestellten dadurch unterscheidet, dass er einen Bonus bekommt. Erweitern Sie das Diagramm darüber hinaus um eine Klasse Manager. Ein Manager ist leitender Angestellter, der zusätzlich einen Dienstwagen hat. Zu diesem sollen der Wagentyp und das Kennzeichen im System verwaltet werden.

  c) Implementieren Sie die neuen Klassen in Java. Neben den neuen Attributen soll jede neue Klasse auch ein Methode `print()` zur Ausgabe der Attribute besitzen.

Aufgaben (II)

d) Erweitern Sie die Klassen um Konstruktoren, die als Parameterliste jeweils die Attribute der Klasse besitzen.

e) Schreiben Sie eine Klasse Anwendung, in der sie folgende Objekte anlegen und auf der Konsole ausgeben:

- Einen Arbeiter Frank Meier mit der Personalnummer 4711 und einem Stundensatz von 14,67.
- Eine Facharbeiterin Steffi Müller mit der Personalnummer 4712, der Fachrichtung Chemie und einem Stundensatz von 18,33.
- Einen Manager Karl-Heinz Kaiser, der einen Dienstwagen vom Typ Mazda Cabrio mit dem Kennzeichen S-ZZ 999 fährt.

Aufgaben (III)

- **Aufgabe 2**

  Es soll Software zum Arbeiten mit reellen Matrizen erstellt werden.

  Folgendes soll möglich sein:

  - Anlegen von Matrizen über einen Konstruktor, dem als Parameter die Anzahl der Zeilen und Spalten übergeben werden
  - Ausgeben der Matrizen
  - Setzen und Ausgeben von Attributen soll über `set`- und `get`-Methoden erfolgen
  - Skalarmultiplikation, d.h. Multiplikation der Matrix mit einer Zahl
  - Addition zweier Matrizen
  - Legen Sie in der `main`-Methode Objekte an und testen Sie anhand derer Ihre Implementierung.

Aufgaben (IV)

- **Aufgabe 3**

  Schauen Sie in der Java-Dokumentation (Einstieg über https://docs.oracle.com/javase/8/docs/api/) nach
  - was die Methode `regionMatches` der Klasse `String` macht
  - was die Klasse `StringBuffer` macht
  - welche wesentlichen Aufgaben mit den Methoden der Klasse `Arrays` erledigt werden können