



# PROGRAMMIEREN IN **Java**

TINF16 - Sommersemester 2017  
von Johannes Unterstein - [unterstein@me.com](mailto:unterstein@me.com)

   @unterstein



# EINFÜHRUNG

# AGENDA



- Administratives
- Aufbau der Vorlesung
- Geschichte
- Eigenschaften von Java

# ADMINISTRATIVES

- Du?
- <http://dhw-stuttgart.de/~unterstein>
- I I Termine
  - 4x Theorie, 7x Praxis (Kleines Projekt + Klausurvorbereitung)
- Theorie recht zügig, Fragen bitte zwischendurch
- Pausen und Slides
- Email-Verteiler oder Kurssprecher?
- <https://www.jetbrains.com/shop/eform/students>

4

3. - Wir werden während der Theorie Phase etwas „malen“ und auf Papier arbeiten  
- Vor 3 Jahren hat sich digitale Modellierung als sehr unvorteilhaft gezeigt
4. - Klarer Schwerpunkt auf Praxis!
5. - Verwirrung der Theorie wird in der Praxis abgebaut
6. - 90 Minuten - 15 Minuten Pause - 90 Minuten  
- Zwischendurch 5-10 Minuten, bitte Pause einfordern wenn der Kopf raucht!  
- Die Slides kommen nach Bedarf nach dem jeweiligen Vorlesungstag bzw. zwischendurch. -> <http://www.lehre.dhw-stuttgart.de/~unterstein>  
- Die meisten Slides sind mit Anmerkungen versehen.

# ÜBER MICH ...

- 09/2010 B.Sc. an DHBW Stuttgart - Vector Informatik
- 10/2010 - 12/2011 Softwareentwickler - I&I
- Seit 11/2011 Dozent an der DHBW-Stuttgart
- 01/12 - 06/16 Softwareentwickler - Micromata / Polyas
- Seit 07/2016 Softwareentwickler - Mesosphere

# ... UND ÜBER EUCH?

- Wie heißt ihr?
- Was programmiert ihr an der Arbeit?
- Was programmiert ihr zuhause?
- Welche Sprachen habt ihr schon benutzt?

# AUFBAU DER VORLESUNG

- Objektorientierung, Motivation und Konzepte
- Grundlagen der Sprache Java
- Objektorientierung in Java
- Weitere Spracheigenschaften
- Java Klassenbibliotheken
- Benutzeroberflächen auf dem Desktop
- Ausblick Java 9

# MOTIVATION

## 3 Billion Devices Run Java

Computers, Printers, Routers, BlackBerry Smartphones, Cell Phones, Kindle E-Readers, Parking Meters, Vehicle Diagnostic Systems, On-Board Computer Systems, Smart Grid Meters, Lottery Systems, Airplane Systems, ATMs, Government IDs, Public Transportation Passes, Credit Cards, VoIP Phones, Livescribe Smartpens, MRIs, CT Scanners, Robots, Home Security Systems, TVs, Cable Boxes, PlayStation Consoles, Blu-ray Disc Players...

 #1 Development Platform

**ORACLE**

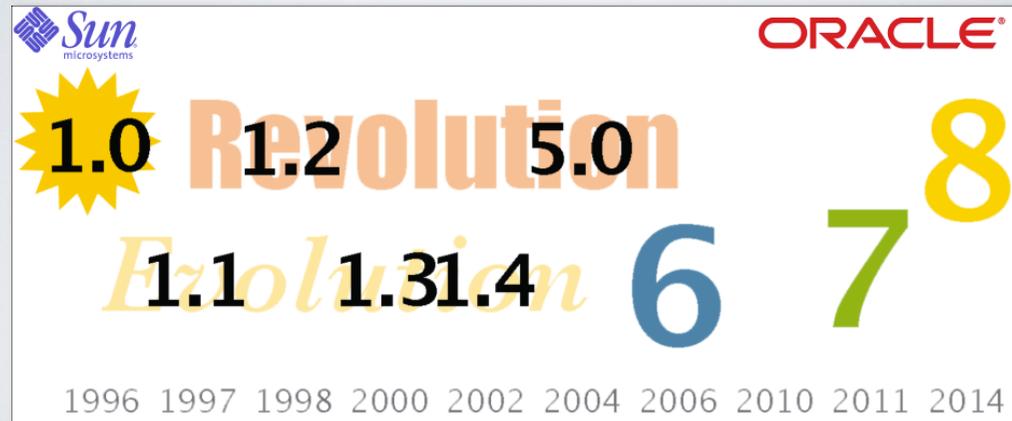
[oracle.com/goto/java](http://oracle.com/goto/java)  
or call +363 1 9631090

Copyright © 2011, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

# GESCHICHTE

- 1991 Sun Microsystems startet geheimes Projekt zur Steuerung von Geräten im Haushalt
  - Portabler Interpreter „Oak“ Bestandteil des Projektes
- Projekt nicht bahnbrechend, „Oak“ gewinnt jedoch durch Applets an Bedeutung
- Oak wird 1995 in Java umbenannt, 1996 erscheint Version 1.0

# VERSIONEN



10 Grafik aus „Java: The Road Ahead“ von Brian Goetz, JAX 2011

- Es gab in der Geschichte von Java eine Reihe von Revolutionären Releases
  - Zum einen natürlich die 1.0, dann aber auch 1.2 (JIT, Swing, Java 2D, D&D, Collections)
  - und natürlich Version 5: Annotations, statische imports, Enumerations, Überarbeitung der Collections API
- Evolutionäre Releases hatten eher den Charakter Java stabiler, sicherer, performanter ... und besser lesbar und somit wartbarer zu machen
- Version 1.0 und 1.1 wurden als JDK 1.0 bzw. JDK 1.1 bezeichnet
- Version 1.2 bis 5.0 wurden als Java Platform 2 bezeichnet und trugen die Namen J2SE 1.2, J2SE 1.3, ...
- Version 5.0 wurde zunächst als 1.5 veröffentlicht, wurde dann aber aus Marketinggründen in ein Major Release umgewandelt
- Version 6 und folgende trugen nur noch ganze Zahlen und der „Plattform 2“ Name wurde entfernt „Java SE 6“
- Ab Version 6 (2006) ist Java Open Source unter der GPL, zuvor lediglich kostenlos
- Version 7 eher schwach auf der Brust (NIO, Project Coin, ..)
- Version 8 hatte Erweiterungen für funktionale Aspekte (Collections, Lambda, Clojures, Default Implementation in Interfaces, ...)
- Version 9 sollte in Q3/2017 wird allerdings noch kontrovers diskutiert. Features: <https://jaxenter.de/java-9-dokumentation-52963>

# EIGENSCHAFTEN

- Java ist eine objektorientierte Sprache
- Java ist robust und dynamisch
- Java ist portabel

Programmiersprache	Source Code
JDK	Entwicklungswerkzeuge, Java Compiler, ...
JRE	Java Bytecode (.class, .jar)
	Java API
	JVM Java Just-in-time-Compilierung
OS	Mac OS, Linux, Windows, Android, ...

||

- Referenzen statt Zeiger
- Automatische Speicherverwaltung
- Strukturierte Fehlerbehandlung
- Multithreading

1. - Das objektorientierte Programmierparadigma liegt Java zugrunde.
2. - Grundidee: Daten und Funktionen, die auf Daten angewandt werden können, möglichst eng zusammenfassen -> Objekt
  - JDK = Java Development Kit
  - JRE = Java Runtime Engine
  - JVM = Java Virtual Machine
  - Code wird in Bytecode übersetzt ...
  - ... und zur Laufzeit von einer Laufzeitumgebung interpretiert
  - Der Java Just-in-time-Compilierung macht zur Laufzeit aus dem Java Bytecode Maschinencode
  - Verschiedene Laufzeitumgebung (Applets, verschiedene Betriebssysteme, ...)
3. - Java ist stark typisiert, daher schon Typfehlerprüfung während der Kompilierung
  - Java ist dynamisch, da zur Laufzeit (und nicht zur Compilezeit) entschieden wird, welcher Code ausgeführt wird („late binding“)
  - Es gibt keine Pointer, sondern man arbeitet eine Abstraktionsebene höher mit Referenzen
  - Es gibt eine automatische Speicherverwaltung durch den Garbage Collector, also kein manuelles Speicher freigeben mehr
  - Es gibt eine strukturierte Fehlerbehandlung. Fehler werden semantisch getrennt vom Applikationscode behandelt und es gibt ein Konstrukt (Exception), welches höherwertig als „Fehlercodes“ ist.
  - Java ist Multithreading fähig. Eine Java Applikation kann also mehrere Threads beinhalten.
    - Z.b.: Ein Thread spielt Musik ab, ein anderer Thread stellt ein animiertes Bild dar und ein dritter Thread validiert eine Eingabe.



# EDITIONEN

- Gleicher Sprachkern, allerdings unterschiedliche Klassenbibliotheken
  - Standard Edition, Java SE (früher J2SE) - „Java“
  - Enterprise Edition, Java EE (früher J2EE)
    - Unternehmensanwendungen (Geschäftslogik)
  - Micro Edition, Java ME
    - Java für embedded Anwendungen (Chipkarten, PDA, ...)



# OBJEKTORIENTIERUNG

Motivation und Konzepte

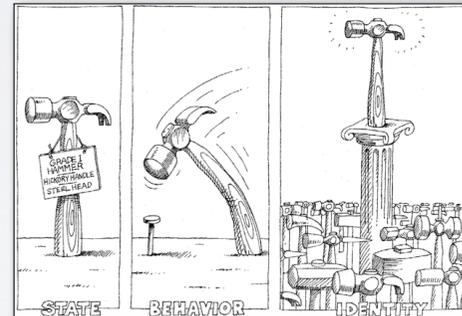
# AGENDA



- Was ist OO?
- Warum OO?
- Konzepte der OO
- Objektorientierte Sprachen
- Übungsaufgabe

# WAS IST OO?

- Die objektorientierte Programmierung versucht eine Teilmenge der realen Welt in Form eines Modells abzubilden
- Daten und Anweisungen werden als Objekt aufgefasst
- Ein Objekt hat einen Zustand, ein Verhalten und eine Identität



Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007

16

1. - Es wird versucht die fachliche Sicht eines Problems (=ein kleiner Ausschnitt der realen Welt) zu modellieren und in Form von Objekten oder Diagrammen zu visualisieren und mit Verhalten zu versehen.
2. - Ein Objekt ist eben mehr als eine Funktion ohne Kontext, wie man sie in der strukturierten Programmierung findet, sondern vielmehr Funktionen auf einem definierten Zustand (also Kontext).
3. - Beispiel Hammer:  
**Zustand:** Erste Klasse Hammer, Edelstahl Kopf, Holzgriff  
**Verhalten:** Kann Nagel in Brett nageln  
**Identität:** Hämmer sind eindeutig identifizierbar (Seriennummer)

# WARUM OO?

- Komplexe Systeme sind die Herausforderungen unserer Zeit
- Komplexe Systeme in großen Teams
- Wiederverwendbare Systeme bzw. Software



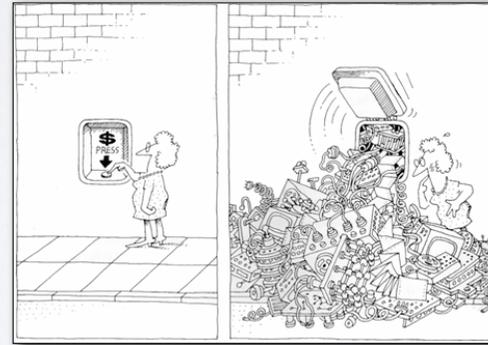
Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007

17

1. - Zerlegung der Komplexität des Problems durch Zerlegung in Teilprobleme mit geringerer Komplexität.  
- Komplexe Systeme, die funktionieren, haben sich aus einfachen Systemen, die funktioniert haben, entwickelt.
2. - Wie Booch in seinem Buch »Objektorientierte Analyse und Design« zitiert, »scheint die strukturierte Programmierung zu versagen, wenn die Applikationen 100 000 Codezeilen oder mehr umfassen«.  
- Diese Art von großen und komplexen Systemen kann nur von großen Teams gebaut werden.  
- Die Objektorientierung adressiert ebenso das Problem, dass viele Menschen zusammen an einer Code Basis arbeiten.  
-> Übersicht  
-> Struktur  
-> Trennung von Code (Separation of Concerns, mehrere Systeme, mehrere Entwickler)
3. - Auch strukturierte Programmierung ermöglicht Wiederverwendung, es handelt sich jedoch um relativ einfache und unabhängige Funktionen.  
- Werden diese Funktionen allerdings komplexer und größer oder stehen in Zusammenhang zueinander wird es schon problematisch.  
- Programmierer durchblickt die Zusammenhänge von globalen Variablen/Methoden nicht auf antrieb.  
- Bei der OOP ist die Wiederverwendung auch von komplexen Systemen möglich.  
- Zum Beispiel durch Vererbung oder Komposition.

# WARUM OO?

- Große Probleme in kleinere Zerlegen  
(Divide et impera)
- Weniger fehleranfällig
- Neue Datentypen
- Leichter anwendbar
- Abbild der realen Welt

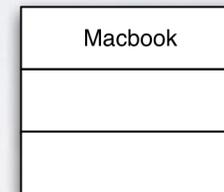


Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007  
18

1. - Zerlegung der Komplexität des Problems durch Zerlegung in Teilprobleme mit geringerer Komplexität.
2. - OOP arbeitet nicht mehr auf zusammenhanglosen Daten und Funktionen, sondern es ergeben sich Semantiken und sinnvolle Trennungen.  
- Innerhalb eines Objektes stehen nur die Daten und Methoden des Objektes zur Verfügung. Die Gefahr der zufälligen Benutzung von Daten fällt weg.
3. - Wiederverwendung von Modellen. Zum Beispiel Typ Koordinate mit X,Y.
4. - Objekte sind wesentlich leichter handhabbar als das zusammenhangslose wirrwarr eines Gesamtkonstruktes.  
- Innerhalb eines Objektes weiß man in der Regel welche Methoden und Daten man verwenden darf.  
- Die Gefahr der zufälligen Benutzung von Daten fällt weg, auch die Übersichtlichkeit steigt.
5. - Wie bereits beschrieben unterstützt OO das natürliche Denkens des Menschen in „Objekten“.  
- Eine Teilmenge der realen Welt wird als fachliches Modell in Form von Objekten modelliert.  
-> Auf dieser Ebene wird Code - nach Verantwortlichkeiten der Objekte - entwickelt.

# ABSTRAKTION

- Trennung von Konzept (Klasse) und Umsetzung (Objekt)
- Ein Objekt ist eine Instanz seiner Klasse
- Eine Klasse beschreibt ...
  - ... wie das Objekt zu bedienen ist (Schnittstelle)
  - ... welche Eigenschaften das Objekt hat
  - ... wie das Objekt hergestellt wird (Konstruktor)

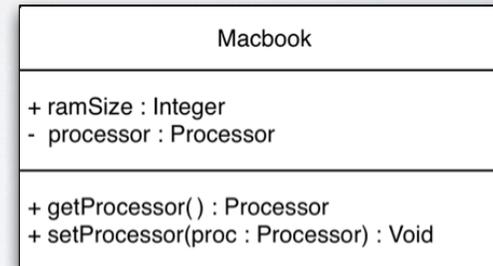


19

1. - In der Objektorientierung wird zwischen dem Konzept und der Umsetzung getrennt.
  - Vergleichbar mit einer Abbildung im Kochbuch und der fertigen Speise. (Konstruktor ist das Rezept)
  - Beispiel: apple.com
  - > Ich gehe in den Store und schaue mir ein Macbook an und kann mir die Eigenschaften anschauen (Klasse)
  - > Ich sage im Store „kaufen“ und veranlasse, dass mir ein neues Macbook (Objekt) erzeugt wird (über einen Konstruktor).
  - Beispiel: Suppe
  - > Ich überlege mir, dass ich Lust auf eine Nudelsuppe habe (Klasse)
  - > Ich schaue in mein Kochbuch und suche nach einem Rezept (Konstruktor) und mit diesem Koche ich die Suppe (Objekt)
- Beispiel Differenzierung Klasse (Schlüsselwort „class“) <-> Objekt:
  - In der Klasse würde die Eigenschaft „Ram Größe“ stehen, in meinem tatsächlichen Objekt kann der Zustand aber nun 8GB oder 16GB sein.

# KAPSELUNG

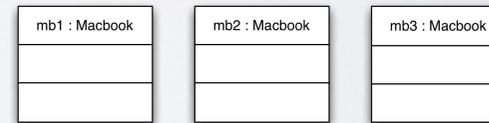
- Zusammenfassung von Daten und Operationen
  - Daten -> Attribute der Klasse (Membervariablen)
  - Operationen -> Methoden der Klasse
- Attribute repräsentieren den Zustand des Objektes





# WIEDERVERWENDBARKEIT

- Abstraktion, Kapselung und Vererbung ermöglichen Wiederverwendung
- Ermöglicht die große Klassenbibliothek in Java
- Erhöhung der Effizienz und Fehlerfreiheit



22

1. - Durch die Tatsache, dass semantisch zusammengehörende Daten auch zusammen gespeichert sind und auch zusammen mit der dazugehöriger Funktionalität abgelegt werden, ergibt sich eine hohe Wiederverwendbarkeit.  
- Ein weiterer großer Aspekt der Wiederverwendung ist die Vererbung (allerdings auch kein Allheilmittel).  
- Sinnvoller Einsatz aus Vererbung und Komposition, näheres zu diesem Thema auf den nächsten Folien.
2. - Durch die Modellierung in Objekten erreicht man bei der von Java mitgelieferten Klassenbibliothek eine sehr große Wiederverwendbarkeit.  
- Wiederverwendbarkeit gab es allerdings auch schon früher, allerdings erleichtert die OOP das Schreiben von wiederverwendbarer Software.
3. - Entwickler müssen nicht jedes Mal das Rad neu erfinden und performante Lösungen entwickeln, wie zum Beispiel das IO Handling geschieht oder Objekte auf grafische Benutzeroberflächen miteinander agieren.  
- Diese Probleme wurden bereits ausreichend in der Java Klassenbibliothek modelliert und gelöst.  
- Dadurch reduzieren sich die Stellen an denen der Entwickler Fehler machen kann und er kann auf bewährten Klassen aufsetzen.  
- Prominentestes Beispiel: Collections und gute Sortierungen

# ASSOZIATION

- Verwendungs- bzw. Aufrufbeziehung
- z.B.: temporäres Objekt in einer Methode (lokale Variable oder Parameter)
- Allgemeinste Art der Beziehung -> Assoziation



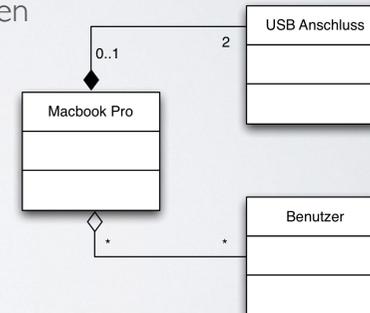
23

Unterscheidung hier innerhalb der Relationen:

- Assoziation
  - Attribut
  - Verwendung (gerichtete Relation) (Kunde "--- benutzt -->" Dienst)
- Aggregation & Komposition (nächste Slides)
- Vererbung (nächste Slides)
- Weiteres Beispiel:
  - Auto benutzt die Straße um zu fahren
  - Koch benutzt Herd zum Kochen

# AGGREGATION & KOMPOSITION

- „Teil-Ganzes“-Beziehungen
  - Aggregation: Teil-Objekt kann alleine existieren
  - Komposition: Teil-Objekt kann nicht alleine existieren
- Kein Unterschied in Java durch Sprachmittel!
- Teil-Objekt wird in Attribut der Klasse gespeichert

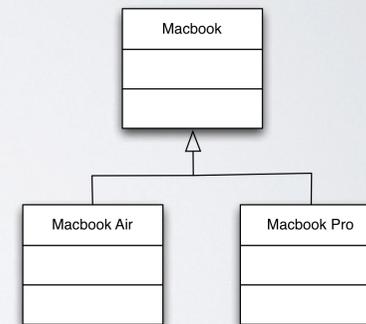


24

1. - Kommt man in seiner Modellierung zu dem Punkt wo man feststellt, dass eine Klasse in der Beziehung "Ist-ein-Teil-von" zu einer anderen Klasse steht, sollte man Aggregation oder Komposition verwenden.
2. - Der Unterschied zwischen Aggregation und Komposition liegt in der Selbstständigkeit der „Teil-Objekt“ Komponente dieser Beziehung.  
- In dem Beispiel kann ein (interner) USB Anschluss nicht ohne dazugehöriges Macbook existieren, ein Benutzer ist (im besten Falle) jedoch auch ohne Macbook existent.
3. - In Java gibt es jedoch keine sprachliche Unterscheidung zwischen diesen beiden Beziehung.  
- Ein Indiz um dies festzustellen ist, ob „Teil-Objekte“ auch außerhalb des „Ganzen-Objekt“ verwendet werden.
4. - In beiden Szenarien wird das „Teil-Objekt“ als Attribut der Klasse des „Ganzen-Objekt“ gespeichert.

# VERERBUNG

- Generalisierung vs. Spezialisierung
- „Ist-ein“ Beziehung
- Abgeleitete Klasse „erbt“ Attribute und Methoden aller super Klassen
- Bildung von Klassenhierarchien
- Keine Mehrfachvererbung in Java



25

- Man spricht im Kontext von Vererbung auch von Generalisierung bzw. Spezialisierung.
  - Die Generalisierung von „Macbook Air“ wäre ein „Macbook“, eine Spezialisierung von „Macbook“ wäre „Macbook Pro“. Schlüsselwort „extends“
    - Generalisierung schränkt die Schnittstelle ein, so dass mehrere Klassen diese Schnittstelle befriedigen können (Klassen-Hierarchie nach oben)
    - Spezialisierung hat eine größere Schnittstelle, so dass nur ein spezialisierteres Objekt diese befriedigen kann (Klassen-Hierarchie nach unten)
- Kommt man in seiner Modellierung zu dem Punkt wo man feststellt, dass eine Klasse in der Beziehung "Ist-ein" zu einer anderen Klasse steht, sollte man Vererbung verwenden  
-> Beispiel: Ein SUV ist ein Auto
- Die abgeleitete Klasse erbt alle Attribute und Methoden von seiner Basisklasse und allen weiteren Superklassen
- Man spricht von Unterklasse/Oberklasse bzw. Subklasse/Superklasse
- Durch das Konstrukt der Vererbung sind große Hierarchien von Klassen möglich
- Booch: „Eine Hierarchie ist eine Anordnung von Ebenen der Abstraktion“ --> Bild von Warum OO (2)
- Es kann sehr feingranular die Funktionalität von Klassen geregelt werden,
  - z.B. Fortbewegungsmittel <- Bodenfahrzeug <- PKW <- SUV <- AudiQFuent
- Es kann aber auch eine unkluge Verwendung der Vererbung zu ungünstigen Erscheinungen kommen, siehe Klasse Stack aus java.util
- Im Gegensatz zu C++ gibt es in Java keine Mehrfachvererbung, da man einer Mehrdeutigkeit vorbeugen wollte und man dadurch gezwungen ist seine Klassenhierarchie sauberer zu planen

# ÜBUNGSaufgabe (15MIN)

- Modellierung DHBW Stuttgart
  - Vorlesungen
  - Professoren, Dozenten, Studenten
  - Vorlesungsräume
  - Standorte
- Bitte sinnvolle Elemente der Klassenhierarchie ergänzen

# ÜBERLADEN & ÜBERLAGERN

## • Überladen

- Innerhalb einer Klasse gibt es mehrere Methoden mit gleichem Namen
- Es zählt Anzahl und Typisierung der Parameter

## • Überlagern/Überschreiben

- Abgeleitete Klassen überschreiben Implementierung der Basisklasse
- Methodenkopf muss identisch sein (fast)

27

1. - Innerhalb einer Klasse kann es mehrere Methoden mit gleichem Namen geben aber unterschiedlichen Parametern.  
- Diese werden anhand der Anzahl und der Typisierungen der Parameter unterschieden.  
- Wichtig: Anhand des Rückgabewertes wird nicht unterschieden und auch nicht wie die Parameter heißen, welche übergeben werden.

Beispiel:

Klasse Auto: Methode `blinkeRechts()` und `blinkeRechts(autobahnmodus: Boolean)`

2. - `@Override` ab Java 5, ab Java 6 auch für Interfaces.  
- Eine abgeleitete Klasse implementiert eine eigene Version einer Methode der Basisklasse (bzw. irgendeiner Superklasse).  
- Wdh.: Methodenkopf = Sichtbarkeit, Name, Parameter, Rückgabewert  
- Sichtbarkeit der Methode in einer Überlagerung kann erweitert werden, aber nicht eingeschränkt.

Beispiel:

Klasse Auto: Abstrakte Klasse Auto hat Methode `blinkeRechts()` und konkrete Klasse OpelManta hat auch Methode `blinkeRechts()`

# POLYMORPHISMUS

- Griechisch: Vielgestaltig
- Variable kann Objekte verschiedener Klassen aufnehmen
  - Allerdings nur in einer Vererbungshierarchie, nur Subklassen
- Late Binding
  - Erst zur Laufzeit wird entschieden, welche Methode aufgerufen wird (überlagerte Methode)

28

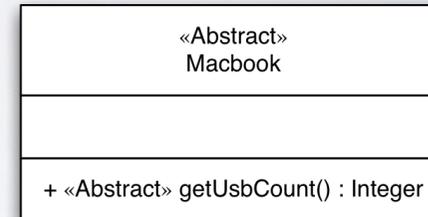
2.
  - Variable vom Typ „Macbook“ kann Objekte der Klassen „Macbook Air“ und „Macbook Pro“ aufnehmen.
  - Umgekehrt ist dies nicht Möglich. Eine Variable vom Typ „Macbook Air“ kann keine Objekte vom Typ „Macbook“ aufnehmen.
  - Dies ist in Verbindung mit Collections und Generics ein sehr mächtiges Konstrukt, später dazu mehr.
3.
  - Late Binding besagt, dass zur Compilezeit noch nicht festgelegt ist, welcher Code tatsächlich ausgeführt wird.
  - Die ausgeführte Methode ist abhängig davon, welche (Sub)Klasse der jeweiligen Variable zugewiesen ist.

Beispiel: Eigentum, blinkende Autos, ...

Beispiel: Klasse Person mit Eigenschaft `lieblingsKuscheltier: Kuscheltier` und Kuscheltier mit der Methode `kuscheln()`  
Nun kann die Methode Kuscheln in vielen Subklassen von Kuscheltier unterschiedlich implementiert sein.

# ABSTRAKTE KLASSEN

- Abstrakte Methode
  - Deklaration der Methode ohne Implementierung
  - Muss überlagert werden
- Abstrakte Klasse
  - Eine Klasse mit mind. 1 abstrakte Methode ist abstrakt
  - Von abstrakten Klassen können keine Instanzen gebildet werden



29

0. Wenn einer Klasse wichtig ist, dass ein bestimmte Funktionalität in der eigenen Klasse vorhanden ist, aber die Klasse in der aktuellen Vererbungshierarchie nicht definieren kann, wie genau diese Funktionalität sein soll, dann würde man üblicherweise zu abstrakten Klassen greifen. Dies besagt:  
-> Die Klasse geht davon aus, dass die Funktionalität da ist, weiß aber im Detail nicht wie sie aussieht und eine erbende Klasse muss dies definieren.
1. - Im Gegensatz zu einer konkret implementierten Methode wird nur die Deklaration der Methode vorgenommen und mit dem Schlüsselwort „abstract“ versehen.  
- In einer Subklasse muss diese Methode zwangsläufig überlagert werden und es ist nicht gestattet dort super.methode() zu verwenden. Zu super, this später mehr
2. - Wenn eine Klasse mindestens eine abstrakte Methode beinhaltet muss die Klasse selbst mit dem Schlüsselwort „abstract“ versehen werden  
- Diese Klasse ist dann nicht mehr instanzierbar und von dieser Klasse muss zwangsläufig geerbt werden, wenn man ein Objekt dieser Klasse haben möchte.  
- Diese Vererbung kann anonym oder in einer benannten Klasse geschehen. Auch hierzu später mehr.  
- Es können also nur Subklassen einer abstrakten Klasse instanziiert werden.

## Beispiel: Macbook

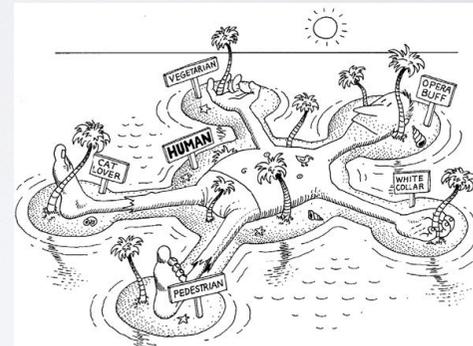
- Ich kann eine abstrakte Umsetzung für ein Macbook schreiben.
- Darin ist festgelegt, dass ich mit der Tastatur Zeichen eingabe kann und diese auf dem Bildschirm angezeigt werden oder, dass ich per USB Schnittstelle eine Maus anschließen kann.
- Es ist aber noch nicht festgelegt, wie viele USB Stecker das Macbook hat oder welche RAM Größe das Macbook hat.
- Wichtig ist nur, dass ich weiß, dass ich meine abstrakte Macbook Klasse fragen kann: „Wie viele USB Stecker habe ich denn?“

## Weiteres Beispiel: Fortbewegungsmittel

- Ich kann nicht in irgendeinen Laden gehen und ein abstraktes Fortbewegungsmittel kaufen, sondern es ist immer ein konkretes Fahrrad, Auto, Motorrad, ...
- Wichtig ist hier nur, dass ich meinem Fortbewegungsmittel den Auftrag erteilen kann: „Bewege mich von A nach B“.

# INTERFACES

- Definiert die öffentliche Schnittstelle einer Rolle
- Definiert ausschließlich public Methoden
- Interfaces werden implementiert, nicht geerbt
- Verwendung wie Klassen



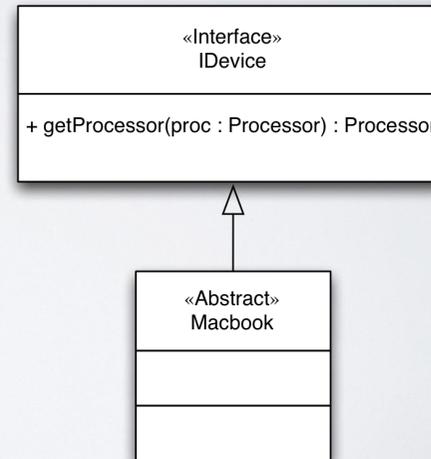
Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007  
30

1.

- Ein Interface (Schlüsselwort „interface“) ist als **Rolle** zu verstehen.
- Ein Interface definiert die Schnittstelle einer Bestimmten Rolle und eine Klasse kann beliebig viele Rollen/Schnittstellen/Interfaces annehmen.
- Unterschied zu einer Klasse:
  - Klasse: Legt das „**Welche Art** ist ein Objekt“ fest. In unserem Beispiel ist das Objekt ein Mensch.
  - Interface: Legt das „**Wie** ist ein Objekt“ fest. In unserem Beispiel ist ein Objekt Vegetarier, ein Opernfan, Katzenliebhaber und noch ein bisschen mehr.
- Damit ein Objekt diese Rolle spielen kann, muss es das Interface implementieren.
- Der Zweck von Interfaces ist die **Spezialisierung**, also die Einschränkung einer gesamten Klasse auf eine bestimmte Sicht, eine Rolle, also ein Interface.
- Der Zweck von abstrakten Klassen hingegen ist die **Generalisierung**, also die Verallgemeinerung von mehreren Implementierungen mit einer gemeinsamen Basis.

# INTERFACES

- Definiert die öffentliche Schnittstelle einer Rolle
- Definiert ausschließlich public Methoden
- Interfaces werden implementiert, nicht geerbt
- Verwendung wie Klassen



31

2. - Ein Interface definiert ausschließlich public Methoden, daher kann der Modifier „public“ auch weggelassen werden.  
 - Weiterhin können in einem Interface Konstanten (Schlüsselwortkombination: static final) festgelegt werden.
  3. - Eine Klasse kann mehrere Interfaces implementieren.  
 - Vorkommen von mehreren Methoden mit gleichem Kopf kein Problem.  
 - Es wird schließlich **keine Implementierung** vorgegeben, sondern es wird nur die Existenz einer Methode gefordert.
  4. - Interfaces können wie Klassen verwendet werden. Parameter, Variablen, ...  
 - Die Verwendung eines Interfaces **reduziert die Sicht** auf ein Objekt auf die Schnittstelle, die im Interface definiert ist.
- Unterschiede abstrakte Klasse zu Interface:
1. Abstrakte Klassen liefern Implementierungen, Interfaces nur Definitionen.
  2. Es kann von einer abstrakten Klasse geerbt werden, aber es können mehrere Interfaces implementiert werden.
  3. Eine abstrakte Klasse dient der Generalisierung, ein Interface dient der Spezialisierung.
- Namenskonventionen:
1. Interface: IDevice, Klasse: Device -> Design nicht so gelungen, aber annehmbar
  2. Interface: Device, Klasse: DeviceImpl -> Noch weniger gelungen
- Frage: Auf was lassen Konvention 1 und 2 schließen?
3. Interface: Sinnvollen fachlichen Namen, Klasse: Ebenfalls sinnvollen fachlichen Namen.  
 -> Interfaces machen erst Sinn, wenn es mehr als eine Implementierung gibt.
- Frage: Was wäre ein sinnvollerer Name für das Interface IDevice?

# KLASSENMETHODEN UND -ATTRIBUTE

- Bzw. statische Methoden und Attribute
- Methoden oder Attribute sind für alle Instanzen gültig
  - bzw. unabhängig von einer konkreten Instanz
- Definition und Aufruf auf Klasse statt auf konkretem Objekt
- Achtung: Statische Methoden können nur auf statische Variablen zugreifen

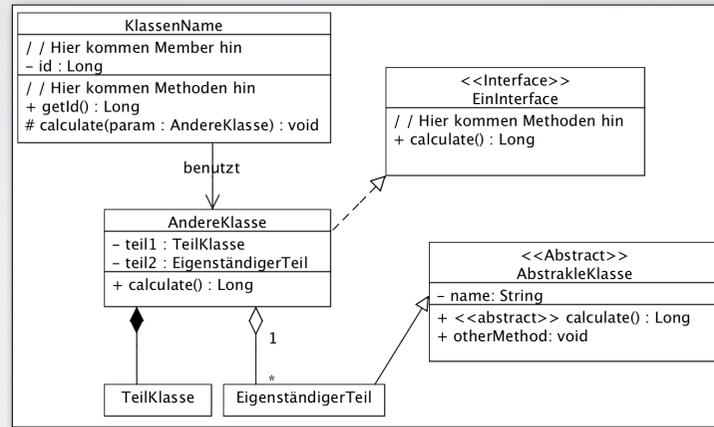
32

2. - Statische Methoden (Schlüsselwort „static“) und Attribute sind für alle Instanzen einer Klasse gültig  
- Beispiel: Klassenzähler, Konstanten, Methoden welche keine Daten zur Verarbeitung brauchen (Factories, Utility Methoden, ..)  
- Beispiel: Auto erbt ja von Fortbewegungsmittel  
    statische Variable für die Anzahl Reifen. Macht kein Sinn jedem Auto einzeln zu sagen, dass es 4 Reifen hat.
4. - Man braucht kein instanziiertes Objekt um den Methodenaufruf zu tätigen, sondern kann mit Klasse.aufruf() die Methode aufrufen  
- Die bekannteste statische Methode in Java dürfte: `public static void main(String args[]) {...}` sein.  
  
- Wdh.: Es gibt 3 unterschiedliche Arten von Attributen (Variablen)
  1. Instanzvariable
  2. Klassenvariable
  3. lokale Variable

# OBJEKTORIENTIERTE SPRACHEN

- OO-Sprache unterstützt OO-Konzepte direkt durch Sprachmittel
- Spezieller Datentyp: Objekt
- In reinen OO-Sprachen (z. B. Smalltalk) ist alles ein Objekt
- Java, C++, C#, Objective C, ... besitzen elementare Datentypen
- Sprachen ohne Vererbung nennt man objektbasiert

# UML



# ÜBUNGSaufgabe (30MIN)

- **Mitarbeiterdatenbank**

- Arbeiter, Angestellter, Manager

- Gemeinsame Daten:

- Personalnummer, Persönliche Daten (Name, Adresse, Geburtstag, ...)

- Arbeiter

- Lohnberechnung auf Stundenbasis

- Stundenlohn

- **Gehaltsberechnung** (Löhne des Unternehmens)

- Angestellter

- Grundgehalt und Zulagen

- Manager

- Grundgehalt und Provision pro Umsatz

- Geschäftsführer (Spezieller Manager)

- Erhält zusätzliche Geschäftsführerzulage