



# PROGRAMMIEREN IN **Java**

TINF13 - Sommersemester 2014  
von Johannes Unterstein - [unterstein@me.com](mailto:unterstein@me.com)



# EINFÜHRUNG

# AGENDA



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

# ADMINISTRATIVES

- Anrede
- II Termine
  - 4x Theorie
  - 7x Praxis (Kleines Projekt + Klausurvorbereitung)
- Theorie recht zügig, Fragen bitte zwischendurch
- Pausen und Slides
- Email-Verteiler oder Kurssprecher?
- Software

4

3. - Wir werden während der Theorie Phase etwas „malen“ und auf Papier arbeiten  
- Letztes Jahr 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 - 20 Minuten Pause - 100 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.dhbw-stuttgart.de/~unterstein>  
- Die meisten Slides sind mit Anmerkungen versehen.
8. - Vor zwei Jahren hatte ich UMLLab im Gepäck, allerdings kam das bei den Studenten nicht so gut an, daher dieses Jahr nur Papier :-).  
- Ich habe IntelliJ Lizenzen für den gesamten Kurs. Daher Entscheidung eclipse <-> IntelliJ, wer möchte.

# ÜBER MICH ...

- 09/2010 B.Sc. an DHBW Stuttgart, Vector Informatik GmbH
- 10/2010 - 12/2011 Softwareentwickler, I&I Internet AG
  - Java Web Applikationen im Hosting & eBusiness Umfeld
- Seit 11/2011 Dozent an der DHBW-Stuttgart
- Seit 01/2012 Softwareentwickler, Micromata GmbH
  - Java Web Applikationen im Kundenauftrag (DHL, VW, ...)



# ... UND ÜBER EUCH?

# AUFBAU DER VORLESUNG

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

# 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...

 Java<sup>™</sup> #1 Development Platform

**ORACLE**

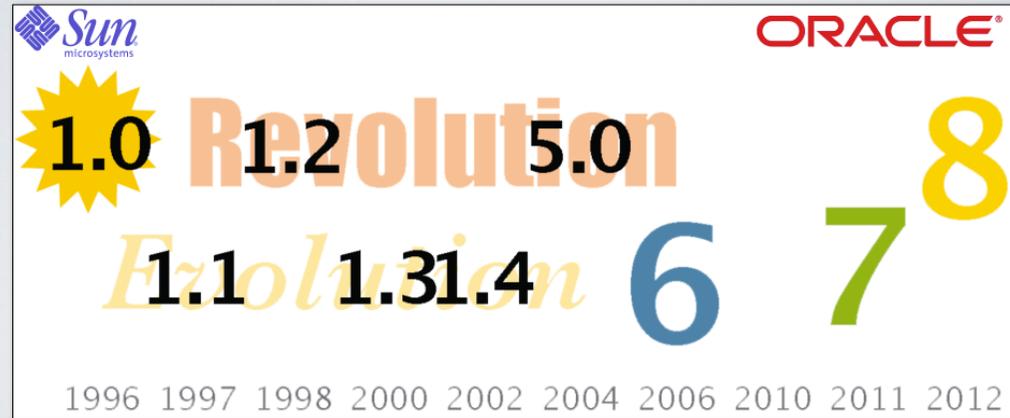
[oracle.com/goto/java](http://oracle.com/goto/java)  
or call +353 1 9031099

Copyright © 2012, 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, Generics, 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
- Roadmap nicht aktuell - Release 8 kam in Q1 2014, weil die Implementierung von Closures noch nicht abgeschlossen waren

# EIGENSCHAFTEN VON JAVA

- 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, ...

11

- 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  
- 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 höher wertiger 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)
  - Enterprise Edition, Java EE (früher J2EE)
    - Unternehmensanwendungen (Geschäftslogik)
  - Micro Edition, Java ME
    - Java für embedded Anwendungen (Chipkarten, PDA, ...)

# LITERATUR

- Handbuch der Java Programmierung, <http://javabuch.de>
  - Guido Krüger, Addison-Wesley
- Java ist auch eine Insel, <http://openbook.galileodesign.de/javainsel7/>
  - Christian Ullenboom, Galileo Press
- [google.com](http://google.com), [stackoverflow.com](http://stackoverflow.com)



# OBJEKTORIENTIERUNG

Motivation und Konzepte

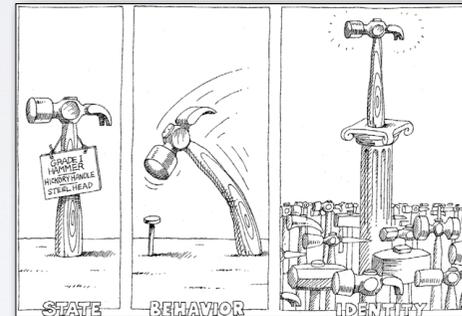
# AGENDA



- 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



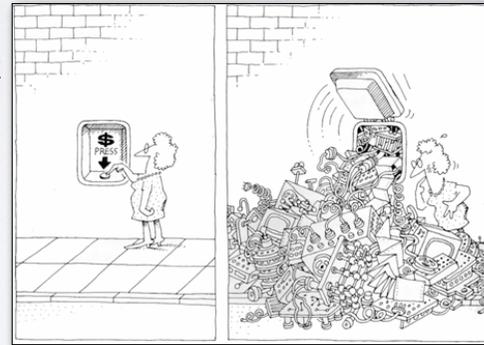
Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007

17

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? (I)

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

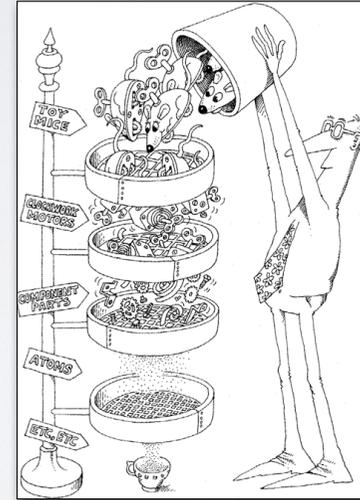


Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007  
18

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 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? (2)

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

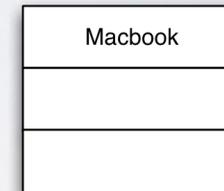


Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007  
19

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.  
- Programmierer 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.  
- Der Programmierer eines Objektes weiß in der Regel welche Methoden und Daten er 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)

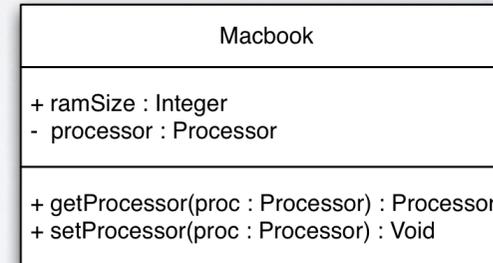


20

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 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.

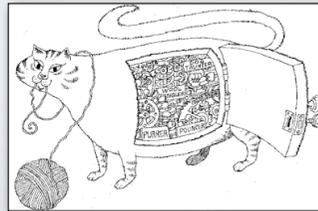
# KAPSELUNG

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



# SICHTBARKEITEN

- Welche Attribute oder Methoden sind für wen sichtbar sind
- Verbergen von Implementierungsdetails



Sichtbarkeit	Eigene Klasse	Subklasse	Package	Alle
private (-)	X	-	-	-
protected (#)	X	X	X	-
public (+)	X	X	X	X
package (-) Standard	X	-	X	-

Hinweis:

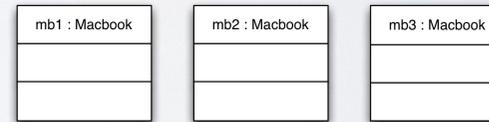
Packages dienen der Zusammenfassung von Klassen zur Strukturierung

22

1. - Nicht jedes Objekt darf alle Attribute oder Methoden von den anderen Objekten sehen bzw. benutzen.  
 - Daher gibt es Sichtbarkeiten „Modifizier“ in Java, welche festlegen wer welche Attribute / Methoden benutzen darf.  
 - Schlüsselworte: private, protected, public
2. - Möglichkeit schaffen, mit der „Außenwelt“ nur über definierte Schnittstellen zu kommunizieren und die eigentliche Implementierung zu verbergen.  
 - Vorteil: Austauschbarkeit, „Außenwelt“ darf nicht auf interne Geheimnisse einer Klasse zugreifen  
 -> Impliziert Entkopplung, wenn ein Objekt vor der „Außenwelt“ die Entwicklungsentscheidung im inneren versteckt, da diese so ggf. auch geändert werden können.  
 - Private ist so eine kleine Mogelpackung. Durch Java Sprachmittel ist es möglich auf private Member von den „Geschwistern“ zuzugreifen.

# WIEDERVERWENDBARKEIT

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



23

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.

# ASSOZIATION

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



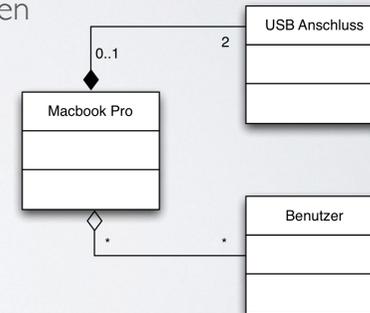
24

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

# 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

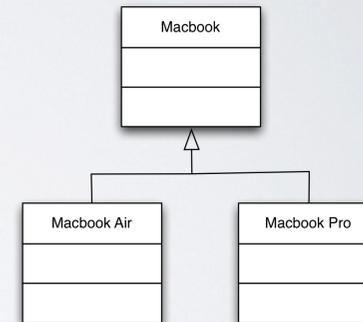


25

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



26

1. - 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“
2. - 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
3. - Die abgeleitete Klasse erbt alle Attribute und Methoden von seiner Basisklasse und allen weiteren Superklassen  
- Man spricht von Unterklasse/Oberklasse bzw. Subklasse/Superklasse
4. - 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 <- AudiQFuenf
5. - 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

# 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 mehr 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.
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.

# 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)

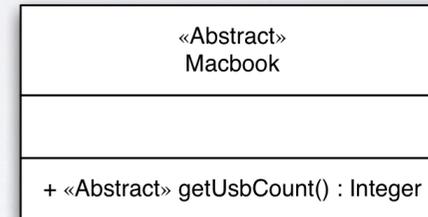
28

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

# ABSTRAKTES

- 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

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

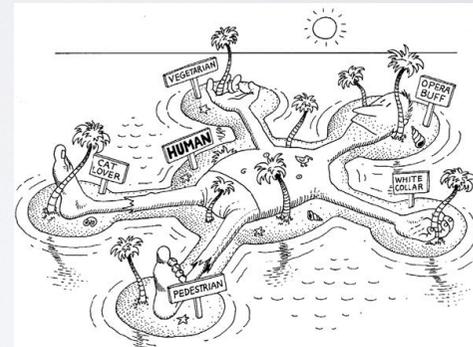
- Ich kann eine abstrakte Umsetzung für ein Macbook schreiben.
- Darin ist festgelegt, dass ich mit der Tastatur Zeichen eingabe kann und diese auf dem Bildschirm angezeigt werden oder, dass ich per USB Schnittstelle eine Maus anschließen kann.
- Es ist aber noch nicht festgelegt, wie viele USB Stecker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

Weiteres Beispiel: Fortbewegungsmittel

- Ich kann nicht in irgendeinem 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 (I)

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

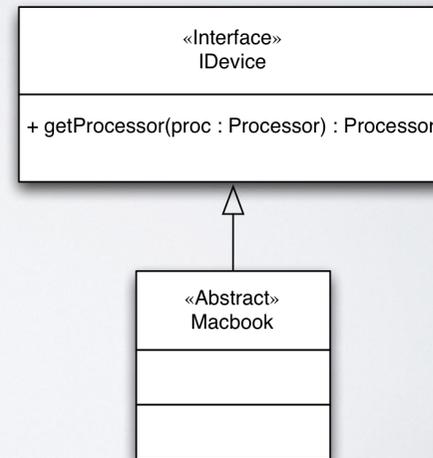


Grafik aus „Object oriented analyses and design“ von Grady Booch, 2007  
30

1. - Ein Interface (Schlüsselwort „interface“) ist als Rolle zu verstehen.  
- 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.

# INTERFACES (2)

- 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 Modifizier „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.
  - Problematisch natürlich, wenn je nach Rolle die Ausführung der Methode unterschiedlich sein soll.  
 Beispiel: Person, Angestellter -> Telefonnummer
4. - Interfaces können wie Klassen verwendet werden. Parameter, Variablen, ...  
 - Die Verwendung eines Interfaces reduziert natürlich 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 -> Gruselig, aber annehmbar
    2. Interface: Device, Klasse: DeviceImpl -> Gruselig^2
 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
- Aufruf auf Klasse statt auf konkretem Objekt
- Achtung: Statische Methoden können nur auf statische Variablen zugreifen

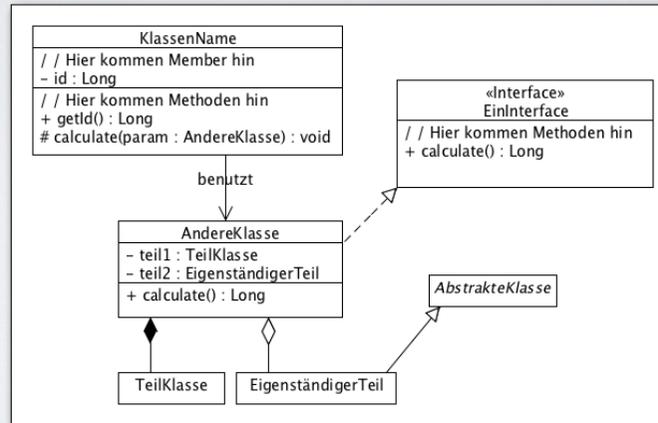
32

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 um
- 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 (15-30MIN)

## • **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



# JAVA

Grundlage der Sprache

# AGENDA

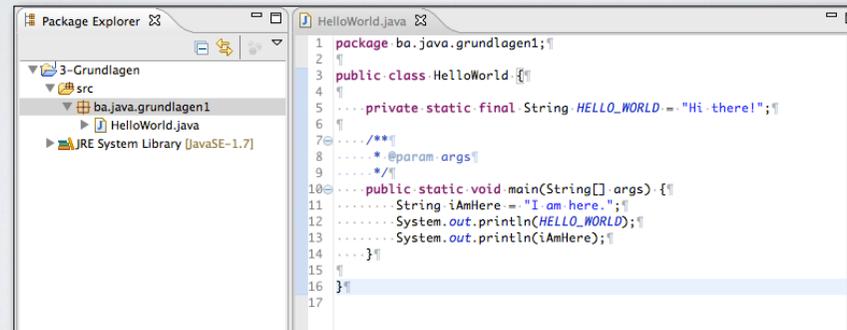


- Allgemeines
- Datentypen
- Ausdrücke
- Anweisungen
- Speichermanagement
- Die Java Umgebung
- Übungsaufgabe

# ALLGEMEINES

- Zeichensatz Unicode
  - Jedes Zeichen mit 16 Bit dargestellt
- Sprachgrammatik an C/C++ angelehnt
  - Datentypen, Ausdrücke und Anweisungen sehr ähnlich, meist sogar identisch

# HELLO!



```
1 package ba.java.grundlagen1;
2
3 public class HelloWorld {
4     ... private static final String HELLO_WORLD = "Hi there!";
5
6
7     ... /**
8     ... * @param args
9     ... */
10    ... public static void main(String[] args) {
11        ... String iAmHere = "I am here.";
12        ... System.out.println(HELLO_WORLD);
13        ... System.out.println(iAmHere);
14    ... }
15
16 }
17
```

1.

- Konventionen:

- KlassenName = Großer Anfangsbuchstabe, CamelCase
- VariablenName = kleiner Anfangsbuchstabe, CamelCase
- Konstante = static final -> Großbuchstaben und underscore

# PRIMITIVE DATENTYPEN

- Primitive Datentypen sind keine Objekte!
- Übergabe durch Wert (Kopie), call by value
- Verfügbare Typen:
  - Ganzzahlige Typen: byte, short, int, long
  - Fließkomma Typen: float, double
  - Zeichentyp: char
  - Logischer Typ: boolean



```
1 package ba.java.grundlagen1;
2
3 public class DatenTypenBeispiel {
4
5     // Deklaration
6     private int meineZahl;
7
8     public DatenTypenBeispiel() {
9         // Initialisierung
10        meineZahl = 5;
11        int meineZweiteZahl = 6;
12    }
```

1. - Primitive Datentypen sind keine Objekte!  
- Dadurch Gewinn an Performance im Vergleich zu „reinen“ OO Sprachen (Was war zum Beispiel eine reine OO Sprache?)  
- Warum sollte eine 1 auch unterschiedlich zu einer 1 sein? Es macht keinen Sinn.

Größen [byte]:

1 byte  
2 short  
4 int  
8 long

4 float  
8 long

2 char

1 boolean (default: false)

# REFERENZTYPEN

- Objekte, Strings, Arrays sowie Enums (ab Java 5)
- Übergabe der Referenz als Wert
- Zeiger != Referenz
- null ist die leere Referenz
- Strings sind Objekte
- Methoden zur Stringmanipulation

```
private void stringBeispiel() {
    ... String ersterString = "Hallo " + concat(String.valueOf(meineZahl));
    ... ersterString += meineZahl;
    ... System.out.println(ersterString);
    ... String zweiterString = "Hallo 55";
    ... System.out.println(zweiterString == ersterString);
    ... System.out.println(zweiterString.equals(ersterString));
    ... System.out.println(ersterString.substring(0, ...
    ... ..ersterString.indexOf("5")).charAt(3));
}
```

41

- Die Referenz selbst wird kopiert. Sie verweist auf das gleiche Objekt.
  - Das Objekt selbst wird nicht übergeben
    - > Die Zuweisung kopiert also lediglich die Referenz auf ein Objekt, das Objekt an sich bleibt unberührt
  - Keine „Dereferenzierung“ wie in C notwendig
  - Gleichheitstest prüft ob zwei Referenzen auf dasselbe Objekt zeigen
- Adresse einer Referenz kann nicht verändert werden, bei einem Zeiger ist dies ja durchaus möglich.
  - Generell gibt es keine Zeigerarithmetik in Java
- Auf null ist prüfbar
- Werden die Strings als Literale dagegen zur Compile-Zeit angelegt und damit vom Compiler als konstant erkannt, sind sie genau dann Instanzen derselben Klasse, wenn sie tatsächlich inhaltlich gleich sind. Wird ein bereits existierender String noch einmal angelegt, so findet die Methode den Doppelgänger und liefert einen Zeiger darauf zurück.
  - Dieses Verhalten ist so nur bei Strings zu finden, andere Objekte besitzen keine konstanten Werte und keine literalen Darstellungen.
  - Mehr zu Strings gibt es allerdings später.
  - Die korrekte Methode, Strings auf inhaltliche Übereinstimmung zu testen, besteht darin, die Methode equals der Klasse String aufzurufen.

Beispiel:  
 Hallo 55  
 false  
 true  
 |

# ARRAYS

- Arrays sind Objekte (Übergabe der Referenz als Wert)
- Verwendung und Zugriff analog zu C / C++

```
private void arrayBeispiel() {  
    ....// Deklaration  
    ....int[] meinArray;  
    ....int auchMeinArray[]; // Schreibweise vermeiden!  
    ....// Initialisierung  
    ....meinArray = new int[5];  
    ....meinArray[0] = 1;  
    ....int[] literate = {1,2,3}; // literale Initialisierung  
}
```

# CASTING

- Primitive Typen
  - Verlustfrei in nächst größeren Datentyp, nicht umgedreht
  - In nächst kleineren Datentyp muss explizit gecastet werden
- Referenztypen
  - Upcast & Downcast
  - Jeder Cast wird geprüft -> Typsicherheit
  - Sowohl Up- als auch Downcasts sind verlustfrei

```
private void castBeispiel() {
    AudiQFuenf q5 = new AudiQFuenf();
    BodenFahrzeug fahrzeug = q5;
    Pkw pkw = (Pkw) fahrzeug;
    // Geht das folgende?
    Pkw pkw2 = (Pkw) new BodenFahrzeug();
}
```

43

1. - Ein „eins“ 1 als Long, ist genauso gut eine 1 als Integer oder als Short. Diese Casts sind verlustfrei durchführbar.
  - Bei Referenztypen sieht das ganze schon etwas anders aus.

Fortbewegungsmittel <- BodenFahrzeug <- PKW <- SUV <- AudiQFuenf

2. - Bei einem Upcast geht es in der Vererbungshierarchie aufwärts. Zum Beispiel von AudiQFuenf zu SUV
  - Bei einem Downcast geht es in der Vererbungshierarchie abwärts. Zum Beispiel von Fortbewegungsmittel nach BodenFahrzeug
  - Vorsicht: Was kann bei einem Downcast passieren?
    - Daher wird auch geprüft
3. - Wenn ein valider Cast vorliegt, kann sowohl beim Up- als auch beim Downcast mit einem vollwertigen Objekt weitergearbeitet werden.
  - Natürlich mit dem Klasseninterface, welches die gecastete Klasse besitzt.

# OPERATOREN

- Arithmetische Operatoren (+, -, \*, /, ++, ...)
- Relationale Operatoren (==, !=, <=, >=, ...)
- Logische Operatoren (!, &&, ||, or, and, ...)
- Bitweise Operatoren (&, |, ^, ...)
- Zuweisungsoperatoren (=, +=, -=, ...)
- Fragezeichen Operator:      a ? b : c      (Wenn „a“, dann „b“, sonst „c“)
- Type-Cast Operator:      (type) a

# OPERATOREN FÜR OBJEKTE

- String-Verkettung:  $a + b$
- Referenzgleichheit:  $==$  bzw.  $!=$
- instanceof-Operator:  $a \text{ instanceof } b$
- new-Operator: Erzeugen von Objekten (auch Arrays)
- Member- und Methodenzugriffe

```
private void instanceOfBeispiel() {  
    ... Pkw pkw = new Pkw();  
    ... System.out.println(pkw instanceof Pkw);  
    ... System.out.println(pkw instanceof BadenFahrzeug);  
    ... System.out.println(pkw instanceof AudiQFuenf);  
    ... int anzahlBlinker = pkw.anzahlBlinker;  
    ... pkw.blinkeRechts();  
}
```

1. - Es genügt, wenn a oder b ein String ist!
3. - true wenn a eine Instanz der Klasse b ist (oder einer ihrer Unterklassen)

# VERZWEIGUNGEN

```
private void verzweigung() {  
    ....int meineZahl = 5;..  
    ....// If/Else-Anweisung..  
    ....if (meineZahl == 5) {..  
    .....// tue etwas..  
    ....} else if (meineZahl == 6) {..  
    .....// tue etwas anderes..  
    ....} else {..  
    .....// tue was ganz anderes..  
    ....}..  
    ....// Switch-Anweisung..  
    ....switch (meineZahl) {..  
    .....case 5:..  
    .....// tue etwas..  
    .....break;..  
    .....case 6:..  
    .....// tue etwas anderes..  
    .....break;..  
    .....default: // tue etwas ganz anderes..  
    ....}..  
}..  
}
```

# SCHLEIFEN

```
private void schleifen() {  
    .....int meineZahl = 0;   
    .....// While-Schleife   
    .....while (meineZahl < 5) {   
    .....    // tue etwas   
    .....    meineZahl++;   
    .....}   
    .....   
    .....meineZahl = 0;   
    .....// Do-While-Schleife   
    .....do {   
    .....    // tue etwas, z.B.   
    .....    System.out.println(meineZahl);   
    .....    meineZahl++;   
    .....} while (meineZahl < 0);   
    .....   
    .....// For-Schleife   
    .....// for(init; test; update)   
    .....// for(int x = 0, y = 1; x < 5 && y == 1; x++)   
    .....for (int x = 0; x < 5; x++) {   
    .....    // tue etwas   
    .....}   
    .....   
    .....int[] array = { 0, 1, 2, 3, 4 };   
    .....// For-Each-Schleife   
    .....for (int x : array) {   
    .....    // tue etwas   
    .....}   
    .....}   
}
```

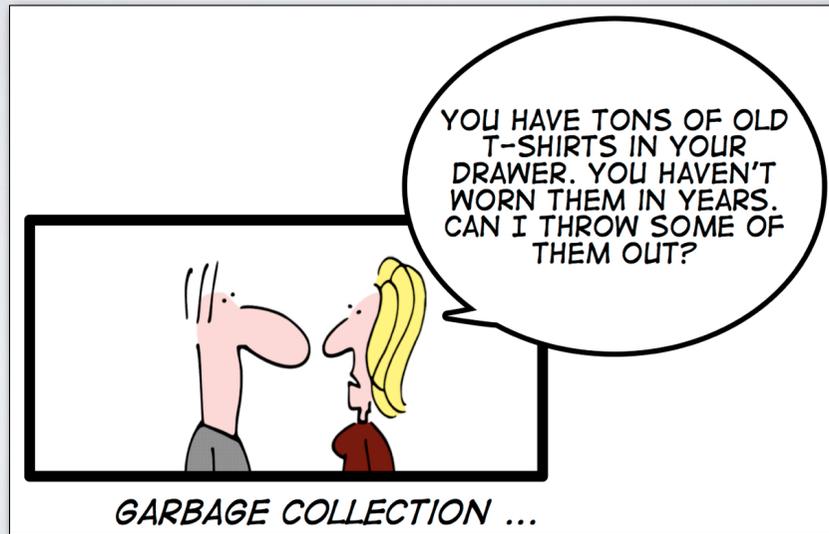
# SPEICHERMANAGEMENT GARBAGE COLLECTION

- Die Müllabfuhr in Java
- Automatisches Speichermanagement
- GC sucht periodisch nicht mehr verwendeten Referenzen
- Achtung: GC befreit nicht von mitdenken!
- Speichermanagement, Heap, Stack und new

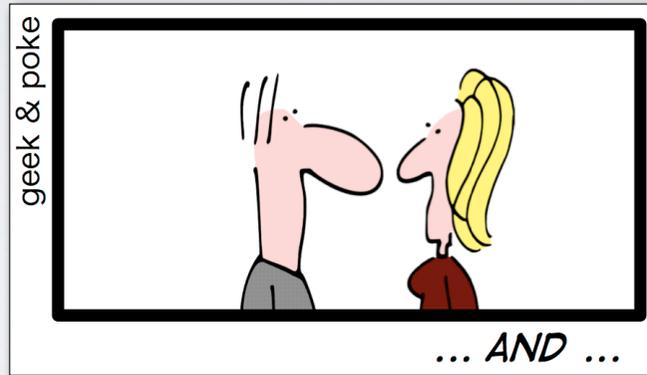
48

1. - Java verfügt über ein automatisches Speichermanagement.
  2. - Dadurch braucht man sich als Java-Programmierer nicht um die Rückgabe von Speicher zu kümmern, der von Referenzvariablen belegt wird.  
- Reservierter Speicher muss nicht mehr explizit frei gegeben werden wie in C/C++.
  3. - Ein mit niedriger Priorität im Hintergrund arbeitender Garbage Collector sucht periodisch nach Objekten, die nicht mehr referenziert werden, um den durch sie belegten Speicher freizugeben.
  4. - Speicher muss manchmal durch setzen von Objekten auf null frei gegeben werden. Bsp.: meinObjekt = null
  5. - Auf dem Heap wird alles abgelegt, was zur Laufzeit mit „new“ erzeugt wird.  
- Der Heap Space einer JVM ist begrenzt, damit ein Java-Programm nicht beliebig viel Speicher vom Betriebssystem abgreifen kann. (Vordefiniert 64mb)  
- Das Java Speichermanagement kümmert sich automatisch darum, dass bei einem „new“ Aufruf genügend Speicher für das gesamte Objekt reserviert wird.  
- Den Stack nutzt die JVM um dort z.B. lokale Variablen abzulegen, welche zuvor z.B. vom Heap geladen wurden. Diese werden aber auch wieder vom Stack entfernt.  
-> Das wird im Laufe des Studiums noch sehr detailliert erklärt ;-)
- Wdh.: Was ist eine Instanzvariable, was ist eine lokale Variable und was ist eine Klassenvariable?

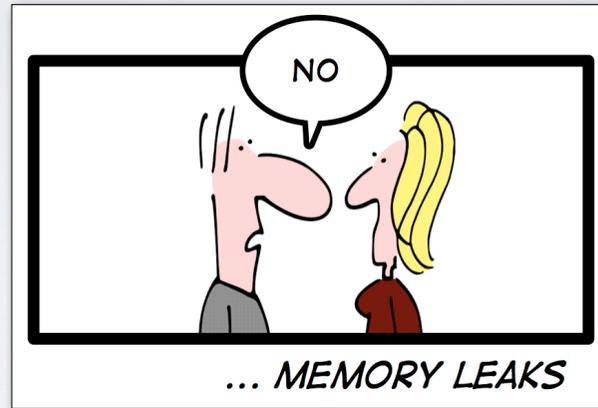
# GARBAGE COLLECTION



# GARBAGE COLLECTION



# GARBAGE COLLECTION



Grafik von <http://geekandpoke.typepad.com/geekandpoke/2010/05/simply-explained.html>

# DIE JAVA UMGEBUNG

- JRE (Java Runtime Environment)
  - Ermöglicht das Ausführen von Java-Programmen „java“
- JDK (Java Development Kit)
  - Ermöglicht das Erstellen (Kompilieren) und Ausführen
- Java Compiler „javac“
- IDE nimmt einem diese Arbeit ab

52

1. - Bestandteil des JRE ist u.A. der Java Interpreter  
- Befehl: java
2. - Bestandteil des JDK ist u.A. der Java Compiler und der Java Interpreter  
- Befehl Compiler: javac
3. - Javac macht aus \*.java Dateien \*.class Dateien, welche die Repräsentation des Codes in Bytecode darstellen.  
- Wdh.: Was ist noch mal Bytecode und warum ist das in Java so?  
- Die \*.class Dateien können später ausgeführt werden  
- Der CLASSPATH zeigt dem Compiler und Interpreter an, in welchen Verzeichnissen er nach Klassendateien suchen soll.
4. - Laut Lehrplan sollt ihr einen „nackten“ Editor benutzen. Empfände ich aber als nicht sinnvoll, da nicht Praxisrelevant.  
- Ich verwende in dieser Vorlesung intelliJ/eclipse als IDE bzw. Sublime Text 2 als „schnellen Editor für Zwischendurch“.  
- Fühlt euch frei andere IDEs zu benutzen, sofern ihr euch damit auskennt!  
-> „Ihr müsst damit arbeiten“  
- Support bekommt ihr nur zu eclipse und intelliJ ;). Wenn ihr euch also nicht sicher seit, bitte eclipse oder intelliJ benutzen.

# ÜBUNGSaufGABE WISSENSTRANSFER

- Implementierung eines Beispiels aus der Informatik-Vorlesung “Algorithmen und Datenstrukturen”
- Beispiel: Quicksort von Arrays
- 30-45 Minuten
- Hinweis: Verwendet auch den Debugger eurer IDE