DHBW Stuttgart
Duale Hochschule Baden-Württemberg

PROGRAMMIEREN IN Java

TINF18 - Sommersemester 2019

von Johannes Unterstein - unterstein@me.com

@unterstein

EINFÜHRUNG

Grafik von http://it.wikipedia.org/wiki/File:Duke-Guitar.png

AGENDA

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

ADMINISTRATIVES

- Du?
- http://dhbw-stuttgart.de/~unterstein
- 11 Termine
  - 4-5x Theorie, 6-7x Praxis
- Pausen und Slides
- Email-Verteiler oder Kurssprecher?
- https://www.jetbrains.com/shop/eform/students

3. - Klarer Schwerpunkt auf Praxis!
4. - Verwirrung der Theorie wird in der Praxis abgebaut
5. - Zwischendurch 5-10 Minuten, bitte Pause einfordern wenn der Kopf raucht!
   - Die Slides kommen nach Bedarf nach dem jeweiligen Vorlesungstag bzw. zwischendurch. -> http://wwwlehre.dhbw-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 - 1&1
- Seit 11/2011 - Dozent an der DHBW-Stuttgart
- 01/2012 - 06/2016 - Micromata / Polyas
- 07/2016 - 01/2018 - Mesosphere
- Seit 02/0218 - Neo4j

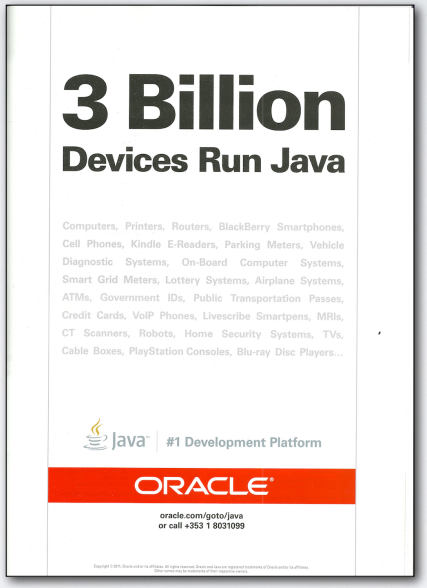
... UND ÜBER EUCH?

- 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
- Java 9 / Java 10 / Java 11 / Java 12 ¯_(ツ)_/¯

MOTIVATION



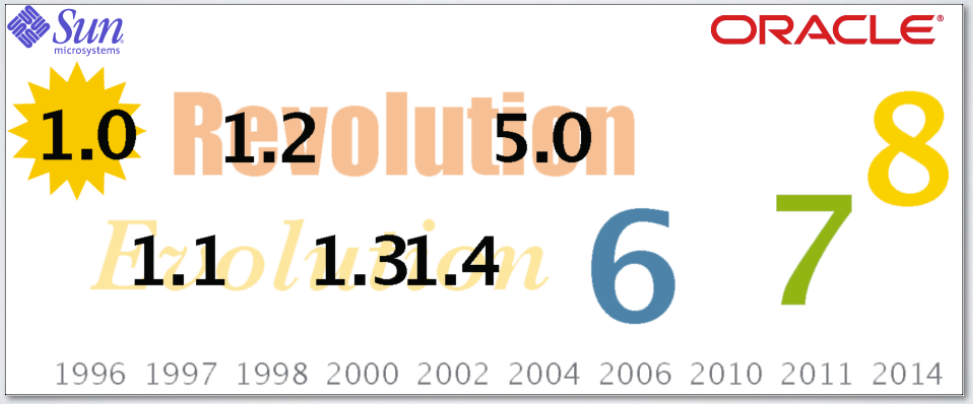
8

Oracle Anzeige in JAX 2011 Magazine

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



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, sichererer, 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 erschienen in Q3/2017 und beinhaltet Projekt Jigsaw. Features: https://jaxenter.de/java-9-dokumentation-52963
- Ab jetzt wird alle 6 Monate eine Java Version released!
- Aktuelle Version Java 12, erschienen im März 2019

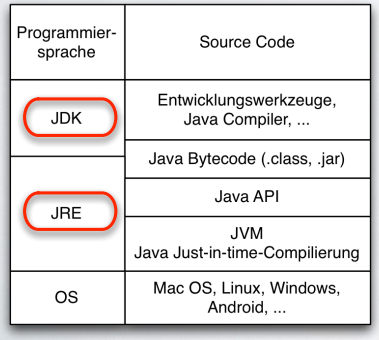
VERSIONEN

- März 2014 Java 8 LTS
- September 2017 Java 9
- März 2018 Java 10
- September 2018 Java 11 LTS
- März 2019 Java 12

- Nach dem Release von Java 9 ist man zu einem Releasezyklus von 6 Monaten über gegangen um kontinuierlich Verbesserungen zu veröffentlichen und nicht wieder so lange Zeiträume ohne Releases zu haben.
- Da nun alle 6 Monate eine neue Version von Java erscheint sind wahrscheinlich auch keine Revolution Releases mehr zu erwarten
- Vollständige Liste der Versionen und ihrer Features: https://en.wikipedia.org/wiki/Java_version_history
- Java 8 war das letzte freie Update für kommerzielle Zwecke.

EIGENSCHAFTEN

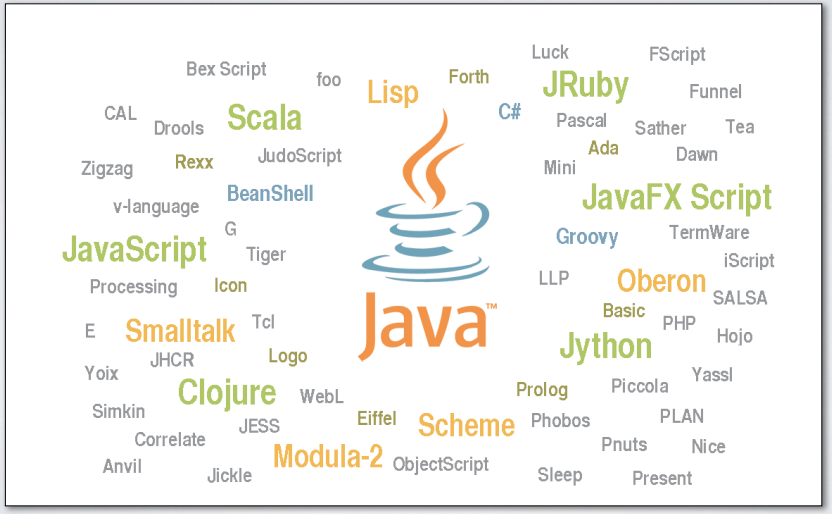
- Java ist eine objektorientierte Sprache
- Java ist portabel



- Java ist robust und dynamisch
  - Referenzen statt Zeiger
  - Automatische Speicherverwaltung
  - Strukturierte Fehlerbehandlung
  - Multithreading

1. - Das objektorientierte Programmierparadigma liegt Java zugrunde.
   - Grundidee: Daten und Funktionen, die auf Daten angewandt werden können, möglichst eng zusammenfassen -> Objekt
2. - 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 Abstaktionsebene 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.

DIE PLATTFORM JVM



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

- Mittlerweile wird Java nicht mehr als die „eine" einzige Sprache verstanden, sondern eher als die „eine" starke Plattform. Microsoft bestreitet diesen Weg des Plattform Gedankens mit .NET und der Common Interface Language schon viel länger.

EDITIONEN

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

OBJEKTORIENTIERUNG

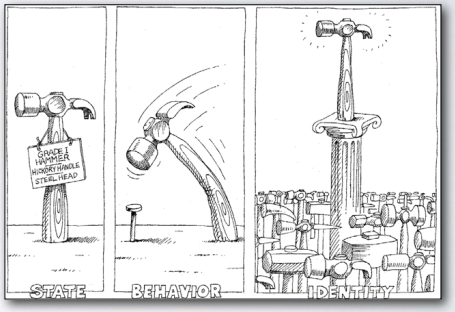
Motivation und Konzepte

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

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 viel mehr 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)

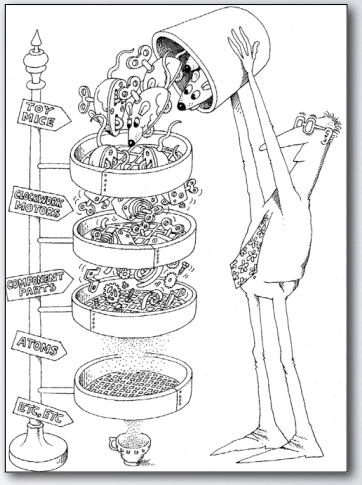
WAS IST OO?

Zustand, Verhalten, Identität ?

- Beispiel Auto: Volvo V70
Zustand: 4 Räder, parkt, Farbe grau
Verhalten: Kann Personen von A nach B bringen
Identität: Seriennummer V70-4711

WARUM OO?

- Komplexe Systeme sind die Herausforderungen unserer Zeit
- Große Probleme in kleinere Zerlegen (Divide et impera)
- Komplexe Systeme in großen Teams
- Wiederverwendbare Systeme bzw. Software

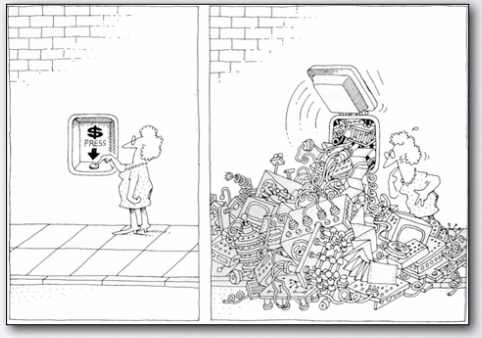


Grafik aus „Object oriented analyses and design" von Grady Booch, 2007

1. - Komplexe Systeme, die funktionieren, haben sich aus einfachen Systemen, die funktioniert haben, entwickelt.
   - 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«.
2. - Zerlegung der Komplexität des Problems durch Zerlegung in Teilprobleme mit geringerer Komplexität.
3. - 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)
4. - 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 anhieb.
   - Bei der OOP ist die Wiederverwendung auch von komplexen Systemen möglich.
     - Zum Beispiel durch Vererbung oder Komposition.

WARUM OO?

- Weniger fehleranfällig
- Neue Datentypen
- Leichter anwendbar
- Abbild der realen Welt

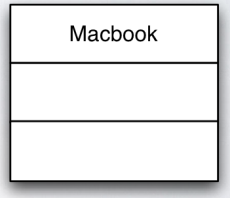


Grafik aus „Object oriented analyses and design" von Grady Booch, 2007

1. - OOP arbeitet nicht mehr auf zusammenhangslosen Daten und Funktionen, sondern es ergeben sich Semantik 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.
2. - Wiederverwendung von Modellen. Zum Beispiel Typ Koordinate mit X,Y.
3. - 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.
4. - 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

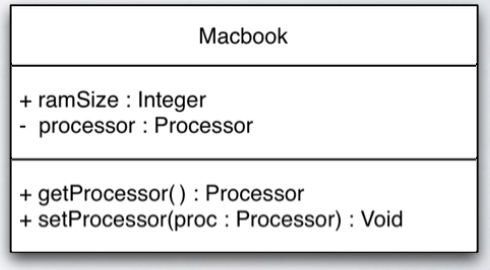
- Trennung von Konzept (Klasse) und Umsetzung (Objekt)
- Ein Objekt ist eine Instanz seiner Klasse
- Eine Klasse beschreibt ...
  - ... wie das Objekt zu bedienen ist (Schnittstelle)
  - ... welche Eigenschaften das Objekt hat
  - ... wie das Objekt hergestellt wird (Konstruktor)



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

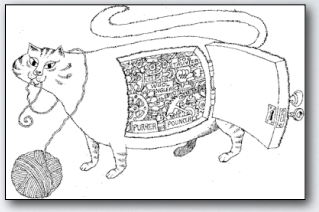


ÜBUNGSAUFGABE (5 MIN)

- Nenne alle **Klassen** in folgendem Text:
- Ein Rechenzentrum besteht aus mehreren Räumen. Ein Raum besteht aus mehreren Racks und in einem Rack sind mehrere Server. In einem Server können bis zu 8 CPU, bis zu 8 Festplatten und bis zu 12 Speicherriegel verbaut werden. Es gibt HighAvailability-Speicherriegel, welche nur in spezielle Server passen.

SICHTBARKEITEN

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



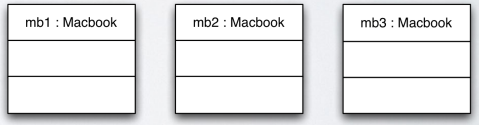
Sichtbarkeit	Eigene Klasse	Subklasse	Package	Alle
private (-)	X	-	-	-
protected (#)	X	X	X	-
public (+)	X	X	X	X
package (~) Standard	X	-	X	-

Hinweis:
Packages dienen der Zusammenfassung von Klassen zur Strukturierung

1. - Nicht jedes Objekt darf alle Attribute oder Methoden von den anderen Objekten sehen bzw. benutzen.
   - Daher gibt es Sichtbarkeiten „Modifier“ 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. „Information Hiding“
     - Vorteil: Austauschbarkeit, „Außenwelt“ darf nicht auf interne Geheimnisse einer Klasse zugreifen
     - Es gibt definierte Schnittellen den Zustand zu verändern und nicht auf direktem Weg der internen Informationen.
   - Private ist so eine kleine Mogelpackung. Durch Java Sprachmittel ist es möglich auf private Member von den „Geschwistern“ zuzugreifen.
3. Entkopplung: Wenn ein Objekt die Implementierungsdetails vor der Außenwelt versteckt, kann es diese Details später ändern, ohne dass die Außenwelt etwas mitbekommt.

WIEDERVERWENDBARKEIT

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



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

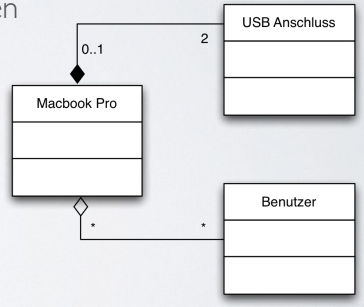
Unterscheidung hier innerhalb der Relationen:

- 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

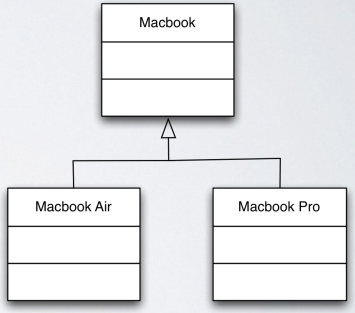
- „Teil-Ganzes“-Beziehungen
  - Aggregation: Teil-Objekt kann alleine existieren
  - Komposition: Teil-Objekt kann nicht alleine existieren
- Kein Unterschied in Java durch Sprachmittel!
- Teil-Objekt wird in Attribut der Klasse gespeichert



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



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“
     - 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)
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

   - Es kann aber auch eine unkluge Verwendung der Vererbung zu ungünstigen Erscheinungen kommen, siehe Klasse Stack aus java.util
5. - 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 (10 MIN)

- Ergänze die letzte Übung um Relationen:
- Ein Rechenzentrum besteht aus mehreren Räumen. Ein Raum besteht aus mehreren Racks und in einem Rack sind mehrere Server. In einem Server können bis zu 8 CPU, bis zu 8 Festplatten und bis zu 12 Speicherriegel verbaut werden. Es gibt HighAvailability-Speicherriegel, welche nur in spezielle Server passen.

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)

1. - Innerhalb einer Klasse kann es mehre 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)

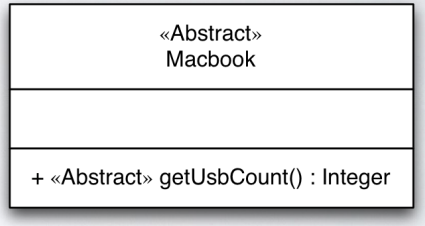
2. - Variable vom Typ „Macbook“ kann Objekte der Klassen „Macbook Air“ und „Macbook Pro“ aufnehmen.
   - Umgekehrt ist dies nicht Möglich. Eine Variable vom Typ „Macbook Air“ kann keine Objekte vom Typ „Macbook“ aufnehmen.
   - Dies ist in Verbindung mit Collections und Generics ein sehr mächtiges Konstrukt, 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



Ziel: Abstrahieren einer Klassenhierarchie, auch wenn noch nicht alle benötigten Informationen vorhanden sind.

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.

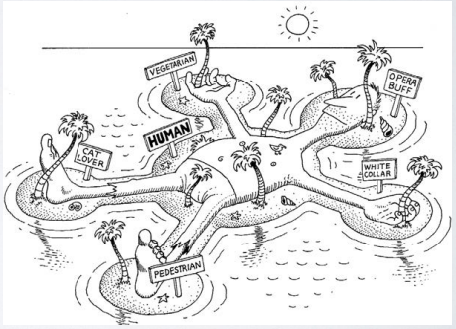
1. - Im Gegensatz zu einer konkret implementierten Methode wird nur die Deklaration der Methode vorgenommen und mit dem Schlüsselwort „abstract“ versehen.
   - In einer Subklasse muss diese Methode zwangsläufig überlagert werden und es ist nicht gestattet dort die Elternmethode (super.method()) zu verwenden. Zu super und 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 instanziierbar 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 eingebe 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



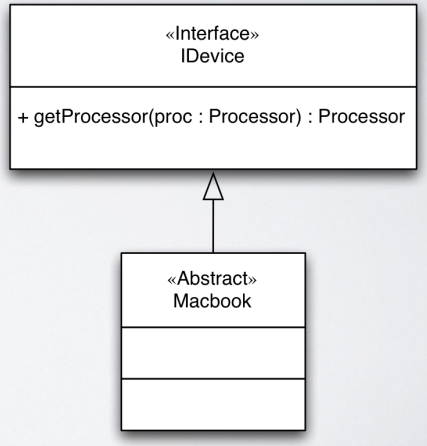
Grafik aus „Object oriented analyses and design" von Grady Booch, 2007

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



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?

ÜBUNGSAUFGABE (5 MIN)

- Gegeben
  - Abstrakte Klasse: Fahrzeug, Methoden: bewege(a, b)
  - Interface Groesse, Methoden: breite(), hoehe(), laenge()
  - Interface Stabelbar, Methoden: stapelbar()
  - Klasse Auto erbt von Fahrzeug und implementiert Groesse und Stabelbar, Methoden: hupe(), blinke()
- Welche Methoden sind jeweils sichtbar?

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

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.

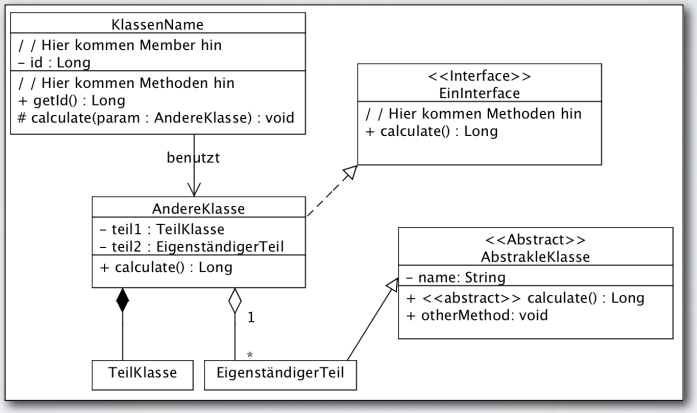
- Wdh.: Es gibt 3 unterschiedliche Arten von Attributen (Variablen)
  1. Instanzvariable
  2. Klassenvariable
  3. lokale Variable

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

5. Beispiel: VB, Pascal

UML


KlassenName
/ / Hier kommen Member hin
– id : Long
/ / Hier kommen Methoden hin
+ getId() : Long
# calculate(param : AndereKlasse) : void
benutzt
<<Interface>>
EinInterface
/ / Hier kommen Methoden hin
+ calculate() : Long
AndereKlasse
– teil1 : TeilKlasse
– teil2 : EigenständigerTeil
+ calculate() : Long
<<Abstract>>
AbstrakleKlasse
– name: String
+ <<abstract>> calculate() : Long
+ otherMethod: void
1
*
TeilKlasse
EigenständigerTeil

ÜBUNGSAUFGABE (15MIN)

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

Mit Tool http://umlet.com/ oder auf Papier

ÜBUNGSAUFGABE (30MIN)

- **Mitarbeiterdatenbank**
  - Arbeiter, Angestellter, Manager
  - Gemeinsame Daten:
    - Personalnummer, Persönliche Daten (Name, Adresse, Geburtstag, ...)
  - Arbeiter
    - Lohnberechnung auf Stundenbasis
    - Stundenlohn
  - Angestellter
    - Grundgehalt und Zulagen
  - Manager
    - Grundgehalt und Provision pro Umsatz
  - Geschäftsführer (Spezieller Manager)
    - Erhält zusätzliche Geschäftsführerzulage
- **Gehaltsberechnung** (Löhne des Unternehmens)

Mit Tool http://umlet.com/ oder auf Papier

JAVA

Grundlage der Sprache

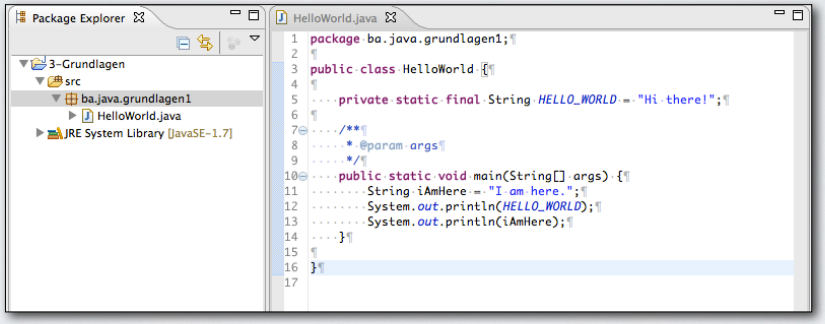
AGENDA

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

ALLGEMEINES

- Zeichensatz Unicode
  - Jedes Zeichen mit 16 Bit (2 Byte) dargestellt
- Sprachgrammatik an C/C++ angelehnt
  - Datentypen, Ausdrücke und Anweisungen sehr ähnlich, meist sogar identisch

HELLO!



1. - Code-Konventionen (Wichtig!):
   - KlassenName = Großer Anfangsbuchstabe, CamelCase
   - VariablenName = kleiner Anfangsbuchstabe, CamelCase
   - Konstante = static final -> Großbuchstaben und underscore

HELLO!

1. - Code-Konventionen (Wichtig!):
   - KlassenName = Großer Anfangsbuchstabe, CamelCase
   - VariablenName = kleiner Anfangsbuchstabe, CamelCase
   - Konstante = static final -> Großbuchstaben und underscore

ÜBUNGSAUFGABE ERSTES PROGRAMM

- Wir implementieren „Hello World“ **ohne** IDE
  - TextEditor eurer Wahl
  - Befehl javac
  - Befehl java
- **15 Minuten**

- Wir verpacken unsere Anwendung in eine **jar** Datei
  - jar cf hello.jar Hello.class
  - java -cp hello.jar Hello
  - java -jar hello.jar ???
- **10 Minuten**

ÜBUNGSAUFGABE ERSTES PROGRAMM

- Wir implementieren „Hello World“ ohne IDE
  - TextEditor eurer Wahl
  - Befehl javac
  - Befehl java
- **15 Minuten**

ÜBUNGSAUFGABE JAR

- Das Manifest
  - Legt MainClass und Meta-Informationen fest
  - jar cfm hello.jar manifest.txt Hello.class
  - java -jar hello.jar
- **10 Minuten**

- Manifest.txt

Manifest-Version: 1.0

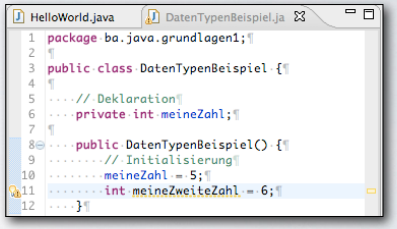
Main-Class: Hello

Siehe https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html

Eine Manifest.txt muss mit einer Leerzeile enden

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



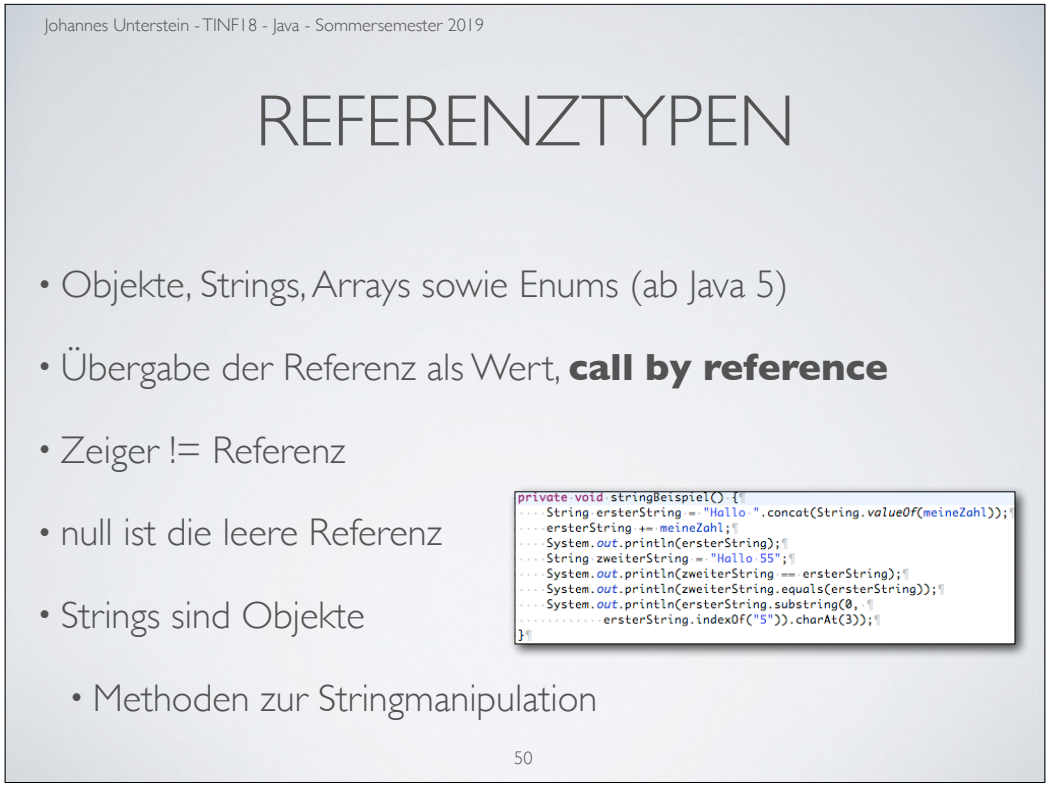
1. - Primitive Datentypen sind keine Objekte!
   - Dadurch Gewinn an Performance im Vergleich zu „reinen“ OO Sprachen (Was war zum Beispiel eine reine OO Sprache?)
   - Warum sollte eine 1 auch unterschiedlich zu einer 1 sein? Es macht keinen Sinn.

Größen [byte]:
1 byte
2 short
4 int
8 long

4 float
8 long

2 char

1 boolean (default: false)



2. - 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 gleich sind
3. - Adresse einer Referenz kann nicht verändert werden, bei einem Zeiger ist dies ja durchaus möglich.
   - Generell gibt es keine Zeigerarithmetik in Java
4. - Auf null ist prüfbar
5. - 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
l

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 (**Umwandeln des Typs**)
  - Verlustfrei in nächst größeren Datentyp, nicht umgedreht
  - In nächst kleineren Datentyp muss explizit gecastet werden
- Referenztypen (**Ändern der Sichtweise**)
  - 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();
}
```


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
- **Referenz**gleichheit: == bzw. !=
- instanceof-Operator: a 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 BodenFahrzeug);
    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

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?

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

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. Empfinde 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/MergeSort/BubbleSort von **Arrays**
- 45 Minuten
- Hinweis: Verwendet auch den Debugger eurer IDE

JAVA

Objektorientierung in Java

AGENDA

- Klassen und Objekte
- Pakete
- Vererbung
- Modifier
- Abstrakte Klasse
- Interfaces
- Lokale Klassen
- Wrapper Klassen
- Übungsaufgabe

SOURCE CODE

- https://github.com/unterstein/dhbw-java-lecture
- ⇔ https://git.io/vr6Pa

KLASSEN

```
package ba.java.oo;

[modifier] class [Name] {

    // Attribute:
    [modifier] [Typ] [Name];

    // Methoden:
    [modifier] [ReturnTyp] [Name]([Parameter]) {
        [Anweisungen]
    }
}
```

```
package ba.java.oo;

public class Klasse {

    // Attribute:
    String einString;

    // Methoden:
    public String getString() {
        return einString;
    }
}
```


[modifier] = Sichtbarkeit, Veränderbarkeit und Lebensdauer

OBJEKTE EINER KLASSE

```
// Initialisierung:
Typ Variable = new Typ();

// Beispiel:
Klasse klasse = new Klasse();
klasse.getString(); // Was gibt das zurück?
```

- Methoden:
  - Rückgabewert **void** möglich

THIS

- Innerhalb einer Methode darf ohne Qualifizierung auf Attribute und andere Methoden zugegriffen werden
  - Der Compiler bezieht diese Zugriffe auf das aktuelle Objekt „this“ und setzt implizit „this.“ vor diese Zugriffe
- Die „this“-Referenz kann auch explizit verwendet werden

```
public void setEinString(String einString) {
    this.einString = einString;
}
```


2. - Hervorheben, dass man auf Member zugreift
   - Ein Member heisst so, wie zum Beispiel ein Parameter
   - Man möchte generell auf einen anderen Scope zugreifen, der nicht der aktuelle ist
3. - Es gibt nicht so eine tiefe Verschachtelung der Scopes in Java, wie es z.B. in C der Fall ist.
   - Generell sind geschweifte Klammern wenig zu empfehlen, da wir schon die Kapselung durch die OOP gegeben haben.
   - Es kann in einem geschweiften Klammern Block keine Variablen mit Namen definiert werden, die es schon außerhalb des Blockes gibt.
     --> Beispiel

```
{
    String test = "lolo";
}
System.out.println(test);
```

oder

```
String test = "test1";
{
    String test = "lolo";
    System.out.println(test);
}
```

führen zu Compilefehlern!

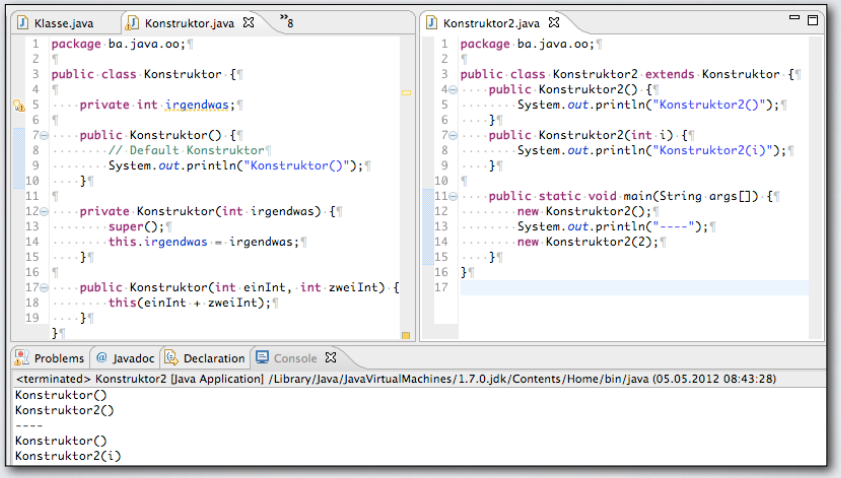
KONSTRUKTOREN

- Spezielle Methoden zur Initialisierung eines Objekts
- Default-Konstruktor
- Verkettung von Konstruktoren
- Werden nicht vererbt

```
package ba.java.oo;¶
¶
public class Konstruktor {¶
¶
    private int irgendwas;¶
¶
    public Konstruktor() {¶
        // Default Konstruktor¶
    }¶
¶
    private Konstruktor(int irgendwas) {¶
        super();¶
        this.irgendwas = irgendwas;¶
    }¶
¶
    public Konstruktor(int einInt, int zweiInt) {
        this(einInt + zweiInt);¶
    }¶
}¶
```


1. - Bsp.: „**Rezept** der Nudelsuppe!“
   - Konstruktor hat gleichen Namen wie Klasse.
   - Konstruktor hat keinen Rückgabewert.
   - Parameterlos oder mit Parametern.
   - Mehrere Konstruktoren mit unterschiedlichen Parametern möglich.
     -> Überladen von Konstruktor-Methoden.
2. - Ist kein Konstruktor explizit angegeben, so wird vom Compiler der Default-Konstruktor angelegt.
   - Default-Konstruktor ist ein Konstruktor ohne Parameter.
   - Default-Konstruktor nimmt keine besondere Initialisierung vor.
   - Enthält eine Klasse nur parametrisierte Konstruktoren, wird **kein** Default-Konstruktor angelegt!
3. - Mittels this oder super lassen sich andere Konstruktoren aufrufen und damit **verketten**.
   - Nützlich um Logik nur einmal abzubilden und in allen Konstruktoren zur Verfügung zu haben.
   - this(..) bzw. super(..) Konstruktoraufrufe müssen stets am Anfang eines Konstruktors stehen!
   - Mehr zu super(...) auf späteren Folien.

KONSTRUKTOREN



- Der Default Konstruktor wird **immer** aufgerufen, auch wenn dieser nicht explizit aufgerufen wird.
- Achtung wichtig: Der Konstruktor einer Superklasse wird **immer vor** der dem Konstruktor einer Subklasse ausgeführt.
- Besitzt die Superklasse keinen Default-Konstruktor, so muss ein Konstruktor explizit aufgerufen werden! (Sonst Compile-Fehler)

ÜBUNGSAUFGABE (20 MIN)

- Implementiere:
  - Klassen: Quadrat erbt von Rechteck
  - Rechteck hat Variablen für Breite und Länge + Getter/Setter
  - Beide haben Default-Konstruktoren und mit Parametern
- Verfolge Aufrufe verschiedener Konstruktoren im Debugger

DESTRUKTOREN

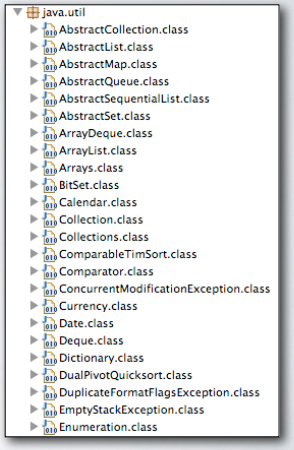
- Keine Destruktoren wie in C++
- Aufräumen beim Löschen des Objektes trotzdem möglich
  - protected void finalize() { ... }
  - Aufruf ist nicht garantiert, eher nicht verwenden!

1. - In C++ muss mittels des Destruktors das Zerstören des Objektes und die Freigabe des Speichers veranlasst werden.
   - In Java muss dies nicht geschehen, da wir ja die Garbage Collection haben.
3. - Die finalize Methode wird unmittelbar vor dem Zerstören des Objektes durch die Garbage Collection aufgerufen.
4. - Da die Methode allerdings auch von der Garbage Collection aufgerufen wird, ergeben sich folgende Nachteile:
     - Zeitpunkt des Aufrufs ist unbestimmt.
     - Es ist nicht garantiert, dass die Methode überhaupt aufgerufen wird. Warum?
   - Zum realen Aufräumen, nach dem ein Objekt nicht mehr benötigt wird, lieber eigene Methode und expliziten Aufruf.

PACKAGES

- Klassen reichen als Strukturelement nicht aus
- Packages sind eine Sammlung von Klassen, die einen gemeinsamen Zweck verfolgen
- Jede Klasse ist Bestandteil von genau einem Package
  - Packages funktionieren wie Verzeichnisse

```
package ba.java.oo;

public class Klasse {
```



1. - In großen Programmen reichen Klassen als Strukturelemente nicht mehr aus.
   - In einer flachen Hierarchie stößt die Übersichtlichkeit mit steigender Klassenzahl an ihre Grenzen.
   -> Vergleiche java.util.* Auf dem Bild ist nur ein winziger Teil dieses Packages.
2. - In diesem Beispiel sieht man auch schön, dass Klassen in einem Package **semantisch** zusammengehören sollten.
     - Was hat ein Calendar mit einer Collection zu tun?
     - Warum ist das Package java.util.* aber immer noch so groß wie es ist?
   - Packages bilden also ein Möglichkeit Klassen besser nach ihrer Semantik zu strukturieren.
3. - Eine Java Klasse ist genau einem Package zugeordnet.
   - Wird kein Package angeben, liegt die Klasse im sogenannten Default-Package.
4. - Vergleichbar ist das ganze mit Verzeichnissen (Packages) und Dateien (Klassen).
     - Eine Datei kann auch nur in einem Verzeichnis liegen (Symlinks zählen nicht) oder eben auf „/“ (Default-Package)

PACKAGES

- **Default** Package (=keine Package-Deklaration)
  - Sollte nur bei kleinen Programmen verwendet werden
- Verwendung von Klassen aus Packages

```
package ba.java.oo;¶
¶
import java.util.Date;¶
public class Packages {¶
    // Import der Klasse verwenden¶
    Date dateImport = new Date();¶
    // Vollqualifizierter Zugriff auf die Klasse¶
    // Import ist in diesem Fall nicht notwendig¶
    java.util.Date dateQualified = new java.util.Date();
}¶
```


1. - Wie bereits gesagt: Ist kein Package angegeben, landet die Klasse im Default Package.
   - Sollte nur bei kleineren Programmen verwendet werden.
   - Oder, wenn man weiß was man tut. Es gibt Frameworks, die darauf aufbauen, das Default Package zu verwenden und in diesem Package nach bestimmten Klassen suchen.

PACKAGES IMPORT

- *-Notation importiert alle Klassen des Packages

```
import java.util.*;
```

  - Bitte nicht benutzen, IDEs sind gut genug
- Automatischer Import von java.lang.*

1. - Mit dem *-Import werden alle Klassen des angegeben Package importiert.
   - Subpackages bleiben davon unberührt und werden nicht importiert.
   - In unserem Beispiel würden somit alle Klassen aus **java.util importiert** werden, die Subpackages wie **java.util.logging.*** würden aber **nicht importiert** werden.
   - **Problem**: Klassen können in unterschiedlichen Packages gleich heißen (z.b.: java.util.List, java.awt.List)
     - Wenn beide Packages über * importiert werden führt dies zum Konflikt
     - Der Compiler weiß nicht, welche Klasse er verwenden soll
   - Der *-Import wirkt sich nicht negativ auf die Performance aus, da der Compiler im Compile Prozess automatisch das * auflöst und nur die Klassen importiert, die tatsächlich benötigt werden.
   - Klassen die explizit einzeln importiert aber nicht verwendet werden, sind nicht von dieser Aufräumarbeit betroffen!
2. - Alle Klassen aus dem Package java.lang sind so wichtig, dass sie immer und automatisch importiert werden.
   - Bestandteil sind z.B. String oder Object
   - Alle anderen Packages müssen hingegen explizit importiert werden.

VERERBUNG

```
[modifier] class [SubKlasse] extends [SuperKlasse] {
    ...
}

// Auto Beispiel
package ba.java.auto;

public class Suv extends Pkw {
    public boolean allrad;
}

// Verwendung im Code
Suv q7 = new Suv();
q7.allrad = true;       // eigenes Attribut
q7.anzahlBlinker = 6; // geerbtes Attribut
q7.blinkeRechts();      // geerbte Methode
```

SUPERKLASSE OBJECT

- Jede Klasse erbt implizit von Object
- Wichtige Methoden von Object
  - public boolean equals(Object obj)
  - public int hashCode()
  - protected Object clone()
  - public String toString()

1. - Jede Klasse ohne Vererbung (ohne extends) erbt implizit direkt von Object.
   - Jede explizit abgeleitete Klasse stammt am obersten Ende der Vererbungshierarchie also auch von Object ab.
   - Object ist damit SuperKlasse von allen Klassen.
2. - Equals vergleicht ob zwei Objekte inhaltlich gleich sind.
   - Wdh.: Was ist der Unterschied zu dem == Operator?
   - Per Default Implementierung von Object ist es allerdings ein „==“ Vergleich! ¯_(ツ)_/¯
   - “equals” kann überschrieben werden, „==“ nicht

   -> Wird von allen Klassen der Java-Klassenbibliothek sinnvoll überschrieben.
3. - hashCode berechnet den numerischen Wert zum Hashing des Objektes.
   - hashCode stellt also die numerisch (mehr oder weniger) eindeutige Repräsentation eines Objektes dar.

   -> Regel: **Überschreibe niemals equals ohne hashCode** und umgekehrt, denn es tut beiden weh ... bzw. tut euch weh!
   -> Jemand eine Vorstellung warum das so ist?
   - Viele Datenstrukturen (z.B. Collections) greifen aus Performance-Gründen auf HashCode zu um Objekte zu strukturieren
     - Diese können dann nicht mehr gefunden werden, wenn Repräsentation von hashCode und equals unterschiedlich ist
   - Wenn zwei Objekte gleich sind laut equals sollten sie auch in einer HashMap gleich sein, welche auf hashCode() zugreift
4. - clone kopiert das Objekt (sofern dies möglich ist -> mehr dazu später).
5. - toString erzeugt eine String Repräsentation des Objektes (Wichtig für +Operator).

DIE SUPER REFERENZ

- Innerhalb einer Methode darf ohne die Super-Referenz auf Attribute und Methoden der Superklasse zugegriffen werden
  - Der Compiler bezieht diese Zugriffe auf die Superklasse und setzt implizit ein „super.“ davor
- Die super Referenz kann explizit zur Konstruktorverkettung verwendet werden oder explitizten Methoden-Aufrufen
- Konstruktoren werden nicht vererbt!

2. - Wiederholung:
   - Bei welchen Modifiern kann denn auf super zugegriffen werden?
3. - Wiederholung:
   - Der **super() Aufruf muss stets am Anfang eines Konstruktors** sein.
   - Wird super() nicht explizit aufgerufen im Konstruktor, übernimmt das implizit der Compiler.
4. - Konstruktoren werden nicht vererbt, aber automatisch oder explizit verkettet.
   -> Gibt es keinen Default-Konstruktor in der SuperKlasse, muss im Konstruktor der SubKlasse ein expliziter Konstruktor Aufruf mit super(..) getätigt werden

ÜBERLAGERN VON METHODEN

- Abgeleitete Klassen re-implementieren eine geerbte Methode einer Basisklasse
  - Die Methode selbst wird übernommen, bekommt allerdings ein neues Verhalten
  - Mittels „super.“ kann die überlagerte Methode der Superklasse noch aufgerufen werden.
    - Verkettung von „super“ nicht möglich
      - „super.super.doSomething()“

```
public class Suv extends Pkw {
    public boolean allrad;

    public void bereiteDifferenzialVor() {
        
    }

    @Override
    public void blinkeRechts() {
        super.blinkeRechts();
        bereiteDifferenzialVor();
    }
}
```


1. - Wdh.: Was ist Überlagern?

ÜBERLAGERUNG VON METHODEN

- Late Binding: Zur Laufzeit wird erst entschieden, welche Methode tatsächlich aufgerufen wird
- Late Binding kostet daher Laufzeit
  - Wenn möglich Methoden private oder final deklarieren

```
Pkw kfz = new AudiQFuenf();
// Ruft die Methode der Klasse Suv auf
kfz.blinkeRechts();
```

77

3. - Warum bringt es Laufzeitvorteile, wenn man Methoden private oder final deklariert?

MODIFIER

- Modifier beeinflussen die Eigenschaften von Klassen, Methoden und Attributen
  - Sichtbarkeit
  - Lebensdauer
  - Veränderbarkeit

SICHTBARKEIT

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

- Achtung: (echte) Klassen und Interfaces können nur public sein oder Standard-Sichtbarkeit besitzen

1. - Auch kein protected als modifier erlaubt für (echte) Klassen!
   - Was unter echter Klasse und mehr oder weniger echten Klassen zu verstehen ist, kommt im nächsten Kapitel.

SICHTBARKEIT

- Kurze Übung: Was gibt folgendes Programm aus?

Pkw.java | AudiQFuenf.java | Sichtbarkeit.java | »15

```
1  package ba.java.oo;
2
3  public class Sichtbarkeit {
4      private int i = 5;
5
6      public Sichtbarkeit() {
7          int i = 30;
8          System.out.println(this.i);
9          i += 5;
10         printInt(i);
11     }
12
13     protected void printInt() {
14         System.out.println(i + 10);
15     }
16
17     protected void printInt(int i) {
18         System.out.println(i + 10);
19     }
20
21 }
22
```

Sichtbarkeit2.java

```
1  package ba.java.oo;
2
3  public class Sichtbarkeit2 extends Sichtbarkeit {
4      private int i = 7;
5
6      public Sichtbarkeit2() {
7          int i = 4;
8          printInt();
9          printInt(this.i + i);
10     }
11
12     @Override
13     protected void printInt(int i) {
14         System.out.println(i + 5);
15     }
16
17     public static void main(String[] args) {
18         new Sichtbarkeit2();
19     }
20 }
21
```


5
40
15
16

ÜBUNGSAUFGABE (15 MIN)

- Implementiere:
  - Klassen: Sichtbarkeit2 erbt von Sichtbarkeit1
  - Versuche deinen Sitznachbarn mit einem komplexen Programmablauf zu verwirren ᕙ(͡°)ᕗ
- Verfolge Aufrufe verschiedener Konstruktoren im Debugger

LEBENSDAUER UND VERÄNDERBARKEIT

- **static**
  - Definition von Klassenmethoden und -attributen
  - Zugriff ohne konkretes Objekt möglich: Klasse.methode()
  - Was gibt „Quadrat q =new Quadrat()“ aus?

```
package ba.java.oo;

public class Quadrat {

    private static int seiten;
    private int laenge;

    static {
        seiten = 4;
        System.out.println("static");
    }

    public Quadrat() {
        System.out.println("dynamic");
    }

    private static int getUmfang(int laenge) {
        return laenge * seiten;
    }

    public int getUmfang() {
        return getUmfang(laenge);
    }
}
```


1. - Wiederholung:
   - Mittels static werden Klassenmethoden und Klassenattribute definiert.
   - Initialisierung kann auch in einem sogenannten static{} Block erfolgen, siehe Beispiel.
   - static{} Block ist allerdings eher unüblich, eher direkte Zuweisung „private static int seiten = 4;“.
   - static{} Block wird ausgewertet, wenn das erste Mal auf eine Klasse durch die JVM zugegriffen wurde.
   - Statische Methoden können nur auf andere statischen Methoden und statische Attribute zugreifen.
   - Der Zugriff ist auch ohne konkretes Objekt möglich: Klasse.methode()

   - Wie lange lebt so eine Klassenvariable im Vergleich zu einer Instanzvariable?

4. - Unter der Annahme, dass auf die Klasse Quadrat noch nicht zugegriffen wurde

LEBENSDAUER UND VERÄNDERBARKEIT

- **final**
  - Finale Attribute, Parameter und Variablen
  - Finale Methoden
  - Finale Klassen
  - Achtung bei finalen Variablen auf Referenztypen!

2. - Finale Attribute, Parameter und Variablen dürfen nicht verändert werden.
   -> Konstante Werte
   - Es ist allerdings möglich, dass man final Variablen deklariert und nichts zuweist und erst später initialisiert. (Nur im Konstruktor !)
3. - Finale Methoden dürfen nicht überlagert werden.
   -> Compiler kann sich Late Binding dieser Methode sparen.
4. - Finale Klassen dürfen nicht abgeleitet (spezialisiert) werden.
   -> Compiler kann sich auf Late Binding aller Methoden der Klasse sparen.
5. - **ACHTUNG**: bei finalen Objektvariablen wird zwar die Variable vor einer neuen Zuweisung geschützt,
   -> Das Objekt selbst kann innerlich allerdings noch verändert werden!

   - Frage: Gilt dies auch für Arrays?

```
final int[] array= new int[5];
array[0]= 1;       // Geht das?
array = new int[6]; // Geht das?
```

LEBENSDAUER UND VERÄNDERBARKEIT

- **transient**
  - Verwendung bei Serialisierung und Deserialisierung
  - Transient Attribute werden dabei ignoriert
- **volatile**
  - Verwendung beim Multithreading
  - Volatile Attribute können asynchron modifiziert werden

1. - Definition Serialisierung: Persistenz eines Objektes in einer Datei.
   - Attribute, die mit transient gekennzeichnet sind, werden beim Serialisieren und Deserialisieren ignoriert.
   -> flüchtige Werte
   Beispiel: Das Geburtsdatum ist angegeben in einem Objekt und das Attribut „Alter“ ist daher als transient gekennzeichnet.
2. - Asynchron: Außerhalb des aktuellen Threads.
   - Wert einer solchen Variable wird daher bei jedem Zugriff neu gelesen.
   - Stellt Datenintegrität in verteilten Prozessen sicher.
   - Verwendung von volatile allerdings ungebräuchlich und selten.
   - Obwohl es ungebräuchlich ist, kurzes einhaken, da volatile technisch sehr interessant ist:
     -> Was bedeutet bei jedem Zugriff neu lesen in der Java Welt?
     - Variable wird nicht aus Register der JVM genommen, sondern tatsächlich neu vom Heap gelesen

ABSTRAKTE KLASSE

- Abstrakte Methode
  - Enthält nur Deklaration, keine Implementierung
- Abstrakte Klasse
  - Kann abstrakte Methoden beinhalten

```
public abstract class Pkw {
    public int anzahlBlinker;

    public abstract void blinkeRechts();
}

public class AudiQFuenf extends Pkw {

    @Override
    public void blinkeRechts() {
    }

}
```


1. - Wiederholung:
   - Eine Abstrakte Methode enthält nur die Deklaration, nicht aber die eigentliche Implementierung der Methode.
   - Definition einer abstrakten Methode mittels des Schlüsselwortes „abstract“.
2. - Eine Klasse, die mindestens eine abstrakte Methode besitzt, muss selbst „abstract“ sein.
   - Eine Klasse kann auch „abstract“ sein, wenn sie keine abstrakte Methode beinhaltet.
   - Eine abstrakte Klasse kann nicht direkt instanziiert werden.

INTERFACES

- Interfaces enthalten
  - Methoden, die implizit public und abstract sind
  - Konstanten
  - **keine** Konstruktoren oder Attribute

Groesse.java

```
1  package ba.java.oo.auto.interfaces;
2
3  public interface Groesse {
4      int getLaenge();
5
6      int getHoehe();
7
8      int getBreite();
   }
```

Auto.java

```
1  package ba.java.oo.auto.interfaces;
2
3  public class Auto implements Groesse {
4
5      @Override
6      public int getLaenge() {
7          return 435;
8      }
9
10     @Override
11     public int getHoehe() {
12         return 160;
13     }
14
15     @Override
16     public int getBreite() {
17         return 210;
18     }
19
20 }
```

INTERFACES

- Mehrfachimplementierung
  - Eine Klasse kann mehrere Interfaces implementieren
- Vererbung von Klassen mit Interfaces
  - Eine Klasse erbt jeweils alle Interfaces seiner Basisklasse
- Ableiten von Interfaces mit „extends“
  - Interfaces können von anderen Interfaces erben

1. - Eine Klasse kann mehrere Interfaces implementieren.
   - Wenn eine Klasse n Interfaces implementiert, dann ist sie mindestens zu n+1 weiteren Datentypen kompatibel.
     -> n Interfaces
     -> mind. 1 Vaterklasse
   - Wdh.: Warum ist es eventuell Problematisch viele Interfaces zu Implementieren, wenn diese z.b. gleiche Methoden definieren?
   - „Person implements PrivatePerson, Angestellter“ -> wo klingelt getTelefonnummer()?
2. - Eine Klasse erbt alle Interfaces, welche auch die Basisklasse implementiert hat.
   - Natürlich werden die Implementierungen auch geerbt.
3. - Interfaces können selbst auch abgeleitet werden, allerdings nur von anderen Interfaces.
   - Das abgeleitete Interface erbt alle Methodendefinitionen des Basis-Interfaces.
   - Eine implementierende Klasse muss damit auch die Methode von allen übergeordneten Interfaces implementieren.

SINN VON INTERFACES

- Trennung **Schnittstelle** von Implementierung
- Beschreibung von **Rollen**
- „**Ein Auto ist keine Größe!**“
- Auslagerung von Konstanten (static final)
- Verwendung als Laufzeit-Flag

1. - Eine Schnittstelle kann mehrere Implementierungen haben.
   - Macht wirklich nur Sinn, wenn es mehrere Implementierungen gibt!
   - Wiederholung: Was habe ich zu Interfacenamen und generell Interfaces im OO Teil gesagt?
2. - Wiederholung: Mensch als Objekt und Tierliebhaber, Vegetarier, Fußgänger als Rollen.
4. - Auslagerung von Konstanten aus Klassen.
   - Diese Konstanten stehen dann in allen Implementierungen des Interfaces zur Verfügung.
   - Somit ist es möglich Konstanten über Klassen hinweg zu definieren.
5. - Logischer Schalter zur Abfrage während der Laufzeit.
   - Es werden also spezielle Rollen (auch Flags oder Marker) durch sogenannte Markerinterfaces kenntlich gemacht.
   - Beispiel: Cloneable, Serializable
   - Schalter für die Methode clone() in der Klasse Objekt, ob eine Klasse klonbar ist.
   - Ist das Interface nicht implementiert, steht die Funktionalität nicht zur Verfügung.

ÜBUNG (15 MIN)

- OO Modellierung folgender Klassen
  - PKW
  - LKW
  - LKW mit Anhänger
  - Taxi
  - Autobus
  - Containerschiff
  - Fähre
  - Floss
  - Yacht
  - **Groesse**

- Ziel: Klassendiagram mit Klassen einer sinnvollen Vererbungshierarchie
- Welche Klassen sind zu ergänzen, welche sind abstrakt?

INTERFACE ODER ABSTRAKTE KLASSE?

- Wiederholung: Unterschied Interface und Abstrakte Klasse?
- Semantischer Unterschied
  - **Generalisierung** <-> **Spezialisierung**
- Pre Java 8: In Praxis leider meist pragmatischere Entscheidungen

1. - 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**.

2. - Ist ein Auto eine Größe? Nein -> Ein Auto hat eine Größe!
   - Ist ein Mensch ein Vegetarier bzw. ist es festgeschrieben „Einmal Vegetarier, immer Vegetarier?“ ? Nein
   - Ist ein Suv ein Auto? Ja
   - Sind dies nur Rollen, die das jeweilige Objekt einnimmt? Ja

3. - Interface als Schnittstellenbeschreibung
   - Abstrakte Klasse, die das Interface implementiert und soweit möglich eine sinnvolle Default-Implementierung bereitstellt
   - Können alle Methoden des Interfaces als default implementiert werden, muss die Klasse nicht abstrakt sein!
   - Teilweise ist auch eine leere Implementierung eine gute Default-Implementierung

DEFAULT IMPLEMENTATION

- Seit Java 8 ist folgendes möglich:


```
public interface DefaultImplementation {

  Double calcSomething(double double1);

  Double calcSomething(double double1, double double2);

  default Double calcSomething(double double1, double double2, double double3) {
    return calcSomething(double1, double2 + double3);
  }
}
```

- Mehrfachvererbung immer noch nicht möglich ;-)

1. Keyword: „default“ in Methoden von Interfaces kann Implementierung hinzufügen
-> Absoluter Bruch mit der bisherigen Semantik der OOP

2. Wenn **zwei** „default“-Implementierungen einer Methode von Interfaces mitgebracht werden muss diese explizit überlagert werden

LOKALE KLASSE

- Lokale Klassen werden auch „Inner Classes“ genannt
- Normalerweise Klassenstruktur innerhalb eines Packages flach
- Große Rolle in Benutzeroberflächen
- Inner Classes sind ein mächtiges Feature

2. - Über Inner Classes kann eine Granularitätsstufe mehr eingezogen werden.
   - Normalerweise ist die Handhabung von Inner Classes eher unhandlich.
   - Es gibt allerdings **zwei Use Cases**, wann dieses Konstrukt gerne benutzt wird:
     - Eine Klasse hat tatsächlich nur eine lokale Bedeutung.
     - Es wird „schnell mal eine Klasse benötigt“ ... die auch nur eine lokale Bedeutung hat ;-)
3. - Insbesondere in der Entwicklung graphischer Benutzeroberflächen spielen Inner Classes eine große Rolle.
   - Bei dem EventListener Pattern von AWT/Swing/SWT sehr gern verwendet.
   - Mehr dazu später.
4. - Inner Classes sind ein mächtiges Feature
     -> Können zum Teil auf **Zustand der umgebenden Klasse** zugreifen! Siehe nächste Folien.
     - Jedoch auch verwirrend bei übermäßigem Einsatz.
     - Der Quelltext ist nur für sehr geübte Augen lesbar.
     - Daher sollte dieses Konzept mit Bedacht eingesetzt werden.
     - Faustsregel: Nur dann Inner Classes verwenden, wenn eine „globale“ (echte) Klasse umständlich einzusetzen wäre.

NICHT STATISCHE LOKALE KLASSE

- Innerhalb des Definitionsteils einer Klasse wird eine neue Klasse definiert
- Definition wie „globale Klasse“
- Instanziierung der inneren Klasse muss innerhalb der äußeren geschehen
- Die lokale Klasse kann auf die Member der Äußeren zugreifen
- Qualifizierung: OuterClass.this.member
- Sowohl auf äußerster Klassenebene, als auch in Methoden möglich

3. - Innerhalb einer Methode oder im Konstruktor
4. - Die äußere Klasse kann ebenso auf die Member der inneren Klasse zugreifen. Sichtbarkeiten spielen dabei keine Rolle, weil alles im **private Scope** liegt.
6. - Man kann eine Klasse auch nur mit der Sichtbarkeit für eine Methode definieren. Diese Klasse kann (wenn die Variablen final deklariert sind) auch auf die Variablen der Methode zugreifen.

ANONYME KLASSE

- Es ist untypisch Klassen in Methoden einen Namen zu geben
- Stattdessen werden diese Klassen anonym deklariert
- Definition und Instanziierung muss in einem Schritt geschehen
- Wichtige Anwendung: Listener bei GUIs
- Die in der Definition angegebene Klasse/Interface wird automatisch abgeleitet/implementiert
- Nur für kleine Klassen!

6. - Wegen der Übersichtlichkeit.
   - Wenn eine anonyme Klasse größer und komplexer wird, hat es meistens den Anspruch auf eine eigene Klasse.

Anonyme Klassen sind schon oft (in der Literatur und aus eigener Erfahrung) große Diskussionspunkte gewesen. Die Übersichtlichkeit wird immer mit der unglaublichen Flexibilität konfrontiert. Eine anonyme Klasse kann dort deklariert werden wo sie gebraucht wird (und auch nur dort) und gibt Java den Charme einer mehr funktional angehauchten Sprache. Das ist Java natürlich nicht, aber es besteht dadurch die Möglichkeit durch wenige Zeilen Code das Verhalten von Objekten so zu manipulieren, wie man es gerne hätte.

BEISPIEL

```
public class InnerClasses {¶
    public InnerClasses() {¶
        ErsteInnerClass eins = new ErsteInnerClass();¶
        eins.i = 5;¶
¶
        // Inner Class in einer Methode, sehr unüblich!¶
        class ZweiteInnerClass {¶
            private double x;¶
        }¶
¶
        ZweiteInnerClass zwei = new ZweiteInnerClass();¶
        zwei.x = 5;¶
¶
        // Anonyme Inner Class¶
        // Erinnerung: Pkw ist abstrakt¶
        Pkw pkw = new Pkw() {¶
¶
            @Override¶
            public void blinkeRechts() {¶
                // blinkblinkblink¶
            }¶
        };¶
        pkw.blinkeRechts();¶
    }¶
¶
    // Klassische Inner Class¶
    private class ErsteInnerClass {¶
        private int i;¶
    }¶
}¶
```

STATISCHE LOKALE KLASSEN

- Innerhalb der Klasse definiert, mit static versehen
- Im Prinzip ist es keine lokale Klasse
- Einziger Unterschied zu gewöhnlicher Klasse:
  - Äußere Klasse als „Präfix“: new OuterClass.InnerClass();
- Benutzt man gerne für „kleine“ Helper Classes ohne Context
  - Oder Demos :)

2. - Kleine Mogelpackung!
   - Der Compiler erzeugt Code, der genau dem Code entspricht, als sei es eine gewöhnliche Klasse.
   - Kein Zugriff auf Membervariablen, da sie static ist und hat somit keine Referenz auf die instanziierende Klasse.
3. - Der einzige Unterschied zu einer gewöhnlichen Klasse ist bei der Instanziierung.
   - Man muss die Umgebende Klasse als „Präfix“ bei der Instanziierung verwenden.
4. - Eine valide Frage wäre, warum man nicht gleich eine gewöhnliche Klasse macht :)
   - Man benutzt diese Art der Klassen gerne, wenn die Daseinsberechtigung der Klasse auf der Existenz der äußeren Klasse basiert
   -> Also zum Beispiel für kleinere Helfer Klassen die eh nur static Methoden haben

WRAPPER KLASSEN

- Zu Jedem primitiven Datentyp in Java gibt es eine korrespondierende Wrapper Klasse
- Kapselt primitive Variable in einer „objektorientierten Hülle“ und stellt Zugriffsmethoden zur Verfügung

Primitiver Typ	Wrapper-Klasse
byte	Byte
short	Short
int	Integer
long	Long
double	Double
float	Float
boolean	Boolean
char	Character
void	Void

UMGANG MIT WRAPPER KLASSEN

```java
public WrapperClasses() {
    // Instanziierung

    // Übergabe des zu kapselnden primitiven Typs
    Integer integer1 = new Integer(1);
    // Meist können auch Strings übergeben werden
    // Vorsicht bei diesem Aufruf.
    // Es kann eine Ausnahme auftreten!
    Integer integer2 = new Integer("1");

    // Rückgabe, auch schon gecastet möglich
    int i = integer1.intValue();
    short short1 = integer1.shortValue();
    String string1 = integer1.toString();

    // Auch das parsen von Strings ist meistens möglich
    // Auch hier kann eine Ausnahme auftreten!
    Integer integer3 = Integer.parseInt("42");
}
```

UMGANG MIT BOXING/UNBOXING

```java
public static void main(String[] args) {
  // List<int> nicht möglich, daher List<Integer>
  List<Integer> alleMeineZahlen = new ArrayList<>();
  alleMeineZahlen.add(12); // auto boxing
  int primitive = alleMeineZahlen.get(0); // auto unboxing
  Integer reference = alleMeineZahlen.get(0); // nix spezielles

  // soweit alles ok, aber bei folgendem Code potentielle Gefahr:
  Integer meineMagischeZahl = null; // Referenztypen dürfen mit null initialisiert werden
  // Es wid auto unboxing versucht, allerdings fliegt es fürchterlich auf die Nase!
  magieMitZahlen(meineMagischeZahl);
}

private static int magieMitZahlen(int i) { return i * 2; }
```

IMMUTABLE

- Mittels Wrapper Klassen erscheint es möglich, die primitiven Typen zu Kapseln und in Methoden zu verändern
- Dies ist allerdings nicht möglich, da die Wrapper Klassen unveränderlich (immutable) sind
- Wie kann man es schaffen primitive Typen veränderlich zu Kapseln?

2. - Wiederholung: Als Parameter für Methoden werden Kopien der Referenzen übergeben.
   - Es gibt auf einer Wrapper Klasse keine Methode „...setValue(...)“
   - Warum geht es nun nicht den Wert einer Wrapper Klasse in einer Methode zu verändern?
   -> Änderungen der Referenz, so dass es der Aufrufer mitbekommen, ist nicht möglich und ohne setValue(..) sieht es schlecht aus.
3. - Man schreibt sich einfach einen eigenen Wrapper, der ein setValue(..) besitzt.

JAVA SOURCEN

- Linux: sudo apt-get install openjdk-8-source
- Windows & Mac
  - http://download.java.net/openjdk/jdk8/
- Attach Source -> src.zip oder entpackter Ordner

ÜBUNGSAUFGABE (60 MIN)

- **Mitarbeiterdatenbank**
  - Arbeiter, Angestellter, Manager
  - Gemeinsame Daten:
    - Personalnummer, Persönliche Daten (Name, Adresse, Geburtstag, ...)
  - Arbeiter
    - Lohnberechnung auf Stundenbasis
    - Stundenlohn, Überstundenzuschlag, Schichtzulage
  - Angestellter
    - Grundgehalt und Zulagen
  - Manager
    - Grundgehalt und Provision pro Umsatz
  - Geschäftsführer (Spezieller Manager)
    - Erhält zusätzliche Geschäftsführerzulage
- **Gehaltsberechnung** (Löhne des Unternehmens)

Auf Basis eurer UML Diagramme

JAVA

Weiterführende Spracheigenschaften

AGENDA

- Strings
- Exceptions
- Enums
- Generics
- Lambdas & Methods
- Bulk-Operations

DIE KLASSE STRING

- Zeichenketten werden in Java als String repräsentiert
- Wie der Typ char sind auch Strings Unicode-fähig
- String hat viele Methoden zur Manipulation
- Keine Kenntnis über inneren Aufbau von Nöten

Vergleiche: http://commons.apache.org

1. - Ein String entspricht einer Reihung von Elementen des Typs char.
   -> Wie in anderen Sprachen auch.
   - Wiederholung: Wie groß ist ein char?
   - char Arrays sollten in Java nur noch in Ausnahmefällen verwendet werden.
3. - String liefert eine Vielzahl von Methoden zur Manipulation und Bestimmung von Eigenschaften der Zeichenkette mit.
   -> Mehr dazu gleich
   - Trotz dieser großen Vielzahl von Methoden gibt es eine Sammlung an Helfermethoden, welche einem den Umgang mit Strings noch weiter erleichtern.
   -> Diese sind aber leider kein Bestandteil dieser Vorlesung ;-). Vgl. http://commons.apache.org
4. - Anders als zum Beispiel bei C muss sich der Entwickler keine Gedanken über den inneren Aufbau von Strings Gedanken machen.
   - Keine Gedanken „verschwenden“, ob man es aktuell mit einem leeren Array von chars zu tun hat.
   - Weiterer Grund: Der innere Aufbau von Strings sind weitestgehend anonym und bedeutungslos für den Entwickler -> „Strings funktionieren einfach“.

WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel() {
    // Konstruktoren:

    // Ein leerer String
    String leerer = new String();
    // Duplizierung
    String duplikat = new String("Hallo");
    // Mittels char-Arrays
    char[] array = { 'H', 'a', 'l', 'l', 'o' };
    String charArray = new String(array);

    // Zeichenextraktion:

    // char charAt(int index)
    // Liefert das Zeichen am angegebenen nullbasierten Index.
    char charAt = duplikat.charAt(0);
    // String substring(int begin, int end)
    // Liefert den Teilstring von inklusiv Index begin bis exklusiv Index end.
    String subString1 = duplikat.substring(1, 3);
    // Eine Variante der Methode ohne end liefert stets den Teilstring bis zum Ende.
    String subString2 = duplikat.substring(3);
    // String trim()
    // Entfernt Leerzeichen am Anfang und Ende
    String trim = "    Hallo    ".trim();

    // Übung:
    System.out.println(leerer);
    System.out.println(duplikat);
    System.out.println(charArray);
    System.out.println(charAt);
    System.out.println(subString1);
    System.out.println(subString2);
    System.out.println(trim);
}
```


Ergebnis:
”

Hallo
Hallo
H
al
lo
Hallo
”

WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel2() {¶
    String string1 = "Hallo!";¶
    String string2 = "Wie gehts?";¶
    AudiQFuenf q5 = new AudiQFuenf();¶

    // Länge der Zeichenkette:¶
    ¶
    // int length()¶
    // Liefert die aktuelle Länge. Der Rückgabewert 0 bedeutet Leerstring.¶
    int length = string1.length();¶

    // Vergleich von Zeichenketten:¶

    // boolean equals(Object obj)¶
    // Analog zur gleichnamigen Methode der Klasse Object.¶
    // Vergleicht zwei Strings auf inhaltliche Gleichheit.¶
    boolean equals1 = string1.equals(string2);¶
    boolean equals2 = string1.equals("Hallo!"); // Was sollte man hier beachten?¶

    // Vergleich mit der String-Darstellung beliebiger Objekte möglich,¶
    // indem zuvor obj .toString() gerufen wird.¶
    boolean equals3 = q5.toString().equals("Ist das ein Q5?");¶

    // boolean equalsIgnoreCase(String s)¶
    // Wie equals, ignoriert jedoch die Unterschiede in Groß-/ Kleinschreibung.¶
    boolean equals4 = "hALlo!".equalsIgnoreCase(string1);¶
    ¶
    // Übung:¶
    System.out.println(length);¶
    System.out.println(equals1);¶
    System.out.println(equals2);¶
    System.out.println(equals3);¶
    System.out.println(equals4);¶
}¶
```


Ergebnis

„
6
false
true
false
true
“

WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel3() {
    String string1 = "Hallo!";
    String stringA = "a";
    String stringB = "b";

    // Vergleich von Zeichenketten:

    // boolean startsWith(String s)
    // Testet, ob ein String mit der angegebenen Zeichenkette beginnt.
    boolean startsWith = string1.startsWith("Ha");
    // boolean endsWith(String s)
    // Testet, ob ein String mit der angegebenen Zeichenkette endet.
    boolean endsWith = string1.endsWith("o!");
    // int compareTo(String s)
    // Lexikalischer Vergleich beider Strings durch paarweisen Vergleich der einzelnen Zeichen
    // von links nach rechts. Ist der aktuelle String kleiner als s, wird ein negativer Wert
    // zurückgegeben. Ist er größer, wird ein positiver Wert zurückgegeben. Bei Gleichheit ist
    // der Rückgabewert 0. => Wichtig für Sortierung und Collections!
    int compare1 = stringA.compareTo(stringB);
    int compare2 = stringB.compareTo(stringA);
    int compare3 = stringA.compareTo(stringA);

    // Übung:
    System.out.println(startsWith);
    System.out.println(endsWith);
    System.out.println(compare1);
    System.out.println(compare2);
    System.out.println(compare3);
}
```


Ergebnis:

„
true
true
-1
1
0
“

WICHTIGSTE METHODEN VON STRINGS

```
public StringBeispiel4() {¶
    String string1 = "Hallo!";¶

    // Suchen in Zeichenketten:¶

    // int indexOf(String s)¶
    // Sucht das erste Vorkommen von s innerhalb der Zeichenkette. Wird s gefunden, wird der¶
    // Index des ersten übereinstimmenden Zeichens zurückgeliefert, ansonsten -1. Eine Variante¶
    // der Methode akzeptiert einen Parameter vom Typ char.¶
    int index1 = string1.indexOf("x");¶
    int index2 = string1.indexOf("l");¶
    // Fehlercode vs. explizite Ausnahme? Was ist euer Gefühl?¶

    // int indexOf(String s, int fromIndex)¶
    // Arbeitet wie die vorige Methode, beginnt allerdings mit der Suche erst bei fromIndex.¶
    // Auch hier akzeptiert eine Variante der Methode einen Parameter vom Typ char.¶
    int index3 = string1.indexOf("l", 3);¶

    // int lastIndexOf(String s)¶
    // Sucht nach dem letzten Vorkommen von s. Eine Variante der Methode akzeptiert einen¶
    // Parameter vom Typ char.¶
    int index4 = string1.lastIndexOf("l");¶

    // Übung:¶
    System.out.println(index1);¶
    System.out.println(index2);¶
    System.out.println(index3);¶
    System.out.println(index4);¶
}¶
```


Ergebnis:

”
-1
2
3
3
”

WICHTIGSTE METHODEN VON STRINGS

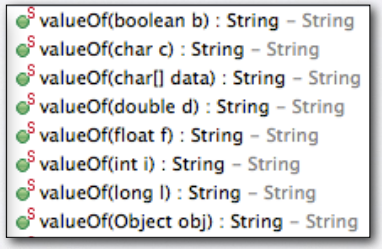
```
public StringBeispiel5() {¶
    String string1 = "Hallo!";¶

    // Ersetzen von Zeichenketten:¶

    // String toLowerCase()¶
    // Wandelt die Zeichenkette in Kleinbuchstaben.¶
    String string2 = string1.toLowerCase();¶
    // String toUpperCase()¶
    // Wandelt die Zeichenkette in Großbuchstaben.¶
    String string3 = string1.toUpperCase();¶
    // String replace(char old, char new)¶
    // Einzelne Zeichen werden ersetzt: old durch new.¶
    String string4 = "Salat".replace("Salat", "Schnitzel");¶
    // String replaceAll(String regex, String new)¶
    // Ersetzt alle Teilstrings, die die Regular Experession regex trifft, durch den neuen¶
    // Teilstring new.¶
    String string5 = "Hallo123".replaceAll("(\\d)", "Zahl");¶
    // String replaceFirst(String regex, String new)¶
    // Ersetzt nur den ersten gefundenen Teilstring, den die Regular Experession regex trifft,
    // durch den neuen Teilstring new.¶
    String string6 = "Hallo123".replaceFirst("(\\d)", "Zahl");¶

    // Übung:¶
    System.out.println(string1);¶
    System.out.println(string2);¶
    System.out.println(string3);¶
    System.out.println(string4);¶
    System.out.println(string5);¶
    System.out.println(string6);¶
}¶
```


Ergebnis:

„
Hallo!
hallo!
HALLO!
Schnitzel
HalloZahlZahlZahl
HalloZahl23
”

KONVERTIERUNG VON STRING

- Oftmals müssen primitive Daten in Strings gewandelt werden
  - Die Klasse String liefert dazu entsprechende statische String.valueOf(..)-Methoden:



WEITERE EIGENSCHAFTEN VON STRINGS

- Die Klasse String ist final
- String ist ein Referenztyp
- Verkettung durch + Operator
  - Beispiel: String string2 = "Hallo" + 123;
- **Strings sind nicht dynamisch**!
  - Inhalt und Länge Konstant
  - Jede Manipulation erzeugt ein neuen neuen String

1. - Eine eigene Ableitung ist nicht möglich
   - Es gibt also keine Möglichkeit, die vorhandenen Methoden auf „natürliche" Art und Weise zu ergänzen oder zu modifizieren.
   - Soll Beispielsweise die Methode „replace" etwas anderes machen, muss dies durch eine lokale Methode des Aufrufers geschehen, welche den String als Parameter bekommt.
   - Frage: Aus welchem Grund macht es Sinn, dass die Klasse String final ist?
   -> Einer der Gründe für diese Maßnahme ist die dadurch gesteigerte Effizienz beim Aufruf der Methoden von String-Objekten.
   -> Anstelle der dynamischen Methodensuche, die normalerweise erforderlich ist, kann der Compiler final-Methoden statisch kompilieren und dadurch schneller aufrufen. Daneben spielten aber auch Sicherheitsüberlegungen und Aspekte des Multithreading eine Rolle.
2. - Jedes String-Literal ist eine eigene Referenz auf ein Objekt
   - Der Compiler erzeugt für jedes Literal ein entsprechendes Objekt und verwendet es anstelle des Literals.
3. - Strings können durch den Plus-Operator verkettet werden.
   - Auch die Verkettung mit anderen Datentypen ist möglich. Es muss nur einer der Typen ein String sein.
   - Beispiel: String s = "Hallo" + 123;
4. - Strings werden wie gesagt Applikationsweit gespeichert und nur über Referenzen angesprochen.
   - D.h. bei der Initialisierung eines Strings wird der Inhalt (und somit auch die Länge) des Strings festgelegt und ist nicht mehr veränderbar.
   - Frage 1: Wie kann dann ein replace Funktionieren?
   -> Die replace Methode nimmt die Veränderung nicht auf dem Original-String vor, sondern erzeugt einen neuen String. Dieser ist mit dem gewünschten Inhalt gefüllt und gibt diese Referenz zurück.
   - Frage 2: Was passiert mit den „alten" String-Objekten?
   -> Strings sind Objekte und wenn keine Referenz mehr auf sie besteht, werden sie von der Garbage Collection weggeräumt.

STRINGBUFFER/-BUILDER

- Arbeitet ähnlich wie String, repräsentiert jedoch dynamische Zeichenketten
- Legt den Schwerpunkt auf Methoden zur Veränderung des Inhaltes
- Das Wandeln von StringBuffer zu String ist jederzeit möglich
- Laufzeitvorteile!

```
public StringBufferBeispiel() {
    String string1 = "Hallo!";
    StringBuffer buffer = new StringBuffer(string1);
    // Konkatenierung in Schleife
    for (int i = 0; i < 20; i++) {
        buffer.append(i);          // string1 += "" + i;
    }
    // Wandlung
    String bufferString = new String(buffer);

    // Übung:
    System.out.println(bufferString);
    System.out.println(buffer);
}
```


2. - Hinzufügen, Löschen, Ersetzen einzelner Zeichen
   - Konkatenieren von Strings
4. - Werden Strings massiv verändert oder konkateniert (z.B. in langlaufenden Schleifen) ist die Verwendung von StringBuffer zu empfehlen!
   - Der Java Compiler ersetzt (laut Spezifikation) nach Möglichkeit String-Konkatenierungen durch StringBuffer!
   -> Dies erhöht die Lesbarkeit enorm und verschlechtert nicht das Laufzeitverhalten.

```
StringBuilder builder = new StringBuilder(“hey!”);

// StringBuffer ist das gleiche wie StringBuilder, nur synchronisiert (Multithreading)
// StringBuffer ist daher langsamer, aber Multithreaded-fähig
StringBuffer buffer = new StringBuffer(“ho!”);
```

ÜBUNGSAUFGABE (15 MIN)

- Implementiere:
  - Programm welches verschiedene Strings manipulieren kann
  - Bitte Schleifen verwenden und Verwendung von String-Konkatenierung und StringBuildern vergleichen

EXCEPTIONS

- Sinn von Exceptions
  - **Strukturierte** und **separate** (!) Behandlung von Fehlern, die während der Ausführung auftreten
  - Exceptions **erweitern** den Wertebereich einer Methode
- Beispiele: Array-Zugriff außerhalb der Grenze, Datei nicht gefunden, ...
- **Begriffe**: Exception, Throwing, Catching

2. - Ausnahmen, also Dinge, die eigentlich nicht passieren sollten, werden separat vom eigentlichen Programmcode behandelt und verarbeitet.
   - Das hat den Vorteil, dass der Quelltext wesentlich besser lesbar ist, da klar ist, was zur regulären Programmausführung und was zur Fehlerbehandlung gehört.
   -> Fehlercodes oft implizit und selbst sehr Fehleranfällig
3. - Exceptions erweitern somit den Wertebereich einer Methode.
   - Exceptions werden in die Signatur einer Methode aufgenommen. Schlüsselwort „throws XYZException“
4. - Wenn auf ein Element in einem Array zugegriffen werden möchte, dass außerhalb der Grenzen des Arrays liegt, wird das Programm nicht beendet.
   - Es fliegt eine IndexOutOfBoundsException. Diese kann separat vom regulären Programmcode behandelt werden.
5. - Exception: Die eigentliche Ausnahme, das Fehlerobjekt (!). Auch eine Exception ist ein Objekt.
   - Bsp: eine Instanz von IndexOutOfBoundsException oder FileNotFoundException.

   - Throwing: Auslösen bzw. werfen einer Exception.
   - Catching: Behandeln bzw. fangen einer zuvor geworfenen Exception.

ABLAUF BEIM VERWENDEN VON EXCEPTIONS

1. Ein Laufzeitfehler oder eine Bedingung des Entwicklers löst eine Exception aus
2. Diese kann nun direkt **behandelt** und/oder **weitergegeben** werden
3. Wird die Exception weitergegeben, kann der Empfänger sie wiederum behandeln und/oder weitergeben
4. Wird die Exception gar nicht behandelt, führt sie zum Programmabbruch und Ausgabe einer Fehlermeldung

- Exceptions werden stets von „innen nach außen“ geworfen, d.h. eine Methode reicht die Exception nach außen an ihren Aufrufer weiter, usw.
- Sobald eine Exception auftritt wird der normale Programmablauf unterbrochen

AUSLÖSEN UND BEHANDELN VON EXCEPTIONS

```
// Behandeln von Exceptions
public ExceptionsBeispiel() {
    // ... tue irgendwas
    try {
        tueEtwasMitEinerDatei();
    } catch (IndexOutOfBoundsException e) {
        // behandeln, weil etwas wegen einem Array kaputt ist
    } catch (FileNotFoundException e) {
        // behandeln, weil etwas wegen einer Datei kaputt ist
    } catch (Exception e) {
        // irgend etwas anderes ist kaputt gegangen.
    }
    // tue irgendwas anderes ...
}

// Auslösen von Exceptions
private void tueEtwasMitEinerDatei()
        throws IndexOutOfBoundsException,
               FileNotFoundException {
    // ...
}
```

FEHLEROBJEKTE

- Fehler**objekte**
  - Instanzen der Klasse Throwable oder ihrer Unterklasse
  - Das Fehlerobjekt enthält Informationen über die Art des aufgetretenen Fehlers
- Wichtige Methoden von Throwable
  - String getMessage(), String toString(), void printStackTrace()

```
java.lang.RuntimeException: Hier ist was dummes passiert ..
        at ba.java.weiteres.ExceptionsBeispiel.tueEtwasMitEinerDatei(ExceptionsBeispiel.java:28)
        at ba.java.weiteres.ExceptionsBeispiel.<init>(ExceptionsBeispiel.java:11)
        at ba.java.weiteres.ExceptionsBeispiel.main(ExceptionsBeispiel.java:32)
```


- Fehlerobjekte sind in Java recht ausgiebig behandelt und die Mutterklasse aller Ausnahmen ist Throwable:
Auszug aus der JavaDoc von Throwable:

```
/**
 * The {@code Throwable} class is the superclass of all errors and
 * exceptions in the Java language. Only objects that are instances of this
 * class (or one of its subclasses) are thrown by the Java Virtual Machine or
 * can be thrown by the Java {@code throw} statement. Similarly, only
 * this class or one of its subclasses can be the argument type in a
 * {@code catch} clause. ...
 */
```

- Ein Fehlerobjekt hat viele Informationen über die Art des aufgetretenen Fehlers.

4. - Wichtige Methoden von Throwable sind:
   1. getMessage(): Liefert eine für den Menschen lesbare (!) Fehlernachricht zurück.
   2. toString(): Liefert eine String Repräsentation des Fehlers zurück: Name der Exceptionklasse und Fehlernachricht.
   3. printStackTrace()
      Schreibt den Trace des aktuellen Fehlerstacks auf den Standart Error Stream (System.err). Über den StackTrace lässt sich der Ursprung des Fehlers zurückverfolgen.

- Fehlerobjekte sind in Java recht ausgiebig behandelt und die Mutterklasse aller Ausnahmen ist Throwable:

Auszug aus der JavaDoc von Throwable:

```
/**
 * The {@code Throwable} class is the superclass of all errors and
 * exceptions in the Java language. Only objects that are instances of this
 * class (or one of its subclasses) are thrown by the Java Virtual Machine or
 * can be thrown by the Java {@code throw} statement. Similarly, only
 * this class or one of its subclasses can be the argument type in a
 * {@code catch} clause. ...
 */
```

- Ein Fehlerobjekt hat viele Informationen über die Art des aufgetretenen Fehlers.

4. - Wichtige Methoden von Throwable sind:
   1. getMessage(): Liefert eine für den Menschen lesbare (!) Fehlernachricht zurück.
   2. toString(): Liefert eine String Repräsentation des Fehlers zurück: Name der Exceptionklasse und Fehlernachricht.
   3. printStackTrace()
      Schreibt den Trace des aktuellen Fehlerstacks auf den Standart Error Stream (System.err). Über den StackTrace lässt sich der Ursprung des Fehlers zurückverfolgen.

FEHLERKLASSEN VON JAVA

- Alle Laufzeitfehler sind Unterklassen von **Throwable**
- Unterhalb von Throwable existieren zwei große Hierarchien
  - **Error** ist Superklasse aller schwerwiegender Fehler
  - **Exception** ist Superklasse aller abnormalen Zustände
- Es können eigene Klassen von Exception abgeleitet werden

1. - Alles was von Throwable erbt, kann in einem catch Block gefangen werden.
3. - Error sollte die Applikation nicht selber fangen, sondern vom System verarbeiten lassen.
   - Auszug aus der JavaDoc von Error.

```
/**
 * An {@code Error} is a subclass of {@code Throwable}
 * that indicates serious problems that a reasonable application
 * should not try to catch.
 */
```

4. - Eine Exception ist ein abnormaler Zustand, den die Applikation selber fangen und behandeln kann (an sinnvoller Stelle).
   - Auszug aus der JavaDoc von Exception:

```
/**
 * The class {@code Exception} and its subclasses are a form of
 * {@code Throwable} that indicates conditions that a reasonable
 * application might want to catch.
 */
```

5. - Es ist zu vermeiden, einfach eine Instanz von Exception zu schmeißen
   - Damit würden sich Fehler nicht im catch Block unterscheiden lassen und das Konzept verliert an Mächtigkeit.

FANGEN VON FEHLERN...

- Es können einerseits durch verschiedene catch-Blöcke unterschiedliche Exception-Typen unterschieden werden
- Andererseits ist es üblich alle Fehler mittels der Oberklasse Exception gemeinsam zu behandeln, wenn eine Unterscheidung an dieser Stelle nicht nötig ist

1. - Sehr gute Unterscheidung der Fehlerfälle.
   - Manchmal nicht sinnvoll auf jede Fehlerart unterschiedlich zu reagieren.
2. - Beispiel GUI: Im Fehlerfall bei einer Aktion „Speichere Datei“ wird dem Benutzer nur angezeigt:
     - „Sorry, konnte nicht gespeichert werden. Versuch es bitte erneut.“
     - Obwohl es unterschiedliche Gründe haben kann.
   - Daher ist diese feingranulare Unterscheidung manchmal nicht sinnvoll.

DIE FINALLY KLAUSEL

- Nach dem try-catch kann **optional** eine finally-Klausel folgen
- Der Code in finally wird **immer** ausgeführt
- Die finally-Klausel ist damit der ideale Ort für Aufräumarbeiten

```
private void tueEtwas() throws FileNotFoundException {
    File file = null;
    try {
        file = new File("irgend/ein/pfad");
        String string = leseDatei(file);
        System.out.println(string);
    } catch (FileNotFoundException e) {
        // Fehlerbehandlung
        System.err.println(e.getMessage());
        throw e;
    } finally {
        schliesseDatei(file);
    }
}

private void schliesseDatei(File file) {
    // Gebe Referenz auf Datei frei
}

private String leseDatei(File file) throws FileNotFoundException{
    // Datei lesen ein und gebe Inhalt zurück
    return null;
}
```


1. - In der finally-Klausel werden üblicherweise Aufräumarbeiten getätigt.
   - Zum Beispiel: Schließen von Dateien, Freigeben von Objekten, ...
2. - Auch wenn die eigentliche Methode durch ein re-throw oder return verlassen wurde.
3. - Daher finden dort die Aufräumarbeiten statt, die unbedingt sein müssen.
   - Vorsicht: Der Zugriff auf Variablen, die im try-Block deklariert wurden ist nicht möglich.
   - Vorsicht2: Der Zugriff auf Variablen, die im try-Block instanziiert wurden, ist mit Vorsicht zu genießen! Auf null prüfen!
   - Ein try-finally ohne catch ist ebenfalls möglich.

TRY WITH RESOURCES

7

```
public void tryWithResources() {
    try (InputStream fileInput =
            new FileInputStream(new File("pfad/zur/meiner/Datei"))) {
        // Hier wird die Datei eingelesen
    } catch (Exception e) {
        // Und hier der Fehler behandelt
    }
}
```

* Mehr Beispiele bei Input/Output-Streams

1. - Durch den try-with-resources Block werden die Ressourcen in der try(..) Klausel automatisch geschlossen.
   - Schlüsselwort ist das AutoCloseable Interface, welches implementiert werden muss.
   - In diesem Beispiel würde fileInput nach Ende des Applikationscodes (oder des Ausnahmecodes) automatisch geschlossen werden.\
2. - Ab Java 8 gibt es auch try-with-multiple-resources

CATCH-OR-THROW

- Jede Exception muss **behandelt oder weitergegeben** werden
  - Eine Exception abzufangen und nicht weiter zu behandeln ist im Allgemeinen nicht sinnvoll und sollte begründet werden.

WEITERGABE VON EXCEPTIONS

- Soll eine Exception weitergegeben werden,
  - muss diese Exception durch throws angegeben sein
  - kann es entweder eine neu instanziierte Exception sein
  - oder eine Exception sein, die von „weiter unten kommt“, welche explizit behandelt wurde und mittels throw weitergeworfen wurde
  - oder eine Exception sein, die von „weiter unten kommt“, welche nicht von explizit im catch Block behandelt wurde

3. - Es passiert irgendwas im Programmablauf, was dazu führt, dass irgendwo im Code „throw new XYZException()“ steht.
4. - Eine Exception wird im catch Block gefangen, eine rudimentäre Behandlung durchgeführt (zum Beispiel geloggt) und dann die gleiche Exceptions mittels „throw existierendeException;“ weitergeworfen werden.
5. - Eine Exception wird nicht im catch Block gefangen. Dann wird diese automatisch weitergeworfen, ohne dass der folgende Code etwas davon mitbekommt.
- Doch wie kann das sein? Exceptions müssen doch immer angegeben werden?
-> Die RuntimeException macht es möglich! Siehe nächste Folie.

RUNTIMEEXCEPTION

- Unterklasse von Exception
- Superklasse aller Fehler, die **nicht behandelt** werden müssen
  - Entwickler kann entscheiden ob er diese Exception fängt
- Ausnahme von der catch-or-throw Regel
- Muss **nicht** in throws **deklariert** werden
- Eingeständnis zum Aufwand, aber gefährlich!

2. - Eine RuntimeException (und alle Klasse, die davon erben) können behandelt werden in einem catch-Block.
   - Sie müssen aber nicht behandelt werden!
3. - Einem Entwickler ist also die Freiheit gegeben, ob er eine RuntimeException fängt, oder eben nicht.
   - Es kann eine Methode, die eine RuntimeException wirft, benutzt werden, ohne dass ein catch-Block von Nöten ist.
4. - Genau aus diesem Grund ist die RuntimeException auch eine Ausnahme von der catch-or-throw Regel.
   - Wiederholung: Was besagt die catch-or-throw Regel?
   - Warum ist es also eine Ausnahme von dieser Regel?
5. - Zudem kommt der unglaublich ungünstige Umstand, dass RuntimeExceptions nicht in die Methodensignatur aufgenommen werden müssen.
   - Warum ist dies problematisch?
6. - Die RuntimeException war ein Eingeständnis an die Java Entwickler, damit der Aufwand für Fehlerbehandlung nicht zu groß wird.
   - In meinen Augen eines der gefährlichsten Features von Java. (Seit JDK1.0)
   - Die Applikation kann „auseinander brechen“, ohne dass es dem verwendeten Entwickler überhaupt klar ist was passieren kann.

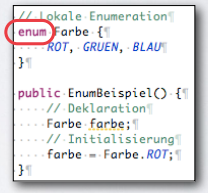
-> In meiner Erfahrung hat die RuntimeException zu sehr großen Systemabstürzen geführt!
-> Einfach aus dem Grund der Unwissenheit!

ÜBUNGSAUFGABE (15 MIN)

- Implementiere:
  - Programm mit mehreren eigenen Exceptions und deren Vererbung
  - Bitte viele Exceptions werfen und fangen und bitte auch mit expliziten catch-Blöcken, catch-all-Blöcken und finally experimentieren

ENUMERATION

- In der Praxis treten Datentypen mit kleinen und konstanten Wertemengen relativ häufig auf
- Verwendung wie andere Typen
  - z.B. Anlegen von Variablen des Typs
  - Variablen können nur vorgegebene Werte annehmen



ENUMS SIND KLASSEN

- Werte von Enums sind Objekte und werden auch so genutzt
- Alle Werte von enums sind singletons
- Die Klasse Enum selbst besitzen weitere Eigenschaften
  - values(), valueOf(String), toString(), equals(..)
  - Werte sind direkt in switch Anweisung verwendbar
- Es gibt spezielle Collections für Enums: EnumSet, EnumMap

1. - Vergleiche Beispiel von eben.
   - Enums werden deklariert und zugewiesen.
   - Namenskonvention: - Schreibweise von Enums an sich wie bei Klassen (Großbuchstabe + CamelCase)
     - Schreibweise von Werten in Großbuchstaben und Unterstrich
2. - Allerdings sind alle Werte von Enums singletons.
   - D.h. alle Werte existieren **genau ein mal** pro Applikation.
   - Wiederholung: Was hat das also zur Folge? Wie oft ist eine Farbe ROT also in einer Applikation vorhanden?
4. - values() liefert einen Array von allen Werten die für das Enum definiert sind.
   - valueOf(„ROT“) liefert den Enum Wert, welcher zur Farbe ROT gehört (Schreibweise wichtig!).
     -> Wird kein Wert zu dem angegebenen String gefunden, wird eine IllegalArgumentException geworfen (=RuntimeException).

   -> die Klasse enum implementiert damit das Interface: Compareable und Serializable.
5. - Werte sind direkt in einer switch Anweisung verwendbar.

   -> War bis Java7 ein echtes Feature, mittlerweile sind aber auch Strings in einer switch Anweisung verwendbar.
   - Von daher eher für alte Java Versionen interessant.
6. - Warum würde eine normale Liste wenig Sinn machen bei Enums (also Aufzählungen)?

   -> Sie sind singletons, eine Liste mit 100 Elementen die alle auf das eine Objekt „Rot“ zeigen macht wenig sinn..

Hinweis: Die Interfaces Compareable, Serializable und Collections werden wir später besprechen!

ENUMS KÖNNEN ERWEITERT WERDEN

- Enums sind Klassen ..
  - .. und lassen sich auch so definieren

```
public enum Farbe {
    ROT(255, 0, 0), GRUEN(0, 255, 0), BLAU(0, 0, 255);

    private int r, g, b;

    private Farbe(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public String toRGB() {
        return "r: " + r + ", g: " + g + ", b: " + b;
    }
}
```


1. - Modifier des Konstruktors ist auf private limitiert.
   - Frage Warum?
   -> Wenn er public wäre, könnten sich externe Klassen ja diesem Konstruktor bedienen und sich eigene Rot Objekte machen.
   - Modifier von Werten des Enums ist implizit public.

GENERICS

```java
package ba.java.weiteres.generics;

public class AlteListe {
  private Object[] data;
  private int size;

  public AlteListe(int maxSize) {
    this.data = new Object[maxSize];
    this.size = 0;
  }

  public void addElement(Object element) {
    if (size >= data.length) {
      throw new ArrayIndexOutOfBoundsException();
    }
    data[size++] = element;
  }

  public Object elementAt(int index) {
    if (size >= data.length) {
      throw new ArrayIndexOutOfBoundsException();
    }
    return data[index];
  }
}
```

```java
package ba.java.weiteres.generics;

public class AlteListeVerwendung {

  public AlteListeVerwendung() {
    // Hmm...eigentlich will ich nur Integer zulassen, geht aber nicht :-(
    AlteListe liste = new AlteListe(10);
    liste.addElement(5);
    liste.addElement(1.5); // führt zu keinem Compiler-Fehler!

    // Der Rückgabetyp ist Object
    Object objectFromListe = liste.elementAt(0);
    Integer integerFromListe = (Integer) objectFromListe;
  }
}
```


Anmerkung:
Man kann in Java keinen neuen generischen Array anlegen. Daher wird hier auf Objekt gearbeitet.

BEDARF NACH GENERICS

- bis Java 1.4
  - Wenn man einen Typ nicht explizit einschränken wollte, musste man auf Object arbeiten
  - Dies ist nicht typsicher -> Rückgabewert Object
  - Daher wurde sehr viel gecastet
  - Vor allem bei Collections ein prinzipielles Problem

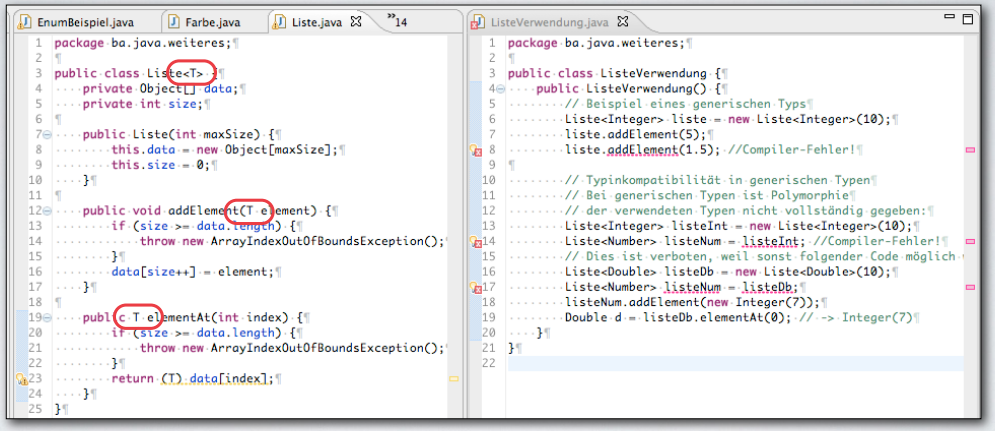
1. - Ohne Generics konnten Klassen, die andere Typen verwenden diese aber nicht explizit einschränkten wollte, lediglich auf Object arbeiten
2. - Dies ist nicht typsicher, das heißt: Die Klasse kann in ihren Methoden zwar alle möglichen Objekte aufnehmen, bei der Rückgabe von gespeicherten Objekten kann der Typ der Objekte allerdings nicht mehr unterschieden werden.
   - Beispiel: Ich möchte eine Liste wo nur „ähnliche Objekte“ drin sind. Also zum Beispiel nur Autos oder Unterklassen.
   -> Nun ist es Schwachsinn für Jede Klasse eine eigene Listenimplementierung zu machen, genau so wie es Schwachsinn ist in diesem Beispiel Autos zusammen mit Kuchen zu speichern (was bei Object allerdings möglich wäre).
3. - Da nur auf Object gearbeitet wurde, musste viel von Object z.B. nach Auto und von Auto nach Object gecastet werden.
4. - Wie bereits in den Beispielen angesprochen ist dies vor allem bei Collections ein sehr großes Problem.
   - Eine Collection ist im Prinzip wie ein Array, nur wesentlich komfortabler.

SINN VON GENERICS

- Ab Java 5
  - Mittels Generics können typsichere Klassen unabhängig vom konkreten verwendeten Typ definiert werden
  - Die Klasse wird so implementiert, dass sie mit jedem Typ zusammenarbeiten kann (Kann eingeschränkt werden)
  - Mit welchem Typ sie zusammen arbeiten muss, wird bei Initialisierung festgelegt
- Generics ersetzen die Arbeit mit Object dennoch nicht (ganz)

3. - Mit welcher konkreten Klasse eine Klasse zusammen arbeiten muss, kann bei der Initialisierung festgelegt und eingeschränkt werden.
5. - Beim Instanziieren einer generischen Klasse muss der konkrete Typ, mit dem gearbeitet werden soll, angegeben werden.
- Danach kann diese Instanz nur mit diesem Typ arbeiten.
- Es gibt jedoch immer wieder Fälle, in denen der konkrete Typ irrelevant ist.
-> In diesen Fällen ist es nach wie vor gut, allgemein auf Object zu arbeiten.
Beispiel: List<Object> meinEigentum;

BEISPIEL



EnumBeispiel.java | Farbe.java | Liste.java | 14

```
1  package ba.java.weiteres;¶
2  ¶
3  public class Liste<T> {¶
4      private Object[] data;¶
5      private int size;¶
6  ¶
7      public Liste(int maxSize) {¶
8          this.data = new Object[maxSize];¶
9          this.size = 0;¶
10     }¶
11 ¶
12     public void addElement(T element) {¶
13         if (size >= data.length) {¶
14             throw new ArrayIndexOutOfBoundsException();¶
15         }¶
16         data[size++] = element;¶
17     }¶
18 ¶
19     public T elementAt(int index) {¶
20         if (size >= data.length) {¶
21             throw new ArrayIndexOutOfBoundsException();¶
22         }¶
23         return (T) data[index];¶
24     }¶
25 }¶
```

ListeVerwendung.java

```
1  package ba.java.weiteres;¶
2  ¶
3  public class ListeVerwendung {¶
4      public ListeVerwendung() {¶
5          // Beispiel eines generischen Typs¶
6          Liste<Integer> liste = new Liste<Integer>(10);¶
7          liste.addElement(5);¶
8          liste.addElement(1.5); //Compiler-Fehler!¶
9  ¶
10         // Typinkompatibilität in generischen Typen¶
11         // Bei generischen Typen ist Polymorphie¶
12         // der verwendeten Typen nicht vollständig gegeben:¶
13         Liste<Integer> listeInt = new Liste<Integer>(10);¶
14         Liste<Number> listeNum = listeInt; //Compiler-Fehler!¶
15         // Dies ist verboten, weil sonst folgender Code möglich
16         Liste<Double> listeDb = new Liste<Double>(10);¶
17         Liste<Number> listeNum = listeDb;¶
18         listeNum.addElement(new Integer(7));¶
19         Double d = listeDb.elementAt(0); //-> Integer(7)¶
20     }¶
21 }¶
22
```


Anmerkung:
Man kann in Java keinen neuen generischen Array anlegen. Daher wird hier auf Objekt gearbeitet.

WILDCARD ? ALS TYPPARAMETER

- „?“ Kann anstelle eines konkreten Typs angegeben werden, wenn der Typ selbst keine Rolle spielt
- Damit geht die Typsicherheit verloren, allerdings explizit
- Nur bei lesendem Zugriff erlaubt.Kein `new Liste<?>` möglich!

```
public void printAll(List<?> l) {
    for (Object o : l) {
        System.out.println(o);
    }
}
```

GEBUNDENE WILDCARDS

- Abgeschwächte Form der ? Wildcard durch Einschränkung auf Subklassen einer gegebenen Superklasse mittels extends
- Es ist auch eine Einschränkung nach oben durch super möglich
- Auch die gebundene Wildcard ist nur lesend erlaubt

```
// Vorsicht: Das ist nicht unsere Klasse Liste!
public void printAllNumber(List<? extends Number> list) {
    List<? extends Number> liste2; // Auch dies ist lesend!
    for(Number numb : list) {
        System.out.println(numb.doubleValue());
    }
}
```


2. - Aber äußerst selten!

PROJECT COIN - DIAMOND OPERATOR


```
public void projectCoin() {
    List<Integer> integerList = new LinkedList<Integer>();
    Map<Integer, List<Integer>> integerZuIntegerListMap =
            new HashMap<Integer, List<Integer>>();
    Map<Integer, Map<Integer, List<Integer>>> blabla =
            new HashMap<Integer, Map<Integer, List<Integer>>>();
    // Das ist zu viel! Daher ProjectCoin ab Java7
    // Der Compiler weiß schon am Besten, was ich initialisieren will,
    // kümmert er sich auch um die Generic Handling
    List<Integer> integerList7 = new LinkedList<>();
    Map<Integer, List<Integer>> integerZuIntegerListMap7 = new HashMap<>();
    Map<Integer, Map<Integer, List<Integer>>> blabla7 = new HashMap<>();
}
```

LAMBDAS

- Anonyme innere Klassen mit neuer Syntax zu verwenden
- Nur für Interfaces mit **einer** Methode


```java
public class Lambdas {
    public static void main(String[] args) {
        Button btn = new Button();
        // Java < 8
        btn.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
        // Java >= 8
        btn.setOnAction(
            event -> System.out.println("Hello World!")
        );
    }
}
```

METHODEN REFERENZEN

- Übergabe von Methoden als Referenzen in andere Methoden

```java
public class Lambdas {
    public int compareByLength(String in, String out) {
        return in.length() - out.length();
    }

    public static void main(String[] args) {
        Lambdas myObject = new Lambdas(args);
        // Man kann in Java 8 die Referenz einer Methode übergeben
        Arrays.sort(args, myObject::compareByLength);
    }
}
```

BULK OPERATIONS

- Bulk Operationen liefern Möglichkeit etwas auf allen Elementen einer Collection zu tun (durch Lambdas)
- z.B. filtern, manipulieren, …

```java
public class BulkOperations {

    public static void main(String[] args) {
        List<String> names = Arrays.asList("Smith", "Adams", "Crawford");
        List<Person> people = find("London");

        // Streams verwenden um zu filtern
        List<Person> anyMatch = people.stream().
            filter(p -> (names.stream().anyMatch(p.name::contains))).collect(Collectors.toList());

        // Statt anyMatch kann man auch gezielter suchen
    }

    public static List<Person> find(String location) { ... }
}
```

ÜBUNGSAUFGABE (15 MIN)

- Implementiere:
  - Programm mit mehreren Nutzungen von Java 8 Streams
  - Bitte experimentiert mit stream(), map(), reduce(), filter() und wandelt Listen zu Streams und zurück.

JAVA

Wiederholung

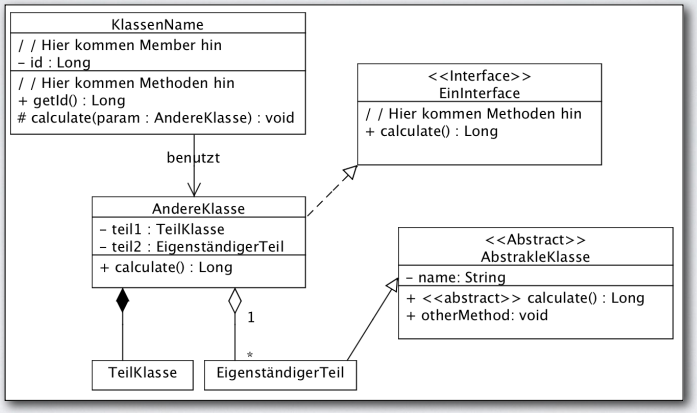
WIEDERHOLUNG ALLGEMEIN

- public static void main(String args[]) {...}
- Namenskonventionen:
  - KlassenNamen beginnen mit Großbuchstaben
  - Variablen- und Methoden-Namen mit kleinem Buchstaben
  - Beide haben CamelCase

WDH REFERENZEN

```java
public class Rechteck {

  private int laenge;

  private int breite;

  public Rechteck(int laenge) { this(laenge, 12); }

  public Rechteck(int laenge, int breite) {
    this.laenge = laenge;
    this.breite = breite;
  }


  private static Rechteck doSomething(Rechteck rechteck) {
    return rechteck;
  }

  public static void main(String[] args) {
    Rechteck rechteck1 = new Rechteck(1, 2);
    Rechteck rechteck2 = new Rechteck(1, 2);
    System.out.println(rechteck1 == rechteck2); // false, obwohl Inhalt gleich
    System.out.println(rechteck1 == doSomething(rechteck1)); // true
    String string1 = "rechteck";
    String string2 = "rechteck";
    String string3 = new String(string2);
    System.out.println(string1 == "rechteck"); // true
    System.out.println(string1 == string2); // true
    System.out.println(string1 == string3); // false
    System.out.println(compareStringsReference(string1, string2)); // true
    System.out.println(compareStringsReference(string1, string3)); // false
    System.out.println(compareStrings(string1, string2)); // true
    System.out.println(compareStrings(string1, string3)); // true
  }

  public static boolean compareStringsReference(String string1, String string2) {
    return string1 == string2;
  }

  public static boolean compareStrings(String string1, String string2) {
    if (string1 != null) {
      return string1.equals(string2);
    }
    return string1 == string2;
  }

}
```

UML


KlassenName
/ / Hier kommen Member hin
– id : Long
/ / Hier kommen Methoden hin
+ getId() : Long
# calculate(param : AndereKlasse) : void
benutzt
<<Interface>>
EinInterface
/ / Hier kommen Methoden hin
+ calculate() : Long
AndereKlasse
– teil1 : TeilKlasse
– teil2 : EigenständigerTeil
+ calculate() : Long
<<Abstract>>
AbstrakleKlasse
– name: String
+ <<abstract>> calculate() : Long
+ otherMethod: void
1
*
TeilKlasse
EigenständigerTeil

ÜBUNG (30 MIN)

- OO-Modellierung einer Bibliothek
  - Die Bibliothek besitzt Bücher und Zeitschriften, welche an Studenten ausgeliehen werden
  - Um die Ausleihfrist zu überprüfen wird notiert, wann etwas ausgeliehen wird
- Ziel: Klassendiagramm mit Klasse, Attributen, Methoden und Beziehungen

JAVA

Java Klassenbibliotheken

AGENDA

- Allgemeines
- Collections
- Utility-Klassen
- Dateihandling
- Reflection
- Weiterführende API

UMFANG DER KLASSENBIBLIOTHEKEN

- Die Klassenbibliothek von Java ist groß und mächtig
- Im Rahmen dieser Vorstellung:
  - Was gibt es wichtiges und wo finde ich es?
- Die Klassenbibliothek ist gut in JavaDoc dokumentiert

4. - Soll ein Element der Klassenbibliothek benutzt werden, lohnt es sich in die Dokumentation zu schauen.
   - Alternativ kann auch direkt der Sourcecode von Java angeschaut werden.
   - Ich würde einfach Control+LinksKlick machen auf einer Klasse, die ich benutzen möchte und einfach in den Source Code rein schauen.

COLLECTIONS

- Eine Collection ist eine Datenstruktur, um Mengen von Daten aufzunehmen und zu verarbeiten
  - Die Verwaltung wird gekapselt
- Ein Array ist einfachste Art der Collection
  - Collections sind aber mächtiger und einfacher zu benutzen
  - Daher werden Arrays nur relativ selten in Java benutzt

2. - Zugriff auf die Daten ist nur über definierte Methoden möglich.
   - Ermöglicht neue Sichten auf die Daten, beispielsweise Zugriff auf die Daten über Schlüssel.
5. - Dass wir in dem Beispiel das letzte mal Arrays benutzt haben war, weil die Collections schon eine reichhaltige API zur Sortierung haben und wir ja von Hand sortieren wollten.

WICHTIGSTE COLLECTIONS

- java.util.*
- Seit Java 5 sind die Collections generisch
- Namens Konvention: <Stil><Interface>
- Wichtigste Interfaces: **List, Set, Map, Queue**
- **Tree** vs. **Hash** Implementierungen

3. - Beispiel: ArrayList, Array ist der Stil und List das Interface
HashMap, Hash ist der Stil und Map das Interface
4. - List = Geordnete Collections, die Duplikate erlauben
Set = Collections ohne Duplikate
Map = Collections, die eine Schlüssel/Wert-Ablage erlaubt
Queue = Collections, die sog. „Warteschlangen“ darstellen, und Objekte meist FIFO (first-in-first-out) behandeln
5. - TreeMap vs. HashMap und TreeSet vs HashSet
JavaDoc von TreeMap
* <p>This implementation provides guaranteed log(n) time cost for the
* {@code containsKey}, {@code get}, {@code put} and {@code remove}
* operations. Algorithms are adaptations of those in Cormen, Leiserson, and
* Rivest's Introduction to Algorithms. */
JavaDoc von HashMap
/* <p>This implementation provides constant-time performance for the basic
* operations (<tt>get</tt> and <tt>put</tt>), assuming the hash function
* disperses the elements properly among the buckets. */

Daher ist die HashMap vorzuziehen, außer man will später auch eine geordnete Ausgabe wieder bekommen.

Ihr werdet unterschiede im Laufe des Studiums noch ausreichend erklärt bekommen :-)

ARBEITSWEISE HASH-TABELLE

```
Map<Integer, String> mitarbeiter = new HashMap<>();
mitarbeiter.put(4711, "Klaus");
mitarbeiter.put(4712, "Peter");
mitarbeiter.put(4747, "Hans");
System.out.println(mitarbeiter.get(4711));
```

- Schlüssel-Wert-Paare
- Berechnung Schlüssel durch Hash-Funktion
- Schlüssel dient als Index eines internen Arrays
- Füllgrad bzw. load factor

- Die Hash-Tabelle arbeitet mit Schlüssel-Werte-Paaren. Aus dem Schlüssel wird nach einer mathematischen Funktion – der so genannten Hash-Funktion – ein Hashcode berechnet.
- Dieser dient dann als Index für ein internes Array.
- Dieses Array hat am Anfang eine feste Größe. Wenn später eine Anfrage nach dem Schlüssel gestellt wird, muss einfach diese Berechnung erfolgen, und wir können dann an dieser Stelle nachsehen.
- Falls eine Kollision auftritt, wird ein kleines Behälterobjekt mit dem Schlüssel und Wert aufgebaut und als Element an die Liste angehängt. Eine Sortierung findet nicht statt.

4. - Maß des Füllstandes
   - Zwischen 0% und 100%.
   - Für performanten Zugriff sollte ein Füllstand von 75% nicht überschritten werden. Standard-Implementierungen beachten dies und vergrößern dynamisch die Array-Größe. Übernimmt Java allerdings für euch!

IMPLEMENTIERUNGEN LIST

- ArrayList
- LinkedList
- Stack
- Vector

1. - ArrayList: Implementation eines „Resizable-Array“.
   Die Größe kann vorgegeben werden, wächst aber auch mit.
2. - LinkedList: Implementierung einer verlinkten Liste.
   Spezielle Methoden zum Zugriff auf das erste und letzte Element.
   Implementiert nicht nur das List Interface sondern auch das Queue Interface.
   Kann daher auch als Stack, Queue oder Double-Ended Queue (= Deque) eingesetzt werden.
3. - Stack: Klassischer LIFO (last-in-first-out) Speicher.
   Bietet die typischen push/pop Operationen.
4. - Vector: Implementation eines „Resizable-Array“
   Verhält sich wie eine ArrayList.
   Existiert historisch länger als eine ArrayList und ist im Gegensatz zu dieser synchronisiert (Multithreading).

An dieser Stelle aber die Vererbung von Stack beachten.

IMPLEMENTIERUNGEN SET

- EnumSet
- HashSet
- TreeSet

1. - EnumSet: Spezialisiertes Set für Enums
   Alle Elemente des Sets müssen von einem gemeinsamen Enum stammen.
2. - HashSet: Set, das auf einer Hashtabelle aufbaut
   Die Reihenfolge der Elemente spielt hier keine Rolle.
   Die Performance der Operationen (add, …) ist konstant.
   Elemente müssen Hash-fähig sein (Methode hashCode()).
3. - TreeSet: Sortiertes Set
   Die Elemente werden automatisch sortiert.
   Dazu müssen die Elemente das Interface Comparable implementieren (Methode compareTo()).

IMPLEMENTIERUNGEN MAP

- EnumMap
- HashMap
- Hashtable (auch klein table!)
- TreeMap

1. - EnumMap: Spezialisierte Map für Enums als Schlüssel.
   Alle Schlüssel müssen von einem gemeinsamen Enum stammen.
2. - HashMap: Implementierung einer Hashtabelle.
   Die Reihenfolge der Elemente spielt hier keine Rolle.
   Die Performance der Operationen (put, …) ist konstant.
   Elemente müssen Hash-fähig sein (Methode hashCode()).
3. - Hashtable: Implementierung einer Hashtabelle.
   Verhält sich wie eine HashMap.
   Existiert historisch länger als eine HashMap und ist im Gegensatz zu dieser synchronisiert (Multithreading)
4. - TreeMap: Sortierte Map.
   Die Schlüssel werden automatisch sortiert.
   Dazu müssen die Schlüssel das Interface Comparable implementieren (Methode compareTo()).

Frage: Kann man in eine Map oder ein Set „null“ adden?
- auf null kann man nicht hashCode() oder compateTo() aufrufen!

DIE KLASSE COLLECTIONS

- Statische Methoden zur Manipulation und Verarbeitung von Collections
- Besonderes die Methoden zum Synchronisieren sind im Zusammenhang mit Multithreading wichtig

```
public CollectionsBeispiel() {¶
    List<Integer> meineListe = new LinkedList<>();¶
    meineListe.add(5);¶
    meineListe.add(2);¶
    meineListe.add(10);¶
    // sortiert Liste aufsteigend¶
    // Integer implementiert das Interface Comparable<Integer>¶
    Collections.sort(meineListe);¶
    // Ein spezieller Vergleicher,¶
    // der zur Sortierung heran gezogen wird¶
    Comparator<Integer> vergleicher = new Comparator<Integer>() {¶
        @Override¶
        public int compare(Integer int1, Integer int2) {¶
            return int2.compareTo(int1);¶
        }¶
    };¶
    // Liste speziell sortieren¶
    Collections.sort(meineListe, vergleicher);¶
    // dreht die Elemente der Liste (wieder) um¶
    Collections.reverse(meineListe);¶
    List<Integer> synchronisierteListe =¶
            Collections.synchronizedList(meineListe);¶
    List<Integer> unmodifizierbareListe =¶
            Collections.unmodifiableList(meineListe);¶
    Integer max = Collections.max(meineListe);¶
    // ...¶
}¶
```


1. - Durchsuchen
   - Sortieren
   - Kopieren
   - Erzeugen unveränderlicher Collections
   - Erzeugen synchronisierter Collections
2. - Nur die älteren Collections sind von Haus aus synchronisiert (Vector, Hashtable, ..).
   - Die neueren Collection-Klassen sind aus Performance-Gründen nicht thread-safe.
   - Diese können aber durch die Methoden von Collection zu solchen gemacht werden.
3. - Ab Java 9 auch `List.of(“Hans”, “Wurst”, “Müller”)` möglich

DIE KLASSE ARRAYS

- Statische Methoden zur Manipulation und Verarbeitung von Arrays
- parallellSort

```
private void arraysBeispiele() {
    Integer[] meinArray = { 5, 2, 10, 25, 1, 20 };
    List<Integer> alsListe = Arrays.asList(meinArray);
    // sortiert Liste aufsteigend
    Arrays.sort(meinArray);
    // Vergleich von Arrays.equals geht auf den Inhalt!
    Integer[] zweiterArray = { 1, 2, 3 };
    boolean vergleich = Arrays.equals(meinArray, zweiterArray);
    // Es können Elemente gesucht werden
    int indexOfSearch = Arrays.binarySearch(meinArray, 20);
    System.out.println(Arrays.toString(meinArray));
    // Gibt "[1, 2, 5, 10, 20, 25]" aus
}
```


1.
- Durchsuchen
- Sortieren
- Vergleichen
- Vorbefüllen
- Inhalt als String ausgeben
- etc.
- Im Endeffekt das gleiche wie Collections nur mit Arrays

ÜBUNGSAUFGABE (15 MIN)

- Implementiere:
  - Programm mit mehreren Collections
  - Bitte experimentiert mit List, Set und Map. Benutzt auch verschiedene equals/hashCode Implementierungen.

WICHTIGE UTILITY KLASSEN

- Utility Klassen sind Klassen, welche sich nicht konkret einordnen lassen, die man aber immer wieder benötigt
- Wichtige Utility Klassen
  - java.util.Random
  - java.security.SecureRandom
  - java.util.GregorianCalendar und java.util.Date

```
private void utilities() {
    Random random =
            new Random(System.currentTimeMillis() % 14930352);
    for(int i = 0; i<10;i++) {
        System.out.println(random.nextInt());
    }
}
```


3. - Random erzeugt Zufallszahlen.
4. - Calendar und Date sind zur Verarbeitung Datums- und Zeitwerten da.
   - Datumsarithmetik ist nicht einfach und spielt mindestens in der Liga von Zeichencodierungsproblemen.
   - Datumsarithmetik nicht hinreichend in der Java Standardbibliothek gelöst.
   - Date und Calendar sind stark veraltet, wird aber noch im Zusammenhang von Datenbanken gerne verwendet.
   - Tut euch selber einen gefallen und verwendet JodaTime (http://joda-time.sourceforge.net) oder die Java 8 Implementierung

WICHTIGE UTILITY KLASSEN

- java.lang.System
  - getProperty() (HomeVerzeichnis, Tempverzeichnis, …)
  - getEnv()
  - lineSeparator()
  - Standard Streams (in, out, err)
  - exit()
  - gc()
  - currentTimeMillis()

1. - http://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html
3. - Beendet das Programm
4. - Fordert die GC auf, sich in Gang zu setzen.
   - Wird im allgemeinen nicht benötigt und ist auch kein Garant, dass die GC startet.

WICHTIGE UTILITY KLASSEN

- java.lang.Runtime
- java.lang.Math
- java.math.BigDecimal & BigInteger

1. - Starten und interagieren mit nativen Prozessen bzw. anderen Programmen.
2. - Die Methoden zur Fließkomma-Arithmetik, z.b. Winkel-Funktionen und Quadratwurzel.
3. - Klassen für beliebig große und genaue Zahlen für arithmetische Operationen.
   - Besonders im Banken-Breich sehr wichtig.
   Wiederholung: Wie viel passt in einen Long? 8 Byte

NEUE DATE API


```java
public class NewDates {

    public static void main(String[] args) {
        Clock clock = Clock.systemUTC(); // UTC der Systemzeit

        Clock clock = Clock.systemDefaultZone(); // Uhrzeit in der Zeitzone des Systems

        long time = clock.millis(); // Millisekunden seit 01.01.1970

        ZoneId zone = ZoneId.of("Europe/Berlin"); // Zeitzone für Namen

        Clock clock = Clock.system(zone); // Setzt eine Zeitzone

        LocalDate date = LocalDate.now(); // LocalDate = menschenlesbare Zeitangaben
        String year = date.getYear();
        String month = date.getMonthValue();
        String day = date.getDayOfMonth();

        Period p = Period.of(2, HOURS);  // Zeitperioden
        LocalTime time = LocalTime.now();
        LocalTime newTime = time.plus(p);
    }
}
```

DATEIEN UND VERZEICHNISSE

- Java bietet sehr umfangreiche API zum Datei- und Verzeichnishandling an
  - Handling von Dateien und Verzeichnisse selbst
  - Streams zum sequentiellen I/O
  - Random I/O
- java.io.*

3. - Streams sind eine objektorientierte Technik zur sequentiellen Ein- und Ausgabe von Dateiinhalten
4. - Der Zeilentrenner sollte niemals hardcodiert werden, sondern immer mit System.getProperty(„line.separator“) abgefragt werden.

HANDLING VON DATEIEN

- Dateien und Verzeichnisse sind Objekte, **nicht aber der Inhalt** von Dateien!

```
public FileBeispiel() {
    // Abstrahiert von Datei und Verzeichnis
    File homeVerzeichnis = new File("/Users/junterstein");
    File präsentation = new File(homeVerzeichnis.getAbsolutePath()
            + "/documents/java12.key");
    // Absolute und relative Namen unterstützt
    File testFile = new File("./test.jar");
    // Abfrage der Datei-/Verzeichnisattribute
    System.out.println(testFile.exists() + ":" + testFile.canRead()
            + ":" + testFile.canWrite());
    // Iterieren über Verzeichnisse
    for (File kind : homeVerzeichnis.listFiles()) {
        if (kind.isFile()) {
            // tue etwas
        }
        if (kind.isDirectory()) {
            // tue etwas anderes
        }
    }
    // Anlegen und Löschen von Dateien und Verzeichnissen
    testFile.delete();
    // mkdir() legt nur den angegebenen Ordner an, wenn der Vater existiert
    // mkdirs() legt alle Ordner an, bis der übergebene Pfad erreicht ist
    new File(homeVerzeichnis.getAbsolutePath()+"/mein/ausgedachter/Pfad").mkdirs();
    // Verwalten Temporärer Dateien
    try {
        File temporäreDatei = File.createTempFile("meinProjektName", "meineDateiEndung");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

JAVA.NIO.FILE.PATHS

- Pfadverkettung bitte mit Path und Paths

```
10    public class PathBeispiel {
11
12      public static void main(String[] args) {
13        File file = new File("mein/ausgedachter/Pfad");
14        File newFile = new File(file.getAbsolutePath() + "../../was/lustiges/dran"); // <-- unüblich
15        Path path = Paths.get(file.getAbsolutePath(), "../../was/lustiges/dran");
16        Path parent = path.getParent();
17        File parentFile = parent.toFile();
18        Path parent2 = parentFile.toPath();
19      }
```

STREAMS

- Sehr abstraktes Konstrukt, zum Zeichen auf imaginäres Ausgabegerät zu schreiben oder von diesem zu lesen
  - Erste konkrete Unterklassen binden Zugriffsroutinen an echte Ein- oder Ausgabe
- Streams können verkettet und geschachtelt werden
  - **Verkettung**: Zusammenfassung mehrerer Streams zu einem
  - **Schachtelung**: Konstruktion von Streams mit Zusatzfunktion
- Bitte apache-commons verwenden: FileUtils, IOUtils, StreamUtils, …

Vergleiche: http://commons.apache.org

2. - Dateien, Strings, Netzwerkkommunikationskanäle, ...
4. - Verkettung Beispiel: mehrere Dateien als ein Stream behandeln mittels SequenceInputStream.
5. - Schachtelung: Konstruktion von Streams, die bestimmte Zusatzfunktionen übernehmen.
   - Am meisten verwendet: Puffern von Zeichen
   - BufferedInputStream(InputStream in)

   - Beide Konzepte sind mit Java Sprachmitteln realisiert und können in eigenem Code erweitert werden.

CHARACTER- UND BYTE-STREAMS

- **Byte**-Streams: Jede Transporteinheit genau 1 Byte lang
  - Problem bei Unicode (> 1 Byte) & Umständlich
- **Character**-Streams: Unicode-fähige textuelle Streams
- Wandlung von Character- und Byte-Streams und umgekehrt möglich

0. - Character Streams heißen Reader/Writer, Byte Streams heißen .. Streams (InputStream, OutputStream)
2. - Und die Transporteinheit steht nur als binäre Dateninformation zur Verfügung.

CHARACTER-STREAMS FÜR DIE AUSGABE

- Abstrakte Klasse Writer
  - OutputStreamWriter
  - FileWriter
  - PrintWriter
  - BufferedWriter
  - StringWriter
  - CharArrayWriter
  - PipedWriter

```
private void charWriter() {
    String s;
    FileWriter fw = null;
    BufferedWriter bw = null;
    try {
        fw = new FileWriter("buffer.txt");
        bw = new BufferedWriter(fw);
        for (int i = 1; i <= 10000; ++i) {
            s = "Dies ist die " + i + ". Zeile";
            bw.write(s);
            bw.newLine();
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        // Java Version < 7 Bedarf etwas
        // mehr aufwand zum stream closen
        try {
            if (bw != null) {
                bw.close();
            }
            if (fw != null) {
                fw.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


1. - Basis alles sequentiellen Character-Ausgaben.
2. - Basisklasse für alle Writer, die einen Character-Stream in einen Byte-Stream umwandeln.
3. - Writer zur Ausgabe in eine Datei als konkrete Ableitung von OutputStreamWriter
4. - Ausgabe von Textformaten.
5. - Writer zur Ausgabepufferung, um die Performance beim tatsächlichen Schreiben (der Datei) zu erhöhen.
6. - Writer zur Ausgabe eines String.
7. - Writer zur Ausgabe eines Streams in ein char-Array.
8. - Writer zur Ausgabe in einen PipedReader -> Linux Pipe Konstrukt.

CHARACTER-STREAMS FÜR DIE EINGABE

- Abstrakte Klasse Reader
  - InputStreamReader
  - FileReader
  - BufferedReader
  - LineNumberReader
  - StringReader
  - CharArrayReader
  - PipedReader

```
private void charReader() {
    String line = null;
    try (BufferedReader br =
            new BufferedReader(new FileReader("config.sys"))) {
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```


1. - Basis aller sequentiellen Character-Eingaben.
2. - Basisklasse für alle Reader, die einen Byte-Stream in einen Cahracter-Stream umwandeln.
3. - Reader zum Einlesen aus einer Datei als konkrete Ableitung von InputStreamReader.
4. - Reader zur Eingabepufferung und zum Lesen kompletter Zeilen.
5. - Ableitung des BufferedReader mit der Fähigkeit, Zeilen zu zählen.
6. - Reader zum Einlesen von Zeichen aus einem String.
7. - Reader zum Einlesen von Zeichen aus einem Char-Array.
8. - Reader zum Einlesen von Zeichen aus einem PipedWriter -> Beispiel 1 Slide vorher.

BYTE-STREAMS

- Bei Character-Streams wird von Readern und Writern (analog zur jeweiligen Basisklasse) gesprochen, hier spricht man von Byte-Streams, also InputStreams und OutputStreams
- Abstrakte Klasse **InputStream** und **OutputStream**
- Funktionieren genauso wie Character-Streams, arbeiten jedoch auf Bytes
- Spezialfall: ZipOutputStream und ZipInputStream im Paket java.util.zip zum Schreiben und lesen von Archivdateien

Vergleiche: http://commons.apache.org

2. - Inklusive einer analogen Klassenhierarchie wie bei Writern und Readern.

RANDOM I/O

- Streams vereinfachen den sequentiellen Zugriff auf Dateien
- Manchmal wahlfreier Zugriff auf Dateien notwendig
- RandomAccessFile stellt entsprechende Methoden zur Verfügung um in Dateien zu navigieren/lesen/schreiben

ÜBUNGSAUFGABE (15 MIN)

- Implementiere:
  - Programm mit Lesen und Schreiben von Dateien
  - Bitte lest und schreibt sowohl Bytes als auch Character in eine Datei.

INTROSPECTION & REFLECTION

- Möglichkeit, zur Laufzeit Klassen zu instanziieren und zu verwenden ohne diese zur Compilezeit zu kennen
- Abfragemöglichkeiten, welche Member eine Klasse besitzt
- Reflection ist ein sehr mächtiges Werkzeug, sollte aber mit Bedacht eingesetzt werden
  - Reflection Code schlägt oft erst zur Laufzeit fehl
  - Macht Code schwer lesbar

1. - Wichtig zum Beispiel für Anwendungen mit PlugIn-Schnittstellen
   - Prominentestes Beispiel: Eclipse

INTROSPECTION & REFLECTION

- Die Klasse java.lang.Class

```
try {
    String meinKlassenName = "ba.java.auto.AudiQFuenf";
    Class<?> meineKlasse = Class.forName(meinKlassenName);
    Object meinObjekt = meineKlasse.newInstance();
    AudiQFuenf meinAudi = (AudiQFuenf) meinObjekt;
} catch (Exception e) {
    e.printStackTrace();
}
```

  - Erreichbar über getClass() der Klasse Object
  - Methoden zur Abfrage der Member der Klasse, sowie weiterer Eigenschaften

```
try {
    String str = "Test";
    Class<?> clazz = str.getClass();
    Method m = clazz.getDeclaredMethod("length");
    Object ret = m.invoke(str);
    System.out.println(ret);
} catch (Exception e) {
    e.printStackTrace();
}
```


3. - Über die Relections API sind ganz hässliche Hacks möglich.
-> Es können modifier von Klassen verändert werden und auch auf private Member zugegriffen werden (lesen und schreibend).
- Welches Tool, was ihr alle schon benutzt habt, verwendet ganz Exzessiv die Reflections API und die Möglichkeit auf private Member zuzugreifen?
-> Jeder Debugger.

SERIALISIERUNG

- Fähigkeit, ein Objekt im Hauptspeicher in ein Format zu konvertieren, um es in eine Datei zu schreiben oder über das Netzwerk zu transportieren
- Deserialisierung: Umkehrung der Serialisierung in ein Objekt
- Manchmal wird Serialisierung zur Persistenz eingesetzt
  - Meist nicht die klassische Serialisierung, sondern XML, JSON
- Standard Serialisierung ist binärbasiert!

1. - Bei Client/Server Anwendungen spielt Serialisierung oft eine Rolle, um Objekte zwischen Client und Server zu transportieren.
   - Das Objekt wird in einem binären Format gespeichert, also nicht für den Menschen lesbar!
3. - In kleineren Anwendungen ohne Datenbank werden Objekte halt in eine Datei geschrieben und können aus dieser wieder geladen werden.
4. - XML bietet eigene Stream Writer und Reader mit Java Sprachmitteln.
5. - Nachteile von Binärserialisierung (Auszug „Java ist auch eine Insel“):

Der klassische Weg von einem Objekt zu einer persistenten Speicherung führt über den Serialisierungsmechanismus von Java über die Klassen ObjectOutputStream und ObjectInputStream. Die Serialisierung in Binärdaten ist aber nicht ohne Nachteile. Schwierig ist beispielsweise die Weiterverarbeitung von Nicht-Java-Programmen oder die nachträgliche Änderung ohne Einlesen und Wiederaufbauen der Objektverbunde. Wünschenswert ist daher eine Textrepräsentation. Diese hat nicht die oben genannten Nachteile.

Ein weiteres Problem ist die Skalierbarkeit. Die Standard-Serialisierung arbeitet nach dem Prinzip: Alles, was vom Basisknoten aus erreichbar ist, gelangt serialisiert in den Datenstrom. Ist der Objektgraph sehr groß, steigt die Zeit für die Serialisierung und das Datenvolumen an. Verglichen mit anderen Persistenz-Konzepten, ist es nicht möglich, nur die Änderungen zu schreiben. Wenn sich zum Beispiel in einer sehr großen Adressliste die Hausnummer einer Person ändert, muss die gesamte Adressliste neu geschrieben werden – das nagt an der Performance.

XML SUPPORT

- eXtensible Markup Language
- XML ist ein semi-strukturiertes Datenformat
- Es können in XML eigene Strukturen abgebildet werden, die jedoch nur mit eigenem allgemeinen Parser verarbeitet werden können
- Java liefert DOM und SAX Parser mit und bietet Möglichkeiten zur Transformation von XML Dokumenten

4. - Unterschied DOM zu SAX?
   - DOM baut den gesamten XML Baum erstmal im RAM auf und dann kann bequem auf dem Baum navigiert und gearbeitet werden.
   - SAX ließt das XML sequentiell ein und schmeißt Events, wenn bestimmte Sachverhalte eintreten.
   -> SAX ist unbequemer, aber wesentlich speicherfreundlicher.
0. - Es ist allerdings die Verwendung von JAXB zu empfehlen.
   - Java Architecture for Xml Binding
   - Erlaubt es normale Java Klassen per Annotation automatisch in XML zu serialisieren und zu deserialisieren.

ANNOTATIONS

- Möglichkeit Java Code mit **Meta** Informationen zu versehen
- Nur ganz kurze Einführung in Zusammenhang mit JAXB!
- @Override, ...

```
 9  public class AnnotationBeispiel {
10
11      public AnnotationBeispiel() {
12          UserBean bean = new UserBean();
13          bean.name = "James";
14          bean.surename = "Hetfield";
15          bean.birthDate = new Date(63, 7, 3); // 3. August 1963
16          // XMLSerializeHelper ist kein Bestandteil von Java
17          System.out.println(XMLSerializeHelper.instance().serialize(bean));
18      }
19
20      @XmlRootElement
21      public static class UserBean implements Serializable {
22          private static final long serialVersionUID = -9091053716626960 39L;
23
24          @XmlElement(name = "vorname")
25          public String name;
26          public String surename;
27          @XmlElement(name = "geburtstag")
28          public Date birthDate;
```

Console

```
<terminated> AnnotationBeispiel [Java Application] /Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/H
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<userBean>
    <vorname>James</vorname>
    <surename>Hetfield</surename>
    <geburtstag>1963-08-03T00:00:00+01:00</geburtstag>
</userBean>
```

JDBC

- Java Database Connectivity
- Schnittstelle zwischen einer Applikation und einer SQL-DB
- Es sind weitere Treiber der Datenbank-Hersteller notwendig

3. - Java gibt nur die Schnittstelle vor, aber nicht die Datenbank-abhängige Implementierung.
0. - Weitere OR-Mapper sind nicht im Sprachkern enthalten.
- Die javax.persistence API kommt in der JavaEE mit und die Implementierung von ... mehr oder weniger guten Drittherstellern ;)

NETZWERK

- Java stellt API zur Socket-Programmierung via TCP/IP bereit
- Über Streams ist das Lesen und Schreiben möglich
- Dies ist im Allgemeinen umständlich, weshalb meist andere Mechanismen verwendet werden

0. - Welche Möglichkeiten über das Netzwerk zu kommunizieren sind etabliert und verbreitet?
   - Webservices - REST/SOAP - Wobei REST schon besser ist :)

RMI

- Remote Method Invocation
- Socket Programmierung erlaubt Austausch von Daten
  - RMI erlaubt transparente Methodenaufrufe auf Server
- RMI abstrahiert
- RMI ist nur eine Möglichkeit für verteilte Objekte

4. - Dem Verwender eines Objektes kommt es vor, als würde er auf einem lokalen Objekt arbeiten.
   - Java RMI kümmert sich um die Übertragung der Objekte über das Netzwerk.
5. - CORBA, ...
   - Der Ansatz von Webservices adressiert ein viel größeres Feld an Möglichkeiten, wie es die bloße RMI kann.
   - Die Vorlesung verteilte Systeme im 4. oder 5. Semester beschäftigt sich sehr intensiv damit --> Aufpassen :)

SICHERHEIT UND KRYPTOHRAPHIE

- Daten müssen oft verschlüsselt werden
  - Verschlüsselung von Dateien
  - Verschlüsselung von Daten bei Netzwerktransport
- Symmetrische, Asymmetrische Verfahren und Signaturen unterstützt

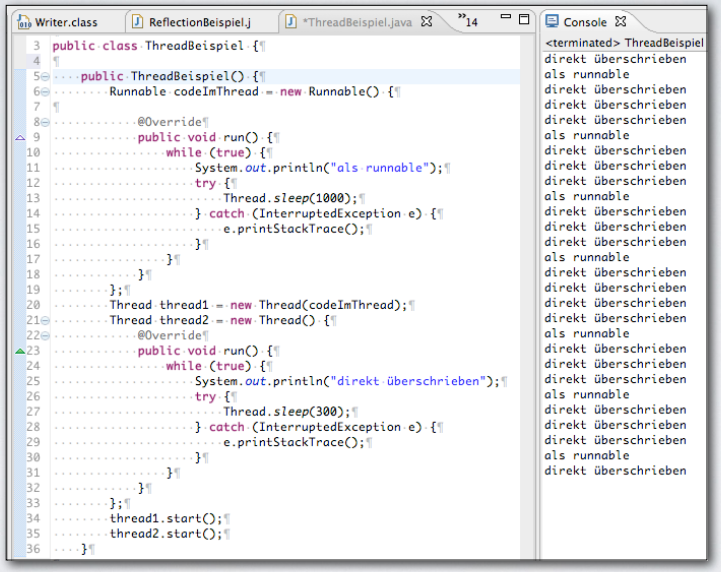
2. - Cipher cipher = Cipher.getInstance(algorithmName);

MULTITHREADING

- Konzepte der Nebenläufigkeit von Programmteilen
  - Ähnelt einem Prozess, arbeitet aber auf einer feineren Ebene
- Threads sind in Java direkt als Sprachkonstrukt umgesetzt
- Threads können synchronisiert werden

1. - Beispiel: GUI Thread mit Sound abspielen, Eingabe erfassen und Eingabe validieren
2. - Vergleiche Vorlesung „Betriebssysteme“
4. - Schlüsselwort synchronized

MULTITHREADING BEISPIEL




```
3  public class ThreadBeispiel {
4
5      public ThreadBeispiel() {
6          Runnable codeImThread = new Runnable() {
7
8              @Override
9              public void run() {
10                 while (true) {
11                     System.out.println("als runnable");
12                     try {
13                         Thread.sleep(1000);
14                     } catch (InterruptedException e) {
15                         e.printStackTrace();
16                     }
17                 }
18             }
19         };
20         Thread thread1 = new Thread(codeImThread);
21         Thread thread2 = new Thread() {
22             @Override
23             public void run() {
24                 while (true) {
25                     System.out.println("direkt überschrieben");
26                     try {
27                         Thread.sleep(300);
28                     } catch (InterruptedException e) {
29                         e.printStackTrace();
30                     }
31                 }
32             }
33         };
34         thread1.start();
35         thread2.start();
36     }
```

ÜBUNG (30 MIN)

- OO-Modellierung einer Bibliothek
  - Die Bibliothek besitzt Bücher und Zeitschriften, welche an Studenten ausgeliehen werden
  - Um die Ausleihfrist zu überprüfen wird notiert, wann etwas ausgeliehen wird
- Ziel: Klassendiagramm mit Klasse, Attributen, Methoden und Beziehungen

JAVA 9 & 10

JAVA 9 FEATURES

- Neues Modulsystem - Projekt Jigsaw
- HTTP/2.0 Unterstützung
- Verbesserung der Streaming API
- List.of("France", "Bulgaria", "Germany");
- Try-with-multiple-resources

JAVA 10 FEATURES

- Verbesserte Typ-Inferenz

```
1  // int
2  var zahl = 5;
3  // String
4  var string = "Hello World";
5  // BigDecimal
6  var objekt = BigDecimal.ONE;
```

-

JAVA 11 FEATURES

- Single File Sources startbar `java HelloWorld.java`
- Viele Wartungsarbeiten, GC, …

JAVA 12 FEATURES

- Viele Wartungsarbeiten, GC, …

JAVA

Tests

TESTING

- Tests sind wichtig
- Tests erleichtern euer Leben und beschleunigen
- Tests bringen Vertrauen und Sicherheit
- Tests bringen nichts, wenn sie nichts testen ¯_(ツ)_/¯

JUNIT

- Externe Bibliothek
- Defacto Standard
- @Test
- Assertions
- import static org.junit.jupiter.api.Assertions.*

ÜBUNGSAUFGABE (15 MIN)

- Implementiere:
  - Schreibt Tests für die Gehaltsberechnung

JAVA

Oberflächen mit ~~Swing~~

Grafik von http://javafx.com

AGENDA

- Grundlagen GUI
- Swing
- JavaFX

GRAPHISCHE BENUTZEROBERFLÄCHEN

- AWT = Abstract Window Toolkit
  - Verwendet die jeweiligen GUI-Komponenten des OS
  - Bietet nur eine Grundmenge von Elementen aller OS
- Swing
  - Swing werden direkt von Java gerendert und ist OS frei
  - Mehr Elemente und bequemere API
  - Oft kein natives OS-Verhalten und etwas langsamer

2. - Wegen Swing sind Java Desktop Applikationen so in Verruf geraten.
   - Man merkt den Applikationen eben an, dass es Java ist.
0. - Weitere Oberflächentechnologien von Java:
   - SWT -> Standard Window Toolkit
   - Von eclipse (entwickelt und verwendet)
   - Verwendet native UI Komponenten, reduziert sich jedoch auf den kleinsten gemeinsamen Nenner aller.
   - Rest wird in pure Java implementiert.

GRAPHISCHE BENUTZEROBERFLÄCHEN

- JavaFX
  - Ab Java 7 Teil vom JDK
  - Aktuelle Version 12
  - Portabel
  - Direkt von Java gerendert

1. - Kann aber auch separat heruntergeladen werden
2. - JavaFX 1.x kann nicht mit 2.x verglichen werden.
   - 2007 hat Sun JavaFX offiziell angekündigt, war aber sehr experimentell.
   - 2008 kam JavaFX 1.0 als Entwicklerkit für Mac und Windows.
   - 2009 kam Version 1.2 mit mehr UI Komponenten und Linux support, aber keiner Abwärtskompatibilität.
   - Bisher war es nicht so wirklich gut benutzbar und nie abwärtskompatibel nach einem Release.
   - 2011 kam Version 2.0 und es wurde bisheriger Scriptansatz verworfen und aktuelle Strategie implementiert.
   - 2.0 unterstützt Hardware-Rendering
3. - JavaFX kann auf dem Desktop ausgeführt werden, allerdings auch als WebAnwendung auf den Client herunter geladen werden.
   - Weiterhin werden JavaFX im Browser laufen, auf SetTop Geräten, Mobilfunkgeräten, ...
   - Eher der webbasierte Weg
   - UI Komponenten können per CSS gestyled werden.
4. - Ähnlich zu Swing.
   - Aber können mit unterschiedlichem Look and Feel für unterschiedliche Betriebssysteme angezeigt werden.

SWING

- Alle Swing UI Komponenten sind java.awt.Container und können andere Komponenten beinhalten
- Ermöglicht ein einheitliches Konzept zum Rendering
- Scope dieses Abschnittes: Grundkonzept kennen

1. - Baut auf AWT auf, verwendet einige AWT Klassen
   – Aber: Komplett in Java entwickelt, verwendet keine Betriebssystemresourcen
   - Pluggable Look-and-Feel
   - Plattformunabhängig, da Rendering über Java Komponenten
3. - http://docs.oracle.com/javase/tutorial/uiswing/components/index.html
   - http://docs.oracle.com/javase/tutorial/ui/features/components.html

SWING
JCOMPONENT

- JComponent ist die Basisklasse aller Swing UI Komponenten
- Erbt von java.awt.Container ->Swing-Elemente sind Container
- JComponent unterstützt viele Features
  - ToolTips, Border, Pluggable Look & Feel, Drag & Drop

2. - Ein Button kann z.b. ein Icon und ein Text beinhalten

HALLO SWING

```
package ba.java.swing;¶
¶
import javax.swing.JFrame;¶
import javax.swing.JLabel;¶
¶
public class SwingApp {¶
    public SwingApp() {¶
        JFrame frame = new JFrame("Erste Swing Applikation");¶
        frame.getContentPane().add(new JLabel("Hallo!"));¶
        frame.setSize(400, 200);¶
        // frame.pack(); für automatische Größenberechnung¶
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);¶
        frame.setVisible(true);¶
    }¶
¶
    public static void main(String[] args) {¶
        new SwingApp();¶
    }¶
}¶
```

SWING KOMPONENTEN

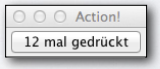
- JFrame
- JComponent
- JPanel
- JScrollPane
- JButton
- JLabel
- JTabbedPane
- JCheckbox
- JComboBox
- JTextField, JTextArea
- JPasswordField
- JSlider, JMenuBar, ...

Vergleiche: http://docs.oracle.com/javase/tutorial/ui/features/components.html

SWING
EVENTS & LISTENER

- Package java.awt.event bzw. java.swing.event
- java.awt.ActionListener ist die Mutter aller EventListener
- Meist anonyme Klasse, die Interface implementiert

```
public class DrueckMich {

    private int count = 0;
    private final String GEDRUECKED = " mal gedrückt";

    public DrueckMich() {
        JFrame derFrame = new JFrame("Action!");
        final JButton derButton = new JButton(count + GEDRUECKED);
        derButton.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                count++;
                derButton.setText(count + GEDRUECKED);
            }
        });
        derFrame.getContentPane().add(derButton);
        derFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        derFrame.pack();
        derFrame.setVisible(true);
    }

    public static void main(String[] argv) {
        new DrueckMich();
    }
}
```



2. - Der ActionListener reagiert auf alle Arten von Events.
   - Meistens macht es Sinn nur auf bestimmte Events zu reagieren, dann nimmt man einen spezielleren Listener, z.b. MouseListener.

0. - Frage: Warum ist „derButton“ als final deklariert?

SWING
LISTENER UND ADAPTER

- Für jeden Typ von Listener existiert ein Interface, das den Listener definiert. Namensschema: <Typ>Listener
- Dazu gibt es jeweils eine passende Klasse, die die Event Informationen hält. Namensschema: <Typ>Event
- Auf Komponenten existiert Methode add<Typ>Listener(..)
- Meist existiert eine Adapterklasse, Schema: <Typ>Adapter

4.
- Falls mehr als eine Methode im Listener Interface definiert ist, dann existiert meistens zusätzlich eine abstrakte Adapterklasse, die das Interface mit leeren Methoden implementiert.
- Warum könnte das sinnvoll sein?
- Beispiel: MouseListener definiert 5 Methoden, aber vielleicht interessiert mich nur mouseClicked?
- Welchen Nachteil hat die Verwendung von Adapterklassen?
- Man verbaut sich den Weg von einer anderen Klasse zu erben.
- Die Klassen die auf etwas reagieren, sind meistens mehr als „nur“ Listener

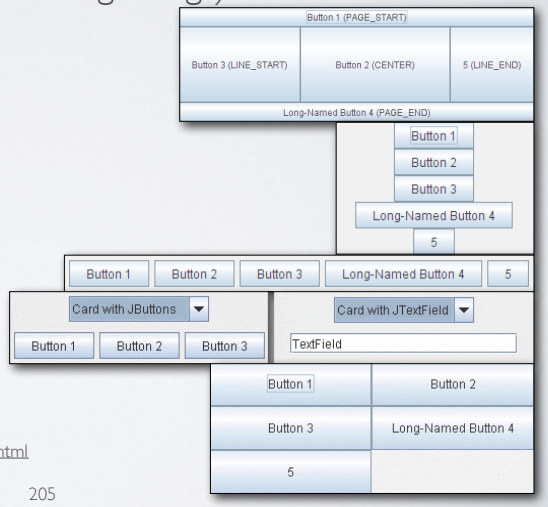
-> anonyme Überschreibung von Adapterklassen ist daher ratsam.

SWING
PROMINENTE EVENTLISTENER

- WindowListener
- ActionListener
- KeyListener
- MouseListener
- MouseMotionListener

SWING LAYOUTMANAGER

- Container#setLayout(LayoutManager mgr)
- BorderLayout
- BoxLayout
- FlowLayout
- CardLayout
- GridLayout

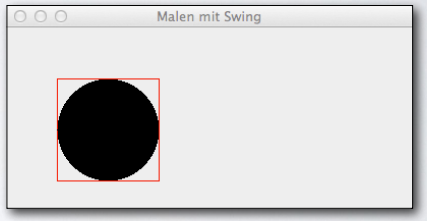


http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

0. - Vergleiche http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
1. - Ein BorderLayout teilt den Container in bis zu 5 Teile (Top, Bottom, Left, Right, Center bzw. N, E, S, W, Center).
   - Aller Platz, der übrig bleibt geht in den Center Bereich.
2. - BoxLayouts setzen alle Komponenten in eine Reihe oder eine Spalte.
   - Elemente können innerhalb der Reihe/Spalte ausgerichtet werden (left, right, center).
3. - FlowLayout stellt alle Elemente in einer Reihe dar.
   - Ein Zeilenumbruch kommt, wenn der Container zu klein wird in der horizontalen.
4. - CardLayout stellt einen Manager dar, der verschiedene Elemente zu verschiedenen Zeitpunkten darstellt.
   - Dabei ist immer nur eine Variante von Components zu einer Zeit sichtbar.
   - Funktionalität wie ein JTabbedPane.
5. - Elemente werden innerhalb einer definierten Anzahl Zeilen und Spalten angeordnet.
   - Elemente haben alle die gleiche Größe.

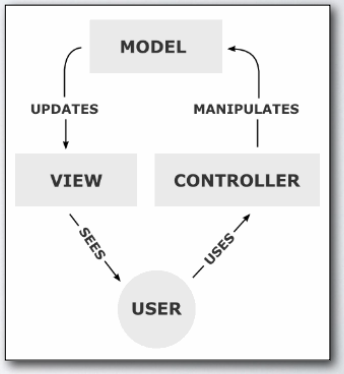
SWING ZEICHNEN

```
public PaintBeispiel() {
    JFrame frame = new JFrame("Malen mit Swing");
    frame.setSize(400, 200);
    JPanel zeichenFläche = new JPanel() {

        // jede Swing JComponent hat eine methode paint(),
        // welche zum Zeichnen überschrieben werden kann.
        @Override
        public void paint(Graphics g) {
            super.paintComponents(g);
            g.setColor(Color.RED);
            g.drawRect(50, 50, 100, 100);
            g.setColor(Color.BLACK);
            g.fillOval(50, 50, 100, 100);
        }
    };
    frame.getContentPane().add(zeichenFläche);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
```



JAVAFX

- Ähnlich zu Swing, aber komfortabler
- Superklasse Node bzw. Control
- MVC
- Sourcen (sind bei Java 8 enthalten)

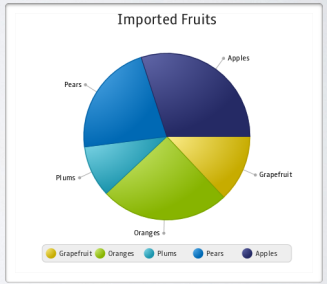


Grafik von http://en.wikipedia.org/wiki/File:MVC-Process.png

1. - Konzept der Layout Manager ist vorhanden, allerdings etwas moderner.
   - EventListener sind analog zu Swing, allerdings auch hier etwas moderner.
   - Eingabeelemente sind ähnlich, aber auch mit modernerer API.
2. - Ähnlich zu Swing, allerdings anderer Name.
3. - Stärkeres MVC Pattern als bei Swing.
   - Es gibt Models, Views und Controller.
   - Model = Datenhaltung.
   - Views = Präsentationsschicht.
   - Controller = Enthalten eigentliche Logik.
   - Wird gleich im Beispiel gezeigt.

Johannes Unterstein - TINF18 - Java - Sommersemester 2019

JAVAFX
GUI ELEMENTE



Grafik von http://docs.oracle.com/javafx/2/ui_controls/overview.htm

Dokumentation von oracle ist sehr gut!

Siehe:
http://docs.oracle.com/javafx/2/ui_controls/overview.htm
http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

JAVAFX SCENEBUILDER

- Grafischer Editor
- .fxml und @FXML
- http://www.oracle.com/technetwork/java/javafxscenebuilder-1x-archive-2199384.html

Gezeigt während der Live-Dome:
Anlegen neues intelliJ JavaFX Projekt
Aufbau GUI
- Library
- Hierarchy
- Scene
- Layout
  - fx:id Attribut

Gui Elemente
  - Containers
  - Layouts
  - Wrap in Layout
- Controls

Modell-Binding
Observables

FXML kann natürlich auch per Texteditor bearbeitet werden
- Wie setzen wir einen Controller
- Wie verwende ich Controls mit fx:id?

WICHTIGE DINGE ÜBER JAVAFX

- .fxml und @FXML
- javafx.fxml.Initializable
- ObservableList und FXCollections
- Inhalt eines Packages
- Dokumentation ist sehr gut

1. - Annotation @FXML macht autobinding von Variable zu konkretem Control in der Oberfläche mittels fx:id.
2. - Interface Initializable kann als Controller einer Oberfläche gesetzt werden.
   - initialize(URL url, ResourceBundle resourceBundle) wird aufgerufen zur Initialisierung.
3. - Es ist von Haus aus möglich CSS zum Styling zu verwenden.
   - Für Designer sehr angenehm, eher der webbasierte Weg.
4. - Inhalt eines Packages sollten in JavaFX sein: „Alles was zu einer View dazugehört.“
       - View Klasse
       - Controller Klasse
       - CSS Datei
       - Helfer Klassen
   - Modell ist separat, da es meistens View-übergreifend ist.
5. - http://docs.oracle.com/javafx/2/ui_controls/overview.htm
   - http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

ZU INSTALLIEREN

- Aktuelles Oracle JDK
- Intelllj oder efxclipse.org -> Vollständige Eclipse-Distribution
- SceneBuilder

http://www.eclipse.org/efxclipse/index.html
http://www.oracle.com/technetwork/java/javafxscenebuilder-1x-archive-2199384.html

ÜBUNGSAUFGABE (30 MIN)

- Implementiere:
  - Implementiere kleine Oberfläche mit JavaFX

HAPPY HACKING