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

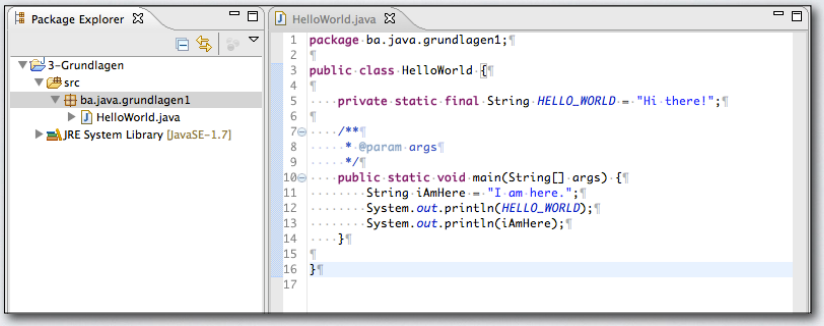
AGENDA

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

ALLGEMEINES

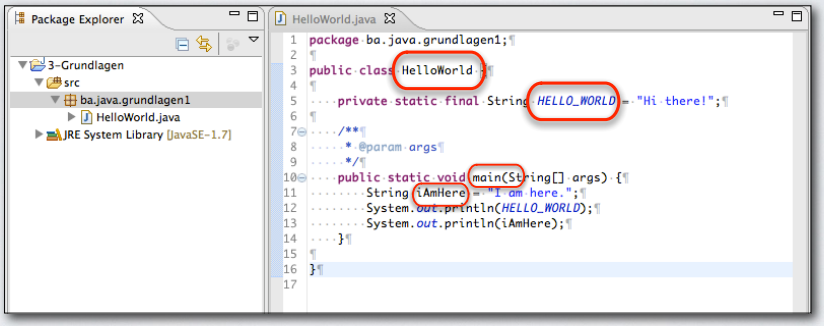
- Zeichensatz Unicode
  - Jedes Zeichen mit 16 Bit (2 Byte) dargestellt
- Sprachgrammatik an C/C++ angelehnt
  - Datentypen, Ausdrücke und Anweisungen sehr ähnlich, meist sogar identisch

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

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)

REFERENZTYPEN

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

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

50

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!

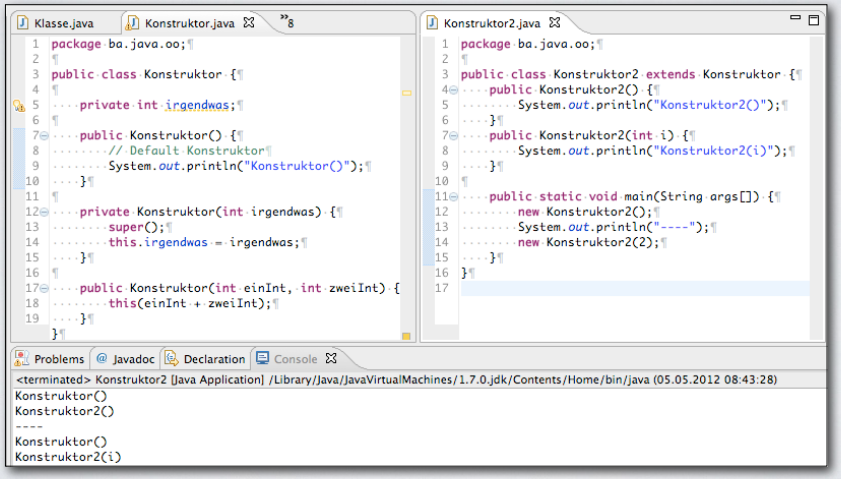
KONSTRUKTOREN

- 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

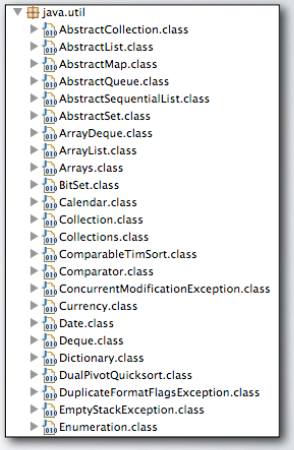
- Keine Destruktoren wie in C++
- Aufräumen beim Löschen des Objektes trotzdem möglich
  - protected void finalize() { ... }
  - Aufruf ist nicht garantiert, eher nicht verwenden!

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)_/¯
   - “eqals” 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();
```


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 ᕦ(̈◕)ᕤ
- 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.

SINN VON INTERFACES

EIN AUTO IST KEINE GRÖßE! :(

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.

SINN VON INTERFACES

EIN AUTO HAT EINE GRÖßE!

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:


```java
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 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