Java Intro

Michael Inden

Freiberuflicher Consultant und Trainer

https://github.com/Michaeli71/JAVA_INTRO

Speaker Intro

- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg

- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch
Blog: https://jaxenter.de/author/minden

Kurse: Bitte sprecht mich an!

https://github.com/Michaeli71/JAVA_INTRO

Agenda

Workshop Contents

- **PART 1: Schnelleinstieg Java**
  - Erste Schritte in der JShell
  - Schnelleinstieg
    - Variablen
    - Operatoren
    - Fallunterscheidungen
    - Schleifen
    - Methoden
    - Rekursion

https://github.com/Michaeli71/JAVA_INTRO

Workshop Contents

- **PART 2: Strings**
  - Gebräuchliche String-Aktionen
  - Suchen und Ersetzen
  - Formatierte Ausgaben
  - Mehrzeilige Strings
  - Einstieg Reguläre Ausdrücke

- **PART 3: Arrays**
  - Gebräuchliche Array-Aktionen
  - Mehrdimensionale Arrays
  - Beispiel: Flood Fill

Workshop Contents

- **PART 4: Klassen & Objektorientierung**
  - Basics
  - Textuelle Ausgaben
  - Gleichheit == / `equals()`
  - Klassen ausführbar machen
  - Imports & Packages
  - Information Hiding
  - Vererbung und Overloading und Overriding
  - Die Basisklasse `Object`
  - Interfaces & Implementierungen

Workshop Contents

- **PART 5: Collections**
  - Schnelleinstieg Listen, Sets und Maps
  - Iteratoren
  - Generics
  - Basisinterfaces für Container
  - Praxisbeispiel Stack und Queue selbst gebaut
  - Sortierung – `sort()` + Comparator

Workshop Contents

- **PART 6: Ergänzendes Wissen**
  - Sichtbarkeits- und Gültigkeitsbereiche
  - Primitive Typen und Wrapper
  - Enums
  - `?`-Operator
  - `switch`
  - Besonderheiten in Schleifen `break` und `continue`
  - Vererbung und Polymorphie
  - Varianten innerer Klassen

Workshop Contents

- **PART 7: Exception-Handling**
  - Schnelleinstieg
  - Exceptions selbst auslösen
  - Eigene Exception-Typen definieren
  - Propagation von Exceptions
  - Automatic Resource Management
  - Checked / Unchecked Exceptions

- **PART 8: Dateiverarbeitung**
  - Verzeichnisse und Dateien verwalten
  - Daten schreiben / lesen
  - CSV-Dateien einlesen

Workshop Contents

- **PART 9: Einstieg in Lambdas und Streams**
  - Syntax von Lambdas
  - Lambdas im Einsatz mit `filter()`, `map()` und `reduce()`
  - Lambdas im Einsatz mit `Collectors.groupingBy()`
  - `takeWhile()` / `dropWhile()`

- **PART 10: Datumsverarbeitung**
  - Einführung Datumsverarbeitung
  - Zeitpunkte und die Klasse `LocalDateTime`
  - Datumswerte und die Klasse `LocalDate`
  - Zeit und die Klasse `LocalTime`

PART 4:
Klassen & Objektorientierung

Immer wieder hört man, Java ist eine objekt-orientierte Sprache. Doch was bedeutet das?

Klassen und Objektorientierung

- Grundlegende Idee: Zustand (Daten) und Verhalten (Funktionen auf den Daten) miteinander verbinden
- Dazu dienen Klassen und Objekte

- Vorüberlegung: Wie könnte man zusammengehörende Informationen bündeln?

```
personFirstname = "Michael";
personLastname = "Inden";
personBirthday = LocalDate.of(1971, 2, 7);

carBrand = "Renault";
carColor = "PETROL";
carHorsePower = 170;
```

- Für wenige Daten noch okay, aber für umfangreiche?!

Grundlagen

- objektorientierten Softwareentwicklung darin, den Programmablauf als ein Zusammenspiel von Objekten und ihren Interaktionen aufzufassen.
- Anlehnung an die reale Welt. Dort sind Objekte, etwa Personen, Autos usw., und ihre Interaktionen ein wesentlicher Bestandteil.
- Dinge werden durch Merkmale und Verhaltensweisen charakterisiert.

- Betrachten wir dies am Beispiel: Im realen Leben ist ein Auto ein Objekt. Ein solches Auto hat Eigenschaften (Attribute), wie z. B. Gewicht und Farbe, und Verhaltensweisen (Methoden), wie z. B. Fahren und Bremsen.

- Eine Klasse kann man sich wie einen Bauplan oder eine Konstruktionsbeschreibung für die Erstellung von Objekten vorstellen.

Grundlagen

- Sprechen wir beispielsweise über ein *spezielles Auto*, etwa das von Hans Mustermann, so reden wir über ein *konkretes Objekt*.
- Sprechen wir dagegen *abstrakter* von Autos, dann meinen wir eine Klasse.
- Eine Klasse ist demnach eine *Strukturbeschreibung* für Objekte und umfasst eine *Menge von Attributen und Methoden*, die in der Regel auf diesen Attributen arbeiten und damit Verhalten definieren.
- Eine Methode »bremsen« kann z. B. das Attribut »Geschwindigkeit« ändern. Schließlich ist ein Objekt eine konkrete Ausprägung (Instanz) einer Klasse.

Klassen definieren

- Mit dem Schlüsselwort class erzeugt man einen Bauplan (eine Klasse) und damit auch einen neuen Typ.
- Standardmäßig startet ein Klassenname in Java mit einem Großbuch-staben.
- Nachfolgend ist dies für Autos vom Typ Car mit drei Attributen, je eins für die Marke, die Farbe und für die Motorleistung, gezeigt.

```
jshell> class Car
   ...> {
   ...>     String brand;
   ...>     String color;
   ...>     int horsePower;
   ...> }
|  created class Car
```

Objekt erzeugen

- Genau wie wir bisher Variablen de- klariert haben, können wir dies für Klassen auch tun, also einen Typ und danach einen Variablennamen angeben. Um nun ein Objekt vom Typ Car zu erstellen, nutzen wir das Schlüsselwort new gefolgt vom Klassennamen.

```
jshell> Car myCar = new Car()
myCar ==> Car@39ed3c8d
```

- Aber Moment: Welche Marke, Farbe und Motorleistung hat das Objekt?
- Und wieso ist die Ausgabe so kryptisch?

Auf Attribute zugreifen

- **Neu erstelltes Objekt**

```
jshell> Car myCar = new Car()
myCar ==> Car@39ed3c8d
```

- **Aber Moment: Welche Marke, Farbe und Motorleistung hat das Objekt?**

```
jshell> myCar.brand
$71 ==> null

jshell> myCar.color
$72 ==> null

jshell> myCar.horsePower
$73 ==> 0
```

- **Offensichtlich sind die Attribute noch nicht mit sinnvollen Werten belegt.**

Werte setzen

- **Werte mit .-Notation und Zuweisung setzen**

```
jshell> myCar.brand = "Audi"
$74 ==> "Audi"

jshell> myCar.color = "BLUE"
$75 ==> "BLUE"

jshell> myCar.horsePower = 220
$76 ==> 220

jshell> myCar
myCar ==> Car@39ed3c8d
```

- **Die Werte sind doch jetzt besetzt, wieso ist die Ausgabe immer noch kryptisch?**

Klassen definieren

- **Für eine String-Ausgabe benötigt es eine Methode `toString()`**

```
jshell> class Car
   ...> {
   ...>     String brand;
   ...>     String color;
   ...>     int horsePower;
   ...>
   ...>     public String toString()
   ...>     {
   ...>         return "Marke: %s / Farbe: %s / PS: %d".
   ...>                formatted(brand, color, horsePower);
   ...>     }
   ...> }
|  replaced class Car
|    update replaced variable myCar, reset to null
```

Klassen definieren

- **Versuchen wir es mal:**

```
jshell> Car myCar = new Car()
myCar ==> Marke: null / Farbe: null / PS: 0

jshell> myCar.brand = "VW"
$80 ==> "VW"

jshell> myCar.color = "YELLOW"
$81 ==> "YELLOW"

jshell> myCar.horsePower = 75
$82 ==> 75

jshell> myCar
myCar ==> Marke: VW / Farbe: YELLOW / PS: 75
```

- **Besser, aber müssen wir wirklich immer alles von Hand belegen?**

Klassen – Wertebelegung beim Erstellen: Konstruktor

```
jshell> class Car
   ...> {
   ...>     String brand;
   ...>     String color;
   ...>     int horsePower;
   ...>
   ...>     public Car()
   ...>     {
   ...>         this.brand = "Unbekannt";
   ...>         this.color = "Unlackiert";
   ...>         this.horsePower = 0;
   ...>     }
   ...>
   ...>     public Car(String brand, String color, int horsePower)
   ...>     {
   ...>         this.brand = brand;
   ...>         this.color = color;
   ...>         this.horsePower = horsePower;
   ...>     }
   ...>
   ...>     public String toString()
   ...>     ...
   ...> }
```

Klassen definieren

- **Versuchen wir es mal:**

```
jshell> Car myFirstCar = new Car()
myFirstCar ==> Marke: Unbekannt / Farbe: Unlackiert / PS: 0

jshell> Car mySecondCar = new Car("Audi", "BLUE", 220)
mySecondCar ==> Marke: Audi / Farbe: BLUE / PS: 220

jshell> Car myThirdCar = new Car("Renault", "PETROL", 120)
myThirdCar ==> Marke: Renault / Farbe: PETROL / PS: 120
```

Klassen – Wertebelegung beim Erstellen: Konstruktor

```
jshell> class Car
   ...> {
   ...>     String brand;
   ...>     String color;
   ...>     int horsePower;
   ...>
   ...>     public Car()
   ...>     {
   ...>         this("Unbekannt", "Unlackiert", 0);
   ...>     }
   ...>
   ...>     public Car(String brand, String color, int horsePower)
   ...>     {
   ...>         this.brand = brand;
   ...>         this.color = color;
   ...>         this.horsePower = horsePower;
   ...>     }
   ...>
   ...>     public String toString()
   ...>     ...
   ...> }
```

Verhalten definieren – Methoden

- **Wäre es nicht schön, sprechende Namen zum Verändern zu nutzen?**

```
...>     public Car()
...>     {
...>         this("Unbekannt", "Unlackiert", 0);
...>     }
...>
...>     public Car(String brand, String color, int horsePower)
...>     {
...>         this.brand = brand;
...>         this.color = color;
...>         this.horsePower = horsePower;
...>     }
...>
...>     public void paintWith(String newColor)
...>     {
...>         this.color = newColor;
...>     }
```

Verhalten definieren – Methoden

- **Wäre es nicht schön, sprechende Namen zum Verändern zu nutzen?**

```
...>     public Car(String brand, String color, int horsePower)
...>     {
...>         this.brand = brand;
...>         this.color = color;
...>         this.horsePower = horsePower;
...>     }
...>
...>     public void paintWith(String newColor)
...>     {
...>         this.color = newColor;
...>     }
...>
...>     public void applyTuningKit()
...>     {
...>         this.horsePower +=150;
...>     }
```

Klassen definieren

- **Versuchen wir es mal:**

```
jshell> Car emptyCar = new Car()
emptyCar ==> Marke: Unbekannt / Farbe: Unlackiert / PS: 0

jshell> emptyCar.paintWith("PURPLE")

jshell> emptyCar.applyTuningKit()

jshell> emptyCar
emptyCar ==> Marke: Unbekannt / Farbe: PURPLE / PS: 150
```

- **Soweit so gut, aber wie ändern wir die Marke und sollten wir das überhaupt nachträglich können?**

Klassen definieren

- **Versuchen wir es mal:**

```
jshell> Car vw = new Car("VW", "RED", 100)
vw ==> Marke: VW / Farbe: RED / PS: 100

jshell> vw.applyTuningKit()

jshell> vw.applyTuningKit()

jshell> vw
vw ==> Marke: VW / Farbe: RED / PS: 400
```

- **Okay, mit «richtigem» Konstruktor ist es besser**
- **Aber sollten wir das gleiche Tuning Kit mehrmals anwenden können?**

Statische Attribute und Methoden

```
class StaticExample
{
    static String staticInfo = "Class wide info";

    static String generateInfo()
    {
        System.out.println("static methods can be called without " +
                           "creating objects");

        return "Special Information";
    }

    String objectMethod()
    {
        System.out.println("object methods must be called on objects");
        System.out.println("static methods/variables are accessible");

        return staticInfo;
    }
}
```

Statische Attribute und Methoden

```
jshell> StaticExample.generateInfo()
static methods can be called without creating objects
$101 ==> "Special Information"

jshell> StaticExample.staticInfo
$102 ==> "Class wide info"

jshell> StaticExample.objectMethod()
|  Error:
|  non-static method objectMethod() cannot be referenced from a static context
|  StaticExample.objectMethod()
|  ^--------------------------^

jshell> new StaticExample().objectMethod()
object methods must be called on objects
static methods/variables are accessible
$103 ==> "Class wide info"
```

Objekte vergleichen

Objekte vergleichen

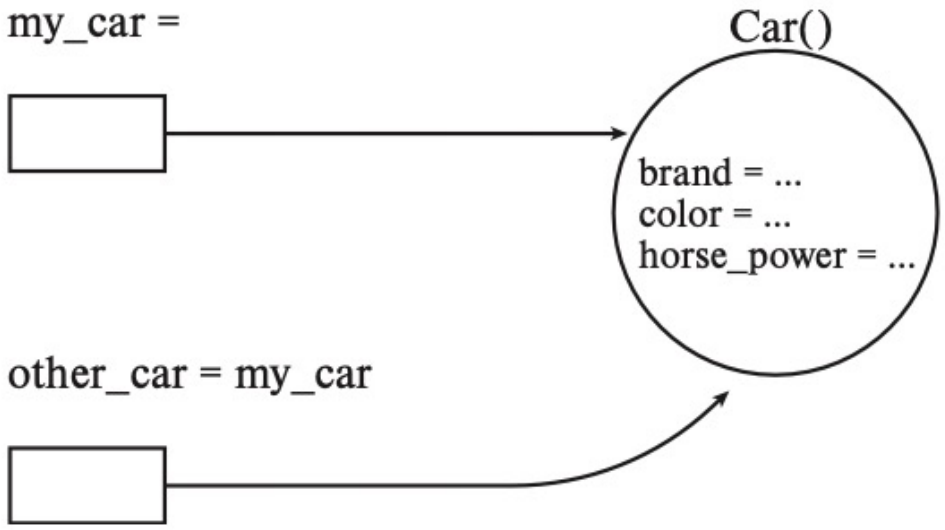
```
Car tomsCar = new Car("Audi", "BLUE", 275)
Car jimsCar = new Car("Audi", "BLUE", 275)

toms_car == jims_car
```

Was ist das Ergebnis?

Objekte vergleichen

```
jshell> Car tomsCar = new Car("Audi", "BLUE", 275)
tomsCar ==> Marke: Audi / Farbe: BLUE / PS: 275

jshell> Car jimsCar = new Car("Audi", "BLUE", 275)
jimsCar ==> Marke: Audi / Farbe: BLUE / PS: 275

jshell> tomsCar == jimsCar
$107 ==> false
```

- **Referenzgleichheit / Identität!**
- **Inhaltliche Gleichheit / semantische Gleichheit => gleiche Werte der Attribute**



Wie prüft man diese beiden Arten der Gleichheit in Java?

Objekte vergleichen

- Operator ‘==' – Mit dem Operator ‘==' werden Referenzen verglichen. Somit wird auf *Identität* geprüft, also, ob es sich um dieselben Objekte handelt.

- ‘`equals()`' – Mit `equals()` werden zwei Objekte bezüglich ihres *Zustands* (d. h. der Wertebelegung der Attribute) verglichen.

```
jshell> tomsCar.equals(jimsCar)
$108 ==> false
```

Wieso geht das denn auch nicht? Was braucht es für semantische Gleichheit in Java?

Objekte vergleichen

- ‘`equals()`' – Mit `equals()` werden zwei Objekte bezüglich ihres *Zustands* (d. h. der Wertebelegung der Attribute) verglichen.
- Zum inhaltlichen Vergleich eigener Klassen ist dort die Methode `equals()` selbst zu implementieren.
  - Standardmäßig erfolgt sonst lediglich ein Vergleich der beiden Referenzen mit dem Operator ‘==’.
  - In eigenen Realisierungen muss der Teil des Objektzustands verglichen werden, der für die semantische Gleichheit, also die inhaltliche Gleichheit, relevant ist.

Implementierung von `equals()`

1. Typprüfung – Um nicht Äpfel mit Birnen zu vergleichen, sichern wir vor dem inhaltlichen Vergleich ab, dass nur Objekte des gewünschten Typs verglichen werden.

2. Objektvergleich – Anschließend werden diejenigen Attributwerte verglichen, die für die Aussage »Gleichheit« relevant sind.

    - Die hierzu notwendigen Attribute des Objekts werden (z. B. in der Reihenfolge ihrer Definition oder des ersten vermuteten Unterschieds) per Operator == auf Gleichheit geprüft.

Implementierung von `equals()`

1. **Typprüfung – Um nicht Äpfel mit Birnen zu vergleichen, sichern wir vor dem inhaltlichen Vergleich ab, dass nur Objekte des gewünschten Typs verglichen werden.**
2. **Objektvergleich – Anschließend werden diejenigen Attributwerte verglichen, die für die Aussage »Gleichheit« relevant sind.**

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit sicherstellen
        return false;

    Car otherCar = (Car) other;
    return Objects.equals(brand, otherCar.brand) &&
           Objects.equals(color, otherCar.color) &&
           horsePower == otherCar.horsePower;
}
```

Klassen ausführbar machen

Klassen ausführbar machen

- Funktionalität objektorientierter Programme ergibt sich aus dem Zusammenspiel verschiedener Klassen.

- Demzufolge muss es zumindest einen Startpunkt, die Hauptapplikation, geben.

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

- public schon mehrmals gesehen => Sichtbarkeiten
- public macht Klasse / Funktionalität für andere Klassen sichtbar

Macht es Sinn die Klasse Car ausführbar zu machen?

Klassen ausführbar machen

- **NEIN!**
- **Die Klasse Car modelliert ja nur Daten.**
- **Besser ist es, eine Hauptapplikation als Klasse zu definieren CarManagementApp**

```
public class CarManagementApplication
{
    Car[] availableCars = new Car[] {  new Car("Renault", "BLUE", 75),
                                       new Car("Renault", "PETROL", 175),
                                       new Car("Ferrari", "RED", 455),
                                       new Car("BMW", "GREEN", 255),
                                       new Car("BMW", "YELLOW", 125),
                                       new Car("VW", "WHITE", 65),
                                       new Car("VW", "BLUE", 105) };

    private CarManagementApplication()
    {
    }
```

Klassen ausführbar machen

- **Besser ist es, eine Hauptapplikation als Klasse zu definieren CarManagementApp**

```
private void filterByBrand(String brand)
{
    for (Car currentCar : availableCars)
    {
        if (currentCar.brand.equals(brand))
            System.out.println(currentCar);
    }
}

private void filterByHorsePowerGreaterThan(int minHorsePower)
{
    for (int i = 0; i < availableCars.length; i++)
    {
        if (availableCars[i].horsePower > minHorsePower)
            System.out.println(availableCars[i]);
    }
}
```

Klassen ausführbar machen

- **Besser ist es, eine Hauptapplikation als Klasse zu definieren CarManagementApp**

```
public static void main(String[] args)
{
    CarManagementApplication app = new CarManagementApplication();

    System.out.println("Alle Renaults im Angebot:");
    app.filterByBrand("Renault");


    System.out.println();

    System.out.println("Alle Autos mit mehr als 150 PS:");
    app.filterByHorsePowerGreaterThan(150);
}
```

Klassen ausführbar machen

- **Programmausführung:**

```
Alle Renaults im Angebot:
Marke: Renault / Farbe: BLUE / PS: 75
Marke: Renault / Farbe: PETROL / PS: 175


Alle Autos mit mehr als 150 PS:
Marke: Renault / Farbe: PETROL / PS: 175
Marke: Ferrari / Farbe: RED / PS: 455
Marke: BMW / Farbe: GREEN / PS: 255
```

Imports und Packages

Imports und Packages

- Funktionalität objektorientierter Programme ergibt sich aus dem Zusammenspiel verschiedener Klassen.
- Die Trennung von Datencontainern und Applikation haben wir schon erfolgreich gesehen
- Wir wollen nun die Applikation noch ein wenig besser strukturieren
- Dabei helfen Packages <=> Verzeichnisse im Filesystem

Imports und Packages im Standard

- Java selbst ist in diverse Packages aufgeteilt
- Wir haben u.a. schon java.util gesehen und dort die Klasse Arrays als Utility-Klasse

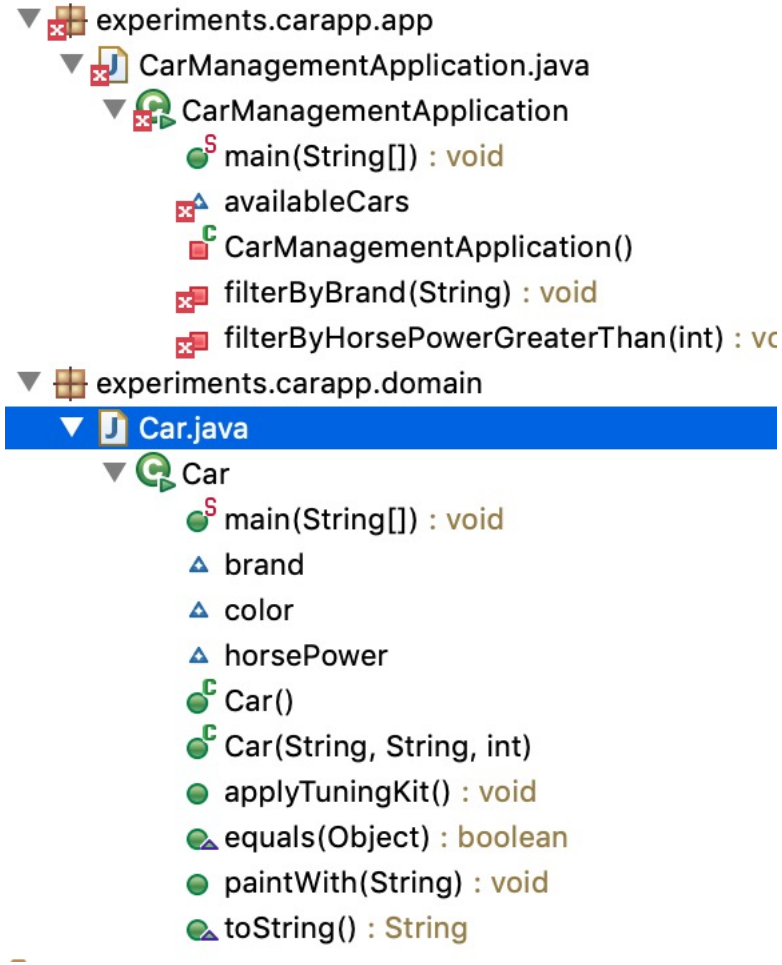
- Es gibt noch viel mehr, etwa Datumsverarbeitung

```
jshell> Year.now()
|  Error:
|  cannot find symbol
|    symbol:   variable Year
|  Year.now()
|  ^--^

jshell> import java.time.Year

jshell> Year.now()
$111 ==> 2021
```

- Funktionalität muss explizit mit import eingebunden werden!

Imports und Packages

- Anlegen von Packages und Befüllung mit den beiden Klassen



Es ging doch zuvor alles, was ist denn jetzt los?

Imports und Packages

- Die Klasse Car ist nun in einem anderen Package => import benötigt

```
public class CarManagementApplication
{
    Car[] availableCars = new Car[] {  new Car("Renault", "BLUE", 75),
                                       new Car("Renault", "PETROL", 175),
                                       new Car("Ferrari", "RED", 455),
                                       new Car("BMW", "GREEN", 255),
                                       new Car("BMW", "YELLOW", 125),
                                       new Car("VW", "WHITE", 65),
                                       new Car("VW", "BLUE", 105) };
```

Imports und Packages

- Die Attribute sind alle ohne Sichtbarkeit definiert => nur im Package selbst sichtbar

```
19      private void filterByBrand(String brand)
20      {
21          for (Car currentCar : availableCars)
22          {
23              if (currentCar.brand.equals(brand))
24                  System.out.println(currentCar);
25          }
26      }
27
28      private void filterByHorsePowerGreaterThan(int minHorsePower)
29      {
30          for (int i = 0; i < availableCars.length; i++)
31          {
32              if (availableCars[i].horsePower > minHorsePower)
33                  System.out.println(availableCars[i]);
34          }
35      }
```

Imports und Packages

- Die Attribute sind alle ohne Sichtbarkeit definiert => nur im Package selbst sichtbar
- Abhilfen:
  1. Sichtbarkeit auf public ändern
  2. Zugriffsmethoden bereitstellen => Es wird Zeit für Datenkapselung

```
19      private void filterByBrand(String brand)
20      {
21          for (Car currentCar : availableCars)
22          {
23              if (currentCar.brand.equals(brand))
24                  System.out.println(currentCar);
25          }
26      }
27
28      private void filterByHorsePowerGreaterThan(int minHorsePower)
29      {
30          for (int i = 0; i < availableCars.length; i++)
31          {
32              if (availableCars[i].horsePower > minHorsePower)
33                  System.out.println(availableCars[i]);
34          }
35      }
```

Imports und Packages

- Zugriffsmethoden bereitstellen => Es wird Zeit für Datenkapselung
- Eclipse bietet Automatik

```
19    private void filterByBrand(String brand)
20    {
21        for (int i = 0; i < availableCars.length; i++)
22        {
23            if (availableCars[i].brand.equals(brand))
24                System.out.printl
25        }
26    }
27
28    private void filterByHorsePow
29    {
30        for (int i = 0; i < avail
31        {
32            if (availableCars[i].horsePower > minHorsePower)
33                System.out.println(availableCars[i]);
34        }
35    }
```

The field Car.brand is not visible

2 quick fixes available:

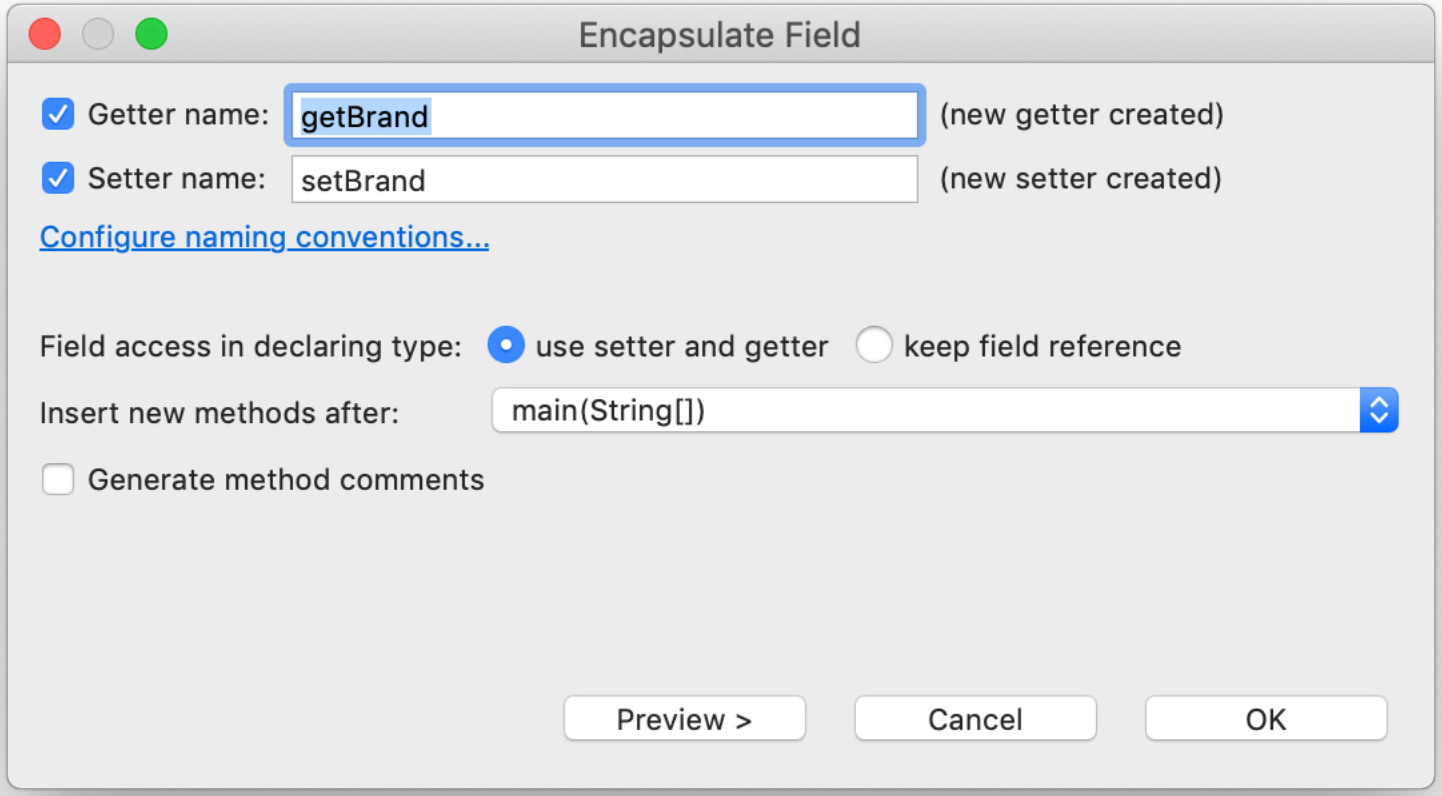
Change visibility of 'brand' to 'public'

Create getter and setter for 'brand'...

Press 'F2' for focus

Imports und Packages

- Zugriffsmethoden bereitstellen => Es wird Zeit für Datenkapselung
- Eclipse bietet Automatik



Imports und Packages

- **Ergebnis der Automatik**

```
private void filterByBrand(String brand)
{
    for (Car currentCar : availableCars)
    {
        if (currentCar.getBrand().equals(brand))
            System.out.println(currentCar);
    }
}

private void filterByHorsePowerGreaterThan(int minHorsePower)
{
    for (int i = 0; i < availableCars.length; i++)
    {
        if (availableCars[i].getHorsePower() > minHorsePower)
            System.out.println(availableCars[i]);
    }
}
```

Imports und Packages

- **Ergebnis der Automatik**

```
public class Car
{
    private String brand;
    String color;
    private int horsePower;

...
```

```
public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

public int getHorsePower() {
    return horsePower;
}

public void setHorsePower(int horsePower) {
    this.horsePower = horsePower;
}
```

Das sieht ja schon fast wie Information Hiding aus?

Imports und Packages

- **Ergebnis der Automatik**

```
public class Car
{
    private String brand;
    String color;
    private int horsePower;

...
```

```
public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

public int getHorsePower() {
    return horsePower;
}

public void setHorsePower(int horsePower) {
    this.horsePower = horsePower;
}
```

Sichtbarkeiten und Information Hiding

Information Hiding – Verstecken von Informationen

- Gerade haben wir Zugriffsmethoden eingeführt, insbesondere um die Kompilierfehler zu entfernen.
- Das führte ganz nebenbei zum Verstecken der Attribute, was man Kapselung bzw. Information Hiding nennt.
- Beim Verstecken von Informationen möchte man erreichen, dass gewisse Informationen nicht nach außen dringen. Wieso ist dies hilfreich?

Information Hiding – Verstecken von Informationen

- Das erkennen wir, wenn wir über folgende Punkte bei Modellierungen von Informationen mithilfe von Attributen in Klassen nachdenken:

- Geheime Informationen – Stellen wir uns Passwörter vor: Diese möchte man sicher nicht einfach öffentlich zugänglich machen.

- Wertebereiche – Zudem gelten für gewisse Attribute bestimmte gültige Wertebereiche. Obwohl die Variablen vielleicht vom Typ `int` oder `long` sind, ist nur ein kleiner Bruchteil der Werte auch tatsächlich gültig. Eine untere Grenze ist oftmals der Wert 0, etwa für PS oder Rabatte. Vielfach gibt es auch Obergrenzen.

Wertebereichsprüfungen

Die Kapselung ermöglicht es uns, innerhalb von Methoden verschiedene Prüfungen zu integrieren und ungültige Werte zurückzuweisen.

```
public void setHorsePower(int horsePower)
{
    if (horsePower <= 0 || horsePower > 2_000)
        throw new IllegalArgumentException("INVALID PS: not in range 1 - 2000");

    this.horsePower = horsePower;
}
```

Information Hiding – Verstecken von Informationen

- Unveränderliche Informationen – Beispielsweise ist der Geburtstag eines Menschen fix. Deshalb sollte sich dieser Wert für eine Klasse `Person` nach der Konstruktion nicht ändern (können).

- Auch die Automarke steht schon zur Produktion fest. Aus einem Audi oder VW wird kein Porsche werden. Etwas Ähnliches haben wir aber in der Einführung schon gemacht, da wurde aus einem VW kurzerhand ein Audi.

- Vielleicht fühlte sich das für Sie dort bereits etwas merkwürdig an, nun wissen Sie, dass Sie damit richtig lagen. Gutes OO-Design vermeidet derartige Überraschungen.

Information Hiding – Sichtbarkeiten

- **Zum Verstecken von Informationen und zum Steuern des Zugriffs benötigen wir eine Festlegung von Sichtbarkeiten.**
- **Dabei haben wir schon die recht intuitiven `public` und `private` genutzt.**
- **Java bietet folgende vier Sichtbarkeiten, die festlegen, ob und wie andere Klassen auf Methoden und Attribute zugreifen dürfen:**
  - `public` – Ist von überall aus zugreifbar.
  - `protected` – Zugriff für alle Klassen im selben Package und für abgeleitete Klassen
  - Package-private oder default (kein Schlüsselwort) – Nur Klassen aus demselben Package haben Zugriff darauf.
  - `private` – Nur die Klasse selbst hat Zugriff.

Information Hiding – Sichtbarkeiten

- Schauen wir uns das Verstecken von Informationen für die Klasse `Car` anhand des Attributs für die Farbe an.
- Diese ist bislang als String modelliert. Stellen wir uns vor, wir würden eine weitere Filterung einbauen wollen, etwa nach der Farbe des Autos.
- Wir könnten direkt auf das Attribut `color` zugreifen. Wie schon angedeutet ist das ungünstig. Deshalb sollten wir Zugriffsmethoden anbieten:

```
public String getColor()
{
    return color;
}

public void setColor(String color)
{
    this.color = color;
}
```

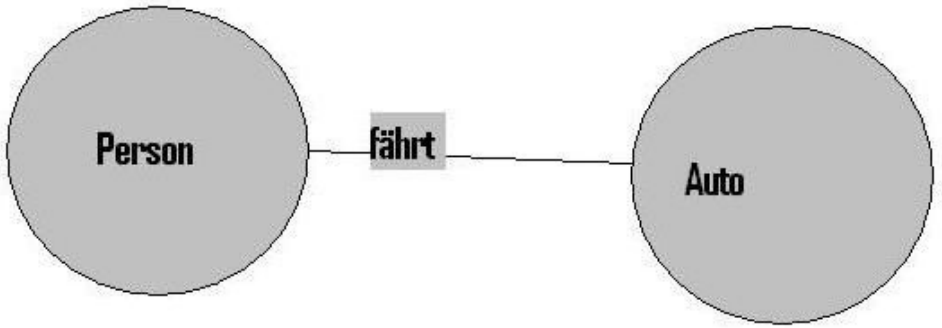
Information Hiding – Verstecken von Informationen

- **Mithilfe von Zugriffsmethoden:**
  - Lassen sich Zugriffe steuern
  - Sichtbarkeiten steuern
  - Wertebereiche prüfen
  - Schreibzugriffe verhindern (keine set()-Methode)
  - Werte zwischenspeichern
  - Aufrufe protokollieren
  - ...
- **Somit sollten bei Verwendung einer Klasse von außerhalb immer Zugriffsmethoden angeboten werden.**

Recap

OO-Grundlagen

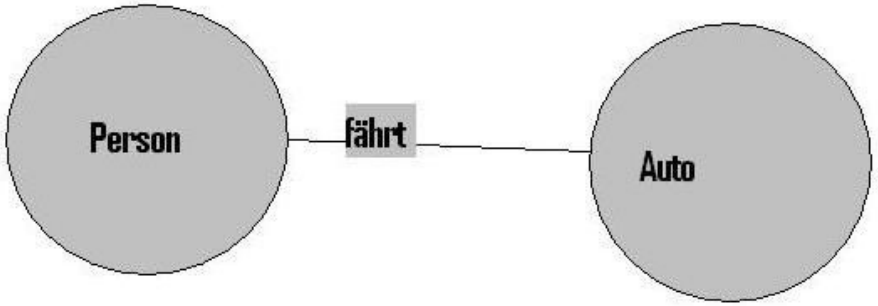
- **Ziel**
  - Zustand (Daten) mit Verhalten (Funktionen auf diesen Daten) kombinieren
  - Denken in Objekten und Zuständigkeiten
  - Programm als ein Zusammenspiel von Objekten



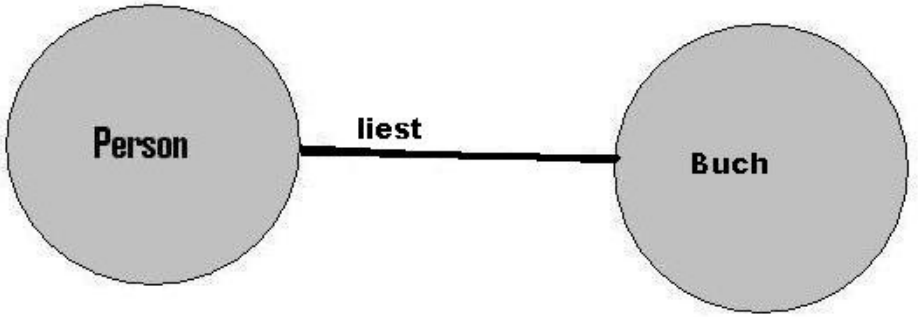
- **Grundgedanken der Objekt-Orientierung**
  - **Datenkapselung** / Information Hiding
  - **Kapselung + Trennung von Zuständigkeiten** / Encapsulation
  - **Wiederverwendbarkeit** / Reuse

- **Assoziation, Aggregation und Komposition**
  - **Assoziation**

    Stehen zwei verschiedene Objekte in Beziehung zu einander, so spricht man ganz allgemein von einer Assoziation.

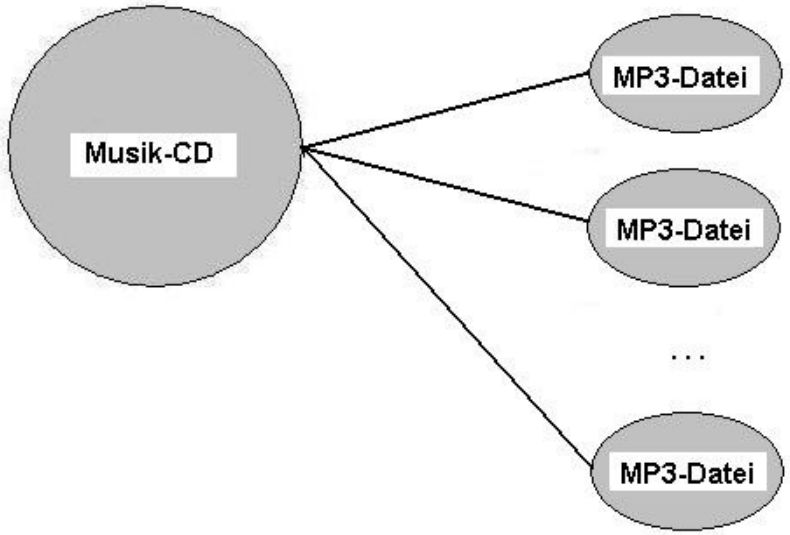


Diese Beziehung kann nach der Zusammenarbeit wieder aufgelöst werden und neue Assoziationen zu anderen Objekten aufgebaut werden.



- **Assoziation, Aggregation und Komposition**
  - **Aggregation**

    Spezialfall einer Assoziation: Sie drückt aus, dass ein Objekt eine Menge von anderen Objekten beinhaltet / referenziert:



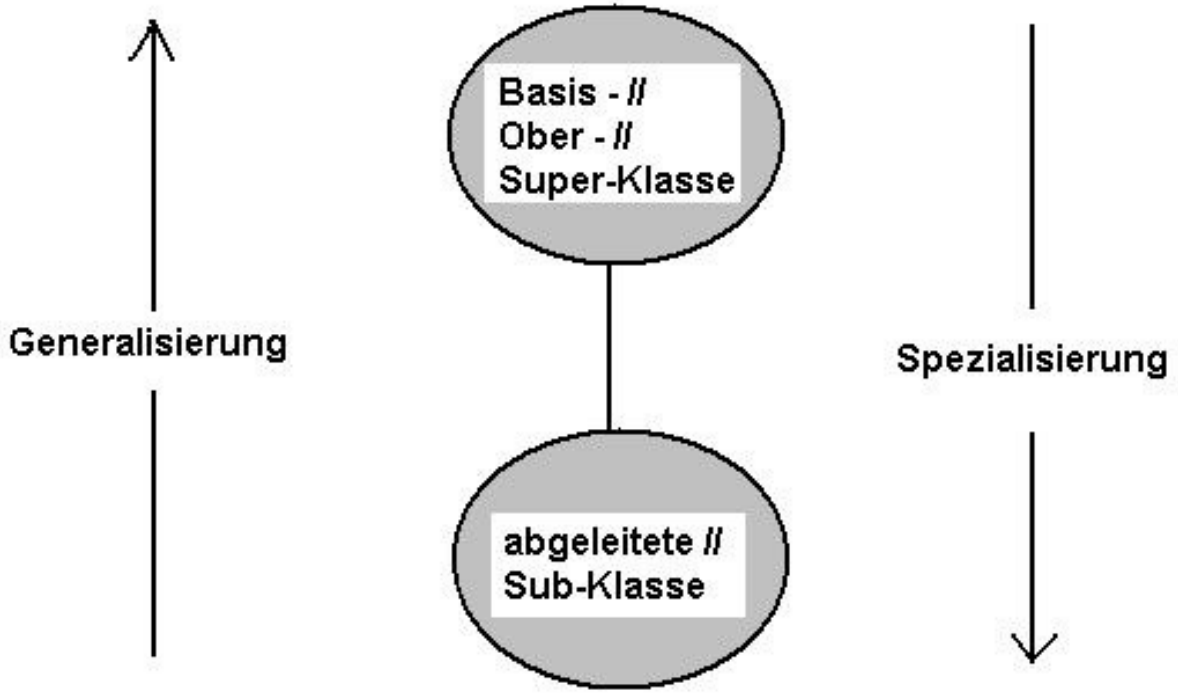
  - **Komposition**

    ,,stärkere“ Ganzes-Teile-Beziehung als Aggregation: Das enthaltene Objekt kann ohne das andere bzw. die anderen nicht

Vererbung

Vererbung

- **Eine spezielle Art, neue Klassen basierend auf bestehenden Klassen zu definieren, nennt sich Vererbung**



- **Die neu entstehende Klasse erweitert oder übernimmt (erbt) durch diesen Vorgang das Verhalten und die Eigenschaften der Basis**

Vererbung

- Vererbungsakt wird durch das Schlüsselwort extends ausgedrückt.
- Eine Subklasse muss lediglich die Unterschiede zu ihrer Basisklasse beschreiben und nicht komplett neu entwickelt werden.

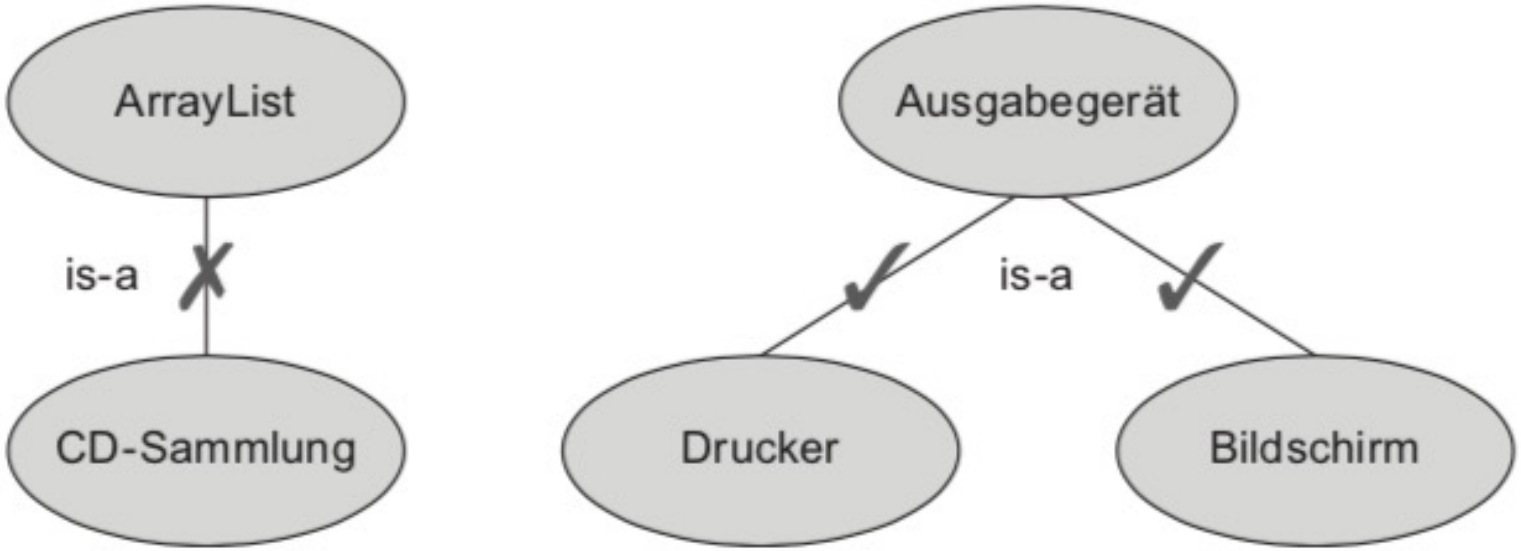
```
class BaseClass
{
    public String method()
    {
    }

    public String otherMethod()
    {
    }
}
```

```
class SubClass extends BaseClass
{
    @Override
    public String method()
    {
        super.method()
        // zusätzliches Verhalten
    }

    public String additionalMethod()
    {
        // zusätzliches Verhalten
    }
}
```

Vererbung – is-a-Beziehung

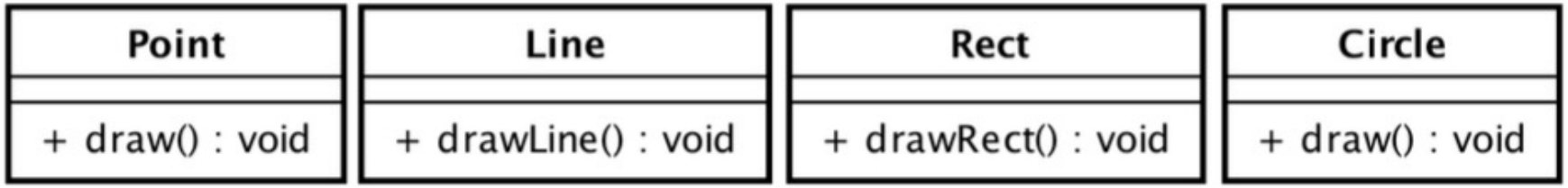
Vererbung ist ein Mittel, um bereits modelliertes Verhalten vorhandener Klassen wiederzuverwenden und zu erweitern und damit die Wartung zu erleichtern.

Vererbung nur einsetzen, wenn „is-a“-Eigenschaft erfüllt ist

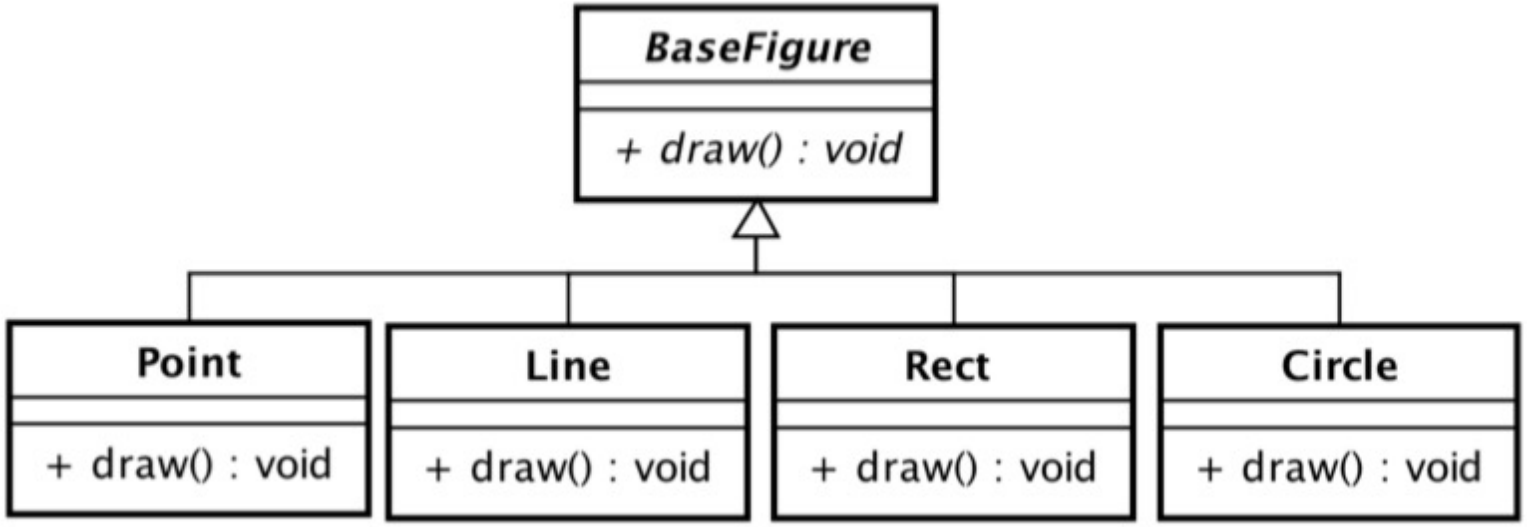


Basisklassen

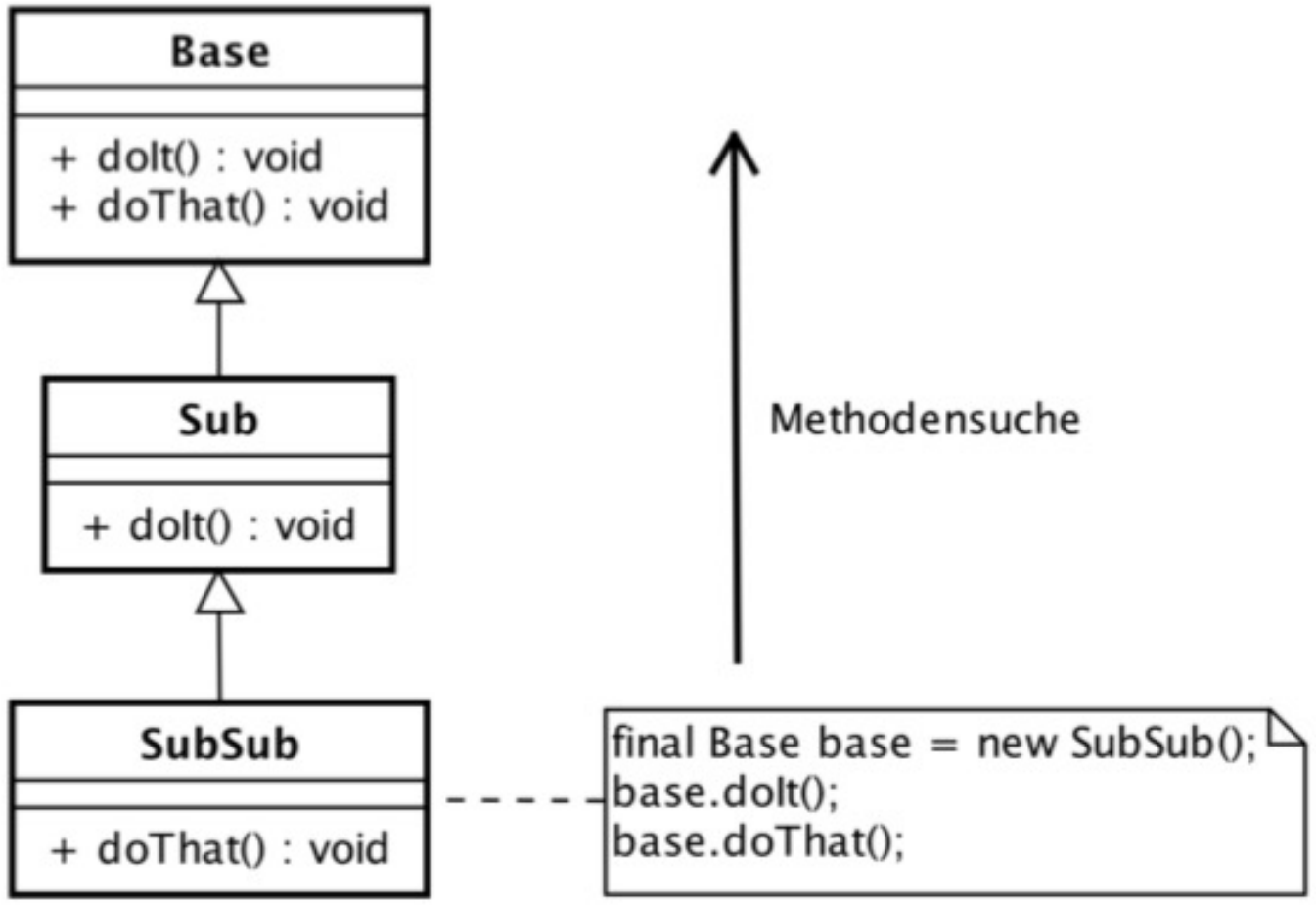
- Schnell entstehen leicht unterschiedliche Namen, etwa neben `draw()` auch die Varianten `drawLine()`, `drawRect()` usw.



- Handhabung der Figuren für nutzende Klassen umständlich und nicht intuitiv



Vererbung – Methodensuche



Vererbung prüfen – instanceof

```
class Base {
}

class Sub extends Base {
}

public class SubSub extends Sub
{
    public static void main(String[] args)
    {
        Base sub = new Sub();
        Base subsub = new SubSub();

        System.out.println(subsub instanceof Sub);
        System.out.println(subsub instanceof Base);
        System.out.println(sub instanceof Base);
    }
}
```

Die Klasse `Object` als die Mutter aller Klassen

- **Die Klasse `java.lang.Object` ist die Basisklasse aller in Java existierenden Klassen.**
- **Jede Referenzvariable besitzt zumindest immer den Typ `Object`. Das ist praktisch, da diese Basisklasse bereits einige Methoden und damit elementare Verhaltensweisen bereitstellt:**

  - `String toString()` – textuelle Repräsentation für ein Objekt zu erzeugen.
  - `boolean equals(Object)` – Vergleich eines Objekts mit einem anderen.
  - `int hashCode()` – Sie bildet den Objektzustand (oder Teile davon) auf eine Zahl ab. Wenn man equals(Object) überschreibt, ist es dringend anzuraten, auch immer hashCode() passend dazu zu überschreiben.
  - `Class<?> getClass()` – Laufzeittyp einer Klasse abfragen. Sie ist wichtig für den Einsatz der fortgeschrittenen Technik Reflection.

Pattern Matching bei `instanceof`

Pattern Matching bei instanceof

- **ALT**

```
final Object obj = new Person("Michael", "Inden");
if (obj instanceof Person)
{
    final Person person = (Person) obj;
    // ... Zugriff auf person...
}
```

Immer diese Casts…
Geht es nicht einfacher?

Pattern Matching bei instanceof

- **ALT**

```
final Object obj = new Person("Michael", "Inden");
if (obj instanceof Person)
{
    final Person person = (Person) obj;
    // ... Zugriff auf person...
}
```

- **NEU**

```
if (obj instanceof Person person)
{
    // Hier kann man auf die Variable person direkt zugreifen
}
```

Pattern Matching bei instanceof

```
Object obj2 = "Hallo Java 14";


if (obj2 instanceof String str)
{
    // Hier kann man str nutzen
    System.out.println("Länge: " + str.length());
}
else
{
    // Hier kein Zugriff auf str
    System.out.println(obj.getClass());
}
```

Pattern Matching bei instanceof

```
if (obj2 instanceof String str2 && str2.length() > 5)
{
    System.out.println("Länge: " + str2.length());
}
```

Overriding vs Overloading

Overriding vs Overloading

- **Overriding (Überschreiben)**
  - **Redefinieren von Verhalten geerbter Methoden (KONTEXT VERERBUNG!=**
  - Spezialisierungen können gewisse Methoden leicht zu modifizieren oder sogar komplett anders zu definieren.
  - Anforderungen an eine überschriebene Methode:
    - Der Methodenname, Parameterliste, Rückgabetyp muss übereinstimmen.
    - Sie darf keine weiteren Exceptions werfen.

Overriding vs Overloading

- **Overloading (Überladen)**
  - Ausgangslage
    - `drawByStartAndEndPos(int x1, int y1, int x2, int y2);`
    - `drawByStartPosAndSize(Point pos, Dimension size);`
  - Wunsch: Definieren von Methoden mit gleichem Namen aber unterschiedlicher Parameterliste => **Handhabung der Klasse vereinfacht sich**
  - Durch das Overloading kann man die „Befehle”, die eine Klasse anbietet bzgl. ihres Namens vereinheitlichen. => zwei `draw()`-Methoden anbieten.
    - `draw(int x1, int y1, int x2, int y2);`
    - `draw(Point pos, Dimension size);`

Overriding vs Overloading

- **Grenzen des Overloadings**
  - **Allerdings sind dem Overloading Grenzen gesetzt.** Die Forderung nach einer unterschiedlichen Parameterliste macht es unmöglich, folgende Methoden zu „überladen”.
    - `drawByStartAndEndPos(int x1, int y1, int x2, int y2);`
    - `drawByStartPosAndSize(int x1, int y1, int w, int h);`
  - **Folge wäre eine *gleiche* Signatur und damit *keine* Unterscheidbarkeit**
    - `draw(int x1, int y1, int x2, int y2);`
    - `draw(int x1, int y1, int w, int h);`

Schnittstellen und Implementierung

Schnittstellen

- Wenn Aufrufe an Methoden anderer Klassen erfolgen, dient dazu eine sogenannte Use-Beziehung: Eine Klasse verwendet eine andere.
- Dabei gibt es einen Aufrufer und einen Bereitsteller von Funktionalität. Man spricht zum Teil auch von Client und Server.
- Häufig werden die Begriffe Schnittstelle und Interface analog zueinander genutzt. Mit beidem sind diejenigen Methoden einer Klasse gemeint, die von anderen Klassen aufgerufen werden können.
- Schnittstellen definieren ein »Angebot von Verhalten«. Im einfachsten Fall ergibt sich die Schnittstelle, die ein Client von einem Server nutzen kann, implizit über die öffentlichen Methoden.
- Manchmal ist es jedoch zweckmäßiger, Methoden zu gruppieren und diesen Funktionsblöcken einen eigenen Namen zu geben. Dies ist durch die Technik der Interfaces möglich.

Schnittstellen

- Java kennt dazu das Schlüsselwort `interface` zur Definition der Schnittstelle einer Klasse in Form einer Menge von Methoden ohne Implementierung.
- Man spricht hier von abstrakten Methoden. Zu deren Kennzeichnung dient das Schlüsselwort `abstract`.

```
interface IGraphicFigure
{
    public abstract void resize();
    public abstract void draw();
}
```

Realisierung / Implementierung

- Das Implementieren eines Interface durch eine Klasse wird Realisierung genannt und durch das Schlüsselwort `implements` ausgedrückt.
- Eine Realisierung eines Interface durch eine Klasse bedeutet, dass sich ein Objekt dieser Klasse so verhalten kann, wie es das Interface vorgibt.

```
class Circle implements IGraphicFigure
{
    @Override
    public void resize()
    {
    }

    @Override
    public void draw()
    {
    }
}
```

Exercises Part 4

https://github.com/Michaeli71/JAVA_INTRO

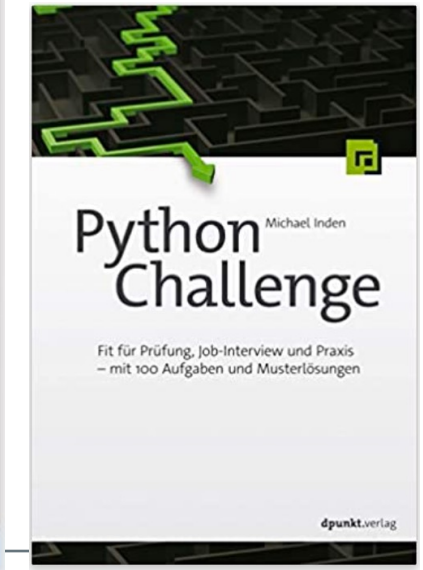
Thank You


HELP
IS ON THE
WAY


Michael Inden
Einfach Java
Gleich richtig programmieren lernen
dpunkt.verlag


5. Auflage
Michael Inden
Der Weg zum Java-Profi
Konzepte und Techniken für die professionelle Java-Entwicklung
dpunkt.verlag


Michael Inden
Java Challenge
Fit für das Job-Interview und die Praxis – mit mehr als 100 Aufgaben und Musterlösungen
dpunkt.verlag


Michael Inden
Python Challenge
Fit für Prüfung, Job-Interview und Praxis – mit 100 Aufgaben und Musterlösungen
dpunkt.verlag