

# Grundlagen der Programmierung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Vorlesungsskript zum Sommersemester 2020

9. Vorlesung (22. Juni 2020)



**Dr. Jin Gerlach**  
**Christian Olt**

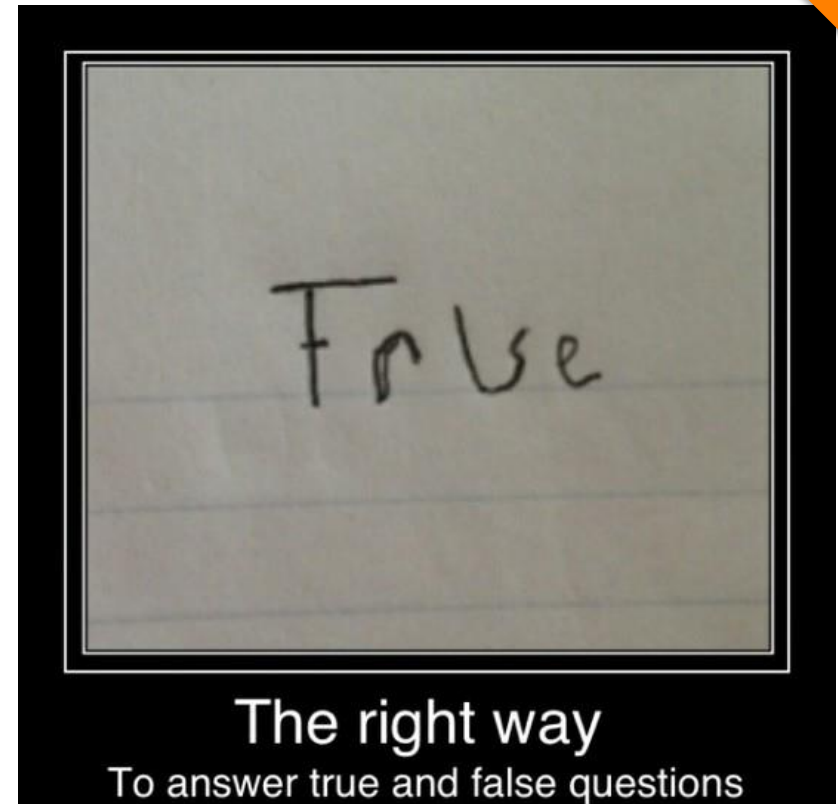
Fachgebiet Wirtschaftsinformatik | Software & Digital Business  
Fachbereich Rechts- und Wirtschaftswissenschaften  
Technische Universität Darmstadt

# True oder false?



<http://pingo.upb.de>  
#9456

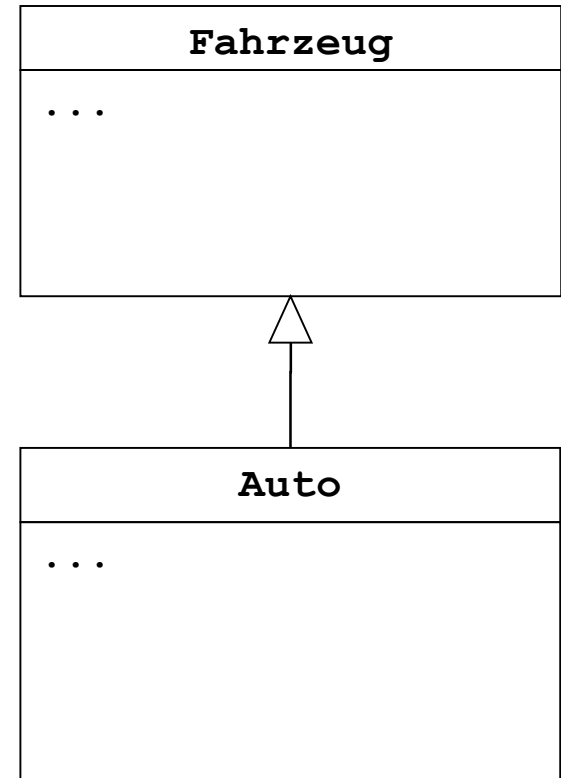
1. Eine Variable vom Typ der Ober-Klasse kann Objekte vom Typ einer Sub-Klasse aufnehmen.
2. Überschriebene und überschreibende Methode besitzen immer die gleiche Signatur.
3. Der statische Typ einer Variablen bestimmt die aufrufbaren Methoden.
4. Der dynamische Typ einer Variablen bestimmt die aufrufbaren Methoden.





# Polymorphismus aka „Dynamisches Binden“

- **Sub-Klasse ist eine Spezialisierung der Ober-Klasse**
- **Polymorphismus** (griechisch für Vielgestaltigkeit) bedeutet: Eine Variable vom **Datentyp einer Ober-Klasse** kann Objekte
  - vom **eigenen** Datentyp (z. B. Fahrzeug)
  - von **allen Datentypen der Sub-Klassen** (z. B. Auto, LKW) speichern
- **Beispiel:** `Fahrzeug f = new Auto(...);`
- Objekte der Sub-Klasse haben alle Methoden und Attribute der Ober-Klasse
  - Alles, was man mit der Ober-Klasse machen kann, geht auch mit der Sub-Klasse





# Statische vs. dynamische Datentypen

- **Konsequenz aus dem Polymorphismus:**

Das Objekt, das in einer Variable gespeichert wurde, hat nicht immer den Datentyp, der bei der Deklaration der Variablen angegeben wurde

- **Unterscheidung:**

- **Statischer Typ** = Datentyp der Variablen
- **Dynamischer Typ** = Datentyp des Objekts

- **Beispiel:**

**Fahrzeug** f = new Auto(...);

Statischer Typ

Dynamischer Typ



# Überschreiben von Methoden

- **Sub-Klassen können geerbte Methoden neu implementieren**
- Dieser Vorgang wird als **Überschreiben** bezeichnet
  - Hierzu wird in der Sub-Klasse eine Methode mit derselben Signatur implementiert, sie überschreibt die entsprechende Methode aus der Ober-Klasse
  - Beim Aufruf einer überschriebenen Methode auf (Objekten) der Sub-Klasse wird die neue Implementierung der überschreibenden Methode aufgerufen
  - Die Sub-Klasse kann auf die überschriebenen Methoden der Ober-Klasse über die Referenz `super` zugreifen:  
`super.ueberschriebeneMethode(...);`



# Beispiel: Überschreiben von Methoden

```
public class Fahrzeug{  
  
    public void druckeInfo() {  
        System.out.println(modell);  
        System.out.println(ps);  
    }  
  
    ...  
}
```

```
public class Auto extends Fahrzeug {  
  
    public void druckeInfo() {  
        super.druckeInfo();  
        System.out.println(tueren);  
        System.out.println(sitzplaetze);  
    }  
  
    ...  
}
```

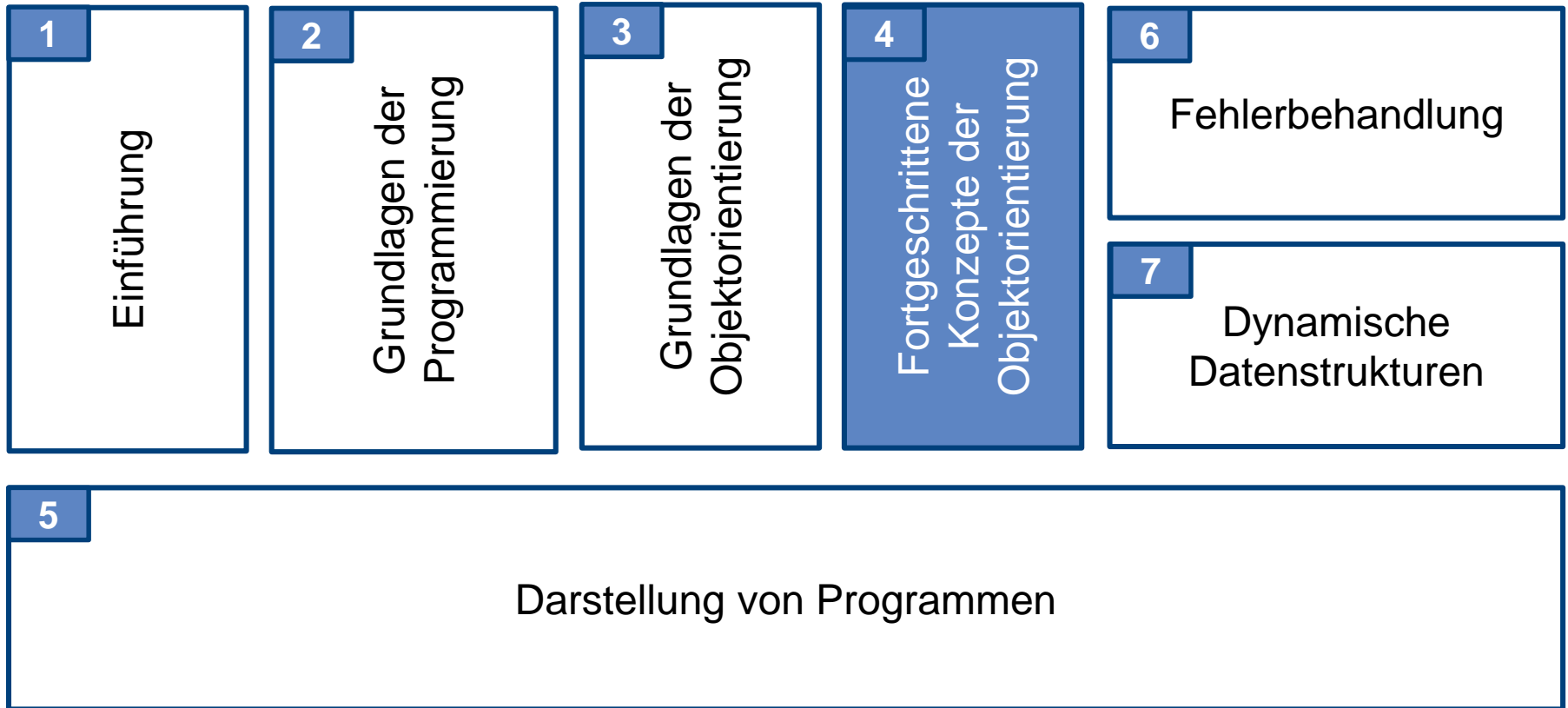
```
public class MyClass {  
    public static void main(String[] args) {  
        Auto auto = new Auto("VW Golf", 50, 5, 5);  
        auto.druckeInfo();  
        Fahrzeug truck = new Fahrzeug("MAN TGS", 300);  
        truck.druckeInfo();  
    } }
```



# Polymorphismus und Methodenaufrufe

- Methodenaufruf bei polymorphen Variablen
  - Der **statische Typ** bestimmt, **welche Methoden** aufgerufen werden können
  - Der **dynamische Typ** bestimmt, **welche Implementierung** der Methode aufgerufen wird
- Die Auswahl der Methode geschieht zur Laufzeit (dynamisch) und unabhängig von der (statischen) Deklaration
  - Falls die Sub-Klasse die **Methode überschreibt**, wird die überschreibende Methode (der Sub-Klasse) aufgerufen
  - Falls die Sub-Klasse die **Methode nicht überschreibt**, wird die geerbte Methode (der Ober-Klasse) aufgerufen

# Thematische Übersicht





# Kapitel 4: Fortgeschrittene Konzepte der Objektorientierung

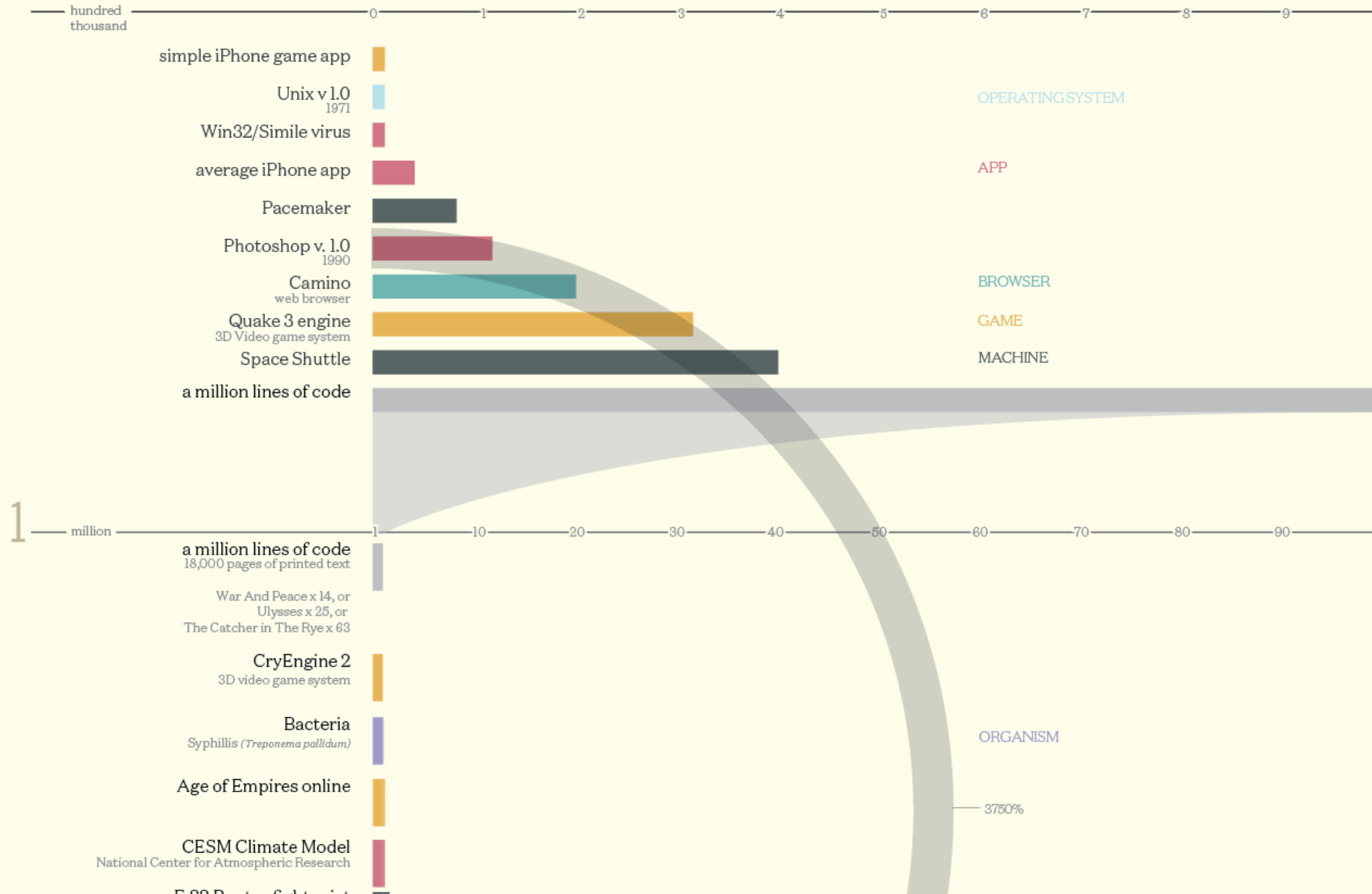
- Pakete als Ordnungsstruktur in Programmen
- Prinzip des „Information Hiding“ und die Sichtbarkeit von Attributen und Methoden in Paket- und Vererbungshierarchien
- Referenzen als Unterschied zwischen Variablen für einfache und strukturierte Datentypen

## Lernziele:

- Pakete als Ordnungsstruktur kennen und anwenden können.
- Das Prinzip des „Information Hiding“ sowie Getter- und Setter-Methoden verstehen.
- Den Unterschied zwischen einfachen Variablen und Referenzvariablen verstehen.

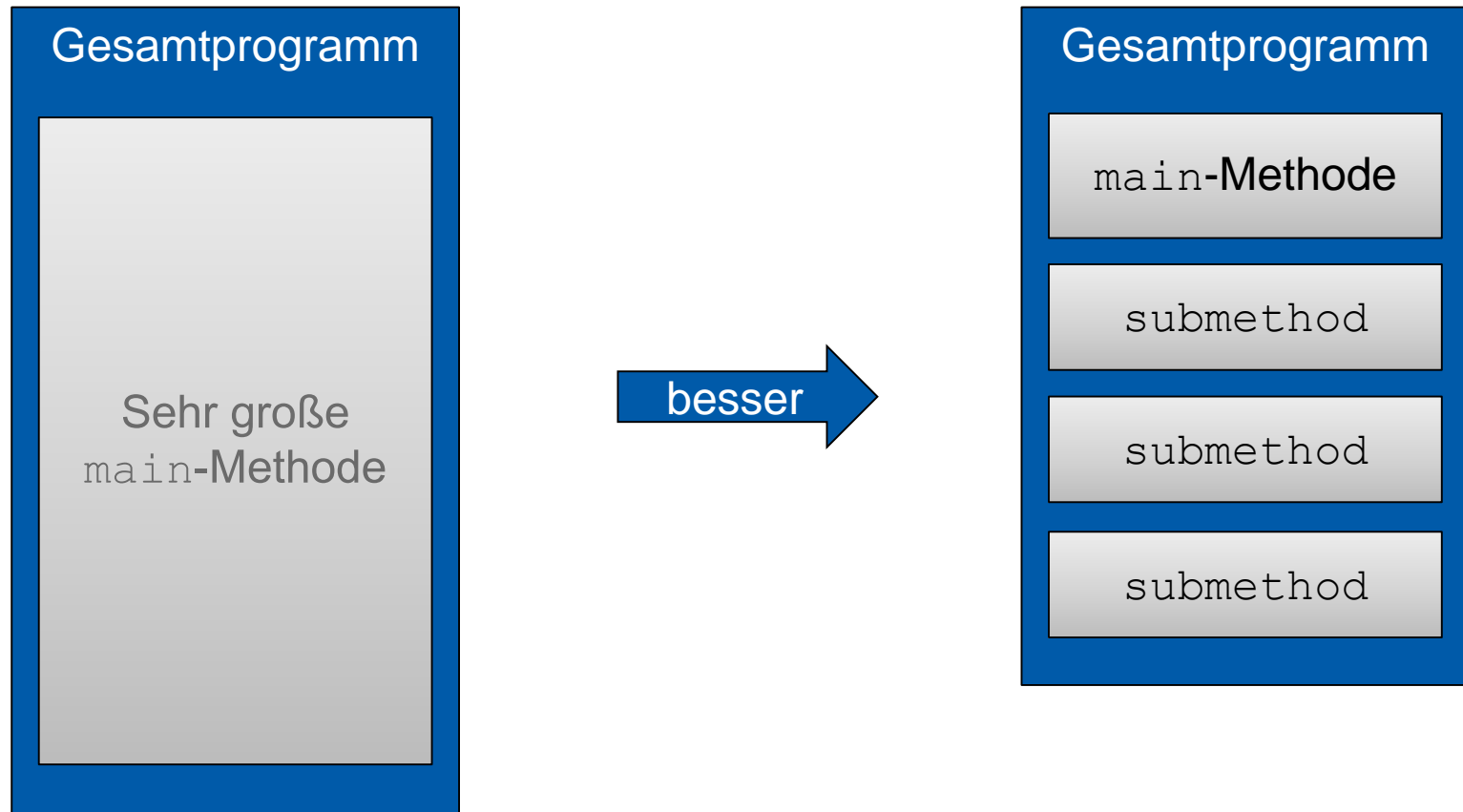
# Codebases

Millions of lines of code



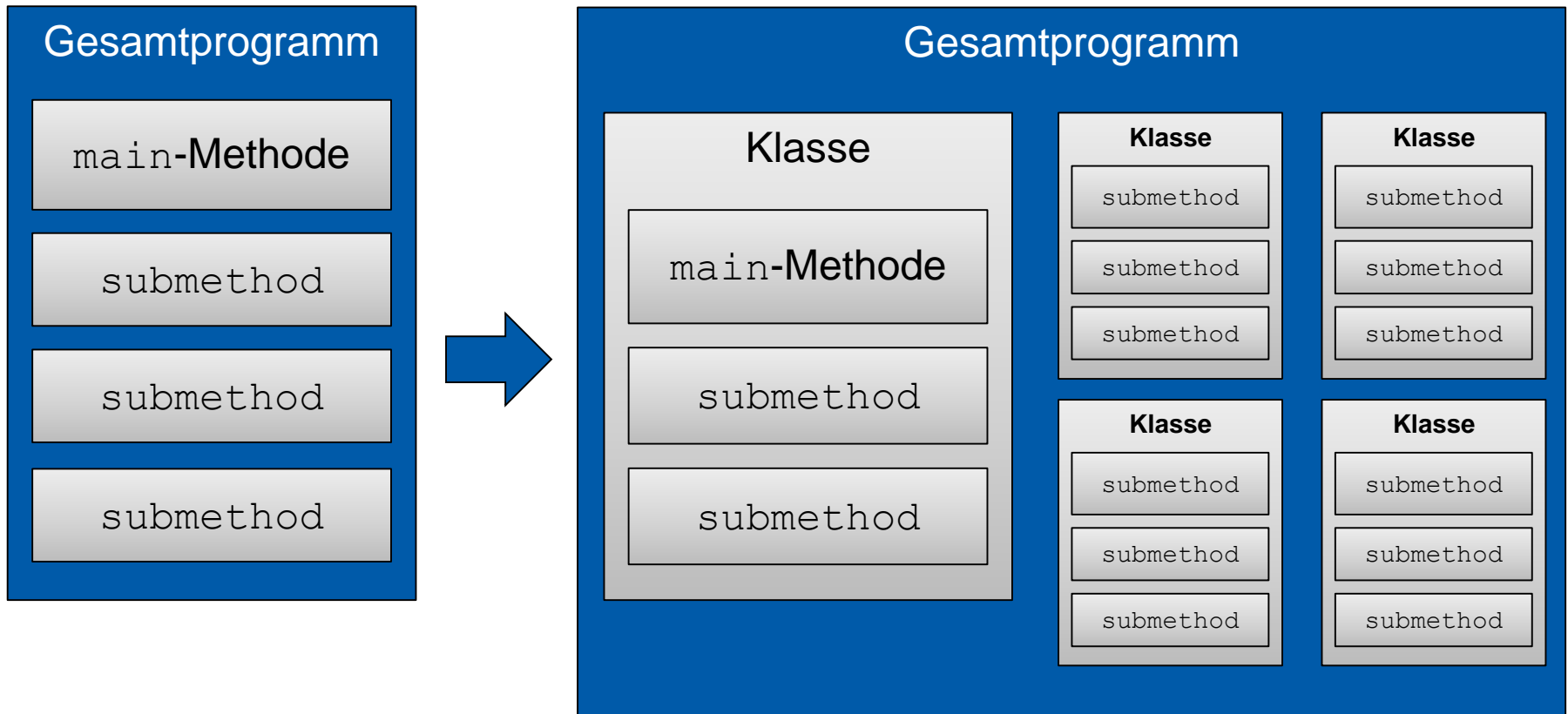
# Modularisierung Teil 1:

## Unterteilung in Methoden



# Modularisierung Teil 2:

## Unterteilung in Klassen

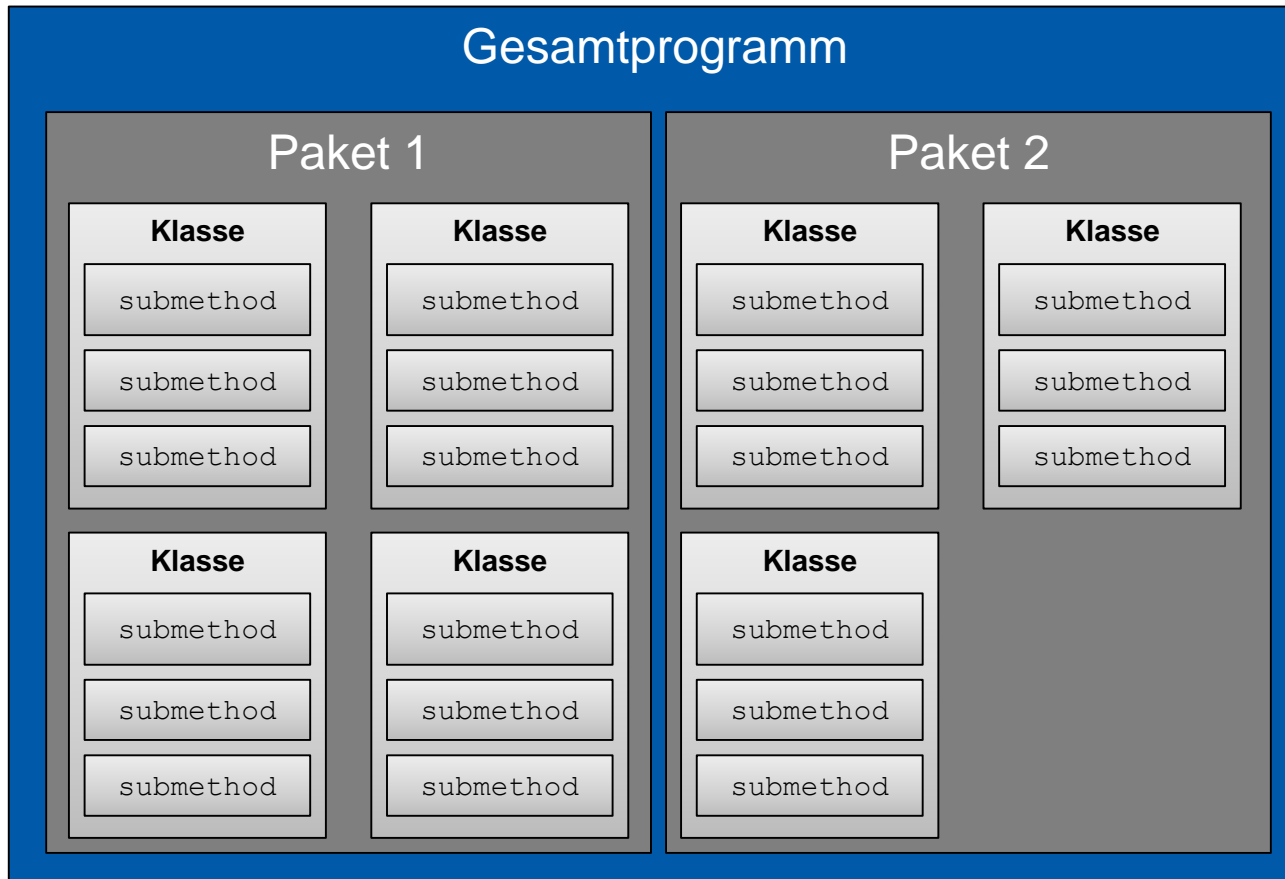


# Modularisierung Teil 3:

## Zusammenfassung zu Paketen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



- Pakete (Packages) sind Zusammenfassungen von Klassen (nach Zweck oder Typ)
- Die mit Java ausgelieferten Klassen sind bereits in Pakete unterteilt:

Paket	Erklärung
<code>java.lang</code>	Enthält grundlegende Sprach- und Systemklassen, wird automatisch geladen
<code>java.io</code>	Enthält eine Vielzahl an Klassen zur Ein- und Ausgabe
<code>java.net</code>	Enthält Klassen für die Netzwerkfunktionalitäten
<code>java.util</code>	Enthält diverse Hilfsklassen



# Eigene Klassen zu Paketen zusammenfassen

- Eigene Klassen können ebenfalls zu Paketen zusammengefasst werden
- Die Anweisung `package paketname` am Anfang der Java-Datei legt fest, zu welchem Paket eine Klasse gehört
  - Vor dieser Anweisung dürfen keine weiteren Anweisungen stehen
  - Die Dateien der Klassen des Pakets `paketname` werden im Unterverzeichnis `paketname` gespeichert



- Pakete können wiederum **Unterpakete** enthalten
  - Syntax: `package paketname.unterpaketname`
  - Die Dateien der Klassen des Pakets `paket.unterpaket` werden im Unterverzeichnis `paket\unterpaket` gespeichert


- Beispiel:

```
package de.tudarmstadt.is.gdp;
```

```
public class MyClass {
```

```
    ...
```

```
}
```



Datei `MyClass.java` (bzw. `MyClass.class`)  
befindet sich im Verzeichnis  
`de\tudarmstadt\is\gdp`



- Direkter Zugriff (über den Klassennamen) nur auf Klassen aus dem **eigenen Paket** und auf Klassen aus Paket `java.lang`
- Es gibt zwei Möglichkeiten zum Zugriff auf Klassen aus anderen Paketen
  - **Möglichkeit 1: Über den vollqualifizierten Namen:**  
`de.tudarmstadt.MyClass c = new de.tudarmstadt.MyClass();`
  - **Möglichkeit 2: Importieren**
    - Import nur einer Klasse :  
`import java.util.Scanner;`
    - Import aller Klassen in Paket `java.util`:  
`import java.util.*;`
    - Die Import-Anweisung steht hinter der `package`-Anweisung und vor dem Klassenkopf!

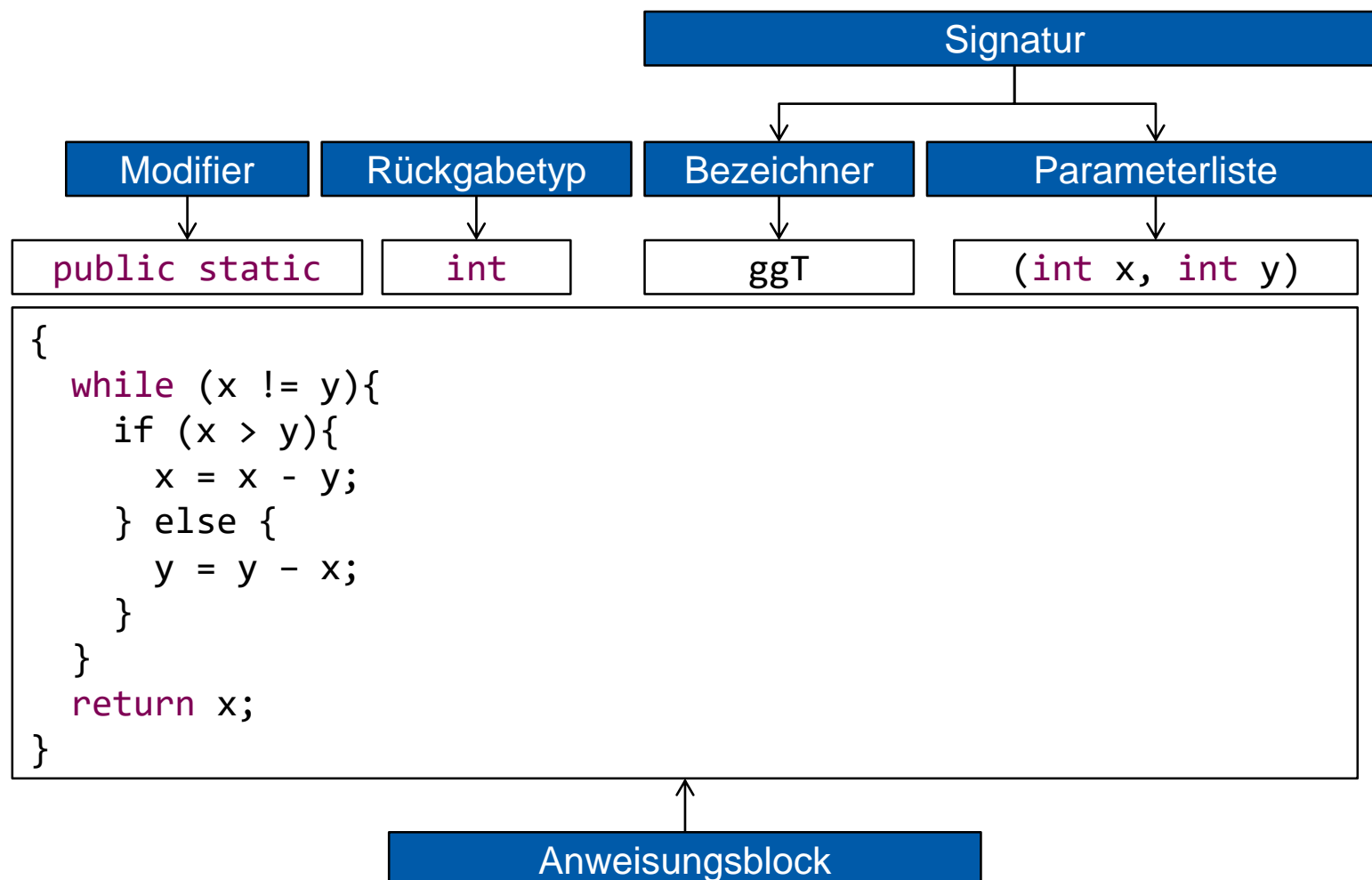
# Das Prinzip des „Information Hiding“

- Datenkapselung/Information Hiding = Verbergen von Implementierungsdetails
  - Kontrollierter Zugriff auf Methoden bzw. Attribute einer Klasse.  
Öffentliche Methoden und Attribute bestimmen die Schnittstelle, die darüber bestimmt, auf welche Weise mit der Klasse interagiert werden kann
  - Klassen können den internen Zustand anderer Klassen nicht in unerwarteter Weise lesen oder ändern (Wissen über die Implementierung einer Klasse für andere Klassen nicht nötig)
- Vorteile: Übersichtlichkeit, da nur die öffentliche Schnittstelle betrachtet werden muss → Bessere Wartbarkeit, Testbarkeit, Stabilität





# Elemente einer Methode



# Modifizier: Öffentliche und private Methoden und Attribute

Java erlaubt verschiedene Einschränkungen bzgl. des Zugriffs auf die Variablen / Methoden eines Objekts.

„Öffentlich“

„Privat“

**public:** auf öffentliche Variablen / Methoden darf von allen Klassen aus zugegriffen werden

**protected:** auf geschützte Variablen/Methoden darf nur von der eigenen Klasse, von Klassen im eigenen Paket sowie von Objekten der Unterklasse zugegriffen werden

Keine Sichtbarkeitsangabe (**package**): auf Variablen / Methoden darf nur im eigenen Paket zugegriffen werden

**private:** auf private Variablen / Methoden darf nur innerhalb der eigenen Klasse zugegriffen werden

# Sichtbarkeit (Beispiel)

## My Apple ID


### Reset your password

You can change or reset the password for your Apple ID account by providing some information.

### Verify Your Identity: Step 1 of 2

Enter the Recovery Key you were provided when setting up two-step verification.

**Recovery Key:**



[Lost your Recovery Key?](#)

CancelNext

# Sichtbarkeit von Methoden und Attribute in Klassenhierarchien

	private	package	protected	public
Gleiche Klasse	<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>
<b>Unterklassen</b> im gleichen Package		<b>X</b>	<b>X</b>	<b>X</b>
Klassen im gleichen Package		<b>X</b>	<b>X</b>	<b>X</b>
<b>Unterklassen</b> in anderen Packages			<b>X*</b>	<b>X</b>
Klassen in anderen Packages				<b>X</b>

\* Zugriff nur innerhalb eines Objekts, das das Attribut / die Methode geerbt hat

# „Getter-“ und „Setter-“Methoden

- Getter- und Setter-Methoden (auch: Zugriffsmethoden) werden verwendet, um Zugriff auf beschränkt sichtbare Attribute zu ermöglichen
  - Getter-Methoden liefern den Wert eines Attributs zurück
  - Setter-Methoden weisen dem Attribut einen Wert zu
- Namensschema:
  - *setAttributname()*
  - *getAttributname()* bzw. *isAttributname()* bei boolean
- Können z. B. Logik für Rechteprüfungen oder Protokollierung beinhalten

# Beispiel: Öffentliche und private Methoden/Attribute

Das Attribut  
hubraum kann von  
außen nicht direkt  
gelesen/verändert  
werden

Änderungen (von  
außen) sind nur  
über die Methode  
setHubraum  
möglich.

```
public class Auto {  
    ...  
    private int hubraum;  
    ...  
    public void setHubraum(int h){  
        if (h > 0) {  
            hubraum = h;  
        }  
    }  
  
    public int getHubraum(){  
        return hubraum;  
    }  
    ...  
}
```



# Aufgabe: Sichtbarkeiten



<http://pingo.upb.de>  
#9456

Gegeben seien zwei Klassen, welche der Zuweisungen in Klasse **A** funktionieren?

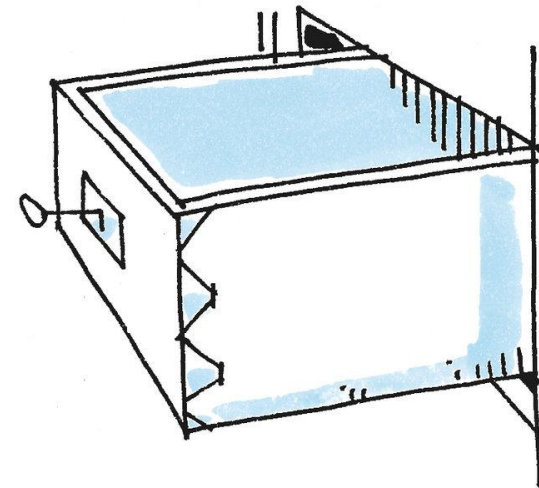
```
package y;
public class B {
    private int a = 1;
    int b = 1;
    protected int c = 1;
    public int d = 1;
}
```

```
package x;
import y.B;
public class A {
    public static void main(String[] args)
    {
        B myObject = new B();
        1 myObject.a = 42;
        2 myObject.b = 42;
        3 myObject.c = 42;
        4 myObject.d = 42;
    }
}
```



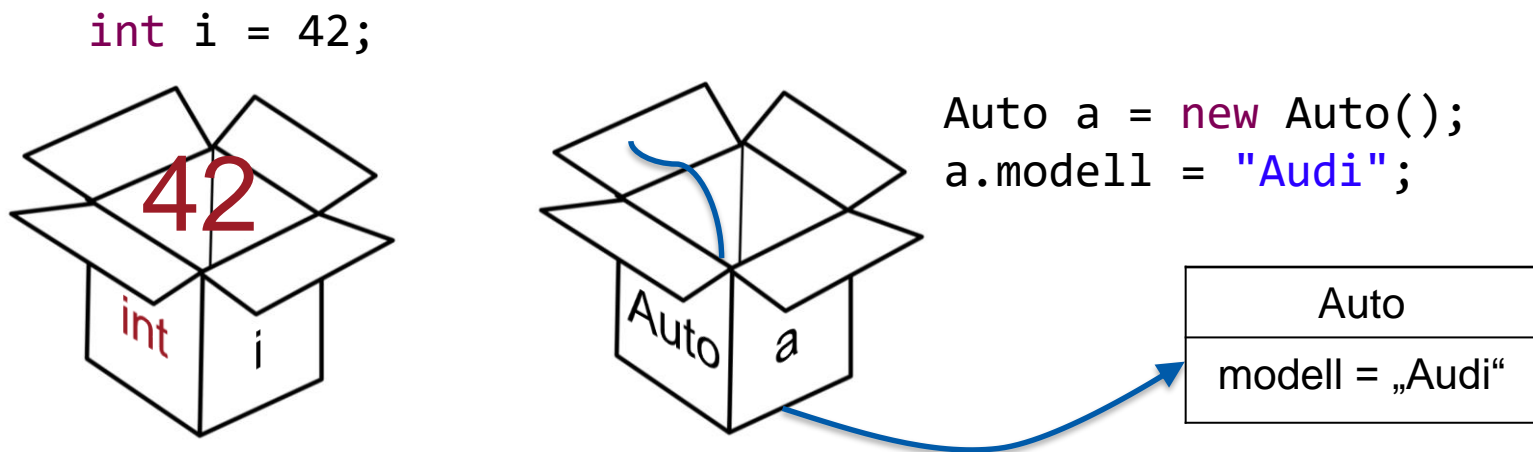
# Variablen und Arten von Datentypen

- **Variablen** sind Behälter für genau einen Wert und es gilt:
  - Variablen haben einen **Datentyp** (z. B. `int`)
  - Variablen haben einen **Bezeichner** (z. B. `x` oder `besteVariableDerWelt`)
  - Variablen haben einen **Wert** (z. B. `42`)
- Wir unterscheiden **zwei Arten von Datentypen**:
  - **Einfache** Datentypen:
    - Wahrheitswert (`boolean`) → `true`, `false`
    - Einzelzeichen (`char`) → z. B.: `'a'`, `'b'`, `'1'`, `'2'`, `'%'`, `'_'`
    - Numerische Datentypen
      - Ganzzahlige Datentypen (`byte`, `short`, `int`, `long`) → z. B. `123`
      - Gleitkommatypen (`float`, `double`) → z. B. `1.25`
  - **Strukturierte** Datentypen (auch: Referenz-Datentypen)



# Werte- und Referenzvariablen

- Variablen eines strukturierten Datentyps verweisen auf Objekte (sie werden daher als **Referenzvariablen** bezeichnet)
- Z. B. ist eine Variable vom Datentyp `Auto` eine Referenzvariable
- Während Variablen einfacher Datentypen (z. B. `int`, `double`) direkt Werte des jeweiligen Datentyps speichern, enthalten Referenzvariablen als Wert lediglich einen Verweis (eine Referenz) auf ein Objekt



# Der wesentliche Unterschied...



Variablen einfachen Datentyps:

```
int a = 1;
```

```
int b = 2;
```

```
a = b;
```

```
b = 42;
```

Am Ende des Programms enthalten  
a und b verschiedene Werte:

a = 2

b = 42

Sie können gleich sein, aber  
niemals identisch!

Variablen strukturierten Datentyps:

```
Auto a = new Auto();
```

```
a.modell = "Audi";
```

```
Auto b = new Auto();
```

```
b.modell = "VW";
```

```
a = b;
```

```
b.modell = "BMW";
```

Am Ende des Programms sind a  
und b identisch, d. h. sie  
verweisen auf dasselbe Objekt,  
dessen Attribut modell auf "BMW"  
gesetzt wurde.

# Der wesentliche Unterschied...

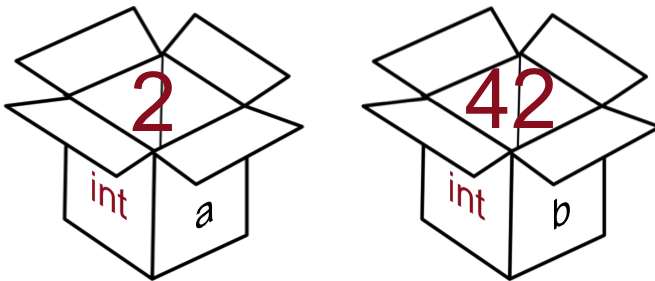
Variablen einfachen Datentyps:

```
int a = 1;
```

```
int b = 2;
```

```
a = b;
```

```
b = 42;
```



Variablen strukturierten Datentyps:

```
Auto a = new Auto();
```

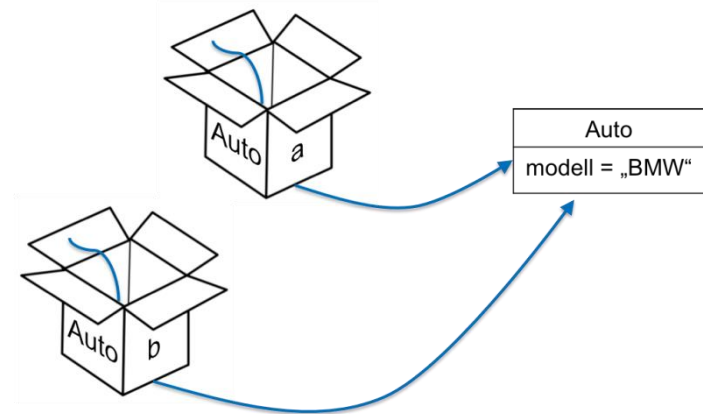
```
a.modell = "Audi";
```

```
Auto b = new Auto();
```

```
b.modell = "VW";
```

```
a = b;
```

```
b.modell = "BMW";
```



- Der Vergleichsoperator `==` überprüft die Übereinstimmung zweier Werte, d.h. bei Objektvariablen werden die gespeicherten Referenzen auf Gleichheit überprüft, aber nicht die Attribute der referenzierten Objekte!

- Beispiel:

```
Auto a = new Auto();  
a.modell="Golf 3";  
Auto b = new Auto();  
b.modell="Golf 3";  
System.out.println(a == b); // Ausgabe: false  
a = b;  
System.out.println(a == b); // Ausgabe: true
```

# Die Urklasse: `java.lang.Object`



- Jede Klasse, die nicht explizit von einer Oberklasse abgeleitet wird, wird in Java automatisch von `java.lang.Object` abgeleitet. Damit ist `java.lang.Object` die **Urklasse**, von der alle anderen Klassen abgeleitet sind!
  - Eine Variable vom Typ `Object` kann Objekte einer beliebigen Klasse aufnehmen!
  - `java.lang.Object` definiert einige Methoden, die von allen Klassen geerbt werden – Beispiele sind:
    - `boolean equals(Object o)` vergleicht zwei Objekte
    - `String toString()` gibt eine Repräsentation des Objekts als String zurück (*nicht aus!* - z. B. für `System.out.println(...)`)
    - `Object clone()` liefert eine Kopie des Objekts



# Die Urklasse: `java.lang.Object`

- Die Implementierung ist meist so allgemein, **dass abgeleitete Klassen diese überschreiben müssen.**
  - Die Methode `Object.toString()` liefert einen String bestehend aus Klassenname und Speicheradresse, z. B. `"Auto@1263249"`
  - Die Methode `Object.equals(Object o)` vergleicht Objekte basierend auf deren Speicheradresse. Dies entspricht einem Vergleich mit dem `"=="` Operator bei einfachen Datentypen. **Der `"=="` Operator vergleicht bei Objekten (d.h. auch bei Strings!) nur die Speicheradressen** und nicht die Werte der Objekte. Deshalb sollte die Methode `equals(Object o)` in jeder Klasse überschrieben werden und dann zum Vergleich genutzt werden. Die Klasse `String` bietet bereits eine überschriebene `equals(Object o)` Methode.





# Vergleich von Objekten mit `equals(...)`

- Durch Überschreiben der Methode `equals(Object o)` kann festgelegt werden, wann Objekte einer Klasse als „gleich“ erkannt werden sollen.
- Der Methode `equals(Object o)` eines Objekts A wird ein weiteres Objekt B übergeben, welches dann mit dem Objekt A verglichen werden kann (z.B. durch Vergleich der Attributwerte)

# Beispiel: Vergleiche mit überschriebenen equals()-Methoden



```
public class Auto {  
    public String modell;  
  
    public boolean equals(Object o) {  
        if (o instanceof Auto){  
            Auto a = (Auto) o;  
            if(modell.equals(a.modell)){  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

```
public class Hauptprogramm{  
  
    public static void main(String[] args) {  
        Auto a1 = new Auto();  
        a1.modell = "Audi";  
        Auto a2 = new Auto();  
        a2.modell = "Audi";  
        System.out.println(a1 == a2);  
        // Ausgabe: „false“  
        System.out.println(a1.equals(a2));  
        // Ausgabe: „true“  
    }  
}
```