

# DIE SÄULEN DER OBJEKTOIENTIERTEN PROGRAMMIERUNG TEIL 2/2

VOM 25/11/2024 BIS 06/12/2024

## OOP DEFINITION

Die **objektorientierte Programmierung** (kurz **OOP**) ist ein auf dem Konzept der Objektorientierung basierendes Programmierparadigma, welches Flexibilität und Wiederverwendbarkeit von Programmen fördert.

Die Grundidee der objektorientierten Programmierung ist, Daten und Funktionen, die auf diese Daten angewandt werden können, möglichst eng in einem sogenannten *Objekt* zusammenzufassen und nach außen hin zu *kapseln*, so dass Methoden fremder Objekte diese Daten nicht versehentlich manipulieren können.

Im Gegensatz dazu beschreibt das vor der OOP vorherrschende Paradigma eine strikte Trennung von Funktionen (Programmcode) und Daten, dafür aber eine schwächere Strukturierung der Daten selbst.

## WARUM OBJEKTORIENTIERT PROGRAMMIEREN?

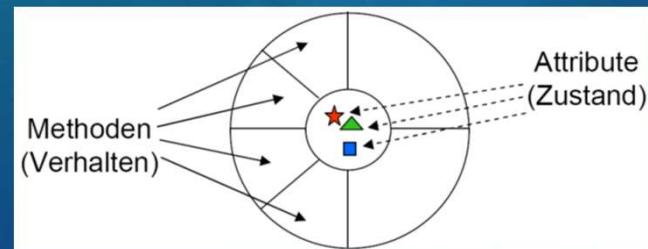
- Software lässt sich schneller entwickeln
- ist leichter zu warten
- weniger Fehleranfällig sowie einfacher zu debuggen
- Source Code ist übersichtlicher
- Wiederverwendbar (Frameworks)

## WAS ZEICHNET OBJEKTORIENTIERTE PROGRAMMIERUNG AUS?

- *Abstraktion*  
Ein Objekt verkörpert einen Arbeiter der verschiedene Eigenschaften hat und verschiedene Aufgaben übernehmen kann.
- *Kapselung*  
Jedes Objekt ist eine Kapsel, das heißt Eigenschaften und Verhaltensweisen die in dem Objekt implementiert werden überschreiben nicht etwaig gleichlautende Eigenschaften außerhalb des Objektes.
- *Vererbung*  
Es können allgemeine Verhaltensweisen und Eigenschaften von Objekten in einer Oberklasse heraus generalisiert werden. So kann doppelter Code auf leichte Weise vermieden werden.
- *Polymorphie*  
Klassen die von einer Oberklasse allgemeine Eigenschaften und Verhaltensweisen geerbt hat können diese überschreiben um so ein spezialisiertes Verhalten zu erzeugen.

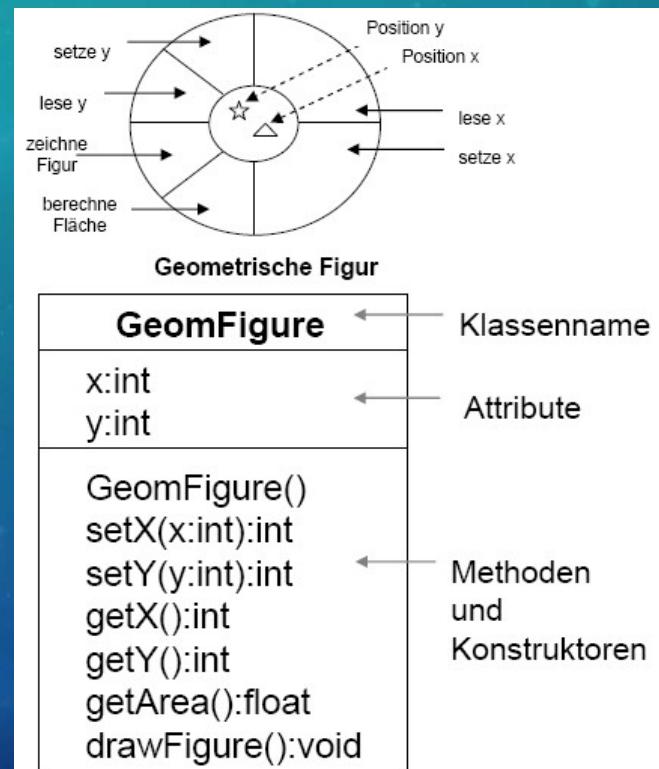
# OOP MIT JAVA: OBJEKTE

- (Software-)Objekte besitzen einen internen **Zustand** und haben ein festgelegtes **Verhalten**
- Ein Objekt
  - verwaltet seinen Zustand durch Variablen, Eigenschaften oder **Attribute** genannt
  - implementiert sein Verhalten durch Funktionen, **Methoden** genannt
- Ein Java Objekt kann zum Beispiel eine Person, ein Ball oder ein Raumschiff sein.



# UML NOTATION

- Die Unified Modeling Language (UML) ist eine Notation zur Spezifikation und Visualisierung von Modellen für Softwaresysteme
- In den folgenden Folien werden **Klassendiagramme** zur Beschreibung von Klassen und deren Beziehungen untereinander verwendet



# KLASSEN UND OBJEKTE (INSTANZEN)

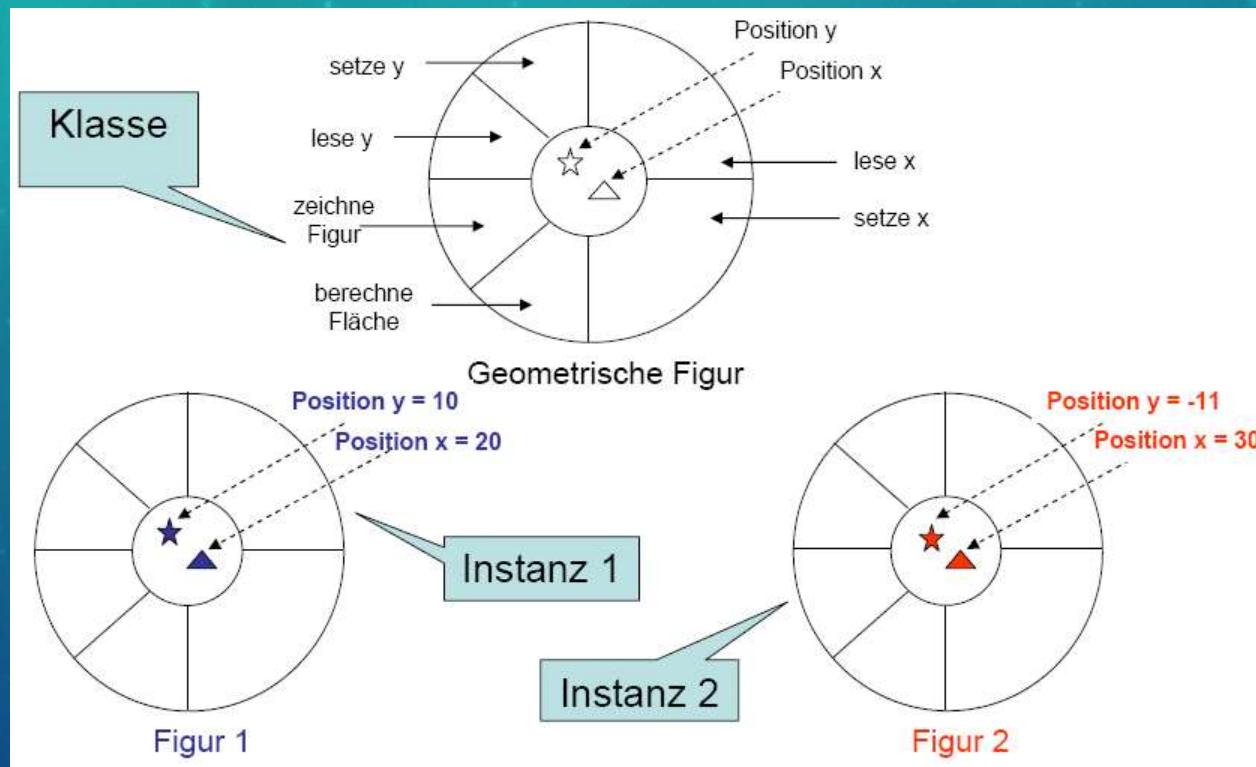
**Klassen sind Vorlagen für die Erzeugung von Objekten**

- Jedes Objekt gehört einer Klasse an.
- Die Attribute und Methoden der Objekte werden in der zugehörigen Klasse definiert
- Alle Objekte einer Klasse besitzen dasselbe Verhalten, da sie dieselbe Implementierung der Methoden besitzen
- Objekte einer Klasse werden auch als **Instanzen** (*instance*) oder **Exemplare** bezeichnet
- Klassen können auch Methoden und Attribute besitzen, die für die Klasse selbst und nicht für jede Instanz gelten. Diese werden als **Klassen-Methoden/-Attribute** bezeichnet (im Gegensatz zu **Instanz-Methoden/-Attribute**)

# ERZEUGUNG VON OBJEKTEN

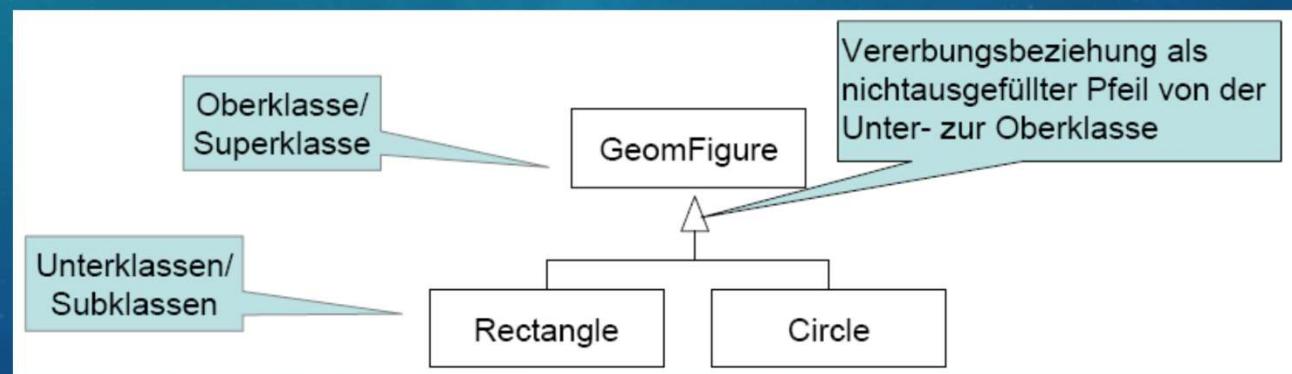
- Es wird Speicherplatz für das Objekt auf dem Heap allokiert (abhängig von den Datentypen der Attribute)
- Bei der Erzeugung wird ein **Konstruktor** (spezielle Methode) ausgeführt, der die Initialisierung des Objekts vornimmt
- Es werden den Attributen Werte zugeordnet, wodurch der Zustand des Objekts definiert wird
  - entweder definiert durch den Konstruktor
  - oder Default-Werte (0 für Zahlen, *null* für Referenzen, etc.)
- Die Erzeugung eines Objekts erfolgt in den meisten Programmiersprachen mit dem **new-Operator**

# BEISPIEL: EINE KLASSE UND ZWEI INSTANZEN (OBJEKTE) DER KLASSE

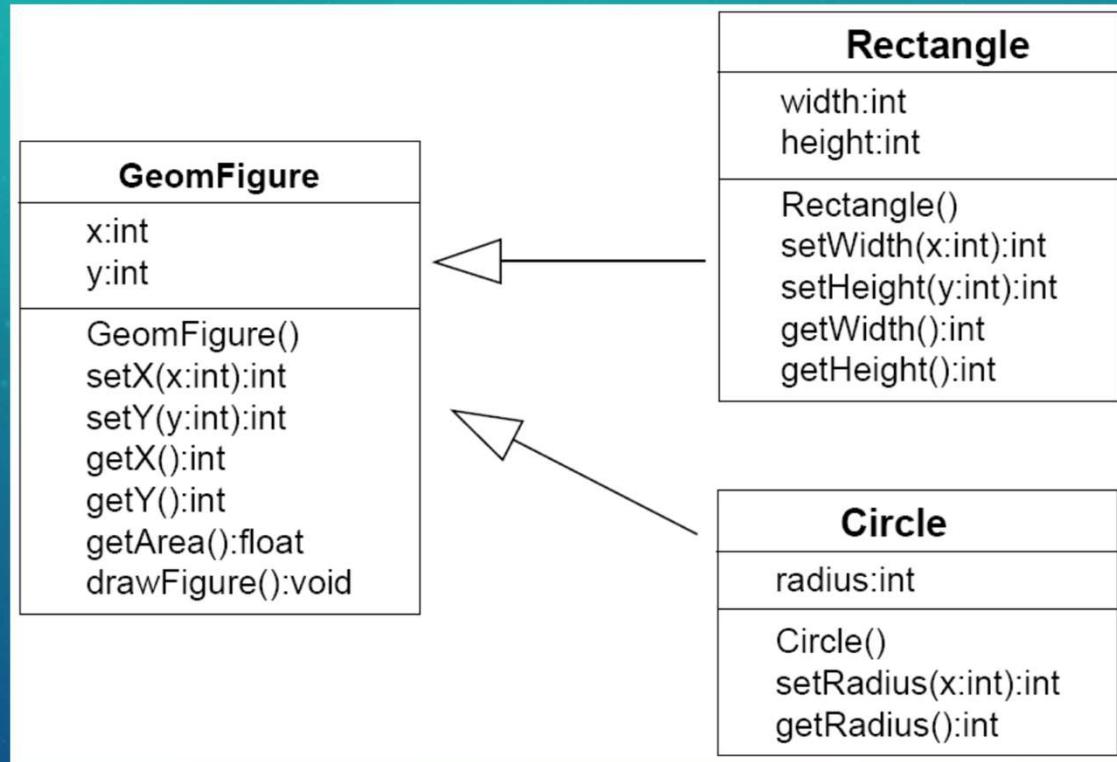


# VERERBUNG

- Vererbung (*inheritance*) ist ein Konzept, wodurch Attribute und Methoden der Oberklasse auch den Unterklassen zugänglich gemacht werden
- Vererbungsbeziehungen zwischen Klassen stellen Generalisierungs- bzw. Spezialisierungsbeziehungen dar



# VERERBUNG



# VERERBUNG: DIE KLASSE SQUARE

- Die Klasse Square soll nun durch Vererbung von der Klasse GeomFigure abgeleitet werden.
- Wir benötigen einen Konstruktor, der neben der x- und y-Koordinate die Seitenlänge des Quadrates erhält.
- Weiterhin solle eine Methode zur Berechnung des Flächeninhaltes implementiert werden sowie die `toString`-Methode überschrieben werden.

# ÜBERSCHREIBEN VON METHODEN

- Hat eine Methode einer Subklasse **die gleiche Signatur** einer Methode der Superklasse, so wird die Superklassenmethode überschrieben (**overriding**)
- GeomFigure definiert eine Methode `toString()`:

```
String tmp="[" + myX + "," + myY + "]";  
return tmp;
```

- Square überschreibt die Methode `toString()`:

```
String tmp=super.toString();  
tmp=tmp+" "+seitenL;  
return tmp;
```

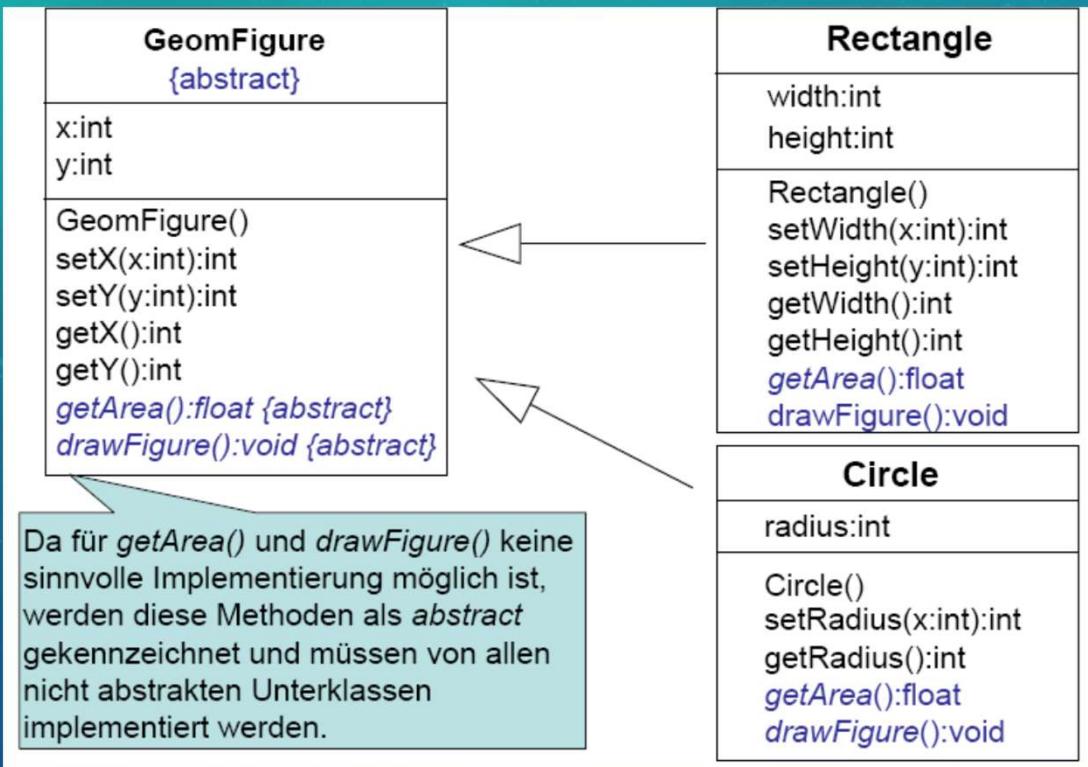
# ÜBERLADEN VON METHODEN

- Gibt es mehrere Methoden mit demselben Namen aber mit **unterschiedlichen Signaturen**, so spricht man von Überladen (**overloading**).
- Beispiel:  
`void println() ...`  
`void println (String s) ...`  
`void println (int i) ...`

# ABSTRAKTE KLASSEN

- Abstrakte Klassen sind Klassen, die mindestens eine **abstrakte Methode** haben
- Abstrakte Methoden besitzen keine Implementierung.
- Von abstrakten Klassen können **keine Instanzen** erzeugt werden
- Unterklassen müssen entweder die abstrakten Methoden implementieren oder sind ebenfalls abstrakt

# BEISPIEL: ABSTRAKTE KLASSEN



# ZUSAMMENFASSUNG

## Objekte

- besitzen Attribute und Methoden
- gehören einer Klasse an

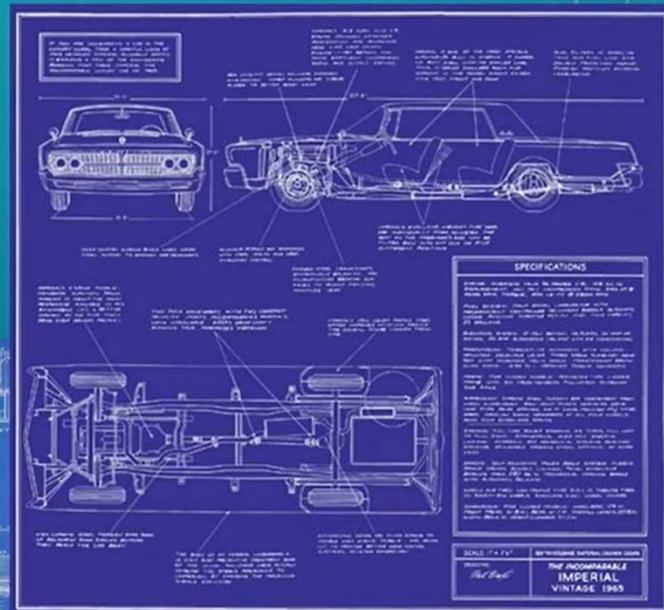
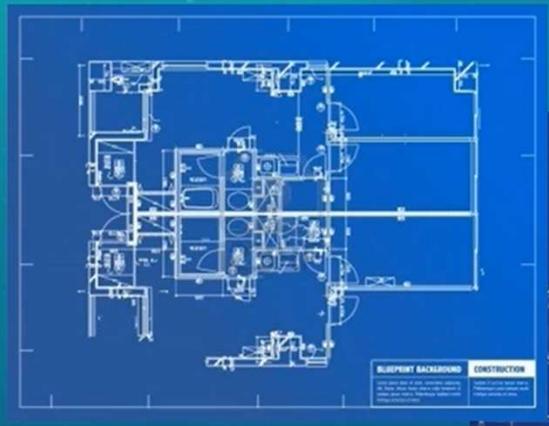
## Klassen

- legen Methoden und Attribute der Objekte fest (Bauplan)
- können Klassen-Methoden-/Attribute besitzen
- können abstrakt sein
- Objekte werden auf dem Heap erzeugt
- den Attributen werden Werte zugeordnet
- Konstruktoren werden ausgeführt
- Vererbung: Attribute und Methoden der Oberklasse werden den Unterklassen zugänglich gemacht

# VORTEILE VON OOP

- Wiederverwendung von Code
- Einfachere Anpassbarkeit von Code, damit auch bessere Wartbarkeit
- Objektorientierte Systeme ermöglichen einen strukturierten Systementwurf
- Software Engineering: OOA, OOD, OOP
- Nachteile: Lernaufwand, bei einfachen Programmen evt. mehr Code

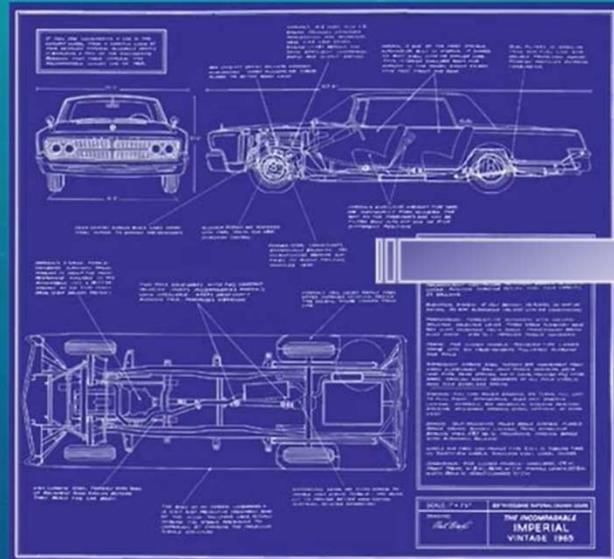
# WAS SIND KLASSEN?



Bauanleitungen - Blueprints

allgemeine Anweisungen zum Erstellen von  
Objekten

# OBJEKTE



Blueprint [Klasse]

Zusammenbauen  
[Initialisieren]

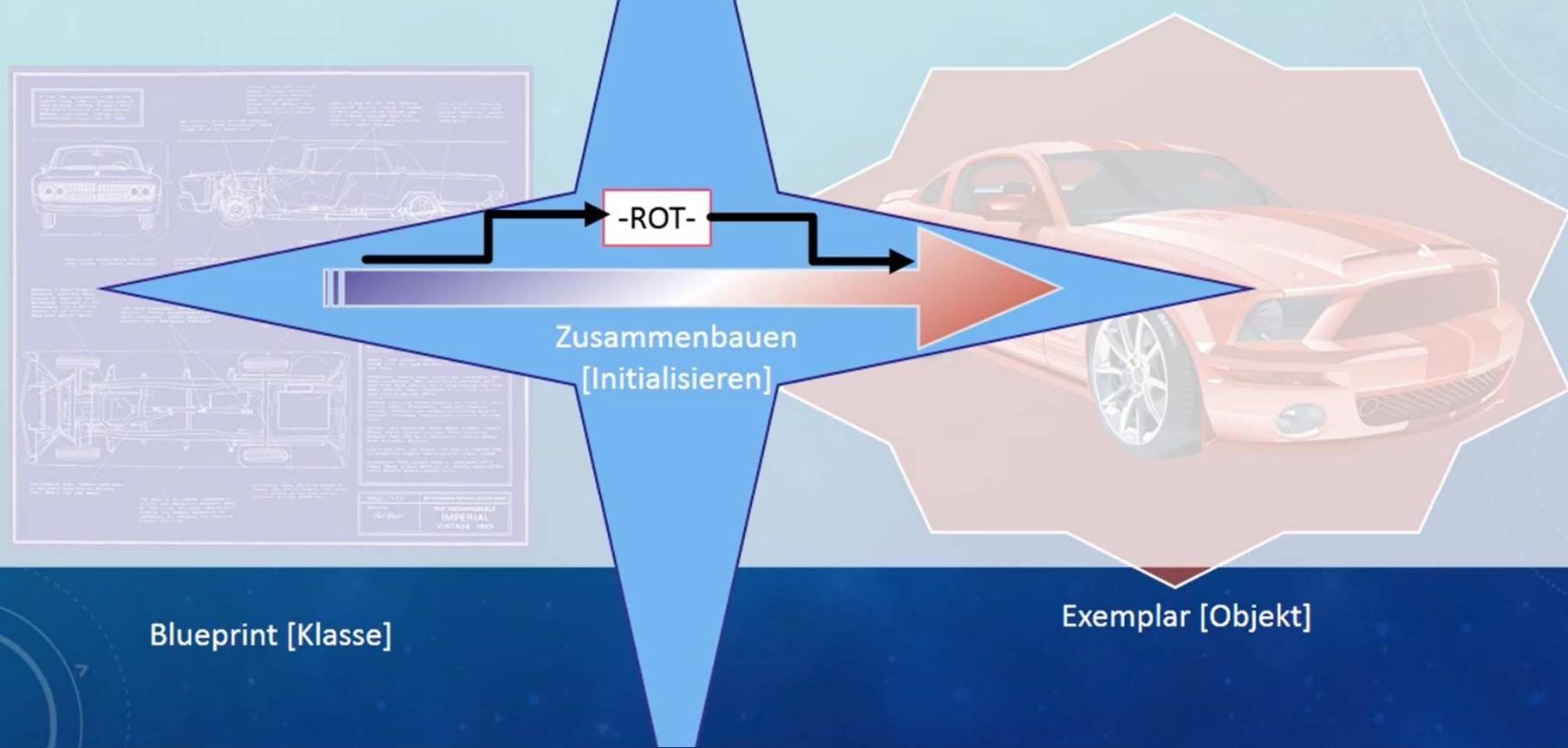


Exemplar [Objekt]

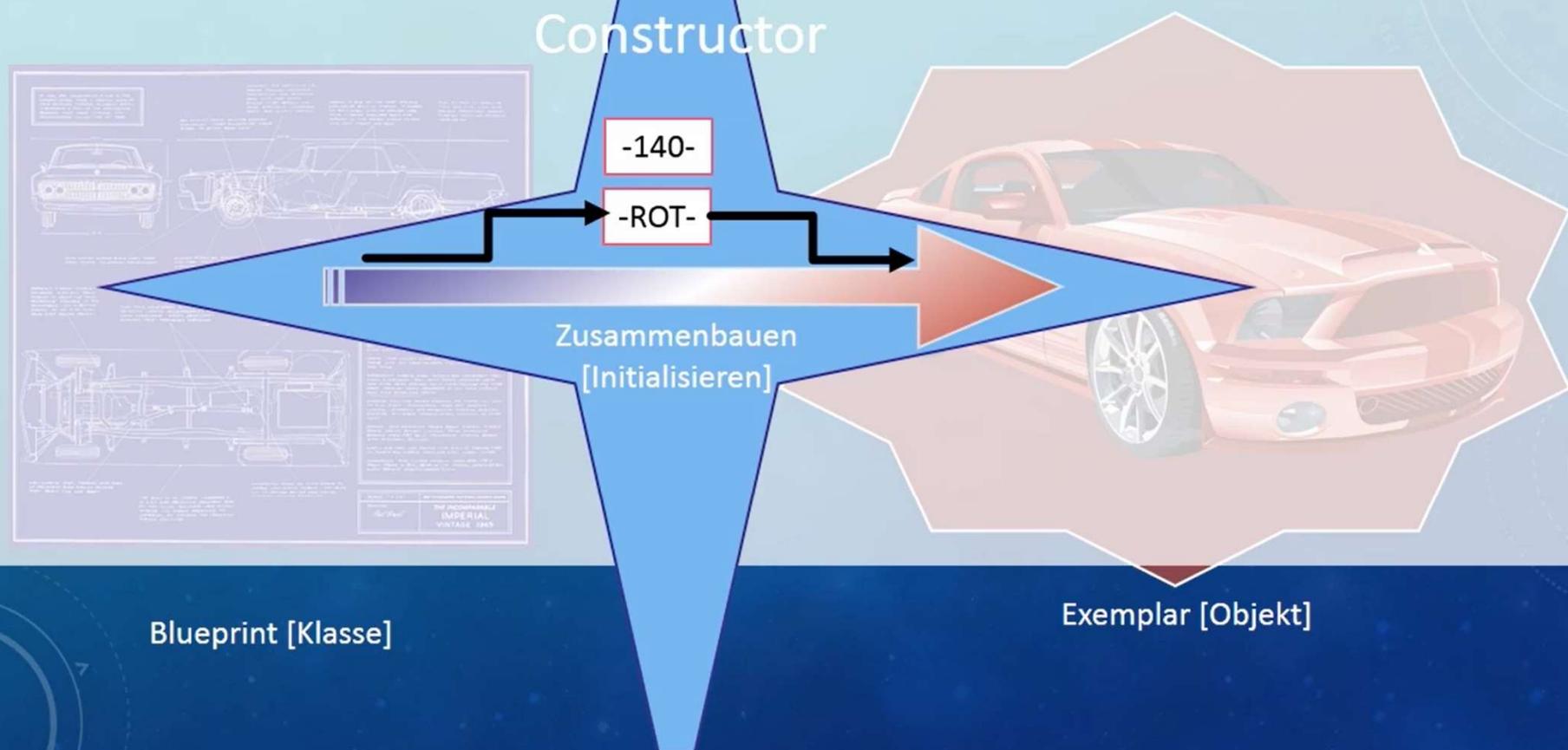
# ATTRIBUTE



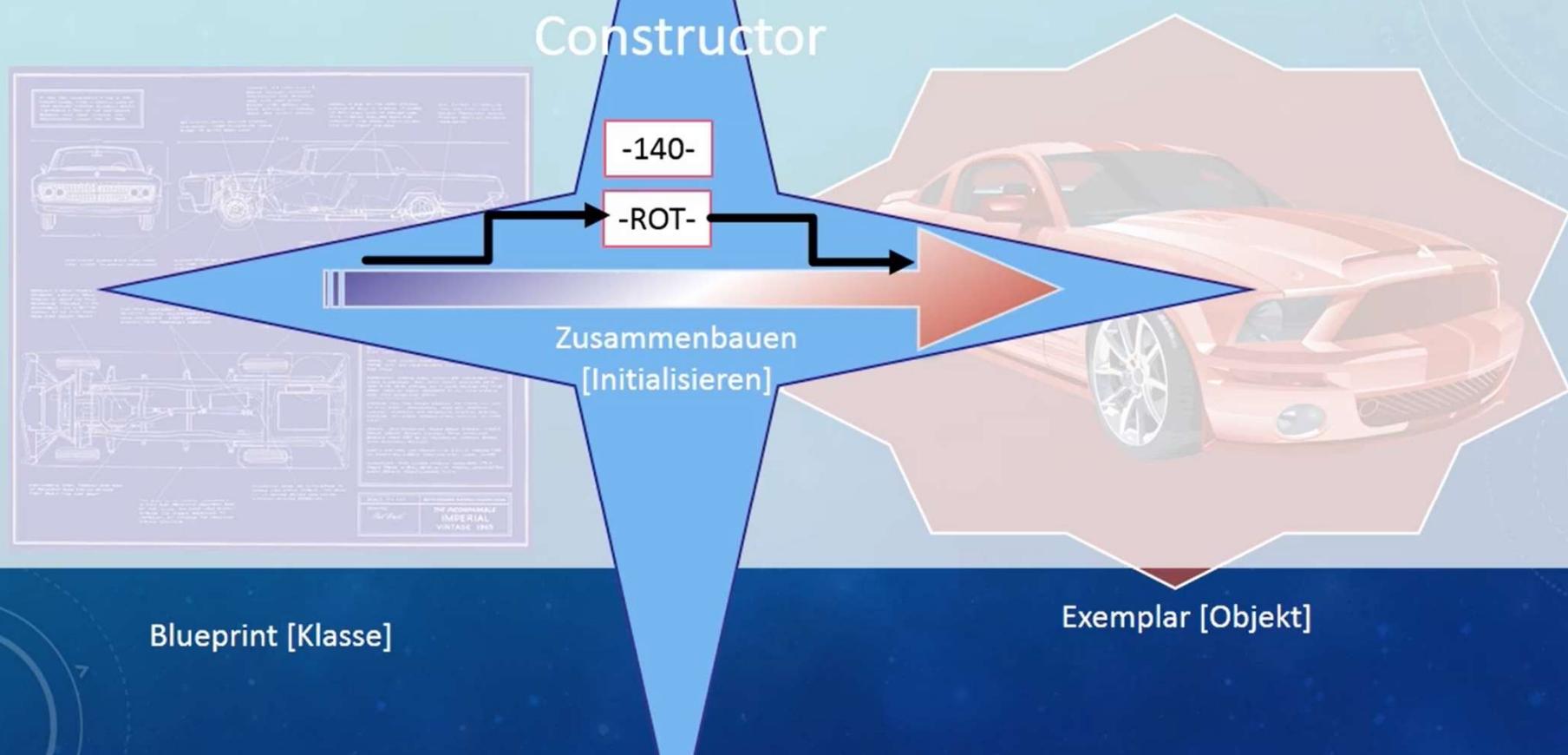
# OBJEKTE



# OBJEKTE



# OBJEKTE



# INITIALISIEREN /BAU VON OBJEKten

Festlegung von  
Attributen für Auto:

Farbe

Maximalgeschwindigkeit



# ATTRIBUTE



# ATTRIBUTE

Objekt	Farbe	Maximalgeschwindigkeit
1	rot	120
2	gelb	110
3	grün	100
4	blau	110
5	lila	100

Attribute immer mit Werten füllen!

# METHODEN



## Attribute

Maximalgeschwindigkeit



Farbe



# METHODEN

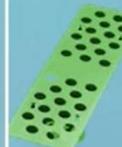


weitere Attribute

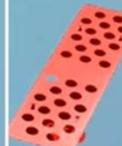
momentaneGeschwindigkeit (<120MPH)

Lenkradausrichtung

## Aufgaben



Beschleunigen



Bremsen



Lenken

## Attribute

Maximalgeschwindigkeit



Farbe



# METHODEN



weitere Attribute

momentaneGeschwindigkeit (<120MPH)

Lenkradausrichtung



## Attribute

Maximalgeschwindigkeit



Farbe



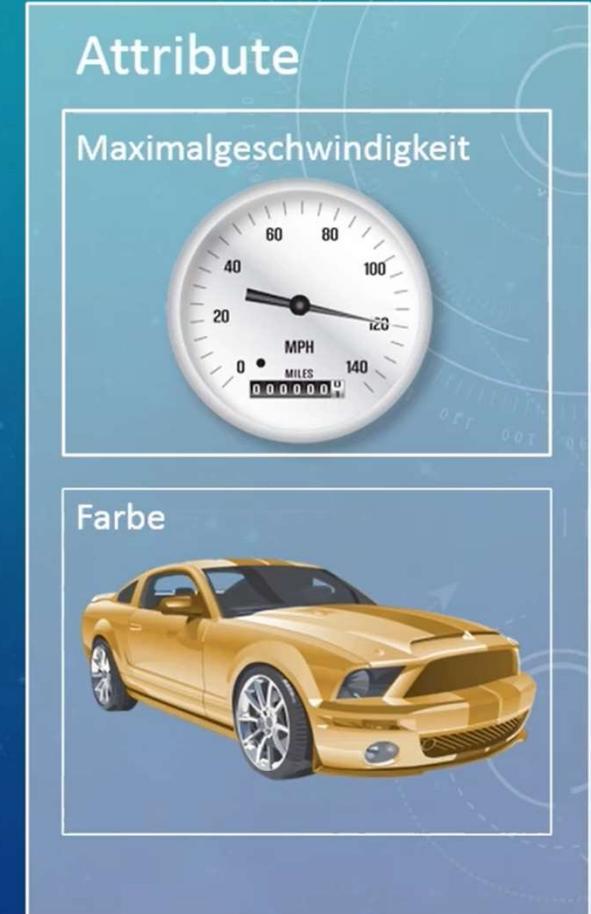
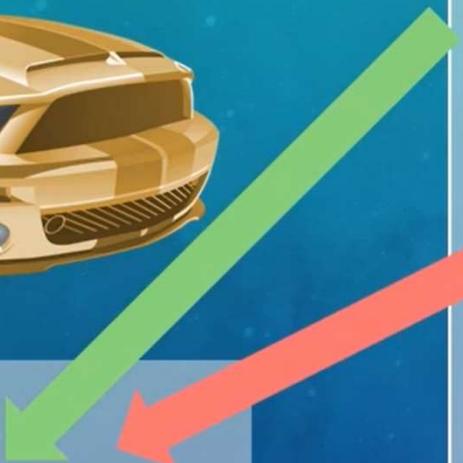
# METHODEN



weitere Attribute

momentaneGeschwindigkeit (<120MPH)

Lenkradausrichtung



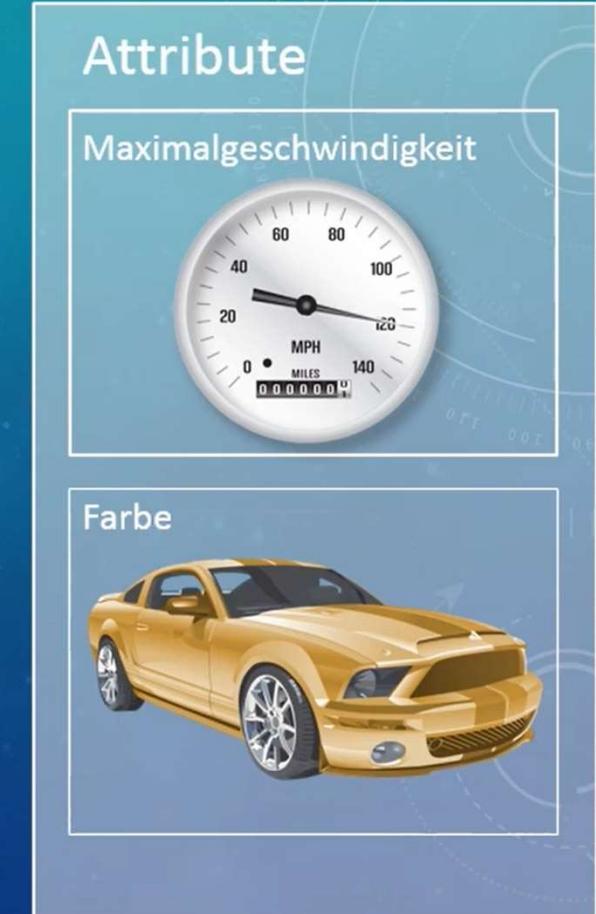
# METHODEN



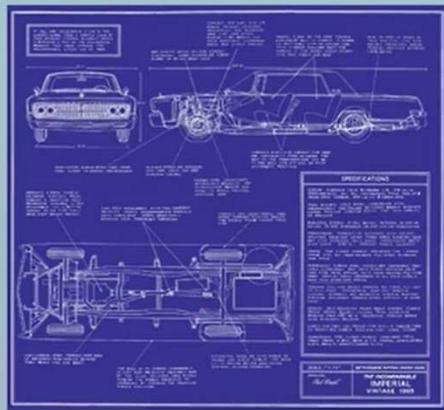
weitere Attribute

momentaneGeschwindigkeit (<120MPH)

Lenkradausrichtung



# ZUSAMMENFASSUNG



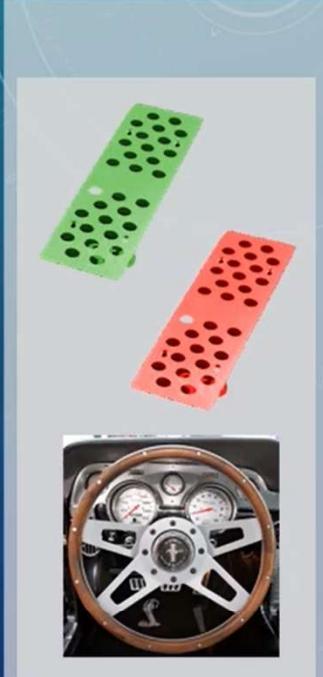
Bauplan /  
Blueprint /  
Klasse



Konstruieren /  
Zusammenbauen /  
Initialisieren



Objekt /  
Exemplar



Aufgaben /  
Methoden

# Handout: Objektorientiertes Design in Java

---

## Einführung in Klassen und Objekte

In der objektorientierten Programmierung (OOP) sind **Klassen** und **Objekte** die zentralen Konzepte. Eine Klasse ist ein **Bauplan**, der beschreibt, wie Objekte aufgebaut werden. Objekte sind **Instanzen** einer Klasse, die spezifische Daten und Verhalten kapseln.

## Vorteile von Klassen und Objekten

- **Kapselung:** Daten und Methoden, die auf diese Daten zugreifen, sind in einer Einheit zusammengefasst.
  - **Wiederverwendbarkeit:** Einmal definierte Klassen können für verschiedene Anwendungen verwendet werden.
  - **Modularität:** Programmcode ist strukturierter und leichter wartbar.
  - **Flexibilität und Erweiterbarkeit:** Klassen können durch Vererbung erweitert werden.
- 

## Aufbau von Klassen

Eine Klasse besteht aus:

1. **Attributen:** Variablen, die den Zustand eines Objekts beschreiben.
2. **Methoden:** Funktionen, die das Verhalten eines Objekts definieren.

## Struktur einer Klasse

```
class Klassenname {  
    // Attribute (Zustand)  
    Datentyp attributName;  
  
    // Methoden (Verhalten)  
    Rückgabewert methodName(Parameterliste) {  
        // Methodenkörper  
    }  
}
```

## Beispiel: Klasse Auto

```
class Auto {  
    // Attribute  
    String farbe;  
    int maximaleGeschwindigkeit;  
    int momentaneGeschwindigkeit;  
    String lenkradAusrichtung;
```

```

// Methoden
void beschleunigen(int geschwindigkeit) {
    momentaneGeschwindigkeit += geschwindigkeit;
}

void bremsen(int geschwindigkeit) {
    momentaneGeschwindigkeit -= geschwindigkeit;
    if (momentaneGeschwindigkeit < 0) {
        momentaneGeschwindigkeit = 0;
    }
}

void lenken(String richtung) {
    lenkradAusrichtung = richtung;
}
}

```

## Objekte: Zugriff, Wertzuweisung und Gültigkeit

Ein **Objekt** ist eine Instanz einer Klasse. Es wird mit dem Schlüsselwort `new` erstellt.

### Objekte erstellen

```

public class Main {
    public static void main(String[] args) {
        // Objekte der Klasse Auto erstellen
        Auto rotesAuto = new Auto();
        rotesAuto.farbe = "Rot";
        rotesAuto.maximaleGeschwindigkeit = 120;

        Auto gelbesAuto = new Auto();
        gelbesAuto.farbe = "Gelb";
        gelbesAuto.maximaleGeschwindigkeit = 110;

        // Attribute ändern
        rotesAuto.beschleunigen(50);
        System.out.println("Momentane Geschwindigkeit: " +
rotesAuto.momentaneGeschwindigkeit);
    }
}

```

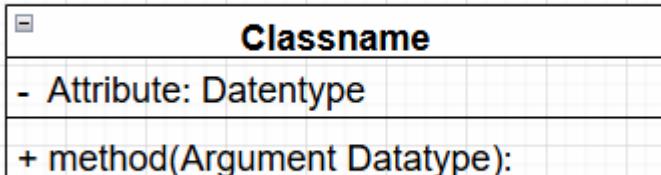
### Eigenschaften eines Objekts ändern

1. Attribute werden direkt oder über Methoden verändert.
2. Methoden werden aufgerufen, um das Verhalten eines Objekts zu definieren.

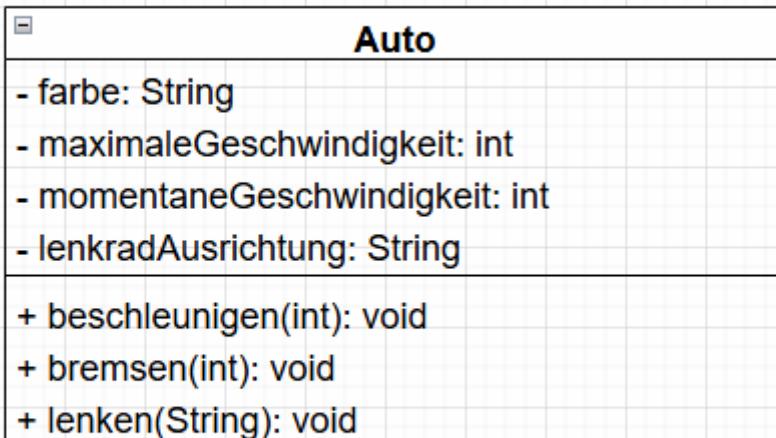
---

## UML-Klassendiagramme

UML-Klassendiagramme sind grafische Darstellungen von Klassen und deren Beziehungen.



Beispiel: Klassendiagramm **Auto**



- +: öffentliche Elemente (**public**)
- -: private Elemente (**private**)

## Erweiterung: Instanzen des Autos

Erstellen von fünf Autos mit unterschiedlichen Attributwerten:

```
public class Main {  
    public static void main(String[] args) {  
        Auto rotesAuto = new Auto();  
        rotesAuto.farbe = "Rot";  
        rotesAuto.maximaleGeschwindigkeit = 120;  
  
        Auto gelbesAuto = new Auto();  
        gelbesAuto.farbe = "Gelb";  
        gelbesAuto.maximaleGeschwindigkeit = 110;  
  
        Auto gruensAuto = new Auto();  
        gruensAuto.farbe = "Grün";  
        gruensAuto.maximaleGeschwindigkeit = 100;
```

```

        Auto blauesAuto = new Auto();
        blauesAuto.farbe = "Blau";
        blauesAuto.maximaleGeschwindigkeit = 110;

        Auto lilaAuto = new Auto();
        lilaAuto.farbe = "Lila";
        lilaAuto.maximaleGeschwindigkeit = 100;
    }
}

```

## Zusammenfassung

Begriff	Beschreibung
<b>Klasse</b>	Ein Bauplan für Objekte. Beschreibt Zustand (Attribute) und Verhalten (Methoden).
<b>Objekt</b>	Eine Instanz einer Klasse mit spezifischen Werten für die Attribute.
<b>Attribute</b>	Variablen innerhalb einer Klasse, die den Zustand eines Objekts speichern.
<b>Methoden</b>	Funktionen innerhalb einer Klasse, die das Verhalten eines Objekts definieren.
<b>UML-Klassendiagramm</b>	Grafische Darstellung einer Klasse mit ihren Attributen und Methoden.

### Vorteile von Klassen

- Modularität:** Erleichtert die Wartung und Erweiterung des Codes.
- Wiederverwendbarkeit:** Klassen können in verschiedenen Programmen genutzt werden.
- Datenkapselung:** Sorgt für besseren Schutz und Organisation der Daten.

Mit Klassen und Objekten wird die Programmierung in Java strukturierter, wiederverwendbarer und leichter wartbar.

# Die vier Grundpfeiler der objektorientierten Programmierung (OOP)

---

Die objektorientierte Programmierung basiert auf vier zentralen Konzepten:

## Abstraktion

- **Definition:** Abstraktion bedeutet, nur die wesentlichen Eigenschaften eines Objekts offenzulegen und unnötige Details zu verbergen.
- **Beispiel:** Ein Auto hat wesentliche Eigenschaften wie **Farbe**, **Geschwindigkeit** und **maximale Geschwindigkeit**, aber die interne Funktionsweise des Motors bleibt verborgen.

## Kapselung (Encapsulation)

- **Definition:** Kapselung bedeutet, dass die internen Details einer Klasse vor externem Zugriff geschützt werden.
- **Umsetzung:** Attribute (Daten) werden als **private** deklariert, und der Zugriff erfolgt über öffentliche Getter- und Setter-Methoden.

## Vererbung (Inheritance)

- **Definition:** Mit Vererbung können Klassen Eigenschaften und Methoden einer übergeordneten Klasse übernehmen.
- **Beispiel:** Ein **Elektroauto** kann von der allgemeinen Klasse **Auto** erben und zusätzliche Eigenschaften wie **Batteriestand** hinzufügen.

## Polymorphismus

- **Definition:** Polymorphismus ermöglicht es, dieselbe Methode in verschiedenen Kontexten unterschiedlich zu verwenden.
  - **Beispiel:** Ein Auto könnte eine **beschleunigen**-Methode haben, die unterschiedlich implementiert wird, je nachdem ob es ein **Elektroauto** oder ein **Benzinauto** ist.
- 

## Kapselung (Encapsulation) im Detail

Kapselung schützt die internen Details einer Klasse und macht sie für andere Teile des Codes unzugänglich. Stattdessen erfolgt der Zugriff nur über definierte Schnittstellen wie Getter und Setter.

---

## Problematik ohne Kapselung – Beispiel: Auto-Klasse

Wenn alle Attribute einer Klasse öffentlich (**public**) sind, können sie von externen Klassen direkt verändert werden, was problematisch ist.

Beispiel: Auto-Klasse ohne Kapselung

```

class Auto {
    public String farbe;
    public int maximaleGeschwindigkeit;
    public int momentaneGeschwindigkeit;
}

```

## Verwendung der Klasse

```

public class Main {
    public static void main(String[] args) {
        Auto auto = new Auto();

        // Direkte Änderung der Attribute
        auto.farbe = "Rot";
        auto.maximaleGeschwindigkeit = 150;
        auto.momentaneGeschwindigkeit = -50; // Problematisch!
    }
}

```

## Probleme ohne Kapselung

### 1. Ungültige Werte können gesetzt werden:

- `momentaneGeschwindigkeit` wird auf `-50` gesetzt, was physikalisch unmöglich ist.
- Es gibt keine Validierung, um solche Änderungen zu verhindern.

### 2. Kein Schutz vor inkorrekt Änderungen:

- Jemand könnte die `maximaleGeschwindigkeit` auf `0` setzen, was das Verhalten der Klasse bricht.

### 3. Unvorhersehbares Verhalten:

- Andere Teile des Codes könnten versehentlich Attribute ändern, was schwer nachvollziehbare Bugs verursacht.

### 4. Keine Kontrolle über die Daten:

- Es ist nicht möglich, zusätzliche Logik (z. B. Protokollierung) auszuführen, wenn sich ein Attribut ändert.

## Lösung: Kapselung mit privaten Attributen

Die Attribute der Klasse **Auto** werden als `private` deklariert und sind von außen nicht direkt zugänglich. Der Zugriff erfolgt ausschließlich über Getter- und Setter-Methoden.

## Beispiel: Auto-Klasse mit Kapselung

```

class Auto {
    private String farbe;
    private int maximaleGeschwindigkeit;
    private int momentaneGeschwindigkeit;

    // Getter und Setter für Farbe
    public String getFarbe() {
        return farbe;
    }

    public void setFarbe(String farbe) {
        this.farbe = farbe;
    }

    // Getter und Setter für maximale Geschwindigkeit
    public int getMaximaleGeschwindigkeit() {
        return maximaleGeschwindigkeit;
    }

    public void setMaximaleGeschwindigkeit(int maximaleGeschwindigkeit) {
        this.maximaleGeschwindigkeit = maximaleGeschwindigkeit;
    }

    // Getter und Setter für momentane Geschwindigkeit
    public int getMomentaneGeschwindigkeit() {
        return momentaneGeschwindigkeit;
    }

    public void setMomentaneGeschwindigkeit(int geschwindigkeit) {
        if (geschwindigkeit >= 0) {
            this.momentaneGeschwindigkeit = geschwindigkeit;
        } else {
            System.out.println("Fehler: Geschwindigkeit kann nicht negativ
sein!");
        }
    }
}

```

---

## Vorteile der Kapselung

### 1. Datenvalidierung:

- Setter können sicherstellen, dass nur gültige Werte gesetzt werden (z. B. keine negativen Geschwindigkeiten).

### 2. Bessere Wartbarkeit:

- Änderungen an der internen Implementierung betreffen nur die Klasse, solange die öffentlichen Getter- und Setter-Methoden gleich bleiben.

### 3. Zusätzliche Logik:

- Getter und Setter können zusätzliche Aktionen ausführen, wie z. B. Protokollierung oder Benachrichtigungen.

#### 4. Schutz vor versehentlichem Zugriff:

- Externe Klassen können Attribute nicht direkt manipulieren, was die Robustheit des Codes erhöht.

---

## Beispiel mit Kapselung

Verwenden der Auto-Klasse

```
public class Main {  
    public static void main(String[] args) {  
        Auto auto = new Auto();  
  
        // Werte mit Setter setzen  
        auto.setFarbe("Rot");  
        auto.setMaximaleGeschwindigkeit(150);  
        auto.setMomentaneGeschwindigkeit(100);  
  
        // Werte mit Getter abrufen  
        System.out.println("Farbe: " + auto.getFarbe());  
        System.out.println("Maximale Geschwindigkeit: " +  
        auto.getMaximaleGeschwindigkeit());  
        System.out.println("Momentane Geschwindigkeit: " +  
        auto.getMomentaneGeschwindigkeit());  
  
        // Ungültige Geschwindigkeit setzen  
        auto.setMomentaneGeschwindigkeit(-20); // Gibt eine Warnung aus  
    }  
}
```

---

## Fazit

Die Kapselung ist ein essenzielles Prinzip der OOP, das:

- **Daten schützt** und **Fehler verhindert**,
- die **Wartbarkeit** und **Erweiterbarkeit** des Codes verbessert,
- sicherstellt, dass nur **gültige Werte** den Attributen zugewiesen werden.

Durch private Attribute und kontrollierten Zugriff über Getter und Setter wird der Code robuster, sicherer und leichter verständlich.

# Konstruktoren in Java

---

Ein **Konstruktor** ist eine spezielle Methode in Java, die verwendet wird, um ein Objekt zu erstellen und zu initialisieren. Im Gegensatz zu normalen Methoden hat ein Konstruktor **keinen Rückgabewert** und **den gleichen Namen wie die Klasse**.

## Merkmale eines Konstruktors

- Wird automatisch aufgerufen, wenn ein Objekt erstellt wird.
  - Hat keinen Rückgabewert (auch kein `void`).
  - Kann überladen werden (mehrere Konstruktoren mit unterschiedlichen Parametern).
  - Wird verwendet, um Attribute eines Objekts während der Instanziierung zu initialisieren.
- 

## Warum Konstruktoren verwenden?

1. **Initialisierung von Attributen:** Konstruktoren ermöglichen es, Attributwerte direkt bei der Erstellung eines Objekts zu setzen.
  2. **Erzwingen von Werten:** Konstruktoren können sicherstellen, dass Objekte immer mit gültigen Werten erstellt werden.
  3. **Lesbarkeit und Effizienz:** Der Code wird klarer, wenn Objekte mit initialisierten Attributen erstellt werden.
- 

## Syntax eines Konstruktors

```
class Klassenname {  
    // Konstruktor ohne Parameter  
    Klassenname() {  
        // Initialisierung  
    }  
  
    // Konstruktor mit Parametern  
    Klassenname(Datentyp parameter1, Datentyp parameter2) {  
        // Initialisierung  
    }  
}
```

## Beispiel: Konstruktor in der `Auto`-Klasse

Hier wird die `Auto`-Klasse mit einem Konstruktor erweitert, der Attribute wie `farbe`, `maximaleGeschwindigkeit` und `momentaneGeschwindigkeit` initialisiert.

```
class Auto {  
    // Attribute
```

```

String farbe;
int maximaleGeschwindigkeit;
int momentaneGeschwindigkeit;

// Konstruktor
Auto(String farbe, int maximaleGeschwindigkeit, int momentaneGeschwindigkeit)
{
    this.farbe = farbe;
    this.maximaleGeschwindigkeit = maximaleGeschwindigkeit;
    this.momentaneGeschwindigkeit = momentaneGeschwindigkeit;
}

// Beispieldaten
void beschleunigen(int geschwindigkeit) {
    momentaneGeschwindigkeit += geschwindigkeit;
}

void bremsen(int geschwindigkeit) {
    momentaneGeschwindigkeit -= geschwindigkeit;
    if (momentaneGeschwindigkeit < 0) {
        momentaneGeschwindigkeit = 0;
    }
}
}

```

## Verwendung eines Konstruktors

Der Konstruktor wird aufgerufen, wenn ein Objekt mit `new` erstellt wird. Hier werden wir Objekte der `Auto`-Klasse mit dem Konstruktor initialisieren:

```

public class Main {
    public static void main(String[] args) {
        // Erstellen von Objekten mit Konstruktor
        Auto rotesAuto = new Auto("Rot", 150, 0);
        Auto gelbesAuto = new Auto("Gelb", 120, 10);

        // Zugriff auf Attribute
        System.out.println("Rotes Auto - Farbe: " + rotesAuto.farbe + ", Maximale
Geschwindigkeit: " + rotesAuto.maximaleGeschwindigkeit);
        System.out.println("Gelbes Auto - Farbe: " + gelbesAuto.farbe + ", "
Momentane Geschwindigkeit: " + gelbesAuto.momentaneGeschwindigkeit);

        // Methodenaufruf
        rotesAuto.beschleunigen(20);
        System.out.println("Rotes Auto - Momentane Geschwindigkeit nach
Beschleunigung: " + rotesAuto.momentaneGeschwindigkeit);
    }
}

```

# Überladene Konstruktoren

In Java können wir mehrere Konstruktoren mit unterschiedlichen Parametern definieren. Dies ist nützlich, wenn wir Objekte mit unterschiedlichen Anfangswerten erstellen möchten.

```
class Auto {  
    String farbe;  
    int maximaleGeschwindigkeit;  
    int momentaneGeschwindigkeit;  
  
    // Konstruktor mit allen Attributen  
    Auto(String farbe, int maximaleGeschwindigkeit, int momentaneGeschwindigkeit)  
{  
        this.farbe = farbe;  
        this.maximaleGeschwindigkeit = maximaleGeschwindigkeit;  
        this.momentaneGeschwindigkeit = momentaneGeschwindigkeit;  
    }  
  
    // Konstruktor mit Standardwerten  
    Auto(String farbe, int maximaleGeschwindigkeit) {  
        this(farbe, maximaleGeschwindigkeit, 0); // Ruft den anderen Konstruktor  
        auf  
    }  
  
    // Beispielmethoden  
    void beschleunigen(int geschwindigkeit) {  
        momentaneGeschwindigkeit += geschwindigkeit;  
    }  
}
```

Verwendung der überladenen Konstruktoren:

```
public class Main {  
    public static void main(String[] args) {  
        // Objekt mit vollständigem Konstruktor  
        Auto blauesAuto = new Auto("Blau", 130, 50);  
  
        // Objekt mit Standardwerten  
        Auto gruensAuto = new Auto("Grün", 110);  
  
        System.out.println("Blaues Auto - Farbe: " + blauesAuto.farbe + ",  
        Momentane Geschwindigkeit: " + blauesAuto.momentaneGeschwindigkeit);  
        System.out.println("Grünes Auto - Farbe: " + gruensAuto.farbe + ",  
        Momentane Geschwindigkeit: " + gruensAuto.momentaneGeschwindigkeit);  
    }  
}
```

## Zusammenfassung

Merkmal	Beschreibung
<b>Konstruktor</b>	Eine spezielle Methode, die ein Objekt initialisiert und denselben Namen wie die Klasse hat.
<b>Kein Rückgabewert</b>	Konstruktoren haben keinen Rückgabewert, auch kein <code>void</code> .
<b>Automatischer Aufruf</b>	Wird automatisch bei der Erstellung eines Objekts mit <code>new</code> aufgerufen.
<b>Überladung</b>	Mehrere Konstruktoren können definiert werden, um verschiedene Initialisierungen zu ermöglichen.

## Vorteile der Konstruktoren

- Saubere Initialisierung:** Attribute können direkt beim Erstellen des Objekts gesetzt werden.
- Flexibilität:** Überladene Konstruktoren ermöglichen unterschiedliche Arten von Initialisierungen.
- Fehlervermeidung:** Konstruktoren stellen sicher, dass ein Objekt immer in einem gültigen Zustand erstellt wird.

# Vererbung und Kapselung in Java - Erklärungen mit Auto-Beispiel

## Kapselung in Java

Kapselung ist ein zentrales Prinzip der objektorientierten Programmierung, das die Daten einer Klasse schützt, indem die Felder einer Klasse als **private** deklariert werden. Der Zugriff auf diese Felder erfolgt ausschließlich über **Getter- und Setter-Methoden**, die als öffentliche Schnittstelle bereitgestellt werden.

### Vorteile der Kapselung:

- Kontrollierter Zugriff auf Daten.
- Verhindert ungewollte Änderungen an den internen Zuständen.
- Erleichtert spätere Änderungen, da die interne Implementierung unabhängig von der öffentlichen Schnittstelle geändert werden kann.

## Vererbung in Java

Vererbung ermöglicht es einer Klasse (Unterklassen), die Eigenschaften und Methoden einer anderen Klasse (Oberklasse) zu übernehmen. Durch Vererbung können Unterklassen vorhandenes Verhalten wiederverwenden und erweitern, ohne die Oberklasse zu verändern.

### Vorteile der Vererbung:

- Wiederverwendbarkeit von Code.
- Ermöglicht die Spezialisierung durch Hinzufügen neuer Funktionen in Unterklassen.
- Fördert eine übersichtliche Klassenhierarchie.

---

## Beispiel: Vererbung und Kapselung kombiniert

Das folgende Beispiel zeigt die Kapselung und Vererbung anhand der Klassen **Auto** (Oberklasse) und **Sportwagen** (Unterklasse).

```
// Oberklasse Auto mit Kapselung
public class Auto {
    // Private Felder (Kapselung)
    private String farbe;
    private int maximaleGeschwindigkeit;
    private int momentGeschwindigkeit;

    // Konstruktor
    public Auto(String farbe, int maximaleGeschwindigkeit) {
        this.farbe = farbe;
        this.maximaleGeschwindigkeit = maximaleGeschwindigkeit;
        this.momentGeschwindigkeit = 0;
    }

    // Getter und Setter (Kapselung)
    public String getFarbe() {
        return farbe;
    }
}
```

```

}

public void setFarbe(String farbe) {
    this.farbe = farbe;
}

public int getMaximaleGeschwindigkeit() {
    return maximaleGeschwindigkeit;
}

public void setMaximaleGeschwindigkeit(int maximaleGeschwindigkeit) {
    this.maximaleGeschwindigkeit = maximaleGeschwindigkeit;
}

public int getMomentGeschwindigkeit() {
    return momentGeschwindigkeit;
}

public void setMomentGeschwindigkeit(int momentGeschwindigkeit) {
    if (momentGeschwindigkeit <= maximaleGeschwindigkeit &&
momentGeschwindigkeit >= 0) {
        this.momentGeschwindigkeit = momentGeschwindigkeit;
    } else {
        System.out.println("Ungültige Geschwindigkeit!");
    }
}

// Methode zum Beschleunigen
public void beschleunigen(int wert) {
    setMomentGeschwindigkeit(this.momentGeschwindigkeit + wert);
}

// Methode zum Bremsen
public void bremsen(int wert) {
    setMomentGeschwindigkeit(this.momentGeschwindigkeit - wert);
}
}

```

## Unterklasse Sportwagen mit zusätzlicher Funktionalität

```

// Unterklasse Sportwagen, erbt von Auto
public class Sportwagen extends Auto {
    private boolean turboModus; // Eigenschaft für den Turbo-Modus

    // Konstruktor
    public Sportwagen(String farbe, int maximaleGeschwindigkeit) {
        super(farbe, maximaleGeschwindigkeit); // Konstruktor der Oberklasse wird
aufgerufen
        this.turboModus = false;
    }
}

```

```

// Methode zum Aktivieren des Turbo-Modus
public void aktiviereTurbo() {
    this.turboModus = true;
    System.out.println("Turbo-Modus aktiviert!");
}

// Methode zum Deaktivieren des Turbo-Modus
public void deaktiviereTurbo() {
    this.turboModus = false;
    System.out.println("Turbo-Modus deaktiviert!");
}

// Überschreiben der Methode beschleunigen
@Override
public void beschleunigen(int wert) {
    if (turboModus) {
        wert *= 2; // Verdoppelt die Beschleunigung im Turbo-Modus
    }
    super.beschleunigen(wert); // Ruft die beschleunigen-Methode der Oberklasse
auf
}
}

```

## Hauptprogramm zur Demonstration

```

public class Hauptprogramm {
    public static void main(String[] args) {
        // Erstellen eines normalen Autos
        Auto auto = new Auto("Rot", 150);
        auto.beschleunigen(50);
        System.out.println("Normales Auto Geschwindigkeit: " +
auto.getMomentGeschwindigkeit();

        // Erstellen eines Sportwagens
        Sportwagen sportwagen = new Sportwagen("Blau", 250);
        sportwagen.beschleunigen(50);
        System.out.println("Sportwagen Geschwindigkeit: " +
sportwagen.getMomentGeschwindigkeit());

        // Turbo-Modus aktivieren und erneut beschleunigen
        sportwagen.aktiviereTurbo();
        sportwagen.beschleunigen(30);
        System.out.println("Sportwagen Geschwindigkeit nach Turbo: " +
sportwagen.getMomentGeschwindigkeit());
    }
}

```

---

## Erklärung

## 1. Kapselung:

- Die Felder `farbe`, `maximaleGeschwindigkeit` und `momentGeschwindigkeit` der Klasse `Auto` sind als `private` deklariert.
- Der Zugriff erfolgt ausschließlich über Getter- und Setter-Methoden, wodurch kontrolliert wird, dass nur gültige Werte zugewiesen werden.

## 2. Vererbung:

- Die Klasse `Sportwagen` erbt von `Auto` mit dem Schlüsselwort `extends`.
- `Sportwagen` fügt die Eigenschaft `turboModus` und Methoden zur Steuerung des Turbo-Modus hinzu.
- Die Methode `beschleunigen` wird in der Unterklasse überschrieben, um das Verhalten im Turbo-Modus zu ändern.

## 3. Polymorphismus:

- Ein `Sportwagen` kann wie ein `Auto` behandelt werden, da er alle Eigenschaften der Oberklasse `Auto` erbt. Gleichzeitig erweitert er das Verhalten durch Turbo-spezifische Funktionen.
- 

Vorteile der Kombination von Kapselung und Vererbung

- **Sichere Daten:** Durch Kapselung bleiben die Felder geschützt und nur über definierte Methoden zugänglich.
- **Wiederverwendbarkeit:** Gemeinsame Funktionen werden in der Oberklasse definiert, und die Unterklassen spezialisieren diese.
- **Flexibilität:** Änderungen in der Oberklasse wirken sich automatisch auf die Unterklassen aus, was den Wartungsaufwand reduziert.

# Handout: Java Klassen, Packages und Package Naming

---

## 1. Was ist eine Klasse in Java?

Eine **Klasse** in Java ist ein Bauplan für Objekte. Sie enthält Eigenschaften (Variablen) und Verhalten (Methoden), die Objekte dieser Klasse besitzen können.

Beispiel einer Auto-Klasse

```
public class Auto {  
    private String marke;  
  
    public Auto(String marke) {  
        this.marke = marke;  
    }  
  
    public void start() {  
        System.out.println(marke + " startet.");  
    }  
}
```

---

## 2. Was ist ein Package in Java?

Ein **Package** ist eine Sammlung von Klassen und Interfaces, die zusammengehören. Es wird verwendet, um Code zu organisieren und Namenskonflikte zu vermeiden.

Warum Packages verwenden?

- **Bessere Organisation:** Gruppiert verwandte Klassen.
- **Namenskonflikte vermeiden:** Klassen in verschiedenen Packages können denselben Namen haben.

Beispiel eines Packages mit einer Auto-Klasse

**Dateistuktur:**

**Datei:** com/vehicles/Auto.java

```
package com.vehicles;  
  
public class Auto {  
    public void start() {  
        System.out.println("Das Auto startet.");  
    }  
}
```

Datei: com/vehicles/Main.java

```
package com.vehicles;

public class Main {
    public static void main(String[] args) {
        Auto auto = new Auto();
        auto.start();
    }
}
```

## Import von Packages

Wenn Klassen aus einem anderen Package verwendet werden sollen, müssen sie importiert werden.

```
import com.vehicles.Auto;

public class Main {
    public static void main(String[] args) {
        Auto auto = new Auto();
        auto.start();
    }
}
```

---

# Package Naming

---

## Konvention zur Benennung von Packages

Die Verwendung von **com.package** (z. B. **com.vehicles**) als Teil von Java-Packagenamen folgt einer bewährten Konvention für die **Organisation und Strukturierung** von Projekten. Diese Konvention wird als **Reverse-Domain-Naming-Konvention** bezeichnet.

---

## Gründe für die Verwendung von **com.package**

### 1. Eindeutigkeit der Packagenamen

Durch die Verwendung von Domains (z. B. **com**, **org**, **net**) wird sichergestellt, dass der Paketname eindeutig ist.

#### Beispiel:

- Zwei Entwickler erstellen eine Klasse namens **Auto**:
    - Paket 1: **com.example.vehicles.Auto**
    - Paket 2: **org.myproject.vehicles.Auto**
  - Es gibt **keinen Konflikt**, da die Packagenamen eindeutig sind.
-

## 2. Strukturierung von Projekten

- **Hierarchische Organisation:**
  - Der Domain-Teil (`com`, `org`) steht oben in der Hierarchie.
  - Danach folgt der spezifische Projekt- oder Modulname (z. B. `example`, `vehicles`).

### Beispiel: Projektstruktur für ein Autoprojekt

```
src/
└── com/
    └── example/
        ├── vehicles/
        │   ├── Auto.java
        │   ├── Werkstatt.java
        └── utils/
            └── Helper.java
```

Diese Struktur:

- Organisiert Code nach Funktionalität oder Verantwortlichkeiten.
- Erleichtert die Navigation in größeren Projekten.

---

## 3. Domain-Referenz

- **Konvention:** Der Domainname des Unternehmens oder Projekts wird verwendet, aber in umgekehrter Reihenfolge.
- **Beispiel:**
  - Firma mit Domain `example.com` → Package-Präfix: `com.example`.

Dies stellt eine Verbindung zwischen dem Projekt und seiner Organisation her und macht den Ursprung des Codes deutlich.

---

## 4. Vermeidung von Namenskonflikten

- **Ohne Domain-Namen:**
  - Wenn zwei Entwickler denselben Paketnamen verwenden, z. B. `vehicles`, könnten Konflikte auftreten.
- **Mit Domain-Namen:**
  - Die Pakete `com.example.vehicles` und `org.myproject.vehicles` sind **eindeutig** und können unabhängig existieren.

---

## 5. Best Practice und Standard in Java

- **Java Naming Conventions:**
  - Java-Package-Namen folgen der **Reverse-Domain-Naming-Konvention**.
- **Framework-Unterstützung:**

- Frameworks wie **Spring** und **Maven** setzen diese Konvention voraus.
- 

## Wie wählt man einen Packagenamen aus?

### 1. Eigene Domain nutzen

- Wenn Sie ein Projekt für Ihre Firma erstellen, verwenden Sie den Firmen-Domainnamen.

#### Beispiel:

- Firma mit Domain `example.com` → Package-Präfix: `com.example`.

### 2. Kein Domainname?

- Nutzen Sie folgende Präfixe:
    - `org` für Organisationen.
    - `net` für Netzwerke.
    - `app` oder `my` für persönliche Projekte.
- 

## Beispiele

### Firma mit Domain: `vehicles.com`

- **Packagenamen:**
  - `com.vehicles.cars` (für Fahrzeugklassen)
  - `com.vehicles.utils` (für Hilfsklassen)

### Persönliches Projekt

- **Packagenamen:**
    - `app.myproject.main`
    - `app.myproject.utils`
- 

## Fazit

Die Verwendung von `com.package`:

1. **Erhöht die Eindeutigkeit** und verhindert Konflikte.
2. **Organisiert den Code** in einer hierarchischen Struktur.
3. **Ermöglicht Zusammenarbeit**, da verschiedene Teams oder Entwickler ihre Pakete klar abgrenzen können.

Auch wenn es nicht zwingend erforderlich ist, ist diese Konvention eine bewährte Praxis, die Professionalität und Ordnung in Java-Projekten fördert.

---

## Praktisches Beispiel

### Dateistruktur für ein Beispielprojekt:

```
src/
└── com/
    └── vehicles/
        ├── Auto.java
        └── Main.java
    └── utils/
        └── Helper.java
```

#### Code für Auto.java:

```
package com.vehicles;

public class Auto {
    public void start() {
        System.out.println("Das Auto startet.");
    }
}
```

#### Code für Main.java:

```
package com.vehicles;

public class Main {
    public static void main(String[] args) {
        Auto auto = new Auto();
        auto.start();
    }
}
```

# Sichtbarkeiten in Java: **private**, **public**, **protected** und **Default**

---

In Java werden Zugriffssichtbarkeiten (Access Modifiers) verwendet, um zu kontrollieren, wo und wie auf Klassen, Methoden, Variablen oder Konstruktoren zugegriffen werden kann. Sie spielen eine zentrale Rolle beim **Encapsulation**-Prinzip (Datenkapselung).

---

## 1. Definition der Sichtbarkeiten

### 1. **public**

- **Definition:** Mitglieder oder Klassen mit **public** sind von überall im Programm zugänglich.
- **Vorteile:**
  - Ermöglicht den Zugriff auf Klassen oder Methoden aus beliebigen Packages.
  - Ideal für Methoden oder Klassen, die eine öffentliche API darstellen.
- **Beispiel:**

```
public class Auto {  
    public String marke; // Von überall zugänglich  
  
    public void fahren() {  
        System.out.println(marke + " fährt.");  
    }  
}
```

---

### 2. **private**

- **Definition:** Mitglieder oder Methoden mit **private** sind nur innerhalb der gleichen Klasse zugänglich.
- **Vorteile:**
  - Schützt sensible Daten vor unbefugtem Zugriff.
  - Erzwingt, dass der Zugriff auf die Daten nur über kontrollierte Methoden (z. B. Getter/Setter) erfolgt.
- **Beispiel:**

```
public class Auto {  
    private String motorTyp = "V8"; // Nur innerhalb der Klasse zugänglich  
  
    private void zeigeMotor() {  
        System.out.println("Motor: " + motorTyp);  
    }  
  
    public void start() {  
        zeigeMotor(); // Zugriff innerhalb der Klasse erlaubt  
    }  
}
```

```
        System.out.println("Das Auto startet.");
    }
}
```

---

### 3. protected

- **Definition:** Mitglieder mit **protected** sind:
  - Zugänglich innerhalb der gleichen Klasse.
  - Zugänglich für Unterklassen (auch in anderen Packages).
  - Zugänglich für Klassen im gleichen Package.
- **Vorteile:**
  - Ermöglicht den Zugriff auf wichtige Funktionalitäten in Unterklassen.
  - Bietet einen Mittelweg zwischen **private** und **public**.
- **Beispiel:**

```
package com.vehicles;

public class Auto {
    protected int baujahr; // Zugänglich in Unterklassen und im gleichen Package

    protected void reparieren() {
        System.out.println("Das Auto wird repariert.");
    }
}
```

Verwendung in einer Unterklasse:

```
package com.garage;

import com.vehicles.Auto;

public class Werkstatt extends Auto {
    public void wartung() {
        reparieren(); // Zugriff auf geschützte Methode
        System.out.println("Wartung abgeschlossen.");
    }
}
```

---

### 4. Default (ohne Modifikator)

- **Definition:** Wenn kein Modifikator angegeben wird, ist der Zugriff nur innerhalb des gleichen Packages möglich.
- **Vorteile:**

- Nützlich für Klassen oder Methoden, die nur für andere Klassen im gleichen Package sichtbar sein sollen.
- **Beispiel:**

```
class Motor {
    void starten() {
        System.out.println("Der Motor startet.");
    }
}
```

## 2. Vorteile und Verwendung von Getter und Setter

Getter und Setter sind Methoden, die den Zugriff auf **private** Variablen kontrollieren. Sie bieten eine kontrollierte Möglichkeit, auf Eigenschaften einer Klasse zuzugreifen oder sie zu ändern.

### **Warum Getter und Setter verwenden?**

1. **Datenkapselung:** Direkter Zugriff auf private Variablen wird verhindert.
2. **Validierung:** Setter können Eingabewerte validieren, bevor sie einer Variablen zugewiesen werden.
3. **Flexibilität:** Getter und Setter ermöglichen Änderungen in der Implementierung, ohne die öffentliche API zu verändern.

### **Beispiel: Getter und Setter**

```
public class Auto {
    private String marke; // Private Variable

    // Getter für 'marke'
    public String getMarke() {
        return marke;
    }

    // Setter für 'marke'
    public void setMarke(String marke) {
        if (marke != null && !marke.isEmpty()) { // Validierung
            this.marke = marke;
        } else {
            System.out.println("Ungültige Marke.");
        }
    }

    public void anzeigen() {
        System.out.println("Auto-Marke: " + marke);
    }
}
```

### **Verwendung der Getter und Setter:**

```

public class Main {
    public static void main(String[] args) {
        Auto auto = new Auto();
        auto.setMarke("BMW"); // Setter verwenden
        System.out.println("Marke: " + auto.getMarke()); // Getter verwenden
    }
}

```

### 3. Zusammenfassung der Zugriffssichtbarkeiten

Modifikator	Gleiche Klasse	Gleiches Package	Unterklassen (anderes Package)	Überall
private	✓	X	X	X
Default	✓	✓	X	X
protected	✓	✓	✓	X
public	✓	✓	✓	✓

### 4. Praxisbeispiel: Anwendung aller Sichtbarkeiten

```

package com.vehicles;

public class Auto {
    private String motorTyp = "V8"; // Nur innerhalb der Klasse zugänglich
    protected int baujahr; // Zugänglich in Unterklassen und im gleichen
    Package
    public String marke; // Von überall zugänglich

    // Getter und Setter für die private Variable
    public String getMotorTyp() {
        return motorTyp;
    }

    public void setMotorTyp(String motorTyp) {
        this.motorTyp = motorTyp;
    }

    protected void reparieren() {
        System.out.println("Das Auto wird repariert.");
    }

    public void starten() {
        System.out.println(marke + " startet.");
    }
}

```

Verwendung in einer Unterklasse:

```
package com.garage;

import com.vehicles.Auto;

public class Werkstatt extends Auto {
    public Werkstatt(String marke, int baujahr) {
        this.marke = marke; // Zugriff auf `public` Feld
        this.baujahr = baujahr; // Zugriff auf `protected` Feld
    }

    public void wartung() {
        reparieren(); // Zugriff auf geschützte Methode
        System.out.println("Wartung für " + marke + " abgeschlossen.");
    }
}
```

---

## Fazit

- **private**: Maximale Kapselung, Zugriff nur innerhalb der Klasse.
- **protected**: Ermöglicht Zugriff für Unterklassen und im gleichen Package.
- **public**: Zugriff von überall, für öffentliche Schnittstellen geeignet.
- **Default**: Zugriff nur innerhalb des gleichen Packages.

Getter und Setter helfen, Daten sicher zu kapseln und bieten eine flexible Kontrolle über die Eigenschaften einer Klasse.

# Vergleich der Zugriffssichtbarkeiten in Java

---

## Sichtbarkeitstabelle

Sichtbarkeit/Modifikator	public	protected	default	private
<b>Innerhalb der Klasse</b>	✓ Ja	✓ Ja	✓ Ja	✓ Ja
<b>Im gleichen Package</b>	✓ Ja	✓ Ja	✓ Ja	✗ Nein
<b>In Unterklassen (anderes Package)</b>	✓ Ja	✓ Ja	✗ Nein	✗ Nein
<b>Außerhalb des Packages</b>	✓ Ja	✗ Nein	✗ Nein	✗ Nein

---

Vorteile:

- **public**: Maximale Sichtbarkeit, ideal für öffentliche APIs.
  - **protected**: Zugang für Unterklassen, aber nicht vollständig öffentlich.
  - **default**: Beschränkung auf das gleiche Package, verhindert ungewollten Zugriff von außen.
  - **private**: Maximale Kapselung, Schutz vor externem Zugriff.
- 

## Codebeispiele

### 1. public, protected, default und private in der Klasse Auto

Datei: com/fahrzeuge/Auto.java

```
package com.fahrzeuge;

public class Auto {
    private String motorTyp = "V8";      // Nur innerhalb der Klasse zugänglich
    protected int baujahr;                // Zugänglich in Unterklassen und im gleichen
                                         // Package
    public String marke;                 // Von überall zugänglich
    String message = "Hallo";           // Default-Sichtbarkeit, zugänglich im
                                         // gleichen Package

    // Getter und Setter für die private Variable
    public String getMotorTyp() {
        return motorTyp;
    }

    public void setMotorTyp(String motorTyp) {
        this.motorTyp = motorTyp;
    }

    protected void reparieren() {
        System.out.println("Das Auto wird repariert.");
    }
}
```

```
    public void starten() {
        System.out.println(marke + " startet.");
    }
}
```

---

## 2. Zugriff und Vererbung in der Klasse Werkstatt

Datei: com/garage/Werkstatt.java

```
package com.garage;

import com.fahrzeuge.Auto;

public class Werkstatt extends Auto {
    public Werkstatt(String marke, int baujahr) {
        this.marke = marke; // Zugriff auf `public` Feld
        this.baujahr = baujahr; // Zugriff auf `protected` Feld
    }

    public void wartung() {
        reparieren(); // Zugriff auf geschützte Methode
        System.out.println("Wartung für " + marke + " abgeschlossen.");
    }
}
```

---

## 3. Zugriff in der Hauptklasse Main

Datei: com/garage/Main.java

```
package com.garage;

import com.fahrzeuge.Auto;

public class Main {
    public static void main(String[] args) {
        Auto auto = new Auto();
        auto.setMotorTyp("Elektro"); // Setter verwenden
        System.out.println("Motor-Typ: " + auto.getMotorTyp()); // Getter
verwenden

        auto.marke = "Tesla"; // Direkter Zugriff auf `public`
        auto.starten(); // Aufruf einer öffentlichen Methode
    }
}
```

## 4. Zusammenfassung

- **public Felder und Methoden:** Zugänglich von überall, auch außerhalb des Packages.
- **protected Felder und Methoden:** Zugänglich innerhalb der gleichen Klasse, im gleichen Package und in Unterklassen (auch in anderen Packages).
- **default Felder und Methoden:** Zugänglich nur innerhalb des gleichen Packages.
- **private Felder und Methoden:** Zugänglich nur innerhalb der gleichen Klasse.

# Handout: Abstrakte Klassen in Java

---

## Definition

Eine **abstrakte Klasse** in Java ist eine Klasse, die mit dem Schlüsselwort **abstract** deklariert wird. Sie kann Methoden enthalten, die entweder:

- **abstrakt** sind (keine Implementierung, nur Methodensignatur), oder
- **konkret** sind (mit Implementierung).

Abstrakte Klassen können nicht direkt instanziert werden. Sie dienen als **Basisklassen**, die von anderen Klassen geerbt werden müssen.

---

## Merkmale einer abstrakten Klasse

1. Kann sowohl **abstrakte Methoden** (ohne Implementierung) als auch **konkrete Methoden** (mit Implementierung) enthalten.
  2. Kann **Konstruktoren, Variablen** und **statische Methoden** enthalten.
  3. Vererbt gemeinsame Funktionalitäten an Unterklassen, zwingt aber die Unterklassen zur Implementierung spezifischer Methoden.
  4. Kann nicht direkt instanziert werden.
- 

## Vorteile abstrakter Klassen

- **Wiederverwendbarkeit:** Gemeinsame Logik kann in der abstrakten Klasse implementiert und von Unterklassen geerbt werden.
  - **Strukturierung:** Erzeugt eine klare Struktur für die Vererbungshierarchie.
  - **Flexibilität:** Unterklassen können spezifische Implementierungen der abstrakten Methoden bereitstellen.
  - **Teilweise Implementierung:** Ermöglicht die Kombination aus vorgefertigtem Code und Methoden, die von Unterklassen implementiert werden müssen.
- 

## Beispiel: Abstrakte Klasse mit Autos

### 1. Abstrakte Klasse **Auto**

```
package com.fahrzeuge;

public abstract class Auto {
    protected String marke; // Geschützt: zugänglich in Unterklassen
    protected int baujahr;

    public Auto(String marke, int baujahr) {
        this.marke = marke;
        this.baujahr = baujahr;
```

```

    }

    // Abstrakte Methode: Muss in Unterklassen implementiert werden
    public abstract void antriebStarten();

    // Konkrete Methode: Kann direkt von Unterklassen geerbt werden
    public void anzeigen() {
        System.out.println("Marke: " + marke + ", Baujahr: " + baujahr);
    }
}

```

## 2. Konkrete Unterklassen

### Klasse ElektroAuto

```

package com.fahrzeuge;

public class ElektroAuto extends Auto {
    private int batteriekapazitaet; // In kWh

    public ElektroAuto(String marke, int baujahr, int batteriekapazitaet) {
        super(marke, baujahr);
        this.batteriekapazitaet = batteriekapazitaet;
    }

    @Override
    public void antriebStarten() {
        System.out.println(marke + " startet den Elektromotor mit einer
Batteriekapazität von " + batteriekapazitaet + " kWh.");
    }
}

```

### Klasse VerbrennungsAuto

```

package com.fahrzeuge;

public class VerbrennungsAuto extends Auto {
    private String kraftstofftyp; // z. B. Benzin oder Diesel

    public VerbrennungsAuto(String marke, int baujahr, String kraftstofftyp) {
        super(marke, baujahr);
        this.kraftstofftyp = kraftstofftyp;
    }

    @Override
    public void antriebStarten() {
        System.out.println(marke + " startet den Verbrennungsmotor mit " +
}

```

```
kraftstofftyp + ".");  
}  
}
```

---

### 3. Hauptklasse

Datei: com/main/Main.java

```
package com.main;  
  
import com.fahrzeuge.*;  
  
public class Main {  
    public static void main(String[] args) {  
        Auto elektroAuto = new ElektroAuto("Tesla", 2022, 75);  
        Auto verbrennungsAuto = new VerbrennungsAuto("BMW", 2020, "Benzin");  
  
        elektroAuto.anzeigen();  
        elektroAuto.antriebStarten();  
  
        verbrennungsAuto.anzeigen();  
        verbrennungsAuto.antriebStarten();  
    }  
}
```

---

## Diagramm der Vererbungshierarchie

```
Abstrakte Klasse: Auto  
|  
+--- Konkrete Klasse: ElektroAuto  
|      - Eigenschaft: batteriekapazitaet (int)  
|      - Methode: antriebStarten() [implementiert]  
|  
+--- Konkrete Klasse: VerbrennungsAuto  
      - Eigenschaft: kraftstofftyp (String)  
      - Methode: antriebStarten() [implementiert]
```

---

## Zusammenfassung

- Abstrakte Klassen sind ideal, um eine gemeinsame Basis für mehrere verwandte Klassen zu schaffen.
- **Abstrakte Methoden** definieren Verhalten, das von Unterklassen implementiert werden muss.
- **Konkrete Methoden** ermöglichen die Wiederverwendung von Code in allen Unterklassen.
- Das Beispiel mit Autos zeigt, wie Sie eine klare Struktur und Wiederverwendbarkeit in Ihrem Code erreichen können.



# Handout: Finale Klassen in Java

---

## Definition

Eine **finale Klasse** in Java ist eine Klasse, die mit dem Schlüsselwort **final** deklariert wird. Eine finale Klasse kann **nicht vererbt** werden. Alle Methoden und Eigenschaften der Klasse sind für die Vererbung gesperrt, aber die Klasse selbst kann instanziert werden.

---

## Merkmale einer finalen Klasse

- Keine Vererbung möglich:** Eine finale Klasse kann nicht als Basisklasse dienen.
  - Methoden und Eigenschaften:** Können wie in normalen Klassen verwendet werden, sind jedoch für die Vererbung gesperrt.
  - Instanzierbar:** Finale Klassen können wie normale Klassen instanziert werden.
- 

## Vorteile einer finalen Klasse

- Sicherheit:** Verhindert, dass die Klasse überschrieben oder manipuliert wird.
  - Effizienz:** Der Compiler kann optimierte Implementierungen vornehmen, da er sicher ist, dass die Klasse nicht erweitert wird.
  - Klarheit:** Signalisiert eindeutig, dass eine Klasse vollständig definiert ist und nicht verändert werden soll.
- 

## Beispiel: Finale Klasse mit Autos

### 1. Finale Klasse Auto

```
package com.fahrzeuge;

public final class Auto {
    private final String marke; // Konstante: Wert kann nicht geändert werden
    private final int baujahr;

    public Auto(String marke, int baujahr) {
        this.marke = marke;
        this.baujahr = baujahr;
    }

    public String getMarke() {
        return marke;
    }

    public int getBaujahr() {
        return baujahr;
    }
}
```

```

    public void starten() {
        System.out.println(marke + " startet. Baujahr: " + baujahr);
    }
}

```

## 2. Versuch der Vererbung (nicht erlaubt)

**Datei:** com/fahrzeuge/Sportwagen.java

```

package com.fahrzeuge;

// Fehler: Die Klasse Auto ist final und kann nicht erweitert werden
public class Sportwagen extends Auto {
    public Sportwagen(String marke, int baujahr) {
        super(marke, baujahr);
    }
}

```

**Fehler:** Cannot inherit from final class 'Auto'

## 3. Verwendung der finalen Klasse

**Datei:** com/main/Main.java

```

package com.main;

import com.fahrzeuge.Auto;

public class Main {
    public static void main(String[] args) {
        Auto auto = new Auto("Tesla", 2022);
        System.out.println("Marke: " + auto.getMarke());
        System.out.println("Baujahr: " + auto.getBaujahr());
        auto.starten();
    }
}

```

## Diagramm der Klassennutzung

Finale Klasse: Auto

- Eigenschaft: marke (final, String)
- Eigenschaft: baujahr (final, int)
- Methode: starten() [implementiert]

## Zusammenfassung

- **Finale Klassen** verhindern die Vererbung und schützen damit die Struktur und Logik der Klasse.
- Sie sind ideal, wenn eine Klasse vollständig definiert ist und keine Erweiterung oder Anpassung durch Unterklassen erlaubt sein soll.
- Das Beispiel mit der finalen Klasse `Auto` zeigt, wie eine Klasse als abgeschlossene Einheit verwendet werden kann.

# Handout: Einführung in OOP mit Java, Interfaces und Adapterklassen

---

## Einführung in die Objektorientierte Programmierung (OOP)

Objektorientierte Programmierung (OOP) ist ein Programmierparadigma, das auf der Struktur von Klassen und Objekten basiert. Es umfasst vier Hauptprinzipien:

1. **Abstraktion:** Vereinfachung komplexer Systeme durch die Reduzierung auf wesentliche Merkmale.
  2. **Kapselung:** Verbergen der Implementierungsdetails eines Objekts.
  3. **Vererbung:** Weitergabe von Eigenschaften und Methoden an Unterklassen.
  4. **Polymorphismus:** Nutzung der gleichen Schnittstelle für unterschiedliche Datentypen.
- 

## Klassenprinzip und Vererbung in Java

In Java ist eine Klasse eine Blaupause für Objekte. Vererbung ermöglicht es, eine Klasse von einer anderen abzuleiten.

Beispiel:

```
// Basisklasse
class Fahrzeug {
    String typ;

    public Fahrzeug(String typ) {
        this.typ = typ;
    }

    public void fahren() {
        System.out.println(typ + " fährt.");
    }
}

// Unterklasse
class Auto extends Fahrzeug {
    public Auto() {
        super("Auto");
    }
}
```

---

## Nachteile und Probleme der Vererbung

1. **Eingeschränkte Mehrfachvererbung:** Java erlaubt keine Mehrfachvererbung von Klassen.
2. **Enger Zusammenhang:** Änderungen in der Oberklasse können Unterklassen beeinträchtigen.
3. **Wiederverwendbarkeit:** Vererbung kann zu starren Abhängigkeiten führen.

## Beispiel: Warum Mehrfachvererbung problematisch ist

```
// In Java nicht möglich:  
class Elektroauto extends Auto, Elektrisch {  
    // Compilerfehler: Mehrfachvererbung ist nicht erlaubt  
}
```

## Lösung: Interfaces in Java

Ein Interface ist ein Vertrag, der festlegt, welche Methoden eine Klasse implementieren muss. Es ist eine Alternative zur Mehrfachvererbung.

### Vorteile von Interfaces

1. Ermöglicht Mehrfachimplementierung.
2. Entkoppelt die Implementierung vom Verhalten.
3. Fördert Flexibilität und Wiederverwendbarkeit.

### Definition eines Interfaces:

```
interface Fahrbar {  
    void starten();  
    void stoppen();  
}
```

### Implementierung eines Interfaces:

```
class Benzinauto implements Fahrbar {  
    @Override  
    public void starten() {  
        System.out.println("Benzinauto startet.");  
    }  
  
    @Override  
    public void stoppen() {  
        System.out.println("Benzinauto stoppt.");  
    }  
}  
  
class Elektroauto implements Fahrbar {  
    @Override  
    public void starten() {  
        System.out.println("Elektroauto startet.");  
    }  
  
    @Override
```

```

    public void stoppen() {
        System.out.println("Elektroauto stoppt.");
    }
}

```

## Direkte Methodenimplementierung in Interfaces

Seit Java 8 können Methoden in Interfaces mit **default** implementiert werden.

```

interface Wartung {
    default void status() {
        System.out.println("Wartungsstatus: Alles in Ordnung.");
    }
}

class Sportwagen implements Wartung {
    // Kann die default-Methode optional überschreiben
}

```

## Anwendung von Interfaces: Beispiel Auto und Elektroauto

Szenario:

Ein Auto kann verschiedene Arten von Fahrzeugen sein, wie z. B. Benzin- oder Elektroauto. Beide haben jedoch ein gemeinsames Verhalten: sie können fahren.

```

interface Fahrzeug {
    void fahren();
}

class Benzinauto implements Fahrzeug {
    @Override
    public void fahren() {
        System.out.println("Das Benzinauto fährt.");
    }
}

class Elektroauto implements Fahrzeug {
    @Override
    public void fahren() {
        System.out.println("Das Elektroauto fährt.");
    }
}

public class Main {
    public static void main(String[] args) {
        Fahrzeug auto1 = new Benzinauto();

```

```

Fahrzeug auto2 = new Elektroauto();

auto1.fahren();
auto2.fahren();
}

}

```

## Zusammenfassung: Unterschiede zwischen Vererbung und Interfaces

Aspekt	Vererbung	Interfaces
Anzahl der Elternklassen	Eine	Mehrere
Modifikatoren	public, private, protected	Nur public
Verwendung	Implementierung gemeinsamer Logik	Definition eines gemeinsamen Verhaltens
Flexibilität	Gering (starke Kopplung)	Hoch (lose Kopplung)

## Quizfragen zur Selbstkontrolle

1. **Was sind die Hauptprobleme der Vererbung in Java?**
2. **Warum bietet ein Interface eine Lösung für Mehrfachvererbungsprobleme?**
3. **Was ist der Unterschied zwischen einer abstrakten Klasse und einem Interface?**
4. **Können Interfaces private Methoden haben?**

## Lösungen zum Quiz

1. Probleme der Vererbung:
  - Keine Mehrfachvererbung.
  - Änderungen in der Oberklasse können Unterklassen beeinflussen.
  - Kann zu starren Abhängigkeiten führen.
2. Interfaces lösen das Problem, da sie Mehrfachimplementierungen erlauben und nur das Verhalten spezifizieren, nicht die Implementierung.
3. Eine abstrakte Klasse kann sowohl abstrakte als auch konkrete Methoden enthalten, während ein Interface nur abstrakte Methoden (bis Java 7) oder default-Methoden (seit Java 8) erlaubt.
4. Seit Java 9 können Interfaces private Methoden enthalten.

## Direkte Methodenimplementierung in Interfaces (Default-Methoden) in Java

Seit Java 8 können Interfaces **Default-Methoden** enthalten. Diese ermöglichen es, **Methoden direkt im Interface** mit einer Standardimplementierung zu definieren. Dadurch entfällt in vielen Fällen der Bedarf an Adapterklassen, da du nur noch die Methoden überschreiben musst, die du wirklich anpassen möchtest.

---

### Was sind Default-Methoden?

Default-Methoden:

- Werden direkt im Interface definiert.
  - Haben eine **Standardimplementierung**, die von den implementierenden Klassen übernommen werden kann.
  - Können von der implementierenden Klasse **überschrieben** werden, wenn eine andere Implementierung benötigt wird.
- 

### Beispiel: Ein Interface mit Default-Methoden

Stell dir ein Interface **AutoAktionen** vor, das Standardmethoden für ein Auto definiert.

```
public interface AutoAktionen {  
    void motorStarten(); // Muss von allen implementierenden Klassen definiert  
    werden  
  
    default void beschleunigen() {  
        System.out.println("Das Auto beschleunigt mit Standardgeschwindigkeit.");  
    }  
  
    default void bremsen() {  
        System.out.println("Das Auto bremst standardmäßig.");  
    }  
  
    default void hupen() {  
        System.out.println("Die Hupe ertönt standardmäßig.");  
    }  
}
```

---

### Implementierung in einer Klasse

Jetzt kannst du das Interface implementieren und nur die Methoden überschreiben, die du anpassen möchtest.

```
public class ElektroAuto implements AutoAktionen {  
    @Override  
    public void motorStarten() {
```

```

        System.out.println("Der Elektromotor startet lautlos.");
    }

    // Keine Überschreibung von `beschleunigen()`, `bremsen()` oder `hupen()`
    // nötig,
    // da die Standardimplementierung aus dem Interface verwendet wird.
}

```

## Ausgabe:

```

public class Main {
    public static void main(String[] args) {
        AutoAktionen auto = new ElektroAuto();
        auto.motorStarten(); // "Der Elektromotor startet lautlos."
        auto.beschleunigen(); // "Das Auto beschleunigt mit
Standardgeschwindigkeit."
        auto.bremsen(); // "Das Auto bremst standardmäßig."
        auto.hupen(); // "Die Hupe ertönt standardmäßig."
    }
}

```

## Überschreiben einer Default-Methode

Wenn eine Klasse eine andere Implementierung für eine Standardmethode benötigt, kann sie die Default-Methode überschreiben.

```

public class Sportwagen implements AutoAktionen {
    @Override
    public void motorStarten() {
        System.out.println("Der Sportwagen-Motor startet mit einem aggressiven
Röhren.");
    }

    @Override
    public void beschleunigen() {
        System.out.println("Der Sportwagen beschleunigt extrem schnell.");
    }
}

```

## Ausgabe:

```

public class Main {
    public static void main(String[] args) {
        AutoAktionen auto = new Sportwagen();
        auto.motorStarten(); // "Der Sportwagen-Motor startet mit einem

```

```

        aggressiven Röhren."
        auto.beschleunigen(); // "Der Sportwagen beschleunigt extrem schnell."
        auto.bremsen(); // "Das Auto bremst standardmäßig."
        auto.hupen(); // "Die Hupe ertönt standardmäßig."
    }
}

```

---

## Vorteile von Default-Methoden

### 1. Flexibilität:

- Implementierende Klassen müssen nur die Methoden überschreiben, die sie ändern wollen.
- Es bleibt eine Standardimplementierung verfügbar, wenn keine spezifische benötigt wird.

### 2. Weniger Boilerplate-Code:

- Du musst keine Adapterklassen oder leere Implementierungen für ungenutzte Methoden schreiben.

### 3. Abwärtskompatibilität:

- Neue Methoden können zu einem bestehenden Interface hinzugefügt werden, ohne alte Implementierungen zu brechen.
- 

## Grenzen von Default-Methoden

### 1. Ambiguitäten:

- Wenn eine Klasse von mehreren Interfaces mit gleichen Default-Methoden erbt, muss die Klasse eine eigene Implementierung der Methode bereitstellen.

```

public interface InterfaceA {
    default void methode() {
        System.out.println("Default in InterfaceA");
    }
}

public interface InterfaceB {
    default void methode() {
        System.out.println("Default in InterfaceB");
    }
}

public class Konflikt implements InterfaceA, InterfaceB {
    @Override
    public void methode() {
        System.out.println("Eigene Implementierung in Konflikt.");
    }
}

```

## 2. Keine Zustände:

- Default-Methoden können keinen Zustand (Instanzvariablen) speichern, da Interfaces in Java keine Instanzvariablen haben dürfen.
- 

## Zusammenfassung

Default-Methoden in Interfaces	Adapterklassen
Einführung seit Java 8.	Seit Beginn von Java.
Ermöglicht Standardimplementierungen direkt im Interface.	Adapterklassen bieten leere Implementierungen für alle Methoden eines Interfaces.
Bessere Flexibilität und weniger Boilerplate-Code.	Besonders hilfreich bei älteren Interfaces mit vielen Methoden.
Ideal für neue und moderne APIs.	Geeignet für ältere APIs wie AWT/Swing.

Default-Methoden machen Adapterklassen in vielen Fällen überflüssig, vor allem bei modernen Java-Programmen. Sie bieten eine elegante und direkte Lösung für Standardimplementierungen.

## Interfaces und Adapterklassen in Java

Du kannst **Interfaces** in Java entweder direkt oder mit Hilfe von **Adapterklassen** verwenden. Welche Variante du wählst, hängt von der Anzahl der Methoden im Interface und von deinem spezifischen Anwendungsfall ab.

---

### 1. Interfaces direkt verwenden

Wenn du ein Interface direkt implementierst, musst du **alle Methoden** des Interfaces in deiner Klasse definieren, selbst wenn du einige davon nicht benötigst. Dies ist der Standardweg und funktioniert gut, wenn das Interface nur wenige Methoden hat.

```
public interface AutoAktionen {  
    void motorStarten();  
    void beschleunigen();  
    void bremsen();  
    void hupen();  
}
```

Wenn du dieses Interface implementierst:

```
public class ElektroAuto implements AutoAktionen {  
    @Override  
    public void motorStarten() {  
        System.out.println("Elektromotor startet lautlos.");  
    }  
  
    @Override  
    public void beschleunigen() {  
        System.out.println("Elektroauto beschleunigt.");  
    }  
  
    @Override  
    public void bremsen() {  
        // Nicht benötigt  
    }  
  
    @Override  
    public void hupen() {  
        // Nicht benötigt  
    }  
}
```

Hier musst du auch die Methoden `bremsen()` und `hupen()` definieren, selbst wenn sie leer bleiben.

---

### 2. Interfaces mit Adapterklassen verwenden

Wenn ein Interface **viele Methoden** hat und du nur wenige davon benötigst, kannst du eine **Adapterklasse** verwenden. Diese Klasse implementiert das Interface und stellt leere Implementierungen für alle Methoden bereit. Dadurch kannst du nur die Methoden überschreiben, die du wirklich benötigst.

#### Wie funktioniert das?

1. Du erstellst eine **Adapterklasse**, die das Interface implementiert und alle Methoden leer lässt.
2. Du erstellst dann eine **spezifische Klasse**, die von der Adapterklasse erbt, und überschreibt nur die benötigten Methoden.

#### Adapterklasse:

```
public abstract class AutoAdapter implements AutoAktionen {  
    @Override  
    public void motorStarten() {}  
    @Override  
    public void beschleunigen() {}  
    @Override  
    public void bremsen() {}  
    @Override  
    public void hupen() {}  
}
```

#### Spezifische Klasse:

```
public class ElektroAuto extends AutoAdapter {  
    @Override  
    public void motorStarten() {  
        System.out.println("Elektromotor startet lautlos.");  
    }  
  
    @Override  
    public void beschleunigen() {  
        System.out.println("Elektroauto beschleunigt.");  
    }  
}
```

---

## Wann benutzt man Adapterklassen?

Du benutzt Adapterklassen, wenn:

1. Ein **Interface viele Methoden** hat, aber du nur wenige davon verwenden möchtest.
2. Du Boilerplate-Code vermeiden möchtest (leere Methoden in deiner Klasse).

#### Beispiel: **MouseListener direkt vs. mit Adapterklasse**

Ohne Adapterklasse:

```
public class MausBeispiel implements MouseListener {  
    @Override  
    public void mouseClicked(MouseEvent e) {  
        System.out.println("Maus geklickt.");  
    }  
  
    @Override  
    public void mousePressed(MouseEvent e) {}  
    @Override  
    public void mouseReleased(MouseEvent e) {}  
    @Override  
    public void mouseEntered(MouseEvent e) {}  
    @Override  
    public void mouseExited(MouseEvent e) {}  
}
```

Mit Adapterklasse ([MouseAdapter](#)):

```
public class MausBeispiel extends MouseAdapter {  
    @Override  
    public void mouseClicked(MouseEvent e) {  
        System.out.println("Maus geklickt.");  
    }  
}
```

## Zusammenfassung: Wann was verwenden?

Direkte Verwendung des Interfaces	Adapterklasse verwenden
Wenn das Interface nur wenige Methoden hat.	Wenn das Interface viele Methoden hat.
Du benötigst alle oder fast alle Methoden des Interfaces.	Du benötigst nur einige wenige Methoden.
Kein zusätzlicher Codeaufwand für eine Adapterklasse.	Du sparst Boilerplate-Code für leere Methoden.

In der Praxis werden Adapterklassen vor allem bei älteren APIs (z. B. AWT/Swing) oder bei sehr großen Interfaces eingesetzt. In modernen Java-Versionen kannst du oft auch **Default-Methoden** in Interfaces verwenden, was den Bedarf an Adapterklassen verringert.

# Java-Versionen und Vergleich zwischen Java 7 und Java 8

---

## Übersicht früherer Java-Versionen

### Java 6

- **Release Date:** Dezember 2006
- **Schlüsselmerkmale:**
  - Verbesserte Performance und neue Monitoring-Tools.
  - Einführung des Compiler API (JSR 199).
  - Erweiterte Desktop-APIs, wie das System Tray API.
  - Verbesserte Unterstützung für Web Services und Scripting.

### Java 7

- **Release Date:** Juli 2011
- **Schlüsselmerkmale:**
  - **Try-with-resources:** Automatische Schließung von Ressourcen wie Dateien und Datenbanken.
  - **Switch mit Strings:** Nutzung von Strings in `switch`-Anweisungen.
  - **Dynamische Typisierung:** Unterstützung dynamischer Sprachen auf der JVM mit `invokedynamic`.
  - **Diamond Operator (<>):** Vereinfachte Generics-Deklarationen.
  - **NIO 2 API:** Verbesserte Dateioperationen und Unterstützung für asynchrone Aufgaben.
  - **Mehrere Ausnahmen in einer catch-Anweisung:** Mehrere Ausnahmetypen können in einem Block behandelt werden.

### Java 8

- **Release Date:** März 2014
- **Schlüsselmerkmale:**
  - **Lambdas und funktionale Programmierung:** Einführung von Lambda-Ausdrücken für eine kompakte und deklarative Programmierung.
  - **Streams API:** Datenströme können mit Methoden wie `filter`, `map` und `reduce` verarbeitet werden.
  - **Default Methods:** Ermöglicht Standardimplementierungen in Interfaces.
  - **Neue Date and Time API:** Bessere Handhabung von Datums- und Zeitoperationen.
  - **Optional-Klasse:** Umgang mit Null-Werten ohne `NullPointerException`.
  - **Nashorn JavaScript Engine:** Ersetzt Rhino für JavaScript-Unterstützung.
  - **Parallel Streams:** Unterstützung paralleler Datenverarbeitung.
  - **Annotations on Types:** Verbesserung der Typannotationen.

---

## Vergleich zwischen Java 7 und Java 8

Feature	Java 7	Java 8
---------	--------	--------

Feature	Java 7	Java 8
<b>Lambda Expressions</b>	Nicht verfügbar	Unterstützt kompakte und deklarative funktionale Programmierung.
<b>Streams API</b>	Nicht verfügbar	Verarbeitung von Datenströmen mit deklarativen Methoden.
<b>Default Methods</b>	Nicht verfügbar	Standardmethoden in Interfaces für Abwärtskompatibilität.
<b>Try-with-resources</b>	Verfügbar	Auch verfügbar.
<b>Switch mit Strings</b>	Verfügbar	Auch verfügbar.
<b>Annotations on Types</b>	Nicht verfügbar	Verfügbar, um Typannotationen zu verbessern.
<b>Neue Date and Time API</b>	Nicht verfügbar	Verfügbar mit besseren Möglichkeiten für Zeitoperationen.
<b>Optional-Klasse</b>	Nicht verfügbar	Verfügbar zur Vermeidung von Nullwert-Problemen.

## Fazit

Java 7 brachte wesentliche Verbesserungen in Bezug auf die Sprachkonstrukte und APIs, während Java 8 einen Paradigmenwechsel hin zur funktionalen Programmierung und modernen APIs einleitete. Java 8 wird als eine der bedeutendsten Versionen in der Java-Geschichte betrachtet.

# Wichtige Bibliotheken in Java: **Math** und **Random**

---

Java bietet viele integrierte Bibliotheken, die Entwicklern leistungsstarke Werkzeuge für häufige Aufgaben bereitstellen. Zwei der am häufigsten verwendeten Klassen sind **Math** und **Random**, die in der Java-Standardbibliothek enthalten sind.

---

## 1. Die Klasse **Math**

Die **Math**-Klasse in Java bietet eine Sammlung statischer Methoden für mathematische Operationen wie Arithmetik, Trigonometrie und Exponentialberechnungen. Sie gehört zum Paket **java.lang** und ist ohne zusätzlichen Import verfügbar.

Wichtige Methoden der **Math**-Klasse

Methode	Beschreibung	Beispiel
<code>Math.abs(double x)</code>	Absolutwert einer Zahl	<code>Math.abs(-5.3) -&gt; 5.3</code>
<code>Math.max(double a, b)</code>	Maximum von zwei Zahlen	<code>Math.max(3, 7) -&gt; 7</code>
<code>Math.min(double a, b)</code>	Minimum von zwei Zahlen	<code>Math.min(3, 7) -&gt; 3</code>
<code>Math.pow(double a, b)</code>	Potenz, <b>a</b> hoch <b>b</b>	<code>Math.pow(2, 3) -&gt; 8.0</code>
<code>Math.sqrt(double x)</code>	Quadratwurzel	<code>Math.sqrt(16) -&gt; 4.0</code>
<code>Math.random()</code>	Zufallszahl zwischen 0 (inkl.) und 1 (exkl.)	<code>Math.random() -&gt; 0.52345</code>
<code>Math.round(double x)</code>	Rundet auf die nächste Ganzzahl	<code>Math.round(4.7) -&gt; 5</code>
<code>Math.ceil(double x)</code>	Rundet auf die nächste größere Ganzzahl	<code>Math.ceil(4.2) -&gt; 5.0</code>
<code>Math.floor(double x)</code>	Rundet auf die nächste kleinere Ganzzahl	<code>Math.floor(4.7) -&gt; 4.0</code>
<code>Math.sin(double x)</code>	Sinus (in Radian)	<code>Math.sin(Math.PI / 2) -&gt; 1.0</code>
<code>Math.cos(double x)</code>	Kosinus (in Radian)	<code>Math.cos(0) -&gt; 1.0</code>
<code>Math.log(double x)</code>	Natürlicher Logarithmus	<code>Math.log(10) -&gt; 2.3025</code>

Beispiele für die Anwendung von **Math**

```

public class MathExample {
    public static void main(String[] args) {
        // Absolutwert
        System.out.println("Absolutwert von -5: " + Math.abs(-5));

        // Maximum und Minimum
        System.out.println("Maximum von 5 und 10: " + Math.max(5, 10));
        System.out.println("Minimum von 5 und 10: " + Math.min(5, 10));

        // Potenz und Quadratwurzel
        System.out.println("2 hoch 3: " + Math.pow(2, 3));
        System.out.println("Quadratwurzel von 16: " + Math.sqrt(16));

        // Runden, Decke und Boden
        System.out.println("Runden von 4.7: " + Math.round(4.7));
        System.out.println("Ceil von 4.2: " + Math.ceil(4.2));
        System.out.println("Floor von 4.7: " + Math.floor(4.7));

        // Zufallszahl zwischen 0 und 1
        System.out.println("Zufallszahl: " + Math.random());

        // Trigonometrie
        System.out.println("Sinus von π/2: " + Math.sin(Math.PI / 2));
        System.out.println("Kosinus von 0: " + Math.cos(0));

        // Logarithmus
        System.out.println("Natürlicher Logarithmus von 10: " + Math.log(10));
    }
}

```

## 2. Die Klasse Random

Die Klasse **Random** im Paket `java.util` bietet erweiterte Methoden zur Generierung von Zufallszahlen. Im Gegensatz zu `Math.random()` kann **Random** gezielt Zufallszahlen in verschiedenen Bereichen und Formaten generieren.

### Wichtige Methoden der **Random**-Klasse

Methode	Beschreibung	Beispiel
<code>nextInt()</code>	Liefert eine zufällige int-Zahl	<code>nextInt()</code> -> -123456789
<code>nextInt(int bound)</code>	Zufällige int-Zahl zwischen <code>0</code> (inkl.) und <code>bound</code>	<code>nextInt(10)</code> -> 7
<code>nextDouble()</code>	Zufällige double-Zahl zwischen 0.0 und 1.0	<code>nextDouble()</code> -> 0.72
<code>nextBoolean()</code>	Liefert <code>true</code> oder <code>false</code>	<code>nextBoolean()</code> -> true
<code>nextLong()</code>	Liefert eine zufällige long-Zahl	<code>nextLong()</code> -> 9876543210L

Methode	Beschreibung	Beispiel
nextFloat()	Liefert eine zufällige float-Zahl zwischen 0.0 und 1.0	nextFloat() -> 0.34

## Beispiele für die Anwendung von Random

```

import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        // Random-Objekt erstellen
        Random random = new Random();

        // Zufällige int-Zahlen
        System.out.println("Zufällige int-Zahl: " + random.nextInt());
        System.out.println("Zufällige int-Zahl zwischen 0 und 10: " +
random.nextInt(10));

        // Zufällige double-Zahl
        System.out.println("Zufällige double-Zahl: " + random.nextDouble());

        // Zufällige boolean-Werte
        System.out.println("Zufälliger boolean: " + random.nextBoolean());

        // Zufällige long-Zahlen
        System.out.println("Zufällige long-Zahl: " + random.nextLong());

        // Zufällige float-Zahlen
        System.out.println("Zufällige float-Zahl: " + random.nextFloat());
    }
}

```

## Vergleich von Math.random() und Random

Eigenschaft	Math.random()	Random
Paket	java.lang	java.util
Verwendung	Liefert nur double zwischen 0 und 1	Liefert verschiedene Datentypen
Flexibilität	Begrenzte Konfiguration	Hohe Flexibilität (Bereich, Typ)
Seed	Kann nicht explizit gesetzt werden	Kann explizit gesetzt werden

## Fazit

- Die **Math-Klasse** eignet sich für mathematische Berechnungen und generiert schnelle Zufallszahlen über **Math.random()**.

- Die **Random-Klasse** bietet mehr Kontrolle und Flexibilität bei der Generierung von Zufallszahlen in verschiedenen Datentypen und Bereichen.
- Beide Klassen sind leistungsstarke Werkzeuge, die je nach Anwendungsfall verwendet werden können.