

Kapitel 7 - Packages und Sichtbarkeit, lokale Klassen, abstrakte Klassen und Methoden, Interfaces, enum

1. Java Grundlagen: Entwicklungszyklus, Entwicklungsumgebung

2. Datentypen, Kodierung, Binärzahlen, Variablen, Arrays

3. Ausdrücke, Operatoren, Schleifen und Verzweigungen

4. Blöcke, Sichtbarkeit und Methoden (Teil 1)

5. Grundkonzepte der Objektorientierung

6. Objektorientierung: Sichtbarkeit, Vererbung, Methoden (Teil 2), Konstruktor

7. Packages, lokale Klassen, abstrakte Klassen und Methoden, Interfaces, enum

8. Arbeiten mit Objekten: Identität, Listen, Komparatoren, Kopien, Wrapper, Iterator

9. Fehlerbehandlung: Exceptions und Logging

10. Utilities: Math, Date, Calendar, System, Random

11. Rekursion, Sortieralgorithmen und Collections

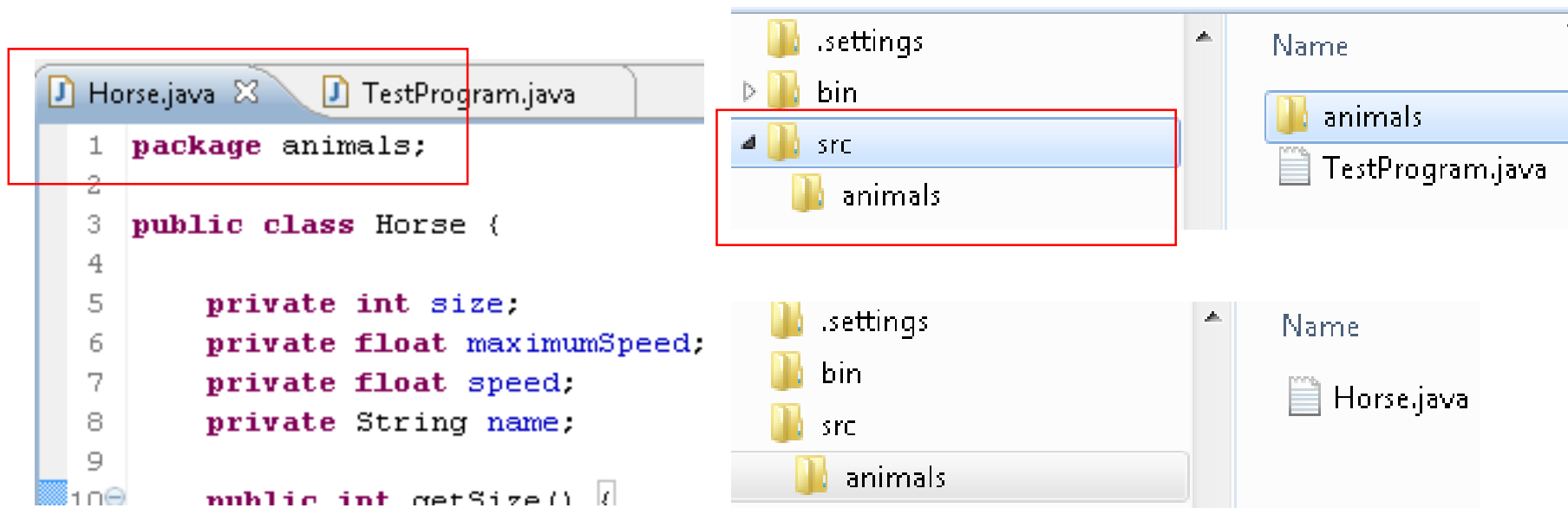
12. Nebenläufigkeit: Arbeiten mit Threads

13. Benutzeroberflächen mit Swing

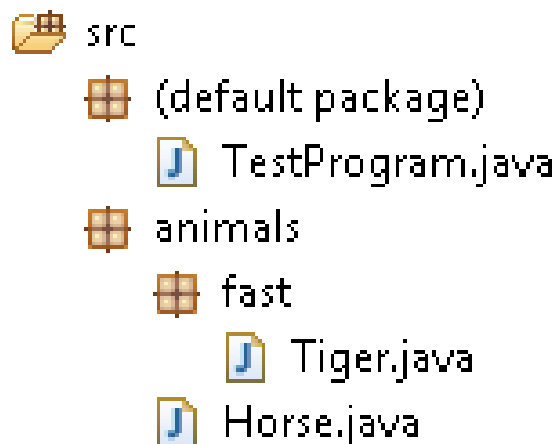
14. Streams: Auf Dateien und auf das Netzwerk zugreifen

Packages

- Packages sind Zusammenfassungen von Klassen zu Paketen
- Jedes Paket entspricht einem Dateisystem-Verzeichnis, in dem die zugehörigen Klassen liegen

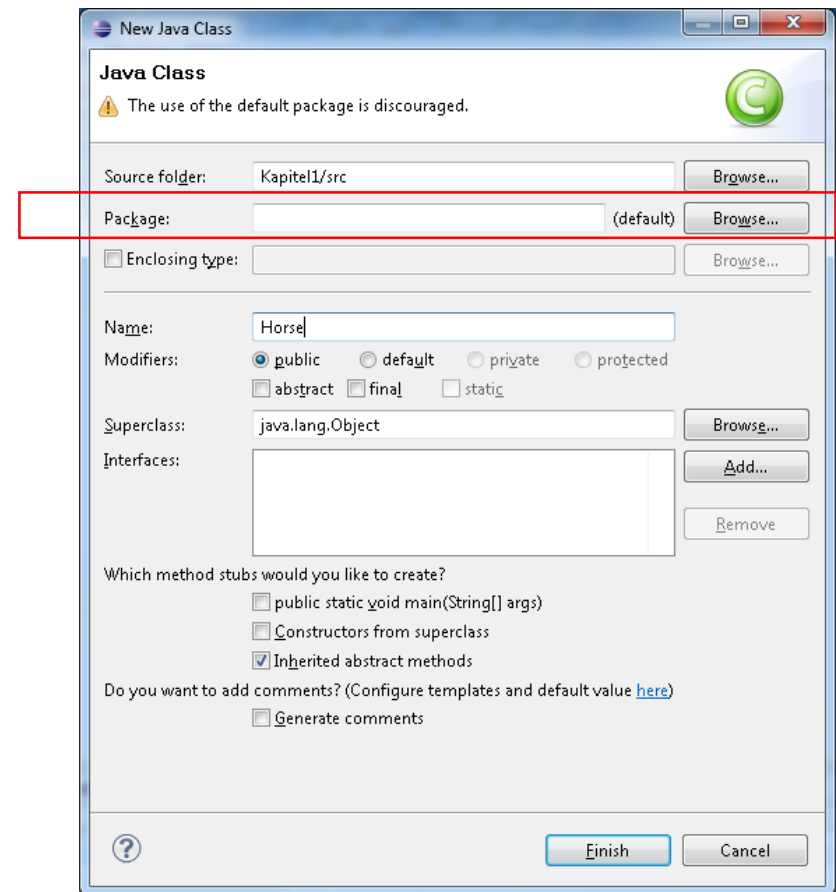
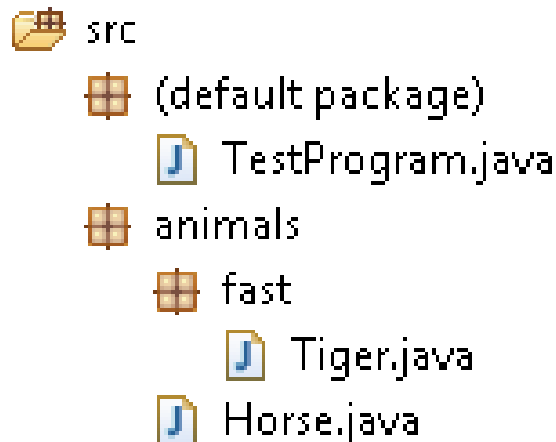


- Packages können wieder Packages enthalten (vgl. Verzeichnisse)
- Der Name einer Klasse innerhalb von Packages setzt sich aus den einzelnen Package-Namen getrennt durch einen „.“ zusammen (Pfad).
- Beispiel: `animals.fast.Tiger`

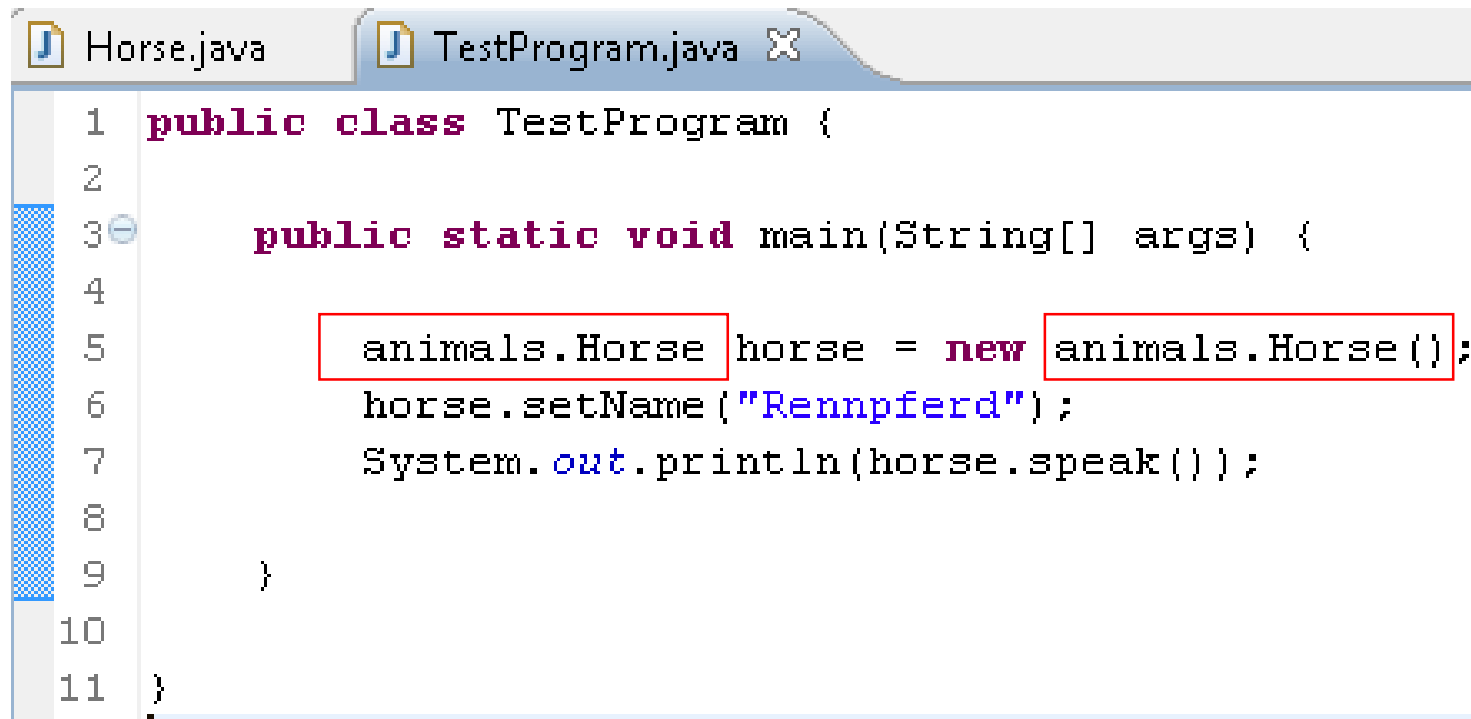


```
Tiger.java
1  package animals.fast;
2
3  public class Tiger {
4
5  }
6
```

- Package in eclipse angeben
- „Default Package“

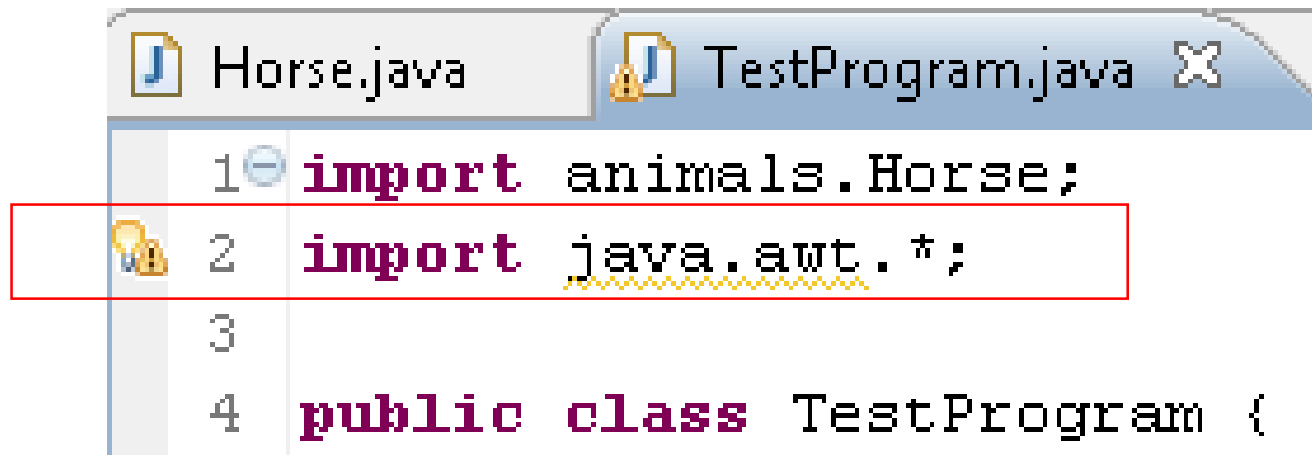


- Referenz auf Klassen, die innerhalb von Packages liegen
- Möglichkeit 1: den absoluten Pfad angeben



```
1 public class TestProgram {
2
3     public static void main(String[] args) {
4
5         animals.Horse horse = new animals.Horse();
6         horse.setName("Rennpferd");
7         System.out.println(horse.speak());
8
9     }
10
11 }
```

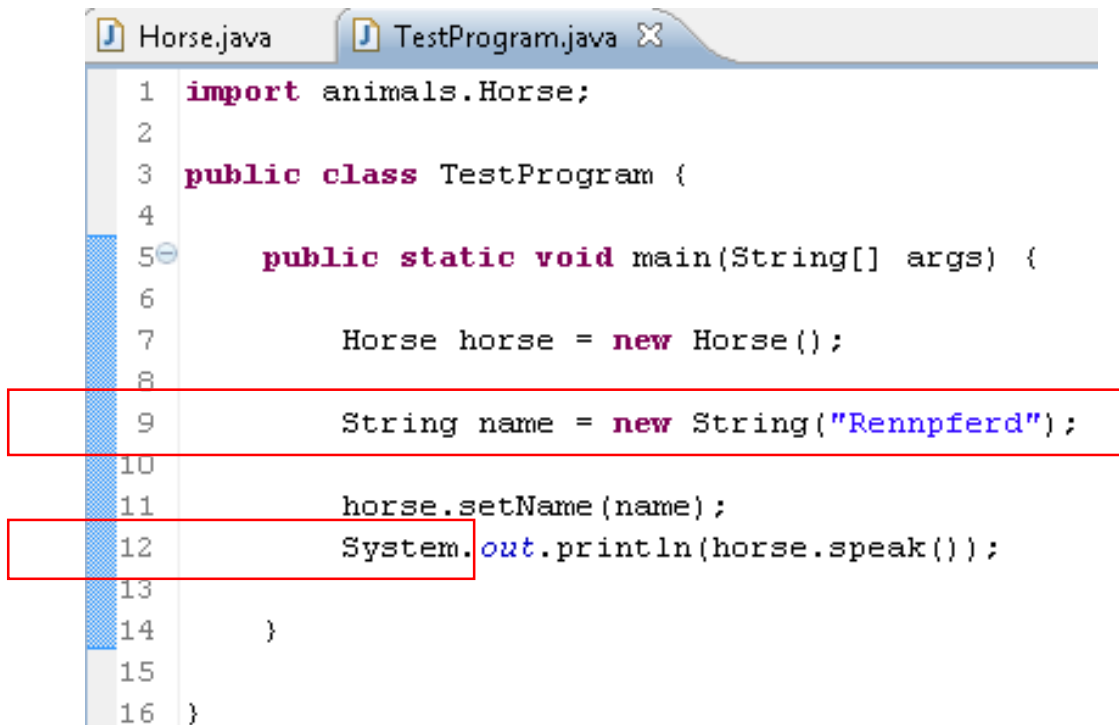
- Referenz auf Klassen, die innerhalb von Packages liegen
- Möglichkeit 2: das Package importieren (bevorzugter Weg)
- Wildcard * um alle Packages unterhalb des angegebenen Packages zu importieren



```
1 import animals.Horse;
2 import java.awt.*;
3
4 public class TestProgram {
```

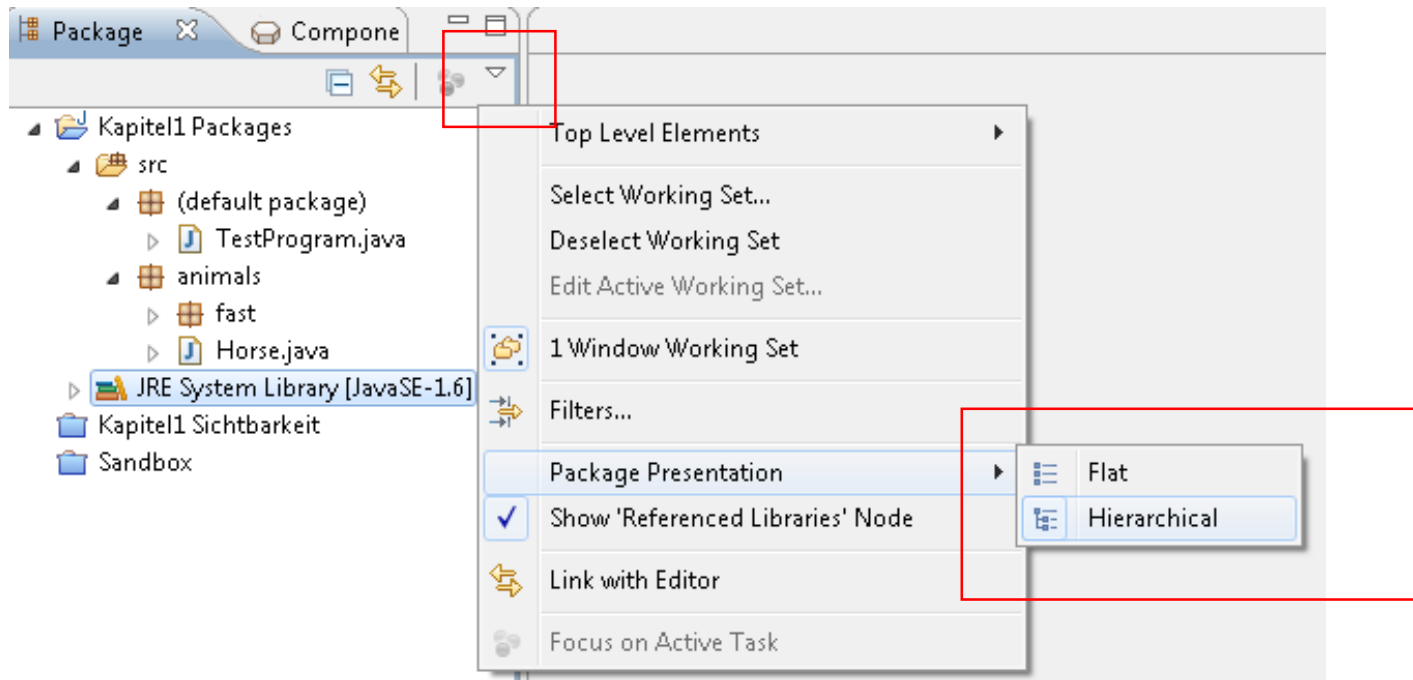

Das Package java.lang muss nicht angegeben werden

- Beispiel: java.lang.String, java.lang.System, etc.



```
1 import animals.Horse;
2
3 public class TestProgram {
4
5     public static void main(String[] args) {
6
7         Horse horse = new Horse();
8
9         String name = new String("Rennpferd");
10
11         horse.setName(name);
12         System.out.println(horse.speak());
13
14     }
15
16 }
```

- Darstellung in eclipse: Flat vs. Hierarchical



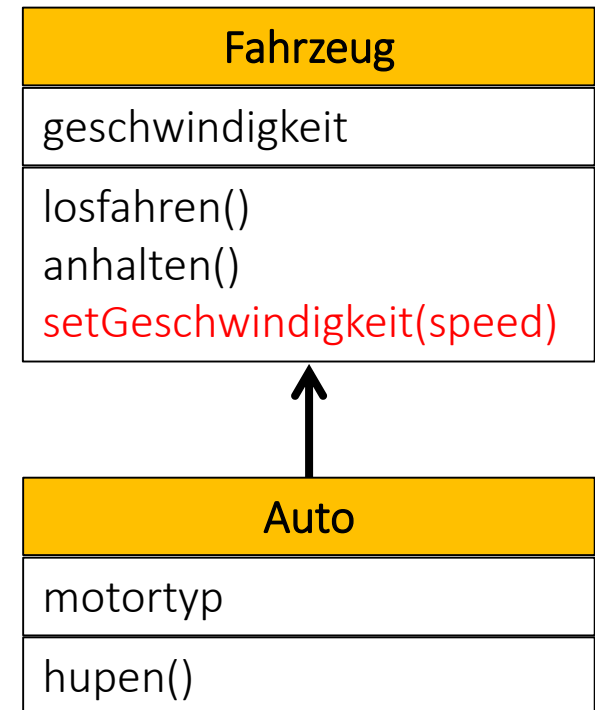
- eclipse: Organize Imports mit STRG + SHIFT + O

Sichtbarkeit von Methoden und Variablen

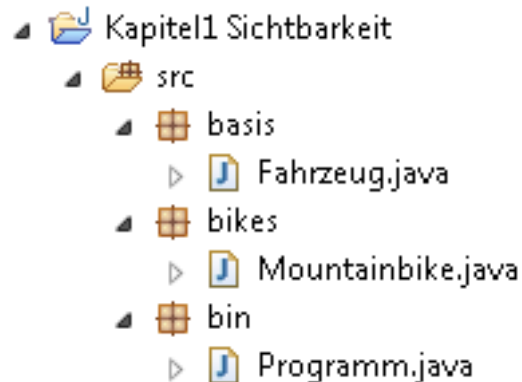
Elemente des Typs **public** sind in der Klasse selbst, in Methoden abgeleiteter Klassen und für den Aufrufer von Instanzen der Klasse sichtbar.

Elemente des Typs **private** sind in der Klasse selbst sichtbar. Für abgeleitete Klassen und Aufrufer sind diese verdeckt.

Elemente des Typs **protected** sind nur in der Klasse selbst und in Methoden abgeleiteter Klassen sichtbar. Zusätzlich können Klassen desselben **Pakets** diese aufrufen.



[Fahrzeug.java, Auto.java]



```
Fahrzeug.java Mountainbike.java Programm.java
1 package basis;
2 public class Fahrzeug {
3
4     private float geschwindigkeit = 0;
5     private String farbname;
6
7     public void setFarbe(String farbname){
8         this.farbname = farbname;
9     }
10
11     // Test 1: getFarbe() auf protected umstellen. Was passiert?
12     // Test 2: getFarbe() auf private umstellen. Was passiert?
13     public String getFarbe(){
14         return this.farbname;
15     }
16
17     public float getGeschwindigkeit() {
18         return geschwindigkeit;
19     }
20
21     public void setGeschwindigkeit(float geschwindigkeit) {
22         this.geschwindigkeit = geschwindigkeit;
23     }
24
25 }
```

protected – Beispiel

Elemente des Typs protected sind nur in der Klasse selbst und in Methoden abgeleiteter Klassen sichtbar. Zusätzlich können Klassen desselben **Pakets** diese aufrufen.

Elemente ohne Modifier (keine Angabe von public, protected oder private) werden als package scoped oder als Elemente mit Standard-Sichtbarkeit bezeichnet. Sie sind nur innerhalb des Pakets sichtbar zu dem sie gehören.

Unterschied protected / package scoped?

[s. Beispiel hierzu in Mountainbike.java, SecondFahrzeug.java]

eclipse: type hierarchy mit STRG + T

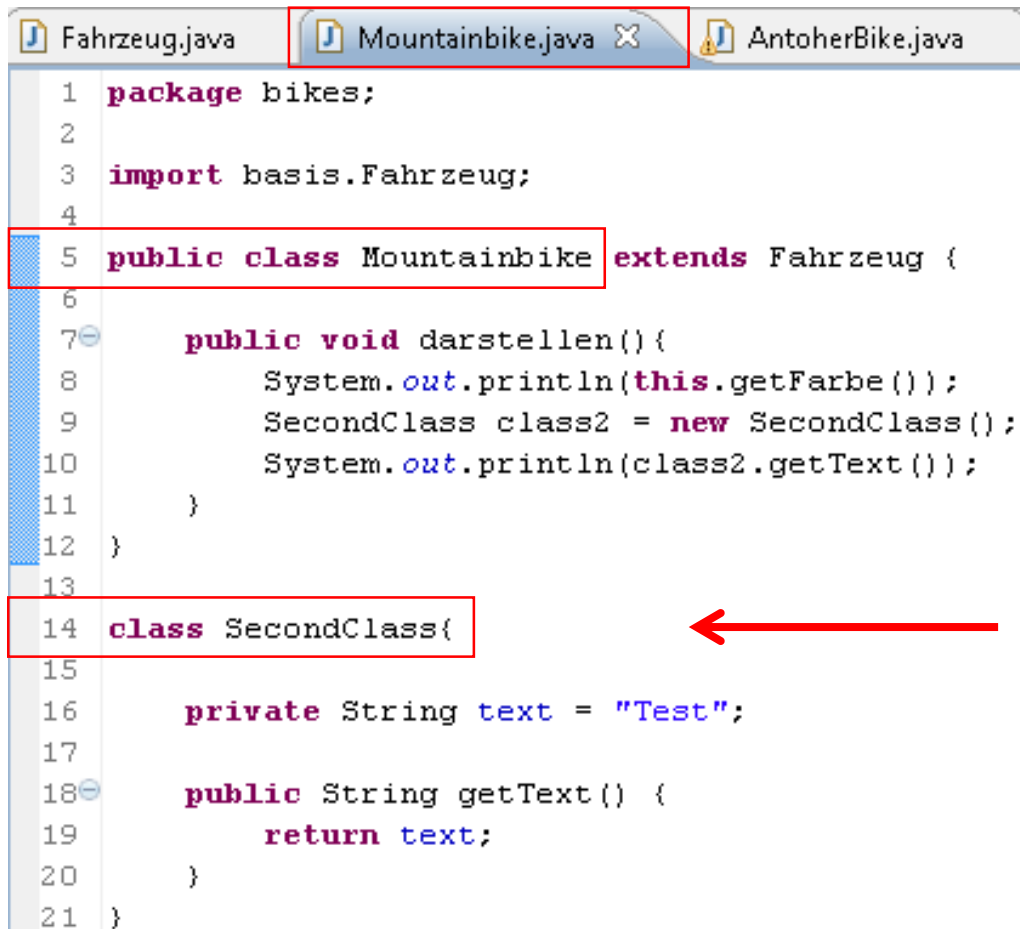
Sichtbarkeit von Klassen

Klassen mit dem modifier **public** sind auch außerhalb des Pakets sichtbar, in dem sie definiert wurden.

Klassen ohne den modifier **public** sind nur innerhalb des Pakets sichtbar, in dem sie definiert wurden (package scoped).

In jeder Quell-Datei (.class / .java) darf nur eine Klasse mit dem modifier **public** angelegt werden.

Beispiel s. nächste Folie

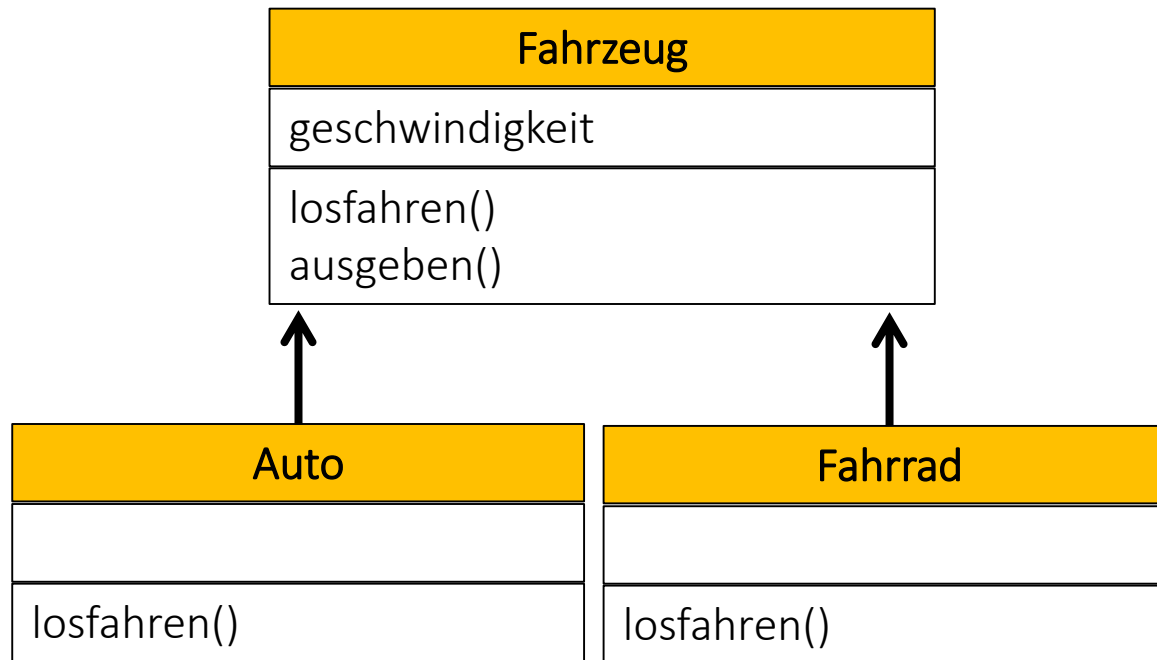


```
1 package bikes;
2
3 import basis.Fahrzeug;
4
5 public class Mountainbike extends Fahrzeug {
6
7     public void darstellen() {
8         System.out.println(this.getFarbe());
9         SecondClass class2 = new SecondClass();
10        System.out.println(class2.getText());
11    }
12 }
13
14 class SecondClass{
15
16     private String text = "Test";
17
18     public String getText() {
19         return text;
20     }
21 }
```

← Zugriff nur innerhalb des
Packages bikes

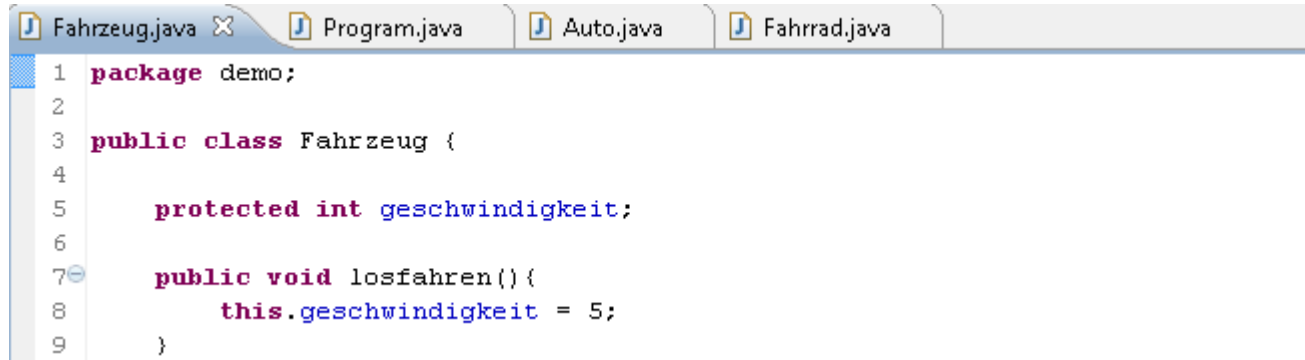
[Fahrzeug.java, Auto.java, [Mountainbike.java](#)]

Überschreiben



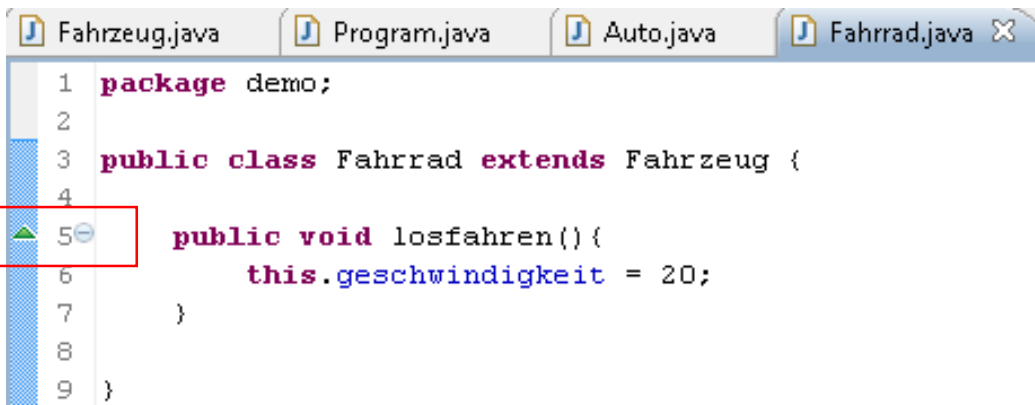
- Überschreiben / Überlagern durch Methoden mit dem gleichen Namen in den Kind-Klassen

[Kapitel 1 - Ueberschreiben]



The screenshot shows the Eclipse IDE with four tabs: Fahrzeug.java, Program.java, Auto.java, and Fahrrad.java. The Fahrzeug.java tab is active, displaying the following code:

```
1 package demo;
2
3 public class Fahrzeug {
4
5     protected int geschwindigkeit;
6
7     public void losfahren(){
8         this.geschwindigkeit = 5;
9     }
```



The screenshot shows the Eclipse IDE with the same four tabs. The Fahrrad.java tab is active, displaying the following code:

```
1 package demo;
2
3 public class Fahrrad extends Fahrzeug {
4
5     public void losfahren(){
6         this.geschwindigkeit = 20;
7     }
8
9 }
```

A red rectangle highlights the line number 5 in the Fahrrad.java file, and a green triangle icon is visible next to it, indicating method overriding.

- Beispiel: losfahren() wird in der Unterklasse Fahrrad überschrieben.
- Eclipse zeigt das an (grünes Dreieck)

[Kapitel 1 - Ueberschreiben]

Überladen

Util
minimum(int a, int b) minimum(float a, float b)

- Methoden mit gleichem Namen gelten als verschoben, wenn sie sich durch die Typen ihrer Parameter unterscheiden
- Java „sucht“ beim Aufruf der Methode auf der Grundlage der übergebenen Parameter die passende Methode

```
5 public static int minimum(int zahl1, int zahl2){  
6     if (zahl1 < zahl2){  
7         return zahl1;  
8     }  
9     return zahl2;  
10 }  
11  
12 public static float minimum(float zahl1, float zahl2){  
13     if (zahl1 < zahl2){  
14         return zahl1;  
15     }  
16     return zahl2;  
17 }
```

- Beispiel: Abhängig vom übergebenen Datentyp wird die passende minimum Methode gewählt
- Wie geht der Operator < mit verschiedenen Datentypen um?

```
12 public static float minimum(float zahl1, float zahl2){  
13     if (zahl1 < zahl2){  
14         return zahl1;  
15     }  
16     return zahl2;  
17 }  
18  
19 public static float minimum(float zahl1, float zahl2, float zahl3){  
20  
21     float min1_2 = minimum(zahl1, zahl2);  
22     float result = minimum(min1_2, zahl3);  
23  
24     return result;  
25 }
```

- Auch die Anzahl der Parameter spielt dabei eine Rolle


```
12 public static float minimum(float zahl1, float zahl2){  
13     if (zahl1 < zahl2){  
14         return zahl1;  
15     }  
16     return zahl2;  
17 }  
18  
19  
20 public static double minimum(float zahl1, float zahl2){  
21  
22     float result = minimum(zahl1, zahl2);  
23  
24     return (double) result;  
25 }
```

- Es ist nicht möglich, Methoden durch verschiedene Rückgabewerte zu überladen.

Beispiel

```
public class Mitarbeiter {
```

```
    // Die Attribute eines Mitarbeiters im Betrieb.
```

```
    private String name;
```

```
    private int personalNummer;
```

```
    private double gehalt;
```

```
    // Konstruktor zur Konstruktion eines Objekts
```

```
    public Mitarbeiter (String name,  
                        int personalNummer, double gehalt) {
```

```
        this.gehalt = gehalt;
```

```
        this.name = name;
```

```
        this.personalNummer = personalNummer;
```

```
    }
```

// Methode zur Selbstdarstellung

```
public void ausgabe() {  
    System.out.printf(  
        "Name = %-10s Personalnummer = %3d Gehalt = %7.2f\n",  
        name, personalNummer, gehalt);  
}
```

← <http://download.oracle.com/javase/6/docs/api/java/util/Formatter.html>

// Diese Methode sollte der Chef öfter anwenden...

```
public void erhoeheGehalt(double betrag) {  
    gehalt += betrag;  
}  
  
public double getGehalt() {  
    return gehalt;  
}  
  
public String getName() {  
    return name;  
}  
  
public int getPersonalNummer() {  
    return personalNummer;  
}
```

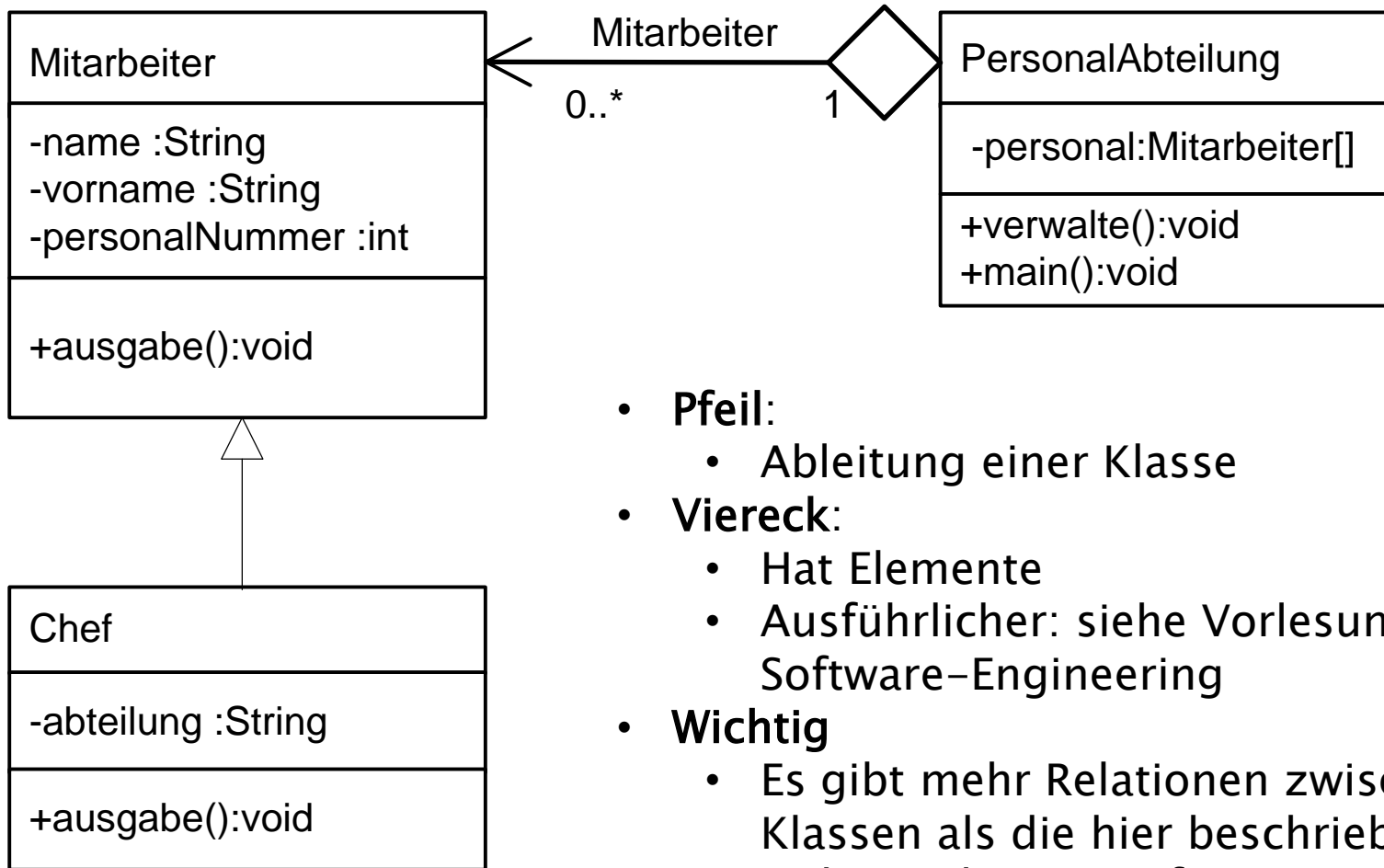
- **Beispiel: Chefs sind auch Mitarbeiter**
 - Mitarbeiter als Oberklasse (auch Basisklasse genannt)
 - Chef ist dann die Unterklasse (auch abgeleitete Klasse genannt)
 - Chef ist eine Spezialisierung von Mitarbeiter oder
 - Chef erbt von Mitarbeiter
- `class Chef extends Mitarbeiter {`
 - Zusätzlich zu Mitarbeitern die Chef-Attribute
 - Zusätzlich zu Mitarbeitern die Chef-Methoden
 - Gleiche Methoden wie bei Mitarbeitern können bei Chefs ein anderes Verhalten zeigen!
 - Konstruktoren für Chefs
- `}`
- Eine Klasse kann von höchstens einer anderen Klasse erben
 - Erinnerung: Java kennt keine Mehrfachvererbung
- Bezug zur Oberklasse mit `super`
 - Aus der Sicht von Chef: Mitarbeiter ist die Oberklasse (Basisklasse)

Beispiel: Chefs sind auch Mitarbeiter

- Eine Personalverwaltung verwaltet Objekte der Klasse Mitarbeiter.
- Sie kann auch Objekte der Klasse Chef verwalten, denn ein Chef ist auch ein Mitarbeiter.
- Deswegen braucht die Verwaltung nicht für Chefs neu programmiert werden.
- Dies ist ein großer Vorteil der Objektorientierung:
 - Einmal eine Verwaltung schreiben, z.B. für Applets
 - Dann läuft diese Verwaltung für alle Applets, z.B.:
 - `public class MyApplet extends Applet ...`
 - MyApplet ist ein Applet!

```
public class Chef extends Mitarbeiter{
    private String Abteilung; // Zusätzliches Attribut
    public Chef (String name, int personalNummer,
                 double gehalt, String abteilung) {
        super (name, personalNummer, gehalt); // Konstruktor
        this.abteilung = abteilung;           // Attribut
    }

    // Methoden
    // Optionale Annotation: ja, wir wollen überschreiben
    @Override
    public void ausgabe () {
        super.ausgabe (); // Darstellung der Oberklasse
        rufen
        System.out.println (" Leitung Abteilung " +
        abteilung);
    }
}
```

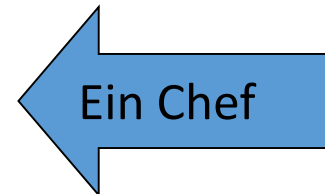


- **Pfeil:**
 - Ableitung einer Klasse
- **Viereck:**
 - Hat Elemente
 - Ausführlicher: siehe Vorlesung Software-Engineering
- **Wichtig**
 - Es gibt mehr Relationen zwischen den Klassen als die hier beschriebenen: siehe Vorlesung Software-Engineering


```
public class Personalverwaltung {
    public static void main (String[] args) {
        // Die Personalabteilung verwaltet das Personal
        Mitarbeiter personal [] = {
            new Mitarbeiter("Hitchcock", 0, 1000),
            new Mitarbeiter("Bond", 7, 2000),
            new Mitarbeiter("Ford", 99, 3000),
            new Chef ("Nealy", 1, 9000, "Sun")
        };
        for (Mitarbeiter p: personal)
            p.ausgabe ();

        for (Mitarbeiter p: personal)
            p.erhoeheGehalt (100);

        for (Mitarbeiter p: personal)
            p.ausgabe ();
    }
}
```

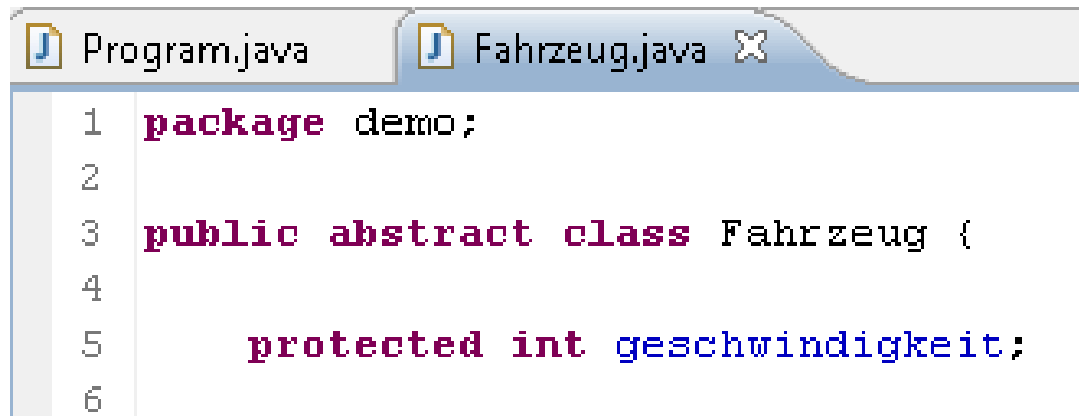


Der Chef ist ein Mitarbeiter. Er wird wie ein Mitarbeiter behandelt.

- Die ausgabe-Methode eines Mitarbeiter-Objekts wird aufgerufen.
- Das Objekt „weiß“, was es ist. Dies sorgt dafür, dass die passende Methode aufgerufen wird.
- Für ein Chef-Objekt ist dies die ausgabe()-Methode der Chef-Klasse, nicht die entsprechende Methode der Basisklasse, denn diese wurde überschrieben.
- Diese Bindung einer Methode an ein Objekt zum spätestmöglichen Zeitpunkt (Ablauf) wird auch als späte Bindung bezeichnet.
- Mit dieser Technik der „artgerechten“ Behandlung kann man Verwaltungssysteme erstellen, die Objekte von Klassen verwalten, die es bei der Entwicklung der Verwaltung noch nicht gab.

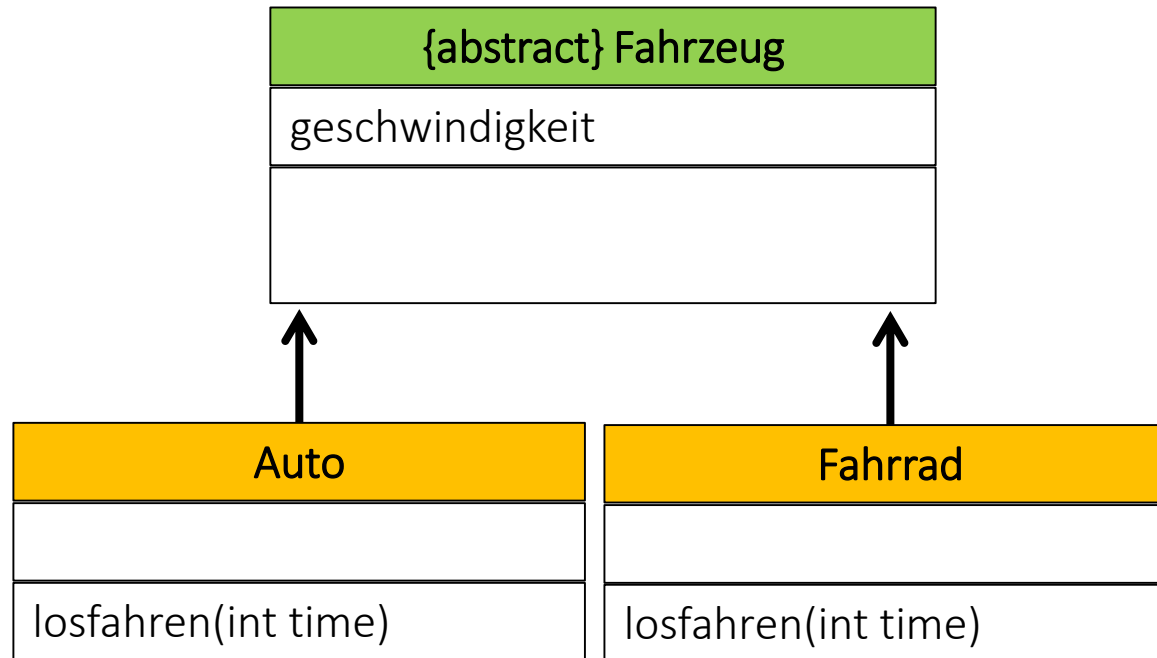
Abstrakte Klassen und Methoden

- Klassen werden wie folgt als abstrakte Klassen definiert:



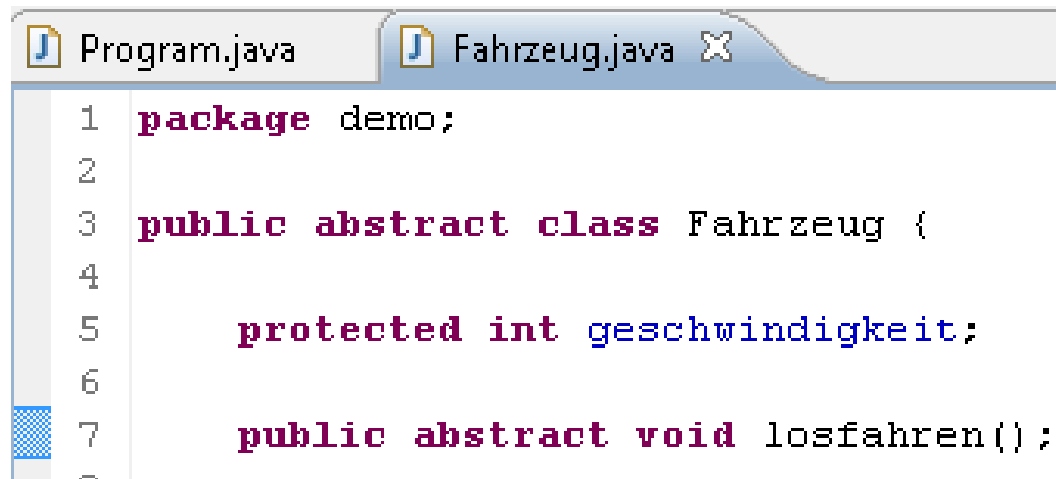
```
1 package demo;
2
3 public abstract class Fahrzeug {
4
5     protected int geschwindigkeit;
6 }
```

- Von abstrakten Klassen können keine Instanzen (Objekte) angelegt werden
- Sie dienen meist als Vaterklassen in einer Vererbungshierarchie



- Es sollen keine Instanzen von Fahrzeug angelegt werden, weil es keine abstrakten Fahrzeuge gibt.
- Es gibt aber „konkrete“ Autos und Fahrräder, die eine gemeinsame Vaterklasse Fahrzeug haben.

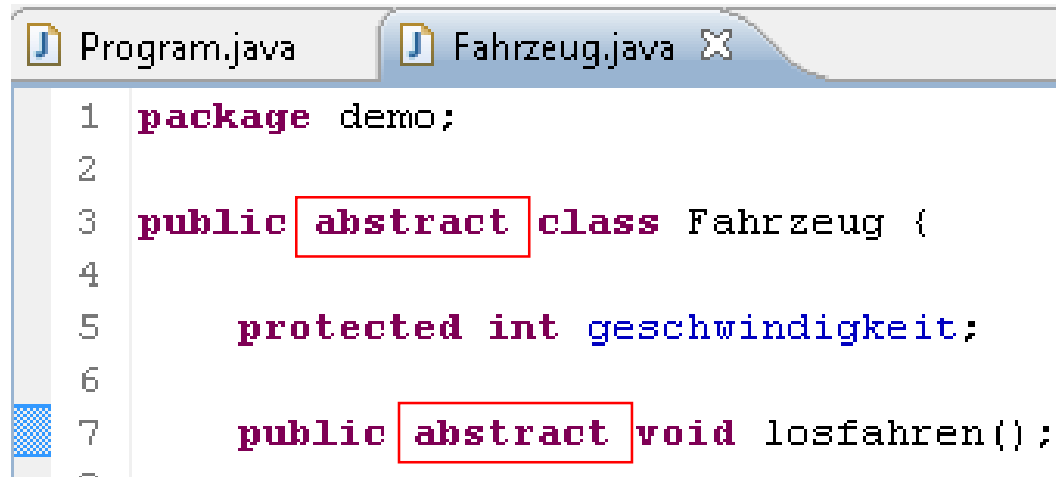
- Methoden werden wie folgt als abstrakte Methoden definiert:



```
1 package demo;
2
3 public abstract class Fahrzeug {
4
5     protected int geschwindigkeit;
6
7     public abstract void losfahren();
8 }
```

- Sie haben keinen Methodenrumpf, d.h. sie werden hier nur definiert aber nicht implementiert. Es fehlen die { }.
- Abstrakte Methoden schreiben vor, dass jede Unterklasse diese Methode implementieren muss („gemeinsames Interface“).

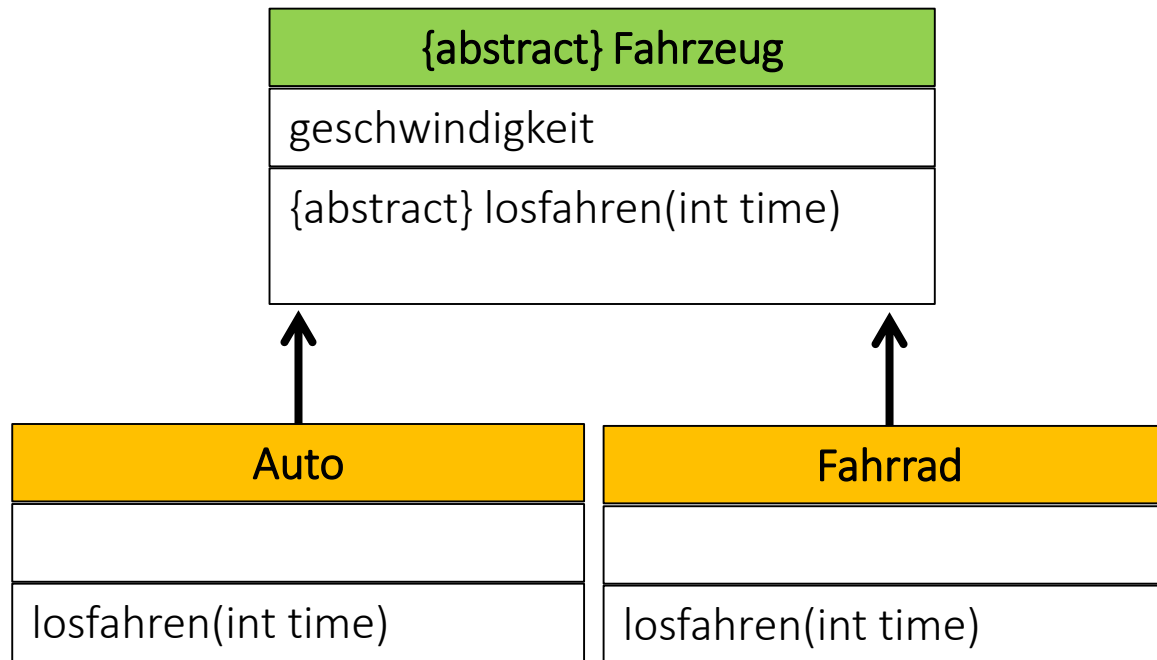
- Zusammenhang zwischen abstrakten Klassen und Methoden:



```
1 package demo;
2
3 public abstract class Fahrzeug {
4
5     protected int geschwindigkeit;
6
7     public abstract void losfahren();
8 }
```

The screenshot shows a code editor with two tabs: 'Program.java' and 'Fahrzeug.java'. The 'Fahrzeug.java' tab is active, displaying the following Java code. The words 'abstract' in the class and method declarations are highlighted with red boxes.

- Eine Klasse, die eine abstrakte Methode enthält, muss selbst abstrakt sein



- Die Methode `losfahren(int time)` muss von allen Kindklassen implementiert werden


```
1 package demo;
2
3 public class Fahrrad extends Fahrzeug {
4
5     public void
6         this.
7     }
8
9 }
10
```

The type Fahrrad must implement the inherited abstract method Fahrzeug.losfahren(int)

2 quick fixes available:

- [Add unimplemented methods](#)
- [Make type 'Fahrrad' abstract](#)

Press 'F2' for focus

Interfaces

- Ein **interface** enthält die Definition von Methodenköpfen (bzw. Methodensignaturen).
- **In keinem Fall enthält es eine Implementierung**
- Beispiel:
 - **public interface** Runnable {
 - **void** run ();
 - }
- Ein Interface legt nur die Anforderungen an eine potentielle Klasse fest, welche diese Schnittstelle implementieren kann.
- Von einem konkreten Objekt einer implementierenden Klasse darf dann erwartet werden, dass es alle im **interface** genannten Anforderungen erfüllt

- Wenn eine Klasse ein **interface** implementiert, dann verhält sie sich wie dieses
- Deswegen werden viele Verwaltungssysteme für **interfaces** geschrieben:
 - Alle Klassen, die dieses **interface** implementieren, können mit so einem System verwaltet werden
- Dieses Verfahren ist grundlegend für die Architektur vieler Java-Systeme

- Wenn eine Klasse eine Schnittstelle implementiert, dann können Objekte dieser Klasse gemäß dieser Schnittstelle benutzt werden, d.h. man kann für solche Objekte die entsprechenden Methoden aufrufen.
- Dabei müssen *alle* im **interface** angegebenen Methoden überschrieben werden

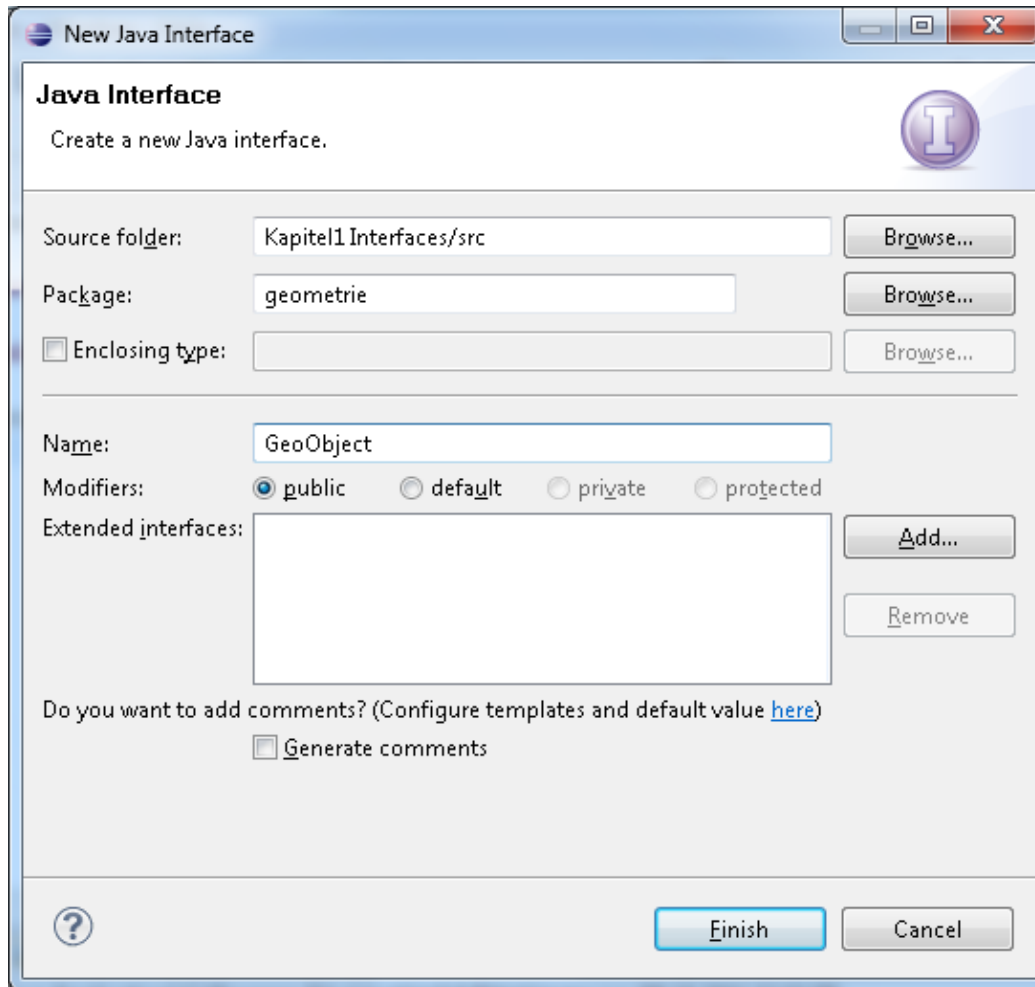
```
public class MeineKlasse implements Runnable {  
    public void run () {  
        // ... Implementierung  
    }  
}
```

- Ein Verwaltungssystem kümmert sich um eine Liste von geometrischen Objekten (z. B. Kreise, Rechtecke, usw.)
- Für jedes Objekt kann es entscheiden, ob dieses einen bestimmten Punkt (x, y) enthält oder nicht
- Das System kann die Objekte zählen, welche einen bestimmten Punkt enthalten

*// Die Schnittstelle ist der Vertrag zwischen
// Implementierung und Anwendung*

```
interface GeoObjekt {  
    public boolean contains (int x, int y);  
    public String getName ();  
}
```

Beispiel: Interfaces mit eclipse anlegen



[Kapitel 1 – Interfaces]

```
GeoObjekt.java  Kreis.java X
2
3 public class Kreis implements GeoObjekt {
4
5     private int r; // r = Radius
6     private int mx, my; // mx, my = Mittelpunkt
7
8     public Kreis(int r, int x, int y) {
9         this.r = r;
10        this.mx = x;
11        this.my = y;
12    }
13
14    public boolean contains(int x, int y) {
15        return (mx - x) * (mx - x) + (my - y) * (my - y) <= r * r;
16    }
17
18    public String getName() {
19        return "Kreis";
20    }
21 }
```

[\[Kapitel 1 – Interfaces\]](#)


```
GeoObjekt.java  Kreis.java  Verwaltungssystem.java  Rechteck.java X
1  package geometrie;
2
3  public class Rechteck implements GeoObjekt {
4
5      private int x, y, b, h; // Linke obere Ecke, Breite, Höhe
6
7      public Rechteck(int x, int y, int b, int h) {
8          this.x = x;
9          this.y = y;
10         this.b = b;
11         this.h = h;
12     }
13
14     public boolean contains(int x, int y) {
15         return this.x <= x && x <= this.x + b && this.y <= y && y <= this.y + h;
16     }
17
18     public String getName() {
19         return "Rechteck";
20     }
21 }
```

[\[Kapitel 1 – Interfaces\]](#)

```
GeoObjekt.java  Kreis.java  Verwaltungssystem.java X
4
5 public class Verwaltungssystem {
6
7     public static void main(String[] args) {
8
9         GeoObjekt[] liste = new GeoObjekt[3];
10        liste[0] = new Kreis(4, 2, 2);
11        liste[1] = new Rechteck(1,1,3,3);
12        liste[2] = new Kreis(1, 2, 2);
13
14        // Zu prüfender Punkt
15        int x = 2;
16        int y = 2;
17
18        int counter = 0;
19
20        for (GeoObjekt obj : liste){
21            // Enthält das Objekt den Punkt?
22            if (obj.contains(x, y)){
23                counter++;
24            }
25        }
```

[Kapitel 1 – Interfaces]

class und interface, extends und implements

class

- enthält Daten und Methoden
- Klasse extends basis
- Klasse erbt Code+Daten von basis
- Mehrfachvererbung **NICHT** möglich. Man kann nur von einer Klasse erben

interface

- enthält nur Methodenköpfe
- Klasse implements basis
- Klasse muss **alle** Methoden von basis implementieren
- Klasse muss die von basis geforderte Schnittstelle implementieren
- Implementieren mehrerer Schnittstellen möglich

Fazit: Zwar sind extends und implements zwei Formen der Vererbung, aber:

- bei extends kann etwas geerbt werden,
- bei implements erbt man nur die Vorschrift, sich gemäß der Schnittstelle zu verhalten

enum

In Java kann man Konstanten in der folgenden Form definieren.

```
public static final int ROT      = 0;
public static final int GRUEN   = 1;
public static final int GELB    = 2;
public static final int MONTAG  = 1;
int farbe = ROT;    // Zuweisung ist sinnvoll
farbe = MONTAG;    // ??? Missverständnis
                  // Für den Compiler ist das kein Problem!
```

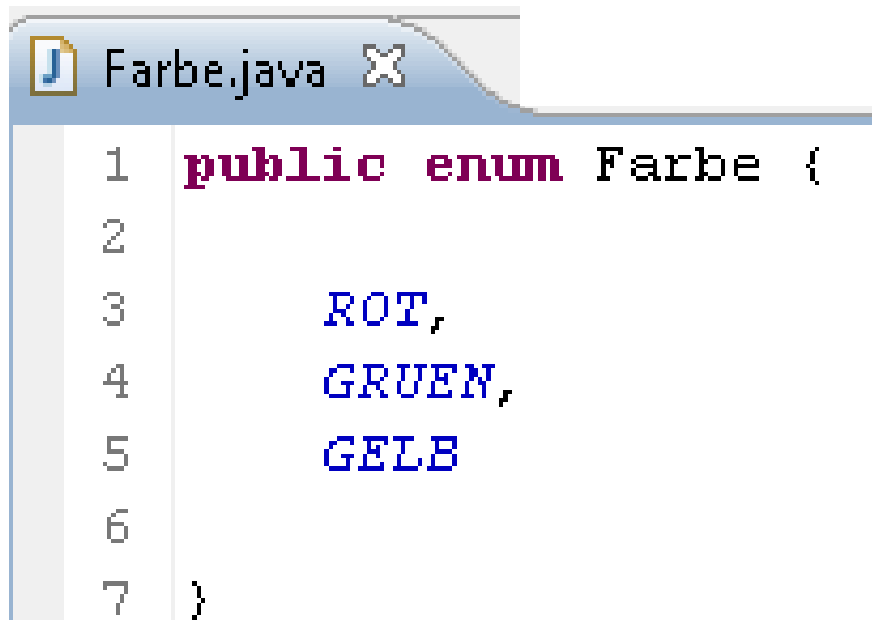
Die zweite Wertzuweisung (rot) in obigem Programmabschnitt illustriert das Problem:

- Konstanten dieser Art sind nicht typgebunden
- Durch dieses Loch im Sicherheitsnetz der Typüberprüfung können Fehler schlüpfen, die erst zur Laufzeit des Programms sichtbar werden

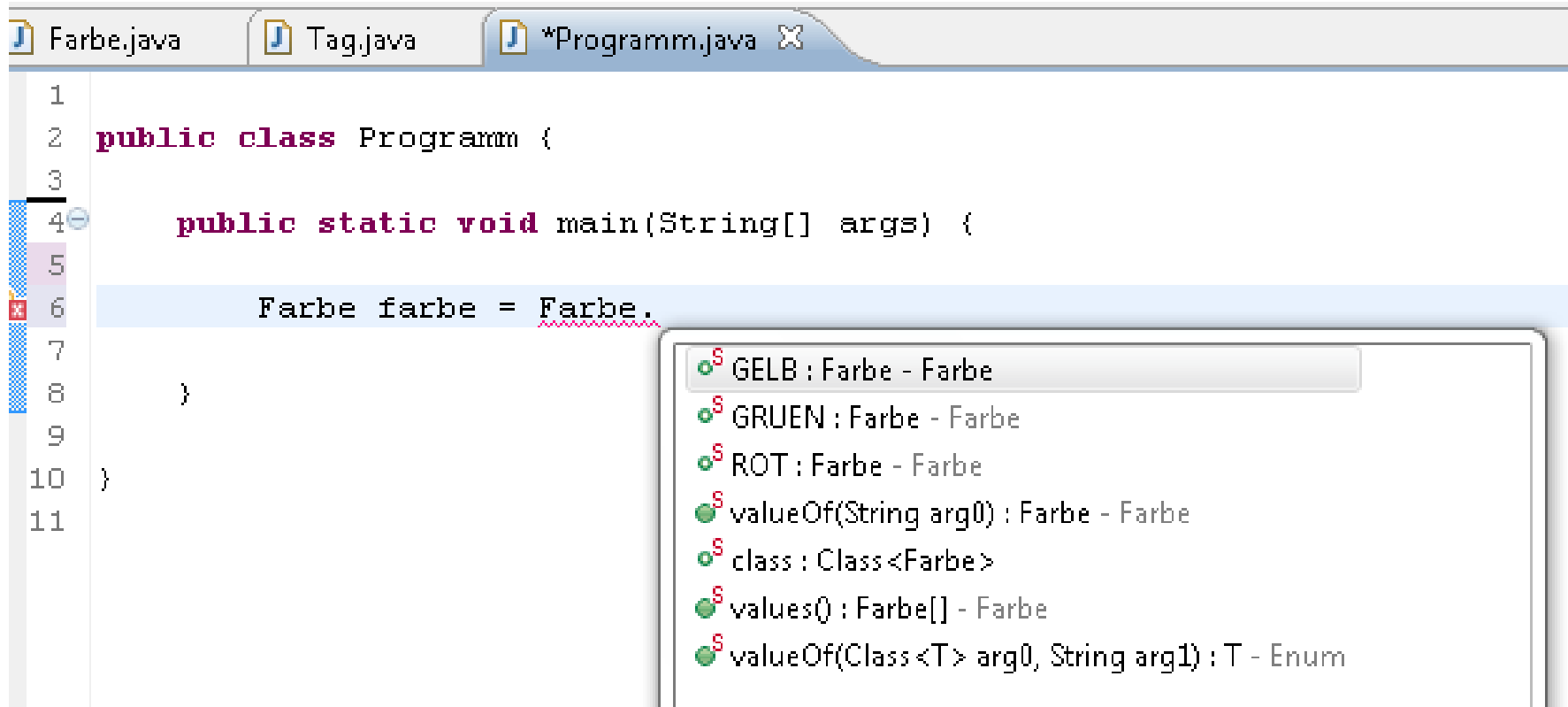
Ein Tester kann mit reinen **int-Daten** auch nicht auf die vom Programmierer gedachte Bedeutung **einer Konstante schließen**

- Was sollte der Test bei farbe ausgeben?

- Eine Aufzählung mit 3 konstanten Werten



```
1 public enum Farbe {  
2  
3     ROT,  
4     GRUEN,  
5     GELB  
6  
7 }
```

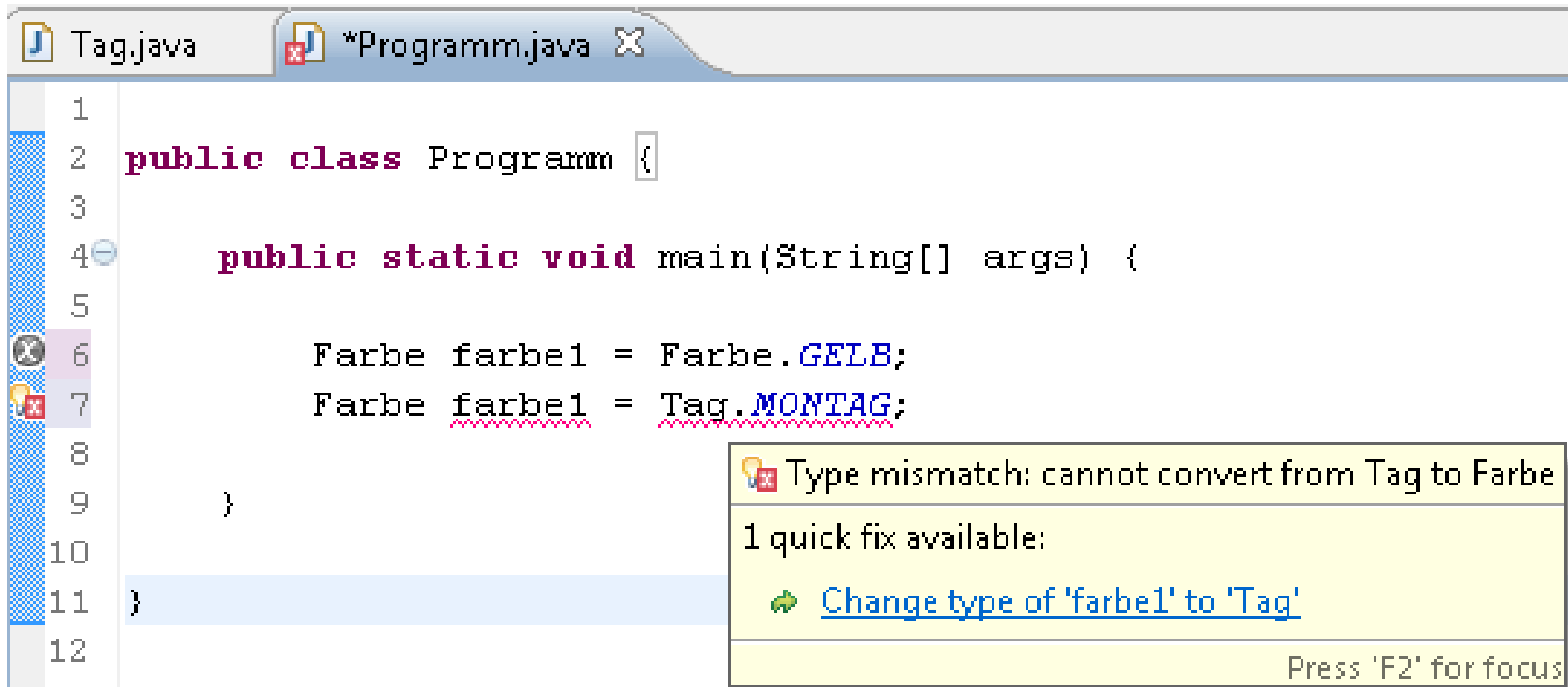


```
1  
2 public class Programm {  
3  
4     public static void main(String[] args) {  
5  
6         Farbe farbe = Farbe.  
7  
8     }  
9  
10 }  
11
```

Farbe

- GELB : Farbe - Farbe
- GRUEN : Farbe - Farbe
- ROT : Farbe - Farbe
- valueOf(String arg0) : Farbe - Farbe
- class : Class<Farbe>
- values() : Farbe[] - Farbe
- valueOf(Class<T> arg0, String arg1) : T - Enum

- Typsicherheit ist gewährleistet (vgl. Nutzung von int-Konstanten)



```
1
2 public class Programm {
3
4     public static void main(String[] args) {
5
6         Farbe farbe1 = Farbe.GELB;
7         Farbe farbe1 = Tag.MONTAG;
8
9     }
10
11 }
12
```

Type mismatch: cannot convert from Tag to Farbe
1 quick fix available:
[Change type of 'farbe1' to 'Tag'](#)
Press 'F2' for focus

enum mit Konstruktoren und Methoden

```
Tag.java X
1  public enum Tag {
2
3      MONTAG (1),
4      DIENSTAG (2),
5      MITTWOCH (3),
6      DONNERSTAG (4),
7      FREITAG (5),
8      SAMSTAG (6),
9      SONNTAG (7);
10
11     private int ordnung;
12
13     Tag(int ordnung) {
14         this.ordnung = ordnung;
15     }
16
17     public boolean istVor(Tag other) {
18         if (this.ordnung < other.ordnung) {
19             return true;
20         }
21         return false;
22     }
23 }
```

[Kapitel 1 – Enum]

```
Tag tag = Tag.MITTWOCH;
```

```
switch (tag) {  
    case MONTAG:  
        System.out.println("Mondays are bad.");  
        break;  
  
    case FREITAG:  
        System.out.println("Fridays are better.");  
        break;  
  
    case SAMSTAG:  
    case SONNTAG:  
        System.out.println("Weekends are best.");  
        break;  
  
    default:  
        System.out.println("Midweek days are so-so.");  
        break;  
}
```

[Kapitel 1 – Enum]