

# 19) Weitere Java-Konstrukte (zum Selbststudium)

Prof. Dr. rer. nat. Uwe Aßmann  
Institut für Software- und  
Multimediatechnik  
Lehrstuhl Softwaretechnologie  
Fakultät für Informatik  
TU Dresden  
Version 15-0.1, 5/4/15

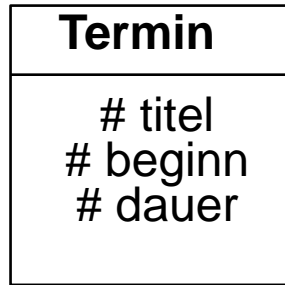
- 1) Bemerkungen zum Selbststudium
- 2) Sichtbarkeiten
- 3) Operationen
  - 1) Konstruktoren
- 4) Ausnahmen
- 5) super
- 6) Casts

- ▶ Obligatorisch:
  - Balzert, verschiedene Abschnitte
  - Boles Kap. 8, 13, 14
- ▶ The Java Language Reference Manual.
- ▶ Freies Java Buch (leider nur Version 1.1, für Grundlagen):
  - [http://www.computer-books.us/java\\_8.php](http://www.computer-books.us/java_8.php)
  - <http://www.computer-books.us/java.php>
- ▶ Kommunikationsdiagramme:
  - <http://www.agilemodeling.com/essays/umlDiagrams.htm>
  - <http://www.agilemodeling.com/artifacts/communicationDiagram.htm>
  - [http://en.wikipedia.org/wiki/Communication\\_diagram](http://en.wikipedia.org/wiki/Communication_diagram)
  - [http://www.sparxsystems.com.au/resources/uml2\\_tutorial/uml2\\_communicationdiagram.html](http://www.sparxsystems.com.au/resources/uml2_tutorial/uml2_communicationdiagram.html)

# Hinweis: Online-Ressourcen!

- ▶ Über die Homepage der Lehrveranstaltung (bei "Vorlesungen") finden Sie die Dateien
  - `Terminv.java`
  - `TaxDeclarationDemo.java`
- ▶ Diese Dateien enthält eine vollständige Umsetzung der fortlaufenden Beispiele "Terminverwaltung" bzw. „Steuererklärung“ in lauffähigen Java-Code.
- ▶ Empfohlene Benutzung:
  - Lesen
  - Übersetzen, Starten, Verstehen
  - Modifizieren
  - Kritisieren

# Laufendes Beispiel Terminverwaltung



```
class Termin {
    ...
    protected String titel;
    protected Hour beginn;
    protected int dauer;
    ...
};
```



```
class Teambesprechung
    extends Termin {
    private Teammitglied[] teilnahme;
    private BesprRaum veranstOrt;
    ...
};
```

# 19.1 Hinweis: Material zum Selbststudium

- ▶ Die folgenden Folien enthalten Material zur Java-Programmierung für das Selbststudium.
- ▶ Das Material wird in den Übungen an verschiedenen Stellen entfaltet
- ▶ Bitte stellen Sie sicher, dass Sie diese Folieninhalte beherrschen

- ▶ Literal
- ▶ Variable, spezielle Variable `this`
- ▶ Operator in Ausdruck
- ▶ Attributzugriff `o.a` / `super.a` / `this.a`
- ▶ Methodenaufruf `o.m(...)` / `super.m(...)` / `this.m(...)`
- ▶ Array-Zugriff `a[i]` / `a[i][j]` / ...
- ▶ Konstruktoraufruf `new <ClassName> (<parameterList>)`
- ▶ Arrayinstanziierung `new <BasisTypName> [<n>]`
- ▶ Konditionalausdruck `<BoolExp> ? <Exp1> : <Exp2>`
- ▶ Cast-Ausdruck `(<TypName>) <Exp>`

# Java im Selbststudium

## Anweisungen (1) <Statement>

- ▶ Variablendeklaration `<Typ> <variable>;`
  - mit Wertsemantik (für primitive Typen)
  - mit Referenzsemantik für Referenztypen (Klassen, Interfaces, Enumerations)
- ▶ Methodenaufruf mit Semikolon `o.m(...);`
- ▶ Konstruktoraufruf mit Semikolon  
`new <ClassName> (<parameterList>);`
- ▶ Zuweisung `<variable> = <wert>;`
- ▶ Leere Anweisung `;`
- ▶ Block `{<StatementList> }`

- `if (<BoolExp>) <Statement1> else <Statement2>`
- `switch (<Exp>) {  
    case <Exp1> : <StatementList1>  
    ...  
    default: <StatementList>`
- `while (<BoolExp>) <Statement>`
- `for (<InitExp>; <BoolExp>; <UpdateExp>) <Statement>`
- `break [<label>];`
- `return [<Exp>];`
- `try <TryBlock>  
    catch (<formalParam1>) <CatchBlock1>  
    ...  
    finally <FinallyBlock>`

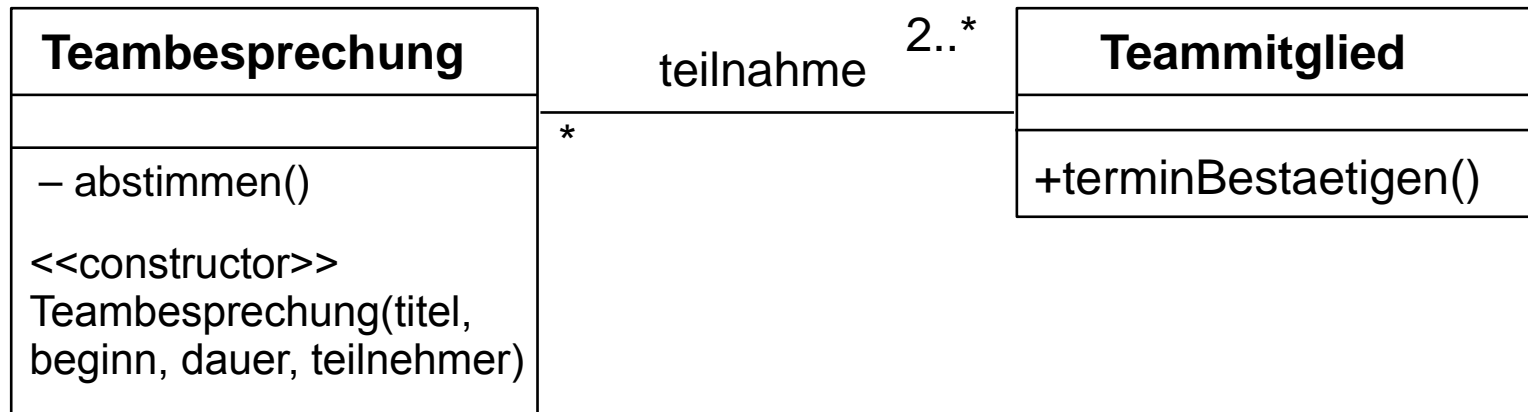


# 19.3 Operationen



... auch Methoden genannt...

# Beispiel Methodenrumpf (1)

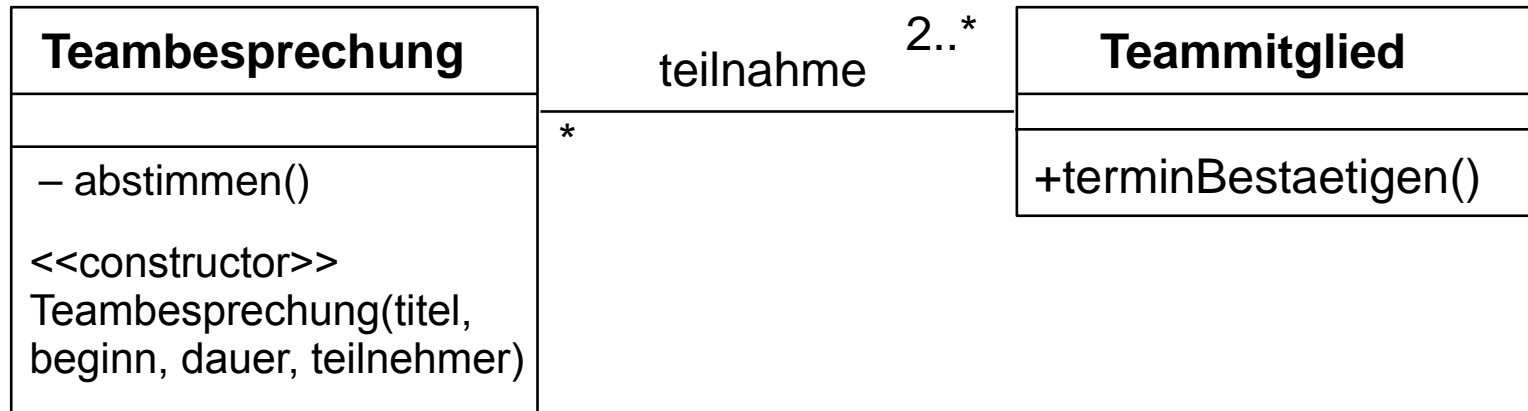


```
class Teambesprechung {

    private Teammitglied[] teilnahme; ...

    private boolean abstimmen (Hour beginn, int dauer) {
        boolean ok = true;
        for (int i=0; i<teilnahme.length; i++)
            ok = ok &&
                teilnahme[i].terminBestaetigen(beginn, dauer);
        return ok;
    }
}
```

# Beispiel Methodenrumpf (2) – Konstruktor für Netzaufbau

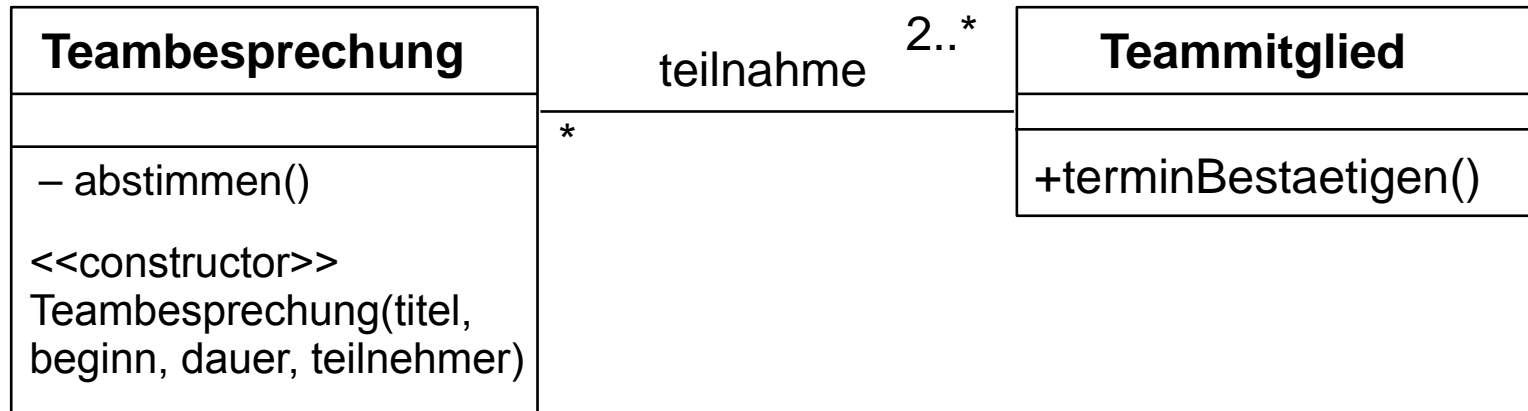


```
class Teambesprechung {

    private Teammitglied[] teilnahme; ...
    private boolean abstimmen (Hour beginn, int dauer) ...

    public Teambesprechung (String titel,
        Hour beginn, int dauer, Teammitglied[] teilnehmer) {
        // ... titel, beginn, dauer lokal speichern
        this.teilnahme = teilnehmer;
        if (! abstimmen(beginn, dauer))
            System.out.println("Termin bitte verschieben!");
        else ...
    } ...
}
```

# Beispiel Methodenrumpf (3) – öffentl. Methode



```
class Teammitglied { ...
    private Teambesprechung[] teilnahme;

    public boolean terminBestaetigen (Hour beginn,int dauer){
        boolean konflikt = false;
        int i = 0;
        while (i < teilnahme.length && !konflikt) {
            if (teilnahme[i].inKonflikt(beginn, dauer))
                konflikt = true;
            else
                i++;
        };
        return !konflikt;
    } ...
}
```

# 19.4 Ausnahmen (Exceptions)



# Ausnahmebehandlung in Java

## ▶ Ausnahme (*Exception*):

- Objekt einer Unterklasse von `java.lang.Exception`
- Vordefiniert oder und selbstdefiniert

## ▶ Ausnahme

### ▪ **auslösen** (*to throw an exception*)

- Erzeugen eines Exception-Objekts
- Löst Suche nach Behandlung aus

### ▪ **abfangen und behandeln** (*to catch and handle an exception*)

- Aktionen zur weiteren Fortsetzung des Programms bestimmen

### ▪ **deklarieren**

- Angabe, daß eine Methode außer dem normalen Ergebnis auch eine Ausnahme auslösen kann (Java: **throws**)
- Beispiel aus `java.io.InputStream`:

```
public int read() throws IOException;
```

# Java-Syntax für Ausnahmebehandlung

Schnittstelle

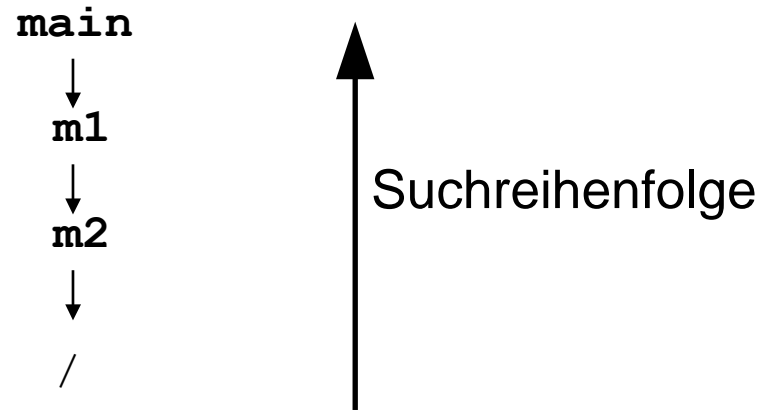
```
class TotalDiv {  
    public static int tDiv (int x, int y)  
        throws ArithmeticException {  
        try {  
            return (x / y);  
        }  
        catch (ArithmeticException e) {  
            System.err.println(„Division by zero“);  
            throw new ArithmeticException();  
        }  
    }  
}
```

Gekapselter  
Block

Ausnahme-  
Fänger

# Dynamische Suche nach Ausnahmebehandlung

- Suche nach Abfangklausel (catch) entlang der (dynamischen) Aufrufhierarchie:



- Bei mehreren Abfangklauseln an der gleichen Stelle der Hierarchie gilt die zuerst definierte Klausel:

```
try {  
  catch (xException e)  
  catch (yException e)
```

The diagram shows a code snippet with two catch clauses. To the right of the code, a vertical arrow points downwards, labeled "Suchreihenfolge" (search order), indicating that the search for an exception handler proceeds from the first (top) catch clause to the second (bottom) catch clause.



# Definition neuer Ausnahmen

Benutzung von benutzerdefinierten Ausnahmen möglich und empfehlenswert !

```
class TestException extends Exception {
    public TestException () {
        super ();
    }
}
class SpecialAdd {
    public static int sAdd (int x, int y)
        throws TestException {
        if (y == 0)
            throw new TestException ();
        else
            return x + y;
        }
}
```

# Deklaration und Propagation von Ausnahmen

- ▶ Wer eine Methode aufruft, die eine Ausnahme auslösen kann, muß
  - entweder die Ausnahme abfangen
  - oder die Ausnahme weitergeben (*propagieren*)
- ▶ Propagation in Java: Deklarationspflicht mittels **throws** (außer bei Error und RuntimeException)

```
public static void main (String[] argv) {  
    System.out.println (SpecialAdd.sAdd (3, 0)) ;  
}
```

Java-Compiler: Exception TestException must be caught, or it must be declared in the throws clause of this method.

# Bruch von Verträgen und Ausnahmen

- ▶ Man kann Verträge auch mit Ausnahmetests prüfen:

```
class ContractViolation {..};  
class ParameterContractViolation extends ContractViolation {..};  
class FigureEditor{  
    draw (Figure figure) throws ContractViolation {  
        if (figure == null) throw new ParameterContractViolation();  
    }  
}
```

- ▶ im Aufrufer:

```
try {  
    editor.draw(fig);  
} catch (ParameterContractViolation) {  
    fig = new Figure();  
    editor.draw(fig);  
}
```

- ▶ Vorteil: kontrollierte Reaktion auf Vertragsbrüche.

# Regeln zum Umgang mit Ausnahmen

- ▶ Gesetz des pragmatischen Programmierers 58: **Bauen Sie die Dokumentation ein**
  - Ausnahmebehandlung niemals zur Behandlung normaler (d.h. häufig auftretender) Programmsituationen einsetzen
  - Ausnahmen sind Ausnahmen, regulärer Code behandelt die regulären Fälle!
- ▶ Gesetz 34: **Verwenden Sie Ausnahmen nur ausnahmsweise**
  - Nur die richtige Dosierung des Einsatzes von Ausnahmen ist gut lesbar
- ▶ Gesetz 35: **Führen Sie zu Ende, was Sie begonnen haben**
  - Auf keinen Fall Ausnahmen “abwürgen”, z.B. durch triviale Ausnahmebehandlung
  - Ausnahmen zu propagieren ist keine Schande, sondern erhöht die Flexibilität des entwickelten Codes.
- ▶ Gesetz 33 **Verhindern Sie das Unmögliche mit Zusicherungen**
  - Vertragsüberprüfungen, generieren Ausnahmen

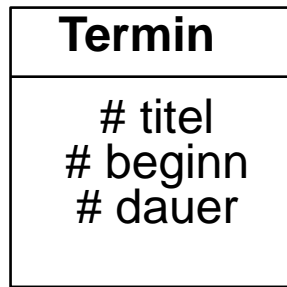
# Wiederholung: Dynamische Suchalgorithmen in der JVM

- ▶ Merkmalsuchalgorithmus
  - Die JVM sucht dynamisch nach Methoden und Attributen entlang der Vererbungshierarchie. Diese ist in der .class-Datei abgespeichert und wird darin abgesucht
  - Verwendung für Polymorphie und Reflektion
- ▶ Ausnahmenbehändlersuche
  - Die JVM sucht dynamisch den Aufrufkeller nach oben ab, ob zur ausgelösten Ausnahme ein Behandler definiert

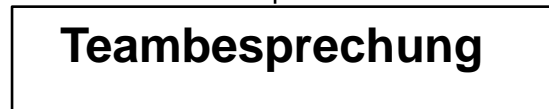
# 19.5 Feinheiten der Vererbung



# Vererbung und Assoziation zusammen



```
class Termin {
    ...
    protected String titel;
    protected Hour beginn;
    protected int dauer;
    ...
};
```



```
class Teambesprechung
    extends Termin {

    private Teammitglied[] teilnahme;
    private BesprRaum veranstOrt;
    ...
};
```

# Aufruf von Oberklassen-Konstruktoren durch super-Konstruktoraufruf

- ▶ "super" ermöglicht Aufruf von Methoden, die in der Oberklasse definiert und *überschattet* sind.
- ▶ Ein Super-Konstruktoraufruf ruft den Konstruktor der Oberklasse auf und kann damit alle Aktionen dessen übernehmen

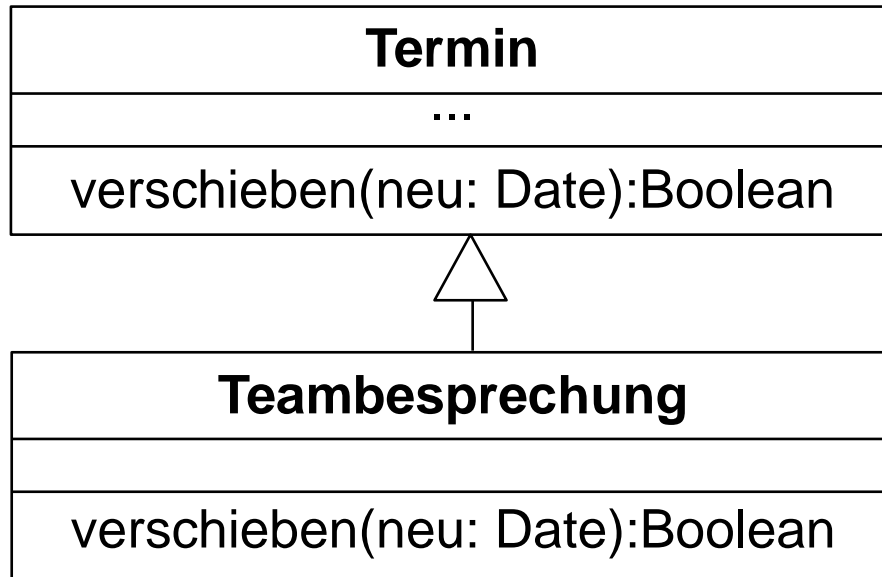
```
class Termin {  
    protected String titel;  
    protected Hour beginn;  
    protected int dauer;  
    ...  
    public Termin  
        (String t, Hour b, Dauer d) {  
        titel = t; beginn = b; dauer = d; };  
};
```

Spezielle Syntax  
für Konstruktoren:  
"super" immer als  
erste Anweisung!

```
class Teambesprechung extends Termin {  
    ...  
    private Teammitglied[] teilnahme;  
    ...  
    public Teambesprechung (String t, Hour b, Dauer d,  
        Teammitglied[] tn) {  
        super(t, b, d);  
        teilnahme = tn; ... };  
};
```



# "super" in überschriebenen Operationen



```
class Termin {  
  
    boolean verschieben  
        (neu: Date) {  
        ...z.B. Aufzeichnung  
            in Log-datei  
        }  
    ...  
}
```

```
class Teambesprechung  
    extends Termin  
  
    boolean verschieben  
        (neu: Date) {  
        ...  
        super.verschieben (neu)  
    }  
    ...  
}
```

"super"  
ermöglicht auch den Zugriff auf  
überschriebene Versionen von  
Operationen, wie sie in einer  
Oberklasse definiert sind.  
  
Andere Syntax als bei Konstruktoren!

# 19.6 Casts



# Problem: Typanpassungen (Casts) mit konkreten Datentypen, Bsp: Geordnete Listen mit ArrayList

```
import java.util.ArrayList;
...
class Bestellung {
    private String kunde;
    private ArrayList liste;
    private int anzahl = 0;

    public Bestellung(String kunde) {
        this.kunde = kunde;
        this.liste = new ArrayList();
    }

    public void neuePosition (Bestellposition b) {
        liste.add(b);
    }

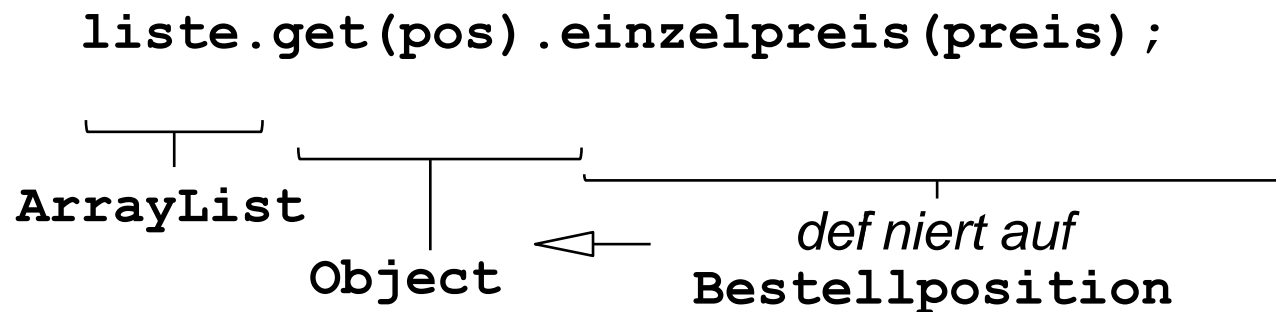
    public void loeschePosition (int pos) {
        liste.remove(pos);
    }
    ...
}
```

# Anwendungsbeispiel mit ArrayList (falsch!)

```
...  
public void sonderpreis (int pos, int preis) {  
    liste.get(pos).einzelpreis(preis);  
}  
...
```

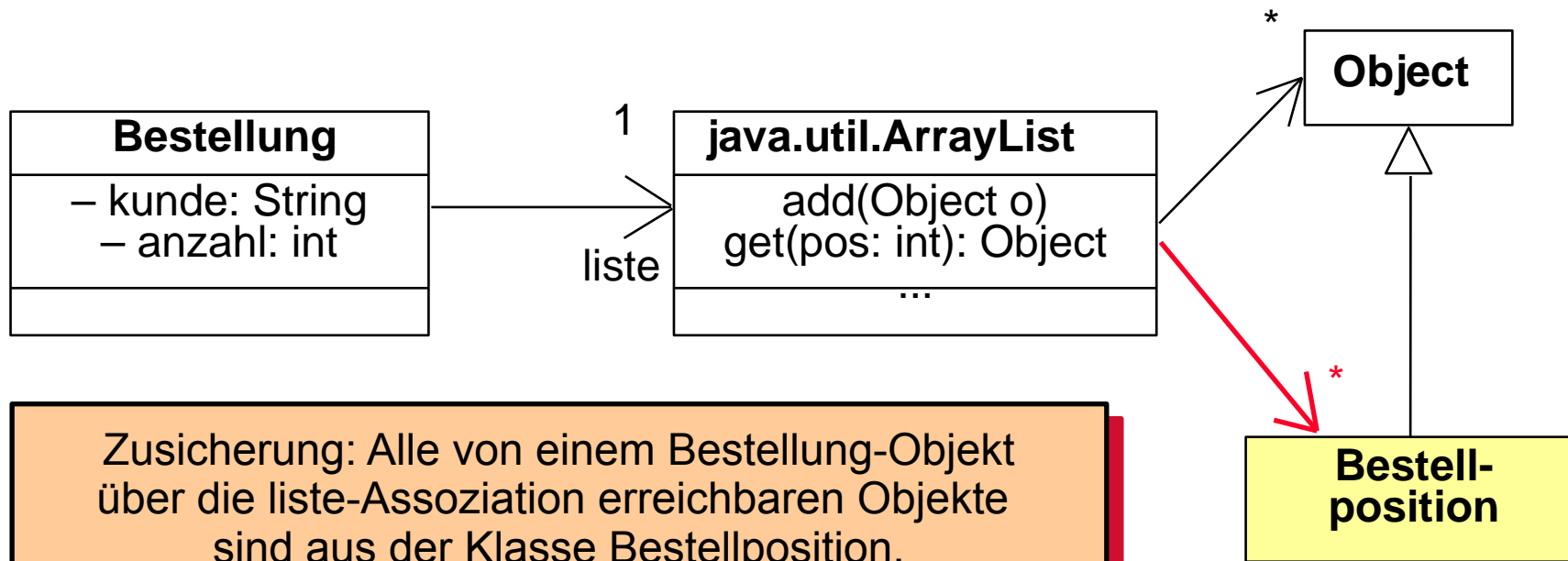
- ▶ Compilermeldung:

„Method einzelpreis(int) not found in class java.lang.Object.“



Spezialisierung von Object auf Bestellposition?

# Typanpassungen auf Elementtypen



## Typanpassung (cast):

- Operationen der Oberklasse passen immer auch auf Objekte der Unterklasse
- Operationen der Unterklasse auf Objekte einer Oberklasse anzuwenden, erfordert explizite Typanpassung (*dynamic cast*):

( *Typ* ) Objekt

hier: (Bestellposition) liste.get(pos)

# Cast im Anwendungsbeispiel mit ArrayList

```
public void sonderpreis (int pos, int preis) {
    ((Bestellposition)liste.get(pos)).einzelpreis(preis);
}

public int auftragssumme() {
    int s = 0;
    for(int i=0; i<liste.size(); i++)
        s +=
            ((Bestellposition)liste.get(i)).positionspreis();
    return s;
}

public void print () {
    System.out.println("Bestellung fuer Kunde "+kunde);
    for(int i=0; i<liste.size(); i++)
        System.out.println(liste.get(i));
    System.out.println("Auftragssumme: "+auftragssumme());
    System.out.println();
}
}
```

Online:  
Bestellung1.java

Anwendungsbeispiel  
mit LinkedList:

Online:  
Bestellung3.java

# Was haben wir gelernt?

- ▶ Kapselung und Modularisierung werden mit Sichtbarkeiten möglich
- ▶ Spezielle Variablen (this, super) erlauben den Zugriff auf Attribute und überschriebene Methoden/Operationen
- ▶ Ausnahmen für Ausnahmen
- ▶ Kommunikationsdiagramme ordnen die Methodenaufrufe über der Fläche an und zeigen die Zeit durch Nummerierung an
- ▶ Klassenoperationen von Objektoperationen unterscheiden