Vererbung von Konstruktoren

- Aufruf-Reihenfolge
- Impliziter Konstruktor-Aufruf
- Expliziter Konstruktor-Aufruf
- Exkurs: Polymorphismus

Konstruktoren und Vererbung

- In einer Klassenhierarchie werden in einem Konstruktor ***immer*** *zuerst* ***automatisch*** die *parameterlosen Konstruktoren* der obersten Klasse(n) aufgerufen
- Dieser Aufruf erfolgt *immer*, es gibt keine Möglichkeit, ihn zu vermeiden (man kann ihn höchstens zu einem anderen Konstruktor „umbiegen“)

```
public class Oben {

    Oben() {
        println("Oben-Konstruktor");
    }

}
```

- Weiteres Beispiel
  auf der nächste Folie

```
public class Mitte extends Oben {

    Mitte() {
        println("Mitte-Konstruktor");
    }
}
```

```
public class Unten extends Mitte {

    Unten() {
        println("Unten-Konstruktor");
    }

}
```

Veranschaulichung

```
public class Object {
    Object() {
        // Belegen von Speicherplatz etc.
    }
} // Object
```

```
public class Oben {

    Oben() {
        println("Oben-Konstruktor");
    }

}
```

```
public class Mitte extends Oben {

    Mitte() {
        println("Mitte-Konstruktor");
    }
}
```

```
public class Unten extends Mitte {

    Unten() {
        println("Unten-Konstruktor");
    }

}
```

```
new Unten();
```

Ausgabe des Programms:

Oben-Konstruktor

Mitte-Konstruktor

Unten-Konstruktor

Konstruktoren und Vererbung: Aufrufreihenfolge

- Das bedeutet, der Rumpf des Konstruktors der *obersten Klasse* in der Hierarchie wird *zuerst* ausgeführt, dann derjenige des nächstniedrigeren usw.
- Sinn dieser Reihenfolge ist, den Klassen *in der Reihenfolge, in der sie konstruiert wurden*, Gelegenheit zu geben, nach Bedarf ihre Felder zu *initialisieren*, Dateien zu öffnen, Netzwerkverbindungen aufzubauen etc.
- Im Konstruktor einer Klasse kann man sich also darauf verlassen, dass alle notwendigen Initialisierungen aller (direkten und indirekten) Oberklassen abgeschlossen sind, bevor die eigenen Anweisungen durchgeführt werden

Der *implizite* Aufruf *parameterloser* Konstruktoren 1/2

- Automatisch aufgerufen werden immer *parameterlose* Konstruktoren der Oberklasse

```
class Mietwagen {
    Mietwagen () {
     ...
    }
    Mietwagen ( String hersteller ) {
     ...
    }
} // Mietwagen

class Bus extends Mietwagen {
    Bus ( String hersteller ) {
     ...
    }
} // Bus
```

Es wird ***nicht*** etwa dieser Konstruktor aufgerufen (der ja vom Parameter ganz gut passen würde...)

An dieser Stelle findet ein (impliziter) Konstruktor-Aufruf statt

Der *implizite* Aufruf *parameterloser* Konstruktoren 2/2

- Der Compiler überprüft, ob ein parameterloser Konstruktor in der direkten Oberklasse existiert
- Existiert kein solcher Konstruktor, meldet der Compiler einen entsprechenden Fehler

"Bus.java": Implicit super constructor Mietwagen() is undefined.
Must explicitly invoke another constructor

- Beispiel

```
class Mietwagen {
    Mietwagen ( String hersteller ) {
     ...
    }

} // Mietwagen

class Bus extends Mietwagen {
    Bus ( String hersteller ) {
     ...
    }
} // Bus
```

?

An dieser Stelle findet ein (impliziter) Konstruktor-Aufruf statt

Der explizite Aufruf von Konstruktoren

- Wie bereits beschrieben, gibt es *keine Möglichkeit*, den Aufruf eines Konstruktors der direkten Oberklasse *zu vermeiden*
- Es gibt jedoch eine Möglichkeit, den Konstruktor-Aufruf auf einen anderen Konstruktor mit Parametern "umzubiegen"
- Dazu wird, wie beim Aufruf überdeckter Methoden, das Schlüsselwort ***super*** benutzt (s. Bsp. *Versicherter* / Student)
- Der *explizite Aufruf* mittels ***super*** *ersetzt* dann den *impliziten Aufruf*
- Der explizite Aufruf mittels ***super*** muss *als Erstes* im Konstruktor stehen, *wo auch der implizite Aufruf stattfinden würde*, um die *Ablaufreihenfolge* von Konstruktoren in einer Klassenhierarchie *einzuhalten*

Beispiel

```java
public class Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;

    public Versicherter( String name, String vorname, int mitgliedsnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.mitgliedsnummer = mitgliedsnummer;
    }
```

```java
public class Rentner extends Versicherter {

    private float jahreAktiverMitgliedschaft;

    public Rentner( String name, String vorname, int mitgliedsnummer,
            float jahreAktiverMitgliedschaft ) {
        super(name, vorname, mitgliedsnummer);
        this.jahreAktiverMitgliedschaft = jahreAktiverMitgliedschaft;
    }
```

```java
public class Student extends Versicherter {

    private String hochschule;
    private int matrikelnummer;

    public Student( String name, String vorname, int mitgliedsnummer,
            String hochschule, int matrikelnummer ) {
        super(name, vorname, mitgliedsnummer);
        this.hochschule = hochschule;
        this.matrikelnummer = matrikelnummer;
    }
```

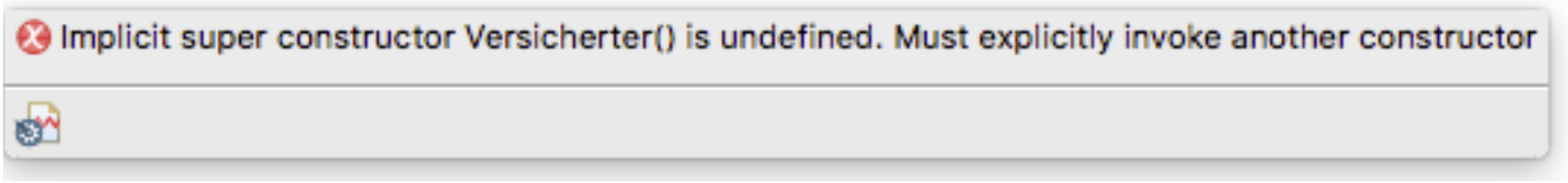
Beispiel

```
public class Versicherter {

    private String name, vorname;
    private int mitgliedsnummer;

    public Versicherter( String name, String vorname, int mitgliedsnummer ) {
        this.name = name;
        this.vorname = vorname;
        this.mitgliedsnummer = mitgliedsnummer;
    }
```

```
public Student( String name, String vorname, int mitgliedsnummer,
        String hochschule, int matrikelnummer ) {
    this.hochschule = hochschule;
    this.matrikelnummer = matrikelnummer;
}
```



Ausblick: Polymorphismus

- In Java gibt es ***überladene*** Konstruktoren (und überladene Methoden)
  - *Eine Klasse* kann mehrere *Konstruktoren* enthalten, sofern diese sich durch die *Anzahl und/oder die Typen ihrer formalen Parameter unterscheiden*
  - Wir kennen das schon von (gleichnamigen) Methoden, zum Beispiel:
    → *println* in der Klasse *MakeItSimple*
  - Bedingung:
    Die Konstruktoren (Methoden) müssen anhand der aktuellen Parameterliste unterscheidbar sein
  - Der *Compiler* bestimmt anhand der Anzahl und des Typs der aktuellen Parameter, welcher Konstrukor (welche Methode) zur Ausführung kommen kann