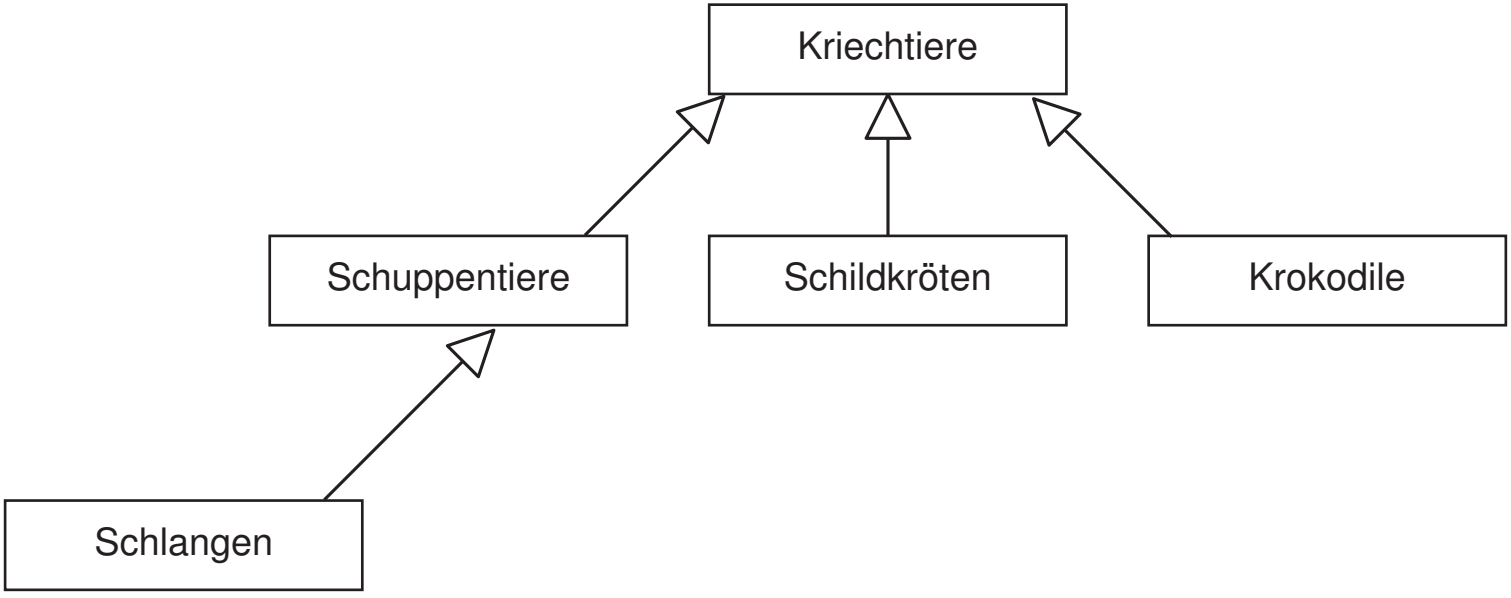
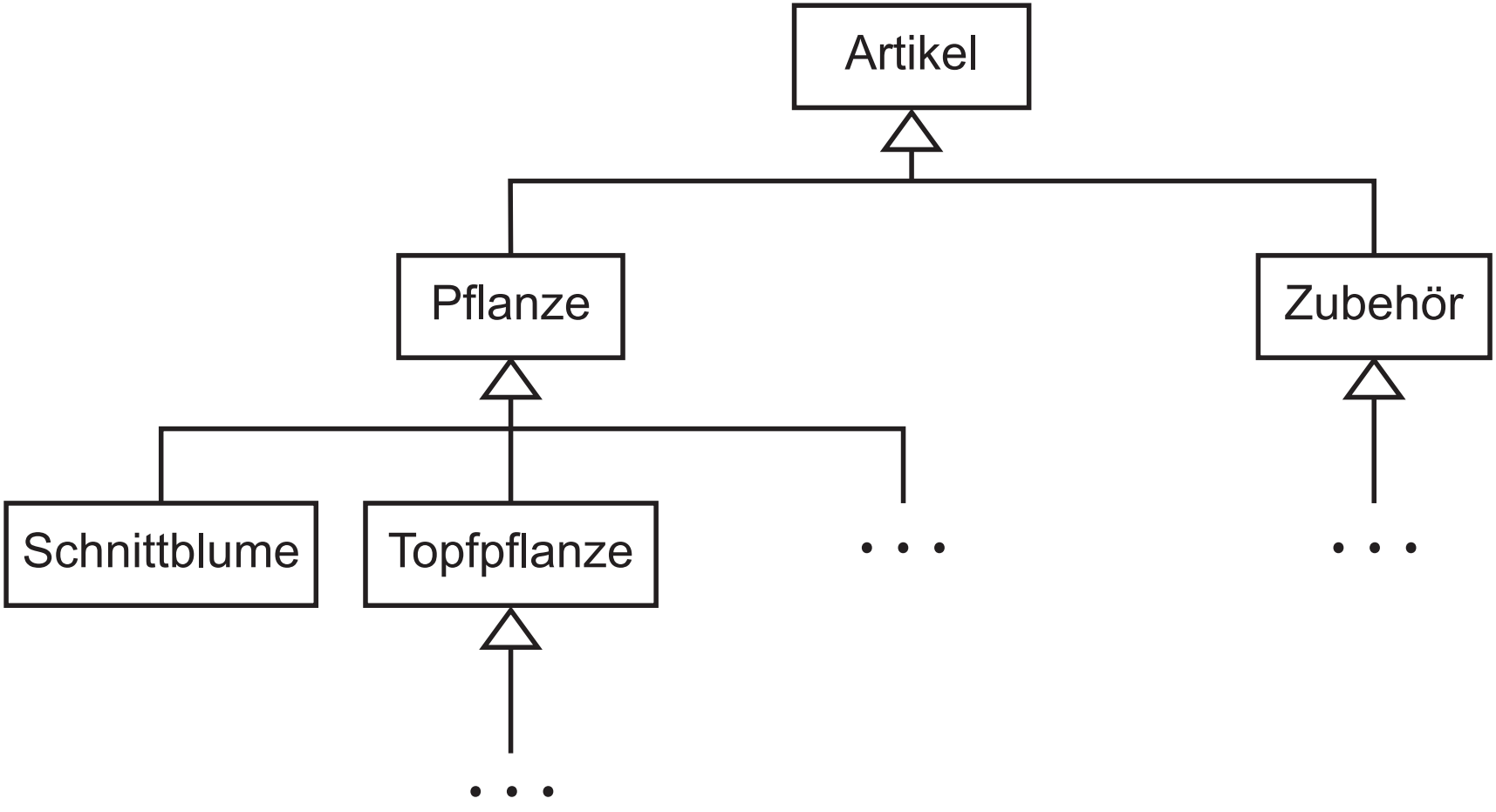
Einführung in die Objektorientierte Programmierung
Vorlesung 9: Vererbung

Sebastian Küpper

Spezialisierung / Generalisierung in der Biologie — Taxonomien



Spezialisierung / Generalisierung von Klassen: Unter- und Oberklasse



Beispiel: Vererbung in Java

```
class Artikel {
   double preis;
   // ...
   double lieferePreis() {
      return this.preis;
   }
   void legePreisFest(final double neuerPreis) {
      this.preis = neuerPreis;
   }
   // ...
}
class Pflanze extends Artikel {
}
```

Beachte: `Pflanze` ist keine leere / nutzlose Klasse

Spezialisierung erhält Funktionalität

Die folgende Anweisungsfolge ist legal:

```
Pflanze p = new Pflanze();
p.legePreisFest(20.0);
double preis = p.preis; // preis hat den Wert 20.0
preis = p.lieferePreis(); // preis hat auch den Wert 20.0
```

Spezialisierung erlaubt neue Funktionalität

```
class Pflanze extends Artikel {
   double lagertemperatur;

   double liefereLagertemperatur() {
      return this.lagertemperatur;
   }
   void legeLagertemperaturFest(final double temp) {
      this.lagertemperatur = temp;
   }
}
```

Neue Funktionalität ist nur in Unterklasse aufrufbar

```
Pflanze p = new Pflanze();
p.legePreisFest(10.0);
p.legeLagertemperaturFest(21.0);
Artikel a = new Artikel();
a.legePreisFest(10.0); // ist moeglich
a.legeLagertemperaturFest(15.0); // ist nicht moeglich,
//  da die Klasse Artikel diese Methode nicht besitzt.
```

Spezialisierung erlaubt Subsitution

```
Artikel a = new Pflanze();
a.legePreisFest(10.0); // ist moeglich
a.legeLagertemperaturFest(15.0); // ist nicht moeglich, da
// die Klasse Artikel diese Methode nicht besitzt.
```

Spezialisierung erlaubt Subsitution

```
Artikel a = new Pflanze();
a.legePreisFest(10.0); // ist moeglich
a.legeLagertemperaturFest(15.0); // ist nicht moeglich, da
// die Klasse Artikel diese Methode nicht besitzt.
```

- Jede Klasse ist ein Typ
- Eine Unterklasse ist ein Untertyp / Subtyp
- Ein Objekt einer Klasse `U` ist kompatibel mit Variable vom Typ `O` gdw `U`=`O` oder `U` ist abgeleitet von `O` (inkl. Transitivität).

Spezialisierung kann verboten werden

```
class Zubehoer extends Artikel {
}
final class DiverseDekoration extends Zubehoer {
   // kann nicht weiter spezialisiert werden.
}
```

Konvertierung

Betrachte die Methode `printInformation(Artikel)`:

```
void printInformation(Artikel a){
   System.out.println("Ich bin eine Pflanze mit der
       Lagertemperatur: " +
       ((Pflanze)a).liefereLagertemperatur());
}
```

Konvertierung

Betrachte die Methode `printInformation(Artikel)`:

```
void printInformation(Artikel a){
   System.out.println("Ich bin eine Pflanze mit der
       Lagertemperatur: " +
       ((Pflanze)a).liefereLagertemperatur());
}
```

```
Artikel a = new Artikel();
Artikel p = new Pflanze();
p.legeLagertemperaturFest(5);
printInformation(p); // OK
printInformation(a); // Laufzeitfehler: ClassCastException
```

Typprüfung

```
void printInformation(Artikel a){
   if(a instanceof Pflanze)
      System.out.println("Ich bin eine Pflanze mit der
          Lagertemperatur: " +
          ((Pflanze)a).liefereLagertemperatur());
   else System.out.println("Ich bin keine Pflanze"');
}
```

Typprüfung

```
void printInformation(Artikel a){
   if(a instanceof Pflanze)
      System.out.println("Ich bin eine Pflanze mit der
          Lagertemperatur: " +
          ((Pflanze)a).liefereLagertemperatur());
   else System.out.println("Ich bin keine Pflanze"');
}
```

```
Artikel a = new Artikel();
Artikel p = new Pflanze();
p.legeLagertemperaturFest(5);
printInformation(p); // OK
printInformation(a); // OK
```

Statischer Typ vs. dynamischer Typ

- Statischer Typ: Typ, mit dem die Variable deklariert wird; unveränderlich
- Dynamischer Typ: Typ des Objekts, das die Variable referenziert; kann durch Zuweisungen geändert werden

Es gilt stets: Dynamischer Typ ist Subtyp von oder gleich statischem Typ.

Statischer Typ vs. dynamischer Typ

- Statischer Typ: Typ, mit dem die Variable deklariert wird; unveränderlich
- Dynamischer Typ: Typ des Objekts, das die Variable referenziert; kann durch Zuweisungen geändert werden

Es gilt stets: Dynamischer Typ ist Subtyp von oder gleich statischem Typ.
Im Beispiel: `Artikel p = new Pflanze();`
Statischer Typ: `Artikel`, dynamischer Typ: `Pflanze`

Substitutionsprinzip für Parameter und Ergebnisse von Methoden

- Dynamischer Typ eines Parameters muss stets Subtyp oder gleich dem statischen Typ sein
- Dynamischer Typ eines Ergebnisses kann der statische Typ oder ein Subtyp sein
- Also: Sowohl Parameter als auch Ergebnisse können spezieller sein als der statische Typ impliziert, aber nicht allgemeiner.

Überschreiben von Methoden

Angenommen wir wollen Artikel mit Zeichenketten repräsentieren:

```
public class Artikel{
   double preis;
   String name;
   public String toString(){
      return "Artikel: " + name + ", Preis: " + preis;
   }
}
```

Überschreiben von Methoden

Was passiert bei Aufruf von `p.toString()` wenn `p` eine `Pflanze` ist?

```
public class Pflanze extends Artikel{
   double lagertemperatur
}
```

Überschreiben von Methoden

Was passiert bei Aufruf von `p.toString()` wenn `p` eine `Pflanze` ist?

```
public class Pflanze extends Artikel{
   double lagertemperatur
}
```

Lagertemperatur wird „unterschlagen“: Ausgabe von:

```
Artikel p = new Pflanze();
p.legePreisFest(5);
p.legeNameFest("Horst");
p.legeLagerTemperaturFest(3);
System.out.println(p.toString());
```

ist: `Artikel: Horst, Preis: 5.`

Überschreiben von Methoden

Lösung: Methode überschreiben.

```
public class Pflanze extends Artikel{
   double lagertemperatur
   public String toString(){
      return super.toString() + ", Lagertemperatur: " +
          lagertemperatur;
   }
}
```

Überschreiben von Methoden

Lösung: Methode überschreiben.

```
public class Pflanze extends Artikel{
   double lagertemperatur
   public String toString(){
      return super.toString() + ", Lagertemperatur: " +
          lagertemperatur;
   }
}
```

Ausgabe von:

```
Artikel p = new Pflanze();
p.legePreisFest(5);
p.legeNameFest("Horst");
p.legeLagerTemperaturFest(3);
System.out.println(p.toString());
```

ist: `Artikel: Horst, Preis: 5, Lagertemperatur: 3.`

Überschreiben von Methoden

Auswirkung des Überschreibens

- Implementation in der Unterklasse ersetzt Implementation in Oberklasse
- Die Wahl der Implementation anhand des dynamischen Typs:
  `Artikel p = new Pflanze();`
  `System.out.println(p.toString());` verwendet Implementation in `Pflanze`.
- Methode aus der Oberklasse kann mit Schlüsselwort `super` aufgerufen werden.

Fehlerquellen beim Überschreiben von Methoden

- Man versucht eine Methode zu überschrieben die als `final` gekennzeichnet ist

Fehlerquellen beim Überschreiben von Methoden

- Man versucht eine Methode zu überschrieben die als `final` gekennzeichnet ist
- Man versucht eine Methode zu überschrieben die als `static` gekennzeichnet ist

Fehlerquellen beim Überschreiben von Methoden

- Man versucht eine Methode zu überschrieben die als `final` gekennzeichnet ist
- Man versucht eine Methode zu überschrieben die als `static` gekennzeichnet ist
- Man verändert die Semantik einer Methode beim Überschreiben

Fehlerquellen beim Überschreiben von Methoden

- Man versucht eine Methode zu überschrieben die als `final` gekennzeichnet ist
- Man versucht eine Methode zu überschrieben die als `static` gekennzeichnet ist
- Man verändert die Semantik einer Methode beim Überschreiben
- Programmteil weiß nichts von abgeleiteten Klassen und erwartet Verhalten der Oberklasse

Fehlerquellen beim Überschreiben von Methoden

- Man versucht eine Methode zu überschrieben die als `final` gekennzeichnet ist
- Man versucht eine Methode zu überschrieben die als `static` gekennzeichnet ist
- Man verändert die Semantik einer Methode beim Überschreiben
- Programmteil weiß nichts von abgeleiteten Klassen und erwartet Verhalten der Oberklasse
- Man überschreibt versehentlich eine Methode

Fehlerquellen beim Überschreiben von Methoden

- Man versucht eine Methode zu überschrieben die als `final` gekennzeichnet ist
- Man versucht eine Methode zu überschrieben die als `static` gekennzeichnet ist
- Man verändert die Semantik einer Methode beim Überschreiben
- Programmteil weiß nichts von abgeleiteten Klassen und erwartet Verhalten der Oberklasse
- Man überschreibt versehentlich eine Methode
- Die Signatur (d.h. Name oder Parametertypen) weicht von der Methode aus der Oberklasse ab – dann wird nicht überschrieben, sondern überladen

Fehlerquellen beim Überschreiben von Methoden

- Man versucht eine Methode zu überschrieben die als `final` gekennzeichnet ist
- Man versucht eine Methode zu überschrieben die als `static` gekennzeichnet ist
- Man verändert die Semantik einer Methode beim Überschreiben
- Programmteil weiß nichts von abgeleiteten Klassen und erwartet Verhalten der Oberklasse
- Man überschreibt versehentlich eine Methode
- Die Signatur (d.h. Name oder Parametertypen) weicht von der Methode aus der Oberklasse ab – dann wird nicht überschrieben, sondern überladen
- Die Signatur stimmt mit einer Methode aus der Oberklasse überein, aber der Rückgabetyp nicht – das ist generell verboten

Verdecken von Attributen

Attributnamen können in Unterklassen neu belegt werden und verdecken das Attribut der Oberklasse:

```
class Ober {
   int x;
}
class Unter extends Ober {
   double x;
   void test() {
      int a = super.x; // Zugriff auf int x aus Ober
      double b = x; // Zugriff auf double x aus Unter
   }
}
```

Die Wurzel aller Klassen: `Object`

- Implizite Oberklasse jeder Klasse ohne `extends`-Anweisung

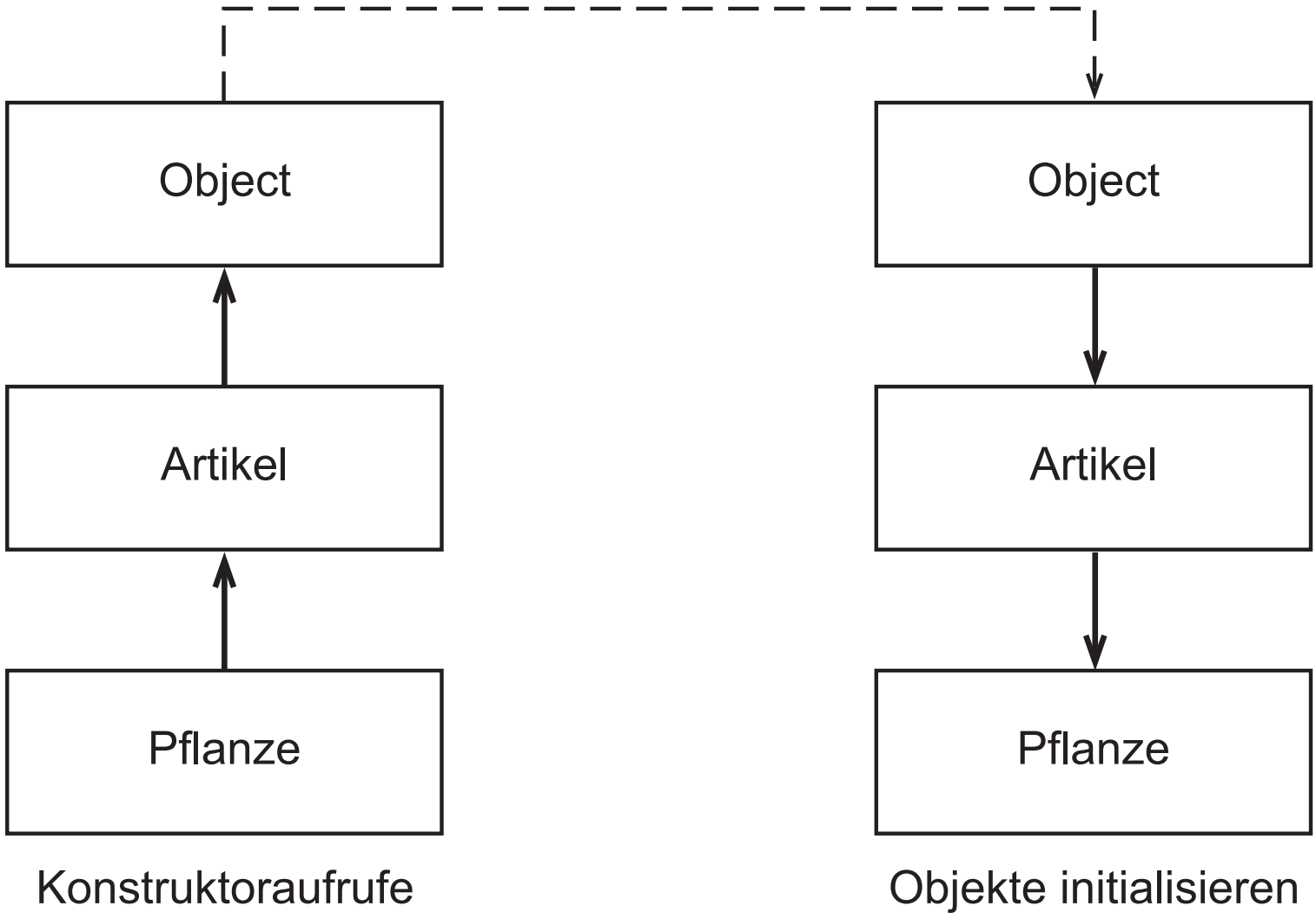
Die Wurzel aller Klassen: `Object`

- Implizite Oberklasse jeder Klasse ohne `extends`-Anweisung
- Enthält Methoden, die alle Klassen anbieten; Überladung empfohlen:
  - `equals()` zum Vergleich mit anderen Objekten gleicher Klasse. Standardimplementation: `a.equals(b)` liefert `a == b` zurück.
  - `toString()` zur textuellen Ausgabe. Standardimplementation: Klassenname und Hashcode des Objekts

Die Wurzel aller Klassen: `Object`

- Implizite Oberklasse jeder Klasse ohne `extends`-Anweisung
- Enthält Methoden, die alle Klassen anbieten; Überladung empfohlen:
  - `equals()` zum Vergleich mit anderen Objekten gleicher Klasse. Standardimplementation: `a.equals(b)` liefert `a == b` zurück.
  - `toString()` zur textuellen Ausgabe. Standardimplementation: Klassenname und Hashcode des Objekts
- `getClass()` gibt ein `Class`-Objekt zurück, das Informationen zur Klasse des Objekts enthält.

Erstellung neuer Objekte



(Nicht-)Standardkonstruktoren

- Standardkonstruktor: Konstruktor ohne Argumente
- Kein Konstruktor angegeben: Implizit Standardkonstruktor gegeben
- Ansonsten muss Standardkonstruktor explizit angegeben werden wenn gewünscht
- Problem: Wie erfolgt Aufstieg bei Konstruktoraufruf?

Ableitung von Klasse ohne Standardkonstruktor

- Erster Befehl in Konstruktor muss Aufruf eines anderen Konstruktors mit `this` oder `super` sein
- Aufrufkette notwendig um Speicherplatz für Klasse korrekt anzulegen

Statische Bindung vs. dynamische Bindung

- Statische Bindung: Welche Methode bei einem Aufruf ausgeführt wird, wird bei Kompilation entschieden.
- Dynamische Bindung: Welche Methode bei einem Aufruf ausgeführt wird, wird zur Laufzeit entschieden.
- Wozu ist dynamische Bindung notwendig?

Statische Bindung vs. dynamische Bindung

- Statische Bindung: Welche Methode bei einem Aufruf ausgeführt wird, wird bei Kompilation entschieden.
- Dynamische Bindung: Welche Methode bei einem Aufruf ausgeführt wird, wird zur Laufzeit entschieden.
- Wozu ist dynamische Bindung notwendig?
- Überladung; dynamischer Typ einer Variable zu Compile-Zeit nicht bekannt

Statische Bindung vs. dynamische Bindung

Regelfall ist dynamische Bindung; erfolgt bei Methodenaufruf

```
Artikel[] artikel = {new Artikel(), new Pflanze(), new
    Artikel()};
for (Artikel a : artikel)
   System.out.println(a.toString());
```

Aufruf von `toString()` zeigt verschiedenes Verhalten für verschiedene Objekte in `artikel`.

Statische Bindung vs. dynamische Bindung

Regelfall ist dynamische Bindung; erfolgt bei Methodenaufruf

```
Artikel[] artikel = {new Artikel(), new Pflanze(), new
    Artikel()};
for (Artikel a : artikel)
   System.out.println(a.toString());
```

Aufruf von `toString()` zeigt verschiedenes Verhalten für verschiedene Objekte in `artikel`.
Wie funktioniert dynamische Bindung?

Vorbereitung zur Compilezeit

- Suche ausgehend vom statischen Typ des Objekts bis zu `Object`
- Es wird die erste Methode gesucht, deren Signatur passend ist
- Wird keine gefunden: Laufzeitfehler
- Anderenfalls setze Suche weiter fort und vergleiche neue Kandidaten mit bisher gefundener Methode:
  - Neuer Kandidat ist spezieller als bisher gefundene Methode: Merk dir den neuen Kandidaten
  - Neuer Kandidat ist genereller oder gleich wie bisher gefundene Methode: Merk dir weiterhin bisher gefundene Methode
  - Neuer Kandidat ist unvergleichbar mit bisher gefundener Methode: Übersetzungsfehler

Speziellere Methode

- Typ `A` ist spezieller als `B` falls `A` Subtyp von `B` ist
- Bei mehreren Parametern: Die Methode, die mehr speziellere Parameter hat ist die speziellere Methode

Dynamische Bindung – zur Laufzeit

- Suche ausgehend vom dynamischen Typ der Variable aufsteigend nach einer Methode, die die zur Compilezeit gefundene überschreibt
- Stoppe, sobald eine Methode gefunden wird, die überschreibt; verwende diese Methode
- Stoppe, wenn statischer Typ gefunden wird; verwende die zuvor gefundene Methode

Statische Bindung

- Geschieht falls die Methode eine Klassenmethode ist.
- Die zur Übersetzungszeit gefundene Methode wird ausgeführt und keine Suche zur Laufzeit vollzogen
- Attribute werden immer statisch gebunden (Verschattung statt Überschreibung)

Beispiel Statische Bindung

```
public class A {
   public int x = 8;
   public int getX() { return x; }
}
public class B extends A {
   public int x = 10;
   public int getX() { return x; }
   public void testBinding() {
      A a = new A();
      B b = new B();
      A ab = new B();
      System.out.print(b.x); // 10
      System.out.print(ab.x); // 8
      System.out.print(b.getX()); // 10
      System.out.print(ab.getX()); // 10
   }
}
```

Fehlerquellen (Übersetzungszeit)

- Keine passende Methode gefunden
- Rückgabetyp der Methode passt nicht zur Verwendung beim Aufruf
- Objektmethode wird an Klasse aufgerufen
- Mehrere Methoden passen gleich gut (Entscheidung uneindeutig)

Fragen zur Vorlesungseinheit

1. Was ist der Unterschied zwischen dynamischer und statischer Bindung?
2. Wozu dient Überschreiben von Methoden?
3. Wie verhindert man, dass eine Methode überschrieben wird?