Informatik B

Vorlesung 2
Konstruktoren, Vererbung

Rückblick

- OOP Grundlagen
- Aufbau einer Klasse
- Aufbau einer Methode
- **`this`**
- *Call-by-value*
- *Call-by-value* mit Seiteneffekt
- Überladen von Methoden

Objekterzeugung

`new`

- Mit dem Operator **`new`** wird ein neues Objekt erzeugt
- Speicherplatz wird für alle benötigten Attribute reserviert
- Die Attributwerte werden mit Defaultwerten (Nullwerten) versehen:
  - **`int`** **`0`**
  - **`double`** **`0.0`**
  - **`boolean`** **`false`**
  - Referenztypen **`null`**
- Eine Referenz auf das neue Objekt wird erzeugt und zurückgeliefert

Objekterzeugung

Konstruktoren

- „Spezielle Methode“
- Werden automatisch bei jedem **new** aufgerufen
- Bei der Objekterzeugung kann der Aufruf eines Konstruktors nicht umgangen werden
- Der Konstruktor versetzt ein mit **new** erzeugtes Objekt in einen sinnvollen Startzustand
- Der Konstruktor konstruiert kein Objekt, sondern „erzeugt“ sinnvolle Daten für das Objekt

Objekterzeugung

Konstruktoren

- Ein Konstruktor hat den gleichen Namen wie eine Klasse
- Syntax:
  [Modifikator] Klassenname([Parameterliste]) {
      // Anweisungen
  }
- Ein Konstruktor ohne Parameter heißt ***Default Constructor***
- Konstruktoren mit Parametern werden ***Custom Constructor*** genannt
- In **jeder** Klasse ist mindestens ein Konstruktor vorhanden, selbst wenn keiner explizit implementiert wurde

Beispiel

```
public class Person {
   String name;
   int alter;
}
```

- Klasse ohne expliziten Konstruktor
- Der Javacompiler fügt einen *Default Constructor* hinzu
- Dieser neue Konstruktor setzt keine Werte, der Compiler wüßte ja auch nicht welche sinnvoll wären
- Der neue Konstruktor ist nur notwendig, um die Syntax bei der Objekterzeugung aufrecht zu erhalten
- Der Compiler fügt **nur** dann einen *Default Constructor* hinzu, wenn **kein** anderer Konstruktor implementiert wurde

Beispiel

- Der vom Compiler hinzugefügte Konstruktor sähe wie folgt aus:

```
public class Person {
    String name;
    int alter;

    public Person() {
    }
}
```

Beispiel

```
public class Person {
      String name;
      int alter;

      public Person(String name, int alter) {
            this.name = name;
            this.alter = alter;
      }
}
```

- In dieser Klasse gibt es keinen *Default Constructor* mehr, da ja ein anderer definiert wurde

Beispiel

```
public class Person {
	String name;
	int alter;

	public Person() {
		name = „NA“;
		alter = 0;
	}

	public Person(String name, int alter) {
		this.name = name;
		this.alter = alter;
	}
}
```

- Hier existiert ein *Default Constructor*, da er explizit implementiert wurde

Copy Constructor

- Erzeugt eine Kopie eines bereits bestehenden Objektes der gleichen Klasse
- Die Referenz des Originalobjektes wird als Parameter übergeben

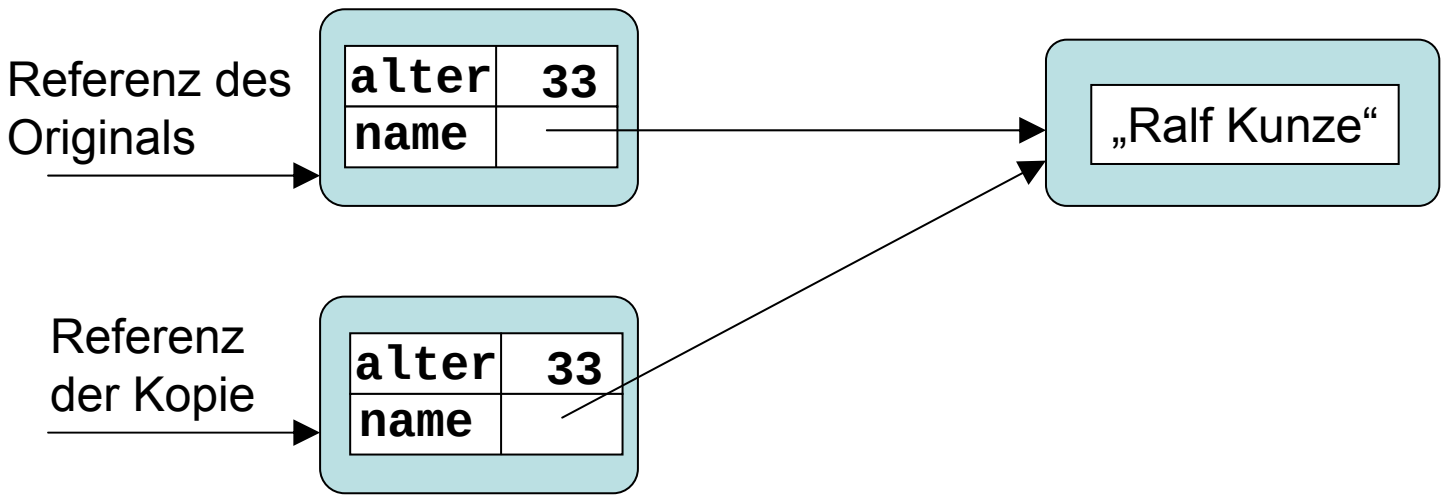
```
public class Person {
    String name;
    int alter;

    public Person(Person p) {
        name = p.name;
        alter = p.alter;
    }
}
```

- Achtung: Beim Attribut **`name`** werden lediglich die Referenzen kopiert!

Copy Constructor

- Problematik bei Attributen mit Referenztypen



Copy Constructor

shallow copy:

- Werden die Attributwerte nur durch Wertzuweisungen kopiert entsteht eine ***shallow copy*** (Flache Kopie)
- Es wird zwar das Objekt kopiert, aber nicht die darin enthaltenen Objekte
- Ob eine *shallow copy* ausreicht hängt von der konkreten Anwendung ab
- In der Regel ist der Ansatz der *shallow copy* zu simpel

Copy Constructor

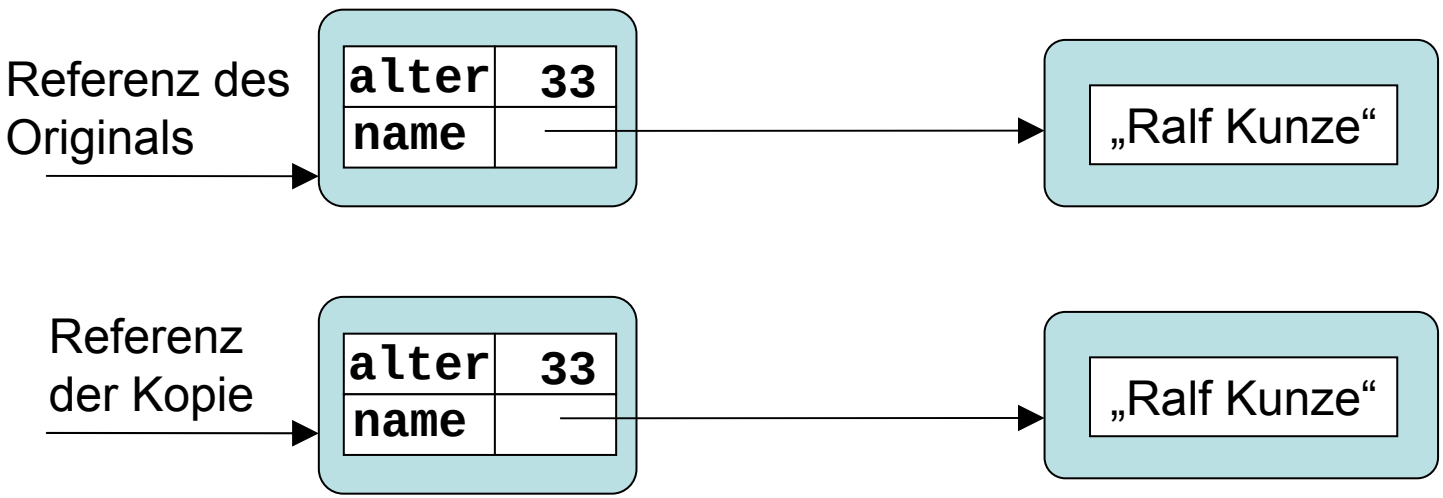
deep copy:

- Bei einer ***deep copy*** (Tiefe Kopie) wird jedes Attribut einzeln kopiert
- Eine Wertzuweisung reicht bei Referenztypen nicht aus
- Die referenzierten Objekte müssen mit geeigneten Methoden kopiert werden
- In der Regel eignet sich dazu der *Copy Constructor* des jeweiligen Objektes
- Das Duplizieren von weitreichenden Objektverkettungen signifikant Rechenzeit und Speicherplatz kosten

Copy Constructor

```
public class Person {
     String name;
     int alter;

     public Person(Person p) {              // deep copy
            alter = p.alter;
            name = new String(p.name);
     }
}
```



Copy Constructor

deep copy

- Enthält eine Klasse nur primitive Datentypen reicht eine Wertzuweisung aus
- Sind nur Referenzen auf nicht änderbare Objekte vorhanden reicht ebenfalls eine Wertzuweisung aus (z.B. ist es entgegen dem vorherigen Beispiel bei **`String`**-Objekten nicht notwendig da diese unveränderbar sind)
- Sind Referenzen auf veränderliche Objekte enthalten müssen diese Objekte kopiert werden
- Im Zusammenhang mit Vererbung erweisen sich *Copy Constructoren* als zu schwach (Abhilfe: **`clone()`**)

Constructor Chaining

- Konstruktoren weisen den Attributen Werte zu
- Konstruktoren übernehmen zusätzlich Tests der übergebenen Parameter
- Bei mehreren Konstruktoren müssten die Tests in jedem einzeln implementiert werden → Fehleranfälligkeit
- Abhilfe: Ein Konstruktor kann einen anderen Konstruktor der gleichen Klasse mittels **`this`** aufrufen

Constructor Chaining

```
public class Person {
     String name;
     int alter;

     public Person(String name, int alter) {
       // Ueberpruefung, ob Variablen korrekte Werte enthalten
            this.name = name;
            this.alter = alter;
     }

     public Person(int alter) {
            this(„NA“,alter);
     }

     public Person() {
            this(„NA“,0);
     }
}
```

Constructor Chaining

- Ein Konstruktor (in der Regel der mit den meisten Parametern) erledigt die meiste Arbeit und wird von den anderen aufgerufen
- Diese Vorgehensweise macht nur Sinn, wenn bei der Initialisierung ähnliche Vorbereitungen/Prüfungen notwendig sind
- Eine *Constructor Chain* mit **`this`** muss als erste Anweisung im Konstruktor stehen
- Nach **`this`** können weitere Anweisungen im Konstruktor enthalten sein

Bedeutungen von `this`

- **`this`** hat zwei Bedeutungen:
  - **`this`** in Kombination mit der Punktnotation bezeichnet das gerade erzeugte Objekt (im Konstruktor), bzw. das aktuelle Objekt an das eine Nachricht geschickt wurde (in einer Methode)
  - **`this`** in der ersten Zeile eines Konstruktors mit einer Parameterliste ruft den entsprechenden Konstruktor derselben Klasse auf

Destruktor

- Nichtreferenzierte Objekte werden vom *Garbage Collector* aus dem Speicher entfernt
- Der *Garbage Collector* wird automatisch vom System aufgerufen oder kann mittels **`System.gc();`** gestartet werden
- Vor dem Löschen eines Objektes wird ein Destruktor aufgerufen:

```
protected void finalize() {
   // Anweisungen
}
```

Destruktor

- Der Aufruf des Destruktors wird nicht garantiert
- Wird das Programm beendet, bevor der *Garbage Collector* aufgerufen wird, wird `finalize()` nicht aufgerufen
- Da Java über eine automatische Speicherverwaltung verfügt, sind Destruktoren weniger Bedeutsam als in anderen Programmiersprachen
- Da der Zeitpunkt des Aufrufes nicht klar definiert ist, sollten Destruktoren mit der nötigen Vorsicht eingesetzt werden
- Sinnvoll, z.B. um Netzverbindungen oder Dateihandles zu schließen

Vererbung

- Eltern geben ihren Kindern Eigenschaften mit
- Vererbung bindet die Klassen noch dichter aneinander
- Klassen werden in gewisser Weise austauschbar
- In Java kann nur von Klassen geerbt werden
- Einschränkungen/Erweiterungen von primitiven Typen (etwa im Wertebereich oder in der Anzahl der Nachkommastellen) sind nicht möglich

Vererbung

Einfach- vs. Mehrfachvererbung

- Einfachvererbung:
  - Eine Klasse ist maximal von einer anderen abgeleitet
- Mehrfachvererbung:
  - Eine Klasse kann beliebig viele Oberklassen haben
- In Java gibt es nur Einfachvererbung
  - Weniger problematisch
  - Sauberer objektorientierter Ansatz
- In C++ gibt es Mehrfachvererbung

Vererbung

- Klassen in Java sind in einer Hierarchie geordnet
- Von **`Object`** erben automatisch alle Klassen, direkt oder indirekt
- Eine Klasse erweitert durch das Schlüsselwort **`extends`** eine andere Klasse
- Sie wird zur Unterklasse (auch Subklasse oder Kindklasse genannt)
- Die Klasse, von der die Unterklasse erbt, heißt Oberklasse (auch Superklasse oder Elternklasse)
- Es werden **alle sichtbaren** Eigenschaften der Oberklasse auf die Unterklasse übertragen
- Eine Oberklasse vererbt Eigenschaften; die Unterklasse erbt sie
- Syntax: **`class`** Unter **`extends`** Ober { }

Vererbung

- Sichtbare Attribute werden an die Unterklasse vererbt. Beispiel: **`vererbung1`**
  - Attribute können überdeckt werden
  - Namenskonflikte können mittels **`super`** gelöst werden
    Beispiel: **`vererbung2`**
- Sichtbare Methoden werden an die Unterklasse vererbt. Beispiel **`vererbung3`**
  - Methoden können überdeckt/überschrieben werden
  - Namenskonflikte können mittels **`super`** gelöst werden

Vererbung

Methoden überschreiben:

- Es existiert in der Unterklasse eine Methode mit der gleichen Signatur wie in der Oberklasse
- Rückgabetyp nicht zur Signatur, doch muss eine Unterklasse diesen Typ übernehmen (gilt immer bei primitiven Typen, bei Referenztypen können es in Unterklassen auch Untertypen sein)
- Implementiert die Unterklasse die Methode neu, so sagt sie auf diese Weise: »Ich kann’s besser.«
- Die überschreibende Methode kann demnach den Funktionscode spezialisieren und Eigenschaften nutzen, die in der Oberklasse nicht bekannt sind.

Vererbung

`super`

- In der überschreibenden Methode kann die Methode der Oberklasse aufgerufen werden: **`super.methode();`** Beispiel: **`vererbung4`**
- Sinnvoll, wenn nicht die ganze Funktionalität neu implementiert werden soll.
- Generell kann innerhalb einer Methode mittels **`super`** auf Methoden der Oberklasse zugegriffen werden
- „Umgehen“ der dynamischen Bindung

Vererbung

- Konstruktoren werden (trotz ihrer Ähnlichkeit zu Methoden) nicht vererbt
- Die Initialisierung der Attribute übernimmt jede Klasse selbst
- In allen Vererbungshierarchien müssen die Attribute sinnvoll belegt werden
- In einem Konstruktor wird **immer** als erstes der Konstruktor der Oberklasse aufgerufen

Vererbung

super (Konstruktor)

- Der Aufruf eines Konstruktors der Oberklasse erfolgt mittels **`super([`**Parameterliste**`])`**
- **`super`** muss in der ersten Zeile eines Konstruktors stehen
- Es kann wahlweise der *Default* als auch der *Custom Constructor* verwendet werden (je nach Verfügbarkeit) Beispiel: **`vererbung4`**
- Wird super nicht explizit hingeschrieben, fügt der Compiler das Statement **`super();`** (*Default Constructor* der Oberklasse) in den Konstruktor ein, falls er nicht existiert => Fehlermeldung

Bedeutungen von **`super`**

- **`super`** hat zwei Bedeutungen
  - **`super`** in Kombination mit der Punktnotation bezeichnet das aktuelle Objekt (interpretiert als Objekt der Oberklasse), an das eine Methode geschickt wurde
  - **`super`** in der ersten Zeile eines Konstruktors mit einer Parameterliste ruft den entsprechenden Konstruktor der Oberklasse auf

Substitutionsprinzip

- Wenn eine Unterklasse **u** die Oberklasse **o** erweitert (von **o** abstammt), kann überall, wo **o** gefordert wird, auch ein **u** übergeben
- Es wurde etwas allgemeines gefordert, die Unterklasse ist nur spezieller
- Wird mehr als gefordert übergeben, kann damit zwar nichts angefangen werden, aber es weist alle geforderten Eigenschaften auf
- Weil an Stelle eines Objekts auch ein Objekt der Unterklasse auftauchen kann, spricht man von Substitution
- Wird z.B. ein Objekt der Klasse Person gefordert, kann auch ein Student die Rolle erfüllen

Vererbung

Dynamisches Binden

- Der Compiler kann nicht entscheiden, welche öffentliche Instanzmethode ausgewählt werden soll
- Durch das Substitutionsprinzip könnte eine Referenz vom Typ **`Person`** auf ein Objekt der Klasse **`Student`** verweisen
- Es soll immer die Methode aufgerufen werden, die dem aktuellen Objekt beim Verfolgen der Vererbungslinie am nächsten ist
- Es wird daher zur Laufzeit entschieden, welche Methode aufgerufen wird

Vererbung

Dynamisches Binden

- Wie findet der Java-Interpreter die richtige Methode:
  - Zunächst wird in der Klasse des beauftragen Objektes geschaut und falls gefunden wird die Methode ausgeführt
  - Wird die Methode nicht gefunden wird in der nächst „höheren“ Klasse geschaut, usw.
  - Spätestens in der obersten Klasse (**`Object`**) wird die Methode gefunden und ausgeführt
- Beispiel: **`dynamischesbinden1`**
- Attribute und nicht öffentliche Methoden werden statisch gebunden, Beispiel: **`dynamischesbinden2`**

Gleichheit von Objekten

- Zwei Objekte sind gleich, wenn sie physikalisch identisch sind (gleiche Speicheradresse)
- Der Operator == testet zwei Referenzen, ob sie dasselbe Objekt referenzieren
- Zwei Objekte können als gleich angesehen werden, wenn die enthaltenen Daten gleich sind
- Wenn getestet werden soll, ob zwei physikalisch verschiedene Objekte gleich sind, ist diese Gleichheit eine Frage der Fachlichkeit
- Operational wird die Gleichheit dann mit der Methode **`equals(Object o)`** geprüft, die in der Klasse **`java.lang.Object`** definiert ist
- **`equals()`** wird vererbt und kann entsprechend überschrieben werden

Gleichheit von Objekten

- Wird **`equals`** überschrieben müssen folgende Regeln gelten:
  - **Symmetrie**: Für zwei Referenzen **`x`** und **`y`** ungleich **`null`**: **`x.equals(y)`** genau dann wenn **`y.equals(x)`**
  - **Reflexivität**: Für die Referenz **`x`** ungleich **`null`** gilt: **`x.equals(x)`** ist **`true`**
  - **Transitivität**: Für drei Referenzen **`x`**, **`y`** und **`z`** ungleich **`null`** gilt: Wenn **`x.equals(y)`** und **`y.equals(z)`** dann auch **`x.equals(z)`**
  - **Konsistenz**: Für zwei Referenzen **`x`** und **`y`** ungleich **`null`** liefert der wiederholte Aufruf von **`x.equals(y)`** entweder immer **`true`** oder immer **`false`**, falls keine Informationen, die zur Berechnung von **`equals()`** verwendet werden, sich ändern
  - Für jede Referenz **`x`** ungleich **`null`** gilt: **`x.equals(null)`** liefert **`false`**.

Gleichheit von Objekten

`hashCode()`

- Wird die Methode **`equals()`** überschrieben, muss ggf. auch die Methode **`hashCode()`** überschrieben werden.
- Es muss gelten:
  - Die Methode **`hashCode()`** liefert für zwei gleiche Objekte (nach Definition von **`equals()`**) denselben Wert zurück
  - In der Regel muss also **`hashCode()`** überschrieben werden, wenn **`equals()`** überschrieben wird
- Beispiel: **`vergleich1`**

Zusammenfassung

- Konstruktor
- Destruktor
- Vererbung:
  - Attribute
  - Methoden
  - Konstruktoren
  - **`super/this`**
- Gleichheit von Objekten

Ausblick

- Typecasting
- Abstrakte Klassen
- Interfaces
- Modifikatoren
- Packages
- …