Informatik B

Vorlesung 3

Casts, Abstrakte Klassen, Interfaces, Packages, Sichtbarkeiten

Rückblick

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

Cast

- Möglicherweise kommt es vor, dass Datentypen konvertiert werden müssen
- Dies nennt sich Typanpassung (engl. typecast) oder auch casten
- Ein Cast wird generell zur Compilezeit ausgewertet und hat zur Laufzeit keine Bedeutung mehr
- Casts von Referenzen sind vor allem durch den Vererbungsmechanismus notwendig
- Casts von primitiven Datentypen schränken den Wertebereich ein

Cast

Referenzen

- Der Compiler prüft, ob ein Methodenaufruf gültig ist, also ob die Methode in der durch den Referenztyp angegebenen Klasse enthalten ist
- Wird einer Referenz ein Objekt einer Unterklasse zugeordnet, können also nur Methoden des Referenztyps aufgerufen werden und nicht des Typs des eigentlich enthaltenen Objektes
- Ist der Programmierer sicher, das in einer Referenz ein Objekt einer Unterklasse des Referenztyps enthalten ist, kann er es durch einen Cast dem Compiler deutlich machen
- Beispiel: **`cast1`**

Cast

Referenzen

- Ein Cast verhindert nicht das Dynamische Binden bei öffentlichen Methoden
- Weiterhin wird entsprechend des Objektes zur Laufzeit die Methode ausgewählt
- Beispiel: **`cast2`**

Cast

Primitive Datentypen

- Ein Cast bei primitiven Datentypen entspricht einer Typanpassung
- Java unterscheidet zwei Arten der Typanpassung:
  - Automatische (implizite) Typanpassung. Daten eines kleineren Datentyps werden automatisch (implizit) dem größeren angepasst. Der Compiler nimmt diese Anpassung selbstständig vor.
  - Explizite Typanpassung. Ein größerer Typ kann einem kleineren Typ mit möglichem Verlust von Informationen zugewiesen werden.

Cast

Primitive Datentypen

- Implizit (*widening conversion*)
  - Werte der Datentypen byte und short werden bei Rechenoperationen automatisch in den Datentyp int umgewandelt
  - Ist ein Operand vom Datentyp long, dann werden alle Operanden auf long erweitert
  - Wird aber short oder byte als Ergebnis verlangt, dann ist dieses durch einen expliziten Typecast anzugeben

Cast

Implizit (*widening conversion*)

- Folgende Typumwandlungen führt Java automatisch aus:

Von Typ	In Typ
`byte`	**`short, char, int, long, float, double`**
`short`	**`int, long, float, double`**
`char`	**`int, long, float, double`**
`int`	**`long, float, double`**
`long`	**`float, double`**
`float`	**`double`**

Cast

Primitive Datentypen

- Explizit (*narrowing conversion*)
  - Die explizite Anpassung engt einen Typ ein
  - Bei der expliziten Typumwandlung von **`double`** und **`float`** in einen Ganzzahltyp kann es zum Verlust von Genauigkeit kommen
  - Eine Einschränkung des Wertebereichs ist möglich
  - Bei der Konvertierung von Fließkommazahlen verwendet Java eine Rundung gegen null
  - Bei der Konvertierung eines größeren Ganzzahltyps in einen kleineren werden einfach die oberen Bits abgeschnitten
  - Eine Anpassung des Vorzeichens findet nicht statt
  - Die Darstellung in Bit zeigt das sehr anschaulich:

```
int m = 123456789;      // 00000111010110111100110100010101
short sm = (short) m;                    // 1100110100010101
System.out.println(sm); // -13035
int n = -123456;        // 11111111111111100001110111000000
short sn = (short) n;                    // 0001110111000000
System.out.println(sn); // 7616
```

Cast

Primitive Datentypen

- Explizit
  - Ein **`short`** hat wie ein **`char`** eine Länge von 16 Bit
  - Doch diese Umwandlung ist nicht ohne ausdrückliche Konvertierung möglich
  - Das liegt am Vorzeichen von **`short`**. Zeichen sind per Definition immer ohne Vorzeichen
  - Würde ein **`char`** mit einem gesetzten höchstwertigen letzten Bit in ein **`short`** konvertiert, käme eine negative Zahl heraus. Ebenso wäre, wenn ein **`short`** eine negative Zahl bezeichnet, das oberste Bit im **`char`** gesetzt, was unerwünscht ist. Die ausdrückliche Umwandlung erzeugt immer nur positive Zahlen

Cast

Primitive Datentypen

- Explizit
  - Leider ist die Typanpassung nicht ganz so einleuchtend, Beispiel: **`cast3`**
  - Wenn Ganzzahl-Ausdrücke vom Typ kleiner **`int`** mit einem Operator verbunden werden, passt der Compiler eigenmächtig den Typ auf **`int`** an
  - Bei der Zuweisung steht auf der rechten Seite ein **`int`** und auf der linken Seite der kleinere Typ **`byte`** oder **`short`**. Mit der ausdrücklichen Typumwandlung erzwingen wir diese Konvertierung und akzeptieren ein paar fehlende Bit

Cast

- Alle nicht öffentlichen Methoden und alle Attribute werden statisch gebunden
- Ein Cast bei Instanzvariablen kann ein verändertes statisches Bindungsverhalten zur Folge haben
- Zur Compilezeit werden die statisch gebundenen Elemente fest verdrahtet
- Beispiel **`cast4`**

Abstrakte Klasse / Interface

- Eine Klasse soll nicht immer komplett ausprogrammiert werden
- Dies ist der Fall, wenn die Oberklasse lediglich Methodensignaturen für die Unterklassen vorgeben möchte, aber nicht weiß, wie sie diese implementieren soll
- In Java gibt es dazu zwei Konzepte:
  - Abstrakte Klassen
  - Schnittstellen (engl. interfaces).

Abstrakte Klassen

- Bisher konnte jede Klasse Objekte bilden
- Das ist nicht immer sinnvoll, z.B. wenn eine Klasse nur in einer Vererbungshierarchie existieren soll
- Sie kann dann als Modellierungsklasse eine Ist-eine-Art-von-Beziehung ausdrücken und Signaturen für die Unterklassen vorgeben
- Eine Oberklasse besitzt dabei Vorgaben für die Unterklasse, das heißt, alle Unterklassen erben die (öffentlichen) Methoden und Attribute
- Um das in Java auszudrücken, wird der Modifizierer **`abstract`** an die Typdeklaration der Oberklasse gesetzt

Abstrakte Klassen

- Von einer abstrakten Klasse können keine Objekte erzeugt werden auch wenn ein Konstruktor definiert werden darf
- Ansonsten verhalten sich die abstrakten Klassen wie normale, sie enthalten die gleichen Eigenschaften und können auch selbst von anderen Klassen erben
- Abstrakte Klassen sind das Gegenteil von konkreten Klassen

Abstrakte Klassen

Abstrakte Methoden

- Das Schlüsselwort **`abstract class`** leitet die Deklaration einer abstrakten Klasse ein
- Eine Klasse kann ebenso abstrakt sein wie eine Methode
- Eine abstrakte Methode gibt lediglich die Signatur und den Rückgabewert vor, und eine Unterklasse implementiert irgendwann diese Methode
- Durch abstrakte Methoden wird ausgedrückt, dass die Oberklasse keine Ahnung von der Implementierung hat und dass sich die Unterklassen darum kümmern müssen

Abstrakte Klassen

Abstrakte Methode

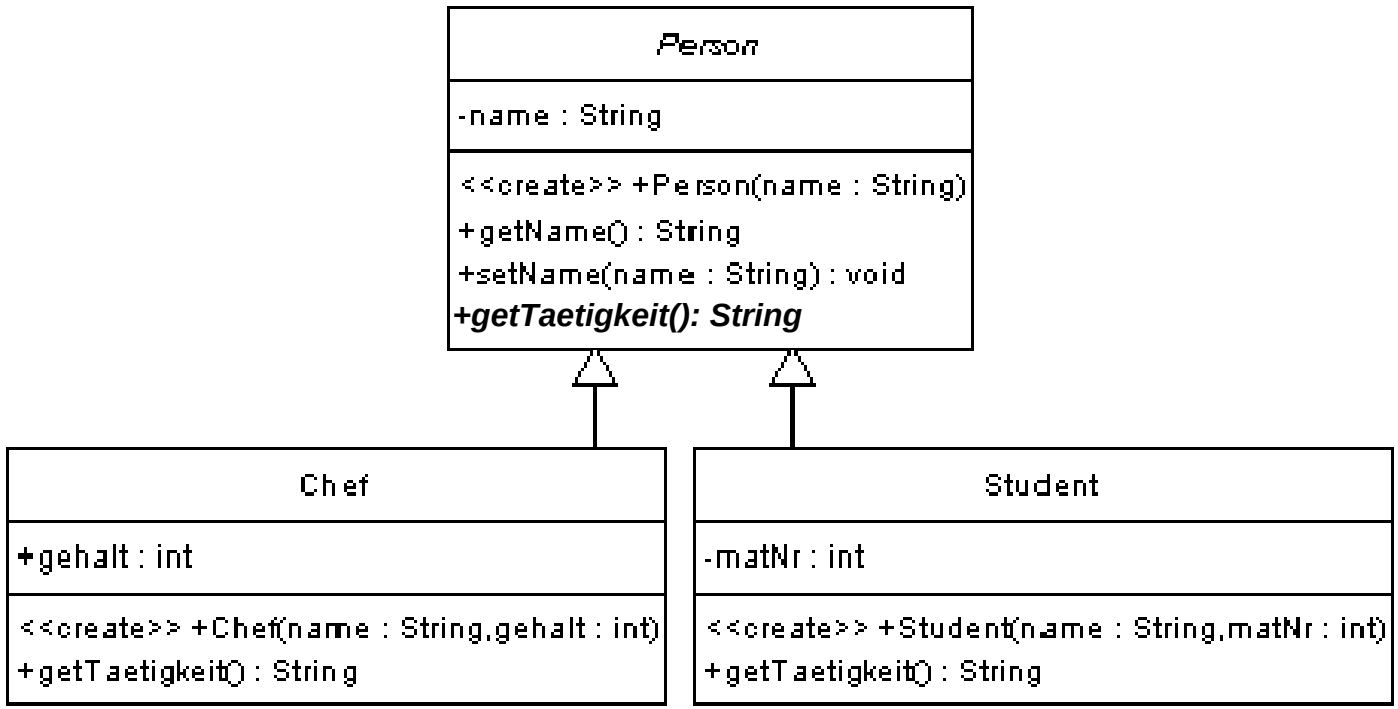
- Ist mindestens eine Methode abstrakt, so ist es automatisch die ganze Klasse
- Das Schlüsselwort **`abstract`** muss dann ausdrücklich vor den Klassennamen geschrieben werden
- Vergisst man das Schlüsselwort **`abstract`** bei einer solchen Klasse, folgt daraus ein Compilerfehler
- Eine Klasse mit einer abstrakten Methode muss abstrakt sein, da sonst ein Exemplar der Klasse konstruiert und genau diese Methode aufgerufen werden könnte
- Beispiel: **`abstrakt1`** (Klasse Person)

Abstrakte Klassen

Vererben von abstrakten Methoden

- Wenn von einer Klasse abstrakte Methoden geerbt werden, hat man zwei Möglichkeiten:
  - Überschreiben aller abstrakten Methoden. Dann muss die Unterklasse nicht mehr abstrakt sein (wobei sie es auch weiterhin sein kann). Von der Unterklasse können Objekte instanziiert werden
  - Die abstrakte Methode wird nicht überschrieben, so dass sie normal vererbt wird. Das bedeutet: Eine abstrakte Methode bleibt in der Klasse, und die Klasse muss wiederum abstrakt sein.
- Beispiel: **`abstrakt1`**

Abstrakte Klassen (UML)



Da die kursive Schrift teilweise schwer zu erkennen ist kann man auch in geschweiften Klammern das Wort **`abstract`** hinzugefügt werden:

`Person`
`{abstract}` **`+getTaetigkeit: String {abstract}`**

Interfaces

- Ein Interface (Schnittstelle) enthält keine Implementierungen
- Es werden nur Methodeköpfe deklariert
  - Modifizierer
  - Rückgabetyp
  - Signatur
- Ein Methodenrumpf wird nicht angegeben
- Der Name einer Schnittstelle endet oft mit *–ble* (**`Accessible`**, **`Adjustable`**, **`Cloneable`**, **`Runnable`**)

Interfaces

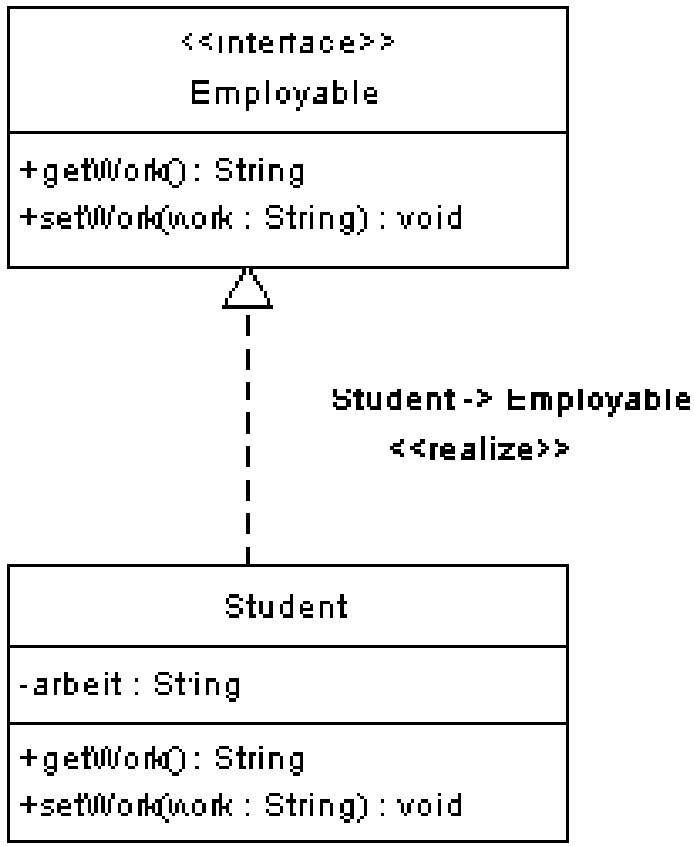
- Die Deklaration eines Interfaces ist ähnlich zu einer abstrakten Klasse
- An Stelle von **`class`** steht das Schlüsselwort **`interface`**
- Bei den abstrakten Methoden muss der Modifizierer **`abstract`** nicht zwingend geschrieben werden, da alle Methoden in Schnittstellen automatisch abstrakt und öffentlich sind
- Die von den Schnittstellen deklarierten Methoden sind – wie auch bei abstrakten Methoden – mit einem Semikolon abgeschlossen
- Methoden in Interfaces haben niemals eine Implementierung
- Ein Interface darf keinen Konstruktor deklarieren
- Instanzvariablen dürfen nicht deklariert werden
- Lediglich Konstanten dürfen in einem Interface definiert werden:
  ```
  final int PI = 3.14159265
  ```
- Beispiel: **`interface1`** (Interface **`Employable`**)

Interfaces

Implementieren von Interfaces

- Verwendet eine Klasse ein Interface, so folgt hinter dem Klassennamen das Schlüsselwort **`implements`** gefolgt vom Namen der Schnittstelle
- Die Implementation mehrerer Interfaces ist erlaubt
- Mehrere Interface Namen werden durch Kommata voneinander getrennt
- Bei der Implementierung eines Interfaces müssen die Methoden in den Unterklassen öffentlich implementiert werden, da die Methoden in Schnittstellen immer automatisch **`public`** sind
- Implementiert eine Klasse nicht alle Operationen aus den Schnittstellen, so erbt sie damit abstrakte Funktionen und muss selbst wieder als abstrakt gekennzeichnet werden
- Beispiel: **`interface1`**

Interfaces (UML)



Interfaces

Marker Interface

- Schnittstellen ohne Methoden sind möglich
- Leere Schnittstellen werden *Marker Interface* (Markierungsschnittstellen) oder *Tag Interface* genannt
- Sie sind nützlich, da mit **`instanceof`** leicht überprüft werden kann, ob eine Klasse eine gewisse Markierung aufweist
- Beispielsweise kann an jedem Objekt die Methode **`clone()`** aufgerufen werden, aber nur wenn der Programmierer die Klasse mit dem **`Cloneable`** Interface markiert hat, sonst erfolgt eine Fehlermeldung

Interfaces

Vorteile

- Über Interfaces lässt sich eine Sicht auf ein Objekt beschreiben
- Jedes Interface ermöglicht eine neue Sicht auf das Objekt, eine Art Rolle
- Implementiert eine Klasse diverse Interfaces, können ihre Exemplare in verschiedenen Rollen auftreten
- Hier wird erneut das Substitutionsprinzip wichtig, bei dem ein mächtigeres Objekt verwendet wird, auch wenn weniger erwartet wird
- Interfaces sind ein gutes Mittel, um Teamarbeit zu unterstützen

Interfaces

Mehrfachvererbung

- Eine Klasse darf nur von einer direkten Oberklasse abgeleitet werden
- Ohne Schwierigkeiten kann eine Klasse mehrere Schnittstellen implementieren
- Dies wird gelegentlich als »Mehrfachvererbung in Java« bezeichnet
- Auf diese Weise besitzt die Klasse ganz unterschiedliche Typen, da sie nun **`instanceof`** der Oberklasse sowie der Schnittstellen ist

Interfaces

Mehrfachvererbung

- Problem bei der Mehrfachvererbung von Klassen ist, dass zwei Oberklassen die gleiche Funktion mit zwei unterschiedlichen Implementierungen vererben könnten
- Die Unterklasse weiß dann nicht, welche Logik sie erbt
- Bei Interfaces gibt es das Problem nicht
- Wenn zwei (oder mehr) Interfaces die gleiche Methode (inklusive Rückgabetyp) vorschreiben würden, gibt es keine zwei verschiedenen Implementierungen
- Die implementierende Klasse bekommt lediglich zweimal die Aufforderung, die Operation zu implementieren

Interfaces

Interfacemethoden, die nicht implementiert werden müssen

- In einem Interface deklarierte Methoden müssen nicht implementiert werden, wenn diese durch die Vererbungshierarchie geerbt wurden
- Wird in einem Interface z.B. die Methode **`equals()`** vorgeschrieben ist die Implementierung nicht nötig
- **`equals()`** ist eine Methode, die jedes Objekt besitzt
- Eventuell überschreibt man **`equals()`**, falls die Implementierung anderes leisten soll
- Der Sinn derartige Methoden dennoch im Interface zu deklarieren besteht darin, die Funktionsweise in der Dokumentation genau anzugeben
- Eine Java-Dokumentation kann nur generiert werden, wenn auch eine Methode im Quellcode vorhanden ist

Abstrakte Klasse vs. Interface

- Eine abstrakte Klasse und ein Interface sind sich sehr ähnlich:
  - Beide schreiben den Unterklassen beziehungsweise den implementierten Klassen Operationen vor, die sie implementieren müssen
  - Ein wichtiger Unterschied ist jedoch der, dass beliebig viele Schnittstellen implementiert werden können, jedoch nur eine Klasse
  - Abstrakte Klassen bieten sich meist in der Design-Phase an, wenn Gemeinsamkeiten in einer Oberklasse ausgelagert werden sollen
  - Abstrakte Klassen können zusätzlichen Programmcode enthalten, was Schnittstellen nicht können
  - Nachträgliche Änderungen an Interfaces sind nicht einfach
  - Einer abstrakten Klasse kann eine konkrete Methode mitgegeben werden, was zu keiner Quellcodeanpassung für Unterklassen führt

Packages

- Ein Java-Paket ist eine logische Gruppierung von Klassen
- Klassen, die zu einem Paket gehören, befinden sich im gleichen Verzeichnis
- Der Name des Pakets ist dann gleich dem Namen des Verzeichnisses (und natürlich umgekehrt)
- Sun hat für sich einige Paketnamen reserviert, die von eigenen Klassen nicht genutzt werden sollen (**`java`**, **`javax`**, **`sun`**)
- Wenn man eigene Klassen in Pakete mit dem Präfix **`java`** setzen würde, schafft man damit Verwirrung, da nicht mehr nachvollziehbar ist, ob das Paket bei jeder Distribution dabei ist

Packages

Hierarchische Strukturen

- Pakete lassen sich in Hierarchien ordnen
- In einem Paket kann wieder ein anderes Paket liegen
- Genauso wie bei der Verzeichnisstruktur des Dateisystems
- Sun definiert das Paket **`java`** für einen Hauptzweig, aber auch **`javax`**. Unter dem Paket **`java`** liegen dann zum Beispiel **`awt`**, **`util`** und andere

Packages

- Der Name einer Klasse ist der Klassenname mit kompletter package Angabe: **`java.lang.String`**
- Man spricht vom vollqualifizierten Klassennamen
- Innerhalb einer Klasse wird die Paketzugehörigkeit durch das Schlüsselwort **`package`** in der ersten Zeile der Klasse deutlich gemacht:

```
package de.uos.inf.rkunze.meinpackage;
```

Packages

- Um Klassen außerhalb des eigenen Pakets nutzen zu können, müssen sie dem Compiler präzise beschrieben werden
  - Volle Qualifizierung des Klassennamens Beispiel: **`personenverwaltung1`**
  - Mittels **`import`** eine Klasse bekannt machen Beispiel: **`personenverwaltung2`**
- Damit nicht alle Klassen eines Pakets einzeln aufgeführt werden müssen, können Sie mit einem Wildcard auf alle sichtbaren Klassen zugreifen: **`import java.util.*;`**
- Achtung! Die Verwendung von Wildcards beim Import von Klassen ist nicht empfehlenswert. Werden nachträglich einem mittels Wildcard importiertem Paket Klassen hinzugefügt kann es zu Namenskonflikten kommen!!! Besser immer nur die wirklich benötigten Klassen importieren.

Packages

Statisches Import

- Das **`import`** hat in Java die Bedeutung, den Compiler über die Pakete zu informieren, so dass eine Klasse nicht mehr voll qualifiziert werden muss, wenn sie im Importteil explizit aufgeführt wird oder wenn das Paket der Klasse genannt ist.
- Falls eine Klasse statische Methoden oder Konstanten vorschreibt, werden ihre Eigenschaften immer über den Klassennamen angesprochen (**`Math.PI`**)
- Es gibt mit dem statischen Import die Möglichkeit, die Klasseneigenschaften wie eigene Funktionen oder Variablen ohne Klassennamen sofort zu nutzen

Packages

Static Import:

```
import static java.lang.Math.max;
import static java.lang.System.out;

class ImportTest {

   public static void main( String[] args ) {
       out.println( max(1,10) );
   }
}
```

Packages

Defaultpackage

- Falls eine Klasse ohne Paket-Angabe implementiert wird, befindet sie sich standardmäßig im unbenannten Paket (engl. *unnamed package*) oder Default-Paket
- Es ist immer sinnvoll, eigene Klassen in Paketen zu organisieren
- Durch die Verwendung von Packages sind feinere Sichtbarkeitseinstellungen möglich

Sichtbarkeit

- Methoden und Variablen können für unterschiedliche Sichtbarkeitsbereiche definiert werden
- Generell sollte man einen soweit wie möglich eingeschränkten Sichtbarkeitsbereich verwenden
- Es sollte möglichst kaum „jemand“ auf Variablen direkt zugreifen können, es sei denn es ist zwingend erforderlich

	Eigene Klasse	Klasse gleiches Paket	Unterklasse im anderen Paket	Klasse in anderen Paketen
`public`	Ja	Ja	Ja	Ja
`protected`	Ja	Ja	Ja	Nein
„paketsichtbar“	Ja	Ja	Nein	Nein
`private`	Ja	Nein	Nein	Nein

Zusammenfassung

- Cast
- Abstrakte Klassen
- Interfaces
- Packages
- Sichtbarkeit

Nächste Woche

- Exceptions
- Dateioperationen