Programmieren in Java

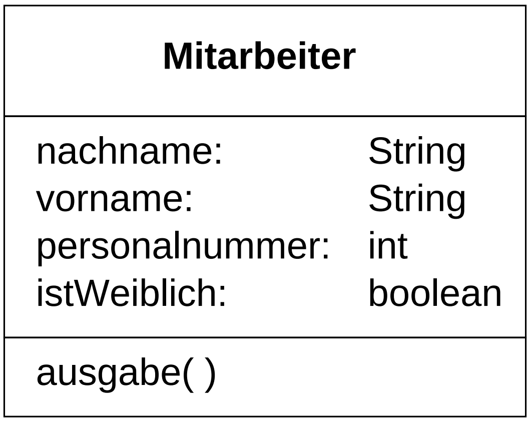
Ein Ergänzungskurs zur
Einführung in die Informatik

Teil 3: Objektorientierung

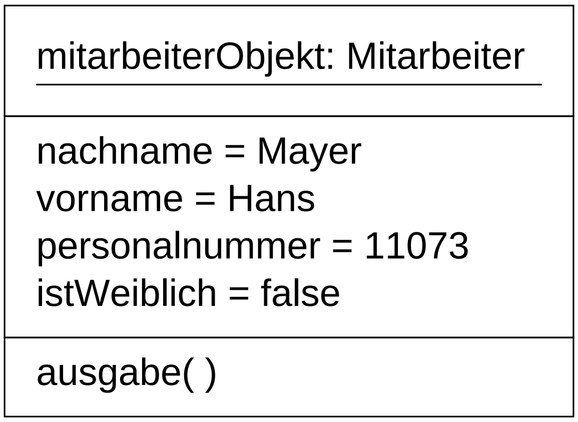
- Klassen und Objekte
- Generalisierung und Spezialisierung
- Vererbung
- Überschreibung von Methoden
- Polymorphismus
- Abstrakte Klassen
- Schnittstellen
- Pakete
- Zugriffsrechte

Klassen und Objekte

- (Objekt-)Klasse: Beschreibung der Eigenschaften einer Gruppe (Klasse) von Objekten („Bauanleitung“)
- Eigenschaften einer Klasse von Objekten bestimmt durch
  - Attribute (Variablen)
    Beschreibung der Objektstruktur
  - Operationen (Methoden)
    Beschreibung des Objektverhaltens
- Objekt: Instanz einer Klasse
  Instanziierung
  → Initialisierung von Attributen (Variablenwerte)
  → Festlegung des anfänglichen Objektzustands
- Graphische Darstellung mit Diagrammen:



Klassenbeschreibung



Objektbeschreibung

Wichtige Ziele der OO-Modellierung

- Problemorientierte Strukturierung des Programms
- getrennte Entwicklung einzelner Systemkomponenten
- Wiederverwendbarkeit von Komponenten ($\rightarrow$ Reduzierung des Implementierungsaufwands)

Wichtige Techniken der OO-Programmierung

- Kapselung (engl.: encapsulation) $\rightarrow$ Kontrollierter Zugriff auf Daten
- Vererbung (engl.: inheritance) $\rightarrow$ Wiederverwendbarkeit von Programmteilen
- Polymorphismus (engl.: polymorphism) $\rightarrow$ Flexible Nutzung abstrakter und konkreter Objekteigenschaften

Klassenhierarchien

- Darstellung von Klassen entsprechend
  - dem Grad ihrer Spezialisierung
  - ihrer Verwandtschaft zu anderen Klassen
- Entstehung durch
  - Generalisierung:
    Zusammenfassung gemeinsamer Eigenschaften
    zweier (oder mehrerer) Klassen
    $\rightarrow$ (direkte) Oberklasse
  - Spezialisierung:
    Hinzufügen zusätzlicher Eigenschaften zu
    bereits vorhandenen Eigenschaften einer Klasse
    $\rightarrow$ (direkte) Unterklasse
- Oberklasse: allgemeiner als ihre Unterklassen
  (verkörpert nur Gemeinsamkeiten der Unterklassen)
- Unterklasse: Teilklasse all ihrer Oberklassen
  (verfügt über alle Eigenschaften der Oberklassen)
- in Java:
  Klasse `Object` ist allgemeinste Klasse
  $\rightarrow$ `Object` Oberklasse für alle Klassen

Kapselung

- strenge Auslegung des Kapselungsprinzips:
  Zugriff auf Attribute *nur* über Methoden möglich
- in Java:
  Grad der Kapselung durch Festlegen von Zugriffsrechten individuell bestimmbar

Vererbung

- Aufgabe:
  Weitergabe der Oberklassen-Eigenschaften an deren Unterklassen
- Syntax:
  `class` *Unterklasse* `extends` *Oberklasse* { ... }
- Beispiel:
  `class Mitarbeiter extends Person { ... }`
- Variablen / Methoden der Oberklasse *ohne* explizite Deklaration auch in der Unterklasse verwendbar
- in Java:
  - höchstens eine direkte Oberklasse
    (keine Mehrfachvererbung)
  - Keine explizite Angabe einer Oberklasse $\rightarrow$
    Klasse `Object` wird als Oberklasse festgelegt

Überschreiben von Methoden

- Deklaration einer Methode in einer Unterklasse:
  Methodenkopf entspricht dem Kopf einer in einer Oberklasse deklarierten Methode
- Überschreibung betrifft nur Methodenrumpf
- Aufgabe:
  einheitlicher Methodenaufruf für Ober-/Unterklasse trotz verschiedener Methodenrümpfe
  (Ausnutzung klassenspezifischer Eigenschaften)
- Beispiel:
  Die Methode `toString()` der Klasse `Object`
- Konstruktoren nicht überschreibbar
  (auch nicht vererbbar, da klassenspezifisch)
- Zugriff auf Methoden/Konstruktoren der Oberklasse (mit Schlüsselwort `super`):
  - Methoden: `super.`*Methodenname*`(...);`
  - Konstruktoren: `super(...);`

Polymorphismus

- Ermöglicht den Aufruf konkreter Methoden auf abstrakter Ebene
- für ein Objekt der Klasse *Unterklasse* gilt:
  - Objekt gehört auch zu allen Oberklassen von *Unterklasse* (Objekt hat alle Eigenschaften der Oberklassen)
  - Für Variablen/formale Parameter mit Oberklasse als Typ: auch Objekte der *Unterklasse* zuweisbar (nur Oberklassen-Eigenschaften werden genutzt)
- Polymorphismus: Aufruf einer überschriebenen Methode $\rightarrow$ der klassenspezifischste verfügbare Methodenrumpf wird ausgeführt

abstrakte Klassen

- konkrete Methodendeklaration (mit Methodenrumpf) nicht immer sinnvoll:
  - Abstraktionsgrad der Klasse zu hoch
  - Klasse nur Ausgangsbasis zur Erstellung konkreter Unterklassen

  → abstrakte Methodendeklaration (ohne Rumpf)
- Syntax für abstrakte Methode:
  `abstract` *Ergebnistyp Methodenname* `(`*Parameter*`);`
- Klasse enthält abstrakte Methode
  → Klasse muß abstrakt deklariert werden
- Syntax für abstrakte Klasse:
  `abstract` *Klassenname* { ... }
- keine Instanziierung abstrakter Klassen möglich!
- Unterklassen abstrakter Klassen:
  Konkretisierung durch Implementierung aller geerbten abstrakten Methoden
  (ansonsten auch Unterklasse abstrakt)

Schnittstellen (interfaces)

- Aufgabe: Beschreibung abstrakter Eigenschaften
- zulässige Komponenten einer Schnittstelle:
  - abstrakte Methoden (`abstract` ...)
  - Konstanten (`static final` ...)
- Syntax für Schnittstellendeklarationen:
  `interface` *Schnittstellenname* {
  abstrakte Methodendeklarationen
  Konstantendeklarationen
  }
- Modifikatoren wie `public` und die Schlüsselwörter `abstract` bzw. `static` innerhalb einer Schnittstellendeklaration *nicht* explizit angeben!
- Schnittstellen als spezielle Form abstrakter Klassen:
  - nicht instanziierbar
  - als Datentyp verwendbar (für beliebige Objekte, deren Klassen die Schnittstelle implementieren)
  - Schnittstellenhierarchien möglich (Vererbung mit `extends`)
  - *nicht* als (abstrakte) Oberklasse in Klassenhierarchien einsetzbar

Implementierung von Schnittstellen

- Implementierung:
  Konkretisierung aller in der Schnittstelle enthaltenen abstrakten Methoden
- Syntax:
  `class` *Klassenname* `implements` *Schnittstellenname*
  { ... }
- Implementierung mehrerer Schnittstellen:
  - Angabe der Schnittstellen als Liste, jeweils durch Kommas getrennt
  - entspricht Konkretisierung verschiedener abstrakter Eigenschaften für eine Klasse
  - ersetzt Mehrfachvererbung in Java

Pakete (packages)

- Zusammenfassung von Klassen zu funktional abgeschlossenen Einheiten

- Syntax:
  `package` *Paketname*`;`

- Paket umfaßt Klassendateien eines Verzeichnisses (Unterverzeichnisse als Unterpakete)

- falls keine Angabe eines Paketnamens:
  Implizite Einteilung in "unsichtbares Paket"
  (mit den Klassendateien desselben Verzeichnisses)

- vollständig qualifizierte Bezeichner:
  Paketname.Klassenname

- Abkürzung der Bezeichner durch Import:

  - Syntax:
    `import` Paketname`.*;`
    `import` Paketname`.`*Klassenname*`;`

  - Angabe von *Paketname* kann entfallen

- Pakethierarchie der Java-Standardklassen:
  Unterpakete des Pakets `java` (z.B. `java.lang`)

Zugriffsrechte

- Modifikatoren für “Sichtbarkeit”:
  - `public`:
    Zugriff auf Klassen/Komponenten auch außerhalb des jeweiligen Pakets uneingeschränkt möglich
  - `protected`:
    Zugriff auf Klassen/Komponenten innerhalb des jeweiligen Pakets uneingeschränkt möglich Für Unterklassen außerhalb des Pakets Zugriff eingeschränkt möglich.
  - Standard (ohne Modifikator):
    Zugriff auf Klassen/Komponenten innerhalb des jeweiligen Pakets uneingeschränkt möglich
  - `private`:
    Zugriff auf Komponenten nur innerhalb der jeweiligen Klasse möglich
    (keine Vererbung!)
- Modifikator `final` unterbindet schreibende Zugriffe:
  - Klassen: keine Unterklassenbildung
  - Methoden: keine Überschreibung
  - Variablen: keine Wertzuweisung nach der Initialisierung ($\rightarrow$ Konstanten)