II.4.2 Abstrakte Klassen und Interfaces

Dienstag, 28. November 2017 09:00

Oft ähnliche Datenstrukturen, die sich nur im Typ der darin gespeicherten Werte unterscheiden. Man sollte hierfür nicht immer neue Klassen implementieren, sondern eine generische Datenstruktur implementieren, die dann für Werte versch. Typen verwendet werden kann.

Mögl. Lösung:
Datentyp Liste, der Werte v. Typ Object speichern kann.

$l = (\frac{5}{4}, \frac{1}{2})$

2 Nachteile

1. Listen können jetzt Werte bel. Typen enthalten

$l = (\text{"hallo"}, \frac{5}{4}, \frac{1}{2})$

2. Listen machen nur Sinn bei Werten, die man auf Gleichheit überprüfen kann

Objekte bel. Typen kann man

i. A. nur mit == auf
Gleichheit überprüfen.
Diese Gleichheit ist oft
nicht sinnvoll.

l.suche(neuBruch(1,2))
wäre false.

- Man sollte Liste und Element so definieren, dass darin nur Werte von solchen Klassentypen gespeichert werden, bei denen es eine Methode "gleich" gibt, die eine sinnvolle inhaltliche Gleichheit implementiert.
  ⇒ Anforderung an Klassen
- Lösung: definiere Oberklasse "Vergleichbar" mit Methode "gleich".
  Dann dürfen Werte nur noch vom Typ "Vergleichbar" (oder einer der U-Klassen v. "Vergleichbar" sein).
- "Vergleichbar" soll nur das Vorhandensein der Methode "gleich" in allen ihren U-Klassen erzwingen. Es ist sinnlos, "gleich" schon in "Vergleichbar" zu implementieren. Alle U-Klassen sollen "gleich" überschreiben.
  ⇒ Abstrakte Klasse

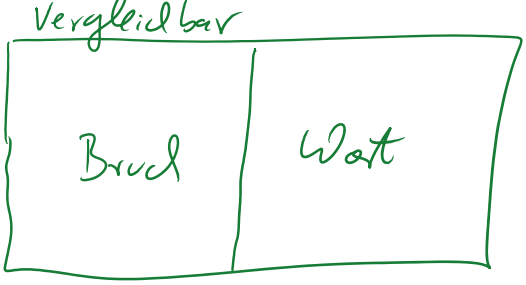
Abstrakte Klasse

Klasse mit Methode enthält bei denen der Methode

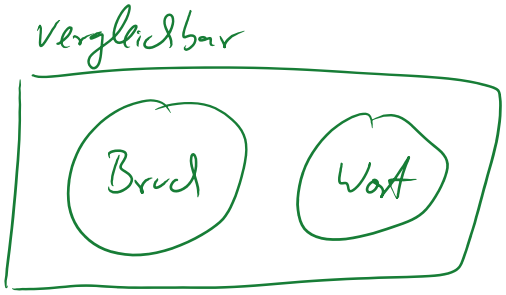
- Klasse, die Methoden enthält, bei denen der Methodenrumpf fehlt (abstrakte Methoden).
  ⇒ ; statt Methodenrumpf
- Diese Methode existiert dann auch in allen U-Klassen der abstrakten Klasse.
- In jeder konkreten U-Klasse muss die abstr. Methode durch eine konkrete Methode überschrieben worden sein.
- Abstr. Klasse kann Konstruktoren enthalten, aber man kann keine Objekte erzeugen, die "nur" vom Typ der abstr. Klasse sind.

  `Vergleichbar v = new Vergleichbar();` ← verboten
- Im Bsp: "gleich" in Klasse Bruch vergleicht Brüche inkl. "Kürzen"
- Wenn Bruch und Wort die 2 U-Klassen von Vergleichbar sind:

Vergleichbar abstrakte Klasse



Vergleichbar nicht abstrakt



- Man kann abstrakte Methoden aufrufen (z.B. wert.gleich)

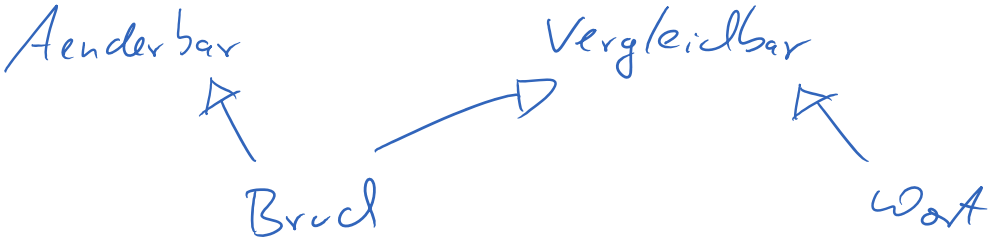
weil weil zur Laufzeit ein Objekt einer konkreten U-Klasse ist, die die abstrakte Methode mit einer konkreten Methode überschrieben hat.

- Generisches Programmieren: Trenne die Implementierung v. "höheren Datenstrukturen" von der Implementierung der darin gespeicherten Werte.

Nachteil v. abstrakten Klassen:

- Wir haben weiterhin nur Einfachvererbung, d.h. jede Klasse hat nur eine direkte Oberklasse.
  Grund: Vermeidg. v. Konflikten beim Erben von konkreten Methoden.
- Abstrakte Klassen dienen dazu, Anforderungen an U-Klassen zu formulieren. Man würde gerne ausdrücken, dass eine U-Klasse mehrere Anforderungen erfüllt.

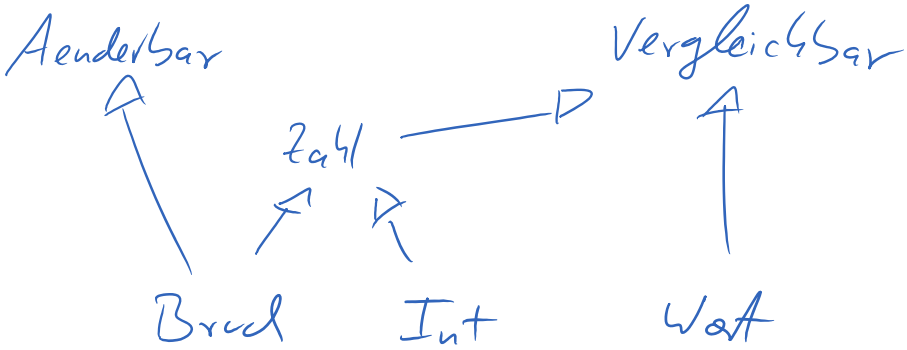
Bsp: Abstr. Klasse "Aenderbar" erzwingt, dass es eine Methode "aenderung" gibt, mit der man die Werte der Attribute ändern kann.



Lösung in Java: Interfaces

- entspreden abstrakten Klassen, in denen es nur noch abstrakte Methoden gist.
- ähnlich zur Schnittstellendokumentation (mit javadoc): beschreibt, welche Methoden v. außen sichtbar sind (mit Namen, Eingabe- u. Resultat-Typ)
- Im Interface sind alle Methoden public und abstract. (Kein Schlüsselwort mehr nötig)
- Abstrakte Methoden sind niemals statisch.
- Wenn Klasse U-Klasse eines Interfaces ist: "implements" statt "extends". In konkreter Klasse müssen alle Methoden d. Interfaces überschrieben worden sein, sonst ist die Klasse abstrakt.
- Bei Interfaces ist Mehrfachvererbung erlaubt.
- Attribute eines Interfaces sind automatisch static und final (Konstante d. Interfaces).

Bsp:



- Interfaces u. die abstrakten Methoden darin können genauso verwendet werden wie andere Klassen.

Datenzugriff mit Interfaces

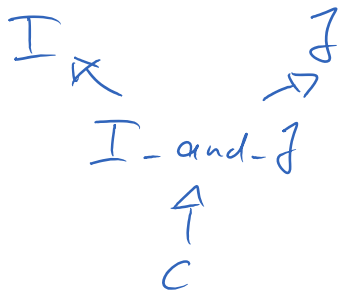
Datenzugriff mit Interfaces

- `I i = z;` ← implizite Typanpassung v. U-Klasse zur O-Klasse / Interface
- `i.b(5);` ← führt die b(int)-Methode aus Klasse C aus.
- `j.b(5);` ← führt die b(double)-Methode aus Klasse C aus

- `i.q(5);` bzw. `j.q(5);` ← führt die q(int)-Methode aus C aus

- `C.y` ← 6, wird v. Interface I geerbt

Interfaces können sich gegenseitig erweitern (d.h. ein Interface kann "Unter-Interfaces" haben).



Seit Java 8: Interfaces dürfen doch auch (manche) konkreten Methoden enthalten

- Idee: Wenn man Interfaces um neue Methoden erweitert, dann müssten alle Klassen geändert werden, die dieses Interface implementieren, da sie die neuen

die dieses Interface implementieren, da sie die neuen Methoden jetzt auch überschreiben müssen.

Schlüsselwort default

Abhilfe: Interface darf default-Implementierung für Methoden enthalten, die benutzt wird, falls Klasse die Methode nicht überschreibt.

- Nachteile:
  - Nicht-Eindeutigkeit bei Mehrfachvererbung ← kann zu Prog-Fehler führen
  - Kaum noch Unterschied zw. abstr. Klassen u. Interfaces
    - * Interfaces haben aber keine Konstruktoren
    - * — " — haben nur static final Attribute

- super bei Interfaces: Konkrete Angabe des Ober-Interfaces, um Mehrfachvererbungs-Uneindeutigkeit aufzulösen: `I.super.m()`

- Statische Methoden v. Interfaces dürfen nur über den Namen des Interfaces aufgerufen werden, nicht über Namen d. U-Klasse.

  `I.w()`, aber nicht `C.w()`.