Objektorientierte Programmierung

Kapitel 22: Aufzählungstypen (Enumeration Types)

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

http://www.informatik.uni-halle.de/~brass/oop14/

Inhalt

Einführung und Motivation (1)

- Ein Aufzählungstyp (engl. “enumeration type”) ist ein Datentyp, der nur eine kleine Anzahl möglicher Werte hat, die durch explizite Aufzählung definiert werden.
- Beispiel:

```
(1)  enum Wochentag {
(2)      MONTAG,
(3)      DIENSTAG,
(4)      MITTWOCH,
(5)      DONNERSTAG,
(6)      FREITAG,
(7)      SAMSTAG,
(8)      SONNTAG
(9)  }
```

Es ist Konvention, dass die einzelnen Aufzählungskonstanten (z.B. `MONTAG`) in Großbuchstaben geschrieben werden.

Einführung und Motivation (2)

- Solche Daten gibt es nicht selten:
  - Monate (Januar, . . . , Dezember)
  - Anreden (Herr, Frau, . . .)
  - Spielkarten-Farben (Kreuz, Pik, Herz, Karo)
  - Gegenstands-Typen in einem Rollenspiel
  - Himmelsrichtungen
  - . . .
- Kennzeichen:
  - Die Anzahl verschiedener Werte ist nicht sehr groß.
  - Die Wertemenge ist fest.

    Nicht abhängig von Benutzereingaben.

Einführung und Motivation (3)

- Ohne spezielles Konstrukt würde man kleine ganze Zahlen verwenden (Codes für die verschiedenen Werte), z.B.

```
(1)  class Wochentag {
(2)      static final int MONTAG = 1;
(3)      static final int DIENSTAG = 2;
(4)      ...
```

- Nachteil: Variablen, die einen Wochentag enthalten sollen, müssen mit dem Typ `int` deklariert werden.
- Der Compiler gibt dann keine Fehlermeldung aus, wenn man z.B. -23 in eine solche Variable speichert.
- Es gibt auch keine Fehlermeldung, wenn man z.B. einen Wochentag mit einer Himmelsrichtung vergleicht.

  Alle solchen Aufzählungen haben ja den Typ `int`.

Einführung und Motivation (4)

- Wenn man möchte, dass einem der Compiler bei der frühzeitigen Fehlererkennung hilft, muss man ihm mehr Information geben:
  - hier also die Aufzählungstypen untereinander und von `int` deutlich unterscheiden.
- Je früher ein Fehler erkannt wird, desto weniger Arbeit verursacht er.

  Meist dauert die Fehlersuche beim Testen ("Debugging") deutlich länger als die Beseitigung eines Fehlers, den der Compiler schon gemeldet hat. Wenn der Fehler erst beim Kunden auftritt, ist es noch schlimmer.
- Fehler, die der Compiler meldet, sind nicht zu übersehen. Fehler, die erst zur Laufzeit (manchmal) auftreten, können unentdeckt bleiben.

Aufzählungstypen: Basiswissen (1)

- Java hat dazu seit Version 1.5 das Konstrukt "`enum`":

```
enum Wochentag { MONTAG, ..., SONNTAG }
```

- Dann ist `Wochentag` eine Klasse, von der es genau die sieben Objekte `MONTAG` bis `SONNTAG` gibt.
- Die Syntax in diesem einfachsten Fall ist also:
  - Schlüsselwort "`enum`",
  - Name des Aufzählungstyps,
  - "`{`",
  - Konstanten des Aufzählungstyps, durch "`,`" getrennt,

    Wie bei initialisierten Arrays ist ein "`,`" nach dem letzten Wert erlaubt.

  - "`}`".

Aufzählungstypen: Basiswissen (2)

- Die Aufzählungskonstanten sind statische konstante Felder in der Klasse, im wesentlichen so:

```
(1)  class Wochentag {
(2)      static final Wochentag MONTAG =
(3)              new Wochentag();
(4)      ...
```

- Wenn man sie im Programm verwenden will, muss man also "Klasse.Konstante" schreiben:

```
(1)  enum Wochentag { MONTAG, ..., SONNTAG }
(2)
(3)  class Test {
(4)      public static void main(String[] args) {
(5)          Wochentag w = Wochentag.MONTAG;
```

Aufzählungstypen: Basiswissen (3)

- In einem `switch` schreibt man nur den Konstanten-Namen:

  Normale Objekt-Referenzen könnte man nicht als case-Label verwenden.

```
(10)    boolean wochenende(Wochentag w) {
(11)        switch(w) {
(12)            case MONTAG:
(13)            case DIENSTAG:
(14)            case MITTWOCH:
(15)            case DONNERSTAG:
(16)            case FREITAG:
(17)                return false;
(18)
(19)            case SAMSTAG:
(20)            case SONNTAG:
(21)                return true;
(22)        }
(23)    }
```

Inhalt

Basis-Methoden (1)

- Häufig werden Aufzählungs-Konstanten nur verwendet, um einige wenige Fälle zu unterscheiden, z.B. mit == oder dem `switch`.
- Aufzählungstypen haben aber einige vordefinierte Methoden.

  Und man kann sogar eigene definieren, s.u.
- `String toString()`:
  Liefert den Namen der Konstante, z.B. funktioniert:

```
System.out.println(w); // Druckt MONTAG
```

- `String name()`: Wie `toString()`.

  Die Methode `toString()` könnte im Aufzählungstyp überschrieben werden, die Methode `name()` ist in der impliziten Oberklasse `Enum` `final` und kann somit nicht modifiziert werden. Falls man ganz sicher sein will, dass es der Name der Konstante ist, wäre `name()` besser.

Basis-Methoden (2)

- `int ordinal()`:
  Diese Methode liefert die Position der Konstante in der Deklaration des Aufzählungstyps.

  Im Beispiel würde `Wochentag.MONTAG.ordinal()` den Wert `0` liefern.

  Wenn `w` den Wert `Wochentag.SONNTAG` enthält, würde `w.ordninal() == 6` sein.

- `static T valueOf(String name)`:
  Diese Methode liefert zum Namen einer Konstanten das entsprechende Objekt.

  Z.B.: `Wochentage.valueOf("MONTAG")` liefert `Wochentage.MONTAG`.

  `static T[] values()`:
  Diese Methode liefert ein Array, das alle Werte (Objekte) des Aufzählungstyps enthält.

Basis-Methoden (3)

- `int compareTo(T e)`:
  Der Vergleich zweier Aufzählungskonstanten ist definiert basierend auf der Reihenfolge, in der sie definiert wurden.

  Das Interface `Comparable` wird implementiert.

- Verschiedene Methoden von `Object` wie
  - `boolean equals(Object o)` und
  - `int hashcode()`

  sind passend (und nicht änderbar) definiert.

  Sie sind in der impliziten Oberklasse `Enum` `final`.

Aufzählungstypen als spezielle Klassen

- Konstruktoren von Aufzählungstypen sind immer `private`, man kann also kein weiteres Objekt außer den explizit aufgezählten Konstanten erzeugen.

  Auch die von `Object` ererbte Methode `clone()` erzeugt nur eine `CloneNotSupportedException`.

- Man kann keine Subtypen (Unterklassen) von Aufzählungstypen deklarieren.
- Selbstverständlich kann man Werte eines Aufzählungstyps einer Variablen von Typ `Object` zuweisen.
- Für Aufzählungstypen `T` gibt es spezielle Implementierungen von `Set<T>` und `Map<T,V>`, nämlich `EnumSet<T>` und `EnumMap<T,V>`. Sie sind besonders effizient (z.B. Bitarray).

Inhalt

Zusätzliche Methoden (1)

- Man kann die Aufzählungstyp-Klasse um eigene Methoden (und Konstruktoren) erweitern. Beispiel (engl. Abkürzungen):

```
(1)  enum Wochentag {
(2)      MONTAG("MON"),
(3)      DIENSTAG("TUE"),
(4)      MITTWOCH("WED"),
(5)      DONNERSTAG("THU"),
(6)      FREITAG("FRI"),
(7)      SAMSTAG("SAT"),
(8)      SONNTAG("SUN");
(9)
(10)     private String en;
(11)     Wochentag(String en) { this.en = en; }
(12)     String enName() { return this.en; }
(13) }
```

Zusätzliche Methoden (2)

- Nach der letzten Konstante kann man also ein Semikolon "`;`" schreiben, und danach mehr oder weniger beliebigen Programmcode wie in einer normalen Klasse.
- Wenn man einen oder mehrere Konstruktoren mit Parametern eingeführt hat, kann man nach den Konstanten die entsprechenden Argument-Werte angeben.
- Konstruktoren müssen immer `private` sein.

  Man braucht es nicht hinzuschreiben, sie sind automatisch `private`.
  Man kann aber z.B. nicht `public` schreiben. Der implizite Aufruf des Konstruktors der Oberklasse `Enum`, bei dem Name und Position der Konstante übergeben werden, wird vom Compiler automatisch eingefügt, ein expliziter Aufruf von `super(...)` ist verboten.
  Zugriffe auf nicht-konstante statische Attribute im Konstruktor sind verboten (sie wären noch nicht initialisiert).

Konstanten-spezifischer Programmcode (1)

- Eigene Implementierung einer Methode für jede Konstante:

```
(1)  enum Direction {
(2)      NORTH {
(3)          void go() { Game.goXY(0, -1); }
(4)      },
(5)      EAST {
(6)          void go() { Game.goXY(1, 0); }
(7)      },
(8)      SOUTH {
(9)          void go() { Game.goXY(0, 1); }
(10)     },
(11)     WEST {
(12)         void go() { Game.goXY(-1, 0); }
(13)     };
(14)     abstract void go();
(15) }
```

Konstanten-spezifischer Programmcode (2)

- Nun hat die Klasse `Direction` eine Methode `go()`, aber jedes Objekt der Klasse (jede Aufzählungskonstante) hat eine unterschiedliche Implementierung.
- Formal werden hier anonyme, geschachtelte Subklassen eingeführt.

  In dieser Vorlesung wurden geschachtelte Klassen nicht behandelt.
- Hier gehört also jede Konstante einer eigenen Subklasse von `Direction` an. In der Subklasse ist die Methode überschrieben.

  Dies ist die einzige Möglichkeit, Subklassen von Aufzählungstypen zu bekommen. Normale Subklassen kann man nicht deklarieren. Man beachte, dass es auch nicht nötig ist, die Klasse `Direction` als `abstract` zu deklarieren, was sonst für eine Klasse mit abstrakten Methoden nötig wäre.

Konstanten-spezifischer Programmcode (3)

- Man könnte diese Aufgabe aber auch mit Attributen lösen:

```
(1)  enum Direction {
(2)      NORTH(0, -1),
(3)      EAST(1, 0),
(4)      SOUTH(0, 1),
(5)      WEST(-1, 0);
(6)
(7)      private int diffX, diffY;
(8)      Direction(int dX, int dY) {
(9)          this.diffX = dX;
(10)         this.diffY = dY;
(11)     }
(12)     void go() {
(13)         Game.goXY(this.diffX, this.diffY);
(14)     }
(15) }
```