3 Enumerationen

Enumerationen sind die heimlichen Sieger in J2SE 5, denn nach C/C++ und vielen endlosen Beteuerungen der Java-Gurus von Sun, dass sie in Java überflüssig sind, weil sie sich doch so schön nachbilden lassen, hat mancher (zumindest der Autor) eine eher konservative Lösung wie *glorified Integers* erwartet. Was nun angesichts der Konkurrenz C# mit `enum`-Klassen geliefert wurde, ist eine wirklich gute Spracherweiterung. Der Begriff „glorified integers“ wird direkt am Anfang der Enum-Spezifikation verwendet, um damit die Implementierungen der anderen Sprachen zu charakterisieren. Alle C-ähnlichen Sprachen verwenden Enumerationen als typ-sichere konstante integrale Typen, also ganzen Zahlen mit mnemotechnischen Namen, wie dieses C#-Beispiel zeigt:

```
enum Days {Sun=1, Mon, Tue, Wed, Thu, Fri, Sat};
```

Abgesehen davon, dass sich `Days` exakt so nicht in Java-Enums nachbilden lässt¹, ist man mit Java 1.5 ansonsten konsequent einen Schritt weiter gegangen und betrachtet Enumerationen als normale Klassen mit speziellen konstanten Mitgliedern und Restriktionen. Dies hat Vorteile bei der Verwendung, da sie die Realisierung von Enumerationen mit komplexen Eigenschaften zulässt. Vorab eine Liste der Features:

Enum-Features!

Aufzählung von Konstanten, optional mit komplexem Verhalten.

- Im Gegensatz zu der trivialen Alternative `static final int ANY_ENUM= 123` sind `enum`-Konstanten typ-sicher.
- Im Gegensatz zu normalen Konstanten können `enum`-Konstante geändert, hinzugefügt oder gelöscht werden, ohne dass die Klassen, die sie verwenden, re-compiliert werden müssen (wobei das Löschen von benutzten Konstanten allerdings zu einer Ausnahme führt).
- Bis auf ihre besonderen Eigenschaften können `enum`-Klassen beliebige Felder und Methoden enthalten, beliebige Interfaces implementieren und ihre Konstanten werden als normale Objekte behandelt.
- Von der Klasse `Enum` geerbte bzw. automatisch generierte Methoden wie `toString()`, `hashCode()`, `equals()`, `values()` und `valueOf()` machen die `enum`-Klassen ohne zusätzlichen Code sofort einsatzfähig.

Unterstützt werden die Enumerationen durch performante Spezialversionen `EnumSet` bzw. `EnumMap` von Containern. Was man Sun angesichts dieser Implementation vorwerfen kann, ist die Tatsache, dass sie ihre jahrelangen Beteuerungen, mittels Enum-Patterns wären eigene Enum-Klassen kein Problem, selber Lügen straft. Denn es war (nahezu) unmöglich, eigene Enums zu schreiben, die sich in allen Situationen semantisch korrekt verhalten.²

¹ Der Grund liegt in der Zuordnung `Sun=1` (siehe 3.5.1 den 1. Punkt).

3.1 Grundlagen zu enum-Klassen

Obwohl die JVM für Enumerationen nicht geändert werden musste, waren neben dem Compiler Änderungen in den `class`-Dateien notwendig. Der Serialisierungs-Mechanismus und auch die Bibliotheken von J2SE 5 mussten ebenfalls angepasst werden. Neben dem Schlüsselwort `class` gibt es nun eine weitere Möglichkeit, eine spezielle Art von Klasse für Konstanten anzulegen.

Deklaration von Enum-Klassen :

Eine Enum-Klasse wird wie folgt deklariert:

```
publicopt enum ClassName Interfacesopt {  
    ConstName (Argument,...)opt { AnonymousClassBody }opt ,  
    ...  
    ;ClassBodyopt  
}
```

Man sieht sofort, dass der Aufbau der Klasse nach einem strikten 5-Punkte-Muster abläuft:

1. Zuerst werden die Konstanten aufgezählt. Man kann ihnen – im Gegensatz zu C/C# – keine ganzen Zahlen (per Gleichheitszeichen) zuordnen. Die erste Konstante hat immer die Zuordnung zur ganzen Zahl 0.
2. Hinter den Konstanten können optional in Klammern Argumente für Konstruktoren angegeben werden, sofern man die entsprechenden Konstruktoren im `ClassBody` der `enum`-Klasse auch angelegt hat. Konstruktoren sind implizit `private`.
3. Selten genutzt, trotzdem möglich: Es kann danach noch ein Block `AnonymousClassBody` kommen, der Methoden implementiert, die für alle Konstanten aufgerufen werden können, aber die man für jede Konstante speziell implementieren muss. Faktisch ist das dann eine anonyme Klasse, die Methoden überschreibt, die im `ClassBody` der `enum`-Klasse definiert sind (da man sie ansonsten von außen nicht aufrufen könnte!).
4. Jede Konstante (inkl. ihrer optionalen Teile) wird durch ein Komma von der folgenden Konstante abgegrenzt. Die letzte Konstante muss durch ein Semikolon von dem abschließenden Code der `enum`-Klassen abgegrenzt werden (sofern dieser existiert)..

Aufbau von enum-Klassen

Der Aufbau folgt einem festen Muster, dargelegt in fünf Punkten.

² Das Problem liegt in der Serialisierung und der Verwendung von verschiedenen Class-Loadern. Siehe hierzu einen sehr guten Beitrag unter http://www.javaworld.com/javatips/jw-javatip122_p.html von V. Roubtsov .

- Der abschließende (optionale) `ClassBody` enthält dann allgemeine Felder, Konstruktoren und Methoden zur `enum`-Klasse. Nur die Methoden können `public` deklariert werden, alles Andere ist wieder implizit `private`.

Laut Draft-Spezifikation sind zwar auch `final` und `abstract` vor `enum` erlaubt, dies wurde aber kurzfristig im Mai 2004 „per Order de Mufti“ verboten, da `final` ohnehin automatisch erzeugt wird bzw. alternativ ein `abstract` genau dann, wenn mindestens eine Methode im `ClassBody` abstrakt erklärt wird.³

Anhand einer trivialen Klasse `TimeSpec` zusammen mit dem compiler-generierten Code werden bereits die hauptsächlichen Eigenarten einer `enum`-Klasse gut verständlich:

enum-Klasse	Compiliert zu:
<pre>enum TimeSpec { AM, PM }</pre>	<pre>final class TimeSpec extends Enum { public static final TimeSpec AM; public static final TimeSpec PM; private static final TimeSpec[] VALUES; static { AM= new TimeSpec("AM", 0); PM= new TimeSpec("PM", 1); VALUES = new TimeSpec[] {AM,PM}; } private TimeSpec(String s,int i){super(s,i);} public static final TimeSpec[] values() ... public static TimeSpec valueOf(String s) ... }</pre>

Anhand des generierten Codes sind folgende grundlegenden Eigenschaften sofort nachzuvollziehen:

- Alle `enum`-Klassen werden automatisch von der sogenannten *primordial* Klasse `Enum` abgeleitet. Es ist nicht erlaubt, diese explizit als Superklasse anzugeben, denn sie ist ohnehin die unvermeidbare Mutterklasse, was mit *primordial* auch gemeint ist.
- Eine `enum`-Klasse verhält sich wie eine Utility-Klasse, sie ist `final`, erlaubt also keine Subklassen und schützt sich zusätzlich aufgrund der `private` deklarierten Konstruktoren vor Anlage von Instanzen.

Generierter enum-Code
Mit Hilfe der Klasse Enum erstellt der Compiler den Code zur enum-Klasse.

---

³ Für weitere Einzelheiten siehe 3.5.2 Details.

- Die in der Liste aufgeführten Konstanten sind *self-typed*, sie werden implizit als `static final` vom Typ ihrer eigenen `enum`-Klasse mit Hilfe eines `static` Initializer angelegt. Dabei rufen sie einen Konstruktor von `Enum` mit ihrem Namen als `String` und einer `int`, beginnend mit 0 auf (siehe auch 1. Punkt zum Anlagemuster).
- Einzelne Konstante können über die statische Methode `valueOf(String constName)` geholt werden, wogegen die Methode `values()` das interne Array aller Konstanten – als Clone! – liefert.

3.2 Klasse Enum & ein Idiom

Bis auf die beiden automatisch generierten statischen Methoden `values()` und `valueOf()` erben `enum`-Klassen alles von ihrer Mutter-Klasse `Enum`. Sie ist ein rekursiv deklariertes generisches Beauty:

```
public class Enum<E extends Enum<E>>
    implements Comparable<E>, Serializable {

    // Liefert die Konstante aus einer Enum-Klasse
    // durch Angabe ihres Namens
    public static <T extends Enum<T>>
        T valueOf(Class<T> enumType, String name);

    // Anlage der Konstante mit Namen und
    // Position in der Konstanten-Liste
    Enum(String name, int ordinal);

    // vergleicht zwei Konstanten anhand ihrer
    // Position in der Konstanten-Liste
    public final int compareTo(E o)

    // liefert die Klasse, in der die Konstante
    // deklariert ist
    public final Class<E> getDeclaringClass();

    // Name der Konstanten
    public final String name();

    // Position der Konstante
    public final int ordinal();
```

```

    // Name der Konstanten
    public String toString()
}

```

Die `Object`-Methoden `clone()`, `equals()` und `hashCode()` wurden oben weggelassen. Alle Instanz-Methoden bis auf `toString()` sind `final` deklariert und somit nicht in eigenen `enum`-Klassen überschreibbar. Die Methode `equals()` vergleicht nur die Identität der Konstanten und hat dieselbe Wirkung wie `==`. Beides ist immer korrekt, da Clonen verboten ist.⁴

Idiom: Superklasse deklariert Subklassen-Methoden

Und nun zu der interessanten Frage, warum die Typ-Variablen `E` und `T` auf die gleiche rekursive Art definiert werden? Die Antwort lautet, dass das generische Konstrukt für die Methoden, in denen `E` und `T` verwendet werden, erzwingen will, dass sie nur Sub-Typen von `Enum` akzeptieren. Das kann man auch allgemeiner formulieren:

C<S extends C<S>>

Was will uns dieser Type-Parameter sagen?

Problem: Wie kann man in einer Superklasse eine Methode für alle (noch unbekannt) Subklassen definieren, die für einen Parameter oder das Resultat der Methode nur genau diese Subklasse selbst zulässt?

Lösung: Die Superklasse muss einen Type-Parameter definieren, der nur Subklasse von ihr selbst sein kann. Die dadurch definierte Typ-Variable repräsentiert somit die unbekannt Subklasse und wird überall da verwendet, wo man den Subklassen-Typ benötigt.

Die einfachste Art, diese Lösung in Code umzusetzen, ist diese:

```

SuperClass<SubClass extends SuperClass<SubClass>>

```

Diese Definition ist eine optimal Lösung. Jede `SubClass` der `SuperClass` übergibt sich selbst als Instanz-Argument für die Superklasse und erfüllt dabei als einzige Klasse ihre eigene Bedingung.

Das Idiom wäre keines, wenn es nur für Enumerationen gut wäre. Deshalb muss man es sofort an einem einfachen Beispiel ausprobieren. Die abstrakte Klasse `Part` (Teile) gibt für alle konkreten Subklassen eine Methode `isEquivalent()` vor, die nur zwei Instanzen derselben Subklasse auf Gleichartigkeit prüfen soll. Diese Restriktion ist durchaus sinnvoll, da

⁴ Sollten Sie linear gelesen haben, so sehen Sie unschwer die Parallelität zu den Objekten der primitiven Wrapper-Klassen. Bei Enumerationen kann man also nicht in die Falle verschiedener Identitäten für denselben Wert laufen (siehe Abschnitt 3.4.1).

man nicht Äpfel mit Birnen oder einen Drucker mit einer Festplatte vergleichen will.

```
// die Type-Variable P repräsentiert eine
// Subklasse von Part
public abstract class Part<P extends Part<P>> {
    protected Part() {}

    // P kann nur dieselbe Subklasse sein und
    // fixiert die Implementation
    public abstract boolean isEquivalent(P part);
}

// Printer übergibt sich Part selbst als Typ
public class Printer extends Part<Printer> {
    // nur Drucker vom gleichen Typ sind äquivalent
    public enum PrinterType {INKJET,LASER,MATRIX};

    private PrinterType pt;

    public Printer(PrinterType typ) {
        pt= typ;
    }

    // diese Methoden-Signatur wurde in der
    // Superklasse Part erzwungen
    public boolean isEquivalent(Printer other) {
        return pt==other.pt;
    }
}
```

Interessant ist der statische Import von Enums, die in einer anderen Klasse enthalten sind. Dies wird im folgenden Test ausgenutzt:

```
// Annahme: Part und Printer im selben Package
import static galileo.kap02.Printer.PrinterType.*;

public class PartTest {
    public static void main(String[] args) {
        // durch statischen Import einfacher Zugriff
        // auf Konstante
        Printer p1= new Printer(INKJET);
        Printer p2= new Printer(LASER);
    }
}
```

```

Printer p3= new Printer(INKJET);

System.out.println(p1.isEquivalent(p2)); ▶ false
System.out.println(p1.isEquivalent(p3)); ▶ true
}
}

```

Änderung der Serialisierung

Die clever konstruierte `Enum`-Superklasse und der vom Compiler automatisch generierte Code entsprechen dem seit Jahren propagierten Java-Idiom für Enumerationen von Sun. Abgesehen davon, dass alleine der Aufwand einer manuellen Lösung wie dieser erheblich ist, würde es die Einzigartigkeit der `enum`-Klassen trotzdem nicht schützen. Denn über die Serialisierung kann man das alles wieder aushebeln. Also hat Sun noch ein wenig nachgeholfen und den Serialisierungs-Mechanismus im Compiler sowie das zugehörige Protokoll abgeändert, so dass es nun nicht möglich ist, per Serialisierung weitere Instanzen zu erschaffen. Das war entscheidend und in dieser Form nur Sun möglich.⁵

3.3 Details zu enum-Klassen

Bis hierher sind `enum`-Klassen auch nichts anderes als „glorified Integers“, sogar mit der Einschränkung, dass der ersten Konstanten immer die ganze Zahl `0` zugeordnet ist. Im Folgenden werden die Details besprochen, die den Qualitätssprung beweisen sollen.

Konstruktoren, Methoden

Betrachtet man im Beispiel aus 3.5.1 den automatisch generierten Konstruktor, so ist dieser `private` und hat die Signatur `(String constName, int ident)`. Schreibt man zusätzlich eigene Konstruktoren, müssen sie entweder auch `private` oder am besten ohne Modifier deklariert sein. Sie können sich gegenseitig mit `this()` aufrufen, der Aufruf von `super()` ist jedoch verboten. Der Grund liegt darin, dass der Compiler die Signatur der selbstgeschriebenen Konstruktoren um `String, int` ergänzen muss, um wieder mittels `super(String, int)` den Konstruktor von `Enum` automatisch aufrufen zu können.

Ergänzen wir also `enum TimeSpec` (aus Beispiel aus 3.5.1) um zwei Konstruktoren und vergleichen ihn mit dem compiler-generierten Code:

enum-Klasse: ClassBody
Verwendung von Konstruktoren und Methoden im enum-Klasse-Körper.

⁵ Wer sich für die Details der Serialisierung interessiert, findet sie bei Sun in einem entsprechenden Dokument unter dem Link <http://java.sun.com/j2se/1.5/pdf/serial-1.5.0.pdf>

enum Konstruktoren, Body	Compiliert zu:
<pre>enum TimeSpec { AM, PM(12,24); // --- ClassBody --- private int from,to; TimeSpec() { this(0,12); } TimeSpec(int from,int to){ this.from= from; this.to= to; } public int getFrom() { return from; } public int getTo() { return to; } public String toString(){ return super.toString() +"["+from+",""+to+""; } }</pre>	<pre>final class TimeSpec extends Enum { ... private TimeSpec(String s,int i) { this(s,i,0,12); } private TimeSpec(String s,int i, int from,int to) { super(s, i); this.from = from; this.to = to; } ... }</pre>

Eigene Konstruktoren sind dazu da, hinter den Konstanten in Klammern weitere Argumente angeben zu können. Soll dies nur optional sein benötigt man zusätzlich einen No-Arg-Konstruktor, der mittels `this()` dann ein Chaining zu einem komplexeren Konstruktor durchführt. Da `toString()` in Klasse `Enum` die einzige nicht-`final` erklärte Methode ist, wurde sie im Beispiel überschrieben. Das macht nur Sinn, wenn man zusätzliche Felder in der `enum`-Klasse hat, deren Werte wichtige Zusatzinformationen zur Konstanten darstellen. Die Zusatzinformationen sind bei `TimeSpec` die Felder `from`, `to`. Es bietet sich also an, sie hinter den Konstanten-Namen anzugeben.

Hier wieder der obligatorische Test:

```
import static package.TimeSpec.*;

// Testklasse und main weggelassen

System.out.println(AM.getFrom()+".."+
    AM.getTo());           ▶ 0..12
System.out.println(AM +" "+"+PM);           ▶ AM[0,12];PM[12,24]

for (TimeSpec ts: TimeSpec.values())
    System.out.println(ts);           ▶ AM[0,12]
                                       ▶ PM[12,24]

System.out.println(
    TimeSpec.valueOf("AM"));           ▶ AM[0,12]
System.out.println(
    Enum.valueOf(TimeSpec.class,"AM")); ▶ AM[0,12]
```

toString() vs. valueOf()

Die letzte beiden Anweisung des o.a. Beispiels zeigen den Einsatz der gleichnamigen statischen `valueOf()`-Methoden von `Enum` und einer `enum`-Klasse. Da `toString()` eine Konstante in einen String umwandelt und `valueOf()` eine String benötigt, um die Konstante einer `enum`-Klasse zu holen, könnte man die Meinung vertreten, `valueOf()` müsste sich irgendwie an `toString()` „anpassen“:⁶

Frage: Welchen String gibt man bei `valueOf()` an, um die Konstante einer `enum`-Klasse zu bekommen?

Antwort: Man kann nur den Namen der Konstanten angeben, ohne Rücksicht auf das, was `toString()` liefert.

Der Grund ist einfach. Beide statische Methode können nicht überschrieben werden. Die Implementationen der `valueOf()`-Methoden in `Enum` und den `enum`-Klassen bleibt immer so wie sie sind und können also auch nur so und nicht anders aufgerufen werden (siehe Beispiel oben). Wollte man überhaupt etwas ändern, müsste man in der eigenen `enum`-Klasse eine zusätzliche statische Methode `valueOf()` mit einer anderen Signatur schreiben. Was bringt das? Mehr Arbeit!

Anpassen von valueOf()?

Die statische Methoden `valueOf()` von `Enum` und den `enum`-Klassen sind unantastbar. Sie können nicht geändert werden.

---

⁶ Induziert wird dies Meinung bzw. Anforderung dadurch, dass der String, den die `toString()`-Methode liefert, im Idealfall – übergeben an einen Konstruktor, der einen String erwartet – wieder eine Art Clone-Objekt des Original-Objekts liefern soll.

Anonyme Klassen der Konstanten

Sun sieht den Einsatz von anonymen Klassen, die optional hinter den Konstanten stehen können, eher als Ausnahme an, womit sie wahrscheinlich Recht haben. In der eingangs vorgestellten `enum`-Deklaration bezeichnet der Block `AnonymousClassBody` nichts anderes als eine anonyme Klasse, die die eigene `enum`-Klasse als Superklasse hat (siehe auch 3. Punkt in Abschnitt 3.5.1). Wird also in der `enum`-Klasse selbst (aus Versehen) eine Methode `abstract` erklärt, muss zwangsläufig hinter jeder Konstanten ein nachfolgender Block stehen, der diese abstrakte Methode implementiert. Nicht jeder will zu diesem „absurden“ Fall unbedingt Code sehen. Trotzdem wird der Vollständigkeit halber in der Klasse `TimeSpec` eine abstrakte `foo()`-Methode deklariert, die ihren Namen redlich verdient hat. Sie führt dann bei der Implementation hinter den Konstanten zu zwei anonymen Klassen `TimeSpec$1.class` und `TimeSpec$2.class`.

```
enum TimeSpec {
    AM {
        // anonyme Klasse, die foo() implementieren muss
        void foo() {System.out.println("AM foo"); }
    },
    PM(12,24) {
        // dito
        void foo() {System.out.println("PM foo");}
    };

    // --- ClassBody ---
    private int from,to;

    TimeSpec() { this(0,12); }
    TimeSpec(int from,int to) {
        this.from= from;
        this.to= to;
    }

    // hiermit werden alle Konstanten gezwungen,
    // eine anonyme Klasse zu implementieren.
    abstract void foo();
}
```

Auch wer für solche akademischen Spielereien kein Verständnis aufbringt, kann schnell in eine ähnliche Situation geraten, und zwar dann, wenn Interfaces implementiert werden, bei denen z.B. eine Methode für bestimmte Konstante individuell implementiert werden muss.

Implementation eines Interfaces

Sucht man nach einem passenden Beispiel, ist die erste Reaktion »Ich könnte Ihnen viele Beispiele nennen, wenn ich nur Eines wüßte«. Auch hier muss ein kleines „an den Haaren“ herbeigezogenes Beispiel die sehr individuelle Implementation eines Interfaces in einer `enum`-Klasse demonstrieren, und zwar gleich in zwei Versionen.

Geldwährungen (Klasse `Currency`) werden anhand bekannter konstanter Idents an der Börse gehandelt. Der Einfachheit halber nehmen wir nur drei und tragen in die Wechselkurs-Tabelle (`exchangeTable`) feste Werte ein.⁷ Weiterhin nehmen wir einmal an, dass – aus welchen Gründen auch immer – für eine Währung, z.B. für den Euro, ein Disagio von 2% beim Wechsel in eine andere Währung erhoben wird.⁸ Und nun setzen wir dies in minimalen `enum`-basierten Code um:

```
interface Convertable {
    double exchangeRate(Currency cId);
}

enum Currency implements Convertable {
    DOLLAR,
    YEN,
    EURO
    // die anonyme Klasse von EURO
    {
        public double exchangeRate(Currency cId) {
            return this==cId? 1.0:
                // --- geht auch mit ordinal() ---
                // return ordinal()===cId.ordinal()? 1.0:
                exchangeTable[ordinal()][cId.ordinal()]*0.98;
        }
    };

    // --- ClassBody ---
    static double[][] exchangeTable =
        { {1.0, 1.07, 0.81},
          {0.0092, 1.0, 0.0076},
          {1.22, 131.3, 1.0} };
}
```

⁷ Das ist bereits unrealistisch, kann aber – wie mir von Studenten glaubhaft versichert wurde – beliebig komplex und dynamisch erweitert werden. Man nennt das dann „Eine internet-basierende Investor-Broker-Applikation“ und die beschäftigt nicht nur ein Semester, sondern auch Programmier-Teams in Banken.

⁸ Auch das ist wieder unrealistisch, da der Euro zur Zeit Stärke beweist. Aber als guter Europäer will man ja nicht gleich eine andere Währung desavouieren.

Interfaces zu Enums

Interfaces können in `enum`-Klassen auf zwei Arten implementiert werden.

```

// die normale Implementation für DOLLAR und YEN
public double exchangeRate(Currency cId) {
    return exchangeTable[ordinal()][cId.ordinal()];
}
}

public class EnumTest {
    public static void main(String[] args) {
        for (Currency c1: Currency.values()) {
            for (Currency c2: Currency.values())
                System.out.print(c1.exchangeRate(c2)+" ");
            System.out.println();
        }
    }
}

```

Konsol-Ausgabe:

```

1.0 1.07 0.81
0.0092 1.0 0.0076
1.1956 128.674 1.0

```

Diese merkwürdige anonyme Klasse kann natürlich auch durch einen einfacheren Code ersetzt werden:

```

enum Currency implements Convertable {
    DOLLAR,
    YEN,
    EURO;

    static double[][] exchangeTable =
        { {1.0, 1.07, 0.81},
          {0.0092, 1.0, 0.0076},
          {1.22, 131.3, 1.0} };

    public double exchangeRate(Currency cId) {
        if (this!=EURO)
            return exchangeTable[ordinal()][cId.ordinal()];
        else
            return this==cId? 1.0:
                exchangeTable[ordinal()][cId.ordinal()]*0.98;
    }
}

```

Anonym vs. if/switch

Welcher Code klarer ist, hängt vom Problem ab.

Ohne zu orakeln, wird zu 99% der letzte Code geschrieben, was aber in nicht-trivialen Fällen zu Spagetti-Code führt. Denn irgendwie erinnert alles zu sehr an die ultra-coole Graphik-Methode `drawShape()`, die für alle möglichen Figuren (`Shape`) mittels `if/switch` den passenden Code implementiert. Kommt eine Figur hinzu oder wird eine geändert, muss man halt wieder irgendwo tief im `if/switch` herumbasteln – sofern man den Source-Code hat – um die Sache anzupassen und anschließend stabil zu bekommen.

SOC-Variante: ⁹

Nach dem *Separation-of-Concerns* (SOC) Prinzip überlässt man das spezifische Verhalten einer Klasse tunlichst ihr selbst.

Verletzt man SOC und bastelt an Klassen „von außen“ herum, muss man zwangsläufig an ihre Details, die Internas heran und wenn sich da etwas ändert, hilft an sich nur Beten.

Betrachtet man die Konstante `EURO` als den Namen der folgenden anonymen Klasse, so ist der Code genau da, wo er hingehört. Kommt eine Konstante hinzu oder ändert sich das Verhalten einer Konstanten, wird auch genau dort der Code ergänzt bzw. geändert. In den meisten Fällen zeigen Konstante allerdings kein komplexes Verhalten und dann ist `if/switch` in der Methode des `ClassBody` unschlagbar.

Idents anonymer Klassen

Konstante mit komplexem Verhalten repräsentieren quasi die Namen ihrer anonymen Klassen.

3.4 for-, switch-Anweisung

Zum automatisch generierten Code einer enum-Klasse gehört die Methode `values()`, die ein Array der Konstanten in der Reihenfolge ihrer Deklaration zurückliefert. Damit ist die Iteration über alle oder von-bis-Bereiche von Konstanten problemlos möglich. Denkt man bei Enums an „glorified Integers“, so ist es ebenfalls nicht abwegig, die `switch`-Anweisung so zu erweitern, dass sie auch `enum`-Konstanten akzeptiert:

enum-Erweiterung von switch:

Zusätzlich zu dem Typ `int` lässt `switch` auch einen `enum`-Ausdruck zu, wobei als `case`-Label die Konstanten der zugehörigen `enum`-Klasse verwendet werden:

```
switch (enumExpression) {
    case EnumConstant:
        ...
}
```

Als `EnumConstant` darf nur der einfache Name (ohne vorangestellter `enum`-Klassen) angegeben werden.

---

⁹ Das Separation of Concerns Konzept ist sehr vielfältig (siehe www.research.ibm.com/hyperspace).

Da `switch{enum}` nichts sensationell Neues darstellt und für C/C++ Programmierer schon lange zum Alltag gehört, reicht wohl eine einfaches Beispiel, um den Mechanismus zu verstehen:

```
// 3 RGB-Konstanten ohne Eigenschaften und Verhalten
public enum RGB {
    RED, GREEN, BLUE
}

// in einer anderen Compilations-Einheit:
public class SwitchTest {
    public static void main(String[] args) {
        RGB[] rgbVal= RGB.values();
        for (RGB c: rgbVal)
            System.out.println(c);
        // Test der letzten Konstanten
        switch (rgbVal[rgbVal.length-1]) {
            case RED:
                System.out.println("Rot"); break;
            case GREEN:
            case BLUE:
                System.out.println("Grün oder Blau"); break;
            default:
                throw new AssertionError("Keine RGB-Farbe");
        }
    }
}
```

Konsol-Ausgabe:

```
RED
GREEN
BLUE
Grün oder Blau
```

Eine triviale Implementation würde die Konstanten im `case` durch ihre Ordinal-Zahlen austauschen, in diesem Fall `RED`, `GREEN`, `BLUE` durch `0`, `1`, `2`. Das Problem dabei ist nur, dass dies bei getrennt compilierten Klassen `RGB.class` und `SwitchTest.class` nicht funktioniert. Denn erweitern wir `RGB` nachträglich um eine Konstante, ohne dass wir `SwitchTest.java` neu compilieren, würde die Ausführung von `SwitchTest.class` zu der Ausnahme `AssertionError` führen (der `default`-Fall). Testen wir mit einer Erweiterung, die für `Red`, `Green`, `Blue` die Nummern erhöht:

```
public enum RGB {  
    NULL, RED, GREEN, BLUE  
}
```

Führen wir die alte `SwitchTest.class` ohne Re-Compilation mit der neuen `RGB.class` aus, gibt dies kein `AssertionError`, sondern die an sich erhoffte Ausgabe:

```
NULL  
RED  
GREEN  
BLUE  
Grün oder Blau
```

Dieses Magic heißt *Indirektion* und wird häufig auch bei der Adressierung von Speichern eingesetzt. Anstatt mittels eines Pointers direkt den Speicher zu adressieren, verwendet man ein Handle – eine sogenannte logische Speicheradresse –. Mit dem Handle ist dann die reale Speicheradresse assoziiert. Alle Klienten, die nur über das Handle den Speicher benutzen, bleiben von Adressänderungen unberührt. Nur der Pointer im Handle muss natürlich umgesetzt werden.

Eine ähnliche Strategie wird auch vom Compiler für Enums gewählt. Eine `switch{enum}` verwendet als `case`-Label nur Handles.¹⁰ Gäbe es allerdings eine feste Zuordnung der Handles zur Konstanten in derselben `class`-Datei, wäre das Problem nicht unbedingt weg. Deshalb lagert der Compiler eine `SwitchMap`, die die Konstanten den Handles zuordnet, in einer statischen inneren Klasse aus. Diese wird in eine eigene `class`-Datei compiliert, die beim Laden mit Hilfe des statischen Initialisierers dynamisch die statisch deklarierte `SwitchMap` aufbaut. Problem beseitigt!¹¹

¹⁰ Die übrigens in Java 1.5 beta 3 mit 1 und nicht mit 0 beginnen.

¹¹ Dies lässt sich mit Pseudo-Code zur Switch-Map und der statischen Initialisierung recht eindrucksvoll belegen, ist aber zur Anwendung nicht unbedingt notwendig.

Dynamisches switch

Die Label in switch-case verwenden nicht die Ordinal-Nummern der Enum-Konstanten.

---

switch verwendet Handles

switch ist unabhängig von Änderungen bei enum-Konstanten.

Diese Dynamik hat den (sehr kleinen) Nachteil, dass das Einfügen von Konstanten zu Inkonsistenzen im `switch` führt, sofern man immer alle Konstanten behandeln will. In vielen Fällen will oder kann man das gar nicht, weil man ohnehin nur für die zu diesem Zeitpunkt bekannten Konstanten Code schreiben kann. Aber für den „Alles-Behandeln-Fall“ gibt es ja das `default`. Hier kann man im schlimmsten Fall eine `AssertionError` erzeugen, der z.B. auch beim `SwitchTest` oben durch die neue Konstante `NULL` ausgelöst wird.

3.5 EnumSet, EnumMap

`EnumSet` bzw. `EnumMap` sind aufgrund ihrer Deklaration voll in das Collection-Framework eingebettet. Sie sind sogenannte High-Performance-Implementierungen, Werden z.B. höchstens 64 Konstante definiert, kann `EnumSet` Set-Operationen auf Bit-Manipulationen mit einer `long`-Variable abbilden, was natürlich die Sache ungemein beschleunigt. Im Folgenden wird nicht auf die interne Implementation eingegangen, sondern auf die Verwendung der beiden Klassen. Die Klassen sind nicht thread-sicher, d.h. alle Methoden sind nicht synchronisiert. Da sie das aber mit den „normalen“ Containern in `java.util` gemein haben, muss man halt die gleichen Vorsichtsmaßnahmen ergreifen.

High Performance

EnumSet und EnumMap sind hoch performante Implementierungen.

EnumSet

Für `EnumSet` wird zugesichert, dass alle Basis-Operationen in konstanter Zeit ablaufen und das in der Regel weitaus schneller als in einer normalen `HashSet`, mithin ist es attraktiv genug, um besprochen zu werden. Über einen Type-Parameter stellt `EnumSet` sicher, dass es nur mit genau einer `enum`-Klasse verwendet wird. Da `EnumSet` zusätzlich `abstract` deklariert ist, können keine Konstruktoren verwendet werden, die die Zukunft dieser Klasse verdunkeln würden. Dafür gibt es statische Factory-Methoden für viele verschiedene Formen der Anlage. Hier nur die speziellen Factory-Methoden der Klasse:

```
public abstract class EnumSet<E extends Enum<E>>
    extends AbstractSet<E>
    implements Cloneable, Serializable {

    // Factory-Methoden
    // Als Argument muss das Klassen-Objekt .class
    // einer
    // enum-Klasse übergeben werden.
    // Erzeugt eine leere EnumSet, die ausschließlich
    // Konstante dieser Klasse akzeptiert
    public static <E extends Enum<E>>
```

EnumSet Factories

EnumSet-Objekte werden ausschließlich über Factory-Methoden erzeugt.

```

        EnumSet<E> noneOf(Class<E> elementType);

// wie noneOf, aber EnumSet enthält alle Konstante
public static <E extends Enum<E>>
    EnumSet<E> allof(Class<E> elementType);

// Erzeugt einen Clone des EnumSet-Arguments
public static <E extends Enum<E>>
    EnumSet<E> copyOf(EnumSet<E> s);

// Erzeugt Komplement des EnumSet-Arguments
public static <E extends Enum<E>>
    EnumSet<E> complementOf(EnumSet<E> s);

// Erzeugt EnumSet mit genau 1 bis 5 Konstanten
public static <E extends Enum<E>>
    EnumSet<E> of(E e
        [,E e2]opt [,E e3]opt [,E e4]opt [,E e5]opt);

// Erzeugt EnumSet mit mehr als 6 Elementen
public static <E extends Enum<E>>
    EnumSet<E> of(E first, E... rest);

// Erzeugt EnumSet mit Konstanten zwischen from, to
// inklusive from und to
public static <E extends Enum<E>>
    EnumSet<E> range(E from,E to);

//...
}

```

Die VarArgs-Variante von `of()` wird aufgrund ihrer Eigenschaft als „geringst-spezifische“ Methoden (siehe Abschnitt 3.4.2 „maximal spezifische Methoden“) erst ab 6 Konstanten aufgerufen. Außer den o.a. Factory-Methoden erbt `EnumSet` natürlich alle regulären `Set`-Methoden, die hier deshalb nicht aufgeführt werden. Die Verwendung von `EnumSet` zeigt das folgende Beispiel:

```

package galileo.kap02;
// eine sehr subjektive Sprachzusammenstellung
public enum Language {
    C, CPP, CSHARP, JAVA, PHP, PERL, PYTHON, RUBY, GROOVY
}

```

```
// Der Einsatz der Factory-Methoden
import java.util.*;
import static java.util.EnumSet.*;
import static galileo.kap02.Language.*;

public class EnumSetTest {
    public static void main(String[] args) {
        EnumSet[] esArr= new EnumSet[] {
            noneOf(Language.class),
            allOf(Language.class),
            of(C, JAVA, GROOVY),
            of(C, CPP, CSHARP, JAVA, PERL, PYTHON, PHP),
            complementOf(of(JAVA)),
            range(PERL, PHP) };
        for (EnumSet es: esArr)
            System.out.println(es);
    }
}
```

```
[ ]
[C, CPP, CSHARP, JAVA, PERL, PYTHON, PHP, RUBY, GROOVY]
[C, JAVA, GROOVY]
[C, CPP, CSHARP, JAVA, PERL, PYTHON, PHP]
[C, CPP, CSHARP, PERL, PYTHON, PHP, RUBY, GROOVY]
[PERL, PYTHON, PHP]
```

Vorsicht bei range()

Die range()-Methode reagiert mit einer Exception auf falsche Grenzen.

Bis auf die Factory-Methode `range()` sieht das so weit recht gut aus. Die `range`-Methode setzt aber voraus, dass `PERL` in der Aufzählung vor `PHP` kommt. Wird `Language` nachträglich so verändert, dass `PHP` vor `PERL` steht, gibt es beim Aufruf der alten `EnumSetTest.class` eine Ausnahme:¹²

```
Exception ... IllegalArgumentException: PERL > PHP
```

EnumMap

Wie erwartet, assoziiert eine `EnumMap` Konstante mit Werten aus einer anderen Klasse. Da bei der Anlage der Map implizit oder explizit die `enum`-Klasse übergeben wird, steht die Anzahl der Konstanten (und damit die Maximalanzahl der Werte) vorab fest, was wie bei der `EnumSet` zu Array-Operationen mit extrem effizienten Zeitverhalten führt. Bei der

¹² Abgesehen davon, dass man so etwas Schreckliches auch nicht machen soll, wäre eine leere Menge anstatt einer Ausnahme auch keine schlechte Lösung gewesen.

`EnumMap` haben sich die Entwickler im Gegensatz zu `EnumSet` entschieden, bei Konstruktionen zu bleiben:

```
public class EnumMap<K extends Enum<K>, V>
    extends AbstractMap<K, V>
    implements java.io.Serializable, Cloneable {

    // Als Argument muss das Klassen-Objekt .class einer
    // enum-Klasse übergeben werden.
    // Erzeugt eine leere EnumMap, die ausschließlich Konstante
    // aus dieser Klasse als Schlüssel akzeptiert
    public EnumMap(Class<K> keyType);

    // Erzeugt einen Clone einer existierende EnumMap
    public EnumMap(EnumMap<K,? extends V> m);

    // Erzeugt einen Clone einer existierenden normalen Map
    // sofern diese mindestens 1 Element mit einer
    // Mapping enthält, um den Schlüssel zu ermitteln
    public EnumMap(Map<K, ? extends V> m);
    //...
}
```

In dem nun folgenden Beispiel werden zusätzlich wieder die neuen `StringBuilder` und `Formatter` Klassen benutzt. Es gibt zwei Enums, die erste `Country` enthält drei Länder-Codes und legt für diese fiktive Importzölle fest. Mit Hilfe der Klasse `Product` werden vier Produkte angelegt, die von den Ländern importiert werden können. Die Produkte haben neben dem Namen einen durchschnittlichen Wert pro Einheit. Die Einheiten sind als Konstanten in der zweiten Enum `Unit` definiert. In die `EnumMap` werden die Länder als Schlüssel und die Arrays der Importmengen als Werte abgelegt. Die Methode `importToString()` wandelt die Importdaten mit Hilfe der Klasse `Formatter` in einen formatierten String um. Der Einfachheit halber sind alle Klassen in einer Compilations-Einheit zusammengefasst.

```
1: package galileo.kap02;
2:
3: import java.util.*;
4: import java.text.*;
5: import java.math.*;
6:
7: // Zugriff auf die Konstanten über einfache Namen
8: import static galileo.kap02.Country.*;
9: import static galileo.kap02.Unit.*;
```

```

10: import static galileo.kap02.Product.PRODUCTS;
11:
12: enum Country {
13:     USA,EU,RUS;

14:     // 10%, 12% und 15% Importzoll
15:     public double basicImportTax() {
16:         switch (this){
17:             case USA: return 0.1;
18:             case EU:  return 0.12;
19:             case RUS: return 0.15;
20:             default:  return Double.NaN;
21:         }
22:     }
23: }
24:
25: // NUM= Stückzahl, M= Meter, M2= m², M3= m³
26: enum Unit {
27:     NUM, KG, M, M2, M3
28: }
29:
30: // Product ist immutable
31: class Product {
32:     // eine „Datenbank“ von Produkten
33:     // keine DBMS, aber zum Testen reicht es
34:     public static final Product[] PRODUCTS;
35:
36:     private String name;
37:     private Unit u;
38:     private double val; // $ Wert pro Einheit
39:
40:     // statischer Initialisierer
41:     static {
42:         PRODUCTS= new Product[] {
43:             new Product("Bananen",KG,0.5) ,
44:             new Product("Erdgas",M3,0.1) ,
45:             new Product("IntelP4/2.8",NUM,180.0) ,
46:             new Product("TFTPanel",M2,3900.0) ,
47:         };
48:     }
49:

```

```

50: public Product(String name,Unit u,double valUSD) {
51:     this.name= name; this.u= u; val= valUSD;
52: }
53:
54: // Getters
55: public String getName() { return name; }
56: public Unit  getUnit() { return u; }
57: public double getUSDValue() { return val; }
58: }
59:
60: public class EnumMapTest {
61:     // Key=   Country-Konstante
62:     // Value= vier Importzahlen
63:     static EnumMap<Country,double[]> import2003=
64:         new EnumMap<Country,double[]>(Country.class);
65:
66:     // Initialisierer übernimmt Aufbau der Map
67:     static {
68:         import2003.put(USA,new double[]{4e6,8e8,8.1e5,2e5});
69:         import2003.put(EU, new double[]{2e6,2e9,2.3e6,1.6e5});
70:         import2003.put(RUS,new double[]{1e7,0,2e5,1.2e5});
71:     }
72:
73:     // Formatierter String mit Importzahlen zu
74:     // Land und Einfuhrwerte
75:     static String importToString(Country c,
76:                                 double[] quantity) {
77:         // zumindest eine rudimentäre Prüfung der
78:         // Precondition (Vorbedingung)
79:         if (quantity==null ||
80:             PRODUCTS.length!= quantity.length)
81:             throw new IllegalArgumentException("quantity");
82:
83:         StringBuilder sb= new StringBuild(c.toString());
84:
85:         // formatter legt alles in sb ab.
86:         // Formatierung von Zahlen in Deutsch
87:         Formatter formatter=
88:             new Formatter(sb,Locale.GERMANY);
89:

```

Anlage einer EnumMap

Key: Länder-Code-Konstante
Value: Array von double

Formatter & Locale

Tabellarische Ausgabe von Zahlen im lokalen Format.

```

90:     // Bedeutung: -12s:
91:     // 12 Stellen, String linksbündig
92:     // denn der Default ist rechtsbündig
93:     formatter.format("%-12s %13s %16s%n",
94:                     ": Produkt", "Menge", "Steuer");
95:
96:     for (int i=0; i<PRODUCTS.length; i++) {
97:         // Bedeutung: ,15.0f Trenner für Tausender
98:         //           15 Dezimalstellen mit 0 Nachkomma
99:         formatter.format("%-12s %,15.0f %-3s %,15.2f\n",
100:        PRODUCTS[i].getName(),
101:        // die Formatierung von großen double-Werten
102:        // führt zu Fehlern in der Sun-Library
103:        // deshalb die interessante Variante BigDecimal
104:        new BigDecimal(quantity[i]),
105:        PRODUCTS[i].getUnit(),
106:        new BigDecimal(quantity[i]*
107:            PRODUCTS[i].getUSDValue()*
108:            c.basicImportTax()));
109:     }
110:     return sb.toString();
111: }
112:
113: // abschließend Test mit EU
114: public static void main(String[] args) {
115:     System.out.print(importToString(EU, import2003.get(EU)));
116: }
117: }

```

Konsol-Ausgabe:¹³

EU: Produkt	Menge	Steuer
Bananen	2.000.000 KG	120.000,00
Erdgas	2.000.000.000 M3	24.000.000,00
IntelP4/2.8	2.300.000 NUM	49.680.000,00
TFTPanel	160.000 M2	74.880.000,00

Enums mit Hilfsklassen sind eine Bereicherung der Sprache, die bereits viel zu lange aus sich warten ließ. Aber nun sind sie da und man kann sich anderen Neuerungen zuwenden.

---

¹³ Die Import-Mengen und Zölle sind frei erfunden. Übereinstimmungen mit realen Werten sind rein zufällig und nicht intendiert.

3.6 Advanced Enums & JSR-108

Dieser Abschnitt ist bewusst an das Ende von Enums geraten, da er übersprungen werden kann. Andererseits wird hier eine erweiterbare Technik vorgestellt, die mit Hilfe von Interfaces und Reflexion weitaus komplexere Probleme angeht als die bisher gezeigten. Um dies zu demonstrieren, suchen wir uns in diesem Fall auch ein ernsthaftes Problem¹⁴ aus, das zu einer eher inaktiven JSR-108 – Java Units Specification – geführt hat. Im Folgenden wird eine radikal andere Lösung als die der Implementation von J.A.D.E¹⁵ angedeutet, die so noch nicht vorgeschlagen werden konnte, da es ja auch kein J2SE 5 gab.

Kurz zum Problem: Nicht nur in der Naturwissenschaft, sondern auch in anderen Disziplinen wie der Wirtschaftswissenschaft gibt es Dinge, die man messen muss. Ein allgemeiner Begriff hierfür ist *Dimension*. Dazu gehören Dinge wie Masse, Länge, Zeit und Geld, wobei die letzte unsere jetzige menschliche Zivilisation dominiert. Dimensionen können nun in verschiedenen konstanten Einheiten – *Units* – gemessen werden. Das Problem bei jeder Programmiersprache ist, dass ihr Zahlensystem prinzipiell dimensionslos ist. Wird also eine Methode, die ein Argument einer bestimmten Dimension und Maßeinheit erwartet, (ohne es zu wissen) mit unterschiedlichen Maßeinheiten aufgerufen, ist das Geschrei gross. Die Methode liefert Ergebnisse, aber leider die falschen. JSR-108 ist also wichtig, nicht trivial und eine gute Implementation spart die wichtigste Dimension (Geld).

Was bietet sich besser an, als Dimensionen und Units in Enums zu gießen (zumal JADE die Zusammenhänge ziemlich umständlich implementiert). Eine sehr einfache Lösung (und deshalb in diesem „Advanced“-Abschnitt nicht weiter verfolgt) besteht darin, Dimensionen und/oder Units in EnumSets zusammenzufassen. Nachfolgend wird eine interessante Alternative geboten, die zum Nachdenken, Erweitern und zur Wiederverwendung in anderen Bereichen anregen soll. Der nachfolgende Code ist allerdings nur eine Machbarkeits-Studie (mehr ist hier nicht möglich). Er zeigt aber recht schön, wie man u.a. das Problem, das eigene Enums nicht erweiterbar sind – da sie `final` sind – umgehen kann. Aber wer lässt sich davon schon beeindrucken, wenn es Interfaces gibt? Denn die darf eine `enum`-Klasse ja implementieren oder aber statisch importieren:

JSR-108

Dimensionen und Units sind in vielen Methoden implizit enthalten und entscheiden über die Brauchbarkeit des Ergebnisses.

---

¹⁴ Zumindest für die Raumfahrt, wenn man an den Verlust des Mars-Orbiters 1999 denkt, der aufgrund von Verwechslungen des metrischen mit dem englischen System denkt.

¹⁵ Siehe <http://jade.dautelle.com>

```

package galileo.kap02;

import java.lang.reflect.*;

// Dieses Interface ist ein Wrapper
// für physikalische Basis-Einheiten
public interface Unit<U extends Unit<U>> {
    enum Length implements Unit<Length> {
        MM,CM,M,KM;

        // Convertierungs-Tabelle
        static final double[][] CTABLE= {
            {1.,10.,1000.,1e6},
            {0.1,1.,100.,1e5},
            {0.001,0.01,1.,1000.},
            {1e-6,1e-5,0.001,1.}
        };

        // siehe unten!
        public boolean isLinear() {
            return true;
        }

        public double convert(Length u) {
            return CTABLE[this.ordinal()][u.ordinal()];
        }
    }

    enum Temperature implements Unit<Temperature> {
        F,C,K;

        public boolean isLinear() {
            return true;
        }

        // Implementation wurde weggelassen!
        public double convert(Temperature u) {
            return Double.NaN;
        }
    }
}

```

Enum-Wrapper-Interface

Das Interface Unit ist Wrapper und gleichzeitig Super-Typ seiner statischen inneren enum-Klassen.

```

enum Mass implements Unit<Mass> {
    MOLE, MG, G, KG;

    public boolean isLinear() {
        // Wegen MOLE wahrscheinlich
        return false;
    }

    // Implementation wurde weggelassen!
    public double convert(Mass u) {
        return Double.NaN;
    }
}

// erzwingt convert mit Unit gleicher Dimension
double convert(U to);

// true:  convert(u1+u2)==convert(u1)+convert(u2) und
//        convert(n*u)==n*convert(u)
// false: sonst
boolean isLinear();
}

```

Es folgt die zugehörige Dimension:

```

import static galileo.kap02.Unit.*;

public enum Dimension {
    LENGTH(Length.class),
    MASS(Mass.class),
    TEMPERATURE(Temperature.class);

    // Class-Datei der zugehörigen Unit
    Class<? extends Unit<?>> unit;

    // setzt Class-Datei der zugehörigen Unit
    Dimension(Class<? extends Unit<?>> unit) {
        this.unit= unit;
    }

    // geht leider nicht: <? extends Enum & Unit<?>>
    // geht auch nicht:  <U extends Enum & Unit<?>>
}

```

Enum<->Enum-Relation

Den Dimension-Enums werden per Class-Datei die Unit-Enums zugeordnet.

```

public Unit<? extends Unit<?>>[] getUnit() {
    try {
        Method m= unit.getMethod("values");
        // Unit<? extends Unit>[] erzeugt Warnung, also:
        return (Unit<?>[])m.invoke(null);
        // sollte nicht passieren!
    } catch (Exception e) {
        return null;
    }
}
}

```

Es gibt zwei Gründe für diese Design-Studie:

- Alles – inklusiver des späteren Test-Programms – wird trotz unvermeidlicher Casts vom Compiler ohne Warnungen akzeptiert!
- Die JSR-108 Implementation von JADE kennt überhaupt keine Klasse/Interface `Dimension`, sie arbeitet nur mit `Unit`, was hart an der Realität vorbeischrämmt. Je mehr ein Programm-Design die Anwendungs-Domain widerspiegelt, um so einfacher ist es zu benutzen bzw. zu warten.

Interface & enum-Klassen

Das Interface `Units` bildet einerseits eine Hülle für die Basis-Konstanten aus der Naturwissenschaft, andererseits schreibt es Methoden für die inneren `Unit`-Klassen vor, die diese zu implementieren haben. Damit nicht Units unterschiedlicher Dimensionen ineinander konvertiert werden können, wird das SubTyp-Idiom eingesetzt. Die Beziehung einer Konstanten vom Typ `Dimension` zu den inneren Enum-Klassen vom Typ `Unit` ist HAS-A. Zum Beispiel hat die Dimension Länge die Maßeinheiten `MM`, `CM`, `M`, `KM`. Diese Beziehung wird durch Übergabe der `class`-Datei `Unit.Length` im Konstruktor von `Dimension` manifestiert.

Generischer Konstruktor: Der Konstruktor in `Dimension` ist „generisch gezuckert“, was in diesem Fall unnötig war, da er von außen nicht benutzt werden kann. Einfach nur `Class` als Typ hätte auch gereicht. Aber `Class` ist nun einmal generisch und deshalb ist es gute Sitte des Klassen-Designers kundzutun, was hier für ein Class-Typ erwartet wird.

Tiger-Bug: Die Methode `getUnit()` stolpert über einen in Java 1.5 nicht behebbaren Bug.¹⁶ Der wirkliche Rückgabe-Typ ist gleichzeitig `Enum` und `Unit`. Aber wie soll man das dem Compiler klarmachen? Die o.a. Lösung zwingt den Klienten bei der Anwendung zu einem Cast, wenn man die `Enum`-Eigenschaften benötigt (siehe unten!). Ansonsten ist die Methode `getUnit()` straight-forward und nutzt Mittel der Reflexion, um die Methode `values()`, die der Compiler für alle `enum`-Klassen generiert, auszurufen.

---

¹⁶ Laut Sun-Mitteilung von Herrn Gafter vom 25.06.2004.

Zum Schluss der obligatorische Test mit geschachtelten foreach zum Ausdruck der Dimensionen mit zugehörigen Einheiten.

```
public class EnumTest2 {
    public static void main(String[] args) {

        // Alle Dimensionen mit allen Units
        for (Dimension d: Dimension.values()) {
            System.out.print("Dimension "+d+": ");
            for (Unit e: d.getUnit())
                System.out.print(e+" ");
            System.out.println();
        }
    }
}
```

```
Dimension LENGTH: MM CM M KM
Dimension MASS: MOLE MG G KG
Dimension TEMPERATURE: F C K
```

```
System.out.println("Umrechnung der Längeneinheiten:");
// nur obere Dreiecks-Matrix der Konvertierung
for (Unit<?> u1: Dimension.LENGTH.getUnit())
    for (Unit<?> u2: Dimension.LENGTH.getUnit())
        // diese Casts sind äußerst unschön!
        if (((Enum)u1).ordinal()<((Enum)u2).ordinal())
            System.out.println(u1+"->"u2+": "+
                // die auch!
                ((Unit.Length)u1).convert((Unit.Length)u2));
    }
}
```

```
Umrechnung der Längeneinheiten:
MM->CM: 10.0
MM->M: 1000.0
MM->KM: 1000000.0
CM->M: 100.0
CM->KM: 100000.0
M->KM: 1000.0
```

<? extends Enum & Unit>

Compiler-Fehler: Dies ist ein J2SE 5-Bug, der zur Zeit nicht behoben werden kann!

Wenn Ihnen das Alles suspekt vorgekommen ist, missachten sie es einfach und wenden sich wieder bodenständigen Problemen zu (wen interessiert schon das Loch auf dem Mars, in das sich der Orbiter verkrochen hat?).