E4 Spezielle Sprachelemente

E4.1 Aufzählungstypen

Mit der J2SE 5.0 wurden Aufzählungstypen (*Enumerations*) durch das Schlüsselwort enum als mächtiges Konzept in die Sprache Java aufgenommen. Aufzählungen bestehen aus einer festen, begrenzten, zur Compilezeit bekannten Anzahl von Konstanten. Typische Beispiele für Aufzählungen wären Wochentage, Monatsnamen, Himmelsrichtungen, Menüoptionen etc. Java-enums stellen eine echte Spracherweiterung dar; sie sind nicht nur eine elegantere, kürzere Schreibweise für bereits vorhandene Konstrukte, sondern eröffnen neue komplexe Programmiertechniken.

In C++ stehen Aufzählungen in einfacher Form ("glorified integers") schon lange zur Verfügung, in Java 5.0 wurden sie nun mit praktisch identischer Syntax, jedoch umfangreichem objektorientiertem Kontext und komplexen Verhaltensmöglichkeiten eingeführt. Java-Aufzählungen sind *besondere finale Klassen*, von denen sich keine seperaten Objekte und Unterklassen erzeugen lassen. Alle enum-Klassen sind implizit von der abstrakten, finalen Klasse `java.lang.Enum` abgeleitet.

Neben "gewöhnlichen" Elementen kann eine Klasse nun auch Aufzählungstypen enthalten, wie in folgendem Beispiel:

```
class Preise {
   private double stueckPreis;
   private enum Mengen{ Klein, Mittel, Gross };
   public Preise( double p ) { stueckPreis = p; }
   public double gesamtPreis( int zahl ) {
      Mengen m = Mengen.Klein;     double preis = 0.0;
      if( zahl > 100 )  m = Mengen.Mittel;
      if( zahl > 1000 ) m = Mengen.Gross;
      switch( m ){
         case Klein :  preis = 1.0*zahl*stueckPreis;   break;
         case Mittel : preis = 0.9*zahl*stueckPreis;   break;
         case Gross :  preis = 0.8*zahl*stueckPreis;   break;
      }
      return preis;
   }
   public static void main( String[] args ){
      Preise tester = new Preise( 100.0 );
```

```
        IO.writeln( "Preis = " + tester.gesamtPreis(780) );
    }
}
```

Die Deklaration und (mittels *Komma*) aufzählende Initialisierung innerhalb {} wird mit Semikolon abgeschlossen:

```
enum Mengen{ Klein, Mittel, Gross };  // Komma beim Aufzählen!
```

Intern wird eine *Klasse* Mengen angelegt, die als statische Elemente die konstanten Objekte Klein, Mittel, Gross enthält. Im Gegensatz zu C++ stehen diese *nicht* einfach für die entsprechende benamte Integer-Aufzählung 0,1,2, sondern repräentieren konstante Objekte ihres eigenen Enum-Typs (hier: Mengen), der nicht zu anderen (einfachen und komplexen) Datentypen gecastet werden kann. Jeder Enumeration stellt einen *eigenen Namensraum* dar, so dass die gleichen Bezeichner in verschiedenen enums der gleichen Klassen verwendet werden können.

In diesem Sinne sind *enums typsicher*, da sie nicht mit int-Werten oder anderen Objekttypen verwechselt werden können. Durch: Mengen m; kann eine Objektvariable vom Typ der Enumeration Mengen deklariert werden, der nur Objekte vom Typ Mengen zugewiesen werden dürfen – dies sind gerade die in der Aufzählung Mengen aufbewahrten Objekte Klein, Mittel, Gross vom Typ der Klasse Mengen. Der Compiler prüft auf Typkompatibilität.

```
Mengen m = Mengen.Klein;  // Ok - typkompatibel
// m = 2;  Fehler - nicht typkompatibel!
```

Das Codebeispiel zeigt, dass enum-Typen in switch-Strukturen verwendet werden können: In case wird das jeweilige enum-Objekt (*ohne* vorangestelltem enum-Klassennamen) genannt. Der Inhalt der Objektvariablen m wird mit den Objekten der enum-Klasse Mengen verglichen, um entsprechend zu verzweigen.

Für enum-Objekte sind auch die *Vergleichsoperationen* == und != definiert, *nicht* jedoch die Operationen <, >, <=, >= :

```
if( m == Mengen.Gross ) IO.writeln( "Grosses Gebinde" );
```

Über den Inhalt einer Enumeration kann in einer Schleife interiert werden; dazu werden innerhalb konventioneller Schleifen Arrays verwendet. Mit Java 5.0 steht zudem eine weitere, syntaktisch neue Form der for-Schleife zur Verfügung.

Der *Inhalt* einer Enumeration kann auf der Klasse selbst und auf ihren Objekten *ausgegeben* werden - für enum wurde intern die Methode toString() entsprechend implementiert. Dabei werden die Namen der gespeicherten Objekte sichtbar (was informativer ist als die Ausgabe einer blossen Zahl wie bei C++):

```
Mengen m = Mengen.Klein;    IO.writeln( "Inhalt:" + m );
IO.writeln( "Inhalt: " + Mengen.Mittel + "  " + Mengen.Gross );
```

- Konsole -
Inhalt: Klein Inhalt: Mittel Gross

Jeder enum-Aufzählungstyp innerhalb einer Klasse stellt selbst eine Klasse mit statischen Methoden dar. U.a. steht die statische Methode `values()` zur Verfügung, die ein Array mit den Konstanten der Aufzählung zurückliefert. In folgendem Beispiel wird dieses Array erzeugt und durchlaufen:

```
class EnumArray {
    public static enum Richtung { Nord, Sued, Ost, West };
    public static void main( String[] args ) {
       Richtung[] aR = Richtung.values();
       for( int i=0; i<aR.length; i++ ) {
          IO.write( "*" + aR[i] );
          IO.write( "*" + Richtung.values()[i] );
       }
    }
}
```

- Konsole -

```
*Nord*Nord*Sued*Sued*Ost*Ost*West*West
```

Aufzählungstypen können *optional* im *Körper* der enum-Klasse sogar mit *weiteren Methoden, Konstruktoren und Attributen* versehen werden, da es sich um Klassen handelt. Konstruktoren und Attribute müssen `private` sein, nur Methoden dürfen auch `public` sein. Auf diese Weise lassen sich via Konstruktoraufruf in den internen enum-Objekten Werte speichern und durch Methoden abfragen. In runden Klammern werden den Objekten die Argumente des Konstruktoraufrufs mitgegeben. Ferner kann auch die Methode `toString()` spezifisch überschrieben werden. In folgendem Beispiel wird die Aufzählung `Mengen` entsprechend erweitert:

```
class Preise {
    private double stueckPreis;
    private enum Mengen{
       // Konstruktoraufrufe mit Wertangaben:
       Klein( 0 ),  Mittel( 100 ),  Gross( 1000 );
       private Mengen( int w ) { grenzWert = w; }   // Konstruktor
       private final int grenzWert;           // Internes Attribut
       public int wert() { return grenzWert; }      // Wertabfrage
       public String toString() { return "Grenze: " + grenzWert; }
    };
    public Preise( double p ) { stueckPreis = p; }
    public double gesamtPreis( int zahl ) {
       Mengen m = Mengen.Klein;     double preis = 0.0;
       if( zahl > Mengen.Mittel.wert() ) m = Mengen.Mittel;
       if( zahl > Mengen.Gross.wert()  ) m = Mengen.Gross;
```

```
        switch( m ){
           case Klein :  preis = 1.0*zahl*stueckPreis;   break;
           case Mittel : preis = 0.9*zahl*stueckPreis;   break;
           case Gross :  preis = 0.8*zahl*stueckPreis;   break;
        }
        return preis;
     }
     public static void main( String[] args ){
        Preise tester = new Preise( 100.0 );
        IO.writeln( "Preis = " + tester.gesamtPreis(780) );
     }
  }
```

Die hinzugefügten Methoden können auf den konstanten enum-Objekten aufgerufen werden, wie im Beispiel die Methode `wert()`. Durch sehr spezielle Konstrukte kann sogar erreicht werden, dass dieselbe Methode sich für die einzelnen enum-Objekte *verschieden* verhält (constant-specific methods). Ferner können enums sogar Interfaces implementieren. Doch gehen wir auf diese fortgeschrittenen enum-Techniken nicht weiter ein, ebensowenig wie auf die neuen speziellen enum-Datencontainer `java.util.EnumSet` und `java.util.EnumMap` des Collection-Frameworks. Zur Vertiefung verweisen wir auf [ESS05] und [SUN05a].

Der Aufzählungstyp enum erspart das Programmieren von *Hilfskonstruktionen*. Dazu gehören methodenlose Klassen, die nur entsprechende Werte als statische Integer-Konstanten enthalten.

```
class Mengen {
   public static final int Klein  = 0;
   public static final int Mittel = 1;
   public static final int Gross  = 2;
}
```

Auf diese Klasse kann aus anderen Klassen direkt zugegriffen werden, um mit ihren `int`-Konstanten `Mengen.Klein` etc. zu arbeiten. Allerdings ist die Klasse `Mengen` *nicht typsicher*, da ihre Konstanten mit gewöhnlichen Integer-Werten *verwechselt* werden können. So würde der Compiler den Aufruf einer Methode:

```
double preis( int zahl, int mengenArt ) { /* ... */ }
```

sowohl in der Form: `preis(100,Mengen.Mittel)` als auch in der sicherlich fehlerhaften Form: `preis(Mengen.Mittel,preis)` akzeptieren. Dagegen kann mit dem obigen enum-Typ `Mengen` *typsicher programmiert* werden. Die Methode hätte nun die Gestalt:

```
double preis( int zahl, Mengen mengenArt ) { /* ... */ }
```

Der Aufruf: `preis(100,Mengen.Mittel)` wäre korrekt, dagegen würde der Compiler den Aufruf: `preis(Mengen.Mittel,100)` natürlich zurückweisen.

Die trick- und lehrreiche Nachahmung einer vollwertigen Aufzählungstyp-Klasse würde mit privatem Konstruktor und öffentlichen Attributen arbeiten, die statische finale Objekte der Klasse selbst sind:

```
class MengenTyp {
   private int mengenArt;
   private MengenTyp( int n ) { mengenArt = n; }
   public static final MengenTyp Klein  = new MengenTyp( 0 );
   public static final MengenTyp Mittel = new MengenTyp( 1 );
   public static final MengenTyp Klein  = new MengenTyp( 2 );
   public int getArt( return mengenArt );
}
```

Der private Konstruktor verhindert das Erzeugen anderer als der *intern* angelegten statischen Objekte. Auf diese kann über die Klasse selbst zugegriffen werden. Der charakteristische Wert `mengenArt` kann über die Methode `getArt()` nur abgefragt, nicht verändert werden. Die Klasse ist typsicher, d.h. die oben genannten Zweideutigkeiten (Verwechselung mit `int`-Werten) bei Methodenaufrufen können nicht auftreten: Eine Methode mit Parameter vom Typ `MengenTyp` kann nur mit den Objekten `MengenTyp.Klein`, `MengenTyp.Mittel` oder `MengenTyp.Gross` aufgerufen werden, nicht mit anderen Objekten oder `int`-Werten. Durch die Methode `int getArt()` könnte die Klasse sogar bei `switch()`-Verzeigungen und Vergleichsoperationen benutzt werden.

Dank der in Java 5.0 eingebauten typsicheren Aufzählungen, steht jedoch all dies als enum-Typ gekapselt *direkt* zur Verfügung.

E4.2 Varargs (Methoden mit variablen Argumenten)

Mit der J2SE 5.0 ("Tiger"-Release) wurde die syntaktische Möglichkeit geschaffen, bei der Deklaration von Methoden die *Parameterzahl als variabel* zu kennzeichnen, so dass die *Methode mit beliebig vielen (typkompatiblen) Parametern aufrufbar* ist. Die übergebenen Parameter (primitive Datentypen oder Objektreferenzen) stehen innerhalb der Methode in einem Array zur Verfügung – somit wird die entsprechende direkte Array-Codierung dem Entwickler abgenommen. Im Methodenkopf ist der übergebene (prinitive oder Referenz-) Datentyp gefolgt von **`...`** und dem Namen des Arrays anzugeben. Folgendes Coding wird somit möglich:

```
class VarArgs {
   public static void varArgs( int ... argumente ) {
      for( int i=0; i<argumente.length; i++ ){
         IO.writeln( "Wert: " + argumente[i] );
      }
   }
   public static void main( String[] args ) {
      varArgs( 10, 20, 30, 40 );  // Parameter mit Komma
   }
}
```

Die Methode `varArgs()` kann mit *null bis beliebig vielen* durch Komma getrennten Argumenten erfolgen. Natürlich ist beim Aufruf *Typkompatibilität* gefordert: Im Beispiel wurde die Methode mit explizitem Parametertyp `int` deklariert, so dass der Aufruf `varMethode(10.0,20.0)` mit `double`-Werten zurückgewiesen werden würde.

Durch den Compiler wird *jede* Varargs-Methode in eine Methode mit Parameterarray übersetzt, so dass die folgenden Methodendeklarationen vom Compiler als *absolut identisch* betrachtet werden und einen Namenskonflikt darstellen:

```
// Kein zulässiges Überladen:
public static void varArgs( int ... argumente ) { /* … */ }
public static void varArgs( int[] argumente ) { /* … */ }
```

Noch flexibler, jedoch *typmäßig völlig unbestimmt* ist die Variante mit Parametertyp Object (Oberklasse aller Java-Klassen, s. Kapitel 13):

```
public static void varArgs( Object ... argumente ) { /* … */ }
```

Dieser Methode können beliebig viele Werte *beliebiger (primitiver und Referenz-) Datentypen* übergeben werden. (Primitive Datentypen wandelt der Compiler in Hüllklassenobjekte um.) Der Aufruf:

```
varArgs( 10, "Hallo", true, 25.6, new Mitarbeiter("Meier") );
```

ist somit möglich (ebenso wie der Aufruf ohne Argumente!). Auf diese Weise wird in Java 5.0 auch die jedem C-Programmierer wohlvertraute Ausgabemethode `System.out.printf()` mit zahlreichen Formatierungsmöglichkeiten realisiert. Variable Argumente können in Kombination mit konventionellen Parametern *nur an der letzten Parameterposition* verwendet werden:

```
void varMult( double d, Object ... argumente ) { /*...*/ }  // OK
```

Von Methoden mit variablen Argumenten machen wir im Rahmen dieses Buches keinen weiteren Gebrauch, da ihre Syntax den Programmiereinsteiger eher verwirrt und gutem Programmierstil nicht förderlich ist. Auch die Firma Sun empfiehlt den eher sparsamen Einsatz dieses neuen Sprachfeatures.

E4.3 Object

public Object() Konstruktor.
protected Object clone() Erzeugt und liefert eine flache Kopie des aufrufenden Objekts (s. Kapitel 11).
public boolean equals(Object obj) Gibt an, ob übergebenes Objekt "identisch" ist mit aufrufendem Objekt. Die `Object`-Implemetierung vergleicht die Referenzen. Nur wenn diese auf das gleiche Objekt zeigen wird `true` zurückgeliefert. Wird diese Methode überschrieben, muss auch `hashcode()` angepasst werden: Objekte, die gemäss `equals()` "gleich" sind, müssen den gleichen Hashcode liefern. (Vertrag zwischen `equals()` und `hashCode()`)
protected void finalize() Wird aufgerufen wenn keine Referenzen mehr auf das unzugreifbare Objekt zeigen, d.h. es zum Löschen durch den Garbage Collector ansteht. Wird überschrieben, um Systemressourcen freizugeben oder Aufräumarbeiten zu erledigen.
public final Class <? extends Object> getClass() Liefert Klassendeskriptor für aufrufendes Objekt, über den Klasseninformationen (Metainformationen) abrufbar sind.
public int hashCode() Liefert dezimalen Hashwert für aufrufendes Objekt. Der Wert ist innerhalb der JVM eindeutig während Ausführung einer Anwendung, kann aber von einer Ausführung zur anderen variieren. Zwei Objekte, die gemäss `equals()` identisch sind, sollten den gleichen Hashcode liefern!
public String toString() Liefert String-Repräsentation des aufrufenden Objekts, das dadurch textuell in lesbarer Form abgebildet wird. Die `Object`-Methode liefert einen String, der sich aus Klassennamen + "@" + hexadezimalem `hashCode()`-Wert zusammensetzt. Unterklassen sollten diese Methode überschreiben.
public final void notify() public final void notifyAll() public final void wait() **public final void wait(long timeout) public final void wait(long timeout, int nanos)** Synchronisation paralleler Ausführungs-Threads (s. Kapitel ...)

Tab. 1: Methoden der Wurzelklasse Object

Die Systemklasse `Object` (aus `java.lang`) ist Basisklasse der gesamten Java-Klassenhierachie ("Mutter aller Klassen"). Die *Methoden von* `Object` sind in Tabelle 1 aufgeführt und kommentiert Für Details verweisen wir auf [SUN05c]. Im Kapitel 14 wird erläutert, dass `Object` als Basistyp die Definition generischer Strukturen mittels Polymorphie erlaubt.

Dass jede Java-Klasse von `Object` erbt, zeigt folgende Demonstration:

```
class TutNix {       // Aufruf geerbter Object-Methoden:
   public static void main( String[] args ) {
      TutNix tn = new TutNix();
      IO.writeln("Hash: " + tn.hashCode() + " " + tn.toString());
   }
}
```

- Konsole -

```
Hash: 1122161 TutNix@111f71
```