Beispiel: Wrapper-Klassen für primitive Typen

- Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende (sog. *Wrapper-*) Klasse, die den primitiven Typ in einer oo Hülle kapselt.
- Es gibt Situationen, bei denen man diese Wrapper-Klassen anstelle der primitiven Typen benötigt. Z.B. werden in Java einige Klassen zur Verfügung gestellt, die eine (dynamische) Menge von beliebigen Objekttypen speichern können. Um darin auch primitive Typen ablegen zu können, benötigt man die Wrapper-Klassen.
- Zu allen numerischen Typen und zu den Typen `char` und `boolean` existieren Wrapper-Klassen.

Beispiel: Wrapper-Klassen für primitive Typen

Wrapper-Klasse	Primitiver Typ
`Byte`	**`byte`**
`Short`	**`short`**
`Integer`	**`int`**
`Long`	**`long`**
`Double`	**`double`**
`Float`	**`float`**
`Boolean`	**`boolean`**
`Character`	**`char`**
`Void`	**`void`**

Beispiel: Wrapper-Klassen für primitive Typen

- Zur Objekterzeugung stellen die Wrapper-Klassen hauptsächlich zwei Konstruktoren zur Verfügung:
  - Für jeden primitiven Typ `type` stellt die Wrapperklasse `Type` einen Konstruktor zur Verfügung, der einen primitiven Wert des Typs `type` als Argument fordert:
    z.B. `public Integer(int i)`
  - Zusätzlich gibt es bei den meisten Wrapper-Klassen die Möglichkeit, einen String zu übergeben:
    z.B. `public Integer(String s)` wandelt die Zeichenkette `s` in einen Integer um, z.B. `"123"` in den `int`-Wert `123`.
  - Beispiele:

```
Integer i1 = new Integer(123);
Integer i2 = new Integer("123");
```

Beispiel: Wrapper-Klassen für primitive Typen

- Kapselung:
  - Der Zugriff auf den Wert des Objekts erfolgt ausschließlich lesend über entsprechende Methoden:
    z.B. `public int intValue()`
  - Die interne Realisierung (wie wird der Wert gespeichert) ist dem Benutzer verborgen.
  - Insbesondere kann der Wert des Objekts nicht verändert werden.
  - Beispiele:

```
int i3 = i1.intValue();
```

Beispiel: Wrapper-Klassen für primitive Typen

- Statische Elemente (u.a.):
  - Wichtige Literale aus dem enstrechenden Wertebereich, z.B. Konstanten:
    `public static int MAX_VALUE` bzw.
    `public static int MIN_VALUE`
    für den maximal/minimal darstellbaren **`int`**-Wert

    oder z.B. Konstanten:
    `public static double NEGATIVE_INFINITY` bzw.
    `public static double POSITIVE_INFINITY`
    für $-\infty$ und $+\infty$
  - Hilfsmethoden wie z.B.
    `static double parseDouble(String s)` der Klasse `Double` wandelt die Zeichenkette `s` in ein primitiven **`double`**-Wert um und gibt den **`double`**-Wert aus.

Beispiel: Wrapper-Klassen für primitive Typen

- Mit Wrapper-Klassen lassen sich i.Ü nun auch Eingaben von der Kommandozeile in entsprechende primitive Typen verwandeln.
- Die Eingaben werden ja automatisch in das `String`-Array geschrieben, das als Eingabe-Parameter der `main`-Methode dient.
- Beim Wechselgeld benötigen wir einen **`int`**-Wert für den Rechnungsbetrag:

```
public class Wechselgeld {
  ... // siehe Teil 5, Folie 120 ff.
  public static void main(String[] args) {
     Integer rI = new Integer(String[0]);
     int r = iR.intValue();

     WG wg = wechselgeld(r)
     ...
  }
}
```

- Der Aufruf `java Wechselgeld 23` berechnet dann das Wechselgeld für den Rechnungsbetrag 23.

Kapitel 4: Aufzählungstypen (Enumerations) i

Motivation

- Klassen stellen ein mächtiges Werkzeug dar, eigene, komplex strukturierte Datentypen zu modellieren und bereit zu stellen.
- Die „Literale“ dieser Datentypen sind die entsprechend strukturierten Objekte.
- In der Praxis hat man es aber auch häufig mit Datentypen zu tun, deren Werte (Literale) aus einem konstanten Wertevorrat stammen (so wie eigentlich auch die primitiven Datentypen), z.B.:
  - Die Menge der Jahreszeiten:
    $\{$FRÜHLING, SOMMER, HERBST, WINTER$\}$
  - Die Menge der Monate:
    $\{$JANUAR, FEBRUAR, . . . , DEZEMBER$\}$
  - Die Menge der Farben:
    $\{$ROT, GRÜN, BLAU, . . . $\}$

Motivation

- Rein vom Konzept her ist das wie gesagt nichts Neues:
- In unseren Modulen, in denen wir die Grunddatentypen und deren Operationen definiert haben, haben wir die Literale ja auch als konstante Funktionen spezifiziert.
- Das ist letztlich auch die Idee, wie man diese Wertemengen definiert.
- Allerdings geht es leider nicht so, wie wir das bisher gemacht haben, nämlich mit einer 0-stelligen Funktion, z.B.

  ```
  static Monat JANUAR() { return new Monat(); }
  ```

  in der Klasse `Monat`.
- Warum nicht?

Motivation

- Die einfachste Möglichkeit solche Wertemengen in einer Programmiersprache zu vereinbaren, wäre, die einzelnen Werte als (globale) Konstanten eines primitiven Typs (typischerweise **`int`**, da es sich um so etwas wie IDs handelt) zu vereinbaren.
- Also in Java z.B. für Monat:

```
static final int JANUAR = 1;
static final int FEBRUAR = 2;
static final int MAERZ = 3;
static final int APRIL = 4;
...
```

Motivation

- In einigen Programmiersprachen muss man dies tatsächlich so machen und es ist zunächst auch nicht verkehrt (in Java bis Version 5.0 z.B.).
- Einziges Problem ist, dass man keine *Typsicherheit* mehr hat: Statt einen eigenen Typ „Monat“ zu haben, arbeitet man mit Werten vom Typ `int`.
- Dadurch kann zur Kompilierzeit nicht sicher gestellt werden, dass man z.B. an einen Methodenparameter einen der zulässigen Aufzählungswerte übergibt.
- Was passiert z.B., wenn man an eine Methode, die einen Wert zwischen 1 und 12 erwartet, den Wert 100 übergibt? Naja, hängt natürlich vom Rumpf der Methode ab ... Wahrscheinlich kommt (ohne dass man es zunächst merkt) ein Blödsinn dabei raus, oder es passiert ein Laufzeitfehler (was ist schlimmer?)

Motivation

- Hier ein konkretes Beispiel:
- Eine Methode `anZahlTage(int monat, boolean schaltJahr)` nimmt eben nun leider als Eingabe für Monat einen Wert vom Typ **`int`**.
- Würde diese Methode mit `anZahlTage(30, false)` aufgerufen werden, wäre das für den Compiler OK.
- Den potentiellen Fehler würde man dann erst zur Laufzeit feststellen (oder eben auch nicht . . .).

Aufzählungstypen

- Für diese Werte-Mengen gibt es in einigen Programmiersprachen (wie gesagt, in Java seit Vers. 5.0) sog. *Aufzählungstypen* (*Enumeration*), die es erlauben, bei der Definition solcher Datentypen die Literale (Werte) explizit festzulegen (aufzuzählen).
- In Java ist dafür ein neues Schlüsselwort verfügbar: **`enum`**.
- Die einfachste Verwendung ist:

  ```
  enum Monat ( JANUAR, FEBRUAR, MAERZ, APRIL, ... )
  ```

- Dies macht einen neuen Typ `Monat` dem Compiler bekannt, den man nun als Typ für (lokale/globale) Variablen, Eingabeparameter von Methoden und Instanzvariablen (Attribute) verwenden kann.
- Dieser Typ kann nur die aufgelisteten Werte annehmen.

Aufzählungstypen

- Intern sind die Literale zwar als **`int`**-Werte codiert, ein Aufzählungs-Typ in Java ist aber als Klasse realisiert, d.h. die einzelnen Literale sind Objekte dieser Klasse.
- Nützliche Eigenschaften von Aufzählungstypen, die standardmäßig durch Java zur Verfügung gestellt werden:
  - Methode `toString`, die den Namen des Literals ausgibt.
    Beispiel: `Monat.JANUAR.toString()` ergibt `"JANUAR"`.
  - Methode `equals` prüft auf Gleichheit.
    Beispiel: `JANUAR.equals(APRIL)` ergibt **`false`**.
  - Variablen von Aufzählungstypen können in **`switch`**-Anweisungen verwendet werden (da es letztlich **`int`**s sind).
  - Methode `values` mit der man die einzelnen Werte der Enumeration durchlaufen kann (mittels eines `Iterator`-Objekts).
  - ...

Aufzählungstypen

- Letztlich steckt dahinter nicht viel Neues, außer, dass die Objekte von der Klasse `Monat` alle intern eine ID besitzen, die aus der Werte-Menge stammt (aber die Objekte können jetzt mit dem in der Aufzählung vereinbarten Namen angesprochen werden, die internen IDs sind gekapselt).
- Enumerations können in Java tatsächlich fast genauso wie Klassen um Attribute, Methoden und Konstruktoren ergänzt werden.
- Ein Blick in ein einschlägiges Lehrbuch zu diesem Thema lohnt sich!
- Zur Veranschaulichung, was man alles u.a. machen kann, noch ein Beispiel:

Aufzählungstypen: Beispiel

```
public enum Farbe
{
  /* ** Die Literal-Namen mit Initialisierung der Attribute durch den Konstruktor ** */
  ROT(255, 0, 0),  // die Parameter fuer den Konstruktor (unten) muessen hier angegeben werden.
  GRUEN(0, 255, 0),
  BLAU(0, 0, 255),
  ... // weitere Farb-Literale
  GELB(255, 255, 0);

  /* ** Attribute, die die RGB-Werte speichern (Konstanten, da nicht veraenderbar) ** */
  private final int r;
  private final int g;
  private final int b;

  /* ** Konstruktor ** */
  public Farbe(int r, int g, int b) {
    this.r = r;
    this.g = g;
    this.b = b;
  }

  /* ** Methoden ** */

  public int getRotAnteil() { return this.r }
  ...
}
```