15. Enumerationen

Prinzip

Gesucht: **Typ mit (kleiner) fester Anzahl von Werten**

z.B.: Farben: rot, blau, grün, gelb
Wochentage: Mo, Di, Mi, Do, Fr, Sa, So
Prioritäten: niedrig, normal, hoch

Mögliche Implementierung als *int*-Konstanten

```
// colors
static final int RED = 0;
static final int BLUE = 1;
static final int GREEN = 2;
```

```
int color = BLUE;
```

```
// priorities
static final int LOW = 0;
static final int NORMAL = 1;
static final int HIGH = 2;
```

```
int priority = HIGH;
```

Problem: Keine Typprüfung zwischen Konstantenmengen

```
int color = HIGH;
```
← Compiler meldet keinen Fehler

=> Java bietet spezielle Enumerationstypen mit Typprüfung

15. Enumerationen

Enumerationstypen

Deklaration durch Aufzählung der Werte (daher auch *Aufzählungstypen*)

```
enum Color {RED, BLUE, GREEN, YELLOW}
enum Day {MON, TUE, WED, THU, FRI, SAT, SON}
enum Weekend {SAT, SON}
```

- Werte sind Konstanten
- müssen innerhalb eines Typ unterschiedliche Namen haben
- werden mit Großbuchstaben geschrieben

Variablen und Zuweisungen

`Color c = Color.BLUE;`	Werte müssen mit Typnamen qualifiziert werden
~~`Color c = Day.WED;`~~	*Color* und *Day* sind nicht kompatibel
~~`Day d = Weekend.SAT;`~~	*Day* und *Weekend* sind nicht kompatibel
~~`Color c = 0;`~~	*Color* und *int* sind nicht kompatibel

Vergleiche

```
if (c == Color.BLUE) ...
if (c != Color.RED) ...
```

Vergleiche mit == und != erlaubt
(Operanden müssen vom selben Enumerationstyp sein)

Werte sind in Deklarationsreihenfolge geordnet

```
Color c1 = Color.RED;
Color c2 = Color.BLUE;
if (c1.compareTo(c2) < 0) ...
```

liefert *true* (*c1 < c2*)

Switch auf Enumerationswerte


```
Color c;
...
switch (c) {
   case Color.RED:
      ...;
      break;
   case Color.BLUE:
      ...;
      break;
   case Color.GREEN:
      ...;
      break;
   case Color.YELLOW:
      ...;
      break;
}
```

Wie in allen Switch-Anweisungen gilt:

- Case-Marken müssen voneinander verschieden sein
- Case-Marken müssen mit Switch-Ausdruck kompatibel sein
- Case-Blöcke müssen mit *break*, *return* oder *throw* enden
- *default*-Zweig möglich

Konversionen

Enum Û String

```
Color c = Color.BLUE;
String name = c.toString();
Out.print(c);
Color c2 = Color.valueOf(name);
```

liefert "BLUE"
liefert "BLUE" (*toString()* wird automatisch angewendet)
c2 == Color.BLUE

Enum Þ int

```
Color c = Color.BLUE;
int val = c.ordinal();
```

liefert 1 (Color.RED == 0, Color.BLUE == 1, Color.GREEN = 2, ...)

Enum Þ Werte-Array

```
Color[] a = Color.values();
```

a[0] == Color.RED, a[1] == Color.BLUE, ...

```
for (Color c: Color.values()) {
   Out.println(c + " = " + c.ordinal();
}
```

Color.RED = 0
Color.BLUE = 1
Color.GREEN = 2
Color.YELLOW = 3

15. Enumerationen

Enumerationstypen als Klassen

Enumerationstypen werden wie Klassen behandelt

```
enum Color {RED, BLUE, GREEN}
```

wird in folgende Klasse übersetzt:

```
class Color {
   static final Color RED   = new Color();
   static final Color BLUE  = new Color();
   static final Color GREEN = new Color();
}
```

Zugriff auf

```
... Color.RED ...
```

liefert statisches Feld RED

Enumerationstypen mit Feldern und Methoden

Enumerationstyp für römische Ziffern

```
enum Roman {
   I(1), V(5), X(10), L(50), C(100), D(500), M(1000);

   private int value;

   Roman(int val) { value = val; }
   int getValue() { return value; }
}
```

Benutzung

```
Roman r = Roman.V;
Out.print(r.getValue());
Out.print(r.ordinal());
```

liefert 5
liefert 1

wird in folgende Klasse übersetzt:

```
class Roman {
   static final Roman I  = new Roman(1);
   static final Roman V = new Roman(5);
   static final Roman X = new Roman(10);
   ...

   private int value;

   Roman(int val) { value = val; }
   int getValue() { return value; }
}
```

Konstruktor darf von außen nicht verwendet werden

Basisklasse Enum

Gemeinsame Basisklasse aller Enumerationstypen

```
class Enum implements Comparable {
   String    toString() {...}
   boolean   equals(Object obj) {...}
   int       compareTo(Object obj) {...}
   int       ordinal() {...}
   ...
}
```

Alle Enumerationstypen erben diese Funktionalität

```
enum Color {RED, BLUE, GREEN}
```

```
class Color extends Enum { ... }
```

Zusätzlich erzeugt der Compiler für jeden Enumerationstyp 2 Methoden

```
class Color extends Enum {
   ...
   Color    valueOf(String name) {...}
   Color[]  values() {...}
}
```