II.2. Objekte, Klassen und Methoden

- 1. Grundzüge der Objektorientierung
- 2. Methoden, Unterprogramme und Parameter
- 3. Datenabstraktion
- 4. Konstruktoren
- **5. Vordefinierte Klassen**

Hüllklassen

- **Primitive Typen (`boolean, char, int, double, ...`) passen nicht ins Konzept von Klassen und Objekten.**
- **Nachteil:**
  - unsystematisch
  - keine Referenzparameter für Objekte primitiver Typen
  - manche Methoden verlangen Klassentypen als Parameter
- **Daher existieren für alle primitive Datentypen sogenannte Hüllklassen:**
  - `Boolean`
  - `Character`
  - `Byte, Short`
  - `Integer, Long`
  - `Float`
  - `Double`

Attribute und Methoden von `Integer`

- **Objekt-Attribut (nicht public): der eingehüllte `int`-Wert**

- **Klassen-Attribute (statisch):**
  - MIN_VALUE : kleinster Wert vom Typ `int (-2.147.483.648)`
  - MAX_VALUE : größter Wert vom Typ `int (2.147.483.647)`

- **Statische Methoden:**
  - `static Integer valueOf (int i)`
  - `static Integer valueOf (String s)`
  - `static int parseInt (String s)`
  - `static String toString (int i)`

- **Methoden:**
  - `String toString ()`
  - `boolean equals (Integer i)`
  - `byte byteValue() , int intValue (), float floatValue (), ...`

Beispiel zur Verwendung von `Integer`

```
Integer x = Integer.valueOf(321);
Integer y = Integer.valueOf("321");
int z = Integer.parseInt("321");

String s1 = Integer.toString (321);
String s2 = x.toString ();

IO.println("x: " + x);
IO.println("y: " + y);
IO.println("z: " + z);
IO.println("s1: " + s1);
IO.println("s2: " + s2);

IO.println ("x == y: " + (x == y));
IO.println ("x.equals(y) : " + x.equals(y));

IO.println ("x.intValue () == z: " +
            (x.intValue () == z));
```

321 ⟵ Integer
321 ⟵
321 ⟵ int

"321" ⟵ String
"321" ⟵

false

true

true

Autoboxing und Unboxing

Automatische Konvertierung zwischen primitivem Datentyp + Hüllklassentyp

```
Integer x = 321;
```

```
Integer x = Integer.valueOf(321);
```
Autoboxing

```
int i = x;
```

```
int i = x.intValue();
```
Unboxing

```
Integer y = x + 2;
```

```
Integer y = Integer.valueOf(x.intValue() + 2);
```
Autoboxing Unboxing

erwartet ein double-Argument

```
Double z = Math.sqrt(y);
```

```
Double z = Double.valueOf(Math.sqrt(y.intValue()));
```
Autoboxing Unboxing

```
Double d = 4;
```

```
Double d = Integer.valueOf(4);
```
Typfehler!

erwartet double-Argument

```
Double d = Double.valueOf(4);
```

Autoboxing und Unboxing nur zwischen prim. Typ und entsprechendem Hüllklassentyp. Keine Typkonversion von Integer und Double

```
Double e = 4.0;
```

```
Double e = Double.valueOf(4.0);
```

Autoboxing und überladene Methoden

```
static int f (int i)          {return 1;}
static int f (Integer x)      {return 2;}
static int f (double d)       {return 3;}
static int f (int... a)       {return 4;}
static int f (Integer... b)   {return 5;}
```

vararg Methoden möglichst nicht überladen!

```
f (Integer.valueOf(1)) = 2
f (1)                  = 4
```

Fehler!

- Autoboxing und Unboxing bei überladenen Methoden wird erst dann gemacht, wenn keine andere Methode passt.
- vararg-Methode wird erst dann ausgeführt wenn auch mit Auto-/Unboxing keine andere Methode passt.

Beispiel zur Verwendung von `String`

```
String s = "Wort";
String t = "Wort";
String u = new String("Wort");
String v = new String("Wort");

IO.println ("s == t: " + (s == t));
IO.println ("s == u: " + (s == u));
IO.println ("s.equals(u): " + s.equals(u));
IO.println ("u == v: " + (u == v));
IO.println ("u.equals(v): " + u.equals(v));

IO.println ("Zeichen in " + u + " an Index 2: " + u.charAt(2));

IO.println ("Laenge von " + u + ": " + u.length());

IO.println ("Zeichen in " + u + " an Index 2: " + u.toCharArray() [2]);
```