Java: Aufzähltypen mit enum, Generics und mehr

Ziele

- Einige nützliche Spracherweiterungen der letzten Java-Versionen kennen und anwenden lernen.
- Statische Imports als Möglichkeit zur Codevereinfachung kennen lernen.
- Strings und enum in switch verwenden können.
- Den Zweck variabel langer Parameterlisten verstehen und diese anwenden können.
- Vorteile des Autoboxing und -unboxing anwenden können.
- enum richtig verwenden können.
- Mit Generics umgehen können.
- Einfache typisierte Klassen erstellen können.

Statische Imports

Bis jetzt

```
double r = Math.cos(Math.PI);
```

Und nun

```
import static java.lang.Math.*;
double r = cos(PI);
```

Strings und enum in switch möglich

```
String s = "Hallo";
switch (s) {
  case "hallo": {
    System.out.println("Klein");
    break;
  }
  case "Hallo": {
    System.out.println("Gross");
    break;
  }
}
```

Groß-/Kleinschreibung wird berücksichtigt

enum siehe hinten

Variabel lange Parameterlisten (vom selben Typ)

```
public static void dupliziere(int anzahl, String... args) {
  while (anzahl-- > 0) {
    for (String s: args)
      System.out.print(s + " ");
    System.out.println();
  }
}
```

- Nur letzter Parameter darf variabel lang sein.
- Technisch gesehen realisiert Java den Parameter als Array.

ZWECK: Definition flexibler Ausgabemethoden wie `printf` in C/C++.

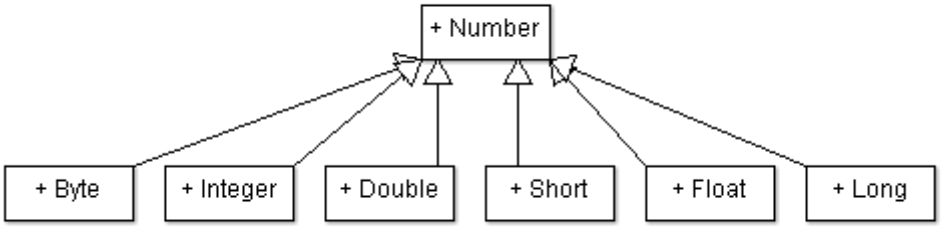
```
dupliziere(2, new String[]{ "das", "war", "was" });
```

Oder

```
dupliziere(2, "das", "war", "was");
```

Autoboxing und -unboxing

Automatisches Ein- und Auspacken von primitiven Typen in und aus Wrapper-Klasse.



Bis jetzt

```
Integer i = new Integer(3);
for (int j = 0; j < i.intValue();j++)
  System.out.println(j+i.intValue());
System.out.println(i.toString());
```

Und nun[1]

```
Integer i = 3;
for (int j = 0; j < i;j++)
  System.out.println(j+i);
System.out.println(i);
```

[1] Der Vergleich auf Gleichheit von `int`/`Integer` und `int`/`Integer` sollte immer über `equals()` erfolgen

Aufzähltypen mit enum

Häufig benötigt man Datentypen mit Werten aus kleinem Wertevorrat:

- TRUE, FALSE
- FRUEHLING, SOMMER, HERBST, WINTER
- KREDITKARTE, BANKEINZUG, NACHNAHME, VORKASSE, RATENZAHLUNG

```
package net.gobbz.enumpackage;

public class Schlussverkauf
{
  public enum Jahreszeit
  {
    FRUEHLING,
    SOMMER,
    HERBST,
    WINTER
  }

  protected Jahreszeit jahreszeit = null;

  public Schlussverkauf(Jahreszeit jahreszeit) {
    this.jahreszeit = jahreszeit;
  }

  public double getSkonto() {
    double ret = 0.0;
    switch(this.jahreszeit) {
      case SOMMER: { ret = 0.3; break; }
      case WINTER: { ret = 0.4; break; }
    }

    if (this.jahreszeit.compareTo(Jahreszeit.SOMMER) <= 0)
      ret += 0.1;
    return ret;
  }
}
```

ACHTUNG: Bei `case` kein `Jahreszeit.SOMMER`

- Technisch gesehen werden Aufzählungstypen in Java als *Klassen* und deren Werte als *Objekte* realisiert.
- Enthalten Methoden `toString()`, `equals()` und implementieren Interface `Comparable` und `Serializable`.
- Collection-Klassen `EnumSet` und `EnumMap` nehmen Mengen auf.

Verwendung:

```
import net.gobbz.enumpackage.Schlussverkauf.Jahreszeit;
import net.gobbz.enumpackage.Schlussverkauf;

public class EnumTest3
{
  public static void main(String[] args) {
    Jahreszeit j1 = Jahreszeit.SOMMER;
    Schlussverkauf s = new Schlussverkauf(j1);
    System.out.println(s.getSkonto());

    Jahreszeit j2 = Jahreszeit.valueOf("FRUEHLING");
    vergleich(j1,j2);

    System.out.println("Jahreszeiten:");
    for (Jahreszeit j : Jahreszeit.values())
      System.out.println(j);
  }

  public static void vergleich(Jahreszeit j1, Jahreszeit j2) {
    System.out.println(j1 + (j1 == j2 ? " == " : " != ") + j2);
  }
}
```

```
0.4
SOMMER != FRUEHLING
Jahreszeiten:
FRUEHLING
SOMMER
HERBST
WINTER
```

Generics

Bis jetzt

```
public static void ausgabeSortiert1(int[] args) {
  Vector v = new Vector();
  for (int i = 0; i < args.length; i++)
    v.addElement(new Integer(args[i]));
  Collections.sort(v);
  for (int i = 0; i < v.size(); i++)
    System.out.print(((Integer)v.elementAt(i)).intValue() + " ");
}
```

Und nun

```
public static void ausgabeSortiert2(int... args) {
  Vector<Integer> v = new Vector<Integer>();
  for (int i : args)
    v.addElement(i);
  Collections.sort(v);
  for (int i: v)
    System.out.print(i + " ");
}
```

- Einfügeoperationen werden typsicher.
- Compiler stellt sicher, dass nur passende Objekte eingefügt werden.
- Umständliche Typkonvertierung entfällt.
- Auch mehrere Typparameter möglich:

```
Hashtable<String,Integer> h =
  new Hashtable<String,Integer>();
```

Einfache typisierte Listenklasse mit typsicherem Iterator

```
public class MiniListe<T> implements Iterable<T>
{
  private Object[] data = null;
  private int size = 0;
  public MiniListe(int maxSize) {
    this.data = new Object[maxSize];
  }
  public void addElement(T element) {
    if (size >= data.length)
      throw new ArrayIndexOutOfBoundsException();
    data[size++] = element;
  }
  public int size() {
    return size;
  }
  public T elementAt(int pos) {
    if (pos >= size)
      throw new NoSuchElementException();
    return (T)data[pos];
  }
  public Iterator<T> iterator() {
    return new Iterator<T>() {
      int pos = 0;
      public boolean hasNext() {
        return pos < size;
      }
      public T next() {
        if (pos >= size)
          throw new NoSuchElementException();
        return (T)data[pos++];
      }
      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
}
```

- `T` steht für den Datentyp, mit dem Klasse später instanziiert wird.
- Interface `Iterable` enthält Methode `iterator()`.
- Es darf kein *typvariabler Generic-Konstruktor* aufgerufen werden z.B. new `T[]`☹ oder new `T[maxSize]`☹. **ABER**:

```
public class MiniListe<T> implements Iterable<T> {
  private T[] data = null;
  public MiniListe(int maxSize) {
    this.data = (T[]) new Object[maxSize];
  }
  …
}
```

Untypisierte Verwendung

```
MiniListe l1 = new MiniListe(10);
l1.addElement(3.14);
l1.addElement("Wort");
for (Object o : l1)
  System.out.println(o);
```

Ganzzahlige Typisierung

```
MiniListe<Integer> l2 = new MiniListe<Integer>(10);
l2.addElement(3);
l2.addElement(4);
l2.addElement(2);
for (Integer i : l2)
  System.out.println(i);
```