Generische Klassen & Methoden

Carsten Gips (HSBI)

Generische Strukturen

```
Vector speicher = new Vector();
speicher.add(1); speicher.add(2); speicher.add(3);
speicher.add("huhu");

int summe = 0;
for (Object i : speicher) { summe += (Integer)i; }
```

Generische Strukturen

```
Vector speicher = new Vector();
speicher.add(1); speicher.add(2); speicher.add(3);
speicher.add("huhu");

int summe = 0;
for (Object i : speicher) { summe += (Integer)i; }
```

```
Vector<Integer> speicher = new Vector<Integer>();
speicher.add(1); speicher.add(2); speicher.add(3);
speicher.add("huhu");

int summe = 0;
for (Integer i : speicher) { summe += i; }
```

Generische Klassen/Interfaces definieren

- **Definition**: "`<Typ>`" hinter Klassennamen

```
public class Stack<E> {
    public E push(E item) {
        addElement(item);
        return item;
    }
}
```

  - `Stack<E>` => Generische (parametrisierte) Klasse (auch: "*generischer Typ*")
  - `E` => Formaler Typ-Parameter (auch: "*Typ-Variable*")

Generische Klassen/Interfaces definieren

- **Definition**: "`<Typ>`" hinter Klassennamen

```
public class Stack<E> {
    public E push(E item) {
        addElement(item);
        return item;
    }
}
```

  - `Stack<E>` => Generische (parametrisierte) Klasse (auch: "*generischer Typ*")
  - `E` => Formaler Typ-Parameter (auch: "*Typ-Variable*")

- **Einsatz**:

```
Stack<Integer> stack = new Stack<Integer>();
```

  - `Integer` => Typ-Parameter
  - `Stack<Integer>` => Parametrisierter Typ

Beispiel I: Einfache generische Klassen

```
class Tutor<T> {
    // T kann in Tutor *fast* wie Klassenname verwendet werden
    private T x;
    public T foo(T t) { ... }
}
```

```
Tutor<String>  a = new Tutor<String>();
Tutor<Integer> b = new Tutor<>();  // ab Java7: "Diamond Operator"

a.foo("wuppie");
b.foo(1);
b.foo("huhu");  // Fehlermeldung vom Compiler
```

Beispiel II: Vererbung mit Typparametern

```
interface Fach<T1, T2> {
    public void machWas(T1 a, T2 b);
}

class SHK<T> extends Tutor<T> { ... }

class PM<X, Y, Z> implements Fach<X, Z> {
    public void machWas(X a, Z b) { ... }
    public Y getBla() { ... }
}

class Studi<A,B> extends Person { ... }
class Properties extends Hashtable<Object,Object> { ... }
```

Beispiel III: Überschreiben/Überladen von Methoden

```
class Mensch { ... }

class Studi<T extends Mensch> {
    public void f(T t) { ... }
}

class Prof<T> extends Mensch { ... }

class Tutor extends Studi<Mensch> {
    public void f(Mensch t) { ... }      // Ueberschreiben
    public void f(Tutor t) { ... }       // Ueberladen
}
```

Vorsicht: So geht es nicht!

```
class Foo<T> extends T { ... }

class Fluppie<T> extends Wuppie<S> { ... }
```

Generische Methoden definieren

- "`<Typ>`" vor Rückgabetyp

```
public class Mensch {
    public <T> T myst(T m, T n) {
        return Math.random() > 0.5 ? m : n;
    }
}
```

Generische Methoden definieren

- "`<Typ>`" vor Rückgabetyp

```
public class Mensch {
    public <T> T myst(T m, T n) {
        return Math.random() > 0.5 ? m : n;
    }
}
```

- "Mischen possible":

```
public class Mensch<E> {
    public <T> T myst(T m, T n) { ... }
    public String myst(String m, String n) { ... }
}
```

Aufruf generischer Methoden

```
class Mensch {
    <T> T myst(T m, T n) { ... }
}
Mensch m = new Mensch();


m.<String>myst("Essen", "lecker");  // Angabe Typ-Parameter


m.myst("Essen", 1);           // String, Integer => T: Object
m.myst("Essen", "lecker");    // String, String  => T: String
m.myst(1.0, 1);               // Double, Integer => T: Number
```

Beispiel methods.GenericMethods

Wrap-Up

- Begriffe:
  - Generischer Typ: `Stack<T>`
  - Formaler Typ-Parameter: `T`
  - Parametrisierter Typ: `Stack<Long>`
  - Typ-Parameter: `Long`
  - Raw Type: `Stack`
- Generische Klassen: `public class Stack<E> { }`
  - "`<Typ>`" hinter Klassennamen
- Generische Methoden: `public <T> T foo(T m) { }`
  - "`<Typ>`" vor Rückgabewert

LICENSE