

Teil III

Java OO

Puzzle #2: Was passiert?

```
class Puzzle2 <U, T> {  
    static class Type <A> {  
        class Constraint <B extends A> extends Type <B> {}  
        <B> Constraint <? super B> oops () { return null; }  
        <B> A coerce (B b) { return pair (this. <B> oops (), b).value; }  
    }  
    static class Sum <T> {  
        Type <T> type; T value;  
        Sum (Type <T> t, T v) { type = t; value = v; }  
    }  
    static <T> Sum <T> pair (Type <T> type, T value) {  
        return new Sum <T> (type, value); }  
    static <T, U> U coerce (T t) {  
        Type <U> type = new Type <U> ();  
        return type. <T> coerce (t); }  
    public static void main (String [] args) {  
        String zero = Puzzle2. <Integer, String> coerce (0); }  
}
```

Gliederung

- 17 Klassen
- 18 Schnittstellen
- 19 Vererbung
- 20 Untertypen

- 21 Generics
- 22 Iteratoren
- 23 Innere Klassen

Java: Features — OO

- Java ist eine klassenbasierte, objektorientierte Sprache
- ein Java Paket besteht aus Schnittstellen und Klassen
- Methoden sind standardmäßig virtuell
- (keine Mehrfachvererbung)
- (kein syntaktischer Zucker für Getter und Setter)

Klassen

```
type Item =  
  abstract Price : int  
type Cart (customer : int) =  
  let mutable items : Item list = []  
  member self.Customer  
    with get () = customer  
  member self.Add (item : Item) =  
    items ← item :: items  
  member self.Total () : int =  
    items |> List.sumBy  
      (fun item → item.Price)
```

```
interface Item { int price (); }  
public class Cart {  
  private final int customer;  
  private final List<Item> items =  
    new ArrayList<Item>();  
  Cart (int customer) {  
    this.customer = customer; }  
  int getCustomer () { return customer; }  
  void add (Item item) {  
    items.add (item); }  
  int total () {  
    int sum = 0;  
    for (Item item : items)  
      sum += item.price ();  
    return sum;  
  }  
}
```

Konstruktoren

- primärer Konstruktor

```
type TrustMe (seed : int) =  
  class  
  ...  
  end
```

- sekundärer Konstruktor

```
type TrustMe (seed : int) =  
  class  
  ...  
  new () = TrustMe (10)  
  ...  
  end
```

- beliebig viele Konstruktoren
- wenn keiner angegeben wird:
0-stelliger Konstruktor wird
bereitgestellt
- Aufruf eines anderen Konstruktors:
this (...)

```
TrustMe (long seed) {  
  this.funds = seed;  
}  
TrustMe () {  
  this (10);  
}
```

Das Objekt selbst

- frei wählbarer Bezeichner

```
member me.AddSeq (is : Item seq) =  
  for i in is do  
    me.Add i
```

- Empfänger ist explizit
- Selbstnachricht: *me.Add*

- vorgegeben: **this**

```
void add (Iterable <Item> is) {  
  for (Item i : is)  
    add (i);  
}
```

- Empfänger ist implizit
- Selbstnachricht: *add* oder **this.add**

Mutter aller Klassen: *Object*

- jede Klasse erbt von genau einer anderen Klasse
- wird keine angegeben: erbt von *System.Object*
- Methoden der Klasse *Object*
 - **protected** *MemberwiseClone* :
unit → *Object*
flache Kopie des Objekts
 - *Equals* : *Object* → *bool*
Test auf Gleichheit
 - *GetHashCode* : *unit* → *int*
Hashwert
 - *ToString* : *unit* → *string*
Stringrepräsentation des Objekts

- jede Klasse erbt von genau einer anderen Klasse
- wird keine angegeben: erbt von *java.lang.Object*
- Methoden der Klasse *Object*
 - **protected** *Object clone* ()
flache Kopie des Objekts
 - *boolean equals* (*Object obj*)
Wertegleichheit
 - *int hashCode* ()
Hashwert
 - *String toString* ()
Stringrepräsentation

Zugriffsrechte

- sichtbar in der Klasse

```
member private self.CallMe () = ...
```

- sichtbar in der gleichen Assembly

```
member internal self.CallMe () = ...
```

- überall sichtbar

```
member public self.CallMe () = ...
```

- ditto

```
member self.CallMe () = ...
```

- 🖱️ Felder sind immer privat

- sichtbar in der Klasse

```
private void callMe () { ... }
```

- sichtbar im gleichen Paket

```
void callMe () { ... }
```

- sichtbar im gleichen Paket *und* in Unterklassen

```
protected void callMe () { ... }
```

- überall sichtbar

```
public void callMe () { ... }
```

- 🖱️ Felder sollten privat sein, können aber zugänglich gemacht werden

Klassenvariablen und -methoden

- „statische“ Anweisung

```
static do  
  printfn "TrustMe is founded."
```

- Klassenvariable

```
static let mutable no = 0
```

- ... wird verändert in ...

```
do no ← no + 1
```

- Klassenmethode

```
static member  
  no_of_accounts = no
```

- „statische“ Anweisung

```
static {  
  print ("TrustMe is founded."); }  
}
```

- Klassenvariable

```
static private int no = 0;
```

- ... wird verändert in ...

```
TrustMe (long seed) {  
  this.funds = seed;  
  no ++; }  
}
```

- Klassenmethode

```
static int no_of_accounts () {  
  return no; }  
}
```

Schnittstellen

```
type Calculator =  
  abstract Abs : unit → unit  
  abstract Add : int → unit  
  abstract Value : int
```

```
interface Calculator {  
  void abs ();  
  void add (int x);  
  int value ();  
}
```

- *explizite* Implementierung von Schnittstellen

```
type HP () =  
  let mutable reg = 0  
  member self.Abs () =  
    reg ← abs reg  
  member self.Add (x : int) =  
    reg ← reg + x  
  member self.Value = reg  
interface Calculator with  
  member s.Add x = s.Add x  
  member s.Abs () = s.Abs ()  
  member s.Value = s.Value
```

- eine Klasse kann beliebig viele Schnittstellen implementieren

Schnittstellen

- *implizite* Implementierung von Schnittstellen (*Problem: Turncoat*)

```
class HP implements Calculator {  
  int reg = 0;  
  public void abs () {  
    reg = Math.abs (reg);  
  }  
  public void add (int x) {  
    reg += x;  
  }  
  public int value () {  
    return reg;  
  }  
}
```

- eine Klasse kann beliebig viele Schnittstellen implementieren

Objektausdrücke aka anonyme Klassen

- Implementierung der Schnittstelle mit einem Objektausdruck

```
let casio () =  
  let mutable reg = 0  
  { new Calculator with  
    member self.Abs () =  
      reg ← abs reg  
    member self.Add (x : int) =  
      reg ← reg + x  
    member self.Value = reg  
  }
```

- Implementierung der Schnittstelle mit einer anonymen Klasse

```
final class Casio {  
  static Calculator casio () {  
    return new Calculator () {  
      int reg = 0;  
      public void abs () {  
        reg = Math.abs (reg);  
      }  
      public void add (int x) {  
        reg += x;  
      }  
      public int value () {  
        return reg;  
      }  
    };  
  }  
}
```


Java: *default* Methoden

```
public interface IStack<T> extends Iterable<T>
{
    public IStack<T> push (T x);
    default IStack<T> push (T... a) {
        for (T x : a)
            this.push (x);
        return this;
    }
    public T pop ();
    default T top () {
        T x = pop ();
        push (x);
        return x;
    }
    public int length ();
    default boolean isEmpty () {
        return (length () == 0);
    }
}
```

Java: Abstrakte Klassen

- *Abstrakte Klassen* dienen der Faktorisierung von Methoden in gemeinsame Oberklassen. Eine abstrakte Klasse definiert ein abstraktes Konzept, das nicht selbst implementiert werden kann.

```
import java.awt.Point;  
abstract class Shape {  
    Point origin;  
    void move (int dx, int dy) {  
        origin.translate (dx, dy);  
    }  
    abstract void rotate (float degrees);  
}
```

-  Eine abstrakte Klasse implementiert nicht alle Methoden und kann deshalb keine Objekte haben.

Vererbung

```
exception Limit
type TrustMeStudent
  (seed : int, limit : int) =
class
  inherit TrustMe (seed)
  let limit = min limit 1000
  override self.Withdraw amount =
    if amount > limit then
      raise Limit
    else
      base.Withdraw amount
end
```

- Zugriff auf die Mitglieder der Basisklasse: **base**

```
class Limit extends Exception { }
class TrustMeStudent extends TrustMe {
  private long limit = 0;
  TrustMeStudent (long seed, long limit) {
    super (seed);
    this.limit =
      Math.min (limit, 1000); }
  @Override
  void withdraw (long amount)
    throws Insufficient, Limit {
    if (amount > limit)
      throw new Limit ();
    else
      super.withdraw (amount); }
}
```

- Zugriff auf die Basisklasse: **super**


Vererbung: Redefinition

- ich erlaube *Withdraw* zu redefinieren

```
type TrustMe (seed : int) =  
  ...  
  member self.Deposit amount =  
    funds ← funds + amount  
  abstract Withdraw : int → unit  
  default self.Withdraw amount =  
    if amount > funds then  
      raise (Insufficient funds)  
    else  
      funds ← funds - amount  
  ...
```

- ich verbiete *deposit* zu redefinieren

```
class TrustMe {  
  ...  
  final void deposit (long amount) {  
    funds += amount;  
  }  
  void withdraw (long amount)  
    throws Insufficient, Limit {  
    if (amount > funds)  
      throw  
        new Insufficient (funds);  
    else  
      funds -= amount;  
  } ...  
}
```

-  Der Typ von *withdraw* hat sich geändert!

Vererbung: Konstruktoren

- primärer Konstruktor

```
type TrustMeStudent  
  (seed : int, limit : int) =  
  class  
    inherit TrustMe (seed)  
    ...  
  end
```

- sekundärer Konstruktor

```
new (limit : int) =  
  TrustMeStudent (10, limit)
```

- Konstruktor

```
TrustMeStudent  
  (long seed, long limit) {  
    super (seed);  
    this.limit =  
      Math.min (limit, 1000);  
  }
```

- Aufruf des Basiskonstruktors:
super (...)
- fehlt der Aufruf, wird **super** ()
ergänzt

- Klasse *Phone*

```
public class Phone
{
    void dial (int no) {...}
}
```

- Klasse *Mobile*

```
public class Mobile extends Phone
{
    void play (String mp3) {...}
    int number (String nick) {...}
    void call (String nick) {
        dial (number (nick));
    }
}
```

Untertypen

- Schnittstellenvererbung

```
type Item =  
  abstract Price : int  
type Coupon =  
  inherit Item  
  abstract Valid : bool
```

- „upcast“: $e :> t$ (statisch)

```
let i : Item = ...  
...  
let c : Coupon = ...  
...  
let cart = [i; ...; c :> Item; ...]
```

- Typanpassung notwendig

- Schnittstellenvererbung

```
interface Item {  
  int price ();  
interface Coupon extends Item {  
  boolean valid ();  
}
```

- „upcast“: $(t) e$ (statisch)

```
Item i = ...  
...  
Coupon c = ...  
...  
var cart = new Item [] { i, ..., c, ... };
```

- Typanpassung automatisch

Untertypen

- „downcast“: $e : ? > t$ (dynamisch)
- Typtest: $e : ? t$ (dynamisch)

```
let mutable total = 0
for i in cart do
  if i :? Coupon &&
    (i :? > Coupon).Valid then
    total ← total + i.Price
printfn "reduction %d" total
```

- downcast: dynamische Prüfung

- „downcast“: $(t) e$ (dynamisch)
- Typtest: $e \text{ instanceof } t$ (dynamisch)

```
int total = 0;
for (var i : cart)
  if (i instanceof Coupon &&
    ((Coupon) i).valid ())
    total += i.price ();
printf ("reduction %d\n", total);
```

- downcast: dynamische Prüfung

Quiz: Überladung

```
class C {  
    void op (C c)          { System.out.print ("1"); }  
}  
  
class SC extends C {  
    // overriding  
    public void op (C c)  { System.out.print ("2"); }  
    // overloading  
    public void op (SC c) { System.out.print ("3"); }  
}  
  
public class Overloading {  
    public static void main (String [] args) {  
        C c1 = new C (); C c2 = new SC (); SC sc = new SC ();  
        c1.op (c1); c1.op (c2); c1.op (sc);    // Which bodies  
        c2.op (c1); c2.op (c2); c2.op (sc);    // will be  
        sc.op (c1); sc.op (c2); sc.op (sc);    // executed ?  
    }  
}
```

Generische Funktionen

- generische Funktion

```
let swap <'a, 'b> (pair : 'a * 'b) : 'b * 'a =  
    (snd pair, fst pair)
```

- Typparameter sind optional

```
let swap (pair : 'a * 'b) : 'b * 'a =  
    (snd pair, fst pair)
```

- Typangaben sind ebenfalls optional

```
let swap pair = (snd pair, fst pair)
```

- generische Funktion

```
public static <A, B>  
    Pair <B, A> swap (Pair <A, B> pair) {  
    return new  
        Pair <B, A>(pair.snd, pair.fst);  
}
```

- Typparameter sind notwendig

Beschränkt generische Funktionen

- beschränkter Typparameter

```
let total <'a when 'a :> Item>  
  (items : 'a seq) : int =  
  let mutable sum = 0  
  for item in items do  
    sum ← sum + item.Price  
  sum
```

- beschränkter Typparameter

```
static <T extends Item>  
  int total (List<T> items) {  
    int sum = 0;  
    for (Item item : items)  
      sum += item.price ();  
    return sum;  
  }
```


Beschränkt generische Funktionen: Quasiordnungen

- Quasiordnung (C#)

```
type IComparable <T> =  
    abstract CompareTo :: T → int
```

- Einschränkung: $'a : comparison$

```
let sort2 <'a when 'a : comparison>  
    (x : 'a, y : 'a) : 'a * 'a =  
    if x ≤ y then (x, y)  
    else (y, x)
```

- kürzer

```
let sort2 (x, y) =  
    if x ≤ y then (x, y)  
    else (y, x)
```

- Quasiordnung

```
interface Comparable <T> {  
    public int compareTo (T that);  
}
```

- Einschränkung:

```
S extends Comparable <S>
```

```
static <T extends Comparable <T>>  
    Pair <T, T> sort2 (T a, T b) {  
    if (a.compareTo (b) ≤ 0)  
        return new Pair <T, T>(a, b);  
    else  
        return new Pair <T, T>(b, a);  
}
```

Java Generics: Wildcards

- Wildcards: *? extends T* und *? super T*

```
static int total (List <? extends Item> items) {  
    int sum = 0;  
    for (Item item : items)  
        sum += item.price ();  
    return sum;  
}
```

- Auszug Schnittstelle *Stream <T>*:

```
<R> Stream <R> map (Function <? super T, ? extends R> mapper)
```

- (F#: Flexible Typen)

```
let total (items : #Item seq) = ...
```

Java Generics: Kovariante Arrays

- Arrays sind kovariant (Bug): wenn $S \preccurlyeq T$, dann $S [] \preccurlyeq T []$

```
Mobile [] harrys = { aMobile };  
Phone [] herberts = harrys;  
herberts [0] = aPhone;  
freds [0].play ("Siberian Khatru");
```

- die Zuweisung `herberts [0] = aPhone` verursacht einen *Laufzeitfehler*:
`java.lang.ArrayStoreException`
- zur Erinnerung: **ref**s und 'mutable data structures' sind invariant

Java Generics: Boxing

- nicht zulässig: *ArrayList* \langle *int* \rangle
- (siehe auch funktionale Schnittstellen)
- „Wrapper“ Klasse: *Integer*
- zulässig: *ArrayList* \langle *Integer* \rangle

Primitiver Typ	„Wrapper“ Klasse
<i>boolean</i>	<i>Boolean</i>
<i>byte</i>	<i>Byte</i>
<i>char</i>	<i>Character</i>
<i>float</i>	<i>Float</i>
<i>int</i>	<i>Integer</i>
<i>long</i>	<i>Long</i>
<i>short</i>	<i>Short</i>
<i>double</i>	<i>Double</i>

- „automatic boxing and unboxing“

F#: Binäre Suchbäume

- persistente binäre Suchbäume

```
type Tree <t when t : comparison> =  
  | Leaf  
  | Node of Tree <t> * t * Tree <t>  
let rec contains key = function  
  | Leaf          → false  
  | Node (l, x, r) → if key ≤ x then contains key l  
                    elif key = x then true  
                    else contains key r  
  
let rec insert key = function  
  | Leaf          → Node (Leaf, key, Leaf)  
  | Node (l, x, r) → if key ≤ x then Node (insert key l, x, r)  
                    else Node (l, x, insert key r)
```

Java: Binäre Suchbäume — Teil 1

- persistente binäre Suchbäume

```
public abstract class Tree<Elem>
{
    abstract boolean contains (Elem elem);
    abstract Tree<Elem> insert (Elem elem);
    public static <T extends Comparable<T>> Tree<T> leaf () {
        ...
    }
    public static <T extends Comparable<T>> Tree<T>
        node (final Tree<T> left, final T key, final Tree<T> right) {
        ...
    }
}
```

- generische Objektconstructoren: *leaf* und *node*
- (abstrakte Klasse dient gleichzeitig als Modul)

Java: Binäre Suchbäume — Teil 2

```
public static <T extends Comparable<T>> Tree<T> leaf () {  
    return new Tree<T>() {  
        public boolean contains (T elem) {  
            return false;  
        }  
        public Tree<T> insert (T elem) {  
            return node (Tree.<T>leaf (), elem, Tree.<T>leaf ());  
        }  
    };  
}
```

Java: Binäre Suchbäume — Teil 3

```
public static <T extends Comparable<T>> Tree<T>
  node (final Tree<T> left, final T key, final Tree<T> right) {
  return new Tree<T>() {
    public boolean contains (T elem) {
      int cmp = elem.compareTo (key);
      if (cmp < 0)
        return left.contains (elem);
      else if (cmp == 0)
        return true;
      else
        return right.contains (elem); }
    public Tree<T> insert (T elem) {
      if (elem.compareTo (key) ≤ 0)
        return node (left.insert (elem), key, right);
      else
        return node (left, key, right.insert (elem)); }
  };
}
```


Aufzähler

- Aufzähler

```
type IEnumerator <T> =  
  abstract Current : T  
  abstract MoveNext : unit → bool  
  abstract Reset : unit → unit
```

- Logik:

- weitere? *MoveNext*
- zum nächsten: *MoveNext*
- Element: *Current*

- aufzählbare Objekte

```
type IEnumerable <T> =  
  abstract GetEnumerator :  
    unit → IEnumerator <T>
```

- Iterator

```
interface Iterator <T> {  
  boolean hasNext ();  
  T next ();  
  void remove ();  
}
```

- Logik:

- weitere? *hasNext*
- zum nächsten: *next*
- Element: *next*

- iterierbare Objekte

```
interface Iterable <T> {  
  Iterator <T> iterator ();  
}
```

„for each“ Schleifen

- „for each“ Schleife

```
for x in xs do  
  ...
```

- „for each“ Schleife

```
for (var x : xs) {  
  ...  
}
```

Java: Innere Klassen

- listenbasierte Implementierung von Stacks

```
public class ListStack<T> implements IStack<T>
{
    static private class Cons<S> {
        S      head;
        Cons<S> tail;
        Cons(S head, Cons<S> tail) {
            this.head = head;
            this.tail = tail;
        }
    }
    private Cons<T> top = null;
    ...
}
```

-  die leere Liste wird durch *null* repräsentiert

Java: Innere Klassen

```
public IStack<T> push (T x) {  
    top = new Cons<T>(x, top);  
    return this;  
}  
  
public T pop () {  
    if (top == null) throw new NoSuchElementException ();  
    else {  
        T x = top.head;  
        top = top.tail;  
        return x;  
    }  
}  
  
public int length () {  
    int i = 0;  
    for (Cons<T> cur = top; cur != null; cur = cur.tail) i++;  
    return i;  
}  
...
```

Java: Innere Klassen

```
...
public Iterator <T> iterator () {
    return new Iterator <T>() {
        private Cons <T> cur = top;
        public boolean hasNext () { return cur != null; }
        public T next () {
            if (cur == null)
                throw new NoSuchElementException ();
            else {
                T head = cur.head;
                cur = cur.tail;
                return head;
            }
        }
    };
}
```