

Teil II

Java Core

Puzzle #1: Was wird ausgegeben?

```
int count = 0;  
for (int i = 0; i < 100; i ++); {  
    count ++;  
}  
println (count);
```

Gliederung

- 3 Kommentare
- 4 Wahrheitswerte
- 5 Zahlen
- 6 Zeichen und Strings
- 7 Blöcke
- 8 Funktionen
- 9 Tupel und Records

- 10 Varianten
- 11 Listen
- 12 Arrays
- 13 Ein- und Ausgabe
- 14 Zustand
- 15 Schleifen
- 16 Ausnahmen

Java: Features — Core

- Java kennt im Wesentlichen nur Klassendefinitionen
 - Records sind spezielle Klassen
 - Varianten werden nicht unterstützt
 - Module sind spezielle Klassen
 - Arrays sind spezielle Objekte
 - Ausnahmen sind spezielle Objekte
- Java kennt im Wesentlichen nur Methoden
 - Funktionen sind spezielle Methoden
- Javas Syntax ist an C angelehnt
- Java unterscheidet zwischen Ausdrücken und Anweisungen
- Bezeichner sind standardmäßig veränderlich
- Kontrollkonstrukte
 - Alternative: **if** (...)
 - Fallunterscheidung: **switch** (...)
 - Schleifen: **for** (...; ...; ...), **while** (...), **do** {...} **while** (...)
- 🖱️ *Annahme*: Sie können F#
- 🖱️ *Vereinfachung*: `printf` statt `System.out.printf`

Kommentare

- einzeilige Kommentare

```
// single-line comment
```

- mehrzeilige Kommentare

```
(*  
multi-line  
comment  
*)
```

- einzeilige Kommentare

```
// single-line comment
```

- mehrzeilige Kommentare

```
/*  
multi-line  
comment  
*/
```

Wahrheitswerte

- Typ: *boolean*
- Wahrheitswerte:

false, true

- Alternative (Ausdruck):

$e_1 ? e_2 : e_3$

- Alternative (Anweisung):

if (*e*)
 s1
else
 s2

- Verknüpfungen: *!*, *&&*, *||*

- Typ: *bool*
- Wahrheitswerte:

false, true

- Alternative:

if *e*₁ *then* *e*₂ *else* *e*₃

- Verknüpfungen: *not*, *&&*, *||*

Ganze Zahlen: beschränkte Genauigkeit

- Typ: *int* (32 Bit), *int64* (64 Bit)

- Numerale:

– 4711, 2147483647
– 815L, 9223372036854775807L

- arithmetische Operationen:

+, −, *, /, %

- Vergleichsoperationen:

<, ≤, =, <>, ≥, >

- ASCII: <= und >=

- Typ: *int* (32 Bit), *long* (64 Bit)

- Numerale:

– 4711, 2147483647
– 815L, 9223372036854775807L

- arithmetische Operationen:

+, −, *, /, %

- Vergleichsoperationen:

<, ≤, ==, !=, ≥, >

- ASCII: <= und >=

Ganze Zahlen: unbeschränkte Genauigkeit

- Typ: *bigint*
- Numerale

```
OI  
1I  
2432902008176I
```

- Fakultät

```
let rec factorial (n : bigint) =  
  if n = OI then  
    1I  
  else  
    n * factorial (n - 1I)
```

- Typ: *BigInteger* aus *java.math*

```
import java.math.BigInteger;
```

- Numerale

```
BigInteger.ZERO  
BigInteger.ONE  
new BigInteger ("2432902008176")
```

- Fakultät

```
public static  
BigInteger factorial (BigInteger n) {  
  if (n.equals (BigInteger.ZERO))  
    return BigInteger.ONE;  
  else  
    return n.multiply (factorial (  
      n.subtract (BigInteger.ONE)));  
}
```


- Typen: *char* und *string*
- Literale


```
let lang = 'C'  
let hiya = "Hi Lisa!"
```

- Operationen

```
let hw = "hello," + " world"
```

- Vergleichsoperationen:
<, ≤, =, <>, ≥, >


Zeichen und Strings

- Typen: *char* und *String*
-  Zeichen und Strings sind unveränderlich
- Literale

```
char lang = 'C'  
String hiya = "Hi Lisa!"
```

- Operationen

```
String hw = "hello," + " world"
```

- Vergleichsoperationen (*char*):
<, ≤, ==, !=, ≥, >
- Vergleichsoperationen (*String*):
s.equals(t), *s.compareTo(t)*
 Vergleich mit ==, != vergleicht nicht den Inhalt der Strings.

- Definitionen (Layout)

```
let a = 4711  
let b = 815  
a + b
```

☞ ein Ausdruck

- Anweisungen

```
let mutable x = 4711  
x ← x + 1  
printf "x is = %d\n" x
```

☞ ebenfalls ein Ausdruck

- ';' ist ein Separator (Operator): er verknüpft zwei Ausdrücke

Definitionen versus Blöcke

- Blöcke (explizite {...} Klammern)

```
{  
  final int a = 4711;  
  final int b = 815;  
  return a + b;  
}
```

☞ kein Ausdruck

- Anweisungen

```
{  
  int x = 4711;  
  x = x + 1;  
  printf ("x is = %d\n", x);  
}
```

- ';' ist ein *Terminator*: er überführt einen Ausdruck in eine Anweisung

Funktionen

- Beispiel: Minimum

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

- Beispiel: Minimum

```
static final int min (int x, int y) {  
  if (x ≤ y)  
    return x;  
  else  
    return y;  
}
```

- alternativ mit *if*-Ausdruck:

```
static final int min (int x, int y) {  
  return x ≤ y ? x : y;  
}
```

Prozeduren

- Typ: $t \rightarrow unit$

```
let dump (xs : int list) =  
  for x in xs do  
    printf "%d\n" x
```

- Ergebnistyp *void*

```
static final  
void dump (List<Integer> xs) {  
  for (Integer x : xs)  
    printf ("%d\n", x);  
}
```

Rekursive Funktionen

- Schlüsselwort *rec*

```
let rec fib n =  
  if n = 0 then 0  
  elif n = 1 then 1  
  else fib (n - 1) + fib (n - 2)
```

- Funktionsname im Rumpf sichtbar

```
static final int fib (int n) {  
  if (n == 0)  
    return 0;  
  else if (n == 1)  
    return 1;  
  else  
    return fib (n - 1) + fib (n - 2);  
}
```

Funktionsausdrücke

- Funktionsausdruck

fun (*a* : *int*) $\rightarrow 2 * a$

- (aka Lambda-Ausdruck)

- Funktionsausdruck

(*int* *n*) $\rightarrow 2 * n$

- (aka Lambda-Ausdruck)

Java: Lambda-Ausdrücke

- Der Rumpf eines Lambda-Ausdrucks kann ein Ausdruck sein:

```
(int n) → 2 * n
```

- Der Typ des formalen Parameters kann ausgelassen werden:

```
n → 2 * n
```

- Der Rumpf eines Lambda-Ausdrucks kann ein Block sein:

```
(int n) → {  
    for (int i = 0; i < n; i++)  
        println ("hello, world");  
}
```

Java: Funktionale Schnittstellen

- Geschichtliches: wie wird der Typ einer Funktion notiert?
 - Mathematik (< 1940): $f(X) \subset Y$
 - Witold Hurewicz (1904–1956): $f : X \rightarrow Y$
 - C: $Y f(X x)$
- In Java sind Funktionen Objekte: zum Beispiel,

```
Function f;
```

- wobei *Function* eine sogenannte *funktionale Schnittstelle* ist,

```
interface Function {  
    R apply (T x);  
}
```

die nur eine abstrakte Methode enthält.

Java: Funktionale Schnittstellen

- Das heißt, wir müssen für jeden Funktionstyp eine neue Schnittstelle definieren?
- Im Prinzip ja, in der Praxis nein, da `java.util.function` über 40 (!) Schnittstellen vordefiniert, zum Beispiel:

<i>IntUnaryOperator</i>	<i>applyAsInt</i>	<i>int</i> → <i>int</i>
<i>Consumer</i> $\langle T \rangle$	<i>accept</i>	<i>T</i> → <i>void</i>
<i>Predicate</i> $\langle T \rangle$	<i>test</i>	<i>T</i> → <i>boolean</i>
<i>Supplier</i> $\langle T \rangle$	<i>get</i>	() → <i>T</i>
<i>Function</i> $\langle A, B \rangle$	<i>apply</i>	<i>A</i> → <i>B</i>

- 🖱️ nötig wegen primitiver Typen

Java: Typisierung von Lambda-Ausdrücken

- Wie wird der Typ eines Lambda-Ausdrucks bestimmt?

```
() → 4711
```

- Ist der Typ *Supplier* *<Integer>*? Oder vielleicht

```
interface Generator {  
    Integer generate ();  
}
```

- Beides! Java verwendet den Kontext, in dem der Lambda-Ausdruck auftritt, um den Typ zu bestimmen.
- Aus diesem Grund können Lambda-Ausdrücke nur in bestimmten Kontexten verwendet werden:
 - Deklaration von Variablen,
 - Zuweisungen,
 - **return** Anweisung, ...

Java: Funktionsapplikation ...

- ... ist Methodenaufruf.
- Beispiel:

```
Function f;
```

- wobei *Function* wie folgt definiert ist

```
interface Function {  
    R apply (T x);  
}
```

- Wir wenden *f* wie folgt an:

```
f.apply (n)
```

Funktionen höherer Ordnung

- Definition

```
let rec map (f : 'a → 'b) = function  
  | []      → []  
  | x :: xs → f x :: map f xs
```

- Anwendung

```
map (fun x → x * x) [3; 5; 7]
```

- Definition

```
static final <A, B>  
List <B> map (Function <A, B> f,  
             List <A> xs) {  
  List <B> ys = new LinkedList <B> ();  
  for (A x : xs) {  
    ys.add (f.apply (x));  
  }  
  return ys;  
}
```

- Anwendung

```
map (x → x * x, Arrays.asList (3, 5, 7))
```

Tupel

- Konstruktion

```
let lisa = ("Lisa", true, 14)
```

- Projektion (Muster)

```
let (name, female, age) = lisa
```

- Tupel werden *nicht* unterstützt
- Bibliothek: Functional Java
www.functionaljava.org
- Konstruktion

```
import static fj.P3;  
import fj.*;  
P3<String, Boolean, Integer>t =  
  p ("Lisa", true, 14);
```

- Muster werden *nicht* unterstützt

```
String name = t._1 ();  
boolean female = t._2 ();  
int age = t._3 ();
```

- (**import static**: direkter Zugriff ohne Punktnotation)

Paare

- Konstruktion

```
let pair = (4711, false)
```

- Projektion

```
fst pair, snd pair
```

- DIY

```
public class Pair <A, B> {  
    public final A fst;  
    public final B snd;  
    public Pair (final A fst,  
                final B snd) {  
        this.fst = fst;  
        this.snd = snd; }  
}
```

- Konstruktion

```
var pair = new Pair (4711, false)
```

- Projektion

```
pair.fst, pair.snd
```

Records

- Typdefinition

```
type Book = {  
  Title   : string  
  Author  : string  
  Pages   : int }
```

- Konstruktion

```
let notw = {  
  Title   =  
    "The Name of the Wind"  
  Author  = "Patrick Rothfuss"  
  Pages   = 662 }
```

- Extraktion

```
let title = notw.Title
```

- Klasse ohne Methoden

```
final class Book {  
  final String title;  
  final String author;  
  final int    pages;  
  
  Book (final String title,  
        final String author,  
        final int    pages) {  
    this.title   = title;  
    this.author  = author;  
    this.pages   = pages;  
  }  
}
```

- ...

Records

- Typdefinition

```
type Book = {  
  Title   : string  
  Author  : string  
  Pages   : int }
```

- Konstruktion

```
let notw = {  
  Title   =  
    "The Name of the Wind"  
  Author  = "Patrick Rothfuss"  
  Pages   = 662 }
```

- Extraktion

```
let title = notw.Title
```

- ...

- Konstruktion

```
var notw = new Book  
  ("The Name of the Wind",  
  "Patrick Rothfuss",  
  662);
```

- Extraktion

```
String title = notw.title;
```


- Typdefinition

```
type Expr =  
  | Const of int  
  | Add of Expr * Expr  
  | Mul of Expr * Expr
```

- Beispiel: Auswertung

```
let rec eval (expr : Expr) =  
  match expr with  
  | Const i      → i  
  | Add (e1, e2) → eval e1 + eval e2  
  | Mul (e1, e2) → eval e1 * eval e2
```

- wird *nicht* unterstützt
- „composite design pattern“
- „visitor design pattern“
- 90°: abstrakter Datentyp
 - mit Schnittstelle und Objektausdrücken oder
 - mit abstrakter Klasse und anonymen Klassen ...

```
abstract class Expr {  
    abstract int eval ();  
  
    static Expr Const (int i) {  
        return new Expr () {  
            int eval () { return i; }  
        };  
    }  
  
    static Expr Add (Expr e1, Expr e2) {  
        return new Expr () {  
            int eval () { return e1.eval () + e2.eval (); }  
        };  
    }  
  
    static Expr Mul (Expr e1, Expr e2) {  
        return new Expr () {  
            int eval () { return e1.eval () * e2.eval (); }  
        };  
    }  
}
```

- Typ: *t option*
- Konstruktion

```
None  
Some 4711
```

- Fallunterscheidung

```
match opt with  
| None → printf "sigh"  
| Some i → printf "got %d\n" i
```

Optionale Werte

- Typ: *Optional* $\langle t \rangle$
- Konstruktion

```
Optional.empty ()  
Optional.of (4711)
```

- Fallunterscheidung

```
if (opt.isEmpty ())  
    printf ("sigh\n");  
else  
    printf ("got %d\n", opt.get ());
```

```
if (opt.isPresent ())  
    printf ("got %d\n", opt.get ());  
else  
    printf ("sigh\n");
```

Listen

- Typ: *'a list*
- Konstruktion

```
let primes = 2 :: 3 :: 5 :: 7 :: 11 :: []
```

- Fallunterscheidung

```
match primes with
```

```
| []      → 0  
| x :: xs → x
```

- Struktur Entwurfsmuster

```
let rec sum = function
```

```
| []      → 0  
| x :: xs → x + sum xs
```

- unveränderliche Listen werden *nicht* unterstützt
- Bibliothek: Functional Java
- Konstruktion

```
import static fj.data.List.list;  
import fj.data.List;  
var primes = list (2, 3, 5, 7, 11);
```

- Struktur Entwurfsmuster

```
public static
```

```
int sum (List <Integer> list) {  
    if (list.isEmpty ()) return 0;  
    else  
        return list.head () +  
                sum (list.tail ());  
}
```

Arrays

- Typ: *int* []
- Definition

```
let primes = [| 2; 3; 5; 7; 11 |]
```

- Arraybeschreibung

```
let squares =  
  [| for i in 0..9 → i * i |]
```

- Größe

```
primes.Length
```

- Subskription

```
primes.[2]
```

- Typ: *int* []
- Deklaration

```
int [] primes = {2, 3, 5, 7, 11};
```

- Größe

```
primes.Length
```

- Subskription

```
primes [2]
```

Arrays

- Arrays sind Werte
- Konstruktion

```
[| 2; 3; 5; 7; 11 |]
```

- Arraybeschreibung

```
[| for i in 1..capacity → 0 |]
```

- Arrays sind Objekte
- Konstruktion

```
new int [] {2, 3, 5, 7, 11}
```

- mit 0 initialisiertes Array

```
new int [capacity]
```

Ein- und Ausgabe

- Ausgabe

```
printf ("%s is %d\n", "x", x);  
System.out.print (4711);  
System.out.print ("hiya");
```

☞ (eigentlich *System.out.printf...*)

- Eingabe

```
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
BufferedReader buffer =  
    new BufferedReader (  
        new InputStreamReader (  
            System.in));  
String line = buffer.readLine ();
```

- Ausgabe

```
printf "%s is %d\n" "x" x  
System.Console.WriteLine 4711  
System.Console.WriteLine "hiya"
```

- Eingabe

```
let line = System.Console.ReadLine ()
```

Veränderliche

- Veränderliche

```
let mutable x = 0  
x ← x + 1  
x ← 42
```

- Konstante (default)

```
let x = 42
```

- Veränderliche (default)

```
int x = 0;  
x = x + 1;  
x = 42;
```

- Konstante

```
final int x = 42
```


Java: Zuweisungen

- akkumulierende Zuweisungen

```
int x = 0;  
x += 1;  
x *= 2;
```

- Ausdrücke mit Effekt

```
x ++  
-- x
```

Veränderliche Records

- Typdefinition

```
type Book = {  
  mutable Title : string  
  mutable Author : string  
  mutable Pages : int }
```

- Konstruktion

```
let notw = {  
  Title =  
    "The Name of the Wind"  
  Author = "Patrick Rothfuss"  
  Pages = 662 }
```

- Modifikation

```
notw.Author ← "P. Rothfuss"
```

- Klasse ohne Methoden

```
final class Book {  
  String title;  
  String author;  
  int pages;  
  Book (String title,  
        String author,  
        int pages) {  
    this.title = title;  
    this.author = author;  
    this.pages = pages;  
  }  
}
```

- ...

Veränderliche Records

- Typdefinition

```
type Book = {  
  mutable Title   : string  
  mutable Author  : string  
  mutable Pages   : int }
```

- Konstruktion

```
let notw = {  
  Title   =  
    "The Name of the Wind"  
  Author  = "Patrick Rothfuss"  
  Pages   = 662 }
```

- Modifikation

```
notw.Author ← "P. Rothfuss"
```

- ...
- Konstruktion

```
var notw = new Book  
  ("The Name of the Wind",  
  "Patrick Rothfuss",  
  662);
```

- Modifikation

```
notw.author = "P. Rothfuss";
```

Veränderliche Arrays

- Arrays sind veränderlich
- Konstruktion

```
let primes = [| 2; 3; 5; 7; 11 |]
```

- Modifikation

```
primes.[2] ← 4711
```

- Arrays sind veränderlich
- Deklaration

```
int [] primes = {2, 3, 5, 7, 11};
```

- Modifikation

```
primes [2] = 4711;
```

Schleifen

- Schleifen

```
for i in 0..99 do
```

```
...
```

```
for i = 0 to 99 do
```

```
...
```

```
for i in 99..-1..0 do
```

```
...
```

```
for i = 99 downto 0 do
```

```
...
```

```
for b in books do
```

```
...
```

```
while i < 100 do
```

```
...
```

- Schleifen

```
for (int i = 0; i ≤ 99; i++) {
```

```
...
```

```
}
```

```
for (int i = 99; i ≥ 0; i--) {
```

```
...
```

```
}
```

```
for (Book b : books) {
```

```
...
```

```
}
```

```
while (i < 100) {
```

```
...
```

```
}
```

```
do {
```

```
...
```

```
} while (i < 100);
```

Java: Schleifen

- Javas **for**-Schleife

```
for (e1; e2; e3) {  
    ...  
}
```

ist eine Abkürzung für eine **while**-Schleife:

```
{  
    e1;  
    while (e2) {  
        ...  
        e3;  
    }  
}
```

- zum Beispiel

```
for (int i = 1; i < 1000; i *= 2)  
    printf ("%d\n", i);
```

Java: Schleifenabbruch

- mit **break**; kann die umschließende Schleife abgebrochen werden
- mit **continue**; kann mit der nächsten Iteration fortgefahren werden
- mit **return...**; kann die aktuelle Methode verlassen werden

```
static boolean palindrome (int [] a) {  
    for (int i = 0, j = a.length - 1; i < a.length / 2; i ++, j --)  
        if (a [i] != a [j])  
            return false;  
    return true;  
}
```

- (Kommaoperator: ';' für Ausdrücke)

Ausnahmen

- Ausnahmen sind Werte vom Typ *exn*

```
exception Zero
```

- Werfen und Fangen

```
try  
  let mutable acc = 1  
  for x in a do  
    if x = 0 then  
      raise Zero  
    else  
      acc ← acc * x  
  acc  
with  
  | Zero → 0
```

- Ausnahmen sind Objekte

```
class Zero extends Exception { }
```

- Werfen und Fangen

```
try {  
  int acc = 1;  
  for (int x : a)  
    if (x == 0)  
      throw new Zero ();  
  else  
    acc *= x;  
  return acc;  
} catch (Zero z) {  
  return 0;  
}
```


Java: Ausnahmen

In Java sind Ausnahmen *Objekte*, Instanzen der Klasse *Throwable* oder einer ihrer Unterklassen, insbesondere *Exception*.

```
public class Insufficient extends Exception {  
    public long n = 0;  
    Insufficient (long n) {  
        this.n = n;  
    }  
}
```

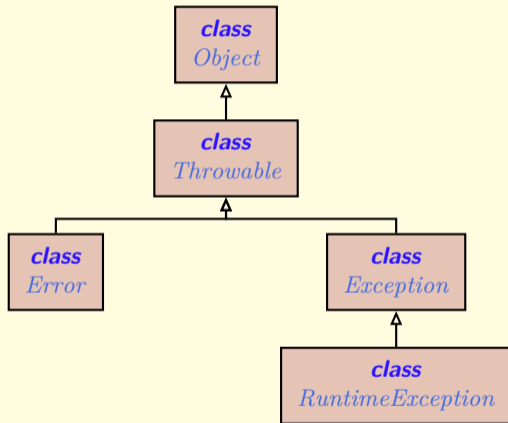
Java: Ausnahmen — “catch or specify”

Zur Schnittstelle einer Methode gehören neben den Parametern und dem Rückgabewert auch eine Liste der Ausnahmen, die möglicherweise in der Methode ausgelöst werden (“catch or specify”).

```
void withdraw (long amount) throws Insufficient
```

☞ Auf diese Weise wird zur Übersetzungszeit sichergestellt, dass alle Ausnahmen auch tatsächlich behandelt werden (checked exceptions).

Java: Ausnahmen — Klassenhierarchie



☞ Ausgenommen von der Prüfung sind lediglich *RuntimeExceptions* (unchecked exceptions).

Java: Ausnahmen — *catch*

```
TrustMe myaccount = new TrustMe ();  
try {  
    myaccount.deposit (100);  
    myaccount.withdraw (50);  
    myaccount.withdraw (60);  
    myaccount.deposit (40);  
    myaccount.withdraw (60);  
    System.out.println ("OK.");  
} catch (Insufficient e) {  
    System.out.println ("Insufficient funds!");  
    System.out.println ("  Can't withdraw " + e.n + "€.");  
}
```

- Alle Ausnahmen, die Instanz von *Insufficient* oder einer ihrer Unterklassen sind, werden von der **catch**-Klausel behandelt.
- Es können auch mehrere **catch**-Klauseln angegeben werden.

Java: Ausnahmen — *finally*

```
public static void dump (String path) throws IOException
{
    BufferedReader reader = null;
    try {
        reader = new BufferedReader (new FileReader (path));
        String line;
        while ((line = reader.readLine ()) != null)
            System.out.println (line);
    } catch (FileNotFoundException e) {
        System.err.println ("Cannot find file " + path);
        e.printStackTrace ();
    } finally {
        System.out.println ("Closing file " + path);
        if (reader != null) reader.close ();
    }
}
```

- *finally*-Block wird immer ausgeführt

Java: Ausnahmen — **try** mit Ressourcen

```
public static void dump (String path) throws IOException
{
    try (BufferedReader reader =
        new BufferedReader (new FileReader (path))) {
        String line;
        while ((line = reader.readLine ()) != null)
            System.out.println (line);
    } catch (FileNotFoundException exc) {
        System.err.println ("Cannot find file " + path);
        exc.printStackTrace ();
    }
}
```

- *reader* wird automatisch geschlossen
- Ressourcen implementieren die Schnittstelle *AutoCloseable* (**throws** *Exception*) oder ihre Unterschnittstelle *Closeable* (**throws** *IOException*)