

Lambda Ausdrücke und funktionale Programmierung

Marcel Lüthi
Departement Mathematik und Informatik

Agenda

- Geschichte: Objektorientierte und Funktionale Programmierung
- Funktionen als Objekte
- Lambda Ausdrücke in Java
- Funktionsobjekte in Java Standardbibliothek

Geschichte: Objektorientierte und Funktionale Programmierung

Erste Programmierung

```
assembly
SUMDIGIN CSECT
    USING  SUMDIGIN,R13
    B      72(R15)
    DC     17F'0'
    STM    R14,R12,12(R13)
    ST     R13,4(R15)
    ST     R15,8(R13)
    LR    R13,R15
    LA    R11,NUMBERS
    LA    R8,1
LOOPK   CH    R8,=H'4'
        BH    ELOOPK
        SR    R10,R10
        LA    R7,1
LOOPJ   CH    R7,=H'8'
        BH    ELOOPJ
        LR    R4,R11
        BCTR  R4,0
        AR    R4,R7
        MVC   D,0(R4)
```

Erste Hochsprachen

```
Program SumOfDigits;

function SumOfDigitBase(n:UInt64;base:LongWord): LongWord;
var
  tmp: UInt64;
  digit,sum : LongWord;
Begin
  digit := 0;
  sum   := 0;
  While n > 0 do
    Begin
      tmp := n div base;
      digit := n-base*tmp;
      n := tmp;
      inc(sum,digit);
    end;
  SumOfDigitBase := sum;
end;
Begin
  writeln(' 1 sums to ', SumOfDigitBase(1,10));
  writeln('1234 sums to ', SumOfDigitBase(1234,10));
  writeln('$FE sums to ', SumOfDigitBase($FE,16));
  writeln('$F0E sums to ', SumOfDigitBase($F0E,16));

  writeln('18446744073709551615 sums to ', SumOfDigitBase(High(UInt64),10));
end.
```

Wichtige Frage

Wie kann man Programme besser strukturieren?

Funktionale Programmierung

- Idee: Komposition von (mathematischen) Funktionen um aus einfachen Teilen komplexe Funktionalität zu bauen
- Mathematische Grundlage: Lambdakalkül
- Aktionen / Berechnungen im Zentrum

Objektorientierte Programmierung

- Idee: Organisation von Code in "selbstorganisierende" Module (Objekte)
- Management von Zustand durch Kapselung
- Objekte im Zentrum

Konzepte entwickelt in 60 und 70er Jahren

Funktionale Konstrukte in Java

*Moderne Programmiersprachen integrieren Konzepte von
Funktionalen Sprachen:*

- Funktionen als Argumente
- Anonyme Funktionen
- (Closures)

Funktionen und Objekte

Funktionsobjekte

Idee: Funktionen sind (seiteneffektfreie) Objekte mit nur einer Methode

Funktionsobjekte: Implementationsstrategie

1. Deklaration: Interface für Funktionen definieren

```
interface Function {  
    int apply(int x);  
}
```

2. Definition der Funktion: Anonymes Objekt erstellen

```
Function square = new Function() {  
    public int apply(int x) { return x * x; }  
}
```

Diskutieren Sie:

- Wie viele verschiedene Interfaces für Funktionen brauchen wir, wenn wir alle Kombinationen von Funktion $T \rightarrow R$ implementieren wollen, wobei T und R jeweils `String`, `Integer` und `Double` sein können?
- Wie könnten wir elegant ein allgemeines Funktionsinterface definieren?

Generische Funktionsobjekte

- Java Generics helfen uns die Funktion nur einmal zu definieren

```
In [40]: interface Function<T, R> {
    R apply(T x);
}
```

Beispielanwendung

```
In [44]: Function<Double, Double> square = new Function<>() {
    public Double apply(Double x) {
        return x * x;
    }
}
```

Anwendungsbeispiel: Transformation von Listenelementen

Gegeben: Liste von Zahlen

```
In [49]: LinkedList<Double> numbers0To10 = new LinkedList<>();  
for (int i = 0; i < 10; i++) {  
    numbers0To10.add(new Double(i));  
}
```

Aufgabe: Führe mathematische Funktion auf Elementen aus

- Bestehende Listenelement dürfen nicht verändert werden
- Es soll neue Liste ausgegeben werden

Lösung: Die map Methode

```
In [51]: static LinkedList<Double> map(LinkedList<Double> list, Function<Double, Double> f) {  
    LinkedList<Double> newList = new LinkedList<>();  
    for (Double v : list) {  
        newList.add(f.apply(v));  
    }  
    return newList;  
}
```

Lösung: Die map Methode

```
In [51]: static LinkedList<Double> map(LinkedList<Double> list, Function<Double, Double> f) {  
    LinkedList<Double> newList = new LinkedList<>();  
    for (Double v : list) {  
        newList.add(f.apply(v));  
    }  
    return newList;  
}
```

Anwendung

```
In [53]: LinkedList<Double> newList = map(numbers0To10, square);  
System.out.println(newList);
```

```
[0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0]
```

Übung

- Können Sie die map Methode so umschreiben, dass als Ausgabe nicht mehr ein Double verlangt wird, sondern ein beliebiger Typ angegeben werden kann?
- Können Sie die map Methode so umschreiben, dass diese nicht mehr nur auf Listen von Double, sondern allgemeinen Listen arbeitet?

```
In [50]: // Ihr Code
```

Lambda Ausdrücke

Lambda Ausdrücke

- Java hat eine spezielle Syntax definiert um Funktionsobjekte zu erstellen.
- Bekannt als lambda Ausdrücke

Parameter -> Ausdruck

Beispiel

`x -> x * x`

Lambda Syntax

```
lambda = ArgList "->" Body
ArgList = Identifier
    | "(" [Type] Identifier { "," [Type] Identifier } ")"
    | "()"
Body = Expression    |  "{" [ Statement ";" ]+ "}"
```

Functional Interface

- Ein *Functional Interface* ist ein Interface oder Abstrakte Klasse mit genau eine Methode
 - Methode entspricht "Berechnung" der Funktion

Beispiel:

```
interface Function<T, R> {  
    R apply(T t);  
}
```

Functional Interface

- Ein *Functional Interface* ist ein Interface oder Abstrakte Klasse mit genau eine Methode
 - Methode entspricht "Berechnung" der Funktion

Beispiel:

```
interface Function<T, R> {  
    R apply(T t);  
}
```

- Lambda Ausdrücke können an ein *Functional Interface* zugewiesen werden.
 - Lambdas bekommen einen Namen

```
Function<Double, Double> f = (Double d) -> d * d;
```

Lambdas als Methodenargumente

- Erlaubt einfache Funktionen mit wenig Code zu erstellen

Beispiel

```
In [25]: LinkedList<Double> ll = new LinkedList<>();
for (int i = 0; i < 10; i++) {
    ll.add(new Double(i));
}

// oben definierte Map Methode jetzt mit Lambdas
map(ll, (Double d) -> d * 2);
```

```
Out[25]: [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0]
```

Lambdas als Rückgabewerte

- Methoden können Funktionen zurückgeben.

```
Function<Double, Double> doubleFun(Double x) {  
    return x -> 2 * x;  
}
```

Übung:

- Schreiben Sie eine Methode `applyTwice`, die eine Funktion f vom Typ `Function<Double, Double>` als Argument nimmt, und eine Funktion zurückgibt, die diese zwei mal hintereinander anwendet, also der Funktion $x \mapsto f(f(x))$ entspricht
- Können Sie diese als generische Funktion schreiben?

In [54]: `// Ihr Code`

Lambdas mit Anweisungsblock

- Rechte Seite von Lambda Ausdruck kann beliebiger Block sein
 - Block muss den richtigen Typ zurückliefern

```
Function<T, R> f = (T t) -> {  
    Statement1;  
    Statement2;  
    return r; // r ist vom Type R  
}
```

Lambdas mit Anweisungsblock

Folgendes funktioniert:

```
In [55]: Function<String, Integer> f = (String s) -> {
    System.out.println(s);
    return Integer.parseInt(s);
};
f.apply("5");
```

5

```
Out[55]: 5
```

Lambdas mit Anweisungsblock

Folgendes funktioniert:

```
In [55]: Function<String, Integer> f = (String s) -> {
    System.out.println(s);
    return Integer.parseInt(s);
};
f.apply("5");
```

5

```
Out[55]: 5
```

Aber hier gibt es einen Typfehler

```
In [56]: Function<String, Integer> f = (String s) -> {
    System.out.println(s);
    return s;
}
```

```
|     return s;
incompatible types: bad return type in lambda expression
java.lang.String cannot be converted to java.lang.Integer
```

Methodenreferenzen

- Wir können Methoden Functional Interfaces zuweisen:

```
Function<Double, Double> f = AClass::aMethod;
```

Beispiel

```
In [36]: Function<Double, Double> cos = Math::cos;
map(numbers0To10, cos)
```

```
Out[36]: [1.0, 0.5403023058681398, -0.4161468365471424, -0.9899924966004454, -0.65364362086361
19, 0.28366218546322625, 0.960170286650366, 0.7539022543433046, -0.14550003380861354,
-0.9111302618846769]
```

Methodenreferenzen

- Wir können Methoden Functional Interfaces zuweisen:

```
Function<Double, Double> f = AClass::aMethod;
```

Beispiel

```
In [36]: Function<Double, Double> cos = Math::cos;
map(numbers0To10, cos)
```

```
Out[36]: [1.0, 0.5403023058681398, -0.4161468365471424, -0.9899924966004454, -0.65364362086361
19, 0.28366218546322625, 0.960170286650366, 0.7539022543433046, -0.14550003380861354,
-0.9111302618846769]
```

oder kürzer

```
In [57]: map(numbers0To10, Math::cos)
```

```
Out[57]: [1.0, 0.5403023058681398, -0.4161468365471424, -0.9899924966004454, -0.65364362086361
19, 0.28366218546322625, 0.960170286650366, 0.7539022543433046, -0.14550003380861354,
-0.9111302618846769]
```

Funktionsobjekte mit mehreren Argumenten

- Idee funktioniert für Funktionen mit beliebig vielen Argumenten

```
In [58]: interface Function2<S, T, R> {
    R apply(S s, T t);
}

Function2<Double, Double, Double> sum = (x, y) -> x + y;

Function2<Double, Double, String> sumAsString = (x, y) -> new Double(x + y).toString();

sumAsString.apply(3.0, 5.0)
```

Out[58]: 8.0

Prädikate

Prädikat: Ein Funktionsobjekt das True oder False zurückgibt

```
interface Predicate<T>
    boolean test(T x);
}
```

Prädikat: Beispiel

```
In [38]: interface Predicate<T> {
    boolean test(T x);
}

double[] array = {0.1, 0.7, -0.5, 1.0};

// returns the number of elements, which have the property specified by Predicate pred
int count(double[] array, Predicate<Double> pred) {

    int counter = 0;

    for (int i = 0; i < array.length; ++i) {
        if (pred.test(array[i]) == true) {
            counter += 1;
        }
    }

    return counter;
}

count(array, x -> x > 0.0);
```

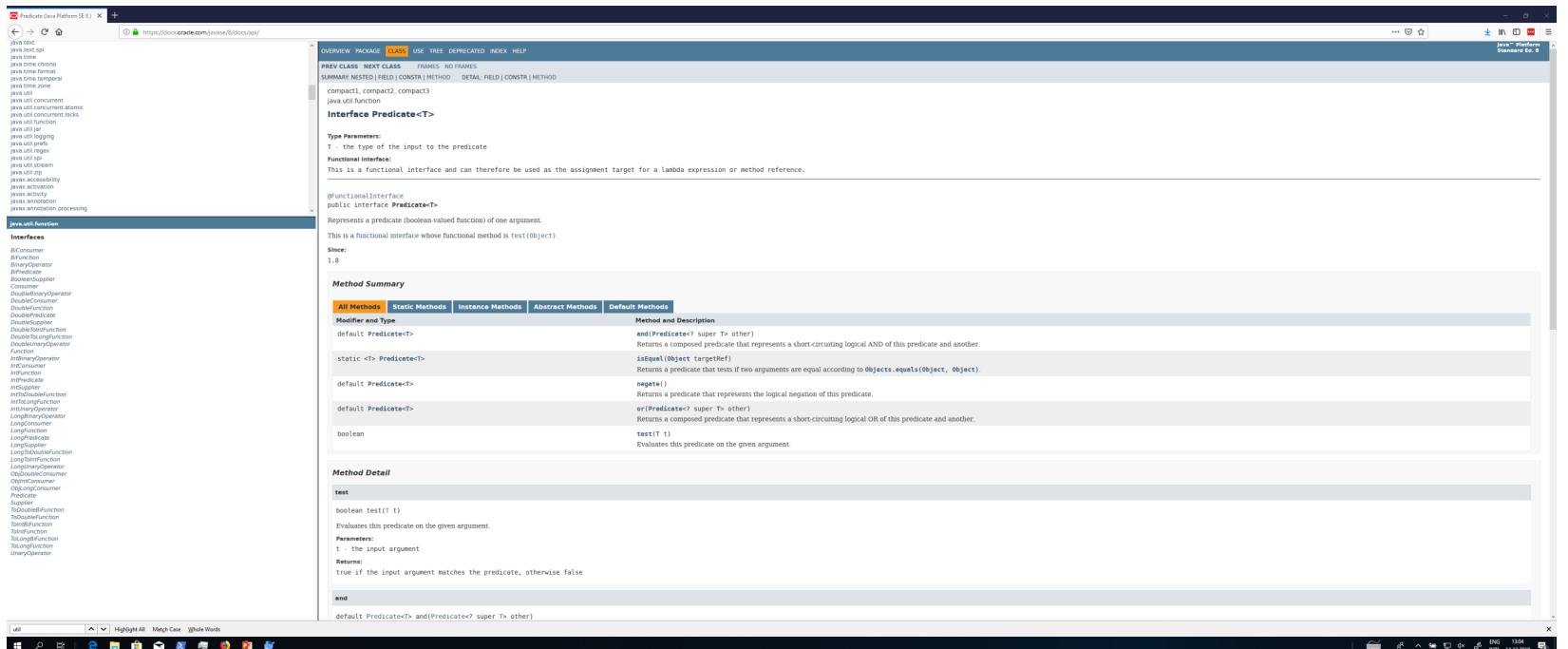
Out[38]: 3

Übung:

- Implementieren Sie die Methode
- Testen Sie diese mit verschiedenen Prädikaten

Funktionsobjekte in der Java Standardbibliothek

- Standard Funktionstypen sind in Java API Dokumentation (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html>) definiert:



The screenshot shows the Java API documentation for the `Predicate` interface. The URL is <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java.util.function/Predicate.html>. The page title is "Predicate - Java Platform SE 11". The left sidebar lists various Java packages and interfaces, including `java.util.function`, `java.util.function.Consumer`, `java.util.function.BiConsumer`, `java.util.function.BinaryOperator`, `java.util.function.Predicate`, `java.util.function.Supplier`, `java.util.function.Consumer`, `DoubleUnaryOperator`, `DoubleBiConsumer`, `DoubleBinaryOperator`, `DoublePredicate`, `DoubleFunction`, `DoubleSupplier`, `DoubleUnaryFunction`, `DoubleBiFunction`, `DoubleBinaryFunction`, `Function`, `Supplier`, `UnaryOperator`, `IntConsumer`, `IntFunction`, `IntPredicate`, `IntSupplier`, `IntUnaryFunction`, `LongConsumer`, `LongFunction`, `LongPredicate`, `LongSupplier`, `LongUnaryFunction`, `LongBiFunction`, `LongBinaryFunction`, `ObjDoubleConsumer`, `ObjDoubleFunction`, `ObjLongConsumer`, `ObjLongFunction`, `Supplier`, `SupplierBiFunction`, `SupplierFunction`, `SupplierSupplier`, `SupplierUnaryFunction`, and `UnaryOperator`.

The main content area shows the `Predicate` interface definition:

```

Interface Predicate<T>

Type Parameters:
T - the type of the input to the predicate

Functional Interface:
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

Annotations:
@FunctionalInterface
public interface Predicate<T>

Description:
Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is test(Object).

Since:
1.8

Method Summary

All Methods Static Methods Instance Methods Abstract Methods Default Methods

Modifier and Type Method and Description
default Predicate<T>
and(Predicate<T> super T> other)
Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

static <T> Predicate<T>
isEqual(Object targetRef)
Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

default Predicate<T>
negate()
Returns a predicate that represents the logical negation of this predicate.

default Predicate<T>
or(Predicate<T> super T> other)
Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

boolean
test(T t)
Evaluates this predicate on the given argument.

Method Detail

test
boolean test(T t)
Evaluates this predicate on the given argument.

Parameters:
t - the input argument
Returns:
true if the input argument matches the predicate, otherwise false

and
default Predicate<T> and(Predicate<T> super T> other)

```

Sourcecode vom Java Funktionsinterface

```
In [59]: public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Komposition und Identität

Übung:

- Versuchen Sie verschiedene Funktionen mit compose zu kombinieren

Sourcecode vom Java Predicate

```
In [60]: public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (+) -> !test(+);
    }
}
```

