TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 04c:
Funktionales Programmieren: Funktionen als Daten

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Funktionen als Daten in Java

Bis jetzt hatten wir Daten und Funktionen. Ein grundsätzliches Konzept funktionaler Abstraktion ist aber, dass Funktionen im Grunde auch nichts anderes als Daten sind und so auch behandelt werden können. Das gibt es auch in Java, haben wir sogar schon in Kapitel 04a gesehen, schauen wir uns in diesem Abschnitt noch einmal aus diesem neuen Blickwinkel heraus an.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct
    = new QuadraticFunction ( 3.3, 5.5, 7.7 );


double y = fct.apply ( x );
```

Der Weg in Java dazu geht über Interfaces wie dieses, das wir schon in Kapitel 03b, Abschnitt zu Interfaces, gesehen haben. Die eigentliche Funktion ist eine Methode eines Objektes einer Klasse, die das Interface implementiert.

In Anlehnung an die Interfaces in java.util.function heißt die Methode auch hier apply. Wie der Name DoubleToDoubleFunction nahelegt, hat apply einen Parameter vom Typ double und liefert auch double zurück.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private DoubleToDoubleFunction fct;
   public X ( DoubleToDoubleFunction fct ) {
      this.fct = fct;
   }
   public DoubleToDoubleFunction getFct () {
      return fct;
  }
}
```

Wie andere Daten auch, können Funktionen in Form von Interfaces ganz normal Attribute von Klassen sowie Parameter und Rückgaben von Methoden sein wie in diesem kleinen Beispiel.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction[ ] functions
    = new DoubleToDoubleFunction [ 5 ];

functions[0] = new QuadraticFunction ( 3.3, 5.5 7.7 );
.....


functions[0].apply ( 123.123 );
```

Und natürlich kann es mit Interfaces auch Arrays von Funktionen geben.

TECHNISCHE UNIVERSITÄT DARMSTADT

Funktionen als Daten in Racket

Nun sehen wir uns das Thema in Racket an. Wie bei allen funktionalen Konstrukten, ist auch dieses in Racket eingebaut und daher einfacher als in Java gelöst.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

In Java ist das alles ziemlich umständlich, weil man immer über Interfaces gehen muss. Jetzt schauen wir uns an, wie einfach das in Racket geht.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )   ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Und wir beginnen konkret mit der ganz normalen Definition einer Konstanten namens add. Allerdings ist der definierende Ausdruck neuen Typs, nämlich eine Funktion. Die Konstante add ist also nicht wie bisher vom Typ Zahl oder String oder von einem Struct-Typ, ...

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;; number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

... sondern von einem Funktionstyp, konkret von dem Typ aller Funktionen, die zwei Zahlen als Parameter haben und eine Zahl zurückliefern. Also genau das, was wir schon vorher, bei der Definition von Funktionen, als Funktionstyp bezeichnet und unter dem Stichwort „Type“ in den Kommentar zur Funktion geschrieben haben.

***Erinnerung*: Wir hatten schon in Kapitel 04a bei der Behandlung von Identifiern gesagt, dass auch Operatoren in Racket nichts anderes als Namen von Funktionen sind, dass also zwischen Operatoren und Funktionen überhaupt nicht unterschieden wird. Das entspricht ja auch den Regeln für die Bildung von Identifiern: Das einzelne Pluszeichen ist ein erlaubter Identifier, nur eben vordefiniert als Name für die Addition.**

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Da add also eine Funktion mit zwei Zahlenparametern *ist*, können wir add auch so verwenden, also mit zwei Zahlen als Parameterwerten aufrufen. Das Ergebnis ist natürlich 5.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Wir können Funktionen dann natürlich auch als Attribute von Structs hernehmen. Dazu ein beispielhafter Struct-Typ mit zwei Attributen, deren Namen schon andeuten, dass sie dafür gedacht sind, Funktionen aufzunehmen.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Der erste Struct vom Struct-Typ functions enthält die gerade definierte Funktion add als Attribut fct1 und die vordefinierte Multiplikation als fct2.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Und der zweite Struct enthält zwei vordefinierte Funktionen, Subtraktion und Division.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Auf die beiden Attribute von Struct a können wir genau so zugreifen wie auf Struct-Attribute von anderen Typen. Da die beiden Attribute aber jeweils eine Funktion mit zwei zahlenwertigen Parametern sind, können sie nur als solche verwendet werden, also als Funktionsaufruf mit zwei Zahlen als Parametern.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Dasselbe geht selbstverständlich auch für die beiden Funktionen von Struct b.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define add + )    ;;  number number -> number
( add 2 3 )
( define-struct functions ( fct1 fct2 ) )
( define a ( make-functions add * ) )
( define b ( make-functions – / ) )
( ( functions-fct1 a ) 2 ( ( functions-fct2 a ) 3 4 ) )
( ( functions-fct1 b ) 9 ( ( functions-fct2 b ) 7 3 ) )
( ( functions-fct2 b ) 5 ( ( functions-fct1 a ) 2 3 ) )
```

Und natürlich ist man nicht darauf festgelegt, alle Funktionen aus a oder alle Funktionen aus b zu nehmen.

Funktionen als Daten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( fct1 n ) ..... )
( define ( fct2 x y ) ..... )
( define ( fct3 a b c ) ..... )

( list fct1 3.14 + fct2 “Hallo“ / fct3 )
```

Natürlich können Funktionen genauso gut auch als Listenelemente verwendet werden, vordefinierte wie Operator + im Beispiel, genauso auch selbstdefinierte wie fct1, fct2 und fct3.

TECHNISCHE UNIVERSITÄT DARMSTADT

Funktionen höherer Ordnung

Wir haben nun gesehen, dass man Konstanten, Struct-Attribute und Listenelemente von Funktionstypen haben kann. Jetzt gehen wir einen Schritt weiter und betrachten Funktionen, die ihrerseits erwarten, dass ihre Parameter von Funktionstypen sind, beziehungsweise die selbst Funktionen zurückliefern.

Funktionen, die Funktionen als Parameter oder Rückgabe haben, bezeichnet man als Funktionen höherer Ordnung. Sie sind ein grundlegendes Konzept der funktionalen Abstraktion.

Funktionen höherer Ordnung


```
// Precondition: a < b, f.apply(a) * f.apply(b) <= 0, epsilon > 0, and
//         the mathematical function represented by f is continuous
public static double findZero ( double a, double b, double epsilon,
                                DoubleToDoubleFunction f ) {
   double m = ( b – a ) / 2;
   if ( m – a < epsilon )
      return m;
   if ( f.apply(a) * f.apply(m) >= 0 )
      return findZero ( m, b, epsilon, f );
   return findZero ( a, m, epsilon, f );
}
```

In Java hatten wir dieses Thema schon weiter vorne in diesem Kapitel in einem Beispiel gesehen, hier noch einmal exakt dasselbe Beispiel. Das ist in Java der Weg, wie man eine Funktion zu einem Parameter einer Methode macht: Ein Interface wird übergeben, und eine der Methoden implementiert dann die eigentliche Funktion.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static DoubleToDoubleFunction
                                standardQuadraticFunction () {
   return new QuadraticFunction ( 1, 0, 0 );
}
```

Funktionen können mit diesem Konstrukt nicht nur Parameter, sondern auch Rückgaben von Funktionen sein, wie Sie an diesem kleinen Beispiel noch einmal sehen.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( number -> number ) number
;;           ( number -> number ) number
;;                    -> number
;; Returns: fct1 applied to x plus fct2 applied to y

( define ( add-fct-results fct1 x fct2 y )
    ( + ( fct1 x ) ( fct2 y ) ) )

( check-expect ( add-fct-results sqrt 9 sqr 6 ) 39 )
```

Kommen wir nun zu einem Beispiel in Racket. Wieder ist in Racket alles viel einfacher. Diese Funktion namens add-fct-results hat insgesamt vier Parameter, zwei Funktionen und zwei Zahlen, immer abwechselnd, und liefert eine Zahl zurück.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( number -> number ) number
;;         ( number -> number ) number
;;                 -> number
;; Returns: fct1 applied to x plus fct2 applied to y

( define ( add-fct-results fct1 x fct2 y )
    ( + ( fct1 x ) ( fct2 y ) ) )

( check-expect ( add-fct-results sqrt 9 sqr 6 ) 39 )
```

So wie hier gezeigt ist es üblich, Parameter von einem Funktionstyp zu beschreiben, eben ein Funktionstyp für sich, zur klareren Trennung von den anderen Parametern wieder in Klammern ähnlich wie bei Listen. Der Typ dieser beiden Parameter ist also derselbe: Funktion, die eine Zahl als Parameter hat und eine Zahl zurückliefert.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( number -> number ) number
;;          ( number -> number ) number
;;                   -> number
;; Returns: fct1 applied to x plus fct2 applied to y

( define ( add-fct-results fct1 x fct2 y )
    ( + ( fct1 x ) ( fct2 y ) ) )

( check-expect ( add-fct-results sqrt 9 sqr 6 ) 39 )
```

Die Parameter fct1 und fct2 werden in der Funktionen add-fct-results gemäß ihrem erwarteten Typ verwendet: Sie werden jeweils mit einem einzelnen Wert aufgerufen, und die Rückgabe beider Funktionen wird als Zahl weiterverarbeitet.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( number -> number ) number
;;           ( number -> number ) number
;;                    -> number
;; Returns: fct1 applied to x plus fct2 applied to y

( define ( add-fct-results fct1 x fct2 y )
    ( + ( fct1 x ) ( fct2 y ) ) )

( check-expect ( add-fct-results sqrt 9 sqr 6 ) 39 )
```

Bei den anderen beiden Parametern erwartet die Funktion add-fct-results, dass sie Zahlen sind. Das ist auch notwendig, denn die beiden Funktionen fct1 und fct2 sollen ja Zahlen als Parameter haben, das muss natürlich zusammenpassen.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: X -> X
;; Returns: the parameter value

( define ( my-identity x ) x )

( my-identity sqrt )

( ( my-identity sqrt  ) 5 )
```

Als nächstes ein extrem einfaches Beispiel für eine Funktion, die eine Funktion zurückliefert.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: X -> X
;; Returns: the parameter value
( define ( my-identity x ) x )

( my-identity sqrt )

( ( my-identity sqrt  ) 5 )
```

Die Rückgabe ist einfach das x, das als Parameter übergeben wird.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: X -> X
;; Returns: the parameter value

( define ( my-identity x ) x )

( my-identity sqrt )

( ( my-identity sqrt  ) 5 )
```

Das ist neu: X ist kein Datentyp, sondern Platzhalter für einen beliebigen Datentyp. Da nach Konvention immer nur Kleinbuchstaben für alle Identifier verwendet werden, können wir Großbuchstaben problemlos ohne Namenskonflikte so verwenden. Die Aussage dieser Type-Klausel ist: Es ist egal, von welchem Typ der Parameter ist, die Rückgabe ist aber garantiert vom selben Typ.

***Erinnerung*: In Kapitel 04b haben wir schon ANY gesehen, womit ausgedruckt wird, dass der Typ beliebig ist. Der Unterschied ist: Bei einer Funktionstyp ANY -> ANY kann der Ergebnistyp ein völlig anderer als der Eingabetyp sein, bei X -> X hingegen sind beide identisch.**

***Vorgriff*: Wir werden später (Fallbeispiele map und fold) sehen, dass die unbekannten Typen nicht immer dieselben sein müssen, dann werden wir beispielsweise X, Y und Z zur Unterscheidung verwenden.**

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: X -> X
;; Returns: the parameter value

( define ( my-identity x ) x )

( my-identity sqrt )

( ( my-identity sqrt  ) 5 )
```

Da es in Racket auch eine Funktion namens identity schon vordefiniert gibt, die übrigens genau dasselbe macht, müssen wir unsere Funktion hier wieder leicht anders nennen.

***Nebenbemerkung*: Warum gibt es eine Funktion identity mit einem Parameter in Racket, die einfach nur ihren Parameter zurückliefert? Weil es Situationen gibt, in denen man eine beliebige Funktion einsetzen möchte, die einen Parameter von einem Typ X hat und auch ein X zurückliefert so wie Methode apply vom Typ DoubleToDoubleFunction. Aber manchmal möchte man in einer solchen Situation einfach nur einen bestimmten Wert haben, muss aber eine Funktion einsetzen, weil eben eine Funktion dort erwartet wird. Dann kann man die Identitätsfunktion mit diesem Wert aufrufen. Die Identitätsfunktion kann also Werte von X überall dort ins Spiel, wo Funktionen X ->X erwartet werden.**

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: X -> X
;; Returns: the parameter value

( define ( my-identity x ) x )

( my-identity sqrt )

( ( my-identity sqrt  ) 5 )
```

Der Wert des farblich unterlegten Ausdrucks ist die Funktion sqrt, der Name der Funktion wird von DrRacket ausgegeben (übrigens auch wenn Sie nur sqrt allein als atomaren Ausdruck in DrRacket eingeben).

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: X -> X
;; Returns: the parameter value

( define ( my-identity x ) x )


( my-identity sqrt )


( ( my-identity sqrt  ) 5 )
```

Da die Rückgabe dieses Aufrufs von my-identity eine Funktion mit einem Zahlenparameter ist, kann die Rückgabe auch gleich auf eine Zahl angewendet werden.

Damit haben wir prinzipiell gesehen, was Funktion höherer Ordnung sind; darauf bauen wir im Folgenden auf.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( real -> real ) real real -> real
( define ( find-zero f a b )
   ( local
     ( ( define m ( / ( + a b ) 2 ) ) )
     ( cond
          [ ( < ( − b a ) epsilon ) m ]
          [  ( > ( * ( f a ) ( f m ) ) 0 ) ( find-zero f m b ) ]
          [ else ( find-zero f a m ) ] ) ) )
```

Wir hatten es nicht erwähnt, aber wir hatten schon vorher in Kapitel 04a, Abschnitt zu Rekursion in Java, ein Beispiel für eine Funktion höherer Ordnung in Java gesehen, da ja die stetige Funktion, von der eine Nullstelle zu berechnen ist, ein Parameter von findZero war. Das geht auch in Racket, nur sieht das sicherlich insgesamt auch hier wieder einfacher aus als in Java.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( real -> real ) real real -> real
( define ( find-zero f a b )
   ( local
     ( ( define m ( / ( + a b ) 2 ) ) )
     ( cond
          [ ( < ( − b a ) epsilon ) m ]
          [  ( > ( * ( f a ) ( f m ) ) 0 ) ( find-zero f m b ) ]
          [ else ( find-zero f a m ) ] ) ) )
```

Wie bei findZero in Java, ist auch bei find-zero in Racket der erste Parameter von einem Funktionstyp, nämlich Funktionen, die reelle Zahlen auf reelle Zahlen abbilden.

Funktionen höherer Ordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( real -> real ) real real -> real
( define ( find-zero f a b )
   ( local
     ( ( define m ( / ( + a b ) 2 ) ) )
     ( cond
          [ ( < ( − b a ) epsilon ) m ]
          [  ( > ( * ( f a ) ( f m ) ) 0 ) ( find-zero f m b ) ]
          [ else ( find-zero f a m ) ] ) ) )
```

Wie üblich in Racket, besteht der Aufruf auch dieser Funktion aus dem Namen der Funktion gefolgt von den aktualen Parametern, nur getrennt durch Whitespaces, und das Ganze in Klammern.

TECHNISCHE UNIVERSITÄT DARMSTADT

Functional Interfaces und Lambda-Ausdrücke in Java

Java Oracle Tutorial: Lambda Expressions

Wir kommen nun zu einem weiteren Kerngedanken der funktionalen Programmierung: Wenn Funktionen ganz normale Daten sind, dann sollte es zu Funktionstypen auch Literale geben, so wie es ja auch für Zahlentypen, Typ String und ein paar weitere Typen jeweils Literale gibt. Aus historischen Gründen werden Literale von Funktionstypen Lambda-Ausdrücke genannt.

In Java gibt es ja eine enge Verknüpfung zwischen Interfaces und Funktionen als Daten. Um in Java zu Lambda-Ausdrücken zu kommen, müssen wir dafür noch eine spezielle Art von Interfaces beleuchten: Functional Interfaces.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Interface kann haben:

- **Nicht implementierte Objektmethoden**
  - **Implizit public**
- **Implementierte Klassenmethoden**
- **Klassenkonstanten**
  - **Implizit public und final**
- **Implementierte Objektmethoden in Form von Default-Methoden**

So, wie Sie Interfaces in Kapitel 03b kennengelernt haben, waren Interfaces ursprünglich in Java konzipiert. Im Laufe der Zeit wurden Interfaces aus verschiedenen pragmatischen Gründen weiterentwickelt, so dass sie inzwischen noch etliche andere Bestandteile enthalten können, die ursprünglich nicht so gedacht waren. Die Unterschiede zwischen Interfaces und abstrakten Klassen sind dadurch ziemlich verwischt, wir arbeiten die verbliebenen Unterschiede gleich noch auf.

Erst einmal mehr zu Interfaces

Ein Interface kann haben:

- **Nicht implementierte Objektmethoden**
  - **Implizit public**
- **Implementierte Klassenmethoden**
- **Klassenkonstanten**
  - **Implizit public und final**
- **Implementierte Objektmethoden in Form von Default-Methoden**

Das ist die Art von Bestandteilen von Interfaces, die wir bisher gesehen und diskutiert haben: Methoden werden nur *definiert*, nicht *implementiert*. Sie können, müssen aber nicht explizit public deklariert sein. Sind sie es nicht, dann sind sie trotzdem public.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Interface kann haben:

- **Nicht implementierte Objektmethoden**
  - **Implizit public**
- **Implementierte Klassenmethoden**
- **Klassenkonstanten**
  - **Implizit public und final**
- **Implementierte Objektmethoden in Form von Default-Methoden**

Wir nehmen zur Kenntnis, dass Klassenmethoden nicht nur definiert, sondern auch implementiert sein dürfen.
***Erinnerung*: Wir haben in Kapitel 03c, Abschnitt „Signatur und Überschreiben / Überladen von**

Methoden“, gesehen, dass bei Klassenmethoden – im Gegensatz zu Objektmethoden – der *statische* Typ darüber entscheidet, welche Implementation verwendet wird, wenn die Methode im Basistyp und in einem direkten oder indirekten Subtyp jeweils implementiert ist.

***Nebenbemerkung*: Aus diesem Grund gibt es bei Klassenmethoden nicht die Probleme, die die Entwickler von Java dazu bewogen haben, Mehrfachvererbung nur leichtgewichtig mit Interfaces statt mit Klassen zu realisieren. Daher spricht überhaupt nichts dagegen, Klassenmethoden mit Implementation in Interfaces zu haben.**

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Interface kann haben:

- **Nicht implementierte Objektmethoden**
  - **Implizit public**
- **Implementierte Klassenmethoden**
- **Klassenkonstanten**
  - **Implizit public und final**
- **Implementierte Objektmethoden in Form von Default-Methoden**

Auch Attribute sind in engen Grenzen in Interfaces möglich: Klassenkonstanten ja, aber sonst keine Art von Attribut. Auch wenn man public beziehungsweise final weglässt, wird jedes Attribut implizit public und final.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Interface kann haben:

- **Nicht implementierte Objektmethoden**
  - **Implizit public**
- **Implementierte Klassenmethoden**
- **Klassenkonstanten**
  - **Implizit public und final**
- **Implementierte Objektmethoden in Form von Default-Methoden**

Schlussendlich hat man auch noch akzeptiert, dass implementierte Objektmethoden aus pragmatischen Gründen wünschenswert sind. Diese nennt man Default-Methoden, und sie werden auch mit dem neuen Schlüsselwort default eingeleitet.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Int {
  int N = 1;
  void m1 ();
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Ein kleines Beispiel zur Illustration.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Int {
  int N = 1;
  void m1 ();
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Auch wenn wie hier weder public noch static noch final dabei steht: Jedes Attribut ist implizit public, static und final, also eine öffentliche Klassenkonstante.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Int {
  int N = 1;
  void m1 ();
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Die Art von Methoden, die wir bisher gesehen haben: nichtimplementierte Objektmethoden, ebenfalls implizit public.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Int {
  int N = 1;
  void m1 ();
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Auch Klassenmethoden sind implizit public.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Int {
  int N = 1;
  void m1 ();
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Das ist nun ein Beispiel für Default-Methoden. Das sind ganz normale Objektmethoden, die auch ganz normal implementiert werden. Aber in Interfaces muss das Schlüsselwort default vorab stehen; public ist wieder implizit.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Int1 {
  default void m () { ..... }
}

public interface Int2 {
  default void m () { ..... }
}

public class X implements Int1, Int2 { ..... }
```

Wie ist es nun sichergestellt, dass Default-Methoden kein Problem bei Mehrfachvererbung von Interfaces verursachen? Probleme bei Mehrfachvererbung gibt es insbesondere dann, wenn dieselbe Methode mit zwei unterschiedlichen Implementationen von zwei verschiedenen Supertypen an dieselbe Klasse vererbt werden. Das wäre bei Default-Methoden von Interfaces prinzipiell möglich. Man hat aber bei der Einführung von Default-Methoden in Java entschieden, dass der Compiler in dem Fall, dass eine Klasse zwei Implementationen derselben Methode erbt, eine Fehlermeldung gibt und die Übersetzung verweigert.

Um das problem zu lösen, muss man Methode m in X implementieren, dann ist wieder alles unzweideutig.

Erst einmal mehr zu Interfaces

Unterschiede Interface / abstrakte Klasse:

- **Interfaces können Mehrfachvererbung.**
- **Abstrakte Klassen können**
  - **von Klassen abgeleitet werden;**
  - **Attribute und Methoden haben, die nicht public sind.**

Es fehlt noch die angekündigte Diskussion, was Interfaces und abstrakte Klassen überhaupt noch voneinander unterscheidet.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterschiede Interface / abstrakte Klasse:

- **Interfaces können Mehrfachvererbung.**
- **Abstrakte Klassen können**
  - **von Klassen abgeleitet werden;**
  - **Attribute und Methoden haben, die nicht public sind.**

Mehrfachvererbung war von Anfang an die Motivation, Interfaces einzuführen, und das ist bis heute der eine Punkt, den Interfaces Klassen voraushaben.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterschiede Interface / abstrakte Klasse:

- **Interfaces können Mehrfachvererbung.**
- **Abstrakte Klassen können**
  - **von Klassen abgeleitet werden;**
  - **Attribute und Methoden haben, die nicht public sind.**

Interfaces können andere Interfaces erweitern; von Klassen abgeleitet werden können sie nicht.

Erst einmal mehr zu Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterschiede Interface / abstrakte Klasse:

- **Interfaces können Mehrfachvererbung.**
- **Abstrakte Klassen können**
  - **von Klassen abgeleitet werden;**
  - **Attribute und Methoden haben, die nicht public sind.**

Wie wir gesehen haben, ist in Interfaces alles implizit public. Das ist bei abstrakten Klassen nicht der Fall.

Functional Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

- Was ist das: ein Interface, bei dem genau eine Methode weder default oder static ist
- Diese Methode heißt die *funktionale Methode* dieses Functional interface

```
public interface Int {
  int N = 1;
  void m1 ();   //  the functional method of Interface Int
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Nun können wir leicht klären, was ein funktionales Interface ist, nämlich ein Interface wie Int unten, dass wir vor ein paar Folien schon einmal gesehen hatten.

Functional Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Was ist das: ein Interface, bei dem genau eine Methode weder default oder static ist**
- **Diese Methode heißt die *funktionale Methode* dieses Functional interface**

```
public interface Int {
  int N = 1;
  void m1 ();   //  the functional method of Interface Int
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Der Begriff Functional Interface ist sehr simpel durch diese kleine Einschränkung bestimmt. Wir werden gleich sehen, dass diese Einschränkung hochgradig zweckmäßig ist.

Functional Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

- Was ist das: ein Interface, bei dem genau eine Methode weder default oder static ist
- Diese Methode heißt die *funktionale Methode* dieses Functional interface

```
public interface Int {
  int N = 1;
  void m1 ();   //  the functional method of Interface Int
  static void m2 () { ..... }
  default void m3 () { ..... }
}
```

Noch eine Begriffsbildung: Jedes Functional Interface hat genau eine funktionale Methode.

***Nebenbemerkung*: Bei Englisch/Deutsch mangelt es dem Autor dieser Folien offenbar an Konsequenz...**

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToDoubleFunction {
  double applyAsDouble ( int n );
}
```

```
public class Mult implements IntToDoubleFunction {
  private double x;
  public Mult ( double factor ) {
    x = factor;
  }
  public double applyAsDouble ( int m ) {
    return m * x;
  }
}
```

Ein einführendes Beispiel, um nun von Functional Interfaces zu Lambda-Ausdrücken in Java zu kommen: ...

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToDoubleFunction {
  double applyAsDouble ( int n );
}
```

```
public class Mult implements IntToDoubleFunction {
  private double x;
  public Mult ( double factor ) {
    x = factor;
  }
  public double applyAsDouble ( int m ) {
    return m * x;
  }
}
```

... Dieses Interface ist in der Java-Standardbibliothek, genauer im Package java.util.function. Wie man sieht, hat das Interface eine Methode applyToDouble, die einen int-Parameter und Rückgabetyp double hat. Das Interface ist ein Functional Interface, und applyAsDouble ist seine funktionale Methode.

***Nebenbemerkung*: Das in Abschnitt „Funktionen als Daten“ weiter vorne in diesem Kapitel definierte Interface DoubleToDoubleFunction war natürlich angelehnt an Interfaces wie IntToDoubleFunction aus der Java-Standardbibliothek.**

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToDoubleFunction {
  double applyAsDouble ( int n );
}
```

```
public class Mult implements IntToDoubleFunction {
  private double x;
  public Mult ( double factor ) {
     x = factor;
  }
  public double applyAsDouble ( int m ) {
     return m * x;
  }
}
```

Diese selbstgebastelte Klasse namens Mult implementiert das Interface IntToDoubleFunction.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToDoubleFunction {
  double applyAsDouble ( int n );
}
```

```
public class Mult implements IntToDoubleFunction {
  private double x;
  public Mult ( double factor ) {
     x = factor;
  }
  public double applyAsDouble ( int m ) {
     return m * x;
  }
}
```

Da Mult nicht abstrakt sein soll, wird die Methode applyAsDouble aus IntToDoubleFunction implementiert.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntToDoubleFunction fct1 = new Mult ( 10 );
double y = fct1.applyAsDouble ( 11 );

IntToDoubleFunction fct2 = x -> x * 10;
double z = fct2.applyAsDouble ( 11 );
```

Auf dieser Folie kommen wir zum eigentlichen Thema dieses Abschnitts, Lambda-Ausdrücke in Java.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntToDoubleFunction fct1 = new Mult ( 10 );
double y = fct1.applyAsDouble ( 11 );

IntToDoubleFunction fct2 = x -> x * 10;
double z = fct2.applyAsDouble ( 11 );
```

Oben ist noch einmal ein Codestück *ohne* Lambda-Ausdruck zu sehen: Wir richten eine Variable des Interface IntToDoubleFunction ein und weisen ihr die Adresse eines neu eingerichteten Objekts vom Typ Mult zu. Der Aufruf von applyAsDouble liefert das Produkt aus 10 und 11 zurück.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntToDoubleFunction fct1 = new Mult ( 10 );
double y = fct1.applyAsDouble ( 11 );

IntToDoubleFunction fct2 = x -> x * 10;
double z = fct2.applyAsDouble ( 11 );
```

Das ist jetzt neu. So sehen – in der einfachsten Form – Lambda-Ausdrücke in Java aus: Zuerst kommen die Namen der Parameter, in diesem Fall nur einer, und den nennen wir x. Dann kommt der Pfeil, und danach der Wert, der durch diesen Lambda-Ausdruck zurückgeliefert werden soll, also Wert des Parameters x mal 10.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntToDoubleFunction fct1 = new Mult ( 10 );
double y = fct1.applyAsDouble ( 11 );

IntToDoubleFunction fct2 = x -> x * 10;
double z = fct2.applyAsDouble ( 11 );
```

Offenbar kann man einen solchen Lambda-Ausdruck einer Variablen vom Typ IntToDoubleFunction zuweisen. Der Compiler übersetzt diese Zeile sinngemäß wie folgt: Er richtet eine namenlose, für uns nicht sichtbare Klasse ein, die das Interface IntToDoubleFunction implementiert und mehr oder weniger exakt so aussieht wie die Klasse Mult, die wir selbst eingerichtet haben: Aus dem Lambda-Ausdruck wird die Methode applyAsDouble.

Zugleich wird ein Objekt dieser namenlosen Klasse eingerichtet, und fct2 verweist auf dieses Objekt.

Die Methode applyAsDouble implementiert der Compiler für die namenlose Klasse so, dass der Wert dieses Objektattributs bei der Abarbeitung als zweiter Faktor der Multiplikation verwendet wird.

Da IntToDoubleFunction ein Functional Interface ist, gibt es keine Zweideutigkeit: Mit dem Lambda-Ausdruck kann nur die Methode applyAsDouble gemeint sein.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Im Quelltext:

```
IntToDoubleFunction fct2 = x -> x * 10;
```

Was der Compiler daraus macht:

```
class <<AnonymousClass>> implements IntToDoubleFunction {
   public double applyAsDouble ( int x ) {
      return x * 10;
   }
}

IntToDoubleFunction fct2 = new <<AnonymousClass>> ();
```

So können Sie sich vorstellen, was im Hintergrund vor sich geht, wenn Sie irgendwo einen Lambda-Ausdruck in einem Java-Quelltext hinschreiben.

Lambda-Ausdrücke in Java

Im Quelltext:

```
IntToDoubleFunction fct2 = x -> x * 10;
```

Was der Compiler daraus macht:

```
class <<AnonymousClass>> implements IntToDoubleFunction {
    public double applyAsDouble ( int x ) {
        return x * 10;
    }
}

IntToDoubleFunction fct2 = new <<AnonymousClass>> ();
```

Wie gesagt, erzeugt der Compiler selbst eine anonyme Klasse, die das betreffende Interface, hier also IntToDoubleFunction implementiert. Wir können diese Klasse nicht selbst sehen oder verwenden, sie ist rein intern. Wie diese Klasse genau heißt, braucht uns auch nicht zu interessieren, uns reicht hier der Platzhalter <<AnonymousClass>> für den eigentlichen Namen.

Lambda-Ausdrücke in Java

Im Quelltext:

```
IntToDoubleFunction fct2 = x -> x * 10;
```

Was der Compiler daraus macht:

```
class <<AnonymousClass>> implements IntToDoubleFunction {
   public double applyAsDouble ( int x ) {
      return x * 10;
   }
}

IntToDoubleFunction fct2 = new <<AnonymousClass>> ();
```

Aus dem Lambda-Ausdruck wird dann die funktionale Methode gebildet: Was links des Pfeils steht, wird zum Parameter, und was rechts vom Pfeil steht, wird zu dem Ausdruck, dessen Wert mit return zurückgeliefert wird.

Lambda-Ausdrücke in Java


```
public class X {
   public static double applyFct
                        ( IntToDoubleFunction fct, int n ) {
   return fct.applyAsDouble ( n );
}



double y = X.applyFct ( x -> x * 3.14, n );
```

Der Vollständigkeit halber sehen wir unten auch einen Lambda-Ausdruck als Parameterwert.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double y = moon.isWaning() ? 10 : 12;

.....

IntToDoubleFunction fct2 = x -> x * y;
double z = fct2.applyAsDouble ( 11 );
```

Eine kleine Variation, bei der der Faktor zur Kompilierzeit noch nicht feststeht, mögliche interne Umsetzung auf der nächsten Folie.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Im Quelltext: `IntToDoubleFunction fct2 = x -> x * y;`

```
class <<AnonymousClass>> implements IntToDoubleFunction {
   int a;
   public <<AnonymousClass>> ( int a ) {
      this.a = a;
   }
   public double applyAsDouble ( int x ) {
      return x * a;
   }
}

IntToDoubleFunction fct2 = new <<AnonymousClass>> ( y );
```

So können wir es uns bei dieser kleinen Variation vorstellen, was der Compiler daraus macht.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Im Quelltext: `IntToDoubleFunction fct2 = x -> x * y;`

```
class <<AnonymousClass>> implements IntToDoubleFunction {
  double a;
  public <<AnonymousClass>> ( double a ) {
      this.a = a;
  }
  public double applyAsDouble ( int x ) {
      return x * a;
  }
}

IntToDoubleFunction fct2 = new <<AnonymousClass>> ( y );
```

Die interne, anonyme Klasse hat nun ein Attribut, in das der aktuale Wert von y gespeichert, und das dann beim Aufruf der funktionalen Methode verwendet wird.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Closure:

- **Information aus dem Entstehungskontext des Lambda-Ausdrucks wird mitgespeichert und bei Verwendung des Lambda-Ausdrucks mitverwendet**
- ***Achtung*: Aktualer Wert wird nicht unbedingt kopiert!**
- **Daher nur konstante oder effektiv konstante Werte**
- **Effektiv konstant (effectively final): eine Variable, deren Wert ab da nicht noch einmal geändert wird**

Der Fachbegriff dafür in der Informatik lautet *Closure.*

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Closure:

- **Information aus dem Entstehungskontext des Lambda-Ausdrucks wird mitgespeichert und bei Verwendung des Lambda-Ausdrucks mitverwendet**
- ***Achtung*: Aktualer Wert wird nicht unbedingt kopiert!**
- **Daher nur konstante oder effektiv konstante Werte**
- **Effektiv konstant (effectively final): eine Variable, deren Wert ab da nicht noch einmal geändert wird**

Informationen aus dem Entstehungskontexts des Lambda-Ausdruckes – hier der aktuale Wert von y – wird zusammen mit dem Lambda-Ausdruck beziehungsweise mit dem, was der Compiler aus dem Lambda-Ausdruck macht, gespeichert.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Closure:

- **Information aus dem Entstehungskontext des Lambda-Ausdrucks wird mitgespeichert und bei Verwendung des Lambda-Ausdrucks mitverwendet**
- ***Achtung*: Aktualer Wert wird nicht unbedingt kopiert!**
- **Daher nur konstante oder effektiv konstante Werte**
- **Effektiv konstant (effectively final): eine Variable, deren Wert ab da nicht noch einmal geändert wird**

***Aber Achtung*: Insbesondere – aber nicht nur – bei Attributen von Referenztypen muss man davon ausgehen, dass das Objekt nicht kopiert, sondern referenziert wird, wie wir das ja bisher immer gesehen haben. Das könnte zu ungeahnten Komplikationen führen, wenn der Wert der Variablen noch einmal geändert würde. Daher ginge Code, bei dem der Wert noch einmal geändert würde, nicht durch den Compiler. Der Fachbegriff für Variable, deren Wert ab einer Stelle nicht mehr geändert wird, lautet *effectively final*.**

Lambda-Ausdruck = Literal

TECHNISCHE UNIVERSITÄT DARMSTADT

int	123
double	3.14159
char	´a´
String	“Hello“
boolean	true
Funktion double -> double	x -> x * x * 3.14159

Hier sehen Sie im Vergleich noch einmal, was wir zu Anfang gesagt hatten: Lambda-Ausdrücke sind Literale von Funktionstypen, also wörtlich hingeschriebene Werte eines Funktionstyps.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntPredicate {
   boolean test ( int x );
   default IntPredicate and ( IntPredicate pred ) { ..... }
   default IntPredicate or ( IntPredicate pred ) { ..... }
   default IntPredicate negate () { ..... }
}
```

Wir betrachten ein wichtiges Fallbeispiel. In Mathematik und Informatik ist ein Prädikat eine boolesche Funktion, also eine Funktion, die entweder true oder false zurückliefert.

Das hier betrachtete Interface finden Sie in java.util.function.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntPredicate {
  boolean test ( int x );
  default IntPredicate and ( IntPredicate pred ) { ..... }
  default IntPredicate or ( IntPredicate pred ) { ..... }
  default IntPredicate negate () { ..... }
}
```

Die funktionale Methode von IntPredicate ist die Methode test. Sie bekommt ein int und liefert ein boolean zurück, ist also ein Prädikat auf dem Datentyp int, daher der Name der Klasse: IntPredicate.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntPredicate {
  boolean test ( int x );
  default IntPredicate and ( IntPredicate pred ) { ..... }
  default IntPredicate or ( IntPredicate pred ) { ..... }
  default IntPredicate negate () { ..... }
}
```

Dieses Interface enthält drei Default-Methoden. Wir werden sie gleich in Aktion erleben. Aber zuerst schauen wir uns das eigentliche Thema, die funktionale Methode test und Lambda-Ausdrücke, kurz an.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class IsOdd implements IntPredicate {
  boolean test ( int m ) {
    return m % 2 == 1;
  }
}

IntPredicate pred1 = new IsOdd ();
IntPredicate pred2 = n -> n % 2 == 1;
```

Die Klasse IsOdd implementiert IntPredicate, und wie der Name IsOdd suggeriert, liefert die funktionale Methode test genau für die ungeraden ganzen Zahlen true zurück, also genau dann, wenn der Rest bei Division durch 2 gleich 1 ist.

Unten sehen Sie wieder den Vergleich: entweder ein explizit eingerichtetes Objekt von Klasse IsOdd oder ein Lambda-Ausdruck, der vom Compiler in ein Objekt genau einer solchen, aber namenlosen und für uns nicht sichtbaren und nicht verwendbaren Klasse umgewandelt wird.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntPredicate pred1 = n -> n % 2 == 1;
IntPredicate pred2 = n -> n > 0;
IntPredicate pred3 = n -> n * n < 100;
IntPredicate pred4 = pred1.negate();
IntPredicate pred5 = pred2.and ( pred3 );
IntPredicate pred6 = pred4.or ( pred5 );
```

Wie angekündigt, kommen wir kurz auf die drei Default-Methoden des Interface IntPredicate zu sprechen.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntPredicate pred1 = n -> n % 2 == 1;
IntPredicate pred2 = n -> n > 0;
IntPredicate pred3 = n -> n * n < 100;
IntPredicate pred4 = pred1.negate();
IntPredicate pred5 = pred2.and ( pred3 );
IntPredicate pred6 = pred4.or ( pred5 );
```

Hier haben wir drei verschiedene Variable von Interface IntPredicate, initialisiert mit drei verschiedenen Lambda-Ausdrücken. Der Compiler macht daraus drei verschiedene namenlose Klassen und je ein Objekt jeder dieser drei Klassen. Die drei hier eingerichteten Variablen des Interface IntPredicate verweisen auf diese drei Objekte.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntPredicate pred1 = n -> n % 2 == 1;
IntPredicate pred2 = n -> n > 0;
IntPredicate pred3 = n -> n * n < 100;
IntPredicate pred4 = pred1.negate();
IntPredicate pred5 = pred2.and ( pred3 );
IntPredicate pred6 = pred4.or ( pred5 );
```

Die Default-Methode negate liefert ein IntPredicate zurück, das bei genau denjenigen ganzen Zahlen true zurückliefert, bei denen das Prädikat, mit dem negate hier aufgerufen wird, false zurückliefert. Konkret liefert also pred4 genau dann true zurück, wenn die Zahl gerade ist.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntPredicate pred1 = n -> n % 2 == 1;
IntPredicate pred2 = n -> n > 0;
IntPredicate pred3 = n -> n * n < 100;
IntPredicate pred4 = pred1.negate();
IntPredicate pred5 = pred2.and ( pred3 );
IntPredicate pred6 = pred4.or ( pred5 );
```

Die Default-Methode and liefert ein Prädikat zurück, das genau bei denjenigen ganzen Zahlen true ist, bei denen sowohl das Prädikat, mit dem and aufgerufen wird, als auch der Parameter der Methode and jeweils true zurückliefern.

Konkret liefert pred5 also genau bei den ganzen Zahlen true zurück, die erstens positiv sind und deren Quadrat zweitens kleiner 100 ist, also: bei den Zahlen 1 bis 9.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntPredicate pred1 = n -> n % 2 == 1;
IntPredicate pred2 = n -> n > 0;
IntPredicate pred3 = n -> n * n < 100;
IntPredicate pred4 = pred1.negate();
IntPredicate pred5 = pred2.and ( pred3 );
IntPredicate pred6 = pred4.or ( pred5 );
```

Die Default-Methode or ist analog zur Default-Methode and, nur dass Verundung durch Inklusiv-Veroderung ersetzt ist. Das Prädikat pred6 liefert also genau für diejenigen Zahlen true zurück, die entweder gerade (pred4) oder im Bereich 1 bis 9 (pred5) sind oder beides.

Dieses kleine Beispiel sollte einen ersten überzeugenden Einblick geliefert haben, wofür diese drei Methoden von IntPredicate und allgemein Default-Methoden so alles gut sein können. Und es sollte auch gezeigt haben, wozu so kleine, einfache Komponenten wie IntToDoubleFunction und IntPredicate gut sind: als Bausteine, die man mittels Default-Methoden zu größeren, komplexeren Komponenten zusammensetzen kann.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
IntPredicate [ ] predicates = new IntPredicate [ 6 ];

predicates [ 0 ] = n -> n % 2 == 1;
predicates [ 1 ] = n -> n > 0;
predicates [ 2 ] = n -> n * n < 100;
predicates [ 3 ] = predicates[0].negate();
predicates [ 4 ] = predicates[1].and ( predicates[2] );
predicates [ 5 ] = predicates[3].or ( predicates[4] );
```

Die sechs Prädikate von der letzten Folie kann man natürlich auch als Array mit sechs Komponenten ausdrücken. Der Komponententyp ist dann das Interface.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public printIfTrue ( int[ ] a, IntPredicate pred ) {
   for ( int i = 0; i < a.length; i++ )
      if ( pred.test ( a[i] ) )
         System.out.println ( a[i] );
}
```

Ein kleines schematisches, aber nicht unrealistisches Anwendungsbeispiel für Prädikate, die als IntPredicate implementierende Klassen realisiert sind.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public printIfTrue ( int[ ] a, IntPredicate pred ) {
    for ( int i = 0; i < a.length; i++ )
        if ( pred.test ( a[i] ) )
            System.out.println ( a[i] );
}
```

Diese Methode soll eine bestimmte Auswahl der Komponenten des Arrays in aufsteigender Reihenfolge der Indizes auf dem Bildschirm ausgeben, und zwar genau die Komponenten, für die das Prädikat pred jeweils true ergibt.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public printIfTrue ( int[ ] a, IntPredicate pred ) {
   for ( int i = 0; i < a.length; i++ )
      if ( pred.test ( a[i] ) )
         System.out.println ( a[i] );
}
```

Hier wird die funktionale Methode test des Functional Interface IntPredicate aufgerufen. Der entscheidende Punkt ist: Für pred kann jedes beliebige Prädikat verwendet werden, beispielsweise eines der Prädikate pred1 bis pred6 von der vorletzten Folie. Die hier gezeigte Methode printIfTrue ist also unabhängig von dem konkreten Prädikat und muss daher nur einmal implementiert werden, um für jedes denkbare Prädikat sofort zur Verfügung zu stehen.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PrintlnIntConsumer implements IntConsumer {
   public void accept ( int n ) {
      System.out.println ( n )
}

public dolfTrue ( int[ ] a, IntPredicate pred, IntConsumer cons ) {
   for ( int i = 0; i < a.length; i++ )
      if ( pred.test ( a[i] ) )
         cons.accept ( a[i] );
}
```

Wir gehen sogar noch einen Schritt weiter und abstrahieren von der konkreten Aktion, also vom Ausgeben auf dem Bildschirm.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PrintlnIntConsumer implements IntConsumer {
   public void accept ( int n ) {
      System.out.println ( n )
}
```

```
public dolfTrue ( int[ ] a, IntPredicate pred, IntConsumer cons ) {
   for ( int i = 0; i < a.length; i++ )
      if ( pred.test ( a[i] ) )
         cons.accept ( a[i] );
}
```

In Package java.util.function gibt es auch ein Interface IntConsumer mit einer Methode accept, die einen Parameter vom Typ int hat und mit diesem Wert irgendetwas macht, aber die Methode accept liefert nichts zurück, ist also void. Hierhin lagern wir die eigentliche Aktion, das Ausgeben auf dem Bildschirm, aus. Das bedeutet natürlich, dass diese Aktion damit austauschbar geworden ist: Überall, wo IntConsumer erwartet wird, kann ein Objekt von PrintIntConsumer, aber statt dessen auch ein Objekt einer beliebigen anderen Klasse, deren Methode accept etwas beliebig anderes macht, verwendet werden.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PrintlnIntConsumer implements IntConsumer {
   public void accept ( int n ) {
      System.out.println ( n )
}

public dolfTrue ( int[ ] a, IntPredicate pred, IntConsumer cons ) {
   for ( int i = 0; i < a.length; i++ )
      if ( pred.test ( a[i] ) )
         cons.accept ( a[i] );
}
```

Beim Prädikat bleibt alles so wie bisher.

Fallbeispiel Prädikate

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class PrintlnIntConsumer implements IntConsumer {
   public void accept ( int n ) {
      System.out.println ( n )
}

public doIfTrue ( int[ ] a, IntPredicate pred, IntConsumer cons ) {
   for ( int i = 0; i < a.length; i++ )
      if ( pred.test ( a[i] ) )
        cons.accept ( a[i] );
}
```

Aber die ausgelagerte Aktion kommt jetzt wieder als Parameter hinein. Anstelle der eigentlichen Aktion wird jetzt die funktionale Methode ausgeführt. Damit ist die eigentliche Aktion auch in dieser Methode doIfTrue austauschbar geworden.

Lambda-Ausdrücke in Java

Komplexere, flexiblere Lambda-Ausdrücke:

```
n -> n % 2 == 1

( int n ) -> { return n % 2 == 1; }

( int n, double x ) ->
    { System.out.print ( n ); System.out.print ( x ); }
```

Wir haben Lambda-Ausdrücke in Java bisher noch nicht in voller Allgemeinheit gesehen. Das holen wir zum Abschluss dieses Abschnitts jetzt nach.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexere, flexiblere Lambda-Ausdrücke:

```
n -> n % 2 == 1
```

```
( int n ) -> { return n % 2 == 1; }
```

```
( int n, double x ) ->
    { System.out.print ( n ); System.out.print ( x ); }
```

Diesen Lambda-Ausdruck hatten wir schon bei IntPredicate gesehen. Das ist aber eigentlich nur eine Kurzform, die nur in bestimmten Fällen möglich ist.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexere, flexiblere Lambda-Ausdrücke:

```
n -> n % 2 == 1

( int n ) -> { return n % 2 == 1; }

( int n, double x ) ->
    { System.out.print ( n ); System.out.print ( x ); }
```

Das ist die absolut äquivalente Langform: Der Parameter n ist in Klammern gesetzt, und anstelle des zurückzuliefernden Ausdrucks steht die ganze return-Anweisung inklusive Semikolon in geschweiften Klammern da. Der Datentyp des Parameters kann, muss aber nicht angegeben sein, denn der Compiler „weiß“ ja, welche Parameter die funktionale Methode hat, auf die der Lambda-Ausdruck passen muss.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexere, flexiblere Lambda-Ausdrücke:

```
n -> n % 2 == 1

( int n ) -> { return n % 2 == 1; }

( int n, double x ) ->
    { System.out.print ( n ); System.out.print ( x ); }
```

Mit dieser allgemeineren Schreibweise ist es dann problemlos möglich, Methoden als Lambda-Ausdrücke zu schreiben, die mehrere Anweisungen enthalten und auch gar nicht unbedingt etwas zurückliefern müssen.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Komplexere, flexiblere Lambda-Ausdrücke:

```
n -> n % 2 == 1

( int n ) -> { return n % 2 == 1; }

( int n, double x ) ->
    { System.out.print ( n ); System.out.print ( x ); }
```

Bei mehr als einem Parameter oder bei einer leeren Parameterliste müssen die Klammern unbedingt sein, nur bei genau einem Parameter wie im obersten Lambda-Ausdruck auf dieser Folie dürfen die Klammern fehlen. Mehrere Parameter werden durch Kommas voneinander getrennt. Wie gesagt, dürfen die Typen der Parameter auch weggelassen werden.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenfassung:

- **In Java abgekürzte Schreibweise für den Aufruf der funktionalen Methode eines (namenlosen, nicht explizit definierten) Functional Interface**
- **Standardform:**

```
( int x, int y ) -> { return x * x + y * y; }
```

- **Kurzform (nur in einfachen Fällen möglich):**

```
( x, y ) -> x * x + y * y
```

Als nächstes kommen wir zu Lambda-Ausdrücken in Racket. Um den Unterschied zu Java möglichst klar zu sehen, fassen wir kurz die Eckpunkte zusammen.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenfassung:

- **In Java abgekürzte Schreibweise für den Aufruf der funktionalen Methode eines (namenlosen, nicht explizit definierten) Functional Interface**
- **Standardform:**

```
( int x, int y ) -> { return x * x + y * y; }
```

- **Kurzform (nur in einfachen Fällen möglich):**

```
( x, y ) -> x * x + y * y
```

In Java sind Lambda-Ausdrücke eigentlich kein eigenständiges Sprachkonstrukt, sondern eine bequeme Abkürzung für die Einrichtung einer Klasse, die ein Functional Interface implementiert, und ein Objekt dieser Klasse. Wir werden gleich in Racket sehen, dass Lambda-Ausdrücke dort durchaus eigenständige Sprachkonstrukte sind.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenfassung:

- In Java abgekürzte Schreibweise für den Aufruf der funktionalen Methode eines (namenlosen, nicht explizit definierten) Functional Interface
- Standardform:

```
( int x, int y ) -> { return x * x + y * y; }
```

- Kurzform (nur in einfachen Fällen möglich):

```
( x, y ) -> x * x + y * y
```

Anstelle der das Functional Interface implementierenden Klasse und der Einrichtung eines Objektes davon können wir auch einfach die funktionale Methode auf diese abgekürzte Art spezifizieren. Der Compiler erledigt den Rest.

Lambda-Ausdrücke in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Zusammenfassung:

- In Java abgekürzte Schreibweise für den Aufruf der funktionalen Methode eines (namenlosen, nicht explizit definierten) Functional Interface
- Standardform:

```
( int x, int y ) -> { return x * x + y * y; }
```

- Kurzform (nur in einfachen Fällen möglich):

```
( x, y ) -> x * x + y * y
```

Wenn die Implementation dieser Methode aus einer einzigen return-Anweisung besteht, kann man die geschweiften Klammern und das return auch weglassen. Und wenn die Situation so einfach ist, dass der Compiler die korrekten Typen für die Parameter selbst ableiten kann, dann können auch diese weggelassen werden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Lambda-Ausdrücke in Racket

Jetzt der Vergleich mit Racket.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Noch einmal Kurzform Java zum Vergleich:**

```
( x, y ) -> x * x + y * y
```

- **Jetzt *allgemeine Form* in Racket:**

```
(  lambda ( x y )  ( + ( * x x ) ( * y y ) )  )
```

Lambda-Ausdrücke in Racket sind vergleichbar mit der Kurzform in Java, die nur bei einer einzelnen return-Anweisung möglich ist.

Lambda-Ausdrücke in Racket

- Noch einmal Kurzform Java zum Vergleich:

```
( x, y ) -> x * x + y * y
```

- Jetzt *allgemeine Form* in Racket:

```
( lambda ( x y )  ( + ( * x x ) ( * y y ) )  )
```

Sie können das Wort lambda in Racket als eine Funktion auffassen, die einen Lambda-Ausdruck als Parameter hat und eine Funktion, die diesen Lambda-Ausdruck implementiert, zurückliefert.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch einmal Kurzform Java zum Vergleich:

```
( x, y ) -> x * x + y * y
```

- Jetzt *allgemeine Form* in Racket:

```
(  lambda ( x y )  ( + ( * x x ) ( * y y ) )  )
```

Hinter dem lambda steht dann der Lambda-Ausdruck. Abgesehen von den Eigentümlichkeiten von Racket – Präfixnotation, keine trennenden Kommas, strikte Klammerung – sieht der Ausdruck in Racket genauso wie in Java aus, nur dass es auch keinen Pfeil gibt.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; add-unary-functions:
;;       ( number -> number ) ( number -> number )
;;       -> ( number number -> number )

( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )
```

Wir schauen uns jetzt Lambda-Ausdrücke in Racket in einer kleinen, aber durchaus sinnvollen Anwendung an.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; add-unary-functions:
;;       ( number -> number ) ( number -> number )
;;       -> ( number number -> number )

( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )
```

Die Besonderheit ist, dass der von der Funktion zurückgelieferte Wert eine Funktion ist, und zwar eine Funktion mit zwei zahlenwertigen Parametern, die eine Zahl zurückliefert.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; add-unary-functions:
;;      ( number -> number ) ( number -> number )
;;      -> ( number number -> number )

( define ( add-unary-functions fct1 fct2 )
     ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )
```

Und zwar bauen wir die zurückgelieferte Funktion aus den beiden Parametern auf. Die Funktion, die zurückgeliefert wird, ruft fct1 mit ihrem eigenen ersten Parameter und fct2 mit ihrem eigenen zweiten Parameter auf. Die zurückgelieferte Funktion hat also irgendwo, ohne dass wir es sehen können, Verweise auf die beiden Funktionen fct1 und fct2 gespeichert, so dass sie die beiden Funktionen aufrufen kann.
Das hatten wir auch schon bei Lambda-Ausdrücken in Java gesehen und Closure genannt. So nennen wir es auch hier. Intern wird das Ganze natürlich mehr oder weniger genauso realisiert, wie wir das in Abschnitt „Functional Interfaces und Lambda-Ausdrücke in Java“ weiter vorne in diesem Kapitel schon gesehen hatten.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )

( define ( times10 a ) ( * a 10 ) )

( define ( div5 a ) ( / a 5 ) )

(  ( add-unary-functions times10 div5  )  3.14  2.71  )

;; Äquivalent zu: ( + ( * 3.14 10 ) ( / 2.71 5 ) )
```

Und so sieht dann die Anwendung dieser Funktion add-unary-functions aus, die zwei unäre Funktionen als Parameter bekommt und daraus eine neue Funktion konstruiert.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )

( define ( times10 a ) ( * a 10 ) )

( define ( div5 a ) ( / a 5 ) )

(  ( add-unary-functions times10 div5  )  3.14  2.71  )

;; Äquivalent zu: ( + ( * 3.14 10 ) ( / 2.71 5 ) )
```

Hier ist die Funktion nochmals, exakt so wie auf der letzten Folie gezeigt.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )

( define ( times10 a ) ( * a 10 ) )

( define ( div5 a ) ( / a 5 ) )

(  ( add-unary-functions times10 div5  )  3.14  2.71  )

;; Äquivalent zu: ( + ( * 3.14 10 ) ( / 2.71 5 ) )
```

Jetzt definieren wir uns irgendwelche zwei Funktionen, die einen zahlenwertigen Parameter haben und eine Zahl zurückliefern und daher als Parameter für add-unary-functions in Frage kommen.

Was diese beiden Funktionen machen, ist uns hier egal, sie dienen nur der Illustration.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )

( define ( times10 a ) ( * a 10 ) )

( define ( div5 a ) ( / a 5 ) )

(  ( add-unary-functions times10 div5  )  3.14  2.71  )

;; Äquivalent zu: ( + ( * 3.14 10 ) ( / 2.71 5 ) )
```

Hier rufen wir die Funktion add-unary-functions auf. Die beiden Parameter sind die beiden Funktionen, die wir gerade davor definiert hatten. Der Rückgabewert dieses Aufrufs von add-unary-function ist, wie oben gesehen, eine Funktion mit zwei zahlenwertigen Parametern.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )

( define ( times10 a ) ( * a 10 ) )

( define ( div5 a ) ( / a 5 ) )

(  ( add-unary-functions times10 div5 )  3.14  2.71  )

;; Äquivalent zu: ( + ( * 3.14 10 ) ( / 2.71 5 ) )
```

Wir hätten die zurückgelieferte Funktion mit define in einer Konstanten abspeichern und dann den Namen dieser Konstanten als Funktionsnamen verwenden können. Das machen wir später in anderen Beispielen.

Hier setzen wir den Aufruf der funktionsgenerierenden Funktion gleich als Aufruf der zurückgelieferten Funktion ein, wenden die zurückgelieferte Funktion also gleich, ohne sie zwischenzuspeichern, auf zwei Zahlen an.

Lambda-Ausdrücke in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( add-unary-functions fct1 fct2 )
      ( lambda ( x y ) ( + ( fct1 x ) ( fct2 y ) ) )

( define ( times10 a ) ( * a 10 ) )

( define ( div5 a ) ( / a 5 ) )

(  ( add-unary-functions times10 div5  )  3.14  2.71  )

;; Äquivalent zu: ( + ( * 3.14 10 ) ( / 2.71 5 ) )
```

Auf die erste Zahl wird durch die von add-unary-function zurückgelieferte Funktion dann fct1 angewandt, also Multiplikation mit 10. Auf die zweite Zahl wird fct2 angewandt, das ergibt Division durch 5. Die von add-unary-functions zurückgelieferte Funktion addiert dann noch die beiden Ergebnisse, und die Summe ist die Rückgabe.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fallbeispiel filter in Racket

Als wichtiges Anwendungsbeispiel schauen wir uns an, wie man Lambda-Ausdrücke in filter, map und fold nutzen kann, als erstes filter.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Da die Funktion filter schon vordefiniert ist, nennen wir sie hier my-filter.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

***Erinnerung*: Im Abschnitt „Funktionen höherer Ordnung“ weiter vorne in diesem Kapitel hatten wir schon die Notation eingeführt, dass X, Y, Z und ähnliches für einen beliebigen, aber festen Typ stehen soll. Eine Liste von X ist also homogen, aber der Typ der Listenelemente ist egal, während eine Liste von ANY heterogen ist, also Elemente von beliebigen Typen bunt gemischt enthalten kann.**

Wichtig ist dabei: Überall, wo derselbe Großbuchstabe steht, muss derselbe Typ eingesetzt werden. In beiden Parametern und im Rückgabetyp muss also X denselben Typ bezeichnen.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Die Filter-Funktionalität ist ja, dass aus einer Liste eines Typs X eine andere Liste desselben Typs X konstruiert wird, indem alle Elemente entfernt werden, die nicht einem gegebenen Prädikat entsprechen.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Dieses Prädikat wird als erster Parameter übergeben. Es muss zum Listentyp passen, das heißt, sein Parameter muss ebenfalls vom Typ X sein.

Bei einem Prädikat auf Zahlen muss X dann gleich number sein; bei einem Prädikat auf Strings muss X Typ String sein; bei einem Prädikat auf Symbolen muss X gleich der Typ Symbol sein; und so weiter.

Wenn das Prädikat beliebige Typen verarbeiten kann, dann kann X auch gleich ANY sein. Dafür hatten wir in Kapitel 04b, Abschnitt zu Structs, schon ein Beispiel: das Prädikat, das genau dann true ergibt, wenn der Parameter vom Struct-Typ student ist.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Ein Filter auf einer leeren Liste ergibt natürlich eine leere Liste.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Für ein Element der Liste ist die entscheidende Frage, ob das Prädikat für dieses Element erfüllt ist oder nicht.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Falls ja, dann gehört dieses Element in die Ergebnisliste. Das heißt, wenn das erste Element das Prädikat erfüllt, dann besteht die Ergebnisliste aus dem ersten Element plus der Anwendung des Filters auf den Rest der Liste, also plus dem Ergebnis des rekursiven Aufrufs von my-filter mit der Restliste.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Erfüllt das erste Element hingegen das Prädikat *nicht*, dann ist das Ergebnis des Filters für die Gesamtliste gleich dem Ergebnis des Filters auf der Restliste, das heißt, das Ergebnis des rekursiven Aufrufs von my-filter mit der Restliste.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Beim Aufruf von my-filter kommt jetzt Lambda ins Spiel. Man muss das Prädikat nicht vorab als boolesche Funktion definieren, sondern kann es auch ad-hoc, also erst im Aufruf definieren. Hier wird ein Prädikat definiert, das eine reelle Zahl erwartet, also Imaginärteil 0, so dass der Größenvergleich definiert ist.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )

(  my-filter  (  lambda ( x )  ( < x 10 )  )  number-list  )
```

Die Liste muss dann entsprechend auch homogen aus reellen Zahlen bestehen, die natürlich beliebig gemischt aus ganzzahligen, rationalen und nichtexakt darstellbaren reellen Zahlen bestehen kann, denn der Größenvergleich mit der Zahl 10 ist für alle diese Darstellungsarten reeller Zahlen definiert und wird ohne Programmabbruch durchgehen.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )
```

```
;; Type: ( list of real ) real -> ( list of real )
( define ( less-than-only lst x )
  ( cond
    [  ( empty? lst )  empty  ]
    [  ( < ( first lst ) x )
            ( cons ( first lst )
                ( less-than-only ( rest lst ) x ) ) ]
    [  else  ( less-than-only ( rest lst ) x )  ] ) )
```

Hier sehen Sie zum Vergleich unter der eben diskutierten Funktion my-filter noch einmal die Funktion less-than-only aus Kapitel 04b, Abschnitt zu rekursiven Funktionen auf Listen. Zum besseren Vergleich ist das Konstrukt mit den zwei if-Verzweigungen durch einen äquivalenten cond-Ausdruck ersetzt.

Wie Sie sehen, sind die beiden Funktionen praktisch identisch, nur dass bei less-than-only das Prädikat bis auf den Vergleichswert festgelegt ist, während es bei my-filter völlig offengehalten ist. Durch Festlegung des Prädikats bei less-than-only ist allerdings auch der Typ der Listenelemente stark eingeschränkt, nämlich auf reelle Zahlen, und anstelle des allgemeinen Prädikats wird der Vergleichswert für das konkrete Prädikat als Parameter übergeben.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )
```

```
;; Type: ( list of ANY ) -> ( list of student )
( define ( all-students input-list )
  ( cond
    [  ( empty? input-list )  empty  ]
    [  ( student? ( first input-list ) )
            ( cons ( first input-list )
              ( all-students ( rest input-list ) ) ) ]
    [  else  ( all-students ( rest input-list ) )  ] ) )
```

Etwas anders sieht die Situation aus bei Funktion all-students aus Kapitel 04b, Abschnitt zu Structs, siehe unten; oben weiterhin my-filter wie auf den letzten Folien.

Fallbeispiel filter in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> boolean ) ( list of X ) -> ( list of X )
( define ( my-filter pred lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ ( pred ( first lst ) )
          ( cons ( first lst ) ( my-filter pred ( rest lst ) ) ) ]
        [ else ( my-filter pred ( rest lst ) ) ] ) )
```

```
;; Type: ( list of ANY ) -> ( list of student )
( define ( all-students input-list )
  ( cond
    [  ( empty? input-list )  empty  ]
    [  ( student? ( first input-list ) )
            ( cons ( first input-list )
              ( all-students ( rest input-list ) ) ) ]
    [  else  ( all-students ( rest input-list ) )  ] ) )
```

Hier ist X gleich ANY, und die Ergebnisliste ist daher eigentlich auch eine Liste von X gleich ANY. Aber da wir wissen, dass das Prädikat alles wegfiltert, was nicht vom Typ student ist, können wir den Elementtyp der Ergebnisliste im Kommentar auf student einschränken, was sicherlich eine wichtige Information für den Nutzer der Funktion all-students ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fallbeispiel map in Racket

Nach filter als nächstes nun map.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )

( define ( my-map fct lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ else ( cons ( fct ( first lst ) )
                      ( my-map fct ( rest lst ) ) ) ] ) )

(  my-map  (  lambda ( x )  ( * x 10 )  )  number-list  )
```

Auch hier nennen wir die Funktion my-map, da map vordefiniert ist.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )

( define ( my-map fct lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ else ( cons ( fct ( first lst ) )
                      ( my-map fct ( rest lst ) ) ) ] ) )

(  my-map  (  lambda ( x )  ( * x 10 )  )  number-list  )
```

Bei map wird aus einer Liste eines Typs X eine Liste eines Typs Y konstruiert. Die beiden Typen X und Y können identisch sein, können aber auch verschieden sein, daher unterscheiden wir hier die beiden Listentypen durch unterschiedliche Buchstaben, behalten aber im Hinterkopf, dass X und Y auch identisch sein *können*.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )

( define ( my-map fct lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ else ( cons ( fct ( first lst ) )
                      ( my-map fct ( rest lst ) ) ) ] ) )

(  my-map  (  lambda ( x )  ( * x 10 )  )  number-list  )
```

Für den Aufbau der Ergebnisliste verwendet my-map eine Funktion, die ein Element von Y aus einem Element von X konstruiert. Zu jedem Element der Eingabeliste soll die Ergebnisliste an derselben Position ein Element haben, das aus ersterem durch Anwendung von fct entsteht. Insbesondere sind beide Listen gleich lang.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )

( define ( my-map fct lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ else ( cons ( fct ( first lst ) )
                      ( my-map fct ( rest lst ) ) ) ] ) )

(  my-map  (  lambda ( x )  ( * x 10 )  )  number-list  )
```

Das Ergebnis von my-map bei einer leeren Liste ist natürlich leer.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )

( define ( my-map fct lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ else ( cons ( fct ( first lst ) )
                      ( my-map fct ( rest lst ) ) ) ] ) )

(  my-map  (  lambda ( x )  ( * x 10 )  )  number-list  )
```

Wenn die Liste nicht leer ist, dann besteht das Ergebnis von fct aus der Anwendung der Funktion auf das erste Element der Liste plus die rekursiven Anwendung von my-map auf die Restliste.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )

( define ( my-map fct lst )
    ( cond
        [ ( empty? lst ) empty ]
        [ else ( cons ( fct ( first lst ) )
                      ( my-map fct ( rest lst ) ) ) ] ) )

(  my-map  (  lambda ( x )  ( * x 10 )  )  number-list  )
```

Jetzt können wir my-map wieder mit einem Lambda-Ausdruck aufrufen. Diese Funktion passt zu einer Liste von beliebigen Zahlen, die diesmal auch komplex sein dürfen. Entsprechend wird eine Liste von Zahlen zurückgeliefert. In diesem Beispiel sind also X und Y identisch.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )
;; X = student, Y = natural

;; enrollment-number: student -> natural
( define ( enrollment-number studi )
      ( student-enrollment-number studi ) )

;; the-enrollment-numbers:
;;      ( list of student ) -> ( list of natural )
( define ( the-enrollment-numbers list-of-students )
       ( my-map enrollment-number list-of-students  ) )
```

Ein weiteres Beispiel für die Anwendung von my-map, nun mit zwei verschiedenen Datentypen X und Y, aber diesmal nicht mit einem Lambda-Ausdruck, sondern zur Abwechslung und zum Vergleich mit einer Funktion, die außerhalb des Aufrufs von my-map extra definiert wird.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )
;; X = student, Y = natural

;; enrollment-number: student -> natural
( define ( enrollment-number studi )
      ( student-enrollment-number studi ) )

;; the-enrollment-numbers:
;;      ( list of student ) -> ( list of natural )
( define ( the-enrollment-numbers list-of-students )
       ( my-map enrollment-number list-of-students  ) )
```

Wir hatten den Struct-Typ student definiert, die Eingabeliste soll eine Liste von student sein. Die Ergebnisliste soll die Matrikelnummern aller dieser Studierenden enthalten, also Typ number. Wir können sogar weitergehen und garantieren, dass in der Ergebnisliste nur natürliche Zahlen vorkommen werden, denn Matrikelnummern sind bei uns natürliche Zahlen.

***Nebenbemerkung*: Eine solche Listenoperation heißt in der Informatik auch Projektion: Die Studierenden werden auf eines ihrer Attribute, die Matrikelnummer, „projiziert“.**

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )
;; X = student, Y = natural

;; enrollment-number: student -> natural
( define ( enrollment-number studi )
      ( student-enrollment-number studi ) )

;; the-enrollment-numbers:
;;      ( list of student ) -> ( list of natural )
( define ( the-enrollment-numbers list-of-students )
      ( my-map enrollment-number list-of-students  ) )
```

Das ist die Funktion, die wir für my-map definieren. Sie erwartet also ein Objekt vom Typ student als Parameter und liefert eine natürliche Zahl zurück.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )
;; X = student, Y = natural

;; enrollment-number: student -> natural
( define ( enrollment-number studi )
      ( student-enrollment-number studi ) )

;; the-enrollment-numbers:
;;      ( list of student ) -> ( list of natural )
( define ( the-enrollment-numbers list-of-students )
       ( my-map enrollment-number list-of-students  ) )
```

Mit der Schreibweise, die wir schon gesehen haben, greifen wir auf das Attribut enrollment-number des Parameters studi zu.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )
;; X = student, Y = natural

;; enrollment-number: student -> natural
( define ( enrollment-number studi )
      ( student-enrollment-number studi ) )

;; the-enrollment-numbers:
;;      ( list of student ) -> ( list of natural )
( define ( the-enrollment-numbers list-of-students )
       ( my-map enrollment-number list-of-students  ) )
```

Wir schreiben eine Funktion, die aus einer Liste von Studierenden eine Liste von deren Matrikelnummern erzeugt.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )
;; X = student, Y = natural

;; enrollment-number: student -> natural
( define ( enrollment-number studi )
      ( student-enrollment-number studi ) )

;; the-enrollment-numbers:
;;      ( list of student ) -> ( list of natural )
( define ( the-enrollment-numbers list-of-students )
      ( my-map enrollment-number list-of-students  ) )
```

Aber diese Funktion macht eigentlich nichts selbst, sondern delegiert die Aufgabe an die generische Funktion my-map von der letzten Folie und verwendet dafür die oben definierte Funktion enrollment-number zur Extraktion der Matrikelnummer eines Studierenden.

Fallbeispiel map in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X -> Y ) ( list of X ) -> ( list of Y )
( define ( my-map fct lst )
   ( cond
      [ ( empty? lst ) empty ]
      [ else ( cons ( fct ( first lst ) ) ( my-map fct ( rest lst ) ) ) ] ) )
```

```
;; Type: ( list of number ) -> ( list of number )
;; Returns: the square roots of the elements of list
( define ( sqrts lst )
   ( cond
      [ ( empty? lst ) empty
      ( else cons ( sqrt ( first lst ) ) ( sqrts ( rest lst ) ) ] ) ) )
```

Hier sehen Sie die eben besprochene Funktion my-map im Vergleich mit der Funktion sqrts aus Kapitel 04b, Abschnitt „Rekursive Funktionen auf Listen“. Allerdings haben wir sqrts nicht eins-zu-eins kopiert, sondern die if-Verzweigung durch einen äquivalenten cond-Ausdruck ersetzt, um die Parallelität möglichst genau herauszuarbeiten: In my-map ist die Funktion als Parameter offengehalten, in sqrts ist eine bestimmte Funktion fest verdrahtet, nämlich die Berechnung der Quadratwurzel. Als Konsequenz ist bei sqrts der Typ X ebenfalls wieder stark eingeschränkt, ähnlich wie bei less-than-only im Gegensatz zu my-filter weiter vorne.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fallbeispiel fold in Racket

Als letztes der drei Fallbeispiele nun fold, die Faltungsoperation.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Auch hier wieder der Name my-fold statt einfach nur fold.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Die Funktion my-fold berechnet aus einer homogenen Liste einen einzelnen Wert, der vom selben Typ sein kann oder auch nicht. Daher unterscheiden wir wie bei my-map die Typen durch Benennung mit X und Y, behalten aber im Hinterkopf, dass X und Y auch identisch sein *können*.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Zur Lösung der Aufgabe von fold – Berechnung eines einzelnen Wertes aus einer homogenen Liste – wird eine Funktion benötigt, die das jeweils aktuelle Listenelement mit dem momentanen Zwischenergebnis verknüpft, woraus ein neues Zwischenergebnis entsteht. Die Zwischenergebnisse sind natürlich vom selben Typ wie das Endergebnis, also Y. Bei einer Liste von X ist das aktuelle Listenelement dann vom Typ X.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Wir brauchen auch noch einen initialen Wert, mit dem wir dann die erste Verknüpfung mit einem Element der Liste bewerkstelligen können.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Wenn die Liste leer ist, dann ist dieser initiale Wert auch das Endergebnis.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Ansonsten wird das Endergebnis von my-fold berechnet, indem my-fold rekursiv auf die Restliste angewendet wird und das daraus resultierende Zwischenergebnis dann mit dem ersten Element der Liste verknüpft wird.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                    ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Das klassische Beispiel für fold ist die Addition der Elemente einer Liste von Zahlen. Die Verknüpfungsfunktion addiert das aktuelle Listenelement x auf die momentane Zwischensumme y. In diesem Beispiel realisieren wir die Verknüpfungsfunktion wieder durch einen Lambda-Ausdruck.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

 ;; X = number, Y = number
 (  my-fold  (  lambda ( x y )  ( + x y )  )  0  number-list  )
```

Die Summe über eine leere Menge von Summanden hat per Definition den Wert 0, daher ist 0 der geeignete initiale Wert.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
( define ( my-fold fct init lst )
    ( cond
        [  ( empty? lst )  init  ]
        [  else  ( fct ( first lst )
                       ( my-fold fct init ( rest lst ) ) )  ] ) )

;; X = student, Y = natural
(  my-fold
       0  (  lambda ( x y )  ( + ( student-fee x ) y )  lst  )
```

Eine kleine Variation des Beispiels, bei dem die Typen X und Y nun unterschiedlich sind. Aus einer Liste von Studierenden wird also die Summe aller Gebühren berechnet.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
;; X = student, Y = ( list of symbol )

;; add-last-name:
;;       student ( list of symbol ) -> ( list of symbol )
( define ( add-last-name studi lst )
     ( cons ( student-last-name studi ) lst ) )

(  my-fold add-last-name empty student-list  )
```

Auch für my-fold wieder zum Vergleich ein Beispiel, bei dem die Verknüpfungsfunktion nicht ein ad-hoc definierter Lambda-Ausdruck, sondern eine separat definierte Funktion ist.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
;; X = student, Y = ( list of symbol )

;; add-last-name:
;;        student ( list of symbol ) -> ( list of symbol )
( define ( add-last-name studi lst )
     ( cons ( student-last-name studi ) lst ) )

(  my-fold add-last-name empty student-list  )
```

Eigentlich ist die Faltungsoperation generell dafür gedacht, skalare Werte wie eben bei der Addition zu produzieren. Der Ergebnistyp kann aber durchaus strukturiert sein, etwa wie hier eine homogene Liste.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
;; X = student, Y = ( list of symbol )

;; add-last-name:
;;       student ( list of symbol ) -> ( list of symbol )
( define ( add-last-name studi lst )
     ( cons ( student-last-name studi ) lst ) )

(  my-fold add-last-name empty student-list  )
```

Die Zwischenergebnisse, die durch die Verknüpfungsfunktion add-last-name mit den einzelnen Elementen der Eingabeliste verknüpft werden, sind dann ebenfalls Listen mit demselben Elementtyp.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
;; X = student, Y = ( list of symbol )

;; add-last-name:
;;       student ( list of symbol ) -> ( list of symbol )
( define ( add-last-name studi lst )
    ( cons ( student-last-name studi ) lst ) )

(  my-fold add-last-name empty student-list  )
```

Die Verknüpfung besteht dann darin, dass der Nachname des aktuellen Listenelements an das Zwischenergebnis angehängt wird, so dass das nächste Zwischenergebnis eine um ein Element erweiterte Liste ist.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
;; X = student, Y = ( list of symbol )

;; add-last-name:
;;       student ( list of symbol ) -> ( list of symbol )
( define ( add-last-name studi lst )
     ( cons ( student-last-name studi ) lst ) )

(  my-fold add-last-name empty student-list  )
```

Bei einer Faltung, die schrittweise eine Liste von Grund auf aufbaut, ist empty der geeignete initiale Wert.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: ( X Y -> Y ) Y ( list of X ) -> Y
;; X = student, Y = ( list of symbol )

;; add-last-name:
;;       student ( list of symbol ) -> ( list of symbol )
( define ( add-last-name studi lst )
     ( cons ( student-last-name studi ) lst ) )

(  my-fold add-last-name empty student-list  )
```

Dieser Aufruf von my-fold liefert letztendlich dasselbe Ergebnis, als wenn wir my-map aufgerufen hätten mit einer Funktion, die aus einem student-Objekt dessen Nachnamen extrahiert.

Fallbeispiel fold in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Nebenbemerkung:

- **In Racket gibt es *kein* fold, sondern *zwei* eingebaute Funktionen:**
  - **foldl: linksassoziativ**
  - **foldr: rechtsassoziativ**
- **Beispiel sukzessive Potenzbildung „a hoch b hoch c hoch d“ aus Liste (list e f g) und initialem Wert x:**
  - **foldl: (g hoch (f hoch (e hoch x)))**
  - **foldr: (e hoch (f hoch (g hoch x)))**
- **Unser my-fold: simuliert foldr**

In Racket gibt es keine Funktion fold, sondern zwei Funktionen, foldl und foldr, die sich im Ergebnis in bestimmten Fällen unterscheiden. Mathematisch gesprochen, rechnet jede Faltung die Elemente einer Liste schrittweise nach irgendeiner gegebenen Berechnungsvorschrift (Parameter fct) zusammen, und foldl macht das von links nach rechts, foldr von rechts nach links. Für assoziative Berechnungsvorschriften wie die Addition macht das keinen Unterschied, für nichtassoziative in der Regel schon. Unser my-fold baut foldr nach.

TECHNISCHE UNIVERSITÄT DARMSTADT

Kombinationen aus fold, filter und map

Bevor wir zum vierten und letzten Fallbeispiel, dem vergleich zweiter Listen, kommen, schauen wir uns kurz anhand eines einfachen Beispiels an, welche Ausdruckskraft diese drei Funktionen erreichen, wenn man sie kombiniert.

Kombinationen aus fold+filter+map

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( make-struct student ( ... fee ... ) )

;; Type: ( list of ANY ) -> number
;; Returns: the sum of the fees of all student in the list
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Als Anwendungsbeispiel nehmen wir wieder Studierende.

Kombinationen aus fold+filter+map

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( make-struct student ( ... fee ... ) )

;; Type: ( list of ANY ) -> number
;; Returns: the sum of the fees of all student in the list
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Wir fügen noch ein Attribut ein, die Gebühren, die ein Studierender semesterweise zu zahlen hat. Falls diese Gebühren nicht für alle Studierenden gleich sind, dann wird man die Gebühren sinnvollerweise zu einem Attribut des Struct-Typs student machen.

Kombinationen aus fold+filter+map

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( make-struct student ( ... fee ... ) )

;; Type: ( list of ANY ) -> number
;; Returns: the sum of the fees of all student in the list
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Allerdings kann die Eingabeliste Elemente beliebigen Typs beliebig gemischt enthalten. Daher wenden wir auf die Eingabeliste zuerst einen Filter an, konkret unsere selbstgebastelte Funktion my-filter. Die Rückgabe dieses Ausdrucks ist die Liste aller Studierenden aus der Eingabeliste, alle anderen Elemente der Liste sind herausgefiltert.

Kombinationen aus fold+filter+map

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( make-struct student ( ... fee ... ) )

;; Type: ( list of ANY ) -> number
;; Returns: the sum of the fees of all student in the list
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Diese Ergebnisliste, die nur aus Studierenden besteht, ist nun die Eingabeliste für my-map. Der erste Parameter extrahiert aus jedem Studierenden den Wert des Attributs fee. Das Ergebnis des Aufrufs von my-map ist also eine Liste der Gebührenhöhen aller Studierenden, die in der Eingabeliste von sum-of-student-fees zu finden sind.

Kombinationen aus fold+filter+map

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( make-struct student ( ... fee ... ) )

;; Type: ( list of ANY ) -> number
;; Returns: the sum of the fees of all student in the list
( define ( sum-of-student-fees list )
   ( my-fold
     ( lambda ( x y ) ( + x y ) )
     0
     ( my-map
        ( lambda ( stud ) ( student-fee stud ) )
        ( my-filter ( lambda ( x ) ( student? x ) ) list ) ) ) )
```

Und diese Liste aus Zahlen wiederum ist dann der erste Parameter für die abschließende Faltung. Wir haben schon gesehen, dass dieser Lambda-Ausdruck zusammen mit dem initialen Wert 0 die Elemente einer zahlenwertigen Liste aufsummiert, in diesem Fall also wie gewünscht die Gebührenhöhen der einzelnen Studierenden in der Eingabeliste von sum-of-student-fees.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fallbeispiel: Vergleich zweier Listen

Nun das angekündigte letzte Fallbeispiel. Auch dieses Beispiel haben wir schon gesehen, und zwar unter dem Namen equal-at-some-position. Vergleichen Sie die folgende Funktion Punkt für Punkt mit equal-at-some-position und sehen Sie sich noch einmal den zugehörigen Aha-Effekt an!

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: (X Y -> boolean) (list of X) (list of Y) -> boolean
;; Returns: true iff  there is at least one common
;;    position of list1 and list2 with matching values
( define ( match-at-some-position pred list1 list2 )
   ( cond
      [ ( empty? list1 ) #f ]
      [ ( empty? list2 ) #f ]
      [ ( pred ( first list1 ) ( first list2  ) ) #t ]
      [ else ( match-at-some-position
                        pred ( rest list1 ) ( rest list2 ) ) ] ) )
```

Wir betrachten eine Neuimplementation von equal-at-some-position, die nicht wie die erste Implementation auf einen konkreten Fall zugeschnitten, sondern jetzt maximal generisch ist.

Wie wir gleich besprechen werden, beinhaltet dies auch, dass die Vergleichsoperation nicht unbedingt ein Test auf Gleichheit sein muss, sondern ebenfalls maximal generisch gehalten wird. Daher nennen wir die Funktion auch nicht mehr equal, sondern match, um anzudeuten, dass es um *irgendeine* Art von Abgleich geht.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: (X Y -> boolean) (list of X) (list of Y) -> boolean
;; Returns: true iff  there is at least one common
;;    position of list1 and list2 with matching values
( define ( match-at-some-position pred list1 list2 )
   ( cond
      [ ( empty? list1 ) #f ]
      [ ( empty? list2 ) #f ]
      [ ( pred ( first list1 ) ( first list2  ) ) #t ]
      [ else ( match-at-some-position
                        pred ( rest list1 ) ( rest list2 ) ) ] ) )
```

Beide Listen müssen jeweils von einem homogenen Datentyp sein, sonst wird keine Abgleichoperation möglich sein. Aber die beiden Listen müssen nicht wie bei equal-at-some-position vom selben Datentyp sein. Die Abgleichoperation liefert natürlich boolean zurück und fällt daher wieder unter den Fachbegriff *Prädikat*, wieder abgekürzt pred.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: (X Y -> boolean) (list of X) (list of Y) -> boolean
;; Returns: true iff  there is at least one common
;;    position of list1 and list2 with matching values
( define ( match-at-some-position pred list1 list2 )
   ( cond
      [ ( empty? list1 ) #f ]
      [ ( empty? list2 ) #f ]
      [ ( pred ( first list1 ) ( first list2  ) ) #t ]
      [ else ( match-at-some-position
                        pred ( rest list1 ) ( rest list2 ) ) ] ) )
```

Wie bei equal-at-some-position: Falls eine der beiden Listen leer ist, kann es keine Position mit einem Match geben.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: (X Y -> boolean) (list of X) (list of Y) -> boolean
;; Returns: true iff  there is at least one common
;;    position of list1 and list2 with matching values
( define ( match-at-some-position pred list1 list2 )
   ( cond
      [ ( empty? list1 ) #f ]
      [ ( empty? list2 ) #f ]
      [ ( pred ( first list1 ) ( first list2  ) ) #t ]
      [ else ( match-at-some-position
                        pred ( rest list1 ) ( rest list2 ) ) ] ) )
```

Und der Fall, dass zwar beide Listen nicht leer sind, aber an der ersten Position kein Match ist, wird ebenfalls so wie bei equal-at-some-position gehandhabt.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: (X Y -> boolean) (list of X) (list of Y) -> boolean
;; Returns: true iff  there is at least one common
;;    position of list1 and list2 with matching values
( define ( match-at-some-position pred list1 list2 )
   ( cond
      [ ( empty? list1 ) #f ]
      [ ( empty? list2 ) #f ]
      [ ( pred ( first list1 ) ( first list2  ) ) #t ]
      [ else ( match-at-some-position
                        pred ( rest list1 ) ( rest list2 ) ) ] ) )
```

Das ist jetzt anders als bei equal-at-some-position: Anstelle eines konkreten, festen Vergleichsoperators wenden wir hier den Parameter pred an.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( match-at-some-position = number-list1 number-list2 )

;; Type: number student -> boolean
;; Returns: true iff the student‘s fee does not exceed x
( define ( some-fee-larger x stud ) ( > x ( student-fee stud ) ) )

( match-at-some-position
      some-fee-larger max-list student-list )
```

Die generische Funktion match-at-some-position wenden wir jetzt auf zwei Beispiele an.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( match-at-some-position = number-list1 number-list2 )

;; Type: number student -> boolean
;; Returns: true iff the student‘s fee does not exceed x
( define ( some-fee-larger x stud ) ( > x ( student-fee stud ) ) )

( match-at-some-position
      some-fee-larger max-list student-list )
```

Das erste Beispiel ist genau die Spezialisierung auf den Anwendungsfall der ursprünglichen Funktion equal-at-some-position: zwei Listen von Zahlen, der Abgleich ist der Test auf Gleichheit. Bei der zweiten Implementation von equal-at-some-position, auf Strings, würde hier string=? anstelle des Vergleichsoperators stehen.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( match-at-some-position = number-list1 number-list2 )

;; Type: number student -> boolean
;; Returns: true iff the student‘s fee does not exceed x
( define ( some-fee-larger x stud ) ( > x ( student-fee stud ) ) )

( match-at-some-position
      some-fee-larger max-list student-list )
```

Im zweiten Anwendungsfall sind die Listen von unterschiedlichen Typen: eine Liste von Studierenden und eine Liste von Zahlen. Dafür gibt es keine eingebaute Abgleichoperation, wir müssen eine eigene bauen, die das leistet, was wir wollen.

Vergleich zweier Listen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( match-at-some-position = number-list1 number-list2 )

;; Type: number student -> boolean
;; Returns: true iff the student‘s fee does not exceed x
( define ( some-fee-larger x stud ) ( > x ( student-fee stud ) ) )

( match-at-some-position
      some-fee-larger max-list student-list )
```

Und das ist die Abgleichoperation für das zweite Beispiel. Die Rückgabe ist genau dann true, wenn die Gebührenhöhe eines Studierenden höher als der Wert von x ist. Das zweite Beispiel von match-at-some-position leistet also insgesamt Folgendes: Zu einer Liste von Studierenden gibt es eine Liste von individuellen Maximalwerten, und match-at-some-position liefert genau dann true, wenn die Gebührenhöhe mindestens eines Studierenden seinen individuellen Maximalwert überschreitet.

TECHNISCHE UNIVERSITÄT DARMSTADT

Spezialisierungen von generischen Funktionen in Racket

Bei solchen sehr generischen Funktionen wie filter, fold und map muss man eine ganze Reihe von Parametern immer beim Aufruf befüllen. Manchmal will man das gar nicht, sondern einzelne Parameter ein für allemal festlegen und dann bei Aufrufen der Funktion nicht mehr mit angeben müssen. Wie das geht, sehen wir uns jetzt an.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> number )
( define ( func-to-add-constant-value x )
                    ( lambda ( y ) ( + x y ) ) )

( define add10 ( func-to-add-constant-value 10 ) )

( add10 20 )    ;;  30
```

Als erstes ein sehr einfaches Beispiel.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> number )
( define ( func-to-add-constant-value x )
                    ( lambda ( y ) ( + x y ) ) )

( define add10 ( func-to-add-constant-value 10 ) )

( add10 20 )    ;;  30
```

Diese Funktion soll aus einer Zahl eine Funktion konstruieren, die eine Zahl als Parameter erhält und eine Zahl zurückliefert.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> number )
( define ( func-to-add-constant-value x )
                    ( lambda ( y ) ( + x y ) ) )

( define add10 ( func-to-add-constant-value 10 ) )

( add10 20 )    ;;  30
```

Der Parameter x dieser funktionsgenerierenden Funktion wird in den Lambda-Ausdruck gesteckt.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> number )
( define ( func-to-add-constant-value x )
                    ( lambda ( y ) ( + x y ) ) )

( define add10 ( func-to-add-constant-value 10 ) )

( add10 20 )    ;;  30
```

Der Parameter y der zurückgelieferten Funktion wird dann mit x addiert, das ist der Rückgabewert der zurückgelieferten Funktion.

Wir haben also wieder einen Fall von Closure: In der zurückgelieferten Funktion wird das x als unsichtbare zusätzliche Information gespeichert, um es beim Aufruf der zurückgelieferten Funktion auf deren Parameter y zu addieren.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> number )
( define ( func-to-add-constant-value x )
                    ( lambda ( y ) ( + x y ) ) )

( define add10 ( func-to-add-constant-value 10 ) )

( add10 20 )    ;;  30
```

Hier sehen wir ein weiteres Beispiel dafür, dass Konstanten auch funktionswertig sein können. Der definierende Ausdruck für die Konstante add10 liefert ja eine Funktion zurück. Wir sehen einmal mehr: Funktionen sind eine weitere Art von Daten, die im Prinzip analog zu Zahlen, boolean, Symbolen, Strings und Struct-Typen behandelt werden können, jeweils mit den typspezifisch dafür vorgesehenen Operationen. Typspezifisch für Funktionstypen ist der Aufruf mit der richtigen Anzahl von Parametern.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> number )
( define ( func-to-add-constant-value x )
                    ( lambda ( y ) ( + x y ) ) )

( define add10 ( func-to-add-constant-value 10 ) )

( add10 20 )    ;;  30
```

In diesem konkreten Beispiel wird also die 10 intern in der zurückgelieferten Funktion als Summand gespeichert.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> number )
( define ( func-to-add-constant-value x )
                    ( lambda ( y ) ( + x y ) ) )

( define add10 ( func-to-add-constant-value 10 ) )

( add10 20 )    ;;  30
```

Bei diesem Aufruf von add10 hat der Parameter y den Wert 20 und wird mit dem intern gespeicherten Wert addiert. Das Ergebnis dieses Aufrufs von add10 ist also 30.

Die Spezialisierung besteht in diesem Beispiel darin, dass einer der beiden Summanden festgelegt und nur noch der andere variabel ist. Das ist eine durchaus nicht ungewöhnliche Situation, dass man an verschiedenen Stellen die mathematische Verknüpfung einer festen Zahl mit unterschiedlichen Zahlen haben möchte, zum Beispiel Fixkosten plus variable Kosten.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> boolean )
( define ( func-less-than x ) ( lambda ( y ) ( > x y ) ) )


;; Type: number ( list of number ) -> ( list of number )
( define ( filter-less-than x lst )
        ( my-filter ( func-less-than x ) lst ) )
```

Als zweites und abschließendes Beispiel für Spezialisierungen betrachten wir eine Spezialisierung von my-filter, in der die Element der Eingabelisgte Zahlen sein müssen und das Filterprädikat bis auf einen Zahlenwert festgelegt ist. Anstelle der Filterfunktion ist dann dieser Zahlenwert der Parameter der spezialisierten Funktion filter-less-than.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> boolean )
( define ( func-less-than x ) ( lambda ( y ) ( > x y ) ) )


;; Type: number ( list of number ) -> ( list of number )
( define ( filter-less-than x lst )
        ( my-filter ( func-less-than x ) lst ) )
```

So wie eben beim ersten Beispiel, definieren wir eine Funktion, die eine Funktion zurückliefert. Die zurückgelieferte Funktion wird dann das Filterprädikat sein.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> boolean )
( define ( func-less-than x ) ( lambda ( y ) ( > x y ) ) )


;; Type: number ( list of number ) -> ( list of number )
( define ( filter-less-than x lst )
        ( my-filter ( func-less-than x ) lst ) )
```

Wie es bei einem Filterprädikat für die Elemente einer Liste sein muss, hat die zurückgelieferte Funktion einen einzigen Parameter, der vom Typ der Listenelemente sein muss, und die Rückgabe ist boolean.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> boolean )
( define ( func-less-than x ) ( lambda ( y ) ( > x y ) ) )


;; Type: number ( list of number ) -> ( list of number )
( define ( filter-less-than x lst )
        ( my-filter ( func-less-than x ) lst ) )
```

Die zurückgelieferte Funktion vergleicht ihren Parameter y mit dem Wert x. Das Ganze ist völlig analog zu dem ersten Beispiel für Spezialisierungen soeben: Der Wert von x wird in der zurückgelieferten Funktion intern durch Closure gespeichert und beim Aufruf dann mit y verglichen.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> boolean )
( define ( func-less-than x ) ( lambda ( y ) ( > x y ) ) )


;; Type: number ( list of number ) -> ( list of number )
( define ( filter-less-than x lst )
        ( my-filter ( func-less-than x ) lst ) )
```

Der Vertrag der Spezialisierung ist fast identisch mit dem Vertrag der allgemeinen Funktion my-filter. Der einzige Unterschied ist, dass der erste Parameter nicht mehr das Filterprädikat ist, sondern der Zahlenwert x.

Spezialisierungen in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number -> ( number -> boolean )
( define ( func-less-than x ) ( lambda ( y ) ( > x y ) ) )


;; Type: number ( list of number ) -> ( list of number )
( define ( filter-less-than x lst )
        ( my-filter ( func-less-than x ) lst ) )
```

Die allgemeine Funktion my-filter erwartet als ersten Parameter ein Prädikat. Dieser Aufruf der Funktion func-less-than liefert es. Die Funktion filter-less-than ist damit eine Spezialisierung von filter in dem Sinne, dass alle Elemente entfernt werden, die nicht kleiner als x sind.

Damit sind wir mit Lambda-Ausdrücken und Spezialisierungen und mit dem ganzen Kapitel fertig.

TECHNISCHE UNIVERSITÄT DARMSTADT

Methodennamen als Lambda-Ausdrücke

Wir kommen noch zu einer weiteren nützlichen Variation von Lambda-Ausdrücken in Java.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleConsumer {
    void accept ( double x );
}

public class IntPrinter implements DoubleConsumer {
    public void accept ( double x ) {
       System.out.print(x);
    }
}
```

```
DoubleConsumer cons1 = new IntPrinter();
DoubleConsumer cons2 = x -> { System.out.print(x); }
DoubleConsumer cons3 = System.out::print;
```

In Java gibt es noch eine weitere Kurzform speziell für diejenigen Lambda-Ausdrücke, die aus einem einzelnen Methodenaufruf und sonst nichts bestehen. Der Fachbegriff, unter dem Sie dies finden, lautet method reference.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleConsumer {
    void accept ( double x );
}

public class IntPrinter implements DoubleConsumer {
    public void accept ( double x ) {
       System.out.print(x);
    }
}
```

```
DoubleConsumer cons1 = new IntPrinter();
DoubleConsumer cons2 = x -> { System.out.print(x); }
DoubleConsumer cons3 = System.out::print;
```

Auch dieses Functional Interface findet sich in Package java.util.function. Analog zum schon vorher in Kapitel 04c (Fallbeispiel Prädikate) gesehenen Interface IntConsumer heißt die funktionale Methode von DoubleConsumer wieder accept, und der Rückgabetyp ist wieder void. Der einzige Parameter ist hier vom primitiven Datentyp double, wohingegen er bei IntConsumer vom Typ int war.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleConsumer {
    void accept ( double x );
}

public class IntPrinter implements DoubleConsumer {
    public void accept ( double x ) {
       System.out.print(x);
    }
}
```

```
DoubleConsumer cons1 = new IntPrinter();
DoubleConsumer cons2 = x -> { System.out.print(x); }
DoubleConsumer cons3 = System.out::print;
```

Hier sehen Sie drei äquivalente Ausdrücke. Die ersten beiden Formen kennen Sie aus Kapitel 04c, Abschnitt zu Functional Interfaces und Lambda-Ausdrücken. Die dritte Form ist die, um die es in diesem Abschnitt jetzt geht.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleConsumer {
    void accept ( double x );
}

public class IntPrinter implements DoubleConsumer {
    public void accept ( double x ) {
       System.out.print(x);
    }
}
```

```
DoubleConsumer cons1 = new IntPrinter();
DoubleConsumer cons2 = x -> { System.out.print(x); }
DoubleConsumer cons3 = System.out::print;
```

Dieser Lambda-Ausdruck passt zur funktionalen Methode von Interface DoubleConsumer, wobei der Compiler automatisch eruiert, dass x vom primitiven Typ double ist.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleConsumer {
    void accept ( double x );
}

public class IntPrinter implements DoubleConsumer {
    public void accept ( double x ) {
       System.out.print(x);
    }
}
```

```
DoubleConsumer cons1 = new IntPrinter();
DoubleConsumer cons2 = x -> { System.out.print(x); }
DoubleConsumer cons3 = System.out::print;
```

Der Methodenrumpf besteht aus dem Aufruf einer Objektmethode und aus sonst nichts. Das Objekt ist das public-Attribut out von java.lang.System. Dass out selbst eine *Klassen*variable von System ist, ist hier unerheblich.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleConsumer {
    void accept ( double x );
}

public class IntPrinter implements DoubleConsumer {
    public void accept ( double x ) {
       System.out.print(x);
    }
}
```

```
DoubleConsumer cons1 = new IntPrinter();
DoubleConsumer cons2 = x -> { System.out.print(x); }
DoubleConsumer cons3 = System.out::print;
```

Das ist die angekündigte Kurzform namens method reference im Fall einer Objektmethode: Die Referenz auf das Objekt wird mit dem Methodennamen durch einen doppelten Doppelpunkt verbunden.

Die Methode print ist für out überladen. Daraus, dass ein DoubleConsumer herauskommen soll, kann der Compiler aber schließen, dass diejenige Version der Methode print einzusetzen ist, die einen einzelnen Parameter vom primitiven Datentyp double hat.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleBinaryOperator {
    double applyAsDouble ( double x, double y );
}

public class MaxOfTwoDoubles implements DoubleBinaryOperator {
   public double applyAsDouble ( double x, double y ) {
      return Math.max ( x, y );
   }
}
```

```
DoubleBinaryOperator op1 = new MaxOfTwoDoubles();
DoubleBinaryOperator op2 = ( x, y ) -> Math.max(x,y);
DoubleBinaryOperator op3 = Math::max;
```

Dasselbe jetzt mit Klassen- statt Objektmethoden. Die funktionale Methode von DoubleBinaryOperator heißt applyAsDouble, hat zwei Parameter vom primitiven Datentyp double und liefert auch ein double zurück.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleBinaryOperator {
    double applyAsDouble ( double x, double y );
}

public class MaxOfTwoDoubles implements DoubleBinaryOperator {
   public double applyAsDouble ( double x, double y ) {
      return Math.max ( x, y );
   }
}
```

```
DoubleBinaryOperator op1 = new MaxOfTwoDoubles();
DoubleBinaryOperator op2 = ( x, y ) -> Math.max(x,y);
DoubleBinaryOperator op3 = Math::max;
```

***Erinnerung*: Ein Methodenrumpf, der aus einer einzelnen return-Anweisung und sonst nichts besteht, passt genau auf die Kurzform von Lambda-Ausdrücken in Java, in der nur der Ausdruck, dessen Wert mit return zurückzuliefern ist, angegeben wird und sogar die geschweiften Klammern weggelassen werden.**

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleBinaryOperator {
    double applyAsDouble ( double x, double y );
}

public class MaxOfTwoDoubles implements DoubleBinaryOperator {
   public double applyAsDouble ( double x, double y ) {
      return Math.max ( x, y );
   }
}
```

```
DoubleBinaryOperator op1 = new MaxOfTwoDoubles();
DoubleBinaryOperator op2 = ( x, y ) -> Math.max(x,y);
DoubleBinaryOperator op3 = Math::max;
```

In perfekter Korrespondenz zur allgemeinen Handhabung von Objekt- und Klassenmethoden kann auch hier der Name der Klasse anstelle des Namens der Referenz stehen.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

String::new

Auch Operator new kommt hier als Methode in dieses Konzept hinein. Operator new von einer Klasse ist eine Methode, deren Rückgabetyp diese Klasse ist. Die Parameter sind die des Konstruktors. Welcher Konstruktor aufgerufen wird, hängt vom Kontext ab, das sehen wir uns jetzt an.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public X () { ..... }
  public X ( int n ) {
    .....
  }
  public X ( int n, double x ) {
    .....
  }
}
```

```
public interface A { X m1 (); }
public interface B {
  X m2 ( int m );
}
public interface C {
  X m3 ( int m, double y );
}

A a = X::new;
B b = X::new;
C c = X::new;
```

Wieder ein rein illustratives Beispiel. Links sehen Sie eine Klasse mit mehreren Konstruktoren. Rechts sehen Sie drei Functional Interfaces, deren funktionale Methode jeweils X zurückliefert und dieselbe Parameterliste hat wie einer der drei Konstruktoren.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public X () { ..... }
  public X ( int n ) {
     .....
  }
  public X ( int n, double x ) {
     .....
  }
}
```

```
public interface A { X m1 (); }
public interface B {
  X m2 ( int m );
}
public interface C {
  X m3 ( int m, double y );
}

A a = X::new;
B b = X::new;
C c = X::new;
```

Der Konstruktor ohne Parameter wird vom Compiler in einem Kontext eingesetzt, in dem eine funktionale Methode ohne Parameter erwartet wird. Das ist bei Interface A der Fall.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public X () { ..... }
  public X ( int n ) {
     .....
  }
  public X ( int n, double x ) {
     .....
  }
}
```

```
public interface A { X m1 (); }
public interface B {
  X m2 ( int m );
}
public interface C {
  X m3 ( int m, double y );
}

A a = X::new;
B b = X::new;
C c = X::new;
```

Analoges gilt für den zweiten Konstruktor ...

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public X () { ..... }
  public X ( int n ) {
     .....
  }
  public X ( int n, double x ) {
     .....
  }
}
```

```
public interface A { X m1 (); }
public interface B {
  X m2 ( int m );
}
public interface C {
  X m3 ( int m, double y );
}

A a = X::new;
B b = X::new;
C c = X::new;
```

... und natürlich auch für den dritten Konstruktor.

Selbstverständlich funktioniert diese automatische Auswahl nicht nur bei Konstruktoren und Operator new, sondern genauso auch bei normalen überladenen Methoden: Der Kontext bestimmt, welche Version einer Methode hergenommen wird. Das hatten wir schon bei System.out.print weiter vorne in diesem Abschnitt kurz angerissen.

Gibt es keine Methode mit passender Signatur, dann gibt der Compiler eine Fehlermeldung aus.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StringToStringFunction {
   String applyAsString ( String str );
}

public interface StringSupplier {
   String get ();
}
```

```
StringToStringFunction fct1 = String::new;
StringToStringFunction fct2 = X::allUpperCase;
StringSupplier sup1 = String::new;
StringSupplier sup2 = Y::createHelloWorld;
```

Schauen wir uns ein immer noch schematisches, aber doch schon reales Beispiel an.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StringToStringFunction {
   String applyAsString ( String str );
}

public interface StringSupplier {
   String get ();
}
```

```
StringToStringFunction fct1 = String::new;
StringToStringFunction fct2 = X::allUpperCase;
StringSupplier sup1 = String::new;
StringSupplier sup2 = Y::createHelloWorld;
```

Dieses Interface ist repräsentiert ebenfalls Funktionen, in diesem Fall solche, die Strings auf Strings abbilden.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StringToStringFunction {
   String applyAsString ( String str );
}

public interface StringSupplier {
   String get ();
}
```

```
StringToStringFunction fct1 = String::new;
StringToStringFunction fct2 = X::allUpperCase;
StringSupplier sup1 = String::new;
StringSupplier sup2 = Y::createHelloWorld;
```

Entsprechend gibt es im Package java.util.function auch Interfaces, die einen Wert ohne Parameter, also quasi aus dem Nichts berechnen. Der Name eines solchen Interface ist per Konvention immer der Typ des Wertes plus das Wort „Supplier“, und die funktionale Methode heißt per Konvention immer get.

Natürlich wird der Rückgabewert von get in der Regel nicht wirklich aus dem Nichts, sondern aus Daten heraus berechnet, die nicht als Parameter von get gegeben werden, sondern anders ins Spiel kommen, zum Beispiel als private-Attribute der implementierenden Klasse, die im Konstruktor initialisiert werden. Daran ist dieses Interface angelehnt, dessen funktionale Methode einen String zurückliefert.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StringToStringFunction {
   String applyAsString ( String str );
}

public interface StringSupplier {
   String get ();
}
```

```
StringToStringFunction fct1 = String::new;
StringToStringFunction fct2 = X::allUpperCase;
StringSupplier sup1 = String::new;
StringSupplier sup2 = Y::createHelloWorld;
```

Analog zum rein schematischen Beispiel auf der letzten Folie können passende Konstruktoren mit dieser Schreibweise als Lambda-Ausdrücke verwendet werden. Für die farblich unterlegte Zeile erzeugt der Compiler eine anonyme Klasse, die das Interface StringToStringFunction implementiert, und deren funktionale Methode nichts anderes tut als den passenden Konstruktor von Klasse String aufzurufen. Tatsächlich hat Klasse String einen Konstruktor mit einem einzelnen Parameter von Klasse String. Dieser richtet eine Kopie des übergebenen Strings ein.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StringToStringFunction {
   String applyAsString ( String str );
}

public interface StringSupplier {
   String get ();
}
```

```
StringToStringFunction fct1 = String::new;
StringToStringFunction fct2 = X::allUpperCase;
StringSupplier sup1 = String::new;
StringSupplier sup2 = Y::createHelloWorld;
```

Hier passt hingegen ein anderer Konstruktor der Klasse String, nämlich der ohne Parameter. Dieser richtet ein String-Objekt der Länge 0 ein, also eine leere Zeichensequenz.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface StringToStringFunction {
   String applyAsString ( String str );
}

public interface StringSupplier {
   String get ();
}
```

```
StringToStringFunction fct1 = String::new;
StringToStringFunction fct2 = X::allUpperCase;
StringSupplier sup1 = String::new;
StringSupplier sup2 = Y::createHelloWorld;
```

Natürlich müssen die aufgerufenen Methoden nicht immer Konstruktoren der Klasse String sein. Jede Methode, die auf die funktionale Methode des jeweiligen Interface passt, kann verwendet werden, um Strings zu erzeugen. Dadurch ergibt sich praktisch beliebige Freiheit, wie man Strings erzeugen kann. In den farblich unterlegten Zeilen wird dies beispielhaft demonstriert, Details schauen wir uns auf der nächsten Folie an.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public String allUpperCase ( String str ) {
      return str.toUpperCase();
   }
}
```

```
StringToStringFunction fct2 = X::allUpperCase;
```

```
public class Y {
   public String createHelloWorld () {
      return new String ( “Hello World“ );
   }
}
```

```
StringSupplier sup2 = Y::createHelloWorld;
```

Die beiden farblich unterlegten Texte sind ohne Veränderung von der letzten Folie kopiert. Links sehen Sie zwei Klassen X und Y, die darauf passen würden.

Im oberen Beispiel wird der Parameter nicht einfach kopiert, sondern es wird eine Kopie erstellt, bei der alle Kleinbuchstaben durch Großbuchstaben ersetzt sind. Im unteren Beispiel wird nicht ein leerer String erzeugt, sondern ein String mit Inhalt „Hello World“, also mit 11 fest vorgegebenen Zeichen. Natürlich könnte man die Klasse Y so gestalten, dass nicht ein fester String zurückgeliefert wird, sondern ein String, der beispielsweise wieder durch Attribute der Klasse Y definiert ist.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

String::new

String[ ]::new

Es gibt ja auch Operator new bei Arrays: Dies ist eine Methode mit einem einzelnen Parameter vom Typ int, die Länge des zu erzeugenden Arrays. Rückgabetyp dieser Methode ist Array von String, und zurückgeliefert wird das erzeugte Array.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToStringArrayFunction {
   String[ ] applyAsStringArray ( int n );
}
```

```
IntToStringArrayFunction fct1 = String[ ]::new;
IntToStringArrayFunction fct2 = Z::createIndexStringArray;
```

Um uns das jetzt genauer anzusehen, modifizieren wir das Interface IntToStringFunction leicht.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToStringArrayFunction {
   String[ ] applyAsStringArray ( int n );
}
```

```
IntToStringArrayFunction fct1 = String[ ]::new;
IntToStringArrayFunction fct2 = Z::createIndexStringArray;
```

Die funktionale Methode liefert nun nicht mehr einen String, sondern ein *Array* von Strings. Entsprechend ändern sich die Namen des Interface und der funktionalen Methode im Rahmen der im Zusammenhang mit der letzten Folie schon genannten Konvention.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToStringArrayFunction {
   String[ ] applyAsStringArray ( int n );
}
```

```
IntToStringArrayFunction fct1 = String[ ]::new;
IntToStringArrayFunction fct2 = Z::createIndexStringArray;
```

Der Lambda-Ausdruck sieht dann so aus. Dieser Lambda-Ausdruck passt, wenn wie hier die funktionale Methode genau einen Parameter hat und dieser vom Typ int ist, und wenn die funktionale Methode ein String-Array zurückliefert.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntToStringArrayFunction {
   String[ ] applyAsStringArray ( int n );
}
```

```
IntToStringArrayFunction fct1 = String[ ]::new;
IntToStringArrayFunction fct2 = Z::createIndexStringArray;
```

Auch hier wieder ein Beispiel dafür, dass man auch ganz andere Methoden verwenden könnte. Die Details schauen wir uns auf der nächsten Folie an.

Methodennamen als Lambda

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Z {
   public String[ ] createIndexStringArray ( int n ) {
      String[ ] result = new String [ n ];
      for ( int i = 0; i < n; i++ )
         result[i] = Integer.toString ( i );
  }
   return result;
}
```

```
IntToStringArrayFunction fct2 = Z::createIndexStringArray;
```

Unten sehen Sie nochmals die auf der letzten Folie gezeigte Methode einer Klasse Z. Ihrem Namen nach könnte sie beispielsweise ein String-Array zurückliefern, bei dem jede Komponente eine String-Repräsentation ihres Index ist. Diese Logik ist in der Klasse Z oben realisiert.