



2.

Auflage



Herbert Prähofer

# Funktionale Programmierung in Java und Kotlin

Eine umfassende Einführung

**dpunkt.verlag**



# Inhalt

**Cover**

**Hinweise zur Benutzung**

**Titel**

**Impressum**

**Inhalt**

**Vorwort**

## **1 Einleitung**

1.1 Elementare Konzepte und Begriffe

1.2 Funktionale Programmierung in Java

1.3 Kotlin als Sprache für eine funktionale Programmierung

## **Teil I Grundlagen der funktionalen Programmierung in Java**

2 Sprachliche Grundlagen von Java

2.1 Generics

2.1.1 Typparameter

2.1.2 Typconstraints

2.1.3 Ko- und Kontravarianz

2.1.4 Typinferenz bei Generics

2.1.5 Schwachstellen der Generics in Java

2.2 Default-Methoden

2.3 Lambda-Ausdrücke

2.3.1 Funktionale Interfaces

2.3.2 Methodenreferenzen

2.4 Record-Klassen

2.5 Switch-Ausdrücke

2.6 Zusammenfassung

### 3 Programmieren ohne Seiteneffekte

#### 3.1 Reine Funktionen

##### 3.1.1 Iteration vs. Rekursion

##### 3.1.2 Referenzielle Transparenz und Ersetzungsprinzip

#### 3.2 Funktionale Ausnahmebehandlung mit Optional

#### 3.3 Zusammenfassung

### 4 Funktionale Datenstrukturen

#### 4.1 Algebraische Datentypen

#### 4.2 Pattern Matching

#### 4.3 Paare und Tupel

#### 4.4 Funktionale Listen

#### 4.5 Persistente Collections

#### 4.6 Zusammenfassung

## **Teil II Gestaltungsprinzipien und Entwurfsmuster**

### 5 Programmieren mit Funktionsparametern

#### 5.1 Höhere Funktionen für Collections

##### 5.1.1 Höhere Funktionen bei persistenten Collections

#### 5.2 Flexible Programmschnittstellen

#### 5.3 Algorithmen

##### 5.3.1 Tiefensuche

##### 5.3.2 Verallgemeinerung der Suche

#### 5.4 Entwurfsmuster

##### 5.4.1 Strategie

##### 5.4.2 Kommando

#### 5.5 Eingebettete und bedingte Ausführung

##### 5.5.1 Eingebetteter Code

##### 5.5.2 Bedingte Ausführung

#### 5.6 Auswertung nach Bedarf

- 5.6.1 Faule Iteratoren
- 5.6.2 Unendliche Folgen
- 5.6.3 Faule Iteration über die Knoten eines Graphen
- 5.7 Zusammenfassung
- 6 Reduktion mit Monoiden
  - 6.1 Monoide
  - 6.2 Reduktion
  - 6.3 Monoide in Java
  - 6.4 Reduzierbare Strukturen
- 6.4.1 Reducible
- 6.4.2 Abgeleitete Operationen
  - 6.5 Heben von Monoiden auf Container-Strukturen
- 6.5.1 Reduktion von Optional-Werten
- 6.5.2 Reduktion von Map-Einträgen
  - 6.6 Parallele Reduktion
  - 6.7 Zusammenfassung
- 7 Funktionsketten mit Monaden
  - 7.1 Flüssige Schnittstellen
  - 7.2 Funktoren
    - 7.2.1 Funktor Optional
    - 7.2.2 Gesetze und Eigenschaften
  - 7.3 Monaden
    - 7.3.1 Monade Optional
    - 7.3.2 Gesetze
    - 7.3.3 Bedeutung von Monaden
    - 7.3.4 MonadPlus: Monade mit monoider Kombination
  - 7.4 Zusammenfassung
- 8 Funktionskomposition

## 8.1 Funktionskomposition bei funktionalen Interfaces

### 8.1.1 Funktionskomposition mit Function

### 8.1.2 Logische Verknüpfungen bei Predicate

### 8.1.3 Bilden von Vergleichsketten mit Comparator

## 8.2 Generatoren für Zufallswerte

### 8.2.1 Monade Gen

### 8.2.2 Erzeugen von Generatoren mit map

### 8.2.3 Generatoren für Listen und Wörter

### 8.2.4 Unendliche Iteratoren für Zufallswerte

## 8.3 Kombinator-Parser

### 8.3.1 Parser und Parser-Ergebnisse

### 8.3.2 Monade Parser

### 8.3.3 Kombinationsoperatoren

### 8.3.4 Parser für arithmetische Ausdrücke

## 8.4 Zusammenfassung

## **Teil III Funktionale Systeme und Bibliotheken**

## 9 Streams

### 9.1 Grundlagen von Streams

#### 9.1.1 Ein erstes Beispiel

#### 9.1.2 Externe vs. interne Iteration

#### 9.1.3 Bedarfsauswertung

### 9.2 Klassen von Streams

### 9.3 Stream-Operationen

#### 9.3.1 Erzeugeroperationen

#### 9.3.2 Zwischenoperationen

#### 9.3.3 Terminaloperationen

### 9.4 Collectors

#### 9.4.1 Interface Collector

- 9.4.2 Vordefinierte Collectors
- 9.4.3 Downstream Collectors
- 9.4.4 Eine eigene Collector-Implementierung
  - 9.5 Anwendungsbeispiele
  - 9.5.1 Ergebnisauswertung mit Streams
  - 9.5.2 Wortindex zu einem Text
    - 9.6 Hinweise
    - 9.6.1 Einmal-Iteration
    - 9.6.2 Begrenzung von unendlichen Streams
    - 9.6.3 Zustandslose und zustandsbehaftete Operationen
    - 9.6.4 Reihenfolge von Operationen
    - 9.6.5 Kombinationen von Operationen
    - 9.7 Interne Implementierung
    - 9.7.1 Beispiel
    - 9.8 Zusammenfassung
- 10 Parallele Streams
  - 10.1 Erzeugen von parallelen Streams
  - 10.2 Parallele Ausführung
    - 10.2.1 Spliterators
    - 10.2.2 Ausführung mit Fork/Join-Framework
      - 10.3 Bedingungen bei paralleler Ausführung
      - 10.3.1 Parallele Ausführung und Seiteneffekte
      - 10.3.2 Eigenschaften der Parameter von reduce
      - 10.3.3 Paralleles Sammeln
      - 10.4 Laufzeit
      - 10.5 Zusammenfassung
- 11 Asynchrone Funktionsketten
  - 11.1 Asynchrone Lösung mit Futures

11.2 CompletableFuture

11.3 Kombination von CompletableFutures

11.4 Beispiel Flugsuche

11.5 Zusammenfassung

12 Reaktive Streams

12.1 Grundlagen

12.1.1 Kontrakt von Observable

12.1.2 Erzeugen von Observables

12.1.3 Anmelden und Abmelden von Observer

12.2 Varianten

12.2.1 Single

12.2.2 Completable

12.2.3 Maybe

12.3 Hot und Cold Observables

12.3.1 ConnectableObservable

12.3.2 Beispiel Echtzeitdaten

12.4 Operationen

12.4.1 Abbildungen

12.4.2 Filtern und Teilmengen

12.4.3 Reduktion und Suche

12.4.4 Sammeln

12.4.5 Operationen mit Zeit

12.4.6 Kombinationen

12.4.7 Konvertierungen

12.4.8 Seiteneffekte

12.5 Asynchrone Ausführung

12.5.1 Serialisierung von nebenläufigen Ereignissen

12.5.2 subscribeOn und Scheduler

12.5.3 observeOn

12.6 Fehlerbehandlung

12.6.1 Fehlerereignisse auslösen

12.6.2 Auf Fehler reagieren

12.7 Rückstau und Flusskontrolle

12.7.1 Reduktion der Menge der Ereignisse

12.7.2 Flowables

12.8 Testen reaktiver Streams

12.9 Zusammenfassung

13 Testen mit und von Funktionen

13.1 Funktionsparameter bei JUnit 5

13.2 AssertJ: Eine DSL für Unit-Tests

13.3 Eigenschaftsbasiertes Testen nach QuickCheck

13.3.1 Tests

13.3.2 Shrinken der Werte

13.4 Zusammenfassung

## **Teil IV Funktionale Programmierung in Kotlin**

14 Einführung in die Sprache Kotlin

14.1 Dateien, Packages und Deklarationen

14.2 Basisdatentypen und Arrays

14.3 Funktionen

14.4 Variablen, Ausdrücke und Kontrollstrukturen

14.4.1 Variablen

14.4.2 Kontrollstrukturen

14.5 Klassen, Interfaces und Objekte

14.5.1 Klassen

14.5.2 Interfaces

14.5.3 Objekte

- 14.5.4 Companion-Objekte
- 14.5.5 Spezielle Arten von Klassen
  - 14.6 Das Typsystem von Kotlin
    - 14.6.1 Verband der Typen und Typinferenz
    - 14.6.2 Nullable- und Non-Nullable-Typen
    - 14.6.3 Typtest und Typecasts
    - 14.7 Generics
      - 14.7.1 Typparameter
      - 14.7.2 Typconstraints
      - 14.7.3 Ko- und Kontravarianz
      - 14.7.4 Star-Projektion
      - 14.7.5 Reifizierte Typparameter
    - 14.8 Erweiterungsfunktionen und Erweiterungsproperties
    - 14.9 Delegates
      - 14.9.1 Property-Delegates
      - 14.9.2 Klassen-Delegates
    - 14.10 Zusammenfassung
- 15 Funktionale Datenstrukturen in Kotlin
  - 15.1 Datenklassen
  - 15.2 Paare und Tripel
  - 15.3 Algebraische Datentypen mit Datenklassen
  - 15.4 Pattern Matching
  - 15.5 Funktionale Liste in Kotlin
  - 15.6 Collections in Kotlin
  - 15.7 Zusammenfassung
- 16 Lambda-Ausdrücke und höhere Funktionen
  - 16.1 Formen von Lambda-Ausdrücken
  - 16.2 Funktionsreferenzen und anonyme Funktionen

- 16.3 Closures
- 16.4 Inlining Lambda-Ausdrücke
- 16.5 Java-Interoperabilität und SAM-Interfaces
- 16.6 Höhere Funktionen für FList
- 16.7 Zusammenfassung
- 17 Höhere Funktionen für Collections
  - 17.1 Einführung
  - 17.2 Abbilden, Filtern, Sortieren, Teilmengen
  - 17.3 Reduktion
  - 17.4 Suche und Quantoren
  - 17.5 Bilden von Maps
  - 17.6 Gruppieren und Aufteilen
  - 17.7 Weitere Operationen
  - 17.8 Sequences
  - 17.9 Zusammenfassung
- 18 Lambda-Ausdrücke mit Empfänger
  - 18.1 Grundlagen
  - 18.2 Varianten und Kombinationen
  - 18.3 Scope-Funktionen
  - 18.4 Hierarchische Builder
    - 18.4.1 Ein Builder für Trees
    - 18.4.2 Ein Builder für hierarchische Dokumente
  - 18.5 Zusammenfassung
- 19 Domänenspezifische Sprachen
  - 19.1 Fallbeispiel physikalische Einheiten
  - 19.2 Fallbeispiel Parser-DSL
    - 19.2.1 Parser und Result
    - 19.2.2 Funktionen und Kombinationsoperatoren

- 19.2.3 DSL für Parser-Spezifikationen
  - 19.3 Fallbeispiel Zustandsmaschinen
  - 19.4 Zusammenfassung
- 20 Gestaltung von Systemen und Frameworks
  - 20.1 Koroutinen
    - 20.1.1 Suspend-Funktionen und Koroutinen
    - 20.1.2 Strukturierte Nebenläufigkeit
      - 20.2 Flows
      - 20.3 Jetpack Compose
      - 20.4 Ktor
      - 20.5 Zusammenfassung

## **A Bibliografie**

## **B Laufzeitexperimente Parallele Streams**

- B.1 Experimentelle Anordnungen
- B.2 Experimente

## **Index**

## **Über den Autor**

## 4 Funktionale Datenstrukturen

In der Einleitung haben wir festgehalten, dass die rein funktionale Programmierung ausschließlich mit unveränderlichen Datenstrukturen arbeitet. Sprachen wie Haskell und Scala haben gezeigt, dass dies prinzipiell möglich ist und auch Vorteile bietet. In diesem Kapitel wollen wir untersuchen, wie funktionale Datenstrukturen in Java implementiert und verwendet werden können.

Zunächst wird ein Ansatz zur Implementierung algebraischer Datentypen vorgestellt. Dann sehen wir, wie Pattern Matching in Java unterstützt wird. Anschließend werden wir uns mit der Implementierung von grundlegenden funktionalen Datenstrukturen beschäftigen und schließlich die Implementierung von persistenten Collections diskutieren.

### 4.1 Algebraische Datentypen

In [Kapitel 1](#) wurden algebraische Datentypen als zentrales Konzept der funktionalen Programmierung vorgestellt. In [Listing 1.1](#) haben wir dazu beispielhaft einen algebraischen Datentyp `Expr` für arithmetische Ausdrücke in Haskell eingeführt. Der algebraische Datentyp `Expr` hat mehrere Varianten, wobei jede Variante durch ein Produkt definiert ist. Ein Produkt besteht aus einem Daten-Tag und den Typen der Felder. Beispielsweise wird die Additionsoperation mit dem Daten-Tag `Add` eingeleitet und definiert zwei Felder, die wiederum vom Typ `Expr` sind.

Die wesentlichen Eigenschaften eines algebraischen Datentyps werden im Folgenden noch einmal angeführt:

- Die Daten sind unveränderlich.
- Datenkapselung wird nicht unterstützt, das heißt, alle Felder sind öffentlich und können direkt gelesen werden.
- Es gibt nur die Varianten des Datentyps, die in der Definition angegeben sind. Man sagt, der Datentyp ist geschlossen.

Java bietet mit den Record-Klassen in Kombination mit der Implementierung eines Interface eine Möglichkeit, einen Datentyp mit diesen Eigenschaften zu

implementieren. Dabei entsprechen die Record-Klassen den Produkten, während das Interface den Datentyp mit seinen Varianten repräsentiert. Varianten werden in Java also durch die Ableitung von einem Interface gebildet.

Listing 4.1 zeigt eine erste Implementierung des algebraischen Datentyps Expr. Das Interface Expr definiert den Datentyp. Die das Interface implementierenden Record-Klassen bilden die Varianten von Expr.

```
public interface Expr { }
record Val(double value) implements Expr { }
record Vbl(String name) implements Expr { }
record Add(Expr left, Expr right) implements
Expr { }
record Mult(Expr left, Expr right) implements
Expr { }
record Minus(Expr sub) implements Expr { }
record Recip(Expr sub) implements Expr { }
```

**Listing 4.1** ADT Expr in Java: Version 1

Die Definition entspricht bereits weitgehend dem algebraischen Datentyp aus Listing 1.1. **Record-Klassen** definieren unveränderliche Datenobjekte mit Feldern, auf die mit öffentlichen Methoden zugegriffen werden kann.

Allerdings ist es in dieser Version weiterhin möglich, zusätzliche Klassen von Expr abzuleiten. Das bedeutet, dass der Datentyp Expr noch nicht geschlossen ist. Um die Klassen einzuschränken, wurden in Java die Schlüsselwörter `sealed` und `permits` neu eingeführt:

- Mit `sealed` wird eine Klasse oder ein Interface gekennzeichnet, sodass ihre Unterklassen eingeschränkt werden können.
- Mit `permits` müssen dann die erlaubten Unterklassen einer Klasse oder eines Interface, das als `sealed` deklariert wurde, explizit angegeben werden.

Dies wird in Listing 4.2 für das Interface Expr gezeigt. Das Interface Expr ist mit `sealed` markiert und `permits` listet alle erlaubten implementierenden Klassen auf. Dadurch sind keine weiteren Implementierungen des Interface Expr erlaubt.

Die Definition aus Listing 4.2 löst auch noch ein zweites Problem aus der Definition aus Listing 4.1. In der ersten Version sind die Record-Klassen nicht

`public`, da Java verlangt, dass jede öffentliche Klasse in einer eigenen Datei definiert wird. In [Listing 4.2](#) werden die Record-Klassen als innere Klassen des Interface `Expr` definiert. Dies ermöglicht eine kompakte Definition, bei der alle Klassen automatisch `public` sind. Die Konsequenz ist jedoch, dass auf die inneren Klassen mit dem Präfix `Expr.` zugegriffen werden muss.

Bei inneren Klassen könnte man auch auf die `permits`-Klausel verzichten. Wenn ein Interface oder eine Klasse mit `sealed` markiert ist und keine `permits`-Klausel angegeben ist, sind nur die in derselben Datei definierten Unterklassen erlaubt.

```
public sealed interface Expr
    permits Expr.Val, Expr.Vbl, Expr.Add,
           Expr.Mult, Expr.Minus, Expr.Recip {
    record Val(double value) implements Expr {}
    record Vbl(String name) implements Expr { }
    record Add(Expr left, Expr right) implements
    Expr { }
    record Mult(Expr left, Expr right) implements
    Expr { }
    record Minus(Expr sub) implements Expr { }
    record Recip(Expr sub) implements Expr { }
}
```

**Listing 4.2** *ADT Expr in Java: Version 2*

Damit haben wir nun eine elegante Möglichkeit, algebraische Datentypen in Java zu implementieren. Mit `sealed` und `permits` können wir zudem die Varianten eines algebraischen Datentyps einschränken. Doch warum ist das überhaupt wünschenswert? Beim Pattern Matching ist es von Vorteil, wenn die Varianten eines Datentyps bekannt sind. Dies wird im nächsten Abschnitt genauer gezeigt.

## 4.2 Pattern Matching

In [Abschnitt 1.1](#) haben wir auch Pattern Matching für algebraische Datentypen beschrieben. Pattern Matching ermöglicht die Unterscheidung der Varianten eines algebraischen Datentyps. Zudem können in Patterns Variablen verwendet werden, an die die Feldwerte beim Matchen gebunden werden. Als Beispiel

haben wir in Haskell eine Funktion `eval` für den algebraischen Datentyp `Expr` definiert.

Nach dem Vorbild von Haskell und Scala [23], [24] wurde Pattern Matching in Java eingeführt. In mehreren Releases von Java wurden mehrere Versionen von Pattern Matching implementiert [44]. Bereits in Version 16 wurde ein sogenanntes »Pattern Matching mit `instanceof`« eingeführt. Später folgte Pattern Matching mit Switch-Anweisungen und Switch-Ausdrücken und schließlich Record Patterns. Im Folgenden werden diese drei Varianten anhand von Implementierungen der Methode `eval` für `Expr` vorgestellt:

```
static double eval(Expr expr, Map<String,
Double> bds)
```

Die Methode evaluiert einen Ausdruck `expr` unter Verwendung einer `Map` `bds`, die Bindungen von Variablennamen zu ihren Werten speichert. Die Methode ist damit vergleichbar zur Haskell-Funktion `eval` aus [Abschnitt 1.1](#).

### Pattern Matching mit `instanceof`

Der Java-Operator `instanceof` prüft, ob ein Wert von einem bestimmten Typ ist. Beispielsweise können wir mit `instanceof` testen, ob der Wert einer Variablen `expr`, die mit dem Typ `Expr` deklariert wurde, vom Typ `Expr.Var` ist. Sehr oft wird man dann eine neue Variable einführen, einen Typecast durchführen und dann auf die spezifischen Eigenschaften des Werts zugreifen, wie hier mit der Variablen `vbl` gezeigt:

```
Expr expr = ...;
if (expr instanceof Expr.Vbl) {
    Expr.Vbl vbl = (Expr.Vbl) expr;
    String name = vbl.name();
    ...
}
```

Für die Kombination von Typtest und Typecast gibt es nun eine neue Variante, die als »instanceof mit Pattern Matching« bezeichnet wird: Nach dem Typ für die Prüfung folgt eine »Pattern-Variable«. Wenn der Test erfolgreich war, wird der Wert dieser Variablen zugewiesen und man kann dann in Folge mit dieser spezifischen Variablen arbeiten:

```
if (expr instanceof Expr.Vbl vbl) {
```

```

    String name = vbl.name();
    ...
}

```

So ist es möglich, diese Prüfung mit anderen Prüfungen zu kombinieren und dabei die Pattern-Variable bereits zu verwenden. Beispielsweise kann nach der Typprüfung wie oben eine Prüfung, ob für den Variablennamen ein Eintrag in der Map `bds` existiert, angeschlossen werden. Das bedeutet, die Variable ist gültig, sobald die Prüfung mit `instanceof` erfolgreich war:

```

if (expr instanceof Expr.Vbl vbl &&
    bds.containsKey(vbl.name())) {...}

```

Mit dem `instanceof` mit Pattern Matching können wir die Methode `eval` für Expr-Werte wie in [Listing 4.3](#) gezeigt implementieren. Dabei wird davon ausgegangen, dass die inneren Klassen von Expr durch einen statischen Import importiert wurden und daher eine Qualifizierung mit dem Präfix Expr nicht mehr notwendig ist:

```

import static at.jku.ssw.fp.sect04_1.Expr.*;

```

In einer If-Kaskade werden die verschiedenen Fälle unterschieden und die Werte berechnet. Der zweite Fall behandelt Vbl-Werte, für die eine Bindung in der Map existiert. Der dritte Fall betrifft Vbl-Werte ohne eine solche Bindung. In diesem Fall kann kein Wert berechnet werden und es wird eine Exception geworfen<sup>1</sup>. Auch in der If-Kaskade benötigen wir einen Else-Zweig, da sonst der Compiler einen Fehler meldet, dass nicht alle Fälle abgedeckt sind und nicht immer ein Wert zurückgegeben wird.

```

static double eval(Expr expr, Map<String,
Double> bds) {
    if (expr instanceof Val val) {
        return val.value();
    } else if (expr instanceof Vbl vbl &&
bds.containsKey(vbl.name())) {
        return bds.get(vbl.name());
    } else if (expr instanceof Vbl vbl) {
        throw new
NoSuchElementException(vbl.name());
    }
}

```

```

    } else if (expr instanceof Add add) {
        return eval(add.left(), bds) +
            eval(add.right(), bds);
    } else if (expr instanceof Mult mult) {
        return eval(mult.left(), bds) *
            eval(mult.right(), bds);
    } else if (expr instanceof Minus minus) {
        return -eval(minus.sub(), bds);
    } else if (expr instanceof Recip recip) {
        double subVal = eval(recip.sub(), bds);
        if (subVal == 0.0) throw new
            ArithmeticException("/ by zero");
        return 1.0 / subVal;
    } else {
        throw new IllegalStateException("Matching
            error");
    }
}

```

**Listing 4.3** Methode eval mit Pattern Matching mit instanceof

### Switch mit Type Patterns

Typprüfungen können nun auch mit Switch-Anweisungen und Switch-Ausdrücken durchgeführt werden. Ein Fall wird dabei durch Angabe des Typs und einer Pattern-Variablen angegeben. Man spricht von einem *Type Pattern*.

Im folgenden Switch-Ausdruck wird geprüft, ob der Wert von `expr` vom Typ `Val` bzw. `Vbl` ist. Es ist auch möglich, mit einem neuen Schlüsselwort `when` eine zusätzliche Bedingung anzugeben. Dies wird im zweiten Fall deutlich, wenn geprüft wird, ob für den Variablennamen ein Eintrag in `bds` existiert.

```

switch (expr) {
    case Val val -> val.value();
    case Vbl vbl when bds.containsKey(vbl.name())
        -> bds.get(vbl.name());
}

```

```

    case Vbl vbl -> throw new
      NoSuchElementException(vbl.name());
    ...
  }

```

Listing 4.4 zeigt die vollständige Implementierung von `eval` mit einem Switch-Ausdruck. Der Code ist nun wesentlich übersichtlicher als die Implementierung mit `instanceof`. Auch muss hier kein Default-Fall angegeben werden. Durch die Deklaration von `Expr` als `sealed` kann der Compiler überprüfen, dass alle Fälle im Switch abgedeckt sind und somit kein Default-Fall benötigt wird.

```

double eval(Expr expr, Map<String, Double> bds)
{
    return switch (expr) {
        case Val val -> val.value();
        case Vbl vbl
            when bds.containsKey(vbl.name()) ->
                bds.get(vbl.name());
        case Vbl vbl ->
            throw new
                NoSuchElementException(vbl.name());
        case Add add -> eval(add.left(), bds) +
            eval(add.right(), bds);
        case Mult mult -> eval(mult.left(), bds)
            * eval(mult.right(), bds);
        case Minus minus -> -eval(minus.sub(),
            bds);
        case Recip recip -> {
            double sResult = eval(recip.sub(),
                bds);
            if (sResult == 0.0) throw new
                ArithmeticException("/ by zero");
            yield 1.0 / sResult;
        }
    }
}

```

```
    };  
}
```

#### **Listing 4.4** Methode `eval` mit `Switch` mit `Type Patterns`

Der kritische Leser könnte bei diesem Beispiel vielleicht anmerken, dass man die Methode `eval` viel besser mit einer dynamisch gebundenen Methode in den `Expr`-Klassen implementieren kann. Man definiert also eine abstrakte Methode `eval` im Interface und implementiert sie entsprechend in den einzelnen Klassen. Über den dynamischen Dispatch wird dann die richtige Methode vom System aufgerufen.

In vielen Fällen ist dies wohl auch der bevorzugte Ansatz in Java. Allerdings muss die dynamisch gebundene Methode im Klassensystem selbst vorgesehen sein. Wenn wir das Klassensystem nicht erweitern können oder wollen, müssen wir die Methode außerhalb des Klassensystems implementieren. Bisher gab es in Java keine einfache Möglichkeit der Typunterscheidung. Eine häufig verwendete Variante war das Visitor-Pattern [27], eine elegante, aber auch teure und komplizierte Lösung. Mit `Switch` und `Type Patterns` gibt es nun eine einfache und übersichtliche Möglichkeit, um die Varianten eines Typs zu unterscheiden und entsprechend zu reagieren.

#### **Pattern Matching mit Record Patterns**

Die Variante, die dem Pattern Matching in Haskell am nächsten kommt, verwendet *Record Patterns*, die nur für Record-Klassen verwendet werden können. Sie werden analog zu Record-Klassen gebildet und bestehen im einfachsten Fall aus dem Namen der Klasse und Pattern-Variablen für die Komponenten. Die folgenden Beispiele zeigen Record Patterns für die Klassen `Vbl` und `Add`:

```
Vbl(String n)  
Add(Expr l, Expr r)
```

Matching erfolgt, indem die Klasse verglichen wird und die Werte der Komponenten an die Pattern-Variablen gebunden werden. Dazu werden die Getter-Methoden für die Komponenten verwendet. Das heißt, nach der Typprüfung wird auf die Werte der Komponenten zugegriffen und den Pattern-Variablen zugewiesen.

[Listing 4.5](#) zeigt die Implementierung von `eval` mit Record Patterns. Diese erscheint nun nochmals kompakter und übersichtlicher als die Implementierung in [Listing 4.4](#).

```

public static double eval(Expr expr, Map<String,
Double> bds) {
    return switch (expr) {
        case Val(double value) -> value;
        case Vbl(String n) when bds.containsKey(n) -
        > bds.get(n);
        case Vbl(String n) -> throw new
        NoSuchElementException(n);
        case Add(Expr l, Expr r) -> eval(l, bds) +
        eval(r, bds);
        case Mult(var l, var r) -> eval(l, bds) *
        eval(r, bds);
        case Minus(var s) -> -eval(s, bds);
        case Recip(var s) -> {
            double sResult = eval(s, bds);
            if (sResult == 0.0) throw new
            ArithmeticException("/ by zero");
            yield 1.0 / sResult;
        }
    };
}

```

**Listing 4.5** Methode eval mit Switch mit Record Patterns

Record Patterns erlauben auch rekursive Patterns. Das bedeutet, dass ein Pattern erneut auf eine Komponente angewendet werden kann. Zum Beispiel entspricht das folgende Pattern einem Objekt vom Typ Add, bei der der rechte Unterausdruck ein Val-Wert ist:

```
Add(Expr l, Val(double v))
```

Durch rekursive Patterns und When-Klauseln lassen sich komplexe Matches formulieren. Im Folgenden wird als Beispiel eine Methode simplify implementiert, die einen Ausdruck anhand folgender Regeln vereinfacht:

- $expr + 0.0 = expr$  und  $0.0 + expr = expr$ : Eine Addition mit 0.0 ergibt den anderen Ausdruck.

- $\text{expr} * 0.0 = 0.0$  und  $0.0 * \text{expr} = 0.0$ : Eine Multiplikation mit 0.0 ist immer 0.0.
- $\text{expr} * 1.0 = \text{expr}$  und  $1.0 * \text{expr} = \text{expr}$ : Eine Multiplikation mit 1.0 ergibt den anderen Ausdruck.
- $- (- \text{expr}) = \text{expr}$ : Ein Ausdruck zweimal negiert ist der Ausdruck selbst.
- $1.0 / (1.0 / \text{expr}) = \text{expr}$ : Ein Ausdruck zweimal reziprok genommen ist der Ausdruck selbst.

Zusätzlich sollen Ausdrücke mit konkreten Werten ausgewertet werden.

Listing 4.6 zeigt die Implementierung von `simplify` mit rekursiven Patterns. Die Vereinfachungen für `Add` und `Minus` sind dargestellt, die Implementierungen für `Mult` und `Recip` sind analog. Betrachten wir die Fälle:

- `Val` und `Vbl` lassen sich nicht vereinfachen. Wir verwenden ein `Type` Pattern und geben die Ausdrücke unverändert zurück.
- Bei einem `Add`-Ausdruck werden zunächst die Unterausdrücke rekursiv vereinfacht und mit ihnen ein `Add` mit den vereinfachten Unterausdrücken gebildet (`addSimpl`). Dann werden die Fälle des vereinfachten `Adds` unterschieden:
  - Sind beide Unterausdrücke `Val`-Werte, wird ein `Val` mit der Summe zurückgegeben. Man beachte das rekursive Pattern für das `Add` mit den Patterns für die beiden `Val`-Werte.
  - Wenn der zweite Unterausdruck ein `Val` und ihr Wert gleich `0.0` ist, ist das Ergebnis der linke vereinfachte Unterausdruck (`lS`).
  - Der nächste Fall deckt die Variante ab, bei der der linke Unterausdruck ein `Val` mit einem Wert `0.0` ist.
  - Im Standardfall wird die Addition mit den beiden vereinfachten Unterausdrücken zurückgegeben.
- Die Vereinfachung für `Minus` ist ähnlich. Zuerst wird ein `Minus` mit dem bereits vereinfachten Unterausdruck gebildet. Für diesen werden dann die Fälle für die Vereinfachungen unterschieden:
  - Ist der Unterausdruck ein `Val`, so wird ein `Val` mit dem negierten Wert zurückgegeben.
  - Wenn der Unterausdruck wieder ein `Minus` ist, wird der Unterausdruck des inneren `Minus` zurückgegeben.
  - Andernfalls ist das Ergebnis ein `Minus` mit dem vereinfachten Unterausdruck.

```
public static Expr simplify(Expr expr) {
    return switch (expr) {
```

```

case Val val -> val;
case Vbl vbl -> vbl;
case Add(Expr l, Expr r) -> {
    Add addSimpl = new Add(simplify(l),
        simplify(r));
    yield switch (addSimpl) {
        case Add(Val(double lV), Val(double rV))
            -> new Val(lV + rV);
        case Add(Expr lS, Val(double rV)) when rV
            == 0.0 -> lS;
        case Add(Val(double lV), Expr rS) when lV
            == 0.0 -> rS;
        default -> addSimpl;
    };
}
case Minus(Expr s) -> {
    Minus minusSimpl = new Minus(simplify(s));
    yield switch (minusSimpl) {
        case Minus(Val(double v)) -> new Val(-v);
        case Minus(Minus(Expr e)) -> e;
        default -> minusSimpl;
    };
}
// ... Mult und Recip analog
}

```

**Listing 4.6** *Implementierung der Methode simplify für Expr*

Man beachte, dass die Fälle weitgehend den oben beschriebenen Vereinfachungsregeln entsprechen. Pattern Matching ermöglicht somit eine deklarative Programmierung, die sich eng an der Problemstellung anlehnt.

### 4.3 Paare und Tupel

Paare oder Tupel sind grundlegende Datenstrukturen in der funktionalen Programmierung. Wie in [Abschnitt 3.1](#) beschrieben, werden sie beispielsweise für Funktionen mit mehreren Rückgabewerten verwendet. In funktionalen Sprachen sind Paare und Tupel daher als grundlegende Datenstrukturen direkt in die Sprache integriert. Leider wurden diese Datenstrukturen nicht zusammen mit den funktionalen Konzepten in Java eingeführt. Paare und Tupel sind jedoch einfache Datenstrukturen, und in diesem Abschnitt werden Implementierungen dieser Strukturen vorgestellt.

Die generische Record-Klasse `Pair` in [Listing 4.7](#) implementiert eine Struktur mit den beiden Komponenten `fst` und `snd`. Die Typen der Werte werden durch die generischen Typparameter `A` und `B` festgelegt. Zusätzlich wird eine statische Methode `of` zur Erzeugung von `Pair`-Werten bereitgestellt und `toString` überschrieben. Es ist sinnvoll, die Klasse als `Serializable` zu deklarieren.

Wie sich die Klasse `Pair` für Rückgabewerte nutzen lässt, haben wir bereits im Beispiel in [Abschnitt 3.1](#) gesehen.

```
public record Pair<A, B>(A fst, B snd)
implements Serializable {
    public static <A, B> Pair<A, B> of(A fst, B
    snd) {
        return new Pair<>(fst, snd);
    }
    @Override
    public String toString() {
        return "(" + fst + ", " + snd + ")";
    }
}
```

#### **Listing 4.7** Record-Klasse `Pair`

In der gleichen Weise lassen sich Klassen definieren, die mehrere Werte zusammenfassen. Die Record-Klasse `Tuple3` in [Listing 4.8](#) enthält drei Werte. Die Komponenten der Werte werden mit `v1`, `v2` usw. bezeichnet. Ansonsten ist die Klasse analog zur Klasse `Pair`. Nach demselben Muster können die Klassen `Tuple4` und `Tuple5` für vier bzw. fünf Werte definiert werden<sup>2</sup>.

```
public record Tuple3<A, B, C>(A v1, B v2, C v3)
implements Serializable {
```

```

public static <A, B, C> Tuple3<A, B, C> of(A
v1, B v2, C v3) {
    return new Tuple3<>(v1, v2, v3);
}
@Override
public String toString() {
    return "(" + v1 + ", " + v2 + ", " + v3 +
    ")";
}
}

```

**Listing 4.8** Klasse *Tuple3*

## 4.4 Funktionale Listen

In einer rein-funktionalen Programmierung können auch Datenstrukturen wie Listen und Mengen keine Seiteneffekte haben. Bei funktionalen Datenstrukturen werden bei Änderungen immer neue Objekte erzeugt, die diese Änderungen repräsentieren. Bestehende Objekte bleiben unverändert. Funktionale Datenstrukturen werden daher auch persistente Datenstrukturen genannt.

Dass bei jeder Änderung ein neues Objekt erzeugt wird, erscheint zunächst ineffizient. Dieses Vorgehen hat aber auch entscheidende Vorteile: Erstens können die neuen Objekte auf den bestehenden Strukturen aufbauen, da diese ja unverändert erhalten bleiben. Zum anderen können unveränderliche Objekte problemlos weitergegeben und geteilt werden – ein Vorteil, der insbesondere bei nebenläufiger und paralleler Verarbeitung von Bedeutung ist. Zudem hat die Forschung sehr effiziente Verfahren zur Implementierung persistenter Datenstrukturen hervorgebracht [70]. So gibt es bei Scala eine umfangreiche Bibliothek mit persistenten Collections [67] und mit `vavr.io` [101] gibt es eine Portierung dieser Bibliothek in Java. Persistente Collections werden wir im nächsten Abschnitt beschreiben.

Im Folgenden soll eine funktionale Liste in Java implementiert werden. Wie bereits in [Abschnitt 1.1](#) erwähnt, sind funktionale Listen eine grundlegende Datenstruktur der funktionalen Programmierung. Die hier beschriebene Implementierung greift diese Ideen auf, berücksichtigt aber auch die spezifischen Anforderungen der Programmierung in Java. So werden für die Iteration und den Aufbau von Listen imperative Programmstrukturen verwendet,

ohne jedoch die funktionalen Eigenschaften der Liste aufzugeben. In den folgenden Kapiteln wird diese Implementierung immer wieder verwendet bzw. als Grundlage für die Einführung weiterer Konzepte der funktionalen Programmierung genutzt.

Funktionale Listen sind rekursive Strukturen mit zwei Varianten: erstens die leere Liste und zweitens eine Zelle mit einem Wert und einer Referenz auf die Restliste. In [Listing 4.9](#) wird die abstrakte Klasse `FList` eingeführt. Von dieser Klasse gibt es zwei Unterklassen, `Nil` und `Cons`, die als statische innere Klassen implementiert sind. Die Klasse `Nil` stellt die leere Liste dar. Von dieser wird in `FList` ein einzelnes Objekt `NIL` erzeugt, auf das mit der statischen Methode `empty` zugegriffen wird. Die Klasse `Cons` hat ein Feld `head` vom Elementtyp `E`, ein Feld `tail` für die Referenz auf die Restliste und ein Feld `size` für die Anzahl der Elemente.

Die konkrete Methode `add` der Basisklasse `FList` erzeugt ein Objekt vom Typ `Cons` mit dem übergebenen Element als `head` und der aktuellen Liste als `tail`. Damit wird eine neue Liste mit dem neuen Element als erstem Element und dieser Liste als Restliste erzeugt. Die Größe der neuen Liste ergibt sich aus der Größe der erweiterten Liste plus 1.

## Listen in Lisp

Rekursive Listen wurden bereits in der Sprache Lisp [1] als elementare Datenstruktur eingeführt. Tatsächlich leitet sich der Name »Lisp« von »List Processing« ab. Listen dienen in Lisp sowohl zur Darstellung von Daten als auch von Programmen. Sie sind aus Paaren aufgebaut, wobei das erste Element des Paares auf den Wert und das zweite Element auf die Restliste zeigt.

Paare werden nach der sie erzeugenden Funktion `cons` auch `Cons`-Zellen genannt. Die leere Liste ist ein besonderes Element und wird `Nil` genannt. Die Namensgebung in der Implementierung von `FList` folgt daher der Namensgebung in Lisp. Die Funktionen, die auf das erste Element und den Rest der Liste zugreifen, heißen in Lisp `car` und `cdr` (sprich: kudar). Diese Namen stammen aus der ersten Implementierung von Lisp, wirken fremd, sind aber in der Lisp-Welt allgegenwärtig. In der Implementierung von `FList` werden wir aber die sprechenden und außerhalb der Lisp-Welt gebräuchlichen Namen `head` und `tail` verwenden.

```
public abstract class FList<E>... {
```

```

private static FList NIL = new Nil();
public static <E> FList<E> empty() {
    return (FList<E>)NIL;
}
public FList<E> add(E value) {
    return new Cons<E>(value, this);
}
public abstract boolean isEmpty();
public abstract int size();
public abstract E head();
public abstract FList<E> tail();
public abstract boolean contains(Object o);
...
public static final class Nil<E> extends
FList<E> {
    @Override
    public boolean isEmpty() { return true; }
    @Override
    public int size() { return 0; }
    @Override
    public E head() {
        throw new NoSuchElementException("...");
    }
    @Override
    public FList<E> tail() {
        throw new NoSuchElementException("...");
    }
    @Override
    public boolean contains(Object o) { return
false; }
}

```

```

    ...
}
public static final class Cons<E> extends
FList<E> {
    private final E head;
    private final FList<E> tail;
    private final int size;
    private Cons(E head, FList<E> tail) {
        this.head = head;
        this.tail = tail;
        this.size = tail.size() + 1;
    }
    @Override
    public boolean isEmpty() { return false; }
    @Override
    public int size() { return size; }
    @Override
    public E head() { return head; }
    @Override
    public FList<E> tail() { return tail; }
    @Override
    public boolean contains(Object o) {
        return this.head.equals(o) ||
            this.tail.contains(o);
    }
    ...
}
}

```

**Listing 4.9** Funktionale Liste FList

Die Basisklasse `FList` definiert darüber hinaus eine Reihe abstrakter Methoden: Die Methode `isEmpty` prüft, ob die Liste leer ist, `size` liefert die Größe der Liste, `head` das erste Element, `tail` die Restliste und `contains` prüft, ob ein Element enthalten ist. Diese abstrakten Methoden sind in den konkreten Klassen `Nil` und `Cons` entsprechend implementiert. Beispielsweise liefert `empty` in `Nil` immer `true` und in `Cons` immer `false`. Der Wert von `size` ist bei `Nil` gleich 0 und bei `Cons` gleich dem Feld `size`. Die Methoden `head` und `tail` greifen bei der Klasse `Cons` auf die beiden Felder zu, bei der Klasse `Nil` werfen sie eine `NoSuchElementException`. Schließlich liefert `contains` bei `Nil` immer `false`, bei `Cons` wird das Element mit `head` verglichen und bei Ungleichheit erfolgt ein rekursiver Aufruf mit der Restliste.

### Exceptions beim Zugriff auf `head` und `tail`

Da `head` und `tail` bei einem Zugriff Exceptions werfen, sind diese Methoden keine reinen Funktionen. Allerdings orientieren wir uns bei diesen beiden Methoden an der Implementierung in Haskell, bei der Zugriffe auf die leere Liste zu einem Programmabbruch führen.

Rekursive Listen kann man mit rekursiven Methoden verarbeiten. Zum Beispiel könnte man das Drucken der Listenelemente durch das folgende rekursive Programm implementieren:

```
static <E> void print(FList<E> list) {
    if (! list.isEmpty()) {
        System.out.println(list.head().toString());
        print(list.tail());
    }
}
```

Bei dieser rekursiven Methode wird bei einer nicht-leeren Liste das erste Element ausgegeben und mit der Restliste die Methode rekursiv aufgerufen.

Rekursive Methoden sind in Java jedoch weniger effizient als iterative Lösungen (vgl. [Abschnitt 3.1.1](#)). Daher implementieren wir für `FList` das Interface `Iterable` (siehe [Listing 4.10](#)). Das von der Methode `iterator` zurückgegebene `Iterator`-Objekt verwendet eine veränderliche Variable `current`. Diese wird mit der Liste initialisiert. Die Methode `hasNext` liefert `true`, wenn `current` nicht gleich der leeren Liste ist. Die Methode `next`

liefert das Element head der Liste current und schaltet dann current auf die Restliste weiter.

```
public abstract class FList<E> implements
Iterable<E> {
    ...
    public boolean contains(Object o) {
        for (E e : this) {
            if (e.equals(o)) return true;
        }
        return false;
    }
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            FList<E> current = FList.this;
            @Override
            public boolean hasNext() {
                return ! current.isEmpty();
            }
            @Override
            public E next() {
                E next = current.head();
                current = current.tail();
                return next;
            }
        };
    }
}
```

**Listing 4.10** Implementierung von Iterable für FList

Damit können wir Schleifen verwenden und zum Beispiel die obige Methode `print` wie gewohnt mit einer For-Schleife implementieren:

```
static <E> void print(FList<E> list) {
    for (E e : list) {
        System.out.println(e.toString());
    }
}
```

Auch die Methode `contains`, die wir zuvor in der Klasse `Cons` rekursiv gelöst haben, können wir nun einfacher iterativ lösen (siehe Methode `contains` in [Listing 4.10](#)). Man beachte, dass durch diese iterativen Implementierungen die funktionalen Eigenschaften von `FList` selbst nicht verletzt wurden.

Mit einer Methode `of` soll man aus einer Reihe von Elementen ein `FList`-Objekt erzeugen können. Dies ist aber mit einem rekursiven Verfahren schwierig. Wir wollen uns daher auch hier die Vorteile einer imperativen Implementierung zunutze machen, aber wieder ohne die funktionalen Eigenschaften von `FList` selbst zu verletzen. Wir führen dazu eine Klasse `Builder` für den Aufbau von `FList`-Objekten ein. `Builder` ist eine statische innere Klasse von `FList` ([Listing 4.11](#)). Sie verwendet eine veränderliche `LinkedList` und stellt die Methoden `add` und `build` zur Verfügung. Mit `add` werden die Elemente vorne in der `LinkedList` eingefügt. Bei mehreren Aufrufen von `add` steht also das zuletzt eingefügte Element an erster Stelle. Die Methode `build` startet von einer Startliste (`start`), die im einfachsten Fall mit der leeren Liste initialisiert wird, iteriert über die Elemente der `LinkedList` und erstellt mit `add` jeweils eine neue Liste, bei der das Element vorangestellt ist. Sie gibt schließlich die so erstellte Liste zurück.

```
public abstract class FList<E> implements
Iterable<E> {
    ...
    public static <E> FList<E> of(E...elems) {
        Builder<E> b = new Builder<E>();
        for (E e : elems) b.add(e);
        return b.build();
    }
    public static class Builder<E> {
```

```

private final LinkedList<E> list;
private final FList<E> start;
private Builder(FList<E> start) {
    list = new LinkedList<E>();
    this.start = start;
}
private Builder() {
    this(FList.empty());
}
public void add(E e) {
    list.addFirst(e);
}
public FList<E> build() {
    FList<E> fList = start;
    for (E e: list) fList = fList.add(e);
    return fList;
}
}
}
}

```

**Listing 4.11** *Innere Klasse Builder zum Aufbau von FList-Objekten*

Die Methode `of` kann nun einfach mit einem `Builder` implementiert werden (siehe Methode `of` in [Listing 4.11](#)). Wir werden dieses Verfahren zum Aufbau von `FList`-Objekten in den folgenden Kapiteln noch mehrfach einsetzen.

Für `FList` sind zahlreiche weitere Methoden sinnvoll und oft notwendig, etwa `equals` und `hashCode`, `remove`, `concat` und `subList`. Als Beispiel soll in [Listing 4.12](#) die Implementierung von `remove` gezeigt werden, die alle Vorkommen eines Elements löscht.

Die Methode verwendet eine private rekursive Methode `removeRec` mit der Liste und dem zu entfernenden Element als Parameter. Diese Methode arbeitet mit einer Kombination von iterativem Code und rekursiven Aufrufen. In einer Schleife wird so lange über die Liste iteriert, bis das Element gefunden wurde

oder das Ende der Liste erreicht ist. Wurde das Ende erreicht, war das Element nicht enthalten und die Liste muss daher nicht verändert werden. Damit wird die Liste unverändert zurückgegeben. Wird das zu löschende Element gefunden, wird es übersprungen, und die restliche Liste wird mit einem rekursiven Aufruf behandelt. An die vom rekursiven Aufruf zurückgegebene Liste werden nun die zuvor gesicherten Elemente angefügt und daraus die Ergebnisliste erzeugt. Damit wird immer das letzte Stück, in dem das zu löschende Element nicht mehr enthalten ist, wiederverwendet.

```
public abstract class FList<E> implements
Iterable<E> {
    ...
    public FList<E> remove(Object obj) {
        return removeRec(this, obj);
    }
    private FList<E> removeRec(FList<E> fList,
Object obj) {
        LinkedList<E> ll = new LinkedList<E>();
        FList<E> fL = fList;
        while (! fL.isEmpty() && !
fL.head().equals(obj)) {
            ll.addFirst(fL.head());
            fL = fL.tail();
        }
        if (fL.isEmpty()) return fList;
        fL = removeRec(fL.tail(), obj);
        for (E e: ll) fL = fL.add(e);
        return fL;
    }
}
```

**Listing 4.12** Methode remove bei FList

## Beispielanwendung

Das Arbeiten mit der funktionalen Liste `FList` soll nun an einem Beispiel demonstriert und der Unterschied zu einer herkömmlichen Lösung aufgezeigt werden. Wir implementieren das bekannte N-Damen-Problem mit einem rekursiven Backtracking-Verfahren in zwei Varianten, einmal mit veränderlichen Strukturen und einmal mit unserer unveränderlichen Liste `FList`.

Bei der imperativen Lösung wird der Spielzustand durch ein veränderliches zweidimensionales Boole'sches Array repräsentiert. Im Backtracking-Algorithmus wird eine Position gesetzt und das Verfahren rekursiv mit dem geänderten Spielzustand aufgerufen. Ist der Versuch nicht erfolgreich, wird der Zug rückgängig gemacht (Backtracking) und ein neuer Versuch mit der nächsten Spalte gestartet.

```
// imperative Lösung!
boolean solveNQueens(boolean[][] board, int col)
{
    if (col >= board.length) {
        return true;
    } else {
        for (int i = 0; i < board.length; i++) {
            if (isSafe(board, i, col)) {
                board[i][col] = true;
                if (solveNQueens(board, col + 1))
                    return true;
                board[i][col] = false; // Backtracking
            }
        }
        return false;
    }
}
```

In der funktionalen Lösung wird der Spielzustand durch eine funktionale Liste der besetzten Positionen codiert. Zudem wird ein `Optional` als Rückgabewert verwendet. `Optional.empty()` zeigt dabei an, dass keine Lösung gefunden werden konnte. Beim rekursiven Verfahren wird die Liste, die die aktuell besetzten Positionen enthält, um eine nächste mögliche Position erweitert und

damit die Methode rekursiv aufgerufen. Beim Backtracking-Schritt ist kein Zurücknehmen des Zuges notwendig, da die aktuelle Liste nicht verändert wurde.

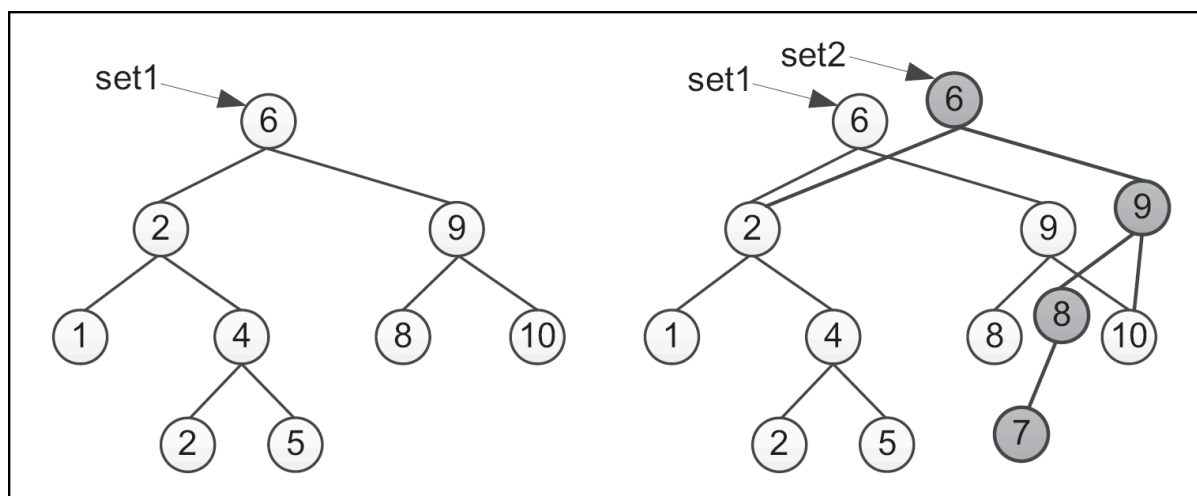
```
static final int N = 8; // Größe des Feldes
Optional<FList<Pos>> solveNQueens(FList<Pos>
board, int col) {
    if (col >= N) {
        return Optional.of(board);
    } else {
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col)) {
                Optional<FList<Pos>> optSol =
                    solveNQueens(board.add(Pos.of(i, col)),
                        col+1);
                if (optSol.isPresent()) return optSol;
            }
        }
        return Optional.empty();
    }
}
```

Der Unterschied zur imperativen Lösung ist also, dass beim Backtracking der Zug nicht zurückgenommen werden muss. In diesem kleinen Beispiel wird der Vorteil der persistenten Datenstruktur noch nicht deutlich. Wenn die Strukturen jedoch komplexer sind und mit vielen und komplexen Änderungen gearbeitet werden muss, kann das Rückgängigmachen von Änderungen schnell schwierig und fehleranfällig werden. Der Vorteil der persistenten Datenstruktur wird noch deutlicher, wenn das Verfahren parallelisiert werden soll. Bei einer veränderlichen Datenstruktur muss diese für parallele Aufrufe kopiert werden. Dies ist bei der funktionalen Lösung nicht notwendig. Man kann einfach die Liste mit den aktuellen Positionen um die nächsten Positionen erweitern und mit diesen jeweils die Methode parallel aufrufen. In [Kapitel 10](#) werden wir uns intensiv mit parallelen Streams beschäftigen und noch genauer auf die Vermeidung von Seiteneffekten eingehen.

## 4.5 Persistente Collections

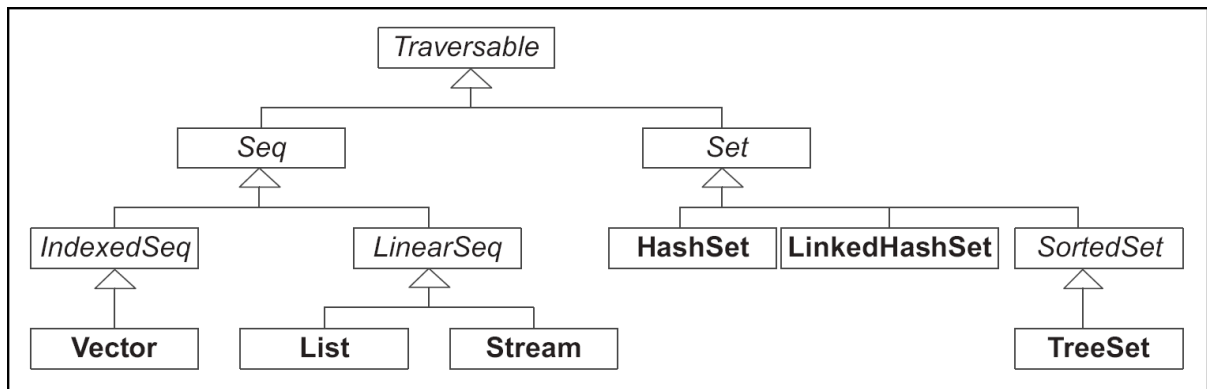
Neben funktionalen Listen können auch Mengen, Maps und andere Collections funktional implementiert werden. Die Arbeit von [70] hat hier wesentliche Vorarbeit geleistet und gezeigt, wie persistente Collections effizient implementiert werden können. Das Prinzip ist das gleiche wie bei den funktionalen Listen: Bestehende Strukturen werden nicht verändert, jede Änderung wird durch eine neue Struktur repräsentiert, aber bestehende Strukturen werden so weit wie möglich wiederverwendet (siehe dazu auch [34]).

Abbildung 4.1 zeigt, wie dies für binäre Suchbäume funktioniert. Angenommen, es existiert eine sortierte Menge `set1`, die, wie im linken Teil von Abbildung 4.1 dargestellt, als binärer Suchbaum gespeichert ist. Nun soll das Element 7 hinzugefügt und damit eine neue Menge `set2` erzeugt werden. Ausgehend von der Wurzel des Baums werden die Knoten kopiert, die sich durch das Einfügen von 7 ändern. Dies ist im rechten Teil von Abbildung 4.1 dargestellt. Wie man sieht, werden die unveränderten Teile wiederverwendet.



**Abb. 4.1** Anfügen eines Elements 7 bei einem unveränderlichen binären Baum

In der Standardbibliothek von Java gibt es nur veränderliche Collections. Die Collection-Bibliothek von Scala [90] bietet leistungsstarke Implementierungen persistenter Collections, und auch in Java gibt es mittlerweile proprietäre Implementierungen. So bietet z. B. die Bibliothek `vavr` [101] eine Portierung der Collections von Scala. Abbildung 4.2 zeigt dazu die Klassenhierarchie. Im Folgenden soll diese Klassenbibliothek verwendet werden, um das Arbeiten mit persistenten Collections anhand eines Beispiels zu demonstrieren.



**Abb. 4.2** Auszug aus der Typhierarchie der funktionalen Collections (abstrakte Typen kursiv, konkrete Collection-Klassen fett)

Zur Demonstration des Arbeitens mit persistenten Collections implementieren wir im Folgenden Algorithmen zur Berechnung von Distanzen und Pfaden zwischen Knoten in einem Graphen. Basis des Verfahrens ist eine Klasse Graph mit Knoten vom generischen Typ N (siehe [Listing 4.13](#)). Die Klasse Graph definiert eine Menge nodes für die Knoten und eine Menge von Pair-Objekten für die Kanten. Die Methode successors liefert für einen Knoten node die Menge der Folgeknoten. Sie verwendet höhere Funktionen. Diese sind hier noch nicht angegeben, wir werden sie erst im nächsten Kapitel beschreiben.

```

public class Graph<N> {
    private final Set<N> nodes;
    private final Set<Pair<N, N>> edges;
    public Graph(Set<N> nodes, Set<Pair<N, N>>
edges) {
        this.nodes = nodes;
        this.edges = edges;
    }
    public Set<N> successors(N node) {
        return ...; // Soll die Menge der Folgeknoten
zu node retournieren
    }
    ...
}
  
```

**Listing 4.13** Klasse für Graphen mit Knoten N

Für Graphen soll nun eine Methode `distances` implementiert werden, die für einen Startknoten `start` die Distanzen zu allen Knoten liefert, wobei die Distanz die Anzahl der Kanten von Start zum Knoten sei. Das Ergebnis ist eine `Map<N, Integer>` mit den Distanzen für jeden Knoten.

Die Methode in [Listing 4.14](#) verwendet eine rekursive Hilfsmethode `distRec`. Diese verwendet die persistenten Collections `Queue` und `HashMap`. In `queue` befinden sich die noch nicht behandelten Knoten, in der `Map result` das aktuelle Resultat mit den Distanzen der Knoten. In jedem Aufruf wird der erste Knoten aus der `Queue` entnommen und behandelt (`node`), die noch nicht behandelten Nachfolger bestimmt (`succs`) und in der `Map` für alle Nachfolger als Distanz die Distanz des aktuellen Knotens plus eins eingetragen. Die Methode wird dann rekursiv aufgerufen, wobei aus der `Queue` der aktuelle Knoten gelöscht wird und die Nachfolger hinten angefügt werden. Das Ergebnis ist berechnet, wenn die `Queue` leer ist.

```
public class Graph<N> {
    ..
    public Map<N, Integer> distances(N start) {
        return distsRec(Queue.of(start),
            HashMap.of(start, 0));
    }
    private Map<N, Integer> distsRec(Queue<N>
        queue, Map<N, Integer> result) {
        if (queue.isEmpty()) return result;
        else {
            N node = queue.head();
            var succs =
                successors(node).removeAll(queue)
                    .removeAll(result.keySet());
            var updtResult = result;
            for (N succ : succs) {
                updtResult = updtResult.put(succ,
                    result.get(node).get() + 1);
            }
        }
    }
}
```

```

        return
        distsRec(queue.tail().appendAll(succs),
        updttdResult);
    }
}
}

```

**Listing 4.14** Methode *distances* für die Berechnung der Distanzen der Knoten zu einem Startknoten

Die Methode `distances` verwendet persistente Collections, aber der Unterschied zu einer Lösung mit variablen Collections ist noch gering. Der wirkliche Vorteil zeigt sich, wenn statt der Distanzen die minimalen Pfade der Knoten zum Startknoten berechnet werden. Die Methode `paths` in [Listing 4.15](#) berechnet die Pfade zum Startknoten in Form von Listen von Knoten. Der einzige Unterschied zur Methode `distances` ist, dass statt einer um eins erhöhten Distanz eine Liste mit einem neuen Knoten in der Map gespeichert wird. Entscheidend ist, dass ein solches Verfahren nur mit persistenten Listen funktioniert. Würde man veränderliche Listen verwenden, so würde nicht nur eine erweiterte Liste für den Nachfolgeknoten erzeugt, sondern auch die Liste des aktuellen Knotens verändert werden. Bei veränderlichen Listen müsste also die Liste des aktuellen Knotens kopiert und dann der neue Knoten eingefügt werden. Ein erheblicher Aufwand, der bei persistenten Listen entfällt.

```

public class Graph<N> {
    ...
    public Map<N, FList<N>> paths(N start) {
        return pathsRec(Queue.of(start),
        HashMap.of(start, FList.of(start)));
    }
    Map<N, FList<N>> pathsRec(Queue<N> queue,
    Map<N, FList<N>>result) {
        if (queue.isEmpty()) return result;
        else {
            N node = queue.head();
            var succs =
            successors(node).removeAll(queue)

```

```

        .removeAll(result.keySet());
var updttdResult = result;
for (N succ : succs) {
    updttdResult =
        updttdResult.put(succ,
            result.get(node).get().add(succ));
}
return
pathsRec(queue.tail().appendAll(succs),
updttdResult);
}
}
}

```

**Listing 4.15** Methode *paths* für die Berechnung der Pfade der Knoten zu einem Startknoten

## 4.6 Zusammenfassung

In diesem Kapitel wurde ein Überblick über die in der funktionalen Programmierung verwendeten Datenstrukturen gegeben. Da es sich bei der funktionalen Programmierung um eine Programmierung ohne Seiteneffekte handelt, sind funktionale Datenstrukturen grundsätzlich unveränderlich.

Das Datentypkonzept klassischer funktionaler Sprachen wie Haskell basiert auf sogenannten algebraischen Datentypen. Algebraische Datentypen bestehen aus Produkten, die Records mit Feldern ähnlich sind, und Varianten von Produkten. In Java können algebraische Datentypen mit Record-Klassen und der Implementierung eines Interface implementiert werden. Die Record-Klassen entsprechen den Produkten, das Interface mit seinen implementierenden Record-Klassen entspricht dem Variantentyp.

Algebraische Datentypen erlauben Pattern Matching. In Java wird Pattern Matching durch den `instanceof`-Operator und durch Switch-Anweisungen und Switch-Ausdrücke unterstützt. Mit Type Patterns können Objekte nach ihrem Typ unterschieden werden. Record Patterns erlauben Muster analog zu Record-Klassen und kommen dem Pattern Matching der funktionalen Sprachen sehr nahe.

Schließlich wurden persistente Collections betrachtet, also Collections, die beim Hinzufügen oder Löschen von Elementen nicht verändert werden. Bei Veränderungen werden immer neue Strukturen erzeugt, dabei jedoch bestehende Strukturen wiederverwendet. Wie wir an Beispielen gezeigt haben, kann ein solches Vorgehen durchaus Vorteile haben. Häufig ist ein schrittweises Entwickeln einer Lösung auf der Grundlage früherer Ergebnisse erforderlich, wobei bereits berechnete Werte weiterhin benötigt werden. Bei veränderlichen Collections müsste man die bestehende Lösungsstruktur kopieren und erst dann die Erweiterung vornehmen. Bei persistenten Strukturen ist dies nicht erforderlich, da bestehende Strukturen unverändert erhalten bleiben.

# Index

## A

accept [54](#)  
aggregate [396](#)  
Aggregatsfunktion [24](#)  
Aggregatsoperation [191](#)  
Algebraischer Datentyp [27](#), [73](#), [360](#)  
Algorithmus [105](#)  
allMatch [206](#)  
allOf [254](#)  
alt [181](#)  
and [163](#)  
andThen [161](#)  
Anweisungsrumpf [49](#)  
Any [338](#)  
anyMatch [206](#)  
anyOf [254](#)  
anyToken [175](#)  
apply [54](#)  
Argument  
    explizit typisiert [49](#)  
    implizit typisiert [49](#)  
assertAll [304](#)  
assertDoesNotThrow [304](#), [305](#)  
Assertions [304](#)  
    assertDoesNotThrow [304](#)  
    assertTimeout [304](#)  
AssertJ [306](#)  
assertThrows [305](#)  
assertTimeout [304](#), [306](#)  
associate [393](#)  
associateBy [394](#)

`associateWith` [394](#)  
Assoziativität [125](#), [151](#), [240](#)  
`async` [438](#)  
Asynchrone Ausführung [247](#)  
Asynchrone Funktionskette [247](#)  
Ausdrucksrumpf [49](#)  
Ausführung  
    asynchrone [247](#)  
    bedingte [116](#)  
    eingebettete [115](#)  
    nicht-deterministische [155](#)  
    parallele [137](#), [232](#), [239](#)  
Ausnahmebehandlung [50](#), [70](#), [154](#), [252](#)  
    bei Lambda-Ausdruck [50](#)  
    funktional [70](#)  
Auswertung  
    nach Bedarf [30](#), [119](#)  
    nicht-strikte [30](#)  
`autoConnect` [275](#)  
`average` [208](#)

## **B**

Backpressure [297](#)  
Backtracking-Verfahren [88](#)  
Bedarfsauswertung [30](#), [119](#), [191](#)  
Bedingte Ausführung [116](#)  
`BooleanSupplier` [55](#)  
    [getAsBoolean](#) [55](#)  
Breitensuche [105](#)  
`buffer` [284](#)  
Builder [228](#), [408](#)  
    hierarchisch [408](#)

## **C**

Callback Hell [262](#)  
Callback-basierte Systeme [262](#)  
Call-by-need [30](#)  
`ChannelFlow` [444](#)

- characteristics [235](#)
- chars [199](#)
- chunked [397](#)
- Clojure [24](#)
- Closure [51](#), [377](#)
- Cold Observable [273](#)
- collect [205](#), [208](#), [209](#), [282](#)
- Collection [100](#)
  - funktionale [89](#)
  - höhere Funktion [100](#)
  - mutable [365](#)
  - persistente [89](#), [100](#)
  - read-only [365](#)
- Collector [209](#)
  - counting [215](#)
  - Downstream [213](#)
  - groupingBy [213](#)
  - joining [212](#)
  - partitioningBy [213](#)
  - toCollection [211](#)
  - toList [211](#)
  - toMap [212](#)
  - toSet [211](#)
- Collectors [211](#)
- Command Pattern [111](#)
- CommonLisp [24](#)
- companion [335](#)
- Companion-Objekt [328](#), [335](#)
- Comparator [164](#)
  - comparing [165](#)
  - comparingInt [165](#)
  - naturalOrder [165](#)
  - nullsFirst [166](#)
  - nullsLast [166](#)
  - reversed [166](#)
  - thenComparing [166](#)
- comparing [165](#)
- comparingInt [165](#)
- Completable [270](#)
- CompletableFuture [251](#)
  - allOf [254](#)

- anyOf [254](#)
- exceptionally [253](#)
- handle [251](#)
- runAfterEitherAsync [254](#)
- supplyAsync [251](#)
- thenAcceptAsync [254](#)
- thenApply [252](#)
- thenCombine [254](#)
- CompletionStage [252](#)
- Composable-Funktion [445](#)
- compose [161](#), [169](#)
- concat [198](#)
- connect [275](#)
- ConnectableObservable [274](#)
  - autoConnect [275](#)
  - connect [275](#)
  - refCount [275](#)
- Cons [82](#)
- Consumer [54](#)
- Controller [108](#)
- Coroutine-Builder [437](#)
- CoroutineScope [438](#)
  - async [438](#)
  - launch [438](#)
  - runBlocking [438](#)
- count [133](#), [207](#)
- counting [215](#)

## D

- Datenklasse [337](#), [357](#)
- Datenstruktur
  - funktionale [357](#)
  - persistente [28](#)
- Daten-Tag [27](#)
- Datentyp
  - algebraischer [27](#), [360](#)
  - geschlossen [73](#)
- Default-Methode [44](#)
- Deklarationsvarianz [347](#)
- Delegate [353](#)

Disposable [267](#)  
distinct [201](#), [391](#)  
distinctUntilChanged [280](#)  
Divide-and-Conquer-Prinzip [232](#)  
doFinally [288](#)  
Domänenspezifische Sprache [415](#)  
    externe [415](#)  
    interne [415](#)  
doOnError [288](#)  
doOnNext [288](#)  
doOnSubscribe [288](#)  
DoubleStream [194](#)  
Downstream Collector [213](#)  
drop [391](#)  
dropWhile [201](#)  
DSL [415](#)

## **E**

eachCount [396](#)  
EBNF-Grammatik [182](#)  
Eingebettete Ausführung [115](#)  
empty [198](#), [265](#)  
Entwurfsmuster [110](#)  
    Kommando [111](#)  
    Strategie [110](#)  
Enumerationsklasse [336](#)  
Ersetzungsprinzip [66](#)  
Erweiterungsfunktion [352](#)  
Erweiterungsproperty [352](#)  
estimateSize [235](#)  
exceptionally [253](#)  
Executable [303](#)  
ExecutorService [248](#)

## **F**

Fauler Iterator [119](#), [122](#)  
Fehlerbehandlung [292](#)  
Fibonacci-Zahl [69](#)

[filter](#) [100](#), [177](#), [260](#), [280](#), [390](#)  
[find](#) [392](#)  
[findAny](#) [206](#)  
[findFirst](#) [206](#)  
[first](#) [392](#)  
[flatMap](#) [147](#), [169](#), [175](#), [200](#), [279](#), [389](#)  
[flatten](#) [147](#)  
[FList](#) [82](#), [363](#)  
    [Cons](#) [82](#)  
    [filter](#) [100](#)  
    [forEach](#) [100](#)  
    [generate](#) [100](#)  
    [head](#) [84](#)  
    [isEmpty](#) [84](#)  
    [Iterable](#) [85](#)  
    [map](#) [99](#)  
    [Nil](#) [82](#)  
    [remove](#) [87](#)  
    [tail](#) [84](#)  
[Flow](#) [441](#)  
[Flowable](#) [297](#)  
[Flüssige Schnittstelle](#) [141](#)  
[Flusskontrolle](#) [295](#)  
[fold](#) [391](#), [396](#)  
[Fold left](#) [126](#)  
[Fold right](#) [126](#)  
[Foldable](#) [130](#)  
[Folding](#) [126](#)  
[Folge, unendliche](#) [121](#)  
[forEach](#) [100](#), [266](#)  
[forEachRemaining](#) [227](#)  
[Fork/Join-Framework](#) [137](#), [237](#)  
[Fork/Join-Thread-Pool](#) [238](#)  
[Freie Variable](#) [27](#), [52](#)  
[fromArray](#) [264](#)  
[fromCallable](#) [264](#)  
[fromFuture](#) [265](#)  
[fromIterable](#) [264](#)  
[Function](#) [53](#), [54](#), [161](#)  
    [accept](#) [54](#)  
    [andThen](#) [161](#)

- apply [54](#)
- Funktion
  - Aggregatsfunktion [24](#)
  - anonyme [376](#)
  - Erweiterungsfunktion [352](#)
  - höherer Ordnung [24](#), [26](#), [98](#)
  - infix [332](#)
  - Kombination [24](#)
  - mit Gedächtnis [67](#)
  - reine [63](#)
- Funktion höherer Ordnung [24](#), [98](#)
- Funktionale Datenstruktur [28](#), [357](#)
- Funktionale Liste [82](#)
  - in Kotlin [363](#)
- Funktionale Verkettung [253](#)
- Funktionales Interface [46](#), [53](#), [54](#)
- Funktionsabschluss [51](#)
- Funktionskette, asynchrone [247](#)
- Funktionskomposition [24](#), [161](#), [253](#)
  - Comparator [164](#)
  - Function [161](#)
- Funktionsparameter [97](#)
- Funktionsreferenz [376](#)
- Funktionsstyp [54](#)
  - echter [54](#)
  - Function0 [372](#)
  - Function1 [372](#)
  - Function2 [372](#)
- Funktor [142](#)
  - Identitätsgesetz [145](#)
  - Kompositionsgesetz [145](#)
  - map [142](#)
  - Optional [144](#)
  - Strukturerhaltung [146](#)
- Future [248](#)

## **G**

- gather [202](#)
- Gatherer [202](#)
  - windowFixed [205](#)

[windowSliding 205](#)  
[Gatherers 204](#)  
[Gebundene Variable 27](#)  
[Gen](#)  
    [compose 169](#)  
    [flatMap 169](#)  
    [map 169](#)  
[generate 100, 198, 265](#)  
[Generics 35, 343](#)  
[Generizität 28, 35, 343](#)  
[Geschachtelte Klasse 336](#)  
[get 54, 71](#)  
[getAsBoolean 55](#)  
[getExactSizeIfKnown 235](#)  
[Getter-Funktion 331](#)  
[groupBy 283, 395](#)  
[GroupedObservable 283](#)  
[groupingBy 395](#)

## H

[handle 251](#)  
[Haskell 25](#)  
[head 84](#)  
[Higher-kinded type parameter 143](#)  
[Höhere Funktion 24, 26, 98](#)  
    [in Kotlin 387](#)  
[Hot Observable 273](#)

## I

[Identitätsgesetz 145, 150](#)  
[ifPresent 71, 118](#)  
[ifPresentOrElse 72](#)  
[ignoreElements 287](#)  
[indexOf 393](#)  
[Infix-Funktion 332](#)  
[Initialisierungsroutine 327](#)  
[Inline-Werteklasse 337](#)  
[Innere Klasse 336](#)

instanceof [75](#)  
Integrator [203](#)  
Interface  
    funktionales [53](#), [54](#)  
interval [265](#)  
ints [199](#)  
IntStream [194](#)  
invoke [372](#)  
isEmpty [84](#)  
isPresent [71](#)  
it [375](#)  
Iterable [85](#)  
Iterable-Erweiterungsfunktion  
    aggregate [396](#)  
    all [393](#)  
    any [393](#)  
    associate [393](#)  
    associateBy [394](#)  
    associateWith [394](#)  
    chunked [397](#)  
    distinct [391](#)  
    drop [391](#)  
    eachCount [396](#)  
    filter [390](#)  
    find [392](#)  
    first [392](#)  
    flatMap [389](#)  
    fold [391](#), [396](#)  
    groupBy [395](#)  
    groupingBy [395](#)  
    indexOf [393](#)  
    joinTo [398](#)  
    joinToString [398](#)  
    last [392](#)  
    map [371](#), [389](#)  
    mapIndexed [389](#)  
    mapTo [389](#)  
    none [393](#)  
    partition [397](#)  
    reduce [391](#)  
    sort [391](#)

- sortDescending [391](#)
- sortedBy [390](#)
- sortedByDescending [390](#)
- sortWith [391](#)
- take [391](#)
- windowed [397](#)
- zip [390](#)
- iterate [198](#)
- Iteration [66](#)
  - externe [190](#)
  - interne [190](#)
- Iterator, fauler [119](#), [122](#)
- IteratorSplitter [228](#)

## **J**

- Java Generics [35](#)
- Jetpack Compose [445](#)
- joinTo [398](#)
- joinToString [398](#)
- JUnit [303](#)
- just [264](#)

## **K**

- Kanonischer Konstruktor [59](#)
- Klassen-Delegate [355](#)
- Kombination von Funktionen [24](#)
- Kombinationsoperator [178](#)
- Kombinator-Parser [173](#), [421](#)
- Kommando-Muster [111](#)
- Kommutatives Monoid [126](#)
- Komponente von Record-Klasse [59](#)
- Kompositionsgesetz [145](#)
- Konstruktor
  - kanonischer [59](#)
  - primärer [327](#)
  - sekundärer [328](#)
- Kontravarianz [37](#), [346](#)
- Koroutine [435](#), [437](#)

Kotlin [31](#), [319](#)  
  Arrays [320](#)  
  Basisdatentypen [320](#)  
  For-Schleife [325](#)  
  If-Ausdruck [324](#)  
  when [325](#)  
  While-Schleife [326](#)  
Kotlin-Schlüsselwort  
  async [438](#)  
  companion [335](#)  
  data [357](#)  
  get [331](#)  
  infix [332](#)  
  it [375](#)  
  launch [438](#)  
  open [328](#)  
  operator [332](#)  
  override [328](#)  
  reified [351](#)  
  sealed [360](#)  
  set [331](#)  
  val [324](#)  
  var [324](#)  
Kotlin-Variable  
  val [324](#)  
  var [324](#)  
Kovarianz [37](#), [346](#)  
Ktor [450](#)  
Ktor Plug-in [450](#)  
Kurzschlussauswertung [206](#)

## L

Lambda-Ausdruck [26](#), [46](#), [97](#), [371](#)  
  Ausnahmebehandlung [50](#)  
  inlined [378](#)  
  mit Empfänger [401](#)  
Lambda-Kalkül [21](#)  
last [392](#)  
Laufzeit [243](#)  
launch [438](#)

Lazy evaluation [30](#), [119](#)  
left [178](#)  
Lifting [134](#), [143](#)  
limit [201](#)  
Lisp [21](#), [24](#), [83](#)  
  Liste [83](#)  
Liste, funktionale [82](#)  
Lock [115](#)  
Logger [117](#)  
LongStream [194](#)

## M

map [99](#), [142](#), [147](#), [149](#), [169](#), [177](#), [260](#), [279](#), [371](#), [389](#)  
mapTo [389](#)  
Marble-Diagramm [279](#)  
max [207](#)  
Maybe [271](#)  
Member-Funktion [328](#)  
Methodenreferenz [56](#)  
  für konkretes Objekt [57](#)  
  Konstruktor [57](#)  
  Objektmethode [57](#)  
  statische Methode [57](#)  
min [207](#)  
Modifier  
  open [328](#)  
  override [328](#)  
Monade [146](#)  
  Assoziativität [151](#)  
  flatMap [147](#)  
  FList [148](#)  
  FSet [155](#)  
  Gen [168](#)  
  Id [154](#)  
  Identitätsgesetz [150](#)  
  IO [159](#)  
  Logged [156](#)  
  map [149](#)  
  MonadPlus [159](#)  
  of [147](#)

Optional [149](#), [154](#)  
Parser [175](#)  
Reader [157](#)  
State [159](#)  
Try [158](#)  
Writer [158](#)  
Monoid [125](#), [127](#)  
  Assoziativität [125](#)  
  kommutatives [126](#)  
  neutrales Element [125](#)  
mult [180](#)

## N

naturalOrder [165](#)  
N-Damen-Problem [88](#)  
Nebenläufigkeit [288](#)  
  strukturierte [440](#)  
negate [163](#)  
Nested class [336](#)  
Neutrales Element [125](#), [240](#)  
never [265](#)  
Nicht-strikte Auswertung [30](#)  
Nil [82](#)  
noneMatch [206](#)  
Non-nullable-Typ [340](#)  
Nothing [339](#)  
Nullable-Typ [340](#)  
nullsLast [166](#)

## O

Objektausdruck [334](#)  
Objektdeklaration [335](#)  
Observable [258](#)  
  buffer [284](#)  
  Cold Observable [273](#)  
  collect [282](#)  
  distinctUntilChanged [280](#)  
  doFinally [288](#)

doOnComplete [288](#)  
doOnError [288](#)  
doOnNext [288](#)  
doOnSubscribe [288](#)  
empty [265](#)  
filter [260](#), [280](#)  
flatMap [279](#)  
forEach [266](#)  
fromArray [264](#)  
fromCallable [264](#)  
fromFuture [265](#)  
fromIterable [264](#)  
generate [265](#)  
groupBy [283](#)  
Hot Observable [273](#)  
ignoreElements [287](#)  
interval [265](#)  
just [264](#)  
Kontrakt [263](#)  
map [260](#), [279](#)  
never [265](#)  
observeOn [291](#)  
onErrorResumeNext [295](#)  
onErrorReturn [294](#)  
publish [274](#)  
range [265](#)  
reduce [281](#)  
retry [295](#)  
sample [296](#)  
scan [281](#)  
serialize [289](#)  
share [275](#)  
single [287](#)  
skip [280](#)  
skipWhile [280](#)  
sorted [260](#)  
subscribe [259](#), [266](#)  
subscribeOn [290](#)  
take [280](#)  
takeWhile [280](#)  
Testen [299](#)

- throttle [296](#)
- throttleFirst [296](#), [297](#)
- throw [265](#)
- timeout [294](#)
- timer [265](#)
- timestamp [285](#)
- toFuture [287](#)
- toMultimap [282](#)
- withLatestFrom [287](#)
- observeOn [291](#)
- Observer [258](#)
  - onComplete [258](#)
  - onError [258](#)
  - onNext [258](#)
  - onSubscribe [258](#)
- of [147](#)
- ofNullable [71](#)
- onComplete [258](#)
- onError [258](#)
- onErrorResumeNext [295](#)
- onErrorReturn [294](#)
- onNext [258](#)
- onSubscribe [258](#)
- open [328](#)
- Operation, assoziative [125](#)
- operator [332](#)
- opt [179](#)
- Optional [70](#), [117](#)
  - Ausnahmebehandlung [70](#)
  - get [71](#)
  - ifPresent [71](#), [118](#)
  - ifPresentOrElse [72](#), [118](#)
  - isPresent [71](#)
  - ofNullable [71](#)
  - orElse [71](#), [117](#)
  - orElseGet [71](#), [118](#)
- or [163](#)
- orElse [71](#), [117](#)
- orElseGet [71](#), [118](#)
- override [328](#)

## P

Paar [81](#), [358](#)

parallel [232](#)

Parallele Ausführung [232](#), [239](#)

Paralleler Stream [231](#)

parallelStream [231](#)

Parametrischer Polymorphismus [27](#)

Parser [173](#), [175](#), [421](#)

alt [181](#)

Alternative [181](#)

anyToken [175](#)

filter [177](#)

flatMap [175](#)

left [178](#)

map [177](#)

mult [180](#)

opt [179](#)

Option [179](#)

right [178](#)

Sequenz [178](#)

then [178](#)

token [177](#)

Wiederholung [180](#)

Parser combinators [173](#)

Parser-DSL [424](#)

Parser-Erweiterungsfunktion

filter [422](#)

flatMap [422](#)

left [422](#)

map [422](#)

mult [422](#)

opt [422](#)

or [422](#)

right [422](#)

then [422](#)

partition [397](#)

Pattern Matching [29](#), [75](#), [361](#)

instanceof [75](#)

mit Smart Casts [361](#)

mit when [361](#)

- mit Zerlegungsdeklaration [362](#)
- Record Pattern [78](#)
- Type Pattern [77](#)
- Pattern-Variable [76](#)
- peek [202](#)
- permits [74](#)
- Persistente Collections [89](#)
- Persistente Datenstruktur [28](#)
- Plug-in, Ktor [450](#)
- Polymorphismus, parametrischer [27](#)
- Predicate [54](#), [163](#)
  - and [163](#)
- Predicate, Funktionskomposition [163](#)
- Primärer Konstruktor [327](#)
- Prioritätssuche [105](#), [109](#)
- Programmierung, rein-funktionale [22](#), [26](#), [63](#)
- Property [328](#), [330](#)
  - Erweiterungsproperty [352](#)
- Property-Delegate [353](#)
- Property-Initialisierer [331](#)
- publish [274](#)

## Q

- QuickCheck [307](#)

## R

- range [198](#), [265](#)
- rangeClosed [198](#)
- Reactive Extension [257](#)
- Reactive Manifesto [257](#)
- ReactiveX [257](#)
- Reaktiver Stream [257](#)
- Record Pattern [78](#), [79](#)
- Record-Klasse [59](#), [74](#)
  - Komponenten [59](#)
  - Pair [81](#)
  - Tuple3 [81](#)
- reduce [129](#), [130](#), [205](#), [281](#), [391](#)

[reduceMap 130](#)  
[ReduceTask 137](#)  
[Reducible 129, 130](#)  
    [count 133](#)  
    [reduce 130](#)  
    [reduceMap 130](#)  
    [sum 133](#)  
    [toSet 133](#)  
[Reduktion 125, 126, 206, 281](#)  
    [Map 135](#)  
    [mit Monoiden 125](#)  
    [Optional 134](#)  
    [parallele 127, 136, 240](#)  
    [von links 126](#)  
    [von rechts 126](#)  
[Reduzierbare Struktur 129](#)  
[refCount 275](#)  
[Referenzielle Transparenz 66](#)  
[Reifizierter Typparameter 350](#)  
[Reine Funktion 63](#)  
[Rein-funktionale Programmierung 22, 26, 63](#)  
[Rekursion 66](#)  
[Rekursive Pattern 79](#)  
[remove 87](#)  
[Result 173](#)  
    [Failure 173](#)  
    [Success 173](#)  
[retry 295](#)  
[reversed 166](#)  
[right 178](#)  
[Rohtyp 43](#)  
[Rückstau 295](#)  
[runAfterEitherAsync 254](#)  
[runBlocking 438](#)  
[RxJava 258](#)

## **S**

[SAM-Interface 379](#)  
[SAM-Konvertierung 380](#)  
[sample 296](#)

[scan](#) [281](#)  
[Scheduler](#) [290](#)  
[Schnittstelle, flüssige](#) [141](#)  
[Schrittfolgensteuerung](#) [429](#)  
[Scope-Funktion](#) [406](#)  
    [also](#) [407](#)  
    [apply](#) [407](#)  
    [let](#) [407](#)  
    [run](#) [407](#)  
    [with](#) [406](#)  
[sealed](#) [74](#)  
[Seiteneffekt](#) [288](#)  
[Sekundärer Konstruktor](#) [328](#)  
[Sequence](#) [398](#)  
[sequential](#) [232](#)  
[Serialisierung](#) [289](#)  
[Setter-Funktion](#) [331](#)  
[share](#) [275](#)  
[Shrinker](#) [312](#)  
[Single](#) [268](#)  
[single](#) [287](#)  
[Single Abstract Method \(SAM\) Typen](#) [46](#)  
[Singleton-Objekt](#) [328](#)  
[skip](#) [201](#), [280](#)  
[skipWhile](#) [280](#)  
[Smart Cast](#) [342](#)  
[sort](#) [391](#)  
[sortDescending](#) [391](#)  
[sorted](#) [201](#), [260](#)  
[sortedBy](#) [390](#)  
[sortedByDescending](#) [390](#)  
[sortedWith](#) [391](#)  
[Spliterator](#) [227](#), [235](#)  
    [characteristics](#) [235](#)  
    [estimateSize](#) [235](#)  
    [forEachRemaining](#) [227](#)  
    [getExactSizeIfKnown](#) [235](#)  
    [tryAdvance](#) [227](#)  
    [trySplit](#) [235](#)  
[Sprache, domänen-spezifische](#) [415](#)  
[Standard ML](#) [25](#)

Star-Projektion [349](#)  
Steuerungssystem [428](#)  
Strategiemuster [110](#)  
Strategy Pattern [110](#)  
Stream [189](#)  
    allMatch [206](#)  
    anyMatch [206](#)  
    average [208](#)  
    chars [199](#)  
    collect [205](#), [208](#), [209](#)  
    concat [198](#)  
    count [207](#)  
    distinct [201](#)  
    dropWhile [201](#)  
    empty [198](#)  
    findAny [206](#)  
    findFirst [206](#)  
    gather [202](#)  
    generate [198](#)  
    iterate [198](#)  
    limit [201](#)  
    max [207](#)  
    min [207](#)  
    noneMatch [206](#)  
    parallel [232](#)  
    paralleler [231](#)  
    parallelStream [231](#)  
    peek [202](#)  
    reaktiver [257](#)  
    reduce [205](#), [206](#)  
    sequential [232](#)  
    skip [201](#)  
    sorted [201](#)  
    stream [197](#)  
    sum [208](#)  
    summaryStatistics [208](#)  
    takeWhile [201](#)  
    toArray [205](#)  
    toList [205](#)  
    unendlich [224](#)  
stream [197](#), [228](#)

- Stream-Operation
  - Bibliotheksfunktion [198](#)
  - Erzeugeroperation [197](#)
  - Generator [198](#)
  - Suche [206](#)
  - Terminaloperation [205](#)
  - zustandsbehaftet [201](#), [224](#)
  - zustandslos [224](#)
  - Zwischenoperation [200](#)
- StreamSupport [228](#)
  - stream [228](#)
- Structured concurrency [440](#)
- Struktur, reduzierbare [129](#)
- Strukturerhaltung [146](#)
- Strukturierte Nebenläufigkeit [440](#)
- Subscribe [259](#), [266](#)
- subscribeOn [290](#)
- Substitutionsprinzip [37](#)
- Suche
  - Breitensuche [105](#)
  - Prioritätssuche [105](#)
  - Tiefensuche [105](#)
  - verallgemeinerte [108](#)
- sum [133](#), [208](#)
- summaryStatistics [208](#)
- Supplier [54](#)
  - get [54](#)
- supplyAsync [251](#)
- Suspend-Funktion [436](#)
- Switch-Ausdruck [60](#)
  - yield [61](#)

## T

- tail [84](#)
- take [280](#), [391](#)
- takeWhile [201](#), [280](#)
- Target typing [49](#)
- test [54](#)
- Testen [299](#), [309](#)
  - QuickCheck [309](#)

- von Observable [299](#)
- TestObserver [300](#)
- TestScheduler [300](#)
- then [178](#)
- thenAcceptAsync [254](#)
- thenApply [252](#)
- thenCombine [254](#)
- thenComparing [166](#)
- Thread-Pool [248](#)
- throttle [296](#)
- throttleFirst [296](#), [297](#)
- throw [265](#)
- Tiefensuche [105](#)
- timeout [294](#)
- timer [265](#)
- timestamp [285](#)
- toArray [205](#)
- toFuture [287](#)
- token [177](#)
- toList [205](#)
- toMultiMap [282](#)
- toSet [133](#)
- Trailing Lambda [373](#)
- Tripel [358](#)
- tryAdvance [227](#)
- trySplit [235](#)
- Tupel [81](#)
- Typ
  - non-nullable [340](#)
  - nullable [340](#)
- Typconstraint [36](#), [344](#)
- Type erasure [43](#)
- Type Pattern [77](#)
- Typinferenz [28](#), [41](#)
- Typparameter [27](#), [35](#), [343](#)
  - höherer Ordnung [143](#)
  - reifizierte [350](#)

## U

- Unendliche Folge [121](#), [171](#)

Union type [27](#)  
Unit [339](#)  
UnsafeVariance [367](#)

## V

Variable  
  freie [27](#), [52](#)  
  gebundene [27](#)  
Variant type [27](#)  
Variantentyp [27](#)  
Verband [338](#)  
Verwendungsvarianz [346](#)

## W

when [79](#), [325](#)  
When-Klausel [79](#)  
windowed [397](#)  
windowFixed [205](#)  
windowSliding [205](#)  
withLatestFrom [287](#)

## Y

yield [61](#)

## Z

Zerlegungsdeklaration [362](#), [376](#)  
zip [390](#)  
Zufallswertegenerator [168](#)  
Zufallszahlengenerator [168](#)