Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 10)

Einführung in Streams in Java

Vergleich von Java Streams mit Haskell

Ein einfaches Beispiel in Haskell

Aufgabe: Berechne die Summe der Quadrate aller geraden Zahlen von 1 bis 10.

```
sum [ x^2 | x <- [1..10], even x ]
```

Erklärung:

- **[1..10]:** Erzeugt eine Liste der Zahlen von 1 bis 10.
- **x <- [1..10]:** Nimmt jedes Element `x` aus der Liste.
- **even x:** Filtert nur die Zahlen, für die `even x` `True` ergibt (also die geraden Zahlen).
- **x^2:** Quadriert jede gefilterte Zahl.
- **sum:** Summiert alle quadrierten Zahlen zu einem Endergebnis.

Äquivalentes Beispiel mit Java Streams

Aufgabe: Berechne die Summe der Quadrate aller geraden Zahlen von 1 bis 10 in Java.

```
int sum = IntStream.rangeClosed(1, 10)
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .sum();
```

Erklärung:

- **IntStream.rangeClosed(1, 10):** Generiert einen Stream von Zahlen von 1 bis 10 (inklusive).
- **filter(x -> x % 2 == 0):** Behält nur die geraden Zahlen im Stream.
- **map(x -> x * x):** Quadriert jede verbleibende Zahl im Stream.
- **sum():** Summiert alle quadrierten Zahlen zu einem Endergebnis.

Vergleich: Haskell vs. Java Streams

Ähnlichkeiten:

- **Datenpipeline:** Beide Ansätze verarbeiten Daten durch eine Sequenz von Operationen.
- **Funktionale Operationen:** Nutzung von Map, Filter und Reduktion (`sum`).
- **Kompakte Syntax:** Beide ermöglichen eine ausdrucksstarke und kurze Darstellung der Berechnung.

Unterschiede:

- **Syntax und Stil:**
  - Haskell nutzt **List Comprehensions**, eine mathematisch inspirierte Notation.
  - Java verwendet die **Stream API** mit **Methodenkette** und **Lambda-Ausdrücken**.
- **Auswertungsstrategie:**
  - Haskell arbeitet standardmäßig mit **Lazy Evaluation**.
  - Java Streams können lazy sein, aber die Auswertung erfolgt meist eager mit dem Aufruf einer Terminal-Operation.

Eager vs. Lazy Evaluation

Verständnis der Auswertungsstrategien in Java und Haskell anhand von Beispielen

Was ist Lazy Evaluation?

Definition:

- **Verzögerte Auswertung** von Ausdrücken bis zu dem Zeitpunkt, an dem deren Werte benötigt werden.
- **Vorteile:**
  - Effiziente Nutzung von Ressourcen.
  - Möglichkeit, mit unendlichen Datenstrukturen zu arbeiten.

Haskell und Lazy Evaluation:

- Haskell verwendet standardmäßig Lazy Evaluation.
- Funktionen erzeugen Werte erst bei Bedarf.

Was ist Eager Evaluation?

Definition:

- **Sofortige und vollständige Auswertung** von Ausdrücken, sobald sie aufgerufen werden.
- **Vorteile:**
  - Einfacheres Debugging.
  - Vorhersehbares Ausführungsverhalten.

Java und Eager Evaluation:

- Java verwendet standardmäßig Eager Evaluation.
- Java Streams kombinieren lazy Zwischenoperationen mit eager Terminal-Operationen.

Haskell-Beispiel: Lazy Evaluation

Aufgabe: Summe der Quadrate aller geraden Zahlen von 1 bis ∞, aber nur die ersten 5 Elemente.

```
sum (take 5 [ x^2 | x <- [1..], even x ])
```

Erklärung:

- **[1..]:** Unendliche Liste von Zahlen ab 1.
- **even x:** Filtert nur gerade Zahlen.
- **x^2:** Quadriert die gefilterten Zahlen.
- **take 5:** Nimmt die ersten 5 Elemente.
- **sum:** Summiert diese Elemente.

Lazy Evaluation ermöglicht die Arbeit mit unendlichen Listen

Java-Beispiel: Eager Evaluation

Aufgabe: Summe der Quadrate aller geraden Zahlen von 1 bis 10.

```
int sum = IntStream.rangeClosed(1, 10)
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .sum();
```

Erklärung:

- **IntStream.rangeClosed(1, 10):** Erzeugt einen Stream von 1 bis 10.
- **filter:** Behält nur gerade Zahlen.
- **map:** Quadriert die Zahlen.
- **sum():** Führt die Auswertung aus (eager).

Die Auswertung erfolgt vollständig bei Aufruf von `sum()`.

Vergleich: Lazy vs. Eager Evaluation

	Haskell	Java Streams
Auswertungsstrategie	Lazy Evaluation	Eager Terminal-Operationen
Datenstrukturen	Können unendlich sein	Meist endlich und festgelegt
Performance	Verzögerte Berechnung spart Ressourcen	Sofortige Berechnung aller Elemente

Demonstration der Lazy Evaluation in Haskell

Unendliche Liste:

```
[1..] -- Liste von 1 bis ∞
```

Verwendung mit Lazy Evaluation:

```
take 5 [ x^2 | x <- [1..], even x ]
-- Nimmt die ersten 5 Quadrate gerader Zahlen
```

Erklärung:

- Die unendliche Liste wird nie komplett erzeugt.
- Nur die benötigten Elemente werden berechnet.

Demonstration der Eager Evaluation in Java

Versuch mit unendlichem Stream:

```
IntStream.iterate(1, n -> n + 1)
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .limit(5)
    .sum();
```

Problem:

- `sum()` ist eine Terminal-Operation und versucht, den gesamten Stream auszuwerten.
- **Lösung:** Nutzung von `limit(5)` um die Anzahl der Elemente zu begrenzen.

Eager Evaluation erzwingt die vollständige Auswertung bis zum Limit.

Einschränkungen von Java Streams

- **Keine echte Lazy Evaluation:** Zwischenoperationen sind lazy, aber Terminal-Operationen sind eager.
- **Arbeit mit unendlichen Streams erfordert Vorsicht:**
  - Verwendung von `limit()`, um die Größe zu begrenzen.
  - Risiko von Endlosschleifen bei falscher Anwendung.

Nicht trivial (am Ende nochmal!)

```java
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Intermediate Operations (lazy)
        Stream<Integer> stream = numbers.stream()
                                        .filter(n -> {
                                            System.out.println("Filter: " + n);
                                            return n % 2 == 0; // Nur gerade Zahlen
                                        })
                                        .map(n -> {
                                            System.out.println("Map: " + n);
                                            return n * 2; // Verdoppeln
                                        });

        // Bis hier wird nichts ausgeführt!
        System.out.println("Pipeline aufgebaut, aber nicht ausgeführt");

        // Terminal Operation (führt die Pipeline aus)
        List<Integer> result = stream.collect(Collectors.toList());
    }
}
```

link

Zusammenfassung der Einführung

- **Haskell:**
  - Nutzt Lazy Evaluation umfassend.
  - Ermöglicht effiziente Arbeit mit großen oder unendlichen Datenstrukturen.
- **Java Streams:**
  - Kombinieren lazy Zwischenoperationen mit eager Terminal-Operationen.
  - Erfordern explizite Begrenzung bei unendlichen Streams.

Verständnis der Auswertungsstrategien ist entscheidend für effizienten Code.

Parallele Verarbeitung

Java Streams:

- **Einfache Parallelisierung:**
  - Durch Ersetzen von `stream()` mit `parallelStream()` oder Aufruf von `parallel()`.
  - Beispiel:

```java
int sum = IntStream.rangeClosed(1, 10)
    .parallel()
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .sum();
```

- **Vorteile:**
  - Nutzung von Mehrkernprozessoren für Performance-Gewinn.

Haskell:

- **Parallelisierung erfordert zusätzliche Bibliotheken** wie `Control.Parallel`.
- **Beispielhafte Nutzung:**
  - Mehr Aufwand in der Implementierung verglichen mit Java Streams.

Fehlerbehandlung und Seiteneffekte

Java Streams:

- **Fehlerbehandlung:**
  - Ausnahmen müssen behandelt oder weitergegeben werden.
  - Checked Exceptions können den Code komplizierter machen.
- **Seiteneffekte:**
  - Sollten vermieden werden, da sie das Verhalten des Streams unvorhersehbar machen können.

Haskell:

- **Rein funktional:**
  - Funktionen sind per Definition frei von Seiteneffekten.
  - Fehlerbehandlung über Monaden wie `Maybe` oder `Either`.

Unterschiede

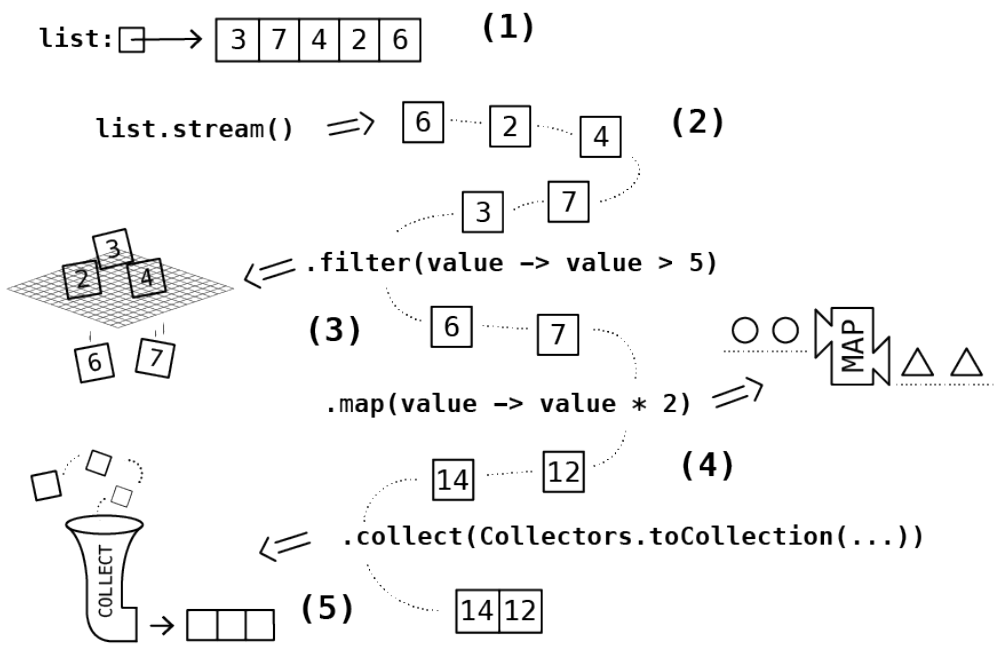
- **Programmierparadigmen:**
  - Haskell ist rein funktional.
  - Java ist objektorientiert mit funktionalen Features.
- **Typisierung:**
  - Haskell nutzt starke, statische Typisierung mit Typinferenz.
  - Java hat auch statische Typisierung, aber mit weniger Typinferenz.
- **Syntax und Lesbarkeit:**
  - Haskell-Code kann für Ungeübte schwer(er) lesbar sein (aber: starke Typinferenz).
  - Java Streams sind (für Java-Entwickler) intuitiver.

Jetzt zu Java "only" : Motivation

- **Java Streams** bringen funktionale Programmierkonzepte in die Java-Welt.
- **Verständnis von Haskell** kann helfen, die funktionalen Aspekte von Streams besser zu verstehen.
- **Bewusstsein für Unterschiede** in Evaluation und Syntax ist wichtig für effiziente und effektive Programmierung.

Streams: Einführung

- **Streams** sind Abstraktionen zur Verarbeitung von Datenflüssen.
- Sie ermöglichen es, Daten **sequenziell** und in **Teilen** zu verarbeiten, anstatt die gesamte Datenmenge auf einmal zu laden.
- Typische Anwendungen umfassen:
  - **Echtzeitdaten** (z.B. Sensoren, Chats)
  - **Dateiverarbeitung**
  - **Netzwerkkommunikation**




```java
List<Integer> list = new ArrayList<>();
list.add(3);
list.add(7);
list.add(4);
list.add(2);
list.add(6);

ArrayList<Integer> values = list.stream()
    .filter(value -> value > 5)
    .map(value -> value * 2)
    .collect(Collectors.toCollection(ArrayLis
```

Eigenschaften von Streams

- **Unidirektional**: Daten fließen nur in eine Richtung (Quelle -> Ziel).
- **Lazy Evaluation**: Operationen auf Streams werden oft erst dann ausgeführt, wenn ein Ergebnis angefordert wird.
- **Sequenziell** oder **Parallel**: Streams können Daten nacheinander oder gleichzeitig verarbeiten.
- **Endliche** oder **unendliche** Datenströme: Manche Streams haben ein Ende, andere laufen unendlich.

Bestandteile eines Streams

1. **Quelle**:
    - Die Datenquelle, von der der Stream liest (z.B. Datei, Netzwerk, Benutzerinput).
2. **Pipeline**:
    - Verarbeitungsschritte, die auf den Datenfluss angewendet werden.
    - Filter, Map, Reduzieren.
3. **Terminal Operation**:
    - Beendet den Stream und liefert ein Ergebnis oder eine Aktion (z.B. Ausgabe, Speicherung).

Funktionsweise von Streams

- **Daten fließen von der Quelle** durch eine Reihe von Verarbeitungsschritten.
- Jeder Schritt kann:
  - **Filtern**: Nur relevante Daten weitergeben.
  - **Transformieren**: Daten umformen (z.B. Zahlen verdoppeln).
  - **Reduzieren**: Mehrere Werte zu einem zusammenfassen (z.B. Summe).

```
[Quelle] --> [Filter] --> [Map] --> [Reduzieren] --> [Ergebnis]
```

Typische Stream-Operationen

1. **Filter**: Entfernt unerwünschte Elemente aus dem Datenfluss.
    - Beispiel: Nur gerade Zahlen passieren.
2. **Map**: Transformiert jedes Element im Stream.
    - Beispiel: Jedes Element wird verdoppelt.
3. **Reduce**: Fasst die Daten zusammen.
    - Beispiel: Summiert alle Elemente.

Vorteile von Streams

- **Effizienter Umgang mit Speicher**: Verarbeitet Daten stückweise, ohne sie komplett zu laden.
- **Lesbarkeit**: Streams ermöglichen eine klare und deklarative Ausdrucksweise.
- **Flexibilität**: Gleiche Methoden für unterschiedliche Datenquellen.
- **Skalierbarkeit**: Datenverarbeitung kann parallelisiert werden.

Herausforderungen bei der Arbeit mit Streams

- **Lazy Evaluation**: Stream-Operationen werden oft erst bei der Terminal-Operation ausgeführt, was Debugging erschwert.
- **Endliche vs. Unendliche Streams**: Nicht alle Streams haben ein klares Ende.
- **Nebenläufigkeit**: Parallelisierbare Streams erfordern sorgfältige Planung und Synchronisation.

Streams in verschiedenen Paradigmen

- **Imperative Programmierung**: Streams bieten eine deklarative Möglichkeit, Daten zu verarbeiten, statt expliziter Schleifen und Zustandsverwaltung.
- **Funktionale Programmierung**: Streams unterstützen typische funktionale Konzepte wie Map, Filter und Reduce.
- **Reaktive Programmierung**: Streams modellieren asynchrone Datenflüsse, ideal für Event-gesteuerte Architekturen.

Streams

- Streams sind ein **leistungsfähiges Konzept** für die Verarbeitung großer oder kontinuierlicher Datenmengen.
- Sie ermöglichen eine **schrittweise, speichereffiziente** Verarbeitung von Daten.
- Streams **abstrahieren** die Datenquelle und bieten eine einheitliche API für verschiedene Arten von Datenströmen.
- Typische Operationen wie **Map, Filter** und **Reduce** machen die Verarbeitung einfach und flexibel.

Sammlungen als Streams verarbeiten

- Sie können Sammlungen (Collections) mit Streams verarbeiten.
- Sie wissen, was ein Lambda-Ausdruck bedeutet.
- Sie kennen die gängigsten Stream-Methoden und können diese in Zwischen- und Terminaloperationen einteilen.

Was ist ein Stream?

- Ein Stream ist eine Möglichkeit, eine Sammlung von Daten zu durchlaufen.
- Index oder aktuelle Variable werden nicht protokolliert.
- Ein Stream verändert nicht die Werte in der ursprünglichen Sammlung, sondern verarbeitet sie.

Beispiel: Verarbeitung von Benutzereingaben

```
Scanner scanner = new Scanner(System.in);
List<String> inputs = new ArrayList<>();
while (true) {
    String row = scanner.nextLine();
    if (row.equals("end")) break;
    inputs.add(row);
}
```

```java
Scanner scanner = new Scanner(System.in);
List<String> inputs = Stream.generate(scanner::nextLine)
    .takeWhile(line -> !line.equals("end"))
    .collect(Collectors.toList());
```

oder auch

```java
Scanner scanner = new Scanner(System.in);
ArrayList<String> inputs = Stream.generate(scanner::nextLine)
    .takeWhile(line -> !line.equals("end"))
    .collect(Collectors.toCollection(ArrayList::new));
```

Erläuterungen

Wir betrachten folgenden Code:

```
List<String> inputs = Stream.generate(scanner::nextLine)
    .takeWhile(line -> !line.equals("end"))
    .collect(Collectors.toList());
```

Überblick

Ziel des Codes:

- **Eingaben einlesen:** Liest Zeilen von der Konsole ein.
- **Beenden bei "end":** Stoppt das Einlesen, wenn der Benutzer "end" eingibt.
- **Sammeln der Eingaben:** Speichert alle eingegebenen Zeilen in einer `List<String>` namens `inputs`.

Stream.generate(scanner::nextLine)

Was passiert hier?

- `Stream.generate(scanner::nextLine)`
  - **Erzeugt einen unendlichen Stream** von Strings.
  - **Supplier:** `scanner::nextLine` ist eine Methodenreferenz, die `scanner.nextLine()` aufruft.
  - **Funktion:** Bei jedem Abruf eines Elements aus dem Stream wird `scanner.nextLine()` aufgerufen, um die nächste Zeile von der Konsole zu lesen.

Funktionsweise von Stream.generate

- **Stream-Erzeugung:**
  - `Stream.generate()` nimmt einen `Supplier<T>` entgegen und erzeugt einen Stream von `T`.
- **Unendlicher Stream:**
  - Da der `Supplier` immer wieder Werte liefert und keine interne Abbruchbedingung hat, ist der Stream potenziell unendlich.
- **Anwendung im Code:**
  - Jedes Mal, wenn der Stream ein Element benötigt, wird `scanner.nextLine()` aufgerufen, um die nächste Benutzereingabe zu erhalten.

takeWhile(line -> !line.equals("end"))

Was passiert hier?

- `takeWhile(line -> !line.equals("end"))`
  - **Zwischenoperation:** Filtert Elemente basierend auf einer Bedingung.
  - **Bedingung:** Nimmt Elemente, solange `line` **nicht** "end" ist.
  - **Kurzschlussverhalten:** Sobald eine Zeile "end" ist, wird die Verarbeitung des Streams beendet.

Funktionsweise von takeWhile

- **Lazy Evaluation:**
  - Die Bedingung wird erst geprüft, wenn ein Element aus dem Stream abgefragt wird.
- **Verarbeitung:**
  - Solange die Bedingung `!line.equals("end")` wahr ist, werden Elemente weitergeleitet.
  - Bei der ersten Zeile, die "end" entspricht, wird der Stream beendet.

collect(Collectors.toList())

Was passiert hier?

- `collect(Collectors.toList())`
  - **Terminaloperation:** Löst die Auswertung des Streams aus.
  - **Sammelt die Elemente:** Alle bis dahin gesammelten Zeilen werden in eine `List<String>` gesammelt.
- **Ergebnis:**
  - Die Variable `inputs` enthält eine Liste aller eingegebenen Zeilen außer "end".

Gesamtzusammenhang des Codes

1. **Stream-Erzeugung:**
    - Startet einen unendlichen Stream, der Zeilen von der Konsole liest.
2. **Bedingte Verarbeitung:**
    - Nimmt Zeilen, solange sie nicht "end" sind.
3. **Sammlung der Ergebnisse:**
    - Sammeln der gefilterten Zeilen in einer Liste.

Schritt-für-Schritt-Ablauf

1. **Benutzereingabe:**
   - Das Programm wartet auf die Eingabe des Benutzers.
2. **Zeile lesen:**
   - `scanner.nextLine()` liest die nächste Zeile.
3. **Bedingung prüfen:**
   - `!line.equals("end")`
     - **Wenn wahr:** Zeile wird weiterverarbeitet.
     - **Wenn falsch:** Stream wird beendet.
4. **Zeile sammeln:**
   - Die Zeile wird in die Liste `inputs` eingefügt.
5. **Wiederholung:**
   - Schritte 1–4 werden wiederholt, bis "end" eingegeben wird.

Beispielhafter Ablauf

Eingaben des Benutzers:

```
Hallo
Wie geht's?
Lernen mit Streams
end
```

Inhalt der Liste `inputs`:

```
["Hallo", "Wie geht's?", "Lernen mit Streams"]
```

Einfluss von Lazy und Eager Evaluation

- **Lazy Evaluation:**
  - `Stream.generate` und `takeWhile` sind lazy.
  - Sie definieren die Pipeline, ohne sofort ausgeführt zu werden.
- **Eager Evaluation:**
  - Die Terminaloperation `collect` ist eager.
  - Erst beim Aufruf von `collect` wird die gesamte Verarbeitung ausgeführt.

Wichtige Aspekte

- **Unendlicher Stream begrenzen:**
  - Ohne `takeWhile` würde `Stream.generate` unendlich laufen.
- **Benutzersteuerung:**
  - Der Benutzer entscheidet durch Eingabe von "end", wann die Eingabe beendet wird.
- **Speichermanagement:**
  - Da die Eingaben in einer Liste gesammelt werden, sollte bei sehr vielen Eingaben der Speicherverbrauch beachtet werden.

Beispiel mit zusätzlicher Verarbeitung

```
List<String> inputs = Stream.generate(scanner::nextLine)
    .takeWhile(line -> !line.equals("end"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

- **Zusätzliche Operation:**
  - `map(String::toUpperCase)` konvertiert jede Zeile in Großbuchstaben.

Beispiel: Statistiken mit Streams

```
long numbersDivisibleByThree = inputs.stream()
    .mapToInt(s -> Integer.valueOf(s))
    .filter(number -> number % 3 == 0)
    .count();
double average = inputs.stream()
    .mapToInt(s -> Integer.valueOf(s))
    .average()
    .getAsDouble();
```

- Anzahl durch 3 teilbarer Zahlen
- Durchschnitt aller Werte

Beispiel IntelliJ `AverageCalculator`

Datenfluss

```
[ "1", "2", "5" ]           // Originale String-Liste
          ⇓
inputs.stream()             // Stream von Strings
          ⇓
.mapToInt(s -> Integer.valueOf(s))
          ⇓
[ 1, 2, 5 ]                 // IntStream von ints
          ⇓
.average()                  // Berechnet Durchschnitt
          ⇓
OptionalDouble[2.6667]      // Ergebnis der Durchschnittsberechnung
          ⇓
.getAsDouble()              // Extrahiert den double-Wert
```

Statefulness von Streams in Java

Was bedeutet "Stateful"?

- Ein **Stream** kann **nur einmal konsumiert** werden.
- Nach einer **Terminal-Operation** wie `sum()`, `collect()` oder `forEach()` ist der Stream **geschlossen**.
- Ein erneuter Zugriff führt zu einer **IllegalStateException**.

Beispiel: Fehler durch mehrfaches Konsumieren

```java
IntStream stream = IntStream.of(1, 2, 3, 4, 5);

// Erste Terminal-Operation (ok)
int sum = stream.sum();

// Zweite Terminal-Operation (führt zu Fehler)
double average = stream.average().getAsDouble(); // IllegalStateException
```

Stream-Methoden (Auswahl)

Methode	Zweck
`stream()`	Erstellt einen Stream von der Sammlung
`mapToInt()`	Konvertiert Werte zu Ganzzahlen
`filter()`	Filtert Werte basierend auf einer Bedingung
`average()`	Berechnet den Durchschnitt
`count()`	Zählt die Anzahl der Werte im Stream

Lambda-Ausdrücke

- Kurzschreibweise für Funktionen
- Beispiel:

```
.stream().filter(value -> value > 5)
```

entspricht:

```
.stream().filter((Integer value) -> {
    return value > 5;
})
```

Quiz

link

Terminaloperationen

- Terminaloperationen beenden einen Stream.
- Beispiele:
  - `forEach` : führt eine Aktion für jedes Element aus.
  - `collect` : sammelt Stream-Elemente in einer Sammlung.
  - `reduce` : kombiniert Elemente.

```
// Beispiel: Sammlung
ArrayList<Integer> positives = list.stream()
    .filter(value -> value > 0)
    .collect(Collectors.toCollection(ArrayList::new));
```

Terminaloperation: forEach

- Die Methode `forEach` führt eine Aktion für jedes Element im Stream aus.
- Sie ist nützlich, um alle Elemente zu durchlaufen und eine Aktion auszuführen, z.B. das Drucken.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.stream()
    .forEach(number -> System.out.println(number));
```

Ausgabe:

```
1
2
3
4
5
```

Terminaloperation: reduce

- Die Methode `reduce` kombiniert die Elemente eines Streams zu einem einzelnen Ergebnis.
- Sie benötigt eine Funktion, die zwei Werte zu einem zusammenführt.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()
    .reduce(0, (subtotal, number) -> subtotal + number);

System.out.println("Summe: " + sum);
```

Ausgabe:

```
Summe: 15
```

Übung: Zahlen, die durch 2, 3 oder 5 teilbar sind

- Implementieren Sie die Methode `public static ArrayList<Integer> divisible(ArrayList<Integer> numbers)`.
- Die Methode gibt eine Liste mit Zahlen zurück, die durch 2, 3 oder 5 teilbar sind.
- Die als Parameter empfangene Liste darf nicht verändert werden.

```
public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(3);
    numbers.add(2);
    numbers.add(-17);
    numbers.add(-5);
    numbers.add(7);

    ArrayList<Integer> divisible = divisible(numbers);
    divisible.stream()
        .forEach(num -> System.out.println(num));
}
```

Ausgabe:

```
3
2
-5
```

Terminaloperation: reduce

- Die `reduce`-Methode kombiniert Stream-Elemente zu einem einzigen Ergebnis.
- Format: `reduce(*initialState*, (*previous*, *object*) -> *action*)`.

```
ArrayList<Integer> values = new ArrayList<>();
values.add(7);
values.add(3);
values.add(2);
values.add(1);

int sum = values.stream()
    .reduce(0, (previousSum, value) -> previousSum + value);
System.out.println(sum);
```

Ausgabe:

```
13
```

Die `reduce`-Methode

- Die Methode `reduce` kombiniert die Elemente eines Streams zu einem einzigen Ergebnis.
- Es gibt verschiedene Signaturen der Methode:

```
Optional<T> reduce(BinaryOperator<T> accumulator);
T reduce(T identity, BinaryOperator<T> accumulator);
U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner);
```

- Am häufigsten genutzt:

  ```
  T reduce(T identity, BinaryOperator<T> accumulator);
  ```

  Diese Methode nimmt zwei Parameter:

  - `identity`: Der initiale Wert für die Berechnung.
  - `accumulator`: Eine Funktion, die zwei Werte zu einem kombiniert.

Alternative: Ohne `identity`

- Es gibt auch die Möglichkeit, die `reduce`-Methode ohne `identity` zu verwenden:

```
Optional<Integer> sum = values.stream()
    .reduce((previous, value) -> previous + value);
```

- Da kein Initialwert angegeben wird, gibt diese Variante ein `Optional` zurück, um mit einem leeren Stream umgehen zu können.
- Beispiel:

```
Optional<Integer> result = List.of(5, 10, 15).stream().reduce((a, b) -> a + b);
System.out.println(result.orElse(0)); // Ausgabe: 30
```

```
Optional<Integer> result = List.of().stream().reduce((a, b) -> a + b);
System.out.println(result.orElse(0)); // Ausgabe: 0
```

```
Optional<Integer> result = List.of().stream().reduce((a, b) -> a + b);
System.out.println(result); // Ausgabe Optional.empty
```

Beispiel: Summe berechnen mit `reduce`

- Beispiel: Berechnung der Summe aller Werte in einer Liste mit `reduce`.

```
ArrayList<Integer> values = new ArrayList<>();
values.add(7);
values.add(3);
values.add(2);
values.add(1);

int sum = values.stream()
    .reduce(0, (previousSum, value) -> previousSum + value);
System.out.println(sum);
```

- Die `reduce`-Methode:
  - Startet mit dem `identity`-Wert `0`.
  - Addiert nacheinander die Werte aus der Liste.

Ergebnis: `13`

Schritt-für-Schritt: Summe berechnen

- Ablauf der `reduce` -Operation:

1. **Startwert:** `previousSum = 0`
2. **Erster Schritt:** `0 + 7 = 7`
3. **Zweiter Schritt:** `7 + 3 = 10`
4. **Dritter Schritt:** `10 + 2 = 12`
5. **Letzter Schritt:** `12 + 1 = 13`

- Am Ende gibt `reduce` den akkumulierten Wert zurück: `13` .

```
System.out.println(sum); // Ausgabe: 13
```

Fazit zu `reduce`

- `reduce` ist eine mächtige Methode zur Akkumulation von Werten in Streams.
- Kann verwendet werden, um Summen, Produkte oder andere Aggregationen zu berechnen.
- Zwei wichtige Varianten:
    i. Mit einem Startwert (`identity`).
    ii. Ohne Startwert, was ein `Optional` zurückgibt.
- Hilfreich für zahlreiche Anwendungsfälle in der funktionalen Programmierung.

reduce: Strings kombinieren

- Die `reduce`-Methode kann auch verwendet werden, um Strings zu kombinieren.

```
ArrayList<String> words = new ArrayList<>();
words.add("First");
words.add("Second");
words.add("Third");
words.add("Fourth");

String combined = words.stream()
    .reduce("", (prevString, word) -> prevString + word + "\n");
System.out.println(combined);
```

Ausgabe:

```
First
Second
Third
Fourth
```

Zwischenoperationen: Einführung

- Zwischenoperationen geben einen Stream zurück und können verkettet werden.
- Beispiele:
  - `map` : Umwandeln von Werten.
  - `filter` : Filtern von Werten.
  - `distinct` : Eindeutige (unique) Werte.
  - `sorted` : Sortieren von Werten.

Beispiel: Filter nach Geburtsjahr

- Geben Sie die Anzahl der Personen aus, die vor 1970 geboren wurden.

```
long count = persons.stream()
    .filter(person -> person.getBirthYear() < 1970)
    .count();
System.out.println("Count: " + count);
```

Beispiel: Vornamen, die mit "A" beginnen

- Geben Sie die Anzahl der Personen aus, deren Vorname mit "A" beginnt.

```
long count = persons.stream()
    .filter(person -> person.getFirstName().startsWith("A"))
    .count();
System.out.println("Count: " + count);
```

Eindeutige (unique) Vornamen in alphabetischer Reihenfolge

- Geben Sie die eindeutigen Vornamen in alphabetischer Reihenfolge aus.

```
persons.stream()
    .map(person -> person.getFirstName())
    .distinct()
    .sorted()
    .forEach(name -> System.out.println(name));
```

Objekte und Streams: Beispiel mit Büchern

- Streams sind auch nützlich bei der Verarbeitung von Objekten, z.B. Bücher und ihre Autoren.
- Beispiel: Durchschnittliches Geburtsjahr der Autoren berechnen.

```
double average = books.stream()
    .map(book -> book.getAuthor())
    .mapToInt(author -> author.getBirthYear())
    .average()
    .getAsDouble();

System.out.println("Durchschnitt: " + average);
```

Warum `getAsDouble`?

Um Folgendes zu umgehen (`average` auf 0 Elementen?):

```
OptionalDouble averageOptional = books.stream()
    .map(book -> book.getAuthor())
    .mapToInt(author -> author.getBirthYear())
    .average();

if (averageOptional.isPresent()) {
    double average = averageOptional.getAsDouble();
    System.out.println("Durchschnittliches Geburtsjahr: " + average);
} else {
    System.out.println("Keine Autoren vorhanden, Durchschnitt nicht berechenbar.
}
```

Dateien und Streams

- Java bietet Streams zum einfachen Verarbeiten von Dateien.
- Die `Files`-Klasse bietet die Methode `lines()`, die eine Datei in einen Stream von Zeilen umwandelt.
- Beispiel:

```
List<String> rows = new ArrayList<>();
try {
    Files.lines(Paths.get("file.txt"))
         .forEach(row -> rows.add(row));
} catch (Exception e) {
    System.out.println("Fehler: " + e.getMessage());
}
```

- Wenn die Datei erfolgreich gelesen wird, enthält die Liste `rows` alle Zeilen der Datei.
- Andernfalls wird eine Fehlermeldung ausgegeben.

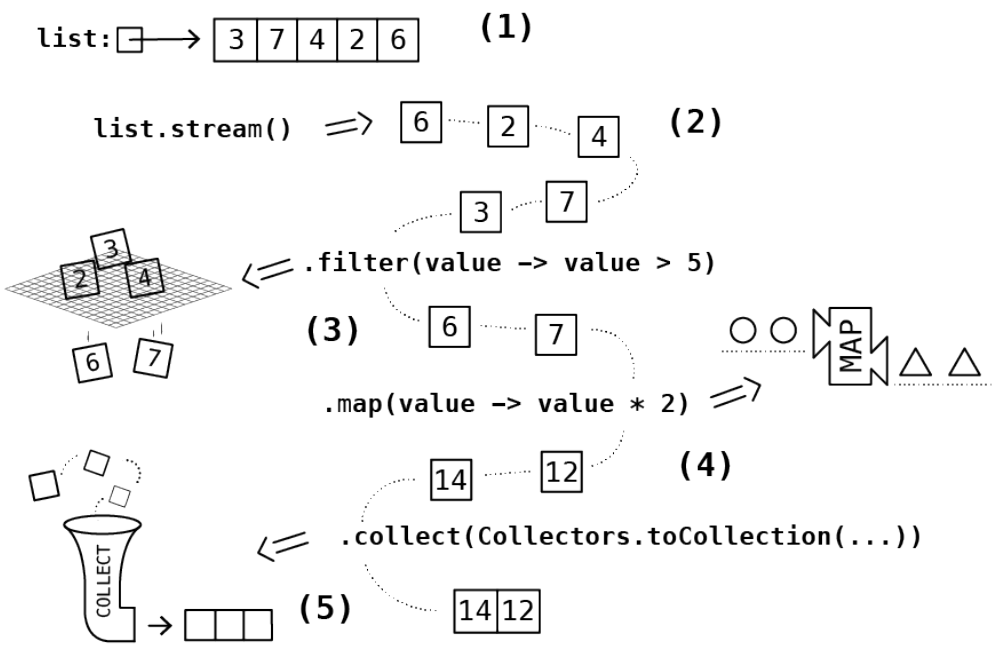
Zusammenfassung

- Streams bieten eine leistungsstarke Möglichkeit, Daten zu verarbeiten.
- Sie sind flexibel und können kombiniert werden.
- Lambda-Ausdrücke machen die Arbeit mit Streams eleganter und effizienter.

Übung: Positive Zahlen

- Implementieren Sie eine Methode `positive(List<Integer> numbers)`, die die positiven Zahlen einer Liste zurückgibt.

```
public static List<Integer> positive(List<Integer> numbers) {
    return numbers.stream()
        .filter(n -> n > 0)
        .collect(Collectors.toList());
}
```




```java
List<Integer> list = new ArrayList<>();
list.add(3);
list.add(7);
list.add(4);
list.add(2);
list.add(6);

ArrayList<Integer> values = list.stream()
    .filter(value -> value > 5)
    .map(value -> value * 2)
    .collect(Collectors.toCollection(ArrayLis
```

Verwendung von `map`, `reduce`, `filter` in Kombination

In diesem Beispiel haben wir eine Liste von Personen, und wir wollen:

1. Personen unter 18 herausfiltern.
2. Alle Hobbys der gefilterten Erwachsenen extrahieren und die Anzahl der **einzigartigen Hobbys** zählen.
3. Eine Zusammenfassung anzeigen, wie viele Hobbys jede Person hat und wie viele Hobbys insgesamt vorhanden sind.

Klassenstruktur

```
class Person {
    String name;
    int age;
    List<String> hobbies;

    // Konstruktoren und Getter ausgelassen
}
```

Beispiel-Liste von Personen

```
List<Person> people = List.of(
    new Person("Alice", 25, List.of("lesen", "wandern", "schwimmen")),
    new Person("Bob", 17, List.of("Videospiele", "wandern")),
    new Person("Charlie", 30, List.of("schwimmen", "kochen")),
    new Person("Diana", 22, List.of("wandern", "fotografie")),
    new Person("Eve", 15, List.of("lesen", "schach"))
);
```

Schritt 1: Personen unter 18 herausfiltern

```
List<Person> adults = people.stream()
    .filter(person -> person.getAge() >= 18)  // Nur Erwachsene
    .collect(Collectors.toList());
```

ohne Streams

```
List<Person> adults = new ArrayList<>();

for (Person person : people) {
    if (person.getAge() >= 18) {  // Nur Erwachsene hinzufügen
        adults.add(person);
    }
}
```

Schritt 2: Einzigartige Hobbys der Erwachsenen sammeln

```
Set<String> uniqueHobbies = adults.stream()
    .flatMap(person -> person.getHobbies().stream())  // Hobbys extrahieren
    .collect(Collectors.toSet());  // Hobbys als Menge
```

ohne Streams

```
Set<String> uniqueHobbies = new HashSet<>();

for (Person person : adults) {
    for (String hobby : person.getHobbies()) {  // Hobbys der Person durchlaufen
        uniqueHobbies.add(hobby);  // Jedes Hobby zur Menge hinzufügen
    }
}
```

Schritt 3: Anzahl der Hobbys pro Person

```
Map<String, Long> hobbiesPerPerson = adults.stream()
    .collect(Collectors.toMap(
        Person::getName,
        person -> person.getHobbies().stream().distinct().count()  // Anzahl der Hobbys pro Person
    ));
```

ohne Streams

```
Map<String, Long> hobbiesPerPerson = new HashMap<>();

for (Person person : adults) {
    // Erstelle eine Menge für eindeutige Hobbys der Person
    Set<String> uniqueHobbies = new HashSet<>(person.getHobbies());

    // Speichere den Namen der Person und die Anzahl der eindeutigen Hobbys
    hobbiesPerPerson.put(person.getName(), (long) uniqueHobbies.size());
}
```

Ausgabe

```
System.out.println("Hobbys pro Person: " + hobbiesPerPerson);
System.out.println("Gesamtanzahl einzigartiger Hobbys: " + uniqueHobbies.size())
```

Beispiel-Ausgabe

```
Hobbys pro Person: {Alice=3, Charlie=2, Diana=2}
Gesamtanzahl einzigartiger Hobbys: 5
```

Alles zusammen

```
Map<String, Long> hobbiesPerPerson = people.stream()
    .filter(person -> person.getAge() >= 18)  // Nur Erwachsene
    .collect(Collectors.toMap(
        Person::getName,  // Name der Person als Schlüssel
        person -> person.getHobbies().stream().distinct().count()  // Anzahl eindeutiger Hobbys als Wert
    ));
```

Aspekt	Stream	Schleife
Code-Kürze	Kürzer und deklarativ: alles in einer Kette	Länger, aber klar strukturiert
Lesbarkeit	Für erfahrene Entwickler eleganter	Für Einsteiger oft verständlicher
Flexibilität	Kombination mit weiteren Operationen einfach	Zusätzliche Arbeit für komplexere Logik
Performance	Ähnlich (Streams können leicht optimiert sein)	Direkte Kontrolle über Iteration, keine zusätzlichen Kosten
Parallelisierung	Einfach mit `.parallelStream()`	Muss manuell implementiert werden

Wichtige Konzepte

- **Filter**: Entfernt alle Personen unter 18 Jahren.
- **Map/flatMap**: Extrahiert und "flacht" die Liste der Hobbys zu einem Strom ab.
- **Reduce**: Summiert die Anzahl der Hobbys pro Person und zählt die **einzigartigen** Hobbys.

`Collectors.toMap` und interne Reduktion

`Collectors.toMap` führt intern eine **Reduktion** durch, um den Stream in eine Map zu transformieren.

So funktioniert es:

Der folgende Code zeigt, wie man `toMap` mit `reduce` explizit nachbilden kann:

```
Map<String, Long> hobbiesPerPerson = people.stream()
    .filter(person -> person.getAge() >= 18)  // Nur Erwachsene
    .reduce(
        new HashMap<>(),  // Startwert: leere Map
        (map, person) -> {
            map.put(
                person.getName(),
                person.getHobbies().stream().distinct().count()
            );
            return map;
        }
    );
```

Hintergrund

(Lambda-Kalkül und Lambda-Ausdrücke)

Lernziele: Lambda-Kalkül und Java

- Sie verstehen den historischen Hintergrund und den Zweck des **Lambda-Kalküls**.
- Sie verstehen das Konzept der **Berechnungsuniversaliät** im Lambda-Kalkül.
- Sie erkennen die Verbindung zwischen dem **Lambda-Kalkül** und **Lambda-Ausdrücken** in Java.

Historischer Hintergrund des Lambda-Kalküls

- Das **Lambda-Kalkül** wurde in den 1930er Jahren von **Alonzo Church** entwickelt.
- Ziel: Ein formales System zur Definition und Manipulation von **Funktionen**.
- **Hintergrund**: Zeitgleich mit der Entwicklung der Turing-Maschine von Alan Turing. Beide Modelle beschäftigen sich mit der Berechenbarkeit.
- Das Lambda-Kalkül wurde als ein universelles und minimalistisches **mathematisches Modell der Berechnung** entwickelt, um zu zeigen, dass Funktionen ohne Maschinen oder Hardware beschrieben werden können.
- Wichtige Entdeckung: **Church-Turing-These**. Sie besagt, dass das Lambda-Kalkül und die Turing-Maschine dieselbe Berechnungsuniversaliät besitzen – beide Modelle sind Turing-vollständig.

Zweck des Lambda-Kalküls

- Der **Hauptzweck** des Lambda-Kalküls ist es, die Berechnung von Funktionen in ihrer **reinsten Form** zu modellieren.
- Es bietet einen formalen Rahmen für die **Definition**, **Applikation** und **Abstraktion** von Funktionen.
- **Lambda-Ausdrücke** beschreiben Funktionen ohne Benennung – sie sind anonym.
    - Beispiel: Die Identitätsfunktion:

      `λx. x`

    - Der Parameter `x` wird einfach zurückgegeben.
- Das Lambda-Kalkül ist eine Grundlage für die **funktionale Programmierung** und wird als **Werkzeug** verwendet, um zu zeigen, wie komplexe Berechnungen nur mit Funktionen und deren Anwendung durchgeführt werden können.

Berechnungsuniversaliät im Lambda-Kalkül

- **Berechnungsuniversaliät**: Das Lambda-Kalkül ist ein Modell, das zeigt, dass jede berechenbare Funktion durch Funktionen und deren Anwendung dargestellt werden kann.
- Dies bedeutet, dass das Lambda-Kalkül **Turing-vollständig** ist – es kann jedes Problem lösen, das auch eine Turing-Maschine lösen kann.
    - Beispiel einer arithmetischen Funktion im Lambda-Kalkül:

      ```
      λx.λy. x + y
      ```

      Dies beschreibt die Addition zweier Zahlen.
- **Abstraktion** und **Applikation** sind die zwei grundlegenden Operationen des Lambda-Kalküls:
    - **Abstraktion**: Erzeugt eine Funktion.
    - **Applikation**: Wendet eine Funktion auf ein Argument an.

Beispiel: Komplexe Lambda-Funktion

- Betrachten wir eine Lambda-Funktion, die das **Maximum** von zwei Werten berechnet:

```
λx.λy. if x > y then x else y
```

- Dieser Ausdruck führt einen Vergleich durch und gibt den größeren Wert zurück.
- **Applikation** eines solchen Ausdrucks:

```
(λx.λy. if x > y then x else y) 5 3  // Ergebnis: 5
```

- Die Idee der anonymen Funktion und ihrer Anwendung auf Argumente zeigt die **Modularität** des Lambda-Kalküls.

Lambda-Ausdrücke: Ihre Verbindung zum Lambda-Kalkül

- **Lambda-Ausdrücke** in modernen Programmiersprachen, einschließlich Java, basieren direkt auf den Prinzipien des **Lambda-Kalküls**.
- Sie ermöglichen es, anonyme Funktionen (ohne Namen) zu definieren und diese direkt zu verwenden.
    - **Syntax eines Lambda-Ausdrucks** in Java:

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

- **Ähnlichkeit** zum Lambda-Kalkül:
    - `λx. x + 1` im Lambda-Kalkül entspricht `(x) -> x + 1` in Java.
- Die Idee der **Berechnungsuniversaliät** bleibt erhalten, auch wenn in Java zusätzliche Konzepte wie **Typen** und **Klassen** existieren.

Beispiel: Lambda-Ausdrücke in Java

- Ein praktisches Beispiel in Java, das zeigt, wie Lambda-Ausdrücke zur **Sortierung** verwendet werden können:

```
List<String> names = Arrays.asList("Anna", "Bob", "Charlie");
names.sort((a, b) -> a.compareTo(b));
names.forEach(name -> System.out.println(name));
```

- Der Lambda-Ausdruck `(a, b) -> a.compareTo(b)` ist ein Ausdruck, der zwei Argumente vergleicht und das Ergebnis zurückgibt.
- Dies ist eine direkte **praktische Umsetzung** der Konzepte des Lambda-Kalküls: Funktionen als Parameter.

Fazit: Lambda-Kalkül und Java

- Das **Lambda-Kalkül** bildet die theoretische Grundlage für **anonyme Funktionen** und die **Berechnungsuniversaliät**.
- **Lambda-Ausdrücke** in Java basieren direkt auf diesen Prinzipien, bieten jedoch eine praktische und modernere Syntax.
- **Bezug**: Während das Lambda-Kalkül als Modell der Berechenbarkeit dient, ermöglichen Lambda-Ausdrücke in Java die Umsetzung dieser Konzepte in der täglichen Programmierung.
    - **Funktionales Paradigma**: Lambda-Ausdrücke in Java bringen funktionale Programmierkonzepte wie **first-class functions** und **closures** in eine objektorientierte Umgebung.

Comparable and Comparator

Lernziele

- Sie wissen, wie Sie das `Comparable`-Interface in Java implementieren.
- Sie wissen, wie Sie mit einem `Comparator`-Interface sortieren.
- Sie verstehen, wie Java Listen und Streams sortiert.
- Sie wissen, wie man Listenelemente nach mehreren Kriterien sortiert (z. B. Personen nach Name und Alter).
- Sie lernen, wie man mehrere Schnittstellen in einer Klasse implementiert.

Das Comparable-Interface

- `Comparable` ermöglicht das Vergleichen von Objekten einer Klasse.
- Die Methode `compareTo()` gibt eine Zahl zurück:
  - Negative Zahl: *this* kommt vor dem anderen Objekt.
  - Positive Zahl: *this* kommt nach dem anderen Objekt.
  - `0`: Beide Objekte sind gleich.

```
public int compareTo(T obj);
```

- Beispiele: Sortieren nach Größe, Namen, oder anderen Attributen.

Beispiel: Vergleich nach Körpergröße

```
public class Member implements Comparable<Member> {
    private String name;
    private int height;

    public Member(String name, int height) {
        this.name = name;
        this.height = height;
    }

    @Override
    public int compareTo(Member other) {
        return this.height - other.height;
    }
}
```

- Die Mitglieder werden anhand ihrer Körpergröße verglichen und sortiert.

Mehrere Sortierkriterien

```java
public class Member implements Comparable<Member> {
    private String name;
    private int height;

    public Member(String name, int height) {
        this.name = name;
        this.height = height;
    }

    public String getName() {
        return name;
    }

    public int getHeight() {
        return height;
    }

    @Override
    public int compareTo(Member other) {
        // Zuerst nach Körpergröße vergleichen
        int heightComparison = Integer.compare(this.height, other.height);
        if (heightComparison != 0) {
            return heightComparison; // Wenn die Höhen unterschiedlich sind, Rückgabe
        }
        // Wenn die Körpergrößen gleich sind, nach Namen vergleichen
        return this.name.compareTo(other.name);
    }
}
```

`compareTo` Quiz

Mehrere Sortierkriterien mit `Comparator`

- Sie können mehrere Kriterien zum Sortieren verwenden.

```
Comparator<Member> comparator = Comparator
    .comparing(Member::getHeight)
    .thenComparing(Member::getName);
Collections.sort(members, comparator);
```

- Mit `comparing` wird das erste Kriterium (Höhe) und mit `thenComparing` das zweite (Name) verwendet.

ohne Streams

```
Comparator<Member> comparator = new Comparator<>() {
    @Override
    public int compare(Member m1, Member m2) {
        // Zuerst nach der Höhe vergleichen
        int heightComparison = Integer.compare(m1.getHeight(), m2.getHeight());
        if (heightComparison != 0) {
            return heightComparison;  // Wenn die Höhen unterschiedlich sind, Rü
        }
        // Falls die Höhen gleich sind, nach dem Namen vergleichen
        return m1.getName().compareTo(m2.getName());
    }
};
```

Sortierung mit Lambda-Ausdrücken

- Sortieren nach Geburtsjahr (implizit: `Comparator`):

```
persons.stream()
    .sorted((p1, p2) -> p1.getBirthYear() - p2.getBirthYear())
    .forEach(p -> System.out.println(p.getName()));
```

- Sortieren nach Namen:

```
persons.stream()
    .sorted((p1, p2) -> p1.getName().compareTo(p2.getName()))
    .forEach(p -> System.out.println(p.getName()));
```

- Mit Lambda-Ausdrücken können Sie benutzerdefinierte Sortierungen erstellen.
- Wird `persons` verändert?
- `sorted` benötigt im Gegensatz zu üblichen Zwischenoperationen **alle** Elemente

Falls `Person` bereits `Comparable` implemetiert geht auch

```
persons.stream()
    .sorted()  // Verwendet die natürliche Ordnung, falls vorhanden
    .forEach(p -> System.out.println(p.getName()));
```

Alternativen

```
persons.sort((p1, p2) -> p1.getBirthYear() - p2.getBirthYear());
persons.forEach(p -> System.out.println(p.getName()));
```

```
persons.sort(Comparator.comparing(Person::getBirthYear));
```

- Wird `persons` verändert?

Beispiel: Identifizierbare und sortierbare Objekte

- Klasse, die **mehrere Schnittstellen** implementiert:

```
public interface Identifiable {
    String getId();
}

public class Employee implements Identifiable, Comparable<Employee> {
    private String name;
    private String employeeId;

    public Employee(String name, String employeeId) {
        this.name = name;
        this.employeeId = employeeId;
    }

    @Override
    public String getId() {
        return this.employeeId;
    }

    @Override // Identifiable garantiert, dass das funktioniert
    public int compareTo(Employee other) {
        return this.getId().compareTo(other.getId());
    }

    @Override
    public String toString() {
        return this.name + " (" + this.employeeId + ")";
    }
}
```

- Hier ist `Employee` sowohl identifizierbar als auch sortierbar.

Noch ein Beispiel: Sortieren nach mehreren Attributen

- Mehrere Kriterien für die Sortierung:

```
List<Employee> employees = new ArrayList<>();
employees.add(new Employee("Alice", "E002"));
employees.add(new Employee("Bob", "E001"));

Comparator<Employee> comparator = Comparator
    .comparing(Employee::getName)
    .thenComparing(Employee::getId);

Collections.sort(employees, comparator);

employees.forEach(e -> System.out.println(e));
```

- Erst nach Name, dann nach ID sortieren.

Vergleich von Comparable und Comparator

- Sie verstehen den Unterschied zwischen `Comparable` und `Comparator`.
- Sie wissen, wann man `Comparable` oder `Comparator` verwendet.
- Sie können Beispiele für beide Schnittstellen in Java implementieren.

Comparable: Signatur und Beispiel

- Wird innerhalb der Klasse implementiert.
- Definiert die natürliche Reihenfolge von Objekten.
- Signatur:

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

- Beispiel:

```
public class Member implements Comparable<Member> {
    private String name;
    private int height;

    @Override
    public int compareTo(Member other) {
        return this.height - other.height;
    }
}
```

- **Verwendung**: Für Klassen, die ihre eigene natürliche Reihenfolge haben.

Comparator: Signatur und Beispiel

- Wird außerhalb der Klasse verwendet.
- Definiert eine benutzerdefinierte Reihenfolge.
- Signatur:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- Beispiel (sehr einfach):

```
public class MemberHeightComparator implements Comparator<Member> {
    @Override
    public int compare(Member m1, Member m2) {
        return Integer.compare(m1.getHeight(), m2.getHeight());
    }
}
```

- **Verwendung**: Für mehrere oder alternative Sortierkriterien.

Vergleich: Comparable vs Comparator

Vergleichspunkt	Comparable	Comparator
Ort der Implementierung	In der Klasse selbst	Extern, in einer separaten Klasse
Methode	`compareTo(T obj)`	`compare(T o1, T o2)`
Verwendungszweck	Definiert natürliche Reihenfolge	Definiert benutzerdefinierte Reihenfolge
Veränderbarkeit	Ändert die Klasse	Keine Änderung der Klasse erforderlich
Beispiel	Sortieren nach einem Attribut	Sortieren nach mehreren Attributen oder Kriterien

- **Comparable**: Wird in der Klasse implementiert, um die natürliche Reihenfolge zu definieren.
- **Comparator**: Wird extern verwendet, um flexiblere oder alternative Sortierungen zu ermöglichen.

Beispiel: Sortieren mit Comparable

- Sortieren mit `Comparable`:

```
List<Member> members = new ArrayList<>();
members.add(new Member("Alice", 170));
members.add(new Member("Bob", 180));

Collections.sort(members); // Verwendet compareTo()
```

- Die Liste wird **in-place** nach dem Attribut `height` (Körpergröße) sortiert, das in `compareTo` der Klasse `Member` definiert ist.

```
@Override
public int compareTo(Member other) {
    return this.height - other.height;
}
```

Beispiel: Sortieren mit Comparator

- Sortieren mit einem `Comparator`:

```
List<Member> members = new ArrayList<>();
members.add(new Member("Alice", 170));
members.add(new Member("Bob", 180));

Collections.sort(members, new MemberHeightComparator());
```

- Der `MemberHeightComparator` wird verwendet, um die Liste nach Höhe zu sortieren.

```
public class MemberHeightComparator implements Comparator<Member> {
    @Override
    public int compare(Member m1, Member m2) {
        return Integer.compare(m1.getHeight(), m2.getHeight());
    }
}
```

Mehrere Kriterien mit Comparator

- Sie können mehrere Kriterien für die Sortierung verwenden:

```
Comparator<Member> comparator = Comparator
    .comparing(Member::getHeight)
    .thenComparing(Member::getName);
Collections.sort(members, comparator);
```

- `comparing`: Erstes Kriterium (Höhe).
- `thenComparing`: Zweites Kriterium (Name).

```
members.forEach(m -> System.out.println(m));
```

Was passiert hier?

```
Comparator<Member> comparator = Comparator
    .comparing(Member::getHeight)
    .thenComparing(Member::getName);
```

- Dieser Code definiert eine **Reihenfolge** für die Objekte der Klasse **Member**.
- Zuerst wird die **Größe** (Höhe) der Mitglieder verglichen.
- Falls zwei Mitglieder gleich groß sind, wird der **Name** als nächstes Kriterium verwendet.
- Dies ermöglicht eine **mehrstufige Sortierung**: erst nach **Größe**, dann nach **Name**.

Lambda-Ausdruck und Methodenreferenz (`::`)

- `Member::getHeight` ist eine Kurzform für einen **Lambda-Ausdruck**.

Vergleich:

- **Lambda-Ausdruck:**

```
(Member m) -> m.getHeight();
```

- **Methodenreferenz:**

```
Member::getHeight;
```

- Beide Varianten sagen dem Comparator, dass er die **Größe** eines Member-Objekts verwenden soll, um zwei Objekte zu vergleichen.
- Die **Methodenreferenz** ist eine kompaktere und lesbarere Variante des Lambda-Ausdrucks.

Wie `comparing` und `thenComparing` zusammenarbeiten

- `comparing(Member::getHeight)` erstellt den ersten Schritt: Es wird ein Comparator erzeugt, der die **Größe** vergleicht.
- Danach kommt `thenComparing(Member::getName)`: Es fügt den zweiten Schritt hinzu, der die **Namen** der Mitglieder vergleicht, falls die Größen gleich sind.
- Der ursprüngliche Vergleich wird durch `thenComparing` **überschrieben**, da der neue Comparator die gesamte Vergleichslogik übernimmt.
- `thenComparing` baut einen neuen Comparator auf, der alle vorherigen Vergleiche enthält und erweitert.

Was passiert mit den `Comparator`s?

- Wenn `comparing` aufgerufen wird, erzeugen wir einen **Comparator**, der nur nach **Größe** vergleicht.
- Mit `thenComparing` entsteht ein **neuer** Comparator, der sowohl den **Größenvergleich** als auch den **Namensvergleich** enthält.
- **Der alte Comparator existiert nicht mehr als separates Objekt**, er wird in den neuen Comparator integriert und überschrieben.

Kein Nebeneinander von Comparatoren

- **Jeder neue Comparator** enthält die Logik des vorherigen.
- Es gibt keine Co-Existenz von Comparatoren: Der neue Comparator integriert alle vorherigen Vergleiche.
- `thenComparing` erzeugt einen neuen **zusammengesetzten Comparator**, der Schritt für Schritt alle Kriterien abarbeitet.

Beispiel:

```
List<Member> members = ... // Liste von Membern
members.sort(heightComparator);  // Verwendet nur den ursprünglichen Vergleich
members.sort(heightAndNameComparator);  // Der erweiterte Comparator enthält all
```

Das Prinzip der Komposition

Komposition bedeutet, dass einfache Bausteine kombiniert werden können, um komplexere Logiken zu erstellen.

- Zuerst erstellen wir einen **Comparator**, der nach **Größe** sortiert.
- Anschließend erweitern wir ihn um einen **zweiten Schritt**: den Vergleich nach **Namen**.
- Jeder Schritt wird in den neuen Comparator integriert.

Vorteile der Komposition:

- **Wiederverwendbarkeit**: Jeder Baustein (z.B. der Größenvergleich) wird im neuen Comparator wiederverwendet.
- **Flexibilität**: Wir können nach Belieben neue Kriterien hinzufügen, ohne den ursprünglichen Code zu verändern.
- **Lesbarkeit und Wartbarkeit**: Der Code bleibt verständlich, da er Schritt für Schritt aufgebaut wird.

Fazit:

- Das **Prinzip der Komposition** fördert die **saubere Struktur** und **erleichtert den Aufbau komplexer Logik**, indem einfache Bausteine zusammengeführt werden.
- Jedes neue Kriterium ergänzt die bestehende Logik und ersetzt nicht die vorherige.

Comparable vs Comparator

- Verwenden Sie **Comparable**, wenn Sie eine natürliche Reihenfolge in der Klasse definieren möchten.
- Verwenden Sie **Comparator**, wenn Sie mehr Flexibilität oder alternative Sortierlogiken benötigen.
- **Comparator** ermöglicht auch das Sortieren nach mehreren Kriterien.

```
Comparator<Member> comparator = Comparator
    .comparing(Member::getHeight)
    .thenComparing(Member::getName);
```

Lernziele: Weitere nützliche Techniken

- Sie kennen die Probleme, die mit der Verkettung von Strings verbunden sind, und wissen, wie diese mit der StringBuilder-Klasse vermieden werden können.
- Sie verstehen reguläre Ausdrücke und können eigene schreiben.
- Sie kennen den Enumerated-Typ (enum) und wissen, wann er verwendet werden sollte.
- Sie wissen, wie man einen Iterator verwendet, um durch Datensammlungen zu gehen.

StringBuilder: Vermeidung von ineffizientem String-Verkettung

```
String numbers = "";
for (int i = 1; i < 5; i++) {
    numbers = numbers + i;
}
System.out.println(numbers);
```

- Das obige Programm erzeugt einen neuen String in jedem Schleifendurchlauf (`String` ist immutable).
- Jedes `+` -Zeichen erstellt einen neuen String im Speicher, was ineffizient ist.
- Lösung: Verwenden Sie die **StringBuilder-Klasse**, um String-Manipulationen effizienter zu machen.

```
StringBuilder numbers = new StringBuilder();
for (int i = 1; i < 5; i++) {
    numbers.append(i);
}
System.out.println(numbers.toString());
```

- Nur ein String wird erstellt, was viel effizienter ist.

String-Verkettung mit Zeilenumbrüchen

- Wenn Sie String-Verkettung mit Zeilenumbrüchen verwenden, führt jede `+`-Operation zu mehreren neuen Strings:

```
String numbers = "";
for (int i = 1; i < 5; i++) {
    numbers = numbers + i + "\n";
}
System.out.println(numbers);
```

- Dieses Programm erstellt 9 Strings! Verwenden Sie stattdessen:

```
StringBuilder numbers = new StringBuilder();
for (int i = 1; i < 5; i++) {
    numbers.append(i).append("\n");
}
System.out.println(numbers.toString());
```

- Effiziente Erstellung eines Strings mit **StringBuilder**.

`String` Quiz

`StringBuilder` Quiz

Warum Iteratoren verwenden?

- **Iteratoren** bieten mehr **Kontrolle** und **Flexibilität** als for-each oder for-Schleifen.
- Sie ermöglichen sicheres Entfernen und Hinzufügen von Elementen während der Iteration.
- Iteratoren sind universell einsetzbar und funktionieren für alle Collection-Typen.

Sicheres Entfernen während der Iteration

- Hier: **for-each-Schleife**: Entfernen von Elementen kann eine **ConcurrentModificationException** auslösen (die Liste, über iterieret wird, wird verändert!).

```
for (String element : list) {
    if (element.equals("B")) {
        list.remove(element);  // Fehler!
    }
}
```

- **Iterator**: Sicheres Entfernen mit der Methode `remove()`:

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String element = iterator.next();
    if (element.equals("B")) {
        iterator.remove();  // Sicheres Entfernen
    }
}
```

oder auch

```
for (int i = 0; i < list.size(); i++) {
    if (list.get(i).equals("B")) {
        list.remove(i);  // Entfernt das Element sicher
        i--;  // Index anpassen, um das Überspringen des nächsten Elements zu ve
    }
}
```

- Der Iterator wird informiert!

Mehr Kontrolle über den Iterationsprozess

- Iteratoren bieten präzise Kontrolle, z.B. Elemente überspringen, modifizieren oder entfernen.
- Beispiel:

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String element = iterator.next();
    if (element.equals("C")) {
        iterator.remove();  // Entfernen während der Iteration
    }
}
```

- **For-Schleifen** bieten nicht diese Flexibilität.

Universelles Werkzeug für Sammlungen

- Iteratoren funktionieren mit allen Klassen, die die **Collection-Schnittstelle** implementieren.

```
Iterator<String> iterator = set.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

- Sie sind einheitlich verwendbar für **Listen**, **Sets** und **Maps**.
- **For-Schleifen** erlauben keine Änderungen der Sammlung während der Iteration.

Vermeidung von Indexfehlern

- **Indexfehler** können auftreten, wenn Elemente während einer indexbasierten Schleife entfernt werden:

```
for (int i = 0; i <= list.size(); i++) {  // Fehler: <= statt <
    if (list.get(i).equals("B")) {
        list.remove(i);  // Führt zu IndexOutOfBoundsException!
    }
}
```

- **Grund:** Nach dem Entfernen wird `list.size()` kleiner, aber die Schleife versucht auf ungültige Indizes zuzugreifen.
- **Lösung: Verwende einen Iterator:**

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    if (iterator.next().equals("B")) {
        iterator.remove();  // Sicher
    }
}
```

- **Vorteil:** Konsistente Iteration, keine Fehler bei Modifikationen.

Iteratoren in komplexen Datenstrukturen

- **Iteratoren** sind besonders nützlich bei nicht-linearen Strukturen wie **Bäumen** oder **Graphen**, wo Schleifen möglicherweise nicht ausreichend sind.
- Sie bieten einen kontrollierten Zugang zu Elementen, unabhängig von der zugrunde liegenden Datenstruktur.

(Naïver) Iterator über Primzahlen

- Ein Iterator über Primzahlen kann effizienter sein als eine Schleife. Hier: die nächste Primzahl **on demand** berechnet.

```java
public class PrimeIterator implements Iterator<Integer> {
    private int current = 2;

    public boolean hasNext() {
        return true;  // Es gibt unendlich viele Primzahlen
    }

    public Integer next() {
        int nextPrime = current;
        current = findNextPrime(current);
        return nextPrime;
    }

    private int findNextPrime(int num) {
        int candidate = num + 1;
        while (!isPrime(candidate)) {
            candidate++;
        }
        return candidate;
    }

    private boolean isPrime(int num) {
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
    }
}
```

- Dieser Iterator berechnet die nächste Primzahl nur dann, wenn sie gebraucht wird (lazy evaluation).

Verwendung des Primzahl-Iterators

- Beispiel: Ausgabe der ersten 10 Primzahlen:

```
public class Main {
    public static void main(String[] args) {
        PrimeIterator primeIterator = new PrimeIterator();

        // Ausgabe der ersten 10 Primzahlen
        for (int i = 0; i < 10; i++) {
            System.out.println(primeIterator.next());
        }
    }
}
```

- Vorteile:
  - **Unendliche Mengen**: Berechnung nur auf Abruf.
  - **Flexibilität**: Keine Begrenzung auf eine vorgegebene Anzahl von Iterationen.
  - **Effizienz**: Keine unnötigen Berechnungen, wenn die nächste Primzahl nicht benötigt wird.

Fazit: Iteratoren vs. Schleifen

- Verwenden Sie **Iteratoren**, wenn Sie:
  - Elemente sicher während der Iteration hinzufügen oder entfernen müssen.
  - Mehr Kontrolle über den Iterationsprozess benötigen.
  - Einheitliche Methoden für verschiedene Sammlungstypen anwenden wollen.
  - Mit komplexen Datenstrukturen arbeiten müssen.
  - Unendliche Mengen wie Primzahlen handhaben wollen.
- **Iteratoren** bieten Flexibilität und Sicherheit, die Schleifen manchmal nicht bieten können.

Einführung in Reguläre Ausdrücke

Was sind Reguläre Ausdrücke (Regex)?

- Ein Werkzeug zur Mustererkennung in Zeichenketten
- Erlauben das Auffinden, Extrahieren und Ersetzen von Textsegmenten anhand vordefinierter Muster

Warum Reguläre Ausdrücke?

- **Effizienz:** Komplexe Suchmuster in wenigen Zeichen beschreiben
- **Flexibilität:** Anpassbar an unterschiedliche Textformate und -strukturen
- **Weit verbreitet:** In vielen Programmiersprachen, Texteditoren und Tools unterstützt

Grundlegende Syntaxelemente

- **.** (Punkt): Steht für ein beliebiges Zeichen
- * (Stern): Wiederholt den vorangehenden Ausdruck beliebig oft
- **+**: Wiederholt den vorangehenden Ausdruck mindestens einmal
- **?**: Vorangehendes Zeichen ist optional
- **^**: Markiert den **Anfang** der Zeichenkette (oder Zeile)
- **$**: Markiert das **Ende** der Zeichenkette (oder Zeile)
- **{n}**: Genau n Wiederholungen
- **{n,}**: Mindestens n Wiederholungen
- **{n,m}**: Mindestens n und höchstens m Wiederholungen

Nützliche Zeichenklassen

- **[abc]**: Ein beliebiges Zeichen aus der Gruppe a, b oder c
- **[^abc]**: Ein beliebiges Zeichen, außer a, b, c (Negation in der Zeichenklasse)
- **\d**: Ziffer [0-9] (nur in PCRE-Dialekten)
- **\w**: Wortzeichen [a-zA-Z0-9_]
- **\s**: Leerraumzeichen (Space, Tab, etc.)

Beispiele

- **E-Mail-Validierung:** `^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]$` (aaaaber!)
- **Telefonnummern:** `^\+?\d{1,3}[- ]?\d+$` (amerikanisch)
- **URLs:** `^https?:\/\/[^\s/$.?#].[^\s]*$`

Mit diesen Grundlagen lässt sich bereits ein Großteil der praktischen Anwendungen umsetzen.

Unterschiedliche Spezifikationen

- **POSIX-Standard:** Nutzt `[0-9]` oder `[[:digit:]]` für Ziffern, keine `\d`-Kurzschreibweise
- **Perl/PCRE-Dialekte:** Erlauben `\d` als Abkürzung für `[0-9]` und bieten erweiterte Features
- **Fazit:** Keine einheitliche Spezifikation – je nach Tool oder Sprache gelten unterschiedliche Dialekte, Erweiterungen und Syntaxregeln.

Erklärung des E-Mail-Regex

`^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$`

- `^` **und** `$` : Anfang und Ende der Zeichenkette
- `[A-Za-z0-9._%+-]+` : Benutzername-Teil – mindestens ein erlaubtes Zeichen
- `@` : Literal für das @-Symbol
- `[A-Za-z0-9.-]+` : Domain-Teil – mindestens ein Buchstabe, Ziffer, Punkt oder Minus
- `\.` : Literaler Punkt
- `[A-Za-z]{2,}` : Top-Level-Domain mit mindestens zwei Buchstaben

Ist `peter@.xy.de` eine valide E-Mail?

Nunja...

Siehe https://emailregex.com/

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:
[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f
[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-
[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:
[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)
```

und "korrigiert:

hier

und hier

Reguläre Ausdrücke: Ein Überblick (Java!)

- Ein **regulärer Ausdruck** ist eine kompakte Art, eine Menge von Strings zu definieren.
- Reguläre Ausdrücke werden verwendet, um die Korrektheit von Strings zu überprüfen.
- Beispiel: Überprüfung einer Matrikelnummer, die mit "01" beginnt und 7 Ziffern enthält:

```
String nummer = scanner.nextLine();
if (nummer.matches("01[0-9]{7}")) {
    System.out.println("Richtiges Format.");
} else {
    System.out.println("Falsches Format.");
}
```

- Der reguläre Ausdruck `"01[0-9]{7}"` beschreibt Matrikelnummern, die mit "01" beginnen, gefolgt von 7 Ziffern.

Wichtige Elemente regulärer Ausdrücke

- **Alternation**: Vertikale Linie als "oder"-Operator, z. B. `00|111|0000`
- **Klammern**: Begrenzen den Geltungsbereich eines Ausdrucks, z. B. `0000(0|1)`
- **Quantifizierer**:
  - `*` : 0 oder mehr Wiederholungen
  - `+` : 1 oder mehr Wiederholungen
  - `?` : 0 oder 1 Wiederholung
  - `{a}` : exakt a Wiederholungen
  - `{a,b}` : zwischen a und b Wiederholungen
  - `{a,}` : mindestens a Wiederholungen
- **Beispiel**:

```
String string = "trololololololo";
if (string.matches("tro(lo)+")) {
    System.out.println("Richtiges Format.");
}
```

RegEx Quiz

Es wir häßlich: Java Regex: Kette von geschlossenen Klammern

Beispiel: Matching von einer Reihe von `)` (mind. 1)

Java-Code:

```java
public class RegexExample {
    public static void main(String[] args) {
        String regex = "\\)+";

        System.out.println("))".matches(regex));  // true
        System.out.println("()".matches(regex));  // false
    }
}
```

Erklärung des Regex:

- `\\)`:
  - Die geschlossene Klammer `)` hat in Regex eine spezielle Bedeutung und muss escaped werden, um als normales Zeichen behandelt zu werden.
  - In Java-Strings wird ein Backslash (`\`) zusätzlich selbst escaped, weshalb zwei Backslashes (`\\`) erforderlich sind.
- `+`: Steht für "mindestens eine Wiederholung".
- Und wie matched man `\` ? Mit `\\\\`
  - In einem regulären Ausdruck (Regex) hat der Backslash () eine spezielle Bedeutung und muss deshalb mit einem weiteren Backslash (\) escaped werden.
  - In Java-Strings selbst muss ein einzelner Backslash ebenfalls escaped werden, weshalb `\\` zu `\\\\` wird.

Unterschied zu `grep`:

- `String regex = "\\)+";`
- In **Java's `String.matches()`** wird standardmäßig geprüft, ob der **gesamte String** zum Regex passt (implizites `^...$`).
- Tools wie **`grep`** suchen hingegen standardmäßig nach **Teilmatches**, d.h. der Regex muss nicht den gesamten String abdecken.

Beispiele:

Eingabe	Ergebnis mit `matches()`	Ergebnis mit `grep`-ähnlichem Verhalten (Teilmatch)
`))`	✅ True	✅ True
`()`	❌ False	✅ True

Warum `"()"` nicht passt:

- Der Regex `\\)+` sucht ausschließlich nach einer oder mehreren geschlossenen Klammern `)`.
- Da `matches()` den gesamten String prüft, schlägt das Matching bei `"()"` fehl, weil die offene Klammer `(` nicht im Regex enthalten ist.

Teilmatches in Java:

Um nach Teilmatches zu suchen (ähnlich wie `grep`), kann man `Pattern` und `Matcher.find()` verwenden:

```
import java.util.regex.*;

public class RegexExample {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("\\)+");
        Matcher matcher = pattern.matcher("()");

        System.out.println(matcher.find()); // true
    }
}
```

Zeichenklassen in regulären Ausdrücken

- **Zeichenklassen** spezifizieren eine Menge von zulässigen Zeichen, z. B. `[0–9]` oder `[a–c]`.
- Beispiel: Nur die Zeichen `a`, `b`, und `c` sind erlaubt:

```
String string = "abcabc";
if (string.matches("[a-c]*")) {
    System.out.println("Richtiges Format.");
}
```

- Zeichenklassen bieten eine kompakte Möglichkeit, Mengen von Zeichen zu definieren.

Praktische Übung: Reguläre Ausdrücke

Teil 1: Wochentage überprüfen

Erstellen Sie eine Methode, die überprüft, ob der übergebene String eine Abkürzung eines Wochentags (mon, tue, wed, etc.) ist.

```
public boolean isDayOfWeek(String string) {
    return string.matches("mon|tue|wed|thu|fri|sat|sun");
}
```

Teil 2: Vokale überprüfen

Erstellen Sie eine Methode, die überprüft, ob der String nur Vokale enthält:

```
public boolean allVowels(String string) {
    return string.matches("[aeiou]+");
}
```

Teil 3: Uhrzeit im Format `hh:mm:ss` überprüfen

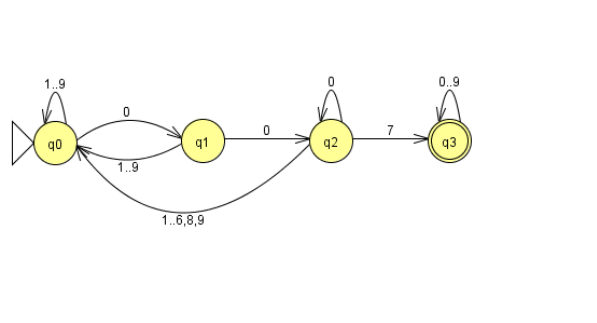
```
public boolean timeOfDay(String string) {
    return string.matches(" ... "); // your task
}
```

Was ist eine formale Sprache?

- Eine **formale Sprache** ist eine Menge von **Wörtern**, die über einem **Alphabet** definiert sind.
  - Ein **Alphabet** besteht aus einer endlichen Menge von Zeichen (z.B. `{a, b}`).
  - Ein **Wort** ist eine endliche Folge von Zeichen aus diesem Alphabet (z.B. `aab`).
- Eine formale Sprache wird durch eine **formale Definition** oder **Regeln** beschrieben.
  - Zum Beispiel können **Grammatiken** oder **reguläre Ausdrücke** verwendet werden, um eine formale Sprache zu definieren.
- Beispiel:
  - Sprache L: alle Wörter, die nur aus `a` und `b` bestehen und mit `a` beginnen.
  - Wörter in L: `a`, `ab`, `aaa`, `aab`, ...

Modelle zur Erkennung formaler Sprachen

- Es gibt verschiedene **formale Modelle**, die formale Sprachen erkennen können.
- Ein wichtiges Modell ist der **endliche Automat (DFA/NFA)**.
  - Ein **endlicher Automat** besteht aus:
    - Zuständen
    - Übergängen zwischen Zuständen
    - Eingabealphabet
    - Start- und Endzuständen
  - Endliche Automaten lesen Zeichen und entscheiden, ob ein Wort zur Sprache gehört.
- Beispiel: Ein endlicher Automat, der `007` erkennt.



- **Reguläre Ausdrücke** und **endliche Automaten** sind äquivalent: Beide erkennen genau die gleichen Sprachen.

Reguläre Sprachen und Programme

- Ein **Java-Programm** ist ebenfalls ein Wort in einer formalen Sprache.
    - In diesem Fall gehört es zu einer **Programmiersprache**, die durch eine formale **Grammatik** definiert ist.
- Eine **Grammatik** ist eine Menge von Regeln, die beschreibt, wie Wörter in einer Sprache gebildet werden.
    - Eine formale Sprache kann durch eine Grammatik definiert werden, die festlegt, welche **Regeln** auf das Alphabet angewendet werden.
- **Reguläre Ausdrücke** können reguläre Sprachen definieren, aber sie können keine Programmiersprachen wie Java vollständig beschreiben.
    - Reguläre Grammatiken erkennen nur **reguläre Sprachen**, was stark einschränkt, welche Strukturen definiert werden können.

Die Chomsky-Hierarchie

- Die **Chomsky-Hierarchie** ist eine Klassifizierung von Grammatiken und Sprachen.
    - Sie teilt Sprachen in vier Klassen ein:
        a. **Reguläre Sprachen** (erkannt durch endliche Automaten und reguläre Grammatiken)
        b. **Kontextfreie Sprachen** (erkannt durch Kellerautomaten)
        c. **Kontextsensitive Sprachen**
        d. **Rekursiv aufzählbare Sprachen** (erkannt durch Turingmaschinen)
- **Reguläre Grammatiken** erkennen **reguläre Sprachen**, aber komplexere Sprachen wie Java erfordern mächtigere Grammatiken (kontextfreie oder kontextsensitive).
- Beispiel:
    - Reguläre Sprache: Wörter, die nur aus `a` und `b` bestehen und mit `a` beginnen.
    - Programmiersprachen wie Java sind **kontextfrei** oder darüber hinaus.

Warum ist "Ist ein Wort in der Sprache?" wichtig?

- Die Frage, ob ein **Wort** in einer **Sprache** liegt, ist zentral in der Informatik.
    - **Sprachakzeptanz**: Endliche Automaten, Kellerautomaten, Turingmaschinen und andere Modelle prüfen, ob ein Wort den Regeln der Sprache entspricht.
- Beispiele:
    - **Java-Compiler** prüfen, ob ein Programm in der Java-Sprache geschrieben ist.
    - **Parser** für Datenformate wie JSON oder XML prüfen, ob die Daten dem Format entsprechen.
- Diese Frage ist entscheidend für die **Formalisierung** und **Validierung** von Programmen, Daten und Algorithmen.

Teaser

Abstract Syntax Tree

Programmierprinzipien: Effiziente Verarbeitung von Sammlungen

- **Streams** bieten eine abstrakte, funktionale Art, Sammlungen zu verarbeiten.
  - Prinzip der **Deklarativen Programmierung**: Fokus auf das **Was** statt auf das **Wie**.
  - **Lesbarkeit und Wartbarkeit**: Streams ermöglichen prägnanten und klaren Code.
  - **Effizienz**: Streams minimieren die Arbeitsspeicherauslastung, indem sie Daten nur bei Bedarf verarbeiten (lazy evaluation).
- **Transformation** statt **Iteration**: Streams fördern die Umwandlung von Daten, ohne den Zustand zu verändern, wodurch der Code leichter nachvollziehbar wird.

Prinzipien des Vergleichs: Comparable-Interface

- **Vergleichbarkeit**: Das Comparable-Interface fördert die **natürliche Ordnung** von Objekten in einer Sammlung.
  - Prinzip der **Objektorientierung**: Objekte "wissen", wie sie sich selbst vergleichen können.
  - **Flexibilität**: Implementierung in eigenen Klassen erlaubt es, Objekte sortierbar zu machen, ohne sie zu verändern.
- **Trennung von Logik und Struktur**: Comparable trennt die Sortierlogik vom restlichen Code und fördert **Kapselung**.
- **Verwendung von Comparatoren** für komplexere Sortierungen: Der Einsatz von Comparatoren erlaubt es, mehrere Kriterien zu kombinieren, was die **Wiederverwendbarkeit** von Code erhöht.

Weitere nützliche Techniken: Effizienz und Präzision

- **StringBuilder** und **Effiziente Ressourcenverwaltung**:
  - Prinzip der **Optimierung**: StringBuilder ermöglicht eine performante Manipulation von Strings ohne unnötigen Speicherverbrauch.
  - **Vermeidung von Redundanz**: Durch effiziente Techniken wie StringBuilder wird unnötige Ressourcenverwendung vermieden.
- **Reguläre Ausdrücke**:
  - **Mustererkennung** und **Präzision**: Reguläre Ausdrücke sind kraftvolle Werkzeuge zur Stringverarbeitung und fördern die Erstellung präziser Logiken.
  - Prinzip der **Abstraktion**: Reguläre Ausdrücke ermöglichen es, komplexe Muster kompakt und verständlich darzustellen, was die Wartbarkeit des Codes verbessert.
- **Iteratoren**:
  - **Flexibilität und Kontrolle**: Iteratoren bieten mehr Kontrolle über den Iterationsprozess und machen die Bearbeitung von Sammlungen flexibler.
  - Prinzip der **Trennung von Zustand und Prozess**: Ein Iterator hält den Zustand der Sammlung und den Prozess der Iteration getrennt.

Schlussgedanken: Kernprinzipien

1. **Abstraktion**: Sammlungen und deren Verarbeitung (z.B. mit Streams und Iteratoren) abstrahieren von der zugrunde liegenden Implementierung. Dies führt zu klarerem, wartbarem Code.
2. **Kapselung und Modularität**: Techniken wie das Comparable-Interface und Comparator fördern die Trennung von Logik und Datenstrukturen, was die Modularität erhöht.
3. **Effizienz**: Ressourcen sparend arbeiten, z.B. durch StringBuilder, Iteratoren und Lazy Evaluation in Streams, ist entscheidend für die Skalierbarkeit und Leistung großer Programme.
4. **Präzision**: Mit Werkzeugen wie regulären Ausdrücken und Sortiermechanismen können Entwickler präzise und kompakte Lösungen für spezifische Probleme schreiben.

Programmierprinzipien unterstützen die **Lesbarkeit**, **Wiederverwendbarkeit** und **Wartbarkeit** des Codes.