

Java StreamApi



- Die Stream Api ist eine effektive Api, um Operationen auf eine Sequenz von Elementen auszuführen
- Die Stream Api bietet die Möglichkeit auf alle Sequenzen von Elementen gleichermaßen zu operieren, z.B. Listen, Arrays, Sql Query results, ...
- Streams bilden einen Wrapper um eine Sequenz von Daten. Ein Stream speichert keine Daten und man kann über Streams nicht auf die Daten zugreifen, allerdings Funktionen auf diese ausführen
- Die Stream Api wurde mit Java 8 eingeführt



Java StreamApi

Warum die Stream Api verwenden? Nehmen wir an man möchte aus einer Liste von Zahlen alle gerade Zahlen in eine separate Liste speichern. Vor Java 8 wäre das nur so möglich:

```
List<Integer> aList = Arrays.asList(6,88,23,14,17,12,32,51,79,94);
List<Integer> evenNumbersList = new ArrayList<Integer>();

for(int num:aList)
{
    if(num %2 == 0){
        evenNumbersList.add(num);
    }
}
```

Mit Java 8:

```
List<Integer> aListWithStream = Arrays.asList(6,88,23,14,17,12,32,51,79,94);
aListWithStream.stream().filter(num -> num %2 == 0); // aListWithStream.stream() = create stream from
a collection
```

Verkettung von Operationen

Die Stream Api ermöglicht die Verkettung von mehreren Operationen. Z.B.:

```
List<String> names =  
Arrays.asList("Sunday", "Monday", "January", "Wednesday", "February", "December",  
"Saturday", "Monday", "Friday");  
  
names.stream()  
    .filter(name -> name.endsWith("day"))  
    .sorted()  
    .map(name -> name.toUpperCase())  
    .forEach(name -> System.out.println(name));
```



Umwandlung von Streams in Listen

Die Stream Api bietet Funktionen um Streams in Listen, Sets, Maps oder Collections umzuwandeln.

```
List<Integer> unsortedList = Arrays.asList(49,8,11,15,22);  
List<Integer> sortedList =  
unsortedList.stream().sorted().collect(Collectors.toList());
```

Optionals

- Optional ist ein generischer Datentyp
- Zweck: Vermeidung von NullPointerException
- Anstatt null zu returnieren, kann man ein leeres „Optional“ returnieren
- Vorteil: Der Compiler lässt es zu, dass man für null Objekt eine Funktion aufruft – z.B.:
// Annahme: Person p ist null, dann wäre folgende Zeile trotzdem möglich
p.getName();


// Annahme: Man verwendet ein Optional<Person> p, dann geht folgende Zeile nicht:
p.getName(); // Zuerst müsste man die Person mit p.get „holen“ und dann würde auffallen, dass
man es vorher auf isEmpty() überprüfen müsste

Erstellung und Prüfung von Optionals

- `Optional.empty()`: Erstellen eines leeren Optional
- `Optional.of(value)`: Erstellen eines Optional mit einem Nicht -Null-Wert
- `Optional.ofNullable(value)`: Erstellen eines Optional, das null sein kann
- `isPresent()`: Überprüfen, ob ein Wert vorhanden ist
- `get()`: Abrufen des Werts (Achtung: Kann `NoSuchElementException`)
- `orElse(defaultValue)`: Bereitstellen eines Standardwerts, falls Optional leer ist
- `orElseGet(Supplier)`: Bereitstellen eines Supplier -Funktionsinterfaces für

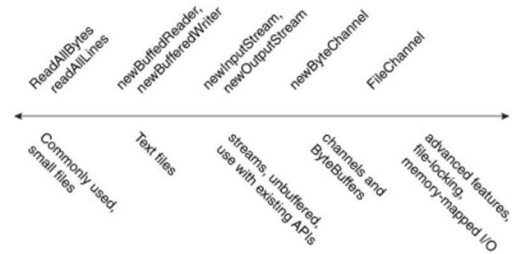


Best Practices

- Vermeidung der Nutzung von `get()` ohne `isPresent()` -Überprüfung
 - Bevorzugung von `orElseGet()` gegenüber `orElse()` für rechenintensive Standardwerte
- 

Fileio

- In Java gibt es einige Möglichkeiten und Klassen mit Files zu operieren
- Von links nach rechts werden dabei die Zugriffe auf Files komplexer in seinen Anwendungsfällen



Klasse Files

- **Files** besteht hauptsächlich aus statischen Methoden, sodass keine Instanz der Klasse erstellt werden muss, um die Methoden zu verwenden.
- Ermöglicht das Ausführen verschiedener Dateioperationen wie Erstellen, Lesen, Schreiben, Kopieren, Verschieben und Löschen von Dateien.
- Arbeitet eng mit der Klasse **Path** zusammen, um Dateipfade zu repräsentieren und zu manipulieren.
- Die meisten Files -Methoden werfen `IOException`, um Fehler bei Dateioperationen zu behandeln.

Lesen / Schreiben mit Files

```
// read from file
Path path = Paths.get("path/to/your/file.txt"); // oder Path.of(...)
try {
    List<String> content = Files.readAllLines(path);
} catch (IOException e) {
    throw new RuntimeException(e);
}

//write to file
Path path = Paths.get("path/to/your/file.txt");
String contentToAdd = "Dieser Text wird angehängt.\n";
try {
    Files.write(path, contentToAdd.getBytes());
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

StandardOpenOptions

- Viele der Lese- und Schreiboperationen nehmen noch optionale Parameter, die es ermöglichen ein bestimmtes Verhalten für Lese- und Schreiboperationen festzusetzen:
- `WRITE` - Opens the file for write access.
- `APPEND` - Appends the new data to the end of the file. This option is used with the `WRITE` or `CREATE` options.
- `TRUNCATE_EXISTING` - Truncates the file to zero bytes. This option is used with the `WRITE` option.
- `CREATE_NEW` - Creates a new file and throws an exception if the file already exists.
- `CREATE` - Opens the file if it exists or creates a new file if it does not.
- `DELETE_ON_CLOSE` - Deletes the file when the stream is closed. This option is useful for temporary files.
- `SPARSE` - Hints that a newly created file will be sparse. This advanced option is honored on some file systems, such as NTFS, where large files with data "gaps" can be stored in a more efficient manner where those empty gaps do not consume disk space.
- `SYNC` - Keeps the file (both content and metadata) synchronized with the underlying storage device.
- `DSYNC` - Keeps the file content synchronized with the underlying storage device.

Lesen/Schreiben mit BufferedStream I/O Klasse

- Mit der BufferedStream I/O Klasse werden BufferedStream I/O Klasse Lese - und Schreiboperationen über Buffer geregelt.
- Diese kann sowohl für Files als auch für andere Datenquellen wie zum Beispiel Sockets verwendet werden.
- Die BufferedStream I/O Klasse enthält sehr effektive Helperfunktionen, zum Beispiel um ein File Zeile für Zeile zu lesen.

Try with Ressource Statement

- Eine Ressource ist ein Objekt, dass am Ende des Programmes geschlossen werden muss.
- Dazu gehören alle Ressourcen, die vom Betriebssystem verwaltet werden.
- Wenn man eine solche Ressource „beantragt“ erhält man einen sog. Handler zurück. Mit diesem Handler kann man auf die Ressource zugreifen, am Ende muss man diese Ressource aber wieder freigeben.
- Verwendet man Try in Kombination mit Ressource Statement, „macht dastry Statement automatisch“ – ansonsten muss man es manuell machen.
- Jede Klasse die das Interface **java.lang.AutoCloseable** implementiert, kann als Ressource verwendet werden (dort befindet sich die close() Funktion, die Implizit aufgerufen wird und die man somit nicht separat aufrufen muss).

BufferedStream ohne Ressource Statement

```
File inputFile = new File( pathname: "input.txt");  
  
BufferedInputStream bufferedInputStream = null;  
try {  
    bufferedInputStream = new BufferedInputStream(new FileInputStream(inputFile));  
    //some file operations  
} catch (FileNotFoundException e) {  
    throw new RuntimeException(e);  
}  
finally {  
    if(bufferedInputStream != null) {  
        try {  
            bufferedInputStream.close();  
        } catch (IOException e) {  
            //...  
        }  
    }  
}
```

BufferedStreammit Ressource Statement

```
File inputFile = new File( pathname: "input.txt");  
  
try (BufferedInputStream bufferedInputStream = new BufferedInputStream(new FileInputStream(inputFile))) {  
    //some file operations  
    System.out.println();  
} catch (IOException e) {  
    throw new RuntimeException(e);  
}
```


Gemeinsames Beispiel Java

Lies aus einem .ini File die Felder und Ihre Konfigurationen ein sodass die Felder am Ende dynamisch generiert werden. Das Format darfst du frei wählen – z.B.:

```
Feld
id=1
name=Schlossstrasse
preis=500
preisProHaus=50
mieteBasis=50
mieteEinHaus=100
--
```

