

1) `findFirst()` und `Optional`

```
Stream.iterate(1,x->x+1)
    .filter(x->x%11==0 && x%13==0)
    .limit(10)
    .forEach(System.out::println);
```

liefert: 143, 286, 429, ..., 1430

>> wir wollen nur die erste Zahl, die durch 11 und 13 teilbar ist:

→ `findFirst()` .. terminal Operation, liefert ein `Optional`-Objekt, kann ausgegeben werden

```
System.out.println(Stream.iterate(1,x->x+1)
    .filter(x->x%11==0 && x%13==0)
    .findFirst()); //terminal Operation
```

liefert: `Optional[143]`

→ der Wert eines `Optional`s kann mit `.get()` ausgelesen werden

```
System.out.println(Stream.iterate(1,x->x+1)
    .filter(x->x%11==0 && x%13==0)
    .findFirst().get());
```

liefert: 143

➔ Wenn kein Wert gefunden wird ??

```
System.out.println(Stream.iterate(1,x->x+1)
    .limit(100)
    .filter(x->x%11==0 && x%13==0)
    .findFirst());
```

liefert: `Optional.empty`

→ `get()` führt in diesem Fall zu einer `Exception`

```
System.out.println(Stream.iterate(1,x->x+1)
    .limit(100)
    .filter(x->x%11==0 && x%13==0)
    .findFirst().get());
```

liefert: `.....NoSuchElementException: No value present`

→ Abhilfe: `.orElse(value)`

```
System.out.println(Stream.iterate(1,x->x+1)
    .limit(100)
    .filter(x->x%11==0 && x%13==0)
    .findFirst()
    .orElse(-1));
```

liefert: -1

2) flatMap

→ List <List<..>> zu einem Stream umwandeln

```
List<List<Integer>> vals = new ArrayList<>();  
vals.add(new ArrayList<>(Arrays.asList(1,2,3)));  
vals.add(new ArrayList<>(Arrays.asList(2,4,6)));  
vals.add(new ArrayList<>(Arrays.asList(3,6,9)));  
  
vals.stream().flatMap(x->x.stream()).forEach(System.out::println);  
liefert : 1, 2, 3, 2, 4, 6, 3, 6, 9
```

3) sortieren

.sorted() → "Standardsortierung"

```
Stream.of(1,33,22,-12,68,391,222,-23)  
    .sorted()  
    .forEach(System.out::println);  
liefert : -23, -12, 1, 22, 33, 68, 222, 391
```

.sorted(Comparator) → Sortierung mittels Comparator → compare-Methode !!

```
Stream.of(1,33,22,-12,68,391,222,-23)  
    .sorted((x,y)->x-y)  
    .forEach(System.out::println);  
liefert : -23, -12, 1, 22, 33, 68, 222, 391
```

→ "rückwärts" sortieren

```
Stream.of(1,33,22,-12,68,391,222,-23)  
    .sorted((x,y)->y-x)  
    .forEach(System.out::println);  
liefert : 391, 222, 68, 33, 22, 1, -12, -23
```

.. viele "vordefinierte" Methoden

a) Integer [Long, Double, String, ...].compare

```
Stream.of(1,33,22,-12,68,391,222,-23)  
    .sorted((a,b)->-Integer.compare(a,b))  
    .forEach(System.out::println);
```

b) comparing-Methode in Comparator

```
Stream.of(1,33,22,-12,68,391,222,-23)  
    .sorted(Comparator.comparing(Integer::intValue).reversed())  
    .forEach(System.out::println);
```

c) wenn die Elemente des Streams Comparable implementieren

```
Stream.of(1,33,22,-12,68,391,222,-23)  
    .sorted(Comparator.reverseOrder())  
    .forEach(System.out::println);
```

4) Files

– hat drei statische Methoden fürs Dateisystem:

* **list(directory)**: liefert alle Elemente des Verzeichnisses **directory**

```
Stream<Path> content = Files.list(Paths.get("resources"));
```

* **walk(directory[, maxDepth][, fileVisitOptions])**: liefert alle Elemente aus dem Verzeichnis **directory** und aus allen Unterverzeichnissen (inkl. aller Unterverzeichnissen)

maxDepth ist optional und gibt die maximale Verzeichnis-Tiefe an

fileVisitOptions ist optional und kann derzeit ausschließlich **fileVisitOptions.FOLLOW_LINKS** sein (folge symbolischen Links)

```
Stream<Path> content = Files.walk(Paths.get("../"));
```

* **find(directory, maxDepth, matcher[, fileVistiOptions])**: liefert alle Elemente des Verzeichnisses

```
Stream<Path> files = Files.find(Paths.get("..."),  
5,  
(path,attr) -> String.valueOf(path).endsWith(".java"));
```

– hat eine statische Methode zum Lesen von Textdateien:

* **lines(file)**: zum zeilenweise Lesen aus Textdateien

```
Stream<String> lines = Files.lines(Paths.get("resources/test.txt"),  
Charset.forName("UTF-8"));
```