Technische Hochschule Rosenheim

Modul- Fortgeschrittene Programmierkonzepte

Bachelor Informatik

13- Streams

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Advanced Stream Processing in Java

Korbinian Riedhammer

Technische Hochschule Rosenheim

Streams in Java

Technische Hochschule Rosenheim

Generation

- `Stream.of(...)` with array or varargs
- `Collection.stream()`, if supported
- `Stream.generate(...)` using a generator
- Popular APIs, e.g. `Pattern.compile("\\W").splitAsStream("hello world");`

Intermediate Operations

Terminal Operations

Intermediate Operations

`filter(Predicate<T> p)` removes/skips unwanted elements in the stream.

`map(Function<T, R> f)` transforms a `Stream<T>` into a `Stream<R>` using the provided `Function`

`sorted(Comparator<T> comp)` returns a sorted stream

`concat(Stream<T> s)` appends another stream

`distinct()` removes duplicates

`skip(int n)` and `limit(int n)` skip elements and truncate the stream

`flatMap(...)` flattens a list-of-lists into a single stream

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

Stream<Integer> integerStream = lol.flatMap(al -> al.stream());
integerStream.forEach(System.out::print);  // 12345
```

Terminal Operations

Technische Hochschule Rosenheim

Use `.forEach(Consumer<T> c)` to pass each element to the `Consumer`

Use `reduce` to combine (and optionally map) elements of a stream.

```
Stream.of(1, 3, 3, 7).reduce(0, Integer::sum));
// 14

Stream.of(1, 3, 3, 7).reduce(BigInteger.ZERO,
    (bi, i) -> bi.add(BigInteger.valueOf(i)),
    (bi1, bi2) -> bi1.add(bi2)));  // combine identity with first result
// 14
```

Use `collect` to collect/distribute elements to other structures.

```
List<Integer> list1 = new LinkedList<>();
Stream.of(1, 3, 3, 7).forEach(i -> list.add(i));

// or shorter, using collect
List<Integer> list2 = Stream.of(1, 3, 3, 7).collect(Collectors.toList()));
```

Collectors

Technische Hochschule Rosenheim

```java
// Accumulate names into a TreeSet
Set<String> set = people.stream()
    .map(Person::getName)
    .collect(Collectors.toCollection(TreeSet::new));

// Convert elements to strings and concatenate them, separated by commas
String joined = things.stream()
    .map(Object::toString)
    .collect(Collectors.joining(", "));

// Compute sum of salaries of employee
int total = employees.stream()
    .collect(Collectors.summingInt(Employee::getSalary));

// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));

// Compute sum of salaries by department
Map<Department, Integer> totalByDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment,
        Collectors.summingInt(Employee::getSalary)));

// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing = students.stream()
    .collect(Collectors.partitioningBy(s -> s.getGrade() <= 400));
```

Finding Values in a Stream

Technische Hochschule Rosenheim

Use `findFirst()`, `min()` or `max()` to find values, returns `Optional<T>`!

Verifying Values in a Stream

Use the `allMatch()`, `anyMatch()` and `noneMatch()` functions with a `Predicate<T>`.

Parallel Processing

Use `parallelStream()` for concurrent processing.

Well Done!

Well Done!