Java Streams API Cheatsheet

Java Streams API is a powerful tool for processing sequences of elements in a functional and declarative manner. This cheatsheet provides an overview of key concepts, including `map`, `filter`, `reduce` operations, and examples of stream usage.

Table of Contents

Introduction to Streams

- **Stream**: A sequence of elements that can be processed in a functional and declarative way.
- **Elements**: Stream can work with various data types (e.g., lists, arrays, files).
- **Pipeline**: Streams allow chaining multiple operations together for data transformation.
- **Lazy Evaluation**: Stream operations are evaluated only when necessary, improving efficiency.

Stream Operations

Map

- **Purpose**: Transforms each element in the stream into another element.
- **Method**: `map(Function<T, R> mapper)`
- **Example**:

```
List<String> names = List.of("Alice", "Bob", "Charlie");
List<Integer> nameLengths = names.stream()
                                 .map(String::length)
                                 .collect(Collectors.toList());
// nameLengths: [5, 3, 7]
```

Filter

- **Purpose**: Selects elements from the stream based on a condition.
- **Method**: `filter(Predicate<T> predicate)`
- **Example**:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
List<Integer> evenNumbers = numbers.stream()
                                   .filter(n -> n % 2 == 0)
                                   .collect(Collectors.toList());
// evenNumbers: [2, 4, 6, 8]
```

Reduce

- **Purpose**: Aggregates elements in the stream to a single result.
- **Method**: `reduce(T identity, BinaryOperator<T> accumulator)`
- **Example**:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .reduce(0, (a, b) -> a + b);
// sum: 15
```

Stream Usage Examples

Example 1: Find the Average

```
List<Double> grades = List.of(85.5, 92.0, 78.5, 90.5, 88.0);
double average = grades.stream()
                       .mapToDouble(Double::doubleValue)
                       .average()
                       .orElse(0.0);
// average: 88.7
```

Example 2: Filtering and Counting

```
List<String> words = List.of("apple", "banana", "cherry", "date");
long count = words.stream()
                  .filter(word -> word.startsWith("c"))
                  .count();
// count: 1
```

Example 3: Joining Strings

```
List<String> fruits = List.of("apple", "banana", "cherry");
String result = fruits.stream()
                      .collect(Collectors.joining(", "));
// result: "apple, banana, cherry"
```

Java Streams API offers a concise and expressive way to work with data sequences. These examples showcase the versatility and power of streams for various data manipulation tasks.