Modern Java Recipes
Ken Kousen, Kousen IT, Inc.
Modern Java Recipes

Cool Stuff You May Not Know
Videos

O'Reilly video courses: See Safari Books Online for details

Groovy Programming Fundamentals
Practical Groovy Programming
Mastering Groovy Programming
Learning Android
Practical Android
Gradle Fundamentals
Gradle for Android
Spring Framework Essentials
Advanced Java Development
Modern Java Recipes

Examples are from the book

Source code:

https://github.com/kousen/java_8_recipes
https://github.com/kousen/java_9_recipes
The Basics

- Streams
- Lambda Expressions
- Method References
Beyond the Basics

- Lazy streams
- The `peek` method
- `anyMatch`, `allMatch`, `noneMatch`
- Using `collect` and `reduce`
- `flatMap` and why should you care
- Using `Optional` as intended
- Deferred execution
- Partitioning and Grouping
- Downstream collectors
- Date-Time examples
Functional interfaces in java.util.function

**Consumer** → single arg, no result
   void accept(T t)

**Predicate** → returns boolean
   boolean test(T t)

**Supplier** → no arg, returns single result
   T get()

**Function** → single arg, returns result
   R apply(T t)
Functional Interfaces in java.util.function

**BiFunction** → binary function from T and U to R

```java
R apply(T, U)
```

**UnaryOperator** extends **Function** (T and R same type)

**BinaryOperator** extends **BiFunction** (T, U, and R same type)
Streams

A sequence of elements

Does not store the elements

Does not change the source

Operations are lazy when possible

Closed when terminal expression reached
Streams

A stream carries values from a source through a pipeline (zero or more intermediate operations) to a terminal operation.

Remember:

No elements are processed until the terminal operation.

LazyStreams.java
**count**

`long count()`

Returns the count of elements in this stream. This is a special case of a reduction and is equivalent to:

```java
    return mapToLong(e -> 1L).sum();
```

This is a terminal operation.

**Returns:**

the count of elements in this stream
Debugging with the peek() method

```java
def public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```
The peek() method

```java
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> {
            System.out.println(n);
            return n;
        })
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```
The peek() method

```java
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .peek(n -> System.out.printf("original: %d%n", n))
        .map(n -> n * 2)
        .peek(n -> System.out.printf("doubled: %d%n", n))
        .filter(n -> n % 3 == 0)
        .peek(n -> System.out.printf("filtered: %d%n", n))
        .sum();
}
```

`PeekDemo.java`, `PeekDemoTest.java`
anyMatch, allMatch, noneMatch

Short-circuiting, terminal operations on Stream

Return true if any, every, or none of the elements satisfy predicate

boolean anyMatch(Predicate)

boolean allMatch(Predicate)

boolean noneMatch(Predicate)
public boolean isPrime(int num) {
    int limit = (int) (Math.sqrt(num) + 1);
    return num == 2 || num > 1 && IntStream.range(2, limit)
        .noneMatch(divisor -> num % divisor == 0);
}
Prime number checker

@Test
public void testIsPrime() {
    assertTrue(Stream.of(2, 3, 5, 7, 11, 13, 17, 19)
                .allMatch(calculator::isPrime));
}

@Test
public void testIsPrimeWithComposites() {
    assertFalse(Stream.of(4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20)
                 .anyMatch(calculator::isPrime));
}

Primes.java, PrimesTest.java
Careful using any/all/none with empty stream

```java
@Test
public void emptyStreamsDanger() {
    assertTrue(Stream.empty().allMatch(e -> false));
    assertTrue(Stream.empty().noneMatch(e -> true));
    assertFalse(Stream.empty().anyMatch(e -> true));
}
```
What about that three-arg form of `collect`?

```java
<R> R collect(Supplier<R> supplier,
        BiConsumer<R,? super T> accumulator,
        BiConsumer<R,R> combiner)
```

We're used to using:

```java
<R,A> R collect(Collector<? super T, A, R> collector)
```
Utility methods in Collectors

Stream.of( ... )

.collect( Collectors.toList() ) → creates an ArrayList

.collect( Collectors.toSet() ) → creates a HashSet

.collect( Collectors.toCollection( Supplier ) )

→ creates the supplier (LinkedList::new, TreeSet::new, etc)

.collect( Collectors.toMap( Function, Function ) )

→ creates a map; first function is keys, second is values
collect with 3 args

Supplier\(<R>\)  → way to instantiate the result container

BiConsumer\(<R,T>\)  → way to add a single T to an R

BiConsumer\(<R,R>\)  → way to combine two R's
collect

Pseudo-code from the JavaDocs:

```java
R result = supplier.get();
for (T element : this stream)
    accumulator.accept(result, element);
return result;
```

Where is the combiner?
Imagine the process in parallel:

Elements: 1 ... 25 | 26 ... 50 | 51 ... 75 | 76 ... 100

Results:  R1      R2      R3      R4 supplier/accumulator

Total:     R        combiner
From a "strings as streams" example:

```java
.collect(StringBuilder::new, // Supplier
    StringBuilder::appendCodePoint, // Accumulator
    StringBuilder::append); // Combiner
```
Reduction Operations

Reduction operations

Terminal operations that produce

one value from a stream

average, sum, max, min, count, ...
Reduction Operations

Implementing Stream.reduce

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)
Transforming Streams

map:

<R> Stream<R> map(Function<T, R> mapper)

flatMap:

<R> Stream<R> flatMap(Function<T, Stream<R>> mapper)

Map from single element to multiple elements
Remove internal structure

FlatMapDemo.java
Deferred Execution

Logging

```java
log.info("x = " + x + ", y = " + y);  // arg is String

String formed even if not info level

log.info(() -> "x = " + x + ", y = " + y);

Only runs if at info level

Arg is a Supplier<String>
```
Deferred Execution

Any time you write a method that takes a complicated argument that should only be evaluated conditionally consider an overloaded method that takes a `Supplier`

For example:

```java
obj.myMethod( ... complicated ...)
```

```java
obj.myMethod(() → ... complicated ...)
```
Deferred Execution

Then, inside `myMethod`,

Check to see if argument is needed

If so, create it by calling `get()`

If not, just `return`
Optional

Alternative to returning object or null

`Optional<T> value`

- `isPresent()` → boolean
- `get()` → return the value

Remember: Never call `get` unless you're sure `Optional` is not empty
Optional

`ifPresent()` accepts a function

```java
optional.ifPresent( ... do something ...)```

`orElse()` provides an alternative

```java
optional.orElse(... default ...)```

`orElseGet()` supplies a default value

```java
optional.orElseGet(Supplier<? extends T> other)
```

`orElseThrow()` throws an exception

```java
optional.orElseThrow(Supplier<? extends X> exSupplier)
```
Partitioning and Grouping

Two methods in Collectors:

- `partitioningBy` → produce a Map with exactly two entries
  - true → list of elements that satisfy predicate
  - false → list of elements that don't satisfy predicate

- `groupingBy` → produce a Map with many entries
  - each key maps to list of elements with that key
Downstream Collectors

Same idea as partition or grouping, but
you don't want lists back
instead you want to post-process the lists
Downstream Collectors

Use Collectors methods with two (or more) arguments

first arg is the **Predicate** for partition or grouping

second arg post-processes the returned lists

data.stream()
  .collect(Collectors.partitioningBy(
    Predicate,
    Collectors.counting())); // or other downstream collector
# Downstream Collectors

<table>
<thead>
<tr>
<th>Stream method</th>
<th>Collectors method</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>counting</td>
</tr>
<tr>
<td>map</td>
<td>mapping</td>
</tr>
<tr>
<td>min</td>
<td>minBy</td>
</tr>
<tr>
<td>max</td>
<td>maxBy</td>
</tr>
<tr>
<td>sum</td>
<td>summingInt, summingLong, summingDouble</td>
</tr>
<tr>
<td>summarizing</td>
<td>summarizingInt, summarizingLong, summarizingDouble</td>
</tr>
</tbody>
</table>
Dates and Times

Java 8 Date-Time: java.time package

AntarcticaTimeZones.java

FunnyOffsets.java

DaysToElection.java
Summary

- Lazy streams
- Stream methods
  - collect
  - anyMatch, allMatch, noneMatch
  - reduce
  - peek
  - map vs flatMap
- Deferred Execution
- Partitioning, grouping, and downstream collectors
- Date/Time API
Thank you
GREAT INDIAN DEVELOPER SUMMIT 2019
Conference: April 23-26, Bangalore

Register early and get the best discounts!

www.developersummit.com  @greatindiandev  bit.ly/gidslinkedin