Object Oriented Programming in Java

Combining Lambdas Expressions

Suhel Hammoud

Combining Lambda Expressions

- `java.util.function` interfaces have default methods.
- These methods enable chaining and combining lambda expressions.
- Purpose: write simpler, more readable, and expressive code.

Chaining Predicates with Default Methods

Problem Statement

- Need to filter strings that are:
  - non-null
  - non-empty
  - shorter than 5 characters

Direct Lambda:

```java
Predicate<String> p = s -> (s != null) && !s.isEmpty() && s.length() < 5;
```

Predicate Chaining Example

- Improved version using and() default method:

```
Predicate<String> nonNull = s -> s != null;
Predicate<String> nonEmpty = s -> !s.isEmpty();
Predicate<String> shorterThan5 = s -> s.length() < 5;

Predicate<String> p = nonNull.and(nonEmpty).and(shorterThan5);
```

- Clearer, intent-driven code.

How It Works at the API Level

- `and()` is:
  - an instance method of `Predicate<T>`
  - takes another `Predicate<T>` as argument
  - returns a new `Predicate<T>`
- Must be a `default` method (only one abstract method allowed)
- Other helpful methods:
  - `or()`: combine with logical OR
  - `negate()`: logical NOT

Expressive Predicate Composition

```java
Predicate<String> isNull = Objects::isNull;
Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNullOrEmpty = isNull.or(isEmpty);
Predicate<String> isNotNullNorEmpty = isNullOrEmpty.negate();
Predicate<String> shorterThan5 = s -> s.length() < 5;

Predicate<String> p = isNotNullNorEmpty.and(shorterThan5);
```

- Combines method references and default methods.
- Improves clarity despite complexity.

Creating Predicates with Factory Methods

Factory Method 1: Predicate.isEqual

```
Predicate<String> isEqualToDuke = Predicate.isEqual("Duke");
```

- Tests if input equals "Duke".

Factory Method 2: Predicate.not

```
Predicate<Collection<String>> isEmpty = Collection::isEmpty;
Predicate<Collection<String>> isNotEmpty = Predicate.not(isEmpty);
```

Chaining Consumers with Default Methods

- Consumer<T> can be chained using andThen().

```java
Logger logger = Logger.getLogger("MyApplicationLogger");
Consumer<String> log = message -> logger.info(message);
Consumer<String> print = message -> System.out.println(message);

Consumer<String> logAndPrint = log.andThen(print);
```

- Executes log first, then print.

Chaining and Composing Functions

Chaining with andThen

```
Function<T, R> f1;
Function<R, V> f2;
Function<T, V> result = f1.andThen(f2);
```

- Applies f1, then passes result to f2.

Composing with compose

```
Function<T, R> f1;
Function<R, V> f2;
Function<T, V> result = f2.compose(f1);
```

- Also applies f1 first, then f2.

Chaining vs Composing Functions

- `f1.andThen(f2) == f2.compose(f1)`
- Order is the same, syntax is different.
- You can mix function types as long as:
    - output of `f1` is compatible with input of `f2`.

Creating an Identity Function

- Factory method: Function.identity()
- Returns a function that returns its input:

```java
Function<String, String> id = Function.identity();
```

- Works for any type T:

```java
Function<T, T> id = Function.identity();
```