Object Oriented Programming in Java

Using Lambdas Expressions in Your Application

Suhel Hammoud

Using Lambdas Expressions in Your Application

- Java SE 8 introduced lambda expressions.
- Major rewrite of JDK API: more updates than with generics in Java SE 5.
- Many existing interfaces became functional automatically.
- Your old interfaces may be implemented with lambdas without any modifications.

Discovering the java.util.function Package

- New package: java.util.function, part of java.base module.
- Functional interfaces heavily used in Collections and Stream API.
- Organized around four main types of interfaces.
- Understanding the core ones helps you understand the rest.

Creating or Providing Objects with Supplier<T>

Implementing the Supplier<T> Interface

- A Supplier<T> does not take arguments and returns an object.
- Interface has only one method: T get();.
- Example:

```
Supplier<String> supplier = () -> "Hello Duke!";
```

- Captures variables from surrounding scope (effectively final).

Using a Supplier<T>

- Example: Generating random numbers:

```
Random random = new Random(314L);
Supplier<Integer> rnd = () -> random.nextInt(10);

for (int index = 0; index < 5; index++) {
    System.out.println(rnd.get() + " ");
}
```

- Captured variable random is effectively final.

Specialized Suppliers

- Avoid unnecessary boxing/unboxing for performance.
- Example using `IntSupplier`:

```
Random random = new Random(314L);
IntSupplier rnd = () -> random.nextInt();
```

- Call `getAsInt()` instead of `get()`.
- Specialized Suppliers: `IntSupplier`, `BooleanSupplier`, `LongSupplier`, `DoubleSupplier`.

Consuming Objects with Consumer<T>

Implementing and Using Consumers

- A Consumer<T> takes an argument and returns nothing.
- Interface has one abstract method: void accept(T t);.
- Example:

```
Consumer<String> printer = s -> System.out.println(s);
```

Using Specialized Consumers

- Example:

```
Consumer<Integer> printer = i -> System.out.println(i);
```

- Auto-boxing may occur — can impact performance.
- Specialized Consumers: IntConsumer, LongConsumer, DoubleConsumer.

Consuming Two Elements with a BiConsumer

- BiConsumer<T, U> takes two arguments.
- Example:

```
BiConsumer<Random, Integer> randomNumberPrinter =
    (random, number) -> {
        for (int i = 0; i < number; i++) {
            System.out.println("next random = " + random.nextInt());
        }
    };
randomNumberPrinter.accept(new Random(314L), 5);
```

- Specialized versions: ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>.

Passing a Consumer to an Iterable

- forEach(Consumer) applies a Consumer to each element.
- Example:

```
List<String> strings = List.of("A", "B", "C");
Consumer<String> printer = s -> System.out.println(s);
strings.forEach(printer);
```

- Enables internal iteration and improves readability.

Testing Objects with Predicate<T>

Implementing and Using Predicates

- A Predicate<T> tests an object, returning a boolean.
- Interface:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

- Example:

```
Predicate<String> length3 = s -> s.length() == 3;
```

Using Specialized Predicates

- Example:

```
Predicate<Integer> isGreaterThan10 = i -> i > 10;
```

- Specialized Predicates: IntPredicate, LongPredicate, DoublePredicate.
- Reduces boxing/unboxing overhead.

Testing Two Elements with a BiPredicate

- `BiPredicate<T, U>` takes two arguments.
- Example:

```
BiPredicate<String, Integer> isOfLength = (word, length) -> word.length()
== length;
```

- No specialized primitive versions.

Passing a Predicate to a Collection

- `removeIf(Predicate)` method removes elements matching the predicate.
- Example:

```
List<String> strings = new ArrayList<>(List.of("one", "two", "three",
"four", "five"));
Predicate<String> isEvenLength = s -> s.length() % 2 == 0;
strings.removeIf(isEvenLength);
System.out.println(strings);
```

- Note:
  - Mutates the collection.
  - Cannot be used on immutable lists (like `List.of()`).

Mapping Objects to Other Objects with Function<T, R>

Implementing and Using Functions

- A Function<T, R> transforms a value.
- Interface:

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

- Example:

```
Function<String, Integer> toLength = s -> s.length();
```

Using Specialized Functions

- Specialized for different argument and return types:
  - Input: T, int, long, double
  - Output: T, int, long, double
- Specialized interfaces: `IntFunction<T>`, `ToIntFunction<T>`, etc.

UnaryOperator<T> and Specialized Functions

- UnaryOperator<T> = Function<T, T>.
- Example: Mathematical operations like sqrt, sin, log.
- Specialized versions: IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator.

Passing a Unary Operator to a List

- Example:

```
List<String> strings = Arrays.asList("one", "two", "three");
UnaryOperator<String> toUpperCase = word -> word.toUpperCase();
strings.replaceAll(toUpperCase);
System.out.println(strings);
```

- Modifies list elements in place.

Mapping Two Elements with a BiFunction

- `BiFunction<T, U, R>` takes two arguments, returns a value.
- Example:

```
BiFunction<String, String, Integer> findWordInSentence =
    (word, sentence) -> sentence.indexOf(word);
```

- Specialized versions: `IntBinaryOperator`, `ToIntBiFunction<T>`, etc.

Wrapping up the Four Categories of Functional Interfaces

- Four Main Categories:
  - Supplier: no argument, returns a value.
  - Consumer: takes an argument, returns nothing.
  - Predicate: takes an argument, returns a boolean.
  - Function: takes an argument, returns a transformed value.
- Variants for two arguments: BiConsumer, BiPredicate, BiFunction.
- Specialized interfaces avoid boxing/unboxing for primitives.
- Extensions: `UnaryOperator<T>`, `BinaryOperator<T>` for same-type transformations.