Handlers and Listeners

Josh Wilcox (jw14g24)

March 26, 2025

Table of Contents

Nested Classes

- A nested class is simply a class within a class
- There are 4 types:
  - **Static Nested**
  - **Inner Classes (Non-Static)**
    - Direct access to the attributes of the class *it is nested in*
  - **Local Inner Classes**
    - A class that is defined within the body of the method
    - The scope of the class is purely the method
  - **Anonymous Inner Classes**
    - Nested classes defined without a name

Benefits of nested classes

- **Logical Grouping:** Nested classes allow logically grouping classes that are only used in one place, improving code organization and readability.
- **Encapsulation:** They help encapsulate helper classes, making them inaccessible from outside the enclosing class.
- **Improved Readability:** By keeping related classes together, the code becomes easier to understand and maintain.
- **Access to Enclosing Class Members:** Inner classes (non-static) have direct access to the members (fields and methods) of their enclosing class, even private ones.
- **Reduced Namespace Pollution:** Since nested classes are scoped within their enclosing class, they do not pollute the top-level namespace.

Lambdas

- Lambdas act like references to **methods** rather than references to **objects**
- They can reference an existing function or can define a function on the fly - anonymous function
- They behave like function pointers in `C` (kind of)
- They can be defined like the following:

```
Interface lambdaName=InputArg -> function body;
```

- For example:

```
Comparator<Student> comparator = (o1, o2) -> o1.age - o2.age
```

Lmabda and Functional Interfaces

- `Predicate<T>` - Checks if a condition is true

```
Predicate<String> isLongWord = word -> word.length() > 5;

System.out.println(isLongWord.test("Hello")); // false
System.out.println(isLongWord.test("Lambda")); // true
```

- `Function<T,R>` - Takes an input type `T` and returns type `R`

```
Function<Integer, String> intToString = num -> "Number: " + num;

System.out.println(intToString.apply(10)); // Number: 10
```

- `BinaryOperator<T>` - Takes two values of the same type `T` and returns one of type `T`.

```
BinaryOperator<Integer> add = (a, b) -> a + b;
System.out.println(add.apply(3, 7)); // 10
```

- `BiPredicate<T,U>` - Checks a condition with two arguments of types `T` and `U`.

```
BiPredicate<String, Integer> isLongerThan = (str, len) -> str.length() > len;

System.out.println(isLongerThan.test("Hello", 3)); // true
System.out.println(isLongerThan.test("Hi", 3)); // false
```

Method Reference Operator ::

- The **Method Reference Operator (::)** is a shorthand notation for writing lambdas that call a specific method.
- It allows you to refer to methods or constructors directly by their names.
- There are four types of method references:
  - **Static Method Reference:** *ClassName*`::staticMethodName`

```
Function<String, Integer> parseInt = Integer::parseInt;
System.out.println(parseInt.apply("123")); // 123
```

  - **Instance Method Reference of a Particular Object:** *instance*`::instanceMethodName`

```
String str = "Hello";
Supplier<Integer> length = str::length;
System.out.println(length.get()); // 5
```

  - **Instance Method Reference of an Arbitrary Object of a Particular Type:** *ClassName*`::instanceMethodName`

```
Function<String, String> toUpperCase = String::toUpperCase;
System.out.println(toUpperCase.apply("hello")); // HELLO
```

  - **Constructor Reference:** *ClassName*`::new`

```
Supplier<List<String>> listSupplier = ArrayList::new;
List<String> list = listSupplier.get();
```

- Method references improve code readability and reduce boilerplate when using lambdas.