1Z0-808 Exam Topic Reviewer

TopicId: 1032
Topic: Lambda Expressions and Functional Interfaces

August 5, 2025

Introduction: The Functional Revolution in Java

Welcome, everyone. Before Java 8, handling events or passing behavior to a method often required creating bulky anonymous inner classes. It worked, but it was verbose. Java 8 introduced lambda expressions to provide a clear, concise way to represent anonymous functions, paving the way for a more functional style of programming. For the 1Z0-808 exam, mastering lambdas is not optional—it's fundamental. They are the gateway to the powerful Streams API and are tested extensively.

1 What is a Lambda Expression?

A lambda expression is essentially a short, anonymous method. It has no name, but it does have parameters, a body, and an optional return type. Its primary purpose is to implement the single abstract method of a functional interface.

1.1 Lambda Syntax: The Rules of Conciseness

The core syntax is simple: **(parameters) -¿ expression** or **(parameters) -¿ { statements; }**. Let's break down the variations. The compiler is smart, so you can often omit redundant information.

- **Full Syntax:** The types of the parameters are explicitly declared.

```
(String s1, String s2) -> s1.concat(s2)
```

- **Type Inference:** Most of the time, the compiler can infer the parameter types from the context (the functional interface).

```
(s1, s2) -> s1.concat(s2)
```

- **Single Parameter:** If there is only one parameter (and its type is inferred), you can omit the parentheses.

```
// Represents a function that takes a String and returns its length
s -> s.length()
```

- **No Parameters:** If there are no parameters, you must use empty parentheses.

```
// Represents an action that prints to the console
() -> System.out.println("Executing!")
```

- **Body Syntax:**
  - For a single expression, curly braces and the `return` keyword are optional. The result of the expression is implicitly returned.

    ```
    (int a, int b) -> a + b
    ```

- For a body with multiple statements, you **must** use curly braces, and if the method needs to return a value, you **must** use an explicit `return` statement.

```
(int a, int b) -> {
    int sum = a + b;
    System.out.println("Sum is: " + sum);
    return sum;
}
```

2 Functional Interfaces: The Target for Lambdas

A lambda expression, by itself, doesn't have a type. It only gets a type when it's assigned to a *functional interface*.

- **Definition:** A functional interface is any interface that contains **exactly one abstract method** (SAM).
- **Exam Trap:** Methods from `java.lang.Object` (like `equals()`, `hashCode()`, `toString()`) that are declared in an interface do **not** count towards the single abstract method limit.
- **Default and Static Methods:** An interface can have multiple `default` or `static` methods and still be a functional interface.

2.1 The `@FunctionalInterface` Annotation

This annotation is like `@Override`. It's not mandatory, but it's a best practice. It tells the compiler to verify that the interface meets the SAM requirement. If it doesn't, a compile-time error is generated. Using it prevents someone from accidentally adding another abstract method later and breaking existing lambdas.

```
@FunctionalInterface
interface Speaker {
    void speak(String message); // The single abstract method

    // This is ok!
    default void shout() {
        System.out.println("LOUD NOISES!");
    }
}

// Usage:
Speaker s = msg -> System.out.println("I say: " + msg);
s.speak("Hello"); // Prints "I say: Hello"
```

3 Standard Built-In Functional Interfaces

Java provides a set of common functional interfaces in the `java.util.function` package. You **must** know these for the exam.

- `Predicate<T>`
  Abstract Method: `boolean test(T t)` Use: Evaluates a condition. Example: `s -> s.isEmpty()`
- `Consumer<T>`
  Abstract Method: `void accept(T t)` Use: Performs an action with an object, returns nothing. Example: `s -> System.out.println(s)`
- `Supplier<T>`
  Abstract Method: `T get()` Use: Provides an object, takes no input. Example: `() -> new ArrayList<>()`
- `Function<T, R>`
  Abstract Method: `R apply(T t)` Use: Transforms an object of type T into one of type R. Example: `s -> s.length()`
- `UnaryOperator<T>` (extends `Function<T, T>`)
  Abstract Method: `T apply(T t)` Use: A special `Function` where input and output types are the same. Example: `s -> s.toUpperCase()`
- `BinaryOperator<T>` (extends `BiFunction<T, T, T>`)
  Abstract Method: `T apply(T t1, T t2)` Use: Combines two objects of the same type into a single object of that same type. Example: `(i1, i2) -> i1 + i2`

Note: There are also primitive specializations like `IntPredicate`, `LongSupplier`, `DoubleFunction` to avoid the performance cost of boxing and unboxing primitives.

4 Lambdas and Variable Scope: The Final Rule

This is a major source of tricky exam questions. Lambdas can *capture* variables from their enclosing scope.

- **Local Variables:** A lambda can access local variables from the enclosing method, but only if they are **final** or **effectively final**.
- **Effectively Final:** This means the variable's value is never changed after it's initialized. You don't need to write the `final` keyword, but you must treat it as if it were there.
- **Instance/Static Variables:** Lambdas can freely access and modify instance or static variables of the class. The 'final' restriction does not apply to them.

Example (Exam Focus):

```
public class ScopeTest {
    int instanceVar = 10; // Instance variable
    static int staticVar = 100; // Static variable
    public void testScope() {
        int localFinalVar = 1; // final
        int localEffectivelyFinalVar = 2; // effectively final
        int localMutableVar = 3; // NOT effectively final

        // VALID: can access final local var
        Consumer<String> c1 = s -> System.out.println(s + localFinalVar);

        // VALID: can access effectively final local var
        Consumer<String> c2 = s -> System.out.println(s + localEffectivelyFinalVa

        // COMPILER ERROR: localMutableVar is modified later
        // Consumer<String> c3 = s -> System.out.println(s + localMutableVar);
        localMutableVar++;

        // VALID: can access and modify instance/static variables
        Consumer<String> c4 = s -> {
            instanceVar++;
            staticVar++;
            System.out.println("instanceVar is now " + instanceVar);
        };
        c4.accept("");
    }
}
```

Key Takeaways for the 1Z0-808 Exam

- Know the lambda syntax variations cold. When are `()`, {}, and `return` required or optional?
- A functional interface has exactly one abstract method. `default`, `static`, and `Object` methods don't count.
- Memorize the main functional interfaces from `java.util.function`: `Predicate`, `Consumer`, `Supplier`, `Function`, and the two operators.
- The most critical rule: Lambdas can only capture **final** or **effectively final** local variables. This rule does not apply to instance or static fields.