Java Lambda Expressions and Method References Tutorial

Table of Contents

---

1. Anonymous Inner Classes

What are Anonymous Inner Classes?

Anonymous inner classes are classes without a name that are defined and instantiated at the same time. They're commonly used for implementing interfaces or extending classes for one-time use.

Use Cases:

- Event handling in GUI applications
- Implementing callback functions
- Quick implementations of interfaces
- Custom comparators for sorting

Example:

```java
```

```java
// Traditional anonymous inner class
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running in anonymous class");
    }
};

// Comparator example
List<String> names = Arrays.asList("John", "Alice", "Bob");
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

Problems with Anonymous Inner Classes:

- Verbose syntax
- Reduced readability
- Boilerplate code
- Performance overhead

---

2. Functional Interfaces

What are Functional Interfaces?

A functional interface is an interface that has exactly one abstract method. They can have multiple default or static methods.

Key Points:

- **Single Abstract Method (SAM)**: Only one abstract method
- **@FunctionalInterface**: Optional annotation for compile-time checking
- **Lambda Target**: Can be implemented using lambda expressions
- **Built-in Support**: Java 8+ provides many built-in functional interfaces

Examples:

java

```java
@FunctionalInterface
public interface Calculator {
    int calculate(int a, int b);

    // Default methods are allowed
    default void print() {
        System.out.println("Calculating...");
    }
}

@FunctionalInterface
public interface StringProcessor {
    String process(String input);
}
```

---

3. Lambda Expressions

What are Lambda Expressions?

Lambda expressions are a concise way to represent anonymous functions. They provide a clear and concise way to implement functional interfaces.

Syntax:

```
(parameters) -> expression
(parameters) -> { statements; }
```

Key Points:

- **Concise Syntax**: Reduces boilerplate code
- **Type Inference**: Compiler infers parameter types
- **Functional Programming**: Enables functional programming paradigms
- **Immutable**: Encourages immutable programming

Examples:

```
java
```

```java
// Simple lambda
Runnable r = () -> System.out.println("Hello Lambda");

// With parameters
Calculator add = (a, b) -> a + b;
Calculator multiply = (a, b) -> {
    int result = a * b;
    System.out.println("Result: " + result);
    return result;
};

// String processing
StringProcessor toUpper = s -> s.toUpperCase();
StringProcessor reverse = s -> new StringBuilder(s).reverse().toString();
```

---

4. Method References

What are Method References?

Method references are a shorthand notation for lambda expressions that call a single method. They make code more readable and concise.

Types of Method References:

1. Static Method Reference

Syntax: `ClassName::staticMethod` **Byte-sized Points**:

- References static methods directly
- No object instance needed
- Clean syntax for utility methods

Example:

```java
// Lambda: x -> Math.sqrt(x)
Function<Double, Double> sqrt = Math::sqrt;

// Lambda: (a, b) -> Integer.compare(a, b)
Comparator<Integer> comparator = Integer::compare;
```

2. Instance Method Reference

Syntax: `instance::instanceMethod` **Byte-sized Points**:

- References method on specific object instance
- Object is bound at creation time
- Useful for callback patterns

Example:

```java
String prefix = "Hello ";
Function<String, String> greeter = prefix::concat;
System.out.println(greeter.apply("World")); // Hello World

PrintStream out = System.out;
Consumer<String> printer = out::println;
```

3. Instance Method Reference of Arbitrary Object

Syntax: `ClassName::instanceMethod` **Byte-sized Points**:

- First parameter becomes the object to call method on
- Remaining parameters become method arguments
- Common with collection operations

Example:

```java
// Lambda: s -> s.length()
Function<String, Integer> lengthFunction = String::length;

// Lambda: s -> s.toUpperCase()
Function<String, String> upperCase = String::toUpperCase;

List<String> names = Arrays.asList("john", "alice", "bob");
names.stream().map(String::toUpperCase).forEach(System.out::println);
```

4. Constructor Reference

Syntax: `ClassName::new` **Byte-sized Points**:

- Creates new instances using constructor
- Works with any constructor (no-arg, parameterized)
- Useful for factory patterns

Example:

```java
// Lambda: () -> new ArrayList<>()
Supplier<List<String>> listSupplier = ArrayList::new;

// Lambda: s -> new StringBuilder(s)
Function<String, StringBuilder> builderFunction = StringBuilder::new;

// Creating objects from stream
List<String> names = Arrays.asList("John", "Alice");
List<Person> persons = names.stream()
    .map(Person::new)
    .collect(Collectors.toList());
```

---

5. Stream API Basics

What is Stream API?

Stream API provides a functional approach to processing collections of objects. It allows for declarative programming style with operations like filter, map, reduce.

Key Stream Methods:

filter()

Byte-sized Points:

- Filters elements based on a predicate
- Returns a new stream with matching elements
- Intermediate operation (lazy evaluation)
- Preserves order of elements

Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> evenNumbers = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList()); // [2, 4, 6]
```

map()

Byte-sized Points:

- Transforms each element using a function
- One-to-one mapping of elements
- Intermediate operation
- Changes element type if needed

Example:

```java
List<String> names = Arrays.asList("john", "alice", "bob");
List<String> upperNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList()); // [JOHN, ALICE, BOB]

List<Integer> lengths = names.stream()
    .map(String::length)
    .collect(Collectors.toList()); // [4, 5, 3]
```

forEach()

Byte-sized Points:

- Performs action on each element
- Terminal operation
- Returns void
- Side-effect operation

Example:

```java
List<String> names = Arrays.asList("John", "Alice", "Bob");
names.stream().forEach(System.out::println);

// With lambda
names.stream().forEach(name -> System.out.println("Hello " + name));
```

collect()

Byte-sized Points:

- Terminal operation that accumulates elements
- Transforms stream back to collection
- Highly customizable with Collectors class

- Mutable reduction operation

Example:

```java
List<String> names = Arrays.asList("John", "Alice", "Bob");

// To List
List<String> list = names.stream().collect(Collectors.toList());

// To Set
Set<String> set = names.stream().collect(Collectors.toSet());

// Joining strings
String joined = names.stream().collect(Collectors.joining(", "));
```

reduce()

Byte-sized Points:

- Combines stream elements into single result
- Takes binary operator for combining elements
- Returns Optional for empty streams
- Immutable reduction operation

Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Sum using reduce
Optional<Integer> sum = numbers.stream().reduce((a, b) -> a + b);
// Or with method reference
Optional<Integer> sum2 = numbers.stream().reduce(Integer::sum);

// With identity value
Integer sum3 = numbers.stream().reduce(0, Integer::sum); // 15
```

sorted()

Byte-sized Points:

- Sorts stream elements
- Natural ordering or custom comparator

- Intermediate operation
- Creates new sorted stream

Example:

```java
List<String> names = Arrays.asList("John", "Alice", "Bob");

// Natural sorting
List<String> sorted = names.stream()
    .sorted()
    .collect(Collectors.toList()); // [Alice, Bob, John]

// Custom comparator
List<String> byLength = names.stream()
    .sorted(Comparator.comparing(String::length))
    .collect(Collectors.toList()); // [Bob, John, Alice]
```

distinct()

Byte-sized Points:

- Removes duplicate elements
- Uses equals() method for comparison
- Intermediate operation
- Maintains encounter order

Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 3, 4);
List<Integer> unique = numbers.stream()
    .distinct()
    .collect(Collectors.toList()); // [1, 2, 3, 4]
```

limit()

Byte-sized Points:

- Limits stream to first n elements
- Short-circuiting intermediate operation
- Useful for pagination
- Preserves encounter order

Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> first3 = numbers.stream()
    .limit(3)
    .collect(Collectors.toList()); // [1, 2, 3]
```

skip()

Byte-sized Points:

- Skips first n elements
- Intermediate operation
- Useful with limit for pagination
- Returns remaining elements

Example:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> after2 = numbers.stream()
    .skip(2)
    .collect(Collectors.toList()); // [3, 4, 5, 6]
```

---

6. Built-in Functional Interfaces

Predicate<T>

Byte-sized Points:

- Tests a condition and returns boolean
- Method: `boolean test(T t)`
- Used in filtering operations
- Can be combined with and(), or(), negate()

Example:

```java
```

```java
Predicate<Integer> isEven = n -> n % 2 == 0;
Predicate<String> isLong = s -> s.length() > 5;

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> evenNumbers = numbers.stream()
    .filter(isEven)
    .collect(Collectors.toList());

// Combining predicates
Predicate<Integer> isPositiveEven = n -> n > 0 && n % 2 == 0;
Predicate<Integer> combined = isEven.and(n -> n > 3);
```

Function<T, R>

Byte-sized Points:

- Transforms input of type T to output of type R
- Method: `R apply(T t)`
- Used in mapping operations
- Can be chained with andThen(), compose()

Example:

```java
Function<String, Integer> stringLength = String::length;
Function<Integer, String> intToString = Object::toString;

List<String> names = Arrays.asList("John", "Alice", "Bob");
List<Integer> lengths = names.stream()
    .map(stringLength)
    .collect(Collectors.toList());

// Chaining functions
Function<String, String> processString = stringLength
    .andThen(intToString)
    .andThen(s -> "Length: " + s);
```

Consumer<T>

Byte-sized Points:

- Accepts input and returns nothing
- Method: `void accept(T t)`

- Used for side-effects
- Can be chained with andThen()

Example:

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

List<String> names = Arrays.asList("John", "Alice", "Bob");
names.forEach(printer);

// Chaining consumers
Consumer<String> combined = printer.andThen(upperPrinter);
```

Supplier<T>

Byte-sized Points:

- Supplies a value without taking input
- Method: `T get()`
- Used for lazy evaluation
- Factory method pattern

Example:

```java
Supplier<String> stringSupplier = () -> "Hello World";
Supplier<Double> randomSupplier = Math::random;
Supplier<List<String>> listSupplier = ArrayList::new;

// Lazy evaluation
Optional<String> optional = Optional.empty();
String result = optional.orElseGet(stringSupplier);
```

BiPredicate<T, U>

Byte-sized Points:

- Tests condition with two inputs
- Method: `boolean test(T t, U u)`
- Returns boolean result
- Can be combined like Predicate

Example:

```java
BiPredicate<String, Integer> lengthCheck = (s, len) -> s.length() == len;
BiPredicate<Integer, Integer> isEqual = Integer::equals;

boolean result = lengthCheck.test("Hello", 5); // true
```

BiFunction<T, U, R>

Byte-sized Points:

- Takes two inputs and produces one output
- Method: `R apply(T t, U u)`
- Used in reduce operations
- Can be chained with andThen()

Example:

```java
BiFunction<Integer, Integer, Integer> add = Integer::sum;
BiFunction<String, String, String> concat = String::concat;

Integer sum = add.apply(5, 3); // 8
String combined = concat.apply("Hello", " World"); // Hello World
```

BiConsumer<T, U>

Byte-sized Points:

- Accepts two inputs and returns nothing
- Method: `void accept(T t, U u)`
- Used for side-effects with two parameters
- Can be chained with andThen()

Example:

```java
```

```java
BiConsumer<String, Integer> printWithLength =
    (s, len) -> System.out.println(s + " has length " + len);

Map<String, Integer> map = new HashMap<>();
BiConsumer<String, Integer> mapPutter = map::put;
```

UnaryOperator<T>

Byte-sized Points:

- Special Function where input and output are same type
- Method: `T apply(T t)`
- Extends Function<T, T>
- Common in mathematical operations

Example:

java

```java
UnaryOperator<Integer> square = x -> x * x;
UnaryOperator<String> toUpper = String::toUpperCase;

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
List<Integer> squared = numbers.stream()
    .map(square)
    .collect(Collectors.toList()); // [1, 4, 9, 16]
```

BinaryOperator<T>

Byte-sized Points:

- Special BiFunction where both inputs and output are same type
- Method: `T apply(T t1, T t2)`
- Extends BiFunction<T, T, T>
- Used in reduction operations

Example:

java

```java
BinaryOperator<Integer> max = Integer::max;
BinaryOperator<String> concat = String::concat;

List<Integer> numbers = Arrays.asList(1, 5, 3, 9, 2);
Optional<Integer> maximum = numbers.stream().reduce(max); // 9

Integer sum = numbers.stream().reduce(0, Integer::sum); // 20
```

---

7. Practice Questions

Anonymous Inner Classes & Functional Interfaces (6 questions)

1. Convert the following anonymous inner class to a lambda expression and then to a method reference:

```java
Comparator<String> comp = new Comparator<String>() {
    public int compare(String a, String b) {
        return a.compareToIgnoreCase(b);
    }
};
```

2. Create a custom functional interface called `TriFunction<T, U, V, R>` that takes three parameters and returns a result. Implement it using lambda expressions to calculate the area of a triangle.
3. Write a program that uses an anonymous inner class to implement a `Validator` interface that checks if a string contains only digits. Then convert it to lambda expression.
4. Create a functional interface `StringTransformer` with a method that takes two strings and returns a string. Implement it to merge two strings with a separator using lambda expressions.
5. Convert this anonymous inner class to lambda: Create a `Runnable` that prints numbers 1 to 5 with a 1-second delay between each number.
6. Design a functional interface `Calculator` with a method that performs arithmetic operations. Create implementations for add, subtract, multiply, and divide using lambda expressions.

Lambda Expressions (6 questions)

7. Write lambda expressions to sort a list of `Person` objects by age (ascending), then by name (descending), and finally by city (ascending).
8. Create a lambda expression that takes a list of integers and returns a new list containing only the squares of even numbers.
9. Using lambda expressions, write code to find all strings in a list that start with a specific letter and have length greater than 3.

10. Write a lambda expression that takes two `Optional<Integer>` values and returns their sum if both are present, otherwise returns 0.

11. Create lambda expressions to implement different discount calculation strategies: 10% off for students, 15% off for seniors, and 5% off for everyone else.

12. Write a lambda expression that groups a list of employees by their department and calculates the average salary for each department.

Method References (6 questions)

13. Convert these lambda expressions to method references:

```java
Function<String, Integer> f1 = s -> s.length();
Predicate<String> f2 = s -> s.isEmpty();
Consumer<String> f3 = s -> System.out.println(s);
```

14. Create a method reference for a constructor that creates `Person` objects from a single string parameter (assuming appropriate constructor exists).

15. Write code using method references to convert a list of strings to uppercase, then sort them, and finally print each one.

16. Use method references to create a `Comparator` that compares `Employee` objects first by salary (descending) and then by name (ascending).

17. Convert this code to use method references:

```java
list.stream()
  .filter(x -> x.isActive())
  .map(x -> x.getName())
  .forEach(x -> System.out.println(x));
```

18. Create method references for: getting current time, generating random numbers, and creating empty lists.

Stream API - filter() (5 questions)

19. Given a list of integers, use `filter()` to find all numbers that are divisible by both 3 and 5.

20. Filter a list of `Product` objects to find products with price between $50 and $200 and rating above 4.0.

21. From a list of strings, filter out all strings that contain digits or special characters, keeping only alphabetic strings.

22. Filter a list of dates to find all dates that fall on weekends in the current year.

23. Create a complex filter that finds all employees who are either managers with more than 5 years of experience OR senior developers with salary above $80,000.

Stream API - map() (5 questions)

24. Use `map()` to convert a list of file paths (strings) into a list of `File` objects, then map them to their file sizes.
25. Given a list of `Order` objects, use `map()` to extract customer emails, then transform them to lowercase and remove duplicates.
26. Map a list of integers to their binary string representations, then map those to their lengths.
27. Transform a list of `Person` objects to a list of formatted strings in the format "Name: [name], Age: [age]".
28. Use nested mapping to flatten a list of `Department` objects (each containing a list of `Employee` objects) into a single list of employee names.

Stream API - forEach() (4 questions)

29. Use `forEach()` to print all elements in a list with their index positions.
30. Given a map of student names and their grades, use `forEach()` to print each student's grade status (Pass/Fail based on grade >= 60).
31. Use `forEach()` to save each `Customer` object in a list to a database (simulate with print statements).
32. Create a side effect using `forEach()` that counts how many strings in a list start with each letter of the alphabet.

Stream API - collect() (5 questions)

33. Use `collect()` to group a list of `Transaction` objects by currency and calculate the total amount for each currency.
34. Collect employee names into a comma-separated string, but only for employees in the IT department.
35. Use `collect()` to create a map where keys are product categories and values are lists of product names in that category.
36. Collect the top 5 highest-paid employees into a custom `EmployeeSummary` object that contains their names and average salary.
37. Use partitioning collector to separate a list of numbers into even and odd numbers, then collect each partition into separate sorted lists.

Stream API - reduce() (5 questions)

38. Use `reduce()` to find the longest string in a list of strings.

39. Reduce a list of `BankAccount` objects to calculate the total balance across all accounts.
40. Use `reduce()` to find the oldest person in a list of `Person` objects.
41. Create a custom reduction that concatenates all non-empty strings in a list with a delimiter, skipping null values.
42. Use `reduce()` with three parameters (identity, accumulator, combiner) to calculate the product of all positive numbers in a parallel stream.

Stream API - sorted() (4 questions)

43. Sort a list of `Book` objects first by publication year (descending), then by title (ascending), and finally by author (ascending).
44. Create a custom sorting logic that sorts strings by length first, and then alphabetically for strings of the same length.
45. Sort a list of `Employee` objects using multiple criteria: department name, then by salary (descending), then by hire date (ascending).
46. Sort a mixed list of positive and negative integers such that negative numbers come first (sorted ascending), followed by positive numbers (sorted descending).

Built-in Functional Interfaces (8 questions)

47. Create a `Predicate<String>` that checks if a string is a valid email address, then combine it with another predicate that checks minimum length using `and()`.
48. Use `Function<T,R>` to create a chain of transformations: String → trim → toLowerCase → remove vowels → reverse.
49. Create a `Consumer<List<String>>` that prints each string in the list along with its length, then chain it with another consumer that prints the total count.
50. Design a `Supplier<Password>` that generates secure random passwords with configurable length and character sets.
51. Use `BiPredicate<Person, Integer>` to check if a person's age is within a specified range, then use it to filter a list of people.
52. Create a `BiFunction<List<Integer>, Integer, List<Integer>>` that takes a list and a threshold, returning a new list with all elements greater than the threshold multiplied by 2.
53. Implement a `UnaryOperator<String>` that capitalizes the first letter of each word in a sentence, then use it in a stream operation.
54. Use `BinaryOperator<BigDecimal>` to create a precise calculator for financial calculations, implementing add, multiply, and percentage operations while maintaining proper precision.

Complex Combination Questions (6 questions)

55. Given a list of `Order` objects, filter orders from the last 30 days, group them by customer, map to total order values, and collect the top 10 customers by spending.
56. Process a list of log entries: filter by error level, map to extract timestamp and message, sort by timestamp, and collect into a formatted report string.
57. From a list of `Student` objects, find students with GPA above 3.5, group by major, calculate average GPA per major, and sort majors by average GPA descending.
58. Take a list of file paths, filter for image files, map to File objects, check if they exist and are readable, collect existing files, and sort by file size.
59. Process a list of `SalesRecord` objects: filter by date range and product category, map to revenue values, reduce to total revenue, and format the result as currency.
60. Given a nested structure of `Company` → `Department` → `Employee`, flatten to all employees, filter by salary range and years of experience, group by department, and create a summary report showing department names with employee counts and average salaries.

Summary

This tutorial covered:

- **Anonymous Inner Classes**: Traditional approach and their limitations
- **Functional Interfaces**: The foundation for lambda expressions
- **Lambda Expressions**: Concise syntax for functional programming
- **Method References**: Even more concise syntax for simple operations
- **Stream API**: Functional approach to collection processing
- **Built-in Functional Interfaces**: Ready-to-use functional types
- **60 Practice Questions**: Comprehensive coverage of all topics

Next Steps:

1. Practice writing lambda expressions for common scenarios
2. Experiment with method references in different contexts
3. Master the Stream API for collection processing
4. Learn advanced topics like parallel streams and custom collectors
5. Apply functional programming principles in real projects

Key Takeaways:

- Lambda expressions make code more readable and concise

- Method references provide the most concise syntax when possible
- Stream API enables functional programming with collections
- Built-in functional interfaces cover most common use cases
- Combining these features leads to more expressive and maintainable code