Java 8 Lambda Level 1

5. Java 8 - Lambda Expressions

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example,neccessary hinds, points to remember,usual mistakes,etc.

start lvl1

Perfect! Let's start **Level 1** of Java 8 **Lambda Expressions**. We'll grow step by step, like a game. 🕹️

Level 1: Lambda Basics

Concept

Lambda expressions provide a **clear and concise way to represent a single method interface (functional interface)** using an expression rather than a full class or anonymous class.
They are introduced to **simplify functional programming** in Java.

Functional Interface: An interface with **only one abstract method**.
Examples: `Runnable`, `Comparator`, `ActionListener`.

Syntax

```
java

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

Examples:

1️⃣ No parameters, no return value (like Runnable):

```
java

Runnable r1 = () -> System.out.println("Hello from Lambda!");
r1.run();
```

Explanation:

- `Runnable` has a single abstract method `run()`.
- Lambda `() -> System.out.println("Hello from Lambda!")` implements it.
- Shorter than writing an anonymous inner class.

2️⃣ With parameters and return value:

```
java

Comparator<Integer> comp = (a, b) -> a - b;

System.out.println(comp.compare(10, 20)); // Output: -10
```

Explanation:

- `Comparator` interface has one method `compare(T o1, T o2)`
- `(a, b) -> a - b` implements `compare()` in a concise way.
- Returns a negative number if `a < b`, zero if equal, positive if `a > b`.

Real-World Use Case

1. Sorting a list of objects:

```
java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
System.out.println(names); // [Alice, Bob, Charlie]
```

2. Using threads more cleanly:

```
java

new Thread(() -> System.out.println("Thread running")).start();
```

Points to Remember

- Lambda can only be used with **functional interfaces**.
- Parentheses can be omitted if there is **only one parameter**:
  `(x) -> x * x` → `x -> x * x`
- Braces `{}` are optional if there is a **single statement**.
- Return keyword is needed if using **multiple statements in braces**.

Usual Mistakes

- Trying to use lambda with interfaces **having more than one abstract method**.
- Misplacing parentheses for multiple parameters.
- Forgetting to include `return` when using braces with multiple statements.

✅ Mini Challenge (Level 1)

Convert this anonymous inner class to a lambda:

```java
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running");
    }
};
r.run();
```

If you want, I can take you **Level 2**, where we explore **lambdas with collections and streams** for more practical power.

Do you want to move to **Level 2**?

next

Awesome! Let's level up. 🎮

Level 2: Lambda with Collections

In **real-world Java**, lambdas shine when used with **Collections** because they reduce boilerplate code for iteration, filtering, and processing.

1️⃣ Iterating a List

Before Java 8:

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
for(String name : names) {
    System.out.println(name);
}
```

With Lambda (Java 8):

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));
```

Explanation:

- `forEach` is a default method in `Iterable` interface.
- `name -> System.out.println(name)` is the lambda expression executed for each element.

2️⃣ Filtering a List

```
java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(System.out::println);
```

Explanation:

- `stream()` converts collection to a Stream.
- `filter()` takes a lambda to select elements.
- `System.out::println` is a **method reference** shorthand for `x -> System.out.println(x)`.

Output:

```
nginx

Alice
```

3️⃣ Transforming a List

```
java

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> upperNames = names.stream()
                               .map(name -> name.toUpperCase())
                               .toList();
System.out.println(upperNames);
```

Output:

```
csharp

[ALICE, BOB, CHARLIE]
```

Explanation:

- `map()` transforms each element using the lambda.
- `toList()` collects the result back into a list.

Real-World Use Case

- **Filtering users based on criteria** (e.g., age > 18)
- **Transforming data** (e.g., names to uppercase before saving in DB)
- **Iterating over collections without boilerplate loops**

Points to Remember

- Lambdas + Streams = **powerful combination** for functional-style programming.
- Use **method references** when possible (`ClassName::methodName`) for brevity.
- Streams **don't modify the original collection**; they return a new stream or result.

Usual Mistakes

- Forgetting to call `collect()` or `toList()` after a stream if you want a list.
- Using **mutating operations inside streams**, which is not recommended.
- Confusing `map()` (transform) vs `filter()` (select).

✅ Mini Challenge (Level 2)

Given this list of integers, use a **lambda and stream** to print only the even numbers:

```
java

List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
```

If you want, we can move to **Level 3**, where we explore **multi-parameter lambdas, return values, and functional interfaces like Predicate, Function, and Consumer**.

Do you want to move to **Level 3**?

next

Perfect! Let's advance to **Level 3**. 🎮

Level 3: Functional Interfaces with Lambda

In Java 8, **functional interfaces** from `java.util.function` package make lambdas **reusable and powerful**.

1️⃣ Consumer Interface

Use case: Perform an action on a single input.

```
java

import java.util.function.Consumer;
import java.util.Arrays;
import java.util.List;

public class LambdaConsumerExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
```

```
        Consumer<String> printer = name -> System.out.println("Hello, " + name);
        names.forEach(printer);
    }
}
```

Explanation:

- `Consumer<T>` has `accept(T t)` method.
- Lambda `name -> System.out.println("Hello, " + name)` implements it.
- Used with `forEach` for clean iteration.

2️⃣ Predicate Interface

Use case: Test a condition (returns boolean).

```
java


import java.util.function.Predicate;
import java.util.Arrays;
import java.util.List;

public class LambdaPredicateExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 12, 18, 7, 20);

        Predicate<Integer> isEven = n -> n % 2 == 0;
        numbers.stream()
               .filter(isEven)
               .forEach(System.out::println);
    }
}
```

Output:

```
12
18
20
```

Explanation:

- `Predicate<T>` has `test(T t)` method.
- `filter()` takes a Predicate to decide which elements to keep.

3️⃣ Function Interface

Use case: Transform data (input → output).

```
java


import java.util.function.Function;
import java.util.Arrays;
import java.util.List;

public class LambdaFunctionExample {
    public static void main(String[] args) {
```

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

        Function<String, Integer> lengthFunc = name -> name.length();
        names.stream()
             .map(lengthFunc)
             .forEach(System.out::println);
    }
}
```

Output:

```
5
3
7
```

Explanation:

- `Function<T, R>` has `apply(T t)` method.
- Transforms each element from one type to another.

Real-World Use Case

1. **Consumer:** Logging or sending notifications for each user.
2. **Predicate:** Filtering database query results in memory.
3. **Function:** Mapping DTO objects to entities or formatting strings.

Points to Remember

- Use **Consumer** for **actions without return value**.
- Use **Predicate** for **boolean checks**.
- Use **Function** for **mapping/transforming data**.
- Combine functional interfaces with **Streams** for powerful pipelines.

Usual Mistakes

- Using lambda without matching **functional interface type**.
- Trying to mutate external variables in lambda (they must be **effectively final**).
- Forgetting `.apply()` for `Function` or `.test()` for `Predicate` if used standalone.

✅ Mini Challenge (Level 3)

Given a list of integers, use **Predicate** and **Consumer** to **print only numbers greater than 10**:

```
java

List<Integer> numbers = Arrays.asList(5, 12, 18, 7, 20);
```

If you want, we can move to **Level 4**, where we explore **multi-parameter lambdas, chaining, and advanced stream operations like reduce, sorted, and distinct**.

Do you want to move to **Level 4**?