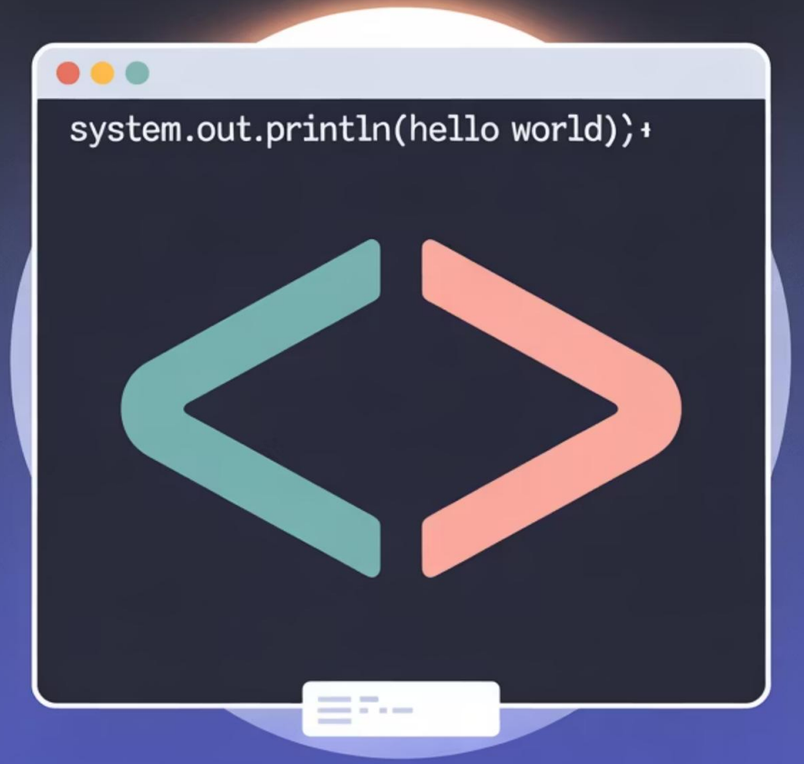
Java 8 Lambdas

- Lambda Expressions
- Basic Syntax
- Practical applications - examples
- Functional interfaces that support lambdas



Introduction to Lambda Expressions

Lambda expressions, introduced in Java 8, represent a significant paradigm shift, bringing functional programming concepts directly into the language.

They offer a concise way to represent anonymous functions, drastically reducing boilerplate code compared to traditional anonymous inner classes.

1

Java 8 Innovation

A cornerstone feature of Java 8, enhancing developer productivity.

2

Anonymous Functions

Enables treating functionality as a method argument or code as data.

3

Concise Code

Reduces verbosity, especially for single-method interfaces.

4

Functional Interfaces

Works seamlessly with interfaces that have exactly one abstract method.

Understanding Lambda Syntax

Lambda expressions follow a straightforward syntax that makes code more readable and maintainable.

They consist of three main parts: parameters, the arrow token (`->`) and the body.

Basic Syntax Forms:

- `(parameters) -> expression`

For single-line expressions, the return keyword is implicit.

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

For multi-line code blocks, requiring explicit `return` for methods that return a value.

Examples:

- `() -> System.out.println("Hello!");`

(No parameters, simple action)

- `name -> System.out.println("Hello " + name);`

(Single parameter, no parentheses needed)

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

(Multiple parameters, implicit return)

- `(int x, int y) -> { return x * y; }`

(Type declaration optional, explicit return)

Lambda with Runnable

One of the most common and illustrative uses of lambda expressions is with the `Runnable` interface, which is a functional interface.

This allows for incredibly concise thread creation.

Before Lambdas:

```
Runnable task1 = new Runnable() {
public void run() {
 System.out.println("Task 1 running...");
 }
};
```

This traditional approach requires an anonymous inner class, leading to more boilerplate code.

With Lambdas:

```
Runnable task2 = () -> System.out.println("Task 2 running...");
```

A single line of code expresses the same functionality, enhancing clarity and reducing visual clutter.

Executing these tasks is straightforward: `new Thread(task1).start();` and *new Thread(task2).start()*;

Lambda with Comparator

Sorting collections is another prime example where lambda expressions simplify code dramatically, particularly when implementing custom comparison logic using the `Comparator` functional interface.

```
List names = Arrays.asList(“Sita”, “Gita”, “Rita”);
```

Without Lambdas:

```
Collections.sort(names, new Comparator() {
public int compare(String a, String b) {
      return b.compareTo(a);
}
}); // descending list of names
```

The traditional approach requires instantiating an anonymous inner class, making the sorting logic less direct.

With Lambdas:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

The lambda expression directly specifies the comparison logic, leading to more readable and maintainable code.

The Backbone of Lambdas

Functional Interfaces

- Functional interfaces are at the core of lambda expressions.
- A functional interface is precisely an interface that contains a single abstract method (SAM)
- This unique abstract method serves as the specific target for any corresponding lambda expression.

Single Abstract Method (SAM)

- The defining characteristic of a functional interface.
- It can include multiple default or static methods, but only one abstract method is permitted.

@FunctionalInterface Annotation

- This annotation is highly recommended.
- It enforces the SAM rule at compile-time, preventing the accidental introduction of additional abstract methods.

Lambda Compatibility

Lambda expressions serve as a concise, inline implementation of the singular abstract method defined within a functional interface.

The Backbone of Lambdas

Common Functional Interfaces

- Java 8 introduced a new package, `java.util.function`, which provides a rich set of predefined functional interfaces.
- These commonly used interfaces cover a wide range of use cases, making it easier to integrate functional programming paradigms into your code.

Predicate`(T -> boolean)`

- Represents a boolean-valued function of one argument.
- Ideal for filtering or testing conditions.
- Its abstract method is **`boolean test(T t)`**

Function`(T -> R)`

- A function that accepts one argument and produces a result.
- Useful for transformations.
- Its abstract method is **`R apply(T t)`**

Consumer`(T -> void)`

- Represents an operation that accepts a single input argument and returns no result.
- Used for side effects.
- Its abstract method is **`void accept(T t)`**

Supplier`(() -> T)`

- Represents a supplier of results.
- Useful when to generate or retrieve a value without any input.
- Its abstract method is **`T get()`**

Functional Interface Examples

Let's see how these common functional interfaces can be implemented concisely using lambda expressions, enhancing the readability and expressiveness of your code.

Predicate:

```
Predicate isEven = x -> x % 2 == 0;
// Example usage:
isEven.test(4); // would return true
```

Checks if a number is even.

Function:

```
Function length = str -> str.length();
// Example usage:
length.apply("Java");// would return 4
```

Calculates the length of a string.

Consumer:

```
Consumer printer = s -> System.out.println(s);
// Example usage:
printer.accept("Hello Lambda!");
```

Prints a given string to the console.

Supplier:

```
Supplier random = () -> Math.random();
// Example usage:
random.get(); //would return a random double
```

Provides a random double value.

Conclusion: Embrace the Evolution

- Java 8's Lambda Expressions are more than just *syntactic sugar*, they are a fundamental shift that empowers developers to write more *concise, readable, and powerful code*.
- Their seamless integration with the *Stream API, in particular, enables highly expressive and efficient* declarative programming for data processing.
- By understanding lambdas and their underlying functional interfaces, we can unlock new possibilities for building modern, efficient Java applications.

> "Lambdas simplify code that uses functional Interfaces, improving the readability and flexibility of Java programs."


Thank You!