

Lambda Expressions Basics

Basics of Lambda Expressions

What is a Lambda Expression?

- Is a method without a name used to pass around functionality as if it were data
- Compact way of passing around behaviour
- Lambda Expressions provide the ability to
 - Assign functions to variables
 - Pass functions as arguments to other functions
 - Return functions from other functions
 - Have anonymous functions

Need for Lambda Expression

- **Enables Functional programming**
- **Less verbose and easier-to-read code thus easier to maintain**
- **Eliminates boiler plate code**
- **Provide support for parallel processing**

Using Anonymous inner class

➤ Consider the below interface Calculation

```
public interface Calculation {  
    double compute(double one, double two);  
}
```

➤ Implementation using Anonymous Inner Class

```
Calculation multiply = new Calculation() {  
    public double compute(double one, double two) {  
        return one * two;  
    }  
};  
  
double result = multiply.compute(3, 6);
```

— Above code is verbose and not easy to read

First Lambda Expression

➤ Shorter syntax using Lambda Expression

```
Calculation multiply = (double one, double two) -> {return one * two;};  
double result = multiply.compute(3,6);
```

- Above Lambda expression creates and assigns a function without a name

```
(parameters) -> {lambda expression body};
```

➤ A lambda expression consists of the following

- A comma-separated list of formal parameters enclosed in parentheses
- The arrow token ->
- A body, which consists of a single expression or a statement block.

Syntax of Lambda Expression

➤ No Parameters

- empty parentheses() signifies no parameters

```
() -> {System.out.println("Hello World");};
```

- Braces{} can be omitted for single statement in body

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

Syntax of Lambda Expression

➤ Single Parameter

```
(String a) -> {a = a.trim();  
              return a.length();  
              };
```

— Parameter data type can be omitted. Parentheses can be omitted for single parameter

```
a -> {a = a.trim;  
      return a.length();  
      };
```

```
a -> a.trim().length();
```

ted

Syntax of Lambda Expression

➤ More than one Parameter (Finding difference between two integers)

```
(int a, int b) -> { if(a > b){  
                    return a - b;  
                } else {  
                    return b - a;  
                }  
};
```

➤ Parameter datatypes can be omitted.

➤ Braces can be omitted for single statement in body

```
(a, b) -> (a > b) ? a - b : b - a;
```

Functional Interface

Functional Interface

- Each Lambda expression represents a single function
 - Can be assigned to a variable or passed as value to a function. **What should be the variable Type?**
- Functional Interface
 - Is an interface with exactly one abstract method declaration
 - Can have static and default methods

```
@FunctionalInterface
public interface Calculation<T> {
    T compute(T one, T two);

    default double sqrt(int num) {
        return Math.sqrt(num);
    }
}
```

Functional Interface Implementation

```
Calculation<Integer> addition = (x, y) -> x + y;  
Integer result = addition.compute(10, 5);  
System.out.println(result);
```

```
@FunctionalInterface  
public interface Calculation<T> {  
    T compute(T one, T  
    two);  
}
```

➤ Type inference

- Compiler matches and infers the parameter/return types of lambda expression with the types defined in the method of functional Interface

➤ @FunctionalInterface annotation

- Avoids accidental addition of more than one abstract method in the functional interfaces
- Optional but good to have annotation for using an interface as a type for Lambda expression

Functional Interfaces in Java 8

- In Java 8, a new package **java.util.function** is added with many built-in functional interfaces
- These interface provide target types for lambda expressions

Interface	Description
Consumer<T>	<ul style="list-style-type: none">• Method : void accept(T t)• Represents an operation that accepts a single input argument and returns no result
Supplier<T>	<ul style="list-style-type: none">• Method : T get()• Represents a supplier of results
Function<T,R>	<ul style="list-style-type: none">• Method : R apply(T t)• Represents a function that accepts one argument and produces a result
Predicate<T>	<ul style="list-style-type: none">• Method : boolean test(T t)• Evaluates this predicate on the given argument

Using Functional interfaces

```
public class FunctionalInterfaceDemo {  
    public static int getSum(int[] arr, Predicate<Integer> condition) {  
        int sum = 0;  
        for (int num : arr) {  
            if (condition.test(num)) {  
                sum += num;  
            }  
        }  
        return sum;  
    }  
  
    public static void main(String[] args) {  
        int[] intArr = {1,2,3,4,5,6};  
        int evenSum = getSum(intArr, x -> x % 2 == 0);  
        System.out.println("Sum of even numbers " + evenSum);  
        System.out.println("Sum of Odd numbers " + getSum(intArr, x -> x%2!=0));  
    }  
}
```

Passing inline function
as value

Interfaces in Java 7

- Many interfaces before Java 8, fit the functional interface definition
- Some have been made as strict Functional interfaces by adding `@FunctionalInterface`
 - Ex – Runnable, Comparator
- Lambda expression implementing Runnable
 - Runnable has only one method 'run', takes no arguments and is a void return type

```
Runnable task = () -> System.out.println("Hello World");  
Thread t = new Thread(task);  
t.start();
```

Sorting List using Lambda expression

- Given below is the signature of the Collections.sort method

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

Sorts the specified list according to the order induced by the specified comparator

- Comparator implementation using Lambda expression

```
List<String> words = Arrays.asList("can", "man", "pan");  
  
Collections.sort(words, (String s1, String s2) -> s2.compareTo(s1));  
  
for(String word : words){  
    System.out.println(word);  
}
```


Accessing variables in Lambdas

- Lambda expression can
 - access instance variable and static variables of the enclosing class
 - Local variables that are declared final or are effectively final

```
public class TestScopes {  
    int incr1;  
    static int incr2;  
  
    void displayIncremented(int num) {  
        final int incr3 = 10;  
        Function<Integer,Integer> increment = x -> x + incr1 + incr2 + incr3;  
        System.out.println(increment.apply(num));  
    }  
}
```

- incr3 should be final or effectively final