What is Lambda :

Lambda expressions are a powerful feature introduced in Java 8 that allows you to write more concise and expressive code. They provide a way to represent functional interfaces (interfaces with a single abstract method) as instances of functional interfaces. Lambda expressions are often used with functional interfaces to implement functional programming concepts in Java.

Here is an in-depth explanation of Java lambda expressions:

1. Syntax: Lambda expressions have the following syntax:

```
(parameters) -> expression/statement block
```

2. Functional Interfaces: Lambda expressions are typically used with functional interfaces. Functional interfaces are interfaces that have only one abstract method and can be implicitly converted to lambda expressions. Examples of functional interfaces in Java include `Runnable`, `Comparator`, `Consumer`, and `Predicate`.
3. Parameter List: The parameter list of a lambda expression can be empty or can contain one or more parameters. If there is only one parameter, you can omit the parentheses. For example:

```
() -> System.out.println("Hello")
x -> x * x
(a, b) -> a + b
```

4. Arrow Token: The arrow token (`->`) separates the parameter list from the body of the lambda expression.
5. Body: The body of a lambda expression can be an expression or a statement block. If the body is an expression, it is evaluated and returned. If the body is a statement block, it can contain multiple statements enclosed in curly braces. For example:

```
() -> 42
x -> {
    int result = x * x;
    System.out.println(result);
}
```

6. Type Inference: In many cases, the types of the lambda expression parameters can be inferred by the compiler based on the context in which the lambda expression is used. This allows you to write more concise code by omitting the type declarations. For example:

```
(String s) -> s.length()
(x, y) -> x + y
```

7. Method References: Lambda expressions can also be used to reference existing methods. Instead of providing a lambda body, you can reference a method directly. Method references provide a way to make the code more readable and concise. For example:

```
System.out::println
Math::max
String::toLowerCase
```

8. Capturing Variables: Lambda expressions can access variables from their enclosing scope. These variables are effectively final or should be effectively final (i.e., their values should not change after they are captured by the lambda expression). For example:

```
int x = 42;
() -> System.out.println(x)
```

9. Benefits: Lambda expressions provide several benefits, including:
   - Conciseness: Lambda expressions allow you to write more compact code.
   - Readability: They make the code more readable by focusing on the functionality rather than the boilerplate code.
   - Flexibility: Lambda expressions provide flexibility in terms of passing behavior as an argument.

Lambda expressions are a key feature of Java 8 and have significantly improved the expressiveness and flexibility of the language. They are widely used in modern Java programming, especially when working with streams, functional interfaces, and functional programming paradigms.

Example in class :

```
package lambda;

@FunctionalInterface
public interface Operation {
    int someOperation(int i, int j);
}
```

```java
package lambda;

public class OperationAdd implements Operation{
    @Override
    public int someOperation(int i, int j) {
        return i+j;
    }
}

package lambda;

import java.util.function.BiConsumer;
import java.util.function.LongPredicate;
import java.util.function.Supplier;

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

        BiConsumer<String, String> biConsumer
                = (a, b) -> System.out.println(a+" "+b);
        biConsumer.accept("John", "Doe");

        Operation oAdd1 = new OperationAdd();
        System.out.println(oAdd1.someOperation(1,2));

        Operation oAdd2 = new Operation() {
            @Override
            public int someOperation(int i, int j) {
                return i+j;
            }
        };
        System.out.println(oAdd2.someOperation(1,2));

        Operation oAdd3 = (i, j) -> {
            return i + j;
        };
        System.out.println(oAdd3.someOperation(1,2));

        Operation oAdd4 = Integer::sum;
        System.out.println(oAdd4.someOperation(1,2));

        Supplier<Integer> mySupplier = () -> 10;
        System.out.println(mySupplier.get());

        LongPredicate isValueGreaterThanTenPredicate = (longValue) ->
                        longValue > 10l;
        System.out.println(isValueGreaterThanTenPredicate.test(9l));

        //-----------------------------------//
        performOperation(Integer::sum, 1, 2);
        performOperation(LambdaTest::add, 1, 2);
        performOperation(oAdd1::someOperation, 1, 2);
        performOperation(LambdaTest::multiply, 1, 2);
        performOperation((i, j) -> i/j, 1, 2);
        performOperation((i, j) -> i-j, 1, 2);

    }
    //ClassName::methodName

    public static void performOperation(Operation o, int a, int b){
        int output = o.someOperation(a, b);
        System.out.println("Output is : "+output);
    }

    public static int add(int m, int n){
        return m+n;
    }

    public static int multiply(int m, int n){
        return m*n;
    }

}
```