Lambda Expression

- Lambda Expression is an anonymous function. That means the function which does not have the name, return types, access modifiers.
- Also does not need to define the data type of input parameters.
- Lambda expression in java implements the functional interface and it can be treated as any other java object.
- It can be used to create threads, comparators and can be used to add event Listeners.
- E.g.,

```
@FunctionalInterface
public interface AFunctionalInterface
{
	public void m1();
}

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

		// provide implementation with lambda
		AFunctionalInterface fl = () ->
		{
			System.out.println("Calling m1 method");
		};

		// function call.
		fl.m1();
	}
}
```

- In above example,
  `() -> {System.out.println("Calling m1 method"); };` is the lambda expression in the above code.
- Lambda expression adds functional programming techniques to Java.

Java Lambda Expression Syntax

Lambda expressions consist of three components as below

1. Argument List
2. Lambda Operator (->)
3. Body

1. Argument List
    - It is the first portion of the Lambda expression in Java.
    - It can be empty or non-empty.
    - It is enclosed by a round bracket.
2. Lambda Operator (->)
    - It's an arrow sign that appears after the list of arguments.
    - It connects the arguments-list with the body of the expression.
3. Body
    - It contains the function body of lambda expression.
    - It is surrounded by curly brackets.

```
//Syntax of the lambda expression
(parameter_list) -> {function_body};
```

OR

A. No Parameter Syntax

```
( ) -> { function_body };
```

a. It is the case when the function does not require any input parameter.

B. One parameter Syntax

```
( p1 ) -> { function_body };
```

a. need to pass one input parameter.
b. while using lambda, we don't need to specify the data type of input parameters because the compiler discovers it automatically.

C. Two Parameter Syntax (multiple parameter syntax)

```
( p1, p2 ) -> { function_body };
```

a. Need to pass parameters.
b. It is separated by **,** .

Java Lambda expression Example

1. Without Using Lambda Expression

```
// without Lambda expression

AFunctionalInterface Afl = new AFunctionalInterface()
{
        @Override
        public void m1() {
                System.out.println("It's a lengthy code");
        }
};

Afl.m1();
```

2. Using Lambda Expression

```
// Using Lambda Expression

AFunctionalInterface fl = () -> {
      System.out.println("Calling m1 method...");
};

fl.m1();
```

3. Lambda Expression with no parameter

```
// Using Lambda Expression

AFunctionalInterface fl = () -> {
      System.out.println("Calling m1 method...");
};

fl.m1();
```

4. Lambda Expression with one parameter

```
// Using Lambda Expression

AFunctionalInterface fl = (i) -> {
      System.out.println("Print i " + i);
};

fl.m1(10);
```

5. Lambda Expression with multiple parameters

```
AFunctionalInterface fl = (fName, lName) -> {
      System.out.println("Hello " + fName + "Hello " + lName);
};

fl.m1("Rasika", "Swapnil");
```

6. Iterating Collections Using the Foreach Loop

```
// Iterating Collections Using the Foreach Loop
List<String> l = new ArrayList<String>();

	l.add("A");
	l.add("B");
	l.add("C");
	l.add("D");
	l.add("E");
	l.add("F");

	l.forEach((element) -> {
		System.out.print(element + " ");
	});
```

7. Lambda Expression With Return Statement

```
FunInt3 f4 = (a,b) -> {
	return (a+b);
};
System.out.println(f4.add(10, 20));
```

8. Lambda Expression - Creating Thread

```
Runnable r = () -> {
	System.out.println("New Thread is running");
};
Thread t = new Thread(r);
t.start();
```

9. Lambda Expression - Comparator
   - When we wish to sort a collection of items that can be compared, we use a comparator.
   - You must only use Comparator if you wish to sort this collection based on several criteria/fields.
   - e.g.,

```
List<Person> personList = new ArrayList<Person>();
personList.add(new Person(2, "Kaju", "Biju"));
personList.add(new Person(1, "Gaju", "Biju"));
personList.add(new Person(0, "", "Biju"));

Collections.sort(personList, (p1, p2) -> {
	return p1.fName.compareTo(p2.fName);
});

personList.forEach((person) -> {
System.out.println("Person " + person);
});
```

10. Lambda Expression - Filter Collection Data

```
List<Person> personList = new ArrayList<Person>();
personList.add(new Person(2, "Kaju", "Biju"));
personList.add(new Person(1, "Gaju", "Biju"));
personList.add(new Person(0, "", "Biju"));

Stream<Person> stP = personList.stream().filter((p) -> p.age >= 1);

stP.forEach((person) -> {
	System.out.println(person);
});
```

11. Lambda Expression - Event Listener

```
JButton button = new JButton("Click Me!");
button.addActionListener(e ->
	System.out.println("Handled by Lambda listener")
);
```

Lambdas as Objects

- A lambda expression can be assigned to a variable and passed around like any other object.
- e.g.,

```
@FunctionalInterface
public interface MyComprator
{
	public boolean compare(int a, int b);
}

MyComprator myComp = (i,j) -> i > j;
boolean result = myComp.compare(10, 9);
System.out.println(result);
```

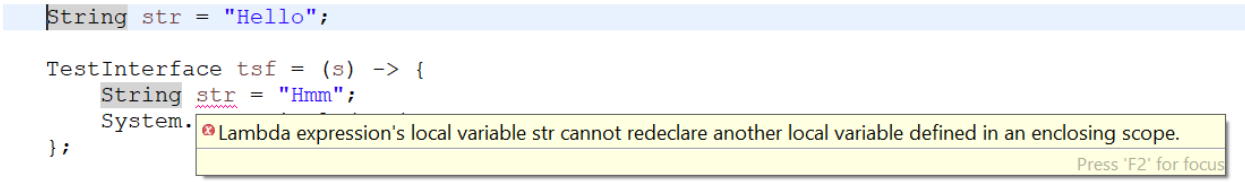
- assign the return value of Lambda expression to boolean variable result, which shows that Lambda can be treated as a java object.

Accessible Variable

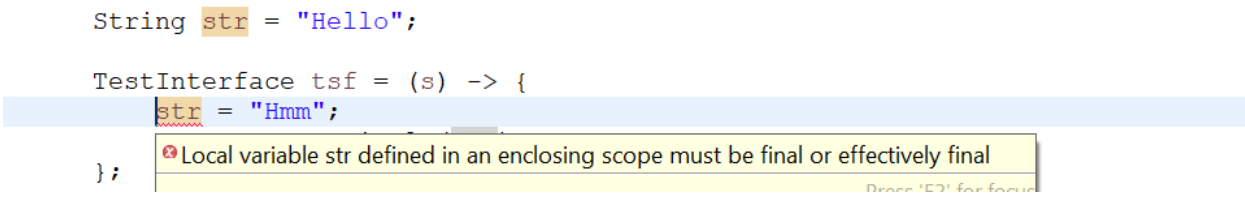
- Java lambdas can access the following types of variables
    1. Local Variables
    2. Instance Variables
    3. Static Variables

1. Local Variables

    - Can access to local variables of the enclosing scope.
    - But we need to **follow some rules** in the case of local variables.
    - **can't declare a local variable with the same name** that is already declared in the enclosing scope of a lambda expression.
      Since a lambda expression can't define any new scope as an anonymous inner class does.



    - **can't assign any value to a local variable declared outside** the lambda expression inside the lambda expression.
      Because local variables declared outside of a lambda expression may be final or effectively final.



    - **this() and super() references inside a lambda expression body are the same as their enclosing scope.**
      Because lambda expressions can't define any new scope.

2. Instance Variables
    - the value will be altered inside the lambda.

```
public class MyInstanceVariable {
    int i;
    public void show() {
        AFunctionalInterface a1 = () -> {
            i =10;
            System.out.println(i);
        };
        a1.m1();
    }
}
```

3. Static Variables
    - Static variables can also be accessed via a Java lambda expression.

```
    static String s1 = "String is";

    public void display() {
        TestInterface t1 = (s1) -> {
            System.out.println(s1 + " immutable");
        };

        t1.testLocalVariable(s1);
    }
```