A lambda expression is said to be capturing if it either accesses instance variables of it's enclosing class or local variables (final or effectively final) from it's enclosing scope.

A lambda expression can capture the three types of variables given below:

- Static variables
- Instance variables
- Local variables

If a lambda expression captures a local variable then the variable should be either **final** or **effectively final**.

## What is effectively final?

**Effectively final** is a new concept that was introduced in Java 8. A non-final, local variable whose value is never changed after initialization is known as **effectively final**.

Before Java 8, we cannot use a non-final, local variable in an anonymous class. If you need to access a local variable in an anonymous class, then it should be declared as **final**. This restriction is relaxed in Java 8. Now, the compiler, itself can check if the value of a variable is not changed after the assignment. Then, it is effectively final.

Let's look at an example of a capturing lambda. In the below example, our lambda is capturing a local variable called `i`. The value of this variable is initialized once and never changed, so it is **effectively final**.

```
import java.util.function.UnaryOperator;

public class CapturingLambdaDemo {

    public static void main(String args[]){

        int i = 5;

        UnaryOperator<Integer> operator = (input) -> input * i;

        System.out.println(operator.apply(i));

    }
}
```

Run Save Reset

The below code will not compile because we have modified the value of the local variable and it is not final anymore.

```
import java.util.function.UnaryOperator;

public class CapturingLambdaDemo {

    public static void main(String args[]){

        int i = 5;

        i = 7; // Since we have changed the value of i, the below line will not compile.

        UnaryOperator<Integer> operator = (input) -> input * i;

        System.out.println(operator.apply(i));

    }
}
```

Run Save Reset

The below code will compile because the variable is reassigned, but it is not a local variable.

```
import java.util.function.UnaryOperator;

public class CapturingLambdaDemo {

    static int i = 0;

    public static void main(String args[]){

        i = 7;

        UnaryOperator<Integer> operator = (input) -> input * i;

        System.out.println(operator.apply(i));

    }
}
```

Run Save Reset

## Why should local variables be final or effectively final?

Now, let's discuss why the local variable should be *final* or *effectively final* if it is used in a lambda expression.

When a local variable is used in a lambda expression, the lambda makes a copy of that variable. This occurs because the scope of a lambda expression is only until the method is in the stack. If the lambda does not make a copy of the variable, then the variable is lost after the method is removed from the stack.

Now, if the variable is not final or effectively final, it is possible that the value of the variable is changed after using it in the lambda as shown below.

```
import java.util.function.Function;

public class CapturingLambdaDemo {

    public static void main(String args[]){

        Function<Integer, Integer> multiplier = getMultiplier();

        System.out.println(multiplier.apply(10));
    }

    public static Function<Integer,Integer> getMultiplier(){

        int i = 5;
        // The below lambda has copied the value of i.
        Function<Integer, Integer> multiplier = t -> t * i;
        // If you change the value of i here, then the lambda will have old value.
        // So this is not allowed and code will not compile.
        i = 7;
        return multiplier;

    }
}
```

Run Save Reset

I hope that after reading this lesson you have a clear idea of what effectively final variables are and why local variables in lambdas should be effectively final.