**Java Lambda Expressions**

Lambda expressions, sometimes referred to as an anonymous method were first introduced in the update of Java 8. They are still being used in the most recent Java 16. It was implemented as a way of adding functional programming to Java which is an Object Oriented Language.

A Lambda expression is in essence a shorthand way of expressing functional interface instances. A functional interface is an interface that has only one abstract method. Some of the most common ways a lambda expression can be beneficial is they can allow functionality as a method argument, a function can be created without belonging to a class and it can be passed around like an object and executed on demand (geeksforgeeks.com) These expressions are most commonly used for things such as event listeners, runnable or comparable.

In order for a lambda expression to be valid it must match the interface in three ways. The first being that the interface may only have one method. The second is the parameters of the lambda expression must match those of the method in the interface. Lastly, if there is a return type, does the method return type match that of the lambda expression. If all of these are true, then there is a successful match between the interface and lambda.

Lambda expressions are closely related to Anonymous interfaces but there is a major difference in that an anonymous interface can have state member variables and a lambda expression cannot making it known as stateless.

Lambda expression syntax is as follows:

(parameters) -> (code block)

What you notice is that the parameters are to the left of the lambda arrow and are often surrounded by parenthesis, although in some cases they are not required such as with only one parameter. To the right of the arrow is the body of the expression which contains the statements or code block.

Lambda expressions in many ways are similar to methods which means that they too can take in parameters. They do require as mentioned earlier that they match the functional interface method parameters. Lambda expressions may take zero, one or multiple parameters just like a regular method. The types of parameters that can be passed in are also just like methods and can include Objects. It is considered to be best practice to always specify the types of parameters to make sure the Java compiler has no issues in recognizing the parameters coming from the functional interface method. Starting in Java 11 one can use the var keyword that was introduced in Java 10 as a parameter keyword. This can help with local variable inference is some cases, but it is still considered best practice to declare data types as parameters.

Lambda expressions also allow you to return values just like a method and you add the return statement in the same way. You must return the same value as declared in the functional interface method. This is to prevent runtime compiling issues.

One of the things that makes lambda expressions very versatile is that they can be passed around due to the fact they are also objects. This also means that they can be stored as variables as well.

When using a lambda expression, it is also possible to access certain variable and to use them in certain conditions. The three main variables are local variables, instance variables and static variables. Local variables can be captured outside the body of the expression as long as the value never changes. If later on the variable is changed, the compiler will throw an error. With instance variables we use the ‘this’ keyword to access it within the expression. Unlike a local variable that effectively needs to be final, our instance variable can be changed, and it will take in that change. Due to the fact we use the ‘this’ keyword we have to recognize that we are actually referencing the enclosing object and not the lambda expression itself. Static variables are also able to be accessed within a lambda expression. They are accessed in the same way as a regular method and they too like instance variables can be changed after an expression has captured the variable.

Overall, Lambda expressions can be a great way to reduce code bloating and increase readability. It also helps to eliminate some of the drawbacks to anonymous inner classes.

References:

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

<http://tutorials.jenkov.com/java/lambda-expressions.html>

<https://www.geeksforgeeks.org/lambda-expressions-java-8/>