Lambda Expressions

**What is a Lambda Expression?**

A lambda expression is an anonymous function. It comprises of a set of parameters, a lambda operator (->) and a function body. See it as a method without a declaration. (no access modifier, return value declaration, and name).

*For example :-*

(String s) -> { System.out.println(s); }; 🡪 Given a string s prints the string to the main output and returns void.

n -> n % 2 != 0; 🡪 Given a number n returns a boolean indicating if it is odd.

**The Parameters**

* A lambda expression can receive zero, one or more parameters.
* The type of the parameters can be explicitly declared or it can be inferred from the context.
* Parameters are enclosed in parentheses and separated by commas.
* Empty parentheses are used to represent an empty set of parameters.
* When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

**The Body**

* The body of the lambda expression can contain zero, one or more statements.
* When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there is more than one statement then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

**What is the Type of Lambda Expression?**

The type of the lambda expression would be a *function;* but in Java, the lambda expressions are represented as objects, and so they must be bound to a particular object type known as a functional interface. This is called the ***target type***.

Notice that the type of the lambda expression is determined by the compiler from the context based on the target type. This implies that two apparently equal lambda expressions may have different types simply because they are bound to a different target type as demonstrated in the following code.

Callable<String> callMe = () -> "Hello";

PrivilegedAction<String> action = () -> "Hello"

**Lambdas as Objects**

A Java lambda expression is essentially an object. You can assign a lambda expression to a variable and pass it around, like you do with any other object. Here is an example:

public interface MyComparator {

public boolean compare(int a1, int a2);

}

MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);

**Using Filters, Maps, Streams and forEach to apply Lambdas to Java Collections!**

***forEach* : -** The Java 8 forEach operation is available on any list object and it accepts a function to work on every element of the list.

***Streams* : -** The java.util.stream.Stream interface serves as a gateway to the bulk data operations. After the reference to a stream instance is acquired, we can perform the interesting tasks with the collections. The Stream API would allow us to declare either sequential or parallel operations over the stream of data:

Stream<Person> stream = persons.stream();

***Filters* : -** Filtering a stream of data is the first natural operation that we would need. Stream interface exposes a filter method that takes in a Predicate that allows us to use lambda expression to define the filtering criteria:

Stream<Person> personsOver18 = persons.stream().filter(p -> p.getAge() > 18);

***Map* : -** The map operations allows us to apply a function that takes in a parameter of one type, and returns something else.

Stream<Student> students = persons.stream()

.filter(p -> p.getAge() > 18)

.map(person -> new Student(person));

**Collect: converting stream to list**

The Stream API provides a number of “terminal” operations. The collect() method is one of those terminals that allows us to collect the results of the operations.

There’s a Collectors utility class for convenience:

List<Student> students = persons.stream()

.filter(p -> p.getAge() > 18)

.map(Student::new)

.collect(Collectors.toList());