14-lambda-function-expressions

Contents

Lambda

The lambda expression is a form of an annonymous class or another form of a functional interface - in other words a unit which represents an executable action, lambda expressions are also called closures, the closure term comes from the fact that they enclose some sort of state, they are a special case of callback functions, which usually do not have the capability to operate on anything else but the state passed in to them through the function parameters (arguments)

Structure

The lambda expression is consisting of two main components, one is the argument definition list, and the other is the lambda body, this is very similar to a way one would define a callback, however as already mentioned lambda's are not the same as callbacks, lambdas have the ability to capture state that is not explicitly provided to them from the outer scope through arguments, the two structural objects `(arg-list)` and the `{ lambda-body }` are connected with the `->` operator which is only used to construct lambda expression in the java language

```
(/* lambda arguments */) -> {
```

```
 // body of the lambda
}
```

Interfaces

As already mentioned in java 8, the spec has introduced a new functional interface, that is an interface which has only one method, that is not a `default` interface method, i.e only one method that has no implementation. This feature ties real nice with the introduction of lambda expression as already mentioned lambda expression are really an anonymous class which has only one function, that is the lambda expression itself, meaning that functional interfaces are not too far off

```
interface Functional {
    int method();
}

Functional instance = () -> 5; // create a lambda and assign it to a type of
   Functional
```

Interfaces which have only one function without an implementation are by default treated as Functional type interfaces by the java compiler

```
When a lambda expression occurs in a target type context, an instance of a class is
automatically created that implements the functional interface
```

Declaration

As already mentioned when in a context of a target type, the lambda expression has to be of well defined type, what this means is that the lambda has to be assigned to a known type always

```
var func = () -> 5; // compiler error, even though java 11 introduced the
   auto-type var, the compiler can not resolve this
Functional func = () -> 5; // this is valid, a correct, explicit type is
   defined for the lambda
```

Syntax

There are several rules which govern the syntax form of a lambda definition and declaration

- When the lambda has only one parameter it need no be enclosed in brackets
- There is no need to explicitly define the argument types they are deduced from the functional interface signature
- The body of the lambda need not be surrounded by curly brackets when it is representing one single statement
- When the lambda body statement is not surrounded by curly brackets an implicit return is provided by the compiler

```
// implicit functional interface
interface Functional {
    int method(int n);
}

// no need to surround body with {}
// an implicit return is added,
// arguments not enclosed in ()
```

```
// a single statement body
Functional func = n -> 5;

func.method(5); // invocation
```

When one has to explicitly provide the type of one of the lambda arguments, then all arguments have to have their type provided, otherwise it is compiler error

Generics

One important point, lambda expression can not be generic, however the functional interface they are created from can be. This is easy to see why, the functional interface is the contract, while the lambda expression or definition is the actual implementation, therefore the type has to be defined when the lambda expression is created or defined

```
interface Functional<T> {
    T method(T n);
}

Functional<Integer> lambdaOne = n -> n.intValue(); // valid, the interface is
    generic, the implementation is specialized
Functional<Double> lambdaTwo = n -> n.doubleValue(); // valid, the interface
   is generic, the implementation is specialized

lambdaOne.method(5); // invoke the first lambda instance with an integer
lambdaTwo.method(5.5); // invoke the first lambda instance with an double
lambdaTwo.method(5); // that is not valid, auto-promotion is not valid for
   lambda expressions, must match signature exactly
```

Automatic promotion does not work with lambda expression, the signature of the lambda must match exactly with the argument types

Exceptions

Since lambdas can throw an exception, if the exception is of checked type, it has to be defined in the `throws` clause of the functional interface, therefore one can define a functional interface such as

```
interface Functional<T> {
    T method(T n) throws IllegalStateException;
}

Functional<Integer> lambdaOne = n -> n.intValue(); // valid, only an instance
    of the functional interface is defined

try {
    lambdaOne.method(5); // correctly invoke the method and catch the
       exception it has declared it might throw
} catch(Exception e) {
    // do something if exception is caught
}
```

Captures

There are strict rules as to what the lambda can use and capture from the outer scope, while a lambda expression can capture local and enclosing class member variables, there are some exceptions as to how one can use those.

- lambdas can capture local variables, but those have to be effectively final, meaning that they can not be re-assigned in the body of the lambda
- lambdas can capture the `this` of the enclosing class and mutate non-final member variables
- lambda defined in a static context (i.e static method) has no access to `this` of the enclosing class
- lambdas can reference static member variables of the enclosing or other classes

```
public class Clazz {
    private static final Double PI = 3.14; // accessible from the lambda
       expression

    private int value = 0; // accessible in a lambda instance created inside
       non-static method
    private Integer boxed = 12; // accessible in a lambda instance created
       inside non-static method

    public void method() {
        int value = 5; // not declared final, but it is assumed to be by the
           compiler, modifications to this local are compile time error
        Integer inner = new Integer(5); // not declared final, but it is
           assumed to be by the compiler, modifications to this local are
           compile time error

        Functional<Integer> lambda = n -> {
            value = value + inner + this.boxed.intValue(); // compile time
               error, `value` refers to the local variable, which always
               final, but is modified
            this.value = 12; // this is valid assign to the enclosing class
               member variable `value`
            this.boxed = this.value + n; // this is valid, `boxed` can be re-
               assigned, new reference
            return this.value + n + PI; // also valid expression in the
               return statement of the lambda
        };

        lambda.method(3);
    }
}
```

References

There is another related feature to lambda expression which allows one to capture a `reference` to an existing method which is compatible with the `lambda's functional interface declaration`. What that means is that java run-time will automatically create an anonymous class instance and lambda from a method reference to a static or non static method. Java uses a special reference operator `::` where on the left is defined the target or instance reference - either an class instance variable, or a class type

Instance To obtain an instance method reference one has to use the instance variable itself as a prefix to the `::` reference operator, as shown below, the function reference refers to the specific variable instance, and the `this` argument to the lambda is the actual instance itself

```
interface Functional {
    char get(int i);
}

String string = "hello-world"; // create an instance of a String type
Functional ref = string::charAt; // reference of an `instance` method of
   String.charAt

// this lambda reference, has internally captured `this` as the `string`
   instance itself
// therefore internally it would invoke this.charAt(n), where `this` is the `
   string` var

ref.get(0); // will return the character at 0-th position in the `string`
   variable
ref.get(1); // will return the character at 1-th position in the `string`
   variable
```

Static Another way to use method reference is to use a reference to a static method, in this case the `join` method in the `String` class is used which accepts the delimiter and elements to be joined. Note that the signature of the `join` does not specify `String` as arguments but rather `CharSequence` however `CharSequence` is a super class of `String` which means that, lambda signature definitions can specify the sub-class and the correct reference will be resolved, and that is not a compile time error

```
    public static String join(CharSequence delimiter, CharSequence...
       elements) // The original function signature for `join` from `String`
```

```
interface Functional {
    String concat(String delim, String... elements);
}

String string = "hello-world"; // create an instance of a String type
Functional ref = String::join; // reference of an `instance` method of String
   .charAt

// this lambda reference, has internally captured `this` as the `string`
   instance itself
// therefore internally it would invoke this.charAt(n), where `this` is the `
   string` var

ref.concat(",", string); // will use the String.join to concat the entries
```

Generics It is also possible to provide a generic reference to a method, the syntax remains mostly the same however there is a small caveat where the type argument of the generic needs to be specified as well. Below is the original functional interface provided by `java.lang` used to compare any two values of the same type, returns an `int` to signify the status of `a compared to b` - is greater (negative), equal (zero) or greater (positive)

```
public interface Comparator<T> {
    int compare(T o1, T o2); // the actual functional interface being used
       and it's only compare function
```

```
class CustomComparator {
    public static <T extends Number> int comp(T a, T b) { // custom
       implementation matching the functional signature
        return a.intValue() - b.intValue();
    }
}

List<Integer> list = List.of(1, 2, 3, 4, 5, 6); // create a list of integers,
    to find the max entry from
int max = Collections.max(list, CustomComparator::<Integer>comp); // compare
   using the custom implementation
```

In the example above, the `max` method of the `Collections` library is used, which is a generic method that takes a reference to the functional interface `Comparator` having a single function compare with `a` and `b` and returns an `int`, in the `CustomComparator` class above the method signature matches the one in `Comparator` which allows one to pass the function reference of `CustomComparator::comp` as a parameter to the `max` method of `Collections`.

Constructors It is also possible to create references to class' constructors, the premise and idea is the same for regular methods, however there are small differences. When referencing constructor as a functional interface, the signature that needs to be matched is very specific, the return type of a constructor reference is obviously the `class type`, the input is the list of arguments the constructor accepts.

```
public interface Functional {
    Integer create(int n);
}

Functional creator = Integer::new; // reference the constructor of Integer
Integer one = creator.create(1); // create an instance of the integer class
Integer two = creator.create(2); // create an instance of the integer class
```

Predefined

In the core library in Java, there are several functional interfaces which are already defined, that for the most part handle pretty much all the cases and patterns one might expect to use in a lambda context. The list below shows the different combinations of lambda function references that can be built with the default functional interfaces provided by the java core

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type T and return the result, which is also of type T. Its method is called apply().
BinaryOperator<T>	Apply an operation to two objects of type T and return the result, which is also of type T. Its method is called apply().
Consumer<T>	Apply an operation on an object of type T. Its method is called accept().
Supplier<T>	Return an object of type T. Its method is called get().
Function<T, R>	Apply an operation to an object of type T and return the result as an object of type R. Its method is called apply().

Interface	Purpose
Predicate<T>	Determine if an object of type T fulfills some constraint. Return a boolean value that indicates the outcome. Its method is called test().