4 - Lambda Expressions and Functional Programming

< Previous | Next >

- None of the primitive specialized functional interfaces (such as IntFunction, DoubleFunction, or IntConsumer) extend the non-primitive functional interfaces (i.e. Function, Consumer, and so on).
- java.util.function.IntFunction takes int primitive as an argument. It can be parameterized to return any thing.
- Whenever you want to process streams of primitive elements, you should use the primitive specialized streams (i.e. IntStream, LongStream, and DoubleStream) and primitive specialized functional interfaces (i.e. IntFunction, IntConsumer, IntSupplier etc.) to avoid additional cost associated with auto-boxing/unboxing and achieve better performance.
- The reduce method needs a BinaryOperator. This interface is meant to consume two arguments and produce one output. It is applied repeatedly on the elements in the stream until only one element is left. The first argument is used to provide an initial value to start the process. (If you don't pass this argument, a different reduce method will be invoked and that returns an Optional object.)
- The Stream.max method requires an argument of type Comparator.
- `ls.stream().max(Comparator.comparing(a->a)).get();` Comparator.comparing method requires a Function that takes an input and returns a Comparable. This Comparable, in turn, is used by the comparing method to create a Comparator. The max method uses the Comparator to compare the elements int he stream. The lambda expression a->a creates a Function that takes an Integer and returns an Integer (which is a Comparable).
- Integer.max works very differently from Integer.compare. The max method returns the maximum of two numbers while the compare method returns a difference between two numbers.
- All intermediate operations return Stream (that means, they can be chained), while terminal operations don't.
- filter, peek, and map are intermediate operations. count, forEach, sum, allMatch, noneMatch, anyMatch, findFirst, and findAny are terminal operations.
- Remember that while compute and computeIfPresent take a BiFunction as an argument, computeIfAbsent takes a Function.
- A Function's return type can be different from its input but it is possible to use Function where the input type and return type are same.
- UnaryOperator and BinaryOperator always returns the same type as the type of its input(s).
- forEach method expects a Consumer as an argument. Not a Function.
- Remember that Consumer doesn't return anything. Therefore, the body of the lambda expression used to capture Consumer must be an expression of type void.
- Map's forEach method requires a BiConsumer object. A BiConsumer takes exactly two parameters but println method takes only one and therefore cannot be used to implement BiConsumer.
- filter method takes only one argument of type Predicate. If you want to apply multiple filters, you can chain multiple filters to a Stream.
- `Predicate even = (Integer i) -> i % 2 == 0;` : compile error expects Object but found Integer. Fix is `Predicate<Integer> even = (Integer i)-> i%2==0;` of `Predicate even = (Object i)-> ((Integer)i)%2==0;` or even `Predicate even = i -> ((Integer)i)%2==0;` .
- `List<Double> dList = Arrays.asList(10.0, 12.0);` `dList.stream().forEach(x->{ x = x+10; });` Remember that the variables are passed by value. Therefore, when a new Double object is assigned to x by the statement `x = x + 10;` , the original element in the list is not changed. Therefore, the first call to forEach does not change the elements in the original list on which the stream is based.
- Function takes one argument and returns a value. So `Function<Type>` will not compile. It should actually be `Function<T, R>` . Function expects an argument to be passed. Thus, it should be `f.apply(argument);`

# Method/constructor references:

- An important point to understand with method or constructor references is that you can never pass arguments while referring to a constructor or a method. Remember that such references are mere references. They are not actual invocations.
- Basically, when you do `Supplier<MyProcessor> s = MyProcessor:new;` you are telling the compiler to get you the constructor reference of the constructor that does not take any argument. This is because Supplier's functional method does not take any argument.
- On the other hand, when you do `Function<Integer, MyProcessor> f =  MyProcessor::new;` you are telling the compiler to get you the constructor reference of the constructor that takes one Integer argument. The compiler figures this out because the functional method of Function interface requires one argument and you have parameterized it to Integer. So the compiler looks for a constructor that takes an Integer (or int) argument and gives you that constructor's reference. The constructor or the method is not invoked at this time and therefore, no argument is needed at this time.
  Arguments are required only when you actually invoke the constructor or a method.
- Therefore, code such as `MyProcessor::new(10);` doesn't make sense. You cannot pass arguments while taking a reference. You pass arguments when you use the reference to invoke it as done in:
  `MyProcessor mp = f.apply(10);` This works because f is already defined to use a constructor reference that takes a parameter. 10 is passed to that constructor.

# java.util.BiFunction:

1. It is a function that accepts two arguments and produces a result.
2. The types of the arguments and the return value can all be different.

# Three flavors of compute methods of Map:

1. `public V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` If the function returns null, the mapping is removed (or remains absent if initially absent). If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged. Returns: the new value associated with the specified key, or null if none.
2. `public V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)` If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null. Returns: the current (existing or computed) value associated with the specified key, or null if the computed value is null.
3. `public V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)` If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value. If the function returns null, the mapping is removed. Returns: the new value associated with the specified key, or null if none

# Three versions of reduce:

- `Optional<T>  reduce(BinaryOperator<T> accumulator)` Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
- `T  reduce(T identity, BinaryOperator<T> accumulator)` Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
- `<U> U  reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)` Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
- **NOTE**: If you don't pass an identity, the first reduce method will be invoked and that returns an Optional object.

< Previous | Next >