4-lambda-functional-interface

Contents

- Functional interface
  - Functional
    - Method References
    - Constructor references
  - Interfaces
    - Primitive
    - Binary
    - Unary
- Summary

Functional interface

The java utility function namespace package has numerous built-in interfaces. Other packages in the Java library make use of these interfaces defined in this package.

`Before defining a custom functional interface, consider using an existing one, defined in the util.function package, since most of the functional interfaces there cover a lot of the most common patterns used in lambda design expressions, and can also be shared and used with existing internal java libraries like Stream and so on.`

Functional

The table below shows the most commonly used functional interfaces defined in the `java.util.functional` package. What is important to note is that most of them have variations, meaning that there are different versions of some - e.g `IntPredicate`, which is a specialization of the predicate functional lambda interface, for integral conditions and is used in `IntStream`. Another example is the `UnaryOperator`, which is a specialization for Function lambda functional interface, which takes and returns only one singular result

Method	Description
Predicate<T>	Checks a condition and returns a boolean value as result In filter() method in java.util.stream.Stream which is used to remove elements in the stream that don't match the given condition (i.e., predicate) as argument.
Consumer<T>	Operation that takes an argument but returns nothing In forEach() method in collections and in java.util.stream.Stream; this method is used for traversing all the elements in the collection or stream.
Function<T, R>	Functions that take an argument and return a result In map() method in java.util.stream.Stream to transform or operate on the passed value and return a result.
Supplier<T>	Operation that returns a value to the caller (the returned value could be same or different values)

The interface method names for these functional interfaces are rather easy to remember since they extend off of the name of the interface itself.

- `Predicate` - boolean test(T t) - execute the actual predicate action and return a boolean result, based on the target argument
- `Consumer` - void accept(T t) - no result is returned, but some action is executed on the target argument
- `Function` - R apply(T t) - apply action based on the target argument, and return a new result of new type
- `Supplier` - T get() - simply return a result, without accepting any external argument

Method References

Most of the common implementations work by using function references from existing libraries and APIs, to leverage the usage of the functional interfaces the best, for example

```
// predicate using the String::isEmpty, function reference, the argument of
   the predicate will be each string entry in
// the stream, the result, will be calling isEmpty on each of them, which
   returns boolean
Stream.of("", "non-empty")
    .removeIf(String::isEmpty)
    .forEach(System.out::println);

// consumer using the standard out println, which returns no result, however
   takes in at least one argument, to print
// out the elements of the stream in this case
Stream.of("hello", "world")
    .forEach(System.out::println);

// function which maps a string to int, by leveraging the parseInt method,
   from Integer, which takes in one type of
// argument, String in this case, and returns another one, Integer, or throws
    if parsing fails
Arrays.stream("4, -9, 16".split(", "))
    .map(Integer::parseInt)
    .map(i -> (i < 0) ? -i : i)
    .forEach(System.out::println);

// supplier, which is in this case a function without any input arguments
   however it generates something on the output,
```

```
// in this case on each call to nextBoolean, a new random boolean value will
   be generated
Random random = new Random();
Stream.generate(random::nextBoolean)
    .limit(2)
    .forEach(System.out::println);
```

Note that the examples above use function references, which is another feature which is a corner stone of the lambda and function interface feature in Java 8, which inter operates very well with streams, those three, combine effortlessly to achieve very clean and non aggressively verbose functional style of code expression

Something to take away from, is the usage of `instance` methods as function references, such as `String::isEmpty` and `random::nextBoolean`, both of those methods are `instance` methods, however in the example with the String, the implicit `this` argument is passed in as the first argument to `isEmpty` which allows us to reference an instance method, by not referencing any instance at all, the `instance` is each entry in the stream for which the method will be called, and the signature matches the `predicate` interface - `boolean test(String this)`

However with the `random::nextBoolean`, the instance reference is in the declaration, therefore the first argument to `nextBoolean` will be the `random` instance variable. Therefore the signature matches the `supplier` interface - `Boolean get()`

The example with Integer:parseInt, matches the functional interface of Function, however parseInt is a static final method in the Integer class, and in this case no instance method is invoked, the first argument to parseInt is String, the output is Integer, it is as simple as that, it matches the `function` interface - `Integer apply( String int)`

The difference is subtle, but this feature is very powerful, however sometimes hard to get right, or read.

Constructor references

The constructor references for any class are not any different than regular method references, in fact they very much are like static methods, they are not methods that take a reference to `this`, since they are not instance methods. Constructors take any number of arguments, usually none, but sometimes one or more than one, the return of the constructor is obviously the actual class instance or an object of the class type.

```
Supplier<String> newString = String::new;
System.out.println(newString.get());
```

The example above shows how a class constructor can be mapped to an existing functional interface from the java.util package, in this case it matches against the supplier functional interface. With a method reference using `::new` this lambda expression gets simplified as `String::new`.

Interfaces

Below are listed some of the most important functional interfaces provided in the `java.util.function`, those extend the core four ones, with different capabilities, which are useful for different contexts, mainly trying to create all possible cases and combinations between raw class types and raw primitive types

Primitive

There are primitive versions of the functional interface classes provided by `java.util`, these are created to be able to deal with the primitive versions of the stream types like `IntStream` and `DoubleStream` for example, here is a table with the list of primitive functional interfaces

Method	Description
IntPredicate boolean test(int value)	Evaluates the condition passed as int and returns a boolean value as result
LongPredicate boolean test(long value)	Evaluates the condition passed as long and returns a boolean value as result
DoublePredicate boolean test(double value)	Evaluates the condition passed as double and returns a boolean value as result
IntFunction<> R apply(int value)	Operates on the passed int argument and returns value of generic type R
LongFunction<> R apply(long value)	Operates on the passed long argument and returns value of generic type R
DoubleFunction<> R apply(double value)	Operates on the passed double argument and returns value of generic type R
ToIntFunction<> int applyAsInt(T value)	Operates on the passed generic type argument T and returns an int value
ToLongFunction<> long applyAsLong(T value)	Operates on the passed generic type argument T and returns a long value
ToDoubleFunction<> double applyAsDouble(T value)	Operates on the passed generic type argument T and returns an double value
IntToLongFunction long applyAsLong(int value)	Operates on the passed int type argument and returns a long value
IntToDoubleFunction double applyAsDouble(int value)	Operates on the passed int type argument and returns a double value
LongToIntFunction int applyAsInt(long value)	Operates on the passed long type argument and returns an int value
LongToDoubleFunction double applyAsLong(long value)	Operates on the passed long type argument and returns a double value
DoubleToIntFunction int applyAsInt(double value)	Operates on the passed double type argument and returns an int value
DoubleToLongFunction long applyAsLong(double value)	Operates on the passed double type argument and returns a long value
IntConsumer void accept(int value)	Operates on the given int argument and returns nothing
LongConsumer void accept(long value)	Operates on the given long argument and returns nothing
DoubleConsumer void accept(double value)	Operates on the given double argument and returns nothing
ObjIntConsumer<> void accept(T t, int value)	Operates on the given generic type argument T and int arguments and returns nothing
ObjLongConsumer<> void accept(T t, long value)	Operates on the given generic type argument T and long arguments and returns nothing
ObjDoubleConsumer<> void accept(T t, double value)	Operates on the given generic type argument T and double arguments and returns nothing
BooleanSupplier boolean getAsBoolean()	Takes no arguments and returns a boolean value
IntSupplier int getAsInt()	Takes no arguments and returns an int value

Method	Description
LongSupplier long getAsLong()	Takes no arguments and returns a long value
DoubleSupplier double getAsDouble()	Takes no arguments and returns a double value

```
// in this example the predicate function is passed to the filter
   intermediate stream function, the predicate here is of
// the functional interface type `IntPredicate`
IntStream.range(1, 10).filter(i -> (i % 2) == 0).forEach(System.out.println);
```

Binary

The `binary` interfaces from `java.util` simply expand the already existing pattern of other functional interfaces, the idea behind this is that to add one additional input argument to all of the existing functional interfaces - Consumer, Function, Consumer, converting them into a binary or bi functional interfaces where the actual signature of the functional interface remains unchanged, simply an additional argument is added

Method	Description
BiPredicate<, U> boolean test(T t, U u)	Checks if the arguments match the condition and returns a boolean value as result
BiConsumer<, U> void accept(T t, U u)	Operation that consumes two arguments but returns nothing
BiFunction<, U, R> R apply(T t, U u)	Function that takes two argument and returns a result

As shown above the interface remains the same, meaning the method names are unchanged, however they are now expanded with one additional argument

Unary

The `unary` functional interfaces is a variation or specialization of the Function functional interface, it simply marks all input and output type to be of one and the same type `<T>` the signature of the unary functional interface looks like so - `UnaryOperator<T> extends Function<T, T>`

Method	Description
UnaryOperator<> T accept(T t)	Function that takes one argument of the same type and produces result of the same type

`The entirety of the java.util.function packages consists of only functional interfacse . There are only four core interfaces in this package - Predicate, Consumer, Function & Supplier. The rest of the interfaces are primitive versions, binary versions and derived interfaces such as UnaryOperator interfaces. These interfaces differ mainly on the signature of the abstract methods they declare.`

Summary

Built-in interfaces

- Built-in functional interfaces Predicate, Consumer, Function, and Supplier differ mainly based on the signature of the abstract method they declare.
- A Predicate tests the given condition and returns true or false; hence it has an abstract method named "test" that takes a parameter of generic type T and returns boolean type.
- A Consumer "consumes" an object and returns nothing; hence it has an abstract method named "accept" that takes an argument of generic type T and has return type void.
- A Function "operates" on the argument and returns a result; hence it has an abstract method named "apply" that takes an argument of generic type T and has generic return type R.
- A Supplier "supplies" takes nothing but returns something; hence it has an abstract method named "get" that takes no arguments and returns a generic type T.
- The forEach() method defined in `Iterable` (implemented by collection classes) method accepts a Consumer.

Primitive versions

- The built-in interfaces Predicate, Consumer, Function, and Supplier operate on reference type objects. For primitive types, there are specializations available for int, long, and double types for these functional interfaces.
- When the Stream interface is used with primitive types, it results in unnecessary boxing and unboxing of the primitive types to their wrapper types. This results in slower code as well as wastes memory because the unnecessary wrapper objects get created. Hence, whenever possible, prefer using the primitive type specializations of the functional interfaces Predicate, Consumer, Function, and Supplier.
- The primitive versions of the functional interfaces Predicate, Consumer, Function, and Supplier are available only for int, long and double types (and boolean type in addition to these three types for Supplier). You have to use implicit conversions to relevant int version when you need to use char, byte, or short types; similarly, you can use the version for double type when you need to use float. Develop code that uses binary versions of functional interfaces
- The functional interfaces `BiPredicate`, `BiConsumer`, and `BiFunction` are binary versions of Predicate, Consumer, and Function respectively. There is no binary equivalent for Supplier since it does not take any arguments. The prefix "Bi" indicates the version that takes "two" arguments.

Unary versions

- is a functional interface and it extends Function interface.
- primitive type specializations of `UnaryOperator` are `IntUnaryOperator`, `LongUnaryOperator`, and `DoubleUnaryOperator` for int, long, and double types respectively.