SOFTWARE ENGINEERING

Java 8 functional interfaces

Getting to know various out-of-the-box functions such as Consumer, Predicate, and Supplier.

By Madhusudhan Konda. August 7, 2014

Open wooden box (source: Pixabay)

In the first part of this series, we learned that lambdas are a type of functional interface – an interface with a single abstract method. The Java API has many one-method interfaces such as `Runnable`, `Callable`, `Comparator`, `ActionListener` and others. They can be implemented and instantiated using anonymous class syntax. For example, take the `ITrade` functional interface. It has only one abstract method that takes a `Trade` object and returns a boolean value – perhaps checking the status of the trade or validating the order or some other condition.

```
@FunctionalInterface
public interface ITrade {
  public boolean check(Trade t);
}
```

In order to satisfy our requirement of checking for *new* trades, we could create a lambda expression, using the above functional interface, as shown here:


```
ITrade newTradeChecker = (Trade t) -> t.getStatus().equals("NEW");

// Or we could omit the input type setting:
ITrade newTradeChecker = (t) -> t.getStatus().equals("NEW");
```

The expression is expecting a `Trade` instance as an input argument, declared to the left of the arrow token (we could omit the type of the input). The right side of the expression is simply the body of the check method – checking the status of the passed in Trade. The return type is implicitly boolean type as the check method returns boolean. The real power of using lambdas comes when you start creating a multitude of them representing real-world behavioral functions. For example, in addition to what we've already seen, here are the lambda expressions for finding out big trade (i.e, if the

trade's quantity is greater than 1 million) or checking out a newly created large Google trade:

```
// Lambda for big trade
ITrade bigTradeLambda = (Trade t) -> t.getQuantity() > 10000000;

// Lambda that checks if the trade is a new large google trade
ITrade issuerBigNewTradeLambda = (t) -> {
    return t.getIssuer().equals("GOOG") &&
           t.getQuantity() > 10000000 &&
           t.getStatus().equals("NEW");
  };
```

These functions can then be passed on to a method (most probably server side) which takes in an `ITrade` as one of its parameters. Let's say, we have a collection of trades and wish to filter out some trades based on a certain criteria. This requirement can be easily expressed using the above lambda passing to a method which accepts a list of trades too:

```
// Method that takes in list of trades and applies the lambda behavior
for each of the trade in the collection

private List<Trade> filterTrades(ITrade tradeLambda, List<Trade>
trades) {
  List<Trade> newTrades = new ArrayList<>();

  for (Trade trade : trades) {
    if (tradeLambda.check(trade)) {
      newTrades.add(trade);
    }
  }
  return newTrades;
}
```

The above `filterTrades` method expects a collection trade and a lambda to be applied on each of the trades in the collection one by one, by iterating over them. If

the input trade satisfies the pre-conditions defined by the lambda, the trade is added to the cumulative basket, else thrown away. The good thing about this method is that it accepts *any* lambda as long as it implements an `ITrade` interface. The method is shown below by taking the various lambdas:

```
// Big trades function is passed
List<Trade> bigTrades =
  client.filterTrades(bigTradeLambda,tradesCollection);

// "BIG+NEW+ISSUER" function is passed
List<Trade> bigNewIssuerTrades =
  client.filterTrades(issuerBigNewTradeLambda,tradesCollection);

// cancelled trades function is passed
List<Trade> bigNewIssuerTrades =
  client.filterTrades(cancelledTradesLambda,tradesCollection);
```


Pre-Built functions library

There are a lot of re-usable functional requirements that can be captured by functional interfaces and lambdas. The designers of Java 8 have captured the common use cases and created a library of functions for them. A new package called `java.util.function` was created to host these common functions. In our `ITrade` case, all we're doing is checking the boolean value of business functionality based on a condition. You can test a condition on any other object (not just on Trades). For example, you can check whether an employee is a member of a long list of employees; whether your train from London to Paris running on time; whether today is a sunny day, etc. The overall goal is to check for a condition and return true or false based on this. Taking the commonality of such use cases into account, Java 8 introduced a functional interface called `Predicate` that does exactly what we're doing with `ITrade` – checking an input for its correct (true/false) value. Instead of writing our own functional interfaces, we can use the standard library of well defined, multi-faceted functional interfaces. There are a handful of these interfaces added to the library, which we'll discuss in the next section.

java.util.Predicate

We need a function for checking a condition. A `Predicate` is one such function accepting a single argument to evaluate to a boolean result. It has a single method `test` which returns the boolean value. See the interface definition below which is a generic type accepting any type T:

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

From our knowledge of lambdas so far, we can deduce the lambda expression for this `Predicate.` Here are some example lambda expressions:

```
// A large or cancelled trade (this time using library function)!
Predicate<Trade> largeTrade = (Trade t) -> t.isBigTrade();
```

```
// Parenthesis and type are optional
Predicate<Trade> cancelledTrade = t -> t.isCancelledTrade();

// Lambda to check an empty string
Predicate<String> emptyStringChecker = s -> s.isEmpty();

// Lambda to find if the employee is an executive
Predicate<Employee> isExec = emp -> emp.isExec();
```

The invocation of the functionality is similar to what we've seen in the `ITrade` case. The `Predicate`'s `test` is invoked to check the true vale, as shown in the following code snippet:

```
// Check to see if the trade has been cancelled
boolean cancelledTrade = cancelledTrade.test(t);

// Check to find if the employee is executive
boolean executive = isExec.test(emp);
```

java.util.Function

A `Function` is a functional interface whose sole purpose is to return any result by working on a single input argument. It accepts an argument of type T and returns a result of type `R`, by applying specified logic on the input via the `apply` method. The interface definition shown here:

```
// The T is the input argument while R is the return result
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
```

We use a `Function` for transformation purposes, such as converting temperature from Centigrade to Fahrenheit, transforming a String to an Integer, etc:

```
// convert centigrade to fahrenheit
Function<Integer,Double> centigradeToFahrenheitInt = x -> new
Double((x*9/5)+32);

// String to an integer
Function<String, Integer> stringToInt = x -> Integer.valueOf(x);

// tests
System.out.println("Centigrade to Fahrenheit:
"+centigradeToFahrenheitInt.apply(centigrade))
System.out.println(" String to Int: " + stringToInt.apply("4"));
```

Did you notice the two arguments to the `Function` - `String` and `Integer`? This indicates that the function is expecting a `String` and returning the `Integer`. A bit more sophisticated requirement, like aggregating the trade quantities for a given list of trades, can be expressed as a `Function`, shown below:

```
// Function to calculate the aggregated quantity of all the trades -
taking in a collection and returning an integer!
Function<List<Trade>,Integer> aggegatedQuantity = t -> {
  int aggregatedQuantity = 0;
  for (Trade t: t){
    aggregatedQuantity+=t.getQuantity();
  }
  return aggregatedQuantity;
};
```

The aggregation of the above trades can be done using Stream API's "fluent" style:

```
// Using Stream and map and reduce functionality

aggregatedQuantity =
  trades.stream()
  .map((t) -> t.getQuantity())
```

```
    .reduce(0, Integer::sum);

// Or, even better
aggregatedQuantity =
    trades.stream()
    .map((t) -> t.getQuantity())
    .sum();
```

Other functions

Java 8 provides few other functions out of the box, such as `Consumer`, `Supplier` and others. The `Consumer` accepts a single argument but does not return any result:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

This is mostly used to perform operations on the arguments such as persisting the employees, invoking house keeping operations, emailing newsletters etc. The `Supplier`, as the name suggests, supplies us with a result; here's the method signature:

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

For example, fetching configuration values from database, loading with reference data, creating an list of students with default identifiers etc, can all be represented by a supplier function. On top of these functions, Java 8 provides specialized versions of these functions too for specific use cases. For example, the `UnaryOperator` which

extends a `Function` acts only on same types. So, if we know that both the input and output types are the same, we could use `UnaryOperator` instead of `Function`.

```
// Here, the expected input and return type are exactly same.
// Hence we didn't use Function, but we are using a specialized sub-
class

UnaryOperator<String> toLowerUsingUnary = (s) -> s.toLowerCase();
```

Did you notice that the function is declared with one generic type? In this case, both being the same `String` - so, instead of writing `Function`, we managed to shorten it even further by using `+UnaryOperator`. Going a bit further, functions are also provided to represent primitive specializations, `DoubleUnaryOperator`, `LongUnaryOperator`, `IntUnaryOperator` etc. They basically deal with operations on primitives such as a `double` to produce a `double` or `long` to return a `long`, etc. Just like `Function` has children to accommodate special cases, so do other functions like `IntPredicate` or `LongConsumer` or `BooleanSupplier`, etc. For example, `IntPredicate` represents an integer value based function, `LongConsumer` expects a long value returning no result, and `BooleanSupplier` supplies boolean values. There are plethora of such specializations, so I would strongly advocate that you run through the API in detail to understand them.

Two argument functions

Until now, we have dealt with functions that only accept a single input argument. There are use cases that may have to operate on two arguments. For example, a function that expects two arguments but produces a result by operating on these two arguments. This type of functionality fits into two-argument functions bucket such as `BiPredicate`, `BiConsumer`, `BiFunction`, etc. They are pretty easy to understand too except that the signature will have an additional type (two input types and one return type). For instance, the `BiFunction` interface definition is shown below

```
@FunctionalInterface
```

```
public interface BiFunction<T, U, R> {
  R apply(T t, U u);
}
```

The above function has three types – `T`, `U` and `R`. The first two are are input types while the last one is the return result. For completeness, the following example provides a snippet of `BiFunction` usage, which accepts two `Trades` to produce sum of trade quantities. The input types are `Trade` and return type is `Integer`:

```
  BiFunction<Trade,Trade,Integer> sumQuantities = (t1, t2) -> {
    return t1.getQuantity()+t2.getQuantity();
  };
```

Going with the same theme of two argument functions, the `BiPredicate` expects two input arguments and returns a boolean value (no surprise!):

```
// Predicate expecting two trades to compare and returning the
condition's output
BiPredicate<Trade,Trade> isBig = (t1, t2) -> t1.getQuantity() >
t2.getQuantity();
```

As you may have guessed by now, there are specializations of these two-argument functions too. For example a function that operates on the *same* type of operands and emits the same type. Such facility is captured in `BinaryOperator` function. Note that the extends `BinaryOperator` extends `BiFunction`. The following example shows `BinaryOperator` in action – it accepts two `Trades` to produce a merged trade (which is of the same type as the input arguments – remember, the `BinaryOperator` is a special case of `BiFunction`).

```
BiFunction<Trade,Trade,Trade> tradeMerger2 = (t1, t2) -> {
  // calling another method for merger
  return merge(t1,t2);
```

```
};

// This method is called from the lambda.
// You can prepare a sophisticated algorithm to merge a trade in here
private Trade merge(Trade t1, Trade t2){
  t1.setQuantity(t1.getQuantity()+t2.getQuantity());
  return t1;
}
```

Note that we did not pass in all three arguments when declaring the type (expectation is that all the inputs and outputs are of same type). Also, did you notice that the actual logic is carried out by the super function `BiFunction` rather than `BinaryOperator`? This is because the `BinaryOperator` extends the `BiFunction`:

```
//BinaryOperator is a special case of BiFunction.

public interface BinaryOperator<T> extends BiFunction<T,T,T> { .. }
```

Now that we have seen the functions and functional interfaces, there is something I would like to reveal about interfaces. Until Java 8, the interfaces were abstract creatures – you certainly cannot add implementation to it, making it brittle in some ways. However, Java 8 re-engineered this, calling them virtual methods.

Virtual (default) methods

The journey of a Java library begins with a simple interface. Over time, these libraries are expected to evolve and grow in functionality. However, pre-Java 8 interfaces are untouchable! Once they are defined and declared, they are practically written in stone. For obvious reasons, backward compatibility being the biggest one, they cannot be changed after the fact. While it's great that lambdas have been added to the language, there's no point of having them if they cannot be used with existing APIs. In order to add support to absorb lambdas into our libraries, the interfaces needed to evolve. That is, we need to be able to add additional functionality to an already published API. The dilemma is how to embrace lambdas to create or enhance the APIs without losing

backward compatibility? This requirement pushed Java designers to come up with yet another elegant feature – providing *virtual extension* methods or simply *default* methods to the interfaces. This means we can create concrete methods in our interfaces going forward. The virtual methods allow us to add newer functionality too. The collections API is one such example, where bringing lambdas into the equation has overhauled and enhanced the APIs. Let us walk through a simple example to demonstrate the default method functionality. Let's assume that every component is said to have a name and creation date when instantiated. However, should the implementation doesn't provide concrete implementation for the name and creation date, they would be inherited from the interface by default. For out example, the `IComponent` interface defines a set of default methods as shown below:

```
@FunctionalInterface
public interface IComponent {

  // Functional method – note we must have one of these functional
methods only
  public void init();

  // default method – note the keyword default
  default String getComponentName(){
    return "DEFAULT NAME";
  }
  // default method – note the keyword default
  default Date getCreationDate(){
    return new Date();
  }
}
```

As you can see from the snippet above, the interfaces now have an implementation rather than just being abstract. The methods are prefixed with a keyword `default` to indicate them as the default methods.

Multiple inheritance

Multiple inheritance is not new to Java. Java has provided multiple inheritance of types since its inception. If we have an object hierarchy implementing various interfaces,

there are a few rules help us understand which implementation is applied to the child class. The fundamental rule is that the closest concrete implementation to the subclass wins the inherited behavior over others. The immediate concrete type has the precedence over any others. Take, for example, the following class hierarchy.

```
// Person interface with a concrete implementation of name
interface Person{
  default String getName(){
    return "Person";
  }
}
// Faculty interface extending Person but with its own name
implementation
interface Faculty extends Person{
  default public String getName(){
    return "Faculty";
  }
}
```

So, both `Person` and `Faculty` interfaces provide default implementation for name. However, note that `Faculty` extends Person but overrides the behavior to provide its own implementation. For any class that implements both these interfaces, the name is inherited from `Faculty` as it is the closest subtype to the child class. So, if I have a `Student` subclass implementing `Faculty` (and `Person`), the Student's `getName()` method prints the Faculty's name:

```
// The Student inherits Faculty's name rather than Person
class Student implements Faculty, Person{ .. }

// the getName() prints Faculty
private void test() {
  String name = new Student().getName();
  System.out.println("Name is "+name);
}
output: Name is Faculty
```

However, there's one important point to note. What happens if our `Faculty` class does not extend `Person` at all? In this case, Student class inherits `name` from both implementations, thus making the compiler moan. To make our compiler happy, in this case, we must provide the concrete implementation by ourselves. However, if you wish to inherit one of the super type's behaviors, you can do so explicitly, as we see in the code snippet below.

```
interface Person{ .. }

// Notice that the faculty is NOT implementing Person
interface Faculty { .. }

// As there's a conflict, out Student class must explicitly declare
whose name it's going to inherit!
class Student implements Faculty, Person{
  @Override
  public String getName() {
    return Person.super.getName();
  }
}
```

There's a special syntax to obtain the method from a super interface – using super-interface.super.method: `Person.super.getName()`.

Method references

In a lambda expression, when we are calling an already existing method of an existing class or super class, method and class references come in handy. These are new utility features introduced in Java 8 in order to represent lambdas even more concisely and succinctly. Method references are shortcuts for calling existing methods. For example, take a class that has already a method that adds up two integers:

```
public class AddableTest {
  // Add given two integers
  private int addThemUp(int i1, int i2){
```

```
    return i1+i2;
}
```

Because the method of adding the integers already exists, there's no point of creating a lambda expression doing exactly the same. Hence, we refer to this existing method via a method reference (using double colon ::), when create a lambda for `IAddable` implementation:

```
public class AddableTest {
  // Lambda expression using existing method
  IAddable addableViaMethodReference = this::addThemUp;

  // Add given two integers
  private int addThemUp(int i1, int i2){
    return i1+i2;
  }
}
```

Notice the `this::addThemUp` lambda expression in the code above. The `this` refers to an instance of the `AddableTest` class while the bit after the double colon is the call to pre-existing method. Also, take a note that the method reference doesn't add the braces `()` at the end. Should you have another class that implements desired functionality via a static method, you can simply use its method in a lambda expression by using this feature of method references. See the example given below:

```
// Class that provides the functionality via it's static method
public class AddableUtil {
  public static int addThemUp(int i1, int i2){
    return i1+i2;
  }
}

// Test class
public class AddableTest {
  // Lambda expression using static method on a separate class
  IAddable addableViaMethodReference = AddableUtil::addThemUp;
```

```
    ...
}
```

We can also use a constructor reference by simply calling the class's constructor as `Employee::new` or `Trade::new`.

Summary

In this post, we learned more about Java functional interfaces and functions. We dived into understanding the various out-of-the-box functions such as `Consumer`, `Predicate`, `Supplier`, etc. We also looked at virtual methods and method references. In the next post in this series, we will look at the Stream API in detail.

Article image: Open wooden box (source: Pixabay).

Madhusudhan Konda

Madhusudhan Konda is an experienced Java consultant working in London, primarily with investment banks and financial organizations. Having worked in enterprise and core Java for the last 12 years, his interests lie in distributed, multi-threaded, n-tier scalable, and extensible architectures. He is experienced in designing and developing high-frequency and low-latency application architectures. He enjoys writing technical papers and is interested in mentoring.

more