Using Lambdas Expressions in Your Application

The introduction of lambda expressions in Java SE 8 came with a major rewrite of the JDK API. More classes have been updated in the JDK 8 following the introduction of lamdbas than in the JDK 5 following the introduction of generics.

Thanks to the very simple definition of *functional interfaces*, many existing interfaces became *functional* without having to modify them. The same goes for your existing code: if you have interfaces in your application written prior to Java SE 8, they may become functional without having to touch them, making it possible to implement them with lambdas.

Discovering the `java.util.function` package

The JDK 8 also introduces a new package: `java.util.function` with functional interfaces for you to use in your application. These functional interfaces are also heavily used in the JDK API, especially in the Collections Frameworks and the Stream API. This package is in the `java.base` module.

With a little more than 40 interfaces, this package may look a little scary at first. It turns out that it is organized around four main interfaces. Understanding them gives you the key to understand all the others.

Creating or Providing Objects with `Supplier<T>`

Implementing the `Supplier<T>` Interface

The first interface is the `Supplier<T>` interface. In a nutshell, a supplier does not take any arguments and returns an object.

We should really say: a lambda that implements the supplier interface does not take any argument and returns an object. Making shortcuts makes things easier to remember, as long as they are not confusing.

This interface is really simple: it has no default or static method, just a plain `get()` method. Here is this interface:

```
1  @FunctionalInterface
2  public interface Supplier<T> {
3
4      T get();
5  }
```

The following lambda is an implementation of this interface:

```
Supplier<String> supplier = () -> "Hello Duke!";`
```

This lambda expression simply returns the `Hello Duke!` string of characters. You can also write a supplier that returns a new object every time it is invoked:

```
Random random = new Random(314L);
Supplier<Integer> newRandom = () -> random.nextInt(10);

for (int index = 0; index < 5; index++) {
    System.out.println(newRandom.get() + " ");
}
```

Calling the `get()` method of this supplier will invoke `random.nextInt()`, and will produce random integers. Since the seed of this random generator is fixed to the value `314L`, you should see the following random integers generated:

```
1
5
3
0
2
```

Note that this lambda is capturing a variable from the enclosing scope: `random`, making this variable *effectively final*.

Using a `Supplier<T>`

Note how you generated random numbers using the `newRandom` supplier in the previous example:

```
for (int index = 0; index < 5; index++) {
    System.out.println(newRandom.get() + " ");
}
```

Calling the `get()` method of the `Supplier` interface invokes your lambda.

Using Specialized Suppliers

Lambda expressions are used to process data in applications. How fast a lambda expression can be executed is thus critical in the JDK. Any CPU cycle that can be saved has to be saved, since it may represent a significant optimization in a real application.

Following this principle, the JDK API also offers specialized, optimized versions of the `Supplier<T>` interface.

You may have noticed that our second example supplies the `Integer` type, where the `Random.nextInt()` method returns an `int`. So in the code you wrote, there are two things that are happening under the hood:

- the `int` returned by the `Random.nextInt()` is first boxed into an `Integer`, by the auto-boxing mechanism;
- this `Integer` is then unboxed when assigned to the `nextRandom` variable, by the auto-unboxing mechanism.

The auto-boxing is the mechanism by which an `int` value can be directly assigned to an `Integer` object:

```
int i = 12;
Integer integer = i;
```

Under the hood, an object is created for you, wrapping that value.

The auto-unboxing does the opposite. You may assign an `Integer` to an `int` value, by unwrapping the value within the `Integer`:

```
Integer integer = Integer.valueOf(12);
int i = integer;
```

This boxing / unboxing does not come for free. Most of the time, this cost will be small compared to other things your application is doing, like getting data from a database or from a remote service. But in some cases, this cost may be not acceptable, and you need to avoid paying it.

The good news is: the JDK gives you a solution with the `IntSupplier` interface. Here is this interface:

```
@FunctionalInterface
public interface IntSupplier {

    int getAsInt();
}
```

Notice that you can use the exact same code to implement this interface:

```
Random random = new Random(314L);
IntSupplier newRandom = () -> random.nextInt();
```

The only modification to your application code is that you need to call `getAsInt()` instead of `get()`:

```
for (int i = 0; i < 5; i++) {
    int nextRandom = newRandom.getAsInt();
    System.out.println("next random = " + nextRandom);
}
```

The result of running this code is the same, but this time no boxing / unboxing occurred: this code is more performant than the previous one.

The JDK gives you four of these specialized suppliers, to avoid unnecessary boxing / unboxing in your application: `IntSupplier`, `BooleanSupplier`, `LongSupplier` and `DoubleSupplier`.

> *You will see more of these specialized version of functional interfaces to handle primitive types. There is a simple naming convention for their abstract method: take the name of the main abstract method (`get()` in the case of the supplier), and add the returned type to it. So for the supplier interfaces we have: `getAsBoolean()`, `getAsInt()`, `getAsLong()`, and `getAsDouble()`.*

Consuming Objects with `Consumer<T>`

Implementing and Using Consumers

The second interface is the `Consumer<T>` interface. A consumer does the opposite of the supplier: it takes an argument and does not return anything.

The interface is a little more complex: there are default methods in it, which will be covered later in this tutorial. Let us concentrate on its abstract method:

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    // default methods removed
}
```

You already implemented consumers:

```
Consumer<String> printer = s -> System.out.println(s);
```

You can update the previous example with this consumer:

```
for (int i = 0; i < 5; i++) {
    int nextRandom = newRandom.getAsInt();
    printer.accept("next random = " + nextRandom);
}
```

Using Specialized Consumers

Suppose you need to print integers. Then you can write the following consumer:

```
Consumer<Integer> printer = i -> System.out.println(i);`
```

Then you may face the same auto-boxing issue as with the supplier example. Is this boxing / unboxing acceptable in your application, performance-wise?

Do not worry if this is not the case, the JDK has you covered with the three specialized consumers available: `IntConsumer`, `LongConsumer`, and `DoubleConsumer`. The abstract methods of these three consumers follow the same convention as for the supplier, since the returned type is always `void`, they are all named `accept`.

Consuming Two Elements with a BiConsumer

Then the JDK adds another variant of the `Consumer<T>` interface, which takes two arguments instead of one, called quite naturally the `BiConsumer<T, U>` interface. Here is this interface:

```
@FunctionalInterface
public interface BiConsumer<T, U> {

    void accept(T t, U u);

    // default methods removed
}
```

Here is an example of a biconsumer:

```
BiConsumer<Random, Integer> randomNumberPrinter =
        (random, number) -> {
            for (int i = 0; i < number; i++) {
                System.out.println("next random = " + random.nextInt());
            }
        };
```

You can use this biconsumer to write the previous example differently:

```
randomNumberPrinter.accept(new Random(314L), 5));
```

There are three specialized versions of the `BiConsumer<T, U>` interface to handle primitive types: `ObjIntConsumer<T>`, `ObjLongConsumer<T>` and `ObjDoubleConsumer<T>`.

Passing a Consumer to an Iterable

Several important methods have been added to the interfaces of the Collections Framework, that are covered in another part of this tutorial. One of them takes a `Consumer<T>` as an argument and is extremely useful: the `Iterable.forEach()` method. Here is a simple example, that you will see everywhere:

```
List<String> strings = ...; // really any list of any kind of objects
Consumer<String> printer = s -> System.out.println(s);
strings.forEach(printer);
```

This last line of code will just apply the consumer to all the objects of the list. Here it will simply print them one by one on the console. You will see another way to write this consumer in a later part.

This `forEach()` method exposes a way to access an internal iteration over all the elements of any `Iterable`, passing the action you need to take on each of these elements. It is a very powerful way of doing so, and it also makes your code more readable.

Testing Objects with `Predicate<T>`

Implementing and Using Predicates

The third interface is the `Predicate<T>` interface. A predicate is used to test an object. It is used for filtering streams in the Stream API, a topic that you will see later on.

Its abstract method takes an object and returns a boolean value. This interface is again a little more complex than `Consumer<T>`: there are default methods and static methods defined on it, which you will see later on. Let us concentrate on its abstract method:

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    // default and static methods removed
}
```

You already saw an example of a `Predicate<String>` in a previous part:

```
Predicate<String> length3 = s -> s.length() == 3;
```

To test a given string, all you need to do is call the `test()` method of the `Predicate` interface:

```
String word = ...; // any word
boolean isOfLength3 = length3.test(word);
System.out.prinln("Is of length 3? " + isOfLength3);
```

Using Specialized Predicates

Suppose you need to test integer values. You can write the following predicate:

```
Predicate<Integer> isGreaterThan10 = i -> i > 10;
```

The same goes for the consumers, the supplier, and this predicate. What this predicate takes as an argument is a reference to an instance of the `Integer` class, so before comparing this value to 10, this object is auto-unboxed. It is very convenient but comes with an overhead.

The solution provided by the JDK is the same as for suppliers and consumers: specialized predicates. Along with `Predicate<String>` are three specialized interfaces: `IntPredicate`, `LongPredicate`, and `DoublePredicate`. Their abstract methods all follow the naming convention. Since they all return a `boolean`, they are just named `test()` and take an argument corresponding to the interface.

So you can write the previous example as follow:

```
IntPredicate isGreaterThan10 = i -> i > 10;
```

You can see that the syntax of the lambda itself is the same, the only difference is that `i` is now an `int` type instead of `Integer`.

Testing Two Elements with a BiPredicate

Following the convention you saw with the `Consumer<T>`, the JDK also adds a `BiPredicate<T, U>` interface, which tests two elements instead of one. Here is the interface:

```
@FunctionalInterface
public interface BiPredicate<T, U> {

    boolean test(T t, U u);

    // default methods removed
}
```

Here is an example of such a bipredicate:

```
Predicate<String, Integer> isOfLength = (word, length) -> word.length() == length;
```

You can use this bipredicate with the following pattern:

```
String word = ...; // really any word will do!
int length = 3;
boolean isWordOfLength3 = isOfLength.test(word, length);
```

There is no specialized version of `BiPredicate<T, U>` to handle primitive types.

Passing a Predicate to a Collection

One of the methods added to the Collections Framework takes a predicate: the `removeIf()` method. This method uses this predicate to test each element of the collection. If the result of the test is `true`, then this element is removed from the collection.

You can see this pattern in action in the following example:

```
List<String> immutableStrings =
        List.of("one", "two", "three", "four", "five");
List<String> strings = new ArrayList<>(immutableStrings);
Predicate<String> isEvenLength = s -> s.length() % 2 == 0;
strings.removeIf(isEvenLength);
System.out.println("strings = " + strings);
```

Running this code will produce the following result:

```
strings = [one, two, three]
```

There are several things worth pointing out on this example:

- As you can see, calling `removeIf()` mutates this collection.
- So you should not call `removeIf()` on an immutable collection, like the ones produced by the `List.of()` factory methods. You will get an exception if you do that because you cannot remove elements from an immutable collection.
- `Arrays.asList()` produces a collection that behaves like an array. You can mutate its existing elements, but you are not allowed to add or remove elements from the list returned by this factory method. So calling `removeIf()` on this list will not work either.

Mapping Objects to Other Objects with `Function<T, R>`

Implementing and Using Functions

The fourth interface is the `Function<T, R>` interface. The abstract method of a function takes an object of type `T` and returns a transformation of that object to any other type `U`. This interface also has default and static methods.

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

```

```
    // default and static methods removed
}
```

Functions are used in the Stream API to map objects to other objects, a topic that will be covered later on. A predicate can be seen as a specialized type of function, that returns a `boolean`.

Using Specialized Functions

This is an example of a function that takes a string and returns the length of that string.

```
Function<String, Integer> toLength = s -> s.length();
String word = ...; // any kind of word will do
int length = toLength.apply(word);
```

Here again, you can spot the boxing and unboxing operations in action. First, the `length()` method returns an `int`. Since the function returns an `Integer`, this `int` is boxed. But then the result is assigned to a variable `length` of type `int`, so the `Integer` is then unboxed to be stored in this variable.

If performance is not an issue in your application, then this boxing and unboxing is really not a big deal. If it is, you will probably want to avoid it.

The JDK has solutions for you, with specialized versions of the `Function<T, R>` interface. This set of interfaces is more complex than the one we saw for the `Supplier`, the `Consumer<T>`, or the `Predicate` categories because specialized functions are defined both for the type of the input argument, and the returned type.

Both the input argument and the output can have four different types:

- a parameterized type `T`;
- an `int`;
- a `long`;
- a `double`.

Things do not stop here, because there is a subtlety in the design of the API. There is a special interface: `UnaryOperator<T>` which extends `Function<T, T>`. This unary operator concept is used to name the functions that take an argument of a given type, and return a result of the same type. A unary operator is just what you would expect. All the classical math operators can be modeled by a `UnaryOperator<T>`: the square root, all the trigonometric operators, the logarithm, and the exponential.

Here are the 16 specialized types of functions you can find in the `java.util.function` package.

Parameter types	**T**	**int**	**long**	**double**
T	UnaryOperator<T>	IntFunction<T>	LongFunction<T>	DoubleFunction<T>
int	ToIntFunction<T>	IntUnaryOperator	LongToIntFunction	DoubleToIntFunction
long	ToLongFunction<T>	IntToLongFunction	LongUnaryOperator	DoubleToLongFunction
double	ToDoubleFunction<T>	IntToDoubleFunction	LongToDoubleFunction	DoubleUnaryOperator

All the abstract methods of these interfaces follow the same convention: they are named after the returned type of that function. Here are their names:

- `apply()` for the functions that return a generic type `T`
- `applyAsInt()` if it returns the primitive type `int`
- `applyAsLong()` for `long`
- `applyAsDouble()` for `double`

Passing a Unary Operator to a List

You can transform the elements of a list with a `UnaryOperator<T>`. One could wonder why a `UnaryOperator<T>` and not a basic `Function`. The answer is in fact quite simple: once declared, you cannot change the type of a list. So the function you apply can change the elements of the list, but not their type.

The method that takes this unary operator passes it to the `replaceAll()` method. Here is an example:

```
List<String> strings = Arrays.asList("one", "two", "three");
UnaryOperator<String> toUpperCase = word -> word.toUpperCase();
strings.replaceAll(toUpperCase);
System.out.println(strings);
```

Running this code displays the following:

```
[ONE, TWO, THREE]
```

Note that this time we used a list created with the `Arrays.asList()` pattern. Indeed you do not need to add or remove any element to that list: this code just modifies each element one by one, which is possible with this particular list.

Mapping Two Elements with a BiFunction

As for the consumer and predicate, functions have also a version that takes two arguments: the bifunction. The interface is `BiFunction<T, U, R>`, where `T` and `U` are the arguments and `R` the returned type. Here is the interface:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

    // default methods removed
}
```

You can create a bifunction with a lambda expression:

```
BiFunction<String, String, Integer> findWordInSentence =
    (word, sentence) -> sentence.indexOf(word);
```

The `UnaryOperator<T>` interface has also a sibling interface with two arguments: the `BinaryOperator<T>`, that extends `BiFunction<T, U, R>`. As you would expect, the four basic arithmetic operations can be modeled with a `BinaryOperator`.

A subset of all the possible specialized versions of bifunction has been added to the JDK:

- `IntBinaryOperator`, `LongBinaryOperator` and `DoubleBinaryOperator`;
- `ToIntBiFunction<T>`, `ToLongBiFunction<T>` and `ToDoubleBiFunction<T>`.

Wrapping up the Four Categories of Functional Interfaces

The `java.util.function` package is now central in Java, because all the lambda expressions you are going to use in the Collections Framework or the Stream API implement one of the interfaces from that package.

As you saw, this package contains many interfaces and finding your way there may be tricky.

Firstly, what you need to remember is that there are 4 categories of interfaces:

- the suppliers: do not take any argument, return something
- the consumers: take an argument, do not return anything
- the predicates: take an argument, return a boolean
- the functions: take an argument, return something

Secondly: some interfaces have versions that take two arguments instead of one:

- the biconsumers
- the bipredicates
- the bifunctions

Thirdly: some interfaces have specialized versions, added to avoid boxing and unboxing. There are too many to list them all. They are named after the type they take. For example: `IntPredicate`, or the type they return, as in `ToLongFunction<T>`. They may be named after both: `IntToDoubleFunction`.

Lastly: there are extensions of `Function<T, R>` and `BiFunction<T, U, R>` for the case where all the types are the same: `UnaryOperator<T>` and `BinaryOperator<T>`, with specialized versions for the primitive types.

More Learning

Last update: *October 26, 2021*

Writing Your First Lambda Expression ← **Using Lambdas Expressions in Your Application** → Writing Lambda Expressions as Method References

Home > Tutorials > Lambda Expressions > Using Lambdas Expressions in Your Application