Writing Lambda Expressions as Method References

You saw that a lambda expression is in fact the implementation of a method: the only abstract method of a functional interface. Sometimes people call these lambda expressions "anonymous methods", since it is just that: a method that has no name, and that you can move around your application, store in a field or a variable, pass as an argument to a method or a constructor and return from a method.

Sometimes you will be writing lambda expressions that are just calls to specific methods defined somewhere else. And in fact, you already did that when you wrote the following code:

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

Written like that, this lambda expression is just a reference to the `println()` method defined on the `System.out` object.

This is where the *method reference* syntax comes in.

Your First Method Reference

Sometimes a lambda expression is just a reference to an existing method. In that case you can write it is as a *method reference*. The previous code then becomes the following:

```
Consumer<String> printer = System.out::println;
```

There are four categories of method references:

- Static method references
- Bound method references
- Unbound method references
- Constructor method references

The `printer` consumer belongs to the unbound method references category.

> *Most of the time your IDE will be able to tell you if a particular lambda expression may be written as a method reference. Do not hesitate to ask it!*

Writing Static Method References

Suppose you have the following code:

```
DoubleUnaryOperator sqrt = a -> Math.sqrt(a);
```

This lambda expression is in fact a reference to the static method `Math.sqrt()`. It can be written in that way:

```
DoubleUnaryOperator sqrt = Math::sqrt;
```

This particular method reference refers to a static method and is thus called a *static method reference*. The general syntax of a static method reference is `RefType::staticMethod`.

A static method reference may take more than one argument. Consider the following code:

```
IntBinaryOperator max = (a, b) -> Integer.max(a, b);
```

You can rewrite it with a method reference:

```
IntBinaryOperator max = Integer::max;
```

Writing Unbound Method References

Methods That Do Not Take Any Argument

Suppose you have the following code:

```
Function<String, Integer> toLength = s -> s.length();
```

This function could be written as a `ToIntFunction<T>`. It is just a reference to the method `length()` of the class `String`. So you can write it as a method reference:

```
Function<String, Integer> toLength = String::length;
```

This syntax may be confusing at first since it really looks like a static call. But in fact it is not: the `length()` method is an instance method of the `String` class.

You can call any getter from a plain Java bean with such a method reference. Suppose you have `User` class with a `getName()` defined on it. You can then write the following function:

```java
Function<User, String> getName = user -> user.getName();
```

as the following method reference:

```java
Function<String, Integer> getName = User::getName;
```

Methods That Do Not Take Any Argument

Here is another example that you already saw:

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

This lambda is in fact a reference to the `indexOf()` method of the `String` class, and can thus be written as the following method reference:

```java
BiFunction<String, String, Integer> indexOf = String::indexOf;
```

This syntax may look more confusing than the easier examples `String::length` or `User::getName` that are pretty straightforward. A good way to mentally reconstruct the lambda written in the classical way is to check the type of this method reference. That will give you the arguments this lambda is taking.

The general syntax of a unbound method reference is the following: `RefType:instanceMethod`, where `RefType` is the name of a type, and `instanceMethod` is the name of an instance method.

Writing Bound Method References

The first example of a method reference you saw was the following:

```java
Consumer<String> printer = System.out::println;
```

This method reference is called a *bound method reference*. This method reference is called *bound* because the object on which the method is called is defined in the method reference itself. So this call is *bound* to the object given in the method reference.

If you consider the *unbound* syntax: `Person::getName`, you can see that the object on which the method is called is not part of this syntax: it is provided as an argument of the lambda expression. Consider the following code:

```java
Function<User, String> getName = User::getName;
User anna = new User("Anna");
String name = getName.apply(anna);
```

You can see that the function is applied to a specific instance of `User`, that is passed to the function. This function then operates on that instance.

This is not the case in the previous consumer example: the `println()` method is called on the `System.out` object, that is part of the method reference.

The general syntax of a bound method reference is the following: `expr:instanceMethod`, where `expr` is an expression that returns an object, and `instanceMethod` is the name of an instance method.

Writing Constructor Method References

The last type of method reference you need to know is the *constructor method reference*. Suppose you have the following `Supplier<List<String>>`:

```
Supplier<List<String>> newListOfStrings = () -> new ArrayList<>();
```

You can see that in the same way as the rest: this boils down to be a reference on the empty constructor of `ArrayList`. Well, method reference can do that. But since a constructor is not a method, this is another category of method references. The syntax is the following:

```
Supplier<List<String>> newListOfStrings = ArrayList::new;
```

You can notice that the diamond operator is not needed here. If you want to put it, then you need to also provide the type:

```
Supplier<List<String>> newListOfStrings = ArrayList<String>::new;
```

You need to be aware of the fact that if you do not know the type of a method reference, then you cannot tell what it does exactly. Here is an example:

```
Supplier<List<String>> newListOfStrings = () -> new ArrayList<>();
Function<Integer, List<String>> newListOfNStrings = size -> new ArrayList<>(size);
```

Both variables `newListOfStrings` and `newListOfNStrings` can be written with the same syntax `ArrayList::new`, but they do not refer to the same constructor. You just need to be careful with that.

Wrapping Up Method References

Here are the four types of method references.

Name	Syntax	Lambda equivalent
Static	`RefType::staticMethod`	`(args) -> RefType.staticMethod(args)`
Bound	`expr::instanceMethod`	`(args) -> expr.instanceMethod(args)`

Name	Syntax	Lambda equivalent
Unbound	`RefType::instanceMethod`	`(arg0, rest) -> arg0.instanceMethod(rest)`
Constructor	`ClassName::new`	`(args) -> new ClassName(args)`

Last update: *October 26, 2021*

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

Home > Tutorials > Lambda Expressions > Writing Lambda Expressions as Method References