Published Aug 12, 2016 Esteban Herrera

Introduction

In Java, we can use references to objects, either by creating new objects:

```
List list = new ArrayList();
store(new ArrayList());
```

Or by using existing objects:

```
List list2 = list
isFull(list2)
```

But what about a reference to a method?

If we only use a method of an object in another method, we still have to pass the full object as an argument. Wouldn't it be more practical to just pass the method as an argument? For example:

```
isFull(list.size)
```

In Java 8, thanks to lambda expressions, we can do something like this. We can use methods as if they were objects, or primitive values.

Java 8 Method Reference

A method reference is the shorthand syntax for a lambda expression that executes just **ONE** method. Here's the general syntax of a method reference:

```
Object :: methodName
```

We know that we can use lambda expressions instead of using an anonymous class. But sometimes, the lambda expression is really just a call to some method, for example:

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

To make the code clearer, you can turn that lambda expression into a method reference:

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

In a method reference, you place the object (or class) that contains the method before the `::` operator and the name of the method after it without arguments.

But you may be thinking:

- How is this clearer?
- What will happen to the arguments?
- How can this be a valid expression?
- I don't understand how to construct a valid method reference...

First of all, a method reference can't be used for any method. **They can only be used to replace a single-method lambda expression.**

So to use a method reference, you first need a lambda expression with one method. And to use a lambda expression, you first need a functional interface, an interface with just one abstract method.

In other words:

Instead of using AN ANONYMOUS CLASS
you can use A LAMBDA EXPRESSION
And if this just calls one method, you can use A METHOD REFERENCE

There are four types of method references:

- A method reference to a *static method*.
- A method reference to an *instance method of an object of a particular type*.
- A method reference to an *instance method of an existing object*.
- A method reference to a *constructor*.

Let's begin by explaining the most natural case, a *static method*.

Static method reference

In this case, we have a lambda expression like the one below:

```
(args) -> Class.staticMethod(args)
```

This can be turned into the following method reference:

```
Class::staticMethod
```

Notice that between a static method and a static method reference, instead of the `.` operator, we use the `::` operator, and that we don't pass arguments to the method reference.

In general, we don't have to pass arguments to method references. However, arguments are treated depending on the type of method reference.

In this case, any arguments (if any) taken by the method are passed automatically behind the curtains.

Where ever we can pass a lambda expression that just calls a static method, we can use a method reference. For example, assuming this class:

```
class Numbers {
  public static boolean isMoreThanFifty(int n1, int n2) {
    return (n1 + n2) > 50;
  }
  public static List<Integer> findNumbers(
    List<Integer> l, BiPredicate<Integer, Integer> p) {
      List<Integer> newList = new ArrayList<>();
      for(Integer i : l) {
        if(p.test(i, i + 10)) {
          newList.add(i);
        }
      }
      return newList;
  }
}
```

We can call the `findNumbers()` method:

```
List<Integer> list = Arrays.asList(12,5,45,18,33,24,40);
```

```
findNumbers(list, new BiPredicate<Integer, Integer>() {
  public boolean test(Integer i1, Integer i2) {
    return Numbers.isMoreThanFifty(i1, i2);
  }
});


findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));


findNumbers(list, Numbers::isMoreThanFifty);
```

Instance method reference of an object of a particular type

In this case, we have a lambda expression like the following:

```
(obj, args) -> obj.instanceMethod(args)
```

Where an instance of an object is passed, and one of its methods is executed with some optional(s) parameter(s).

This can be turned into the following method reference:

```
ObjectType::instanceMethod
```

This time, the conversion is not that straightforward. First, in the method reference, we don't use the instance itself—we use its type.

Second, the other argument of the lambda expression, if any, is not used in the method reference, but it's passed behind the curtains like in the static method case.

For example, assuming this class:

```
class Shipment {
  public double calculateWeight() {
    double weight = 0;

    return weight;
  }
}
```

And this method:

```
public List<Double> calculateOnShipments(
  List<Shipment> l, Function<Shipment, Double> f) {
    List<Double> results = new ArrayList<>();
    for(Shipment s : l) {
      results.add(f.apply(s));
    }
    return results;
}
```

We can call that method using:

```
List<Shipment> l = new ArrayList<Shipment>();
```

```
calculateOnShipments(l, new Function<Shipment, Double>() {
  public Double apply(Shipment s) {
    return s.calculateWeight();
  }
});


calculateOnShipments(l, s -> s.calculateWeight());


calculateOnShipments(l, Shipment::calculateWeight);
```

In this example, we don't pass any arguments to the method. The key point here is that an instance of the object is the parameter of the lambda expression, and we form the reference to the instance method with the type of the instance.

Here's another example where we pass two arguments to the method reference.

Java has a `Function` interface that takes one parameter, a `BiFunction` that takes two parameters, but there's no `TriFunction` that takes three parameters, so let's make one:

```
interface TriFunction<T, U, V, R> {
  R apply(T t, U u, V v);
}
```

Now assume a class with a method that takes two parameters a return a result, like this:

```
class Sum {
  Integer doSum(String s1, String s2) {
    return Integer.parseInt(s1) + Integer.parseInt(s1);
  }
}
```

We can wrap the `doSum()` method within a `TriFunction` implementation by using an anonymous class:

```
TriFunction<Sum, String, String, Integer> anon =
  new TriFunction<Sum, String, String, Integer>() {
    @Override
    public Integer apply(Sum s, String arg1, String arg2) {
      return s.doSum(arg1, arg1);
    }
};
System.out.println(anon.apply(new Sum(), "1", "4"));
```

Or by using a lambda expression:

```
riFunction<Sum, String, String, Integer> lambda =
  (Sum s, String arg1, String arg2) -> s.doSum(arg1, arg1);
System.out.println(lambda.apply(new Sum(), "1", "4"));
```

Or just by using a method reference:

```
TriFunction<Sum, String, String, Integer> mRef = Sum::doSum;
System.out.println(mRef.apply(new Sum(), "1", "4"));
```

Here:

- The first type parameter of `TriFunction` is the object type that contains the method to execute.
- The second type parameter of `TriFunction` is the type of the first parameter.
- The third type parameter of `TriFunction` is the type of the second parameter.
- The last type parameter of `TriFunction` is the return type of the method to execute. Notice how this is omitted (inferred) in the lambda expression and the method reference.
- It may seem odd to just see the interface, the class, and how they are used with a method reference; but this becomes more evident when you see the anonymous class or even the lambda version.

From:

```
(Sum s, String arg1, String arg2) -> s.doSum(arg1, arg1)
```

To

```
Sum::doSum
```

Instance method reference of an existing object

In this case, we have a lambda expression like the following:

```
(args) -> obj.instanceMethod(args)
```

This can be turned into the following method reference:

```
obj::instanceMethod
```

This time, an instance defined somewhere else is used, and the arguments (if any) are passed behind the curtains like in the static method case.

For example, assuming these classes:

```
class Car {
  private int id;
  private String color;



}
class Mechanic {
  public void fix(Car c) {
    System.out.println("Fixing car " + c.getId());
  }
}
```

And this method:

```
public void execute(Car car, Consumer<Car> c) {
  c.accept(car);
}
```

We can call the method above using:

```
final Mechanic mechanic = new Mechanic()
Car car = new Car()
```

```
  public void accept(Car c) {
    mechanic.fix(c)
  }
})

// Using a lambda expression
execute(car, c -> mechanic.fix(c))

// Using a method reference
execute(car, mechanic::fix)
```

The key, in this case, is to use any object visible by an anonymous class/lambda expression and pass some arguments to an instance method of that object.

Here's another quick example using another `Consumer`:

```
Consumer<String> c = System.out::println;
c.accept("Hello");
```

Constructor method reference

In this case, we have a lambda expression like the following:

```
(args) -> new ClassName(args)
```

That can be turned into the following method reference:

```
ClassName::new
```

The only thing this lambda expression does is to create a new object and we just reference a constructor of the class with the keyword `new`. Like in the other cases, arguments (if any) are not passed in the method reference.

Most of the time, we can use this syntax with two (or three) interfaces of the `java.util.function` package.

If the constructor takes no arguments, a `Supplier` will do the job:

```
Supplier<List<String>> s = new Supplier() {
  public List<String> get() {
    return new ArrayList<String>();
  }
};
List<String> l = s.get();
```

```
Supplier<List<String>> s = () -> new ArrayList<String>();
List<String> l = s.get();
```

```
Supplier<List::new;
List<String> l = s.get();
```

If the constructor takes an argument, we can use the `Function` interface. For example:

```
Function<String, Integer> f =
```

```
    public Integer apply(String s) {
      return new Integer(s);
    }
};
Integer i = f.apply(100);
```

```
Function<String, Integer> f = s -> new Integer(s);
Integer i = f.apply(100);
```

```
Function<String, Integer> f = Integer::new;
Integer i = f.apply(100);
```

If the constructor takes two arguments, we use the `BiFunction` interface:

```
BiFunction<String, String, Locale> f = new BiFunction<String, String, Locale>() {
  public Locale apply(String lang, String country) {
    return new Locale(lang, country);
  }
};
Locale loc = f.apply("en","UK");
```

```
BiFunction<String, String, Locale> f = (lang, country) -> new Locale(lang, country);
Locale loc = f.apply("en","UK");
```

```
BiFunction<String, String, Locale> f = Locale::new;
Locale loc = f.apply("en","UK");
```

If you have a constructor with three or more arguments, you would have to create your own functional interface.

You can see that referencing a constructor is very similar to referencing a static method. The difference is that the constructor "method name" is `new`.

Conclusion

Many of the examples presented here are very simple and they probably don't justify the use of lambda expressions or method references.

As mentioned at the beginning, use method references if they make your code **CLEARER**.

For example, you can avoid the one method restriction by grouping all your code in a static method, and create a reference to that method instead of using a class or a lambda expression with many lines.

But the real power of lambda expressions and method references comes when they are combined with another new feature of Java 8; streams.

However, that will be the topic of another tutorial. Thanks for reading.

This tutorial was adapted from a chapter of my Java 8 Professional Programmer Study Guide.

Other Java tutorials you might find interesting:

- Serialization and Deserialization in Java