Trail: Learning the Java Language
Lesson: Classes and Objects
Section: Nested Classes
Subsection: Lambda Expressions

Method References

You use lambda expressions to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

Consider again the `Person` class discussed in the section Lambda Expressions:

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public Calendar getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }}
```

Suppose that the members of your social networking application are contained in an array, and you want to sort the array by age. You could use the following code (find the code excerpts described in this section in the example `MethodReferencesTest`):

```
Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);

class PersonAgeComparator implements Comparator<Person> {
    public int compare(Person a, Person b) {
        return a.getBirthday().compareTo(b.getBirthday());
    }
}

Arrays.sort(rosterAsArray, new PersonAgeComparator());
```

The method signature of this invocation of `sort` is the following:

```
static <T> void sort(T[] a, Comparator<? super T> c)
```

Notice that the interface `Comparator` is a functional interface. Therefore, you could use a lambda expression instead of defining and then creating a new instance of a class that implements `Comparator`:

```
Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);
```

However, this method to compare the birth dates of two `Person` instances already exists as `Person.compareByAge`. You can invoke this method instead in the body of the lambda expression:

```
Arrays.sort(rosterAsArray,
    (a, b) -> Person.compareByAge(a, b)
);
```

Because this lambda expression invokes an existing method, you can use a method reference instead of a lambda expression:

```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

The method reference `Person::compareByAge` is semantically the same as the lambda expression `(a, b) -> Person.compareByAge(a, b)`. Each has the following characteristics:

- Its formal parameter list is copied from `Comparator<Person>.compare`, which is `(Person, Person)`.
- Its body calls the method `Person.compareByAge`.

Kinds of Method References

There are four kinds of method references:

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`ContainingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

Reference to a Static Method

The method reference `Person::compareByAge` is a reference to a static method.

Reference to an Instance Method of a Particular Object

The following is an example of a reference to an instance method of a particular object:

```
class ComparisonProvider {
    public int compareByName(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }

    public int compareByAge(Person a, Person b) {
        return a.getBirthday().compareTo(b.getBirthday());
    }
}
ComparisonProvider myComparisonProvider = new ComparisonProvider();
Arrays.sort(rosterAsArray, myComparisonProvider::compareByName);
```

The method reference `myComparisonProvider::compareByName` invokes the method `compareByName` that is part of the object `myComparisonProvider`. The JRE infers the method type arguments, which in this case are `(Person, Person)`.

Reference to an Instance Method of an Arbitrary Object of a Particular Type

The following is an example of a reference to an instance method of an arbitrary object of a particular type:

```
String[] stringArray = { "Barbara", "James", "Mary", "John",
    "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

The equivalent lambda expression for the method reference `String::compareToIgnoreCase` would have the formal parameter list `(String a, String b)`, where `a` and `b` are arbitrary names used to better describe this example. The method reference would invoke the method `a.compareToIgnoreCase(b)`.

Reference to a Constructor

You can reference a constructor in the same way as a static method by using the name `new`. The following method copies elements from one collection to another:

```
public static <T, SOURCE extends Collection<T>, DEST extends Collection<T>>
    DEST transferElements(
        SOURCE sourceCollection,
        Supplier<DEST> collectionFactory) {

        DEST result = collectionFactory.get();
        for (T t : sourceCollection) {
            result.add(t);
        }
        return result;
}
```

The functional interface `Supplier` contains one method `get` that takes no arguments and returns an object. Consequently, you can invoke the method `transferElements` with a lambda expression as follows:

```
Set<Person> rosterSetLambda =
    transferElements(roster, () -> { return new HashSet<>(); });
```

You can use a constructor reference in place of the lambda expression as follows:

```
Set<Person> rosterSet = transferElements(roster, HashSet::new);
```

The Java compiler infers that you want to create a `HashSet` collection that contains elements of type `Person`. Alternatively, you can specify this as follows:

```
Set<Person> rosterSet = transferElements(roster, HashSet<Person>::new);
```

Problems with the examples? Try Compiling and Running the Examples: FAQs.
Complaints? Compliments? Suggestions? Give us your feedback.

Previous page: Lambda Expressions
Next page: When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions