

CS 213 – Software Methodology

Spring 2023
Sesh Venugopal

Feb 20

Lambda Expressions – Part 2

Method References

Method References

Consider a `consume` method with a `Consumer` parameter:

```
// consuming method
public static <T> void consume(List<T> list, Consumer<T> cons) {
    for (T t: list) { cons.accept(t); } }
}
```

Here's a call to this method, with a lambda for the `Consumer` argument:

```
// call to consuming method
List<Integer> list = Arrays.asList(2,3,16,8,-10,15,5,13);
consume(list, i -> System.out.println(i));
```

Alternatively, we can pass a `method reference` to `System.out.println`:

```
// passing method reference
consume(list, System.out::println );
```

A method reference is a lambda written with a `::` and method name, instead of an actual call to the method with parameters

Method References

A method reference is a way to rewrite a lambda to pass just the name of a method, instead of an actual call to it

```
// consuming method
public static <T> void consume(List<T> list, Consumer<T> cons) {
    for (T t: list) { cons.accept(t); } }

// passing method reference
consume(list, System.out::println);
```

`System.out.println` accepts an argument and does not return a value

So it can work like a `java.util.function.Consumer` function, and in the `accept` method, each item in the list will be passed in as argument to `System.out.println`

Method Reference: Instance Method

Recall the earlier example of a map method that took a `java.util.function.Function` as parameter:

```
public static <T,R> List<R>
map(List<T> list, Function<T,R> f) {
    List<R> result = new ArrayList();
    for (T t: list) { result.add(f.apply(t));}
    return result;
}
```

It was used to map color names to their lengths like this:

```
// map color names to their lengths
List<Integer> lengths = map(colors, s -> s.length());
```

`length()` is
an instance method
of `String`



The lambda can be simplified by using a method reference instead:

```
// map color names to their lengths
List<Integer> lengths = map(colors, String::length);
```

Instance Method Reference: Example 2

```
class Student {  
    ...  
    public boolean  
    isSenior() { ... }  
}
```

```
public static List<T>  
filter(List<T> list, Predicate<T> p) {  
    List<T> result = new ArrayList<T>();  
    for (T t: list) {  
        if (p.test(t)) {  
            result.add(t);  
        }  
    }  
    return result;  
}
```

```
List<Student> students = new ArrayList<Student>();  
... // populate list  
  
// filter seniors using method reference  
System.out.println(filter(students, Student::isSenior));
```



equivalent to

```
s -> s.isSenior()
```

Method Reference Example: Sorting

Say we want to sort the students list by year

`java.util.Comparator` is a functional interface with a single abstract `compare` method

Version 1: Write a named `Comparator` class and pass an instance

```
class Student {  
    public static final int FRESHMAN=1;  
    public static final int SOPHOMORE=2;  
    public static final int JUNIOR=3;  
    public static final int SENIOR=4;  
    private int year;  
    ...  
    public int getYear() {  
        return year;  
    }  
}
```

```
class YearComparator  
implements Comparator<Student> {  
    public int compare(  
        Student s1, Student s2) {  
        return s1.getYear() -  
            s2.getYear();  
    }  
}
```

```
// sort with instance of YearComparator  
students.sort(new YearComparator());
```

`java.util.List` interface has a default `sort` method that takes a `Comparator` argument

Method Reference Example: Sorting

Version 2: Pass an instance of an anonymous `Comparator` implementation

```
// sort with instance of anonymous YearComparator implementation
students.sort(new Comparator<Student>() {
    public int compare(Student s1, Student s2) {
        return s1.getYear() - s2.getYear();
    }
});
```

Version 3: Pass a lambda

```
students.sort((s1,s2) -> s1.getYear() - s2.getYear());
```


Method Reference Example: Sorting

Version 4: Use lambda with `comparing` method of `Comparator`

```
students.sort(comparing(s -> s.getYear()));
```


static method
of `Comparator`


function that extracts
key from type of objects
to be compared

`comparing` method returns a `Comparator` instance
that uses key extracted by given function (e.g. `getYear()`)

Version 5: Use method reference with `comparing` method

```
students.sort(comparing(Student::getYear));
```

Code above requires:

```
import static java.util.Comparator.comparing;
```

static methods can
be imported!!

Constructor as Method Reference

```
class Student {  
    ...  
    public Student(int year, boolean commuter, String major) {...}  
    public Student(int year, String major) {...}  
    public Student(int year) {...}  
    public Student() {...}  
}
```

1. No-arg constructor used for `java.util.function.Supplier` instance

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

or

```
Supplier<Student> s = Student::new;  
Student student = s.get();  
  
Student student =  
    ((Supplier<Student>)Student::new).get();
```

2. 1-arg constructor used for `java.util.function.IntFunction` instance

```
IntFunction<Student> func = Student::new;  
Student student = func.apply(Student.SOPHOMORE);
```

Constructor as Method Reference

3. 2-arg constructor used for `java.util.function.BiFunction` instance

```
parameter, parameter, return  
BiFunction<Integer,String,Student> bifunc = Student::new;  
Student student = bifunc.apply(Student.SOPHOMORE,"CS");
```

Example: Generating a list of students, mapping from years to instances

```
static List<Student>  
generate(List<Integer> years, IntFunction<Student> func) {  
    List<Student> result = new ArrayList<Student>();  
    for (Integer i: years) {  
        result.add(func.apply(i));  
    }  
    return result;  
}
```

Call:

```
List<Student> students = generate(  
    Arrays.asList(Student.JUNIOR, Student.FRESHMAN, Student.SENIOR),  
    Student::new);
```

Composing Predicates and Functions

Composing Predicates

```
public static<T> List<T>
filter(List<T> list, Predicate<T> p) {
    List<T> result = new ArrayList<T>();
    for (T t: list) {
        if (p.test(t)) {
            result.add(t);
        }
    }
    return result;
}
```

```
Predicate<Student> cs_major = s -> s.getMajor().equals("CS");
```

```
Predicate<Student> senior = s -> s.getYear() == Student.SENIOR;
```

```
Predicate<Student> junior = s -> s.getYear() == Student.JUNIOR;
```

Composing Predicates

Predicates can be composed to make compound conditions:

```
filter(students,          // ? CS seniors
       cs_major.and(senior));
```

```
filter(students,          // ? CS juniors or seniors
       cs_major
       .and(junior.or(senior)));
```

```
filter(students,          // ? Students who are not
       cs_major            CS juniors or seniors
       .and(junior.or(senior))
       .negate());
```

```
filter(students,          // ? CS majors who are not
       cs_major            juniors or seniors
       .and((junior.or(senior))
       .negate()
       ));
```

Composing Functions

```
public static<T,R> List<R>
map(List<T> list, Function<T,R> f) {
    List<R> result = new ArrayList<R>();
    for (T t: list) {
        result.add(f.apply(t));
    }
    return result;
}
```

```
Function<Integer,Integer> f = i -> i*i;
```

```
Function<Integer,Integer> g = i -> i+2;
```

```
List<Integer> list = Arrays.asList(3,8,-10,15,5);
```

```
map(list, f.andThen(g));    // g(f(x)) = [11, 66, 102, 227, 27]
```

```
map(list, f.compose(g));    // f(g(x)) = [25, 100, 64, 289, 49]
```