

CS 213 – Software Methodology

Spring 2017

Sesh Venugopal

Lecture 10: Feb 16

Lambda Expressions – Part 2

Consumer Interface

The `java.util.function.Consumer` interface “consumes” its single argument, returning nothing

```
interface Consumer<T> {  
    void accept(T t);  
    ...  
}  
  
public static <T> void  
consume(List<T> list,  
        Consumer<T> cons) {  
    for (T t: list) {  
        cons.accept(t);  
    }  
}  
  
// print colors, capitalized  
consume(colors, s ->  
        System.out.println(  
            Character.toUpperCase(s.charAt(0)) +  
            s.substring(1)));
```

Syntactic Sugar with Method References

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

For example, here is a lambda passed to a method, to match a `Consumer` parameter

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

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

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

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

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: Static Method

There are three variations to method references.

The first variation is to pass a static method reference, as in the previous example of passing `System.out::println`

In general, if a class `X` has static method `staticM`, then this takes the form `X::staticM`

Method Reference: Instance Method

The second variation is to pass a reference to an instance method

For example, the previous example of mapping color names to their lengths can be rewritten as follows:

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

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

Method Reference:

Instance Method Example

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

```
public static List<Student>  
filter(List<Student> students,  
        Predicate<Student> p) {  
    List<Student> result =  
        new ArrayList<Student>();  
    for (Student student: students) {  
        if (p.test(student)) {  
            result.add(student);  
        }  
    }  
    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 students list by year

Can write various versions of passing comparison function

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;  
    ...  
    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

Version 5: Use method reference with `comparing` method

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

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

```
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:

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

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_majors = s -> s.getMajor().equals("CS");
```

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

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

Composing Predicates

Predicates can be composed to make compound conditions:

```
filter(students, // CS seniors
       cs_majors.and(seniors));
```

```
filter(students, // CS juniors or seniors
       cs_majors
       .and(juniors.or(seniors)));
```

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

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

Composing Functions

```
public static<T,R> List<R>
filter(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);
```

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

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