CS 213 – Software Methodology

Sesh Venugopal

Lambda Expressions – Part 2

Method References

Method References

Consider a `consume` method with a `java.util.function.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));
```

Instead, 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

```
// 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, which is exactly what the `Consumer`.`accept` method is supposed to do

So sending the method reference syntax as an argument is like aliasing `cons`.`accept` with `System.out.println` in the `consume` method code, it is as if you are replacing `cons`.`accept` with `System.out.println`

Method Reference: Static Method

There are three variations to method references.

The first variation is to pass a reference to a static method, as with `System.out::println` – `println` is a static method in `System.out`

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

Method Reference: Instance Method

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

Recall the earlier example of a mapmethod 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;
}
```

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

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());
```

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;
  ...
  public int getYear() {
     return year; // field in class
  }
}
```

```
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));
```

Code above requires:

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

static methods can be imported!!

Method Reference: Constructor

```
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 and Functions

Composing Predicates

```
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;
```

```
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;
}
```

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

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

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

```
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;
}
```

```
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]
```