CS 213 – Software Methdology

Spring 2016

Lecture 7/8: Feb 9/11

Interfaces – Part 3

Lambda Expressions

Example: Array List Filtering

Pick even numbers out of an array list

```
List<Integer> result =
      new ArrayList<Integer>();
for (Integer i: list) {
    if (i % 2 == 0) {
       result.add(i);
    }
}
return result;
```

Pick numbers > 10 out of array list

```
List<Integer> result =
      new ArrayList<Integer>();
for (Integer i: list) {
    if (i > 10) {
       result.add(i);
    }
}
return result;
```

There may be other conditions for filtering numbers out of this array list that an application may need to use elsewhere in the code (e.g. pick odd numbers, pick multiples of 5, etc.)

How to work this without having to write pretty much the same code every time a new filtering condition needs to be implemented?

Passing Behavior to Method

Setup: Have a method that takes as parameters the array list, *and a function*

Technically, there's no way to pass a method (function) as a parameter

But there are ways to pass a method through a very light object, with simple syntax that makes it appear as if we are just passing a function

Defining Filter Method to Accept Function

Start with defining an interface with a SINGLE method.
This makes it a *functional interface*

```
public interface IntPicker {
   boolean pick(int i);
}
```

Next, implement the filter method with an instance of the functional interface as the second parameter

```
public List<Integer>
filter( List<Integer> list, IntPicker picker) {
   List<Integer> result = new ArrayList<Integer>();
   for (Integer i: list) {
    if (picker.pick(i)) {
       result.add(i);
    }
}
return result;
```

Passing argument for function: v1

Named interface implementation

For each type of filter, make a named class that implements the interface:

```
public class EvenPicker
implements Picker {
   public boolean pick(int i) {
      return i % 2 == 0;
   }
}
```

```
public class GreaterThan10Picker
implements Picker {
   public boolean pick(int i) {
      return i > 10;
   }
}
```

Call the filter method:

```
List<Integer> list = Arrays.asList(2,3,16,8,-10,15,5,13);

List<Integer> evens = filter(list, new EvenPicker());

List<Integer> greaterThan10s = filter(list, new GreaterThan10Picker());
```

Passing argument for function: v2

Anonymous interface implementation

Write anonymous interface on the fly when calling the filter method:

```
List<Integer> list = Arrays.asList(2,3,16,8,-10,15,5,13);

List<Integer> evens = filter(list,
                             new IntPicker() {
                                 public boolean pick(int i) {
                                    return i % 2 == 0;
                                 }
                             });

List<Integer> greaterThan10s = filter(list,
                             new IntPicker() {
                                 public boolean pick(int i) {
                                    return i > 0;
                                 }
                             });
```

Passing argument for function: v3

Named Lambda Expression

A lambda expression can be used to define the method of a functional interface, in a simplified syntax:

```
IntPicker evenPicker = (int i) -> i % 2 == 0;
```

Since the method `pick` is defined to accept an `int` and return a `boolean`, the LHS of the expression is the `int` input, and the RHS is the `boolean` return

```
IntPicker greaterThan10Picker = (int i) -> i > 10;
```

Call the filter method:

```
List<Integer> list = Arrays.asList(2,3,16,8,-10,15,5,13);

List<Integer> evens = filter(list, evenPicker);

List<Integer> greaterThan10s = filter(list, greaterThan10Picker);
```

Passing argument for function: v4

On-the-fly Unnamed Lambda Expression

Call the filter method:

```
List<Integer> list = Arrays.asList(2,3,16,8,-10,15,5,13);

List<Integer> evens = filter(list, (int i) -> i % 2 == 0);

List<Integer> greaterThan10s = filter(list, (int i) -> i > 10);
```

Type of LHS var can be dropped since it can be unambiguously resolved:

```
List<Integer> evens = filter(list, i -> i % 2 == 0);

List<Integer> greaterThan10s = filter(list, i -> i > 10);
```

Lambda Expressions (or just lambdas)

A lambda expression gets compiled into an object that implements a *functional interface,* with types resolved according to context

```
List<Integer> evens = filter(list, i -> i % 2 == 0);
```

Because filter takes an instance of IntPicker as 2nd parameter, the matching lambda expression argument gets compiled to an instance of IntPicker

Because the method (name irrelevant) in the `IntPicker` functional Interface takes a single `int` parameter and returns a `boolean`, the LHS of the lambda is an `int` type var, and the RHS returns a `boolean`

Multiple statements in RHS must be in a braces-block:

```
x -> { x++; System.out.println(x); }
```

Extending filter method to work on some boolean test on ANY type

Want to make filter method work on ANY type, not just `int`, based on a boolean test

Java has a pre-defined interface for this very purpose, in the package `java.util.function`:

```
interface Predicate<T> {
    boolean test(T t);
    ...
}
```

(There are other methods in this interface, which are either `static` or `default`. So this is a functional interface because only the `test` method needs to be implemented)

Extending filter method to work on some boolean test on ANY type

```
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;
}
```

Calls for `Integer` list:

```
List<Integer> list =
   Arrays.asList(2,3,16,8,-10,15,5,13);
List<Integer> evens =
   filter(list, i -> i % 2 == 0);
List<Integer> greaterThan10s =
   filter(list, i -> i > 10);
```

Calls for `String` list:

```
List<String> colors =
   Arrays.asList(
   "red","green","orange","violet",
   "blue","white","yellow","indigo");
List<Integer> shortColors =
   filter(colors, s -> s.length() < 4);
List<Integer> longColors =
   filter(colors, s -> s.length() > 5);
```

Beyond Predicates: Applying Non-Boolean Functions

The `java.util.function.Function` interface helps with this:

```
interface Function<T,R> {
    R apply(T t);
    ...
}
```

```
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;
}
```

Calls:

```
// square all numbers in list
List<Integer> squares = filter(list, i -> i * i);

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

Consumer Interface

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

```
interface Consumer<T> {
    void consume(T t);
    ...
}
```

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

```
// print colors, capitalized
consume(colors, s ->
                  System.out.print(
                     Character.toUpperCase(s.charAt(0)) +
                     s.substring(1) + "   "));
```

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: Student Class Example

Method reference for seniors, applying a predicate for filtering:

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

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

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

equivalent to

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

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

```
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;
  }
}
```

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

Method Reference: Student Class 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();
                  }
              });
```

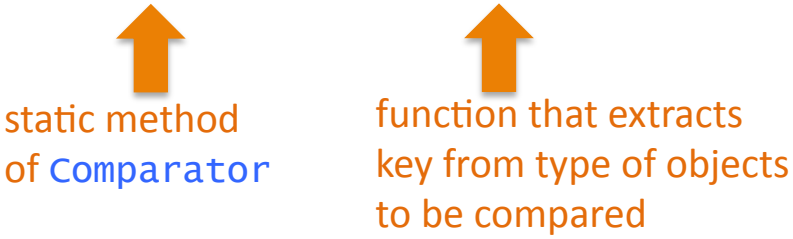
Version 3: Pass a lambda

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

Method Reference: Student Class Example - Sorting

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

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



`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.Function` 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]

List<Integer> list = Arrays.asList(3,8,-10,15,5);
filter(list, f.compose(g));  f(g(x)) = [25, 100, 64, 289, 49]
```