

CS 213 – Software Methodology

Spring 2019

Sesh Venugopal

Lecture 9: Feb 19

Lambda Expressions – Part 1

Example: Array List Filtering

Pick even numbers out of a 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 a list

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

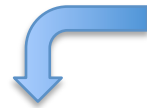
There may be other conditions for filtering numbers out of a list that an application may need to use elsewhere (e.g. pick odd numbers, pick multiples of 5, etc.)

How to redo this so that we can maintain a single scaffolding (loop through list and apply condition), and change ONLY the actual condition depending on requirement?

Passing Behavior to Method

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

`method(list, function)`



function to be
applied to each
member of the list

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

But, as of Java 8, there is a way to pass a method through a very light object, with simple syntax that *makes it appear as if we are just passing a function*

Define Behavior in Functional Interface

Start by defining an interface that has only ONE 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 function argument : v1

Named interface implementation

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

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

```
public class GreaterThan10Picker
implements IntPicker {
    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 function argument: 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 > 10;  
                                         }  
                                     });
```

Passing function argument: v3

Named Lambda Expression

A **lambda expression** is essentially a simplified syntax to define the method of a **functional interface**:

```
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 function argument: 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,  IntPicker  
                                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); }
```

Some Pre-Defined Functional Interfaces in `java.util.function`

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

Want to make boolean filter method work on ANY data type, not just `int`

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

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

← functional method (the single abstract method of the interface)

There are other methods in this interface, which are either `static` or `default`, that are not abstract (fully implemented). So this is a functional interface because a single method, `test`, is abstract.

Generalizing 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<String> shortColors =
    filter(colors, s -> s.length() < 4);
List<String> 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>  
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;  
}
```

Calls:

```
// square all numbers in list  
List<Integer> squares = map(list, i -> i * i);  
  
// map color names to their lengths  
??
```

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)));
```