

Recitation 4 Solution

Lambda Expressions

1. For each of the following expressions, tell whether it is valid or not. If valid, explain the reasoning. If not valid, explain why.

1. `() -> {}`

ANSWER

Valid. Corresponds to a method that takes no arguments, returns void, and has an empty body, e.g.

```
public void stuff() {}
```

2. `() -> "Hello"`

ANSWER

Valid. Corresponds to this:

```
public String stuff() { return "Hello"; }
```

3. `() -> { return "Goodbye"; }`

ANSWER

Valid. Corresponds to this, with an explicit return statement:

```
public String stuff() { return "Goodbye"; }
```

The explicit return, although correct, is unnecessary.

4. `(Integer i) -> return i+10;`

ANSWER

Invalid. Since `return` is a control flow statement, it has to be enclosed within braces.

5. `(String s) -> { "Bourne Ultimatum"; }`

ANSWER

Invalid. `"Bourne Ultimatum"` is an expression, not a statement. You can do either of the following to get a correct lambda expression:

- Move expression out of the braces:

```
(String s) -> "Bourne Ultimatum"
```

- Do a return statement:

```
(String s) -> { return "Bourne Ultimatum"; }
```

2. Which of the following are functional interfaces?

1.

```
public interface Sum1 {  
    int sum(int i, int j);  
}
```

ANSWER

Yes.

2.

```
public interface Sum2 extends Sum1 {  
    double sum(double i, double j);  
}
```

ANSWER

No. `Sum2` has two methods.

3.

```
public interface Rectangle {  
    double getWidth();  
    double getHeight();  
    default double area() {  
        return getWidth()*getHeight();  
    }  
}
```

ANSWER

No. There are two abstract methods.

-
3. Write a sample lambda for each of the following:

1. A boolean expression
2. Creating an object
3. Consuming from an object
4. Select/extract from an object
5. Combine two values
6. Compare two objects

For the questions involving objects, make up some class name - you don't have to write up the class. When parameters are required, specify the type(s).

SOLUTION

Requirement	Sample Lambda
A boolean expression	<code>(String s) -> s.length() > 0</code>
Creating an object	<code>() -> new Student()</code>
Consuming from an object	<code>(String s) -> { System.out.println(s.length()); }</code>
Select/extract from an object	<code>(String s) -> s.length()</code>
Combine two values	<code>(int i, int j) -> i*j</code>
Compare two objects	<code>(Student s1, Student s2) -> s1.getMajor().compareTo(s2.getMajor())</code>

-
4. Which of the following are valid uses of lambdas?

1.

```
public interface Executor {  
    void execute();  
}  
public void do(Executor ex) {  
    ex.execute();  
}  
do(() -> {});
```

ANSWER

Yes. The lambda takes no args and returns nothing, which matches the `execute` method of the `Executor` interface.

2.

```
public interface Proc<T> {
    T process();
}
public Proc<String> get() {
    return () -> "I am a go getter!";
}
```

ANSWER

Valid. The lambda in the return takes no args and a `String`, which matches the `process` method of the `Proc` interface, with the binding of `String` to the generic type `T`.

3.

```
Predicate<Student> p = (Student s) -> s.getMajor();
```

ANSWER

Invalid. The lambda should return a boolean.

4.

```
BiFunction<Integer,Integer,String> bif = (int i, int j) -> ""+i+j;
```

ANSWER

Invalid. The args for the lambda must be `Integers`. Auto conversion to `int` will not be done. (If you omit the data type for the arguments, it will work just fine.)

-
5. This question refers to the `Student` class presented in lecture (see Sakai -> Resources -> Feb 16 -> `Student.java`)

1. Write a NAMED lambda expression using a method reference to check if a student is a senior.

ANSWER

```
Predicate<Student> is_senior = STUDENT::isSenior;
```

2. Write a NAMED lambda expression using a method reference to get the major of a student.

ANSWER

```
Function<Student,String> major = STUDENT::getMajor;
```

3. Given the following filter method:

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

For each of the following, write one or more `Predicate` instances as NAMED lambda expressions that can be passed to the `filter` method to get the required set of students. (Note: when composing predicates,

you want to use named lambda expressions in the composition, otherwise the syntax gets unwieldy/unacceptable.)

1. All non-CS majors

ANSWER

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

2. All CS and Physics majors who are commuters

ANSWER

```
Predicate<Student> physics_major = s -> s.getMajor().equals("Physics");  
Predicate<Student> commuter = Student::getCommuter;  
Predicate<Student> pred = (cs_major.or(physics_major)).and(commuter);
```

3. Math seniors who are not commuters

ANSWER

```
Predicate<Student> math_major = s -> s.getMajor().equals("Math");  
Predicate<Student> pred = (math_major.and(is_senior)).and(commuter.negate());
```

4. Resident non-Math non-freshman students

ANSWER

```
Predicate<Student> is_freshman = Student::isFreshman;  
Predicate<Student> pred = commuter.negate().and(math_major.negate()).and(is_freshman.negate());
```