211lab08

Lab 08 – Anonymous Classes

Anonymous classes in Java are analogous to lambda-expressions in Scheme. The purpose of this lab is to practice writing such classes. We will use the Intermediate language level for this lab.

Lambda-expressions in Java

The abstract concept of a unary function can be modeled by the following interface.

```
interface Lambda {
    Object apply(Object arg);
}
```

An anonymous Lambda class has the following form

```
new Lambda() {
    Object apply(Object arg) {
        // concrete code to compute something and return the result
        // ...
    }
}
```

Since Object is the superclass of all concrete classes in Java, in practice, we often have to perform type-casting on the `arg` in order to perform the desired operation. More than often, we need to type-cast the returned result in order to make proper use of it.

Finger exercises:

Use DrJava to compile the above Lambda interface.
Now create and compile a new file containing the following class.

```
    class Lab8Exercises {
        static Lab8Exercises ONLY = new Lab8Exercises ();
        private Lab8Exercises () {}
    }
```

Add the following method to Lab8Exercises.

```
    Object doSomething() {
        return new Lambda() {
           Object apply(Object arg) {
                return (Integer)arg * (Integer)arg;}
        }.apply(5);
    }
```

(Yes, the code is hard to read because of the "ugly" syntax.) What does it do? Use the Interactions pane to test it out.

Now remove the type cast (Integer) on arg in the above code. Does it compile? Do you see why the type cast is necessary? OK, put back the type cast so that everything compile
Add the following method.

```
    int x_minus_y(int x, int y) {
        return (Integer)new Lambda() {
            Object apply(Object arg) {
                return (Integer)arg - y;}
        }.apply(x);
    }
```

What does it do?
Now remove the first Integer type cast that immediately follows the return in the above code. Does it compile? Put pack the type cast so that everything compiles.

The method in exercise 5 is an example of what is called "Curry-ing"(in honor of the mathematician Curry) where a function of two variables x, y is expressed a function of x whose value is a function of y. Any function of two variables, f (x, y), can be represented as a combination of two functions, each of which is a function of one variable. For example,

- suppose f(x, y) = x + y;
- let F be a function of one variable x, such that F(x) is a function G of one variable y, where G(y) {{ f(x, y) }} x + y;
- then F(x) (y) {{ x + y }} f(x, y);

In the same manner, any function of three variables can be represented as a combination of three functions, each of which is a function of one variable via currying.

What does this anonymous inner class do?

```
Lambda whatIsIt() {
   return new Lambda() {
        Object apply(Object s1) {
            return new Lambda() {
                Object apply(Object s2) {
                    return new Lambda() {
                        Object apply(Object s3) {
                            return s1 + " " + s2 + " " + s3;
                        }
                    };
                }
            };
        }
    };
}
```

Actually the above does not compile in the Intermediate language level. In full Java syntax, the code would like the following (and would compile):

```
public class Currying {
    Lambda whatIsIt() {
       return new Lambda() {
            public Object apply(final Object s1) {
                return new Lambda() {
                    public Object apply(final Object s2) {
                        return new Lambda() {
                            public Object apply(final Object s3) {
                                return s1 + " " + s2 + " " + s3;
                            }
                        };
                    }
                };
            }
        };
    }
}
```

IMPORTANT CONCEPT: CLOSURE

Notice in exercise #5 how the anonymous inner class `new Lambda` inside of `x_minus_y` is allowed to reference y in its computation? y is said to be in the closure of this anonymous inner class.

In functional programming, the closure of a function (lambda) consists of the function itself and the environment in which the function is defined. In Java, a function is replaced by a class. An inner class is only defined in the context of its outer object (and the outer object of the outer object, etc...). An inner class together with its nested sequence of outer objects in which the inner class is well-defined is the equivalent of the notion of closure in functional programming. Such a notion is extremely powerful. Just like knowing how to effectively use lambda expressions and higher order functions is key to writing powerful functional programs in Scheme, effective usage of anonymous inner classes is key to writing powerful OO programs in Java.

More Exercises

Here is the stub code for ObjectList and its concrete subclasses.

```
abstract class ObjectList {
  ObjectList cons(Object n) { return new ConsObjectList(n, this); }

  /** Applies f to each element in this list and returns the list containing
    * the results of this application. */
  abstract ObjectList map(Lambda f);

  /** Returns a String representation of this list as specified in each concrete subclass. */
  abstract String listString();

  /** Returns s String containing the elements of this, where each element is preceded by ' ' */
  abstract String listStringHelp();
}

class EmptyObjectList extends ObjectList {
  static EmptyObjectList ONLY = new EmptyObjectList();
  private EmptyObjectList() { }

  ObjectList map(Lambda f) { return null; /*to do */ }
  String listString() { return "()"; }
  String listStringHelp() { return ""; }
}



class ConsObjectList extends ObjectList {
  Object first;
  ObjectList rest;

  ObjectList map(Lambda f) { return null; /* ... rest.map(f) ...to do*/ }

  String listString() { return "(" + first + rest.listStringHelp() + ")"; }
  String listStringHelp() { return " " + first + rest.listStringHelp(); }
}
```

Implement the `map` method for `ObjectList` .
Add and implement the following method for class Lab8Exercises.

```
    /**
     * Assuming the list L contains integers, returns the list of booleans
     * whose n-th element is true if the number at the n-th element in L is
     * positive, false otherwise.
     */
    ObjectList checkPositive(ObjectList L) {
        return null; // to do
    }
```

Add and implement the following method for class Lab8Exercises.

```
    /**
     * Assuming the list L contains integers, returns the list whose n-th
     * element is the square of the n-th element L.
     */
    ObjectList squareList(ObjectList L) {
        return null; // to do
    }
```

Add and implement the following method for class Lab8Exercises.

```
    /**
     * Assuming the list L contains integers, returns the list of integers
     * whose n-th element is the lesser of the n-th element of L and upperBound
     */
    Object cutOff(ObjectList L, int upperBound) {
        return null; // to do
    }
```

Create and compile a file containing the following class

```
class Pair {
    Object key;
    Object val;
}
```

Now write the `lookup` method for `ObjectList` .

Add and implement the following method for class Lab8Exercises.

```
    /**
     * Assuming the list dict contains Pairs, returns the list of Objects whose
     * n-th element is the Pair in dict whose key is the n-th element of the
     * list L.
     */
    ObjectList translate(ObjectList L, ObjectList dict) {
        return null; // to do
    }
```

—

Access Permissions: (Please don't edit)

- Set ALLOWTOPICCHANGE = Main.TeachersComp211Group