In Java, we have seen and developed methods that took in no arguments or several arguments. But we have not seen any examples of methods that take in other methods as arguments. This is because Java does not allow this. Methods can only take objects and primitives as arguments. Scheme, on the other hand, does not have this restriction. As it would be very useful to somehow pass methods to methods, we need to come up with a way to get around this restriction. The solution is to design classes whose sole purpose is to hold the methods that we want to pass to other methods. In the object-oriented design literature, this technique is called the *command* pattern. Here we are using this pattern in a purely functional fashion. (Source: www.cs.rice.edu/~cork/teachjava/2002/notes/current/node52.html)

PART 1:

You will see examples of two methods, **mapIntToInt** and **orMapInt** that require some method be passed to them. When you have finished PART 1, you should understand how interfaces and concrete subclass are created to represent the method that needs to be passed as an argument to other methods.

The method **mapIntToInt** requires as an argument a method of type (int →int) while **orMapInt** requires a method of type (int → boolean). Since these two arguments are different types, this means that we will need two separate interfaces:

1) Open the Lab 7 – Part 1 project.

2) The method **mapIntToInt** belongs to the **AListInt** class. Its purpose is to use *this* (which is a list of int) and the passed class (which represents a method) to produce a new list of int. The new list of int that is produced depends <u>entirely</u> on the class (which represents a method) passed. For example, suppose we wanted to take a list of int and pass a method (via a class) that multiplies each int in the list by 3. The list (2, -5, 0, 8) would give us the new list (6, -15, 0, 24). Still another example would be to pass a method (via a class) that squares each int in the list. Our list example above would give us the new list (4, 25, 0, 64). There are many other examples like this that you can think of. For this lab, we will work with the two classes (which ultimately will represent methods) called **AddOne** and **AbsVal**. **AddOne** will represent the method that adds 1 to each member of the int list. **AbsVal** will represent the method that produces a similar list to the original, except each int is replaced by its absolute value.

3) Open only the interface called **IInt2Int**, the family of **AListInt** classes and the class called **AddOne** and **AbsVal**. Since **AddOne** and **AbsVal** will represent methods of type (int → int), we created an interface called **IInt2Int** [which you can think of as standing for "Interface for methods of type (int → int)"]. This interface will hold only one method called **apply**. Notice that **apply** is also of type (int → int). It is this method that

AddOne and **AbsVal** will implement differently to achieve the effect of "adding one" and "absolute value"**.** Think of **apply** as a place holder for any method of type (int → int).

In **AddOne,** we implement **IInt2Int** interface and write up the method **apply** as required by implementing **IInt2Int**. In this version of **apply**, we see the code that actually adds 1 to an integer. In **AbsVal,** everything is exactly the same as **AddOne** except the code for **apply**. This version of **apply** uses the Java built in method called **Math.abs** to return the absolute value of an int.

4) In the class **ConsInt**, the method **mapIntToInt** takes in an object of type **IInt2Int** (which will be either **AddOne** or **AbsVal**) and then uses the **apply** method to process recursively *this* (which is a **AListInt**) to get a new list which depends entirely on the argument of type **IInt2Int**. Let's take a closer look at the body of the **mapIntToInt**:

return **new ConsInt**(**i2i.*apply***(**this.first**), **this.rest.mapIntToInt(i2i)**);

i2i is the name of the object of type **IInt2Int** that was passed to the method **mapIntToInt**. It will be either **AddOne** or **AbsVal**. ***apply*** is a method in **i2i**, so **i2i.*apply*** is the **apply** method of class **AddOne** or **AbsVal** and (**this.first**) is the argument to that method. **this.rest.mapIntToInt(i2i)** is the recursive call to **mapIntToInt(i2i)** using *this.rest* and we put this altogether with **new ConsInt** because we want to return a list of integers.

5) In the class **EmptyInt**, the method **mapIntToInt** has only an empty list to work with, so the only sensible thing to return is *this* (the **EmptyInt** class).

6) Open the class called **AListIntTest**. In section A2 you will find an instance of the class **AddOne** called **Add1** and in section A3, you will find an instance of the class **AbsVal** called **AbsValue**. These are used in the test cases of section A5. Go ahead and run these two test cases.

7) You will now design a class called **SubOne** which will resemble **AddOne** except that its version of the method **apply** will return a number one less than its input. The outline of the class **SubOne** already exists, so you only need to replace the bogus code that is sitting in the **apply** method. Is the **SubOne** class already implementing **IIint2Int**? If not, then make it so.

8) Create an example of **SubOne** as indicated in section A4. Comment out the existing test cases of section A5 of the class **AListIntTest** and create 2 test cases for **Sub1** as indicated in A5. Go ahead and run these test cases to see if you get the correct results.

9) Leave open the family of **AListInt** classes and **AListIntTest** class. Close all others. Open the interface called **IInt2Bool** and the class called **IsEven**.

10) The method **orMapInt** also belongs to the **AListInt** class. Its purpose is to use *this* (which is a list of int) and the passed method (via the class) to produce a boolean. The boolean that is returned depends entirely on the class (which represents the method) that is passed. For example, suppose we have a method that inputs a int and returns true if the int is greater than 6 and false otherwise. Then our method **orMapInt** would use that method (via the class) and *this* to return true if ANY of the int are greater than 6. Since the method (via the class) that needs to be passed to **orMapInt** is of type (int → boolean), we develop an interface called **IInt2Bool** [which you can think of as standing for "Interface for methods of type (int → boolean)"]. As we have seen before, this interface will hold only one method called **apply**. Notice that **apply** is also of type (int → boolean). It is this method that **IsEven** will implement to achieve the effect of "checking for evenness"**.** Again, think of **apply** as a place holder for any method of type (int → boolean).

In **IsEven,** we implement **IInt2Bool** interface and write up the method **apply** as required by implementing **IInt2Bool**. In this version of **apply**, we see the code that actually checks if an int is even. This code just checks whether the reminder after division by two is equal to zero.

11) In the class **ConsInt**, the method **orMapInt** takes in an object of type **IInt2Bool** (which is **IsEven**) and then uses the **apply** method to process recursively *this* (which is a **AListInt**) to get a boolean which depends entirely on the argument of type **IInt2Bool**. Let's take a close look at the body of the **orMapInt**:

return **i2b.*apply***(**this.first**) **||** **this.rest.orMapInt(i2b)**);

i2b is the name of the object of type **IInt2Bool** that was passed to the method **orMapInt**. It will be **IsEven**. ***apply*** is a method in **i2b**, so **i2b.*apply*** is the **apply** method of class **IsEven** and (**this.first**) is the argument to that method. **this.rest.orMapInt(i2b)** is the recursive call to **orMapInt(i2b)** using *this.rest* and we put this altogether with || because we want to return a boolean if any **i2b.*apply***(**this.first**) is ever true.

12) In the class **EmptyInt**, the method **orMapInt** needs to return either true or false. Since **orMapInt** will return true if it finds at least one true, then the answer for **EmptyInt** should be one that does not affect any of the booleans returned recursively in the **ConsInt** list. So the value should be false. If this is not clear, then think about what the answer should be for the **EmptyInt** if the **ConsInt** consisted of all negative int and **i2b.apply** was the method that checked for non-negative int. The answer to **orMapInt** should be false because no int on this list is 0 or positive. But **orMapInt** will return true if the answer for **EmptyInt** is true because it takes only one true to make an *or* statement true.

13) Open the class called **AListIntTest**. In section B2 you will find an instance of the class **IsEven** called **AllEven.** This is used in the test cases of section B4. Go ahead and run these three test cases.

14) You will now design a class called **IsPositive** which will resemble **IsEven** except that its version of the method **apply** will return a boolean if any the int are greater than 0. The outline of the class **IsPositive** already exists, so you only need to replace the bogus code that is sitting in the **apply** method. Is the **IsPositive** class already implementing **IIint2Bool**? If not, then make it so.

15) Create an example of **IsPositive** as indicated in section B3. Comment out the existing test cases of section B4 of the class **AListIntTest** and create 3 test cases for **AnyPositive** as indicated in B4. Go ahead and run these test cases to see if you get the correct results.

PART 2:

We saw in PART 1 that before we could test **orMapInt** and **mapIntToInt** we needed to create instances of the **IsEven, AddOne** and **AbsVal** classes. But this in turn required the creation of the concrete classes **IsEven, AddOne** and **AbsVal** classes. Is there anyway to simplify this? When you have finished PART 2, you will see that we can indeed simplify this requirement of creating concrete classes by the use of *anonymous inner classes*.

16) Open the Lab 7 – Part 2 project.

17) You will notice that all of the previous classes are still the same EXCEPT that **IsEven, AddOne** and **AbsVal** classes are missing! If these classes are missing, how can we write something like:

```
AbsVal   AbsValue  = new AbsVal()
AddOne   Add1 = new AddOne()
IsEven   AnyEven = new IsEven()
```

in the **AListIntTest** class?

Before we can explain how to actually do this, we need to explain how we create instances of anonymous classes. Suppose we have an interface called **Command** with one method called **execute** of type (String → boolean) that does something with a string, but we don't have any concrete classes that actually implement **Command**. When we want to create an instance of an object of type **Command** where **execute** returns true if the given String has length less than 6, we do it as follows:

```
Command  IsLessThan6  =  new Command( ) {
                          boolean execute (String givenString) {
                             return (givenString.length( ) < 6);
                           }
                         }
```

Now you can you use **IsLessThan6** as if it was an object in a concrete class with some name that you had created, which extends the interface **Command**.

So how do we create an instance of **IInt2Int** that is just like **AbsVal**?

```
IInt2Int  AbsValue =  new IInt2Int( ) {
                            int apply (int number) {
                               return Math.abs(number);
                              }
                          }
```

Where did we get the code for the body of **apply**? From the method **apply** in the concrete class called **AbsVal** from PART 1! Go ahead write this up in A3 and run the test cases in A5 to see if you get the correct results.

18) In A2 and A4, write up instances of **IInt2Int** that are just like A2 and A4 of PART 1. Go ahead and run the test cases in A5 to see if you get the correct results.

19) In B2 and B3, write up instances of **IInt2Bool** that are just like B2 and B3 of PART 1. Go ahead and run the test cases in B4 to see if you get the correct results.

20) Would you believe that we can do even better than this? If we are not interested in reusing the anonymous class we create, then we can forgo giving it a name and use it directly in a test case:

```
actual (List4.mapIntToInt (new IInt2Int( ) {
              int apply (int number) {
                 return Math.abs(number);
              }
           }) );
```

21) Redo all your test cases from section A and section B so that all of your test cases look like number 20) above. This means commenting all anonymous inner classes that have names assigned to them [as in number 17) above].

22) Can you make up an instance of **IInt2Int** and **IInt2Bool** in which the body of **apply** is different that anything you have seen today? Here are some possible ideas for **IInt2Int**: doubling each number, reversing the sign of each number, cubing each number and then subtracting one. Some ideas for **IInt2Bool**: is each number in the list odd, is each number in the list a perfect square, is each number in the list 1 more than a multiple of 4?

23) Can you make up an interface called **IString2Bool** of type (String → boolean) and a method called **mapStringToBool** in a family of classes called **AListString** that is similar to what we have seen with **IInt2Bool**, **mapIntToBool** and the **AListInt** family of classes?