5 Abstracting with Function Objects

Goals

In this lab you will learn how to abstract over the functional behavior.

5.1 Abstracting with Function Objects

Download the files in *Lab5.zip*. The folder contains the files *ImageFile.java*, *ISelectImageFile.java*, *SmallImageFile.java*, *IListImageFile.java*, *MTListImageFile.java*, *ConsListImageFile.java*, and *ExamplesImageFile.java*.

Starting with partially defined classes and examples will give you the opportunity to focus on the new material and eliminate typing in what you already know. However, make sure you understand how the class is defined, what does the data represent, and how the examples were constructed.

Create a new **Project** *Lab5-su10* and import into it all of the given files. Also import *tester.jar* from the previous lab.

We will now practice the use of *function objects*. The only purpose for defining the class `SmallImageFile` is to implement one method that determines whether the given `ImageFile` object has the desired property (a predicate method). An instance of this class can then be used as an argument to a method that deals with `ImageFile`s.

1. Start with defining in the `ExamplesImageFile` class the missing tests for the class `SmallImageFile`.

2. Design the method `allSmallerThan40000` that determines whether all items in a list are smaller that 40000 pixels. The method should take an instance of the class `SmallImageFile` as an argument.

3. We now want to determine whether the name in the given `ImageFile` object is shorter than 4. Design the class `NameShorterThan4` that implements the `ISelectImageFile` interface with an appropriate predicate method.

   Make sure in the class `ExamplesImageFile` you define an instance of this class and test the method.

4. Design the method `allNamesShorterThan4` that determines whether all items in a list have a name that is shorter than 4 characters. The

method should take an instance of the class `NameShorterThan4` as an argument.

5. Design the method `allSuchImageFile` that that determines whether all items in a list satisfy the predicate defined by the `select` method of a given instance of the type `ISelectImageFile`. In the `ExamplesImageFile` class test this method by abstracting over the method `allSmallerThan40000` and the method `allNamesShorterThan4`.

6. Design the class `GivenKind` that implements the `ISelectImageFile` interface with a method that produces `true` for all `ImageFile`s that are of the given `kind`. The desired `kind` is given as a parameter to the constructor, and so is specified when a new instance of the class `GivenKind` is created.

   Hint: Add a field to represent the desired `kind` to the class `GivenKind`.

7. In the `ExamplesImageFile` class use the method `allSuch` and the class `GivenKind` to determine whether all files in a list are *jpg* files. This should be written as a test case for the method `allSuchImageFile`.

   Do it again, but now ask about the *giff* files.

8. If you have some time left, design the method `filterImageFile` that produces a list of all `ImageFile`s that satisfy the `ISelectImageFile` predicate. Test it with as many of your predicates as you can.

9. Follow the same steps as above to design the method `anySuchImageFile` that that determines whether there is an item a list that satisfies the predicate defined by the `select` method of a given instance of the type `ISelectImageFile`.

10. Finish the work at home and save it in your portfolio.

    Food for thought: Think how this program would be different if we have instead worked with lists of `Book`s, or lists of `Shape`s.

5.2 Understanding Equality

Note: This material is covered in pages 321 - 330 in the textbook. Read it carefully.

1. Download the file *Lab5a.zip*. Create a Java Project and add following files to it's source directory.

   - *Account.java*
   - *Checking.java*
   - *Savings.java*
   - *Credit.java*
   - *ExamplesBankAccts.java*

   We now want to define a method that will determine whether an account is the same as the given account. We may need such method to find the desired account in a list of accounts.

   Of course, now that we have the abstract class it would be easy to compare just account number and the name on the account. But, maybe, we want to make sure that the customer's data match the data we have on file exactly - including the balances, the interest rates, and the minimum balances - as applicable.

   The design of the method `same` is similar to the technique described in the textbook. The relevant classes and examples that were handed out in the class can be found in the file *Coffee.java*. You may want to look at the code there as you work through this problem.

2. Begin by designing the method `same` for the abstract class `Account`.

3. Make examples that compare all kinds of accounts - both of the same kind and of the different kinds. For the accounts of the same kind you need both the expected `true` answer and the expected `false` answer. Comparing any checking account with another savings account must produce `false`.

4. Now that you have sufficient examples, follow with the design of the `same` method in one of the concrete account classes (for example the `Checking` class). Write the template and think of what data and methods are available to us.

5. You will need a helper method that determines whether the given account is a Checking account. So, design the method `isChecking` that determines whether this account is a checking account. You need to design this method for the whole class hierarchy - the abstract class `Account` and all subclasses. Do the same to define the methods `isSavings` and `isCredit`.

6. We are not done. This helps with the first part of the `same` method. We need another helper method that tells Java that our account is of the specific type. Here is the method header and purpose for the checking account case:

   ```
   // produce a checking account from this account
   Checking toChecking();
   ```

   In the class `Checking` the body will be just

   ```
   // produce a checking account from this account
   Checking toChecking(){
     return this; }
   ```

   Of course, we cannot convert other accounts into checking account, and so the method should throw a `RuntimeException` with the appropriate message. We need the same kind of method for every class that extends the `Account` class.

7. Finally, we can define the body of the `same` method in the class `Checking`:

   ```
   // produce a checking account from this account
   boolean same(Account that){
     if (that.isChecking()){
       return that.toChecking().sameChecking(this);
     } else {
       return false;
     }
   }
   ```

   That means, we still need the method `sameChecking` but this only needs to be defined within the `Checking` class and can be defined with a `private` visibility.

   Finish this - with appropriate test cases.

8. Finish designing the `same` method for the other two account classes.

Alternative approaches - bad and good

Note 1 - Incorrect alternative:

The method above can be written with two Java language *features*, the `instanceof` operator and *casting* as follows:

```
// produce a checking account from this account
boolean same(Account that){
  if (that instanceof Checking){
    return ((Checking)that).sameChecking(this);
  } else {
    return false;
  }
}
```

However, this version is problematic and not safe.

If the class `PremiumChecking` extends `Checking`, then any object constructed with a `PremiumChecking` constructor will be an instance of `Checking` and the trouble that can result is illustrated in the example *TestSame.java*. You can make a simple project and run the examples, but we include the output from the *tester* for illustration.

Note 2 - A correct alternative:

In the lecture we have introduced another version that also works correctly. It requires us to add a new method for each `class` that `implements` the common interface.

Lecture Notes for the lecture on equality for unions of classes show this technique for the classes that represent geometric shapes (`IShape`, `Circle`, `Rect`, and `Combo`).

Here the methods were:

```
// is this shape the same as the given shape?
boolean sameShape(IShape that);

// is this shape the same as the given circle?
boolean sameCircle(Circle that);

// is this shape the same as the given rectangle?
boolean sameRect(Rect that);

// is this shape the same as the given circle?
boolean sameCombo(Combo that);
```