3 Understanding Equality

The goal of this lab is to make you think about the different measures of equality one can consider, and learn how to design the methods that verify this equality, and how to design tests that are based on these different measures of equality.

3.1 Extensional Equality

As you may have noticed, until now students were not required to know anything about how to compare two objects for equality. The *tester* library compared any two values and reported the results, without burdening the programmer with the task of defining methods that assess the equality of two objects.

This is not an omission or an accident. Defining methods that compare two objects for equality is not a simple task, as there are several ways in which two objects may be considered *equal*. Some languages provide more than one function/method for evaluating equality. Java only gives us `equals` that is typically not very useful. Java `equals` method by default checks the *intensional equality* of two objects: do the two names refer to the same instance. All of the comparisons the *tester* library did for us checked the *extensional equality* of two objects: do the two objects represent the same values?

We will now learn how to define methods that verify that two objects represent the same value. We name these methods `same...`, not `equals` on purpose. We do not want to override the Java `equals` method until we know enough to do it correctly — with overriding the `hashcode` method as well.

Equality of simple objects

Start a new project *Bookstore*. Import into it the files from **Bookstore.zip** and add the *tester* library to the project.

1. Start looking at the class `Book`. We have defined the method `sameBook`, together with the tests for the method. The method body is very straightforward. Notice the tests: we cover the possibility that the two books will differ in the title, in the name of the author, and in the price.

2. Modify the class `Book` so that the `author` field is of the type `Author` and add the class `Author` that has two fields, author's name and the year of birth.

   Design the method `sameAuthor` for the class `Author`, and modify the method `sameBook` for the class `Book` accordingly.

   There is nothing surprising here - the design follows a *cookie-cutter* pattern, so does the test suite for the methods.

Equality of unions

Now look at the file `TestSame`. It contains a very straightforward definition of the method `same` for the class `A` and the class `B` that extends class `A`, but the design is flawed. The problem is that an `instanceof` a class `B` that extends class `A` is also an `instanceof` class `A`, and if the class `B` also implemented an interface `C`, it would also be an `instanceof` the interface `C`.

There are two approaches to designing extensional equality comparisons for unions of data.

The first one is based on *double dispatch* as follows:

In the interface (or the abstract class) that represents the union we define the method `boolean sameUnion(Union that)` that is implemented by every implementing class (or subclass). In the implementing class `UnionA` the method delegates the task to the object `that`, asking it to verify that it represents the same value as `this` instance of the class `UnionA`:

```
// is this instance of UnionA same as the given instance of Union?
public boolean sameUnion(Union that){
  return that.sameUnionA(this);
}

// is this instance of UnionA same as the given instance of UnionA?
public boolean sameUnionA(UnionA that){
  return ... field by field comparison
}
```

But now, the interface must define the method `sameUnion-x` for every class that implements it and every class has to implement these methods as well, returning false in all but one case. If the union is defined as subclasses of a common superclass, the superclass defines one method per subclass, returning false in every case, and each subclass overrides only the method that compares `this` with another instance of the same class.

1. Design the *double-dispatch* based method `sameBook` for the classes that extend the abstract class `ABook`.

2. Think carefully what it takes to create a complete test suite for this method, and the relevant helper methods.

3. *Food for thought — do it later.*

   Add a new class `PaperbackBook` that extends the class `PrintBook`. Then think what needs to be done to implement properly the `sameBook` method.

The second technique resembles the incorrect one that relied on the `instanceof` operator, but avoids the pitfall we have uncovered. Each class that is a member of the union implements two methods:

```
// was this object defined using a constructor for UnionA
public boolean isUnionA();

// return this object as an instance of UnionA
public UnionA asUnionA();
```

Of course the first method in every other class returns `false`, the second method in every other class throws an exception, indicating that `this` object cannot be cast to the desired type.

The `isSame` method is then defined in the class `UnionA` as:

```
// is this UnionA object the same as the given Union object?
public boolean isSame(Union that){
  if (that.isUnionA())
    return this.sameUnionA(that.asUnionA());
  else
    return false;
}
```

1. The file `Coffee.java` illustrates this design with minimal overhead. Read the code and make sure you understand why the output consists of just four `false`s.

2. Design the method `isSameBook` and all the needed helper methods so that it determines the equality of two books, using the technique shown in the **Coffee.java** file.