Abstracting over Datatypes; Using/Reading Javadocs

The goal of this lab is to understand how we can design more general (generic) programs by defining/designing common behavior and structured data, such as that for lists, as parametrized datatypes.

7.1 Parametrized Datatypes

Begin by downloading `Lab7.zip` and building a project that contains all the files as well as the latest version of the Tester Jar, `tester.jar`.

Your project should have the following files:

- `Book.java`
- `Song.java`
- `Image.java`
- `ILo.java`
- `Examples.java`

Run the project and make sure all tests pass, then work through the following exercises.

1. The file `Examples.java` contains tests for the method `totalValue` in the classes that represent lists of items of the type `T`.

   If you un-comment the `testValue` method in the `Examples` class then the program breaks. Modify the `Book`, `Song`, and `Image` classes so that the method `totalValue` works correctly for the list classes with items of the type `Book`, `Song`, and `Image`, and all the tests pass.

2. Now design a method `makeString` for the list classes that produces readable `String` representations of the elements of type `T` in the list.

(a) Design a method `makeString` for each of `Book`, `Song`, and `Image` that produces a `String` representing all (or some of) the data in `this` instance of the class. For example, you may construct a `String` that contains the book title and author's name; the song's title and artist; etc.

(b) Define an interface `MakeString<T>` that represents a `makeString` method for objects of type `T`. As above, method implementations will produce a `String` representation of the entire object, or a part of it.

(c) Design the method `makeStrings` for the list classes with items of type `T` that produces a list of `String`s, which is the result of applying the `makeString` method to every item in the list.

Test your methods on the lists of books, songs, and images, in the manner similar to that shown in the previous examples.

3. We would like to generalize the method `filter` we have seen previously so that it works for arbitrary lists of items. The method produces a list of all items that satisfy some predicate. We modify the `ISelect` interface so it can be applied to any type of data:

```
// a method to decide whether this item
// has the desired property
interface ISelect<T>{
   // does this data item have the desired property?
   public boolean select(T data);
}
```

Design the method `filter` that produces a list of all items in the list (parametrized by the type `T`) that satisfy the given predicate (an instance of a class that implements the `ISelect<T>` interface). Test it by selecting all books that cost less than $25, all songs that play for more than 180 minutes, and all images with the `"jpeg"` file type.

4. The `makeStrings` method consumed `this` list of items of the type `T` and produced a list of items of the type `String`.

Think of the *Racket* function `map`. It consumes a list of `X`, a function of the type `X -> Y`, and produces a list of `Y` by applying the given function to every item in the list.

So, our `makeStrings` method is a `map` from lists of the type `T` (we used `Song`s, `Book`s, and `Image`s) to lists of items of the type `String`.

(a) Design the interface `ITransform<T, S>` that represents a method `transform` that converts the given item of the type `T` to an item of the type `S`. The interface will be parametrized over two (possibly different) datatypes, `T` and `S`.

(b) Design three classes that implement this interface as follows:

- from the type `Book` to the type `String` e.g. the book title
- from the type `Image` to the type `Integer`, e.g. the image size, or width, or height
- from the type `Song` to the type `Boolean`, e.g. by the given artist, or short song...)

Notice that we use the types `Integer` and `Boolean` instead of the primitive types **`int`** and **`boolean`**. These upper-case types are so called *wrapper classes* (or *boxed* types) that allow us to use a primitive data type as if it were a regularly defined class.

Java automatically converts instances of these classes to and from their primitive values when required, so primitive values may be used where the wrapper type is required and vice versa. There is one exception: ***only the wrapper classes may be used as type parameters***, i.e., you cannot have an instance of `ILo<int>`.

(c) Design the method `map` for the classes that represent a list of items of the type `T`. The method header will be:

```
// Produce a list with items of type S from this
//   list  of items of type T by applying the
//   given function to every item in this list
public <S> ILo<S> map(ITransform<T, S> transform);
```

7.2 Reading/Using Java Documentation

Until now, our purpose statements were sufficient for someone trying to understand how our program works and where to make changes, even if another person wants to improve the program we have written. However, if we design a program that represents a reusable datatype, such as lists or binary search trees parametrized over the type of data they contain, a client of our code may not be interested in the implementation details, only the fields, constructors, and methods that can be used, called, or overridden.

Most modern general-purpose programming languages come with a special (embedded) language for writing purpose statements that can then be translated into documentation. Typically documentation is generated as cross-referenced web pages, which allow a client programmer to understand and use a library without looking at actual code.

JavaDoc Basics

1. Go to the *javalib* web site:

   `http://www.ccs.neu.edu/javalib`

   Go to the `Tester` link on the left, then look at *JavaDocs* tab and open the documentation for the latest version of the *tester* library. The web site you see has the documentation for all public fields and methods in the library. Click on the `Tester` tab in the left pane and you will see a description of the `Tester` class.

2. Scroll through the descriptions of the methods until you find the documentation for `checkInexact`. Click on the method and you will see a detailed description of the method - its purpose, its parameters, and the return value it produces.

3. Now look at the method `checkRange` in the *Method Summary* section. You can see that there is a number of methods with this name: some that consume an argument of the type `java.lang.Comparable<T>` and some that consume an argument of the type `java.util.Comparator<T>`.

   These are two interfaces defined in standard Java libraries. The first is a part of the Java language package (`java.lang`). The classes and interfaces defined there are automatically imported for every Java program. For example, the class `String` is specified in the documentation as `java.lang.String`; we have been using it all along without the need for specific `import` statements.

   The interface `java.util.Comparator<T>` is a part of the `java.util` package in the **Java Collections Framework**: a library of classes and interfaces for dealing with collections of data.

Java Collections Framework

Go to the JavaDoc web site for Java libraries at:

http://java.sun.com/javase/6/docs/api/

1. Scroll through the *All Classes* frame on the left till you find `Comparable` and `Comparator`. You can see in the description that

there is a lot of detail in there, much more than we would expect from such a simple function object. We will address some of these issues in the lectures.

2. It looks like we could replace our interface ICompareBooks for binary search trees with the interface Comparator<T>. When you do you will need to add an **import** java.util.*; statement at the beginning of your program. Other than renaming all the previous uses (and the implemented method names) your program should work as before.

Note: Finish this lab and include your work in your portfolio for Assignment 7.