8 Introducing Type Parameters

Goals

In the first part of this lab you will learn how to correctly design tests for the methods that change the state of an object.

In the second part of the lab you will learn to abstract over the functional behavior.

In the third part you will get the first introduction to the abstraction over the type of data using type parameters.

8.1 Abstracting over the Datatype: Generics

Introducing type parameters

For this part download the files in Lab8Sp2007-s2.zip. The folder contains the files ImageFile.java, ISelect.java, SmallImageFile.java, AList.java, MTList.java, ConsList.java, and Examples.java.

Create a new Project Lab8Part1 and import into it all the given files. Add the variable to include tester.jar in the project.

Look at the files. The interface ISelect now includes a type parameter T:

```java
public interface ISelect<T> {
    /* Return true if this Object of the type T should be selected */
    public boolean select(T o);
}
```

That means that the implementing class can decide what type of data should be used as the argument to the select method. This allows us to define the class SmallImageFile as follows:

```java
/* Select image files smaller than 40000 */
public class SmallImageFile implements ISelect<ImageFile> {
    /* Return true if the size of the given ImageFile is smaller than 40000 */
    public boolean select(ImageFile o) {
        return o.height * o.width < 40000;
    }
}
```

The implementation of the same method in the class ImageFile is then greatly simplified:

```java
// is this imagefile the same as the given one?
public boolean same(ImageFile that){
```
Look at the class definition for the class ImageFile to see the use of the type parameter there.

Moreover, the classes that represent a list of arbitrary items can now specify the type of items that can be included in the list construction.

Re-do all of the problems from the previous part, but using the type parameters

1. In the Examples class design the tests for the class SmallImageFile, just as you did before.

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

3. Design the class NameShorterThan4 that implements the ISelect<ImageFile> interface with a method that determines whether the name in the given ImageFile object is shorter than 4.

   Make sure in the class Examples 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 allSuch<T> that that determines whether all items in a list (of items of the type T) satisfy the predicate defined by the select method of a given instance of the type ISelect<T>. In the Examples class test this method by abstracting over the method allSmallerThan40000 and the method allNamesShorterThan4.

6. For the second etude, at home, follow the same steps as above to design the method anySuch 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 ISelect.
Equality - again

Look now at the implementation of the method `contains` for the classes that represent a list of `<T>`. We know this would not work very well, because the comparison for equality requires that we are testing against an identical object, not just an object that contains the same data.

The test harness includes a parametrized `ISame` interface defined as follows:

```java
interface ISame<T>{
    // is this object the same as the given one?
    boolean same(T that);
}
```

1. Modify the class `ImageFile` so that it implements the `ISame<T>` interface, by defining a method that compares `this ImageFile` against the given one.

2. Modify the method definition of the `contains` method so that it expects that the object passed as argument implements the `ISame<T>` interface and use the given argument to invoke the `ISame` method.

3. Of course, you are following the DESIGN RECIPE and so you have made examples and tests for your method.

4. If you are brave, modify the definition of the classes that represent lists of objects of the type `<T implements ISame<T>>`. It allows you to rewrite the code for the `contains` method so that `this` can invoke the `same` method.

5. Java provides a similar interface that can be used to compare two objects — `Comparator`. If you have the time, ask the TA to show you where to find the documentation for Java libraries, and look up the information on the `Comparator` interface.