CMPS 12M
Introduction to Data Structures Lab
Lab Assignment 6

The goal of this assignment is to gain experience with an advanced feature in Java called *Generics*. (This is similar to the notion of *Templates* in C++.) Begin by reading the online Java Tutorial section on Generics at

http://docs.oracle.com/javase/tutorial/java/generics/index.html

Another good resource is section 9.4 (p.445-454) of our text. Actually both of these resources offer a lot more than is necessary for this assignment. The first few subsections of the Java tutorial, or the first 3 or 4 pages of section 9.4 should suffice.

Java Generics allow you to abstract over types. Consider the IntegerList ADT, several incarnations of which were covered in class. It would be a simple exercise to convert this to a List of Strings perhaps, and call the new ADT StringList. Likewise we could write DoubleList, or BooleanList or any other kind of List ADT. More convenient would be to build a single List-of-anything ADT that could be reused with any data type. One way to accomplish this would be to create a List of Objects or ObjectList ADT. This is what the book has done in Chapter 4 (p.211-213) based on an Object array, and in Chapter 5 (p.245-240) based on a linked list whose nodes hold Object references. Study these examples if you have not already done so. (Note this is similar to what you did in pa4 by building a Queue of Objects.) This approach can be problematic however since nothing would stop the ObjectList client from creating a non-homogeneous List, i.e. one that contains different kinds of Objects. For instance you could create a List whose first element was a String, second element an Integer, third element a Scanner, fourth element another String, and fifth element a List! Such structures can be useful, but Java is a strongly typed language and type mismatch errors may ensue if the Client forgets which type of Object is in which position. A more Java-like structure would be a homogeneous List that is rigorously checked by Java's type checking mechanism. This is possible using Java Generics. A limitation of both approaches is that one cannot create Lists of primitive types such as int or double, only Lists of reference types. This is no real limitation though since every primitive type has a corresponding reference type like Integer for int or Double for double.

Consider a simple class Pair that encapsulates two data values of the same type, String say.

```
class Pair{
   // fields
   private String first;
   private String second;

   // constructor
   Pair(String f, String s){first = f; second = s;}

   // ADT operations:  getters and setters
   String getFirst(){return first;}
   String getSecond(){return second;}
   void setFirst(String f){first = f;}
   void setSecond(String s){second = s;}

   //  override an Object method
   public String toString(){
      return "("+first+", "+second+")";
   }
}
```

To convert this ADT to a Generic Pair we introduce a type parameter `T`. This parameter can be thought of as a variable for which one can substitute a (reference) type.

```
class Pair<T>{
   // fields
   private T first;
   private T second;

   // constructor
   Pair(T f, T s){first = f; second = s;} //constructor

   // ADT operations: getters and setters
   T getFirst(){return first;}
   T getSecond(){return second;}
   void setFirst(T f){first = f;}
   void setSecond(T s){second = s;}

   // override an Object method
   public String toString(){
      return "("+first.toString()+", "+second.toString()+")";
   }
}
```

Notice the use of type T's `toString()` function within Pair's `toString()`. This is possible since every Java Object, including instances of T whatever they may be, have a `toString()` function. With the Generic Pair in hand we can create a Pair of Strings as follows.

```
Pair<String> ps = new Pair<String>("happy", "sad");
```

As one would expect, the name of the new type `Pair<String>` is also the name of a constructor of that type. The same generic Pair class can be used to create a pair of Doubles. Recall that `Double` is the reference type that encapsulates the `double` primitive type, and that `Double()` is its constructor.

```
Pair<Double> pd = new Pair<Double>(new Double(2.5), new Double(5.7));
```

Actually it's not necessary to call the Double constructor in this case since the compiler can infer that from the type argument `Double`. Thus

```
Pair<Double> pd = new Pair<Double>(2.5, 5.7);
```

is entirely equivalent. The following test file exercises some of these options.

```
class PairTest{
   public static void main(String[] args){
      Pair<String> ps = new Pair<String>("happy", "sad");
      Pair<Double> pd = new Pair<Double>(2.5, 5.7);
      System.out.println(ps);
      System.out.println(pd);
      ps.setFirst("very");
      pd.setSecond(-3.4);
      System.out.println(ps);
      System.out.println(pd);
      Pair<String> ps2 = new Pair<String>("one", "two");
      Pair<Pair<String>> pps = new Pair<Pair<String>>(ps, ps2);
      System.out.println(pps);
   }
}
```

A complication arises when we try to add an `equals()` method to our Pair-of-anything ADT. Following the IntegerList example found on the webpage, we might try the following as a first draft.

```
public boolean equals(Object rhs){
   boolean eq = false;
   Pair<T> R = null;

   if(rhs instanceof Pair<T>){
      R = (Pair<T>) rhs;
      eq = (this.first.equals(R.first) && this.second.equals(R.second));
   }
   return eq;
}
```

Inserting this into our parameterized Pair class we find that it does not compile. The problem is that the `instanceof` operator does not work on parameterized types. Instead we can call the `getClass()` function on both `this` and `rhs`, then compare the returned values. The public function `getClass()` belongs to the Object superclass (like `toString()` and `equals()`) and returns the runtime class associated with an Object. Our second draft is:

```
public boolean equals(Object rhs){
   boolean eq = false;
   Pair<T> R = null;

   if(this.getClass()==rhs.getClass()){
      R = (Pair<T>) rhs;
      eq = (this.first.equals(R.first) && this.second.equals(R.second));
   }
   return eq;
}
```

Compiling as before we find that there are no errors, but there is a note to the effect that Pair.java uses unchecked or unsafe operations. If we re-compile with the warnings flag `-Xlint`, we get explicit warnings about the cast operation: `(Pair<T>) rhs`.

```
%javac -Xlint Pair.java
Pair.java:29: warning: [unchecked] unchecked cast
         R = (Pair<T>) rhs;
                       ^
  required: Pair<T>
  found:    Object
  where T is a type-variable:
    T extends Object declared in class Pair
1 warning
```

Even though we put the cast inside the conditional `if(this.getClass()==rhs.getClass()){..}`, the compiler is not able to prove that a `ClassCastException` will not be thrown at runtime, and it therefore provides a warning.

You should always compile your Java programs with the `-Xlint` flag and pay close attention to the ensuing warnings, since they often mean there is some logical inconsistency in the program. This is one situation however where there is no logical flaw in the program and no natural way to get rid of the warning. Java provides a mechanism for dealing with such circumstances called *Annotations*. Annotations provide data about a program that is not part of the program itself, and has no direct effect on the operation of the code they annotate. They have several uses, among them to inform the compiler to detect certain (otherwise undetected) errors or to suppress warnings. Annotations always begin with the '`@`' character. You can learn more about them at

http://docs.oracle.com/javase/tutorial/java/annotations/

The annotation we need is `@SuppressWarnings("unchecked")` placed before the function definition. Our third and final draft of function `equals()` is now:

```
@SuppressWarnings("unchecked")
public boolean equals(Object rhs){
   boolean eq = false;
   Pair<T> R = null;

   if(this.getClass()==rhs.getClass()){
      R = (Pair<T>) rhs;
      eq = (this.first.equals(R.first) && this.second.equals(R.second));
   }
   return eq;
}
```

Compiling this with the `-Xlint` flag results in no errors or warnings. The complete version of the parameterized Pair class is posted on the webpage along with a test client that exercises all ADT operations, including the `equals()` method.

What to Turn In

Convert the IntegerList ADT on the course webpage from a List of ints to a List of anything using Java Generics. Use as starting point either the Linked List version or the Array Doubling version, both found on the course website under Examples/Lecture/IntegerListADT/. Your new generic type will be called simply List and will be defined in a file called `List.java`. Your List class will implement a generic List interface defined in the file `ListInterface.java` also found on the webpage under Examples/Labs/lab6/. Test your List by writing a file called `ListTest.java` that exercises all ADT operations. The same directory contains files `ListIndexOutOfBoundsException.java`, `ListClient.java`, `makefile` and `model-out`. The makefile makes the executable jar file `ListClient` which should produce output identical to the contents of `model-out`, provided your List ADT works properly. Submit the following files to lab6.

Written and submitted by you:
```
List.java
makefile
ListTest.java
```

Provided in Examples/Labs/lab6 and submitted unchanged:
```
ListInterface.java
ListIndexOutOfBoundsException.java
ListClient.java
```

The following files found in Examples/Labs/lab6 are not to be submitted: `Pair.java`, `PairTest.java`, `model-out`. As always start early and ask for help if anything is not completely clear.