CSCI 2110- Data Structures and Algorithms
Laboratory No. 3
Week of September 25th, 2017

This lab will get you familiarized with **Generics in Java**.

Marking Scheme
Each exercise carries 10 points. Your final score will be scaled down to a value out of 10.
Working code, Outputs included, Efficient, Good basic comments included: 10/10
No comments: subtract one point
Unnecessarily inefficient: subtract one point
No outputs: subtract two points
Code not working: subtract up to six points depending upon how many methods are incorrect.

Error checking*: Unless otherwise specified, you may assume that the user enters the correct data types and the correct number of input entries, that is, you need not check for errors on input.*

Submission*: All submissions are through Brightspace. Log on dal.ca/brightspace using your Dal NetId. Submissions are pretty straightforward. Instructions will be also be given in the first lab.*
Deadline for submission: Sunday, October 1st, 2017 at 11.55 p.m. (five minutes before midnight).

What to submit*:*
A zip file containing all source codes and a text document containing sample outputs.

Exercise 0 (no marks): Simple exercise that was discussed in the lectures. Try the program, change the parameters and understand how generics in Java works.

```java
public class Grade<T>{
   private T value;
   public Grade(T entry){
      value = entry;
   }
   public T getValue(){
      return value;
   }
   public void setValue(T entry){
      value = entry;
   }
   public String toString(){
      return ""+value;
   }
   public static void main(String[] args){
      Grade<String> m1 = new Grade<String>("A");
      Grade<Integer> m2 = new Grade<Integer>(90);
      System.out.println(m1);
      System.out.println(m2);
      m1.setValue("A+");
      m2.setValue(65);
      System.out.println(m1);
      System.out.println(m2);
   }

}
```

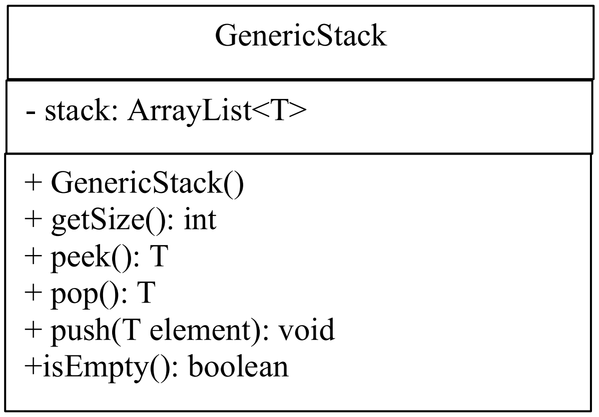
Exercise 1: Write a Point class that has xpos and ypos as its fields, get and set methods, and a toString method. It must be written as a generic class, which means that it should work for any type of object, that is, the xpos and ypos could be either Integer objects, or Double objects, or String objects.

A sample test program is given below. Modify the test program to accept input data from the user and try it for different input values.

```
public class PointTester
{
      public static void main(String[] args)
      {
            Point<Integer> point1 = new Point<Integer>(10,20);
            Point<Double> point2 = new Point<Double>(14.5, 15.6);
            Point<String> point3 = new Point<String>("topleftx",
            "toplefty");
            System.out.println(point1);
            System.out.println(point2);
            System.out.println(point3);
      }

}

The output is:
XPOS: 10    YPOS: 20

XPOS: 14.5  YPOS: 15.6

XPOS: topleftx  YPOS: toplefty
```

The following exercises (2 to 5) illustrate the concept of building data structures in "layers". You will design a Generic Stack and Generic Queue data structure using an ArrayList. Although an ArrayList itself is a generic data structure, the idea behind layering is that the arraylist can be changed to another data structure such as a linked list and the applications running the queue will not notice the difference.

Exercise 2: You would recall from CS1101 that a stack is a last in, first out data structure that has push, pop and peek as its basic operations. In this exercise, you will create a simple generic stack class that is implemented with an ArrayList to store the stack elements. The UML diagram is given below.



The initial part of the code is given below:

```
import java.util.ArrayList;
```

```
public class GenericStack<T> {
      private ArrayList<T> stack;

      //continue


}
```

As an example of its use, write a small test (demo or client program) to create two stacks, one to hold strings and another to hold integer objects, and test the methods:

```
GenericStack<String> stack1 = new GenericStack<String>();
stack1.push("London");
stack1.push("Paris");
stack1.push("Halifax");

GenericStack<Integer> stack2 = new GenericStack<Integer>();
stack2.push(1);
stack2.push(2);
stack2.push(3);
```

Exercise 3: Suppose that you want to store student records in the generic stack that you created in Exercise 2. For this, first define a StudentRecord class that holds three attributes: String firstName, String lastName, int bannerID. Add the appropriate constructor, get, set and toString methods.

Next, write a client (demo) program that does the following:

1. Create an empty stack, stack1, that can hold student records.
2. Read a file that contains student records, one on each line, such as:
```
Ichabod Crane 123456
Brom Bones 456321
Emboar Pokemon 111222
Rayquazza Pokemon 333111
Cool Dude 101010
Trend Chaser 654321
Chuck Norris 112233
Drum Dude 111222
```

   You may assume that each line has three components only: first name, last name and Id number. You would read the ID number as a String and then convert it to an Integer object.
   A StringTokenizer program is useful to read a file in which the records are arranged as shown above and split them into first name, last name and ID number.

3. As each line in the file is read, create a StudentRecord object and push it into stack1.
4. Repeat until all the lines in the file are read.
5. Create another stack2 that can hold string objects.
6. Pop stack1 item by item and push only the last name in each student record into stack2.
7. Pop stack2 and display the items. With the example set of records given above, this would display the last names in the reverse order:
```
        Dude
        Norris
        Chaser
```

```
Dude
Pokemon
Pokemon
Bones
Crane
```

Here is the structure of the solution to this exercise. (It is assumed that you have a separate StudentRecord class)

```
import java.util.Scanner;
import java.io.*;
import java.util.StringTokenizer;

public class Exercise3{
   public static void main(String[] args)throws IOException{

      //TODO: Create stack1 to hold StudentRecord objects

      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter the filename to read from: ");
      String filename = keyboard.nextLine();

      File file = new File(filename);
      Scanner inputFile = new Scanner(file);
      StringTokenizer token;
      while (inputFile.hasNext()){
         String line = inputFile.nextLine();
         token = new StringTokenizer(line, " ");
         String f = token.nextToken();
         String l = token.nextToken();
         String bString = token.nextToken();

         //TODO: Convert bString to an Integer object
         //Create a StudentRecord object with the first name, last name and
         //ID number, push it into stack1

      }
      inputFile.close();

      //TODO: Continue with the rest of the steps 5, 6 and 7

   }
}
```

Writing static methods in generics

Although the above exercise doesn't really specify that you have to write methods in your demo program, in case you decide to write a static generic method, the header for the method should be written as follows:

```
public static <T> returnType methodName(input parameters)
```

Here's a simple generics program with a static method:

```
public class GenericMethodDemo{
     public static void main(String[] args) {
          Integer[] integers = {4, 5, 6, 7};
          String[] strings = {"A", "B", "C", "D"};
          GenericMethodDemo<Integer> print(integers);
          GenericMethodDemo<String> print(strings);
}
public static<T> void print(T[] list){
     for(int i=0; i<list.length; i++)
          System.out.print(list[i] + " ");
     System.out.println();
}
}
```

Exercise 4: A queue is a first in first out linear data structure in which items are added at one end and removed from the other end.

A generic Queue class can be built with the following specification:

Queue() :	creates an empty queue
void enqueue(T item):	add new item to the end of the queue
T dequeue():	remove and return the item from the front of the queue
int size():	return the number of elements in the queue
boolean isEmpty():	return true if the queue is empty, false otherwise.
void clear():	clear the queue
T peek():	return the entry from the front of the queue, null if the queue is empty

In addition to the above basic operations, it is useful to have the following for queue applications:

int positionOf(T item):	Return the position of the specified item and -1 if not found.
void remove(T item):	Remove the first occurrence (from front) of specified item
T first():	Return the first item in the queue(front), null if queue is empty
T next():	Return the next item in the queue relative to the previous call to first or next. Return null if end of queue is reached.

Implement the generic Queue class using an ArrayList to store the Queue elements. Ensure appropriate error checks.

```
public class GenericQueue<T>
{
          private ArrayList<T> queue;
          int cursor;  //the cursor is mainly used for the first and the next methods.


          //continue rest of the code


}
```

Write a demo program to test the various methods in the Queue class.

Exercise 5: You are to develop a simple PrintQueue class using the Queue class. It is modeled similar to the Unix print command. The following is the specification:

PrintQueue()

Creates an empty print queue

void lpr (String owner, int jobId)

Enqueues a job with the specified owner name and job id

void lpq()

Prints all the entries in this queue

void lprm(int jobId)

Removes the active job at the front of the queue if jobId matches, error message otherwise

void lprmAll(String owner)

Removes all jobs from the queue that have been submitted by the owner

As an example, you can create a print queue pq as follows:

swilliams	309
ronaldinho	300
marionjones	312
swilliams	267
davidh	135
ronaldinho	301

pq.lprm(309) → removes the first entry from the queue

pq.lprmAll(ronaldinho) → removes all entries of ronaldinho from the queue

As can be seen, each item in the queue has two entries:
owner: a String
jobId: an int
So first, you need to create a simple Job class to hold these two entries:

```
public class Job
{
        private String owner;
        private int jobId;

        public Job(String o, int j)
        {
                owner = o;
                jobId = j;
        }
        public String getOwner()
        {
                return owner;
        }
        public int getJobId()
        {
                return jobId;
        }
}
```

Then complete the following PrintQueue class. Write a demo program to illustrate the various operations of the PrintQueue class.