Dwayne Fraser COP 4331 003
Homework # 5
Problem #1

```
/*
DWAYNE FRASER
HOMEWORK 5
L.1
 */

package q1;

interface Functor<R,T> {

   R apply(T param);
 }

/*
DWAYNE FRASER
HOMEWORK 5
L.1
 */

package q1;

public interface Functor2 <R,T1,T2> {
         R apply(T1 t1, T2 t2);
}

/*
DWAYNE FRASER
HOMEWORK 5
L.1
 */

package q1;

public class LengthFun implements Functor <Integer, String> {

   private static void print(String x) {
         System.out.println(x);
   }

   /*
```

```
    Apply Function
     */

    @Override
    public Integer apply(String y) {
        return y.length();
    }

    public static void main(String[] args) {

       String str = "Test String";

          LengthFun x = new LengthFun();

          Integer y = x.apply(str);

          print(str + y);

          // Lambda:
          Functor<Integer, String> z = string -> string.length();
          print(str + z.apply(str));



    }
}

/*
DWAYNE FRASER
HOMEWORK 5
L.1
 */

package q1;

import java.util.LinkedList;
import java.util.Arrays;



public class MyList <T> extends LinkedList <T> {

    private static final long serialVersionUID = 1L;
```

```
        public MyList() {
                super();
        }


        public <R> MyList <R> map(Functor <? extends R,? super T> fo) {

        MyList <R> x = new MyList<R>();

                for (T e: this) {
                        x.add(fo.apply(e));
                }
                return x;
        }


        public T reduce(Functor2 <T,T,T> func, T initval) {
                T val = initval;
                for (T e:this) {
                        val = func.apply(val, e);
                }
                return val;
        }


private static void print(String s) {
        System.out.println(s);
}


public static void main(String[] args) {


        TimesTwoFun X = new TimesTwoFun();
        MyList<Integer> Y = new MyList<>();
        Y.addAll(Arrays.asList(-2,1,0,4));

        MyList <Integer> list1 = Y.map(X);
        print(Y + " returns " + list1);


        MyList <Integer> list2 = Y.map(x -> 2 * x);
        print(Y + " returns " + list2);
        Summer summer = new Summer();

        MyList<Integer> summerlist = new MyList<>();
        summerlist.addAll(Arrays.asList(3, -1, 1, 4));

        Integer sumOfAll = summerlist.reduce(summer, 0);
        print(summerlist + " with " + summer.getClass() + " = " + sumOfAll);

        Integer sumOfAll2 = summerlist.reduce((x,y) -> x + y, 0);
        print(summerlist + " with lambda = " + sumOfAll2);

  }
}

class TimesTwoFun implements Functor<Integer, Integer> {
        @Override
        public Integer apply(Integer param) {
                return param * 2;

        }
}

class Summer implements Functor2<Integer,Integer,Integer> {
        @Override
        public Integer apply(Integer t1, Integer t2) {
           return  t1 + t2;
        }
}
```

Problem #2

```
/*
DWAYNE FRASER
HOMEWORK 5
10.1
 */

package q2;

import java.util.*;
```

```
public class LQueue <E> implements MyQueue<E> {

        private LinkedList<E> elements;

        public LQueue() {
                elements = new LinkedList<E>();
        }

        @Override
        public E head() {
                E h = elements.getFirst();
                return h;
        }

        @Override
        public E dequeue() {
                return elements.removeFirst();
        }

        @Override
        public void enqueue(E e) {
                elements.add(e);
        }

        @Override
        public int size() {
                return elements.size();
        }

        @Override
        public boolean isEmpty() {
                return size() == 0;
        }


        @Override
        public void addAll(Collection <? extends E> c) {
            elements.addAll(c);
        }

}

/*
DWAYNE FRASER
```

```
HOMEWORK 5
10.1
 */

package q2;

import java.util.Collection;

public interface MyQueue <E> {

	E head();


	E dequeue();


	void enqueue(E e);

	/**
	 * returns the size of the queue
	 * @return the size of the queue
	 */

	int size();

	/**
	 * returns true if the queue is empty
	 * @return true if the queue is empty
	 */
	boolean isEmpty();

	void addAll(Collection <? extends E> c);
}

/*
DWAYNE FRASER
HOMEWORK 5
10.1
 */

package q2;

import java.util.*;
```

```java
public class QueueTest {

    private static void log(String msg) {
            System.out.println(msg);
    }


    public static void main(String[] args) {
        LQueue <Integer> q = new LQueue <Integer>();

        try {
                Integer x = q.head();

                log(" head() with empty queue: FAILED");
        } catch (NoSuchElementException ex) {
                log(" head() with empty queue: passed");
        }

        try {
                Integer x = q.dequeue();
                log(" dequeue() with empty queue: FAILED");
        } catch (NoSuchElementException ex) {
                log(" dequeue() with empty queue: passed");
        }

        q.enqueue(1);
        q.enqueue(2);
        int s = q.size();
        q.enqueue(3);

        if (q.size() - s != 1) {
                log(" size() not checked");
        }

        if (! q.head().equals(1)) {
                log(" head() failed");
        }
        if (! q.dequeue().equals(1)) {
                log(" dequeue() failed: 1");
        }
        if (! q.dequeue().equals(2)) {
                log(" dequeue() failed: 2");
        }
        if (! q.dequeue().equals(3)) {
            log(" dequeue() failed: 3");
        }

        try {
            Integer x = q.dequeue();
            log(" dequeue() with empty queue: FAILED (second test)");
        } catch (NoSuchElementException ex) {
            log(" dequeue() with empty queue: passed (second test)");
        }

        LQueue <Integer> q2 = new LQueue <Integer>();
        q2.enqueue(1);
        q2.enqueue(2);

        Collection<Integer> col = new LinkedList<>();
        col.add(3);
        col.add(4);

        q2.addAll(col);
        if (! q.dequeue().equals(1)) {
            log(" dequeue() failed: 1");
        }
        if (! q.dequeue().equals(2)) {
            log(" dequeue() failed: 2");
        }
        if (! q.dequeue().equals(3)) {
            log(" dequeue() failed: 3");
        }
        if (! q.dequeue().equals(4)) {
            log(" dequeue() failed: 2");
        }

        log("Seems ok");
    }
}

/*
DWAYNE FRASER
HOMEWORK 5
10.2
 */

package q2;
```

```
import java.io.PrintStream;


public class Stdout {

	static private Stdout theInstance = null;


	private PrintStream stdoutStream = null;


	private Stdout() {
		this.stdoutStream = System.out;
	}

	public void println(String s) {
		this.stdoutStream.println(s);
	}


	public static Stdout getInstance() {
		if (theInstance == null) {
			theInstance = new Stdout();
		}
		return theInstance;
	}


	public static void main(String[] args) {


		Stdout stdout = Stdout.getInstance();
		stdout.println("testing...testing...testing...");
	}

}
```

Problem #3

```
/*
DWAYNE FRASER
HOMEWORK 5
7.1
 */
```

```
package q3;

import  java.io.Serializable;

public class Pair <K, V> implements Serializable, Cloneable {
   private final static long serialVersionUID = 1;

   public Pair(K k, V v) {
         this.key = k;
         this.value = v;
   }

   private Pair() {
   }

   public K k() { return key; }

   public V v() { return value; }


   public boolean equals(Object otherObject) {
         if (this == otherObject) return true;
         if (otherObject == null) return false;
         if (getClass() != otherObject.getClass()) return false;

         Pair <?,?> other = (Pair <?,?>)otherObject;
         return k().equals(other.k()) && v().equals(other.v());
   }


   public int hashCode() {
         return 13 * k().hashCode() + v().hashCode();
   }

   public String toString() {
         return k().toString() + ":" + v().toString();
   }

   public Object clone() {

         Pair <K,V> copy = new Pair <K,V>(k(), v());

         return copy;
```

```
    }


    private K key;
    private V value;
}

/*
DWAYNE FRASER
HOMEWORK 5
7.1
 */

package q3;

import java.io.*;

public class PairTest {
    public static void main(String[] args) {

            Pair <Integer, String> x = new Pair <Integer, String>(1, "one");
            Pair <Integer, String> y = new Pair <Integer, String>(1, "onetwo");
            Pair <Integer, String> z = new Pair <Integer, String>(1, "one");

            assert z.equals(x): "Fail";
            assert ! z.equals(y): "Fail";

            String filename = "serialized.dat";
            try {
                ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream(filename));
                os.writeObject(z);
                os.close();

                ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
                Pair <Integer, String> p1restored = (Pair <Integer, String>) in.readObject();
                assert z.equals(p1restored): "Fail";
            } catch (Exception ex) {
                ex.printStackTrace(System.err);
            }

            Pair <Integer, String> p1copy = (Pair <Integer, String>) z.clone();
            assert z.equals(p1copy): "Fail";
            assert z.hashCode() == p1copy.hashCode(): "Fail";

    }
}
```