Java Programming
MTech Preparatory Semester 2016
Practice programs – 2

Problem 1:

1. Implement a class LList that implements the functionality of a singly-linked list. The outline of the expected methods and a simple **main** to test the class are below. Add any helper classes you need for the implementation.

2. We would like to improve the efficiency of some of the methods of this implementation. If **size** and **append** need to be fast (constant time as against O(n)), what changes would you make?

3. Convert this implementation to a doubly-linked list. What methods change in your implementation?

```
public class LList {
      // pos specifies the location in the list where the operation
      // needs to be performed

      public void insert(int pos, Object obj) {
      }

      public void remove(int pos) {
      }

      public Object get(int pos) {
      }

      public int find(Object obj) { // return the position of obj
      }

      public int size() {
      }

      public void clear() {
      }

      public void append(Object obj) { // add to the end
      }

      public String toString() {
      }

      public static void main(String[] args){
         LList bookList = new LList();
         System.out.println(bookList);
         bookList.append("Harry Potter I");
         System.out.println(bookList);
         bookList.insert(0, "Hamlet");
```

```
            System.out.println(bookList);
            bookList.insert(0, "Cosmos");
            System.out.println(bookList);
            bookList.insert(1, "Java");
            System.out.println(bookList);
            bookList.remove(1);
            System.out.println(bookList);
            bookList.insert(1, "C++");
            bookList.insert(2, "LISP");
            bookList.insert(2, "Calvin & Hobbes");
            System.out.println(bookList);
            int pos = bookList.find("LISP");
            bookList.remove(pos);
            System.out.println(bookList);
            // autoboxing and unboxing
            bookList.clear();
            bookList.append(1);
            bookList.append(1);
            bookList.append(2);
            bookList.append(3);
            System.out.println(bookList);
            pos = bookList.find(2);
            bookList.remove(pos);
            System.out.println(bookList);
      }

}
```

Problem 2:

This example is to understand the behaviour of the garbage collector.

Play around with the size of the array in Blob and the variable n in the main to see when you run out of memory.

Assume you know the size of memory needed for this application (fix the size of space and n), but this is larger than the memory allocated to the application. How can you still get the program to run successfully?

```
public class GCdemo {

      private class Blob {

            int space[10000]
      }

      public static void main(String[] args) {

            // run this with increasing values of n.

            // How large can n be?
```

```
            int n = 10000;
            Blob blobStore[] = new Blob[n];
            for(int i=0;i<n;i++) {
                  Blob b = new Blob();
                  // what happens if you comment the following
                  blobStore[i] = b;
            }
      }
}
```

Problem 3:

JSON (JavaScript Object Notation) is a lightweight data-interchange format as defined below. Implement a set of classes to create and modify JSON objects.

We will work with a subset of JSON to simplify the programming.

The following are our definitions (modified from www.json.org):

> A JSON **object** is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma)
>
> A JSON **array** is an ordered collection of *values*. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).
>
> A **value** can be a string in double quotes, or a JSON *object*, or an array of *values*. These structures can be nested. [Note: this is a subset of the actual JSON definition]

Thus, the JSON structure for a student could look as follows:

{

“name” : “Ramesh”,

“age” : “21”,

“interests” : [“football”, “quiz”, “cooking”]

“grades” : [{“course” : “CS100”, “grade” : “A”},

{“course” : “HS120”, “grade” : “B”},

{“course” : “CS200”, “grade” : “A-“}

]

“hostel addr” : {“hostel” : “Block G”,

“room” : “345”

}

}

Define a class **JsonObject** which can store such structures. To support this, you will probably need to also create classes **JsonValue** (and maybe **JsonArray)** that implement the definitions above.

We want to be able to read an input like the above and construct a JsonObject that represents this data. To this, we should be able to add name/value pairs to a JsonObject. And finally, be able to write out the JSON object similar to the above example.

A possible outline of the JsonObject class is:

```
public class JsonObject {
	public JsonObject() {
	}

	// add a name/value pair to the object
	public void add(String name, JsonValue value) {
	}

	// return the value corresponding to a name.
	// return null if the name does not exist
	public JsonValue get(String name) {
	}
	public String toString() {
	}
}
```

class JsonValue is the tricky one, since it needs to be able to store a string, an array or a JsonObject. While there are many ways of doing this, you could try something on the following lines (you are encouraged to find a better implementation!)

```
public class JsonValue {

	private String str;

	private JsonObject jObj;

	private JsonValue[] jArray;

	/* Implement 3 constructors each of which takes one of
String, JsonObject or array of JsonValues as argument, and
assigns it to the appropriate field. Only one of these three
fields should be non-null */

	// … the constructors


	// we will need 3 different methods for accessing the
appropriate fields.

	// Note that this helps with strong-typing

	String getString() {

	}

	JsonObject getObj() {

	}


	JsonValue[] getArray() {

	}

}
```

The main program should read in a file containing a valid JSON object (as in the above example), build up the JsonObject corresponding to the input, and write out the JsonObject to the console. You will essentially need to implement a parser that can construct the appropriate object based on the character that is read in.

Note: there are many readily available JSON libraries. Do not use any of them! You should not need to import any classes apart from the ones needed for input/output.