Reactions to Two Software Approaches for Object Persistence

Victor Matos
Computer and Information Science Department
Cleveland State University
Cleveland, Ohio 44114
USA

Rebecca Grasser
Computer Science and Information Technology
Lakeland Community College
Lakeland, Ohio 44094
USA

Abstract

Persistence is the mechanism to extend the lifetime of an arbitrary software object beyond the execution of the application that creates the object. In this study we observed student's reactions to two specific preservation schemes: Java JDK serialization (JS), and Google's Json-based serialization (JSON). We instructed a group of undergraduate students on the two previous preservation strategies. Our subjects were asked to write a program to persist (encode-write-read-decode cycle) a sequence of common object patterns using JS and JSON, and identify one of the two approaches as their preferred strategy for future use. Three weeks later the students took an unannounced exam containing two serialization problems. The retention test indicates that not only a larger number of students still preferred JSON over JS, but also the correctness of their solutions was significantly higher than that of classic JS. A primer on JSON encoding is provided as an appendix.

Keywords: Object-Oriented Programming, Java objects persistence, serialization, Java Script Object Notation (JSON), Google Script Object Notation (GSON), learner's reactions.

1. Introduction

Java objects are volatile. Their life time is limited to the life of the thread in which they are created; once the application ends its objects are destroyed. To counterpoint this transient behavior, the Java JDK Persistence scheme offers a way to encode and permanently save objects for further use at other time. In this manner, previously created objects could be reused instead of been recreated from scratch. The study of this subject is typically included in the CS1 and/or CS2 Object-Oriented programming courses. The general principle is quite simple; roughly speaking the data held in an object is encoded and written to a disk file which provides a permanent home for safekeeping the object's state. At some time in the future another application could read the disk data image and with little effort reconstitute the object.

On a larger application context, preserving and reusing Java objects could involve the participation of different physical platforms (embedded, mobile, networked, distributed, etc.) as well as variety of encoding strategies. Among those options we have: Java JDK serialization, plain-text manipulation, SQL database transfer, CSV, XML, and JSON encoding. Contemporary computer applications are developed using a mixture of those hardware and software elements. Contemporary software systems easily range from small apps entirely embedded in a mobile device; to very sophisticate distributed systems involving a large set of heterogeneous computing resources disperse over a wide geographical area. A challenge for the instructor (for whom primarily this article is written) is to choose the most appropriate tool for the students. That preference should be given to the one strategy that is better in the longer term, rather than the one that is easier to learn. Each of the previous alternatives has a number of advantages and disadvantages that need to be considered in the making of this decision; more on the subject is discussed in the next sections.

2. Brief Background

2.1 Java JDK Serialization

We suspect that most instructors use Java JDK serialization (Oracle Corp, 2005) to illustrate *how to do* the saving and retrieving of persistent objects. There are several reasons to support this choice. First, the Serializable interface is part of the Java framework and all it is needed for a class to be preserved is to implement this method-less interface. In addition, most Java textbooks include a section on this particular approach. A second argument in favor of Java serialization is its simplicity. A *serializable* class needs only to write(read) its objects to(from) an Object IO Stream. The core of the Java JDK Serialization process is illustrated in the following code fragment (exception handling is omitted for brevity)

```
//Serialize Person object
Person p1 = new Person("Daenerys Targaryen", 16);
FileOutputStream fileOut = new FileOutputStream("c:/temp/Person.ser");
ObjectOutputStream objOut = new ObjectOutputStream(fileOut);
objOut.writeObject(p1);
objOut.close();
fileOut.close();

// Deserialize Person object
FileInputStream fileIn = new FileInputStream("c:/temp/Person.ser");
ObjectInputStream objIn = new ObjectInputStream(fileIn);
Person p2 = (Person) objIn.readObject();
objIn.close();
fileIn.close();
```

In this fragment the Person object p1 is instantiated (see Figure 1-a, 1-b), and transferred to disk. Its persistent image (Figure 1-c) is later brought back during deserialization to create p2, a clone of p1.

2.2 Problems with Java Serialization

The motivation for our work is based on various drawbacks affecting Java JDK serialization. First, there is the problem of *lack of transparency*. Serialized data is encoded in a compact binary representation; therefore the student cannot *see* the permanent object's image and gain an appreciation for what has been saved of the object (see Figure 1-c). A typical doubt raised at this point can be summarized in this commonly asked question: *Does the persistent object include its methods or just data?* We know, the answer is *just data* [1], however the question is perfectly valid and the lack of transparency in the permanent version of the saved object tends to obscure the comprehension of the subject.

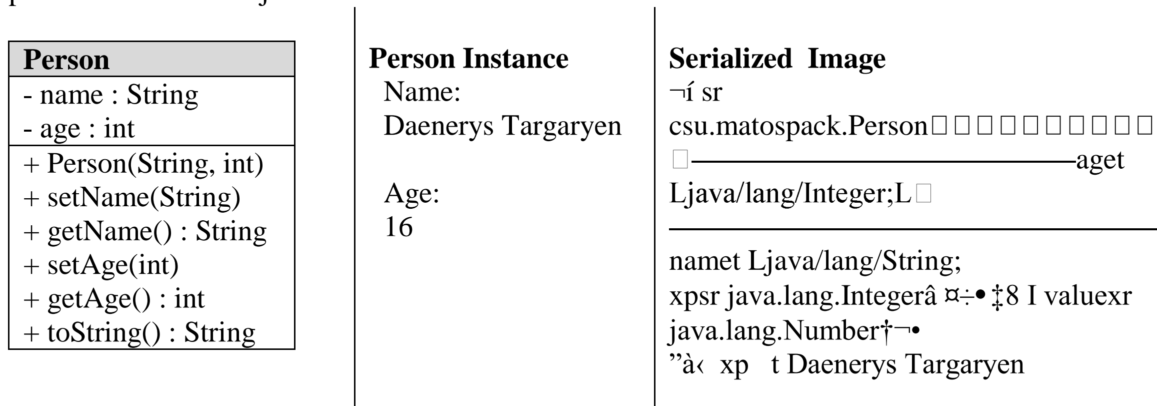

Figure 1: An Instance of the Person class is Shown after it is Serialized

[1] In reality serializing saves a bit more than just the object's data. The outstream.writeObject(obj) method writes the specified object (obj) to the ObjectOutputStream (outstream), but it also includes the object's class name, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes, literal data is saved as strings, and numeric values are stored in binary format.

The second main objection against Java serialization is its *limited inter-operability* capability, i.e. serialized Java data works only in a Java-to-Java interaction. As a consequence of this limitation, Java serialization is of very limited value in complex systems involving software components written in multiple languages. A final restriction in the use of Java serialization is the need to own a copy of the class definition. The source code must include the clause "implements Serializable" for the class to be persisted. For instance, in our example the Person class must be defined as follows: public class Person implements Serializable {…}

2.3 An Alternative Solution: JSON-based Serialization

JSON stands for JavaScript Object Notation (Bray, 2014). This is a text-based "human-readable" format intended for the representation of structured data. JSON's original purpose is to act as a language-independent data interchange vehicle. Objects are written in JSON as a curly-braced (**{ }**) delimited string. The various properties inside the object are recursively annotated as a comma separated set of 'key-value' pairs. Array elements are enclosed inside square brackets (**[ ]**) and separated by commas. JSON was originally proposed for data serialization of web-based applications exchanging data in an AJAX style environment; however it is language and platform independent and it is available for a large number of programming environments (Singh & Leitch & Wilson, 2013), (Microsoft, 2007), (Jackson Project, 2014).

A JSON-based serialization scheme offers a number of positive features, among them we have (1) transparency – the developer can *see* objects as plain text, (2) it can be used on cross-platform applications involving a variety of operating systems and programming languages, (3) it is relatively easy to use and understand, (4) no need to posses original source code to add Serializable annotation support, (5) the technology is already well understood, diffused, and supported.

2.4 A Gson Example

For this study we have opted to work with the open-source Google's Gson API (Singh & Leitch & Wilson, 2013). This is a small and easy to use library for encoding and decoding JSON formatted data. It supports Java Generics and can be used to serialize and deserialize complex objects including generic classes in their hierarchical structures.

The following code fragment briefly illustrates how Gson is used. It works on a modified version of the Person class shown in Figure 1. The new Person class mentioned here, has the Java clause 'implements Serializable' removed from its definition. The first step needed to apply GSON is to create an object from which the class methods toJson() and fromJson() could be invoked.

The act of preserving a Java object can be accomplished by first encoding the object's data and then writing the resulting string on disk. Decoding begins with acquiring a JSON string from permanent storage. It is followed by the instantiation of an appropriated Java Object that feeds its class-variables with the corresponding values parsed from the encoded string. Consider the example:

```
Gson gson = new Gson();
Person p1 = new Person("Daenerys Targaryen", 16);
String jsonString = gson.toJson( p1 ); ➔ {"name":"Daenerys Targaryen","age":16}
Person p2 = gson.fromJson (jsonString, Person.class) ➔ a clone of p1
```

The gson.toJson(p1) expression transforms the Person instance p1 into the JSON formatted string *JsonString* holding {"name":"Daenerys Targaryen","age":16}. The method fromJson(*JsonString, JavaType*) performs the opposite operation.

Unlike Java serialized data, JSON strings do not include a copy of the source class definition they represent. The second parameter passed to fromJson() is responsible for guiding the reconstruction of the equivalent Java object. In general, this parameter is a just a reference to a base class type, as for instance Student.class, Course.class, Person.class etc. However, there is one special GSON case that occurs when using generic collections. For this case a class called com.google.gson.reflect.TypeToken should be used. For instance, the type of an ArrayList<Person> object could be obtained as follows:

```
Type arrPersonType = new TypeToken<ArrayList<Person>>(){}.getType();
```

The new TypeToken object allows GSON to remember the generic type Person held in the ArrayList and could be passed to the fromJson() method to guide the decoding phase.

3. The Experiment – Students Perceptions

We performed an experiment to evaluate student's reactions to Java JDK Serialization (JS) and JSON based serialization. We were interested in four issues related to the use of those two technologies (a) assessing perceived difficulty of each scheme, (b) finding student's overall choice of software, (c) developing awareness on their relative performance, and (d) estimating long term student's retention.

We used two small groups of undergraduate students taking their second OO programming course (a typical population: 30 CIS majors, approx. 2.6 average GPA, 10% female). In preparation for an assignment, we trained the students during a single lesson on both JS and JSON techniques. We gave equal deference to each approach, the tone of the lecture was neutral and we refrained from promoting any particular scheme. At the end of the lesson, students were given a handout summarizing the lecture (see Appendix A & B), and an assignment to be completed in one week (see Appendix C).

The assignment consisted of three parts (a) serializing objects found in five common scenarios, (b) choosing an overall software tool for future serialization work, and (c) implementing a benchmark to assess the relative performance of JS and JSON on a set of size-varying samples. Results were to be transcribed in a questionnaire summarizing their choices and observations.

The first part of the assignment mirrored the serialization cases discussed in class. It asked to use JS and JSON to preserve a number of objects from two classes: Employee and Department. The Employee class consisted of the attributes *employee name* and *ID number*. The Department class included the class variables: *department name*, *address*, and *personnel* list (implemented as an ArrayList<Employee> class).

The specific cases of data preservation included: (1) a simple String object, (2) an Employee object, (3) an ArrayList holding a few Employee objects, (4) a PartTimeEmployee object (an extension of Employee class), and (5) a Department object sustaining a 1:many relationship with the Employee class. For each case, students had to identify the more *difficult-to-use* tool. Figure 2 summarizes the choices made by the students. Overall, 39% of the students found JS to be the most difficult strategy to be used, while only 23% of the surveyed population judged JSON to be harder than JS. The majority (47%) considered both techniques to be of a similar nature.

Perceived Difficulty	Java	JSON	Same
1: Encode string	7	3	16
2: Simple object	5	7	14
3: List of objects	5	12	9
4: Subclass	11	5	10
5: 1:Many	11	3	12
Total	**39%**	**23%**	**47%**

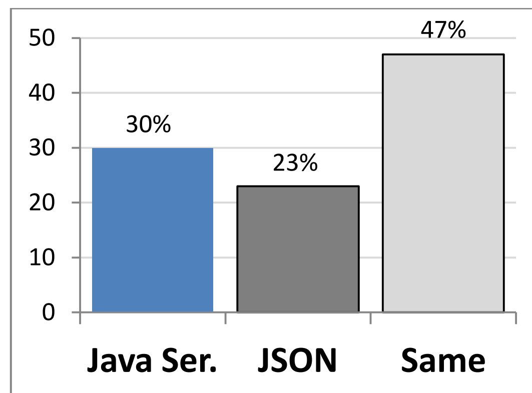

Figure 2: Perceived Difficulty of Software Tool: Java vs. Json

The second part of the questionnaire asked to identify only one of the two tools as the preferred software tool for future serialization work. From our pool of accepted assignments, 73% of the participants choose JSON and 27% favored Java JDK (only a few questionnaires were labeled as 'incomplete' and not considered in this work).

The goal of the third part of the assignment was to allow the participants to gain an appreciation for the relative speed of each method. A Department object holding a size-varying list of employees was used to estimate time performance. The personnel list held: 1K, 5K, 10K, 100K, and finally 500K randomly create employees. Students created the common data sets and used JS and JSON preserving schemes on the five Department samples. Figure 3 shows an example of the output supplied by one student.

Speed (# emps)	**Java** (*msec*)	**Json** (*msec*)
1K	152	251
5k	491	179
10k	800	202
50k	2,771	322
100K	5,509	636
500k	28,288	3,844

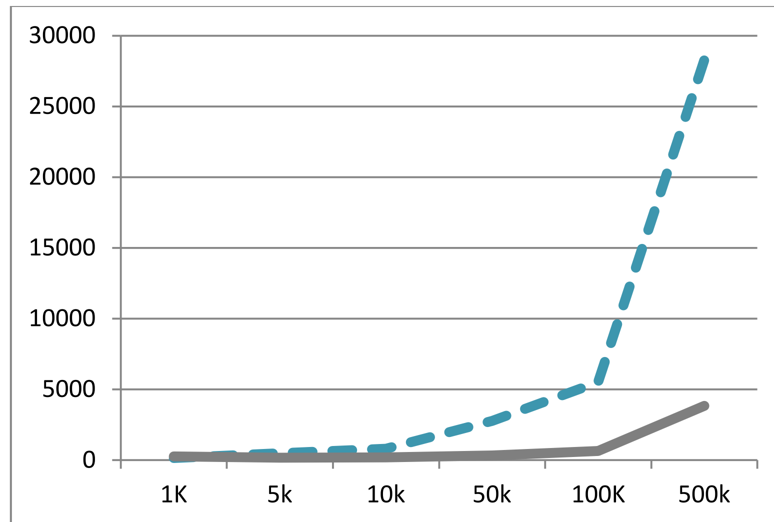

Figure 3: Benchmarking Execution of JS and JSON Serialization/ Deserialization Cycle Measured on Various Department Samples Where Personnel Size Changes from 1K To 500K Employees

4. 'Long Term' Retention Test

Three weeks after the students turned in their assignments, we gave them an unannounced pop-quiz. To motivate the students, their correct answers were considered for extra-points, while invalid answers had *no influence* on their final grades. The test had two questions, the first asked to preserve a simple Employee object, the second asked to do the same on an ArrayList of Employees (this pattern was identified during the first part of the experiment as the one single case in which JSON appeared to be harder to use than JS, see Figure 2).

A total of 40 students took the test, 16 of them used JSON, 12 used Java, and 12 use an unacceptable/wrong approach. The average grade of JSON users was 75% while JS users obtained a 54% grade.

An interesting fact is that we anticipated a 3:1 rate of JSON vs. JS followers, however in the test we found that only 40% of the takers preferred JSON, while 30% chose JS. This rate of technology adoption is lower than expected, due to the drop from 70% to 40% among JSON followers. It is also significant the much better collective average grade obtained by JSON adopters (75%) as compared to the one obtained by JS adopters (54%)

5. Discussion

From Figure 2 we observe that almost half of the subjects (47%) considered the effort of using JS and JSON as *comparable*. The rest of the users, marginally qualify JSON as simpler than JS. A distribution of this kind should make you believe that both tools are so similar that it does not matter which to use. However, the almost 3:1 rate of adoption of JSON as future tool for object preservation (73% adopters) is a significant (an unexpected) result of this experience. From a software-writing standpoint this is quite surprising. Observe that both schemes almost require the same amount of statements to work, keep also in mind the following differences (1) JS requires classes to implement the Serializable interface, whereas JSON does not, (2) JSON must import an external library while JS does not.

The only case in which a significant advantage appears to favor JS emerges when an ArrayList of objects is encoded (see Figure 2, case 2). Almost twice the programmers picked JS as simpler than JSON. This is not a surprise remember that the JSON solution uses the TypeToken clause which adds to the complexity of this case. Notwithstanding this exception, the overall choice is in favor of JSON.

Figure 3 suggests a significant advantage of JSON vs. JS in terms of computing time. For small samples the two techniques are quite similar, however as object size increases JS becomes significantly slower than JSON. Not shown in our experiment is the fact that space requirements are different. JS produces smaller disk images than those equivalent datasets encoded in JSON. Therefore speed-wise JSON is better than JS, and for space-wise demands the situation is the opposite (perhaps it is important to recognize that various students reached this conclusion although space consumption was no included in the original study).

6. Conclusion

Java JDK Serialization is perhaps the most common strategy used in the Java programming world to provide object persistence. This software approach is usually included in CS1 and CS2 textbooks. However, from this experience we believe that the choice of JSON to educate novice developers on the subject of object persistence should be preferred over the classical Java JDK serialization technique.

Our evidence is empirical and our test pool small, however it points to the user's preference in favor of the JSON scheme on both scenarios: (a) ideal working conditions defined as a stress-free deadline scenario, with the developer having access to notes and any other documentation, and, (b) situations in which the programmer is under pressure to solve a problem quickly and does not have access to supporting documentation.

We believe the preference of JSON over JS, is primarily due to the easier to remember human-readable nature of the JSON scheme, as well as its capability to be employed as a data-exchange option in complex applications involving other than Java programming language. It is also important to acknowledge the better retention of JSON notation over JS. In light of these findings, perhaps programming instructors should experiment with JSON instead of classical JS as a better pedagogical tool.

6. References

Singh, I. & Leitch, J. & Wilson, J. Ed. (2013). Google-Gson: A Java library to convert JSON to Java objects and vice-versa. Retrieved July 9, 2014 from https://code.google.com/p/google-gson/

Bray, T. (2014) The JavaScript Object Notation (JSON) Data Interchange Format. Retrieved July 9, 2014, from http://tools.ietf.org/html/rfc7159

Microsoft - An Introduction to JavaScript Object Notation (JSON) in JavaScript and .NET (2007). Retrieved July 9, 2014, from http://msdn.microsoft.com/en-us/library/bb299886.aspx

HBO - The Game of Thrones Viewer's Guide (2014). Retrieved July 9, 2014 from http://viewers-guide.hbo.com/game-of-thrones/season-4/episode-10/houses

Oracle Corp. Java Object Serialization Specification Version 6.0 (2005). Retrieved July 9, 2014, from http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial-arch.html#4176

Jackson Project (2014). Retrieved July 9, 2014, from https://github.com/FasterXML/jackson

Appendix A. Object Persistence - Learning GSON

The following code fragments use the Person class depicted in Figure 1 and fictional characters from (HBO 2014). This set of code fragments is not an exhaustive review of all features of JSON encoding. For additional material see reference (Singh & Leitch & Wilson, 2013)

1. Encoding Simple Data Types.

```
// Create an instance of a Gson-JSON encoder
Gson gson = new Gson();                    // must import  com.google.gson.* API

// PART1. serialize primitive data elements (strings, numbers)
String jsonString = gson.toJson("Game of Thrones");
String javaValue1 = gson.fromJson(jsonString, String.class);
System.out.println(javaValue1);            // prints Game of thrones

// PART2. deserialize primitive data elements
String jsonString2 = gson.toJson(4321);
Integer javaValue2 = gson.fromJson(jsonString2, Integer.class);
System.out.println(jsonString2);           // prints 4321
```

Java and JSON represent strings and numbers the same way; strings are surrounded by double quotes, numbers are not.

2. Encoding Simple Objects

```
// PART1. Serialize a single Person object
Person person0 = new Person("Daenerys Targaryen", 18);
jsonString = gson.toJson(person0);
System.out.println("Json string: " +  jsonString );

// PART2. Deserialize a single Person object
Person personDecoded = gson.fromJson(jsonString, Person.class);
System.out.println("Java object: " +  personDecoded );
```

The previous fragment produces the following output:

```
Json string: {"name":"Daenerys Targaryen","age":18}
Java object: Person(name=Daenerys Targaryen, age=18)
```

JSON objects are delimited by enclosing curly-braces ({ }). For each (non-static, non-transient) class variable the encoded JSON string includes a comma-separated <key, value> pair indicating the variable's name (inside quotes) and its corresponding value. In our example the object's data is annotated in the JSON string as the pairs "name":"Daenerys Targaryen", "age":18.

3. Encoding a List of Objects

```
// PART1. create a (Generic) list of Person objects
ArrayList<Person> lstPerson = new ArrayList<Person>();
lstPerson.add(new Person("Daenerys Targaryen", 17));
lstPerson.add(new Person("Tiryion Lannister", 30));
lstPerson.add(new Person("Arya Stark", 11));

// convert Java collection to JSON string
String jsonList = gson.toJson(lstPerson);
System.out.println("Json list: " + jsonList );

// PART2. use Java reflection to find the list's type
Type arrPersonType = new TypeToken<ArrayList<Person>>(){}.getType();
ArrayList<Person> lst2 = gson.fromJson(jsonList, arrPersonType);
System.out.println("Java list:" + lst2 );
```

The list of Person objects is encoded as follows:

```
Json list: [{"name":"Daenerys Targaryen","age":18}, {"name":"Tiryion Lannister",
"age":30},  {"name":"Arya Stark","age":11}]
```

Observe that the array is delimited by square-braces ([ ]). Each element in the array is a Person object including the attributes "name" and "age". Objects are enclosed inside curly-braces({}) and separated by commas. The Gson TypeToken class is used to extract the collection's generic type information needed to decode the JSON string and regenerate the appropriate Java object. A word of caution, an ArrayList() with multiple type of objects cannot be serialized. You must commit to a particular type T in the definition of Arraylist<T>.

4. Encoding a Subclass

Assume the PersonHouse class is an extension of the Person class. It contains the additional house attribute. Consider the following example:

```
// PART1. Encode subclass
PersonHouse person4 = new PersonHouse("Jon Snow", 23, "Stark");
String jsonPerHome = gson.toJson(person4);
System.out.println("Json subclass: " + jsonPerHome);

// PART2. Decode string
PersonHouse person4again = gson.fromJson(jsonPerHome, PersonHouse.class);
System.out.println("Java subclass: " + person4again);
```

The output produced by this fragment is

```
Json subclass: {"House":"Stark","name":"Jon Snow","age":23}
Java subclass: PersonHouse(House=Stark Person(name=Jon Snow, age=23))
```

5. Encoding a (1: Many) Relationship

Assume the existence of a House class including the properties: houseName and characters (represented as a dynamic list of Person objects).

```
ArrayList<Person> theStarks = new ArrayList<Person>();
theStarks.add(new Person("Catelyn Stark", 40));
theStarks.add(new Person("Sansa Stark", 14));
theStarks.add(new Person("Bran Stark", 9));

//PART1. Encoding complex 1:m class
House starkHouse = new House("Stark", "Winterfell", theStarks);
String jsonHouse = gson.toJson(starkHouse);
System.out.println("Json House: " + jsonHouse);

//PART2. Decoding complex 1:m class
House javaHouse = gson.fromJson(jsonHouse, House.class);
System.out.println("Java House: " + javaHouse);
```

The output produced by this example is:

Json House: {"houseName":"Stark","location":"Winterfell","personLst":[{"name":"Catelyn Stark","age":40},{"name":"Sansa Stark","age":14},{"name":"Bran Stark","age":9}]}

Java House: House(houseName=Stark, location=Winterfell, personLst=[Person(name=Catelyn Stark, age=40), Person(name=Sansa Stark, age=14), Person(name=Bran Stark, age=9)])

Keep in mind that classes participating in a circular reference cannot be encoded. For instance, each Employee instance could refer back to his/her department by including a key such as deptName, but not a direct reference to a department object.

Appendix B. Parsing a JSON string

The following is a custom-made alternative decoding approach based on traversing and parsing the encoded data structure looking for the possible occurrence of jsonElements (which could be: jsonObject, jsonArray, or jsonPrimitive tokens)

```
private static void jsonHouseNodeParsing(String jsonHouseStr) {
    String result = "";
    try {
        JsonElement jelement = new JsonParser().parse(jsonHouseStr);
        JsonObject jobject = jelement.getAsJsonObject();

        String jHouseName = jobject.get("houseName").toString();
        String jLocation = jobject.get("location").toString();
        JsonPrimitive jLocation = jobject.getAsJsonPrimitive("location");
        System.out.println(jHouseName + "\n" + jLocation);
        JsonArray jarray = jobject.getAsJsonArray("personLst");

        for (int i = 0; i < jarray.size(); i++) {
            jobject = jarray.get(i).getAsJsonObject();
            result = jobject.get("name").toString() + " "
                        + jobject.get("age").toString();
            System.out.println(" " + result);
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }

}// jsonNodeParsing
```

For example:

JSON encoded string	Decoded Nodes
{"houseName":"Stark", "location":"Winterfell", "personLst":[{"name":"Catelyn Stark","age":40}, {"name":"Sansa Stark","age":14}, {"name":"Bran Stark","age":9}]}	"Stark" "Winterfell" "Catelyn Stark" 40 "Sansa Stark" 14 "Bran Stark" 9

Appendix C. Student's Questionnaire

Name: ____

Tasks:

1. Implement the Department, Employee,and PartTimeEmployee Java classes depicted below.
2. You will write a JS and a JSON solution for each of the five cases listed in Table1.
3. For CASE5 (see Table1) write a method to create and populate a Department object to which you will add as many employees as indicated in Table 3. The data for each employee will be randomly generated. Each time an employee is created call the .addEmployee(emp) method to attach the employee to the department. Test this process with different samples. Each sample consists of a single department associated to: 1000, 5000, 10000, 50000, and 100000 employees. Tabulate the results obtained from each sample using JS and JSON.
4. Provide your evaluation of programming difficulty for JS and JSON on each of the cases listed in Table1.
5. In case you have to choose only one of the two strategies, indicate which you prefer? (Table2)

Perceived Difficulty (Check **HARDEST** to use)	**Java Serialization**	**JSON Encoding**	**Same**
CASE1: Encode a string value			
CASE2: Simple objects (use Employee class)			
CASE3: List of objects (ArrayList< Employees>)			
CASE4: Subclasses (PartTimeEmployee)			
CASE5: 1:many relationship (Department class)			

Table1: Perceive Difficulty of JS and JSON Encoding

	[Check Mark Only One] Which strategy do you prefer to use in the future?
Java Serialization	
JSON Encoding	

Table 2: Preferred Overall Encoding Strategy for Future Use

Relative Performance. Preserving a Department object having different number of employees. Performance is recorded in *msec* and must include: encode + decode + IO Write&read ops.	**Java Serialization**	**JSON Encoding**
1k (Number of the Employees in the list)		
5k		
10k		
50k		
100K		
500k		

Table 3: Combined Serialization-Deserialization Execution Time for Various Samples

Employee-Department Class Diagrams