FACULTY OF ENGINEERING AND ARCHITECTURE

DEPARTMENT INTEC
RESEARCH GROUP IBCN

Geavanceerde softwareontwikkeling

Lab Session I

Inheritance vs Composition

GHENT UNIVERSITY

Introduction to JSON

JSON (JavaScript Object Notation) is a lightweight, language independent data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language.

JSON succeeds XML as the most popular choice for data representation format, largely due to the rise of JavaScript (which natively supports JSON) both on the client-side (mobile & web applications) as on the server-side (e.g. Node.js). The format is also frequently used in conjunction with REST APIs and web-based services (cfr. the Facebook and Twitter APIs).

To represent data as JSON, we can use two basic structures:

- A JSON object, which is a mapping of keys to values.
- A JSON array, which is an ordered list of values.

These structures can be nested, meaning arrays can consist of objects, while the value of a key in an object can be another object or an array, etc… Primitive JSON values (values that are not an object or array) are limited to boolean values (true/false), integral numbers (e.g. 1, -1, 1000) floating point numbers (e.g. 1.0, 3.1415) and string values.

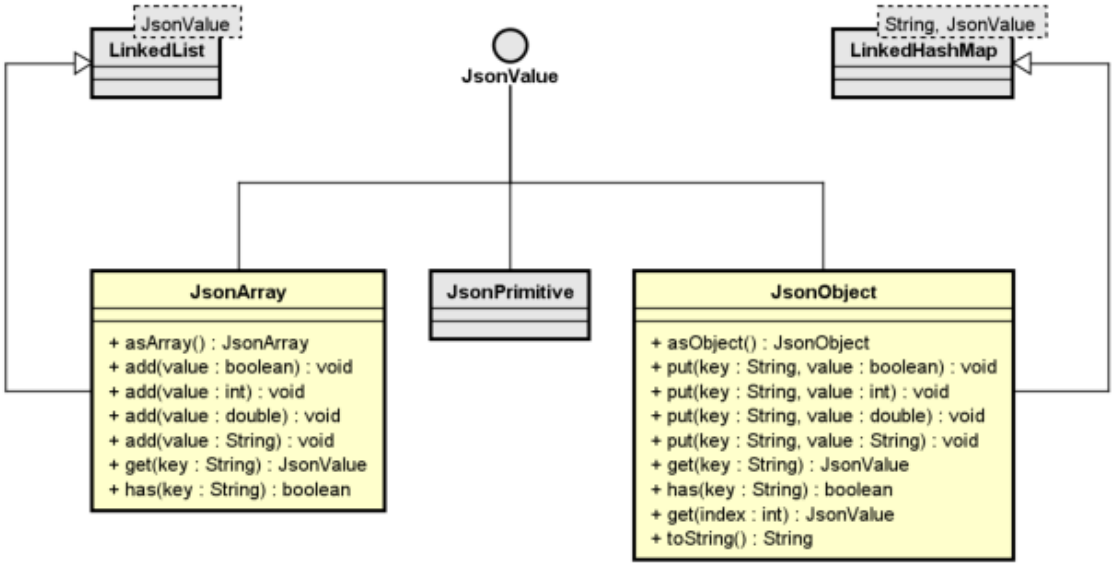
The following representation of a user account is a simple example of a JSON object:

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumber": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

As you can see, JSON objects are indicated using curly brackets ({}) containing a comma-separated list of key, value-entries where the key is put between quotes, followed by a colon and then the value of the entry. Arrays are indicated using square brackets ([]) containing a comma-separated list of values (other arrays, JSON objects or primitive types).

Implementing a simple JSON library

The goal of the first part of this Lab is to implement a simple JSON library that will allow us to represent and export JSON data in Java. The design of such a library relies heavily on the use of inheritance, abstract classes, interfaces and polymorphism, as shown in the class diagram below:



The JsonValue interface is the root of the hierarchy. It specifies all the common operations for any JSON type and is implemented by the two JSON structures, JsonArray and JsonObject, and by JsonPrimitive which will represent all the primitive types contained in the JSON data.

The interface JsonValue and the class JsonPrimitive are already implemented. We ask you to complete the library by providing classes for JsonArray and JsonObject. The good thing is you don't need to start from scratch:

- A JsonArray is an ordered list of JsonValue instances, so you can extend from LinkedList.
- A JsonObject is a mapping of keys to JsonValue instances, so you can extend from LinkedHashMap.

Most of the required methods for JsonArray and JsonObject are pretty straightforward and we've also provided Javadoc documentation for the JsonValue interface. Three methods that might need more explanation are:

- **asObject():** This method will only be implemented by JsonObject. Invocating this method on any other JsonValue instance should result in an UnsupportedOperationException. The goal of this method is to easily access object operations without needing to rely on casts (which are ugly and verbose).
- **asArray():** This method will only be implemented by JsonArray. Invocating this method on any other JsonValue instance should result in an UnsupportedOperationException.

- **get(index : int):** This utility method is also implemented for JsonObject and should return the value for the key with the specified index (this is easy to implement as the mapping is represented in an ordered fashion by extending from LinkedHashMap).

An important feature of our JSON library is the ability to export the JSON model to a textual representation so it can be stored in a file, or transferred using HTTP. Use the toString() function to implement this feature based on what you've learned about JSON in the introduction. You can validate the resulting output using one of the many validator applications you can find online (e.g. https://jsonformatter.curiousconcept.com/).

Finally, we've also provided a JsonObjectBuilder (which follows the builder design pattern that will be explained in one of the theory classes) to quickly build JSON objects. You can now test your JsonArray and JsonObject implementations by executing the following code in your main-method:

```
//Create a nested JSON object containing user records
JsonArray userRecords = new JsonArray();

userRecords.add(new JsonObjectBuilder()
 .add("id", 1)
 .add("name", "John Doe")
 .add("address", new JsonObjectBuilder()
  .add("street", "Abbey Road")
  .add("number", 7)
  .add("city", "London"))
 .build());

userRecords.add(new JsonObjectBuilder()
 .add("id", 2)
 .add("name", "Jane Doe")
 .add("address", new JsonObjectBuilder()
  .add("street", "Abbey Road")
  .add("number", 8)
  .add("city", "London"))
 .build());

//Print the records
System.out.println(userRecords);

//Retrieve the address of the first user
System.out.println(userRecords.get(0).get("address"));
```

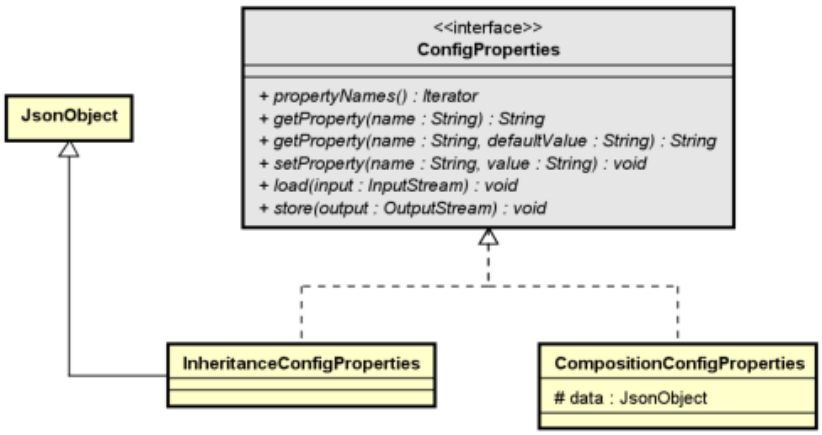
Notice how the structure of the JsonObjectBuilder code mimics the way the resulting JSON would look in formatted text-representation.

Storing JSON properties: Inheritance vs. Composition

In this second part of the Lab, we will use the JSON library to develop a mechanism for storing and loading application properties using JSON. The interface ConfigProperties describes the operations that are required for this type of mechanism:

- **propertyNames():** returns an iterator that iterates over the available property names in the configuration.
- **getProperty(name : String):** returns the property value (a String) for the specified property name.
- **getProperty(name : String, defaultValue : String):** returns the property value (a String) for the specified property name. If no value can be found for this property name, the supplied default value is returned.
- **setProperty(name : String, value : String):** sets the specified value for the specified property name.
- **load(input : InputStream):** loads a configuration from the specified InputStream.
- **store(output : OutputStream):** writes a configuration to the specified OutputStream.

We ask you to provide two alternative implementations for the ConfigProperties interface: one based around inheritance and one using composition (as shown in the diagram below).



Note: (de)serializing properties are beyond the scope of this labo, so you can just provide an empty implementation for the load and store methods!

It seems logical to start with the inheritance-based solution. After all, we quickly managed to implement the JsonObject class by extending LinkedHashMap and a mapping of config keys to config string values that should be stored as a JSON text-representation is actually nothing more than a specialized version of a JsonObject right?

Unfortunately, the inheritance-based solution has some undesired side-effects. Take a look at the following code using this implementation:

```
InheritanceConfigProperties props1 = new InheritanceConfigProperties();
props1.setProperty("key1", "value1");
//The JsonObject interface is exposed as well, so we can do invalid operations like:
props1.put("key2", false); //Properties file should not be able to store booleans!

//Print its properties
Iterator < String > iter1 = props1.propertyNames();
while (iter1.hasNext()) {
 String name = iter1.next();
 //This still works because the second key is automatically converted to String
 System.out.println("props1." + name + " = " + props1.getProperty(name));
}
```

By extending the JsonObject class, we're exposing the JSON Object API as part of the ConfigProperties implementation. Users can perform operations such as storing a Boolean, which will work, but is not actually the intended behavior of the class. This in turn will lead to breaking changes when we would decide to change the internal representation of our ConfigProperties implementation[1].

For this particular case, the composition-based solution is certainly to be preferred. The internal representation of the ConfigProperties implementation is nicely encapsulated in the protected field of the implementing class. Users of the class can only see the intended API of the ConfigProperties interface and changes to the internal representation can be made transparently without introducing breaking changes. Code usage example:

```
CompositionConfigProperties props2 = new CompositionConfigProperties();
props2.setProperty("key1", "value1");
//Only the ConfigProperties interface is exposed, can't set a boolean:
//props2.put("key2", false);
props2.setProperty("key2", "false");

//Print its properties
Iterator < String > iter2 = props1.propertyNames();
while (iter2.hasNext()) {
 String name = iter2.next();
 System.out.println("props2." + name + " = " + props2.getProperty(name));
}
```

[1] One could remark that this is also solved by using the ConfigProperties interface instead of the implementation class directly. This is indeed another very useful form of encapsulation that can help you a lot in large projects. However, concrete classes being used as public API is something that occurs in practice and in that case composition is definitely to be preferred over inheritance. A real life example of a design flaw is the java.util.Properties class (see https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html) which we used as inspiration for this Labo example.

Submitting your solution

1. Important: labs are solved individually
2. Bundle your source solution files in one zip file named ***lab1_name_surname.zip*** (e.g. *lab1_bruno_volckaert.zip*)
3. List of files to be included in the zip:
    a. JsonArray.java
    b. JsonObject.java
    c. InheritanceConfigProperties.java
    d. CompositionConfigProperties.java
4. Send this zip-file via Minerva dropbox to **Wannes Kerckhove, Thomas Dupont** and **Bruno Volckaert**, deadline for sending this is exactly one week from the start of this lab. I.e. for this lab the deadline is Friday 13th of October at 15:59.