Project: Javascript Object Notation

Description

Javascript Object Notation (JSON) is a format for exchanging and storing data. JSON is similar to XML but requires less information to encode the same information. The main idea of JSON is to store key-value pairs as text. The key describes a type of information (e.g., "name") and the value gives the data for that particular object (e.g., "Mike"). For instance, http://www.reddit.com/.json gives the front-page of Reddit as a JSON string.

The specification of JSON is given in terms of **objects**, **arrays** and **key-values**.
The following specification is a slight simplification of the real JSON standard:

1. an **object** in JSON has the format { key : value , key : value , … }. There can be zero or more key-value pairs (see #2 and #4) separated by commas.

2. a **key** is a string (see #3).
3. **strings** are always surrounded by quotation marks. In this assignment, they contain ASCII characters and spaces only. Strings will not contain quotation marks.

4. a **value** (in #1) can be:

    a. the constants true and false.

b. a string (see #3)

c. a number (floating point or integer, without quotation marks)

d. an object (i.e., an entire JSON object surrounded by { }, see #1)

e. an array (see #5).

5. an **array** has the format [value , value , …]. There can be zero or more values separated by commas.

I assume there is always exactly one space between commas, colons, square brackets and braces. Further, strings have no space between the quotation mark and the first/last character of the identifier (i.e., strings have the form "name" rather than " name " with a space before n and a space after e).

As an example, here is a JSON object:

```
{ "name" : "Anne Vestor" , "userid"
    : "avestor", "cash on hand" :
    10000,
    "portfolio" : [ {"name": "BTC" , "quantity" : 3 } , {

"name": "ETH", "quantity": 5 } ]

}
```

Interface 1: Value

```
public interface Value {

    String toString();

    boolean equals(Value v);

}
```

The class implementing the Value interface should store anything that can be represented as a value in the description of a JSON object (see #4 in the description of JSON above; that is, a value can be an array, an object, a boolean value, a number or a string). ToString method should replicate the structure of a JSON value as closely as possible to the input format given for JSON: output on a single line, with spaces between tokens.

The interface will be extended by JSONObject and JSONArray (below) but also should be implemented by (**at least**) four concrete classes for storing booleans, integers, doubles and Strings from item #4a-4c above (see the Factory section below for details on how the concrete classes are created).

Interface 2: JSONObject

```
public interface JSONObject extends Value { void
addKeyValue( Value key, Value v); Value getValue (Value
key);

    JSONIter iterator();

}
```

The JSONObject interface represents the types of structures generated by #1 in the JSON specification above.

The method addKeyValue add a key-value pair to the JSON Object. By definition, the key-value pair is added after all key-value pairs previously added. If there is already a key-value pair with the same key, then the value is updated to the new value.

The method getValue should take a key and return the value associated with the object (if the key does not exist in the JSON Object, return null).

Two JSONObjects are equal if they have the same set of keys and if, for each key, the value associated with a key in one JSONObject is the same as value associated with it in the other JSONObject.

For information on the iterator method, see the section on iterators below.

Interface 3: JSONArray

```
public interface JSONArray extends Value { void
    addValue(Value v);

    JSONIter iterator ();

}
```

The JSONArray is a particular type of interface that can represent any JSON Array, a particular type of Value. This represents the types of structures generated by #5 in the JSON specification above. The method addValue should add a value to the array. By definition, the value is added after all values previously added. Duplicate values in an array are allowed – each new value is simply added to the end of the JSON array.

Two JSONArrays are equal if they have the same Values in the same order.

For information on the iterator method, see the section on iterators below.

Iterators

I created two iterators for the two classes that satisfy the JSONObject and JSONArray interfaces. An iterator is an object that allows a user to traverse a set of data without knowing anything about the underlying representation.

```
public interface JSONIter {

    boolean hasNext();

    Value getNext();

}
```

The methods work as follows:

- When initialized, an iterator starts at the start of the collection. In this case, this is at first element added to a JSON Object or JSON Array by the add methods described in the sections above.

- The hasNext() method tests whether another element exists in the collection after the current position. It returns true if there is another element and false otherwise.

- The getNext() method gets the next element in the collection and advances the position in the collection to the next element. If it is called when no more elements exist (i.e., when hasNext() returns false), then it should return null.

- If an iterator is initialized on an empty collection, the methods behave as if the end of the collection has already been reached, i.e., getNext() returns null and hasNext() returns false.

A typical loop involving an iterator would look like this:

```
JSONIter iter = jsonObject.iterator(); while
(iter.hasNext()) {

    Value v = iter.getNext();

    // use v ..

}
```

Two different classes that implement this interface: one that iterates over a JSONArray (for which getNext() returns Values that are **the elements of the array**) and one that iterates over a JSONObject (for which getNext() returns Values that are **the keys of key-value pairs stored in an object**).

Create Factory

To properly manage the interface between the interfaces and the concrete classes implemented for these interfaces, I defined a small class called JSONFactory that has three methods:

- public static JSONObject getJSONObject()

- public static JSONArray getJSONArray()

- public static Value getJSONValue(ValueEnum v, Object o)

The first two of these methods should simply return a new (empty) object of the concrete types that implement the appropriate interfaces. For instance, if project uses class **X** to implement the interface JSONArray, then the method getJSONArray should only create a new (empty) instance of class **X** and return it.

For the third method, return a new object of one of the four "primitive" types of Values in JSON Objects: boolean, int, double and string. The first parameter should be the type of the Value, which is defined by the ValueEnum enumerated type:

```
public enum ValueEnum {

    BOOL, INT, DOUBLE, STRING;

}
```

The second parameter is a Java Object whose dynamic type is one of four concrete Values in JSON Objects (Boolean, Double, Integer and String). The dynamic type of the Object o will always match the value of the enumerated type v.

This class is necessary since JSONParser.java (which is provided to you) does not know the name of the classes you have used to implement the Value, JSONObject and JSONArray interfaces.

Testing

Construct a set of at least **ten** different unit tests for the JSONParser in combination with your code. To do unit testing in Java

Part 2: Querying JSON Objects

.A query is a command that asks for information about a JSON Object. The queries have the form

“key1.key2….keyn”

which is interpreted as “find the key labelled key1, then within its value, find the key labelled key2, … then find the key labelled keyn, and return its value". For arrays, the key will also include an index in [square brackets]. So, an array key would read key[j], meaning “find key, then in the array of values corresponding to key, find element j."

For instance, if you had the following JSON Object

```
{ "store": {

    "book": [

      { "category": "reference", "author": "Nigel Rees",
        "title": "Sayings of the Century",

        "price": 8.95

      },

      { "category": "fiction", "author":
        "Evelyn Waugh",
        "title": "Sword of Honour",

        "price": 12.99

      },

      { "category": "fiction", "author": "Herman
        Melville", "title": "Moby Dick", "isbn":
        "0-553-21311-3",

        "price": 8.99

      },

    ],

    "bicycle": {
      "color": "red",

      "price": 19.95

    }

  }

}
```

The result of the query “store.book[1].category” would be “fiction”. The result of “store.book[2].title” would be “Moby Dick”. A few other notes and simplifications about queries:

- A query could fail to have an answer (for instance, “store.unicorn.breed” in the above example).
- A query could return an array of values (for instance, “store.book”) or a JSON Object (for instance, “store.book[0]”)
- Queries do **NOT** contain quotation marks. So a query looks like "store.book" and not look like ""store"."book"".

This query language is a simplified version of the JSONpath language. The example above is taken from this site: http://goessner.net/articles/JsonPath/, used under Creative Commons License.

Handling Queries

The interface is called the JSONQueryManager, which allows you to load a JSON object and then make several queries about the JSON object.

To implement this query manager, I implemented the following interface:

```
public interface JSONQueryManager{

    public void loadJSON (String JSON);

    public Value getJSONValue (String query) throws

IllegalStateException;

}
```

The methods are described as follows:

- loadJSON: This method takes a description of a JSON object. The program should then be allowed to make queries on the JSON object.

- getJSONValue: This method returns the JSON value associated with a particular query. The value should be returned as an object satisfying the JSON Value interface from Part 1. If the query is not successful, return null.

If a getJSONValue() call is made without a loadJSON() call previously, the method should throw a IllegalStateException.