Due:

March 18, 2019, **before 11:59 pm.**

Notes

- Please follow both the "Programming Standards" and "Assignment Guidelines" for all work you submit. Programming standards 1-25 are in effect.
- Hand-in will be via the D2L Dropbox facility. Make sure you leave enough time before the deadline to ensure your hand-in works properly.
- All assignments in this course carry an equal weight.

Assignment 3: 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.

In our specification, you can 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 } ]
}
```

This object represents the type of information that you would have been associated with an Investor in Assignment 1. The object is written on several lines for ease of reading, but will typically be represented by a single string on one line. (Recall that in Java, if you have String variable, you must escape the quotes, i.e., each quote should be given as \".)

In this assignment, you will implement a JSON query tool in Java.

Part 1: Implement JSON Storage code (50 % of assignment value)

You are provided with the file JSONParser.java, which parses Strings into JSON structures. The code **requires some additional classes** that are responsible for the storage of the JSON Object after it is read from the String. In particular, you need to implement three interfaces, two iterators and one Factory.

Once you have implemented these methods, you can test the parser by calling the JSONParser.read() method and passing a JSON string as a parameter.

Interface 1: Value

You need to implement the Value interface:

```
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). Your 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).

Your equals() method should check if the parameter of type Value has the same concrete type as the current object, and then test the contents. For the concrete classes for booleans, integers, doubles and Strings, this should test equality of the contents. For arrays and objects, see the description of equality testing below.

Interface 2: JSONObject

You need to implement the JSONObject interface:

```
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 should 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

You need to implement the JSONArray interface:

```
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

You should also create 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. You should implement the following iterator interface:

```
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 ..
}
```

Note that you may have to write 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 you implement for these interfaces, you must define 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 your 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, you should 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 that you want to return, 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

You have been provided with the JSONParser.java file for the assignment. Do not modify this file – your code **must** work with the original file provided with the assignment. When testing your file after your final submission, an original copy of the file will be used for testing, so your code must work with this version.

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

1. create a new class to test the parser.
2. Import: "`import static org.junit.Assert.*;`"
3. Create public void methods in your class to test conditions about the parser/data structure combinations.
4. Annotate each test method with the line "`@Test`" before the method.

At least five of your tests should test boundary conditions: an empty JSON Object, and a JSON Object with only one key-value (for each possible concrete Value type defined by the

ValueEnum enumerated type). You will be graded on your tests, so write useful tests for all unit tests.

The markers will be running your tests, so ensure that they pass.

Part 2: Querying JSON Objects (50 % of the assignment value)

After implementing the back-end of the JSON Parser in Part 1, you should implement a tool for making JSON queries. 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

After implementing the interfaces required in Part 1, you need to implement an interface that allows JSON strings to be parsed and queried by another class. 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, your code should implement 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. You are responsible for providing an appropriate message to the constructor for this exception.

Add the following method to your JSONFactory class:

- `public static JSONQueryManager getJSONQueryManager()`

As with the other two factory methods, this method should only return a new instance of the concrete class implementing the JSONQueryManager interface.

Testing

To test Part 2, the markers will give your code several commands that satisfy the JSONQueryManager interface. You will not be given the file that will be used for this testing. You are expected to write code that satisfies the interface without knowing the particular code that will be used to test it. This will simulate building a general-purpose tool that will be used by users whose queries are unknown to you.

Construct a set of at least **five** different unit tests for the JSONQueryManager.

Data Structures

As with Assignments 1 and 2, you should not use any arrays in your code except for minor input parsing (.split) if necessary. You should also not use any Java Collections data structures. All data structures should be linked data structures that you write.

Hand-in

Submit all your source code for all classes. Submit all files on umlearn.

Submit a single zip file only. Do not submit any of the provided files (interfaces, enums). You only need to submit and document your own code. Submit the files that include your JUnit tests.

Note: there are no main methods for your assignment. Just submit all of your source code including your JUnit Tests. The markers will have separate tests that they will use to check your code.