Object-oriented programming with Java

Exam programming task

JSON (JavaScript Object Notation) is a lightweight, text-based, language-independent data exchange format that is easy for humans and machines to read and write. JSON can represent two structured types: *objects* and *arrays*. An object is an unordered collection of zero or more name/value pairs. An array is an ordered sequence of zero or more values. The values can be *strings*, *numbers*, *booleans*, *null*, and these two structured types. Listing is an example from Wikipedia that shows the JSON representation of an object that describes a person. The object has string values for first name and last name, a number value for age, an object value representing the person's address, and an array value of phone number objects.

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

JSON is often used in Ajax applications, configurations, databases, and RESTful web services. All popular websites offer JSON as the data exchange format with their RESTful web services.

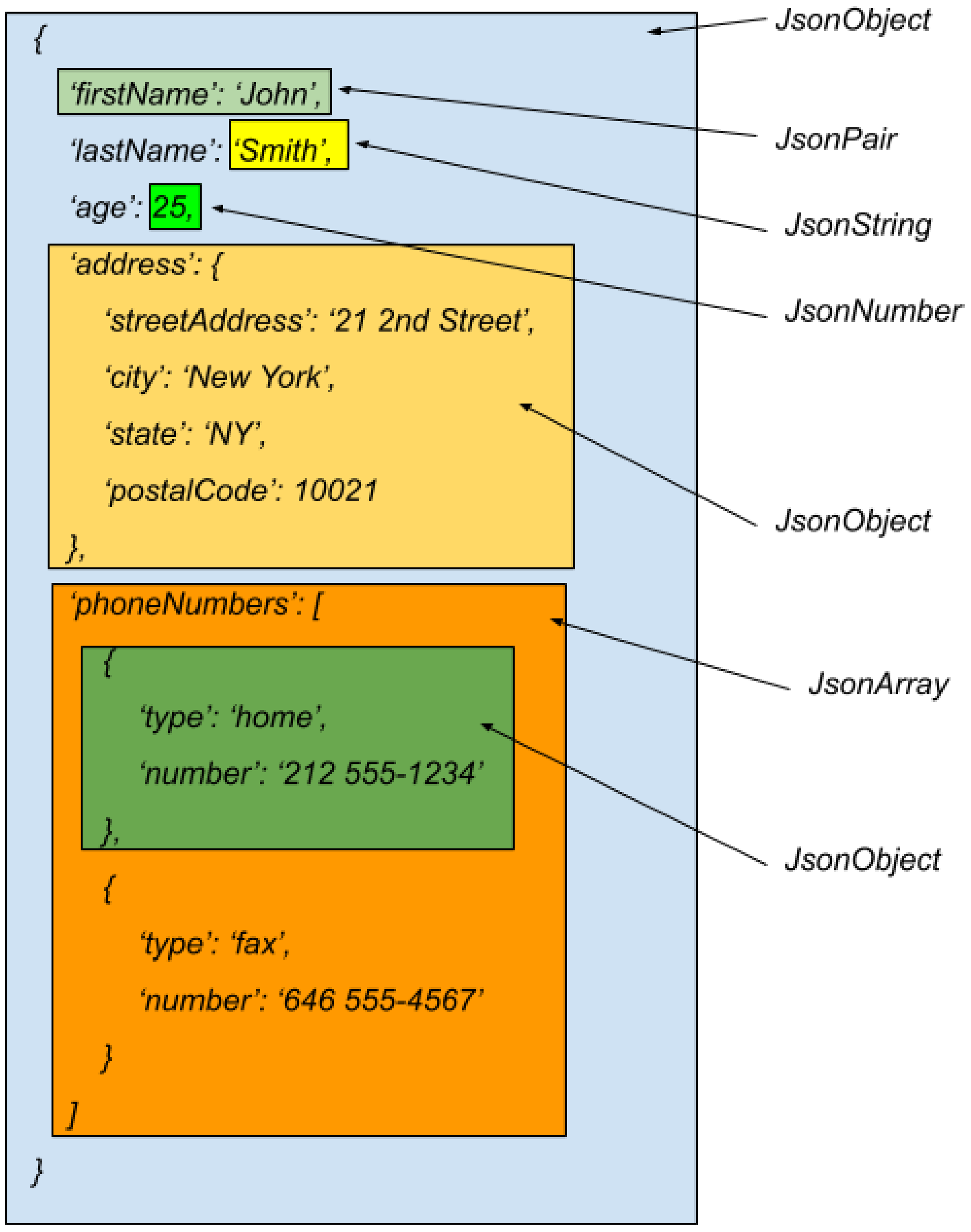
JSON representation as an object-oriented model
To represent JSON as an object-oriented model in Java we can map each JSON data type to Java class, and then use Decorator pattern to “wrap” primitive JSON data types (number, string, boolean, null) into JSON array and/or object.

Open and examine provided project, starting from *json* package
JSON's basic data types and their mapping to Java classes with constructors are:

- Number -> *JsonNumber(Number number)* (already implemented)
- String -> *JsonString(String string)* (already implemented)
- Boolean -> *JsonBoolean(Boolean bool)* (not implemented)
- Array -> *JsonArray(Json… jsons)* (already implemented)
- Object -> *JsonObject(JsonPair… jsonPairs)* (not implemented)
- Null -> *JsonNull()* (not implemented)

JsonPair is an auxiliary class for representing name/value pair - *JsonPair(String name, Json value):*

```
public class JsonPair extends Tuple<String, Json>{
   public JsonPair(String name, Json value) {
       super(name, value);
   }
}
```



All these classes (except *JsonPair*) extend *Json* abstract class:

```java
public abstract class Json {
   public abstract String toJson();
}
```

toJson() method converts current data type to JSON representation - specially formatted string.
For example *JsonNumber* class has the following implementation:

```java
public class JsonNumber extends Json {
   private final Number number;

   public JsonNumber(Number number) {
       this.number = number;
   }

   @Override
   public String toJson() {
       return number.toString();
   }
}
```

and the following usage:

```java
Json jYear = new JsonNumber(2);
print(jYear); // 2
```

According to definition *JsonArray* can hold zero or more items of JSON's data types. For example, list of marks will be represented as a:

```java
Json jMarks = new JsonArray(new JsonNumber(3), new JsonNumber(4));
print(jMarks); // [3, 4]
```

To convert *JsonArray* to JSON's representation you should convert each of its elements to this format and then join them separating by delimiter (comma), and finally put all this text to square brackets.

JsonObject holds an unordered collection of zero or more name/value pairs - *JsonPair* objects:

```java
JsonPair name = new JsonPair("name", new JsonString("Andrii"));
JsonPair surname = new JsonPair("surname", new JsonString("Rodionov"));
JsonPair year = new JsonPair("year", jYear);
JsonPair marks = new JsonPair("marks", jMarks);
JsonObject jsonObj = new JsonObject(name, surname, year, marks);
print(jsonObj); // {'name': 'Andrii', 'surname': 'Rodionov', 'year': 2, 'marks':
[3, 4]}
```

toJson() method for *JsonObject* can return name/value pairs (in any order).
name separated from value by “: “. Since value is a Json type you should call *toJson()* method for it too.
name/value pairs should be separated by comma with space, and all this text should be enclosed into braces.

Tasks

Part 1

1. Implement *JsonBoolean* and *JsonNull* classes. Tests that should pass: `json.JsonBooleanTest`, `json.JsonNullTest`
2. Implement *JsonObject* class constructor and *toJson()* method (you can use standard Java collections). The following tests should pass:
   2.1. For zero name/value pairs - `json.JsonObjectToJsonTest.testToJsonWithZeroPairs()`
   2.2. For one name/value pairs - `json.JsonObjectToJsonTest.testToJsonWithOnePair()`
   2.3. For two name/value pairs with different names and type of values - `json.JsonObjectToJsonTest.testToJsonWithTwoPairs()`
   2.4. For several name/value pairs with different names and type of values - `json.JsonObjectToJsonTest.testToJsonPairsWithSameNamesButDifferentValues()`

3. If two name/value pairs have the same name, the second pair should overwrite value from the previous pair - `json.JsonObjectToJsonTest.testToJsonPairsWithSameNamesButDifferentValues()`

Part 2

Write `app.JSONApp.sessionResult()` method body, returning the following *JsonObject*

```
{
  'name': 'Andrii',
  'surname': 'Rodionov',
  'year': 2,
  'exams': [
          {
            'course': 'OOP',
            'mark': 3,
            'passed': true
          },
          {
            'course': 'English',
            'mark': 5,
            'passed': true
          },
          {
            'course': 'Math',
```

'mark': 2,
'passed': false
}
]
}

Test that should pass - `app.JSONAppTest.testSessionResult()`

Part 3

Very often we need to represent our Java objects as JSON to transfer them via RESTful web services or save them into special NoSQL DBs (for example MongoDB).
So if some domain object can be represented in a JSON format it should implement `Jsonable` interface:

```
public interface Jsonable {
   public JsonObject toJsonObject();
}
```

`toJsonObject()` method returns *JsonObject* representation of the current object. For example:

```
public class BasicStudent implements Jsonable {

   protected final String name;
   protected final String surname;
   protected final Integer year;

   public BasicStudent(String name, String surname, Integer year) {
       this.name = name;
       this.surname = surname;
       this.year = year;
   }

   @Override
   public JsonObject toJsonObject() {
       // should return JsonObject representation of BasicStudent
       return null;
   }
}
```

When we create *BasicStudent* instance and call **`toJsonObject()`** method, expected result should be the following:

```
BasicStudent basicStudent = new BasicStudent("Andrii", "Rodionov", 2);
print(basicStudent.toJsonObject()); // {'name': 'Andrii', 'surname': 'Rodionov',
'year': 2}
```

So your task is to implement **`toJsonObject()`** method for *BasicStudent* class. After this the following test should pass - `domain.BasicStudentTest.testToJsonObject()`

Part 4
To improve *JsonObject* class usability we decided to add some additional methods:

- *void add()* - add new name/value pair to existing *JsonObject*. If pair with the same name already exists, its value should be updated
- *boolean contains(String name)* - returns true if this *JsonObject* contains the specified name/value pair
- *Json find(String name)* - returns *Json* value for specified name, otherwise returns null
- *JsonObject projection(String… names)* - returns new *JsonObject* containing only pairs matching specified names. If no pairs match - should return empty *JsonObject*. For example:

```
JsonObject jsonObject = new JsonObject(
        new JsonPair("name", new JsonString("Andrii")),
        new JsonPair("surname", new JsonString("Rodionov")),
        new JsonPair("year", new JsonNumber(2)),
        new JsonPair("marks",
                new JsonArray(
                        new JsonNumber(3), new JsonNumber(4), new JsonNumber(2)
                )
        )
);

JsonObject jsonObjectProjection = jsonObject.projection("surname", "age", "year",
"marks");
print(jsonObjectProjection); // {'surname': 'Rodionov', 'year': 2, 'marks': [3, 4]}
```

The following tests should pass - `json.JsonObjectAddFindProjectionTest`

Part 5*
Student class extends *BasicStudent* and contains additional property for storing exams results.
You should implement *Student* class and override *toJsonObject* method.
Student class constructor should have the following signature:

```
public Student(String name, String surname, Integer year,
                                                Tuple<String, Integer>... exams) {
   // ToDo
}
```

Array of tuples contains exams results: first parameter is a subject name, second - exam mark. For example new *Student* object can be instantiated in a following way:

```
new Student(
        "Andrii",
        "Rodionov",
        3,
        new Tuple<>("OOP", 3),
        new Tuple<>("English", 5),
        new Tuple<>("Math", 2)
);
```

For this Student instance *toJsonObject* method should return the following *JsonObject*

```
{
  'name': 'Andrii',
  'surname': 'Rodionov',
  'year': 3,
  'exams': [
          {
            'course': 'OOP',
            'mark': 3,
            'passed': true
          },
          {
            'course': 'English',
            'mark': 5,
            'passed': true
          },
          {
            'course': 'Math',
            'mark': 2,
            'passed': false
          }
  ]
}
```

Please pay attention that *'passed'* status should be calculated manually. If mark is less than 3, *'passed'* status should be *false*, otherwise - *true.*

Your task is to implement *Student* class constructor and *toJsonObject* method. After this the following tests should pass - `domain.StudentTest`

Project submission

1. Create new repository in your GitHub account with the following name: ***apps19<surname>-exam***

2. Push your **src** code to this repository
3. Create new Build Job at CI Hudson with name ***apps19<surname>-exam**,* using **JobsTemplate**
   Роботи здані в іншому форматі не перевіряються.

4. Before start building your project at CI Hudson, try to run tests in your local IDE
5. Build your project at CI Hudson