Introduction to JSON

John Girata

CS 2340 Spring 2011

1 Overview

JSON, short for JavaScript Object Notation, is a data transfer and serialization standard inspired by JavaScript, a browser scripting language for building interactive websites. JavaScript has liminted support for object-oriented programming and uses a very simple syntax for defining classes. For example, in Java we might write a class to represent a car like this:

```
public class Car {
    private int numWheels = 4;
    private int numDoors = 2;
    private String color;

    public Car(String color) {
        this.color = color;
    }

    public void drive() {
        // ...
    }
}
```

In JavaScript, the same class would look something like this:

```
var Car = Class.create({
    numWheels: 4,
    numDoors: 2,

    initialize: function(var color) {
        this.color = color;
    },

    drive: function() {
        // ...
    }
});
```

As you can see, JavaScript's syntax for defining classes is very minimalistic. This makes it an excellent candidate for a lightweight object serialization standard.

2 Syntax

JSON defines four basic data types (strings, integers, doubles, and booleans) and two types which are collections of the basic types (objects and arrays).

2.1 Objects

Objects are the basic building blocks in JSON, much like classes in Java. All data represented using JSON is enclosed within an object. Objects start with an opening brace, contain comma-separated key-value pairs,

and end with a closing brace. The key-value pairs consist of a string mapping to a value that can be any of the six data types (including another JSON object). This object is one way to represent the `Car` class from above:

```
{
    numWheels: 4,
    numDoors: 2,
    color: "Blue"
}
```

Suppose we also want to transfer some information about the driver. In Java we would probably have a class to represent a person, so it would make sense to have a corresponding object in JSON:

```
{
    numWheels: 4,
    numDoors: 2,
    color: "Blue",
    driver: {
        name: "Bill",
        age: 24
    }
}
```

2.2 Arrays

The syntax for arrays is also borrowed from JavaScript. They begin with an opening bracket, contain comma-separated values, and end with a closing bracket. Continuing the previous example, suppose we want to transfer a list of passengers with the information about our car (for simplicity, we'll only store the passengers' names):

```
{
    numWheels: 4,
    numDoors: 2,
    color: "Blue",
    driver: {
        name: "Bill",
        age: 24
    },
    passengers: ["Alice", "Bob", "Charlie"]
}
```

3 JSON in Java

Support for JSON is not provided with the JDK, but an excellent open source library exists. A link to download the library we used is in the links section at the end of this document.

3.1 Basics

Most users will only need to use the `JSONObject` and `JSONArray` classes. `JSONObject`'s default constructor creates an empty object and the `put()` method can be used to set a single key-value pair, much like the behavior of `java.util.Map`. For example, we can construct an object to represent our car using this:

```
JSONObject object = new JSONObject();

object.put("numWheels", 4);
object.put("numDoors", 2);
object.put("color", "Blue");
```

Similarly, we can add the array of passengers by first creating an instance of `JSONArray`, then adding that to the `JSONObject`:

```
JSONArray passengers = new JSONArray();

passengers.put("Alice");
passengers.put("Bob");
passengers.put("Charlie");

object.put("passengers", passengers);
```

When you are ready to transfer your object over a network or save it to a text file, use `JSONObject.toString()` to convert the object to a string.

3.2 Retrieving JSON Data

`JSONObject` can also deserialize JSON data. Pass the string representing the data into `JSONObject`'s constructor to build a map of the data:

```
String json = "{ numWheels: 4, numDoors: 2, color: \"Blue\" }";
JSONObject object = new JSONObject(json);
```

`JSONObject` and `JSONArray` provide a number of methods for accessing the fields in the data:

```
getBoolean(...):  boolean
getDouble(...):  double
getInt(...):  int
getLong(...):  long
getString(...):  String
getJSONObject(...):  JSONObject
getJSONArray(...):  JSONArray
```

Each method takes in a key (for `JSONObject`) or index (for `JSONArray`) and attempts to convert the data to the method's associated type. Each method throws a checked `JSONException` if the element either does not exist or cannot be converted to the corresponding type. Normally the type of the data associated with each value would be known beforehand. But if it is not, `getString(...)` can always be used safely as that method will never throw a `JSONException` for having the wrong type (all data is stored internally as a `String`)

4 Using JSON

This section describes a system that uses JSON for serializing objects. Its design is very similar to my team's design, but this system differs slightly based on things we learned during the semester. Also, many details unrelated to serialization (such as exception handling) have been omitted. For our actual code, see the end of this document for a link to our Google Code project page.

4.1 `JSONSerializable` Interface

To use JSON for serialization in our project, we first created an interface `JSONSerializable`. The general contract of this interface has two requirements:

1. The class must have a method `toJSONObject()` that returns the object encoded as a `JSONObject`.
2. The class must have a constructor that takes in a `JSONObject` from which to deserialize the object.

For example, consider the `User` class:

```
public abstract class User implements JSONSerializable {
    private String username;
    private int age;

    public User(JSONObject json) {
        if (json.has("username")) {
            this.username = json.getString("username");
        }

        if (json.has("age")) {
            this.age = json.getInt("age");
        }
    }

    public JSONObject toJSONObject() {
        JSONObject json = new JSONObject();

        json.put("username", this.username);
        json.put("age", this.age);

        return json;
    }
}
```

4.2 Using `JSONSerializable`

We took advantage of `JSONSerializable` in several components in our system, one of which is the database. The methods in our `Database` interface only work on objects implementing `JSONSerializable`:

```
public interface Database {
    /**
     * Saves one object to the database.
     */
    void save(JSONSerializable json);

    /**
     * Loads all objects in the database.
     */
    void load();
}
```

This allows us to abstract all class-specific code out of the database and into the class itself. As a result, a database implementation does not need to worry about the format of the name in `Patient` or the variable type of the date in `Appointment`.

A simple implementation of this interface is our `FlatFileDatabase` class. It stores objects in a directory structure and each object's JSON string is stored in a plain text file. The save() method is trivial: it takes in an object that implements `JSONSerializable`, retrieves the JSON string, and saves it to a text file.

The `load()` method is a bit more complicated. Because an interface can only define required methods (not required constructors), there is no way to enforce the general contract of `JSONSerialization` at compile-time. To get around this issue, we used the Java Reflection API to dynamically call the required constructor (the fully-qualified class name of each object is stored when the object is saved). If the required constructor does not exist, a very "loud" error is produced, and the hope is that this error would be encountered in a testing environment (and not in a production environment).

```
public class FlatFileDatabase implements Database {
    public void save(JSONSerializable object) {
        String json = object.toJSONObject().toString();
        FileWriter writer = new FileWriter(...);

        writer.write(json);
        writer.close();
    }

    public void load() {
        try {
            String json = // Read JSON from file...

            Class<?> clazz = Class.forName(/* new object's class */);
            Class<?> jsonClass = Class.forName("org.json.JSONObject");
            Constructor<?> con = clazz.getConstructor(jsonClass);
            con.newInstance(new JSONObject(json));
        } catch (NoSuchMethodException e) {
            // Make a lot of noise...
        }
    }
}
```

5 Links and Further Reading

http://json.org/	JSON project homepage.
http://json.org/java/	JSON library for Java.
http://json.org/javadoc/	JSON Java API documentation.
http://cyberdoc.googlecode.com/svn/trunk/lib/json.jar	JSON library packaged as a JAR.
http://code.google.com/p/cyberdoc	Google Code project page.
http://download.oracle.com/javase/tutorial/reflect/	Java Reflection API tutorial.