**minimal-json**

A fast and minimal JSON parser and writer for Java. It's not an object mapper, but a bare-bones library that aims at being

* **minimal**: no dependencies, single package with just a few classes, small download size (< 35kB)
* **fast**: high performance comparable with other state-of-the-art parsers (see below)
* **lightweight**: object representation with minimal memory footprint (e.g. no HashMaps)
* **simple**: reading, writing and modifying JSON with minimal code (short names, fluent style)

Minimal-json is fully covered by unit tests, and field-tested by the [Eclipse RAP project](http://eclipse.org/rap) and others (see below). The JAR contains a **valid OSGi** bundle manifest and can be used in OSGi environments without modifications.

**Usage**

The class Json is the entrypoint to the minimal-json API, use it to parse and to create JSON.

**Parse JSON**

You can parse JSON from a String or from a java.io.Reader. You *don't* need to wrap your reader in a BufferedReader, as the parse method uses a reading buffer.

JsonValue value = Json.parse(string);

**JSON values**

JSON values are represented by the type JsonValue. A JsonValue can contain a JSON array, object, string, number, or one of the literals true, false, and null. To transform a JsonValue into a Java type, use the methods asString, asInt,asFloat, asArray etc., depending on the expected type. To query the actual type, use one of the methods isString,isNumber, isArray, isObject, isBoolean, and isNull, for example:

if (value.isString()) {

String string = value.asString();

// ...

} else if (value.isArray()) {

JsonArray array = value.asArray();

// ...

}

**JSON arrays**

The method asArray returns an instance of JsonArray. This subtype of JsonValue provides a get method to access the elements of a JSON array:

JsonArray array = Json.parse(reader).asArray();

String name = array.get(0).asString();

int quantity = array.get(1).asInt();

You can also iterate over the elements of a JsonArray, which are again also JSON values:

for (JsonValue value : jsonArray) {

// ...

}

**JSON objects**

Similar to JsonArray, the type JsonObject represents JSON objects, the map type in JSON. Members of a JSON object can be accessed by name using the get method.

JsonObject object = Json.parse(input).asObject();

String name = object.get("name").asString();

int quantity = object.get("quantity").asInt();

There are also shorthand methods like getString, getInt, getDouble, etc. that directly return the expected type. These methods require a default value that is returned when the member is not found:

String name = object.getString("name", "Unknown");

int age = object.getInt("quantity", 1);

You can also iterate over the members of a JSON object:

for (Member member : jsonObject) {

String name = member.getName();

JsonValue value = member.getValue();

// ...

}

**Example: Extract nested contents**

Let's take the following JSON as an example:

{

"order": 4711,

"items": [

{

"name": "NE555 Timer IC",

"cat-id": "645723",

"quantity": 10,

},

{

"name": "LM358N OpAmp IC",

"cat-id": "764525",

"quantity": 2

}

]

}

The following snippet extracts the names and quantities of all items:

JsonArray items = Json.parse(json).asObject().get("items").asArray();

for (JsonValue item : items) {

String name = item.asObject().getString("name", "Unknown Item");

int quantity = item.asObject().getInt("quantity", 1);

...

}

**Create JSON values**

The entrypoint class Json also has methods to create instances of JsonValue from Java strings, numbers, and boolean values, for example:

JsonValue name = Json.value("Alice");

JsonValue points = Json.value(23);

And there are methods for creating empty arrays and objects as well. Use these together with add to create data structures:

JsonObject user = Json.object().add("name", "Alice").add("points", 23);

// -> {"name": "Alice", "points": 23}

JsonArray user = Json.array().add("Bob").add(42);

// -> ["Bob", 42]

You can also create JSON arrays conveniently from Java arrays such as String[], int[], boolean[], etc.:

String[] javaNames = {"Alice", "Bob"};

JsonArray jsonNames = Json.array(names);

**Modify JSON arrays and objects**

You can replace or remove array elements based on their index. The index must be valid.

jsonArray.set(1, 24);

jsonArray.remove(1);

Likewise, members of JSON objects can be modified by their name. If the name does not exist, set will append a new member.

jsonObject.set("quantity", 24);

jsonObject.remove("quantity");

JsonObject also provides a merge method that copies all members from a given JSON object.

jsonObject.merge(otherObject);

**Output JSON**

The toString method of a JsonValue returns valid JSON strings. You can also write to a java.io.Writer using writeTo:

String json = jsonValue.toString();

jsonValue.writeTo(writer);

Both methods accept an additonal parameter to enable formatted output:

String json = jsonValue.toString(WriterConfig.PRETTY\_PRINT);

jsonValue.writeTo(writer, WriterConfig.PRETTY\_PRINT);

For more details, have a look at the [JavaDoc](http://www.javadoc.io/doc/com.eclipsesource.minimal-json/minimal-json/).

**Concurrency**

The JSON structures in this library (JsonObject and JsonArray) are deliberately **not thread-safe** to keep them fast and simple. In the rare case that JSON data structures must be accessed from multiple threads, while at least one of these threads modifies their contents, the application must ensure proper synchronization.

Iterators will throw a ConcurrentModificationException when the contents of a JSON structure have been modified after the creation of the iterator.

**Performance**

I've spent days benchmarking and tuning minimal-json's reading and writing performance. You can see from the charts below that it performs quite reasonably. But damn, it's not quite as fast as Jackson! However, given that Jackson is a very complex machine compared to minimal-json, I think for the minimalism of this parser, the results are quite good.

Below is the result of a performance comparison with other parsers, namely [org.json](http://www.json.org/java/index.html) 20141113, [Gson](http://code.google.com/p/google-gson/) 2.3.1, [Jackson](http://wiki.fasterxml.com/JacksonHome) 2.5.0, and [JSON.simple](https://code.google.com/p/json-simple/) 1.1.1. In this benchmark, two example JSON texts are parsed into a Java object and then serialized to JSON again. All benchmarks can be found in [com.eclipsesource.json.performancetest](https://github.com/ralfstx/minimal-json/tree/master/com.eclipsesource.json.performancetest).

[rap.json](https://github.com/ralfstx/minimal-json/blob/master/com.eclipsesource.json.performancetest/src/main/resources/input/rap.json), ~ 30kB

[caliper.json](https://github.com/ralfstx/minimal-json/blob/master/com.eclipsesource.json.performancetest/src/main/resources/input/caliper.json), ~ 83kB

**Who is using minimal-json?**

Here are some projects that use minimal-json:

* [Hazelcast](http://hazelcast.org/)
* [Eclipse RAP](http://eclipse.org/rap)
* [Box.com Java SDK](http://opensource.box.com/box-java-sdk/)
* [jshint-eclipse](https://github.com/eclipsesource/jshint-eclipse)
* [tern.java](https://github.com/angelozerr/tern.java)
* [Human JSON](https://github.com/laktak/hjson-java)

**Include**

You can include minimal-json from Maven Central by adding this dependency to your pom.xml:

<dependency>

<groupId>com.eclipsesource.minimal-json</groupId>

<artifactId>minimal-json</artifactId>

<version>0.9.4</version>

</dependency>

Development snapshots are available on [oss.sonatype.org](https://oss.sonatype.org/content/repositories/snapshots/com/eclipsesource/minimal-json/minimal-json/).

**Build**

To build minimal-json on your machine, checkout the repository, cd into it, and call:

mvn clean install

A continuous integration build is running at [Travis-CI](https://travis-ci.org/ralfstx/minimal-json).

**License**

The code is available under the terms of the [MIT License](http://opensource.org/licenses/MIT).