Towards a JSON API for the JDK

Paul Sandoz paul.sandoz at oracle.com
Thu May 15 20:30:42 UTC 2025

- Previous message (by thread): Integrated: 8357075: Remove leftover COMPAT locale data tests
- Next message (by thread): Towards a JSON API for the JDK
- **Messages sorted by:** [date] [thread] [subject] [author]

Hi,

We would like to share with you our thoughts and plans towards a JSON API for the JDK.
Please see the document below.

-

We have had the pleasure of using a clone of this API in some experiments we are conducting with
ONNX and code reflection [1]. Using the API we were able to quickly write code to ingest and convert
a JSON document representing ONNX operation schema into instances of records modeling the schema
(see here [2]).

The overall out-of-box experience with such a minimal "batteries included" API has so far been positive.

Thanks,
Paul.

[1] https://openjdk.org/projects/babylon/
[2] https://github.com/openjdk/babylon/blob/code-reflection/cr-examples/onnx/opgen/src/main/java/oracle/code/onnx/opgen/OpSchemaParser.java#L87

Towards a JSON API for the JDK

One of the most common requests for the JDK is an API for parsing and generating
JSON. While JSON originated as a text-based serialization format for JSON
objects ("JSON" stands for "JavaScript Object Notation"), because of its simple
and flexible syntax, it eventually found use outside the JavaScript ecosystem as
a general data interchange format, such as framework configuration files and web
service requests/response formats.

While the JDK cannot, and should not, provide libraries for every conceivable
file format or protocol, the JDK philosophy is one of "batteries included",
which is to say we should be able to write basic programs that use common
protocols such as HTTP, without having to appeal to third party libraries.
The Java ecosystem already has plenty of JSON libraries, so inclusion in
the JDK is largely meant to be a convenience, rather than needing to be the "one
true" JSON library to meet the needs of all users. Users with specific needs
are always free to select one of the existing third-party libraries.

Goals and requirements

Our primary goal is that the library be simple to use for parsing, traversing,
and generating conformant JSON documents. Advanced features, such as data
binding or path-based traversal should be possible to implement as layered
features, but for simplicity are not included in the core API. We adopt a goal
that the performance should be "good enough", but where performance
considerations conflict with simplicity and usability, we will choose in favor
of the latter.

API design approach

The description of JSON at `https://json.org` describes a JSON document using
the familiar "railroad diagram":
![image](https://www.json.org/img/value.png)

This diagram describes an algebraic data type (a sum of products), which we
model directly with a set of Java interfaces:

```
interface JsonValue { }
interface JsonArray extends JsonValue { List<JsonValue> values(); }
interface JsonObject extends JsonValue { Map<String, JsonValue> members(); }
interface JsonNumber extends JsonValue { Number toNumber(); }
interface JsonString extends JsonValue { String value(); }
interface JsonBoolean extends JsonValue  { boolean value(); }
interface JsonNull extends JsonValue { }
```

These interfaces have (hidden) companion implementation classes that admit
greater flexibility of implementation than modeling them directly with records would permit.
Further, these interfaces are unsealed. We compromise on the sealed sum of products to enable
alternative implementations, for example to support alternative formats that
encode the same information in a JSON document but in a more efficient form than
text.

The API has static methods for parsing strings into a `JsonValue`, conversion to
and from purely untyped representations (lists and maps), and factory methods
for building JSON documents. We apply composition consistently, e.g, a
JsonString has a string, a JsonObject has a map of string to JsonValue, as
opposed to extension for structural JSON values.

It turns out that this simple API is almost all we need for traversal. It gives
us an immutable representation of a document, and we can use pattern matching to
answer the myriad questions that will come up (Does this object have key X? Does
it map to a number? Is that number representable as an integer?) when going
from an untyped format like JSON to a more strongly typed domain model.
Given a simple document like:

```
    {
        "name": "John",
        "age": 30
    }
```

we can parse and traverse the document as follows:

```
JsonValue doc = Json.parse(inputString);
if (doc instanceof JsonObject o
    && o.members().get("name") instanceof JsonString s
    && s.value() instanceof String name
    && o.members().get("age") instanceof JsonNumber n
    && n.toNumber() instanceof Long l && l instanceof int age) {
        // use "name" and "age"
    }
```

Later, when the language acquires the ability to expose deconstruction patterns
for arbitrary interfaces (similar to today's record patterns, see
https://openjdk.org/projects/amber/design-notes/patterns/towards-member-patterns),
this will be simplifiable to:

```
JsonValue doc = Json.parse(inputString);
if (doc instanceof JsonObject(var members)
    && members.get("name") instanceof JsonString(String name)
    && members.get("age") instanceof JsonNumber(int age)) {
        // use "name" and "age"
    }
```

So, overtime, as more pattern matching features are introduced we anticipate
improved use of the API. This is a primary reason why the API is so minimal.
Convenience methods we add today, such as a method that accesses a JSON
object component as say a JSON string or throws an exception, will become
redundant in the future.

JSON numbers

The specification of JSON number makes no explicit distinction between integral
and decimal numbers, nor specifies limits on the size of those numbers.
This is a common source of interoperability issues when consuming JSON
documents. Generally users cannot always but often do assume JSON numbers are
parsable, without loss of precision, to IEEE double-precision floating point
numbers or 32-bit signed integers.

In this respect the API provides three means to operate on the JSON number,
giving the user full control:

1. Underlying string representation can be obtained, if preserving syntactic
   details such as leading or trailing zeros is important.
2. The string representation can be parsed to an instance of `BigDecimal`, using
   `toBigDecimal` if preserving decimal numbers is important.
3. The string representation can be parsed into an instance of `Long`, `Double`,
   `BigInteger`, or `BigDecimal`, using `toNumber`. The result of this method
   depends on how the representation can be parsed, possibly losing precision,
   choosing a suitably convenient numeric type that can then be pattern
   matched on.

Primitive pattern matching will help as will further pattern matching features
enabling the user to partially match.

Prototype implementation

The prototype implementation is currently located into the JDK sandbox
repository
under the `json` branch, see
here https://github.com/openjdk/jdk-sandbox/tree/json/src/java.base/share/classes/java/util/json
The prototype API javadoc generated from the repository is also available at
https://cr.openjdk.org/~naoto/json/javadoc/api/java.base/java/util/json/package-summary.html

Testing and conformance

The prototype implementation passes all conformance test cases but two, available
on https://github.com/nst/JSONTestSuite. The two exceptions are the ones which the
prototype specifically prohibits, i.e, duplicated names in JSON objects
(https://cr.openjdk.org/~naoto/json/conformance/results/parsing.html#35).

Performance

Our main focus so far has been on the API design and a functional implementation.
Hence, there has been less focus on performance even though we know there are a
number of performance enhancements we can make eventually.
We are reasonably happy with the current performance. The
implementation performs well when compared to other JSON implementations
parsing from string instances and traversing documents.

An example of where we may choose simplicity and usability over performance
is the rejection of JSON documents containing objects that in turn contain members
with duplicate names. That may increase the cost of parsing, but simplifies the user
experience for the majority of cases since if we reasonably assume JsonObjects are
map-like, what should the user do with such members, pick one the last one? merge
the values? or reject?

A JSON JEP?

We plan to draft JEP when we are ready. Attentive readers will observe that
a JEP already exists, JEP 198: Light-Weight JSON API (https://openjdk.org/jeps/198). We will
either update this JEP, or withdraw it and draft a new one.

- Previous message (by thread): Integrated: 8357075: Remove leftover COMPAT locale data tests
- Next message (by thread): Towards a JSON API for the JDK
- **Messages sorted by:** [date] [thread] [subject] [author]

More information about the core-libs-dev mailing list