Conception de logiciels intranet : intergiciels à objets répartis (NSY102)

Sérialisation JSON : interface JsonValue

Dans le TP précédent, nous avons vu comment implanter directement la sérialisation JSON, depuis et vers, un type de données algébrique particulier. L'inconvénient majeur de cette approche est d'être obligé d'implanter le *parser* et le *printer* pour chaque type de données.

Deux manières standards de résoudre ce type de problème sont :

1. Introduire un type de données intermédiaire permettant de factoriser le code du *parser* et du *printer*.
2. Mécaniser la génération du code du *parser* et du *printer* à partir de la définition du type de données.

Ces deux solutions sont spécifiées dans les normes de Jakarta EE (qui remplace Java EE depuis la version 8) :

1. Specification JSR 374 – JSON-P (JSON Processing)
2. Specification JSR 367 – JSON-B (JSON Binding)

Dans ce TP, nous allons nous inspirer de cette spécification pour réaliser une implantation de la première solution. Le type de données choisi dans la spécification pour représenter les données JSON prend la forme d'une interface `JsonValue`, et de plusieurs classes qui l'implantent, qui sont documentées ici :

– Interface `JsonValue`

Les interfaces permettant de sérialiser et désérialiser les données JSON sont `JsonReader` et `JsonWriter`, et elles sont documentées ici :

– Interface `JsonReader`

– Interface `JsonWriter`

Nous donnons dans ce TP une représentation alternative de `JsonValue`, basée sur les *records*, et nous verrons comment implanter la sérialisation de cette représentation (le code du *parser* est fourni). Puis nous verrons comment implanter la conversion des types de données algébriques depuis et vers cette représentation. Enfin, comme exemple d'utilisation, nous appliquerons cette sérialisation pour implanter à nouveau l'accès en lecture à une liste distante de type générique.

Questions.

1. Nous introduisons le type de données algébrique suivant pour représenter des données JSON[1]. Ce type est inspiré des classes de même nom présentes dans la spécification Jakarta EE :

```
sealed interface JsonValue {
    record JsonObject(Map<String, JsonValue> m) implements JsonValue {}
    record JsonArray(List<JsonValue> l) implements JsonValue {}
    record JsonNumber(Double d) implements JsonValue {}
    record JsonString(String s) implements JsonValue {}
    record JsonBoolean(Boolean b) implements JsonValue {}
}
```

1. Définition basée sur `https://www.infoq.com/articles/data-oriented-programming-java`

Implanter et tester la méthode `toString` de la classe `Printer` (*subpackage* `json`), les classes `Lexer` et `Parser` sont intégralement fournies.

2. Les classes `JsonReader` et `JsonWriter` sont fournies (*subpackage* `json`). Tester ces classes en utilisant les classes `StringReader` et `StringWriter` de la JDK.

3. L'interface `JsonSerializer<A>` (*subpackage* `data`) contient deux méthodes permettant de convertir un type de données `A` depuis et vers `JsonValue`. Cette interface est définie ainsi :

```
interface JsonSerializer<A> {
    JsonValue toJson(A data);
    A fromJson(JsonValue value);
}
```

   Compléter et tester :

   – le record `StringSerializer` qui implante `JsonSerializer<String>`.

   – le record `DoubleSerializer` qui implante `JsonSerializer<Double>`.

   – le record `IntegerSerializer` qui implante `JsonSerializer<Integer>`.

   Les classes `DataReader` et `DataWriter` sont fournies (*subpackage* `json`). Tester ces classes en utilisant les classes `StringReader` et `StringWriter` de la JDK.

4. Si on sait sérialiser des données de type `A`, on sait aussi sérialiser des données de type `List<A>`. Compléter et tester (*subpackage* `data`) :

   – le record `ListSerializer<A>` qui implante `JsonSerializer<List<A>>`.

5. Compléter et tester :

   – le record `ExprSerializer` qui implante `JsonSerializer<Expr>`.

6. On rappelle le protocole utilisé aux TP précédents, pour implanter l'accès en lecture à une liste distante de type générique :

```
sealed interface Request {
    record GetSize() implements Request {}
    record GetValue(Integer index) implements Request {}
    record Stop() implements Request {}
}

sealed interface Answer<A> {
    record Size<A>(Integer index) implements Answer<A> {}
    record Value<A>(A element) implements Answer<A> {}
}
```

   Compléter et tester :

   – le record `RequestSerializer` qui implante `JsonSerializer<Request>`.

   – le record `AnswerSerializer<A>` qui implante `JsonSerializer<Answer<A>>`.

   Le code du serveur (classe `Server<A>`) et le code qui implante la partie client de la liste distante (classe `RemoteList<A>`) sont fournis (le protocole est toujours celui vu dans les TP précédents). Tester ce code avec des éléments de type `String` (les communications utilisent donc les classes `RequestSerializer` et `AnswerSerializer<String>`).

A Input/Output streams et Reader/Writer

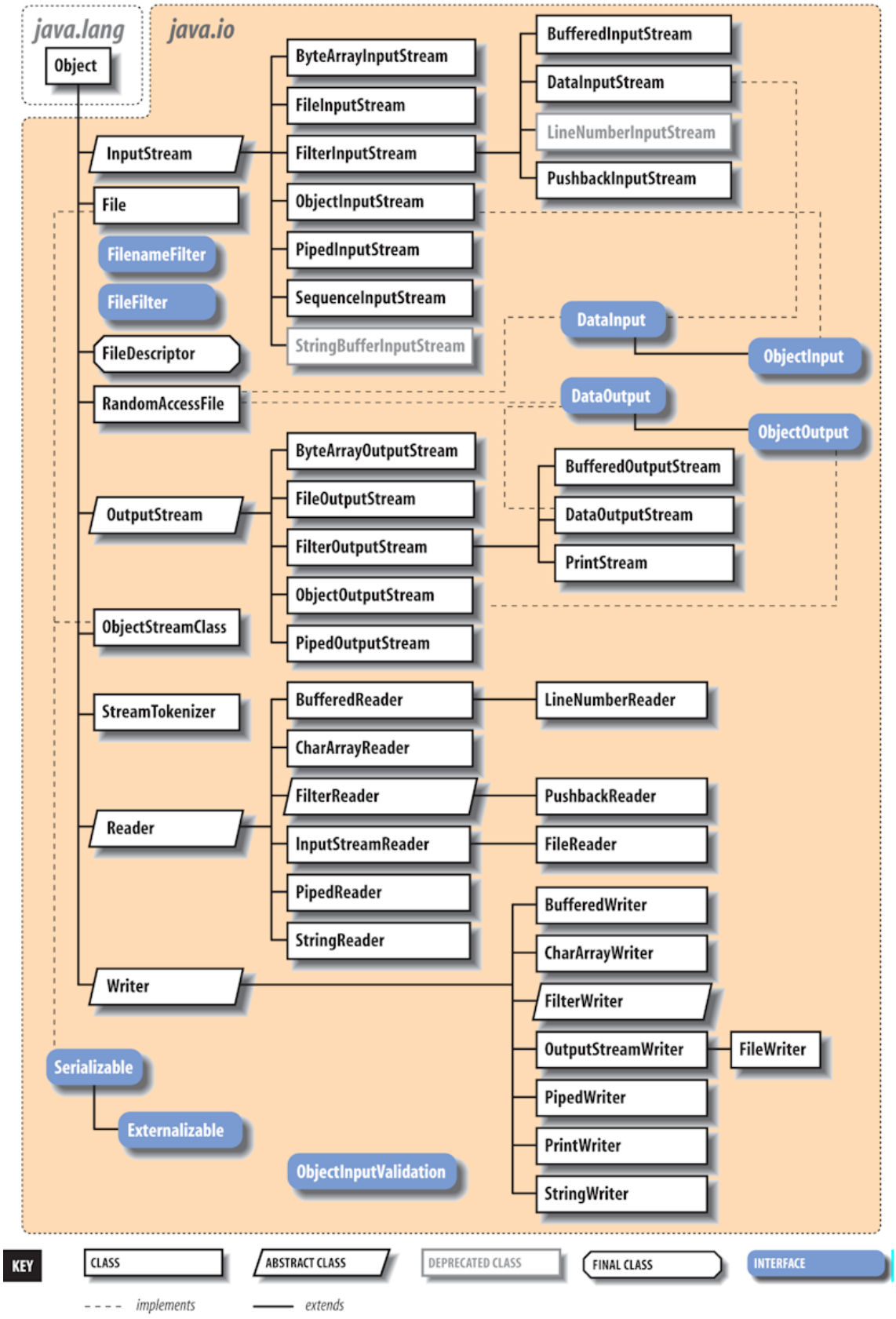


Figure tirée de « Learning Java », (4th Edition, 2013) de P. Niemeyer, D. Leuck.

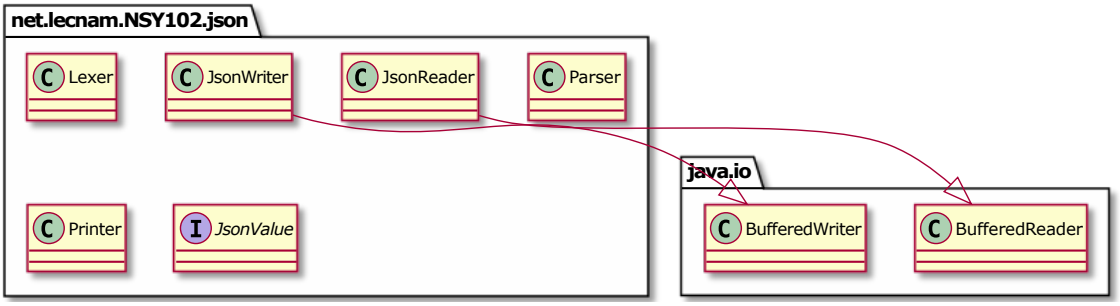
B Type Token utilisé par le « Lexer » et le « Parser »

L'architecture du Parser suit le schéma classique :

Le lexer consomme les caractères provenant d'une instance `Reader` et renvoie un résultat de type `List<Token>`. Le type `Token` est lui-même défini ainsi :

```
sealed interface Token {}
record T_Int(Integer i) implements Token {}
record T_Double(Double d) implements Token {}
record T_Ident(String s) implements Token {}
record T_String(String s) implements Token {}
record T_LCURL() implements Token {}
record T_RCURL() implements Token {}
record T_LSQUARE() implements Token {}
record T_RSQUARE() implements Token {}
record T_COMMA() implements Token {}
record T_COLON() implements Token {}
record T_UNKNOWN() implements Token {}
```

C Diagramme de classe du package json



D Diagramme de classe du package data