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

Sérialisation JSON

La sérialisation (*serialization* ou *marshalling*) est un mécanisme permettant d'encoder des données sous la forme d'une sequence d'octets (donc binaire ou textuel) puis de décoder à nouveau cette séquence pour reconstruire les données. La sérialisation peut donc être utilisée pour assurer la persistance de données (sérialisation dans un fichier) ou pour transférer des données à travers le réseau.

La sérialisation peut utiliser un encodage propriétaire, spécifique à une plateforme (par exemple la sérialisation des objets par la JVM) ou utiliser un format publique multi-plateformes (par exemple, textuel, en JSON ou XML).

Une spécification pour la sérialisation JSON de haut niveau (JSON Binding) est disponible ici :

Specification JSR 367 – JSON-B

Une implantation de référence, Eclipse Yasson, est fournie ici :

https://github.com/eclipse-ee4j/yasson

Plusieurs autres implantations de cette spécification existent (avec des niveaux de compatibilité variables) comme par exemple Jackson :

https://github.com/FasterXML/jackson

Dans ce TP, nous verrons comment implanter nous-même une sérialisation en JSON des types de données algébriques. La difficulté technique provient essentiellement de l'analyse lexicale et syntaxique nécessaire au décodage, un code source partiel est donc fourni.

Remarque. Le code source fourni a été en partie généré mécaniquement, il n'est donc pas idiomatique. En effet, en Java on utiliserait plutôt la classe StreamTokenizer pour implanter le Lexer :

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/StreamTokenizer.html

Exercices.

1. Le code fourni contient l'interface `Expr` qui correspond au type algébrique vu lors d'un TP précédent (étendu par un constructeur `Avg` qui prend une liste d'expressions en paramètre). Implanter une méthode `toJSON` qui affiche une expression au format JSON.

Par exemple, l'expression :

```
Plus[e1=Const[i=3], e2=Times[e1=Const[i=6], e2=Const[i=5]]]
```

devra être convertie sous la forme :

```
{
        "@type": "Plus",
        "e1": {
                "@type": "Const",
                "i": 3
        },
        "e2": {
                "@type": "Times",
                "e1": {
                        "@type": "Const",
                        "i": 6
                },
                "e2": {
                        "@type": "Const",
                        "i": 5
                }
        }
}
```

2. Etudier la classe `Parser` du code fourni qui contient un analyseur syntaxique pour le type `Expr` vu lors d'un TP précédent. Tester ce code sur l'exemple précédent, ainsi que sur d'autres exemples (contenant `Avg` en particulier).

3. On rappelle le protocole utilisé au TP précédent, dans le cas où les éléments de la liste sont de type `String` :

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

sealed interface Answer {}
record Size(Integer index) implements Answer {}
record Value(String element) implements Answer {}
```

   Implanter la sérialisation JSON pour ces interfaces.

4. Modifier le code du client et du serveur de manière à utiliser la sérialisation JSON définie ci-dessus. Les canaux de communication devront être implantés en utilisant les classes `PipedOutputStream` et `PipedInputStream`. On pourra par exemple implanter des décorateurs pour les classes `BufferedReader` et `BufferedWriter` premettant de lire et d'écrire des données de type `Request` et `Answer`.

A Input/Output streams et Reader/Writer

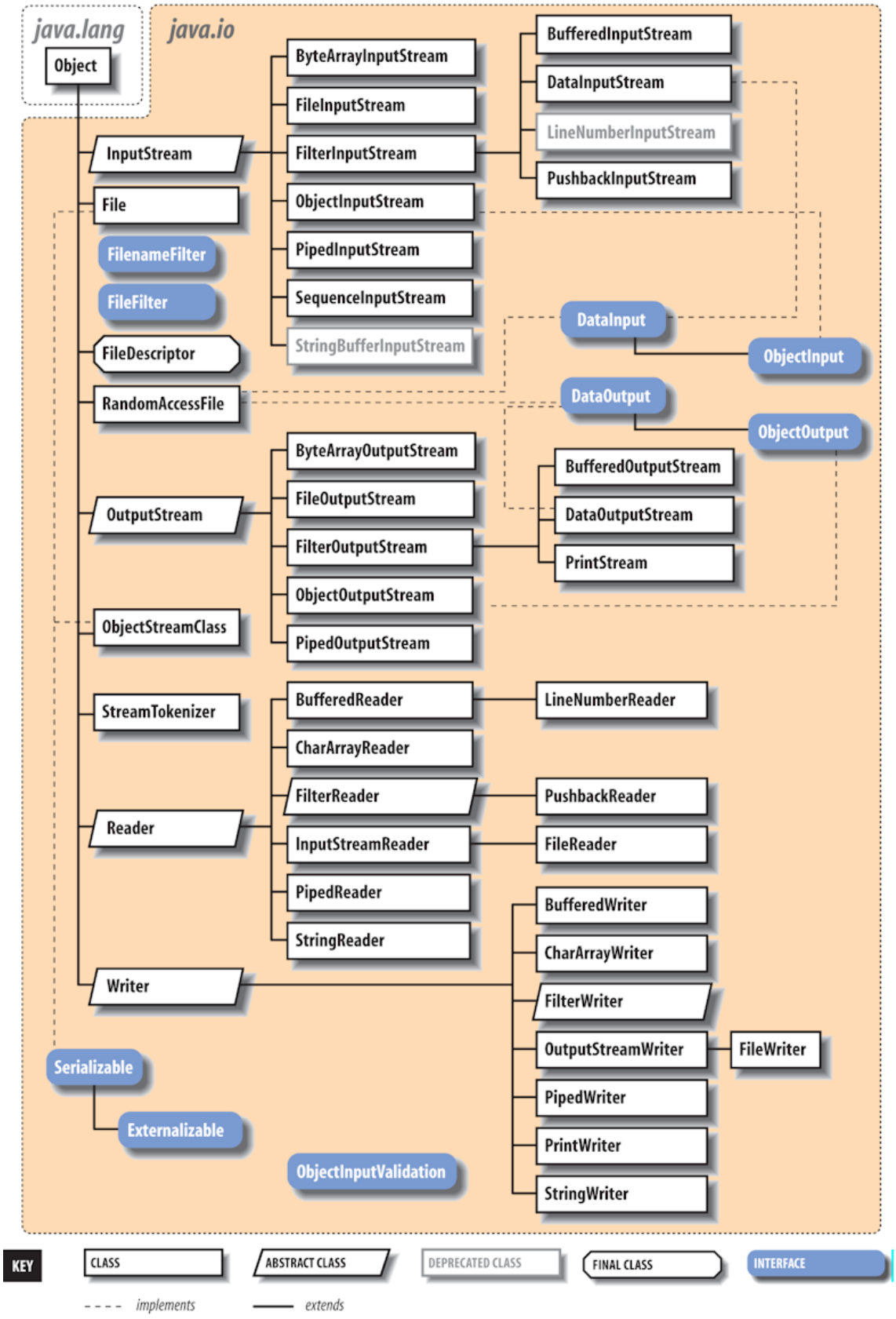

Figure tirée de « Learning Java », (4th Edition, 2013) de P. Niemeyer, D. Leuck.

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_Ident(String s) implements Token {}
record T_String(String s) implements Token {}
record T_LANGLE() implements Token {}
record T_RANGLE() 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_EQUAL() implements Token {}
record T_UNKNOWN() implements Token {}
```