Kayyssa Haïssou,
Ewen Horville, Baptiste NEVEJANS

RAPPORT

PROJET JSON

Nevejans Baptise ; Haïssous Kayyissa ; Horville Ewen

SOMMAIRE

Kayyssa Haïssou,
Ewen Horville, Baptiste NEVEJANS

1. Introduction

Ce projet a pour objectif de réaliser un inspecteur de JSON, afin de faciliter l'affichage de ces données. Pour procéder correctement à sa réalisation, il est divisé en plusieurs étapes avec dans un premier temps la simplification de l'affichage puis la coloration de certains éléments pour plus de clartés.

2. Fonctionnalités du programme

Phase 1 :

La première phase consiste à récupérer le chemin d'accès d'un fichier JSON et a passé son contenu en une chaîne de caractères. On construit l'arbre à partir du premier caractère puis on l'affiche avec des règles de traitement données par les normes du JSON. L'affichage se fait comme ceci :

Nous avons travaillé le plus possible la phase 1 afin de factoriser le code utilisé pour les phases suivantes ainsi que pour adapter les structures abstraites à d'autres langages. La classe PrettyPrinter en est un exemple notable. La classe a été conçue en dehors de toute syntaxe et est donc compatible avec le JSON comme le PHP. Il nous suffit donc d'étendre la classe et de changer quelques variables pour arriver à une syntaxe complètement différente :

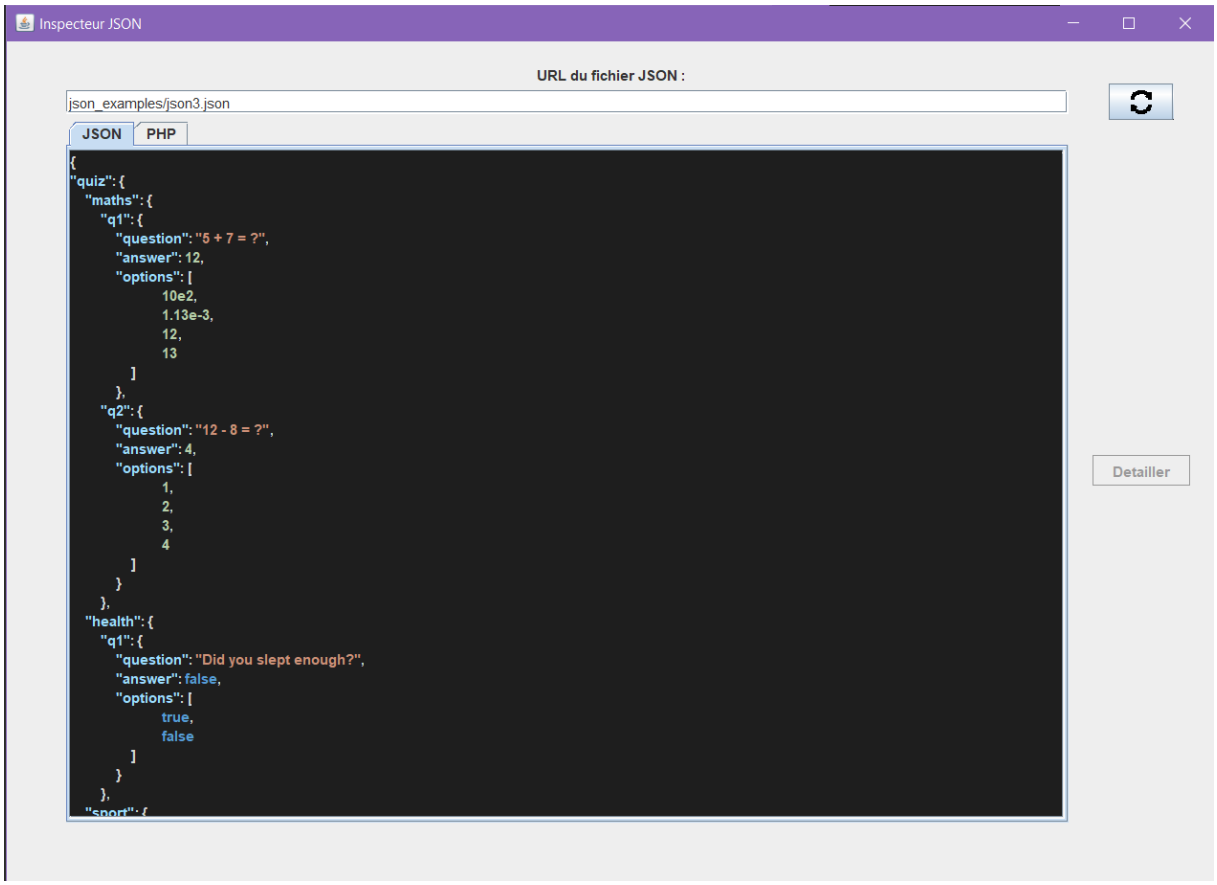
Kayyssa Haïssou,
Ewen Horville, Baptiste NEVEJANS

```
Array
(
[glossary] => Array
  (
  [title] => "example glossary"
  [GlossDiv] => Array
    (
    [GlossList] => Array
      (
      [GlossEntry] => Array
        (
        [GlossTerm] => "Standard Generalized Markup Language"
        [GlossSee] => "markup"
        [SortAs] => "SGML"
        [GlossDef] => Array
          (
          [para] => "A meta-markup language, used to create markup languages such as DocBook."
          [GlossSeeAlso] => Array
            (
            [0] => "GML"
            [1] => "XML"
            )
          )
        [ID] => "SGML"
        [Acronym] => "SGML"
        [Abbrev] => "ISO 8879:1986"
        )
      )
    )
  )
)
```

Kayyssa Haïssou,
Ewen Horville, Baptiste NEVEJANS

Phase 2 :

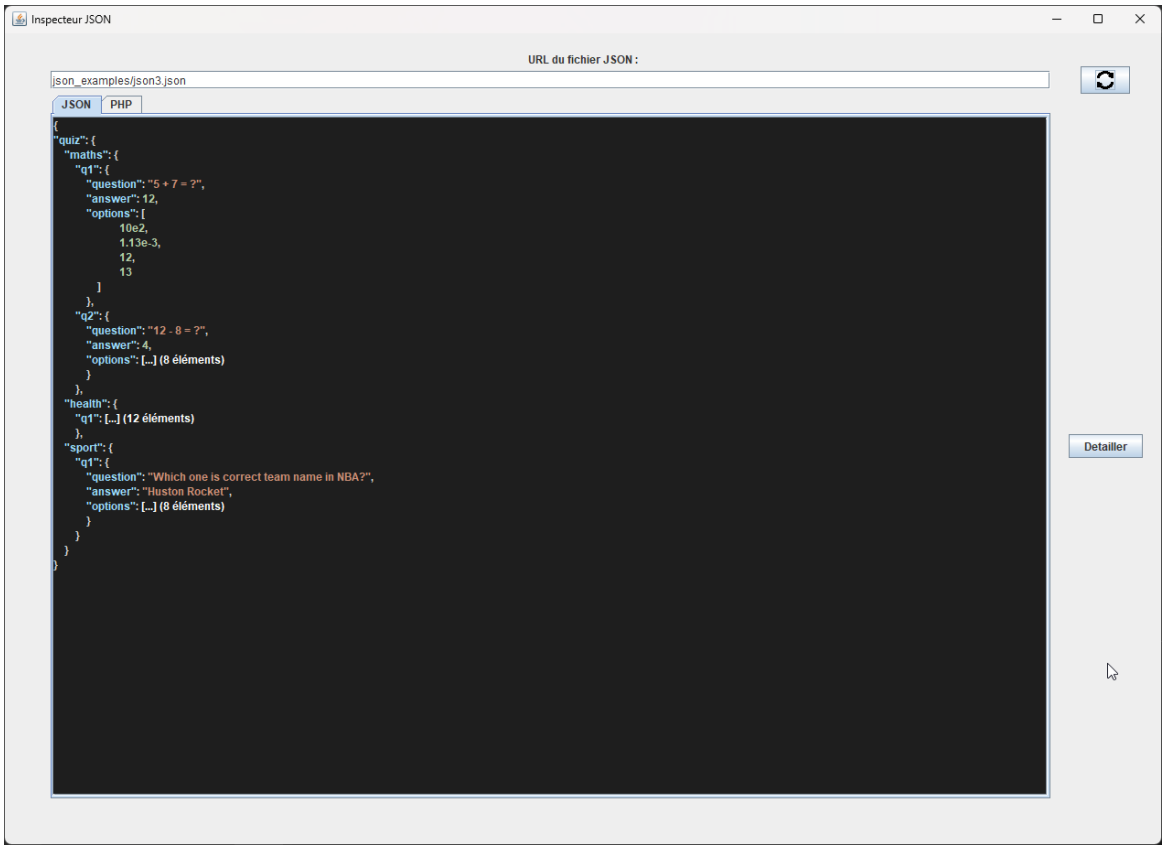
Dans cette phase on met en place l'affichage avec une fenêtre. Cette fenêtre s'affiche en appelant la classe Accueil, l'affichage du panneau Json est fait en appelant la classe AffichageJSON et les couleurs avec les classes ColoredNode et ColoredString permette la gestion de toute les couleurs en fonction des ValueType



Kayyssa Haïssou,
Ewen Horville, Baptiste NEVEJANS

Phase 3 :

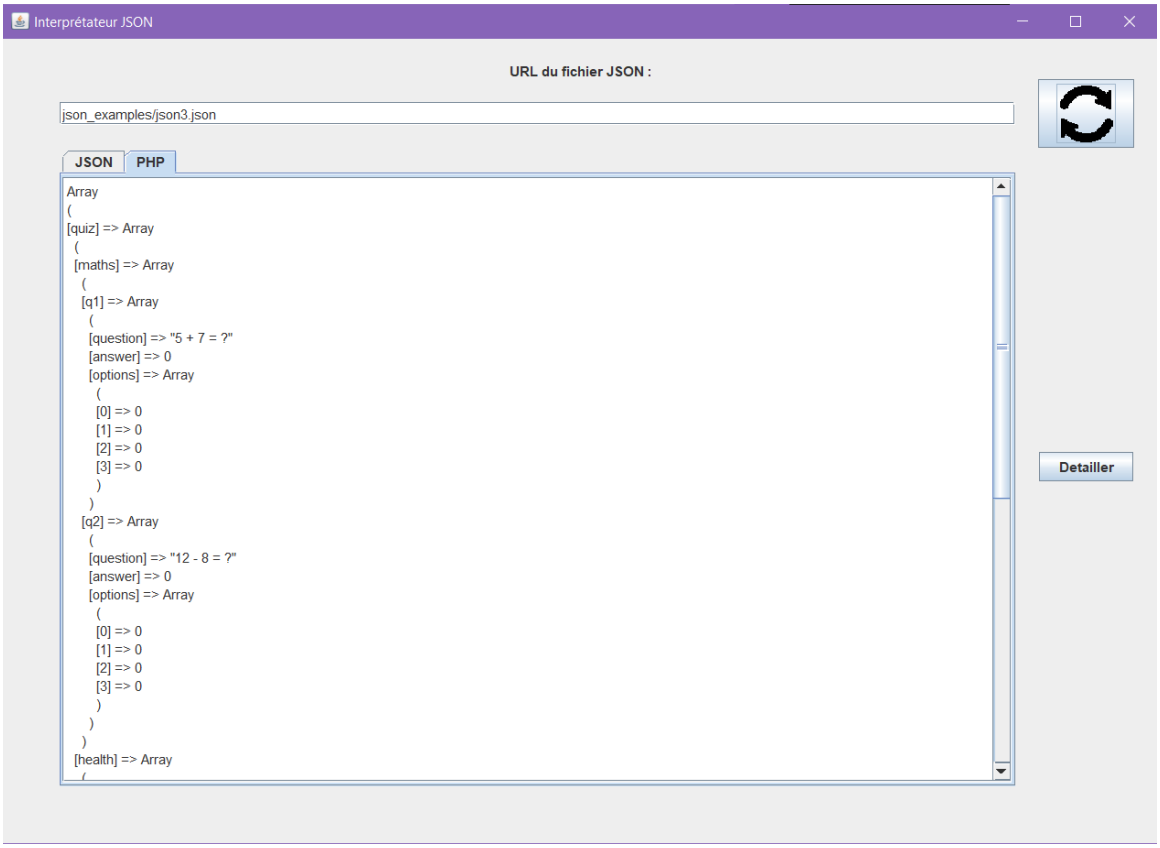
Durant la phase 3, nous avons dû implémenter une mécanique de repli syntaxique. Cela permet de masquer un long bloc d'information et le remplacer par un raccourci. Il ne suffit que de cliquer sur ce raccourcis pour déplier le bloc ou sur l'accolade ouvrante afin de le replier.



Kayyssa Haïssou,
Ewen Horville, Baptiste NEVEJANS

Phase 4 :

Durant cette phase, le but est de passer du JSON vers le PHP, mais puisque la version JSON et PHP existent déjà on a fait le choix de générer le JSON et le PHP en même temp et de changer de panneau avec les onglets JSON et PHP.



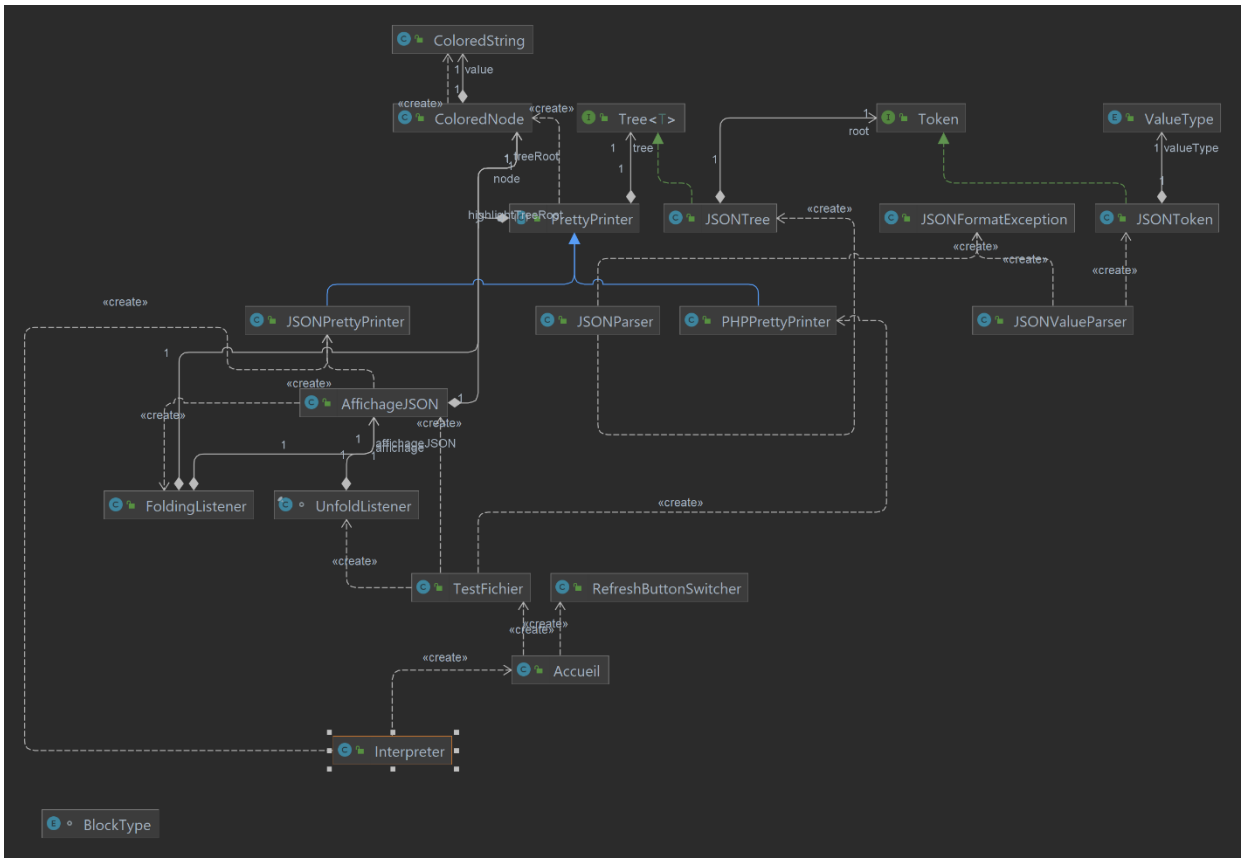
Kayyssa Haïssou,
Ewen Horville, Baptiste NEVEJANS

3. Structure du programme

Le programme est séparé en 4 sous parties majeures : l'API, l'implémentation JSON / PHP, la vue et le contrôleur. L'API contient notamment les squelettes de classes implémentées par le JSON / PHP ainsi que certaines classes utilisées par l'entièreté du projet.

API : A des fins de compatibilité et d'universalité, la plupart des classes créées ont été pensées en dehors de toute syntaxe particulière. Cela les rends très versatiles et factorise beaucoup de code qui aurait été répété sinon.

JSON / PHP : Les classes présentes dans les dossiers JSON / PHP sont des implémentations des classes et interfaces de l'API. Elles y implémentent notamment le moyen de traiter un fichier JSON et le convertir en données abstraites afin de pouvoir ensuite les retranscrire en JSON ou en PHP sans aucune conversion ultérieure.



4. L'arbre de syntaxe abstraite

Le langage Java étant fortement typé, il est impossible de stocker différents types de variables dans une structure et espérer les récupérer via une seule et même méthode. Nous avons donc opté pour la généricité afin de pouvoir stocker différentes classes sous le même nom. Cela les rends difficiles à exploiter si l'on veut utiliser des méthodes / propriétés uniques à ces classes mais très facile lors d'opérations simples (telles que toString).

Nous avons donc une classe *Tree* dont le rôle est de contenir la racine de l'arbre ainsi que d'insérer au bon endroit les valeurs demandées et une interface *Token* contenant les valeurs stockées dans l'arbre. Ces tokens peuvent représenter des primitifs comme des tableaux / dictionnaires. L'arbre utilise des chemins d'accès similaires à ceux utilisés dans une explorateur de dossiers afin de déterminer le chemin à suivre.

Prenons maintenant un exemple de fichier JSON afin d'illustrer l'analyse syntaxique ainsi que la conversion :

```json
{
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized Markup Language",
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
                        "para": "A meta-markup language, used to create markup languages such as DocBook.",
                        "GlossSeeAlso": ["GML", "XML"]
                    },
                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

L'analyse commence tout d'abord par vérifier l'ordre des blocs (tableaux / dictionnaires). Si un bloc s'ouvre, il doit automatiquement être suivi par sa paire ou un autre bloc ouvrant. L'analyse s'assure aussi que le fichier n'est pas vide.

Maintenant, on cherche la première valeur présente dans le JSON. Si c'est une valeur primitive, elle est convertie et la conversion est terminée. Si c'est un tableau / dictionnaire, le convertisseur délimite les bornes du bloc et le traite récursivement jusqu'à trouver des valeurs primitives. On répète l'opération jusqu'à la fin du bloc le plus haut, auquel cas nous avons fini de traiter le fichier donné.

5.Structures de données abstraites utilisées

Nous avons utilisé un arbre de syntaxe abstraite afin de représenter les données du fichier JSON.

Des piles ont aussi été utilisées lors de l'analyse syntaxique afin de vérifier l'ordre et l'imbrication des différents blocs (tableaux, objets).

6.Conclusion

Conclusion Kayyissa : Au début je ne savais pas trop dans quelle direction aller pour mener à bien ce projet. Ewen a heureusement pris les devants pour guider le groupe. J'ai trouvé certaines façons de faire très intéressantes, comme le fait de coder PrettyPrinter de telle manière que plusieurs types de syntaxes peuvent être utilisées pour afficher l'arbre syntaxique. Je me suis principalement occupé de la partie graphique (phase 2), et j'ai beaucoup aimé le découpage en phase qui permet de chercher sans cesse des solutions à partir de la base de code déjà construite.

Conclusion Ewen : L'algorithmie étant l'une des disciplines de la programmation que je connais le mieux et que j'aime le plus, ce projet fût fort rafraichissant et m'a permis de mettre en pratique des notions vues précédemment telles que la notion d'API abstraite afin de faire interface entre deux sous-parties différentes d'un programme. J'ai pu aussi tester mes compétences en essayant de concevoir un système de *pretty print* permettant de s'abstraire de toute syntaxe et donc de pouvoir l'adapter le plus facilement d'un langage à l'autre.

Conclusion Baptiste : J'ai trouvé ce projet très intéressant. Il m'a permis de comprendre et de m'améliorer davantage en développement d'applications. Je me suis concentré sur la première et la quatrième partie du projet en gérant le formatage du Json et du Php. L'inspecteur de JSON a tout de même été compliqué à mettre en œuvre à quelques moments.