TD Algorithmique Avancée : Programmation Java/JSON avec Jackson

Quelques mots sur JSON :

JSON (**J**ava**S**cript **O**bject **N**otation) est un format léger d'échange de données,
- → facile à écrire et à lire pour des humains
- → et facilement analysable ou générable par des machines.

Format de référence pour l'échange de données textuelles sur le Web, JSON a été créé à l'origine comme un sous-ensemble du langage JavaScript, ce qui est rappelé par son acronyme **JavaScript Object Notation**.
Mais aujourd'hui ***JSON est un format indépendant de tout langage*** dont les conventions d'écriture se basent sur des éléments familiers des langages héritant du C comme C++, C#, Java ou PHP pour ne citer qu'eux...
En termes de spécifications techniques, **JSON** est un format générique basé sur **2 structures** :

- → **Une collection de couples nom/valeur** : Divers langages la réifient[1] par un objet, un enregistrement, une structure, un dictionnaire, une table de hachage, une liste typée ou un tableau associatif.
- → **Une liste de valeurs ordonnées**. La plupart des langages la réifient par un tableau, un vecteur, une liste ou une suite.

C'est ces structures de données, communes à la plupart des langages de programmation modernes au nommage près, qui ont permis à JSON de devenir un format d'échange idéal et qui expliquent les raisons de son succès. En effet, le fait que JSON se base sur les structures des langages de programmation favorise son utilisation lors la mise en place d'une interopérabilité entre système.

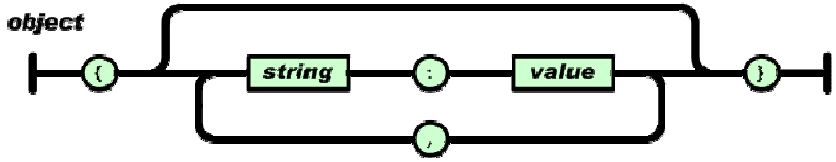
Un exemple simple de données au format JSON visant à modéliser un utilisateur :

```
{
   "name":{
      "last":"Blasquez",
      "first":"Isabelle"
   },
   "gender":"FEMALE",
   "verified":false,
   "msgs":[
      "Message 1",
      "Message 2",
      "Message 3"
   ]
}
```

→ En **JSON** un ***objet*** est représenté par un **ensemble de couples nom/valeur** non ordonnés entre **{ }**

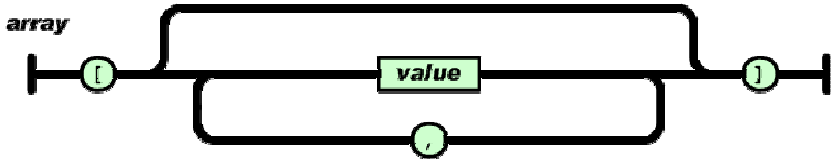
Un objet commence par **{** (accolade gauche) et se termine par **}** (accolade droite).
Chaque nom est suivi de **:** (deux-points).

Au sein d'un objet, chaque couple nom/valeur est séparé par **,** (une virgule).



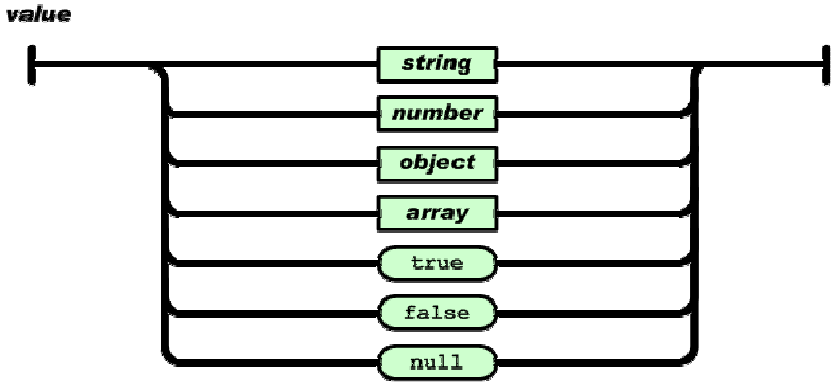
Extrait : http://www.json.org/json-fr.html

→ ***Une liste de valeurs*** est encadrée par des crochets **[ ]**. Les valeurs sont séparées par **,** (virgule)



Extrait : http://www.json.org/json-fr.html

→Une ***valeur*** peut être soit une chaîne de caractères entre guillemets, soit un nombre, soit true ou false ou null, soit un objet soit un tableau. Ces structures peuvent être imbriquées.



Extrait : http://www.json.org/json-fr.html

Comparaion JSON/XML :

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

The same text expressed as XML:

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Extrait : http://json.org/example.html
D'autres extraits sont disponibles sur cette page

En comparaison, le format JSON ne fait pas de différence entre attribut et élément comme en XML. Il est donc moins sujet à variation, et est globalement plus concis et plus lisible. Les valeurs sont typées alors qu'en XML on ne dispose que de chaînes de caractères.
Ainsi, même si JSON et XML permettent tous les deux de représenter de l'information structurée lisible par un humain, on remarquera tout de même que **le format JSON a l'avantage de d'être bien plus léger et concis qu'XML, beaucoup moins verbeux.**

Comment manipuler le format de donner JSON ?
Manipuler du JSON dans les applications web est devenu quasiment incontournable aujourd'hui.
Le site **http://www.json.org/** référence pour chaque langage de programmation, une liste de bibliothèques permettant de réaliser cette opération.

Le site http://www.json.org/ référence également de nombreux liens vers des outils (Editeur, parseur,...) , ainsi que les bibliothèques permettant de manipuler JSON dans de nombreux langages de programmation...

Quelques mots sur Jackson :

Pourquoi Jakson ?
Concernant Java, de nombreuses bibliothèques permettant de manipuler JSON sont disponibles (voir annexe I)
Nous nous intéresserons plus précisément dans ce tutoriel à la bibliothèque **Jackson** qui est une des bibliothèques qui allie à l'heure actuelle puissance et légéreté. Le gros point fort de Jackson se situe au niveau de sa très ***faible empreinte mémoire lors de la désérialisation de gros volumes de données.***
En ne créant qu'un minimum d'objets temporaires lors des opérations de désérialisation, Jackson permet de "soulager" le Garbage Collector de la JVM. ***Jackson est donc recommandé dans le cadre du développement d'applications Android***, mais peut tout aussi bien être utilisé dans projet JAVA standard.

Quelques mots sur Jackson
Jackson est une bibliothèque open-source Java qui permet d'interroger simplement des services **JSON**.
Le site officiel de Jackson est **http://wiki.fasterxml.com/JacksonHome**

Jackson est une bibliothèque complète car elle propose au développeur **3 approches pour la manipulation de données JSON :**

→ le **DataBinding**, qui permet d'établir un ***mapping entre les POJO Java et l'arbre de JSON***. C'est ***l'approche la plus simple*** à utiliser pour transformer le flux JSON en objet métier (désérialisation) et inversesement transformer un objet métier en flux JSON (sérialisation).

→ le **TreeModel**, similaire à DOM (fournit une représentation arborescente en mémoire du document JSON) : c'est ***l'approche la plus flexible***.

→la **Streaming API**, dont le fonctionnement est proche de SAX, permettant de lire et d'écrire un document JSON en s'appuyant sur un système évènementiel : c'est ***l'approche la plus performante***.

Chacune de ces approches apporte son lot d'avantages et d'inconvénients se révélant plus ou moins adaptée suivant le contexte d'utilisation.

Sources :
http://www.json.org/ (en français: http://www.json.org/json-fr.html)
http://fr.wikipedia.org/wiki/JavaScript_Object_Notation
http://jsonformatter.curiousconcept.com/
https://www.projet-plume.org/ressource/format-json
http://blog.excilys.com/2010/02/25/android-pour-lentreprise-6-oubliez-gson-jackson-rocks-my-world/

JSON est décrit dans le RFC 4627 : D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), 2006 (http://www.ietf.org/rfc/rfc4627.txt)

Remarque :
[1] Quelques mots sur la ***réification*** (extrait de http://fr.wikipedia.org/wiki/R%C3%A9ification)
La réification consiste à transformer ou à transposer une abstraction en un objet concret, à appréhender un concept comme une chose concrète. Le terme est aussi employé à propos des personnes vivantes. En informatique, la réification consiste à transformer un concept en un objet informatique.

Installation de Jackson :

Jackson se présente sous la forme d'une **bibliothèque open source autonome** découpée en plusieurs modules maven. Vous pouvez retrouver tous ces modules sur : **https://github.com/FasterXML.**
Ces modules Maven peuvent également être téléchargés directement sous la forme d'archive JAR.

Jackson est donc une librairie JSON très complète et bien documentée. La documentation vous permettant d'installer Jackson est disponible sur **http://wiki.fasterxml.com/JacksonDownload**

Pour installer Jackson, deux possibilités s'offrent à vous, vous pouvez :

→ soit directement télécharger les modules sous la forme d'archives JAR et les inclure dans votre projet en les ajoutant au CLASSPATH

→ soit utiliser un projet Maven. Et comme Jackson est découpée en plusieurs modules Maven, il est donc configurable à souhait : une fois le coeur de l'API ajouté en tant que dépendance, les autres modules ne seront à ajouter qu'en cas de besoin (voir Annexe II).

Dans le cadre de ce tutoriel, nous allons utiliser la première solution.

Création du projet tutoJackson :

→Commencez par lancer Eclipse et créer un Projet Java **tutoJackson.**
Ajouter au CLASSPATH du projet les JAR : **jackson-databind-2.1.1.jar** et **jackson-core-2.1.1.jar** et **jackson-annotations-2.1.1** que vous pouvez récupérer sur la zone libre
ou directement la page : http://wiki.fasterxml.com/JacksonDownload

Remarque : Jackson est une librairie modulaire : seul le jar **jackson-core** est indispensable, mais il doit être utilisé conjointement avec d'autres jars en fonction des besoins

→Créer dans votre projet les packages suivant :
com.iut.tutoJackson.metier
com.iut.tutoJackson.essai

Une liste d'utilisateurs au format JSON pour commencer ...

La première partie de ce tutoriel va consister à travailler à la ***sérialisation/désérialisationd'une liste d'utilisateurs*** modélisé à l'aide de la classe **User** ci-contre.

User
-firstname: String -lastname: String -login: String -twitter: String -web: String

→ Dans le package **com.iut.tutoJackson.metier,** créer rapidement la classe **User.** (sans oublier le toString !) Cette classe est également disponible sur la zone libre d'où vous pouvez l'importer.

Remarques :
→ Vous pouvez accéder à la javadoc de Jackson à partir du site : **http://jackson.codehaus.org/** (documentation accessible via la frame à droite du site qui renvoie sur le nouveau site **fasterxml**)

→ Un outil utile pour formater et vérifier vos fichiers JSON au cours du tutoriel : **JSON Formatter et validator** **http://jsonformatter.curiousconcept.com/**

I. Le Data Binding

Le **DataBinding** permet d'établir un ***mapping entre les POJO Java et l'arbre de JSON***. C'est ***l'approche la plus simple*** à utiliser pour transformer le flux JSON en objet métier (désérialisation) et inversesement transformer un objet métier en flux JSON (sérialisation)

Sur le site officiel de Jackson, on peut treouver de la documentation sur la mise en place du data Binding sur : **http://wiki.fasterxml.com/JacksonDataBinding** et **http://wiki.fasterxml.com/JacksonInFiveMinutes** et **https://github.com/FasterXML/jackson-databind**

Au coeur de l'API Jackson, se trouve la classe **ObjectMapper** qui propose les méthodes de lecture et d'écriture des données **JSON : l**es méthodes utilisées pour le data Binding seront les méthode ***writeValue*** et ***readValue*** .

La documentation concernant le package **jackson-databind** est consultable sur : **http://fasterxml.github.com/jackson-databind/javadoc/2.1.0/**

➢ Data Binding : POJO Java → JSON (Sérialisation)

Serialization

There is no real difference between flavors when serializing Java objects -- differences only matter when reading JSON and mapping (binding) it to Java objects. So serialization is achieved by:

```
ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(dst, myBean); // where 'dst' can be File, OutputStream or Writer
```

void	**writeValue**(File resultFile, Object value) Method that can be used to serialize any Java value as JSON output, written to File provided.
void	**writeValue**(JsonGenerator jgen, Object value) Method that can be used to serialize any Java value as JSON output, using provided JsonGenerator.
void	**writeValue**(OutputStream out, Object value) Method that can be used to serialize any Java value as JSON output, using output stream provided (using encoding JsonEncoding.UTF8).
void	**writeValue**(Writer w, Object value) Method that can be used to serialize any Java value as JSON output, using Writer provided.

Extraits de : http://wiki.fasterxml.com/JacksonDataBinding et http://fasterxml.github.com/jackson-databind/javadoc/2.1.0/

→ Pour commencer dans le package **com.iut.tutoJackson.essai**, créer une classe **TestJacksonDataBinding**

Dans le main, nous allons commencer par sérialiser un objet de type User dans un fichier JSON.
D'après les extraits de code précédents, pour ***procéder à la sérialisation*** vous devez écrire dans votre **main**, au minimum le code suivant (sans oubliez les exceptions qui vont bien !)

```
File dest = new File("users.json");
User user = new User("Isabelle","Blasquez","iblasquez","@iblasquez","www.iutagile.com");
mapper.writeValue(dest,user);
```

→ Exécutez et ouvrir le fichier **user.json** pour visualiser son contenu.

→ Nous souhaitons maintenant sérialiser deuxobjets de type User dans notre fichier.
Si vous modifiez le code que vous venez d'écrire de la manière suivante : que constatez-vous dans le nouveau fichier **user.json** ?

```
ObjectMapper mapper = new ObjectMapper();
File dest = new File("users.json");
User user = new User("Isabelle","Blasquez","iblasquez","@iblasquez","www.iutagile.com");
User user2 = new User("Sacha",
                      "Bourg-Palette","sbourgpalette","@SachaBourgPalet","www.pokemon.com");

mapper.writeValue(dest,user);
mapper.writeValue(dest,user2);
```

Le fichier obtenu est de la forme suivante : seul le dernier **User** a été mémorisé dans le fichier **JSON**.

```
{
   "firstname":"Sacha",
   "lastname":"Bourg-Palette",
   "login":"sbourgpalette",
   "twitter":"@SachaBourgPalet",
   "web":"www.pokemon.com"
}
```

→ En effet, Un seul objet Java ne peut être sérialisé dans un fichier JSON : les données dans un fichier **JSON** sont donc mémorisées sous forme d'arbre : un seul objet à la racine.
Ainsi, si l'on souhaite sérialiser plusieurs utilisateurs dans un même fichier **JSON**, il sera nécessaire de sérialiser une ***collection*** contenant les différents objets à sérialiser.

Pour commencer, nous pouvons utiliser une simple **ArrayList.**
Modifiez votre code de la manière suivante et exécutez-le.

```
ArrayList<User> users = new ArrayList<User>();
users.add(user);
users.add(user2);
mapper.writeValue(dest,users);
```

```
[
   {
      "firstname":"Isabelle",
      "lastname":"Blasquez",
      "login":"iblasquez",
      "twitter":"@iblasquez",
      "web":"www.iutagile.com"
   },
   {
      "firstname":"Sacha",
      "lastname":"Bourg-Palette",
      "login":"sbourgpalette",
      "twitter":"@SachaBourgPalet",
      "web":"www.pokemon.com"
   }
]
```

Remarquez que le fichier ***users.json*** commence cette fois-ci par un **crochet [** et se termine aussi par un **crochet]** , ce qui conformément à la convention JSON indique que le fichier JSON contient un ***array*** (voir page 2 du tutoriel), et qui est cohérent avec le code Java écrit qui utilise **ArrayList** et conforme au mapping des types JSON ↔ Java indiqué en page 5 de ce tutoriel.

→ Si vous souhaitez maintenant que votre fichier **JSON** apparaisse de la manière suivante, c-a-d avec un premier noeud nommé **users**, il est nécessaire de créer en Java une nouvelle classe qui aura un attribut **users** de type **User** et d'effectuer la sérialisation **JSON** (appel de la méthide **writeValue**) sur un objet de cette classe.

```
{
   "users":[
      {
         "firstname":"Isabelle",
         "lastname":"Blasquez",
         "login":"iblasquez",
         "twitter":"@iblasquez",
         "web":"www.iutagile.com"
      },
      {
         "firstname":"Sacha",
         "lastname":"Bourg-Palette",
         "login":"sbourgpalette",
         "twitter":"@SachaBourgPalet",
         "web":"www.pokemon.com"
      }
   ]
}
```

✓ Dans le paquetage **com.iut.tutoJackson.metier**, créer une classe `ListeUsers` de la forme suivante :

```
public class ListeUsers {

    private List<User> users = new ArrayList<User>();
    public List<User> getUsers() return users;
    public void setUsers(List<User> lettres) {
        this.users = users;
    }
}
```

✓ Modifier le code de votre fichier de la manière suivante et exécutez-le. Cconsultez le nouveau fichier **JSON** obtenu.

```
ListeUsers users = new ListeUsers();
users.getUsers().add(user);
users.getUsers().add(user2);
mapper.writeValue(dest,users);
```

Remarquez que le fichier ***users.json*** commence cette fois-ci par une **accolade {** et se termine aussi par une **accolade }** , ce qui conformément à la convention JSON indique que le fichier JSON contient un ***objet*** (voir page 1 du tutoriel), et qui est cohérent avec le code Java écrit qui sérialise un objet de type **ListeUsers** .

→ Utilisation d'un objet de type **ObjectWriter** pour formatter le fichier de sortie **users.json**

Il est possible d' obtenir une sortie formatée du fichier **JSON** généré en récupérant un objet de type ObjectWriter depuis l'objet de type **ObjectMapper** et en appliquant la méthode writeValue sur l'objet de type **ObjectWriter**
Pour obtenir un fichier **users.json** formaté, il suffit de modifier le code de la manière suivante :

```
ObjectMapper mapper = new ObjectMapper();
// ... code déjà écrit ...
ObjectWriter writer = mapper.writerWithDefaultPrettyPrinter();
writer.writeValue(dest,users);
```

Tapez ce code, exécutez-le et visualisez le fichier **users.json**

➢ Data Binding : JSON→ POJO Java(Désérialisation)

Full Data Binding

When using full data binding, deserialization type must be fully specified as something other than `Object.class`. For example:

```
MyBean value = mapper.readValue(src, MyBean.class); // 'src' can be File, InputStream, Reader, String
```

The main complication is handling of Generic types: if they are used, one has to use `TypeReference` object, to work around Java Type Erasure:

```
MyType<String> genValue = mapper.readValue(src, new TypeReference<MyType<String>>() { });
```

(why? Because that is one of limited number of ways of passing full type information; mechanism called "Supertype Token")

Extrait de : http://wiki.fasterxml.com/JacksonDataBinding

→ ***Désérialisation d'un fichier JSON ne contenant qu'un seul objet de type User :***

Importer dans votre projet le fichier **users_1.json** (qui correspond en fait à la première copie d'écran de la page 6).
Dans un fichier test, nous allons maintenant procéder à la désérialisation, toujours en s'appuyant sur un objet **ObjectMapper**, mais en s'appuyant cette fois-ci sur une méthode **readValue**.
D'après ce qui précède, pour ***procéder à la désérialisation*** vous devez donc écrire dans votre **main**, au minimum le code suivant (sans oubliez les exceptions qui vont bien !)

```
ObjectMapper mapper = new ObjectMapper();
File src = new File("users_1.json");
User = mapper.readValue(src,User.class);

System.out.println(user); pour visualiser dans la console !
```

Exécutez et visualisez le contenu du fichier **user_1.json** dans la console !

→ ***Désérialisation d'un fichier JSON contenant une liste de valeurs de type User :***

Importer dans votre projet le fichier **users_2.json** (qui correspond en fait à la deuxième copie d'écran de la page précédente). Dans un fichier test, nous allons maintenant procéder à la désérialisation de ce fichier qui contient deux objets de type User.
Cet exemple va nous permettre d'illustrer le deuxième point de l'extrait précédent c-a-d l'utilisation de la classe `TypeReference` pour la désérialisation de type générique.

Ecrire le code permettant de procéder à la désérialisation de ce fichier...

Le code que vous venez d'écrire pour procéder à la désérialisation de **users_2.json** devrait être similaire au code suivant ;-)

```
ObjectMapper mapper = new ObjectMapper();
File src = new File("users_2.json");
ArrayList<User> users = mapper.readValue(src,new TypeReference<ArrayList<User>>() { });
    //Affichage du résultat
    for (User user : users)
        {
        System.out.println(user);
        System.out.println("----------------");
        }
```

→ ***Désérialisation d'un fichier JSON contenant un objet contenant une liste de Users :***

Importer dans votre projet le fichier **users_3.json** (qui correspond en fait à la copie d'écran de la page 7).
Dans un fichier test, écrire le code nécessaire pour procéder à la dé-sérialisation de ce fichier **JSON**.

Le Data Binding que nous venons de voir jusqu'ici est appelé dans la documentation "Full Data Binding".
En outre, Jackson rend également possible le Data Binding non typé sur des données brutes appelé "Simple Data Binding" ou "Raw Data Binding".

➢ Simple Data Binding (ou Raw Data Binding) : Désérialisation

Tout d'abord, il faut savoir que par défaut, Jackson se base sur les conventions suivantes pour réaliser le mapping entre les données JSON et les POJO Java.

Types used for Simple data binding are:

JSON Type	Java Type
`object`	`LinkedHashMap<String,Object>`
`array`	`ArrayList<Object>`
`string`	`String`
`number (no fraction)`	`Integer`, `Long` or `BigInteger` (smallest applicable)
`number (fraction)`	`Double` (configurable to use `BigDecimal`)
`true|false`	`Boolean`
`null`	`null`

Extrait de : http://wiki.fasterxml.com/JacksonDataBinding

Dans le cas où vous n'avez pas (et/ou ne voulez pas créer) de classes Java spécifiques en correspondance directe avec le format des données du flux JSON, il est quand même possible d'effectuer un Data Binding non typé sur ces données JSON, données dites données "brutes" (Raw Data Binding).

Lors de la désérialisation, le type de correspondance Java utilisé devra alors être **Object.class** (ou **Map.class**, **List.class**, **String []. Class**), comme l'indique l'extrait suivant issu de http://wiki.fasterxml.com/JacksonDataBinding

```
Object root = mapper.readValue(src, Object.class);
Map<?,?> rootAsMap = mapper.readValue(src, Map.class);
```

Also please note that to enable generic type information (like "Map<String,Object>"), you have to use `TypeReference` container as explained above

→ Dans un fichier test, écrire le code suivant (sans oublier les exceptions!) qui permet, à partir des remarques précédentes, de procéder à la dé-sérialisation d'un fichier **JSON (users-3.json) en utilisant un raw databinding.**

```
ObjectMapper mapper = new ObjectMapper();
File src = new File("users_3.json");
Map<String, ArrayList<User>> users = mapper.readValue(src,
                                new TypeReference<LinkedHashMap<String, ArrayList<User>>>() { });
ArrayList<User> usersList = users.get("users"); // Récupération de la liste des utilisateurs

//Affichage du résultat dans la console !
for (User user : usersList)
    {
    System.out.println(user);
    System.out.println("----------------");
    }
```

→ Si vous souhaitez écrire un code plus lisible, vous pouvez rajouter dans votre package **com.iut.tutoJackson.metier** une classe **Users** de la forme suivante :

```
public class Users extends HashMap<String, ArrayList<User>> {
    private static final long serialVersionUID = 1L;
}
```

et "simplifier" le code précédent de la manière suivante :

```
ObjectMapper mapper = new ObjectMapper();
File src = new File("users_3.json");
Users users = mapper.readValue(src, Users.class);
ArrayList<User> usersList = users.get("users"); // Récupération de la liste des utilisateurs

//Affichage du résultat dans la console !
//... code identique au précédent ...
```

On comprend ainsi peut être mieux avec ce code que la classe **Users** sert de ***"hashmap de correspondance"*** entre les données du fichier JSON et les classes Java.

➢ Simple Data Binding (ou Raw Data Binding) : Sérialisation

Pour l'écriture de données brutes Java dans un flux JSON, l'idée reste la même.
On part d'une Map Java contenant des couples clés / valeurs avec des valeurs pouvant également être des objets plus complexes comme des listes ou des maps, et on utilise la méthode **writeValue()** de l'**ObjectMapper** avec en argument la Map parent. Le code suivant est extrait de : http://wiki.fasterxml.com/JacksonInFiveMinutes

Afficher/masquer les numéros de lignes

```
1 Map<String,Object> userData = new HashMap<String,Object>();
2 Map<String,String> nameStruct = new HashMap<String,String>();
3 nameStruct.put("first", "Joe");
4 nameStruct.put("last", "Sixpack");
5 userData.put("name", nameStruct);
6 userData.put("gender", "MALE");
7 userData.put("verified", Boolean.FALSE);
8 userData.put("userImage", "Rm9vYmFyIQ==");
```

This obviously works both ways: if you did construct such a Map (or bind from JSON and modify), you could write out just as before, by:

```
mapper.writeValue(new File("user-modified.json"), userData);
```

Nous n'implémenterons pas ce code là dans le cadre de ce tutoriel.

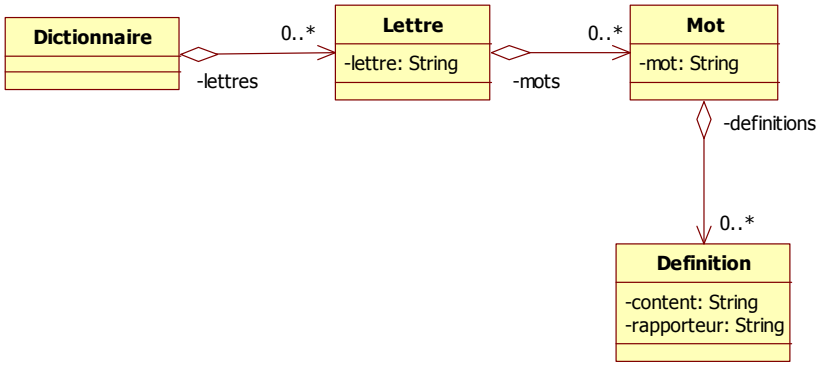
➢ Les annotations Jackson via un exemple un peu plus complet : un dictionnaire au format JSON

Jackson propose un certain nombre d'annotations.
Dans le cadre de ce tutoriel nous allons manipuler, sur un exemple modélisant un dictionnaire, les annotations Jackson suivantes : **@JsonProperty, @JsonIgnore, @JsonIgnoreProperties, @JsonFormat**

La liste complète des annotations Jackson est consultable à l'adresse suivante :
https://github.com/FasterXML/jackson-annotations/wiki/JacksonAnnotations
Des exemples sont proposés sur: https://github.com/FasterXML/jackson-annotations
La documentation concernant les annotations est disponible sur http://fasterxml.github.com/jackson-annotations/javadoc/2.1.0/

Le dictionnaire côté Java

Le diagramme de classes suivant a pour but de modéliser le contenu d'un dictionnaire.
Cette modélisation a été réalisée à partir du cahier des charges suivant :
*"Un dictionnaire se compose d'un **ensemble de lettres**, ayant elles-même **un ensembe de mots imbriqués**. Ces mots auront à leur tour **plusieurs définitions possibles**."*



→Importez dans votre package **com.iut.tutoJackson.metier**, les classes **Dictionnaire**, **Lettre**, **Mot** et **Definition** implémentées à partir du diagramme de classes précédent et disponibles sur la zone libre.

Databinding sur le dictionnaire :

→Importez dans votre package **com.iut.tutoJackson.essai**, la classe **TestDataBindingDico** et implémentez le code nécessaire afin de ***sérialiser un dictionnaire dans un fichier nommé dictionnaire.json*** et ***désérialiser*** ensuite le le dictionnaire depuis le fichier **dictionnaire.json.**

Utilisez de préférence un **full Databinding** afin que votre code soit le plus simple possible.
Utilisez également un **ObjectWriter** pour une meilleure lisibilité du fichier ***dictionnaire.json***

Annotation @JsonProperty : "Property Naming"

L'annotation **@JsonProperty** permet de définir sur un champ, un getter ou un setter le nom de la propriété de mapping pour le document JSON.

→ Dans la classe **Definition**, rajoutez l'annotation **@JsonProperty** comme indiqué ci-dessous, juste au-dessus de la déclaration de l'attribut **content**

```
@JsonProperty(value = "contenu")
private String content;
```

→ Exécutez et ouvrir le fichier **dictionnaire.json** afin de verifier que le flux JSON contient maintenant **contenu** à la place de **conten**t.

Annotation @JsonIgnore : "Property Inclusion"

L'annotation **@JsonIgnore** permet de préciser qu'un champ doit être ignoré lors du mapping.

→ Dans la classe **Definition**, rajoutez l'annotation **@JsonIgnore** comme indiqué ci-dessous, juste au-dessus de la déclaration de l'attribut **rapporteur**

```
@JsonIgnore
private String rapporteur;
```

→ Exécutez et ouvrir le fichier **dictionnaire.json** afin de verifier que le flux JSON ne contient maintenant plus **rapporteur**.

→ Une fois le test effectué, supprimer l'annotation **@JsonIgnore** afin que le **rapporteur** se retrouve bien dans le flux JSON pour la suite du tutoriel.

Annotation @JsonIgnoreProperties : "Property Inclusion"

L'annotation **@JsonIgnoreProperties** permet d'annoter une classe et préciser les différents noms de champs à ignorer lors du mapping.

→ Dans la classe **Definition**, rajouteZ l'annotation **@JsonIgnoreProperties** comme indiqué ci-dessous, juste au-dessus de la déclaration de la classe **Definition**

```
@JsonIgnoreProperties({ "rapporteur", "content" })
public class Definition {
```

→ Exécutez et ouvrir le fichier **dictionnaire.json** afin de verifier que le flux JSON ne contient maintenant plus **rapporteur**, ni **contenu**

→ Une fois le test effectué, supprimer l'annotation **@JsonIgnoreProperties** afin que le **rapporteur** et le **contenu** se retrouve bien dans le flux JSON pour la suite du tutoriel.

A ce stade du tutoriel, vous ne devez garder dans la classe Definition que l'annotation **@JsonProperty** sur l'attribut **content**

Modification de la classe Definition

Definition
-content: String -rapporteur: String -date: java.util.Date

→Dans la classe **Definition**, rajoutez un attribut **date** de type **java.util.Date**, ainsi que son **getteur** et son **setteur** associé.
Ajoutez également l'affichage de la **date** dans la méthode **toString** le plus simplement possible :

```
public String toString() {
        return "\t content : " + content + "\n \t" +
                " rapporteur : "+rapporteur +"\n \t" +
                "date : "+ date;
    }
```

→Exécutez le fichier **TestDataBindingDico**.
Pour l'instant tout va bien, on peut juste constater que le fichier dictionnaire.json contient **date:null**

→Nous allons donc maintenant modifier notre jeu d'essai, afin de rajouter une date à nos définitions.
Modifiez le jeu d'essai du fichier **TestDataBindingDico** de la manière suivante :

```
Definition definition = new Definition();
definition.setContent("Un abricot est ...");
definition.setRapporteur("Kent Back");
definition.setDate(new Date());
// ... un peu plus loin ...
autreDefinition.setDate(new Date());
// ... un peu plus loin ...
definition2.setDate(new Date());
```

→Exécutez et visualiser le fichier **dictionnaire.json** : la date dans le fichier JSON n'est pas lisible par un humain...

Annotation @JsonFormat : "Deserialization and Serialization details"

Depuis Jackson 2.0, l'annotation **@JsonFormat** permet de spécifier le format à utiliser lors de la sérialisation de certains types spécifiques comme la date ou l'heure.

→ Dans la classe **Definition**, rajoutez l'annotation **@JsonFormat** comme indiqué ci-dessous, juste au-dessus de la déclaration de l'attribut **Date**

```
@JsonFormat(shape=JsonFormat.Shape.STRING, pattern="yyyy-MM-dd,HH:00", timezone="CET")
private Date date;
```

→ Exécutez et ouvrir le fichier **dictionnaire.json** afin de verifier que le flux JSON contient maintenant une **date** lisible par un oeil humain et formatée de la manière suivvante : **yyyy-MM-dd,HH:00**

→ Dans la classe **Definition**, modifiez l'annotation **@JsonFormat** comme de la manière suivante :

```
@JsonFormat(shape=JsonFormat.Shape.STRING, pattern="dd-MM-yyyy", timezone="CET")
private Date date;
```

→ Exécutez et ouvrir le fichier **dictionnaire.json** afin de verifier que le flux JSON contient maintenant une **date** au format : **dd-MM-yyyy**

→ Une FAQ sur les dates est disponible sur : **http://wiki.fasterxml.com/JacksonFAQDateHandling**
où il est indiqué que la conversion des dates peut se faire par configuration sur l'ObjectMapper ou par annotation comme nous venons de l'écrire. A priori, il est également recommandé d'éviter d'utiliser le type **java.sql.date** ...

II. Le Tree Model

Utilisée dans la plupart des cas, l'approche par Data Binding peut s'avérer insuffisante dans certains contextes où l'on souhaite avoir un contrôle plus grand sur le contenu de l'arbre associé au document JSON.

Jackson propos une ***approche plus flexible*** que le Data Binding basée sur un concept de **TreeModel** ("modèle d'arbre"). Similaire à **DOM** de XML, cette approche permet de représenter en Java les données JSON sous forme d'arbre composé de noeuds.

L'objet central du **Tree Model** est le **JsonNode**.
La classe **JsonNode** se trouve dans le package **jackson-databind** dont la documentation est consultable sur :
http://fasterxml.github.com/jackson-databind/javadoc/2.1.0/

Sur le site officiel de Jackson, on peut trouver de la documentation sur la mise en place de l'API Tree Model sur :
http://wiki.fasterxml.com/JacksonTreeModel et **http://wiki.fasterxml.com/JacksonInFiveMinutes**

➢ Tree Model API : Lire et mettre à jour un flux au format JSON via un JsonNode

Trees are usually constructed using `ObjectMapper`, similar to how Data Binding is done. There are actually two distinct methods for doing this:

Afficher/masquer les numéros de lignes

```
1  // general method, same as with data binding
2  ObjectMapper mapper = new ObjectMapper();
3  // (note: can also use more specific type, like ArrayNode or ObjectNode!)
4  JsonNode rootNode = mapper.readValue(src, JsonNode.class); // src can be a File, URL, InputStream etc
5
6  // or, with dedicated method; requires a JsonParser instance
7  JsonParser jp = ...; // construct using JsonFactory (can get one using ObjectMapper.getJsonFactory)
8  rootNode = mapper.readTree(jp);
9  // (most useful when binding sub-trees)
```

Extrait de : http://wiki.fasterxml.com/JacksonTreeModel

→ Un objet de type **ObjectMapper** permet également de charger les objets JSON dans un modèle d'arbre (**JsonNode).**
La racine du document JSON, récupérée via un appel à la méthode **readTree()** de l'**ObjectMappe**r, sera ainsi de type **JsonNode**.
Il est à noter que le chargement en mémoire de l'arbre du document JSON sera un peu plus long que lors d'un Data Binding (où l'ObjectMapper charge directement le flux JSON dans un bean ou dans une Map).

→ **Pour accéder aux données de l'arbre** (c-a-d à un nouveau noeud), il suffit d'appeler la méthode **path** ou **get** sur **un JsonNode** (racine ou noeud intermédaire).

JsonNode	**get**(int index) Method for accessing value of the specified element of an array node.
JsonNode	**get**(String fieldName) Method for accessing value of the specified field of an object node.

abstract JsonNode	**path**(int index) This method is similar to get(int), except that instead of returning null if no such element exists (due to index being out of range, or this node not being an array), a "missing node" (node that returns true for isMissingNode()) will be returned.
abstract JsonNode	**path**(String fieldName) This method is similar to get(String), except that instead of returning null if no such value exists (due to this node not being an object, or object not having value for the specified field), a "missing node" (node that returns true for isMissingNode()) will be returned.

Extrait de la documentation de la classe **com.fasterxml.jackson.databind.JsonNode**
sur http://fasterxml.github.com/jackson-databind/javadoc/2.1.0/

→ **Pour convertir les données JSON des noeuds de type JsonNode,** il suffit d'utiliser une méthode de type **asXXX**

boolean	**asBoolean**() Method that will try to convert value of this node to a Java **boolean**.
boolean	**asBoolean**(boolean defaultValue) Method that will try to convert value of this node to a Java **boolean**.
double	**asDouble**() Method that will try to convert value of this node to a Java **double**.
double	**asDouble**(double defaultValue) Method that will try to convert value of this node to a Java **double**.
int	**asInt**() Method that will try to convert value of this node to a Java **int**.
int	**asInt**(int defaultValue) Method that will try to convert value of this node to a Java **int**.
long	**asLong**() Method that will try to convert value of this node to a Java **long**.
long	**asLong**(long defaultValue) Method that will try to convert value of this node to a Java **long**.
stract String	**asText**() Method that will return a valid String representation of the container value, if the node is a value node (method isValueNode() returns true), otherwise empty String.

→ **Pour mettre à jour** un arbre déjà chargé en mémoire, il est nécessaire de caster la racine de type **JsonNode** en **ObjectNode** afin de pouvoir par exemple ajouter de nouvelles données (à l'aide des méthodes **put**) ou supprimer des données (à l'aide d'une méthode **remove**)
Vous pouvez consulter la documentation de la classe **com.fasterxml.jackson.databind.node.ObjectNode** sur http://fasterxml.github.com/jackson-databind/javadoc/2.1.0/

→ **Pour écrire un arbe de type JsonNode dans un flux JSON,** il suffit d'utiliser la méthode **writeValue** sur l'ObjectMapper (comme dans le cas du Data Binding).

Voilà le code qui devrait être écrit pour ***lire le flux de données JSON*** du fichier **users_1.json** dans un **Tree Model**. L'arbre chargé est ensuite modifié et écrit dans un nouveau flux JSON, le fichier **users_1_treeModel.json**

```
// ///////////////
// Lecture
// ///////////////
ObjectMapper mapper = new ObjectMapper();
File src = new File("users_1.json");
JsonNode rootNode = null;
try {
    rootNode = mapper.readTree(src);

    JsonNode nameNode = rootNode.path("firstname");
    System.out.println(nameNode.asText());

    JsonNode lastNameNode = rootNode.path("lastname");
    System.out.println(lastNameNode.asText());

    JsonNode loginNode = rootNode.path("login");
    System.out.println(loginNode.asText());

    JsonNode twitterNode = rootNode.path("twitter");
    System.out.println(twitterNode.asText());

    JsonNode webNode = rootNode.path("web");
    System.out.println(webNode.asText());
} catch (JsonProcessingException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}

// ////////////////////////////////////////////////
// Mise à jour et écriture dans un nouveau flux JSON
// ////////////////////////////////////////////////
ObjectNode currentRootNode = (ObjectNode) rootNode;
currentRootNode.put("nickname", "Isa");

((ObjectNode) rootNode).remove("login");

ObjectWriter writer = mapper.writerWithDefaultPrettyPrinter();
File dest = new File("users_1_treeModel.json");
try {
    writer.writeValue(dest, currentRootNode);
} catch (JsonGenerationException e) {
    e.printStackTrace();
} catch (JsonMappingException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

→ Prenez le temps de comprendre ce code.

→ **Ce code est disponible sur la zone libre dans la classe TestJacksonTreeModelUser**, importez ce fichier dans votre projet et exécutez-le. Visualisez alors le fichier **users_1_treeModel.json**

→ Importez également la classe **TestTreeModelDico. Le code de cette classe permet de manipuler le fichier dictionnaire.json via un Tree Model.** Ce code est un peu plus complexe que le précédent car plusieurs objets sont dans le flux JSON. Remarquez l'utilisation d'un **Iterator** pour parcourir tous ces objets.

```
// ///////////////
// Lecture
// ///////////////
ObjectMapper mapper = new ObjectMapper();
File src = new File("dictionnaire.json");

try {
    JsonNode root = mapper.readTree(src);
    JsonNode lettres = root.path("lettres");
    System.out.println("Lettres");

    for (Iterator<JsonNode> iterator = lettres.iterator(); iterator.hasNext();) {
        JsonNode lettre = iterator.next();
        JsonNode titre = lettre.get("lettre");
        System.out.println(titre.asText());
        JsonNode mots = lettre.get("mots");

        for (Iterator<JsonNode> iterator2 = mots.iterator(); iterator2.hasNext();) {
            JsonNode mot = iterator2.next();
            titre = mot.get("mot");
            System.out.println("==> Mot : " + titre.asText());
            JsonNode definitions = mot.get("definitions");
            System.out.println("DEFINITIONS");

          for (Iterator<JsonNode> iterator3 = definitions.iterator(); iterator3.hasNext();) {
                JsonNode definition = iterator3.next();
                System.out.println("\t --> Définition : "
                        + definition.get("contenu").asText() + " / "
                        + definition.get("rapporteur").asText() + " / "
                        + definition.get("date").asText());
            }
        }
    }

    // ////////////////////////////////////////////////
    // Mise à jour et écriture dans un nouveau flux JSON
    // ////////////////////////////////////////////////
    ObjectNode currentRootNode = (ObjectNode) root;
    currentRootNode.put("note", "Note");
    ObjectWriter writer = mapper.writerWithDefaultPrettyPrinter();
    File dest = new File("dictionnaire_treeModel.json");
    writer.writeValue(dest, currentRootNode);

} catch (JsonProcessingException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

→ Prenez le temps de comprendre ce code suivant et exécutez-le. N'oubliez pas de visualiser le fichier **dictionnaire_treeModel.json**

Ces exemples permettent de montrer en quoi ***l'approche par Tree Model est plus flexible*** que le Data Binding. Cette flexibilité est dû au fait que l'on reste sur un chargement complet en mémoire tout en ayant l'avantage d'être au plus près de la structure arborescente des données JSON.

➢ Tree Model API : Créer un arbre de données from Scracth et l'écrire dans un flux JSON

Constructing Tree Model instances from Scratch

It is also possible to construct tree instances without external JSON source.

For example:

Afficher/masquer les numéros de lignes

```
1  ObjectMapper mapper = new ObjectMapper();
2  JsonNode rootNode = mapper.createObjectNode(); // will be of type ObjectNode
3  ((ObjectNode) rootNode).put("name", "Tatu");
4  // ... and so forth
```

Extrait de http://wiki.fasterxml.com/JacksonTreeModel

La création d'un nouvel arbre de données est possible en créant directement des noeuds à partir de l'***ObjectMapper***. En effet, la classe ***ObjectMapper*** propose les méthodes **createObjectNode()** et **createArrayNode()** qui permettent de créer respectivement de nouveaux noeuds de type **objet** et de type **tableau**.

`createArrayNode()`
Note: return type is co-variant, as basic ObjectCodec abstraction can not refer to concrete node types (as it's part of core package, whereas impls are part of mapper package)

`createObjectNode()`
Note: return type is co-variant, as basic ObjectCodec abstraction can not refer to concrete node types (as it's part of core package, whereas impls are part of mapper package)

Pour illustrer ce qui précède, nous allons créer un nouvel arbre de données avec l'API Tree Model et nous écrirons cet arbre dans le fichier **createTreeModel.json**.
Le fichier JSON que nous souhaitons obtenir à l'exécution devra correspondre au fichier JSON proposé dans la première page de ce tutoriel et repris ci-contre.

```
{
  "name" : {
    "last" : "Blasquez",
    "first" : "Isabelle"
  },
  "gender" : "FEMALE",
  "verified" : false,
  "msgs" : [ "Message 1", "Message 2", "Message 3" ]
}
```

Voilà le code qui devrait être écrit pour ***créer un nouvel arbre de données JsonNode from Scratch et l'écrire dans le fichier*** **createTreeModel.json**

```
// ////////////////////////////////////
// Ecriture d'un Tree Model from Scratch
// ////////////////////////////////////

ObjectMapper mapper = new ObjectMapper();
ObjectNode rootNode = mapper.createObjectNode();
ObjectNode nameObj = rootNode.putObject("name");
nameObj.put("last", "Blasquez");
nameObj.put("first", "Isabelle");

rootNode.put("gender", "FEMALE");

rootNode.put("verified", Boolean.FALSE);

ArrayNode arrayNode = mapper.createArrayNode();
arrayNode.add("Message 1");
arrayNode.add("Message 2");
arrayNode.add("Message 3");
rootNode.put("msgs", arrayNode);

File dest = new File("createTreeModel.json");

try {
    ObjectWriter writer = mapper.writerWithDefaultPrettyPrinter();
    writer.writeValue(dest, rootNode);
} catch (JsonGenerationException e) {
    e.printStackTrace();
} catch (JsonMappingException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

→ Prenez le temps de comprendre ce code.

→ **Ce code est disponible sur la zone libre dans le fichier TestJacksonTreeModelUser**, importez ce fichier dans votre fichier et exécutez-le. Visualisez alors le fichier **createTreeModel.json**

III. La Streaming API

La **Streaming API**, dont le fonctionnement est proche de **SAX**, permet de lire et d'écrire un document **JSON** en s'appuyant sur un système évènementiel : c'est ***l'approche la plus performante*** en termes de temps d'exécution puisque le contenu JSON est ***lu et écrit de manière incrémentale***.

Le code à écrire pour la Streaming API sera plus fastidieux que celui pour le Data Binding, mais l'approche Streaming permettra un contrôle total sur tout ce qui est lu et écrit.
Au niveau performance, on obtient un temps d'exécution 20 à 30% plus rapide dans les cas les plus courants.
Enfin, toutes les opérations réalisées sont exécutées dans le flux courant, ce qui offre une consommation mémoire minimale.

L'API Streaming de Jackson est donc plus performante, rapide et pratique, mais elle est aussi un peu plus complexe à utiliser, car elle nécessite de gérer chaque détail des données JSON.

Attention, la Streaming API n'est pas basée sur l'*ObjectMapper* mais sur l'objet **JsonFactory**.
→ Pour ***écrire des données au format JSON*** un objet de type **JsonGenerator** devra être créé
→ Pour ***lire des données au format JSON*** (c-a-d parser un fichier JSON), un objet ***JsonParser*** devra être créé

Les classes **JsonFactory**, **JsonGenerator** , et **JsonParser** se trouvent dans le package **jackson-core** .
La documentation concernant le package **jackson-core** est consultable sur :
http://fasterxml.github.com/jackson-core/javadoc/2.1.0/

Sur le site officiel de Jackson, on peut trouver de la documentation sur la mise en place de la Streaming API sur :
http://wiki.fasterxml.com/JacksonStreamingApi et **http://wiki.fasterxml.com/JacksonInFiveMinutes**
et **https://github.com/FasterXML/jackson-core**

➢ Streaming API : Ecrire dans un flux au format JSON - JsonGenerator

Most common way to create generators is to pass an external destination (File, `OutputStream` or Writer) into which write resulting JSON content. For this purpose `org.codehaus.jackson.JsonFactory` has extensive set of methods to construct parsers, such as:

```
JsonFactory jsonFactory = new JsonFactory(); // or, for data binding, org.codehaus.jackson.mapper.MappingJsonFactory
JsonGenerator jg = jsonFactory.createJsonGenerator(file, JsonEncoding.UTF8); // or Stream, Reader
```

Extrait de : http://wiki.fasterxml.com/JacksonStreamingApi

Pour écrire un flux JSON avec l'API Streaming, il faut créer un objet de type **JsonGenerator** grâce à un appel de la méthode **createJsonGenerator** de la **JsonFactory**. On appliquera ensuite sur l'objet **JsonGenerator** des méthodes de type **writeXXX** , méthodes qui permettent d'écrire les données "brutes" dans le flux.
N'hésitez pas à consulter la documentation : http://fasterxml.github.com/jackson-core/javadoc/2.1.0/
dont un extrait est proposé ci-dessous.

abstract void	**writeStartArray**() Method for writing starting marker of a JSON Array value (character '['; plus possible white space decoration if pretty-printing is enabled).
abstract void	**writeStartObject**() Method for writing starting marker of a JSON Object value (character '{'; plus possible white space decoration if pretty-printing is enabled).
abstract void	**writeString**(char[] text, int offset, int len) Method for outputting a String value.
abstract void	**writeString**(SerializableString text) Method similar to writeString(String), but that takes SerializableString which can make this potentially more efficient to call as generator may be able to reuse quoted and/or encoded representation.
abstract void	**writeString**(String text) Method for outputting a String value.
void	**writeStringField**(String fieldName, String value) Convenience method for outputting a field entry ("member") that has a String value.
abstract void	**writeTree**(TreeNode rootNode) Method for writing given JSON tree (expressed as a tree where given JsonNode is the root) using this generator.
abstract void	**writeUTF8String**(byte[] text, int offset, int length) Method similar to writeString(String) but that takes as its input a UTF-8 encoded String which has **not** been escaped using whatever escaping scheme data format requires (for JSON that is backslash-escaping for control characters and double-quotes; for other formats something else).

Pour illustrer l'utilisation de la Streaming API, nous souhaitons écrire le fichier **userStreamingAPI.json** correspondant au flux JSON proposé dans la première page de ce tutoriel et repris ci-dessous.

→ Dans votre package **com.iut.tutoJackson.essai**, créer un fichier de test **TestJacksonStreamingAPI.** Dans votre **main**, écrire le code suivant (sans oublier les exceptions!) qui permet, à partir des remarques précédentes de procéder à l'écriture d'un flux **JSON** dans un fichier **userStreamingAPI.json en utilisant les méthodes write de la streaming API.**

```
JsonFactory f = new JsonFactory();
File dest = new File("userStreamAPI.json");

JsonGenerator jsonGenerator;
jsonGenerator = f.createJsonGenerator(dest, JsonEncoding.UTF8);
jsonGenerator.useDefaultPrettyPrinter();//retrouvez cette méthode dans la javadoc !

jsonGenerator.writeStartObject();
jsonGenerator.writeObjectFieldStart("name");
jsonGenerator.writeStringField("last", "Blasquez");
jsonGenerator.writeStringField("first", "Isabelle");
jsonGenerator.writeEndObject(); // for field 'name'
jsonGenerator.writeStringField("gender", "FEMALE");
jsonGenerator.writeBooleanField("verified", false);
jsonGenerator.writeEndObject();
jsonGenerator.close();// Ne pas oublier de fermer le generator
```

```
{
   "name":{
      "last":"Blasquez",
      "first":"Isabelle"
   },
   "gender":"FEMALE",
   "verified":false
}
```

→ Exécutez le code ci-dessus, vous devriez obtenir une copie d'écran similaire à la copie ci-contre:

```
{
  "name" : {
    "last" : "Blasquez",
    "first" : "Isabelle"
  },
  "gender" : "FEMALE",
  "verified" : false,
  "msgs" : [ "Message 1", "Message 2", "Message 3" ]
}
```

→ Complétez votre code afin de faire apparaître le flux JSON complet c-a-d avec l'objet **msgs** comme indiqué sur la copie d'écran ci-contre.

Remarque : L'écriture des données d'un utilisateur dans un flux JSON via la Streaming API est plus verbeuse que l'écriture d'un flux JSON avec le Data Binding. Elle reste cependant plus simple que la version SAX équivalente pour le XML.

➢ Streaming API : Lire un flux JSON - JsonParser

Parsers are objects used to tokenize JSON content into tokens and associated data. It is the lowest level of read access to JSON content.

Most common way to create parsers is from external sources (Files, HTTP request streams) or buffered data (Strings, byte arrays / buffers). For this purpose `org.codehaus.jackson.JsonFactory` has extensive set of methods to construct parsers, such as:

```
JsonFactory jsonFactory = new JsonFactory(); // or, for data binding, org.codehaus.jackson.mapper.MappingJsonFactory
JsonParser jp = jsonFactory.createJsonParser(file); // or URL, Stream, Reader, String, byte[]
```

Extrait de : http://wiki.fasterxml.com/JacksonStreamingApi

Pour lire un flux JSON avec la Streaming API, il faut créer un objet de type **JsonParser** grâce à un appel de la méthode **createJsonParser** de la **JsonFactory**.
Le fonctionnement du parser s'appuie sur un ***système de jeton*** (**token**).
En mode streaming, chaque JSON « string » est considéré comme un seul jeton, et chacun des jetons est traité de façon incrémentale. Le parser permet alors de parcourir le contenu du document **JSON** et de récupérer les informations souhaitées en s'appuyant sur le contenu du jeton courant.
A tout moment, la localisation au sein du fichier en cours d'analyse est possible grâce à la méthode **getCurrentLocation()**, ce qui offre réellement un contrôle total sur les données JSON.

Voilà le code qui devrait être écrit pour ***lire le flux de données JSON*** écrit précédemment dans le fichier **userStreamingAPI.json**

```
JsonFactory factory = new JsonFactory();
File src = new File("userStreamAPI.json");

try {
    JsonParser jsonParser = factory.createJsonParser(src);

    if (jsonParser.nextToken() == JsonToken.START_OBJECT) {

        // on boucle jusqu'à ce que le jeton soit égal à "}"
        while (jsonParser.nextToken() != JsonToken.END_OBJECT) {

            String fieldname = jsonParser.getCurrentName();
            jsonParser.nextToken();

            if ("name".equals(fieldname)) {

                while (jsonParser.nextToken() != JsonToken.END_OBJECT) {
                    String namefield = jsonParser.getCurrentName();
                    jsonParser.nextToken();

                    if ("first".equals(namefield)) {
                        System.out.println(jsonParser.getText()); // affichage
                    } else if ("last".equals(namefield)) {
                        System.out.println(jsonParser.getText()); // affichage
                    } else {
                        throw new IllegalStateException("Non reconnu '"
                                + fieldname + "'!");
                    }
                }


            } else if ("gender".equals(fieldname)) {
                System.out.println(jsonParser.getText()); // affichage

            } else if ("verified".equals(fieldname)) {
                if (jsonParser.getCurrentToken() == JsonToken.VALUE_TRUE)  // affichage
                    System.out.println(Boolean.TRUE);
                else
                    System.out.println(Boolean.FALSE);
```

```
                } else if ("msgs".equals(fieldname)) {

                    // le jeton courant est "[", on va avancer sur le jeton suivant grâce
                    // à la 1ere instruction dans le while jsonParser.nextToken

                    // les messages sont dans un array
                    // on boucle jusqu'à ce que le jeton soit égal à "]"
                    while (jsonParser.nextToken() != JsonToken.END_ARRAY) {

                        // display msg1, msg2, msg3
                        System.out.println(jsonParser.getText());
                    }
                } else {
                    throw new IllegalStateException("Non reconnu '"
                            + fieldname + "'!");
                }
            }
        }

        // Fermer le parser
        jsonParser.close();


    } catch (JsonGenerationException e) {
        e.printStackTrace();
    } catch (JsonMappingException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

→ Prenez le temps de comprendre ce code, puis

→ Importez la classe **TestJacksonStreamingAPIToken** disponible sur la zone libre est exécutez-là.
Vous devriez retrouvez le contenu du fichier JSON dans laconsole.

Remarque : Dans le code précédent, nous avons seulement procéder à l'affichage des données récupérées à l'aide d'un `System.out.println`, il serait bien sûr possible à partir de ces données de recréer un objet Java de type **Utilisateur** comme le montre l'exemple de la documentation de l'API Streaming de Jackson à l'adresse suivante : http://wiki.fasterxml.com/JacksonInFiveMinutes

IV. Application au Cabinet Medical

Dans cette dernière partie, nous vous demandons de mettre en place la persistance des données de l''application CabinetMedical dans un flux **JSON**. Nous travaillerons dans le cas simple où les patients n'ont pas d'ascendant.

Sources : Webographie / Bibliographie

http://www.json.org/ (en français: http://www.json.org/json-fr.html)
http://fr.wikipedia.org/wiki/JavaScript_Object_Notation
http://jsonformatter.curiousconcept.com/
https://www.projet-plume.org/ressource/format-json
http://blog.excilys.com/2010/02/25/android-pour-lentreprise-6-oubliez-gson-jackson-rocks-my-world/

JSON est décrit dans le RFC 4627 : D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), 2006 (http://www.ietf.org/rfc/rfc4627.txt)

https://github.com/FasterXML
http://wiki.fasterxml.com/JacksonHome
http://wiki.fasterxml.com/JacksonDownload
http://wiki.fasterxml.com/JacksonInFiveMinutes
http://wiki.fasterxml.com/JacksonDataBinding
http://wiki.fasterxml.com/JacksonTreeModel
http://fasterxml.github.com/jackson-databind/javadoc/2.1.0/
http://wiki.fasterxml.com/JacksonStreamingApi
http://fasterxml.github.com/jackson-core/javadoc/2.1.0/
https://github.com/FasterXML/jackson-databind
https://github.com/FasterXML/jackson-annotations/wiki/JacksonAnnotations
https://github.com/FasterXML/jackson-annotations
http://fasterxml.github.com/jackson-annotations/javadoc/2.1.0/
https://github.com/FasterXML/jackson-core
http://wiki.fasterxml.com/JacksonFAQDateHandling
http://jackson.codehaus.org/

http://www.tutos-android.com/parsing-json-jackson-android
http://sourcetutorial.com/?p=443
http://sourcetutorial.com/?p=439

"Jackson : la bibliothèque Java qui simplifie la manipulation du JSON" -
pages 66-69 -Magazine Programmez n°160 de Février 2013

Annexe I : Autres Bibliothèques pour manipuler un flux JSON

→ *Une série de billets sur le* **blog Java Trop Bien** *sur le thème* **:**

Quelle API pour convertir des objets Java en JSON ?

Les 5 API les plus connues pour convertir des objets Java en JSON (et inversement).
- json.org (http://www.json.org/java)
http://javatropbien.free.fr/index.php/2011/12/17/quelle-api-pour-convertir-des-objets-java-en-json-partie-1/

- Json-lib (http://json-lib.sourceforge.net)
http://javatropbien.free.fr/index.php/2011/12/22/quelle-api-pour-convertir-des-objets-java-en-json-partie-2/

- XStream (http://xstream.codehaus.org)
http://javatropbien.free.fr/index.php/2011/12/23/quelle-api-pour-convertir-des-objets-java-en-json-partie-3/

- Jackson (http://jackson.codehaus.org)
http://javatropbien.free.fr/index.php/2011/12/27/quelle-api-pour-convertir-des-objets-java-en-json-partie-4/

- Gson (http://code.google.com/p/google-gson)
http://javatropbien.free.fr/index.php/2011/12/28/quelle-api-pour-convertir-des-objets-java-en-json-partie-5/

→ *Sur le bog de* **Xebia : Comparatif des librairies JSON**

http://blog.xebia.fr/2010/08/18/comparatif-des-librairies-json/

→ *Tutoriel pour découvrir* ***GSon*** *:*
Concernant les projets Androit, une autre bibliothèque est également très utilisée : c'est la bibliothèque Gson**(http://code.google.com/p/google-gson)**
Gson est intéressant car maintenu par les équipe Google, mais Jackson est désormais adopté par de plus en plus de frameworks et Jackson est plus performant que Gson en terme de temps de désérialisation**,** sur Android.
Si vous voulez quand même en savoir plus sur Gson, vous pouvez consulter le tutoriel **"Accéder aux services Web via Android"** disponible sur le site **developpez.com** à l'adresse suivante :
http://acesyde.developpez.com/tutoriels/android/web-services-android/

Annexe II : Configuration du POM pour utiliser Jackson dans un projet Maven

Jackson se présente sous la forme d'une **bibliothèque open source autonome** découpée en plusieurs modules maven. Vous pouvez retrouver tous ces modules à l'adresse suivante : **https://github.com/FasterXML.**

A partir de cette page, lorsque l'on clique sur un module en particulier, on accède entre autres au **readme** du module qui donne des indications sur les dépendances à déclarer dans le POM du projet pour utiliser le module Maven, mais aussi quelques exemples simples d'utilisation du module en question, etc...

Pour la réalisation de ce tutoriel, nous avons besoin de configurer le POM de la manière suivante :

```
<dependencies>
   <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-core</artifactId>
      <version>2.1.2</version>
   </dependency>

   <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.1.2</version>
   </dependency>
</dependencies>
```

Comme Jackson est découpé en plusieurs modules Maven, il est donc configurable selon ses bsoins : une fois le coeur de l'API ajouté en tant que dépendance, les autres modules ne seront à ajouter que si nécessaire.