La classe CaJson permet de désérialiser des flux JSON (ou assimilables). Elle ne comporte qu'une propriété **@data** représentant l'objet résultant. Le constructeur **initialize** accepte un seul argument : **dta** qui prend **nil** pour valeur s'il n'est pas précisé lors de l'instanciation d'un nouvel objet. La propriété **@data** de l'objet prend alors également **nil** pour valeur. Lorsque l'argument **dta** est renseigné, il correspond au flux JSON à désérialiser exprimé sous la forme d'une chaine de caractéres. Il comporte alors soit l'expression JSON, soit le nom du fichier où celle-ci est stockée. La distinction entre un nom de fichier et une expression JSON s'effectue à partir du premier caractère de l'argument. Les expressions JSON acceptées débutent toujours par une accolade («{») ou un crochet ouvrant («[»). Les méthodes **read** et **readFile** appelées permettent respectivement de désérialiser l'objet **@data** à partir de l'argument ou du fichier désigné par celui-ci.

```
class CaJson
    attr_reader :data
    def initialize(dta=nil)
        case dta
        when nil
            @data=nil
        when /^[\{|\[]/
            @data=read(dta)
        else
            @data=readFile(dta)
        end
    end
```

La méthode **read** transmet le flux JSON reçu en argument à la méthode d'analyse **_read**. Elle renvoie son résultat à l'appelant (en l'occurrence **initialize** qui l'affecte à la propriété **@data**). La méthode **readFile** reçoit le nom du fichier à traiter en argument (**nme**). Elle ouvre le fichier en lecture seule et le lit ligne à ligne. A l'issue de la lecture, le contenu est stocké dans la variable **dta**, et transmis à la méthode **read** précédemment décrite.

```
    def read(dta)
        @data=_read(dta)
    end

    def readFile(nme)
        dta=""
        fle=File.new(nme,'r')
        while (lne=fle.gets)
```

```
            dta+=lne
        end
        fle.close
        read(dta)
    end
```

La méthode **_read** analyse l'expression JSON délivrée en argument (sous la forme d'une chaîne de caractères) et renvoie l'objet résultant de sa désérialisation. Elle n'effectue aucun contrôle syntaxique. Par conséquent, l'expression JSON soumise doit être correctement constituée. La méthode **_read** est récursive : dès qu'une séquence { … } ou […] est rencontrée lors du traitement, la méthode s'appelle elle-même pour en analyser le contenu. La variable **res** contient l'objet (initialement **nil**) résultant de l'analyse de l'expression JSON passée en argument (**dta**). Elle peut être une séquence { … } correspondant à un dictionnaire associatif (*hash*), une séquence […] décrivant un tableau ou une valeur élémentaire (booléen, entier ou chaîne de caractères). Lorsque l'argument ne débute pas par un caractère d'ouverture de séquence («{» ou «[») ou un guillemet «"», il correspond obligatoirement à une valeur élémentaire booléenne ou entière. La comparaison avec les expressions «true», «false» permet d'associer à l'objet résultant les valeurs booléennes **true** et **false**, à défaut l'expression est convertie en entier (**.to_i**).

```
def _read(dta)
        res=nil
        exp=nil
        hsh=nil
        if ! "\"{[".index(dta[0..0]) then
            if dta=='true' then
                res=true
            elsif dta=='false' then
                res=false
            else
                res=dta.to_i
            end
        elsif dta[0..0] == '"'
            res=dta[1..dta.length-2]
        else
```

Lorsque l'expression JSON à analyser débute par une accolade ouvrante («{»), elle correspond à la description d'un dictionnaire associatif (*hash*). La variable **res** est initialisée avec un dictionnaire vide **{}** (ne contenant aucune association). Le drapeau **hsh** est mis à vrai afin d'indiquer que l'expression JSON manipulée correspond à un dictionnaire associatif. L'expression à analyser **exp** se limite alors à la portion comprise entre l'accolade ouvrante

«{» et la dernière accolade fermante «}» de l'expression JSON soumise (**dta**). En Ruby, un dictionnaire associatif **dct**, associant respectivement les valeurs **1** et **2** à **a** et **b**, se déclare **dct={ 'a'=>1, 'b'=>2 }**. L'accès aux éléments s'effectue à partir de leurs clefs respectives, soit **dct["a"]** et **dct["b"]** qui renvoient **1** et **2**. Le dernier cas possible concerne la description d'un tableau. L'expression JSON commence alors par un crochet ouvert («[»). La variable **res** est initialisée avec un tableau vide **[]** (ne contenant aucun élément). Le drapeau **hsh** est mis à faux : l'expression manipulée ne correspond pas à un dictionnaire associatif (mais à un tableau). L'expression à analyser se limite alors à la portion comprise entre le crochet ouvrant «[» et le dernier crochet fermant «]» de l'expression JSON soumise (**dta**).

```
if dta[0..0] == '{' then
    res={}
    hsh=true
    exp=dta[1..dta.rindex('}')-1].strip
else
    res=[]
    hsh=false
    exp=dta[1..dta.rindex(']')-1].strip
end
```

Pour les dictionnaires associatifs et tableaux, l'analyse se poursuit par l'examen, caractère par caractère, du contenu de la variable **exp** (dotées des associations clef : valeur du dictionnaire ou des éléments du tableau). L'objectif est d'en extraire les paires clef : valeur ou les éléments. La position du premier caractère de l'entité à extraire est mémorisée dans **deb**. Les variables **pos** et **sep** indiquent respectivement la position du caractère en cours d'analyse et celle du séparateur («:») dans les paires clef : valeur (dictionnaire associatif). La drapeau **qte** précise si le contenu analysé est une valeur élémentaire de type chaîne de caractères (**true**) ou non (**false**). Les accolades et crochets peuvent être imbriqués. A chaque accolade ou crochet ouvrant doit correspondre une accolade ou crochet fermant. Le niveau d'imbrication est mémorisé dans **lvl**. Lorsque lvl prend 0 pour valeur, l'entité manipulée (clef : valeur) ou élément correspond à l'objet **res** d'où la limitation de prise en charge du séparateur («:»). La soumission de chaque caractère est effectuée par **exp.each_byte**. Il est important de noter qu'un caractère stocké sur plusieurs octets (unicode) sera transmis en plusieurs fois : octet par octet. Cet aléas n'est pas pris en charge car les caractères ayant une importance syntaxique (guillemets, virgule, accolades et crochet) sont codés sur un seul octet. En l'état les guillemets ne peuvent être utilisés au sein

d'une chaîne de caractère. Ultérieurement un caractère d'exception («\») sera introduit à cet effet. Les expressions «\"» et «\\» seront associées aux guillemets («") et à l'antislah («\»). Lorsque le séparateur de paires clef : valeur ou d'éléments est rencontré («,») selon la nature de l'entité analysée (dictionnaire associatif ou tableau) déterminée par la valeur de **hsh**, la clef (**sbk**) et la valeur (**sbv**) sont extraites et ajoutées à **res** : **res**[*clef*] = *valeur* ou **res** << *valeur*.

```
                qte=false
                deb=0
                pos=0
                sep=0
                lvl=0
                exp.each_byte { |car|
                    if qte
                       qte=(car!=34)
                    else
                        case car
                        when 34 # "
                            qte=true
                        when 44 # ,
                            if lvl==0 then
                                if hsh then
                                    sbk=exp[deb..sep-1].strip
                                    sbv=exp[sep+1..pos-1].strip

res[sbk[1..sbk.length-2]]=_read(sbv)
                                else
                                    sbv=exp[deb..pos-1].strip
                                    res << _read(sbv)
                                end
                                deb=pos+1
                            end
                        when 58 # :
                            if lvl==0
                                sep=pos
                            end
                        when 123,91 # { [
                            lvl+=1
                        when 93,125 # ] }
                            lvl-=1
                        end
                    end

                    pos+=1
                }
```

Au terme du traitement la dernière paire clef : valeur ou le dernier élement est finalement ajouté au dictionnaire associatif ou au tableau.

```
                if hsh then
                    sbk=exp[deb..sep-1].strip
                    sbv=exp[sep+1..exp.length-1].strip
                    res[sbk[1..sbk.length-2]]=_read(sbv)
                else
                    sbv=exp[deb..exp.length-1].strip
                    res << _read(sbv)
                end
            end
            return res
        end
    end
```

Les exemples ci-après illustrent l'emploi de la classe CaJSON.

Exemple 1 :

```
    dta=CaJson.new
    dta.read('{ "name1": "val,{ue1", "name2" : "value2",
"name3" : [1, 2, "value", { "n3.3.1" : "331", "n3.3.2":
true } ] }')
    print dta.data
```

Exemple 2 :

```
    dta=CaJson.new('{ "name1" : "value1" }');
    print dta.data
```

Exemple 3 :

Le fichier sur disque dwdata.txt contient : `{ "name1" : "value1" }` .

```
    dta=CaJson.new('dwdata.txt');
    print dta.data
```