Serializzazione

La persistenza dei dati

Da quando esistono i computer, i programmatori si sono scontrati con la necessità di far "vivere" i dati al di fuori dei programmi. Chiariamo il concetto con un esempio semplicissimo. Supponiamo di voler realizzare una rubrica telefonica in cui memorizzare nomi e indirizzi di amici e conoscenti. A seconda di quanto è abile il programmatore e sofisticato lo strumento, si possono fare applicazioni con un'elegante interfaccia utente e ricche di possibilità (magari capaci di comporre il numero di telefono e richiamare in continuazione se occupato). Tutte queste realizzazioni hanno un punto in comune: quando il programma non è attivo o quando il computer è spento i dati (numeri e nomi) devono restare memorizzati in modo permanente, questa necessità viene chiamata **persistenza**. Tradizionalmente, ogni programmatore decideva un formato per la rappresentazione dei dati persistenti che venivano scritti in uno o più file. Ad esempio, riferendoci alla rubrica di cui sopra, si poteva decidere di delimitare il nome con il carattere "$", oppure di riservargli un campo fisso di 30 byte o ancora di memorizzare prima il cognome e poi il nome, ecc...

La completa libertà sintattica del formato ha generato una babele di rappresentazioni, per cui un produttore di software diverso da quello originario non può fare uso dei dati altrui (a volte si tratta di migliaia di schede bibliografiche o di tutte le leggi di uno stato) se quello originario non dichiara esplicitamente e chiaramente il formato dei suoi dati. Un'altra complicazione è causata dal fatto che spesso i dati sono scritti in formati dipendenti dalla macchina, assolutamente illeggibili con programmi di uso generale o su computer di tipo diverso da quello originario.

Il formato XML

Il linguaggio SGML

SGML (**S**tandard **G**eneralized **M**arkup **L**anguage), è uno standard per la descrizione logica dei documenti. Discende dal Generalized Markup Language della IBM.

L'idea centrale dello standard è un tipo di marcatura generica chiamata "marcatura descrittiva" che definisce la struttura logica dei documenti.

L'organizzazione di un documento non è espressa usando la codifica dei sistemi di scrittura, che è finalizzata alla presentazione grafica, ma sono evidenziate le parti in cui è strutturato il documento (ad esempio paragrafi, capitoli) insieme ad altre particolarità del testo (come note, tabelle, intestazioni).

Il linguaggio HTML

HTML (**H**yper**T**ext **M**arkup **L**anguage) è un linguaggio di pubblico dominio basato su SGML la cui sintassi è stabilita dal World Wide Web Consortium (**W3C**). È stato sviluppato alla fine degli anni ottanta da Tim Berners-Lee al CERN di Ginevra. Verso il 1994 ha avuto una forte diffusione, in seguito ai primi utilizzi commerciali del web.

Nel corso degli anni, seguendo lo sviluppo di Internet, l'HTML ha subito molte revisioni, ampliamenti e miglioramenti, che sono stati indicati secondo la classica numerazione usata per descrivere le versioni dei software.

Il linguaggio XML

Il **progetto XML**, ebbe inizio alla fine degli anni ottanta nell'ambito della SGML Activity del W3C,

XML (*EXtensible Markup Language*) è un linguaggio di marcatura derivato dal SGML. Un documento XML è scritto in formato testo con *tag* anch'essi in formato testo, è quindi leggibile e modificabile con un comune text editor su ogni tipo di computer.

Molte sono le sue analogie con HTML, ma anche profonde e significative le differenze. Innanzitutto XML non è un linguaggio orientato solamente alla visualizzazione, ma un formato assolutamente generale per descrivere dati. La visualizzazione di un documento XML è solo una delle numerose possibilità. Poiché XML, come HTML, deriva da SGML, i tag sono scritti nello stesso modo usando le parentesi angolate

```
<NOMETAG>          </NOMETAG>
```

ma la sintassi è più rigida di quella dell'HTML: ogni tag aperto deve essere anche chiuso, non si possono aprire e chiudere tag in un ordine "sparso", vi è differenza tra maiuscole e minuscole. In altre parole, mentre la riga seguente è riconosciuta senza problemi dai browser HTML

```
<P><FONT size="+1"><tt> testo </font></TT>
```

per seguire la sintassi XML bisogna modificarla. As esempio, si può scrivere come segue

```
<P><FONT size="+1"><tt> testo </tt></FONT></P>
```

Con questi importanti vincoli si può dimostrare che un file XML ha una struttura ad albero.

Un'utile abbreviazione permette di sostituire un tag aperto e immediatamente chiuso

```
<BR></BR>
```

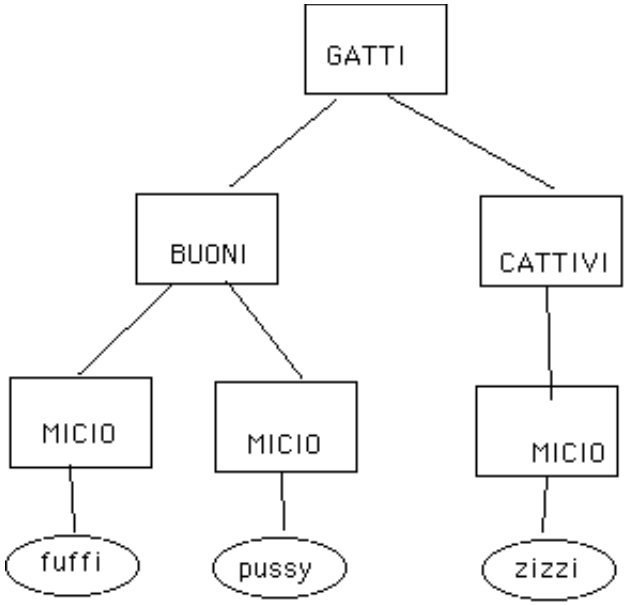
con un solo tag con la barra di chiusura in fondo

```
<BR/>
```

Ecco un primo esempio di file XML, sintatticamente corretto. Siccome i tag sono completamente liberi, ne abbiamo inventati alcuni per l'occasione.

```
<GATTI>
    <BUONI>
        <MICIO>fuffi</MICIO>
        <MICIO>pussy</MICIO>
    </BUONI>
    <CATTIVI>
        <MICIO>zizzi</MICIO>
    </CATTIVI>
</GATTI>
```

La struttura ad albero è mostrata in figura



Struttura ad albero della lista dei gatti

Una possibile interpretazione semantica è quella di un elenco di gatti buoni e cattivi; naturalmente questa è solo una deduzione psicologica indotta dai nomi. Da un punto di vista informatico il documento seguente:

```
<xyzz> <uihh> <z35>trtrt</z35> <z35>peytuut</z35>
</uihh> <hhuu> <z35>hfyryy</z35> </hhuu> </xyzz>
```

ha esattamente la **stessa** struttura, ma risulta certamente più difficile attribuirgli un significato semplicemente guardandolo.

In genere un tag XML può essere arricchito aggiungendovi degli attributi che consistono in coppie nome-valore. Anche in queste vi è completa libertà di scelta.

```
<MICIO RAZZA="soriano" PELO="lungo" VACCINATO="YES">
   fuffi
</MICIO>
```

La totale generalità della notazione XML permette di usarla per descrivere praticamente qualsiasi cosa, con il vantaggio sostanziale di una struttura sintattica univoca.

Parsing

La cosa più semplice che si può fare con un documento XML è leggerlo, farne l'analisi sintattica e utilizzarlo all'interno di un programma. Poiché le regole sintattiche dello XML sono sempre le stesse, il codice che effettua l'analisi sintattica (**parsing**) può essere scritto una volta per tutte. Nulla ovviamente impedirebbe ai programmatori di fare tutto da soli, ma le comunità XML (come **`www.xml.org`** e **`xml.apache.org`**) mettono a disposizione un insieme di strumenti già pronti.

Citiamo due pacchetti software che sono disponibili in Java, in C++ e in altri linguaggi.

- **SAX** (*Simple API for XML)*. Si tratta di un parser, che durante la lettura del file XML, ne individua i vari elementi e per ognuno di essi esegue le opportune azioni. Questo approccio si presta bene sia al trattamento di documenti molto semplici (come l'elenco dei gatti) sia per la elaborazioni di documenti molto grandi che, per la loro lunghezza, non potrebbero essere contenuti nella memoria centrale.
- **DOM** (*Document Object Module*), parser che legge l'intero documento XML e lo trasforma in un albero che risiede in memoria centrale. Questo approccio è adatto nel caso di documenti che richiedono un'elaborazione complessa (il fatto che l'albero risieda in memoria aumenta la flessibilità di programmazione), ma al tempo stesso di dimensioni tali da non creare problemi di spazio.

In Java il parser SAX è compreso nella API del linguaggio (**`javax.xml.`**... **`org.xml.sax.`**...).

Trasformazione di una lista di attributo-valore in una `HashMap`.

Esempio di dati (`file1.xml`)

```
<DATA>
<ITEM KEY="antonio"  VALUE="65774" />
<ITEM KEY="gerolamo" VALUE="57774" />
<ITEM KEY="matteo"   VALUE="78987" />
<ITEM KEY="pippo"    VALUE="45454" />
<ITEM KEY="zuzzurro" VALUE="48754" />
</DATA>
```

Ecco il codice di lettura `XMLTable` che usa il parser `SAX2` fornito dalle API di Java

```
import java.io.*;
import java.util.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class XMLTable extends DefaultHandler {
   private HashMap<String, String> table;
   private File file;

   public XMLTable(String path) {
      this.table = new HashMap<String, String>();
      this.file = new File(path);
      if (!file.exists())
         throw new RuntimeException(" file " + path + " not found");
   }

   public Map<String, String> read() {
      try {
         SAXParserFactory.newInstance().newSAXParser().parse(file, this);
      } catch (Throwable t) {
         t.printStackTrace();
      }
      return table;
   }
```

```
/*
 * il metodo seguente ridefinisce startElement dell'interfaccia DefaultHandler e
 * viene invocato dal parser XML ogni volta che si legge un elemento
 * (ITEM nel nostro caso) name è il nome dell'elemento atts la lista degli attributi
*/
 public void startElement(String d1, String d2, String name, Attributes atts)
       throws SAXException {
    if (name.equals("ITEM")) {
       if (atts.getLength() != 2)
          throw new SAXException("wrong number of attributes: "+ atts.getLength());
       String key = null, val = null;
       for (int i = 0; i < atts.getLength(); i++) {
          String aName = atts.getQName(i);
          if (aName.equals("KEY"))   key = atts.getValue(i);
          if (aName.equals("VALUE")) val = atts.getValue(i);
       }
       if (key == null) throw new SAXException("key undefined");
       if (val == null) throw new SAXException("value undefined");
       table.put(key, val);
    }
 }
}
```

La classe XMLBatch

Vediamo il codice di una semplice classe che implementa `Iterator` e legge coppie chiave-valore da un file `XML` restituendole una per volta sotto forma di `Map`

Ecco un esempio di dati in formato XML (`file2.xml`)

```
<INPUT>
<TEST N="10" PGM="0" />
<TEST N="10" PGM="1" />
<TEST N="100" PGM="3" />
<TEST N="1000" PGM="5" />
<TEST N="10000" PGM="6" />
<TEST N="100000" PGM="7" />
<TEST N="2000000" PGM="7" />
</INPUT>
```

Ogni item di tipo `TEST` specifica una prova da eseguire dando sia il valore di *N* che il programma da eseguire, questa tecnica che è assolutamente generale permette di automatizzare in modo molto flessibile l'esecuzione ripetuta di ogni tipo di codice.

Ed ecco il codice di `XMLBatch` che usa il parser `SAX2` fornito dalle API di Java.

La scelta effettuata è stata quella di creare una classe che implementa `Iterable` e rende l'iteratore del `Vector V` interno alla classe.

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.util.*;

public class XMLBatch extends DefaultHandler implements
       Iterable<Map<String, String>> {

    private Vector<Map<String, String>> V = new Vector<Map<String, String>>();
    private Map<String, String> H = new HashMap<String, String>();

    public XMLBatch(String filename) {
        File file = new File(filename);
        try {
            SAXParserFactory.newInstance().newSAXParser().parse(file, this);
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
```

```
/*
 * il metodo seguente ridefinisce startElement dell'interfaccia DefaultHandler e
 * viene invocato dal parser XML ogni volta che si legge un elemento
 * (ITEM nel nostro caso) name è il nome dell'elemento atts la lista degli attributi
 */
    public void startElement(String d1, String d2, String name, Attributes atts)
            throws SAXException {
        if (name.equals("TEST")) {
            for (int i = 0; i < atts.getLength(); i++)
                H.put(atts.getQName(i), atts.getValue(i));
            V.addElement(new HashMap<String, String>(H));
        }
    }

    public Iterator<Map<String, String>> iterator() {
        return V.iterator();
    }
}
```

Si noti che usando una `Map H` locale alla classe e mettendo nel vettore ogni volta una sua copia si lasciano invariate tutte le coppie chiave valore definite in precedenza nel file XML e non ridefinite nella riga corrente.

Organizzazione di un programma principale

Si noti che ogni test che usa come input una sequenza di coppie "chiave-valore" può essere automatizzato in questo modo senza modificare `XMLBatch` .

```
public static void main(String[] args) {
    data = new XMLBatch(filename);
    for(Map<String, String> h : data) testGenerico(h);
}
```

dove

```
void testGenerico(Map m)
```

è un metodo che effettua un'azione a partire dai dati contenuti in un elemento del file XML memorizzati in una classe che implementa `Map`.

Per chi si diverte a programmare a riga di comando (*sembra incredibile ma esistono ancora*) si può modificare il main per leggere il file di XML dalla lista degli argomenti.

```
public static void main(String[] args) {
       data = new XMLBatch(args[0]);
       for(Map<String, String> h : data) testGenerico(h);
}
```

Il programma di prova

Ecco un programma che prova le due classi.

```
import java.util.*;

public class testXML {

    public static void main(String[] args) {

        System.out.println(" test XMLTable");
        Map m = (new XMLTable("./file1.xml")).read();
        System.out.println("tabella letta");
        System.out.println(m);

        System.out.println();
        System.out.println(" test XMLBatch");

        XMLBatch data = new XMLBatch("./file2.xml");
        for(Map<String,String> H : data) testGenerico(H);
    }

    static void testGenerico(Map m) {
        System.out.println(" parametri: " + m);
    }

}
```

Ed ecco i risultati

```
test XMLTable
tabella letta
{zuzzurro=48754, pippo=45454, matteo=78987, gerolamo=57774, antonio=65774}

 test XMLBatch
 parametri: {N=10, PGM=0}
 parametri: {N=10, PGM=1}
 parametri: {N=100, PGM=3}
 parametri: {N=1000, PGM=5}
 parametri: {N=10000, PGM=6}
 parametri: {N=100000, PGM=7}
 parametri: {N=2000000, PGM=7}
```

Serializzazione Java

Java mette a disposizione un semplice modo per serializzare classi utilizzando i metodi

`void writeObject(Object o)` della classe `ObjectOutputStream`

`Object readObject()` della classe `ObjectInputStream`

Vediamo un semplice esempio.

```
import java.io.Serializable;
public class SimpleClass implements Serializable {

    private static final long serialVersionUID = 1L;

    Integer i;

    public SimpleClass() {this.i = 10;}

    public String toString() {return "il valore interno e' "+i;}

    public void change() {i *= 2;}

}
```

```java
import java.io.*;

public class SerMain {

    public static void main(String[] args) {
        SimpleClass sc = new SimpleClass();
        System.out.println(sc);

        try {
            ObjectOutputStream objectOut = new ObjectOutputStream(
                new FileOutputStream(new File("object")));
            objectOut.writeObject(sc);
            objectOut.close();
        } catch (IOException e) {e.printStackTrace();}

        sc.change();
        System.out.println(sc);

        try {
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream( new File( "object" ) ) );
            sc = (SimpleClass) ois.readObject();
            ois.close();
        } catch (Exception e) {e.printStackTrace();}

        System.out.println(sc);

    }
}
```

Ed ecco i risultati

```
il valore interno e' 10
il valore interno e' 20
il valore interno e' 10
```

Vantaggi e svantaggi della serializzazione Java

- È facile serializzare oggetti complicati

- Non vi è compatibilità con gli altri linguaggi
- Problema del `serialVersionUID`

Sono possibili personalizzazioni del formato di serializzazione (sottoclasse `Externalizable`)

Un'alternativa per la serializzazione: JSON

Note tratte da https://www.json.org/json-it.html

JSON (JavaScript Object Notation) è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. Si basa su un sottoinsieme del Linguaggio di Programmazione JavaScript.

JSON è un formato di testo completamente indipendente dal linguaggio di programmazione, ma utilizza convenzioni conosciute dai programmatori di linguaggi della famiglia del C, come C, C++, C#, Java, JavaScript, Perl, Python, e molti altri. Questa caratteristica fa di JSON un linguaggio ideale per lo scambio di dati.

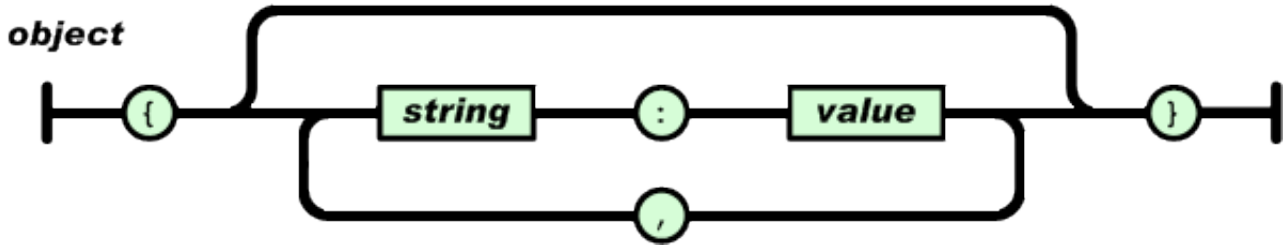
JSON è basato su due strutture:

- Un insieme di coppie nome/valore. In diversi linguaggi, questo è realizzato come un oggetto, un record, uno struct, un dizionario, una tabella hash, un elenco di chiavi o un array associativo.
- Un elenco ordinato di valori. Nella maggior parte dei linguaggi questo si realizza con un array, un vettore, un elenco o una sequenza.

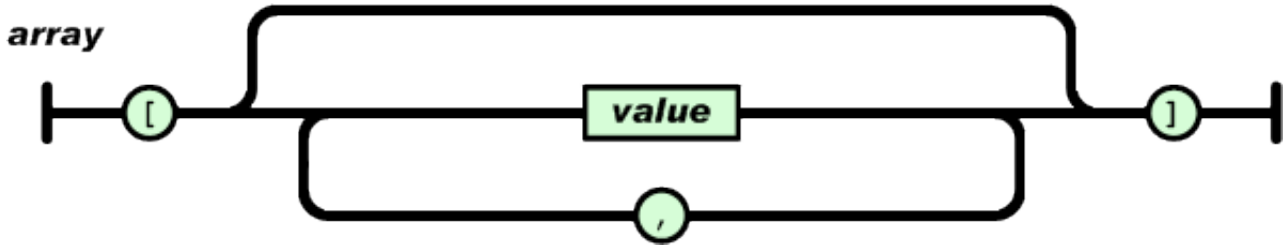
Queste sono strutture di dati universali. Virtualmente tutti i linguaggi di programmazione moderni li supportano in entrambe le forme. E' sensato che un formato di dati che è interscambiabile con linguaggi di programmazione debba essere basato su queste strutture.

In JSON, assumono queste forme:

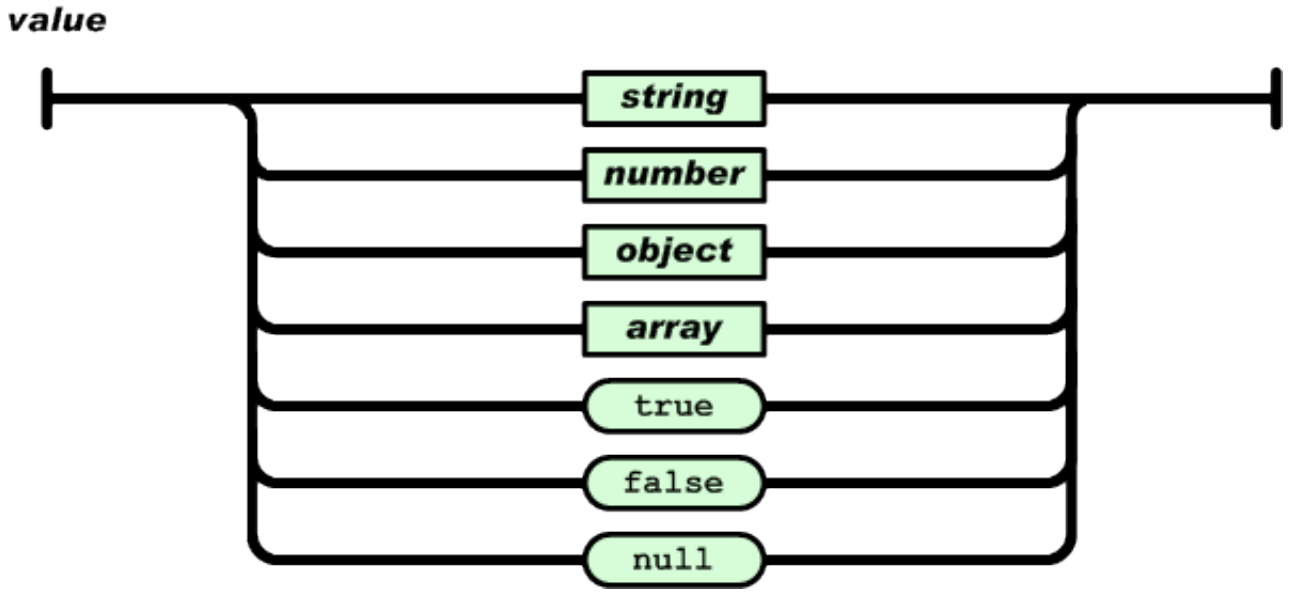
Un oggetto è una serie non ordinata di nomi/valori. Un oggetto inizia con parentesi graffa sinistra e finisce con *parentesi graffa destra*. Ogni nome è seguito da (due punti) e la coppia di nome/valore sono separata da *virgola*.



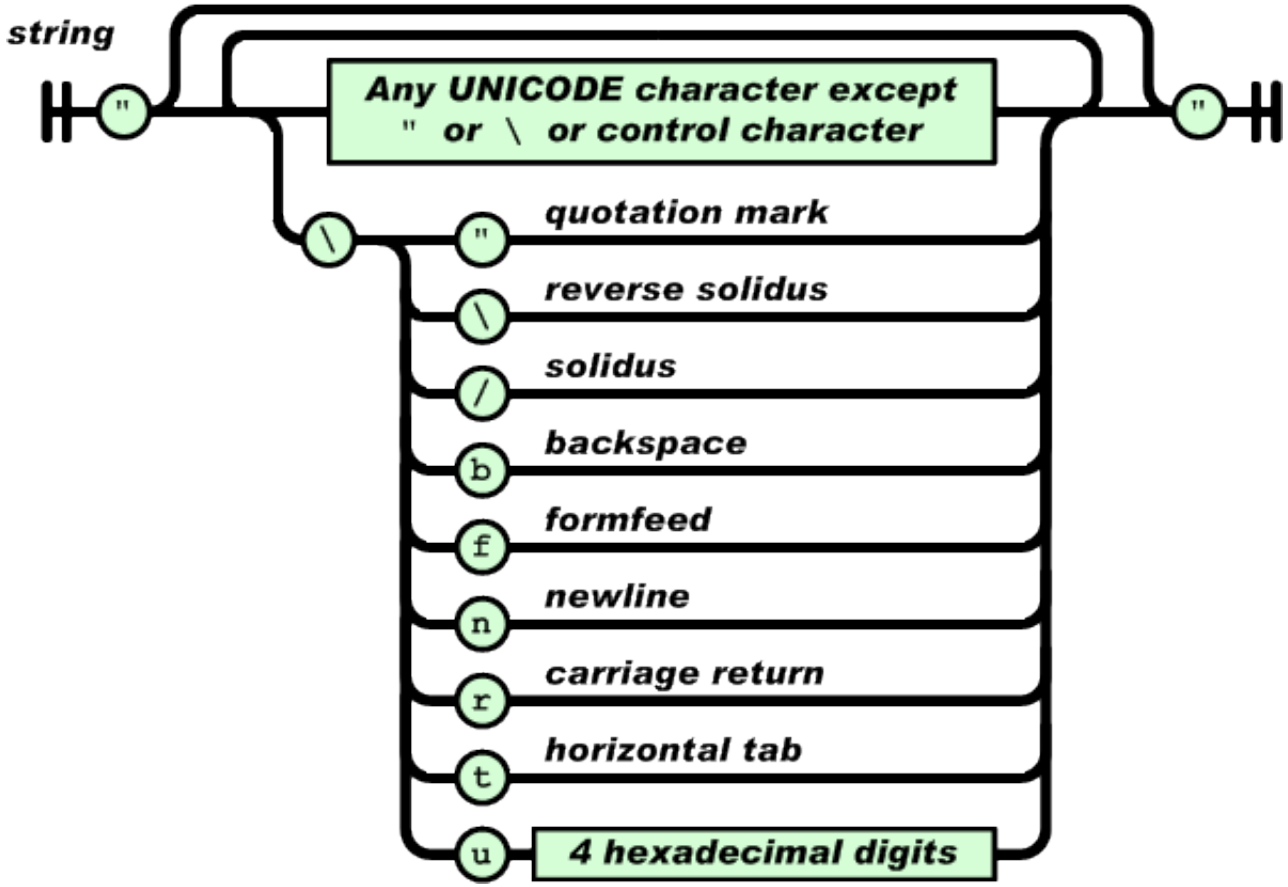
Un array è una raccolta ordinata di valori. Un array comincia con *parentesi quadra sinistra* e finisce con *parentesi quadra destra*. I valori sono separati da *virgola*.



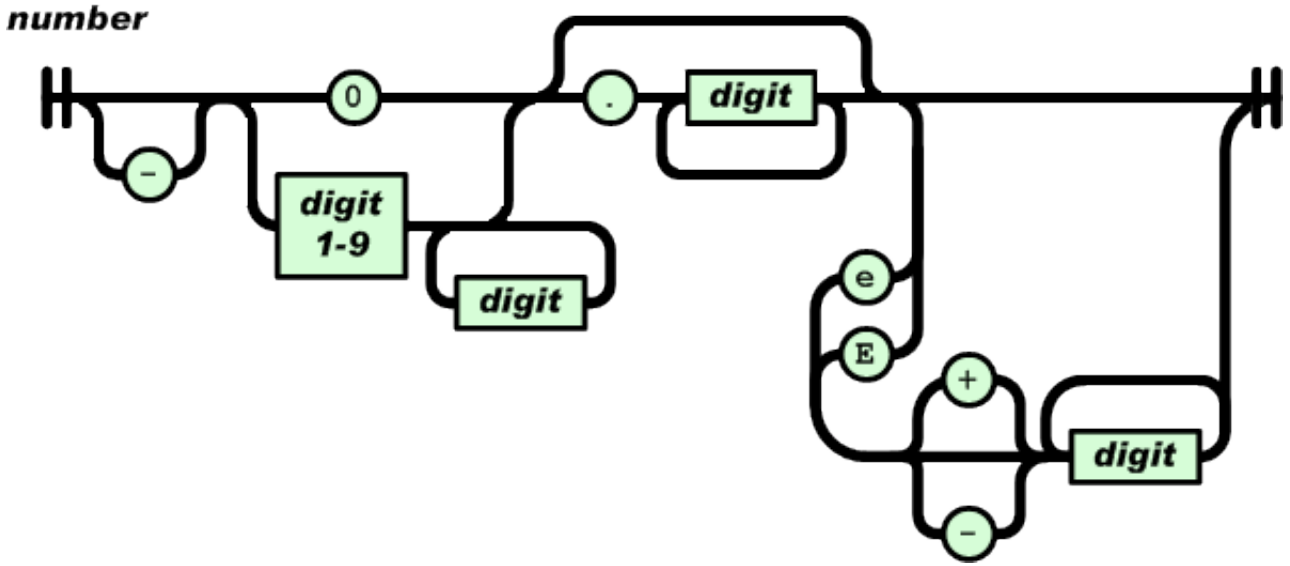
Un valore può essere una stringa tra virgolette, o un numero, o vero o falso o nullo, o un oggetto o un array. Queste strutture possono essere annidate.



Una stringa è una raccolta di zero o più caratteri Unicode, tra virgolette; per le sequenze di escape utilizza la barra rovesciata. Un singolo carattere è rappresentato come una stringa di caratteri di lunghezza uno. Una stringa è molto simile a una stringa C o Java.



Un numero è molto simile a un numero C o Java, a parte il fatto che i formati ottali e esadecimali non sono utilizzati.



I caratteri di spaziatura possono essere inseriti in mezzo a qualsiasi coppia di token.

A parte alcuni dettagli di codifica, questo descrive totalmente il linguaggio.

ESEMPI di file JSON

Un array di caratteri


```
["a", "\u00e0", "b", "c", "d", "e", "\u00e8", "\u00e9", "f", "g",
"h", "i", "\u00ec", "\u00ee", "j", "l", "m", "n", "o", "\u00f2",
"p", "q", "r", "s", "t", "u", "\u00f9", "v", "z"]
```


Una tabella


```
{"rs": 0.00219465061543244 9, "v\u00e0": 3.421460183405392e-07,
"nm": 3.043298794713217 2e-06, "ra": 0.013740061873685957, "et":
0.005644923095119 15, "sn": 1.4928371010753002e-05, "ne":
0.013629152540688095}
```