



PROGRAMMA **ARNALDO**

Parsing

A.A 2022/23

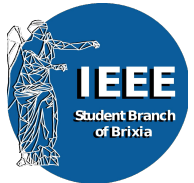
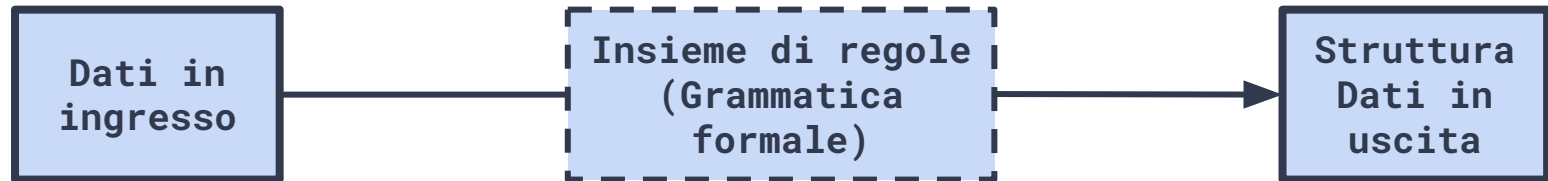


Cos'è il parsing?



Il **parsing** è un processo che analizza una stringa di simboli facente parte di un dato linguaggio e conforme alle regole di una grammatica formale.

In informatica, il **parsing** è un processo che analizza un flusso di dati in ingresso e costruisce una struttura dati utilizzando un insieme di regole fornite una grammatica formale, dotando i dati in ingresso di una rappresentazione strutturale e controllando la correttezza sintattica.



A cosa serve il parsing?

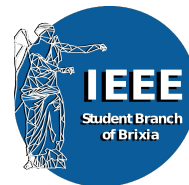


La **necessità** del parsing nasce dal bisogno di **estrapolare** informazioni a partire da un certo dato: in mancanza di questo passaggio, tale dato rimane un'espressione incomprensibile e inutilizzabile

Un esempio: ജിഇനാസ

Con buona probabilità, questa stringa vi risulterà assolutamente incomprensibile: non conoscendo la lingua, il dato è inutilizzabile e da esso risulta impossibile estrarre l'informazione

Analogamente, il **significato** di qualunque stringa rimane oscuro ad una macchina, per quanto possa sembrare ovvio ad un essere umano, a meno che non vengano fornite con essa le informazioni necessarie al suo parsing



Esempio 3.1.1 [Numero di telefono]

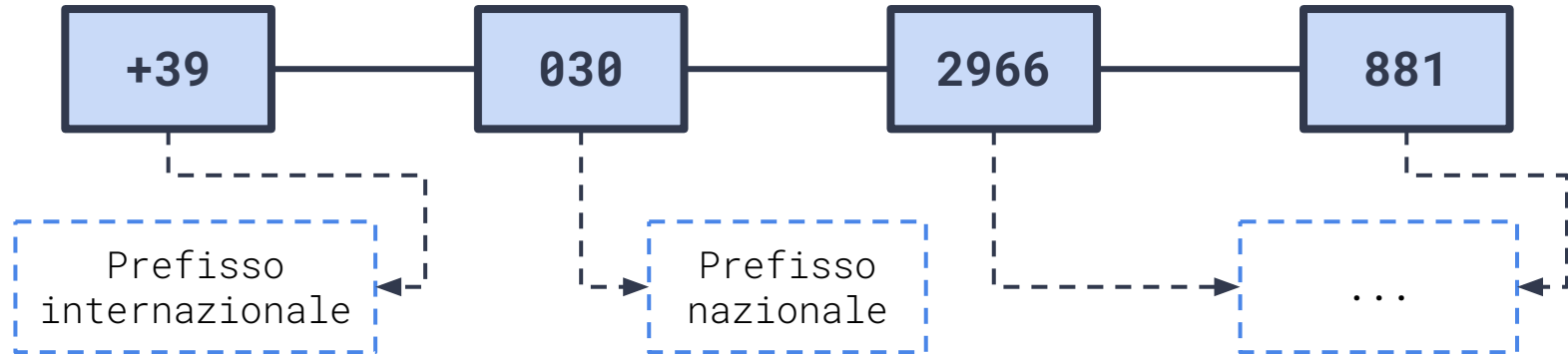


Dato d'ingresso: +39 0302966881

Regole:

- Gli unici caratteri ammessi sono cifre e il segno “+”
- La regola generale può essere riassunta dalla stringa **+xx xxx xxxx xxx**

Informazione ricavata:



Esempio 3.1.2 [scanf]



Un esempio di parsing molto basilico nel linguaggio C è dato dalla funzione **scanf**
Essa permette infatti di leggere input da tastiera:

```
int y, m, d;  
scanf("%d-%d-%d",&y, &m, &d);
```

La regola seguita dalla scanf è la stringa
"YYYY-MM-DD", rappresentante una data in formato
ISO 8601

Queste istruzioni **non** effettuano però nessun
controllo sui valori dei dati inseriti!



Esempio 3.1.3 [compilatore]

Un impiego molto importante del parsing è nella **compilazione** del codice C.

Esso è necessario nella conversione di codice di alto livello in codice di basso livello (linguaggio macchina, eseguibile dall'hardware)

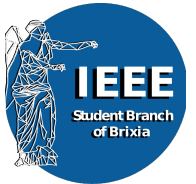
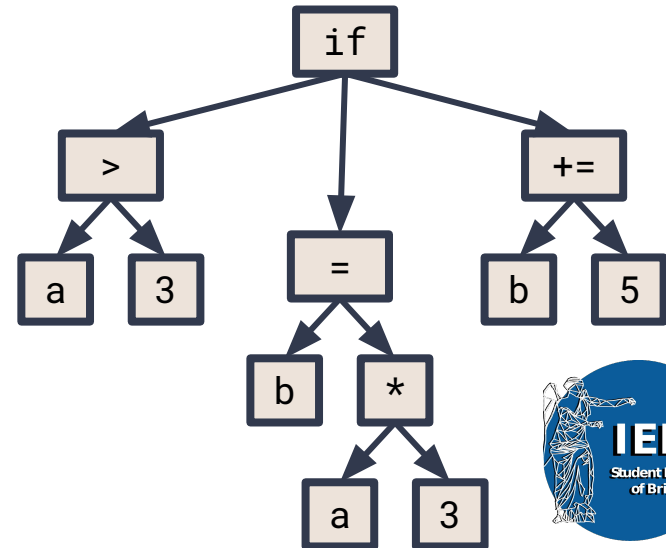
```
if (a > 3) {  
    b = a * 3;  
}  
else {  
    b += 5;  
}
```



I compiti del parsing sono principalmente 2:

- effettuare l'analisi sintattica
- costruire una struttura dati funzionale alla conversione in linguaggio macchina

Questa struttura dati (un albero sintattico), permette al compilatore di "comprendere" il significato del codice scritto dal programmatore

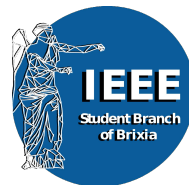


Esempio 3.1.4 *[parser HTML]*

HTML è il linguaggio di markup standard per la creazione di pagine web.

In generale il web browser richiede un documento HTML a un server, e procede a rappresentarlo a seguito di una necessaria operazione di parsing

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```



Esercizio 3.1.1: [Codice Fiscale Beta]



Problema: Dato un codice fiscale, verificare la sua validità

Input: Un codice fiscale (es: FSTPLA98M01B157E)

Output:

- True se il codice fiscale inserito è valido
- False altrimenti

Vincoli: Nessuno (l'input può essere una qualunque stringa)

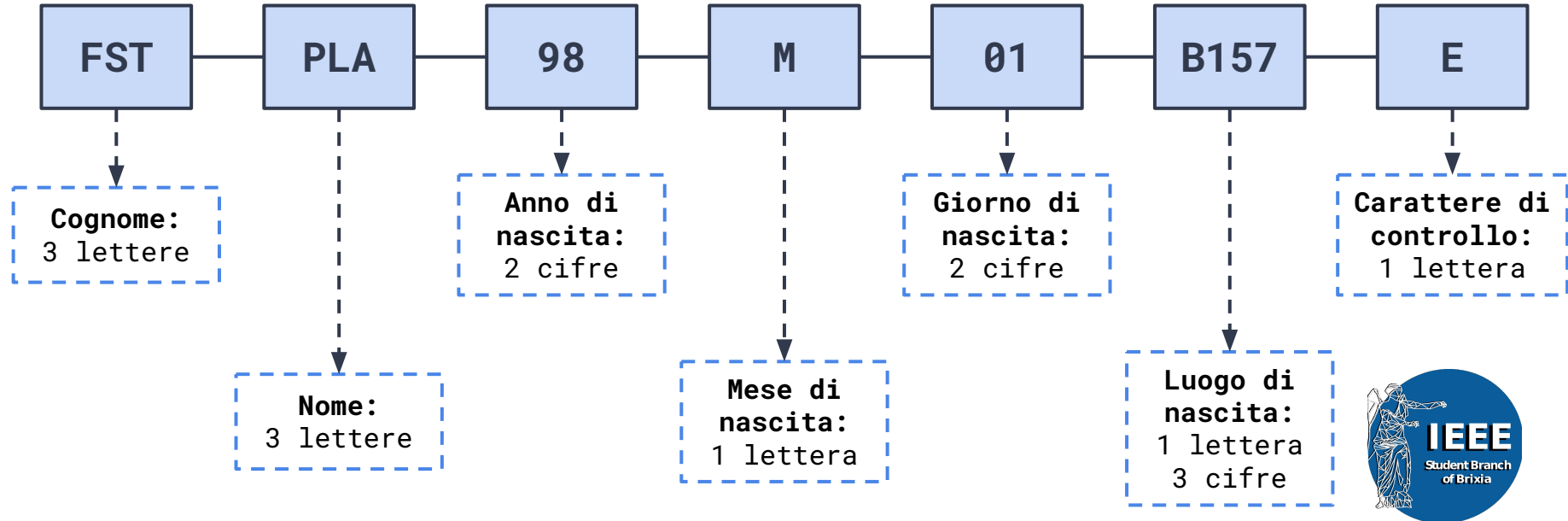
Tempo: 20 minuti

Note: per la struttura del codice fiscale e i controlli da effettuare, vedasi la prossima slide.



Esercizio 3.1.1: [Codice Fiscale Beta]

Struttura del codice fiscale: 16 caratteri divisi in 7 gruppi



Esercizio 3.1.1: [Codice Fiscale Beta]



Controlli principali:

Caratteri e cifre nelle posizioni corrette (FSTPLA98M01BK57E è un esempio di codice errato).

Controlli aggiuntivi:

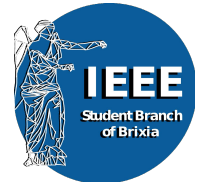
Validità del giorno (valore compreso fra 1 e 31 o fra 41 e 71).

Correttezza della lettera del mese (valori ammessi: A, B, C, D, E, H, L, M, P, R, S, T).

Controlli extra [complicati]:

Numero di giorni in un mese (B30 non è ammesso, il 30 febbraio non esiste). [complicato]

Validità di nome e cognome.



```
<?xml version="1.0" encoding="UTF-8"?>
<programmaArnaldo>
  <!-- LISTA LEZIONI -->
  <lezione id='0'>
    <nome>Prerequisiti</nome>
    <insegnante>Roberto</insegnante>
  </lezione>
  <lezione id='1'>
    <nome>JFC e liste</nome>
    <insegnante>Jacopo</insegnante>
  </lezione>
  <lezione id='2'>
    <nome>Debugger</nome>
    <insegnante>Stefano</insegnante>
  </lezione>
  <lezione id='3'>
    <nome>Git e GitHub</nome>
    <insegnante>Enrico</insegnante>
  </lezione>
</programmaArnaldo>
```

XML

Introduzione e panoramica



Cos'è XML?



XML (**eXtensible Markup Language**) è un linguaggio di **markup**, pensato per essere leggibile sia da umani sia da macchine.

La sua utilità risiede nella possibilità di descrizione dell'informazione e nell'organizzazione dei dati per mezzo di una struttura ordinata e (talvolta) autoesplicativa.



XML fornisce la base per il formato dei file dei principali office suite (Microsoft Office, LibreOffice, iWork)



I file di configurazione di molte applicazioni utilizzano XML.
Un esempio ne è Eclipse (file nascosti `.classpath` e `.project` in ogni progetto)



Nella programmazione Android, file XML sono utilizzati per svariati motivi: UI e varie personalizzazioni

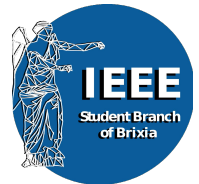
Struttura file XML [XML declaration]

In testa al file XML si trova la **XML declaration** (o prologo), che specifica alcune informazioni riguardo al documento stesso, in questo caso la versione XML utilizzata e il character-encoding:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Il costrutto base dell'XML è il **tag**. Esso inizia con "<" e termina con ">" e può essere di 3 tipi:

- Tag d'apertura (start tag): <nome>
- Tag di chiusura (end tag): </nome>
- Tag vuoto(empty/element tag): <empty/>



Struttura file XML [Attributi]



I tag d'apertura e i tag vuoti possono contenere **uno o più** attributi:

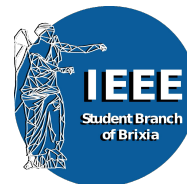
```
<autore nome='Roberto' />  
<tag id='0'>Testo</tag>
```

Un **attributo** è un costrutto consistente di una coppia nome-valore, che fornisce ulteriori dati riguardo a ciò che è specificato nel tag. Nell'esempio precedente:

- "nome" → nome, "Roberto" → valore.
- "id" → nome, "0" → valore dell'attributo

Non c'è nessun limite al numero degli attributi, a patto che abbiano tutti nomi diversi:

```
<rect fill="red" x="20" y="20" width="100" height="100" stroke="blue" />
```



Struttura file XML [Commenti]



Un **commento** è un particolare tipo di elemento che viene ignorato durante la lettura del documento XML:

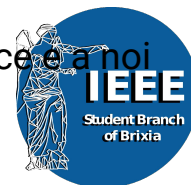
```
<!-- COMMENTO -->  
<!-- COMMENTO  
    SU PIÙ  
    LINEE -->
```

Un commento inizia con la stringa <!--, termina con --> e può occupare una o più righe

Il testo nei commenti non contiene nessun dato utile alla macchina: durante il parsing essi vengono saltati e si passa immediatamente all'elemento successivo.

L'utilità dei commenti è esclusivamente di **assistere il programmatore** a districarsi in file che possono diventare molto lunghi e complessi.

Nota più ampia sui commenti: **SONO FONDAMENTALI!** Servono a voi se dovete tornare sul codice e a noi per correggere più comodamente



Struttura file XML [Documento ben formattato]



```
<?xml version="1.0" encoding="UTF-8"?>
<programmaArnaldo>
  <!-- LISTA LEZIONI -->
  <lezione id='0'>
    <nome>Prerequisiti</nome>
    <insegnante>Roberto</insegnante>
  </lezione>
  <lezione id='1'>
    <nome>JFC e liste</nome>
    <insegnante>Jacopo</insegnante>
  </lezione>
  <lezione id='2'>
    <nome>Debugger</nome>
    <insegnante>Stefano</insegnante>
  </lezione>
  <lezione id='3'>
    <nome>Git e GitHub</nome>
    <insegnante>Enrico</insegnante>
  </lezione>
</programmaArnaldo>
```

Un documento XML deve rispettare una struttura precisa per essere considerato **ben formattato** (*well-formed*).

Il file deve contenere in testa la XML **declaration**

A seguire, l'intero documento deve essere contenuto in un elemento, detto **radice** (o root)

Tutti i tag contenuti nel documento devono essere bilanciati, cioè i tag d'apertura hanno un tag di chiusura corrispondente ed è rispettato l'ordine di annidamento.

L'**ordine di annidamento** è rispettato se ogni tag di chiusura si trova immediatamente dopo il tag d'apertura corrispondente oppure se fra di essi si trovano uno o più elementi che rispettino l'ordine di annidamento



Librerie XML in Java

Lettura e scrittura di file XML



Struttura file XML [Elementi]



Un **elemento** è il componente logico fondamentale del file XML.

```
<lezione id="0">  
  <nome>Prerequisiti</nome>  
  <insegnante>Roberto</insegnante>  
</lezione>
```

Un elemento inizia con un **tag d'apertura** e termina con il **tag di chiusura** corrispondente (avente lo stesso nome). In alternativa, un elemento può consistere solamente di un **tag vuoto**.

Un elemento può contenere del **testo** (o content), contenuto fra tag d'apertura e di chiusura, e/o uno o più elementi, detti **elementi figli**.

Nell'esempio, l'elemento lezione contiene due elementi figli, in ordine nome e insegnante (quest'ultimo ha content pari a Roberto)



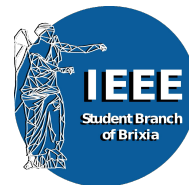
Lettura di file XML [inizializzazione]



```
XMLInputFactory xmlif = null;
XMLStreamReader xmlr = null;
try {
    xmlif = XMLInputFactory.newInstance();
    xmlr = xmlif.createXMLStreamReader(filename, new FileInputStream(filename));
} catch (Exception e) {
    System.out.println("Errore nell'inizializzazione del reader:");
    System.out.println(e.getMessage());
}
```

Questo frammento di codice serve a creare ed istanziare la variabile `xmlr` di tipo `XMLStreamReader`, che sarà utilizzata per leggere il file XML.

Si noti il costrutto try-catch, necessario ad “afferrare” e processare eccezioni lanciate da codice nel blocco try ed evitare che risultino come scritte rosse in console.



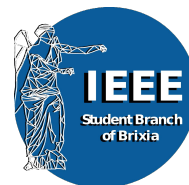
Metodi di XMLStreamReader [eventi]



Questi metodi fondamentali di **XMLStreamReader** permettono di navigare all'interno dell'operazione di parsing, passando all'evento successivo di volta in volta

- **hasNext():boolean**
Restituisce true se ci sono eventi di parsing disponibili
- **next():int**
Passa all'evento successivo e restituisce un intero identificante il tipo di evento
- **getEventType():int**
Restituisce un intero identificante il tipo di evento

I metodi `next()` e `getEventType()` restituiscono valori interi: che cosa rappresentano?
La chiave per interpretare questi valori è nella classe **XMLStreamConstants**



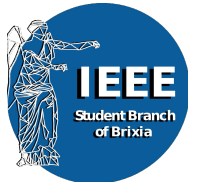
La classe XMLStreamConstants



La classe **XMLStreamConstants** fornisce alcune costanti rappresentanti il tipo di evento di parsing, pensate per l'utilizzo congiunto con **XMLStreamReader** :

- **START_DOCUMENT**
Rappresenta l'inizio del documento
- **START_ELEMENT**
Rappresenta la lettura di un *tag d'apertura*
- **END_ELEMENT**
Rappresenta la lettura di un *tag di chiusura*
- **COMMENT**
Rappresenta la lettura di un commento
- **CHARACTERS**
Rappresenta la lettura di testo all'interno di un elemento

Attenzione: un tag vuoto (<tag/>) viene interpretato come due eventi distinti: **START_ELEMENT** e **END_ELEMENT**.

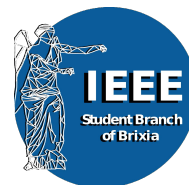


Metodi di XMLStreamReader [singolo evento]



Questi metodi di **XMLStreamReader** possono essere chiamati solo a seguito di specifici eventi (specificati tra parentesi quadre), altrimenti viene lanciata un'eccezione.

- **getLocalName():String**
Restituisce il nome del tag d'apertura o di chiusura [START_ELEMENT, END_ELEMENT]
- **getText():String**
Restituisce il valore del testo del commento o il *content* dell'elemento [CHARACTERS, COMMENT]
- **getAttributeCount():int**
Restituisce il numero di attributi per il tag d'apertura letto [START_ELEMENT]
- **getAttributeLocalName(index:int):String**
Restituisce il nome dell'attributo in posizione index [START_ELEMENT]
- **getAttributeValue(index:int):String**
Restituisce il valore dell'attributo in posizione index [START_ELEMENT]



Lettura di file XML [ciclo di parsing]



```
while (xmlr.hasNext()) {           // continua a leggere finché ha eventi a disposizione
    switch (xmlr.getEventType()) {   // switch sul tipo di evento
        case XMLStreamConstants.START_DOCUMENT: // inizio del documento: stampa che inizia il documento
            System.out.println("Start Read Doc " + filename); break;
        case XMLStreamConstants.START_ELEMENT: // inizio di un elemento: stampa il nome del tag e i suoi attributi
            System.out.println("Tag " + xmlr.getLocalName());
            for (int i = 0; i < xmlr.getAttributeCount(); i++)
                System.out.printf(" => attributo %s->%s%n", xmlr.getAttributeLocalName(i), xmlr.getAttributeValue(i));
            break;
        case XMLStreamConstants.END_ELEMENT: // fine di un elemento: stampa il nome del tag chiuso
            System.out.println("END-Tag " + xmlr.getLocalName()); break;
        case XMLStreamConstants.COMMENT:
            System.out.println("// commento " + xmlr.getText()); break; // commento: ne stampa il contenuto
        case XMLStreamConstants.CHARACTERS: // content all'interno di un elemento: stampa il testo
            if (xmlr.getText().trim().length() > 0) // controlla se il testo non contiene solo spazi
                System.out.println("-> " + xmlr.getText());
            break;
    }
    xmlr.next();
}
```

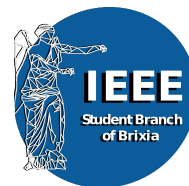
Scrittura di file XML [inizializzazione]



```
XMLOutputFactory xmlof = null;
XMLStreamWriter xmlw = null;
try {
    xmlof = XMLOutputFactory.newInstance();
    xmlw = xmlof.createXMLStreamWriter(new FileOutputStream(filename), "utf-8");
    xmlw.writeStartDocument("utf-8", "1.0");
} catch (Exception e) {
    System.out.println("Errore nell'inizializzazione del writer:");
    System.out.println(e.getMessage());
}
```

Questo frammento di codice serve a creare ed istanziare la variabile `xmlw` di tipo `XMLStreamWriter`, che sarà utilizzata per scrivere il file XML. Viene inoltre inizializzato il documento XML.

Si noti il costrutto try-catch, necessario ad “afferrare” e processare eccezioni lanciate da codice nel blocco try ed evitare che risultino come scritte rosse in console.

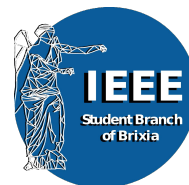


Metodi di XMLStreamWriter [gestione]



Questi metodi forniti dalla classe **XMLStreamWriter** sono necessari alla gestione della scrittura (inizio scrittura, chiusura del file, ecc...)

- **writeStartDocument(encoding:String, version:String):void**
Scrive l'inizio del documento e la XML declaration
- **writeEndDocument():void**
Termina la scrittura del documento, chiudendo anche tag, se necessario
- **flush():void**
Permette di scrivere su file i dati ancora in attesa nel buffer
- **close():void**
Chiude il writer e libera tutte le risorse ad esso associate
Questo metodo **DEVE** essere chiamato al termine della scrittura

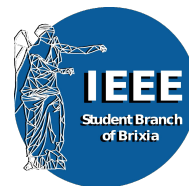


Metodi di XMLStreamWriter [scrittura]



Questi metodi forniti dalla classe `XMLStreamWriter` permettono la scrittura vera e propria

- `writeStartElement(localName:String):void`
Scrive un tag d'apertura con un dato nome
- `writeEndElement():void`
Scrive il tag di chiusura dell'ultimo tag ancora aperto
- `writeAttribute(localName:String, value:String):void`
Scrive un attributo con la coppia nome-valore fornita
[da chiamare immediatamente dopo `writeStartElement(...)`]
- `writeCharacters(text:String):void`
Scrive del testo dato come content all'interno dell'elemento attualmente aperto
- `writeComment(data:String):void`
Scrive un commento, con testo fornito



Scrittura di file XML



```
String[] autori = {"Roberto", "Jacopo", "Enrico", "Stefano"}; // esempio di dati da scrivere
try { // blocco try per raccogliere eccezioni
    xmlw.writeStartElement("programmaArnaldo"); // scrittura del tag radice <programmaArnaldo>
    xmlw.writeComment("INIZIO LISTA"); // scrittura di un commento
    for (int i = 0; i < autori.length; i++) {
        xmlw.writeStartElement("autore"); // scrittura del tag autore...
        xmlw.writeAttribute("id", Integer.toString(i)); // ...con attributo id...
        xmlw.writeCharacters(autori[i]); // ...e content dato
        xmlw.writeEndElement(); // chiusura di </autore>
    }
    xmlw.writeEndElement(); // chiusura di </programmaArnaldo>
    xmlw.writeEndDocument(); // scrittura della fine del documento
    xmlw.flush(); // svuota il buffer e procede alla scrittura
    xmlw.close(); // chiusura del documento e delle risorse impiegate
} catch (Exception e) { // se c'è un errore viene eseguita questa parte
    System.out.println("Errore nella scrittura");
}
```

output.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<programmaArnaldo>
  <!-- INIZIO LISTA -->
  <autore id="0">Roberto</autore>
  <autore id="1">Jacopo</autore>
  <autore id="2">Enrico</autore>
  <autore id="3">Stefano</autore>
</programmaArnaldo>
```

```
{
  "store":{
    "book": [
      {
        "category":"reference",
        "author":"Nigel Rees",
        "title":"Sayings of the Century",
        "price":8.95
      },
      {
        "category":"fiction",
        "author":"J. R. R. Tolkien",
        "title":"The Lord of the Rings",
        "isbn":"0-395-19395-8",
        "price":22.99
      }
    ],
    "bicycle":{
      "color":"red",
      "price":19.95
    }
  }
}
```

JSON

JavaScript Object Notation



Cos'è il JSON?



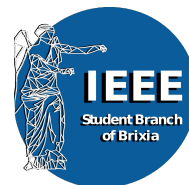
Il **JSON (JavaScript Object Notation)** è un formato standard per lo scambio di dati che utilizza un testo leggibile da umani per immagazzinare e trasmettere dati sotto forma di coppie attributo-valore e liste.



E' un formato di testo
completamente indipendente dal
linguaggio di programmazione



E' supportato nativamente da tutti i
principali database tra cui ad
esempio MySQL



Come mai dovremmo conoscerlo?

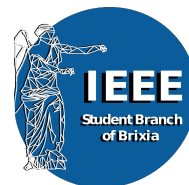


Il *JSON* ha iniziato a svilupparsi durante i primi anni 2000 e ha ottenuto grande successo ora grazie alla sua grande compatibilità con **JavaScript** (linguaggio di programmazione utilizzato nei browser), cosa che il formato XML non aveva.

Oggi JSON infatti oggi costituisce uno **standard** per lo scambio di dati tra client web e mobile e servizi di back-end. Tuttavia presenta alcuni limiti:

- Non presenta uno schema fisso
- Permette di immagazzinare un solo tipo di numero
- Non permette di immagazzinare date
- Non ammette commenti
- Non è verboso a differenza del formato XML

Nonostante ciò, è una notazione molto comoda e facile da utilizzare.





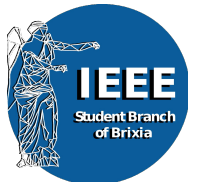
La struttura dei file JSON si basa su due paradigmi:

- **Un elenco ordinato di valori.** Nella maggior parte dei linguaggi questo si realizza con un array, un vettore, un elenco o una sequenza.
- **Un insieme di coppie chiave/valore** (Noto anche come dizionario, una tabella hash, un elenco di chiavi o un array associativo)

Questa struttura formata da coppie non ordinate chiave/valore è delimitata da due graffe ed è denominata come **oggetto**.

Anche un oggetto JSON può essere associato ad una chiave in questo modo:

```
"bicycle" : {  
  "color" : "red",  
  "price" : 19.95  
}
```



Struttura JSON



```
{  
  "nome": "Marco",  
  "cognome": "Rossi",  
  "matricola": "723688",  
  "anno di corso": 2,  
  "laureato": false  
}
```



I formati accettati per i valori in un file JSON sono

- String
- Numero
- Boolean (true o false)
- null

```
{  
  "corsi": [  
    {  
      "nome_corso": "Algebra",  
      "id": 0,  
      "docente": "Pellegrini",  
      "cfu": 9  
    },  
    {  
      "nome_corso": "Analisi I",  
      "id": 1,  
      "docente": "Gervasio",  
      "cfu": 9  
    }  
  ]  
}
```



Questa struttura invece rappresenta una collezione ordinata di oggetti delimitata dalle parentesi quadre in arancione, un **array** JSON.

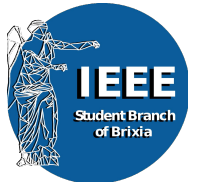


Parsing JSON in Java: Scrittura



Per effettuare il parsing in Java sono disponibili varie librerie, tra cui la libreria [com.google.gson](#):

```
String filePath = "./save.json";
FileWriter fileWriter = new FileWriter(new File(filePath));
Gson gson = new Gson();
ArrayList<User> users = new ArrayList<>();
users.add(new User("test@test.it", "vgfyuhiejdwgy78u9i0tuhygui9h8tijfdkoei09f"));
users.add(new User("mario@gmail.com", "ugihjpidg0pjthfd80yguiweqbfnoijpgeihog"));
gson.toJson(users, fileWriter);
fileWriter.close();
```

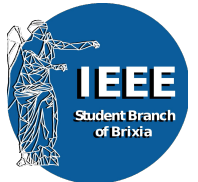


Parsing JSON in Java: Scrittura



Questo è il risultato se aprite il file `./save.json`:

```
[  
  {  
    "email": "test@test.it",  
    "password": "vgfyuhiejdwgy78u9i0tuhygui9h8tijfdkoei09f"  
  },  
  {  
    "email": "mario@gmail.com",  
    "password": "ugihjpigd0pjthfd80yguiweqbfnoijpgeihog"  
  }  
]
```

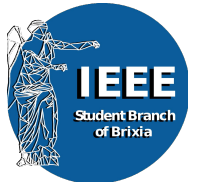


Parsing JSON in Java: Lettura



Per quanto riguarda invece la lettura con la libreria com.google.gson:

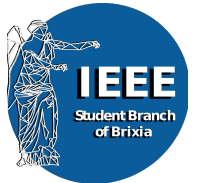
```
String filePath = "./save.json";  
FileReader fileReader = new FileReader(new File(filePath));  
Gson gson = new Gson();  
Type userType = new TypeToken<ArrayList<User>>(){}.getType();  
ArrayList<User> users = gson.fromJson(fileReader, userType);  
fileReader.close();
```





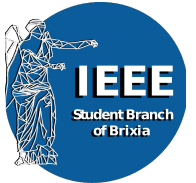
RTFM

Read The *Funny* Manual



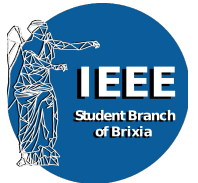


- Parsing (Wikipedia): <https://en.wikipedia.org/wiki/Parsing>
- Struttura Codice Fiscale:
https://it.wikipedia.org/wiki/Codice_fiscale#Descrizione
- XML: <https://en.wikipedia.org/wiki/XML>
- Documentazione XMLStreamReader:
<https://docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamReader.html>
- Documentazione XMLStreamWriter:
<https://docs.oracle.com/javase/7/docs/api/javax/xml/stream/XMLStreamWriter.html>
- Dubbi su XML?
 - <http://www.html.it/guide/guida-xml-di-base>: semplice ripasso dei fondamentali ed uso dell'xml
 - <https://www.w3schools.com/xml/default.asp>: tutorial completo
 - <https://www.w3.org/TR/xml/#sec-intro>: documentazione ufficiale





- Storia del formato JSON
<https://en.wikipedia.org/wiki/JSON>
- Repository Github di com.google.gson
<https://github.com/google/gson>
- Articolo in cui si ha un confronto tra JSON e XML
<https://www.guru99.com/json-vs-xml-difference.html>

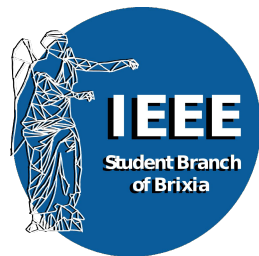


Presentazione realizzata per lo
Student Branch IEEE
dell'Università degli Studi di
Brescia, in occasione del
Programma Arnaldo 2023

*Si prega di non modificare o
distribuire il contenuto di tale
documento senza essere in possesso
dei relativi permessi*

corazzinamarco33@ieee.org
matteo.boniotti@ieee.org
stefano.agnelli@ieee.org
kibo@ieee.org

ieeesb.unibs.it



Grazie per l'attenzione