PARSING JSON - JAVA

Come detto i documenti JSON vengono utilizzati per lo scambio di dati fra applicazioni. I file JSON dunque rappresentano generalmente dei dati che un'applicazione A invia ad un'altra applicazione B. L'operazione di "estrazione", da parte dell'applicazione B, dei dati ricevuti in formato JSON, per poi poterli manipolare, è chiamata **parsing**. Il **parsing** in Java consente dunque di istanziare oggetti Java (ad esempio un oggetto libro) "leggendo" gli attributi da assegnare agli oggetti (ad esempio il titolo, l'autore, il numero di pagine del libro) da un file JSON. Questa operazione consiste, in pratica, nella deserializzazione di oggetti JSON in Java.

Vi sono diverse librerie che consentono di effettuare sia la deserializzazione (oggetti JSON→ oggetti Java) sia la serializzazione (oggetti Java→ oggetti JSON). Le librerie più utilizzate sono Jackson, Gson, Json-P, org.Gson.

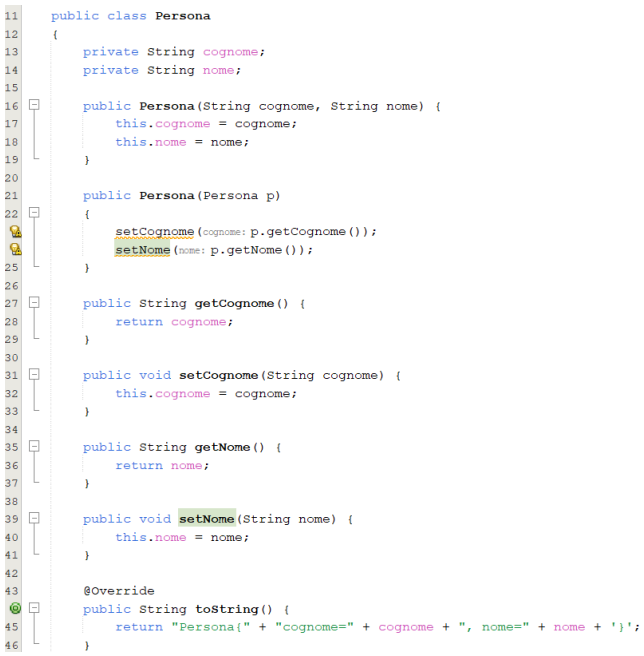
Vedremo ora la libreria Gson, che è una libreria open source creata da Google ed è descritta anche dal libro di testo.

Per l'installazione della libreria di Google Gson, si veda l'apposito file "INSTALLARE LIBRERIA GSON CON MAVEN NETBEANS"

Cap.1 Serializzare un semplice oggetto Java in formato JSON

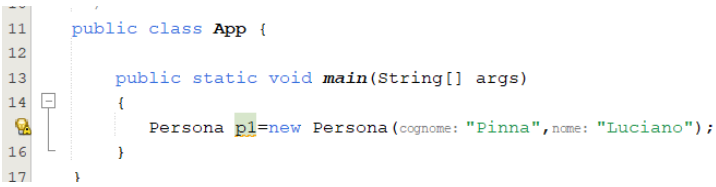
Serializzare un oggetto in formato JSON significa trasformare un oggetto in una stringa in formato JSON e poi salvarlo su un file di testo con estensione .json.

1. creiamo una semplice classe da serializzare, ad esempio la classe Persona, con attributi di tipo String cognome e nome, metodi getter e setter, costruttore, costruttore di copia, toString.



```
public class Persona
{
    private String cognome;
    private String nome;

    public Persona(String cognome, String nome) {
        this.cognome = cognome;
        this.nome = nome;
    }

    public Persona(Persona p)
    {
        setCognome(cognome: p.getCognome());
        setNome(nome: p.getNome());
    }

    public String getCognome() {
        return cognome;
    }

    public void setCognome(String cognome) {
        this.cognome = cognome;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public String toString() {
        return "Persona{" + "cognome=" + cognome + ", nome=" + nome + '}';
    }
```

2. Nella classe APP, nel metodo main, istanziamo una persona



```
public class App {

    public static void main(String[] args)
    {
        Persona p1=new Persona(cognome: "Pinna",nome: "Luciano");
    }
}
```

3. Nella classe APP creiamo un metodo privato statico (per poterlo invocare direttamente dalla classe) che restituisce l'oggetto JSON persona. Il valore di ritorno sarà una stringa in formato JSON, il parametro passato sarà un oggetto Java Persona. Chiamiamo tale metodo JavaToJson (Persona p). Nella classe App testiamo il metodo creato. Spieghiamo i metodi e le classi utilizzate:

```java
public class App {

    public static void main(String[] args)
    {
        Persona p1=new Persona(cognome: "Pinna",nome: "Luciano");
        System.out.println(x: JavaToJson(p: p1));
    }

    private static String JavaToJson(Persona p)
    {
        Gson gson=new Gson();
        String personaJson=gson.toJson(src: p);
        return personaJson;
    }
}
```

Istanzia un oggetto di **classe Gson** chiamato gson. La classe Gson fa parte della libreria Gson importata.

metodo **toJson** della classe Gson: converte l'oggetto passato come parametro jn una stringa Json che rappresenta lo stesso oggetto.

L'output risulta così:

```
{"cognome":"Pinna","nome":"Luciano"}
```

L'output è corretto, ma per formattarlo meglio, in modo che ogni coppia nome-valore sia su una nuova riga, si utilizza un apposito metodo "setPrettyPrinting". Il metodo setPrettyPrinting fa parte di una serie di metodi che consentono di specificare alcune caratteristiche della serializzazione/deserializzazione e che vedremo man mano. Per utilizzare "setPrettyPrinting" è necessario costruire l'oggetto Json NON DIRETTAMENTE DAL COSTRUTTORE, ma tramite un altro oggetto chiamato Gson Builder. Chiamiamo tale modo di istanziazione dell'oggetto Gson **"istanziazione custom di un oggetto Gson".** Il codice spiegato è il seguente:

```java
public class App {

    public static void main(String[] args)
    {
        Persona p1=new Persona(cognome: "Pinna",nome: "Luciano");
        System.out.println(x: JavaToJson(p: p1));
    }

    private static String JavaToJson(Persona p)
    {
        // Gson gson=new Gson();
         GsonBuilder builder=new GsonBuilder();
         builder.setPrettyPrinting();
         Gson gson=builder.create();

         String personaJson=gson.toJson(src: p);
         return personaJson;
    }
}
```

crea il GsonBuilder

nel GsonBuilder si possono specificare delle caratteristiche della serializzazione / deserializzazione per mezzo di appositi metodi, invocandoli in modo concatenato uno dopo l'altro. In questo caso invochiamo il metodo che migliora la formattazione: setPrettyPrinting.

crea l'oggetto di classe Gson dal GsonBuilder

La formattazione dell'output ora è quella standard per JSON:

```
{
  "cognome": "Pinna",
  "nome": "Luciano"
}
```

Oltre a "setPrettyPrinting", altri metodi invocabili che specificano le caratteristiche della serializzazione / deserializzazione, e che vedremo successivamente, sono riportati nella seguente tabella:

Metodo	Descrizione
`disableHtmlEscaping()`	Disabilita l'esclusione, attiva di default, dei caratteri tipici del linguaggio HTML come «<» e «>».
`disableInnerClassSerialization()`	Disabilita la serializzazione di classi innestate.
`excludeFieldsWithModifiers(int... modifier)`	Disabilita la serializzazione/de-serializzazione degli attributi definiti con il/i modificatore/i specificato/i.
`excludeFiledsWithoutExposeAnnotation()`	Disabilita la serializzazione/de-serializzazione degli attributi privi dell'annotazione `@Expose`.
`generateNonExecutableJson()`	Attiva la produzione di codice JSON non eseguibile in JavaScript mediante inserimento di specifici caratteri.
`registerTypeAdapter(Type type, Object adapter)`	Registra un istanziatore degli oggetti per le classi che non dispongono di un costruttore privo di parametri.
`serializeNulls()`	Abilita la serializzazione di valori nulli.
`serializeSpecialFloatingPointValues()`	Abilita la serializzazione dei valori numeri *floating-point* speciali (`NaN`, `+Infinity`, `-Infinity`).
`setDateFormat(String pattern)`	Imposta il formato per le date e gli orari in base alla stringa *pattern* (vedi TABELLA **12**).
`setFieldNamingPolicy(FieldNamingPolicy policy)`	Imposta una politica di trasformazione dei nomi degli attributi Java in JSON (vedi TABELLA **13**).
`setLongSerializationPolicy(LongSerializationPolicy policy)`	Imposta una politica di trasformazione dei valori numerici **`long`**/**`Long`** del linguaggio Java in JSON (vedi TABELLA **14**).
`setPrettyPrinting()`	Abilita la formattazione del codice JSON prodotto dalla serializzazione.

Per memorizzare l'oggetto sul file .json creiamo un altro metodo statico nella classe App e lo chiamiamo WriteStringToFile()

Qui andrà messo il contenuto del file json da creare

```
private static void WriteStringToFile(String pathName, String content) throws IOException
{
    Files.write(path: Paths.get(first: pathName), bytes: content.getBytes());
}
```

Oppure (meglio), per memorizzare l'oggetto sul file .json importiamo la nostra classe di utilità TextFile e la relativa eccezione FileException e scriviamo l'output del metodo JavaToJson() su un file di testo a cui diamo l'estensione .json, ad esempio: "persona.json". Ricordarsi di chiudere il TextFile dopo aver scritto.

Infine invochiamo il metodo appena realizzato nel main della classe App. La classe App diventa così (per facilità di lettura non sono state gestite le eccezioni).

```
17  public class App {
18
19      public static void main(String[] args) throws IOException
20      {
21          Persona p1=new Persona(cognome: "Pinna",nome: "Luciano");
22          WriteStringToFile(pathName: "persona.json", content: JavaToJson(p: p1));
23          System.out.println(x: JavaToJson(p: p1));
24      }
25
26      private static String JavaToJson(Persona p)
27      {
28          // Gson gson=new Gson();
29           GsonBuilder builder=new GsonBuilder();
30           builder.setPrettyPrinting();
31           Gson gson=builder.create();
32
33           String personaJson=gson.toJson(src: p);
34           return personaJson;
35      }
36
37      private static void WriteStringToFile(String pathName, String content) throws IOException
38      {
39          Files.write(path: Paths.get(first: pathName), bytes: content.getBytes());
40      }
41  }
42
```

L'output, migliorato, ora è il seguente, e questo sarà pure il modo con cui verranno scritti i dati sul file Json:

```
{
  "cognome": "Pinna",
  "nome": "Luciano"
}
```

Riassumendo, le classi e metodi della libreria Gson, visti in questo capitolo sono:

Classe **Gson():** istanzia oggetti che espongono i metodi per svolgere la serializzazione

- metodo **String toJson(Persona p)**: restituisce una stringa in formato Json dell'oggetto passato come parametro (serializzazione).

Classe **GsonBuilder**: permette di creare un'istanza della classe Gson specificando precedentemente opportuni metodi che descriveranno il comportamento dell'oggetto Gson creato:

- metodo **setPrettyPrinting**: fa sì che la formattazione delle stringhe restituite dal metodo **toJson**, abbiano le coppie parametro – valore, ognuna su una nuova riga
- metodo **create**(): istanzia l'oggetto Gson() a partire dal JsonBuilder

Cap.2 Deserializzare un semplice oggetto Java da un file JSON

Deserializzare un oggetto da un file JSON significa leggere dal file con estensione .json una stringa che rappresenta un oggetto ed istanziare il relativo oggetto nel programma in Java. Tale operazione è detta anche parsing.

- Scriviamo nella classe App dell'esempio precedente, un metodo (JsonToJava) che, a partire da una stringa in formato json passata come parametro (vedremo successivamente come leggere la stringa da un file), istanzia e restituisce un oggetto di classe "Persona", I nomi degli attributi devono coincidere con i nomi delle proprietà Json, i valori assunti dagli attributi saranno quelli dei valori assegnati alle proprietà Json:

```
public class App {

    public static void main(String[] args) throws IOException
    {

       //esempio deserializzazione oggetto semplice
       Persona p2;
       p2=jsonToJava("{\n" +
        "   \"cognome\": \"Brizzi\",\n" +
        "   \"nome\": \"Luigina\"\n" +
        "}");
        System.out.println("Persona deserializzata:\n"+p2.toString());
    }

    private static Persona jsonToJava(String stringaJson)
    {
        Gson gson=new GsonBuilder().setPrettyPrinting().create();
        Persona p=gson.fromJson(json: stringaJson, classOfT: Persona.class);
        return p;
    }

}
```

Istanzia un oggetto di **classe Gson** chiamato gson. Utilizzando il GsonBuilder come visto prima.

metodo **fromJson** della classe Gson: converte la stringa che rappresenta un oggetto json passato come parametro jn una oggetto Java.

Come primo parametro si passa la stringa Json

Come secondo parametro si passa il nome della classe (Persona) seguito da .class. Tale parametro indica la classe in cui "convertire" l'oggetto Json

Cosa succede se gli attributi della classe e i nomi delle proprietà Json non coincidono? Ad esempio se si tenta di convertire il seguente oggetto Json in un' istanza di Persona?

{
"cognome":"Brizzi",
"nazionalita":"italiana"
}

Questa proprietà non ha un corrispondente attributo nella classe Persona

Viene comunque creato un oggetto Java istanza di Persona dove gli attributi che non hanno la corrispondente proprietà vengono posti a null. Nell'esempio, il valore "nome" dell'oggetto Java creato viene posto a null.

OSSERVAZIONE: Anche se l'ordine in cui sono scritte le proprietà è diverso da quello degli attributi, la deserializzaizone avviene comunque. Ad esempio se in Json si mette prima la proprietà "nome" e poi la proprietà "cognome", i valori di cognome e nome vengono comunque assegnati correttamente all'oggetto Persona istanziato.

Aggiungiamo anche in questo caso un metodo (readStringFromFile) che legge la stringa Json da deserializzare da un file persona.json. Il codice diventa il seguente:

Oppure, meglio: realizziamo un metodo che legge il file .json utilizzando la nostra classe TextFile

```
17  public class App {
18
19      public static void main(String[] args) throws IOException
20      {
21
22          //esempio deserializzazione oggetto semplice
23          Persona p2;
24          String stringaJson=readStringFromFile(pathname: "persona.json");
25          p2=jsonToJava(stringaJson);
26          System.out.println("Persona deserializzata:\n"+p2.toString());
27      }
28
29      private static Persona jsonToJava(String stringaJson)
30      {
31          Gson gson=new GsonBuilder().setPrettyPrinting().create();
32          Persona p=gson.fromJson(json: stringaJson, classOfT: Persona.class);
33          return p;
34      }
35
36      private static String readStringFromFile(String pathname) throws IOException
37      {
38          byte[] content=Files.readAllBytes(path: Paths.get(first: pathname));
39          return new String(bytes: content);
40      }
41
42  }
```

Legge il file come array di byte

Trasforma l'array di byte in una stringa

Riassumendo, le classi e metodi della libreria Gson, visti in questo capitolo sono:

- NomeClasse identificativo = Gson.fromJson (String stringaFormatoJason, NomeClasse.class):

  Istanzia e restituisce un oggetto di classe NomeClasse.

Cap.3 Serializzare e Deserializzare in Json un oggetto Java nidificato

Per serializzare/deserializzare un oggetto Java fra i cui attributi vi sono altri oggetti, il codice è lo stesso visto nel caso di oggetto semplice. Automaticamente i metodi toJson e fromJson creeranno le strutture nidificate sia nella serializzazione che nella deserializzazione.

Attenzione: in alcuni casi la libreria Gson non può accedere ad alcuni degli attributi da serializzare/deserializzare (perché tali attributi fanno parte di librerie del JDK o di librerie di terze parti, sono privati e non consentono accesso tramite getter e setter, …). In questo caso la serializzazione/deserializzazione fallisce e viene mostrata la seguente eccezione: "inaccesibleObjectException". Qui è spiegato il problema: https://github.com/google/gson/blob/main/Troubleshooting.md

Questo è ciò che accade, ad esempio, se si aggiunge alla classe Persona un attributo DataNascita, di classe LocalDate. In questo caso la libreria Gson non può "accedere" agli attributi privati di questa classe (Year, Month, Day), quando si tenta di serializzare/deserializzare un'istanza di questo tipo dunque si genera un'eccezione "inaccesibleObjectException". La soluzione è quella di definire un type adapter ossia una classe che rende disponibili tali attributi alla libreria Gson, e viene mostrata in fondo a questi appunti nel cap.9.

Per esempio, aggiungiamo al progetto una classe Automobile (con attributi: marca, modello e targa e i relativi metodi principali). Aggiungiamo alla classe Persona un attributo automobile, istanza di classe Automobile che rappresenta l'automobile che la persona possiede. Il codice per serializzare/deserializzare un oggetto di classe Persona che ha un attributo di classe Automobile è il seguente:

```java
public class App
{
    public static void main(String[] args) throws IOException
    {
        //serializzazione
        Persona p=new Persona(cognome: "Pinna",nome: "Luciano", new Automobile(marca: "Fiat",modello: "Panda",targa: "we23ed"));
        String personaJson=javaToJson(p);
        WriteStringToFile(pathName: "persona.json", content: personaJson);

        //deserializzazione
        personaJson=readStringFromFile(pathname: "persona2.json");
        Persona p2=jsonToJava(stringaJson: personaJson);
        System.out.println("persona deserializzata:\n"+p2.toString());
    }

    private static String javaToJson(Persona p)
    {
       // Gson gson=new Gson();
        GsonBuilder builder=new GsonBuilder();
        builder.setPrettyPrinting();
        Gson gson=builder.create();

        String personaJson=gson.toJson(src: p);
        return personaJson;
    }

    private static Persona jsonToJava(String stringaJson)
    {
        Gson gson=new GsonBuilder().setPrettyPrinting().create();
        Persona p=gson.fromJson(json: stringaJson, classOfT: Persona.class);
        return p;
    }
```

Serializzazione, analogo ad oggetto semplice produce la seguente stringa:

```
{
  "cognome": "Pinna",
  "nome": "Luciano",
  "automobile": {
    "marca": "Fiat",
    "modello": "Panda",
    "targa": "we23ed"
  }
}
```

Deserializzazione, analogo ad oggetto semplice, genera un oggetto Persona a partire dalla seguente stringa letta dal file persona2.json:

```
{
  "cognome": "Brizzi",
  "nome": "Luigina",
  "automobile": {
    "marca": "Audi",
    "modello": "a4",
    "targa": "uj434uj"
  }
}
```

```java
    private static String readStringFromFile(String pathname) throws IOException
    {
        byte[] content=Files.readAllBytes(path: Paths.get(first: pathname));
        return new String(bytes: content);
    }

    private static void WriteStringToFile(String pathName, String content) throws IOException
    {
        Files.write(path: Paths.get(first: pathName), bytes: content.getBytes());
    }
}
```

Metodi per scrivere/leggere su/da file .json

Cap.4 Serializzare e Deserializzare in Json un oggetto Java con attributi di tipo array o arrayList di altri oggetti.

In JSON un elenco di oggetti viene sempre rappresentato come un array, quindi nelle operazioni di serializzazione / deserrilizzazione è indifferente se l'elenco di oggetti in java è un array o una collection (arrayList, linkedList ecc.). Anche in questo caso il codice per la serializzazione e deserializzazione è lo stesso visto per gli oggetti semplici. Il metodo toJson converte automaticamente un attributo di tipo array o arrayList in un array Json. Il metodo fromJson converte automaticamente una proprietà array di Json nella corrispondente array o arrayList in base a come tale attributo è definito nella classe Java.

Per esempio modifichiamo la classe Persona sostituendo l'attributo "automobile" con un attributo "automobili" che è un array di 10 (numero massimo) automobili.

La classe Persona diventa la seguente (si mostra solo la parte iniziale con i costruttori, si è trascurata l'indipendenza degli oggetti per semplicità …):

```
16  public class Persona
17  {
18      private String cognome;
19      private String nome;
20      private Automobile[] automobili;
21
22      public Persona(String cognome, String nome, Automobile[] automobili)
23      {
24          this.cognome = cognome;
25          this.nome = nome;
26          this.automobili=automobili;
27      }
28
29      public Persona(Persona p)
30      {
            setCognome(cognome: p.getCognome());
            setNome(nome: p.getNome());
            setAutomobili(elencoAuto: p.getAutomobili());
34      }
35
36      public String getCognome() {
37          return cognome;
38      }
39
40      public void setCognome(String cognome) {
41          this.cognome = cognome;
42      }
43
```

Il codice della classe App viene modificato in parte solo nel metodo main, i metodi per serializzazione e deserializzazione non cambiano, neppure quelli per scrivere/leggere su/da file .json (non riportati):

```java
public class App
{
    public static void main(String[] args) throws IOException
    {
        //creo un array con due automobili
        Automobile auto1=new Automobile(marca: "Fiat",modello: "Panda",targa: "we23ed");
        Automobile auto2=new Automobile(marca: "Audi",modello: "A4",targa: "iu87ed");
        Automobile[] elencoAutomobili={auto1,auto2};

        //serializzazione
        Persona p=new Persona(cognome: "Pinna",nome: "Luciano", automobili: elencoAutomobili);
        String personaJson=javaToJson(p);
        WriteStringToFile(pathName: "persona.json", content: personaJson);

        //deserializzazione
        personaJson=readStringFromFile(pathname: "persona2.json");
        Persona p2=jsonToJava(stringaJson: personaJson);
        System.out.println("persona deserializzata:\n"+p2.toString());
    }
    private static String javaToJson(Persona p)
    {
       // Gson gson=new Gson();
        GsonBuilder builder=new GsonBuilder();
        builder.setPrettyPrinting();
        Gson gson=builder.create();

        String personaJson=gson.toJson(src: p);
        return personaJson;
    }
    private static Persona jsonToJava(String stringaJson)
    {
        Gson gson=new GsonBuilder().setPrettyPrinting().create();
        Persona p=gson.fromJson(json: stringaJson, classOfT: Persona.class);
        return p;
    }
```

Serializzazione, analogo ad oggetto semplice produce la seguente stringa:

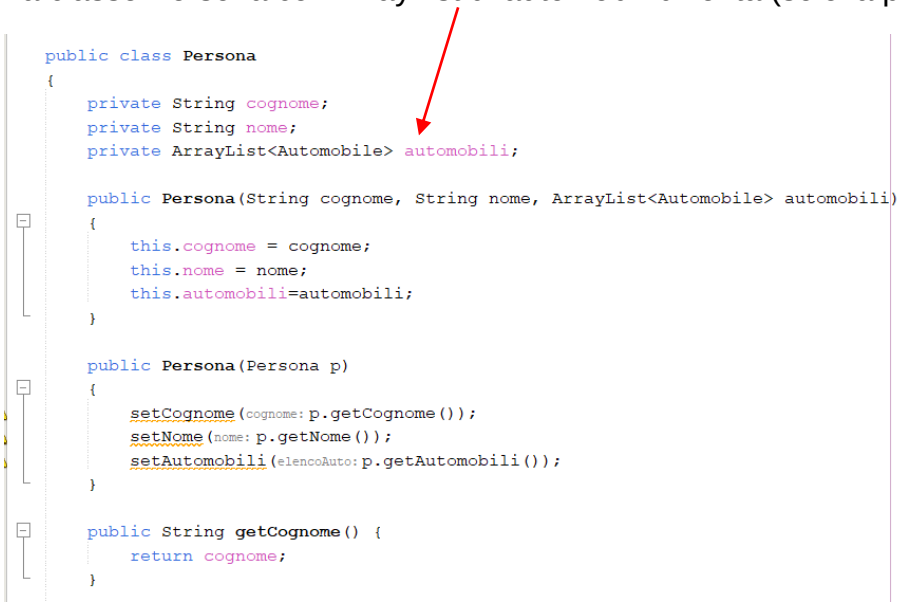
```
{
  "cognome": "Pinna",
  "nome": "Luciano",
  "automobili": [
    {
      "marca": "Fiat",
      "modello": "Panda",
      "targa": "we23ed"
    },
    {
      "marca": "Audi",
      "modello": "A4",
      "targa": "iu87ed"
    }
  ]
}
```

Deserializzazione, analogo ad oggetto semplice, genera un oggetto Person a partire dalla seguente stringa letta da file persona2.json:

```
{
  "cognome": "Brizzi",
  "nome": "Luigina",
  "automobili": [
    {
      "marca": "Audi",
      "modello": "A4",
      "targa": "ee23ee"
    },
    {
      "marca": "Ford",
      "modello": "Focus",
      "targa": "ui89sd"
    }
  ]
}
```

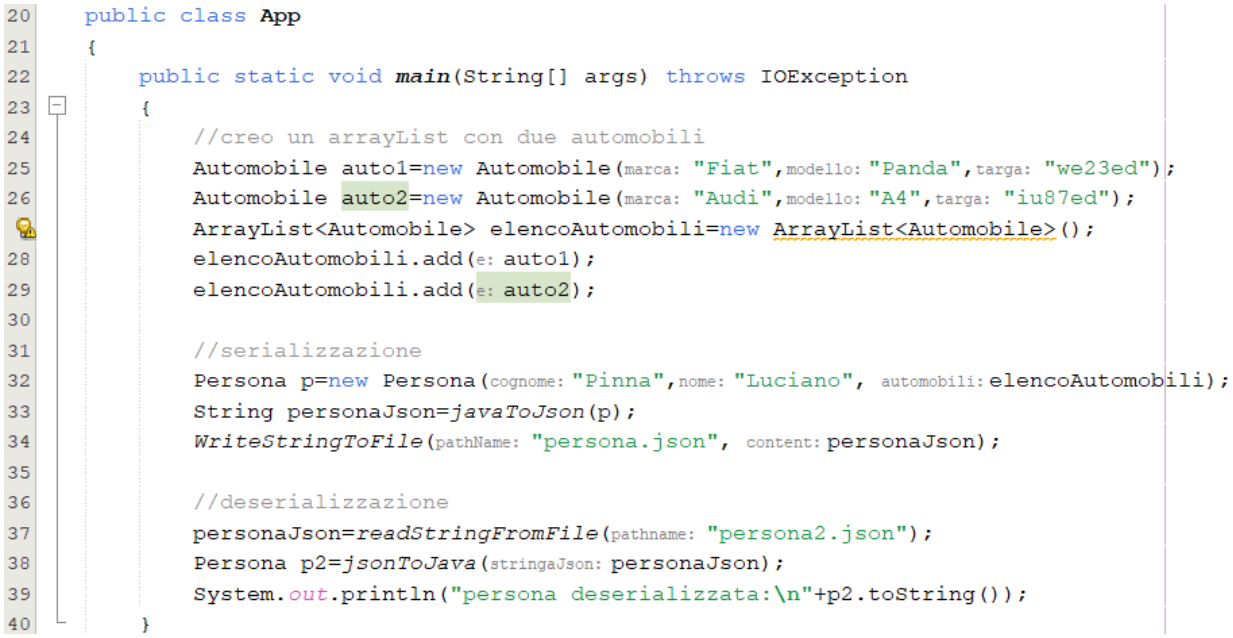
Se anziché definire le automobili con un array si definiscono con un arrayList (come mostrato di seguito) il codice funziona comunque. Il file persona1.json ottenuto con l serializzazione è identico, l'oggetto persona ottenuto dallo stesso file persona2.json con la deserializzazione avrà automaticamente un arrayList di automobili anziché un array di automobili.

La classe Persona con ArrayList di automobili diventa (solo la parte iniziale..)



```
public class Persona
{
    private String cognome;
    private String nome;
    private ArrayList<Automobile> automobili;

    public Persona(String cognome, String nome, ArrayList<Automobile> automobili)
    {
        this.cognome = cognome;
        this.nome = nome;
        this.automobili=automobili;
    }

    public Persona(Persona p)
    {
        setCognome(cognome: p.getCognome());
        setNome(nome: p.getNome());
        setAutomobili(elencoAuto: p.getAutomobili());
    }

    public String getCognome() {
        return cognome;
    }
```

La classe App viene modificata solo nel metodo Main per istanziare l'arrayList di automobili



```
20  public class App
21  {
22      public static void main(String[] args) throws IOException
23      {
24          //creo un arrayList con due automobili
25          Automobile auto1=new Automobile(marca: "Fiat",modello: "Panda",targa: "we23ed");
26          Automobile auto2=new Automobile(marca: "Audi",modello: "A4",targa: "iu87ed");
27          ArrayList<Automobile> elencoAutomobili=new ArrayList<Automobile>();
28          elencoAutomobili.add(e: auto1);
29          elencoAutomobili.add(e: auto2);
30
31          //serializzazione
32          Persona p=new Persona(cognome: "Pinna",nome: "Luciano", automobili: elencoAutomobili);
33          String personaJson=javaToJson(p);
34          WriteStringToFile(pathName: "persona.json", content: personaJson);
35
36          //deserializzazione
37          personaJson=readStringFromFile(pathname: "persona2.json");
38          Persona p2=jsonToJava(stringaJson: personaJson);
39          System.out.println("persona deserializzata:\n"+p2.toString());
40      }
```

Il risultato della serializzazione e deserializzazione è lo stesso del caso con l'array.

Cap.5 Serializzare e Deserializzare in\da Json un array di oggetti Java

Quando si vuole serializzare un array di oggetti Java, ciò che si ottiene in Json è un array di oggetti Json, ossia un file Json in cui l'elemento radice è un array. Ad esempio consideriamo di voler serializzare un array di oggetti di classe Persona (per semplicità consideriamo la classe Persona semplice, che ha come unici attributi cognome e nome, poiché comunque, come abbiamo visto, anche se gli oggetti hanno attributi più complessi, il codice per la serializzazione/deserializzazione non cambia).

Il metodo usato per la serializzazione, "toJson", non cambia rispetto a quanto già visto, infatti accetta come parametro anche un array. Il codice della classe App è il seguente:

```
20  public class App
21  {
22      public static void main(String[] args) throws IOException
23      {
24          Persona[] elencoPersone=new Persona[2]; //array di persone
25          //serializzazione
26          Persona p1=new Persona(cognome: "Pinna",nome: "Luciano");
27          Persona p2=new Persona(cognome: "Massini",nome: "Pietro");
28          elencoPersone[0]=p1;
29          elencoPersone[1]=p2;
30
31          String personaJson=javaToJson(elencoP: elencoPersone);
32          WriteStringToFile(pathName: "elencoPersone.json", content: personaJson);
33
34      }
35      private static String javaToJson(Persona[] elencoP)
36      {
37         // Gson gson=new Gson();
38          GsonBuilder builder=new GsonBuilder();
39          builder.setPrettyPrinting();
40          Gson gson=builder.create();
41          String personaJson=gson.toJson(src: elencoP);
42          return personaJson;
43      }
44
45      private static void WriteStringToFile(String pathName, String content) throws IOException
46      {
47          Files.write(path: Paths.get(first: pathName), bytes: content.getBytes());
48      }
49  }
```

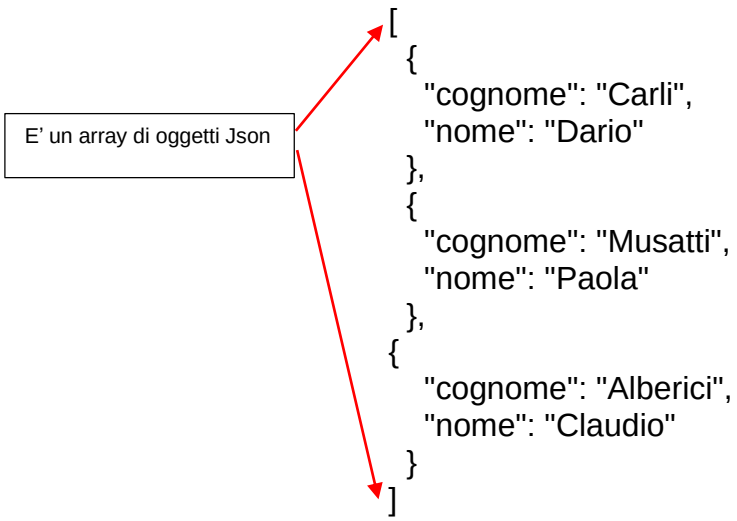
Serializzazione, analogo ad oggetto semplice. Il metodo toJson accetta come parametro anche un array

Si ottiene il seguente file elencoPersone.json:

```
[
  {
    "cognome": "Pinna",
    "nome": "Luciano"
  },
  {
    "cognome": "Massini",
    "nome": "Pietro"
  }
]
```

E' un array di oggetti Json

Per la deserializzazione cambia molto poco rispetto ai casi già visti, bisogna solamente ricordarsi che nel metodo “fromJson” va specificato che la classe verso cui si vuole effettuare la deserializzazione è un array di oggetti e non un semplice oggetto. Per esempio se si vuole deserializzare in un array java il seguente file “persone.json” che contiene un array di oggetti json:



Il codice è il seguente:

```
20  public class App
21  {
22      public static void main(String[] args) throws IOException
23      {
25          Persona[] elencoPersone=null;
26          String elencoPersoneJson;
27          elencoPersoneJson=readStringFromFile(pathname: "persone.json");
28          elencoPersone=jsonToJava(stringaJson: elencoPersoneJson);
29          for(Persona p:elencoPersone)
30          {
31              System.out.println(x: p.toString());
32          }
33      }
34
35      //Restituisce un array di persone
36  private static Persona[] jsonToJava(String stringaJson)
37      {
38          Gson gson=new GsonBuilder().setPrettyPrinting().create();
39          Persona[] elencoPersone;
40          elencoPersone=gson.fromJson(json: stringaJson, classOfT: Persona[].class);
41          return elencoPersone;
42      }
43
44
45     private static String readStringFromFile(String pathname) throws IOException
46      {
47          byte[] content=Files.readAllBytes(path: Paths.get(first: pathname));
48          return new String(bytes: content);
49      }
50
51
52  }
```

Deserializzazione: va specificato che la classe restituita dal metodo fromJson è **un array** di oggetti

Cap.6 Serializzare e Deserializzare in\da Json un ArrayList (o altra collection) di oggetti Java

Rispetto al caso precedente (array di oggetti in Java), in questo caso la deserializzazione richiede un'informazione in più, infatti quando si ha un array di oggetti Json (l'elemento radice del documento è un array), al momento della deserializzazione è necessario specificare se, in Java, tale elenco di oggetti deve diventare un ArrayList oppure una LinkedList oppure, eventualmente un'altra collection di dati.

Pertanto la serializzazione è analoga al caso degli array in Java ma la deserializzazione no. Il codice per serializzare in Json un ArrayList di Persone è il seguente:

```
20  public class App
21  {
22      public static void main(String[] args) throws IOException
23      {
24          Persona p1=new Persona (cognome: "Pinna", nome: "Luciano");
25          Persona p2=new Persona (cognome: "Massini", nome: "Pietro");
26
            ArrayList<Persona> elencoPersone=new ArrayList<Persona>();
28          elencoPersone.add(e: p1);
29          elencoPersone.add(e: p2);
30          String elencoPersoneJson=javaToJson(elencoPersone);
31          WriteStringToFile(pathName: "elencoPersone.json", content: elencoPersoneJson);
32
33      }
34
35      private static String javaToJson(ArrayList<Persona> elencoPersone)
36      {
37          // Gson gson=new Gson();
38           GsonBuilder builder=new GsonBuilder();
39           builder.setPrettyPrinting();
40           Gson gson=builder.create();
41           String personaJson=gson.toJson(src: elencoPersone);
42           return personaJson;
43      }
44      private static void WriteStringToFile(String pathName, String content) throws IOException
45      {
46          Files.write(path: Paths.get(first: pathName), bytes: content.getBytes());
47      }
48  }
```

Serializzazione, analogo ad oggetto semplice. Il metodo toJson accetta come parametro anche un ArrayList

Si ottiene, come nel caso precedente, il seguente file elencoPersone.json:

E' un array di oggetti Json

```
[
  {
    "cognome": "Pinna",
    "nome": "Luciano"
  },
  {
    "cognome": "Massini",
    "nome": "Pietro"
  }
]
```

Per la deserializzazione va specificato con un' apposita istruzione il tipo di collection Java verso cui deserializzare. Per esempio se si vuole deserializzare in un ArrayList Java il seguente file "persone.json" che contiene un array di oggetti json:

E' un array di oggetti Json

```
[
  {
    "cognome": "Carli",
    "nome": "Dario"
  },
  {
    "cognome": "Musatti",
    "nome": "Paola"
  },
 {
    "cognome": "Alberici",
    "nome": "Claudio"
  }
]
```

Il codice, spiegato, è il seguente:

```
19  public class App
20  {
21      public static void main(String[] args) throws IOException
22      {
23          ArrayList<Persona> elencoPersone;
24          String elencoPersoneJson=readStringFromFile(pathname: "elencoPersone.json");
25          elencoPersone=jsonToJava(stringaJson: elencoPersoneJson);
26          System.out.println(x: elencoPersone.toString());
27      }
28
29      private static ArrayList<Persona> jsonToJava(String stringaJson)
30      {
31          Gson gson=new GsonBuilder().setPrettyPrinting().create();
32          Type elencoPersoneType=new TypeToken<ArrayList<Persona>>(){}.getType();
33          ArrayList<Persona> elencoPersone=gson.fromJson(json: stringaJson, typeOfT: elencoPersoneType);
34          return (ArrayList<Persona>) elencoPersone;
35      }
36      private static String readStringFromFile(String pathname) throws IOException
37      {
38          byte[] content=Files.readAllBytes(path: Paths.get(first: pathname));
39          return new String(bytes: content);
40      }
41  }
```

legge l'array di oggetti Json dal file, restituisce un stringa.

Con questa istruzione si "sceglie" in quale tipo di dato collection java si vuole "trasformare" l'elenco di oggetti JSON (ArrayList, un Linked List ecc…)

Si specifica che l'elenco Json deve essere "trasformato nel tipo di dato indicato

Classe Type importata da **java.lang.reflect.Type;** attenzione perché ci possono essere più librerie nei suggerimenti

Classe TypeToken importata da **com.google.gson.reflect.TypeToken**

attenzione perché ci possono essere più librerie nei suggerimenti

Cap.7 Serializzare e Deserializzare in\da Json valori Null con il metodo serializableNulls

Quando si vuole serializzare un oggetto Java che ha un attributo con valore null, di default la proprietà Json corrispondente a tale attributo non viene creata. Questa scelta è stata fatta per far sì che il file Json sia il più piccolo possibile.

Nei prossimi codici per semplicità non verrà mostrata la parte che salva e legge su/da file, ma semplicemente la generazione/lettura di stringhe Json.

Esempio: aggiungiamo alla classe Persona un attributo eta di tipo Integer **(attenzione: è importante osservare che sto utilizzando la classe wrapper Integer anziché int, vedremo dopo perché)**

```
13    public class Persona
14    {
15        private String cognome;
16        private String nome;
17        private Integer eta;
18
19        //.........
```

Nella classe App istanziamo un oggetto Persona con eta=null e vediamo che la stringa json ottenuta non contiene la proprietà eta:

```
19  public class App
20  {
21      public static void main(String[] args) throws IOException
22      {
23          Persona p1=new Persona(cognome: "Pinna",nome: "Luciano",eta:null);
24          Gson gson=new GsonBuilder().setPrettyPrinting().create();
25          String jsonPersona=gson.toJson(src:p1);
26           System.out.println(x: jsonPersona);
27      }
28  }
```

Risultato

```
{
  "cognome": "Pinna",
  "nome": "Luciano"
}
```

La proprietà "eta" non viene creata

Poiché in alcuni casi si vuole che compaiano in Json anche i valori null, (ad esempio poiché è richiesto dalle API che si utilizzano) è possibile modificare l'opzione di default in modo che gli attributi con valore null vengano comunque serializzati. Ciò sui ottiene con il metodo "**serializeNulls**" invocato in maniera concatenata al GsonBuilder. Il metodo "**serializeNulls**" è uno dei metodi, già citati, che specificano il comportamento della serializzazione/deserializzazione.

```java
public class App
{
    public static void main(String[] args) throws IOException
    {
        Persona p1=new Persona(cognome: "Pinna",nome: "Luciano",eta:null);
        Gson gson=new GsonBuilder().setPrettyPrinting().serializeNulls().create();
        String jsonPersona=gson.toJson(src:p1);
        System.out.println(x: jsonPersona);
    }
}
```

Risultato

```
{
  "cognome": "Pinna",
  "nome": "Luciano",
  "eta": null
}
```

La proprietà "eta" viene creata

Il problema non si pone con la deserializzazione poiché in qualsiasi caso, la stringa json sia senza attributo "eta" sia con attributo "eta"="null", durante la deserializzazione assegna all'attributoi eta il valore null, indipendentemente dall'utilizzo o meno del metodo "**serializeNulls**" .

OSSERVAZIONE: è necessario porre molta attenzione alla deserializzazione di tipi di dato nativi quando i corrispondenti valori nella stringa Json non sono presenti. Poiché i tipi di dato nativi sono non "nullable" in Java, ossia non possono assumere valore null poiché non sono oggetti, ad essi verrà assegnato valore 0 se sono numerici e false se sono booleani. Per esempio se l'attributo eta fosse int anziché Integer, la deserializzazione seguente porterebbe ad ottenere un oggetto Java Persona con eta = 0:

```java
public class App
{
    public static void main(String[] args) throws IOException
    {

        String personaJson="{\n" +
         "   \"cognome\": \"Pinna\",\n" +
         "   \"nome\": \"Luciano\"\n" +
         "}";
        Gson gson=new GsonBuilder().setPrettyPrinting().serializeNulls().create();
         Persona p=gson.fromJson(json: personaJson, classOfT: Persona.class);
          System.out.println(x: p.toString());
```

```
Persona{cognome=Pinna, nome=Luciano, eta=0}
```

ATTENZIONE: all'attributo eta viene assegnato valore 0. ma avere 0 significa che si ha meno di un anno di vita, mentre avere eta=null significa non conoscere il valore dell'eta. Sono due cose diverse. Attenzione. Il problema si risolve utilizzando per eta la classe wrapper Integer anziché int.

Cap. 8 Altre cose che si possono fare: (non fare questo capitolo)

1. Modificare i nomi delle proprietà con i quali verranno serializzati gli attributi. Ad esempio far sì che l'attributo "nome" venga serializzato in Json avendo come nome della proprietà "name" anziché "nome". Per fare questo si utilizza un'annotazione **"@serializedName**"

Esempio

```
15    public class Persona
16    {
17        private String cognome;
18        @SerializedName(value="name", alternate="nome")
19        private String nome;
20        private Integer eta;
21
22        //.........
```

Serializzazione:

La proprietà json avrà come nome "name" anziché "nome"

Deserializzazione:

Indica che anche gli oggetti che hanno una proprietà chiamata "name" vanno deserializzati assegnando il valore all'attributo "nome". Quindi sia

"nome": "Luciano"

sia

"name":Luciano"

verranno deserializzati correttamente

La serializzazione produrrà questo oggetto Json

{

"cognome": "Pinna",

"**name**": "Luciano",

"eta": null

}

2. Selezionare quali attributi serializzare/deserializzare con il tag @expose e il metodo excludeFieldsWithoutExposeAnnotation

Invocando il metodo "excludeFieldsWithoutExposeAnnotazion" durante un'istanziazione custom di un oggetto Gson, accade che:

tutti gli attributi privi dell'annotazione @Expose() non verrano serializzati e neppure deserializzati

tutti gli attributi in cui è presente l'annotazione @Expose() senza parametri verrano serializzati e deserializzati

tutti gli attributi in cui è presente l'annotazione con parametri @Expose(serialize true/false, deserialize=true/false) seguiranno le indicazioni verranno o non verranno serializzati/deserializzati in base alle indicazioni dei parametri.

Esempio:

Classe Persona con attributi senza @Expose()

```
public class Persona
{
    private String cognome;
    private String nome;
    private Integer eta;

    //.........
```

```
public class App
{
    public static void main(String[] args) throws IOException
    {
        //serializzazione
        Persona p1=new Persona(cognome: "Pinna",nome: "Luciano",eta:27);
        Gson gson=new GsonBuilder().setPrettyPrinting().excludeFieldsWithoutExposeAnnotation().create();
        String jsonPersona=gson.toJson(src:p1);
        System.out.println(x: jsonPersona);

        //deserializzazione
        String jsonPersona2="{\n" +
                            "  \"cognome\": \"Pinna\",\n" +
                            "  \"nome\": \"Luciano\",\n" +
                            "  \"eta\": 27\n" +
                            "}";
        Persona p2=gson.fromJson(json: jsonPersona2, classOfT: Persona.class);
        System.out.println("Persona 2:"+p2.toString());

    }
}
```

jsonPersona={ },

nessun attributo serializzato

Persona 2:Persona {cognome=null, nome=null, eta=null}

nessun attributo deserializzato

Modificando gli attributi della classe Persona nel seguente modo si ottiene che tutti gli attributi verranno serializzati/deserializzati dal codice precedente della classe APP

```
public class Persona
{
    @Expose()
    private String cognome;
    @Expose()
    private String nome;
    @Expose()
    private Integer eta;

    //.........
```

Modificando gli attributi della classe Persona nel seguente modo:

```
16    public class Persona
17    {
18        @Expose(serialize=true, deserialize=false)
19        private String cognome;
20        @Expose(serialize=false, deserialize=true)
21        private String nome;
22        @Expose()
23        private Integer eta;
24
```

"cognome" verrà serializzato ma non deserializzato

"nome" non verrà serializzato ma verrà deserializzato

""eta" verrà serializzato e deserializzato

dal codice precedente della classe APP, che mostriamo di nuovo per comodità:

```
public class App
{
    public static void main(String[] args) throws IOException
    {
        //serializzazione
        Persona p1=new Persona(cognome: "Pinna",nome: "Luciano",eta:27);
        Gson gson=new GsonBuilder().setPrettyPrinting().excludeFieldsWithoutExposeAnnotation().create();
        String jsonPersona=gson.toJson(src:p1);
        System.out.println(x: jsonPersona);

         //deserializzazione
        String jsonPersona2="{\n" +
                            "   \"cognome\": \"Carli\",\n" +
                            "   \"nome\": \"Dario\",\n" +
                            "   \"eta\": 32\n" +
                            "}";
        Persona p2=gson.fromJson(json: jsonPersona2, classOfT: Persona.class);
         System.out.println("Persona 2:"+p2.toString());

    }
}
```

{
"cognome": "Pinna",
"eta": 27
}
"nome" non viene serializzato

Persona 2:Persona{cognome=null, nome=Dario, eta=32}

"cognome" non viene deserializzato

Cap. 9 Consentire a Gson l’accesso ad attributi inaccessibili mediante TypeAdapter: esempio con la classe LocalDate

Come già detto nel cap. 3 di questi appunti. Se aggiungiamo alla classe Persona un attributo dataNascita di tipo LocalDate, la serializzazione/deserializzazione non va a buon fine poiché la libreria Gson non può accedere agli attributi privati della classe LocaldDate(Year, Month, Day).

Esempio:

```
public class Persona
{

    private String cognome;
    private String nome;
    private LocalDate dataNascita;

    public Persona(String cognome, String nome, LocalDate d)
    {
        this.cognome = cognome;
        this.nome = nome;
        this.dataNascita=d;
    }

     public Persona(String cognome, String nome, int anno, int mese, int giorno)
    {
        this.cognome = cognome;
        this.nome = nome;
        this.dataNascita=LocalDate.of(year: anno,month: mese,dayOfMonth: giorno);
    }

    public LocalDate getDataNascita() {
        return dataNascita;
    }

    public void setDataNascita(LocalDate dataNascita) {
        this.dataNascita = dataNascita;
    }

    //.........
```

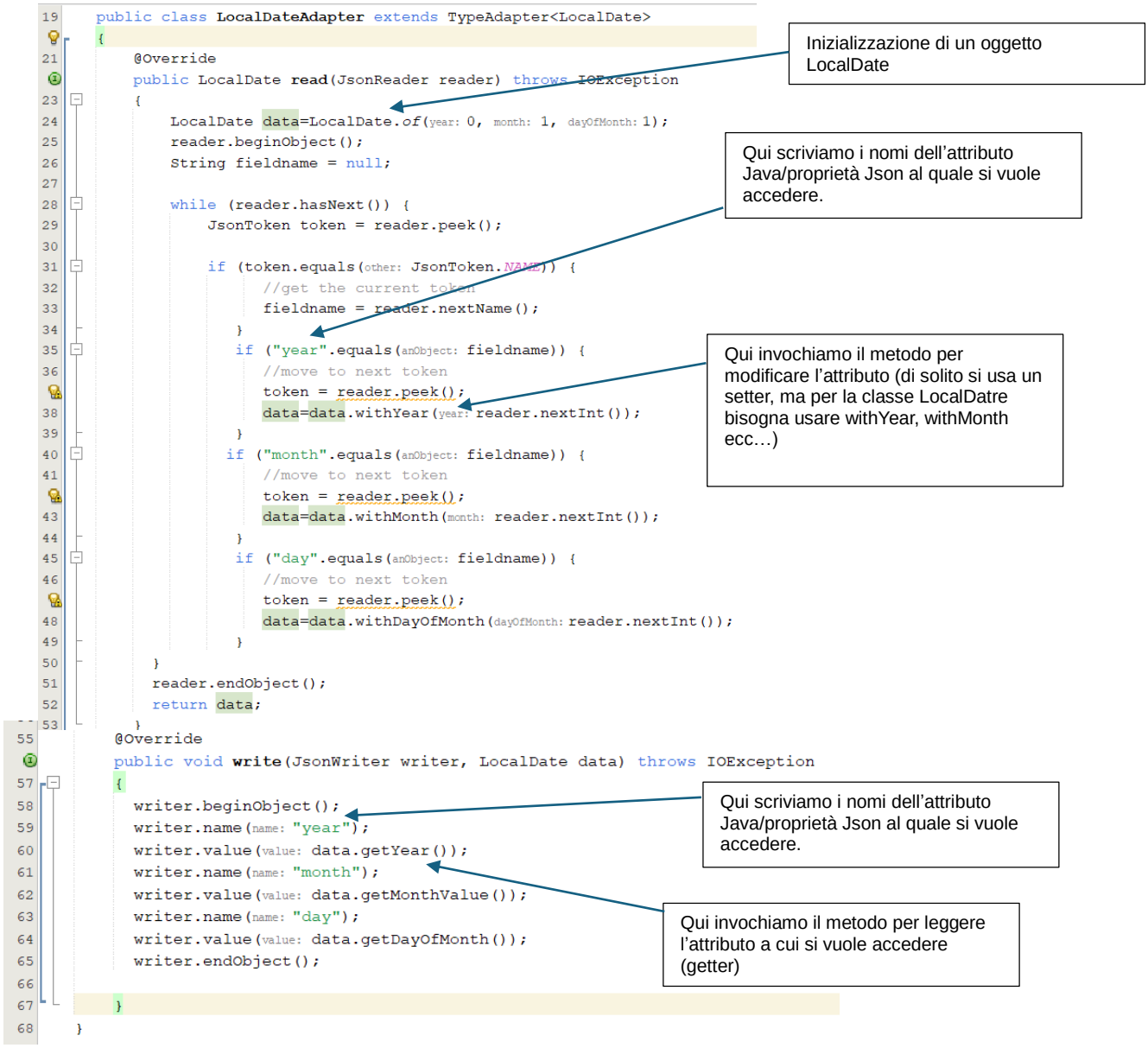
Il seguente codice per serializzare/deserializzare, nella classe App genera l’eccezione InaccessibleObjectException

```java
public class App
{
    public static void main(String[] args) throws IOException
    {
        //serializzazione
       LocalDate dataNascita=LocalDate.of(year: 2000,month: 10,dayOfMonth: 20);

       Persona p1=new Persona(cognome: "Pinna",nome: "Luciano",d: dataNascita);
       Gson gson=new GsonBuilder().setPrettyPrinting().create();
       String jsonPersona=gson.toJson(src:p1);
       System.out.println(x: jsonPersona);

        //deserializzazione
       String jsonPersona2="{\n" +
"  \"cognome\": \"Albrici\",\n" +
"  \"nome\": \"Giulio\",\n" +
"  \"dataNascita\": {\n" +
"    \"year\": 1987,\n" +
"    \"month\": 9,\n" +
"    \"day\": 22\n" +
"  }\n" +
"}";
       Persona p2=gson.fromJson(json: jsonPersona2, classOfT: Persona.class);
        System.out.println("Persona 2:"+p2.toString());

    }
}
```

```
Exception in thread "main" com.google.gson.JsonIOException: Failed making field 'java.time.LocalDate#year' accessible; either in
crease its visibility or write a custom TypeAdapter for its declaring type.
See https://github.com/google/gson/blob/main/Troubleshooting.md#reflection-inaccessible
        at com.google.gson.internal.reflect.ReflectionHelper.makeAccessible(ReflectionHelper.java:76)
        at com.google.gson.internal.bind.ReflectiveTypeAdapterFactory.getBoundFields(ReflectiveTypeAdapterFactory.java:388)
        at com.google.gson.internal.bind.ReflectiveTypeAdapterFactory.create(ReflectiveTypeAdapterFactory.java:161)
        at com.google.gson.Gson.getAdapter(Gson.java:628)
        at com.google.gson.internal.bind.ReflectiveTypeAdapterFactory.createBoundField(ReflectiveTypeAdapterFactory.java:201)
        at com.google.gson.internal.bind.ReflectiveTypeAdapterFactory.getBoundFields(ReflectiveTypeAdapterFactory.java:395)
        at com.google.gson.internal.bind.ReflectiveTypeAdapterFactory.create(ReflectiveTypeAdapterFactory.java:161)
        at com.google.gson.Gson.getAdapter(Gson.java:628)
        at com.google.gson.Gson.toJson(Gson.java:928)
        at com.google.gson.Gson.toJson(Gson.java:899)
        at com.google.gson.Gson.toJson(Gson.java:848)
        at com.google.gson.Gson.toJson(Gson.java:825)
        at com.mycompany. persona json.App.main(App.java:30)
Caused by: java.lang.reflect.InaccessibleObjectException: Unable to make field private final int java.time.LocalDate.year access
ible: module java.base does not "opens java.time" to unnamed module @1517365b
        at java.base/java.lang.reflect.AccessibleObject.throwInaccessibleObjectException(AccessibleObject.java:391)
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(AccessibleObject.java:367)
        at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible(AccessibleObject.java:315)
        at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:183)
        at java.base/java.lang.reflect.Field.setAccessible(Field.java:177)
        at com.google.gson.internal.reflect.ReflectionHelper.makeAccessible(ReflectionHelper.java:68)
        ... 12 more
```

Per permettere a Gson di accedere agli attributi (Year, Month, Day) della classe LocalDate è necessario realizzare un'estensione della classe TypeAdapter che fa parte della libreria Gson. Il nostro TypeAdapter esteso dovrà poi essere "registrato" con una opportuna istruzione, sulla libreria Gson. Per la classe TypeAdapter vanno definiti due metodi astratti chiamati "read" e "write". Un esempio di ciò che si deve fare lo si può trovare qui: https://www.tutorialspoint.com/gson/gson_custom_adapters.htm.

Adattando l'esempio mostrato nel link, al nostro caso, realizziamo la classe LocalDateAdapter nel seguente modo passando come parametro <LocalDate>




```
19  public class LocalDateAdapter extends TypeAdapter<LocalDate>
20  {
21      @Override
22      public LocalDate read(JsonReader reader) throws IOException
23      {
24          LocalDate data=LocalDate.of(year: 0, month: 1, dayOfMonth: 1);
25          reader.beginObject();
26          String fieldname = null;
27
28          while (reader.hasNext()) {
29              JsonToken token = reader.peek();
30
31              if (token.equals(other: JsonToken.NAME)) {
32                  //get the current token
33                  fieldname = reader.nextName();
34              }
35              if ("year".equals(anObject: fieldname)) {
36                  //move to next token
37                  token = reader.peek();
38                  data=data.withYear(year: reader.nextInt());
39              }
40              if ("month".equals(anObject: fieldname)) {
41                  //move to next token
42                  token = reader.peek();
43                  data=data.withMonth(month: reader.nextInt());
44              }
45              if ("day".equals(anObject: fieldname)) {
46                  //move to next token
47                  token = reader.peek();
48                  data=data.withDayOfMonth(dayOfMonth: reader.nextInt());
49              }
50          }
51          reader.endObject();
52          return data;
53      }
54      @Override
55      public void write(JsonWriter writer, LocalDate data) throws IOException
56      {
57          writer.beginObject();
58          writer.name(name: "year");
59          writer.value(value: data.getYear());
60          writer.name(name: "month");
61          writer.value(value: data.getMonthValue());
62          writer.name(name: "day");
63          writer.value(value: data.getDayOfMonth());
64          writer.endObject();
65
66      }
67  }
```

Modificando la classe App con l'istruzione che consente di "dire" a Gson di utilizzare questa classe per accedere agli attributi della classe LocalDate, la serializzazione/deserializzazione ora funziona:

```
19  public class App
20  {
21      public static void main(String[] args) throws IOException
22      {
23          //serializzazione
24          LocalDate dataNascita=LocalDate.of(year: 2000,month: 10,dayOfMonth: 20);
25
26          Persona p1=new Persona(cognome: "Pinna",nome: "Luciano",d: dataNascita);
27          Gson gson=new GsonBuilder().setPrettyPrinting().registerTypeAdapter(type: LocalDate.class,new  LocalDateAdapter()).create();
28          String jsonPersona=gson.toJson(src:p1);
29          System.out.println(x: jsonPersona);
30
31           //deserializzazione
32          String jsonPersona2="{\n" +
33           "  \"cognome\": \"Albrici\",\n" +
34           "  \"nome\": \"Giulio\",\n" +
35           "  \"dataNascita\": {\n" +
36           "    \"year\": 1987,\n" +
37           "    \"month\": 9,\n" +
38           "    \"day\": 22\n" +
39           "  }\n" +
40           "}";
41          Persona p2=gson.fromJson(json: jsonPersona2, classOfT: Persona.class);
42          System.out.println("Persona 2:"+p2.toString());
43      }
44  }
```

Dice a Gson di accedere agli attributi di LocalDate tramite la classe LocalDateAdapter appena creata.

L'output infatti è il seguente:

Output serializzazione

```
{
  "cognome": "Pinna",
  "nome": "Luciano",
  "dataNascita": {
    "year": 2000,
    "month": 10,
    "day": 20
  }
}
Persona 2:Persona{cognome=Albrici, nome=Giulio, dataNascita=1987-09-22}
```

Output deserializzazione