Reti e Laboratorio III
Modulo Laboratorio III
A.A. 2025-2026

docente: Laura Ricci

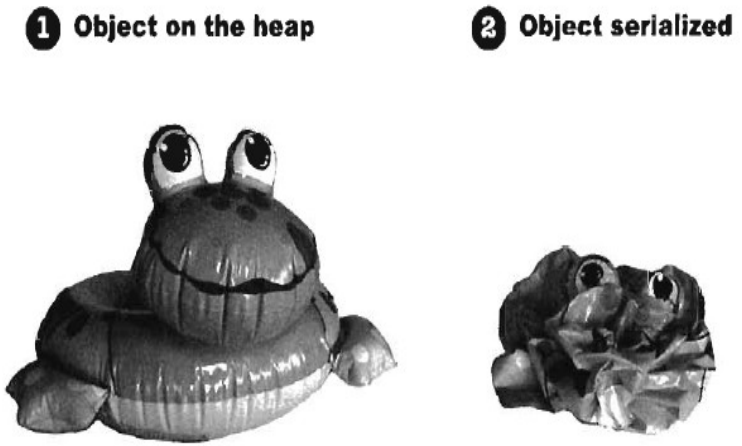
laura.ricci@unipi.it

Lezione 10

Serializzazione, JSON
la libreria GSON
21/11/2025

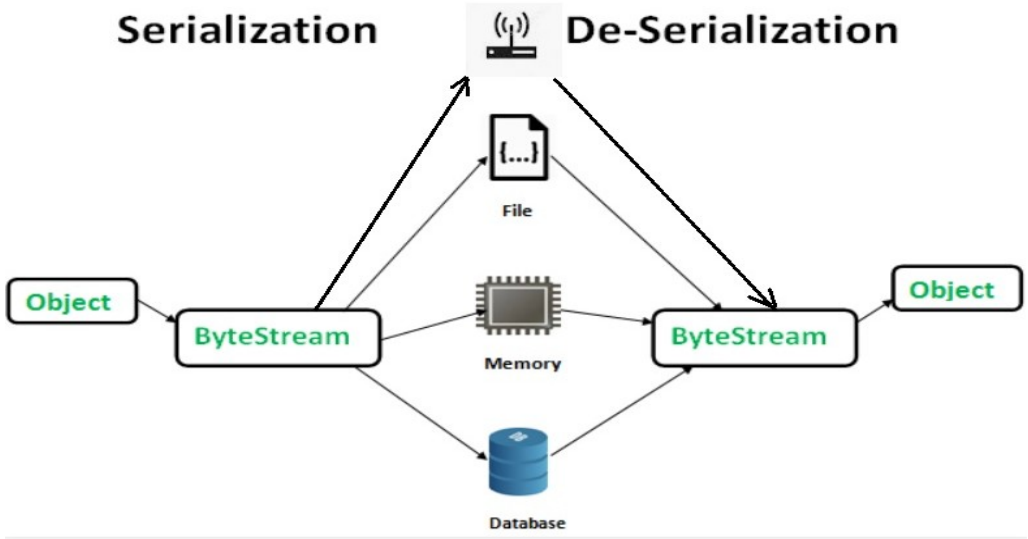
SCRIVERE/LEGGERE OGGETTI DA STREAM

- gli oggetti esistono in memoria fino a che la `JVM` è in esecuzione: per la loro persistenza al di fuori della `JVM`, occorre
  - creare una rappresentazione dell' oggetto indipendente dalla `JVM` usando meccanismi di serializzazione
- ogni oggetto è caratterizzato da uno stato e da un comportamento
  - comportamento: specificato dai metodi della classe
  - stato: "vive" con l'istanza dell'oggetto
  - la serializzazione effettua il flattening dello stato dell'oggetto
  - la deserializzazione ricostruisce lo stato dell'oggetto



PERSISTANZA ED INVIO DI OGGETTI

- l'oggetto serializzato può quindi essere scritto su un qualsiasi stream di output



- come useremo la serializzazione in questo corso?
  - per inviare oggetti su uno stream che rappresenta una connessione TCP
  - per generare pacchetti UDP, si scrive l'oggetto serializzato su uno stream di byte e poi si genera un pacchetto UDP

SERIALIZZAZIONE: INTEROPERABILITA'

- caratteristica auspicabile di un formato di serializzazione
  - non vincolare chi scrive e chi legge ad usare lo stesso linguaggio
- la portabilità può limitare le potenzialità della rappresentazione: una rappresentazione che corrisponde all'intersezione di tutti i vari linguaggi
- principali formati per la serializzazione dei dati che consentono l'interoperabilità tra linguaggi/macchine diverse
  - `XML`
  - `JSON-JavaScript Object Notation`
- XML linguaggio di markup che utilizza tag, simile a HTML
  - più verboso e pesante ripetto a `JSON`
  - maggiore occupazione di memoria

JAVASCRIPT OBJECT NOTATION (JSON)

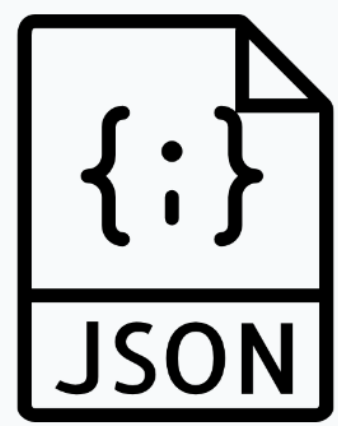
- JSON (JavaScript object notation): formato nativo di Javascript
- espresso con una sintassi molto semplice, facilmente parsabile
- consente di scambiare dati in un formato semplice e human-oriented
- molto utilizzato, anche a livello REST, per accedere a dati mediante API che un server mette a disposizione
- basato su 2 strutture di base:
  - oggetto: insieme di coppie (chiave: valore)
  - liste ordinate di valori
  - + composizione ricorsiva di oggetti e liste
- la composizione ricorsiva genera un documento JSON che ha una struttura ad albero

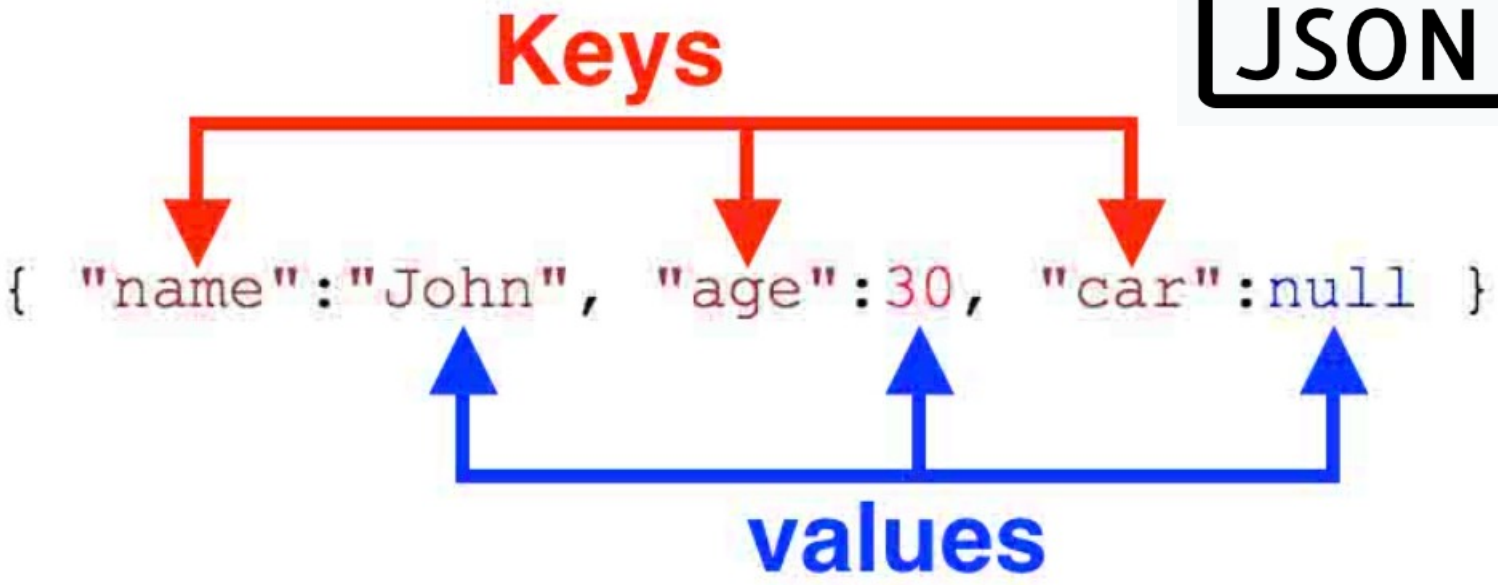
JSON

- coppie (chiave: valore)
  - le chiavi devono esser stringhe `{"name":"John"}`
- i tipi di dato ammissibili per i valori sono:
  - `String`
  - `Number` (`int` o `float`)
  - `object` (`JSON` object, la struttura può essere ricorsiva)
  - `Array`
  - `Boolean`
  - `null`

JSON OBJECT

- una serie non ordinata di coppie *(nome, valore)*
- delimitato da parentesi graffe
- le coppie sono separate da virgole





JSON ARRAY

- una raccolta ordinata di valori

```
["Ford", "BMW", "Fiat"]
```

- delimitato da parentesi quadre e i valori sono separati da virgola
  - un valore può essere di tipo `string`, un numero, un `boolean`, un oggetto `JSON` o un array
  - queste strutture possono essere annidate
- mapping diretto con alcuni tipi di dato Java, array, list, vector, ...

JSON: STRUTTURA RICORSIVA

JSON E XML: CONFRONTO

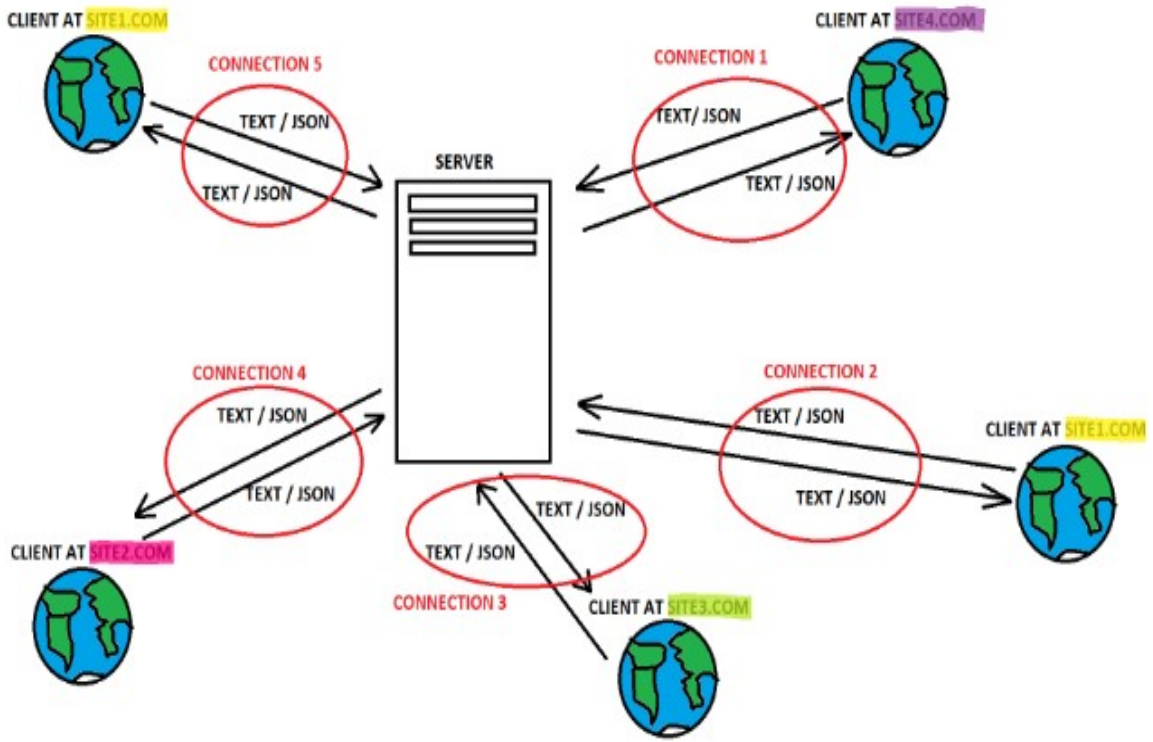
JSON Example

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

XML Example

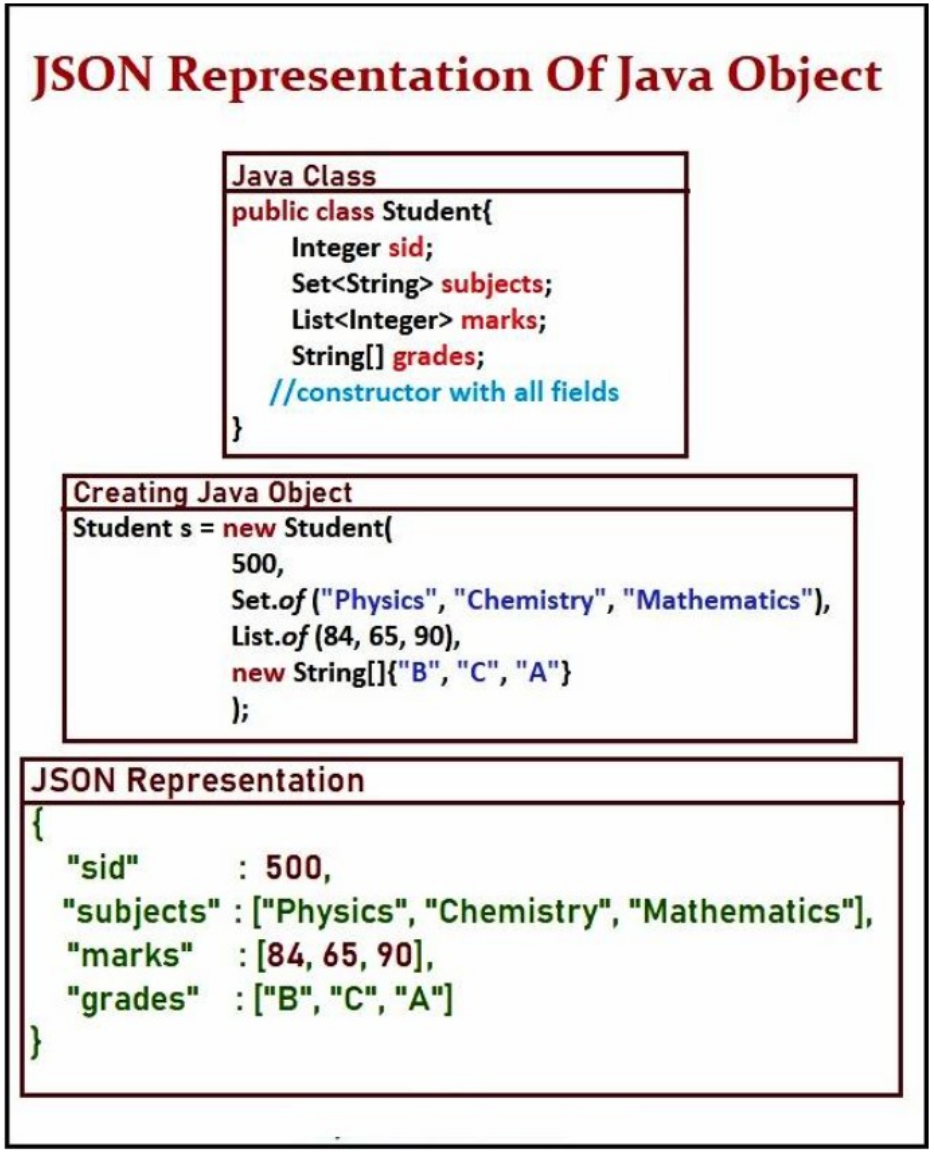
```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

JSON: INTERAZIONE CLIENT SERVER



- `JSON` è in genere il formato dei dati scambiati tra client e server
- ma cosa accade se client/server sono codificati in `Java`?
- necessaria trasformazione `Java/JSON` e viceversa
- diverse librerie proposte

DA JAVA A JSON



- quali librerie per la traduzione?
- GSON
- JSON-Simple
  - leggera e semplice, ma… scarsa documentazione
- JACKSON
- FastJSON
- …

GOOGLE GSON

- una libreria open-source basata su `Java`, sviluppata da `Google` per la serializzazione degli oggetti `Java` in `JSON` e viceversa
- trasforma facilmente oggetti `Java` in `JSON` e viceversa, senza richiedere configurazioni complicate
- gestisce liste, mappe, generics e strutture annidate
- configurabile tramite `GsonBuilder`
- esclusione selettiva dei campi tramite annotazioni o strategie di esclusione personalizzate
- formattazione `JSON` "pretty printing": per generare un `JSON` leggibile per debugging o logging.
- flessibilità: diverse modalità di utilizzo tramite diverse `API`

API PER LA SERIALIZZARE E DESERIALIZZARE

- `data binding API`
  - converte i dati da `JSON` a `Java` e viceversa usando dei "data type adapters"
  - usa reflection
  - supporta anche adapters per tipi generici
- `tree model API:`
  - crea in memoria una rappresentazione dell'oggetto `JSON` mediante un albero (simile a `XML DOM` parser)
  - introduce il nuovo tipo di dato `JsonObject` per rappresentare elementi `JSON:` trasforma elementi `JSON` in strutture `Java`
- `streaming API`
  - tokenizza il documento `JSON:` legge `JsonTokens`
  - buona performance nelle operazioni di lettura/scrittura

LA CLASSE GSON

- classe base: `Gson`
- due modi diversi per creare un'istanza della classe `Gson`, che effettua le traduzioni
- il più semplice `Gson gson = new Gson()`

  crea un oggetto `Gson` settando alcune configurazioni di default (setta tutti i campi a `null`, usa una data di default per oggetti `Date`,…)
- quello che consente di effettuare l'overriding della configurazione di default, settando diversi paramentri

```
Gson gson = new GsonBuilder().create()
```

  - serializzazione ad hoc degli oggetti `Date`
  - definizione di strategie per escludere alcuni campi dalla serializzazione
  - specifiche politiche di serializzazione
  - version control
  - prettyprinting

DATA BINDING API

- serializzazione Java → JSON: toJson()

```
String json = gson.toJson(user);
```

- deserializzazione JSON → Java: fromJson()

```
Gson gson = new Gson();
User user = gson.fromJson(jsonString, User.class);
```

- esegue la traduzione in modo automatico nella maggior parte dei casi, mappando i campi JSON sui campi dell'oggetto Java tramite reflection.
- converte JSON da e verso POJO (Plain Old Java Object)
  - classi Java "semplici"
  - ha solo campi (variabili)
  - metodi getter e setter (non sempre obbligatori)
  - non estende classi particolari
  - non implementa interfacce obbligatorie

DATA BINDING API: SERIALIZZAZIONE

```
public class Person
      { private String name;
        private int age;
        private transient String fiscalID;
        Person(String name, int age, String fiscalID)
            { this.name = name;
              this.age = age;
              this.fiscalID=fiscalID;}
         }

import com.google.gson.*;
public class ToGSON
    { public static void main(String[] args)
    { Person p = new Person("Alice", 59,"XC5F");
      Gson gson = new Gson();
serializzazione
      String json = gson.toJson(p);
      System.out.println(json);
       }}
```

```
$java ToGSON
{"name":"Alice","age":59}
```

SERIALIZZAZIONE: FORMATTARE L'OUTPUT

```
public class Person
     { private String name;
       private int age;
       private transient String fiscalID;
       Person(String name, int age, String fiscalID)
            { this.name = name;
              this.age = age;
              this.fiscalID=fiscalID;}
  public class ToGSON
    public static void main(String[] args)
       {
         Person p = new Person("Alice", 59);
         Gson gson = new GsonBuilder()
                        .setPrettyPrinting()
                        .create();
         String json = gson.toJson(p);
         System.out.println.println(json);
        }}
```

```
$java ToGSON
{
  "name": "Alice",
  "age": 59
}
```

SERIALIZZARE OGGETTI: OSSERVAZIONI

- se un campo è marcato come `transient`, non viene incluso nell'oggetto serializzato: importante per dati sensibili
- non si possono serializzare oggetti con riferimenti circolari, altrimenti si otterrebbe una ricorsione infinita
- tutti i campi della classe corrente ed ereditati dalle superclassi vengono serializzati
- l'invocazione del metodo `toJson` provoca l'invocazione del metodo `obj.getClass()` per determinare i campi che occorre serializzare
- esempio di uso del metodo `getclass()`

```
import java.util.*;
public class Test {
 public static void main(String[] args) throws ClassNotFoundException
     { Test1 t= new Test1();
       Class myClass = t.getClass();
       System.out.println("Class represented by myClass: " + myClass.toString());
              System.out.println("Fields of myClass: "+
                                    Arrays.toString(myClass.getFields()));  }
}
```

JAVA REFLECTIONS

- funzionalità per cui è possibile scrivere codice di un linguaggio la cui funzione è analizzare codice dello stesso linguaggio
- "il codice riflette su sè stesso"
- perchè le reflection?
  - tutti gli strumenti collegati al codice `Java` sono scritti essi stessi in `Java`
  - componenti `Java` che hanno bisogno di manipolare altri componenti `Java` es: compilatore, caricatore ("classloader")
- idea base
  - ad ogni classe `Java` corrisponde un oggetto della classe `java.lang.Class` attraverso i metodi della classe è possibile analizzare tutte le caratteristiche della classe
  - `java.lang.Class` una "metaclasse", ovvero una classe le cui istanze (oggetti) rappresentano altre classi

DATA BINDING API: DESERIALIZZAZIONE

```
import com.google.gson.Gson;
public class GsonFromJson {
  class User {
    private final String firstName;
    private final String lastName;
    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName; }
    public String toString() {
        return new StringBuilder().append("User{").append("First name: ")
                .append(firstName).append(", Last name: ")
                .append(lastName).append("}").toString(); }
                        }
```

DATA BINDING API: DESERIALIZZAZIONE

```
public static void main(String[] args) {
   String json_string =  "{\"firstName\":\"Laura\", \"lastName\": \"Ricci\"}";
   Gson gson = new Gson();
   User user = gson.fromJson(json_string, User.class);
   System.out.println(user);
    } }
```

Output:

User{First name: Laura, Last name: Ricci}

DESERIALIZZAZIONE: IMPLEMENTAZIONE

- legge il `JSON`
- lo confronta con la classe target (es. `User.class`)
- usa la reflection per trovare i campi corrispondenti:
  - nome del campo
  - tipo (`String`, `int`, oggetti, liste, ecc.)
- converte i valori del `JSON` nei valori `Java`
- crea l'oggetto `Java` e assegna i campi, corrispondenza di default per nome
- impostazione valori di default
  - oggetti → `null`
  - numeri → `0`
  - boolean → `false`
- ignora i campi del `JSON` non presenti nella classe `Java`

SERIALIZZARE OGGETTI ANNIDATI

```
import java.util.*;
public class RestaurantWithMenu {
    String name;
    List<RestaurantMenuItem> menu;
     public RestaurantWithMenu (String name, List<RestaurantMenuItem> menu )
       {this.name=name;
        this.menu= menu;
       }}
import java.util.*;
public class RestaurantMenuItem {
    String description;
    float price;
    public RestaurantMenuItem (String description, float price)
        {this.description=description;
         this.price= price;        }
     public String toString() {return description+price;}}
```

SERIALIZZARE OGGETTI ANNIDATI

```
import java.util.*;
import com.google.gson.*;
public class Restaurants {
public static void main (String args[])
    { List<RestaurantMenuItem> menu = new ArrayList<>();
      menu.add(new RestaurantMenuItem("Spaghetti", 9.99f));
      menu.add(new RestaurantMenuItem("Steak", 14.99f));
      menu.add(new RestaurantMenuItem("Salad", 6.99f));
      RestaurantWithMenu restaurant =
                        new RestaurantWithMenu("AllWhatYouCanEat", menu);
      Gson gson = new GsonBuilder()
            .setPrettyPrinting()
            .create();
      String restaurantJson= gson.toJson(restaurant);
      System.out.println(restaurantJson);}}
```

SERIALIZZARE OGGETTI ANNIDATI

```
{
  "name": "AllWhatYouCanEat",
  "menu": [
    {
      "description": "Spaghetti",
      "price": 9.99
    },
    {
      "description": "Steak",
      "price": 14.99
    },
    {
      "description": "Salad",
      "price": 6.99
    }
  ]
}
```

SERIALIZZARE OGGETTI ANNIDATI

```
import java.util.*;
import com.google.gson.Gson; import com.google.gson.GsonBuilder;
enum Degree_Type { TRIENNALE, MAGISTRALE}
public class Student {
    private String firstName;
    private String lastName;
    private int studentID;
    private String email;
    private List<String> courses;
    private Degree_Type Dg;
    public Student(String FName, String LName, int SID, String email,
                List<String> Clist, Degree_Type DG )
        {this.lastName=LName; this.lastName=LName; this.studentID=SID;
         this.email= email; this.courses=Clist; this.Dg=DG;};
    public String toString()
        { return "name:"+firstName+"  surname:"+lastName+"  ID:"+studentID+"
                email:"+email+"  corsi:"+courses+"  Degree:"+Dg;}}
        // Metodi getter e setter
```

SERIALIZZARE COMPOSIZIONE DI OGGETTI

```
public static void main (String args[])
    {List <String> ComputerScienceCourses = Arrays.asList("Reti", "Architetture");
     List <String> MathCourses = Arrays.asList("Analisi", "Statistica");
     // Instantiating students
     Student max = new Student("Mario", "Rossi", 1254, "mario.rossi@uni1.it",
                       ComputerScienceCourses, Degree_Type.TRIENNALE);
     Student amy = new Student("Anna", "Bianchi", 1328, "anna.bainchi@uni1.it",
                       MathCourses, Degree_Type.MAGISTRALE);
     // Instantiating Gson
     Gson gson = new GsonBuilder()
                 .setPrettyPrinting()
                 .create();
     // Converting JAVA to JSON
     String marioJson = gson.toJson(mario);
     String annaJson = gson.toJson(anna);
     System.out.println(marioJson);
     System.out.println(annaJson);}}
```

```
$java Student
{
  "lastName": "Rossi",
  "studentID": 1254,
  "email": "mario.rossi@uni1.it",
  "courses": [
    "Reti",
    "Architetture"
  ],
  "Dg": "TRIENNALE"
}
{
  "lastName": "Bianchi",
  "studentID": 1328,
  "email": "anna.bainchi@uni1.it",
  "courses": [
    "Analisi",
    "Statistica"
  ],
   "Dg": "MAGISTRALE"}
```

DESERIALIZZAZIONE CON DATA BINDING

- in generale l'idea della deserializzazione con data binding è quella di prendere l'oggetto JSON e tradurlo in un "custom object" di JAVA, mediante le funzionalità offerte dall'API data binding
- la struttura dell'oggetto JAVA dovrebbe "rispecchiare" il contenuto del file JSON
- semplice quando il mapping è diretto, come nell'esempio precedente, ma...
- talvolta il formato del file JSON può essere molto diverso rispetto all'oggetto che si vuole costruire, quindi la corrispondenza può essere complicata da costruire
  - occorre strutturare il mapping degli oggetti del file JSON alle classi/sottoclassi JAVA
  - nel caso generale, necessario utilizzare il meccanismo delle reflection

DESERIALIZZAZIONE COLLEZIONI E GENERICI

- in Java, i tipi generici esistono solo a livello di compilazione: a runtime, la JVM cancella le informazioni sui parametri di tipo
- per liste o mappe occorre usare un TypeToken che conservi tutte le informazioni sul tipo

```
Type listType = new TypeToken<List<User>>(){}.getType();

List<User> users = gson.fromJson(jsonString, listType);
```

- per deserializzare il seguente JSON e memorizzarlo in una lista di User

```
[
   { "name": "Alice", "age": 25 },
   { "name": "Bob", "age": 30 }
]
```

```
Type listType = new TypeToken<List<User>>() {}.getType();

List<User> users = gson.fromJson(json, listType);
```

GSON: TREE MODEL API

- costruisce una rappresentazione ad albero del documento JSON direttamente in memoria
- JsonObject : nodi dell'albero
- analogo al parser DOM usato per XML
- approccio più flessibile perché non richiede di conoscere la struttura del JSON in anticipo per effettuare la conversione in oggetto JAVA
- utilizza il parser: JsonParser
  - fornisce un puntatore al nodo radice dell'albero dopo aver letto il JSON.
  - il nodo radice può essere utilizzato per attraversare l'intero albero
    - si può navigare liberamente, aggiungere, modificare o rimuovere nodi

TREE MODEL API: CLASSI PRINCIPALI

Tipo	Descrizione
`JsonElement`	Nodo generico dell'albero
`JsonObject`	Oggetto JSON `{...}`
`JsonArray`	Array JSON `[...]`
`JsonPrimitive`	Valore semplice (String, Number, Boolean)
`JsonNull`	Null JSON

GSON: TREE MODEL

```
import com.google.gson.*;
public class GSONPrimTypes {
    public static void main (String args[])
     {
     String jsonString =
        "{\"name\":\"Mario Rossi\", \"age\":21,\"verified\":false,\"marks\":
           [30,28,25]}";
        //crea l'albero da JSON
        JsonElement rootNode = JsonParser.parseString(jsonString);
        JsonObject details = rootNode.getAsJsonObject();
        JsonElement nameNode = details.get("name");
        System.out.println("Name: " +nameNode.getAsString());
        JsonElement ageNode = details.get("age");
        System.out.println("Age: " + ageNode.getAsInt());
        JsonElement verifiedNode = details.get("verified");
        System.out.println("Verified: " + (verifiedNode.getAsBoolean() ?
                                                        "Yes":"No"));
```

GSON: TREE MODEL

```
JsonArray marks = details.getAsJsonArray("marks");

for (int i = 0; i < marks.size(); i++) {

   JsonPrimitive value = marks.get(i).getAsJsonPrimitive();

   System.out.print(value.getAsInt() + " ");  }
```

GSON: TREE MODEL

```
{
  "name": "AllWhatYouCanEat",
  "menu": [
    {
      "description": "Spaghetti",
      "price": 9.99

    },
    {

      "description": "Steak",
      "price": 14.99
    },
    {
      "description": "Salad",
      "price": 6.99
    }
  ]
  }
```

- supponiamo di aver già creato il file `restaurant.json` in cui è memorizzata la struttura `JSON` sopra
- nelle slide successive mostriamo come si effettua la deserializzazione con tree-based `API`

GSON: TREE MODEL

```
import com.google.gson.*; import java.io.*; import java.util.*;

public class GSONComplexObject {

public static void main(String[] args) {

   File input = new File("restaurant.json");

   try {

     JsonElement fileElement = JsonParser.parseReader(new FileReader(input));

     JsonObject fileObject = fileElement.getAsJsonObject();

     //extracting basic fields

     String identifier = fileObject.get("name").getAsString();

     System.out.println("name is="+identifier);

     JsonArray jsonArrayOfItems =fileObject.get("menu").getAsJsonArray();

     List <RestaurantMenuItem> menuitems = new ArrayList <RestaurantMenuItem>();
```

GSON: TREE MODEL

```
  for (JsonElement menuElement: jsonArrayOfItems) {
      //Get the JsonObject
      JsonObject itemJsonObject = menuElement.getAsJsonObject();
      String desc= itemJsonObject.get("description").getAsString();
      float price = itemJsonObject.get("price").getAsFloat();
      RestaurantMenuItem restaurantel = new RestaurantMenuItem(desc, price);
      menuitems.add(restaurantel);
      }
  System.out.println("Items are"+menuitems);
  }
catch (FileNotFoundException e) {e.printStackTrace();}
catch (Exception e) {e.printStackTrace();} }}
```

```
Stampa
name is=AllWhatYouCanEat
Items are[Spaghetti 9.99, Steak 14.99, Salad 6.99]
```

CONVERSIONE OGGETTI JSON IN HASHMAP

- consideriamo il seguente oggetto `JsonArray`

```
[
     {"name" : "Mario Rossi", "age": 35},
     {"name": "Anna Bianchi", "age": 41}
]
```

- come si può trasformare il `JsonArray` in una hashmap?
  - `Tree model API`: iterare sui suoi elementi e costruire singolarmente gli elementi della mappa
  - `Data Binding API`: usare il metodo `fromJson`

CONVERSIONE CON DATA BINDING

```
static Map<String,Integer> convertUsingIterative1(JsonArray jsonArray ) {
    Map<String, Integer> hashMap = new HashMap<>();
    Gson gson = new Gson();
    Type listType = new TypeToken< List<Map<String, Integer> >>() {}.getType();
    List<Map<String, Object>> list =  gson.fromJson(jsonArray, listType);
    for (Map<String, Object> entry : list) {
        String type = (String) entry.get("name");
        Integer amount = ((Double) entry.get("age")).intValue();
       // Gson parses numbers as Double
        hashMap.put(type, amount);}
    return hashMap;
    }}
```

CONVERSIONE MEDIANTE TREE-BASED API

```
package JSONHashMap;
import com.google.gson.*;
import java.util.*;
public class JHMAp {
public static void main(String args[])
{   JsonArray ja= new JsonArray();
    JsonObject jsonObject1 = new JsonObject();
    jsonObject1.addProperty("name", "Mario Rossi");
    jsonObject1.addProperty("age", 35);
    ja.add(jsonObject1);
    JsonObject jsonObject2 = new JsonObject();
    jsonObject2.addProperty("name", "Giovanni Bianchi");
    jsonObject2.addProperty("age", 41);
    ja.add(jsonObject2);
    Map<String, Integer> hashMap=JHMAp.convertUsingIterative(ja);
    System.out.println(hashMap.get("Mario Rossi"));
    System.out.println(hashMap.get("Giovanni Bianchi"));}
```

CONVERSIONE MEDIANTE ITERAZIONE

```
static Map<String, Integer> convertUsingIterative(JsonArray jsonArray) {
        Map<String, Integer> hashMap = new HashMap<>();
        for (JsonElement element : jsonArray) {
            JsonObject jsonObject = element.getAsJsonObject();
            String type = jsonObject.get("name").getAsString();
            Integer amount = jsonObject.get("age").getAsInt();
            hashMap.put(type, amount);
         }
        return hashMap;
    }}
```

GSON STREAMING API

- streaming: utile supporto quando si deve lavorare su uno stream di oggetti JSON
- immaginiamo di avere un file JSON di 1.5 G che contiene un insieme di documenti, con i relativi metadati
  - un unico oggetto JSON contenente tutti i documenti?
  - caricare tutto l'oggetto e deserializzarlo con i metodi visti è improponibile, perchè la struttura in RAM avrebbe grosse dimensioni
- GSON streaming offre metodi il caricamento incrementale di parti dell'oggetto
- utile quando
  - l'oggetto ha dimensione troppo grossa
  - non si dispone dell'intero oggetto da deserializzare, perchè ad esempio l'oggetto viene inviato in streaming su una connessione di rete

GSON STREAMING API

- la streaming API di Gson permette di leggere e scrivere JSON in modo sequenziale, token per token, usando due classi principali:
  - JsonReader → lettura streaming (input)
  - JsonWriter → scrittura streaming (output)
- non carica l'intero JSON in memoria
  - ideale per file grandi o stream di rete
  - si evita il carico di memoria degli altri due modelli
- elabora il JSON token per token
  - BEGIN_OBJECT, NAME, STRING, NUMBER, END_OBJECT…
- altissime prestazioni
  - più veloce e leggera del data binding o del tree model
- richiede più codice e gestione manuale
  - controllare la struttura del JSON durante la lettura

GSON STREAMING API

- struttura base: `JsonToken` rappresenta una struttura, un nome o un valore all'interno di una stringa `JSON`
- esistono i seguenti tipi di `JsonToken`:
  - `BEGIN_OBJECT` : legge il token { e passa all'interno dell'oggetto
  - `END_OBJECT` : legge il token } e ritorna al livello superiore
  - `BEGIN_ARRAY` : legge il token [ed entra nel contesto dell'array
  - `END_ARRAY` : legge il token] ed esce dal contesto dell'array
  - `HASNEXT` : ritorna TRUE se ci sono ancora token all'interno dell'oggetto o dell'array
  - `NEXTNAME` : legge il nome della proprità
  - `NEXTSTRING, NEXTINT,..` : leggono i valori della proprietà
  - `PEEK` : legge il prossimo token senza consumarlo
  - SKIPVALUE : salta il valore corrente

GSON STREAMING API: JSONWRITER

```
import com.google.gson.stream.JsonWriter; import java.io.FileWriter; import java.io.IOException;
public class GsonStreamWriter {
     public static void main(String... args){
        JsonWriter writer;
        try { writer = new JsonWriter(new FileWriter("result.json"));
              writer.beginObject();                    //  {
              writer.name("name").value("Steve");      //      "name": "Steve"
              writer.name("surname").value("Jobs");    //      "surname": "Job"
              writer.name("birthyear").value(1955);    //      "birthyear": 1955
              writer.name("skills");                   //      "skills":
              writer.beginArray();                     //      [
              writer.value("JAVA");                    //          "JAVA"
              writer.value("Python");                  //          "Python"
              writer.value("Rust");                    //          "Rust"
              writer.endArray();                       //      ]
              writer.endObject();                      //  }
              writer.close();
        } catch (IOException e) {  System.err.print(e.getMessage());}}}
```

GSON STREAMING API: JSONREADER

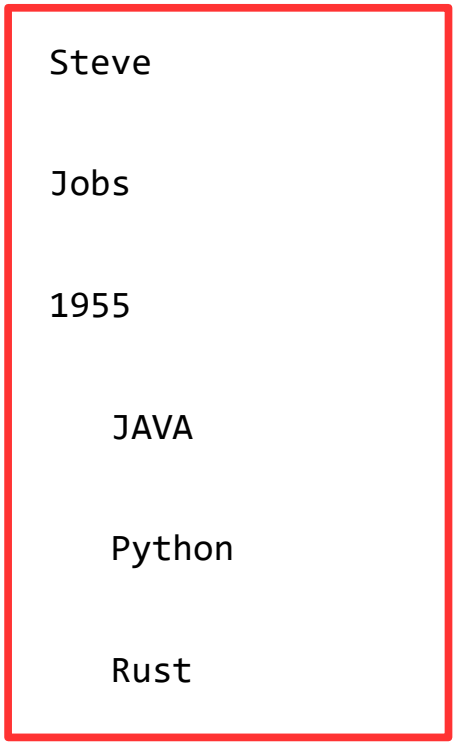
```
import com.google.gson.stream.JsonReader; import java.io.FileNotFoundException;
import java.io.FileReader; import java.io.IOException;
public class GSONStreamReader {
   public static void main(String... args){
      JsonReader reader;
      try {
          reader = new JsonReader(new FileReader("result.json"));
          reader.beginObject();
          while (reader.hasNext()){
              String name = reader.nextName();
              if ("name".equals(name)){
                    System.out.println(reader.nextString());
                } else if ("surname".equals(name)){
                    System.out.println(reader.nextString());
                } else if ("birthyear".equals(name)){
                    System.out.println(reader.nextString());
```

GSON STREAMING API: JSONREADER

```
 } else if ("skills".equals(name))
          { reader.beginArray();
            while (reader.hasNext()){
                System.out.println("\t" + reader.nextString());}
            reader.endArray();
        } else {
            reader.skipValue();
        }
    }
    reader.endObject();
    reader.close();
} catch (FileNotFoundException e) { System.err.print(e.getMessage());
} catch (IOException e) {  System.err.print(e.getMessage());}}}
```

GSON STREAMING API: JSONREADER

Stampa prodotta dal programma



```
Steve
Jobs
1955
    JAVA
    Python
    Rust
```

CONFRONTO TRA LE API

- uso ideale
  - Data Binding
    - JSON stabile → POJO
  - Tree Model
    - JSON variabile/dinamico → struttura ad albero
  - Streaming
    - JSON enorme → parsing veloce senza memoria

GSON IN ECLIPSE

- scaricare il `jar di JSON`
- in `Eclipse`
  - creare una user library per `GSON`
  - aprire il menu **`Windows`**→ **`preferences`**
  - **`Java`** →**`Build path`** → **`User libraries`**→ **`New`**: inserire un nuovo **`User Library Name`** (ad esempio gson_lib).
  - quindi selezionare il nome e cliccare **`Add External JARs`** e reperire il `Jar` scaricato. Applicare e chiudere.
- Inserire la libreria come libreria esterna nel progetto che utilizza `GSON`
  - tasto destro sul nome del progetto → **`JAVA Build Path`** → **`Add External Archives`** selezionare la libreria scaricata

ASSIGNMENT 10

- viene dato un file `JSON` compresso (in formato `GZIP`) contenente i conti correnti di una banca
- ogni conto corrente contiene il nome del correntista ed una lista di movimenti
- per ogni movimento vengono registrati la data e la causale del movimento
- l'insieme delle causali possibili è fissato: `Bonifico, Accredito, Bollettino, F24, PagoBancomat`
- i movimenti registrati per un conto corrente possono essere molto numerosi.
- la struttura del file `JSON` è descritta in un file allegato all'assignment
- progettare un'applicazione che attiva un insieme di thread
- uno di essi legge dal file gli oggetti "conto corrente" e li passa, uno per volta, ai thread presenti in un thread pool
- si vuole trovare, per ogni possibile causale, quanti movimenti hanno quella causale
- i thread cooperano, condividendo una opportuna struttura dati opportunamente sincronizzata, al calcolo dei movimenti per ogni causale
- la lettura dal file deve essere fatta utilizzando l'`API GSON` per lo streaming