Reti e Laboratorio 3
Modulo Laboratorio 3
A.A. 2024-2025

docente: Laura Ricci

laura.ricci@unipi.it

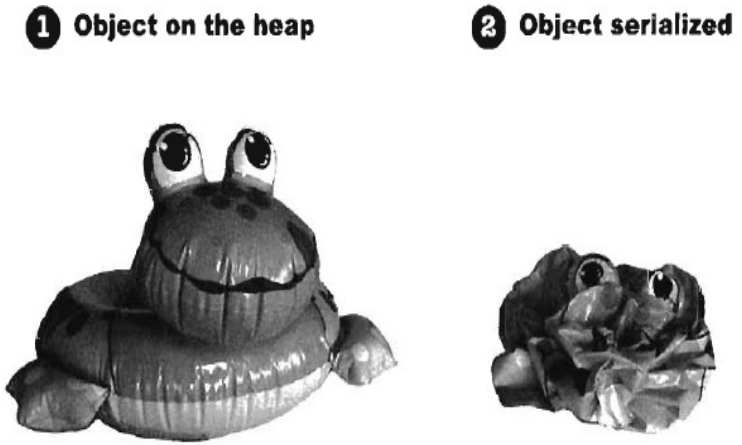
Lezione 7

Serializzazione:
JSON GSON e Java native serialization

28/10/2024

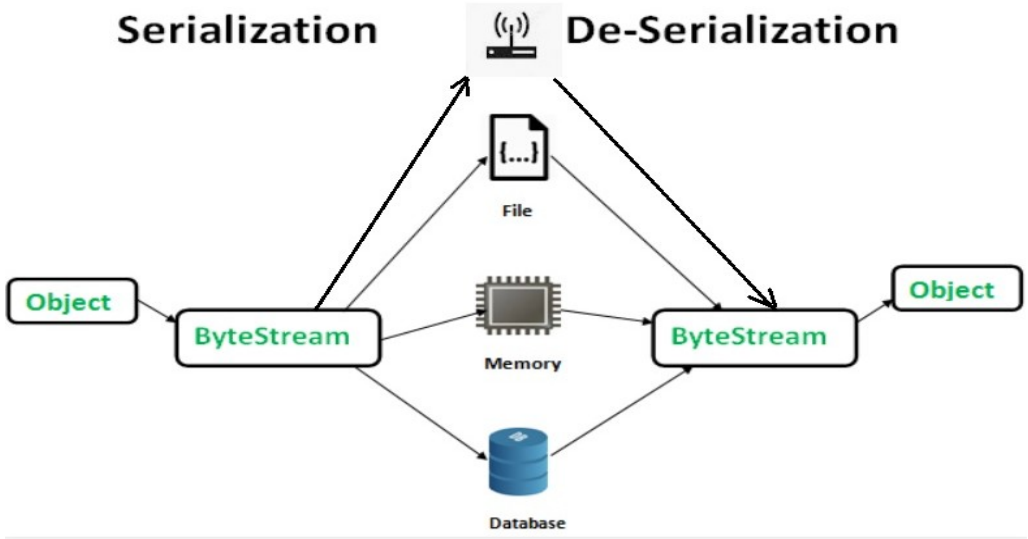
SCRIVERE/LEGGERE OGGETTI DA STREAM

- 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)

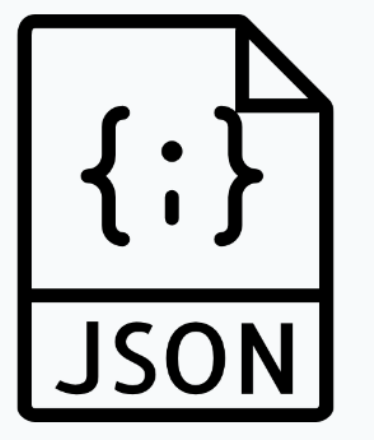
- JSON (JavaScript object notation file): formato nativo di Javascript
- espresso con una sintassi molto semplice, facilmente parsabile
- consente di scambiare dati in un formato semplice e human-oriented
- molto utilizzato, anche a livello REST, per accedere a dati mediante API che un server mette a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

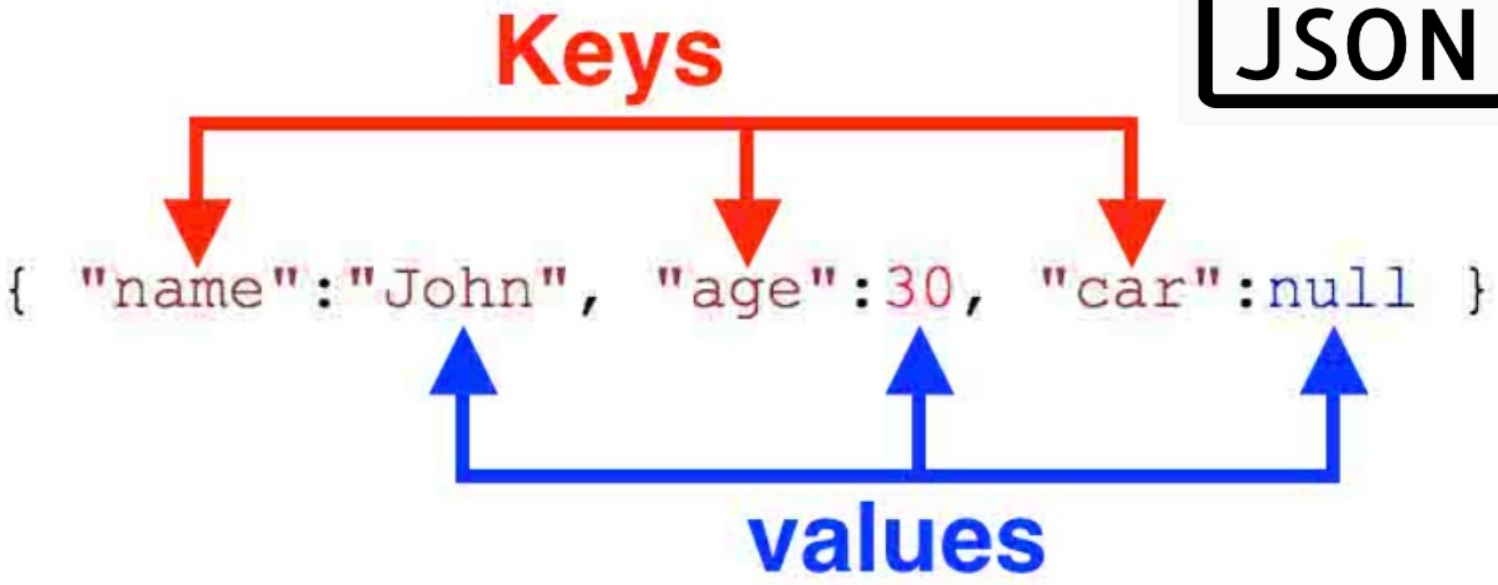
JSON

- 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

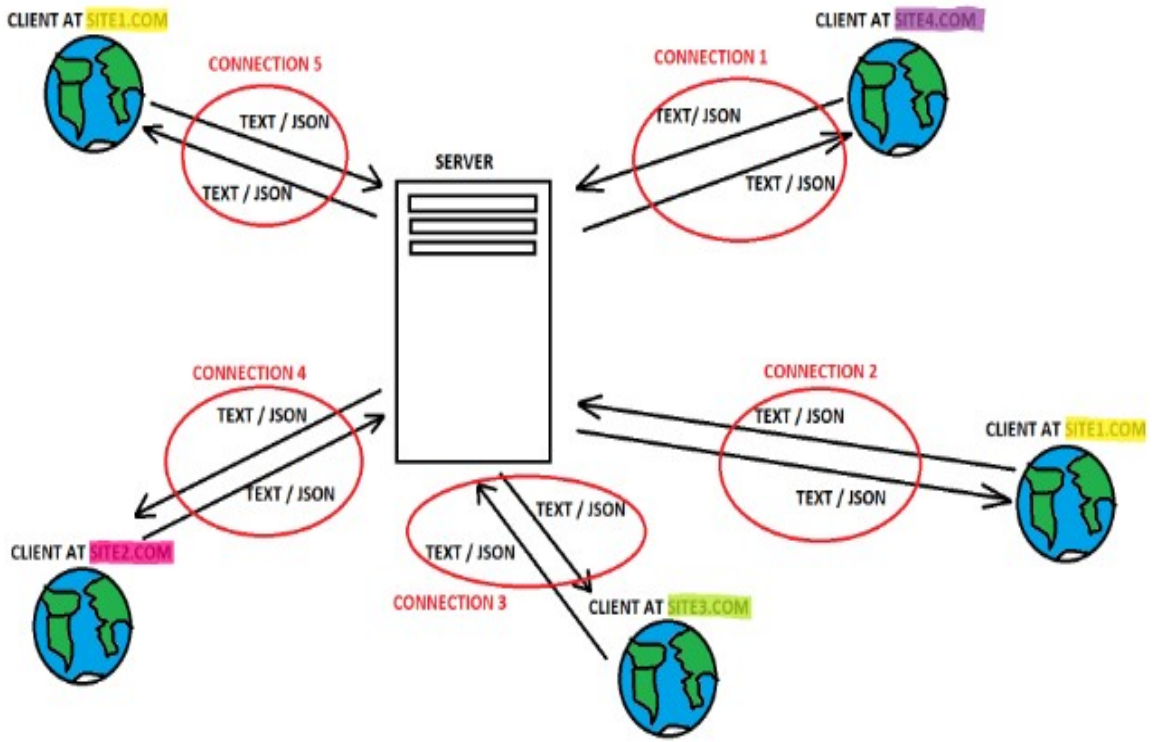
JSON Example

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

XML Example

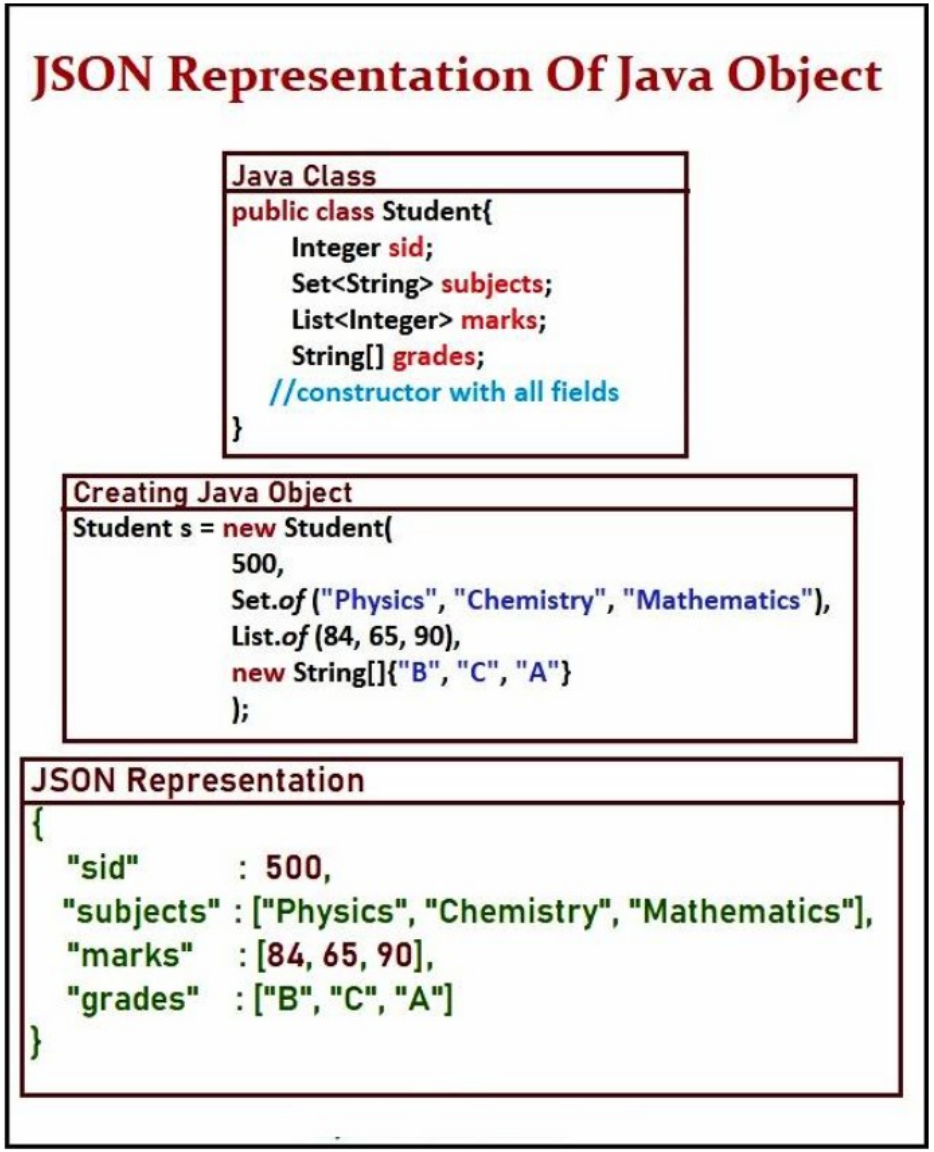
```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>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

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
- …

GSON: GOOGLE JSON

- libreria per serializzare/deserializzare oggetti `Java` in/da `JSON`
- `toJson()` e `fromJson()` metodi per la serializzazione e la deserializzazione
- serializzazione semplice, deserializzazione può richiedere reflection
- supporto per JAVA generics ed oggetti arbitrariamente complessi
- possibile personalizzare la serializzazione
- flessibilità: diverse `API`
  - data binding
  - tree model
  - streaming

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 librerai come libreria esterna nel progetto che utilizza `GSON`
  - tasto destro sul nome del progetto → **`JAVA Build Path`** → **Add `External Archives`** selezionare la libreria scaricata

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

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)
  - albero di `JsonElements`
  - è possibile poi trasformare gli elementi `JSON` in strutture `JAVA`
- `streaming API`
  - tokenizza il documento `JSON`: legge `JsonTokens`
  - buona performance nelle operazioni di lettura/scrittura

SERIALIZZAZIONE DI DATI PRIMITIVI

- *serializzazione con data binding*: dato un oggetto JAVA, restituisce la rappresentazione JSON dell'oggetto

  `toJson(Object src)`

- partiamo dal caso si oggetti semplici

```
package GSONPrimitive;
import com.google.gson.*;
public class GSONPrimTypes {
  public static void main (String args[])
        {
         // Serialization
         Gson gson = new Gson();
         String g1=gson.toJson(1);             // ==> 1
         String g2=gson.toJson("abcd");        // ==> "abcd"
         int[] values = { 1 };
         String g3=gson.toJson(values); }      // ==> [1] }
```

SERIALIZZARE OGGETTI

```
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? Filosofia di base:
  - 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

SERIALIZZARE OGGETTI COMPOSTI

```
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 COMPOSTI

```
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 COMPOSTI

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

SERIALIZZAZIONE OGGETTI COMPOSTI

```
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 ADAPTER

```
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(); }
                        }
```

DESERIALIZZAZIONE: DATA BINDING

```
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 CON DATA BINDING

- in generale l'idea della deserializzazione con data binding è quella di prendere l'oggetto `JSON` e tradurlo in un proprio "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
- alternativa: usare la `tree model API`
- un esempio nelle slide successive

TREE MODEL API

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

creiamo il file `restaurant.json`

in cui memorizziamo la struttura `JSON`

a fianco

nella slide successiva vedremo come

deserializzare questa struttura

TREE MODEL API

```
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>();
```

TREE MODEL API

```
  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 MEDIANTE ITERAZIONE

```
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;
    }}
```

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;
   }}
```

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

- è una API a basso livello che legge e scrive JSON come una sequenza di token discreti
- classi principali `JsonReader` e `JsonWriter`.
- struttura base: `JsonToken` che rappresenta una struttura, un nome o un valore all'interno di una stringa `JSON`
- esistono i seguenti tipi di `JsonToken`:
  - `BEGIN_ARRAY` — opening of a JSON array
  - `END_ARRAY` — closing of a JSON array
  - `BEGIN_OBJECT` — opening of JSON object
  - `END_OBJECT` — closing of JSON object
  - `NAME` — a JSON property name
  - `STRING` — a JSON string
  - `NUMBER` — a JSON number (double, long, or int)
  - `BOOLEAN` — a JSON boolean value
  - `NULL` — a JSON null
  - `END_DOCUMENT` — the end of the JSON stream.

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

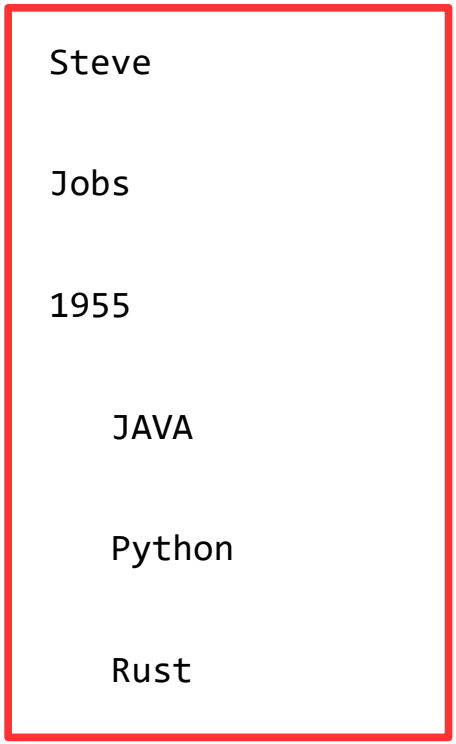
```
import com.google.gson.stream.JsonReader; import java.io.FileNotFoundException;
import java.io.FileReader; import java.io.IOException;
public class GSONStreamReader {
   public static void main(String... args){
      JsonReader reader;
      try {
          reader = new JsonReader(new FileReader("result.json"));
          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



SERIALIZZAZIONE JAVA: HOW TO DO

- `Serializable Interface`
  - per rendere un oggetto "persistente", l'oggetto deve implementare l'interfaccia `Serializable`
  - marker interface: nessun metodo, solo informazione su un oggetto per il compilatore e la `JVM`
  - controllo limitato sul meccanismo di linearizzazione dei dati
  - tutti i tipi di dato primitivi sono serializzabili
  - gli oggetti, se implementano `Serializable`, sono serializzabili
    - a parte alcuni oggetti....(vedi slide successive)
- `Externizable Interface`
  - estende `Serializable`
  - consente creare un proprio protocollo di serializzazione
    - ottimizzare la rappresentazione serializzata dell'oggetto
    - implementazione metodi `readExternal` e `writeExternal`

SERIALIZZAZIONE JAVA: HOW TO DO

```
import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;
public class PersistentTime implements Serializable
       { private static final long serialVersionUID = 1;
         private Date time;
         public PersistentTime()
             {time = Calendar.getInstance().getTime(); }
         public Date getTime()
              {return time; } }
```

in rosso le parti relative alla serializzazione

Regola #1: per serializzare un oggetto persistente la classe di cui l'oggetto è istanza deve implementare l'interfaccia `Serializable` oppure ereditare l'implementazione dalla sua gerarchia di classi

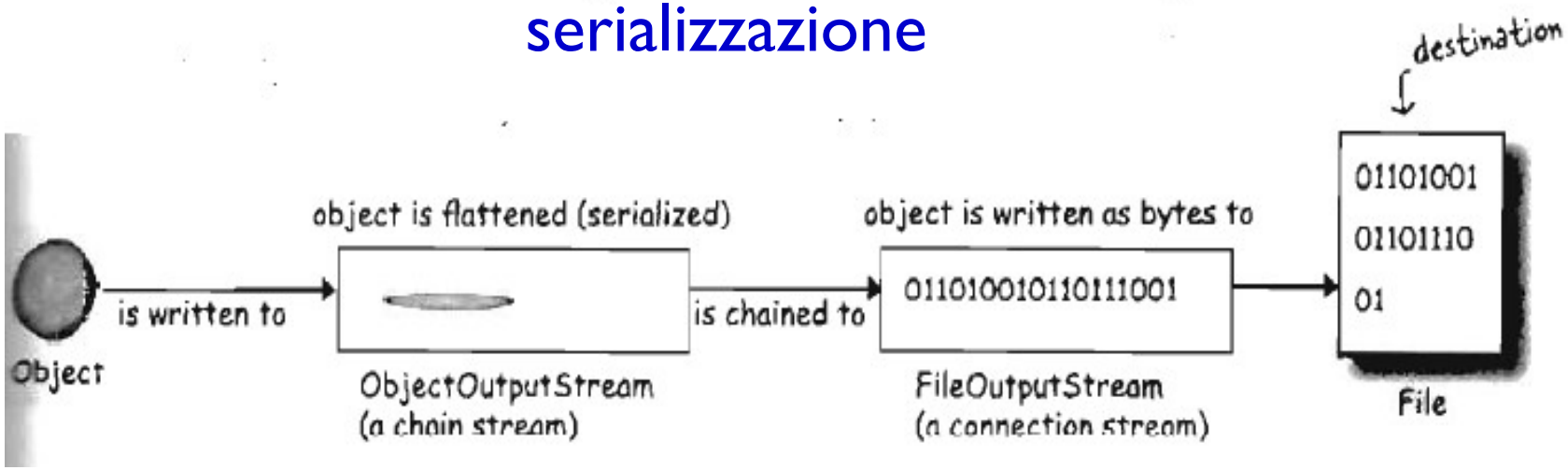
SERIALIZZAZIONE JAVA: HOW TO DO

```
import java.io.*;
public class FlattenTime
   {public static void main(String [] args)
       {String filename = "time.ser";
        if(args.length > 0) { filename = args[0]; }
        PersistentTime time = new PersistentTime();
        try{
           FileOutputStream fos = new FileOutputStream(filename);
           ObjectOutputStream out = new ObjectOutputStream(fos);)
                { out.writeObject(time);}
           catch(IOException ex) {ex.printStackTrace();
         }}}
```

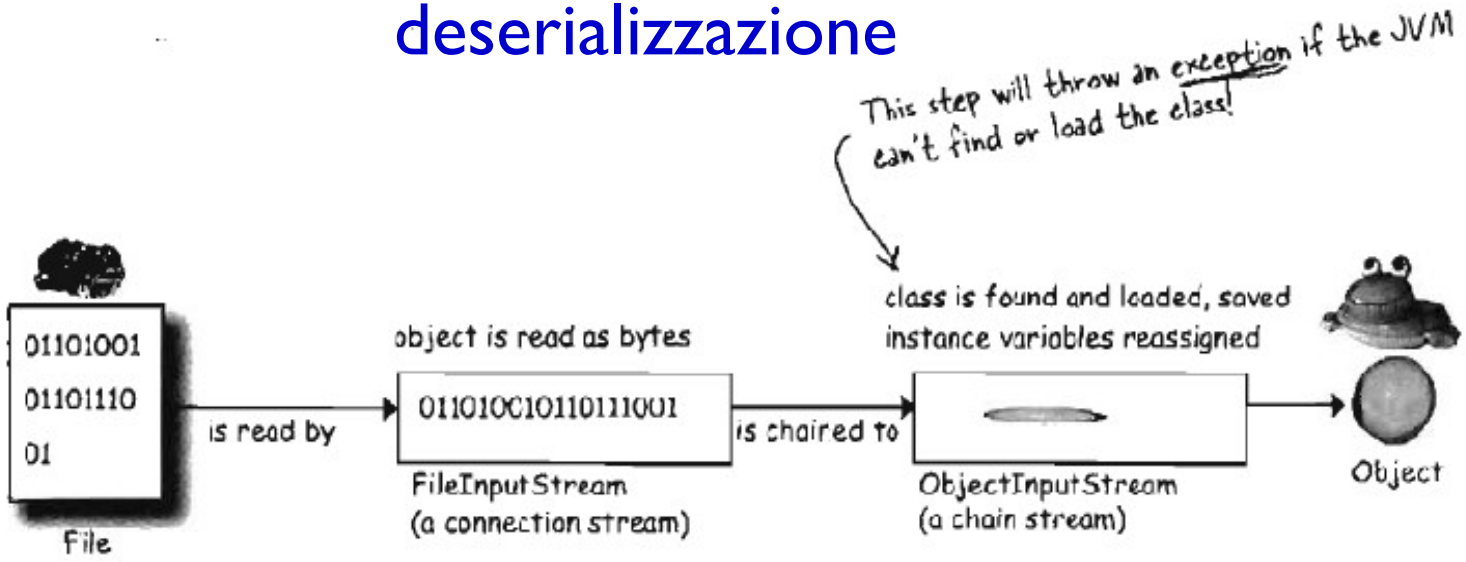
- la serializzazione vera e propria è gestita dalla classe `ObjectOutputStream`
- tale stream deve essere concatenato con uno stream di bytes, che può essere un `FileOutputStream`, uno stream di bytes associato ad un socket, uno stream di byte generato in memoria,...

SERIALIZZAZIONE E DESERIALIAZZAZIONE

serializzazione



deserializzazione



DESERIALIZZAZIONE

```
public class InflateTime
    {public static void main(String [] args)
      {String filename = "time.ser";
       if(args.length > 0)
              {filename = args[0]; }
       PersistentTime time = null; FileInputStream fis = null;
       ObjectInputStream in = null;
       Try {
         FileInputStream fis = new FileInputStream(filename);
         ObjectInputStream in   = new ObjectInputStream(fis);
         time = (PersistentTime)in.readObject();}
  catch(IOException ex)
     { ex.printStackTrace(); }
  catch(ClassNotFoundException ex)
     {ex.printStackTrace();}
```

in rosso le parti relative alla deserializzazione

DESERIALIZZAZIONE

```
// print out restored time
   System.out.println("Flattened time: " + time.getTime());
   System.out.println();
     // print out the current time
   System.out.println("Current time: "+
                          Calendar.getInstance().getTime());}
}
```

Output ottenuto:

```
Flattened time: Mon Mar 12 19:11:55 CET 2012
Current   time: Mon Mar 12 19:16:24 CET 2012
```

`ClassNotFoundException`: l'applicazione tenta di caricare una classe, ma non trova nessuna definizione di una classe con quel nome

DESERIALIZZAZIONE

- il metodo `readObject()` legge la sequenza di bytes memorizzati in precedenza e crea un oggetto che è l'esatta replica di quello originale
  - `readObject` può leggere qualsiasi tipo di oggetto, è necessario effettuare un cast al tipo corretto dell'oggetto
- la JVM determina, mediante informazione memorizzata nell'oggetto serializzato, il tipo della classe dell'oggetto e tenta di caricare quella classe o una classe compatibile
- se non la trova viene sollevata una `ClassNotFoundException` ed il processo di deserializzazione viene abortito
- altrimenti, viene creato un nuovo oggetto sullo heap
  - lo stato di tutti gli oggetti serializzati viene ricostruito cercando i valori nello stream, senza invocare il costruttore (uso di Reflection)
  - si percorre l'albero delle superclassi fino alla prima superclasse non-serializzabile. Per quella classe viene invocato il costruttore

COSA NON E' SERIALIZZABILE?

- oggetti contenenti riferimenti specifici alls `JVM` o al SO (JAVA native class)
  - `Thread`, `OutputStream`, `Socket`, `File`, non possono essere ricreati, perché contengono riferimenti specifici al particolare ambiente di esecuzione
- le variabili marcate come `transient`
  - ad esempio variabili che non devono essere scritte per questioni di privacy, come un numero di carta di credito
- le variabili statiche: sono associate alla classe e non alla specifica istanza dell'oggetto che si sta serializzando
  - lette dalla classe in fase di deserializzazione
- tutti i componenti di un oggetto devono essere serializzabili: se ne esiste uno non serializzabile e non transient si solleva una `notSerializableException`
  - regola #2: per rendere un oggetto persistente occorre marcare tutti i campi che non sono serializzabili come transient

ASSIGNMENT 7

- 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.