

# Reti e Laboratorio III

## Modulo Laboratorio III

### AA. 2023-2024

docente: Laura Ricci

[laura.ricci@unipi.it](mailto:laura.ricci@unipi.it)

## Lezione I I

### Serializzazione:

# JSON e Java native serialization

7/12/2023

# 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

1 Object on the heap

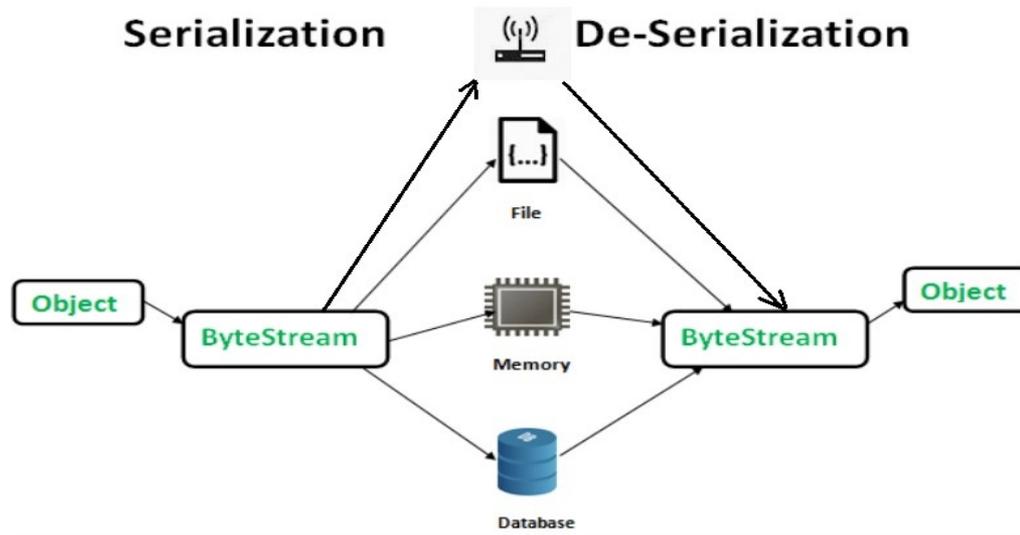


2 Object serialized



# PERSISTENZA ED INVIO DI OGGETTI

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



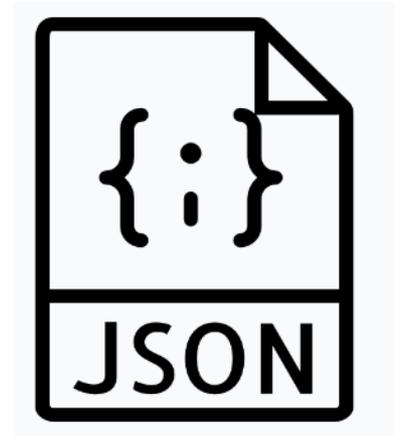
- come useremo la serializzazione in questo corso?
  - per inviare oggetti su uno stream che rappresenta una connessione TCP
  - per generare pacchetti UDP, si scrive l'oggetto serializzato su uno stream di byte e poi si genera un pacchetto UDP

# 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
- formati per la serializzazione dei dati che consentono l'interoperabilità tra linguaggi/macchine diverse
  - XML
  - JSON-JavaScript Object Notation
- JSON: formato nativo di Javascript, ha il vantaggio di essere espresso con una sintassi molto semplice e facilmente riproducibile

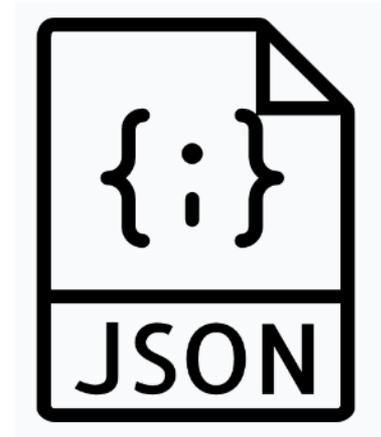
# JAVASCRIPT OBJECT NOTATION (JSON)

- formato lightweight per l'interscambio di dati, indipendente dalla piattaforma poichè è testo, scritto secondo la notazione JSON
  - non dipende dal linguaggio di programmazione
  - “self describing”, semplice da capire e facilmente parsabile
- basato su 2 strutture:
  - coppie (chiave: valore)
  - liste ordinate di valori
- una risorsa JSON ha una struttura ad albero
  - composizione ricorsiva di coppie e liste



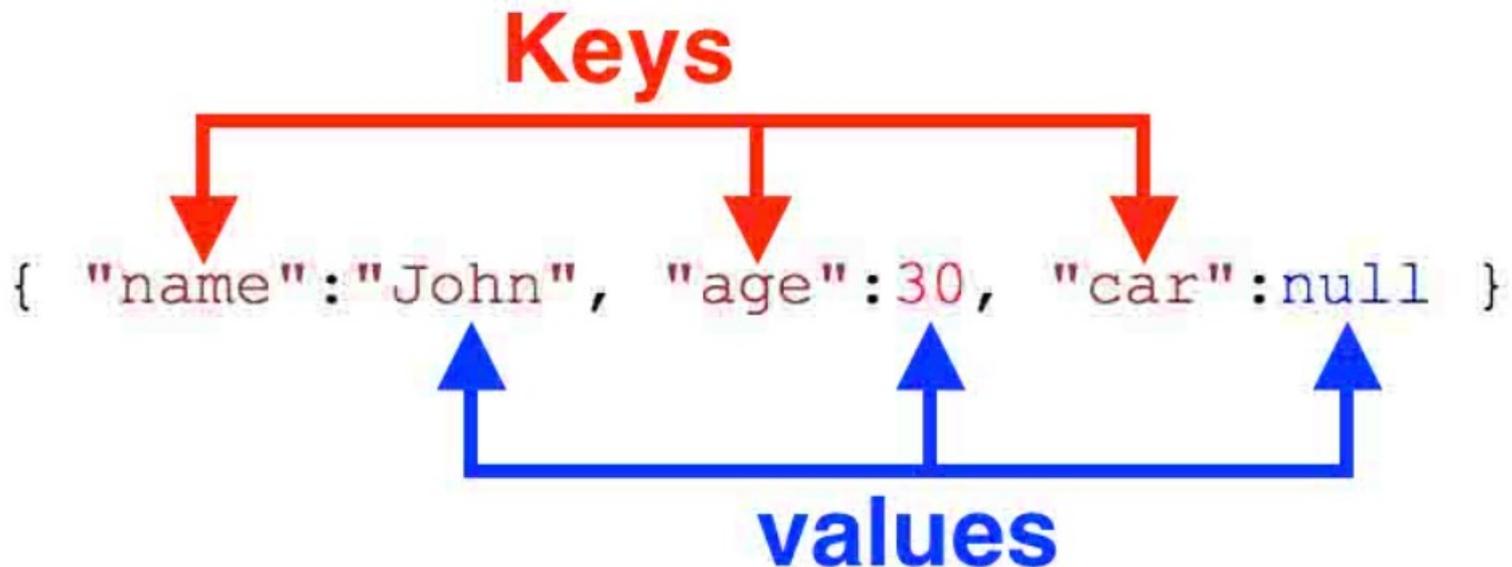
# JAVASCRIPT OBJECT NOTATION

- coppie (chiave: valore)
  - le chiavi devono essere 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 `array`, `list`, `vector`, di `JAVA` etc.

# JSON: STRUTTURA RICORSIVA



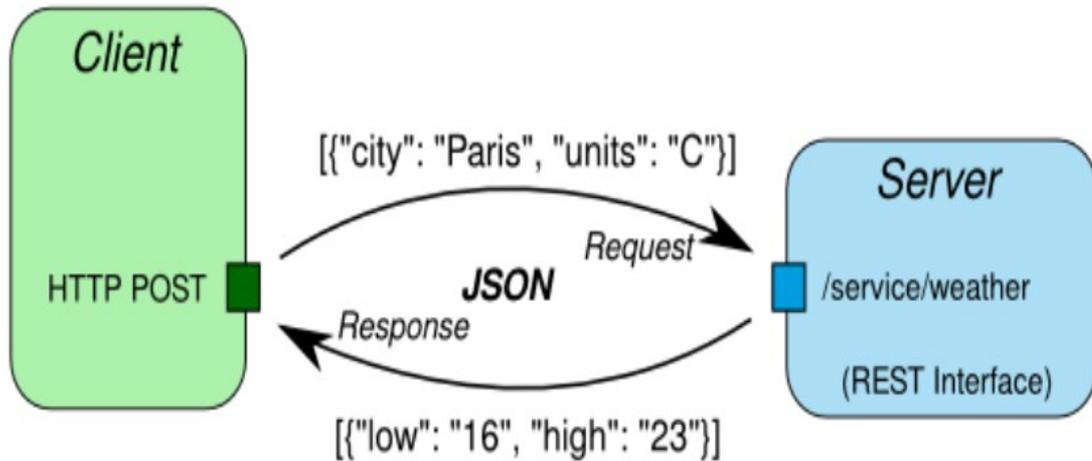
## JSON Example

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

## XML Example

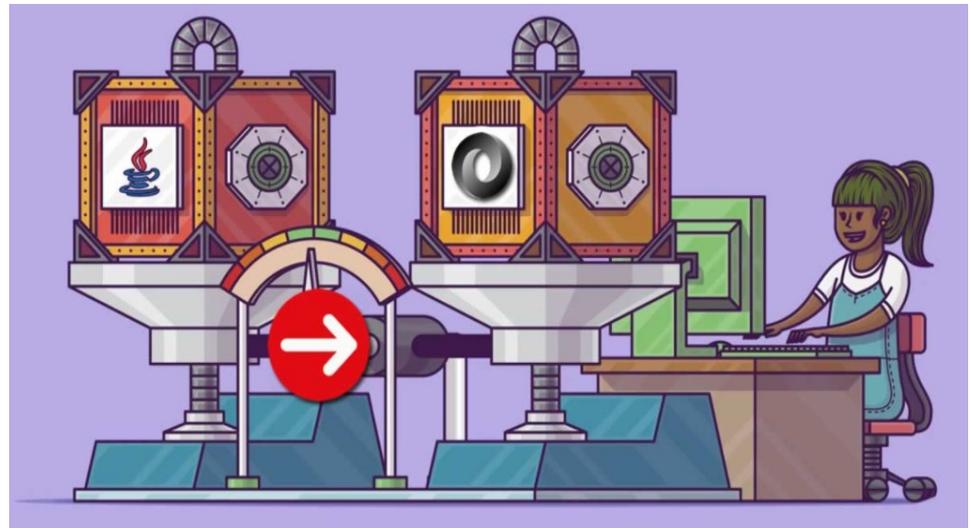
```
<employees>  
  <employee>  
    <firstName>John</firstName> <lastName>Doe</lastName>  
  </employee>  
  <employee>  
    <firstName>Anna</firstName> <lastName>Smith</lastName>  
  </employee>  
  <employee>  
    <firstName>Peter</firstName> <lastName>Jones</lastName>  
  </employee>  
</employees>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

# JSON/REST/HTTP



- client e server interagiscono mediante interfaccia REST
- JSON è in genere il formato dei dati scambiati

- cosa accade se la applicazione è scritta in JAVA?
- necessaria trasformazione JAVA/JSON e viceversa



# DA JAVA A JSON

```
JSON
=====
{
  "id":1,
  "name": SiAm,
  "color": Cream ,
  "breed": Siamese
}
```

JSON string is understood by any program because it's

INTEROPERABLE – program and platform independent

```
Java Obj
=====
```

```
1L
SiAm
Cream
Siamese
```

Representation of ' cat obj'

While it's clear to us that our cat object has: 1L,SiAm, Cream, Siamese

only our java application will understand what these things are.

Our JSON string is understood by every application

Quali librerie per la traduzione?

- GSON
- JACKSON
- JSON-Simple
  - leggera e semplice, ma...  
scarsa documentazione
- FastJSON
- ...

# GSON: GOOGLE GSON

- libreria per serializzare/deserializzare oggetti Java in/da JSON
  - `toJson()` e `fromJson()` semplici metodi per la serializzare e la deserializzazione
  - serializzazione semplice, deserializzazione richiede reflection
  - supporto per JAVA generics ed oggetti arbitrariamente complessi
  - possibile personalizzare la serializzazione
- scaricare JAR ed inserirlo come libreria esterna nel progetto
  - scaricare GSON
  - importare la libreria in Eclipse
  - tasto destro sul nome del progetto → JAVA Build Path → Add External Archives selezionare la libreria scaricata

# SERIALIZAZIONE/DESERIALIZAZIONE CON GSON

- GSON fornisce il supporto per trasformare oggetti JSON in oggetti JAVA e viceversa
  - una classe JAVA con la stessa struttura dell'oggetto JSON
- consideriamo il seguente oggetto JSON e la corrispondente classe JAVA

```
{ "name": "Alice",  
  "age" : 45  
}
```

```
class Person  
{ String name;  
  int age; }
```

- metodi base offerti da GSON per il passaggio da JAVA a JSON sono
  - *serializzazione*: dato un oggetto JAVA, restituisce la rappresentazione JSON dell'oggetto  
`toJson(Object src)`
  - *deserializzazione*: da una stringa in formato JSON ad oggetto JAVA  
`fromJson(String json, Class<T> classOfT)`  
`fromJson(JsonElement json, java.lang.reflect.Type typeOfT)`

# LA CLASSE GSON

- due modi diversi di creare una istanza della classe Gson, che effettua le traduzioni
- il più semplice `Gson gson = new Gson();`

crea un oggetto Gson settando alcune configurazioni di default (omette 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 parametri

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

- serializzazione ad hoc degli oggetti Date
- definizione di strategie per escludere alcuni campi dalla serializzazione
- specifiche politiche di serializzazione,...

- Gson presenta tre tipi di API per la serializzazione/deserializzazione
  - data binding API: converte i dati usando dei “data type adapters”
  - tree model API:
    - crea in memoria una rappresentazione dell’oggetto JSON mediante un albero (simile a XML DOM parser)
    - crea un albero di JsonElements
  - streaming API
    - tokenizza il documento JSON
    - legge JsonTokens con JsonReader e JsonWriter
    - buona performance nelle operazioni di lettura/scrittura

# SERIALIZAZIONE CON DATA BINDING

```
import com.google.gson.Gson;

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

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

serializzazione



```
Gson gson = new Gson();
String json = gson.toJson(p);
```

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

# SERIALIZAZIONE: FORMATTARE L'OUTPUT

```
import com.google.gson.Gson;

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

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 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);}}
```

# SERIALIZAZIONE DELL'OGGETTO

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

# SERIALIZZARE 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"
}
```

# DESERIALIZAZIONE

```
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(); }

    }

}
```

# DESERIALIZAZIONE

```
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}
```

# DESERIALIZAZIONE CON REFLECTION

- la deserializzazione in un oggetto composto richiede in generale informazioni aggiuntive
- occorre indicare, a run time, il tipo (la classe) utilizzata per la deserializzazione
- uso del meccanismo delle Reflection
  - capacità di analizzare ed interagire a run time con le classi
- Gson usa l'informazione sul tipo dell'oggetto JAVA a cui un Json text deve essere mappato
- tuttavia, usando i generici, questa informazione viene persa durante la serializzazione
- necessario l'uso della classe `com.google.gson.reflect.TypeToken` per memorizzare il tipo del generico oggetto

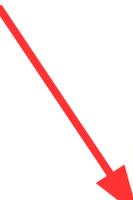
# DESERIALIZAZIONE CON REFLECTION

```
import com.google.gson.*;
import java.lang.reflect.*;
import com.google.gson.reflect.*;

public class RestaurantRefelection {
public static void main(String[] args) {
    try {
        String JsonRestaurant="{\"name\": \"AllWhatYouCanEat\", \"menu\": \"
            + \"[{\"description\": \"Spaghetti\", \"price\": 9.99}, \"
            + \"{\"description\": \"Steak\", \"price\": 14.99}, \"
            + \"{\"description\": \"Salad\", \"price\": 6.99}]\"";

        Gson gson = new Gson();
        Type restaurantType =new TypeToken<RestaurantWithMenu>() {}.getType();
        RestaurantWithMenu rm=gson.fromJson(JsonRestaurant, restaurantType);
        System.out.println(rm);
    }
    catch (Exception e) {e.printStackTrace();}
```

Reflection



```
Type restaurantType =new TypeToken<RestaurantWithMenu>() {}.getType();
RestaurantWithMenu rm=gson.fromJson(JsonRestaurant, restaurantType);
System.out.println(rm);
```

# INTERAZIONE DI RETE IN JSON

- JSON è un formato interoperabile utilizzato soprattutto per scambiare dati in rete
  - nel caso del progetto sia il client che il server saranno implementati in JAVA, per cui potrebbe essere utilizzata anche la serializzazione nativa di JAVA.
  - ma è possibile considerare anche un client/server JAVA che riceve dati JSON generati da una applicazione implementata con un linguaggio diverso
- due possibili scenari
  - il client/server invia al server/client un oggetto JSON che rappresenta una singola entità
    - esempio: il client invia ad un servizio social i dati del proprio profilo
  - il client/server invia al server/client un oggetto JSON che contiene la rappresentazione di uno stream di entità
    - esempio: un server invia al client tutti i post pubblicati sul suo profili social

# CLIENT JSON: INVIO SINGOLA ENTITA'

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

public class Restaurants {

public static void main (String args[])

{ if (args.length!=2) return;

String host= args[0]; int port = Integer.parseInt(args[1]); DataOutputStream os;
try (Socket s= new Socket(host, port));{
    os= new DataOutputStream(s.getOutputStream());    try with resources
    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 Gson();

    String restaurantJson= gson.toJson(restaurant);
    os.writeUTF(restaurantJson);
} catch(Exception e) {}}}
```

invio oggetto JSON  
sullo stream (una stringa)

# SERVER JSON: RICEZIONE DI UNA SINGOLA ENTITA'

```
import java.net.*; import java.io.*; import com.google.gson.*; import java.lang.reflect.*;
import com.google.gson.reflect.*;
public class ServerRestaurant {
public static void main (String args[])
{ if (args.length!=1) return;
  int port = Integer.parseInt(args[0]);
  try (ServerSocket s= new ServerSocket(port));{
    DataInputStream is= new DataInputStream(s.accept().getInputStream());
    System.out.println("accettato");
    String json= is.readUTF();
    Gson gson = new Gson();
    Type restaurantType =new TypeToken<RestaurantWithMenu>() {}.getType();
    RestaurantWithMenu rm=gson.fromJson(json,  restaurantType);
    System.out.println(rm);
  }
  catch (Exception e) {}
```

# 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").getString();

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

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

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

# DESERIALIZAZIONE COMPOSIZIONE DI OGGETTI

```
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]

# GSON STREAMING API

- streaming: utile supporto quando si invia 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 il file avrebbe grosse dimensioni
- GSON streaming offre metodi il caricamento incrementale di parti dell'oggetto
- utile
  - quando l'oggetto ha dimensione troppo grossa
  - quando 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());
                }
            }
        }
    }
}
```

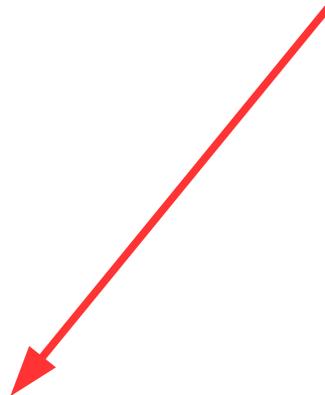
# GSON STREAMING API: JSONREADER

```
} 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

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



# 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`

# SERIALIZAZIONE 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

# SERIALIZAZIONE JAVA: HOW TO DO

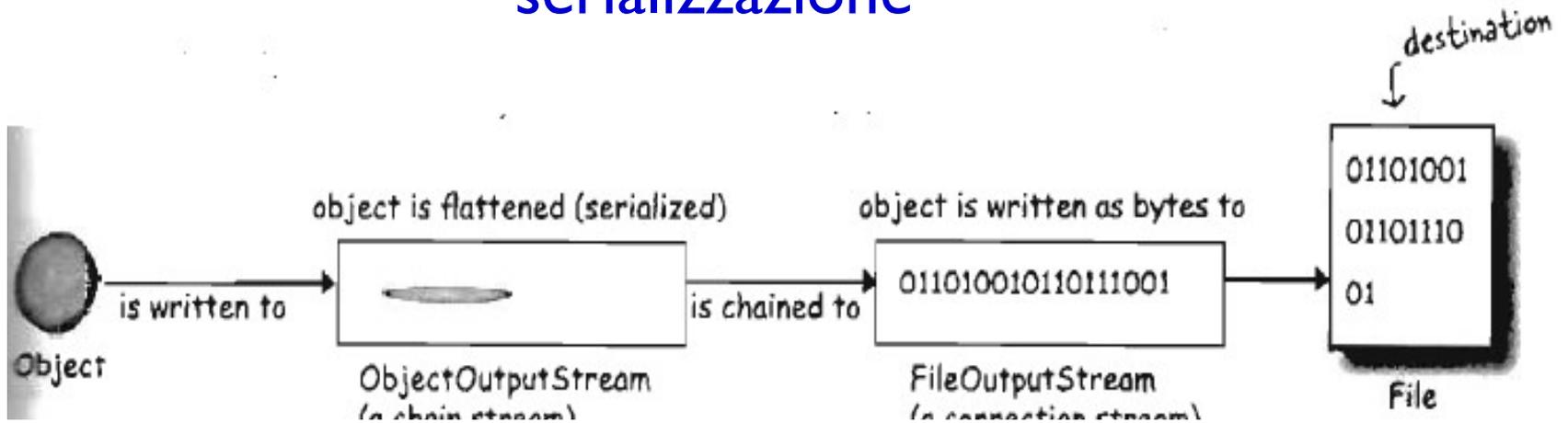
```
import java.io.*;

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();}
        }
    }
}
```

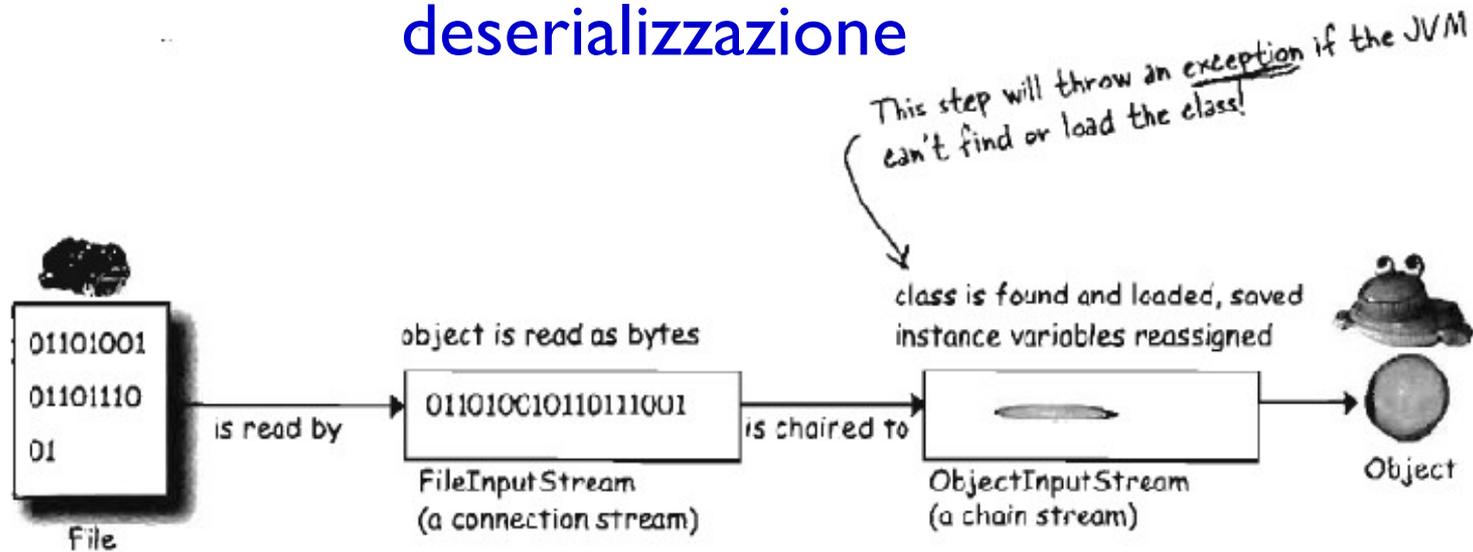
- la serializzazione vera e propria è gestita dalla classe `ObjectOutputStream`
- tale stream deve essere concatenato con uno stream di bytes, che può essere un `FileOutputStream`, uno stream di bytes associato ad un socket, uno stream di byte generato in memoria,...

# SERIALIZAZIONE E DESERIALIZAZIONE

## serializzazione



## deserializzazione



# DESERIALIZAZIONE

```
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**

# DESERIALIZAZIONE

```
// 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

# DESERIALIZAZIONE

- 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 alla 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, es. numero 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`

# ANCORA SUL MULTITHREADING

---

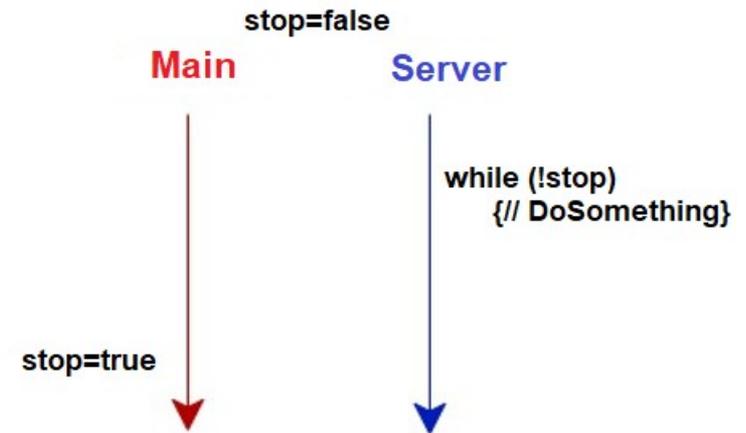
- Variabili volatile
- Variabili Atomic

# PERCHE' VOLATILE?

```
public class Server extends Thread
{
    boolean stop = false; int i;

    public void run()
    {
        while(! stop) {};
        System.out.println("Server is stopped....");
    }

    public void stopThread()
    {
        stop = true;
    }
}
```



```
public class StoppingAThread
{
    public static void main(String args[]) throws InterruptedException
    {
        Server myServer = new Server();
        myServer.start();

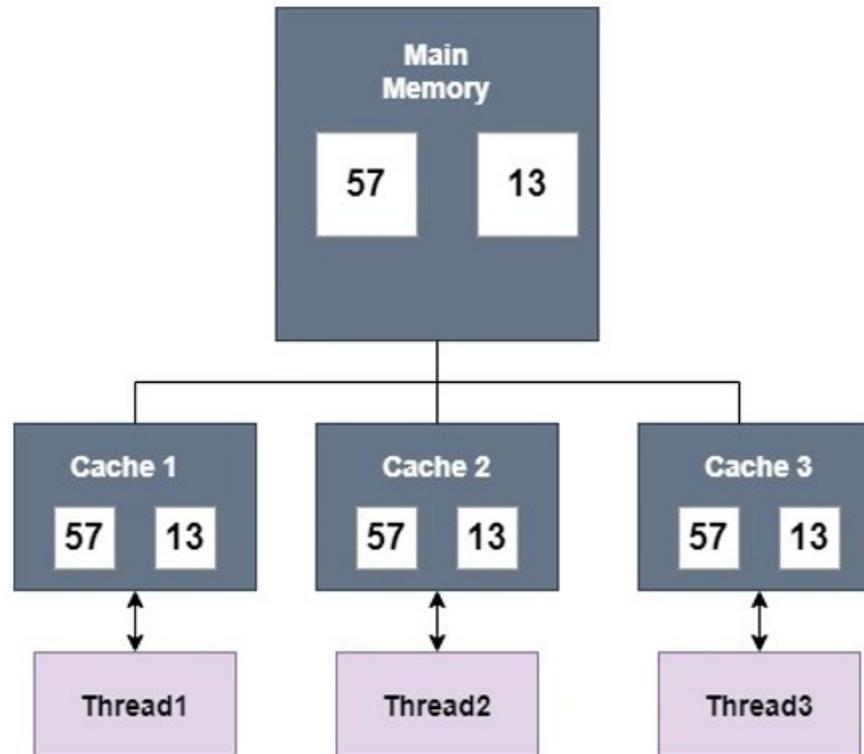
        System.out.println(Thread.currentThread().getName() + " is stopping Server thread");
        Thread.sleep(1000);
        myServer.stopThread();

        System.out.println(Thread.currentThread().getName() + " is finished now");
    }
}
```



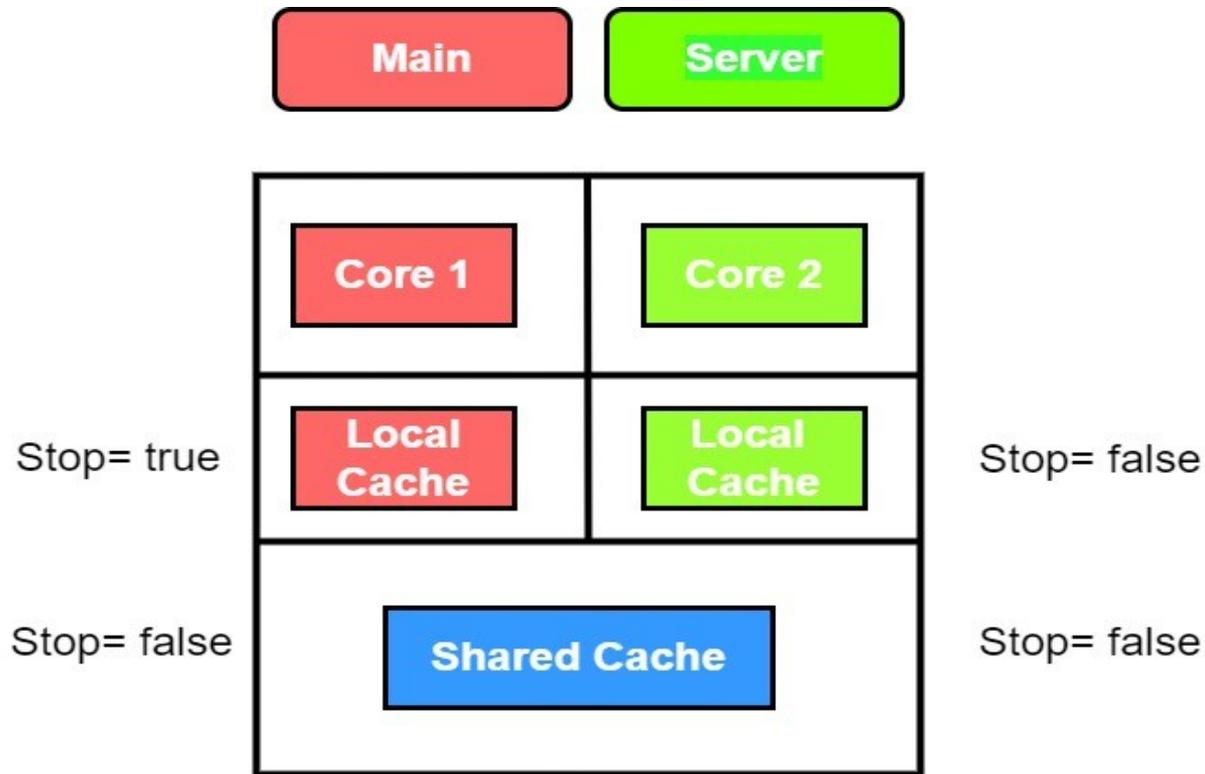
**il programma non termina!**

# IL PROBLEMA DELLA VISIBILITA'



architettura di riferimento, utile per capire il problema della visibilità

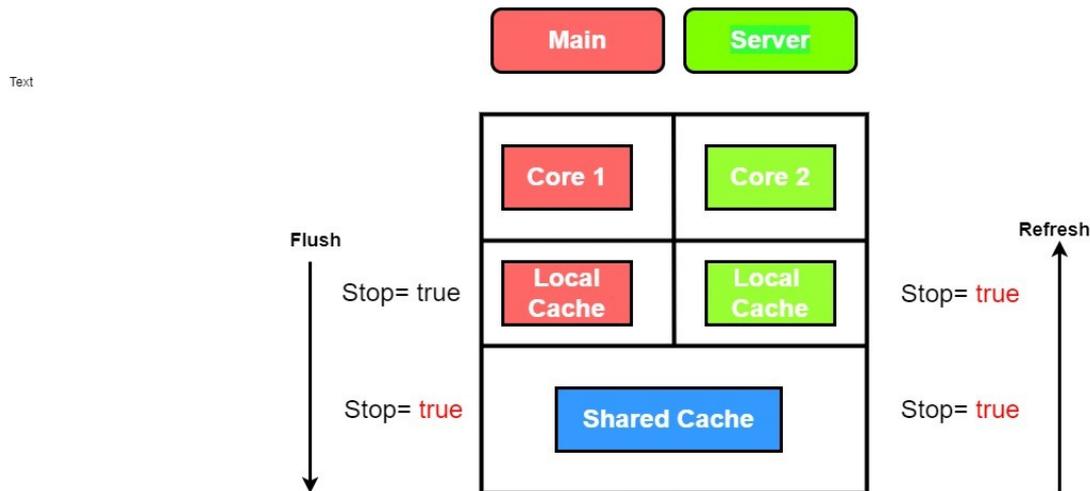
# IL PROBLEMA DELLA VISIBILITA'



- quando il `Main` aggiorna `Stop`, è possibile che la modifica non sia riportata nella memoria condivisa
- il `Main` aggiorna la variabile `Stop` nella propria cache, ma la modifica non viene riportata nella memoria condivisa

# IL MODIFICATORE VOLATILE

- il problema riguarda la “**visibilità**” della modifica, non la sincronizzazione: read e write di un booleano sono atomiche
- modifichiamo la dichiarazione con la keyword **volatile**  
`volatile boolean stop = false`
- l'aggiornamento ad una variabile **volatile** è sempre effettuato nella main memory
  - flush della cache
- il valore della variabile **volatile** è sempre letto dalla memoria



# VOLATILE: VISIBILITA' DI SCRITTURE

- tutte le scritture su una variabile volatile sono riportate direttamente nella memoria condivisa
- inoltre, tutte la variabili visibili dal thread che sta eseguendo la modifica vengono anche sincronizzate sulla memoria condivisa
- esempio:

```
this.nonVolatileVarA = 34;
```

```
this.nonVolatileVarB = new String("Text");
```

```
this.volatileVarC = 300;
```

- quando viene eseguita la terza istruzione, sulla variabile volatileC, i valori delle due variabili non-volatile vengono sincronizzati in memoria condivisa

# VOLATILE: VISIBILITA' DI LETTURE

- quando viene letto il valore di una variabile volatile, viene garantito che tale valore venga letto direttamente dalla memoria condivisa
- inoltre, viene fatto il refresh di tutte la variabili visibili dal thread che sta eseguendo la lettura
- esempio:

```
c = other.volatileVarC;
```

```
b = other.nonVolatileB;
```

```
a = other.nonVolatileA;
```

- la prima istruzione è la lettura di una variabile volatile. Quando questa variabile viene letta dalla memoria, viene effettuato il refresh anche delle due altre variabili

# UNA SOLUZIONE ALTERNATIVA

```
public class Server extends Thread
```

```
{ Boolean stop = false; int i;
```

```
public void run()
```

```
{ synchronized(stop) {};
```

```
while(! stop)
```

```
{synchronized(stop) {}};
```

```
System.out.println("Server is stopped...");}
```

```
public synchronized void stopThread(){ stop = true; }}
```

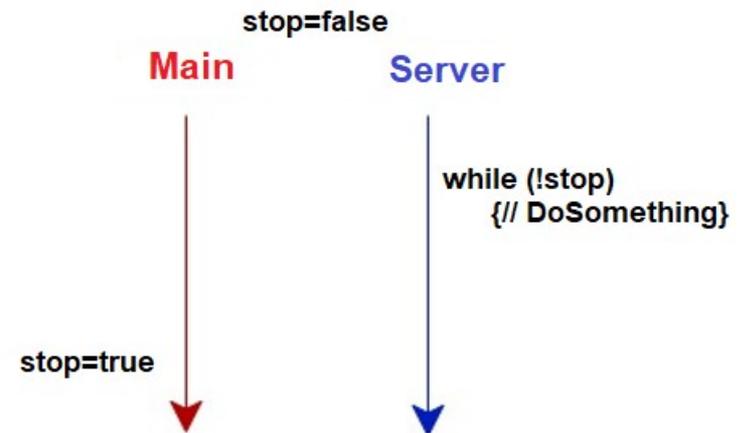
```
public class StoppingAThread
```

```
{...}
```

sincronizzarsi sulla variabile stop ha lo stesso

effetto di usare il modificatore volatile

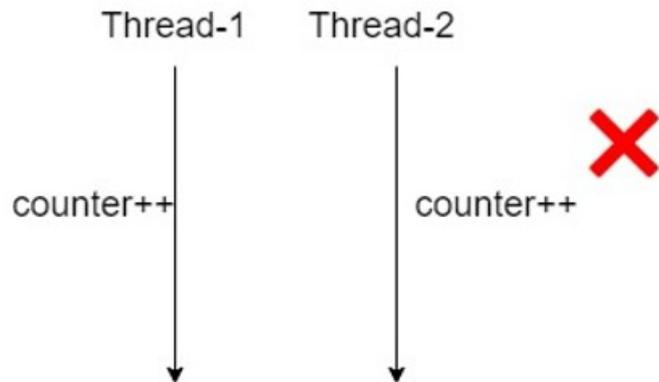
la variabile deve essere definita Boolean, per poter acquisire la lock



# SINCRONIZZAZIONE: VISIBILITA'

- blocchi e metodi sincronizzati forniscono garanzia di visibilità simile a quella offerta dal modificatore `volatile`
- quando un thread entra in un metodo o blocco sincronizzato, viene effettuato un refresh di tutte le variabili visibili dal thread
- quando un thread esce da un blocco sincronizzato, tutte le variabili visibili dal thread vengono scritte in memoria
- monitor garantisce sia sincronizzazione che visibilità
- quando usare `volatile`?
  - quando la variabile condivisa è di tipo semplice
  - per acquisire la lock occorrerebbe fare il cast al corrispondente oggetto
  - tipico del pattern “termina l'esecuzione di un thread”

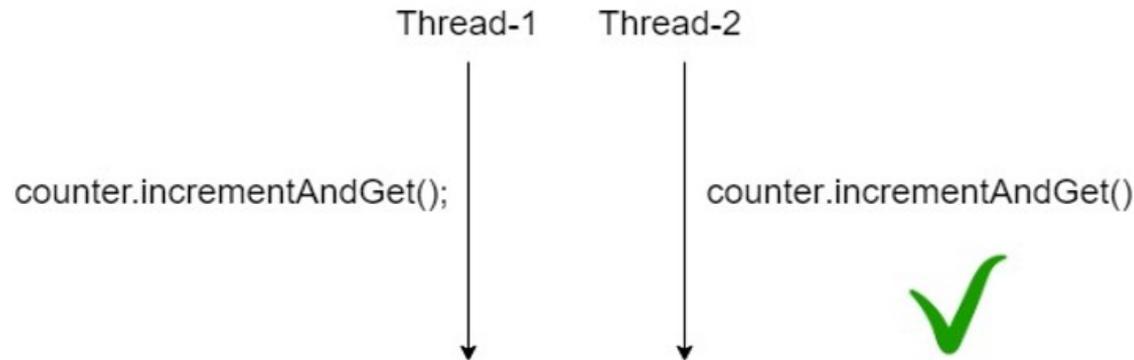
# SINCRONIZZAZIONE SU VARIABILI



Thread-1	Thread-2
Read value (=1)	
	Read value (=1)
Add 1 and write (=2)	
	Add 1 and write (=2)

- l'incremento di una variabile (volatile o meno) **non è atomico**
- se più thread provano ad incrementare una variabile in modo concorrente, un aggiornamento **può andare perduto** (anche se la variabile è volatile)
- ovviamente il problema può essere risolto con le lock
- soluzione alternativa: usare le variabili `Atomic`

# ATOMIC VARIABLES



```
AtomicInteger value = new AtomicInteger(1);
```

- operazioni atomiche che non richiedono sincronizzazioni esplicite o lock: è la JVM che garantisce la atomicità
  - incrementAndGet(): atomically increments by one
  - decrementAndGet(): atomically decrements by one
  - compareAndSet(int expectedValue, int newValue)
- molte altre classi
  - AtomicLong
  - AtomicBoolean

# ATOMIC VARIABLES: UN ESEMPIO

```
import java.util.concurrent.*; import java.util.concurrent.atomic.*;
public class AtomicIntExample {
public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(2);
    AtomicInteger atomicInt = new AtomicInteger();
    for(int i = 0; i < 10; i++){
        CounterRunnable runnableTask = new CounterRunnable(atomicInt);
        executor.submit(runnableTask);
    }
    executor.shutdown(); }
class CounterRunnable implements Runnable {
    AtomicInteger atomicInt;
    CounterRunnable(AtomicInteger atomicInt){this.atomicInt = atomicInt;}
    @Override
    public void run() {
        System.out.println("Counter- " + atomicInt.incrementAndGet());}}
```

# JAVA.UTIL.CONCURRENT.ATOMIC

