
Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

# Programmazione Concorrente e Distribuita
# Serializzazione

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

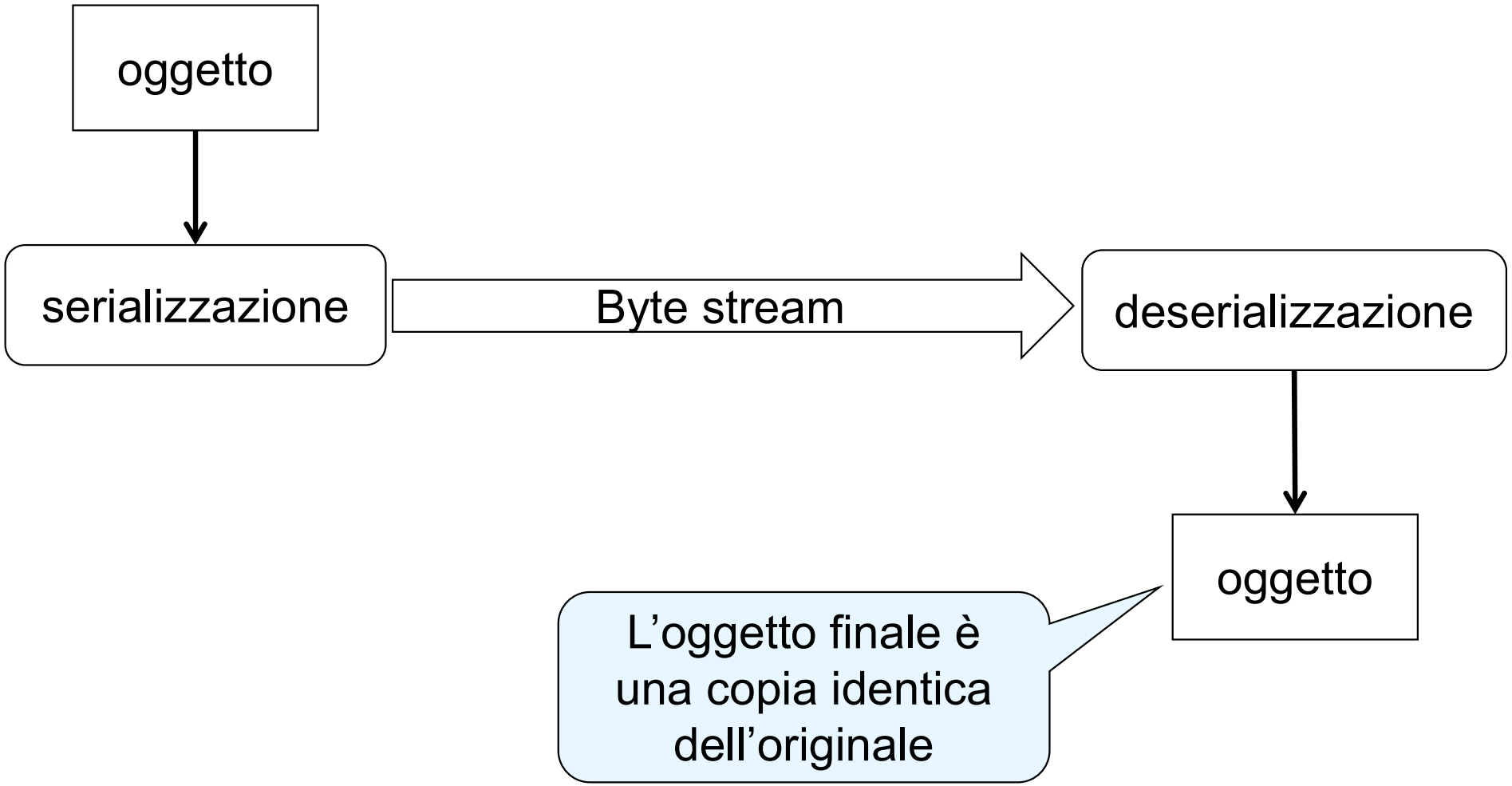
# Serializzazione

- È l'operazione di conversione di **un oggetto** (o di un grafo di oggetti) in una sequenza di byte
- Permette di scrivere l'oggetto su uno stream di byte.


# Deserializzazione

- La Deserializzazione è l'operazione inversa, che permette la lettura da uno stream di byte di un oggetto o del grafo di oggetti precedentemente serializzato
- Vengono ripristinati i valori degli attributi e lo stato dell'oggetto originale

# Serializzazione e deserializzazione



# Serializzazione di un tipo primitivo

- La scrittura di tipi primitivi su uno stream è molto semplice.
- Esempio:

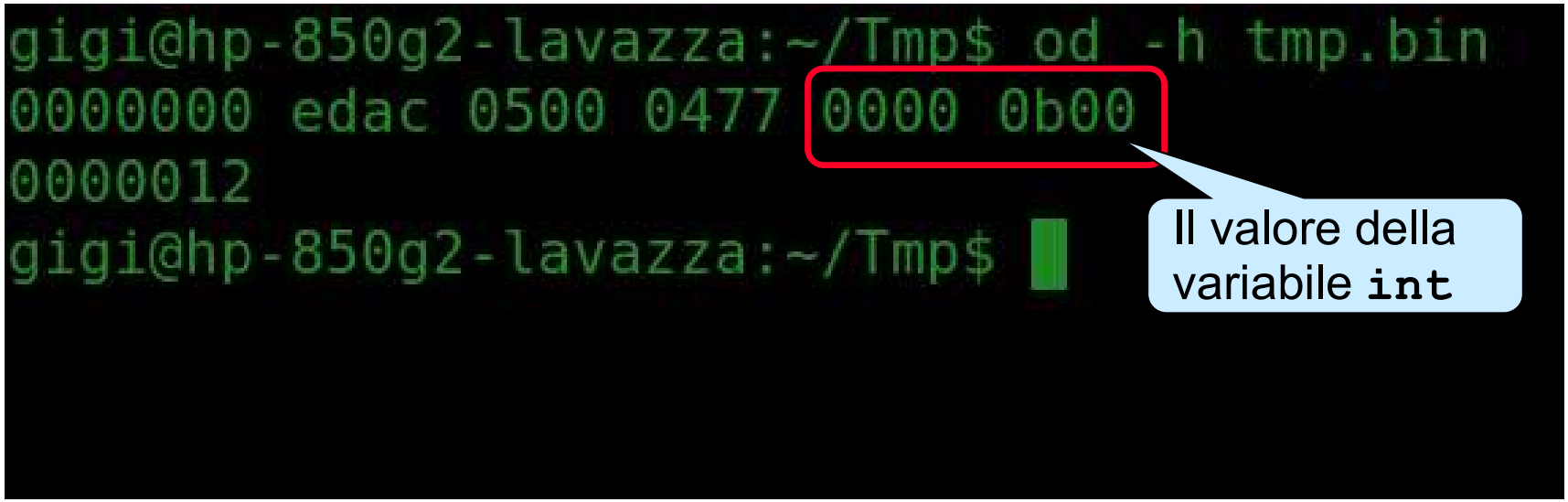
```
public class Primitive {
  public static void main(String[] args) throws IOException {
    String fileName = "tmp.bin";

    ObjectOutput os = new ObjectOutputStream(new
                                  FileOutputStream(fileName));
    int i=11;
    os.writeInt(i);
    os.flush();
    os.close();
  }
}
```

Byte stream

C'è un'operazione apposita per gli interi

# Serializzazione di un tipo primitivo

- Oltre ai 4 byte della variabile intera, il file contiene altre informazioni, che (almeno per ora) non ci interessano



# Deserializzazione di un tipo primitivo

- La lettura di tipi primitivi da uno stream è pure molto semplice.
- Esempio:

```
public class Primitive {
  public static void main(String[] args) throws IOException {
    String fileName = "tmp.bin";
    ObjectInput is = new ObjectInputStream(
                                  new FileInputStream(fileName));
    int j = is.readInt();
    System.out.println("read: "+j);
    is.close()
  }
}
```

# Le interfacce `ObjectOutput` e `ObjectInput`

- Consentono di scrivere e leggere (serializzandoli)
  - Tipi primitivi
  - **Oggetti**

# Interface `ObjectOutput`

| Modifier and Type | Method | Description |
|---|---|---|
| void | close() | Closes the stream. |
| void | flush() | Flushes the stream. |
| void | write(byte[] b) | Writes an array of bytes. |
| void | write(byte[] b, int off, int len) | Writes a sub array of bytes. |
| void | write(int b) | Writes a byte. |
| void | writeObject(Object obj) | Write an object to the underlying storage or stream. |

# Interface `ObjectInput`

| Modifier and Type | Method | Description |
|---|---|---|
| `int` | `available()` | Returns the number of bytes that can be read without blocking. |
| `void` | `close()` | Closes the input stream. |
| `int` | `read()` | Reads a byte of data. |
| `int` | `read(byte[] b)` | Reads into an array of bytes. |
| `int` | `read(byte[] b, int off, int len)` | Reads into an array of bytes. |
| `Object` | `readObject()` | Read and return an object. |
| `long` | `skip(long n)` | Skips n bytes of input. |

# Serializzazione di Oggetto su uno stream

```
public class Primitive {
  public static void main(String[] args) throws IOException,
                                                 ClassNotFoundException {
    String fileName = "tmp.bin";
    ObjectOutput os = new ObjectOutputStream(
                                  new FileOutputStream(fileName));
    os.writeObject("Oggi");
    os.flush();
    os.close();
    ObjectInput is = new ObjectInputStream(
                                  new FileInputStream(fileName));
    String s = (String)is.readObject();
    System.out.println("read: "+s); // stampa la stringa "Oggi"
    is.close();
  }
}
```

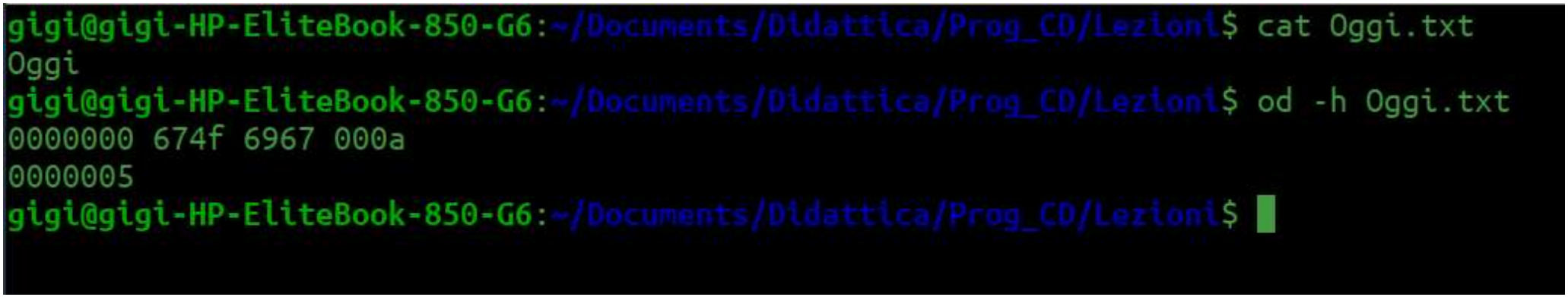
# Cosa c'è nel file

| DEC | HEX | CHARACTER | DEC | HEX | CHARACTER |
|---|---|---|---|---|---|
| 64 | 0x40 | @ | 96 | 0x60 | ` |
| 65 | 0x41 | A | 97 | 0x61 | a |
| 66 | 0x42 | B | 98 | 0x62 | b |
| 67 | 0x43 | C | 99 | 0x63 | c |
| 68 | 0x44 | D | 100 | 0x64 | d |
| 69 | 0x45 | E | 101 | 0x65 | e |
| 70 | 0x46 | F | 102 | 0x66 | f |
| 71 | 0x47 | G | 103 | 0x67 | g |
| 72 | 0x48 | H | 104 | 0x68 | h |
| 73 | 0x49 | I | 105 | 0x69 | i |
| 74 | 0x4A | J | 106 | 0x6A | j |
| 75 | 0x4B | K | 107 | 0x6B | k |
| 76 | 0x4C | L | 108 | 0x6C | l |
| 77 | 0x4D | M | 109 | 0x6D | m |
| 78 | 0x4E | N | 110 | 0x6E | n |
| 79 | 0x4F | O | 111 | 0x6F | o |
| 80 | 0x50 | P | 112 | 0x70 | p |
| 81 | 0x51 | Q | 113 | 0x71 | q |
| | | | | x72 | r |
| | | | | x73 | s |
| | | | | x74 | t |
| | | | | x75 | u |
| | | | | x76 | v |
| | | | | x77 | w |
| 88 | 0x58 | X | 120 | 0x78 | x |
| 89 | 0x59 | Y | 121 | 0x79 | y |
| 90 | 0x5A | Z | 122 | 0x7A | z |

```
gigi@hp-850g2-lavazza:~/Tmp$ od -h tmp.bin
0000000 edac 0500 0074 4f04 6767 0069
0000013
gigi@hp-850g2-lavazza:~/Tmp$
```

# NB

- Cosa succede se scrivo “Oggi” in un file con un normale editor?
- Trovo le stesse informazioni che trovo nel file scritto con **`writeObject`**?



# Serializzazione di più Oggetti su uno stream

```
public static void main(String[] args) throws IOException,
                                               ClassNotFoundException {
  String fileName = "tmp.bin";
  ObjectOutput os = new ObjectOutputStream(
                              new FileOutputStream(fileName));
  os.writeObject("Oggi");
  os.writeObject(new Date());
  os.flush();
  os.close();
  ObjectInput is = new ObjectInputStream(
                              new FileInputStream(fileName));
  String s = (String) is.readObject();
  Date d = (Date) is.readObject();
  System.out.println(s + " " + d);
  is.close();
}
```

# Cosa c'è nel file

```
gigi@hp-850g2-lavazza:~/Tmp$ od -h tmp.bin
0000000 edac 0500 0074 4f04 6767 7369 0072 6a0e
0000020 7661 2e61 7475 6c69 442e 7461 6865 816a
0000040 4b01 7459 0319 0000 7078 0877 0000 5b01
0000060 516d 6a71 0078
0000065
gigi@hp-850g2-lavazza:~/Tmp$
```

# Object serialization

- Perché ci serve memorizzare sullo stream tutte le informazioni sui tipi e sulle dimensioni degli oggetti o tipi primitivi?
- Dopo la serializzazione, si può leggere dallo stream e deserializzare.
  - Informazioni su tipo e byte occupati sono utilizzati per ricreare l'oggetto in memoria.

- `ObjectOutputStream` contiene un metodo che serializza un oggetto e lo invia al flusso di output:

```
public final void writeObject(Object x)
        throws IOException
```

- `ObjectInputStream` contiene il metodo per la deserializzazione:

```
public final Object readObject()
      throws IOException, ClassNotFoundException
```

# Serializzazione di array di Oggetti su uno stream

```
public class ArrayDiStringhe {
  public static void main(String[] args)
      throws IOException, ClassNotFoundException {
    String[] p = new String[]{"rosso","giallo","blu"};
    String fileName = "tmp.ser"; // ".ser" -> convenzione java
    ObjectOutput os = new ObjectOutputStream(
                              new FileOutputStream(fileName));
    os.writeObject(p);
    os.flush();
    os.close();
    ObjectInput is = new ObjectInputStream(
                              new FileInputStream(fileName));
    String[] p1 = (String[]) is.readObject();
      for (String s: p1)
         System.out.print(s + ", "); // rosso, giallo, blu,
      is.close();
  }
}
```

# Serializzazione di tipi non primitivi

- Consideriamo una classe **`Point`** definita da noi
  - La classe rappresenta un punto nello spazio euclideo in base alle due variabili di istanza x e y
- Per renderla serializzabile dobbiamo «attrezzarla» appositamente
- Nel codice di questa classe evidenziamo le parti relative alla serializzazione

```
import java.io.Serializable;
public class Point implements Serializable{
  private static final long serialVersionUID = 1;
  private int x, y;
  public Point(int x,int y){
    this.x=x;
    this.y=y;
  }
  public String toString(){
    return "("+x+", "+y+")";
  }
}
```

# Interfaccia `Serializable`

- Per rendere un oggetto serializzabile bisogna:
  - Far implementare alla classe l'interfaccia **`Serializable`**
  - Definire una versione **`serialVersionUID`** relativa alla classe (se non inserito, la JVM provvede a calcolarlo in maniera autonoma, e questo potrebbe portare a errori).
    - Questo serve in fase di deserializzazione per verificare che la classi usate da chi ha serializzato l'oggetto e chi lo sta deserializzando siano compatibili; se, infatti, le versioni non corrispondono sarà sollevata una **`InvalidClassException`**.

- ATTENZIONE: non abbiamo definito nessun metodo relativo all'interfaccia implementata: in generale non è necessario; sarà nostro compito solo in casi particolari.

# Serializzazione su stream di byte

```
public class TheMain {
  String fileName;
  TheMain(String fn){
    fileName=fn;
  }

  // methods

  public static void main(String[] args) {
    if(args.length==1) {
      TheMain tm = new TheMain(args[0]);
      tm.point_to_file();
      tm.point_from_file();
    } else {
      System.out.println("filename missing");
    }
  }
}
```

# Serializzazione su stream di byte

```
void point_to_file() {
    Point p=new Point(2,3);
    ObjectOutputStream output=null;
    try{
      output=new ObjectOutputStream(
                    new FileOutputStream(fileName));
      output.writeObject(p);
      output.close();
    }
    catch (FileNotFoundException | IOException e){ }
    System.out.println("Serializzazione completata.");
  }
```

Non dobbiamo fare nulla di speciale!

oggetto di classe **Punto**

# Deserializzazione da uno stream di byte

```
void point_from_file() {
  ObjectInputStream ois = null;
  Point p = null;
  try{
    ois = new ObjectInputStream(
                  new FileInputStream(fileName));
    p = (Point) ois.readObject();
    ois.close();
    System.out.println(p);
  }
  catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
  }
}
```

# Deserializzazione di Point

```
gigi@hp-850g2-lavazza:~/Documents/Didattica/Prog_CD/Esempi$ od -h dati.dat
0000000 edac 0500 7273 0500 6f50 6e69 0074 0000
0000020 0000 0000 0201 0200 0049 7801 0049 7901
0000040 7078 0000 0200 0000 0300
0000052
gigi@hp-850g2-lavazza:~/Documents/Didattica/Prog_CD/Esempi$
```

- L'esecuzione della deserializzazione produce:

  `(2, 3)`

# Serializzazione in Java

- La scrittura degli oggetti deve avvenire su uno stream di tipo **`ObjectOutputStream`**
- La lettura degli oggetti deve avvenire da uno stream di tipo **`ObjectInputStream`**
- Per poter essere serializzato un oggetto deve essere istanza di una classe che implementa l'interfaccia **`Serializable`**

# Esempio di classe serializzabile

```
class Employee implements Serializable{
    private static final long serialVersionUID = 1;
    String name;
    int number;
    float salary;
    Employee(String name, int no, float sal){
      this.name=name;
      this.number=no;
      this.salary=sal;
    }
    void display(){
        System.out.println(name+" "+number+" "+ salary);
    }
}
```

# Regole di serializzazione in Java

- Tutti i tipi primitivi sono serializzabili.
- Un oggetto è serializzabile se la sua classe o la sua superclasse implementano l'interfaccia **`Serializable`**.
- Un oggetto può essere serializzabile anche se la sua superclasse non lo è. Ma bisogna che la superclasse abbia un costruttore senza argomenti.
- I campi di classe con modificatore **`transient`** non vengono serializzati.
  - quindi i campi che non supportano la serializzazione vanno qualificati come **`transient`**.
- I campi **`static`** non vengono serializzati (in quanto parte della classe, non dell'oggetto).
- Se le variabili membro di un oggetto serializzabile hanno un riferimento a un oggetto non serializzabile, il codice verrà compilato, ma verrà generata una **`RuntimeException`**.

# Serializzazione in Java: classe e superclasse

```
import java.io.*;

class Employee extends Person implements Serializable{
  private static final long serialVersionUID = 1;
  int serialNumber;
  float salary;
  Employee(String name, int no, float sal){
      super(name);
      serialNumber = no;
      salary = sal;
  }
  void display(){
    System.out.println(name+"    "+serialNumber+
                       "    "+salary);
  }
}
```

# Main: serializza e deserializza

```
import java.io.*;
public class TheMain {
  String fileName;
  ObjectOutputStream output=null;
  ObjectInputStream ois = null;
  TheMain(String fn) throws FileNotFoundException, IOException{
    fileName=fn;
    output=new ObjectOutputStream(new
                                  FileOutputStream(fileName));
    ois = new ObjectInputStream(new FileInputStream(fileName));
  }
```

# Main: serializza e deserializza

```
public static void main(String[] args) {
  if(args.length==1) {
    TheMain tm=null;
    try {
      tm = new TheMain(args[0]);
    } catch (IOException e) {
      System.err.println("Initialization failure");
      System.exit(0);
    }
    tm.exec();
  } else {
    System.out.println("filename missing");
  }
}
```

# Main: serializza e deserializza

```
void exec() {
  Employee empl=new Employee("Rossi", 100, 3000);
  empl.display();
  try {
    output.writeObject(empl);
    output.close();
  } catch (IOException e) {
    System.err.println("Problems serializing");  return;
  }
  System.out.println("Serializzazione completata.");
  Employee newEmpl=null;
  try {
    newEmpl = (Employee) ois.readObject();
    ois.close();
  } catch (ClassNotFoundException | IOException e) {
    System.err.println("Problems de-serializing"); return;
  }
  newEmpl.display();
}
```

Scrive l'oggetto serializzato su file

Legge l'oggetto serializzato da file

# Comportamento del programma

- Dipende da come è definite la classe **`Person`**, che **`Employee`** estende
- Casi:
  - **`Person`** è serializzabile
  - **`Person`** non è serializzabile e non ha un costruttore di default (senza parametri)
  - **`Person`** non è serializzabile e ha un costruttore di default (senza parametri)

# Caso 1: superclasse serializzabile

```
import java.io.Serializable;

class Person implements Serializable{
  private static final long serialVersionUID = 1;

  String name;
  Person(String n) {
    this.name = n;
  }
}
```

# Caso 1: output

```
Rossi   100     3000.0
Serializzazione completata.
Rossi   100     3000.0
```

# Caso 2: superclasse non serializzabile senza costruttore di default

```
class Person {
      String name;
      Person(String n) {
            this.name = n;
      }
}
```

# Caso 2: output

```
Rossi   100     3000.0
Serializzazione completata.
Employee deserialization failed
```

# Caso 3: superclasse non serializzabile con costruttore di default

```
class Person {
      String name;
      Person(){
            this.name = "anonimo";
      }
      Person(String n) {
            this.name = n;
      }
}
```

senza questo costruttore la serializzazione di **`Employee`** fallisce

# Caso 3: output

```
Rossi    100     3000.0
Serializzazione completata.
anonimo    100     3000.0
```

# Serializzazione in Java: riferimenti ad object NON serializzabile

```
class Person {
  public String name;
  Person(String n) { this.name = n; }
}
class Employee implements Serializable{
  private static final long serialVersionUID = 1;
  String name;  int serNum;  float salary;
  Person parent;

  Employee(String name, int no, float sal,
           String pname){
     this.name=name; serNum = no; salary = sal;
     parent = new Person(pname);
  }
  void display(){
    System.out.println(name+" "+serNum+" "+salary);
  }
  String getParentName(){ return parent.name; }
}
```

non serializzabile

NB: stavolta **Employee** non estende **Person**

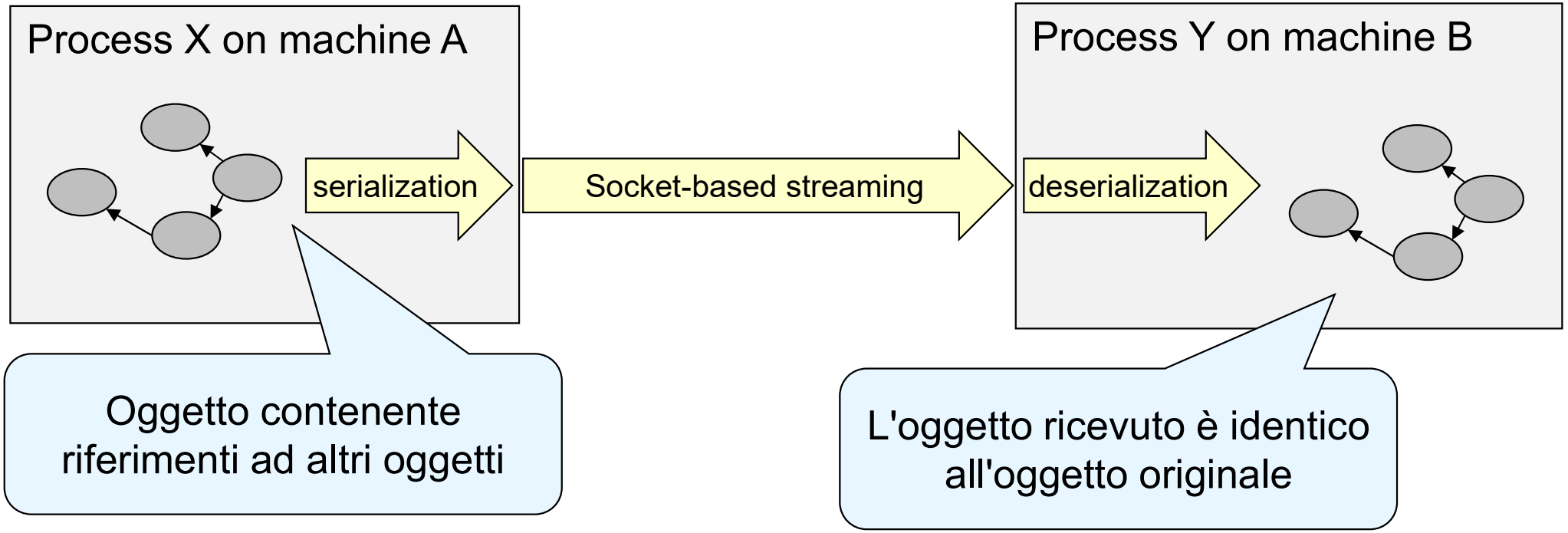
Riferimento a oggetto non serializzabile:

# Riferimento non serializzabile: output

```
Rossi Anselmo   100     3000.0
Serializzazione completata.
Employee deserialization failed
```

# Stream di oggetti e Socket

- Per trasferire dati da un client ad un server (o viceversa) in Java si possono utilizzare anche stream di oggetti trasferiti tramite socket
  - **`ObjectInputStream`**
  - **`ObjectOutputStream`**
- Gli oggetti devono implementare l'interfaccia **`java.io.Serializable`**



# Serializzazione Client/Server

- Esempio:
  - Il Server è in attesa di una connessione
  - Il Client si connette e manda un oggetto di classe Employee al server
  - Il server riceve l'oggetto e lo visualizza

# Serializzazione Client/Server

```
class Person implements Serializable{
  private static final long serialVersionUID = 1;
  String name;
  Person(String n) { this.name = n; }
}

class Employee extends Person implements Serializable{
  private static final long serialVersionUID = 1;
  int serNum;
  float salary;
  Employee(String name, int no, float sal){
      super(name);
      serNum = no;
      salary = sal;
  }
  void display(){
    System.out.println(name+"   "+serNum+"    "+salary);
  }
}
```

# Serializzazione Client/Server: Client

```
public class Client {
  void exec() {
    Socket socket=null;
    InetAddress addr=null;
    try {
      addr = InetAddress.getByName(null);
      socket = new Socket(addr, Server.PORT);
      ObjectOutputStream obj_out_s = new ObjectOutputStream(
                                        socket.getOutputStream());
      Employee empl=new Employee("Tizio", 123, 3000);
      empl.display();
      obj_out_s.writeObject(empl);
    } catch (IOException e) { e.printStackTrace();
    } finally {
      try { socket.close(); } catch (IOException e) { }
    }
    System.out.println("Client finished");
  }
  public static void main(String[] args) {
    new Client().exec();
  }
}
```

# Serializzazione Client/Server: Server

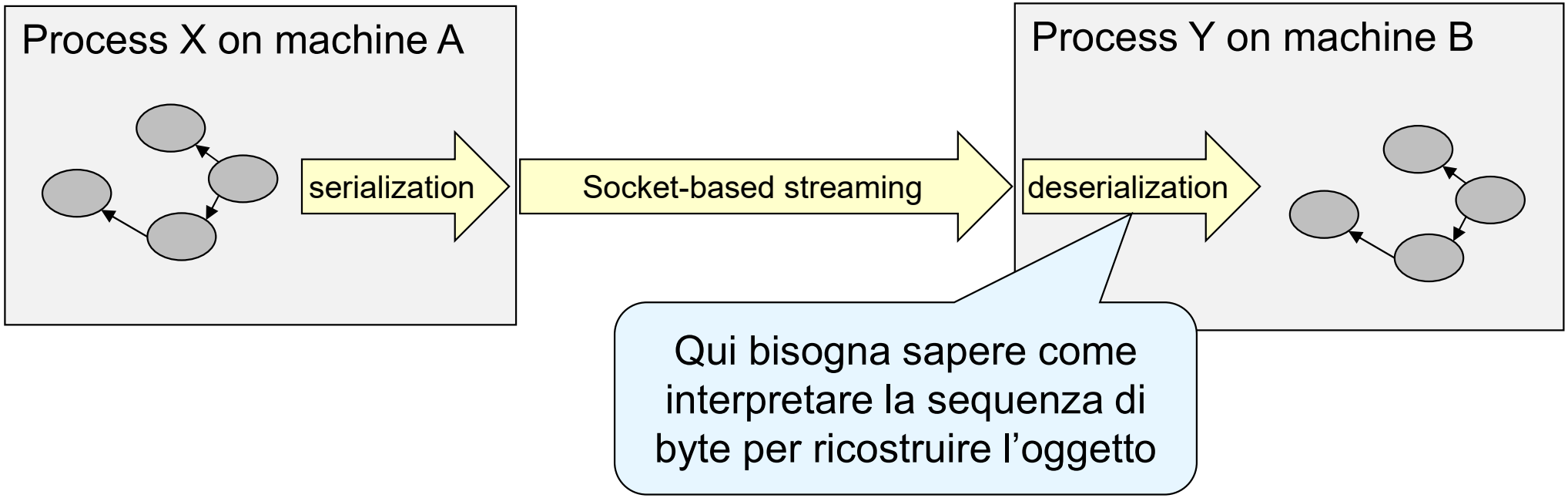
```
public class Server {
  public static final int PORT = 9999;
  Employee obj =null;
  public static void main(String[] args) {
    new Server().exec();
  }
}
```

# Serializzazione Client/Server: Server

```
void exec() {
  Socket socket=null;
  ServerSocket s=null;
  try {
    s = new ServerSocket(PORT);
    socket = s.accept();
    System.out.println("Connection accepted: " + socket);
    ObjectInputStream obj_in_s =new ObjectInputStream(
                                    socket.getInputStream());
    Employee emp = (Employee) obj_in_s.readObject();
    emp.display();
    obj_in_s.close();
  } catch (IOException | ClassNotFoundException e) {
  } finally {
    System.out.println("closing...");
    try { socket.close(); s.close(); } catch (IOException e){}
  }
}
```

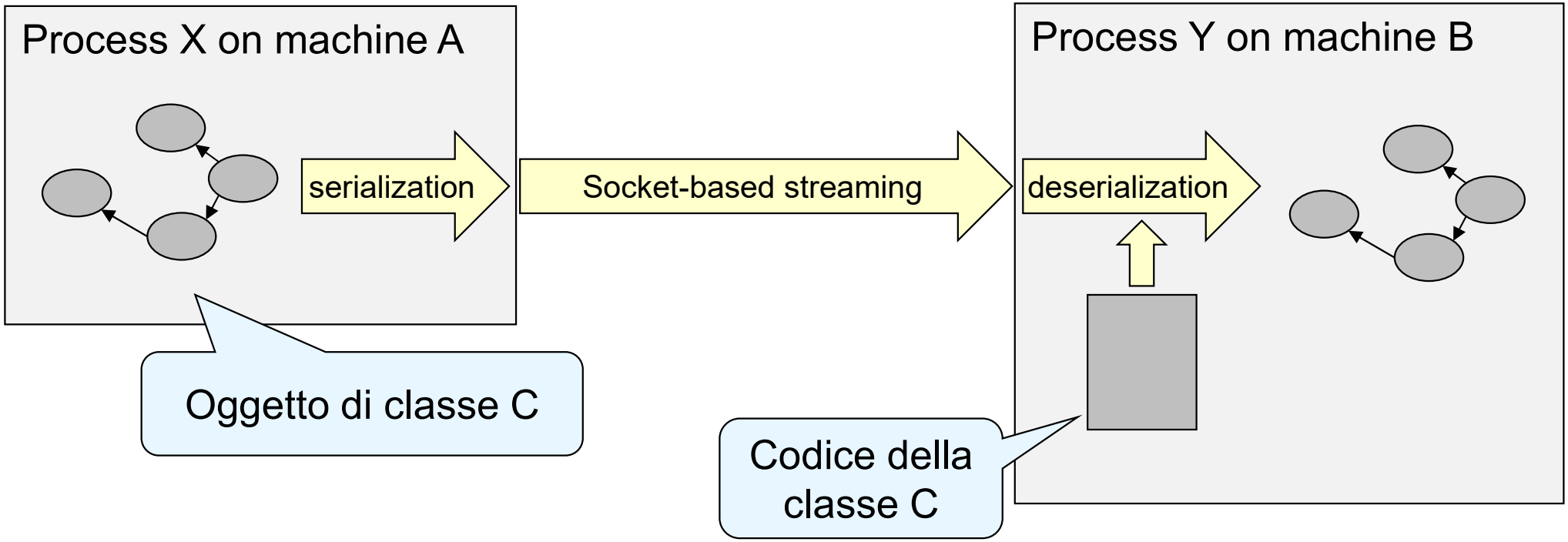
# Serializzazione Client/Server: ricostruzione dell'oggetto spedito

- Quando si trasferisce un oggetto serializzato si riceve una sequenza di byte e bisogna ricostruire un oggetto identico a quello originale.
- Problema: come interpretare la sequenza di Byte?
- Ci vogliono le «istruzioni di montaggio»

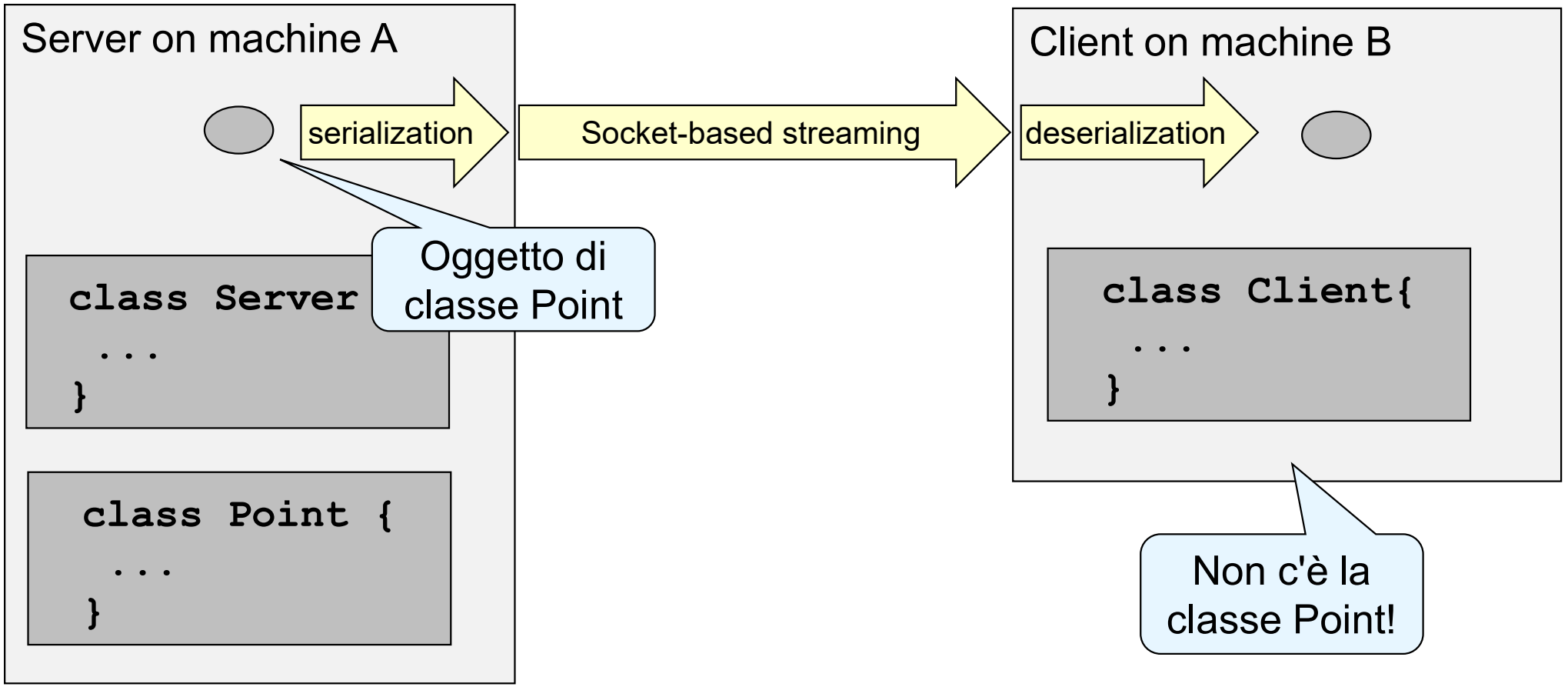


# Serializzazione Client/Server: ricostruzione dell'oggetto spedito

- Se l'oggetto è di tipo primitivo, la JVM sa cosa fare.
- Altrimenti, bisogna conoscere la classe dell'oggetto
  - La classe dell'oggetto fornisce le «istruzioni di montaggio»
- Se il Client non conosce la classe dell'oggetto ricevuto, si verifica l'eccezione **`ClassNotFoundException`**



# Serializzazione Client/Server: esempio problematico



# Serializzazione Client/Server: ClassNotFoundException

```
public class SenderServer {
  public static void main(String[] args) throws IOException {
    ServerSocket s = new ServerSocket(9996);
    Point p = new Point(6,11);
    Socket sc = s.accept();
    try {
      ObjectOutputStream  oos =
           new ObjectOutputStream(sc.getOutputStream());
      oos.writeObject(p); // scrittura dell'oggetto Punto
      oos.close();
    } finally {
      sc.close();
    }
  }
}
```

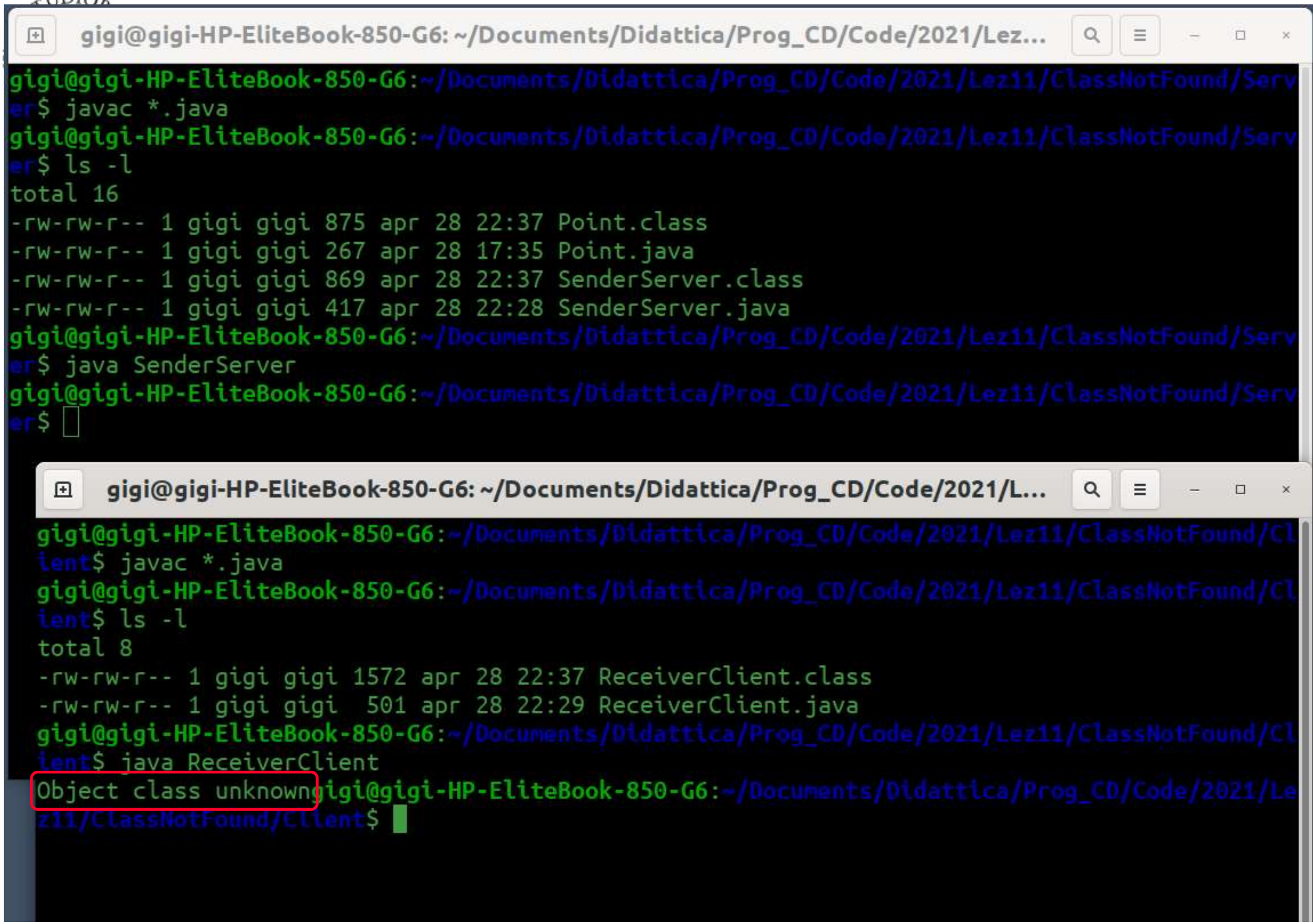
# Serializzazione Client/Server: ClassNotFoundException

```
public class ReceiverClient {
  public static void main(String[] args) throws IOException {
    InetAddress addr = InetAddress.getByName(null);
    Socket socket = new Socket(addr, 9996);
    try {
       ObjectInputStream ois =
              new ObjectInputStream(socket.getInputStream());
       Object p = ois.readObject();
       System.out.print("Object is: " + p);
    } catch(ClassNotFoundException e){
       System.out.print("Object class unknown");
    } finally {
       socket.close();
    }
  }
}
```

Legge l'oggetto come Object

Si verifica l'eccezione **`ClassNotFoundException`**, perché la classe di **`p`** non è nota

- Soluzione: bisogna che il programma client contenga la definizione della classe **`Punto`** (descrizione che deve essere la stessa usata dal server).


gigi@gigi-HP-EliteBook-850-G6: ~/Documents/Didattica/Prog_CD/Code/2021/Lez...
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Server$ javac *.java
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Server$ ls -l
total 16
-rw-rw-r-- 1 gigi gigi 875 apr 28 22:37 Point.class
-rw-rw-r-- 1 gigi gigi 267 apr 28 17:35 Point.java
-rw-rw-r-- 1 gigi gigi 869 apr 28 22:37 SenderServer.class
-rw-rw-r-- 1 gigi gigi 417 apr 28 22:28 SenderServer.java
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Server$ java SenderServer
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Server$
gigi@gigi-HP-EliteBook-850-G6: ~/Documents/Didattica/Prog_CD/Code/2021/L...
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Client$ javac *.java
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Client$ ls -l
total 8
-rw-rw-r-- 1 gigi gigi 1572 apr 28 22:37 ReceiverClient.class
-rw-rw-r-- 1 gigi gigi  501 apr 28 22:29 ReceiverClient.java
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Client$ java ReceiverClient
Object class unknowngigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassNotFound/Client$

# Controllo di versione

- Consideriamo la seguente sequenza di azioni:
  1. Definiamo una classe C
     - Serializzabile
     - Avente **`serialVersionUID=1`**
  2. Creiamo un'istanza, la serializzamo e la salviamo in un file usando uno **`ObjectStream`** .
  3. Aggiorniamo la definizione della classe C
     - aggiungendo un nuovo campo
     - Cambiando **`serialVersionUID`**, che diventa 2.
- Cosa succede quando un programma contenente la nuova definizione di C tenta di deserializzare l'oggetto salvato?
- Viene generata una **`java.io.InvalidClassException`** perché è cambiato l'identificatore univoco.

# Controllo di versione

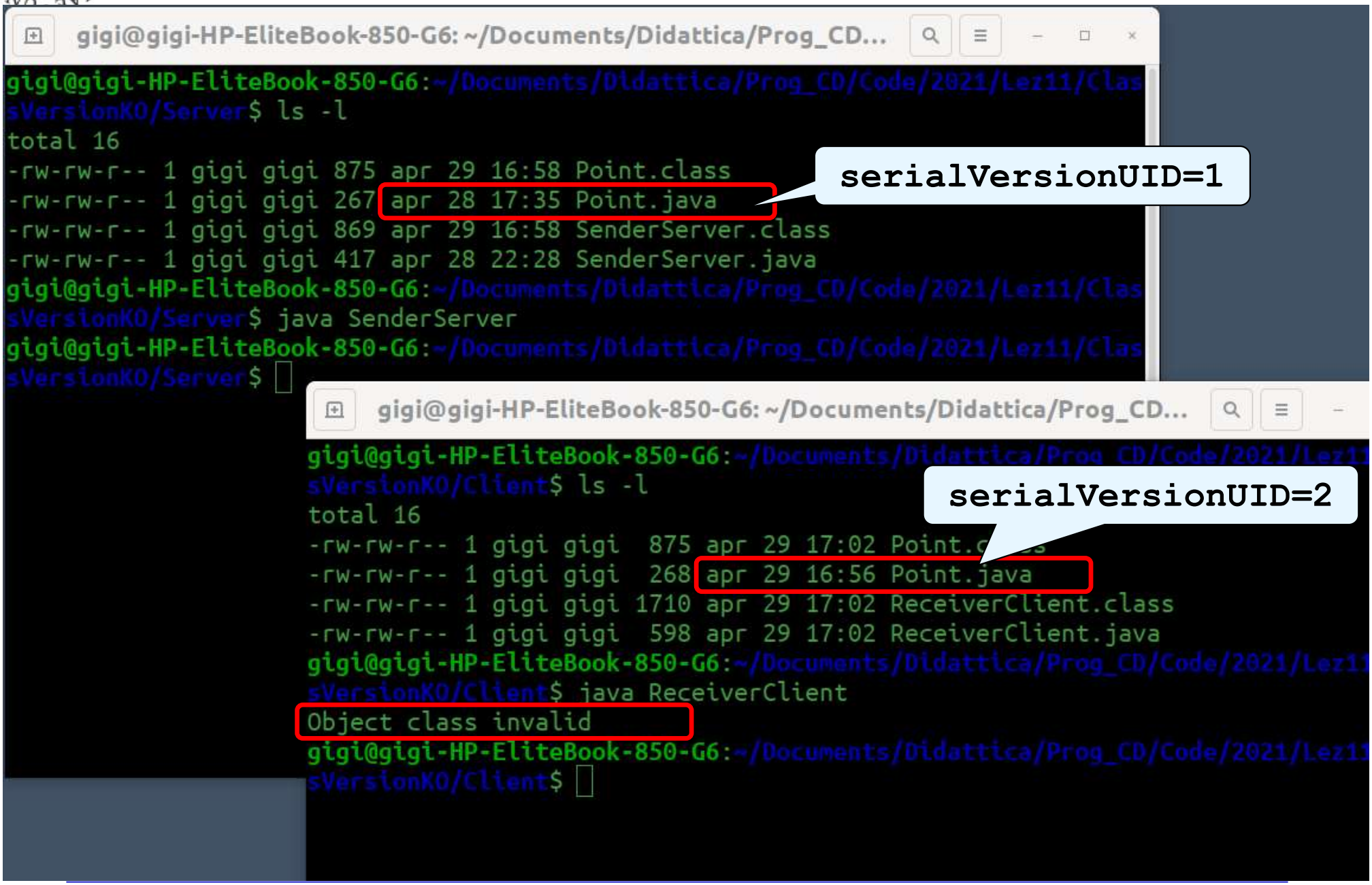
- Se l'identificatore della classe non equivale all'identificatore dell'oggetto serializzato, viene generata una eccezione.

- Se si desidera avere il controllo delle versioni, è sufficiente inserire il campo **`serialVersionUID`** manualmente.
- Altrimenti java usa un valore di default
  - Per conoscerne il valore si può utilizzare **`serialver`**.

# Controllo di versione

- Riprendiamo l'esempio in cui il client manda al server una istanza di Point
- Il server usa **`Point.java`** versione 1
- Il client usa **`Point.java`** versione 2

# Controllo di versione: esempio



# Quando la default serialization non è abbastanza

- Finora abbiamo visto casi in cui Java non ha bisogno d'aiuto per fare la serializzazione e deserializzazione: basta fornirgli una classe dichiarata serializable.

- Possono esserci casi in cui la serializzazione base fornita da Java non basta.
- Vediamo un esempio.

# PersistentClock

```
import java.io.*;

public class TheMainClock {
  public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException{
    PersistentClock pc = new PersistentClock(1000);
  }
}
```

# PersistentClock

```
import java.io.*;
import java.util.*;
public class PersistentClock extends Thread {
  private long animationSpeed;
  public PersistentClock(int animSpeed) {
    this.animationSpeed = animSpeed;
    start();
  }
  public void run() {
    System.out.println("clock "+ this.getName()+": inizio");
    for(int i=0; i<10; i++) {
      try {
        System.out.print(new Date()+"\r");
        Thread.sleep(animationSpeed);
      } catch (InterruptedException e) { }
    }
    System.out.println("clock"+this.getName()+" terminated");
  }
}
```

# PersistentClock: output

```
clock Thread-0: inizio
Sun May 01 13:14:07 CEST 2022
Sun May 01 13:14:09 CEST 2022
Sun May 01 13:14:10 CEST 2022
Sun May 01 13:14:11 CEST 2022
Sun May 01 13:14:12 CEST 2022
Sun May 01 13:14:13 CEST 2022
Sun May 01 13:14:14 CEST 2022
Sun May 01 13:14:15 CEST 2022
Sun May 01 13:14:16 CEST 2022
Sun May 01 13:14:17 CEST 2022
clockThread-0 terminated
```

# Quando la default serialization non è abbastanza

- Vogliamo serializzare il PersistentClock.
- Procediamo come al solito.

# Quando la default serialization non è abbastanza

```
import java.io.*;
import java.util.*;
public class PersistentClock extends Thread
                              implements Serializable{
  private static final long serialVersionUID = 1;
  private long animationSpeed;
  public PersistentClock(int animSpeed) {
    this.animationSpeed = animSpeed;
    start();
  }
  public void run() {
    System.out.println("clock "+ this.getName()+": inizio");
    for(int i=0; i<10; i++) {
      try {
        System.out.print(new Date()+"\r");
        Thread.sleep(animationSpeed);
      } catch (InterruptedException e) { }
    }
    System.out.println("clock"+this.getName()+" terminated");
} }
```

# Quando la default serialization non è abbastanza

```
import java.io.*;

public class TheMainClock {
public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException{
    new TheMainClock().exec();
  }
}
```

1. Crea il clock
2. Dopo un po' lo salva serializzato
3. Dopo la terminazione del clock, ripristina (deserializzando) la copia salvata

# Quando la default serialization non è abbastanza

```
private void exec() throws IOException, ClassNotFoundException  {
  PersistentClock pc = new PersistentClock(1000);
  try { Thread.sleep(3000); } catch (InterruptedException e) { }
  FileOutputStream  fos = new FileOutputStream("pippo.ser");
  ObjectOutput os = new ObjectOutputStream(fos);
  os.writeObject(pc);
  os.flush(); os.close();
  System.out.println("Main: clock saved");
  try { pc.join(); } catch (InterruptedException e) {  }
  System.out.println("Main: clock terminated, restoring");
  ObjectInput is= new ObjectInputStream(
                          new FileInputStream("pippo.ser"));
  pc = (PersistentClock) is.readObject();
  System.out.println("Main: clock restored");
  is.close();
}
```

cosa succede quando deserializziamo questo oggetto?

# Quando la default serialization non è abbastanza

```
clock Thread-0: inizio
Sun May 01 13:21:14 CEST 2022
Sun May 01 13:21:15 CEST 2022
Sun May 01 13:21:16 CEST 2022
Main: clock saved
Sun May 01 13:21:17 CEST 2022
Sun May 01 13:21:18 CEST 2022
Sun May 01 13:21:19 CEST 2022
Sun May 01 13:21:20 CEST 2022
Sun May 01 13:21:21 CEST 2022
Sun May 01 13:21:22 CEST 2022
Sun May 01 13:21:23 CEST 2022
clockThread-0 terminated
Main: clock terminated, restoring
Main: clock restored
```

La deserializzazione è avvenuta, ma non succede nulla ...

# Quando la default serialization non è abbastanza

- Dove sta il problema?
- Abbiamo ripristinato un oggetto di tipo Thread, ottenendo una copia dell'oggetto originale (un nuovo Thread)
  - In quale stato?
- Lo stato di un nuovo oggetto dipende da ciò che viene fatto nel costruttore, ma quando si deserializza un oggetto il costruttore non viene chiamato.
- Deserializzando un oggetto **`PersistentClock`** non riusciamo a ripristinare il suo stato correttamente perché il costruttore **`public PersistentClock(int animationSpeed)`** non viene chiamato.
  - Quindi non viene fatto lo **`start()`** del Thread.

# Come risolvere il problema «a mano»

```
private void exec() throws IOException, ClassNotFoundException  {
    PersistentClock pc = new PersistentClock(1000);
    try { Thread.sleep(3000); } catch (InterruptedException e) {}
    FileOutputStream  fos = new FileOutputStream("pippo.ser");
    ObjectOutput os = new ObjectOutputStream(fos);
    os.writeObject(pc);
    os.flush();
    os.close();
    System.out.println("Main: clock saved");
    try { pc.join(); } catch (InterruptedException e) {  }
    System.out.println("Main: clock terminated, restoring");
    ObjectInput is= new ObjectInputStream(
                            new FileInputStream("pippo.ser"));
    pc = (PersistentClock) is.readObject();
    pc.start();
    System.out.println("Main: clock restored");
    is.close();
  }
```

# Come risolvere il problema automaticamente

- Possiamo ridefinire la deserializzazione in modo da ottenere automaticamente quel che ci serve?

# Serializzazione personalizzata

- Java consente di personalizzare **`writeObject()`** e **`readObject()`**
- Si possono definire due metodi nelle classi da serializzare:

```
private void writeObject(ObjectOutputStream o)
                                        throws IOException;
private void readObject(ObjectInputStream i)
          throws IOException, ClassNotFoundException;
```

- Se una classe fornisce questi metodi, il meccanismo di serializzazione/deserializzazione li chiama invece di applicare la serializzazione di default

- Aggiungiamo questi metodi a **`PersistentClock`**

# Serializzazione personalizzata

Metodo eseguito nella serializzazione

```
private void writeObject(ObjectOutputStream out)
                                        throws IOException {
    out.defaultWriteObject();
  }
```

Scrive i campi non static e non transient

# Deserializzazione personalizzata

Metodo eseguito nella deserializzazione

```
private void readObject(ObjectInputStream in)
                throws IOException, ClassNotFoundException {
  in.defaultReadObject();
  this.start();
}
```

Leggei i campi non static e non transient salvati precedentemente

Completa il ripristino del PersistentClock facendo lo start del nuovo Thread

# Output con deserializzazione personalizzata

```
clock Thread-0: inizio
Sun May 01 14:18:20 CEST 2022
Sun May 01 14:18:21 CEST 2022
Sun May 01 14:18:22 CEST 2022
Main: clock saved
Sun May 01 14:18:23 CEST 2022
Sun May 01 14:18:24 CEST 2022
Sun May 01 14:18:25 CEST 2022
Sun May 01 14:18:26 CEST 2022
Sun May 01 14:18:27 CEST 2022
Sun May 01 14:18:28 CEST 2022
Sun May 01 14:18:29 CEST 2022
clockThread-0 terminated
Main: clock terminated, restoring
Main: clock restored
clock Thread-1: inizio
Sun May 01 14:18:30 CEST 2022
Sun May 01 14:18:31 CEST 2022
Sun May 01 14:18:32 CEST 2022
...
```

readObject personalizzato contiene **`start()`**

# Gestione degli attributi `static`

- La serializzazione Java li ignora.
- In fase di deserializzazione, il valore dell'attributo resta quello corrente della classe.

- Vediamo come salvare e ripristinare anche gli attributi **`static`**.

# Altro esempio di personalizzazione

```
import java.io.*;
public class TheMain {
  public static void main(String[] args)
      throws IOException, ClassNotFoundException {
    DemoClass p = new DemoClass(false);
    ObjectOutput os = new ObjectOutputStream(
                                  new FileOutputStream("tmp.ser"));
    DemoClass.incSdat();  DemoClass.incSdat();
    System.out.println(p);
    os.writeObject(p);
    os.flush();  os.close();
    DemoClass.incSdat();  DemoClass.incSdat();
    ObjectInput is = new ObjectInputStream(
                                  new FileInputStream("tmp.ser"));
    DemoClass p1 = (DemoClass) is.readObject();
    System.out.println(p1);
    is.close();
  }
}
```

chiama **DemoClass.writeObject**

chiama **DemoClass.readObject**

# Altro esempio di personalizzazione

```
import java.io.*;

public class DemoClass implements Serializable {
  private int _dat=3;
  private static int _sdat=0;
  boolean personalizedSerialization;

  DemoClass(boolean p){
    personalizedSerialization=p;
  }

  public static void incSdat() {
    _sdat++;
  }

  public String toString(){
     return "DemoClass ["+ _sdat+"]"+_dat;
  }
```

static: serializzazione di default non lo salva

# Altro esempio di personalizzazione

```
  private void writeObject(ObjectOutputStream o)
                                              throws IOException {
    o.writeInt(_dat);
    o.writeBoolean(personalizedSerialization);
    if(personalizedSerialization) {
        o.writeInt(_sdat);
    }
  }
  private void readObject(ObjectInputStream i)
                  throws IOException, ClassNotFoundException {
    _dat=i.readInt();
    personalizedSerialization=i.readBoolean();
    if(personalizedSerialization) {
        _sdat=i.readInt();
    }
  }
}
```

# Esecuzione

- Nessuna personalizzazione
  - `DemoClass p = new DemoClass(false)`
- Output

```
DemoClass [2]3
DemoClass [4]3
```

Valore corrente dell'attributo **`static`** nella classe

- Serializzazione personalizzata
  - `DemoClass p = new DemoClass(true)`
- Output

```
DemoClass [2]3
DemoClass [2]3
```

Valore salvato