



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

- L'operazione di scrittura su stream di byte di **un intero oggetto** o di un grafo di oggetti prende il nome di serializzazione

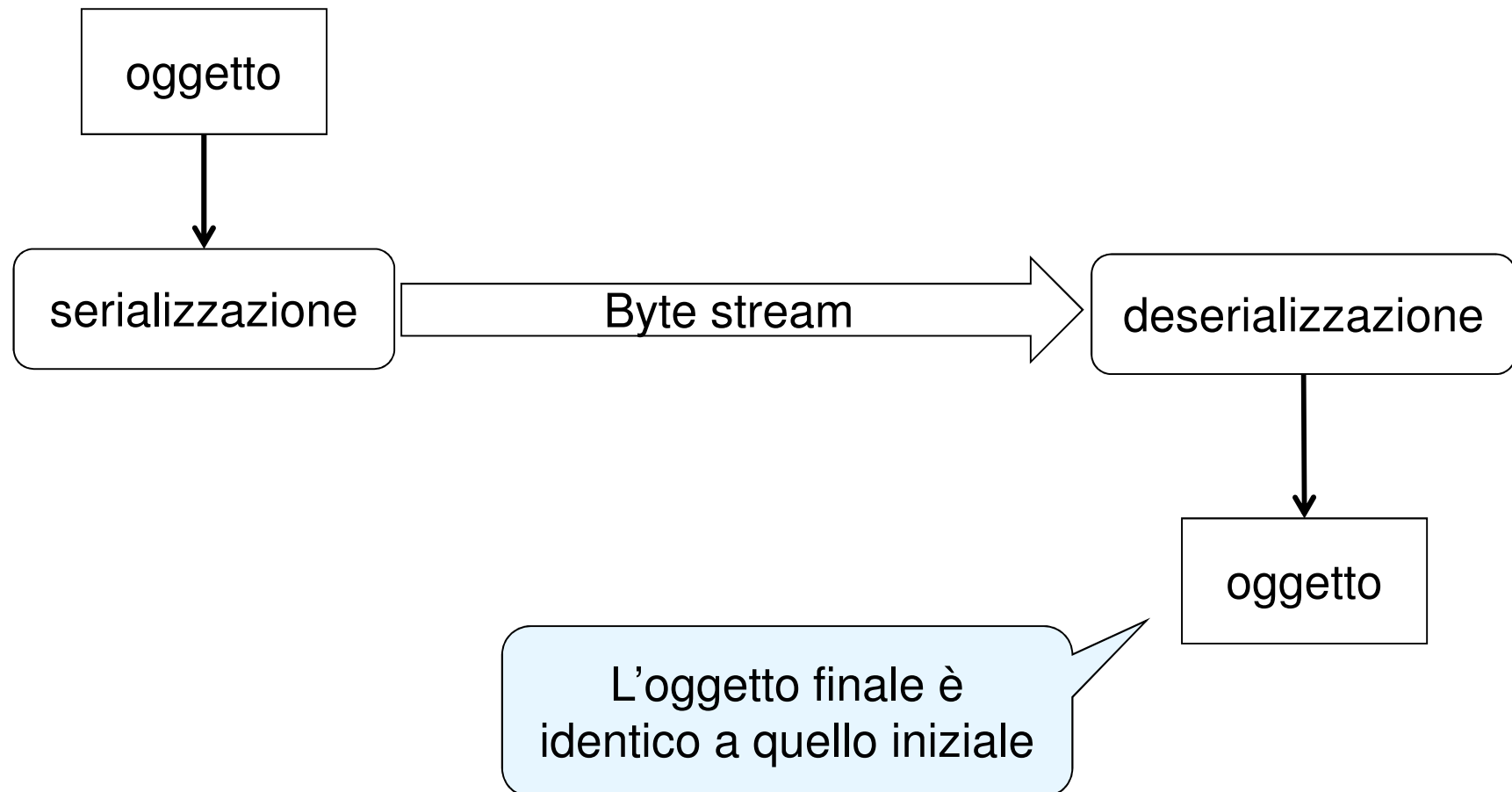


Deserializzazione

- La Deserializzazione è l'operazione inversa, che comporta la lettura da uno stream di byte di un oggetto o del grafo di oggetti 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();  
    }  
}
```

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

```
gigi@hp-850g2-lavazza:~/Tmp$ od -h tmp.bin
00000000 edac 0500 0477 0000 0b00
00000012
gigi@hp-850g2-lavazza:~/Tmp$
```

Il valore della
variabile `int`



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
<code>void</code>	<code>close()</code>	Closes the stream.
<code>void</code>	<code>flush()</code>	Flushes the stream.
<code>void</code>	<code>write(byte[] b)</code>	Writes an array of bytes.
<code>void</code>	<code>write(byte[] b, int off, int len)</code>	Writes a sub array of bytes.
<code>void</code>	<code>write(int b)</code>	Writes a byte.
<code>void</code>	<code>writeObject(Object obj)</code>	Write an object to the underlying storage or stream.



Interface Object Input

Modifier and Type	Method	Description
int	<code>available()</code>	Returns the number of bytes that can be read without blocking.
void	<code>close()</code>	Closes the input stream.
int	<code>read()</code>	Reads a byte of data.
int	<code>read(byte[] b)</code>	Reads into an array of bytes.
int	<code>read(byte[] b, int off, int len)</code>	Reads into an array of bytes.
Object	<code>readObject()</code>	Read and return an object.
long	<code>skip(long n)</code>	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
			114	0x72	r
			115	0x73	s
			116	0x74	t
			117	0x75	u
			118	0x76	v
			119	0x77	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
00000000 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`?

```
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Lezioni$ cat Oggi.txt
Oggi
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Lezioni$ od -h Oggi.txt
00000000 674f 6967 000a
00000005
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Lezioni$ █
```



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
00000000 edac 0500 0074 4f04 6767 7369 0072 6a0e
00000020 7661 2e61 7475 6c69 442e 7461 6865 816a
00000040 4b01 7459 0319 0000 7078 0877 0000 5b01
00000060 516d 6a71 0078
00000065
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, ma vedremo dopo che questo 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
00000000 edac 0500 7273 0500 6f50 6e69 0074 0000
00000020 0000 0000 0201 0200 0049 7801 0049 7901
00000040 7078 0000 0200 0000 0300
00000052
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**



Serializzazione in Java: esempio

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



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 serNum;
    float salary;
    Employee(String name, int no, float sal){
        super(name);
        serNum = no;
        salary = sal;
    }
    void display(){
        System.out.println(name+", "+serNum+", "+salary);
    }
}
```



Main: serializza e deserializza

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

Scrive su file oggetto serializzato

Legge oggetto serializzato da file



Main: serializza e deserializza

```
void employee_to_file() {
    Employee empl=new Employee("Rossi", 100, 3000);
    empl.display();
    ObjectOutputStream output=null;
    try{
        output=new ObjectOutputStream(new
                                   FileOutputStream(fileName));
        output.writeObject(empl);
        output.close();
    }
    catch (FileNotFoundException e) { }
    catch(IOException ioe){ }
    System.out.println("Serializzazione completata.");
}
```



Main: serializza e deserializza

```
void employee_from_file() {
    ObjectInputStream ois = null;
    Employee emp = null;
    try {
        ois = new ObjectInputStream(new FileInputStream(fileName));
    } catch (IOException e) {
        System.err.println("stream creation failed");
        System.exit(0);
    }
    try {
        emp = (Employee) ois.readObject();
    } catch (ClassNotFoundException | IOException e) {
        System.err.println("Employee deserialization failed");
        System.exit(0);
    }
    try {
        ois.close();
    } catch (IOException e) {}
    emp.display();
}
```



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

Riferimento a oggetto non serializzabile:



Riferimento non serializzabile: output

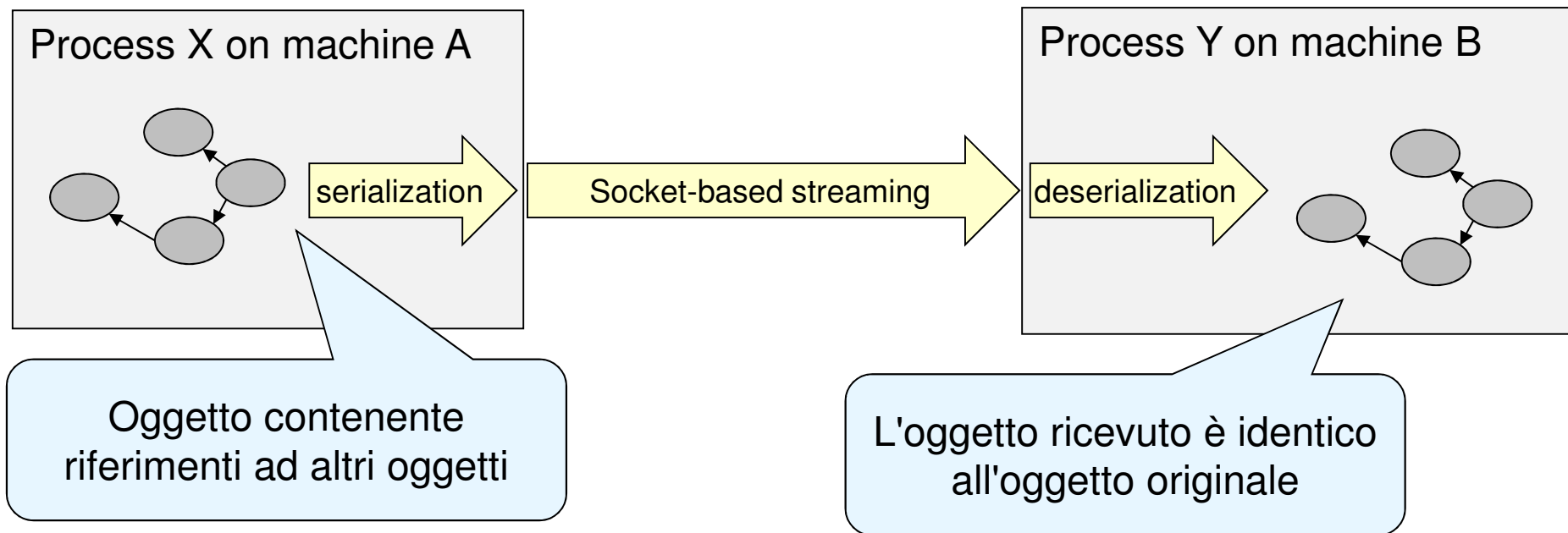
Rossi Anselmo 100 3000.0

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 {
    Employee empl;
    Client() {
        empl=new Employee("Emerenziano Paronzini", 123, 3000);
    }
    void exec() {
        Socket socket=null;
        try {
            InetAddress addr = InetAddress.getByName(null);
            socket = new Socket(addr, 9999);
            ObjectOutputStream obj_out_s =
                new ObjectOutputStream(socket.getOutputStream());
            empl.display();
            obj_out_s.writeObject(empl);
        } catch (IOException e) {}
        finally {
            try { socket.close(); } catch (IOException e) { }
        }
    }
    public static void main(String[] args) {
        new Client().exec();
    }
}
```



Serializzazione Client/Server: Server

```
public class Server {
    Employee obj =null;
    void exec() {
        Socket socket=null;  ServerSocket s=null;
        try {
            s = new ServerSocket(9999);
            socket = s.accept();
            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 {
            try {socket.close(); s.close();} catch (IOException e) {}
        }
    }
    public static void main(String[] args) {
        new Server().exec();
    } }
```



Serializzazione Client/Server: ClassNotFoundException

- Quando si trasferisce un oggetto serializzato si riceve un oggetto binario identico a quello originale.
- Problema: come interpretare la sequenza di Byte?
- Se l'oggetto è di tipo primitivo, la JVM sa cosa fare.
- Altrimenti, bisogna conoscere la classe dell'oggetto
- Se il Client prova ad accedere al “class bytecode” dell’oggetto ricevuto, allora deve avere il bytecode della classe.
- Vediamo un esempio di questo tipo di problema, con eccezione **ClassNotFoundException**



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 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 serializziamo 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 si 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 `Employee`
- Il server usa `Point.java` versione 1
- Il client usa `Point.java` versione 2



Controllo di versione: esempio

```
gigi@gigi-HP-EliteBook-850-G6: ~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassVersionK0/Server$ ls -l
total 16
-rw-rw-r-- 1 gigi gigi 875 apr 29 16:58 Point.class
-rw-rw-r-- 1 gigi gigi 267 apr 28 17:35 Point.java
-rw-rw-r-- 1 gigi gigi 869 apr 29 16:58 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/ClassVersionK0/Server$ java SenderServer
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassVersionK0/Server$
```

serialVersionUID=1

```
gigi@gigi-HP-EliteBook-850-G6: ~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassVersionK0/Client$ ls -l
total 16
-rw-rw-r-- 1 gigi gigi 875 apr 29 17:02 Point.class
-rw-rw-r-- 1 gigi gigi 268 apr 29 16:56 Point.java
-rw-rw-r-- 1 gigi gigi 1710 apr 29 17:02 ReceiverClient.class
-rw-rw-r-- 1 gigi gigi 598 apr 29 17:02 ReceiverClient.java
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassVersionK0/Client$ java ReceiverClient
Object class invalid
gigi@gigi-HP-EliteBook-850-G6:~/Documents/Didattica/Prog_CD/Code/2021/Lez11/ClassVersionK0/Client$
```

serialVersionUID=2



Quando la default serialization non è abbastanza

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



PersistentClock

```
public class PersistentClock implements Runnable{
    private Thread animator;
    private static final long serialVersionUID = 1;
    private long animationSpeed;
    public PersistentClock(int animSpeed) {
        this.animationSpeed = animSpeed;
        animator = new Thread(this);
        animator.start();
    }
    public void run() {
        while (true) {
            try {
                System.out.print(new Date()+"\r");
                Thread.sleep(animationSpeed);
            } catch (InterruptedException e) {}
        }
    }
    public static void main(String[] args) throws IOException{
        PersistentClock p = new PersistentClock(1000);
    }
}
```



PersistentClock: output

```
Thu Apr 29 18:38:31 CEST 2021
Thu Apr 29 18:38:32 CEST 2021
Thu Apr 29 18:38:33 CEST 2021
Thu Apr 29 18:38:34 CEST 2021
Thu Apr 29 18:38:35 CEST 2021
Thu Apr 29 18:38:36 CEST 2021
Thu Apr 29 18:38:37 CEST 2021
Thu Apr 29 18:38:38 CEST 2021
Thu Apr 29 18:38:39 CEST 2021
```



Quando la default serialization non è abbastanza

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



Quando la default serialization non è abbastanza

```
public class PersistentClock implements Serializable, Runnable{
    private static final long serialVersionUID = 1;
    transient private Thread animator; // non serializzare
    private long animationSpeed;
    public PersistentClock(int animSpeed) {
        this.animationSpeed = animSpeed;
        animator = new Thread(this);
        animator.start();
    }
    public void run() {
        while (true) {
            try {
                System.out.print(new Date()+"\r");
                Thread.sleep(animationSpeed);
            } catch (InterruptedException e) { }
        }
    }
}
```

Senza transient
si ha errore



Quando la default serialization non è abbastanza

```
public class TheSaver {
    String fileName;
    PersistentClock pc;
    TheSaver(String s){
        pc = new PersistentClock(1000);
        fileName=s;
    }
    void savePersistentClock() {
        FileOutputStream fos;
        try {
            fos = new FileOutputStream(fileName);
            ObjectOutputStream os = new ObjectOutputStream(fos);
            os.writeObject(pc);
            os.flush();
            os.close();
        } catch (IOException e) {
            System.err.println("KO"); System.exit(0);
        }
    }
}
```

cosa succederà quando
deserializzeremo questo oggetto?



Quando la default serialization non è abbastanza

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

    TheSaver ts=null;
    if(args.length==1) {
        ts=new TheSaver(args[0]);
    } else {
        ts=new TheSaver("../tmp.ser");
    }
    ts.savePersistentClock();
    Thread.sleep(10000);
    System.out.println("Shutdown");
    System.exit(0);
}
}
```



Quando la default serialization non è abbastanza

```
public class TheRestorer {
    String fileName;
    PersistentClock pc;
    TheRestorer(String s){ fileName=s; }
    void restorePersistentClock() {
        ObjectInput is;
        try {
            is = new ObjectInputStream(new FileInputStream(fileName));
            pc = (PersistentClock) is.readObject();
            System.out.println("Persistent clock restored");
            is.close();
        } catch (IOException | ClassNotFoundException e) { }
    }
    public static void main(String[] args) throws IOException{
        TheRestorer tr=null;
        if(args.length==1) { tr=new TheRestorer(args[0]); }
        else { tr=new TheRestorer("../tmp.ser"); }
        tr.restorePersistentClock();
    } }
```

cosa succede?



Quando la default serialization non è abbastanza

- Cosa succede quando deserializziamo il persistent clock?
- L'oggetto viene ricostruito, ma non l'animatore.
- In pratica la ricostruzione è parziale.



Quando la default serialization non è abbastanza

- Nel caso visto, lo stato dell'oggetto dipende fortemente da ciò che viene fatto nel costruttore.
- Il problema è che quando si deserializza un oggetto il costruttore non viene chiamato.
 - ▶ Come giusto, il costruttore dell'oggetto **PersistentClock** viene chiamato solo quando viene creata una nuova istanza di una classe.
- Se deserializziamo un oggetto **PersistentClock** non riusciamo a ripristinare il suo stato correttamente perché il costruttore **public PersistentClock(int animationSpeed)** non viene chiamato.
 - ▶ Quindi non viene ripristinato l'animatore, che originalmente veniva lanciato dal costruttore.



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();
    startAnimation();
}
```

Legge i campi non static e non transient salvati precedentemente

```
private void startAnimation() {
    animator = new Thread(this);
    animator.start();
}
}
```

Completa il ripristino del PersistentClock attivando l'animator



Output con deserializzazione personalizzata

Persistent clock restored

Thu Apr 29 19:18:04 CEST 2021

Thu Apr 29 19:18:05 CEST 2021

Thu Apr 29 19:18:06 CEST 2021

Thu Apr 29 19:18:07 CEST 2021

Thu Apr 29 19:18:08 CEST 2021

Thu Apr 29 19:18:09 CEST 2021

Thu Apr 29 19:18:10 CEST 2021

Thu Apr 29 19:18:11 CEST 2021

Thu Apr 29 19:18:12 CEST 2021



Altro esempio di personalizzazione

```
public class DemoClass implements Serializable {  
    private int _dat=3;  
    private static int _sdat=2;  
    private void writeObject(ObjectOutputStream o)  
                                throws IOException {  
        o.writeInt(_dat);  
        o.writeInt(_sdat);  
    }  
    private void readObject(ObjectInputStream i)  
        throws IOException, ClassNotFoundException {  
        _dat=i.readInt();  
        _sdat=i.readInt();  
    }  
    public String toString(){  
        return "DemoClass: " + _dat;  
    }  
}
```

static: serializzazione di default non lo salva



Altro esempio di personalizzazione

```
public static void main(String[] args)
    throws IOException, ClassNotFoundException {
    DemoClass p = new DemoClass();
    ObjectOutput os = new ObjectOutputStream(
        new FileOutputStream("tmp.ser"));
    os.writeObject(p); // chiama DemoClass.writeObject
    os.flush();
    os.close();
    ObjectInput is = new ObjectInputStream(
        new FileInputStream("tmp.ser"));
    DemoClass p1 = (DemoClass) is.readObject();
    System.out.print(p1);
    is.close();
}
}
```