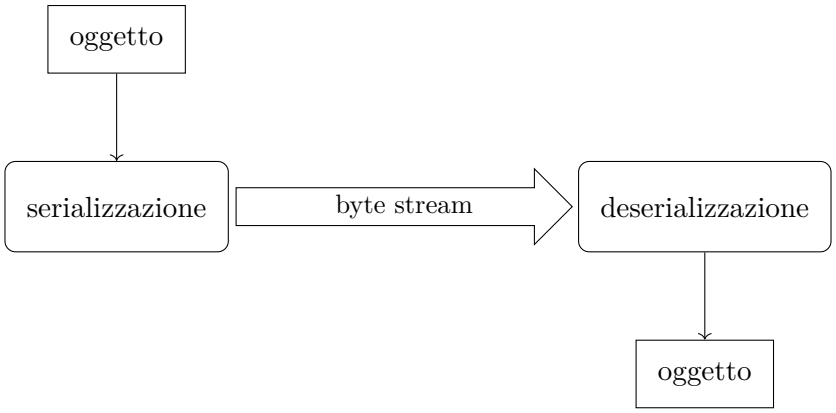
Azzolini Riccardo

Serializzazione

1 Serializzazione e deserializzazione

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

La **deserializzazione** è l'operazione inversa, che comporta la lettura da uno stream di byte di un oggetto o grafo di oggetti serializzato e la ricostruzione dell'oggetto originale (in particolare, viene ripristinato lo stato di quest'ultimo, ovvero i valori degli attributi).



Grazie a queste due operazioni, gli oggetti (che non hanno una struttura sequenziale) possono essere salvati all'interno di file, trasferiti sulla rete tramite socket, ecc.

2 Serializzazione di un tipo primitivo

In Java, la scrittura di tipi primitivi su uno stream di byte avviene semplicemente usando gli appositi metodi della classe `ObjectOutputStream`:

```
import java.io.*;

public class PrimitiveSerialization {
    public static void main(String[] args) throws IOException {
```

```
        String fileName = "tmp.bin";
        try (
            ObjectOutput os =
                new ObjectOutputStream(new FileOutputStream(fileName))
        ) {
            int i = 11;
            os.writeInt(i);
            os.flush();
        }
    }
}
```

I contenuti del file `tmp.bin` creato da questo programma, qui mostrati in esadecimale, sono:

```
ac ed 00 05 77 04 00 00 00 0b
```

Oltre ai 4 byte del valore intero 11 (`00 00 00 0b`), sono presenti delle informazioni aggiuntive, che servono all'operazione di deserializzazione per determinare come ricostruire i valori originali dai dati serializzati.

Analogamente alla serializzazione, la deserializzazione di un tipo primitivo consiste semplicemente nell'invocazione di un apposito metodo della classe `ObjectInputStream`:

```
import java.io.*;

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

3 Serializzazione di un oggetto

Come altro esempio, il seguente codice serializza e deserializza un oggetto – in particolare, una stringa:

```
import java.io.*;

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

        try (
            ObjectOutput os =
                new ObjectOutputStream(new FileOutputStream(fileName))
        ) {
            os.writeObject("Oggi");
            os.flush();
        }

        try (
            ObjectInput is =
                new ObjectInputStream(new FileInputStream(fileName))
        ) {
            String s = (String) is.readObject();
            System.out.println("Read: " + s);
        }
    }
}
```

Anche in questo caso il file `tmp.bin` contiene, prima della codifica della stringa (`4f 67 67 69`), alcune informazioni aggiuntive necessarie alla deserializzazione:

```
ac ed 00 05 74 00 04 4f 67 67 69
```

4 Serializzazione di più oggetti

È anche possibile serializzare più oggetti su uno stream. In tal caso, quando poi si usa `readObject()` per effettuare la deserializzazione, bisogna sapere in che ordine sono stati scritti i dati per convertirli ai tipi corretti (in questo esempio, prima `String` e poi `Date`).

```
import java.io.*;
import java.util.Date;

public class ObjectsSerialization {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
```

```
        String fileName = "tmp.bin";

        try (
            ObjectOutput os =
                new ObjectOutputStream(new FileOutputStream(fileName))
        ) {
            os.writeObject("Oggi");
            os.writeObject(new Date());
            os.flush();
        }

        try (
            ObjectInput is =
                new ObjectInputStream(new FileInputStream(fileName))
        ) {
            String s = (String) is.readObject();
            Date d = (Date) is.readObject();
            System.out.println(s + " " + d);
        }
    }
}
```

Già solo con due oggetti, il contenuto del file `tmp.bin` comincia a diventare piuttosto complicato:

```
ac ed 00 05 74 00 04 4f 67 67 69 73 72 00 0e 6a
61 76 61 2e 75 74 69 6c 2e 44 61 74 65 68 6a 81
01 4b 59 74 19 03 00 00 78 70 77 08 00 00 01 72
75 28 90 40 78
```

5 Interfacce `ObjectOutput` e `ObjectInput`

L'interfaccia `ObjectOutput`, già vista negli esempi precedenti, consente di scrivere (serializzare) tipi primitivi e oggetti. Oltre alle "solite" operazioni di `flush` e `close`, essa mette a disposizione:

- vari metodi per scrivere byte:

  ```
  void write(int b) throws IOException;
  void write(byte[] b) throws IOException;
  void write(byte[] b, int off, int len) throws IOException;
  ```

- metodi per scrivere valori di tipi primitivi:

```
void writeInt(int v) throws IOException;
void writeDouble(double v) throws IOException;
// altri...
```

- un metodo per la scrittura di oggetti:

  ```
  void writeObject(Object obj) throws IOException;
  ```

Analogamente, l'interfaccia `ObjectInput` permette la lettura (deserializzazione) di tipi primitivi e oggetti. Essa fornisce dei metodi di lettura analoghi a quelli di scrittura presenti in `ObjectOutput`,

```
int read() throws IOException;
int read(byte[] b) throws IOException;
int read(byte[] b, int off, int len) throws IOException;
int readInt() throws IOException;
double readDouble() throws IOException;
// altri...
Object readObject() throws ClassNotFoundException, IOException;
```

un metodo `close`, e, inoltre:

- `skip`, per saltare `n` byte di input:

  ```
  long skip(long n) throws IOException;
  ```

- `available`, che indica quanti byte possono essere letti senza bloccarsi (in altre parole, quanti byte sono immediatamente disponibili):

  ```
  int available() throws IOException;
  ```

Osservazione: Siccome il tipo restituito da `readObject` è `Object`, spetta al programmatore convertire l'oggetto deserializzato al tipo giusto.

Le interfacce `ObjectOutput` e `ObjectInput` sono implementate rispettivamente dagli stream di byte `ObjectOutputStream` e `ObjectInputStream`.

6 Serializzazione di un array di oggetti

Anche gli array (in questo esempio, un array di oggetti) possono essere serializzati con `writeObject` e deserializzati con `readObject`:

```
import java.io.*;
import java.util.Arrays;

public class StringArray {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
```

```
        String fileName = "tmp.bin";

        try (
            ObjectOutput os =
                new ObjectOutputStream(new FileOutputStream(fileName))
        ) {
            String[] array = new String[]{"rosso", "giallo", "blu"};
            os.writeObject(array);
            os.flush();
        }

        try (
            ObjectInput is =
                new ObjectInputStream(new FileInputStream(fileName))
        ) {
            String[] array = (String[]) is.readObject();
            System.out.println(Arrays.toString(array));
        }
    }
}
```

7 Serializzazione di tipi definiti dall'utente

Si consideri la seguente classe `Punto`, che rappresenta un punto nello spazio euclideo a due dimensioni, mediante due variabili di istanza `x` e `y`:

```
public class Punto {
    private int x, y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "Il punto ha coordinate " + x + " e " + y;
    }
}
```

Per renderla serializzabile, bisogna "attrezzarla" appositamente:

```
import java.io.Serializable;
```

```
public class Punto implements Serializable {
    private static final long serialVersionUID = 1;
    private int x, y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "Il punto ha coordinate " + x + " e " + y;
    }
}
```

- La classe deve implementare l'interfaccia `Serializable` – ma non è necessario implementare alcun metodo relativo a tale interfaccia, tranne che in alcuni casi particolari.
- Bisogna definire una costante, chiamata `serialVersionUID`, il cui valore funge da numero di versione per la classe. Esso serve in fase di deserializzazione, per verificare che le classi usate da chi ha serializzato l'oggetto e chi lo sta deserializzando siano compatibili: se le versioni non corrispondono, verrà sollevata una `InvalidClassException`.

  Tecnicamente, la definizione di questa costante è facoltativa, ma, se omessa, la JVM calcola un numero di versione in modo autonomo, e ciò potrebbe portare a errori.

Una volta fatto ciò, le istanze della classe possono essere

- serializzate con `writeObject`:

```
import java.io.*;

public class SerializzaPunto {
    public static void main(String[] args) throws IOException {
        try (
            ObjectOutput os = new ObjectOutputStream(
                new FileOutputStream("dati.dat")
            )
        ) {
            Punto p = new Punto(2, 3);
            os.writeObject(p);
            os.flush();
        }
        System.out.println("Serializzazione completata.");
```

```
    }
}
```

- deserializzate con `readObject`:

```
import java.io.*;

public class DeserializzaPunto {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {
        try (
            ObjectInput is = new ObjectInputStream(
                new FileInputStream("dati.dat")
            )
        ) {
            Punto p = (Punto) is.readObject();
            System.out.println(p);
        }
    }
}
```

L'esecuzione del programma di serializzazione crea il file `dati.dat`, che contiene:

```
ac ed 00 05 73 72 00 05 50 75 6e 74 6f 00 00 00
00 00 00 00 01 02 00 02 49 00 01 78 49 00 01 79
78 70 00 00 00 02 00 00 00 03
```

Poi, il programma di deserializzazione legge tale file, ricostruisce l'oggetto originale, e lo stampa (mediante il metodo `toString` di `Punto`):

```
Il punto ha coordinate 2 e 3
```

8 Regole di serializzazione in Java

Complessivamente, le regole a cui è soggetta la serializzazione in Java sono le seguenti:

- Tutti i tipi primitivi sono serializzabili.
- Un oggetto è serializzabile se la sua classe o la sua superclasse implementano l'interfaccia `Serializable`.
- Una classe può implementare `Serializable` anche se la sua superclasse non è a sua volta serializzabile, purché tale superclasse abbia un costruttore senza argomenti.

- Se si desidera serializzare solo alcuni campi di una classe, si contrassegnano quelli da *non* serializzare con l'apposito modificatore `transient`.
- I campi `static` di una classe non vengono serializzati (in quanto parte della classe, e non dell'oggetto, mentre la serializzazione riguarda appunto gli oggetti).
- Se i campi di un oggetto serializzabile contengono un riferimento a un oggetto non serializzabile, la serializzazione solleverà una `NotSerializableException`.

8.1 Esempio: superclasse non serializzabile

```
import java.io.Serializable;

class Person { // Non serializzabile
    String name;

    Person() { name = ""; } // Costruttore senza argomenti
    Person (String name) { this.name = name; };
}

class Employee extends Person implements Serializable {
    private static final long serialVersionUID = 1;
    private int number;
    private double salary;

    Employee(String name, int number, double salary) {
        super(name);
        this.number = number;
        this.salary = salary;
    }

    void display() {
        System.out.println(name + " " + number + " " + salary);
    }
}
```

In questo esempio, se la superclasse non avesse il costruttore senza argomenti `Person()`, la deserializzazione delle istanze di `Employee` fallirebbe (in particolare, verrebbe sollevata una `InvalidClassException`).

8.2 Esempio: riferimento a un oggetto non serializzabile

```
import java.io.Serializable;
```

```
class Person { // Non serializzabile
    String name;

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

class Employee implements Serializable {
    private static final long serialVersionUID = 1;
    private int number;
    private double salary;
    private Person parent; // Riferimento a oggetto non serializzabile

    Employee(String name, int number, double salary) {
        this.number = number;
        this.salary = salary;
        this.parent = new Person(name);
    }

    void display() {
        System.out.println(parent.name + " " + number + " " + salary);
    }
}
```

In quest'altro esempio, la classe `Person` non è serializzabile, e la classe `Employee` contiene un attributo di tipo `Person`, ovvero un riferimento a un oggetto non serializzabile. Di conseguenza, la serializzazione delle istanze di `Employee` fallisce, sollevando una `NotSerializableException`.