Java Object Serialization

- Quando un client invoca un metodo di un server remoto gli argomenti vengono trasmessi dal client al server, mentre i risultati vengono trasmessi in senso inverso.

- Prima di essere trasmessi, sia gli argomenti che i risultati vengono *codificati*.

- Similmente, alla ricezione, prima di essere utilizzati, sia gli argomenti che i risultati devono essere *decodificati*.

- Il processo di codifica viene chiamato *marshalling* mentre quello di decodifica viene chiamato *unmarshalling*.

- Se gli argomenti o i risultati di una invocazione remota contengono oggetti (locali), allora RMI utilizza un processo di *serializzazione* di oggetti (object serialization).

- La *serializzazione* serve a convertire un insieme di oggetti, che contengono referenze ad altri oggetti, in un *flusso* (stream) di bytes.

- Tale flusso può essere spedito in rete attraverso un socket, può essere memorizzato in un file, o semplicemente essere manipolato per elaborazioni successive.

- Si noti che attraverso la serializzazione è possibile salvare in un file lo stato di un oggetto. Stato che può essere recuperato in un secondo momento, eventualmente dopo l'esecuzione dell'applicazione che ha creato l'oggetto stesso.

L'interfaccia Serializable

- Tutti gli oggetti (locali) che vengono utilizzati come argomenti o risultati di invocazioni remote *devono essere serializzabili.* Più precisamente devono osservare le seguenti condizioni:

  – La classe di un oggetto serializzabile deve implementare l'interfaccia `java.io.Serializable` (oppure la sua estensione `java.io.Externalizable`);

– Il punto precedente riguarda la parte di stato definita localmente nella classe in questione. Ma come ci si comporta con quelle *variabili* che vengono ereditate da una o più superclassi?

  * Se la superclasse implementa anch'essa `Serializable` allora non vi sono problemi.
  * Altrimenti, bisogna informare direttamente il meccanismo di serializzazione sulle variabili della superclasse usando i metodi `ObjectOutputStream.writeObject()` e `ObjectInputStream.readObject()` per gestire direttamente la serializzazione e la deserializzazione. In tal caso le superclassi non-serializzabili devono avere un costruttore di default senza argomenti che sia accessibile dalla classe in questione (se la più vicina superclasse è `Object` allora il problema non si pone poichè esiste gia il costruttore `Object()` e la classe `Object()` non ha variabili da gestire).
  * Un'alternativa a tale procedura è quella di dichiarare la

classe in questione `Externalizable` (chi è interessato può approfondire).

– Le variabili di tipo oggetto di una classe serializzabile devono:

  * o riferirsi ad un oggetto che sia anch'esso serializzabile
  * o essere dichiarate come `static` o `transient`.
  * oppure rappresentare un oggetto nullo; si noti che è l'oggetto che deve essere serializzabile al momento dell'invocazione remota, non la sua classe.

– Se utilizzati è necessario ridefinire i metodi `equals()` e `hashCode` ereditati dalla classe `java.lang.Object`.

– Una classe serializzabile non deve necessariamente implementare dei metodi visto che l'interfaccia `Serializable` è vuota.

• Se durante la fase di marshalling o unmarshalling viene incontrata (direttamente o indirettamente) un oggetto non serializzabile allora viene lanciata un'eccezione `NotSerializableException` che si manifesta presso il client sotto forma di `MarshalException` oppure `UnmarshalException`, a seconda se il problema appaia nella codifica degli argomenti o nella decodifica dei risultati.

- Si noti che codificare un oggetto in un file è cosa relativamente semplice. Mentre è tutt'altro che banale ricostruire un oggetto, possibilmente su una macchina differente, a partire da un file.

- Ad esempio, in C e C^{++} la serializzazione/deserializzazione è abbastanza gravosa per il programmatore che deve gestire tutta una serie di variabili.

- In Java invece, tutte queste problematiche vengono gestite dal sistema di serializzazione.

Vediamo adesso un 'po più in dettaglio cosa avviene nella serializzazione quando si intende passare un oggetto.

Grafi di oggetti

- Quando si serializza un oggetto, si costruisce un grafo di cui l'oggetto in questione è la radice. Si aggiungono poi tutti gli oggetti a cui l'oggetto può fare riferimento, e così via ricorsivamente.

- In un grafo di oggetti un arco tra un oggetto A e uno B indica che l'oggetto A può invocare un metodo dell'oggetto B, e quindi ha una variabile del tipo della classe di B.

- Dato un oggetto appartenente ad un grafo di oggetti il suo *in-degree* è dato dal numero di archi incidenti nell'oggetto; l'*out-degree* il numero archi uscenti dall'oggetto.

- La serializzazione e la successiva deserializzazione devono preservare il grafo degli oggetti associato ad un oggetto serializzabile.

- Gli oggetti di un grafo sono tutti distinti.

- Si noti, che un oggetto può essere riferito più di una volta (in-degree > 1).

- Perciò la costruzione del grafo degli oggetti richiede l'uso di memoria per evitare di codificare due volte un medesimo oggetto.

Processo di Serializzazione

- La serializzazione può essere fatta implicitamente dal sistema RMI, durante il marshalling, oppure può essere gestita direttamente dal programmatore utilizzando il metodo `ObjectOutputStream.writeObject()`.
- Quando un oggetto viene serializzato, si memorizza su uno stream di output lo stato (cioè le variabili) di tutte le sue superclassi e le variabili proprie dell'oggetto.
- Java assegna un *serial number* a ciascun oggetto del grafo.
- Le variabili `static` e `transient` vengono ignorate poichè la serializzazione è intesa per trasmettere lo stato dell'oggetto e non della sua classe.
- Il processo di serializzazione viene applicato ricorsivamente a tutti gli elementi del grafo.

- Ogni oggetto del grafo viene serializzato *una volta sola*; successivi riferimenti all'oggetto sono codificati attraverso un meccanismo di condivisione delle referenze, utilizzando i serial numbers.

- Comunque, ad essere più precisi, l'informazione memorizzata nello stream è ben di più che il grafo degli oggetti coinvolti. Infatti, oltre agli oggetti, vengono memorizzate anche le segnature delle classi relative.

- Ma di quali classi stiamo parlando? Se, ad esempio, a è un oggetto di una classe A_n. In generale, esiste una relazione di sottoclasse che conduce fino alla classe `Object`:

  ```
  A_1(Object) > A_2 > ...... > A_n.
  ```

- Allora, per ciascun oggetto, oltre al grafo degli oggetti, viene memorizzato nello stream la seguente informazione:

  - La descrizione della classe più vicina all'instanza, ovvero A_n
  - Le variabili dell'oggetto visto come un'istanza di A_{n-1}

- Le variabili dell'oggetto visto come un'istanza di A_{n-2}
- ... e così via fino ad arrivare ad A_1.

Nella *descrizione della classe* sono riportate tutta una serie di informazioni come:

- versione ID della classe (un intero usato per validare i file .class);
- Un booleno ad indicare se sono stati usati i metodi `writeObject()` e `readObject()`;
- numero dei campi serializzabili;
- descrizione dei metodi e delle variabili (nome e tipo);
- una decrizione della superclasse (se serializzabile);
- eventuali informazione prodotte dall'invocazione del metodo `ObjectOutputStream.annotateClass` (come vedremo in seguito tale informazione è cruciale in Java RMI).

Quindi, **non solo lo stato dell'oggetto, ma anche il suo tipo viene memorizzato**.

Processo di Deserializzazione

- La deserializzazione può essere fatta implicitamente dal sistema RMI, durante l'unmarshalling, oppure può essere gestita direttamente dal programmatore utilizzando il metodo `ObjectInputStream.readObject()`.

- **IMPORTANTE:** Durante la serializzazione si può lanciare un eccezione `IOException` oppure `ClassNotFoundException` se non è possibile caricare (dinamicamente tramite codebase) la classe corrispondente ad un oggetto del grafo che viene deserializzato.

- Quando si deserializza un oggetto, viene ricostruito il grafo degli oggetti per ricostruire la causalità delle invocazioni.

- Le uniche differenze che vi sono tra un oggetto prima e dopo la serializzazione sono:

  – il metodo `hashcode()` (se non è stato reimplementato)

  – l'assenza dei campi `static` o `transient`

– il comportamento sotto gli operatori `==` e `!=`; l'oggetto originale e la sua deserializzazione hanno differenti chiavi in `java.util.Hashtable` e quindi risultano differenti rispetto al metodo `equals()`.