UnicastRemoteObject

- RMI fornisce diverse classi base per definire server remoti:

  ```
  UnicastRemoteObject < RemoteServer < RemoteObject
  ```

  dove $A < B$ significa che A è una sottoclasse di B.

- `UnicastRemotObject` è una classe del package `java.rmi.server` ed è la classe più usata per implementare server remoti.

- `UnicastRemoteObject` supporta server remoti per comunicazioni point-to-point (e non broadcast o multipoint).

- **IMPORTANTE:** Referenze remote (stub) a server unicast sono valide solo fintantochè è in esecuzione il processo che ha creato l'oggetto remoto.

- Ricordiamo che la realizzazione di un server remoto richiede sostanzialmente tre passi:

  - Implementazione di una o più interfacce remote
  - Definizione di una classe che modelli il comportamento del server
  - Esportazione di un'istanza di tale classe.

Esistono tre modi per definire un server remoto unicast:

1. Estendendo `UnicastRemoteObject`:

```
public class ServerRemoto extends UnicastRemoteObject
                           implement IntRemota {
   public ServerRemoto() throws RemoteException {
     super(); // E' qui che il server viene esportato
    .....
   }
   .....
   public void main(String[] args){
      ServerRemoto server = new ServerRemoto();
      Naming.bind(''MioServer'',server);
```

```
    }
}
```

- Il codice di lancio di questo server si occupa solo di (i) creare un'istanza del server remoto e di (ii) registrare una referenza remota del server nel registro RMI.
- Un server unicast viene *esportato automaticamente* al momento della costruzione, ovvero quando viene invocato il suo costruttore.
- Al momento dell'esportazione il server viene registrato col sistema RMI e messo in ascolto su una porta TCP.
- `UnicastRemoteObject` ha tre possibili costruttori:

```
protected UnicastRemoteObject()
protected UnicastRemoteObject(int port)
protected UnicastRemoteObject(int port,
          RMIClientSocketFactory csf,
          RMIServerSocketFactory ssf)
```

- Il primo costruttore crea ed esporta un nuovo oggetto unicast usando una porta anonima, ovvero una porta scelta dal sistema RMI; per economizzare risorse, il sistema RMI potrebbe usare una porta già in uso presso un altro server riutilizzando il medesimo server socket.
- Il secondo ed il terzo costruttore specificano la porta su cui si vuole lanciare il server.
- Il terzo costruttore specifica anche i socket da usare.
- Poichè il server viene esportato automaticamente al momento della costruzione, ne consegue che i costruttori devono poter lanciare `RemoteException`. Discorso simile per i costruttori di quegli oggetti che estendono `UnicastRemoteObject`.
- Un server che estende `UnicastRemoteObject` eredita la semantica per server remoti di `RemoteObject`. Mentre la semantica per cloning e serializzazione viene ereditata da `UnicastRemoteObject`.
- Più precisamente `UnicastRemoteObject` implementa il metodo `Object.clone` clonando l'intero stato del server remo-

to ed esportandolo sulla stessa porta dell'oggetto originale utilizzando però un altro socket.

- Comunque, la classe non implementa l'interfaccia `Cloneable`, questo è lasciato alle sottoclassi.
- I due vantaggi principali di `UnicastRemoteObject` sono:
  (a) La connessione automatica al meccanismo di runtime di RMI per quanto riguarda esportazione (e de-esportazione).
  (b) Una semantica appropriata per testare l'uguaglianza di server remoti.

  D'altrocanto definire un server remoto estendendo direttamente `UnicastRemoteObject` può essere causa di due tipi di problemi:
  (a) In Java è consentito estendere una sola classe, e quindi il server in questione non può estendere altre classi.
  (b) Il server remoto potrebbe essere esposto ad invocazioni remote premature, visto che l'esportazione avviene nel momento stesso della creazione.

2. Un'altro modo per definire server remoti è quello di estendere una classe tra `RemoteServer` o `RemoteObject` al posto di `UnicastRemoteObject`. Ad esempio:

```
public class ServerRemoto extends RemoteServer
                          // oppure RemoteObject
                          implements IntRemota {
   public ServerRemoto() throws RemoteException {
    .....
   }
   ......
   public void main(String[] args){
      ServerRemoto obj = new ServerRemoto();
      ........
      RemoteStub  stub = UnicastRemoteObject.exportObject(obj);
      Naming.bind(``MioServer'',stub);
   }
}
```

Vediamo un 'po meglio queste due classi base.

Remote Object

- `java.rmi.server.RemoteObject` fornisce la semantica per oggetti remoti, *siano essi server o stub*, riscrivendo opportunamente i metodi `hashCode`, `equals`, `toString`.
- I primi due metodi sono implementati per consentire a referenze di `RemoteObject` di essere memorizzate in tabelle hash ed essere confrontate:
  - Il metodo `equals` ritorna `true` se due `RemoteObject`s sono referenze allo stesso server remoto. Nota che l'uguaglianza non è testata confrontando il contenuto dei server remoti perchè ciò richiederebbe due invocazioni remote con relativo trattamento delle `RemoteException`. Se i `RemoteObject`s in questione sono stubs allora `equals` ne ricava gli stub e li confronta.
  - Il metodo `hashCode` ritorna lo stesso valore per tutte le referenze remote che puntano allo stesso `RemoteObject`.
  - Il metodo `toString` ritorna una rappresentazione sotto

forma di stringa del valore la referenza remota allo stesso oggetto.

RemoteServer

- `java.rmi.server.RemoteServer` è una classe abstract per `UnicastRemoteObject` e `Activatable`. Tra l'altro, fornisce i metodi statici `getClientHost` e `getLog` con la seguente segnatura:

```
public static String getClientHost()
                     throws ServerNotActiveException
public static PrintStream getLog()
```

- Il primo ritorna l'hostname del client corrente. Il secondo fornisce uno stream con i log al server RMI.
- Il comportamento dell'operazione di clonazione deve essere definita dal programmatore.
- Indipendentemente dall'uso di `RemoteObject` o `RemoteServer`, il codice di lancio è differente rispetto al caso in cui si estende

`UnicastRemoteObject`. *La creazione del server remoto non comporta l'esportazione automatica del server.* Invece è necessario invocare il metodo statico `UnicastRemoteObject.exportObject` che si occupa di esportare il server remoto. Esistono tre forme dello stesso metodo:

```
static RemoteStub exportObject(Remote obj)
static Remote exportObject(Remote obj, int port)
static Remote exportObject(Remote obj, int port,
      RMIClientSocketFactory csf,
      RMIServerSocketFactory ssf)
```

- Contrariamente a quanto potrebbe sembrare dalla segnatura, questi metodi ritornano tutti `RemoteStub` che possono essere passati come parametro per la `bind` (o `rebind`).
- **IMPORTANTE**: Il metodo `exportObject` può essere invocato dall'oggetto stesso, da chi ha costruito l'oggetto, oppure da chi l'ha ricevuto in una comunicazione locale.

3. Infine è possibile creare una classe remota senza estendere nessuna

delle classi menzionate prima; ma eventualmente estendendo la classe che si preferisce. Ad esempio:

```
public class ServerRemoto extends MiaClasse
                          implement IntRemota {
  public ServerRemoto() throws RemoteException {
    .....
  }
   .....
  public void main(String[] args){
    ServerRemoto obj = new ServerRemoto();
    ............
    RemoteStub  stub = UnicastRemoteObject.exportObject(obj);
    Naming.bind(``MioServer'',stub);
  }
}
```

- In questo caso la semantica dei server remoti, così come il comportamento per la clonazione, è demandata completamente al programmatore.

- Il meccanismo di esportazione è esattamente uguale al caso precedente.
- Inoltre poichè non stiamo estendendo `RemoteObject` il programmatore della classe remota deve prendersi cura di reimplementare i metodi `hashCode`, `equals`, e `toString`, ereditati dalla classe `Object`, in maniera appropriata per oggetti remoti.
- La riscrittura di tali metodi non richiede molto codice. Essenzialmente bisogna preoccuparsi di tre cose:
  - l'host su cui è in esecuzione il server;
  - l'identità della JVM che esegue il server;
  - l'identità dell'oggetto server all'interno della JVM.

Come detto precedentemente se un server estende `UnicastRemoteObject` allora non può estendere altre classi. Quest'inconveniente può essere agevolmente aggirato spezzando il server in due:

1. Un server remoto che estenda `UnicastRemoteObject` ed imple-

menti l'interfaccia remota. Tale implementazione inoltra tutte le richieste ad un secondo server (il vero server).

2. Un secondo server (che eventualmente estende una qualche classe) che implementi anch'essa l'interfaccia remota. Tale server riceverà le richieste dal primo server utilizzando eventualmente anche metodi della superclasse.

Una seconda potenziale limitazione dei server unicast è che possono prematuramente ricevere richieste remote. Ad esempio:

```
public class PrintManager_Impl extends UnicastRemoteObject
                               implements PrinterManager{
   public PrintManager_Impl(){
       super(5150); // porta usuale per print managers.
   // trova tutte le stampanti registrate
   // ottieni la referenza di ciascuna stampante,
   // e ottieni le informazioni sulle loro code
   // cosi' da consentire agli utenti di ricavare
   // informazioni su tutte le stampanti.
```

L'oggetto è esportato subito dopo aver invocato super(5150), ovvero il costruttore di `UnicastRemoteObject`. A tal punto il server potrebbe ricevere invocazioni remote prima che il costruttore abbia terminato il set-up.

Unexporting

- Un server remoto viene de-esportato quando viene de-registrato dal sistema RMI (non dal registro RMI!)
- Un server remoto è automaticamente de-esportato quando viene rimosso dal garbage collector locale.
- Un server remoto può essere rimosso dal garbage collector locale solo se:
    1. Le referenze remote esistenti (presso i clients) sono sono state a loro volta rimosse dal garbage collector delle rispettive JVM.
    2. Non esistono referenze locali al server.

- Di norma si preferisce la de-esportazione automatica, ma in alcuni casi può essere desiderabile forzare la de-esportazione di un server (Ad esempio se il server è oggetto di un attacco da parte di malintenzionati).

- La de-esportazione può essere forzata con il metodo `UnicastRemoteObject.unexport` con la seguente segnatura:

```
public static boolean unexportObject(Remote obj,
      boolean force) throws NoSuchObjectException.
```

- Se il parametro `force` è messo a true allora la de-esportazione viene forzata brutalmente anche durante un'invocazione remota al server.

- L'eccezione `NoSuchObjectException` viene lanciato se l'oggetto non è correntemente esportato.

Threads, Sockets, e Porte

- Durante l'esportazione di un server remoto avvengono un certo numero di cose:

- Viene creato un `ServerSocket` su una porta TCP che ascolta ed accetta invocazioni al server su quella porta.

- Quando il Server socket accetta una richiesta da parte di un client stabilisce una connessione via socket con questi.

- Una volta stabilita la connessione socket col cliente, le invocazioni RMI vengono inoltrate verso il server (ed il metodo in questione) su un thread determinato dall'implementazione.

- **Importante:** Facendo riferimento alla specifica della sun: invocazioni diverse ad un medesimo server remoto possono essere eseguite nello stesso thread oppure in thread differenti.

Server sockets

- Si noti che all'esportazione di un server, il sistema RMI creerà un nuovo Server socket, su una nuova porta TCP, solo se necessario.
- Se possibile, al fine di economizzare risorse, il sistema RMI tenterà di utilizzare un Server socket già in uso.
- Tale condivisione è possibile purchè i due server siano stati esportati dallo stesso host e dalla stessa JVM con:
  - Numero di porta omesso, oppure uguale a 0, oppure con lo stesso numero di porta.
  - Client socket factory omesso, nullo, o uguale.
  - Server socket factory omesso, nullo, o uguale.
- Si noti che in generale, a meno della presenza di firewalls, i server remoti vengono esportati attraverso una porta di default.
- È strettamente necessario avere più porte TCP solo quando un host ha in esecuzione più servers su JVM differenti.

Client sockets

Se un client ha una coneessione via socket con un server remoto, allora può utilizzare la medesima connessione per accedere un secondo server remoto purchè questi sia stato esportato dallo stesso host del primo con:

- Numero di porta omessa, oppure uguale a zero, oppure con lo stesso numero di porta.
- Client Socket Factory omessa, oppure nulla, oppure uguale socket factory

Serializzazione di server remoti

- Server remoti esportati, pur se serializzabili, vengono passati per referenza remota (si passa lo stub). Indipendentemente dal fatto che il server remoto estenda o meno `UnicastRemoteObject`.
- Se però un server unicast serializzabile viene passato dopo essere stato de-esportato allora il passaggio avviene attraverso il proces-

so di serializzazione che, oltre a memorizzare lo stato del server, memorizza anche la porta usata dal server ed i socket utilizzati.

- Il ricevente, al momento della deserializzazione riceverà il server unicast e lo esporterà automaticamente sulla porta passata utilizzando i medesimi socket.

- Questa procedura consente il salvataggio di server RMI su file ed anche mobilità di codice.