RMI: metodi *equals* e *hashCode*

- Per verificare se due oggetti remoti contengono gli stessi dati, la chiamata indirizzata al metodo equals() avrebbe bisogno di contattare i server dove si trovano gli oggetti e comparare i rispettivi contenuti

- La chiamata potrebbe fallire, ma equals della classe Object non è in grado di sollevare un'eccezione di tipo RemoteException

- Di conseguenza in un' interfaccia remota non è possibile definire un metodo equals

- Lo stesso dicasi per hashCode()

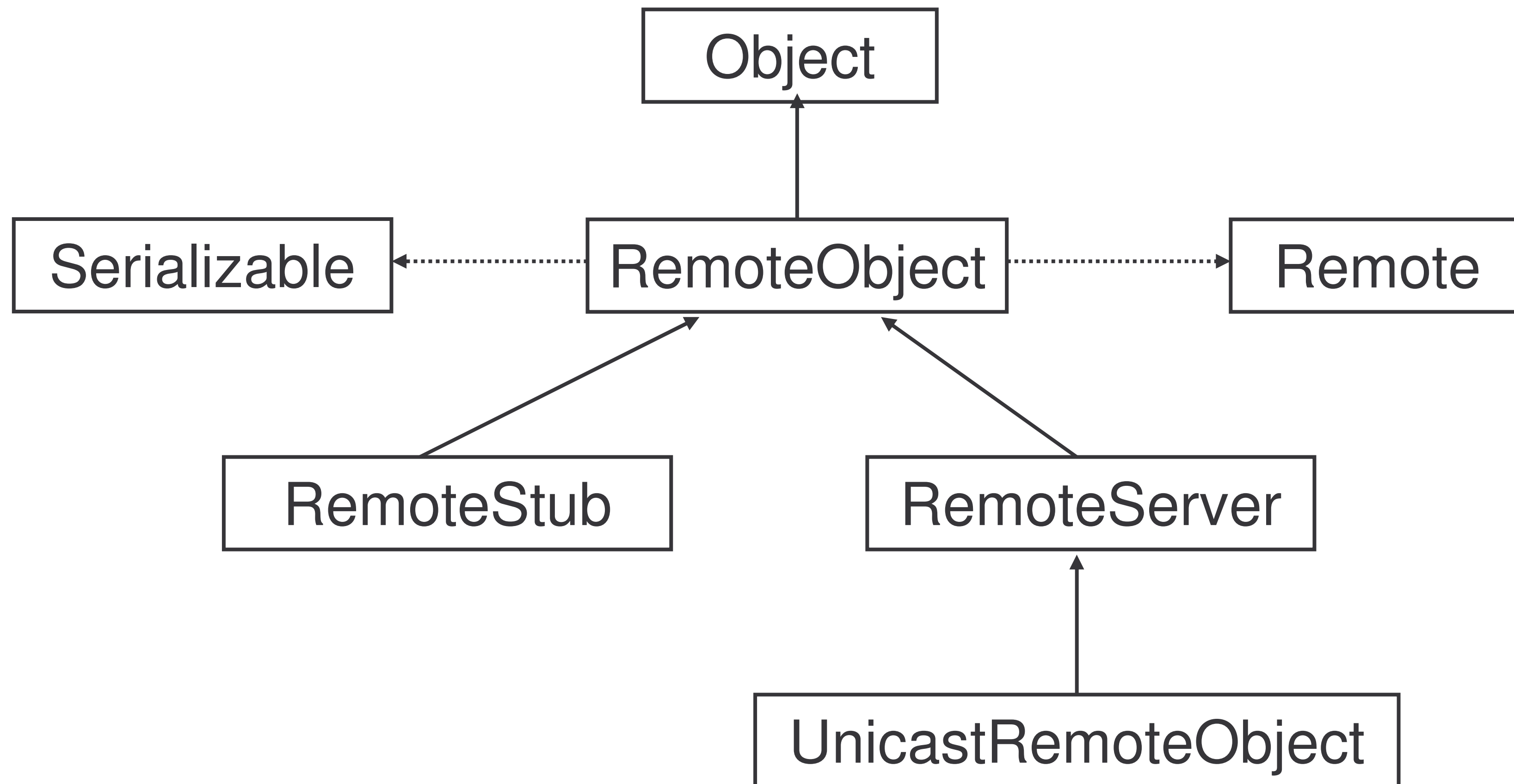
RMI: metodi *equals* e *hashCode*

- E' necessario quindi ridefinire i metodi equals e hashCode della classe RemoteObject, che è superclasse di tutti gli oggetti remoti e stub

- I metodi di RemoteObject non esaminano il contenuto degli oggetti ma solo il riferimento all' oggetto server.

- Se si ridefiniscono i metodi nell'implementazione dell' oggetto remoto, saranno utilizzabili solo localmente sul server ma non avranno effetto sugli stub, poiché essi sono generati automaticamente (continuano ad usare equals di Object)

Struttura gerarchica delle classi ed interfacce RMI



RMI: clone

- clone non può sollevare RemoteException (come equals e hashCode)
- Non ha senso creare un clone di uno stub: per avere un altro riferimento all'oggetto remoto basta duplicare il suo riferimento
- Possiamo definire un metodo remoteClone per l' implementazione dell'oggetto remoto:

```
interface SomeInterface extends java.rmi.Remote
{  public Object remoteClone() throws RemoteException,
        cloneNotSupportedException;
   […]
}

class SomeInterfaceImpl extends UnicastRemoteObject
        implements SomeInterface
{  public Object remoteClone() throws RemoteException,
        cloneNotSupportedException
   {    return clone();
   }
}
```

RMI: parametri remoti inappropriati

Consideriamo un metodo remoto con la seguente interfaccia

```
void paint(Graphics g) throws
RemoteException
```

Considerazioni:

- Graphics (o meglio, le sue sottoclassi) interagiscono con il codice grafico nativo, memorizzando puntatori a blocchi di memoria, che servono a questi metodi. Tali puntatori vengono memorizzati come valori interi nell'oggetto Graphics.
- Questi valori sul server non sarebbero validi
- Graphics non è Serializabile, quindi non può essere inviata via RMI

1. Dynamic class downloading

- Una delle capacità più significative della piattaforma Java è quella di permettere il caricamento a run-time delle classi da un URL (Uniform Resource Locator) a una JVM in esecuzione su un processo separato e di solito su un differente sistema fisico

- Il risultato di tutto questo è che un'applicazione può eseguire un programma (ad esempio un' applet) che era inesistente su disco fino a quel momento

2. Dynamic class downloading

- Per esempio, una JVM in esecuzione all'interno di un Web Browser può scaricare il bytecode per le sottoclassi e per tutte le altre classi necessarie a quell' applet
- Il sistema sul quale è in esecuzione il browser è probabile che non abbia mai eseguito quest'applet prima, e che non l' abbia mai installata neanche su disco
- Una volta che tutte le classi necessarie sono state scaricate dal server, il browser può iniziare l' esecuzione dell'applet usando le risorse locali del sistema sul quale il client browser è in esecuzione.

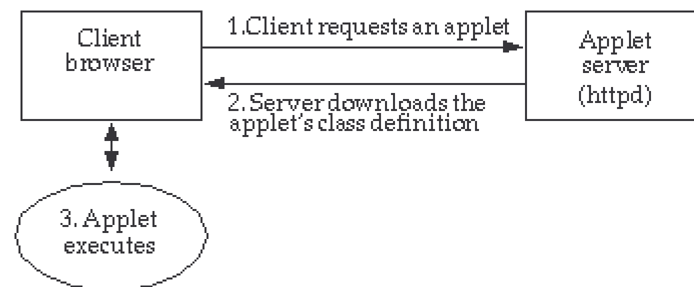


Figure 1: Downloading applets

3. Dynamic class downloading in RMI

- Usando le API RMI una JVM (quindi non solo quelle nei browser) può scaricare classi java, incluse le classi stub RMI, permettendo chiamate a metodi su oggetti remoti.
- Potrebbe essere necessario scaricare anche sottoclassi di classi di cui sono già disponibili gli stub

4. Dynamic class downloading in RMI

- **Quando un programma java usa un Class Loader, il class loader necessita di sapere la locazione (o locazioni) dalla quale caricare le classi necessarie al programma stesso.**
- **Di solito, un class loader è usato insieme ad un server HTTP che mette a disposizioni le classi**
- **Un "*codebase*" può essere definito per indicare la fonte, o il luogo, da cui caricare le classi per una Java Virtual Machine**
- **Si può associare il CLASSPATH come un "local codebase", perchè rappresenta la lista dei luoghi su disco da cui si possono caricare le classi. La variabile CLASSPATH viene consultata ogni volta che si caricano classi da disco.**

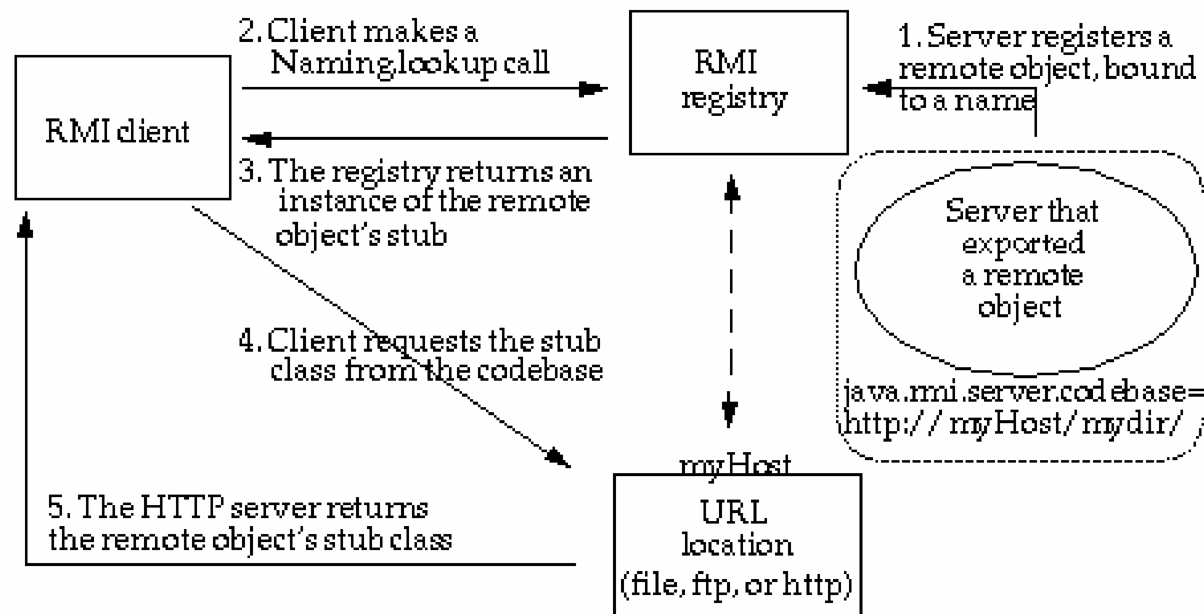
Come viene usato *codebase* in RMI

- Il valore della propietà java.rmi.server.codebase rappresenta una o più locazioni URL dalle quali gli stubs (e altre classi necessarie agli stubs) possono essere caricati.
- Così come le applets, le classi necessarie per eseguire chiamate a metodi remoti possono essere caricate anche da un URLs del tipo "file:///". Questo richiede che il client e il server siano fisicamente sullo stesso host.
- Generalmente, le classi necessarie per eseguire chiamate a metodi remoti dovrebbero essere accessibili attraverso risorse di rete, come un server HTTP o FTP.

Come viene usato *codebase* in RMI

- **Prima di far partire il server, è necessario che l' RMI registry sia in esecuzione (attraverso il comando rmiregistry)**
- **Dobbiamo assicurarci che la shell nella quale verrà eseguito rmirgistry non abbia la variabile ambiente CLASSPATH settata o una variabile CLASSPATH settata che non include il path delle classi stubs dell'oggetto remoto che vuoi rendere disponibili ai client.**
- **Se si esegue rmiregistry ed esso può cercare le classi stub nel CLASSPATH (o nella directory corrente), esso non ricorderà che le classi stub possono essere caricate dal codebase del server, specificato attraverso la proprietà java.rmi.server.codebase nel momento in cui viene avviata l'applicazione server**

Come viene usato *codebase* in RMI



Come viene usato *codebase* in RMI

1. **Il codebase dell'oggetto remoto è specificato dal server dell' oggetto remoto attraverso la proprietà *java.rmi.server.codebase***

   ```
   java -Djava.rmi.server.codebase=http://webserver/export/
   ```

2. **Il server RMI registra un oggetto remoto con l' rmiregistry**

3. **Il codebase settato sulla JVM del server è annotato al riferimento dell' oggetto remoto nell' RMI registry**

Come viene usato *codebase* in RMI

4. **Il client RMI chiede un riferimento ad un determinato oggetto remoto. Il riferimento (istanza dello stub dell'oggetto remoto) è quello che il client userà per fare chiamate ai metodi dell'oggetto remoto.**

5. **RMI registry ritorna un riferimento (istanza dello stub) alla classe richiesta**

6. **Se la classe stub può essere trovata localmente nel CLASSPATH, in cui viene sempre cercato prima di cercare nel codebase, il client caricherà la classe localmente**

Come viene usato *codebase* in RMI

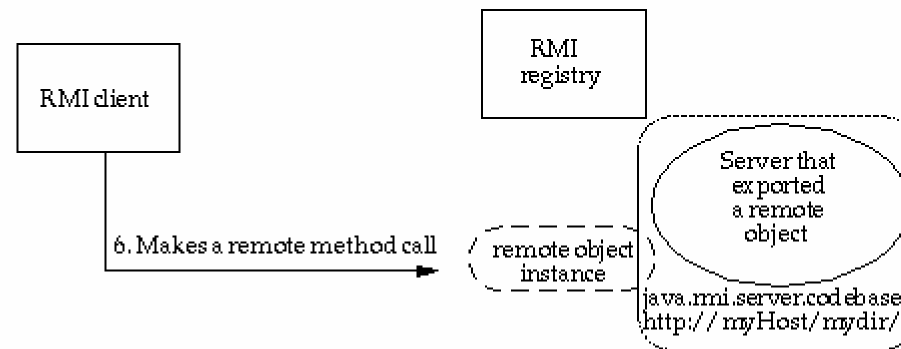
7. Se la classe stub non viene trovata nel CLASSPATH del client, il client tenterà di recuperare la classe stub dal codebase dell'oggetto remoto

8. Il client richiede la classe stub attraverso il codebase. Il codebase che il client usa è l' URL che era stato annotato all'istanza dello stub quando la classe stub è stata caricata dal registry

9. La definizione della classe stub (e altre classi che sono necessarie) è caricata dal client

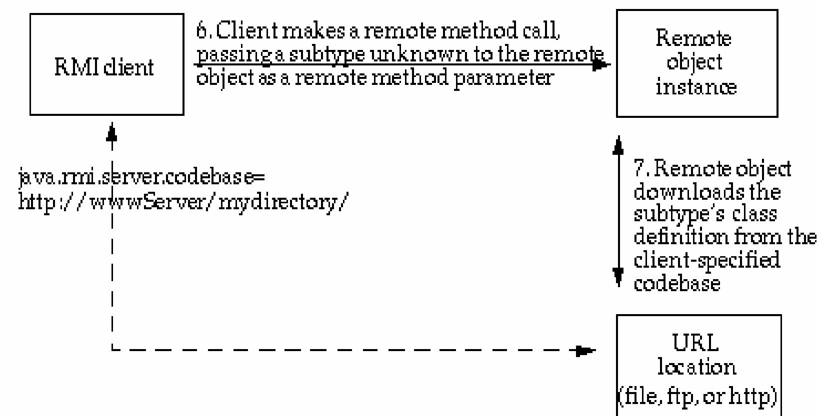
Come viene usato *codebase* in RMI

10. **Ora il client ha tutte le informazioni necessarie per invocare metodi sull'oggetto remoto. Ma a differenza dell' applet che usa il codebase per poter eseguire codice sulla sua JVM, il client RMI usa il codebase dell'oggetto remoto per eseguire codice su una JVM remota**



Altri usi di codebase

- In aggiunta al caricamento delle classi stub e delle loro classi associate, la proprietà java.rmi.server.codebase può essere usata per specificare una locazione dalla quale delle classi, non solo stubs, possono essere caricate.
- Quando un client effettua una chiamata a un metodo di un oggetto remoto, si possono verificare tre casi, dipendenti dal tipo(i) dei parametri(o) del metodo(i)



Altri usi di codebase

1. **Nel primo caso, tutti i parametri del metodo(i) (e il valore di ritorno) sono tipi di dato primitivi, quindi l'oggetto remoto conosce come interpretare i parametri dei metodi, e non c'è necessità di controllare il CLASSPATH o il codebase**
2. **Nel secondo caso, almeno un parametro del metodo remoto o il valore di ritorno è un oggetto per il quale però l'oggetto remoto può cercare la definizione della classe localmente nel suo CLASSPATH**

Altri usi di codebase

3. **Nel terzo caso il metodo remoto riceve un istanza di un oggetto per il quale l'oggetto remoto non può cercare la definizione della classe localmente nel suo CLASSPATH. La classe dell'oggetto inviata dal client è un sottotipo del tipo di parametro dichiarato (implementazione di interfaccia o sottoclasse)**
   - **Così come il codebase delle applets, il codebase del client è usato per caricare classi per altre JVMs**
   - **Se la propietà codebase è settata sull'applicazione client, allora quel codebase è annotato all'istanza del sottotipo quando la stessa viene caricata dal client.**

Esempi d'uso

Il valore del codebase può riferirsi a:

- L' URL di una directory in cui le classi sono organizzate in package
- L'URL di un file JAR
- Una stringa contenente istanze multiple di file JAR e/o directory secondo i criteri precedenti

Nota: Quando il valore del codebase si riferisce a un URL di una directory, il valore deve terminare con un "/"

Esempi d'uso

- Se le classi da caricare si trovano su un server HTTP chiamato ad esempio "webserver", nella directory "export" (sotto la web root), il codebase deve essere settato come segue:

  - Djava.rmi.server.codebase=http://webserver/export/

- Come prima, ma in un file JAR chiamato ad esempio "mystuff.jar", nella directory "public" (sotto la web root), il codebase deve essere settato come segue:

  -Djava.rmi.server.codebase=http://webserver/public/mystuff.jar

- Se le classi da caricare si trovano in due file JAR "myStuff.jar" and "myOtherStuff.jar" e questi due file si trovano su due server separati (chiamati "webserver1" " webserver2 "), il codebase deve essere settato come segue:

  -Djava.rmi.server.codebase="http://webserver1/myStuff.jar http://webserver1/myOtherStuff.jar"

Client-side callbacks

- **In molte architetture, un server può avere la necessità di effettuare una chiamata ad un client**
- **Affinché questo sia possibile, un client RMI deve comportarsi anche da server RMI**
- ***Client Callback*: meccanismo che permette ad un client di essere *notificato dal server* al verificarsi dell'evento per cui si è registrato e messo in attesa.**

  Notifica del server = Invocazione di un metodo remoto del client
- **Può non essere pratico per un client estendere la classe `java.rmi.server.UnicastRemoteObject`**
- **Un oggetto remoto può rendersi pronto per uso remoto chiamando il metodo statico:**

```
UnicastRemoteObject.exportObject(remote_objec)
```

Client-side callbacks

Su lato server bisogna mantenere un riferimento al client su cui fare la callback

- 1. definire l'interfaccia remota del server, come sempre, in cui però ci sia anche il metodo che il client invoca per inviare al server il riferimento su cui fare la callback
- 2. implementare l'interfaccia remota del server
- 3. istanziare l'implementazione dell'interfaccia remota del server e pubblicarla sul registry

Client-side callbacks

Su lato client bisogna fornire il *metodo remoto* che il server deve invocare per fare la callback:

- **1. *definire* l'interfaccia remota del client, in cui viene dichiarato il metodo di callback invocato dal server**
- **2. *implementare* l'interfaccia remota del client, ovvero implementare il metodo di callback**
- **3. *istanziare* l'implementazione dell'interfaccia remota del client e passarla al server che la utilizzarà per invocare il metodo remoto di callback**

Client-side callbacks

- **Il client esporta ed implementa un' interfaccia remota**
- **Nell' esempio seguente, il client implementerà l' interfaccia Time Monitor, che è progettata per essere chiamata da un servizio che fornisce la data e il tempo corrente**
- **Il server non può chiamare il client finché non conosce la locazione del client stesso**
- **E' compito del client registrarsi con il server. Il client fa questo attraverso il metodo registerTimeMonitor del server, passando al server come parametro di questo metodo un riferimento a se stesso**

NOTA: Verrà usata una via alternativa per settare RMI registry

Client-side callbacks

```
import java.rmi.*;
import java.util.Date;
public interface TimeMonitor extends java.rmi.Remote
{  public void time(Date d) throws RemoteException;
}


import java.rmi.*;
public interface TimeServer extends java.rmi.Remote
{  public void registerTimeMonitor(TimeMonitor tm)
        throws RemoteException;
}
```

Client-side callbacks

```
import java.util.Date;
class TimeTicker extends Thread
{  private TimeMonitor tm;
   TimeTicker( TimeMonitor tm ){
     this.tm = tm;
    }
    public void run(){
     while(true)
       try{
          sleep( 2000 );
          tm.time(new Date());
        }catch ( Exception e ){System.out.println(e);}
     }
}
```

Client-side callbacks

```
import java.net.*;
import java.io.*;
import java.util.Date;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;
public class TimeServerImpl implements TimeServer
{  private static TimeServerImpl  tsi;
   public static void main (String[] args)
   {            try
        {       tsi = new TimeServerImpl();
                LocateRegistry.createRegistry(1099);
                System.out.println("Registry created");

                UnicastRemoteObject.exportObject(tsi);
                Naming.rebind("TimeServer", tsi);
                System.out.println( "Bindings Finished" );
                System.out.println( "Waiting for Client requests"
   );
        } [to be continued…]
```

Client-side callbacks

```
catch (Exception e)
        {       System.out.println(e);
        }
    }
   public void registerTimeMonitor( TimeMonitor tm )
   {            System.out.println( "Client requesting a
   connection" );

        TimeTicker tt;
        tt = new TimeTicker( tm );
        tt.start();
        System.out.println( "Timer Started" );
   }
}  // class TimeServerImpl
```

Client-side callbacks

```
import java.util.Date;
import java.net.URL;
import java.rmi.*;
import java.rmi.server.*;
public class TimeClient implements TimeMonitor
{  private TimeServer ts;
        public TimeClient()
        {       try
        {       System.out.println( "Exporting the Client" );
                UnicastRemoteObject.exportObject(this);
                ts = (TimeServer)Naming.lookup(
                        "rmi://localhost:1099/TimeServer");
                ts.registerTimeMonitor(this);
          }
          catch (Exception e)
          {     System.out.println(e);
          }
   }
   [to be continued…]
```

Client-side callbacks

```
public void time( Date d )
    {   System.out.println(d);
    }
} //class TimeClient

public class Main
{  public static void main (String[] args)
    {          new TimeClient();
    }
}
```

Sicurezza

- **Java 2 garantisce la sicurezza utilizzando delle classi (gestori della sicurezza) che controllano l' ammissibilità delle operazioni svolte da un programma**
- **La possibilità di scaricare codice è controllata e regolata da un *security manager* che opera in accordo ad una policy**
- **Può essere necessario specificare un file di policy**
- **In un classico file di policy usato in RMI si permette al codice scaricato, ad esempio da un certo codebase, di:**
  - **Connettersi o accettare connessioni di rete sulle porte il cui numero è maggiore di 1024 (es. rmiregistry 1099)**
  - **Connettersi alla porta 80 (the port for HTTP)**

- **Questo è il codice di un file di policy generale (client.policy):**

```
grant { permission java.net.SocketPermission "*:1024-65535",
"connect,accept";
        permission java.net.SocketPermission "*:80",    "connect";
        // permission java.security.AllPermission;// ATTENZIONE!!!
};
```

Sicurezza

- Il client e il server devono essere eseguiti specificando la proprietà *java.security.policy* per indicare il file di policy che contiene i permessi che si intendono accordare:

  Es. lato client:

```
java –Djava.security.policy=client.policy Client
```

- Per settare il gestore della sicurezza di default è necessario utilizzare all' interno della classe client o server:

```
System.setSecurityManager(new RMISecurityManager());
```

  Così facendo rendiamo visibile il nuovo file di policy: aggiorniamo lo stato dei permessi

- Il S.M. garantisce che le classi che vengono caricate non eseguano operazioni per le quali non siano abilitate
- se il S.M non è specificato non è permesso nessun caricamento di classi da parte sia del client (stub) che del server