Laboratorio di Reti
Lezione 11
Callbacks,
RMI Support

3/12/2020

Laura Ricci

IL MECCANISMO DELLE CALLBACK

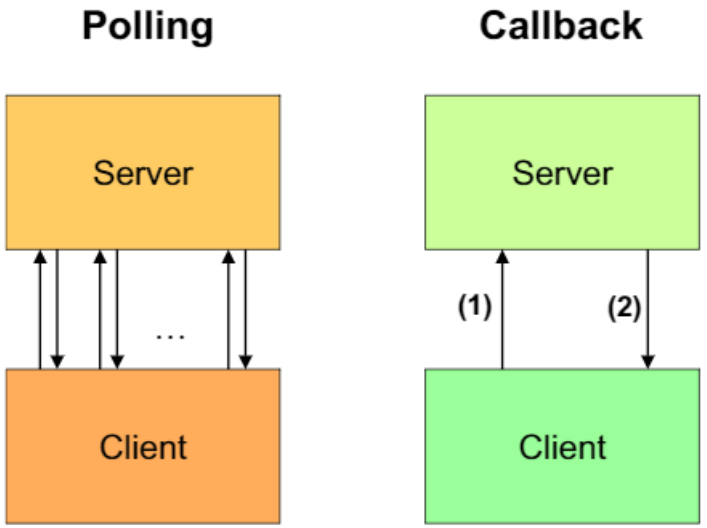
- consente di realizzare il pattern `Observer` in ambiente distribuito
- utile quando:
  - un client è interessato allo stato di un oggetto remoto
  - vuole ricevere una notifica asincrona quando lo stato viene modificato.

applicazioni:

- un utente partecipa ad una chat e vuol essere avvertito quando un nuovo utente entra nel gruppo.
- lo stato di un gioco multiplayer viene gestito da un server.
  - i giocatori notificano al server le modifiche allo stato del gioco.
  - ogni giocatore deve essere avvertito quando lo stato del gioco subisce modifiche.
- gestione distribuita di un'asta:
  - ogni volta che un utente fa una nuova offerta, tutti i partecipanti all'asta devono essere avvertiti

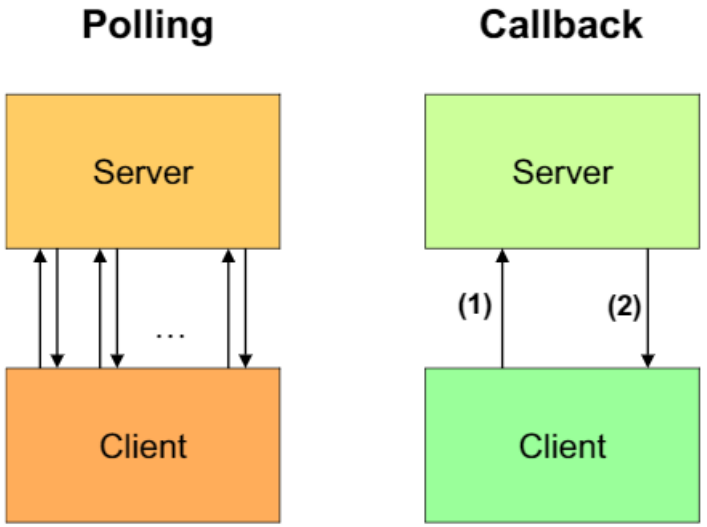
MECCANISMI DI NOTIFICA: POLLING

- il client verifica l'occorrenza dell'evento atteso, interrogando ripetutamente il server
  - invocazione ripetuta di un metodo remoto, tramite RMI
  - svantaggio
    - uso non efficiente delle risorse del sistema
    - alto costo



MECCANISMI DI NOTIFICA: CALLBACKS

- notifica asincrona del verificarsi dell'evento
  - il client registra il suo interesse per un evento,
  - il server notifica il verificarsi di tale evento
  - vantaggi:
    - il client non si blocca
    - viene avvertito quando l'evento avviene
- noto anche come paradigma publish-subscribe



CALLBACK VIA JAVA RMI

- si può utilizzare RMI sia per:
  - l'interazione client-server (registrazione dell'interesse per un evento)
  - quella server-client (notifica del verificarsi di un evento)
- il server definisce un'interfaccia remota **`ServerInterface`** con un metodo remoto utilizzato dal client per registrare il suo interesse per un certo evento
  - bootstrap della callback
- il client definisce una interfaccia remota **`ClientInterface`** con un metodo remoto utilizzato dal server per notificare un evento al client
- il client reperisce il riferimento all'oggetto remoto tramite il **`registry`**
- il client invia al server lo stub del suo ggetto remoto
  - Il server non utilizza il registry per individuare l'oggetto remoto del client,
  - lo riceve dal client come parametro del metodo di registrazione della callback

CALLBACK VIA RMI: IL SERVER

- definisce un oggetto remoto `ROS` che implementa `ServerInterface:`
  - contiene il metodo per la registrazione della callback
  - parametro del metodo di registrazione: riferimento allo stub del client
- quando riceve una invocazione del metodo remoto
  - riceve `ROC`, riferimento all'oggetto remoto del client
  - lo memorizza in una propria struttura dati
- al momento della notifica, utilizza `ROC` per invocare il metodo remoto sul client, per la notifica

CALLBACK VIA RMI: IL CLIENT

- definisce un oggetto remoto ROC che implementa `ClientInterface`
  - contiene un metodo che consente la notifica dell'evento atteso.
  - questo metodo verrà invocato dal server.
- ricerca l'oggetto remoto ROS del server che contiene il metodo per la registrazione mediante un servizio di `Registry`
- al momento della registrazione sul server, passa al server lo stub di ROC
- non registra l'oggetto remoto ROC in un registro
- la notifica asincrona viene implementata mediante due invocazioni remote sincrone

CALLBACK: UN ESEMPIO

Un server gestisce le quotazioni di borsa di un titolo azionario. Ogni volta che si verifica una variazione del valore del titolo, vengono avvertiti tutti i client che si sono registrati per quell'evento.

Il server definisce un oggetto remoto che fornisce metodi per

- consentire al client di registrare/cancellare una callback
- avvertire i client registrati quando si verifica una variazione sul titolo

Il client vuole essere informato quando si verifica una variazione

- registra una callback presso il server
- aspetta per un certo intervallo di tempo durante cui riceve le variazioni di quotazione
- alla fine cancella la registrazione della propria callback presso il server

L'INTERFACCIA DEL CLIENT

```
import java.rmi.*;
public interface NotifyEventInterface extends Remote {
    /* Metodo invocato dal  server per notificare un evento ad un
       client remoto. */
    public void notifyEvent(int value) throws
                                                RemoteException;}
```

`notifyEvent(int value)` è il metodo

- esportato dal client
- utilizzato dal server per la notifica di un nuova quotazione del titolo azionario

L'INTERFACCIA DEL CLIENT: IMPLEMENTAZIONE

```
import java.rmi.*;
import java.rmi.server.*;
public class NotifyEventImpl extends RemoteObject implements
                                        NotifyEventInterface {
  /* crea un nuovo callback client */
  public NotifyEventImpl( ) throws RemoteException
        { super( ); }

  /* metodo che può essere richiamato dal servente per notificare una
  nuova quotazione del titolo */
  public void notifyEvent(int value) throws RemoteException {
        String returnMessage = "Update event received: " + value;
        System.out.println(returnMessage); }
        }
```

LANCIO DEL CLIENT

```
import java.rmi.*; import java.rmi.registry.*; import java.rmi.server.*;
import java.util.*;
public class Client {
   public static void main(String args[ ]) {
      try
         {System.out.println("Cerco il Server");
          Registry registry = LocateRegistry.getRegistry(5000);
          String name = "Server";
          ServerInterface server =
                        (ServerInterface) registry.lookup(name);
          /* si registra per la callback */
          System.out.println("Registering for callback");
          NotifyEventInterface callbackObj =  new NotifyEventImpl();
          NotifyEventInterface stub = (NotifyEventInterface)
                     UnicastRemoteObject.exportObject(callbackObj, 0)
          server.registerForCallback(stub);
```

ATTIVAZIONE DEL CLIENT

```
// attende gli eventi generati dal server per
// un certo intervallo di tempo;
 Thread.sleep (20000);
/* cancella la registrazione per la callback */
System.out.println("Unregistering for callback");
server.unregisterForCallback(stub);
  } catch (Exception e)
        { System.err.println("Client exception:"+ e.getMessage(  ));}
 } } } }
```

L'INTERFACCIA DEL SERVER

```
import java.rmi.*;
public interface ServerInterface extends Remote
   {
    /* registrazione per la callback */
     public void registerForCallback
        (NotifyEventInterface ClientInterface) throws RemoteException;

    /* cancella registrazione per la callback */
     public void  unregisterForCallback  (NotifyEventInterface
                                ClientInterface) throws RemoteException;


   }
```

IL SERVER: IMPLEMENTAZIONE

```
import java.rmi.*; import java.rmi.server.*; import java.util.*;
public class ServerImpl extends RemoteObject implements ServerInterface
    { /* lista dei client registrati */
     private List <NotifyEventInterface> clients;
      /* crea un nuovo servente */
     public ServerImpl()throws RemoteException
          {super( );
           clients = new ArrayList<NotifyEventInterface>( ); };
     public synchronized void registerForCallback
       (NotifyEventInterface ClientInterface) throws RemoteException
          {if (!clients.contains(ClientInterface))
              { clients.add(ClientInterface);
               System.out.println("New client registered." );}};
```

IL SERVER: IMPLEMENTAZIONE

```
/* annulla registrazione per il callback */
  public synchronized void unregisterForCallback
        (NotifyEventInterface Client) throws RemoteException
              {if (clients.remove(Client))
                   {System.out.println("Client unregistered");}
              else { System.out.println("Unable to unregister
                     client."); }
              }

 /* notifica di una variazione di valore dell'azione
 /* quando viene richiamato, fa il callback a tutti i client
  registrati */
  public void update(int value) throws RemoteException
                  {doCallbacks(value);};
```

IL SERVER: IMPLEMENTAZIONE

```
private synchronized void doCallbacks(int value) throws
                                                    RemoteException
        { System.out.println("Starting callbacks.");
          Iterator i = clients.iterator( );
          //int numeroClienti = clients.size( );
          while (i.hasNext()) {
               NotifyEventInterface client =
                                  (NotifyEventInterface) i.next();
               client.notifyEvent(value);
           }
           System.out.println("Callbacks complete.");}
       }
```

ATTIVAZIONE DEL SERVER

```
import java.rmi.server.*; import java.rmi.registry.*;
public class Server   {
        public static void main(String[  ] args) {
           try{ /*registrazione presso il registry */
                ServerImpl server =  new ServerImpl( );
                ServerInterface stub=(ServerInterface)
                          UnicastRemoteObject.exportObject (server,39000);
                String name = "Server";
                LocateRegistry.createRegistry(5000);
                Registry registry=LocateRegistry.getRegistry(5000);
                registry.bind  (name, stub);
             while (true) { int val=(int) (Math.random( )*1000);
                          System.out.println("nuovo update"+val);
                          server.update(val);
                          Thread.sleep(1500);}
               } catch (Exception e) { System.out.println("Eccezione" +e);}}}
```

RMI CALLBACKS: RIASSUNTO

- Il client crea un oggetto remoto, oggetto callback ROC, che implementa un'interfaccia remota che deve essere nota al server
- Il server definisce un oggetto remoto ROS, che implementa una interfaccia remota che deve essere nota al client
- Il client reperisce ROS mediante il meccanismo di lookup di un registry
- ROS contiene un metodo che consente al client di registrare il proprio ROC presso il server
- quando il server ne ha bisogno, reperisce un riferimento ad ROC dalla struttura dati in cui lo ha memorizzato al momento della registrazione e contatta il client via RMI

RMI E CONCORRENZA

- Analizziamo le caratteristiche di un servizio remoto
  - non analizzeremo dettagliatamente l'implementazione
  - studiamo il comportamento tramite un insieme di esempi

- Vogliamo capire:
  - poiché esiste un solo oggetto remoto, i metodi di quell'oggetto possono essere invocati in modo concorrente da client diversi o da thread diversi dello stesso client?

  - in caso affermativo, viene creato un thread per ogni richiesta? Per ogni client?

  - cosa accade se non sincronizzo opportunamente gli accessi sull'oggetto remoto?

RMI E CONCORRENZA

Dalla documentazione ufficiale:

"A method dispatched by the RMI runtime to a remote object implementation (a server) may or may not execute in a separate thread. Calls originating from different clients Virtual Machines will execute in different threads. From the same client machine it is not guaranteed that each method will run in a separate thread"

- invocazioni di metodi remoti provenienti da client diversi (diverse JVM) sono eseguite da thread diversi
  - consente di non bloccare un client in attesa della terminazione dell'esecuzione di un metodo invocato da un altro client
  - ottimizza la performance del servizio remoto
- invocazioni concorrenti provenienti dallo stesso client (ad esempio se le chiamate si trovano in due thread diversi del client) possono essere eseguite dallo stesso thread o da thread diversi.

RMI E CONCORRENZA

- la politica di JAVA RMI di implementare automaticamente multithreading di chiamate diverse presenta il vantaggio di evitare all'utente di scrivere codice per i thread (server side)

- il server non è thread safe
  - richieste concorrenti di client diversi possono portare la risorsa ad uno stato inconsistente

- l'utente che sviluppa il server deve assicurare che l'accesso all'oggetto remoto sia correttamente sincronizzato (metodi `synchronized`, `locks`....)

RMI E CONCORRENZA: UNDER THE HOOD

Per verificare se i metodi dell'oggetto remoto sono invocati in modo concorrente,

- definiamo un oggetto remoto che esporta due metodi
- ogni metodo non fa altro che stampare per un certo numero di volte che è in esecuzione
- attiviamo due client: uno invoca il primo metodo, uno il secondo
  - si ottiene un interleaving delle stampe ?

```
public interface threads  extends java.rmi.Remote {
public void MethodOne()
                    throws java.rmi.RemoteException;
public void MethodTwo()
                    throws java.rmi.RemoteException;
}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
import java.rmi.*;
public class threadsimpl extends RemoteObject implements threads
{
  public threadsimpl() throws RemoteException
   {super();}
  public void MethodOne() throws RemoteException {
    long TimeOne = System.currentTimeMillis();
    for(int index=0;index<25;index++)
      { System.out.println("Method ONE executing");
        // Inserito un ritardo di circa mezzo secondo
        do{
           }while ((TimeOne+500)>System.currentTimeMillis());
       TimeOne = System.currentTimeMillis();
       }  }
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public void MethodTwo() throws RemoteException {
    long TimeTwo = System.currentTimeMillis();
    for(int index=0;index<25;index++)
        {
         System.out.println("Method TWO executing");
         // Inserito un ritardo di circa mezzo secondo
         do{
           }while ((TimeTwo+500)>System.currentTimeMillis());

         TimeTwo = System.currentTimeMillis();
           }
                                                    }
                                                    }
```

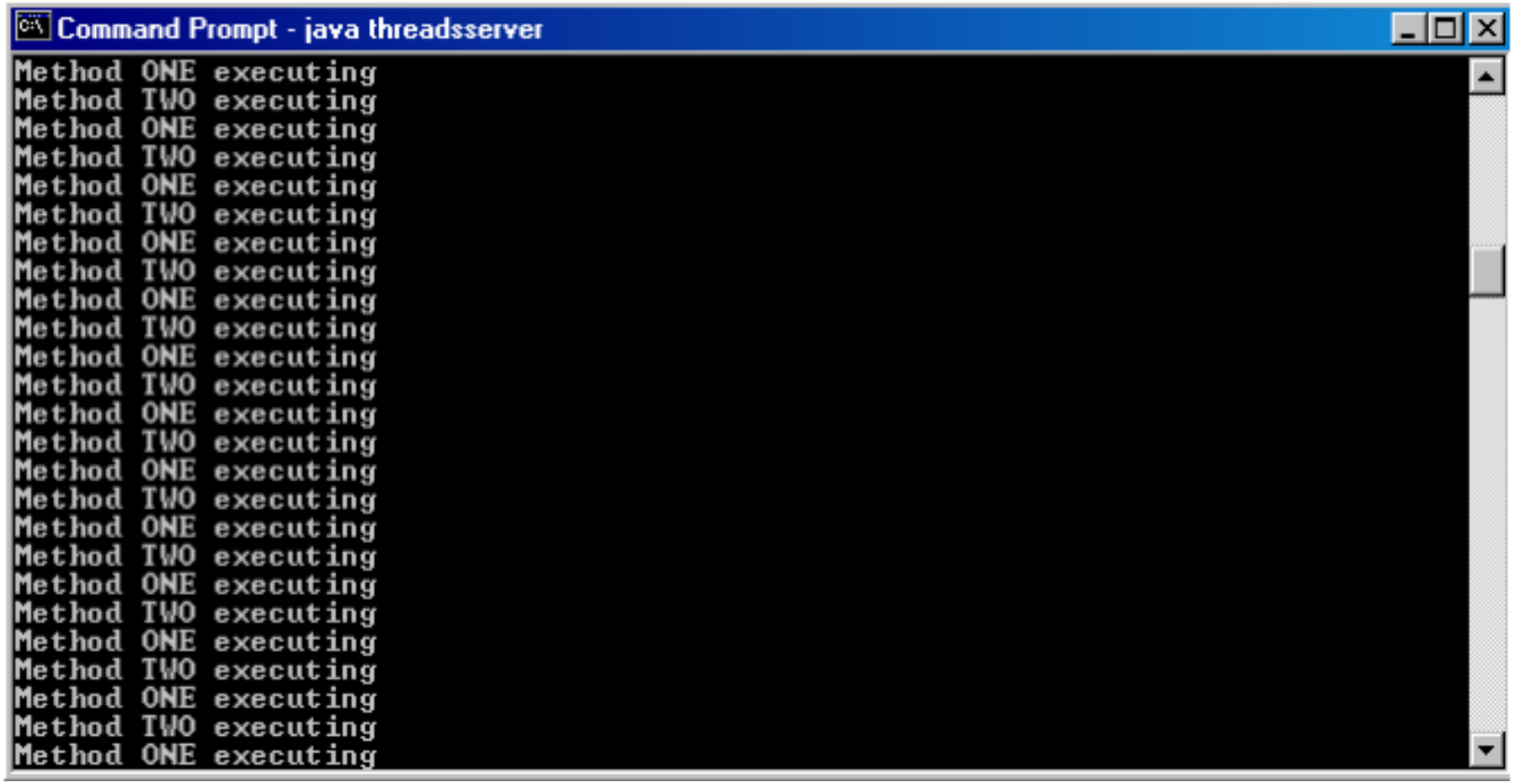
RMI E CONCORRENZA: UNDER THE HOOD

```
public class threadserver {
   public threadserver(int porta) {
     try {LocateRegistry.createRegistry(porta);
          Registry r=LocateRegistry.getRegistry(porta);
          System.out.println("Registro Reperito");
          threadsimpl c = new threadsimpl();
          threads stub =(threads)
                    UnicastRemoteObject.exportObject(c, 0);
        r.rebind("Threads", stub); }
     catch (Exception e) {
                  System.out.println("Server Error: " + e); }  }
    public static void main(String args[]) {
                                  new threadserver(args[0]);  }}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public class threadsclient {
    public static void main(String[] args) {
    try {
        Registry r= LocateRegistry.getRegistry(args[0]);
        threads c = (threads) r.lookup("Threads");
        if (args[1].equals("one"))
                c.MethodOne();
        else if (args[1].equals("two"))
                c.MethodTwo();
                else System.out.println("Error: correct usage -
                        treadsclient port {one|two}");
    }  catch (Exception e){  }}}
```

RMI E CONCORRENZA: UNDER THE HOOD



Una possibile traccia di esecuzione ottenuta attivando un client con argomento "one" ed uno con argomento "two"

RMI E CONCORRENZA: UNDER THE HOOD

- l'esecuzione del metodo `TWO` è iniziata prima che l'esecuzione del metodo `ONE` sia terminata
  - ciò implica che ad i due client sono stati associati due diversi threads, che invocano i metodi dell'oggetto remoto in modo concorrente

- se si vuole rendere "atomica" l'esecuzione di un metodo occorre utilizzare meccanismi opportuni di sincronizzazione (metodi `synchronized`)

- diversi modelli di esecuzione delle richieste provenienti dai client per l'esecuzione di metodi dell'oggetto remoto
  - prelevare le richieste da una coda e servirle sequenzialmente
  - un thread per ogni richiesta. Il thread invoca i metodi dell'oggetto remoto

RMI E CONCORRENZA: UNDER THE HOOD

- come vengono trattate le richieste provenienti da uno stesso client?
- se il client è sequenziale, ci può essere al massimo una richiesta pendente o in esecuzione per volta.
- se il client attiva più threads, questi possono eseguire in parallelo richieste di esecuzione di metordi sull'oggetto remoto
- le richieste vengono eseguite in sequenza o in maniera concorrente?
- il programma successivo indaga questo aspetto (si riferisce al servizio remoto definito nei lucidi precedenti).

RMI E CONCORRENZA: UNDER THE HOOD

```
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.lang.*;
public class threadsclientmod {
      public static void main(String[] args) {
          try {
               Registry r= LocateRegistry.getRegistry(2800);
               threads c = (threads) r.lookup("Threads");
               OneThread t1 = new OneThread(c);
               t1.start();
               TwoThread t2 = new TwoThread(c);
               t2.start();
            }     catch (Exception e){}}}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public class OneThread extends Thread{
   threads x;
   public OneThread(threads c)
     {this.x=c;}
  public void run()
     { try
           {x.MethodOne();}
      catch(Exception e){}
     }}
```

RMI E CONCORRENZA: UNDER THE HOOD

```
public class TwoThread extends Thread{
    threads x;
    public TwoThread(threads c)
      {this.x=c;}
    public void run()
      { try
        {x.MethodTwo();}
         catch(Exception e){}
        }}
```

- l'esecuzione di questo programma consente di capire se invocazioni concorrenti da parte dello stesso client vengono eseguite in modo concorrente o meno, sulla propria macchina

RMI: PASSAGGIO DI PARAMETRI

- Java Standard
  - `void f(int x)` il parametro x è passato per valore
  - `void g(Object k)` il parametro k è passato per riferimento
- Java RMI
  - `void h(Object k)`
    - se `Object` non è un oggetto remoto, il parametro viene passato per valore
    - l'oggetto viene coipiato da un host all'altro
    - usa il meccanismo di serializzazione dell'oggetto (marshalling)
    - se l'oggetto è un oggetto remoto viene passato un riferimento remoto

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;

public interface Point extends Remote {
     public void move(int x, int y) throws RemoteException;
     public String getCoord() throws RemoteException;
}

import java.rmi.*;
public interface Circle extends Remote {
     String SERVICE_NAME = "CircleService";
     Point  getCenter() throws RemoteException;
     double getRadius() throws RemoteException;
     void setCircle(Point c, double r) throws RemoteException;
}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;
import java.rmi.server.*;
public class PointImpl extends UnicastRemoteObject
                                implements Point {
    private int x, y;
    public PointImpl(int x, int y) throws RemoteException {
           this.x = x; this.y = y; }
    public void move(int x, int y) {
           this.x += x;
           this.y += y;
           System.out.println("Point has been moved to: " +
                                       getCoord());  }
    public String getCoord() {
           return "[" + x + "," + y + "]"; }
}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;
import java.rmi.server.*;
public class CircleImpl extends UnicastRemoteObject implements Circle
    { private Point center;
      private double radius;
      public CircleImpl(int x, int y, double r) throws
                                        RemoteException
              {setCircle(new PointImpl(x, y), r); }
      public void setCircle(Point c, double r) throws RemoteException
              {center = c; radius = r;
               System.out.println("Circle defined - Center: " +
                          center.getCoord() + " Radius: " + radius);}
      public Point getCenter() { return center; }
      public double getRadius() { return radius; }}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;
import java.rmi.registry.*;
public class CircleServer {
      public static void main(String args[]) throws Exception {
          Circle circle = new CircleImpl(10, 20, 30);
          LocateRegistry.createRegistry(9999);
          Registry r=LocateRegistry.getRegistry(9999);
          r.rebind(Circle.SERVICE_NAME, circle);
          System.out.println("Circle bound in registry");
     }}
```

Output:

```
Circle defined - Center: [10,20] Radius: 30.0
Circle bound in registry
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*
import java.rmi.registry.*;
public class CircleClient {
   public static void main(String[] args) throws Exception {
          Registry reg= LocateRegistry.getRegistry(9999);
          Circle circle = (Circle) reg.lookup(Circle.SERVICE_NAME);
          System.out.println(circle);
          double r = circle.getRadius() * 2;
          Point  p = circle.getCenter();
          System.out.println(p);
          p.move(30, 50);
          System.out.println("Circle - Center: " + p.getCoord() +
                             " Radius: " + r);
          circle.setCircle(p, r); } }
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

Sul Server:

```
Point has been moved to: [40,70]
Circle defined - Center: [40,70] Radius: 60.0
```

Sul Client:

- i valori restituiti per `Circle` e `Point`, sono riferimenti remoti

Proxy[Circle,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint: [192.168.1.25:54074](remote),objID:[-13ec7e8b:1546e1cec0e:-7fff, 4655578021548762323]]]]

Proxy[Point,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint: [192.168.1.25:54074](remote),objID:[-13ec7e8b:1546e1cec0e:-7ffe, 2122418885530499326]]]]

- i valori restituiti per Center e Radius sono i loro valori reali

Circle - Center: [40,70] Radius: 60.0

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

Supponiamo ora che il server remoto non esporti l'oggetto `Point`, l'oggetto viene definito come un oggetto locale

```
public class PointLoc extends java.io.Serializable {
       private int x, y;
       public PointLoc (int x, int y) { this.x = x; this.y = y; }
       public void move(int x, int y) {
         this.x += x;
         this.y += y;
          System.out.println("Point has been moved to: " + getCoord());
        }
       public String getCoord() {
           return "[" + x + "," + y + "]"}}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

- il client non può più interagire con l'oggetti remoto Point, perchè non esportato
- commentiamo le righe con cui il server interagisce con l'oggetto Point

```
import java.rmi.*
import java.rmi.registry.*;
public class CircleClient {
   public static void main(String[] args) throws Exception {
          Registry reg= LocateRegistry.getRegistry(9999);
          Circle circle = (Circle) reg.lookup(Circle.SERVICE_NAME);
          System.out.println(circle);
          double r = circle.getRadius() * 2;
          Point  p = circle.getCenter();
          System.out.println(p);
          // p.move(30, 50);
          // System.out.println("Circle - Center: " + p.getCoord() +
                              " Radius: " + r);
          // circle.setCircle(p, r);  } }
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

- dal programma modificato, si ottengono, nel client le seguente stampe

```
Proxy[Circle,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[192.168.1.25:54285](remote),objID:[-16781560:1546e5ae81b:
-7fff, 2510422791299955034]]]]
```

```
PointLoc@1be6f5c3
```

- nel client, viene stampata la precedente stringa, che rappresenta la serializzazione di `Point`
  - passaggio per valore: l'oggetto è stato copiato, non ne è stato passato il riferimento remoto
- se tolgo la parte commentata nel client, viene sollevata una eccezione, perchè il client non può accedere all'oggetto remoto

DYNAMIC CLASS LOADING

- `RMI` offre la capacità di reperire dinamicamente la definizione della classe di un oggetto, se non presente nei `local classpath`
- il meccanismo fornisce la possibilità di definire applicazioni espandibili (aspetto enfatizzato già al momento della presentazione del linguaggio nel 1995) che estendano dinamicamente il proprio comportamento.
- Dynamic class loading
  - possibilità offerta da RMI di scaricare dinamicamente un file `.class` e quindi di eseguirlo
- meccanismi necessari per l'implementazione:
  - individuazione del repository remoto delle classi
  - meccanismi di sicurezza: si scarica un fragmento di codice e poi lo si esegue
    - è possibile che quel codice possa recare danno, in modo intenzionale o meno

CLASS LOADING IN JAVA

- class loader di default
  - utilizzato per caricare la classe, il cui metodo main è eseguito utilizzando il comando `JAVA`,
  - ricerca nel `CLASSPATH` locale
  - ricerca successivamente tutte le classi utilizzate, ricercandole nel `CLASSPATH` locale.

- `RMIClassLoader` è utilizzato per il caricamento remoto delle classi, può essere
  - invocato implicitamente al momento della ricezione di oggetti remoti serializzati, ad esempio se si riceve un oggetto di cui non si possiede la classe per la deserializzazione
  - Invocato esplicitamente

DYNAMIC CLASS LOADING: SCENARI

- thin client scaricano dinamicamente il loro codice dal server
  - semplificazione delle applicazioni
  - aggiornamento automatico in caso di nuove versioni/patch
- caricamento dinamico degli stub, da parte del client, richiesto nelle prime versioni di RMI
- polimorfismo:
  - "il cuore" della programmazione ad oggetti.
  - caricamento dinamico delle classi essenziale per una vera programmazione ad oggetti remoti
    - lato server:
      - parametro di una invocazione remota può essere un oggetto di una sottoclasse di quella dichiarata nella interfaccia
    - lato client:
      - gli oggetti restituiti dal server al client possono essere di una sottoclasse del tipo dichiarato come parametro di ritorno

DYNAMIC CLASS LOADING: THIN CLIENT

- sviluppare un client minimale che carica dinamicamente il proprio codice da un'area condivisa dove l'ha memorizzato il server.

- vantaggi di questo approccio:
  - evitare la reinstallazione dei client, quando viene modificato
  - una unica copia del client presente in un'area condivisa
  - minor numero di inconsistenze:
    - se i client condividono la stessa classe nell'area pubblica, non useranno versioni diverse della medesima classe

- il client scarica l'ultima versione del codice dal server e lo esegue:
  - caricamento esplicito di classi dal server
  - metodo RMIClassloader.loadClass :
    - caricare dinamicamente ed in modo esplicito classi remote
    - necessaria URL che riferisce la directory da cui scaricare il codice (classe URL di JAVA)

DYNAMIC CLASS LOADING: THIN CLIENT

```
import java.rmi.server.RMIClassLoader; import java.net.*;import java.security.*;

public class LoadClient
{ public static void main(String[] args) {
  if (args.length == 0) {
       System.err.println("Usage: java LoadClient <remote URL>");
       System.exit(1);}
  System.setSecurityManager(new mySecurityManager());
  try {  URL url = new URL(args[0]);
         Class cl = RMIClassLoader.loadClass(url, "RunAway");
         System.out.println(cl);
         Runnable client = (Runnable) cl.newInstance();
         client.run();
         System.exit(0);
     } catch (Exception e) { System.out.println("Exception: " +
                             e.getMessage());
                             e.printStackTrace();}}}
```

DYNAMIC CLASS LOADING: THIN CLIENT

```
class mySecurityManager extends SecurityManager
   {
    public void checkConnect(String host, int port, Object context)
          {  }
    public void checkPermission(Permission perm)
          {   }
    }
```

DYNAMIC CLASS LOADING: THIN CLIENT

```
public class RunAway  implements Runnable {
      public void run() {
                System.out.println("I made it!");
                         }
        }
```

- per provare il programma in locale:
  - memorizzare la classe `RunAway` nella stessa directory in cui è memorizzata la classe `LoadClient`
  - invocare il programma con argomento `file://.` (la URL)
- in alternativa: se ha la possibilità di accedere ad un web server, indicare la URL del web server

CODE MOBILITY E SICUREZZA

- la mobilità del codice pone ovvi problemi di sicurezza
- da JAVA 2 in poi la politica di sicurezza, impone all'utente di definire esplicitamente i permessi di cui deve disporre un'applicazione scaricata dalla rete
- tali permessi definiscono una sandbox, ovvero uno "spazio virtuale" in cui l'applicazione deve esere eseguita.
  - una sandbox dovrebbe contenere il minimo numero di permessi che consentono l'esecuizione della applicazione
  - i permessi sono elencati in un file di policy
  - due usi diversi dei file di policy
    - amministratore di sistema: utilizza un file di policy per indicare cosa possono fare i programmi lanciati all'interno del sistema
    - sviluppatore dell'applicazione: distribuisce il file di policy che elenca i permessi di cui l'applicazione ha bisogno

IL SECURITY MANAGER

- la garanzia che vengano rispettati i permessi, fissati nei file di policy, durante l'esecuzione del codice, è fornita installando un'istanza della classe `SecurityManager`.

- quando un programma tenta di eseguire un'operazione che richiede un esplicito permesso (ad esempio una lettura su un file o una connessione via socket) viene interrogato il SecurityManager.

- i programmi che scaricano dinamamicamente del codice dalla rete, devono impostare un `SecurityManager`
  - RMI abilita il caricamento dinamico di codice solo in presenza del `SecurityManager`
  - in un'applicazione RMI, se non viene settato un security manager, gli stub e le classi possono essere caricate solamente dal CLASSPATH locale

IL SECURITY MANAGER

- per attivare un SecurityManager:
  - all'inizio del programma:

```
System.setSecurityManager(new SecurityManager())
```

  - oppure definire la proprietà di sistema da linea di comando

```
java -Djava.security.manager applicazione
```

viene creata ed installata un'istanza del SecurityManager prima dell'inizio dell'esecuzione del'applicazione

IL FILE DI POLICY

- la politica di sicurezza deve essere specificata nel file di policy
- semplice esempio di politica (per le prove iniziali...)

```
grant {// Allow everything
        permission java.security.AllPermission;
        };
```

- attribuisce al codice scaricato tutti i diritti possibili
- pericolosa se si scarica codice da un sito non sicuro.
- configurazione di default
  - ogni permesso richiesto dall'applicazione deve essere indicato esplicitamente.
  - esempio: le classi caricate dinamicamente non possono interagire col file-system locale, la rete, o l'interfaccia grafica.
- si possono fornire permessi diversi a classi caricate da codebase differenti.
  - esempio più diritti a quelle classi provenienti da "codebase fidati", meno a quelli provenienti da altri codebase

IL FILE DI POLICY

- una politica più complessa

```
grant {
        permission java.net.SocketPermission
            "*:1024-65635","connect";
       };
```

- questa politica permette ad una applicazione di instaurare qualsiasi connessione di rete su una porta con un numero di porta almeno pari a 1024
- RMI usa come porta di default

RIASSUMENDO.....

Passi necessari per l'attivazione dei controlli di sicurezza:

- scrivere un file di policy. Il file di policy, è un file di testo e contiene le specifiche della "politica di sicurezza" che si intende adottare. Il file deve essere memorizzato all'interno della directory del client

- definire la proprietà `java.security.policy`, che deve fare riferimento al file di policy che contiene la specifica della politica di sicurezza

- lanciare il programma con il seguente comando:

```
java  -Djava.security.policy=policy
```

  dove policy è il nome del file di policy

- oppure specificare da programma il valore della proprietà, mediante `System.setProperty()`.