Lezione 12
Introduzione a Remote Method Invocation

Vittorio Scarano
Corso di Programmazione Distribuita (2003-2004)
Laurea di I livello in Informatica
Università degli Studi di Salerno

Organizzazione della lezione

- Invocazione remota di metodi "fai-da-te"
  - Da un semplice esempio in locale..
  - .. al corrispondente esempio in remoto
  - La sequenza di chiamate
  - Un client più complesso
- Remote Method Invocation
- Le classi ed interfacce di RMI

Chiamate di metodi di oggetti remoti

- L'uso delle primitive di comunicazione TCP/IP
  - è efficiente
  - ma scomodo e poco produttivo quando si devono utilizzare i metodi messi a disposizione dagli oggetti
- Necessario scrivere esplicitamente il trattamento delle comunicazioni
  - definizione di un protocollo di comunicazione
  - parsing del comando
  - gestione degli errori
  - etc. etc.

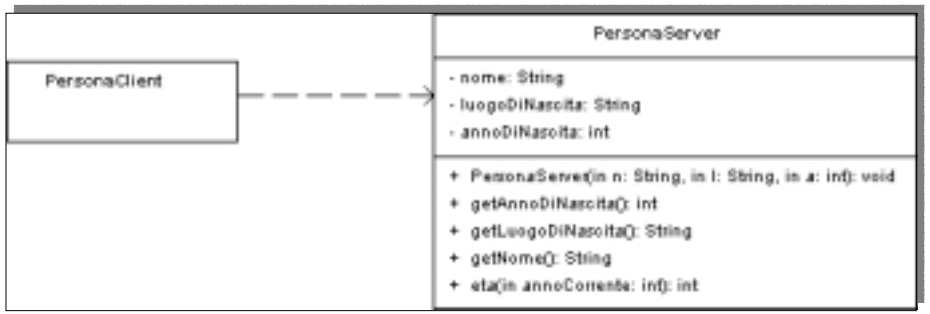
Un esempio di soluzione (senza RMI)

- Senza infrastrutture che permettono la chiamata di metodi remoti (Remote Method Invocation)
- Un esempio semplice:
  - un oggetto PersonServer che contiene nome, luogo e anno di nascita
  - un oggetto PersonClient che ne usa i metodi per accedere ai valori delle variabili di istanza
- Un programma Person
  - in versione locale (Client e Server sono sulla stessa JVM)
  - in versione "remota"

Organizzazione della lezione

- Invocazione remota di metodi “fai-da-te”
  - Da un semplice esempio in locale..
  - .. al corrispondente esempio in remoto
  - La sequenza di chiamate
  - Un client più complesso
- Remote Method Invocation
- Le classi ed interfacce di RMI

Persona locale: il diagramma delle classi



Persona locale: PersonaServer (1)

```
public class PersonaServer {
public PersonaServer (String n, String l, int a) {
     nome = n;
     luogoDiNascita = l;
     annoDiNascita = a;
}
public int getAnnoDiNascita() {
     return annoDiNascita;
}
public String getLuogoDiNascita() {
     return luogoDiNascita;
}
public String getNome() {
     return nome;
}

// continua…
```

- Costruttore
- Metodi di accesso
  - nome
  - luogo di nascita
  - anno di nascita

Persona locale: PersonaServer (2)

```
// ….

public int eta(int annoCorrente) {
     int appoggio = annoCorrente –
               annoDiNascita;
     return appoggio;
}

private String nome;
private String luogoDiNascita;
private int annoDiNascita;
}
```

- Restituisce l’età dato l’anno corrente
- Variabili di istanza

Persona locale: PersonaClient


```
public class PersonaClient {

public static void main(String[] args) {
  PersonaServer p = new
     PersonaServer ( "Pippo",
                     "Topolinia",
                     1925);

System.out.println ("Nome: "+ p.getNome() );
System.out.println ("Luogo di Nascita: "+
          p.getLuogoDiNascita() );
System.out.println ("Anno: "+
          p.getAnnoDiNascita() );
System.out.println ("Questa persona nel 2005
     avra' "+ p.eta(2005)+ " anni.");
}
}
```

- Creazione oggetto (locale)
  - riferimento
- Uso dei metodi dell'oggetto
  - nome
  - luogo di nascita
  - anno di nascita
  - età nel 2005

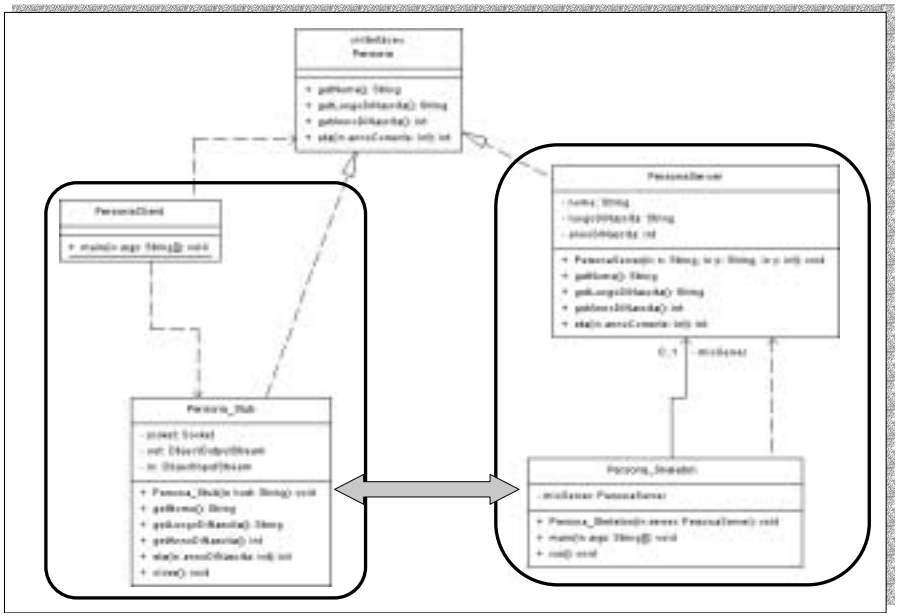
Organizzazione della lezione

- Invocazione remota di metodi "fai-da-te"
  - Da un semplice esempio in locale..
  - .. al corrispondente esempio in remoto
  - La sequenza di chiamate
  - Un client più complesso
- Remote Method Invocation
- Le classi ed interfacce di RMI

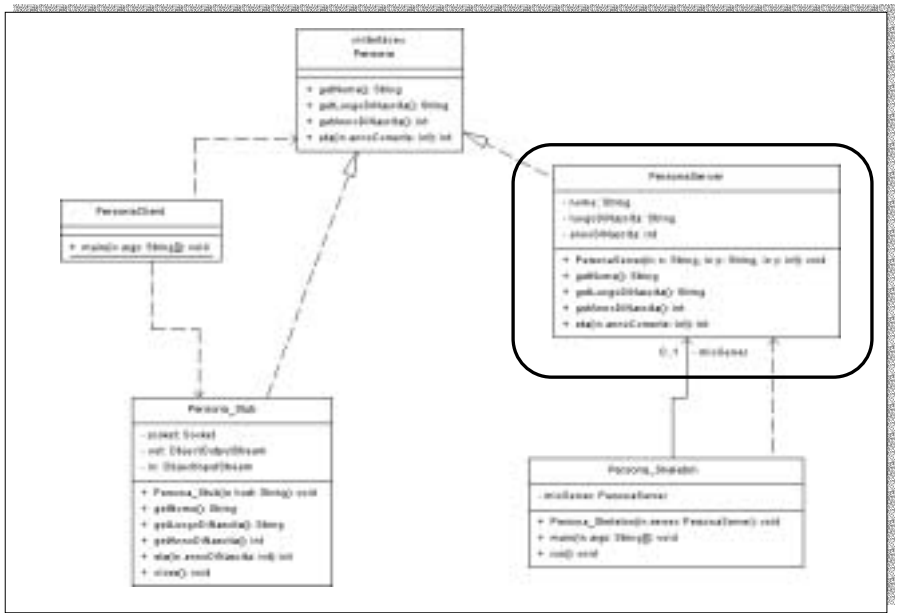
Persona remoto: struttura

- Si creano due classi che servono per la comunicazione
  - *stub* : lato client
    - serve a offrire i metodi del server verso il client
    - gestisce la comunicazione verso lo skeleton
  - *skeleton*: lato server
    - chiama i metodi del server come richiesto dallo stub
    - restituisce i valori di ritorno allo stub
- Come fa il client a conoscere cosa offre lo stub?
  - sia stub che il server implementano una interfaccia comune che indica i metodi offerti dal server

Persona remoto: il diagramma



Persona remoto: il diagramma



Persona remoto: PersonaServer (1)

```
public class PersonaServer
    implements Persona {
public PersonaServer (String n, String l, int a) {
    nome = n;
    luogoDiNascita = l;
    annoDiNascita = a;
}
public int getAnnoDiNascita() {
    return annoDiNascita;
}
public String getLuogoDiNascita() {
    return luogoDiNascita;
}
public String getNome() {
    return nome;
}

// continua…
```

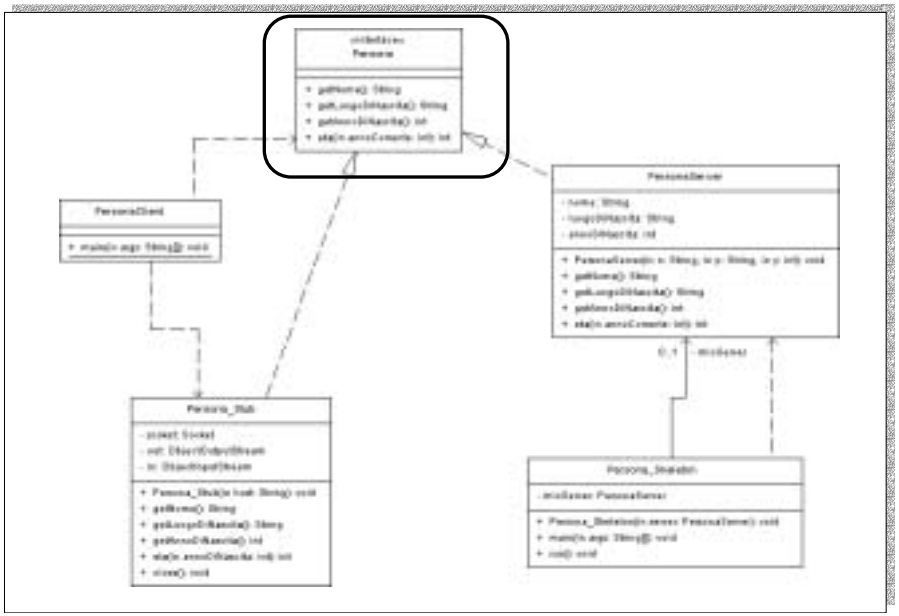
- Identico a PersonServer locale
  - tranne che nella implementazione della interfaccia
- Costruttore
- Metodi di accesso
  - nome
  - luogo di nascita
  - anno di nascita

Persona remoto: PersonaServer (2)

```
// ….

public int eta(int annoCorrente) {
    int appoggio = annoCorrente –
            annoDiNascita;
    return appoggio;
}

private String nome;
private String luogoDiNascita;
private int annoDiNascita;
}
```

- Restituisce l'età dato l'anno corrente
- Variabili di istanza

Persona remoto: il diagramma

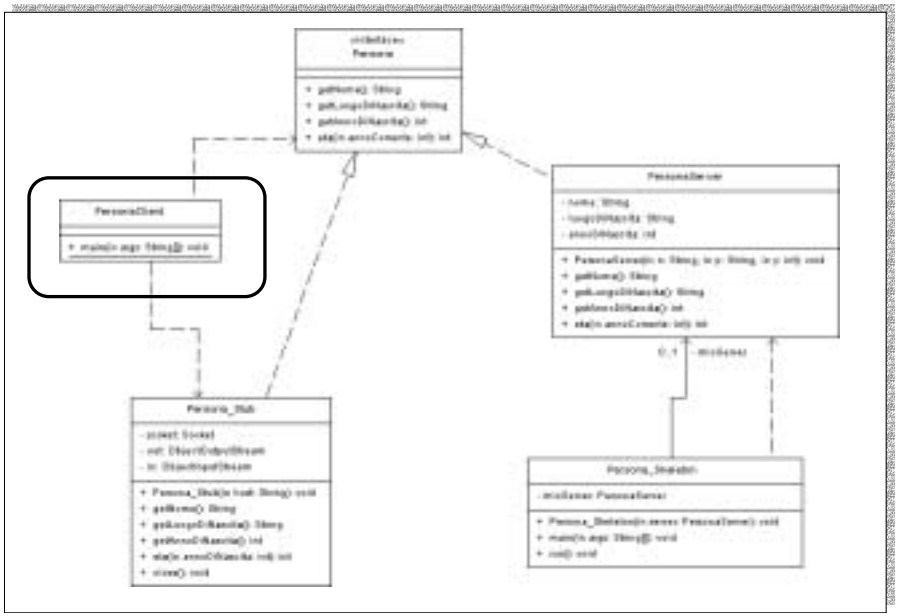


Persona remoto: interface Persona

```
public interface Persona
{
   public String getNome()
            throws Throwable;
   public String getLuogoDiNascita()
            throws Throwable;
   public int getAnnoDiNascita()
            throws Throwable;
   public int eta(int annoCorrente)
            throws Throwable;
}
```

- Interfaccia usata sia dallo Stub che dal server
- Metodi di accesso
  - nome
  - luogo di nascita
  - anno di nascita
- Calcolo della età

Persona remoto: il diagramma



Persona remoto: PersonaClient

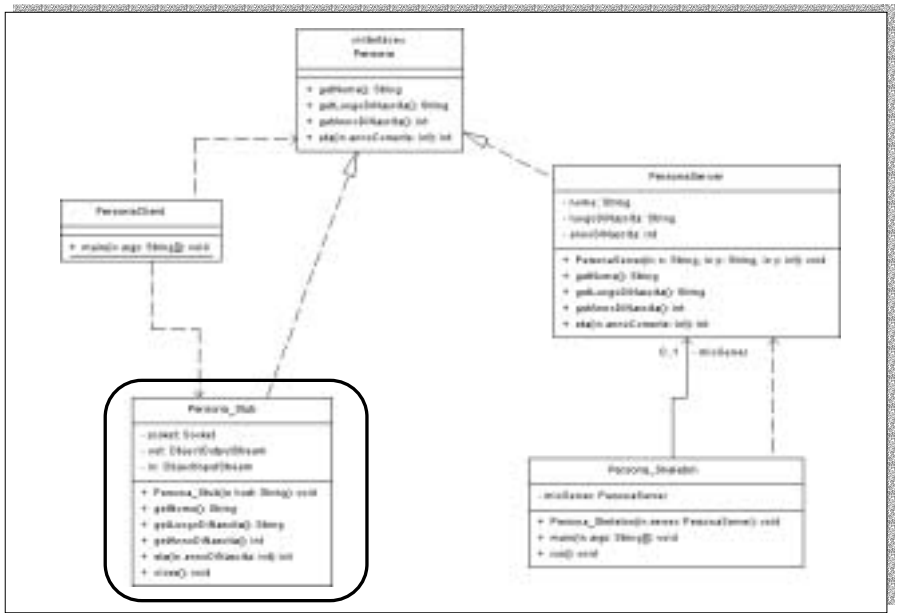
```
public class PersonaClient {
 public static void main(String args[]) {
   try {
     Persona p = new Persona_Stub(args[0]);
     System.out.println ("Nome:" +
          p.getNome());
     System.out.println ("Luogo di Nascita:" +
          p.getLuogoDiNascita());
     System.out.println ("Anno di Nascita:" +
          p.getAnnoDiNascita());
     System.out.println ("Nel 2005 avra' "+
          p.eta(2005)+" anni.");
     ((Persona_Stub) p).close();
   } catch (Throwable t) {t.printStackTrace();}
 }
}
```

- Host del server sulla linea di comando
- Creazione di uno stub
  - variabile di reference di tipo Person
- Chiamate dei metodi sullo stub
  - nome
  - luogo di nascita
  - anno di nascita
  - calcolo della età
- Chiusura

Protocollo tra Stub e Skeleton

- Lo Stub implementa tutti i metodi offerti dalla interface Person
- Il costruttore apre un socket verso lo skeleton
- Per ogni metodo
  - usa lo stream in output per scrivere il nome del metodo da fare eseguire
  - attende il valore restituito e lo restituisce al Client
- Lo skeleton
  - riconosce il metodo da eseguire
  - effettua le chiamate sul server
  - restituisce il valore di ritorno

Persona remoto: il diagramma



Persona remoto: Persona_Stub (1)

```
import java.io.*;
import java.net.*;
public class Persona_Stub
                    implements Persona  {
  public Persona_Stub(String host)
              throws Throwable {
       socket = new Socket (host, 9000);
       out = new ObjectOutputStream(
          socket.getOutputStream());
       in = new ObjectInputStream(
          socket.getInputStream());
    }
  public String getNome ()
              throws Throwable {
       out.writeObject("getNome");
       out.flush();
       return (String) in.readObject();
    }
// continua…
```

- Implementa Persona
- Costruttore:
  - apre un socket su host dello skeleton
  - crea input ed output ObjectStream
- Metodo getNome()
  - scrive il nome del metodo sul socket
  - restituisce la stringa restituita (letta dal socket)

Persona remoto: Persona_Stub (2)

```
   public String getLuogoDiNascita ()
                              throws Throwable {
       out.writeObject("getLuogoDiNascita");
       out.flush();
       return (String) in.readObject();
    }

   public int getAnnoDiNascita ()
              throws Throwable {
       out.writeObject("getAnnoDiNascita");
       out.flush();
       return in.readInt();
    }

// continua…
```

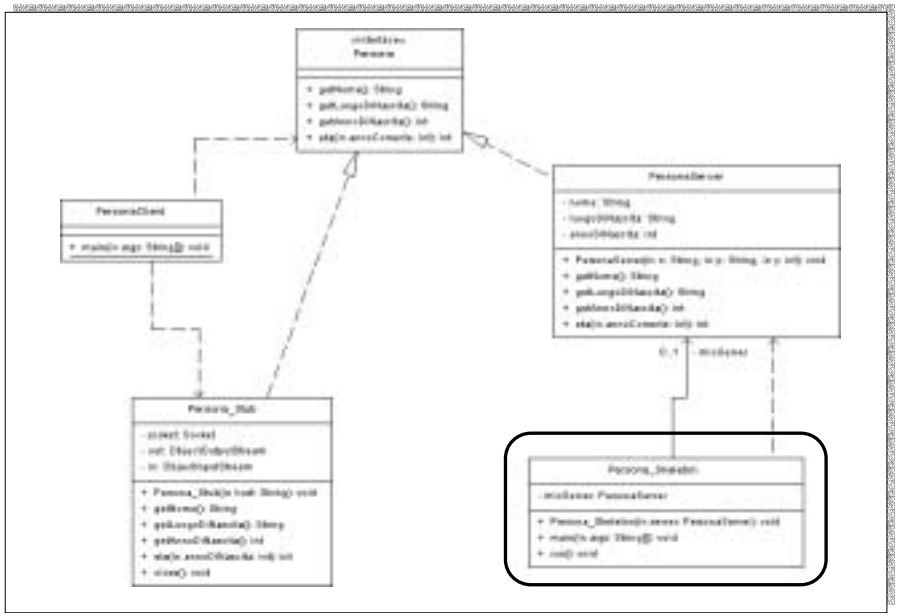
- Metodo getLuogo di Nascita()
  - scrive il nome del metodo sul socket
  - restituisce il valore letto
- Metodo getAnnoDiNascita()
  - scrive il nome del metodo sul socket
  - restituisce il valore letto

Persona remoto: Persona_Stub (3)

```
   public int eta (int annoDiNascita)
              throws Throwable {
       out.writeObject("eta");
       out.writeInt(annoDiNascita);
       out.flush();
       return in.readInt();
    }
   public void close () {
       try { socket.close();
       }
       catch (IOException e) {
       System.out.println ("Chiusura socket!");
       }
    }
    Socket socket;
    ObjectOutputStream out;
    ObjectInputStream in;
}
```

- Metodo eta()
  - scrive il nome del metodo sul socket
  - restituisce il valore letto
- Metodo close()
  - non nella interface
  - permette una chiusura corretta del socket
  - non strettamente necessario
- Variabili di istanza

Persona remoto: il diagramma



Persona remoto: Persona_Skeleton (1)

```
import java.io.*;
import java.net.*;
public class Persona_Skeleton
              extends Thread {
  public Persona_Skeleton(PersonaServer
                  server)  {
      mioServer = server;
   }
   public static void main (String args[]) {
      PersonaServer person = new
              PersonaServer("Pippo",
              "Topolinia",1925);
      Persona_Skeleton skel = new
          Persona_Skeleton(person);
       skel.start();
   }
// continua…
```

- Estende Thread
- Costruttore
  - parametro: un PersonServer
- Main
  - istanzia un oggetto PersonServer
  - passa al suo costruttore il suo riferimento
  - fa partire il thread

Vecchio (2)

```
public void run() {
  Socket socket = null;
  String method;
  try {
     ServerSocket serverSocket =
                  new ServerSocket(9000);
     socket = serverSocket.accept();
     ObjectInputStream inStream =
              new ObjectInputStream (
                       socket.getInputStream());
      ObjectOutputStream outStream =
              new ObjectOutputStream (
                    socket.getOutputStream());
       while (true) {
          method = (String) inStream.readObject();
          if (method.equals("getName")) {
             outStream.writeObject(
                                myServer.getName());
             outStream.flush();
          } else // continua
```

- Apertura ServerSocket
- Alla accettazione
  - preleva stream di input e di output
- Ciclo:
  - leggi dal socket il metodo da invocare
  - chiamata al metodo getName()
  - scrive in output la risposta del PersonServer

Persona remoto: Persona_Skeleton (2)

```
public void run() {
   Socket socket = null;
   String metodo;
   int p;
   System.out.println ("Inizio...");
    try {
      ServerSocket serverSocket = new
              ServerSocket(9000);
     socket = serverSocket.accept();
     ObjectInputStream inStream = new
          ObjectInputStream (
          socket.getInputStream());
     ObjectOutputStream outStream = new
     ObjectOutputStream (
          socket.getOutputStream());
     while (true) {
       metodo = (String) inStream.readObject();
       System.out.println ("Comando:" + metodo);

     // continua….
```

- Apertura ServerSocket
- Alla accettazione
  - preleva stream di input e di output
- Ciclo:
  - leggi dal socket il metodo da invocare
  - chiamata al metodo getName()
  - scrive in output la risposta del PersonServer

Persona remoto: Persona_Skeleton (3)

```
// … dentro al while (true)
  if (metodo.equals("getNome")) {
     outStream.writeObject(mioServer.getNome());
     outStream.flush();
   } else if (metodo.equals("getLuogoDiNascita")) {
         outStream.writeObject(
           mioServer.getLuogoDiNascita());
         outStream.flush();
         } else if (metodo.equals(
                "getAnnoDiNascita")){
                 outStream.writeInt(
           mioServer.getAnnoDiNascita());
                  outStream.flush();
                    } else if (metodo.equals("eta")){
                p = inStream.readInt();
                outStream.writeInt(
                mioServer.eta(p));
                        outStream.flush();
                        } else break;
   } // fine while ….. continua….
```

- getNome()
  - chiama il metodo locale
  - invia il risultato del PersonServer
- getLuogoDiNascita()
  - scrive in output la risposta del PersonaServer
- getAnnoDiNascita()
  - scrive in output la risposta del PersonaServer
- Chiamata al metodo eta()
  - lettura del parametro
  - chiamata metodo locale
  - invia il risultato

Persona remoto: Persona_Skeleton (4)

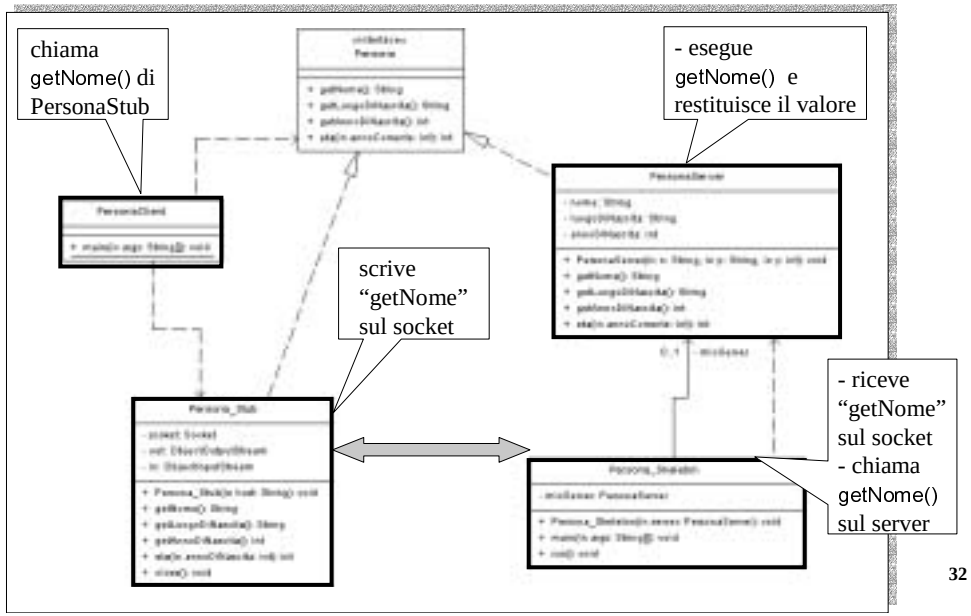
```
// … chiusura del blocco try…
} catch (EOFException e)  {
        System.out.println ("Terminata… ");
        }
        catch (Throwable t)  {
        t.printStackTrace();
        System.out.println ("Sk:"+t.getMessage());
        }
        finally {
        try { socket.close();
        } catch (IOException e) {
                e.printStackTrace();
                System.exit(0);
            }
        }
 } // fine metodo run()

PersonaServer mioServer;
}
```

- Catch
- Comunque alla fine
  - chiude il socket
- Variabile di istanza

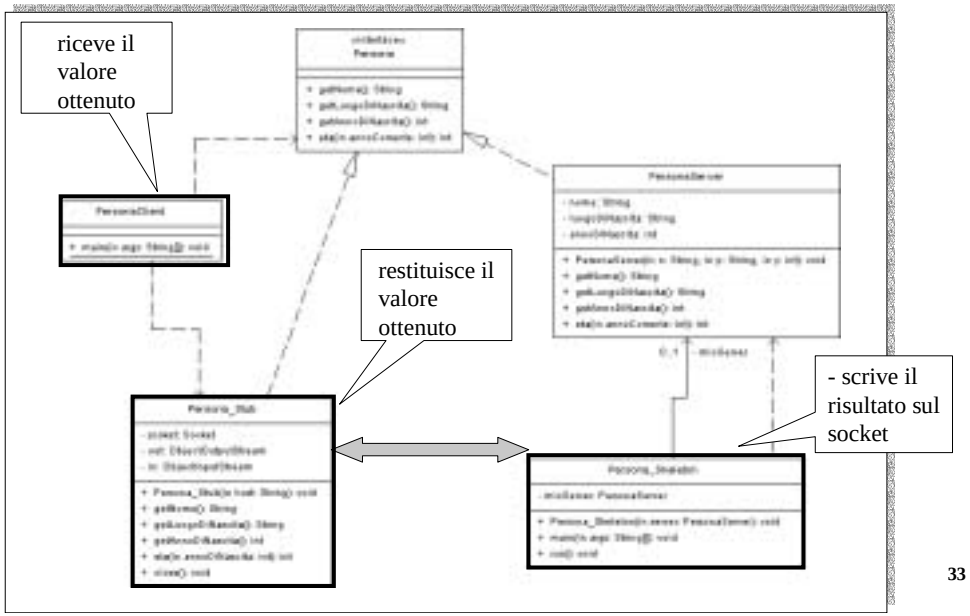
Organizzazione della lezione

- Invocazione remota di metodi “fai-da-te”
  - Da un semplice esempio in locale..
  - .. al corrispondente esempio in remoto
  - La sequenza di chiamate
  - Un client più complesso
- Remote Method Invocation
- Le classi ed interfacce di RMI

Person remoto: la sequenza delle chiamate



Person remoto: la sequenza delle chiamate



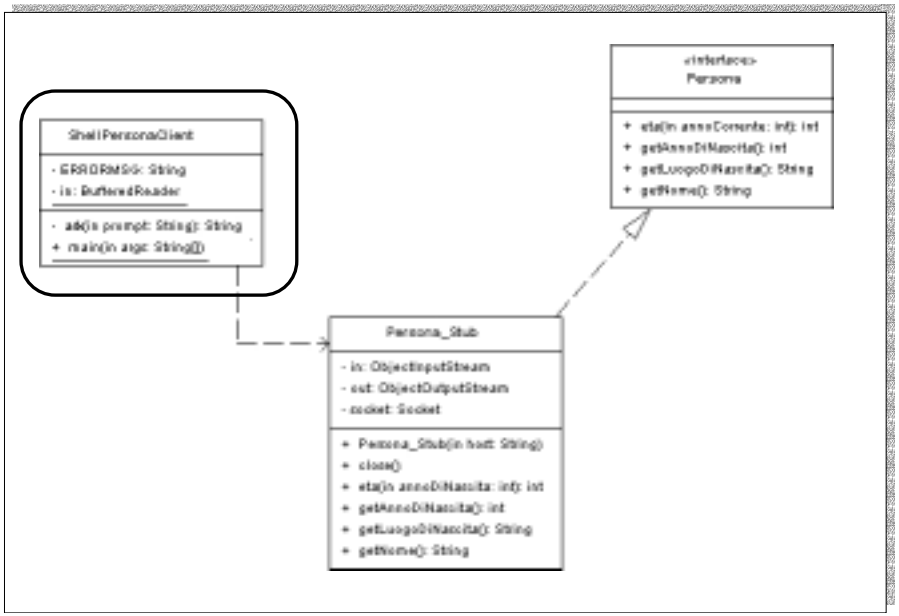
Organizzazione della lezione

- Invocazione remota di metodi “fai-da-te”
  - Da un semplice esempio in locale..
  - .. al corrispondente esempio in remoto
  - La sequenza di chiamate
  - Un client più complesso
- Remote Method Invocation
- Le classi ed interfacce di RMI

Un client più complesso

- ShellPersonaClient permette
  - tramite una semplice shell di comandi
  - la invocazione di metodi selezionati dall’utente
- Il resto delle classi rimane identico
- Struttura di ShellPersonaClient
  - legge da standard input (con un Buffered Stream)
  - fa un semplice parsing dei comandi
    - comandi per chiamate remote (getNome, getLuogoDiNascita, etc.)
    - comando per la chiusura (con il metodo close offerto dallo stub)

Un client più complesso: ShellPersonaClient



Persona remoto: ShellPersonaClient (1)


```
import java.io.*;
public class ShellPersonaClient {
  static final String ERRORMSG = "Cosa?";
  static BufferedReader in = null;
  private static String ask(String prompt)
                  throws IOException {
     System.out.print(prompt+" ");
     return (in.readLine());
   }
  public static void main(String args[]) {
     Persona persona = null;
     String host = args[0];
     String cmd;
     in = new BufferedReader(
      new InputStreamReader(System.in));
     try {
        persona = new Persona_Stub(host);
// continua…
```

- Variabili istanza
- Metodo privato
  - serve a chiedere input di una stringa con un prompt a schermo
- Host del server sulla linea di comando
- Input da tastiera
- Creazione di uno stub
  - variabile di reference di tipo Persona

Persona remoto: ShellPersonaClient (2)


```
    while (!(cmd = ask (“>>")).equals("close")) {
      if (cmd.equals ("getNome"))
          System.out.println ("Nome:" +
              persona.getNome());
       else if (cmd.equals ("getLuogoDiNascita"))
       System.out.println (“Nato in:" +
          persona.getLuogoDiNascita());
       else if (cmd.equals (
              "getAnnoDiNascita"))
             System.out.println (“Nato nel:" +
          persona.getAnnoDiNascita());
           else if (cmd.startsWith("eta")) {
         int p = Integer.parseInt(
                 ask("Anno:"));
         System.out.println ("Nel "+p+
          "avrà "+persona.eta(p)+"anni");
         } else
     System.out.println (ERRORMSG);
} // end while … continua …
```

- Input comando
  - finquando non è close
- Se il comando è getNome
  - eseguilo sullo stub
- Stessa cosa per
  - getLuogoDiNascita()
  - getAnnoDiNascita()
  - eta()
    - in questo caso chiede anche il parametro
- Messaggio di errore

Persona remoto: ShellPersonaClient (3)


```
// continua…. chiusura del try..catch
} catch (Throwable t) {
     t.printStackTrace();
}
finally {
 Persona_Stub p =(Persona_Stub) persona;
 p.close();
}
}// end main


}
```

- Catch
- Comunque alla fine
  - casting necessario per poter chiamare close()

In conclusione…

- Questo è un semplice esempio di come viene reso possibile la invocazione remota di metodi tramite RMI
  - in effetti, tutto il lavoro di scrivere stub e skeleton non viene richiesto al programmatore
  - la cui difficoltà è quindi nello scrivere ed implementare certe interfacce ed usare certi strumenti
    - “*Invocazione remota fai-da-te? No RMI?*” Ahi Ahi Ahi ☺
  - attenzione: non abbiamo trattato il problema del come reperire il riferimento remoto!
- E adesso…. Remote Method Invocation!

Organizzazione della lezione

- Invocazione remota di metodi “fai-da-te”
  - Da un semplice esempio in locale..
  - .. al corrispondente esempio in remoto
  - La sequenza di chiamate
  - Un client più complesso
- Remote Method Invocation
- Le classi ed interfacce di RMI

Remote Method Invocation (RMI)

- RMI è un modello ad oggetti distribuito offerto da Java
- Il modello mantiene le caratteristiche di Java...
  - …e si integra all'interno della semantica di Java.
- Il principio: "*Write once, Run anywhere"*
- Estensione "*ad oggetti"* di Remote Procedure Call
  - fornisce la visione astratta di una chiamata di procedura remota verso altri processi

Gli scopi di RMI - 1

- Supportare in maniera trasparente invocazioni remote di metodi su oggetti su differenti JVM
- Integrare in maniera naturale il modello distribuito all'interno di Java mantenendo le caratteristiche del linguaggio
- Rendere evidenti le differenze tra il modello distribuito e quello locale
- Rendere facile (compatibilmente con gli altri scopi) la realizzazione di applicativi distribuiti in Java

Gli scopi di RMI - 2

- Preservare il type-safety di Java
- Supportare la semantica dei riferimenti (persistenza, lazy activation etc.)
- Mantenere la sicurezza offerta da Java con i security manager e i class loader
- Supportare le callback da server verso applet

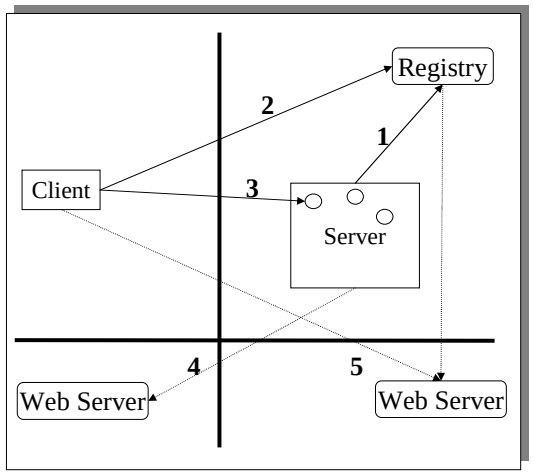
Applicazioni ad oggetti distribuite

- Applicazioni composte da *server* e *client*
  - Il server:
    - crea un certo numero di oggetti
    - li rende accessibili da remoto
    - attende le invocazioni dei client sugli oggetti
  - Il client:
    - preleva la reference ad uno o più oggetti remoti
    - invoca i loro metodi
- Ruolo tra client e server spesso "confuso"
  - "*servent*" in una architettura *peer-to-peer*

Ed RMI?

- RMI fornisce il meccanismo attraverso il quale server e client comunicano
- Una applicazione distribuita deve, quindi, poter:
  - Localizzare oggetti remoti
    - attraverso un *registro*
  - Comunicare con oggetti remoti
    - nascondere per quanto possibile i dettagli della invocazione remota
  - Poter caricare dinamicamente classi dalla rete
    - tra i parametri passati possono esserci anche oggetti

Uno schema di funzionamento per RMI



1. Registrazione oggetto con un nome sul registry da parte del server
2. Il client ricerca il nome sul registry ed ottiene un riferimento remoto
3. Il client invoca il metodo remoto sull'oggetto
- Caricamento dinamico delle classi da Web server
  4. per i parametri del metodo passati al server
  5. per oggetti come valori restituiti dal metodo

Somiglianze tra il modello distribuito e locale

- Un riferimento ad un oggetto remoto può essere passato come argomento (o restituito) da una invocazione di un metodo (locale o remoto)
- Un oggetto remoto può essere soggetto a *casting*
- L'operatore *istanceof* può essere usato per testare le interfacce supportate da un oggetto remoto

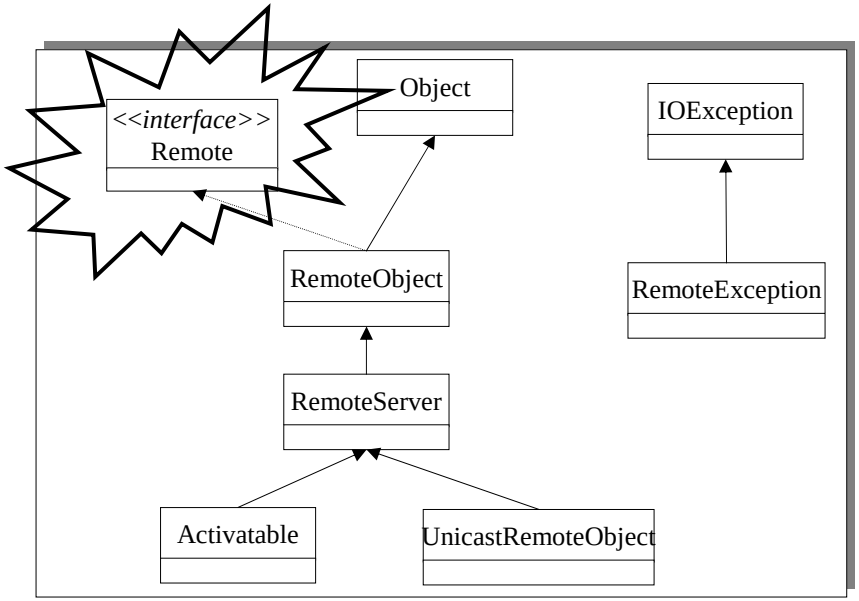
Differenze tra il modello distribuito e locale

- I client interagiscono con le interfacce remote non con la loro implementazione
- Argomenti non remoti e risultati sono passati per copia non per riferimento
- Un oggetto remoto è passato per sempre per riferimento
- Alcuni metodi di *Object* hanno semantica diversa per gli oggetti remoti (*equals, hashCode,toString, clone)*
- Esistono alcune eccezioni remote addizionali

Organizzazione della lezione

- Invocazione remota di metodi “fai-da-te”
  - Da un semplice esempio in locale..
  - .. al corrispondente esempio in remoto
  - La sequenza di chiamate
  - Un client più complesso
- Remote Method Invocation
- Le classi ed interfacce di RMI

Le classi ed interfacce di RMI

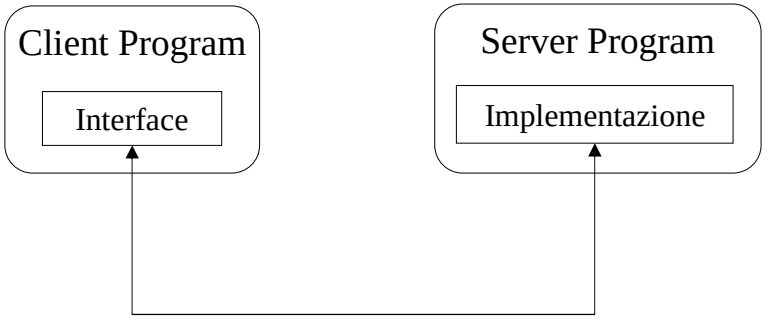


Le interfacce remote

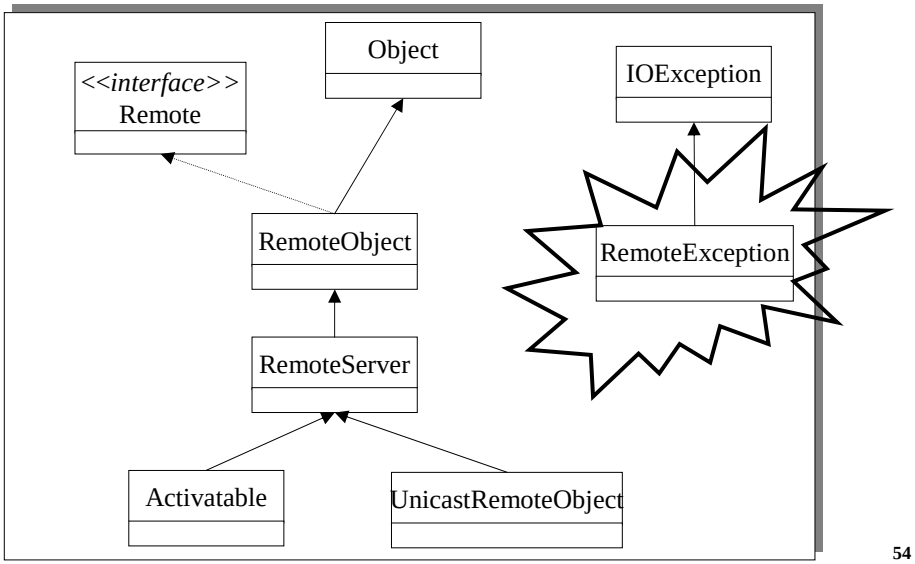
- Una interfaccia remota dichiara metodi che possono essere invocati da una JVM remota
- Requisiti di una interfaccia remota
  - deve derivare (direttamente o indirettamente) da java.rmi.Remote
    - una interfaccia *marker* (definita vuota)
  - tutti i metodi dichiarati devono essere *remoti*
    - devono indicare che lanciano java.rmi.RemoteException o una delle superclassi java.io.IOException o java.lang.Exception)
    - parametri remoti devono essere dichiarati tramite la corrispondente interface

Uso delle *interface* in RMI

- Interface: comportamento
- Classe: implementazione



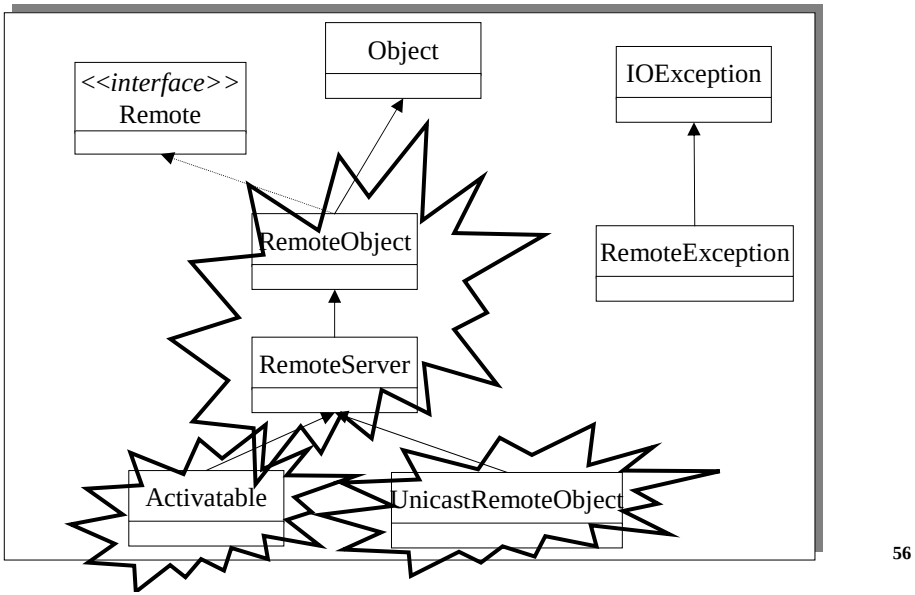
Le classi ed interfacce di RMI



Le eccezioni remote

- La classe java.rmi.RemoteException rappresenta le eccezioni che possono essere generate a runtime
- Alcuni possibili motivi:
  - malfunzionamento della comunicazione
    - server non raggiungibile
    - server rifiuta connessioni
    - connessione chiusa dal server
  - errore durante il trasferimento dei parametri o dei valori di ritorno
  - errori del protocollo RMI

Le classi ed interfacce di RMI



Gli oggetti remoti (server)

- Le classi sono:
  - java.rmi.server.RemoteObject
    - superclasse (astratta): implementa hashCode(), equals() e toString()
  - java.rmi.server.RemoteServer
    - superclasse (astratta) per server (gestione log, etc.)
  - java.rmi.server.UnicastRemoteObject
    - superclasse (astratta) per un oggetto remoto il cui riferimento è valido solo se è attivo il server
  - java.rmi.activation.Activatable
    - superclasse (astratta) per oggetti attivabili: cominciano la loro esecuzione all'atto della chiamata e possono "chiudersi" se necessario (*run on-demand*)