Programmazione Java
Corso Pratico
5 - Applicazioni di rete

Maurizio Franco

Java
nasce per poter realizzare
applicazioni distribuite sulla rete

In questa introduzione alle applicazioni distribuite(in rete), faremo una panoramica su classi che ci vengono fornite per rendere possibili di la realizzazione di tali applicazioni.

Panoramica sul packge java.net

- InetAddress
- URLConnection
- URL
- ServerSocket
- Socket

Applicazioni di rete

- InetAddress: virtualizzazione dell'indirizzo IP(di un host)nella rete;
- URLConnection: gestione di una connessione tra applicazione ed un url.
- URL: puntatore nel www di una risorsa
- ServerSocket: gestione connessioni socket standard lato server;
- Socket: gestione connessioni socket standard lato client;

Classe InetAddress

Astrazione dell'indirizzo IP.

Gli indirizzi numerici sono complessi da ricordare per gli esseri umani, pertanto si preferisce usare delle stringhe di caratteri:

www.google.com

InetAddress: alcuni metodi di utilità

static InetAddress getLocalHost() throws UnknownHostException: restituisce l'indirizzo IP della macchina locale;

static InetAddress getByName(String host) throws UnknownHostException: restituisce l'indirizzo IP associato a un dato host name;

String getHostName(): restituisce il nome della macchina legato all'ip fornito

```
public class TestInetAddress {

    public static void main(String[] args) throws
    UnknownHostException {
     InetAddress address = InetAddress.getLocalHost();
     System.out.println(address);
     address = InetAddress.getByName("starwave.com");
     System.out.println(address);
     InetAddress sw[] = InetAddress.getAllByName("www.nba.com");
     for (int i=0; i<sw.length; i++)
     System.out.println(sw[i]);
    }

}
```

Classe URL

E' la rappresentazione astratta di una risorsa nel world wide web.
Come ad esempio un puntamento ad un file o una directory.

Esempio:

URL xxxUrl = new URL(“http://java.sun.com/index.html”);

Classe URLConnection

Rappresenta la connessione fra la propria applicazione ed un URL.
Un'istanza di questa classe può essere usata per leggere e/o scrivere una risorsa rappresentata da un'istanza della classe URL.

Esempio:

```
URL page = new URL("http://www.google.com");
URLConnection xxxConn = page.openConnection();
xxxConn.connect();
```

Classe ServerSocket

La classe ServerSocket si occupa di gestire i socket lato server;
crea una socket per ogni connessione richiesta da un client.

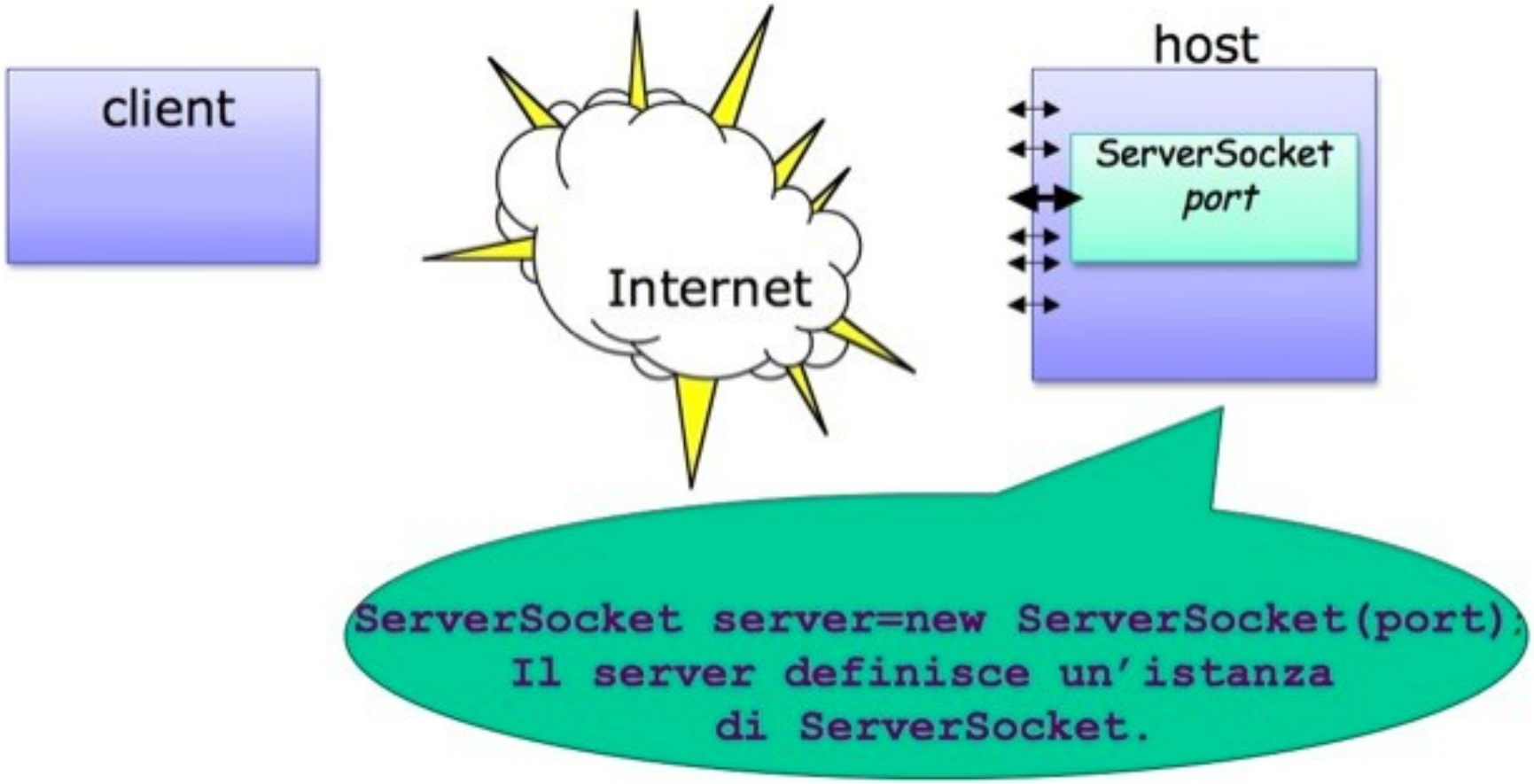
La classe ServerSocket

ha due costruttori:
ServerSocket(int port) throws IOException;
ServerSocket(int port, int backlog) throws IOException.

Il parametro port indica il “port number” sull’host locale (può assumenre valori 1-65535, anche se i valori da 1 a 1023 sono riservati).

Il parametro backlog indica il numero massimo di richieste di connessione che possono essere accordate dal sistema operativo. Utilizzando il primo costruttore tale parametro assume il valore di default di 50.

Applicazioni di rete



ServerSocket

Il metodo più importante è il seguente:

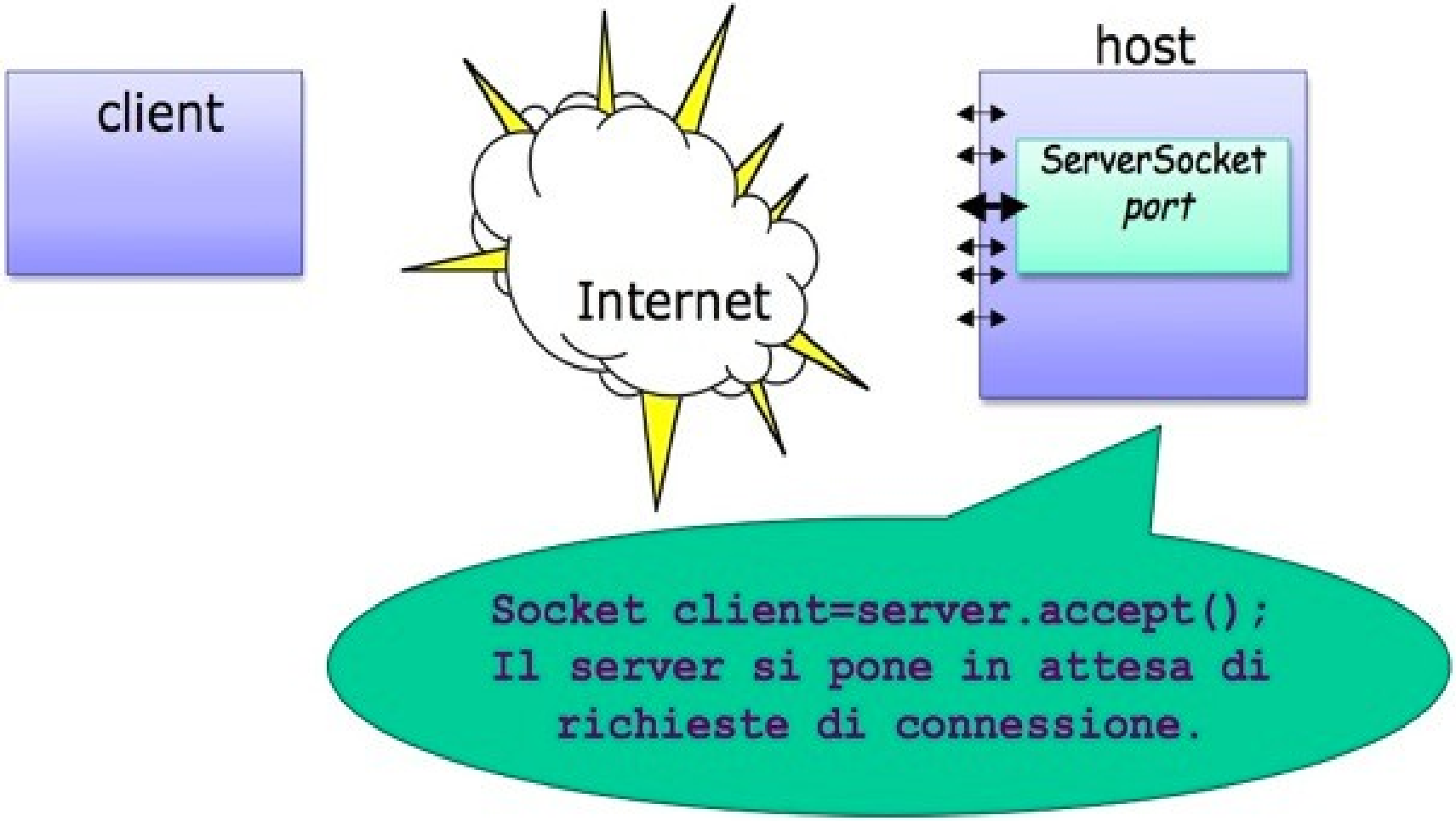
public Socket accept() throws IOException

Comporta l'attesa (sul port number del server) di una richiesta di connessione (listening).

Alla ricezione di una connessione, l'oggetto ServerSocket crea una socket che rappresenta la connessione TCP con il client.

Il riferimento a tale socket verrà restituito al chiamante.

Applicazioni di rete



Esempio:

```
ServerSocket providerSocket = new ServerSocket(2004, 13);

System.out.println("Waiting for connection...");

    Socket connection = providerSocket.accept();
```

Classe Socket

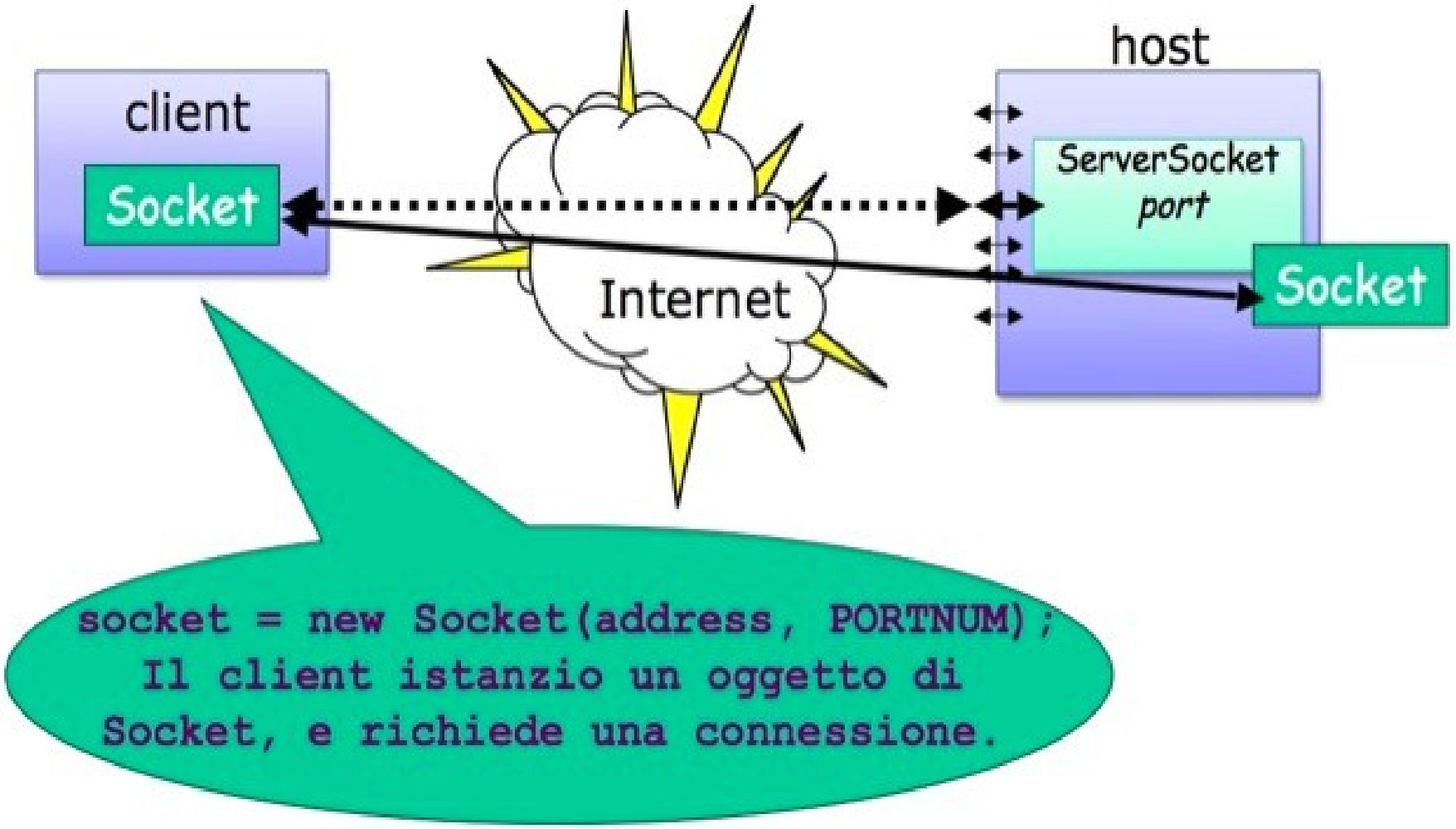
La classeSocket fornisce un'interfaccia socket per i **client** TCP.

Per aprire una connessione bisogna creare un'istanza di tale classe lato client e attendere l'apertura sulla stessa porta indicata, sull'ip della macchina alla quale si stà puntando(server).

Esempio:

Socket conn= new Socket(nomeHost, numport)

Applicazioni di rete



Esempio:

```
Socket requestSocket = new Socket("localhost", 2004);
------------------
InetAddress appoAddress =
InetAddress.getByName("maurizio-server");

Socket requestSocket = new Socket(appoAddress, 2004);
```

Streams dal Socket

Gli streams si ottengono da un oggetto di tipo socket mediante i metodi:

public InputStream getInputStream()

public OutputStream getOutputStream()

Esempio:

```
InputStream in = conn.getInputStream();
OutputStream out= conn.getOutputStream();
```

RMI
Remote Method Invocation

RMI è un insieme di API semplici e potenti che permettono di sviluppare applicazioni distribuite in rete.

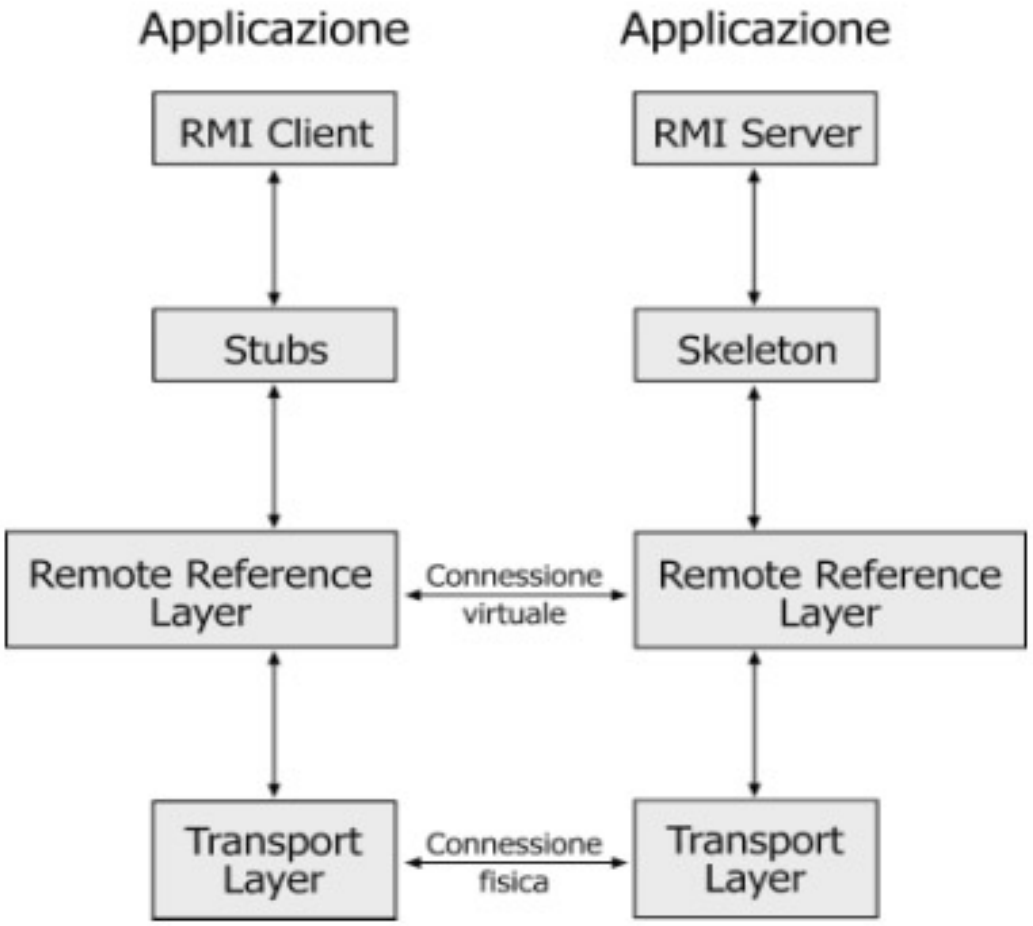
Caratteristiche di RMI:

- si scarica in locale una rappresentazione della classe remota e si lavora su di essa.

- elenganza: una volta ottenuto la rappresentazione della classe remota, detta stub (traduzione “surrogato”) il codice da inserire per richiamare il metodo remoto è uguale a quello da utilizzarsi in caso di chiamata di metodo di una classe locale: nomeIstanza.nomeMetodo(parametri di input).

- la comunicazione tra client e server avviene mediante la serializzazione degli oggetti.

Architettura di RMI:



Architettura di RMI (pt.2) - Livello applicativo:

RMI Client : applicazione che effettua le chiamate ai metodi di oggetti remoti risiedenti sul lato server

RMI Server : applicazione che gestisce gli oggetti serventi

Stub : fornisce una simulazione locale sulla JVM del client dell'oggetto remoto

Skeleton : è l'oggetto remoto in esecuzione sulla JVM del server

Architettura di RMI (pt.3):

Remote Reference Layer

RRL: instaura un collegamento virtuale tra stub e skeleton(di tipo sequenziale e per questo si richiede che i parametri da passare ai metodi siano serializzabili).

Transport Layer

TL: a questo livello si perde la concezione di oggetto remoto e/o locale in quanto si instaura un collegamento fisico per la trasmissione di sequenza di byte (oggetti serializzati) attraverso i socket su protocllo TCP/IP

Architettura di RMI (pt.4):

I livelli RRL e TL si occupano di gestire il protocollo di conversione delle invocazioni dei metodi, dell'impacchettamento dei riferimenti ai vari oggetti del passaggio dei parametri.

RMI, vediamolo in azione....
Si parte da una classe, che poi diventerà il nostro server remoto... per il momento è solo una classe con all'interno un metodo per il concatenamento di due stringhe.

```
public class RmiTestServer {
    public String concat(String a, String b) {
        return a + b;
    }
}
```

I prossimi passi saranno:

- Creare l' interfaccia remota per remotizzare la classe RmiTestServer

- Far estendere l'interfaccia java.rmi.Remote alla classe RmiTestServer

- Sollevare l'eccezione java.rmi.Remote.Exception per ogni metodo dell'interfaccia.

- Controllo dei tipi dei parametri

Creare l' interfaccia remota per remotizzare la classe RmiTestServer

Come in altri casi simili (implementazione dell'interfaccia java.io.Serializable) l'interfaccia java.rmi.Remote è vuota, non contiene alcun metodo.
Essa serve da "marcatore", cioè consente di dichiarare che una classe supporta una determinata caratteristica.
Nel creare un'interfaccia che estende java.rmi.Remote, il programmatore definisce che l'interfaccia è utilizzabile per accedere ad un oggetto remoto (cioè invocarne i metodi).

Ecco l'interfaccia:

```
public interface RmiTestServerInterface extends
  Remote {
    public String concat(String a, String b) throws
  RemoteException;
}
```

Aggiorniamo la nostra classe “server” facendole estendere la classe UnicastRemoteObject ed implementando l'interfaccia RmiTestServerInterface(al contempo solleviamo l'eccezione RemoteException per ogni metodo e costruttore dichiarati) :

```
public class RmiTestServerImplementation extends
   UnicastRemoteObject
        implements RmiTestServerInterface {

     public RmiTestServerImplementation() throws RemoteException
   { }

     public String concat(String a, String b) throws
   RemoteException {
         return a + b;
      }
}
```

Poi ci assicuriamo che tutti I parametri che vengono "trasportati" dal client al server e viceversa siano serializzabili.
E quindi che estandono l'interfaccia Serializable.

A questo punto ci serve ancora una classe server ed una client per eseguire il nostro test.
Ecco la classe server:

```
public class RmiServerTestMain {
    public static void main(String[] args) throws Exception {

  //System.setProperty("java.rmi.server.hostname","192.168.0.1
  57");
        RmiTestServerImplementation server = new
  RmiTestServerImplementation();
        java.rmi.Naming.bind("TestingRmi", server);
        System.out.println("RmiServer : bind done ...");
        System.out.println("MyService is now available ...");
    }
}
```

Prima di scrivere la nostra classe per l'invocazione del client possiamo compilare:

javac *.java

Poi creiamo il nostro stub della classe server, con:

rmic RmiTestServerImplementation

Il quale ci produrrà la classe:

RmiTestServerImplementation_Stub.class

Quindi portiamo sulla macchina sulla quale eseguiremo la classe di prova “client” I file .class rispettivamente dello stub e dell'interfaccia, quindi scriviamo la nostra classe per testare il client:

```
public class RmiClientTestMain {
    public static void main(String[] args) throws Exception {
        String serverIp = args[0] ;
        String urlRmiHost = "rmi://" + serverIp +
  ":1099/TestingRmi";
        RmiTestServerInterface serverRef =
  (RmiTestServerInterface)Naming.lookup(urlRmiHost);
        System.out.println(serverRef.concat("Hello ",
  "world !"));
    }
}
```

La compiliamo e ci ricordiamo di eseguirla con l'ip del server come argomento da linea di comando

Un'ultima annotazione riguarda il nome logico con il quale la nostra classe/il nostro servizio viene registrato all'interno della macchina che lo ospita, nel nostro caso è stata scelta
TestingRmi

Ed ancora la porta sulla quale il nostro servizio viene esposto.
Noi abbiamo deciso di usare la porta di default per le connessioni rmi che è la 1099.

Porta e nome logico del servizio vengono registrati nel rmi registry della macchina server.
La porta tramite rmiregistry tool, un demone che viene lanciato con i comandi:

rmiregistry
oppure
start rmiregistry

O se scegliamo una porta diversa da quella di default(chiaramente da indicare poi nel client)

rmiregistry numero_porta
oppure
start rmiregistry numero_porta

Solo dopo aver avviato lo rmiregistry possiamo far partire il nostro server:

java -Djava.rmi.server.hostname=192.168.0.125 RmiServerTestMain

...ed il nostro client:

java RmiClientTestMain 192.168.0.125

Si rimanda al link:

http://docs.oracle.com/javase/tutorial/rmi/running.html

http://www.mokabyte.it/1997/04/rmi.htm

http://www.mokabyte.it/2000/07/reti5_teoria.htm

E qui per l'installazione dei plugin su eclipse..:

http://www.genady.net/rmi/v20/

http://www.genady.net/rmi/v20/demos/