



# RMI

# Remote Method Invocation

---

Dr. Víctor J. Sosa Sosa



## Introducción

---

- La invocación remota de métodos de Java es un modelo de objetos distribuidos, diseñado específicamente para ese lenguaje, por lo que mantiene la semántica de su modelo de objetos locales, facilitando de esta manera la implantación y el uso de objetos distribuidos.
- En el modelo de objetos distribuidos de Java, un objeto remoto es aquel cuyos métodos pueden ser invocados por objetos que se encuentran en una máquina virtual (MV) diferente.
- Los objetos de este tipo se describen por una o más interfaces remotas que contienen la definición de los métodos del objeto que es posible invocar remotamente.



# Objetos remotos contra procedimientos remotos y sockets

- Los sistemas distribuidos requieren que las partes que los componen y que se ejecutan en diferentes espacios de direcciones, tengan la capacidad de comunicarse entre sí.
- Los *sockets* requieren que las aplicaciones implanten sus propios protocolos para codificar y decodificar los mensajes que intercambian.
- Los *Procedimientos Remotos (RPC)* se realizan mediante la invocación de funciones que se encuentran en espacios de direcciones diferentes. El sistema se encarga de empaquetar los argumentos y enviarlos al proceso que contiene el código que implementa a la rutina remota. Los sistemas codifican los parámetros de la invocación, así como los valores de vuelta en una representación externa de los datos

3



# Objetos remotos contra procedimientos remotos y sockets

- El empleo de objetos distribuidos Java, en lugar de procedimientos remotos, implica varias ventajas como la orientación a objetos misma, movilidad de las aplicaciones Java, los patrones de diseño, la seguridad, la recolección de basura distribuida, etcétera.
- La principal desventaja de los objetos distribuidos de Java, con respecto a las llamadas a procedimientos remotos y Sockets, es definitivamente *el rendimiento*

4



# RMI

---

- **Java/RMI:** Fue diseñada por **JavaSoft** para soportar llamadas a procedimientos remotos entre objetos que se ejecutan sobre Máquinas Virtuales Java (JVM).
- Se trata de una implementación independiente de la plataforma, lo que permite que tanto los objetos remotos como las aplicaciones cliente, residan en sistemas heterogéneos. Sin embargo no es independiente del lenguaje, tanto el objeto servidor Java/RMI como el objeto cliente tienen que ser escritos en Java.
- En primer lugar para que un cliente localice un objeto servidor RMI necesita un mecanismo de nombramiento *RMIRegistry* que se encuentra en la máquina servidora y mantiene la información sobre los objetos servidores disponibles. Los objetos estarán accesibles para los clientes en forma de URL.

5



## Metas del Sistema RMI

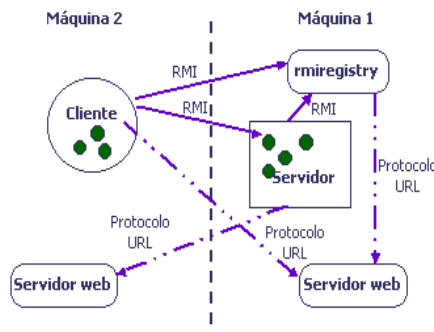
---

- Proporcionar invocación remota de objetos que se encuentran en MVs diferentes.
- Soportar llamadas a los servidores desde los applets.
- Integrar el modelo de objetos distribuidos en el lenguaje Java de una manera natural, conservando en medida de lo posible la semántica de los objetos Java.
- Hacer tan simple como sea posible la escritura de aplicaciones distribuidas.
- Preservar la seguridad proporcionada por el ambiente Java.
- Proporcionar varias semánticas para las referencias de los objetos remotos (persistentes, no persistentes y de "activación retardada").

6

# Funciones RMI

- Localizar objetos remotos.
- Comunicarse con los objetos remotos.
- Cargar el código de operación que implementa a las clases que son pasadas por valor.



- En la figura 1 se muestra una aplicación distribuida, basada en RMI, que utiliza al servidor de nombres rmiregistry para obtener referencias de objetos remotos

7

## Esqueletos y Stubs

- *Los cabos (STUBS) forman parte de las referencias y actúan como representantes de los objetos remotos ante sus clientes.*
- *Acciones:*
  - Inicia una conexión con la MV que contiene al objeto remoto.
  - empaqueta (marshals) y transmite los parámetros de la invocación a la MV remota.
  - Espera por el resultado de la invocación.
  - desempaqueta (unmarshals) y devuelve el valor de retorno o la excepción.
  - Devuelve el valor a quien lo llamó

8



# Esqueletos y Stubs

- El esqueleto es responsable de despachar la invocación al objeto remoto.
- Acciones:
  - desempaqueta los parámetros necesarios para la ejecución del método remoto.
  - Invoca el método de la implantación del objeto remoto.
  - empaqueta los resultados y los envía de vuelta al cliente.
- Tanto cabos como esqueletos, son generados por un compilador llamado rmic.

9



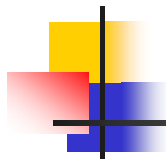
# Interface Remota

- Un objeto distribuido es accesible a través de una **Interface Remota**.
  - Métodos que pueden ser llamados de forma distribuida.
- Es una interface Java con las siguientes restricciones:
  - Debe extender la interface marca **java.rmi.Remote**.
  - Todas las operaciones deben lanzar al menos la excepción **java.rmi.RemoteException**

```
import java.rmi.*;

public interface Contador extends Remote {
    public void inc() throws RemoteException;
    public void dec() throws RemoteException;
    public void set(int valor) throws RemoteException;
    public int get() throws RemoteException;
}
```

10



# Clase Implementación

- Clase que implementa la interface remota.
- Ninguna restricción de implementación.

```
public class ContadorImpl implements Contador {  
    private int valor = 0;  
    public void inc () { valor ++; }  
    public void dec () { valor --; }  
    public void set (int valor) { this.valor = valor; }  
    public int get() { return valor; }  
}
```

11



# Activación de un Objeto Distribuido

- **JVM:**
  - Actúa de registro de los objetos distribuidos que se ejecutan en la máquina virtual.
  - Colabora con otras JVM para la comunicación
  - Identifica las llamadas a los objetos.
  - ...
  - Patrón **Broker**.
- Para que un objeto distribuido pueda recibir llamadas remotas es necesario **activarlo**:
  - **UnicastRemoteObject.exportObject(obj);**

12

# Referencias Remotas

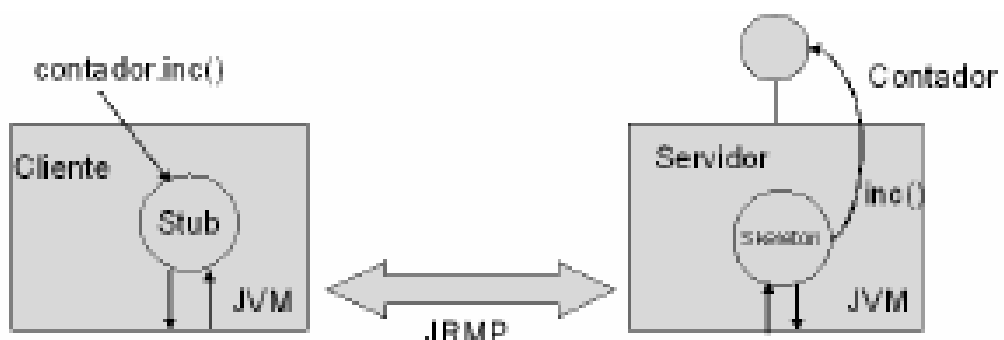
- Una **referencia remota** es una referencia Java que permite acceder al objeto distribuido:
  - Una referencia siempre apunta a un objeto en la JVM.
  - Una referencia remota apunta a un **objeto proxy**.
- Proxy (**Stub**):
  - Clase que implementa la interface remota.
  - Implementa los métodos delegando en el objeto distribuido.
  - **Transparencia para el cliente.**
- **Delegación:** protocolo JRMP\*.
  - El proxy *codifica los parámetros* de la llamada.
  - Indica a la JVM que quiere *enviar un mensaje* a un objeto distribuido (conoce su dirección de transporte e identidad).
  - *Espera la respuesta* y devuelve el valor de la llamada.

\*Java Remote Method Protocol

13

# Referencias Remotas

- **Recepción** de peticiones:
  - La JVM destino recibe una petición de acceso a un objeto distribuido.
  - Identifica el objeto y delega el procesamiento en su objeto **Skeleton**
  - La JVM destino devuelve la respuesta a la JVM peticionaria.



14



# Referencias Remotas

---

- **Compilador RMI:**
  - Genera automáticamente la clase *stub* y *skeleton* asociada a la **implementación** de un objeto remoto.
  - > **rmic** ContadorImpl
  - Trabaja con la clase *compilada* (.class).
- Modos de obtener la referencia remota (*stub*):
  - Utilizar un registro conocido por todos los procesos:
  - **Registro RMI**
  - El *servidor* almacena la referencia en el registro con un *nombre descriptivo*.
  - El *cliente* recupera la referencia utilizando el nombre.
- Otras alternativas:
  - Valor de retorno o parámetro de una llamada remota.

15



## Registro RMI

---

- Aplicación Java que actúa como registro de referencias.
- Utilizado para registrar los **objetos iniciales** de una aplicación.
- Accedemos a un registro utilizando la clase:
  - **java.rmi.registry.Naming**
  - Métodos *de clase* para:
    - **Consultar:** Contador c = (Contador) **Naming.lookup**("contador");
    - **Registrar:** **Naming.rebind**("contador", cImpl);
    - **Listar registro:** String[] referencias = **Naming.list**("//localhost");
- Lanzar el registro:
  - > **rmiregistry**
- **Restricciones:**
  - Sólo pueden modificar el registro los procesos de la misma máquina.
  - El registro no debe poder acceder a los *.class* de los objetos remotos.
- **URL RMI:**
  - [rmi:][//host/]nombre "rmi://www.cenidet.edu.mx/contador",  
"//www.cenidet.edu.mx/contador"

16





## Paso de Parámetros

- Distinta semántica para el paso de parámetros y valores de retorno:
  - Los **tipos primitivos** se pasan por valor.
  - La referencia a un **objeto distribuido** se pasa por referencia:
    - Se envía el *stub* del objeto.
  - Los **objetos normales serializables** se pasan por valor:
    - Se serializa el objeto y se envía una copia
    - No hay *semántica de réplica*.
  - El resto de **objetos no serializables** no pueden pasarse como parámetros.

17



## Descarga Dinámica de Código

- **Problema:**
  - El cliente accede al OD utilizando una copia del *stub* (referencia) que depende de la clase implementación.
  - Las clases *stub* no conviene distribuirlas.
- **Solución:**
  - Descargar las clases *stub* dinámicamente:
    - Cuando va instanciarse el proxy de la referencia remota.
- **Consecuencias:**
  - Hay que *controlar* el código descargado - > instalar un *SecurityManager*
    - Disponemos de **RMISecurityManager**.
  - Los servidores de objetos deben dejar **accesibles** sus **clases stub** utilizando un servidor web o FTP (URL).

18



# Utilización de Hilos

---

- El sistema de tiempo de ejecución de RMI no garantiza que la activación de las invocaciones de los objetos remotos se proyecte sobre diferentes hilos de ejecución. Pero debido a que es posible que las invocaciones remotas de un mismo objeto se ejecuten concurrentemente, es necesario que las implantaciones de los objetos sean **thread-safe**.

19



# Recolección de Basura

---

- RMI utiliza un algoritmo de recolección de basura basado en un contador de referencias.
- El sistema en tiempo de ejecución de RMI da seguimiento a todas las referencias activas de los objetos remotos.
- El algoritmo de recolección de basura distribuido, interactúa con el recolector de basura de la MV local, manteniendo referencias normales y débiles a los objetos.
- Implementación de una interfaz:
  - `java.rmi.server.Unreferenced`

20



# Carga dinámica de clases

---

- RMI utiliza el mecanismo de serialización de objetos de Java para transmitir datos entre máquinas, pero además agrega la información de localización necesaria para permitir que las definiciones de las clases se puedan cargar a la máquina que recibe los objetos.
- Cuando se desempaquetan los valores de retorno o los parámetros de una invocación para convertirlos en objetos activos dentro de la MV que los recibe, es necesario poseer las definiciones de las clases de todos estos objetos.

21



# Carga dinámica de clases

---

- Para soportar la carga dinámica de clases, RMI utiliza subclases especiales de:
  - Para el manejo de flujos de objetos empaquetados
    - `java.io.ObjectOutputStream` y
    - `java.io.ObjectInputStream`,
- Para incluir información sobre la localización de los archivos de clase, que contienen la definición de los objetos contenidos en el flujo se utilizan las subclases especializan al método:
  - `annotateClass` de la clase `ObjectOutputStream`
  - `resolveClass` de la clase `ObjectInputStream`,

22



# Seguridad en RMI

---

- RMI proporciona un mecanismo alternativo basado en el protocolo HTTP (confiable para el firewall), para permitir a los clientes que se encuentran detrás del firewall, invocar métodos de objetos que se encuentren del otro lado de él.

23



# Seguridad en RMI

---

- Para atravesar el firewall, la capa de transporte de RMI incluye la llamada remota dentro del protocolo HTTP, como el cuerpo de una solicitud POST, mientras que los valores de retorno se reciben en el cuerpo de la respuesta HTTP. La capa de transporte de RMI puede formular la solicitud POST, de alguna de las dos maneras siguientes:
  1. Si el proxy firewall permite entregar una solicitud HTTP directamente sobre cualquier puerto de la máquina destino.
  2. Si el proxy firewall sólo entrega solicitudes HTTP a ciertos puertos HTTP bien conocidos, la llamada se envía a un servidor HTTP que se encuentre escuchando en el puerto 80

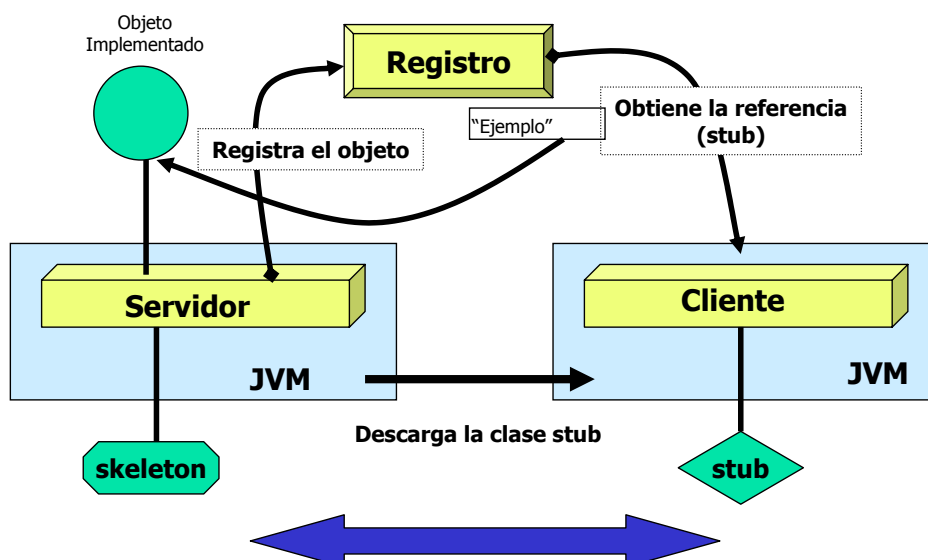
24

# Protocolo de activación

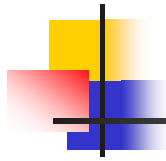
- Construcción de un objeto remoto
- Definición de la interfaz remota
- Implantación del objeto activable
- Cliente del objeto activable
- Compilación y ejecución del código

25

## Visión General



26

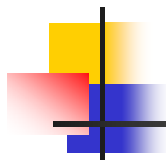


# Proceso de Desarrollo

---

- 1. Definir la ***interface remota***
- 2. Programar la ***clase implementación***
- 3. **Compilar** la *clase implementación*
- 4. Ejecutar del **compilador de stubs** con la clase compilada: **rmic**
- 5. Arrancar el registro RMI en el servidor:  
**rmiregistry**
- 6. Ejecutar la aplicación **servidor**.
- 7. Ejecutar la aplicación **cliente**.

27



# Despliegue

---

- **Servidor:**
  - Interface remota.
  - Clase implementación.
  - Programa servidor.
  - Clase *skeleton*.
  - Clase *stub* (accesible a través de URL)
- **Cliente:**
  - Interface remoto.
  - Programa cliente.

28



## Limitaciones RMI antes de Java 2

---

- Las **referencias no son persistentes**:
  - Si cae el servidor todas las referencias distribuidas dejan de ser válidas.
- **Todos los objetos** remotos deben estar **instanciados** en el servidor:
  - Disminuye el rendimiento cuando se mantienen muchos objetos.
- La **comunicación RMI no es segura**.
- Estos problemas se han solucionado en Java 2 ...
  - Pero, es preferible utilizar **CORBA** que el modelo avanzado RMI.

29



## Conclusiones

---

- La utilización de objetos en el desarrollo de sistemas distribuidos, presenta varias ventajas que permiten ocultar las dificultades inherentes a la distribución en niveles de abstracción inferiores.
- La invocación remota de métodos en Java parte del hecho de correr sobre una plataforma heterogenea.
- RMI posee todas las características de seguridad que hereda de la plataforma Java misma.

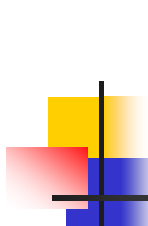
30



# Conclusiones

- Se deberá suministrar una infraestructura en la que se puedan construir aplicaciones cliente/servidor teniendo presente la evolución actual de la informática.
- Sabemos que el Java es el lenguaje que utiliza el cliente y el servidor en la Web. Por lo tanto es importante evaluar cómo cada medio se integra con Java.
- Los objetos Java deben ser capaces de comunicarse con todos los objetos de la red, también con los escritos en C++ y con los objetos *Smalltalk* (herencia de los sistemas COBOL).

31



## Comparaciones: CORBA / DCOM / RMI

Características	CORBA	DCOM	RMI
Nivel de abstracción	4	4	4
Integración con Java	4	4	4
Soporte de SO	4	2	4
Todas las implementaciones de Java	4	1	4
Facilidad de configuración	3	3	3
Invocación distribuida de métodos	4	3	3

(4 es la máxima puntuación y 0 es la mínima)

32





# Comparaciones: CORBA / DCOM / RMI

Características	CORBA	DCOM	RMI
Eficiencia	3 (3.5mseg.)	3 (3.8mseg.)	3 (3.3-5.5mseg.)
Seguridad a nivel de cable	4	4	3
Transacciones a nivel de cable	4	3	0
Referencias persistentes a objetos	4	1	0
Servicios de nombres basados en URLs	4	2	2
Invocaciones multilenguaje	4	4	0
Escalabilidad/interoperabilidad	4	2	1
Estándar abierto	4	2	2

(4 es la máxima puntuación y 0 es la mínima)

33



## PRACTICAS Y EJEMPLOS:

Ejemplo: HOLA MUNDO



# Ejemplo RMI: Hola Mundo!

*/\* HelloInterface.java \*/*

*Interfaz*

```
import java.rmi.*;
/**
 * Interfaz remota para el ejemplo "Hola Mundo!".
 */
public interface HelloInterface extends Remote {
    /**
     * Método que se invocará remotamente.
     * @return el mensaje del objeto remoto, tal como "Hola Mundo!".
     * @exception Se lanza una RemoteException si la invocación remota falla.
     */
    public String say() throws RemoteException;
}
```

35



# Ejemplo RMI: Hola Mundo!

*/\* Archivo Hello.java: \*/*  
import java.rmi.\*;  
import java.rmi.server.\*;

*Serviente*

```
/** Clase Remota para el ejemplo de "Hola, mundo!" . */
public class Hello extends UnicastRemoteObject implements HelloInterface {
    private String message;
    /** Construye un objeto remoto
     * @param msg, es el mensaje del objeto remoto, tal como "Hola, mundo!".
     * @exception se lanza RemoteException si el handle del objeto no puede ser construido. */
    public Hello (String msg) throws RemoteException {
        message = msg;
    }
    /**
     * Implementacion del método invocable de manera remota.
     * @return el mensaje del objeto remoto, tal como "Hola, mundo!".
     * @exception si la invocacion remota falla se lanza un RemoteException.
     */
    public String say() throws RemoteException {
        System.out.println ("Serviente recibio peticion de say()");
        return message;
    }
}
```

36



# Ejemplo RMI: Hola Mundo!

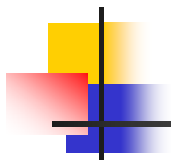
*/\* Programa Client \*/*

*cliente*

```
import java.rmi.*;
import java.rmi.Naming;
import java.io.*;

public class Client
{
    public static void main (String[] argv) {
        try {
            HelloInterface hello =
                (HelloInterface) Naming.lookup ("//localhost/Hello");
            System.out.println (hello.say());
        } catch (Exception e) {
            System.out.println ("HelloClient exception: " + e);
        }
    }
}
```

37



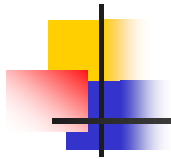
*/\* El programa Server \*/*

Servidor activador y registrador  
del objeto

```
import java.rmi.*;
import java.rmi.Naming;
import java.io.*;

public class Server
{
    public static void main (String[] argv) {
        try {
            Naming.rebind ("Hello", new Hello ("Hola, Mundo!"));
            System.out.println ("Servidor Listo!");
        } catch (Exception e) {
            System.out.println ("Servidor de Hola Mundo Fallo: " + e);
        }
    }
}
```

38



# Compilar y ejecutar un RMI

## Cómo Compilar y ejecutar el RMI:

1.- Asegurarse de que el CLASSPATH indique la ruta donde estarán las clases de nuestros programas.

2.- compilar todos los archivos .java:

```
javac *.java
```

3.- Crear los Stubs y Skeletons de los sirvientes que implementan los servicios que serán remotos (utilizamos rmic). Usamos rmic con los archivos que tengan la implementación!. :

```
rmic sirviente.class (no es necesario el .class)
```

Si no encuentra las clases, es que hay un problema con el CLASSPATH. Asegurarse de añadir el directorio actual al classpath: CLASSPATH=C:\JDK1.3\.;;

4.- Activar el rmiregistry. Asegurarse de hacerlo cada vez que quiera probar un programa nuevo. Si no se hace de esa manera surgirán cosas imprevistas.

en msdos: start **rmiregistry**

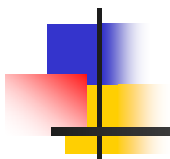
en unix: **rmiregistry** &

5.- Ejecutar primero el servidor y luego el cliente:

```
java server
```

```
java cliente
```

39



## PRACTICAS Y EJEMPLOS:

### Ejemplo: CHAT



## Ejemplo de un RMI

---

- Aplicación que puede recibir 10 usuarios(clientes) a la vez para enviarse mensajes unos a otros.
- CHAT

41



## Código: Creación Objeto Servidor

---

```
// ChatServer
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChatServer extends Remote
{
    public void register (ChatReceiver cr) throws RemoteException;
    public void send (String message) throws RemoteException;
}
```

Importa las utilerías del RMI



42

## Código: Creación Objeto Servidor

```
// ChatServer  
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

Hereda todos los atributos, variables  
y métodos o funciones

```
public interface ChatServer extends Remote  
{  
    public void register (ChatReceiver cr) throws RemoteException;  
    public void send (String message) throws RemoteException;  
}
```

43

## Código: Creación Objeto Servidor

```
// ChatServer  
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

Declara por medio del método  
`register` la estructura o variables  
que se manejarán para enviar a los  
clientes conectados el mensaje

```
public interface ChatServer extends Remote  
{  
    public void register (ChatReceiver cr) throws RemoteException;  
    public void send (String message) throws RemoteException;  
}
```

44

## Código: Creación Objeto Servidor

```
// ChatServer
import java.rmi.Remote;
import java.rmi.RemoteException;
```

Declaración de la estructura por medio de la cual se envía el mensaje al servidor

```
public interface ChatServer extends Remote
{
    public void register (ChatReceiver cr) throws RemoteException;
    public void send (String message) throws RemoteException;
}
```

45

## Código: Creación Objeto Servidor

```
// ChatServerImpl
Import java.net.MalformedURLException;

Import java.rmi.Naming;
Import java.rmi.Remote;
Import java.rmi.RemoteException;

Import java.rmi.server.UnicastRemoteObject;

Import java.util.ArrayList;
Import java.util.List;
Import java.util.Iterator;

Public class ChatServerImpl
    extends UnicastRemoteObject
    implements ChatServer
```

Declara a ChatServerImp como subclase del UnicastRemoteObject

46

# Código: Creación Objeto Servidor

```
// ChatServerImpl
```

```
Import java.net.MalformedURLException;
```

```
Import java.rmi.Naming;
```

```
Import java.rmi.Remote;
```

```
Import java.rmi.RemoteException;
```

```
Import java.rmi.server.UnicastRemoteObject;
```

```
Import java.util.ArrayList;
```

```
Import java.util.List;
```

```
Import java.util.Iterator;
```

```
Public class ChatServerImpl  
    extends UnicastRemoteObject  
    implements ChatServer
```

Se definen las funciones que se  
declararon en el ChatServer

47

# Código: Creación Objeto Servidor

```
// ChatServerImpl
```

```
{
```

```
    public static void main (String [ ] args)
```

```
    {
```

```
        try {
```

```
            new ChatServerImpl ( ).go( );
```

```
        } catch (Exception e) {
```

```
            System.err.println (e);
```

```
        }
```

```
    }
```

Declaración del programa Principal  
El cual es estándar para todos los  
Programas principales de Java

```
Private void go( )
```

```
    throws MalformedURLException, RemoteException
```

```
{
```

```
    Naming.rebind ("rmi://192.168.0.35/ChatService", this);
```

```
}
```

48



# Código: Creación Objeto Servidor

```
// ChatServerImpl
```

```
{  
    public static void main (String [ ] args)  
    {  
        try {  
            new ChatServerImpl ( ).go( );  
        } catch (Exception e) {  
            System.err.println (e);  
        }  
    }  
}
```

Crea un objeto de la clase  
ChatServerImpl para la conexión

```
Private void go( )  
    throws MalformedURLException, RemoteException  
{  
    Naming.rebind ("rmi://192.168.0.35/ChatService", this);  
}
```

49

# Código: Creación Objeto Servidor

```
// ChatServerImpl
```

```
{  
    public static void main (String [ ] args)  
    {  
        try {  
            new ChatServerImpl ( ).go( );  
        } catch (Exception e) {  
            System.err.println (e);  
        }  
    }  
}
```

Recibe el mensaje que se envía  
al querer crear el objeto, si hubo  
un error lo va a mandar a imprimir

```
Private void go( )  
    throws MalformedURLException, RemoteException  
{  
    Naming.rebind ("rmi://192.168.0.35/ChatService", this);  
}
```

50

# Codigo: Creación Objeto Servidor

```
// ChatServerImpl
```

```
{  
    public static void main (String [ ] args)  
    {  
        try {  
            new ChatServerImpl ( ).go( );  
        } catch (Exception e) {  
            System.err.println (e);  
        }  
    }  
}
```

Crea la función "go" para establecer el puerto donde se realizará la conexión

```
Private void go( )  
    throws MalformedURLException, RemoteException  
{  
    Naming.rebind ("rmi://192.168.0.35/ChatService", this);  
}
```

51

# Codigo: Creación Objeto Servidor

```
// ChatServerImpl
```

```
{  
    public static void main (String [ ] args)  
    {  
        try {  
            new ChatServerImpl ( ).go( );  
        } catch (Exception e) {  
            System.err.println (e);  
        }  
    }  
}
```

Dirección IP donde se corre el Servidor

```
Private void go( )  
    throws MalformedURLException, RemoteException  
{  
    Naming.rebind ("rmi://192.168.0.35/ChatService", this);  
}
```

52

# Código: Creación Objeto Servidor

```
// ChatServerImpl
Private List receivers;
Public ChatServerImpl ()
    throws RemoteException
{
    receivers = new ArrayList(10);
}
Public void register (ChatReceiver cr)
    throws RemoteException
{
    receivers.add (cr);
}
Public void send (String message)
    throws RemoteException
{
    System.out.println ("ChatServerImpl received: " + message);
    Iterator iter= receivers.iterator ( ) ;
    While (iter.hasNext ( )) {
        ChatReceiver cr= (ChatReceiver) iter.next ( ) ;
        cr.receive (message);
    } } }
```

Se declara la estructura del objeto de  
Conexión de los clientes que se van  
Conectando a la sesión

53

# Código: Creación Objeto Servidor

```
// ChatServerImpl
Private List receivers;
Public ChatServerImpl ()
    throws RemoteException
{
    receivers = new ArrayList(10);
}
Public void register (ChatReceiver cr)
    throws RemoteException
{
    receivers.add (cr);
}
Public void send (String message)
    throws RemoteException
{
    System.out.println ("ChatServerImpl received: " + message);
    Iterator iter= receivers.iterator ( ) ;
    While (iter.hasNext ( )) {
        ChatReceiver cr= (ChatReceiver) iter.next ( ) ;
        cr.receive (message);
    } } }
```

Se van agregando las conexiones. Se  
Registran al Servidor

54

# Código: Creación Objeto Servidor

```
// ChatServerImpl
Private List receivers;
Public ChatServerImpl ()
    throws RemoteException
{
    receivers = new ArrayList(10);
}
Public void register (ChatReceiver cr)
    throws RemoteException
{
    receivers.add (cr);
}
Public void send (String message)
    throws RemoteException
{
    System.out.println ("ChatServerImpl received: " + message);
    Iterator iter= receivers.iterator ( ) ;
    While (iter.hasNext ( )) {
        ChatReceiver cr= (ChatReceiver) iter.next ( ) ;
        cr.receive (message);
    } } }
```

Se declara el método para enviar el mensaje

55

# Código: Creación Objeto Servidor

```
// ChatServerImpl
Private List receivers;
Public ChatServerImpl ()
    throws RemoteException
{
    receivers = new ArrayList(10);
}
Public void register (ChatReceiver cr)
    throws RemoteException
{
    receivers.add (cr);
}
Public void send (String message)
    throws RemoteException
{
    System.out.println ("ChatServerImpl received: " + message);
    Iterator iter= receivers.iterator ( ) ;
    While (iter.hasNext ( )) {
        ChatReceiver cr= (ChatReceiver) iter.next ( ) ;
        cr.receive (message);
    } } }
```

Imprime todo el mensaje que recibió  
A todos los clientes conectados

56


## Código: Creación Interfaz Remota Cliente

```
// ChatReceiver
```

```
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

```
public interface ChatReceiver extends Remote  
{  
    public void receive (String message) throws  
        RemoteException;  
}
```

Declaración de la interfaz del método o procedimiento remoto y hereda atributos del `RemoteException`, tiene un parámetro que envía o recibe una variable tipo `String`

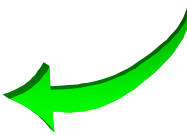


57

## Código: Creación Objeto Remoto Cliente

```
// ChatReceiverImpl  
import java.io.BufferedReader;  
import java.io.InputStreamReader;  
Import java.io.IOException;  
  
Import java.net.MalformedURLException;  
  
Import java.rmi.Naming;  
Import java.rmi.NotBoundException;  
Import java.rmi.Remote;  
Import java.rmi.RemoteException;  
  
Import java.rmi.server.UnicastRemoteObject;  
  
Public class ChatReceiverImpl  
    extends UnicastRemoteObject  
    implements ChatReceiver
```

Define a `ChatReceiverImpl` como subclase del `UnicastRemoteObject`



58

## Código:

# Creación Objeto Remoto Cliente

```
// ChatReceiverImpl
import java.io.BufferedReader;
import java.io.InputStreamReader;
Import java.io.IOException;

Import java.net.MalformedURLException;

Import java.rmi.Naming;
Import java.rmi.NotBoundException;
Import java.rmi.Remote;
Import java.rmi.RemoteException;

Import java.rmi.server.UnicastRemoteObject;

Public class ChatReceiverImpl
    extends UnicastRemoteObject
    implements ChatReceiver
```

Se definen las funciones que se  
declararon en el ChatReceiver

59

## Código:

# Creación Objeto Remoto Cliente

```
// ChatReceiverImpl
{
    public static void main (String[ ] args)
    {
        try {
            new ChatReceiverImpl ().go ();
        } catch (Exception e) {
            System.err.println(e);
        }
    }
    Private void go()
        throws IOException, MalformedURLException,
            NotBoundException, RemoteException
    {
        ChatServer cs= (ChatServer) Naming.Lookup
            ("rmi://192.168.0.80/ChatService");

        cs.register (this);
```

Declaración del programa Principal  
El cual es estándar para todos los  
Programas principales de Java

60

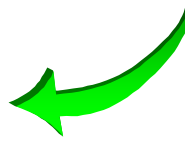
# Código: Creación Objeto Remoto Cliente

```
// ChatReceiverImpl
{
    public static void main (String[ ] args)
    {
        try {
            new ChatReceiverImpl ().go ();
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

Private void go()
    throws IOException, MalformedURLException,
        NotBoundException, RemoteException
{
    ChatServer cs= (ChatServer) Naming.Lookup
("rmi://192.168.0.80/ChatService");

    cs.register (this);
}
```

Crea un objeto de la clase  
ChatReceiverImpl para la conexión



61

# Código: Creación Objeto Remoto Cliente

```
// ChatReceiverImpl
{
    public static void main (String[ ] args)
    {
        try {
            new ChatReceiverImpl ().go ();
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

Private void go()
    throws IOException, MalformedURLException,
        NotBoundException, RemoteException
{
    ChatServer cs= (ChatServer) Naming.Lookup
("rmi://192.168.0.80/ChatService");

    cs.register (this);
}
```

Crea la función "go" para establecer  
el puerto donde se realizará la conexión con  
el Servidor



62

# Código: Creación Objeto Remoto Cliente

```
// ChatReceiverImpl
{
    public static void main (String[ ] args)
    {
        try {
            new ChatReceiverImpl ().go ();
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}

Private void go()
    throws IOException, MalformedURLException,
        NotBoundException, RemoteException
{
    ChatServer cs= (ChatServer) Naming.Lookup
    ("rmi://192.168.0.80/ChatService");
    cs.register (this);
}
```

Hace la búsqueda del  
servicio ChatService

63

# Código: Creación Objeto Remoto Cliente

```
// ChatReceiverImpl
BufferedReader br= new BufferedReader (new InputStreamReader (System.in));
String message;
while ( (message = br.readLine ()) != null) {
    cs.send (message);
}
}

Public ChatReceiverImpl ( )
    throws RemoteException
{
}

Public void receive (String message)
    throws RemoteException
{
    System.out.println (message);
}
}
```

Aquí se construye el mensaje que  
Será enviado al Servidor, para que  
éste a su vez lo envíe a los que esten  
conectados

64



# Código:

## Creación Objeto Remoto Cliente

```
// ChatReceiverImpl
BufferedReader br= new BufferedReader (new InputStreamReader (System.in));
String message;
while ( (message = br.readLine ()) != null) {
    cs.send (message);
}
}
Public ChatReceiverImpl ( )
    throws RemoteException
{
}
Public void receive (String message)
    throws RemoteException
{
    System.out.println (message);
}
}
```

Con esta función se pone a monitorear los mensajes enviados por los Clientes conectados al Servidor

65

## PRACTICAS Y EJEMPLOS:

Ejemplo: HOLA MUNDO USANDO  
APPLETS



# Ejemplo RMI Hola Mundo con Applet

---

```
package examples.hello;
```

*Interfaz*

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Hello extends Remote {  
    String sayHello() throws RemoteException;  
}
```

67



# Ejemplo RMI Hola Mundo con Applet

---

```
package examples.hello;
```

*Servidor*

```
import java.rmi.Naming;  
import java.rmi.RemoteException;  
import java.rmi.RMISecurityManager;  
import java.rmi.server.UnicastRemoteObject;  
  
public class HelloImpl extends UnicastRemoteObject implements Hello {  
  
    public HelloImpl() throws RemoteException {  
        super();  
    }  
  
    public String sayHello() {  
        return "Hola Mundo!";  
    }  
  
    public static void main(String args[]) {  
  
        System.setSecurityManager (new RMISecurityManager() {  
            public void checkConnect (String host, int port) { }  
            public void checkConnect (String host, int port, Object context) { }  
        });  
    }  
}
```

68



# Ejemplo RMI Hola Mundo con Applet

*Servidor..continua*

```
// Crea e instala un gestor de seguridad
// if (System.getSecurityManager() == null) {
//     System.setSecurityManager(new RMISecurityManager());
// }

try {
    HelloImpl obj = new HelloImpl();

    // Hace el binding de esta instancia de objeto con el nombre "HelloServer"
    Naming.rebind("//localhost/HelloServer", obj);

    System.out.println("Se hizo con exito el binding de HelloServer en el registry");
} catch (Exception e) {
    System.out.println("Error en HelloImpl: " + e.getMessage());
    e.printStackTrace();
}
}
```

69



# Ejemplo RMI Hola Mundo con Applet

*Applet*

```
package examples.hello;

import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {
    String message = "Sin_Contenido";

    // "obj" es el identificador que usaremos para referirnos
    // al objeto remoto que implementa a la interfaz "Hello"
    Hello obj = null;

    public void init() {
        try {
            obj = (Hello)Naming.lookup("//" +
                getCodeBase().getHost() + "/HelloServer");
            message = obj.sayHello();
        } catch (Exception e) {
            System.out.println("Excepcion en HelloApplet: " + e.getMessage());
            e.printStackTrace();
        }
    }

    public void paint(Graphics g) {
        g.drawString(message, 25, 50);
    }
}
```

70



# Ejemplo RMI Hola Mundo con Applet

---

Página HTML para llamar  
el Applet: Hola Mundo!

```
<HTML>
  <title>Hello World</title>
  <center> <h1>Programa Ejemplo: Hello Word!</h1> </center>
  <applet codebase="myclasses/"
        code="examples.hello.HelloApplet"
        width=500 height=120>
  </applet>
</HTML>
```

71



---

Volver a [WebObjects](#)

72