Introducción a Java RMI.

aaa

Tabla de contenidos

1 Presentación del Laboratorio.

Esta lección presenta algunos aspectos básicos de la confección de aplicaciones mediante Java RMI, y contiene la minilección y el enunciado de la Lección 3.

La tecnología de *Invocación de Métodos Remotos* (Remote Method Invocation) en Java, es especialmente sencilla, y permite con muy poco esfuerzo realizar aplicaciones completas y complejas que interaccionan entre sí mediante el principio de la invocación de objetos. Cuestiones como concurrencia, paso de objetos entre procesos, y ubicación de objetos remotos, se resuelven de un modo casi transparente para el diseñador de bajo nivel. De hecho, Java RMI es una de las plataformas preferidas para la creación de plataformas más complejas como el soporte de componentes en J2EE.

Una vez que se encuentre bien definida la interfaz de los objetos remotos que intervienen en nuestro diseño, se encuentra a nuestro alcance toda una constelación de clases y mecanismos de apoyo para la construcción de aplicaciones distribuidas. Sin embargo, es preciso ir paso a paso pues el despliegue de este tipo de aplicaciones no es del todo directo. En este minilaboratorio podrás aprender:

Durante la próxima lección de laboratorio trabajarás con una aplicación Java RMI. Verás que tras estas siglas hay un modelo de programación potente y sencillo. Como trabajo previo a la lección, te propongo el estudio, comprensión y lanzamiento de una aplicación Java RMI muy sencillita que encontrarás más adelante. Después de esta minilección sabrás:

- qué entendemos por RMI,
- de qué partes consta una aplicación RMI: una interfaz remota, un cliente y una implementación.
- cómo se crean objetos remotos y se ponen a disposición del cliente,
- cómo se invoca un método remoto, y
- cómo se lanza la aplicación.

1.1 Consejos para el Laboratorio.

Teclea mejor que copia

En primer lugar, permíteme un consejo que tiene sentido en las minilecciones de laboratorio y que, en cuanto a alumno te puede ser útil, aprovéchalas para teclear el código. Razones:

- Tendrás que hacerlo muchas veces, y tienes que ir acostumbrándote.
- Sirve para que pongas más atención en cada elemento del programa que copias, de modo que fomenta la reflexión sobre lo que copias.
- Si usas un IDE gráfico, verás cómo se comporta según vas introduciendo el texto. Esto es muy útil porque puedes aprender a aprovechar muchas sugerencias que te propone, y a desechar otras.
- Finalmente, haces tuyo el código de dos formas: (1) lo asimilas a un ritmo un poco más personal y (2) puedes cambiar identificadores y formatearlo a tu gusto.

Está claro que en el contexto de un examen o prueba, y cuando ya conoces muy bien lo que estás copiando, el método "copy & paste" es lo más apropiado, pero aprovecha esta ocasión para hacerlo de la forma que te propongo.

Usa el IDE pero también usa el terminal

Los entornos de desarrollo tienen muchas ventajas:

- liberan de tareas repetitivas que restan tiempo a tu trabajo.
- dan acceso a páginas de manual de un modo sencillo, y a veces te proponen código.
- la mayoría comprueban la validez del código sobre la marcha.

Pero también tienen desventajas:

- los parsers pueden comportarse de un modo mediocre, viendo problemas de código donde no los hay, sobre todo cuando el código del proyecto está a medias.
- proponen soluciones a corto plazo a problemas sintácticos locales que se resuelven en otro nivel o en otras partes del código.
- son estresantes: los IDE están llenos de botones, iconos e información que inunda al usuario de estímulos que excitan inecesariamente y dispersan la atención. Los botones y los menús son invitaciones a actuar, y a veces no es lo más apropiado, provocando una respuesta en el programador, que necesita reflexión y concentración. Esto es lo menos adecuado en un examen.

En lo que respecta a la programación distribuida, hay que decir que es cierto que los IDEs permiten simular muchas situaciones parecidas a la ejecución de aplicaciones distribuidas en máquinas distintas, sin embargo es importante que en algún momento tomes el control del terminal y trates de realizar despliegues manualmente. Dejando a un lado el hecho de que una simulación no es el caso real, hay que decir que en ocasiones tendrás que hacerlo así, pues la simulación en el IDE es más compleja que el uso del shell.

Usa proyectos distintos para roles distintos

Se supone que en el desarrollo de una aplicación distribuida puedes ser el programador del servidor o ser el programador de un cliente, siendo estos individuos distintos en espacio y tiempo. Si quieres hacer un buen desarrollo trabaja con proyectos separados, y resuelve los problemas a los que se tendrá que enfrentar el desarrollador de un cliente que tiene una vista limitada del proyecto del servidor.

Un paso previo, a pequeña escala consiste en considerar roles distintos de programación dentro de la misma aplicación. Por ejemplo en el desarrollo de un servidor el encargado de un componente del servicio y el encargado de la integración del componente en el programa suelen ser personas distintas; separa una clase sirviente de su clase lanzadora.

> No acumules muchas responsabilidades dentro de un mismo elemento (clase), separa funciones que no estén relacionadas en términos de roles de diseño en clases distintas.

2 Visión general de una Aplicación RMI

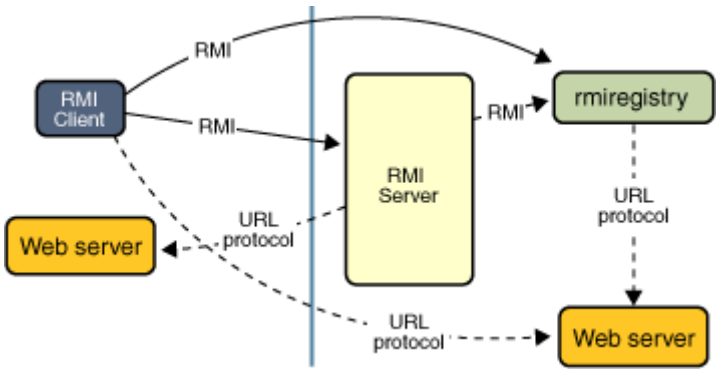
En esta minilección vas a ver cómo conseguimos con Java que un proceso puedal utilizar los métodos de otro objeto que reside en otra máquina distinta. A esto lo llamamos **invocación de métodos remotos**. La invocación de métodos remotos es una de las abstracciones más potentes que existen para construir aplicaciones distribuidas, pues nos permiten diseñar programas distribuidos rompiendo la barrera de la red de un modo *bastantetransparente para el programador* (aunque no del todo).

Las aplicaciones RMI (*Remote Method Invocation*) constan de dos programas, al menos, un cliente y un servidor. Un programa servidor típico, crea ciertos objetos remotos, los pone a disposición de los clientes a través de sus referencias, y espera que los clientes invoquen métodos de estos objetos. Un programa cliente típico, obtiene una referencia remota a uno o más objetos remotos de un servidor, y después invoca métodos de ellos. RMI proporciona el mecanismo por el cual el servidor y el cliente pueden comunicarse y pasar información de uno a otro. Este tipo de aplicaciones caen dentro de la categoría de *aplicaciones de objetos distribuidos.*

Las aplicaciones de objetos distribuidos, efectúan las siguientes tareas:

- **Localizar objetos remotos**: Las aplicaciones pueden usar diversos mecanismos para obtener referencias a los objetos remotos. Por ejemplo, una aplicación puede registrar sus objetos remotos frente a un sencillo servicio de nombres como es el registro RMI. Alternativamente, una aplicación puede pasar y devolver referencias a objetos remotos como parte de otras invocaciones remotas.
- **Comunicarse con objetos remotos**. Los detalles de la comuniación entre los objetos remotos se gestionan por el middleware RMI. Para el programador, las comunicaciónes remotas son equivalentes a invocaciones normales de objetos locales de Java.
- Cargar definiciones de clases para objetos que se pasan como argumento: dado que RMI permite pasar objetos entre los dos lados, proporciona mecanismos para carga de definiciones de clase de objetos, así como para transmitir los objetos en sí.

La siguiente ilustración muestra una aplicación RMI, que utiliza el registro RMI para obtener una referencia a un objeto remoto. El servidor llama al registro para asociar (bind) un nombre a un objeto remoto. El cliente busca el objeto remoto gracias a este nombre en el registro del servidor y después invoca un método sobre él. La ilustración también muestra el hecho de que puede existir un servidor web en el lado del servidor que puede servir para proporcionar tanto al cliente como al servidor las definiciones de las clases que sean precisas según se necesiten.



NOTA al margen: Ventajas de la carga dinámica de código

Aunque no es un punto básico de la tecnología RMI, la carga dinámica de código soporta la utilización de objetos que no han sido contemplados en el momento de la construcción de la aplicación. La plataforma RMI, facilita la carde de estas definiciones en la máquina virtual. Todos los tipos y el **comportamiento** de un objeto, previamente hecho disponible, pueden ser transmitidos de una a otra máquina virtual. RMI pasa los objetos según sus clases reales, por eso el comportamiento de los objetos no cambia cuando se envían a otra máquina virtual. Esta

posibilidad permite introducir nuevos tipos y comportameintos en una máquina virtual, extendiendo dinámicamente el comportamiento de las aplicaciones.

Interfaces, Objetos y Métodos Remotos.

Como cualquier otra aplicación Java, una aplicación distribuida que use Java RMI se compone de clases e interfaces. Las interfaces declaran métodos. Las clases implementan los métodos declarados en las interfaces y, quizás, declaran métodos adicionales. En una aplicación distribuida, algunas implementaciones podrían residir en alguna máquina virtual, pero no en otras. Los objetos con métodos que pueden ser invocados entre máquinas virtuales se denominan *objetos remotos*.

Un objeto se considera remoto cuando implementa una interfaz remota, que dispone de las siguientes características:

- extiende la interfaz java.rmi.Remote, y
- cada método de la interfaz declara lanzar java.rmi.RemoteException, además de las excepciones específicas de la aplicación.

RMI trata un objeto remoto de modo diferente a un objeto no remoto cuando su referencia se pasa de una máquina virtual Java a otra: en lugar de realizar una copia de la implementación del objeto en la máquina receptora, RMI pasa un resguardo remoto por cada objeto remoto. El resguardo actúa como un representante local, o proxy, del objeto remoto y se comporta, para el cliente, como la referencia remota. El cliente invoca métodos del resguardo (stub) local, que se hace cargo de realizar la invocación auténtica sobre el objeto remoto.

Cada resguardo de un objeto remoto implementa el mismo conjunto de métodos que la interfaz remota que implementa el objeto remoto. Esta propiedad permite moldear las referencias del stub a cualquiera de las interfaces que implementa el objeto. Sin embargo, sólo aquellos métodos definidos en una interfaz remota se encuentran disponibles para ser llamados desde la máquina virtual local.

Creación de aplicaciones distribuidas con RMI

Para poner en marcha una aplicación distribuida usando Java RMI se siguen estos pasos:

1. Diseño e implementación de los componentes de la aplicación distribuida.
2. Compilación de las fuentes.
3. Puesta en accesibilidad de las clases en la red.
4. Arranque de la aplicación.

Para esta lección preliminar, vamos a ver una aplicación muy sencilla.

3 Aplicación Hola Mundo - RMI

Esta aplicación que vamos a hacer aquí, es realmente la primera que dice "Hola Mundo", con propiedad, pues es distribuida. Se encuentra en casi cualquier curso introductorio sobre Java, y veremos que es muy sencilla aunque se puede complicar en cualquier momento. Para empezar, trataremos de hacer las cosas fáciles, aunque podremos incluir aspectos interesantes en cualquier momento.

La aplicación en concreto consta de dos procesos, un cliente y un servidor. Éste último dispone de un objeto remoto que implementa un método; al invocar este método nos devuelve un String que contiene un pequeño saludo ("Hola Mundo!"). Para realizar esta tarea de modo sencillo, sigamos los siguientes pasos:

1. Definamos la interfaz remota.
2. Implementemos el servidor.
3. Implementemos el cliente.
4. Compilemos los fuentes.
5. Arranquemos el registro RMI, el servidor y el cliente.

Los archivos precisos para este ejemplo son tres:

- Hello.java - Una interfaz remota.
- Server.java - Una implementación de la interfaz remota anterior, que genera objetos remotos.
- Client.java - Un sencillo cliente que invoca un método de la interfaz remota del objeto remoto.

Les tenemos a continuación, aunque ALTO! en una versión un poco antigua. La razón de ello es que es mejor utilizar estar versiones obsoletas para poder analizar cómo funcionan las versiones actuales de Java RMI, por eso, no utilices este código para compilar, todavía.

La interfaz remota Hello, presenta los métodos públicos de la interfaz del objeto remoto.

```
package helloServer;

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

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

El servidor "OldServer" presenta los métodos públicos y privados del objeto remoto. Has leído bien, "OldServer", más adelante te presentaré una versión más moderna, y más sencilla

```
package helloServer;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class OldServer implements Hello {

    public OldServer() {}

    public String sayHello() {
        return "Hello, world!";
    }

    public static void main(String args[]) {

        try {
            OldServer obj = new OldServer();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);

            // Bind the remote object's stub in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);

            System.err.println("Old version Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

y finalmente el Cliente de prueba.

.

```
package helloClient;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class OldClient {

    private OldClient() {}

    public static void main(String[] args) {

        String host = (args.length < 1) ? null : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registry.lookup("Hello");
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

3.1 Definición de interfaz remota

Un objeto remoto es una instancia de una clase que implementa una *interfaz remota.* Cada interfaz remota extiende la interfaz `java.rmi.Remote` y declara un conjunto de métodos remotos. Cada *método remoto* debe declarar `java.rmi.RemoteException` (o una superclase de RemoteException) en su cláusula throws, además de todas las excepciones propias.

La razón de ello se encuentra en que se añaden nuevos modos de fallo en comparación con la interfaz centralizada. El sistema RMI de Java, resulta no ser totalmente transparente frente a fallos, pero esto es positivo desde el punto de vista del programador, pues puede recuperar cierto tipo de fallos mediante mecanismos muy diversos.

Si quisiéramos detallar con precisión la causa de una excepción remota podrías subclasificar esta excepción, que según el manual contiene:

```
AccessException, ActivateFailedException,
ActivityCompletedException, ActivityRequiredException,
ConnectException, ConnectIOException, ExportException,
InvalidActivityException, InvalidTransactionException,
MarshalException, NoSuchObjectException, ServerError,
ServerException, ServerRuntimeException, SkeletonMismatchException,
SkeletonNotFoundException, StubNotFoundException,
TransactionRequiredException, TransactionRolledbackException,
UnexpectedException, UnknownHostException, UnmarshalException
```

3.2 Implementación del servidor

Una clase *servidor*, en este contexto es la que tiene un método "main" que se encarga de lanzar el servidor como tal, es decir: (1) crea una instancia del objeto remoto, (2) exporta el objeto remoto, y (3) enlaza esa instancia a un nombre, en el servicio de nombres "RMI registry". Por simplicidad, se ha unido la funcionalidad de la clase servidor, con la clase remota, de modo que el código presente un aspecto más compacto. En nuesto caso concreto, la clase "Server" que implementa la interfaz remota "Hello" contiene, además, el método "main" de servicio. A continuación se muestra el código de esta clase, con las observaciones similares de la interfaz "Hello" que ya hemos visto. Tómate la molestia de compararla con la clase Server que has descargado antes.

```
package helloServer;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server extends UnicastRemoteObject implements Hello {

   public Server() throws RemoteException { }

   public String sayHello() throws RemoteException {
      return "Hola, mundo!";
   }

   public static void main(String [ ] args) {
      try {
         Server oRemoto = new Server();
         /* Esto era necesario anteriormente, cuando no se extendía directame
nte
          * UnicastRemoteObject:
            Hello stub = (Hello) UnicastRemoteObject.exportObject(oRemoto,
 0);
          */

         /* Enlaza el stub del objeto remoto en el registro */
         Registry registro = LocateRegistry.getRegistry("localhost");
         /* Anteriormente
            registro.bind("ObjetoHello", stub);
          */
         registro.rebind("ObjetoHello", oRemoto); /* stub); anteriormente */

         System.err.println("Servidor preparado");
      } catch (Exception e) {
         System.err.println("Excepción del servidor: "+e.toString());
         e.printStackTrace();
      }
   }
}
```

La clase de implementación "`Server`" cumple con la interfaz remota "`Hello`", proporcionando una implementación del método remoto "sayHello". El método "sayHello" precisa declarar que lanza RemoteException, de la misma forma que el constructor, pues, ha de respetar la interfaz, y además, tanto la ejecución del método como la ejecución del constructor de los stubs pueden lanzar excepciones de la plataforma.

Nota: Una clase puede definir métodos no indicados en la interfaz remota, pero dichos métodos sólo podrán ser invocados desde el interior de la máquina virtual que ejecuta el servicio, y no pueden ser invocados remotamente.

Creación y exportación de un objeto remoto

El método "main" del servidor debe crear el objeto remoto que proporciona el servicio. Adicionalmente, el objeto remoto debe ser *exportado* al middleware RMI, para que pueda recibir invocaciones remotas. Esto se consigue implicitamente mediante la extensión de "UnicastRemoteObject". En las versiones anteriores a la actual (Java2 o posterior) no es preciso incluirlo explícitamente como en "OldServer". Veamos este código obsoleto:

```
Server oRemoto = new Server();
Hello stub = (Hello) UnicastRemoteObject.exportObject(oRemoto, 0);
```

El método estático "UnicastRemoteObject.exportObject()" exporta el objeto remoto indicado que recibirá invocaciones remotas entrantes mediante un puerto TCP anónimo, y devolverá el resguardo del objeto remoto para pasar a los clientes. Como resultado de "exportObject()", el sistema ya puede comenzar a oír mediante un nuevo socket de servidor o puede usar un socket de servidor compartido para aceptar las llamadas remotas entrantes hacia el objeto remoto. El stub retornado implementa el mismo número de interfaces remotas que la clase del objeto remoto, y contiene el nombre del host y el puerto con el que contactar con el objeto remoto.

Hay que insistir en que en las versiones modernas ya no es preciso efectuar el método "UnicastRemoteObject.exportObject()" a mano, pues se suele extender la clase UnicastRemoteObject. Esta es la razón, además, de porqué no es preciso enlazar "bind", puesto que el propio constructor de la clase anterior ya enlaza el objeto, sólo es preciso reenlazar con el nuevo nombre, "rebind".

Nota: Con respecto a la release J2SE 5.0 y posteriores, ya no es preciso generar mediante el compilador "rmic" (rmi compiler) las clases de stub para los objetos remotos, a menos que el objeto remoto requiera soportar clientes previos a esta versión de máquina virtual. Si tu aplicación necesita dar soporte a estos clientes, necesitarás generar clases stub para los objetos remotos, y depositar esas clases en algún sitio donde puedan ser accedidas por los clientes. Para ver más cuestions sobro cómo depositar estas clases, véase el tutorial de Java codebase tutorial.

Inscripción del objeto remoto en el registro RMI de Java

Para que el invocador (cliente, par, o applet) pueda invocar un método del objeto remoto, el invocador deberá primero obtener un stub del objeto remoto. Como mecanismo de arranque, Java RMI proporciona un API para que las aplicaciones accedan al registro de nombres, de modo que enlacen un nombre a un stub del objeto remoto, y también para que los clientes busquen objetos remotos por este nombre, con el fin de conseguir sus stub.

Un registro RMI de Java, es un servicio de nombres simplificado que permite que los clientes consigan una referencia (y un stub) al objeto remoto. En general los registros se emplean para localizar el primer objeto remoto que necesita el cliente para ponerse en marcha. Típicamente, una vez localizado ese primer objeto, el programador diseña algún otro mecanismo para proporcionarle el resto de los objetos. Un método habitual suele ser devolver referencias remotas desde los métodos remotos.

El método estático "LocateRegistry.getRegistry" sin argumentos, devuelve un stub que implementa la interfaz remota "java.rmi.Registry" y envía la invocación al registro del host local sobre el puerto por defecto 1099. Posteriormente se invoca el método "bind", sobre la referencia "registro" para enlazar la referencia del objeto remoto con el nombre "ObjetoHello". Es importante darse cuenta de que, de algun modo, el registro se comporta también como un servidor que, curiosamente, también funciona mediante la plataforma RMI.

Nota: La llamada a "LocateRegistro.getRegistry" no hace más que devolver una referencia adecuada a un registro. La llamada no comprueba si el registro está ejecutándose de verdad. Si no hubiera un registro ejecutándose en el puerto 1099 del host local cuando se invoca el método "bind", el servidor fallará y devolverá "RemoteException".

3.3 Implementación del cliente

El programa cliente obtiene un stub del registro del host del servidor, busca la referencia del objeto remoto por su nombre concreto en el registro, y después invoca el método sayHello sobre el objeto, usando el stub. He aquí un código fuente de cliente modernizado:

```
package HelloClient;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
   private Client () { }

   public static void main(String [ ] args) {
      String host = (args.length < 1) ? null : args[0];
      try {
         /* OBSOLETO: Esto está un poco rancio,
            Registry registro = LocateRegistry.getRegistry(host);
            Hello stub = (Hello) registro.lookup(***);
          */
         Hello stub = (Hello) Naming.lookup(***); /* ¿Qué hay aquí?*/

         String respuesta = stub.sayHello();
         System.out.println("[Respuesta: "+respuesta+"]");
      } catch (Exception e) {
         System.err.println("<Cliente: Excepcion: "+e);
         e.printStackTrace();
      }
   }
}
```

La versión anterior de este cliente, obtenía el stub del registro mediante "LocateRegistry.getRegistry()" con el nombre del host asociado que se indica en la línea de mandatos. Si no se indicaba ningún nombre de host, entonces usaba "null" como nombre, lo que le indica al registro que debe usar el host local.

Tanto en la versión anterior como en la actual, el cliente debe invocar cierto método lookup(), aunque partiendo de objetos distintos. En el caso antiguo es sobre el registro, y en el caso nuevo es sobree la clase Naming. Mediante este método se obtiene una referencia al objeto remoto alojado en el servidor.

Finalmente, el cliente invoca el método "sayHello()" del stub del objeto remoto, lo que desencadena las siguientes acciones.

- El lado del cliente de la plataforma abre una conexión con el servidor usando la información de host y puerto contenida en la referencia remota (stub) del objeto remoto, y serializa los datos precisos para poder enviar los objetos necesarios en la invocación.
- El lado del servidor de la plataforma acepta la llamada entrante, despacha la llamada al objeto remoto, deserializando los objetos si es preciso. Después serializa el objeto devuelto como consecuencia de la invocación remota.
- El lado del cliente de la plataforma recibe, deserializa, y devuelve el resultado al objeto que desencadenó la invocación remota.

El mensaje de respuesta devuelto por la invocación remota sobre el objeto remoto, se imprime mediante "System.out.println()".

4 Despliegue de la aplicación distribuida

El despliegue es una operación un poco delicada, y no es raro que aun en un caso sencillo podamos perdernos. Si nos olvidamos por un momento de *NetBeans* y tratamos de trabajar con los archivos de clase y los directorios en un terminal de línea de comandos, tendremos una visión más clara de las operaciones que se llevan a cabo. Para ello deberás seguir al pie de la letra las indicaciones que te damos aquí.

En primer lugar, comprueba que las direcciones de las máquinas que figuran como literales en el código son correctas, así como los posibles nombres de los objetos remotos.

1. Crea dos directorios: `cliente` y `servidor`, pues vamos a simular, con una sola computadora el comportamiento que se daría entre dos máquinas distintas. Para simplificar las cosas usa un terminal para trabajar con el cliente y otro terminal distinto para tratar con el servidor. Así tendrás dos consolas donde aparecerán los mensajes bien separados, y podremos simular (al menos un poco) el trabajo en dos máquinas distintas, cada una con su JVM.
2. Deposita una copia de los archivos del servidor y la interfaz al directorio `servidor`.
3. Lleva otra copia de los archivos del cliente y la interfaz al directorio `cliente`. Como ves, la interfaz aparece en los dos lugares, y debe ser el mismo archivo con el mismo nombre, pues la más minima diferencia conllevaría que el cliente y el servidor fueran incompatibles.

Debemos ejecutar con paciencia el despliegue de la aplicación minuciosamente. El punto de partida será un par de directorios con sus respectivos archivos `.java`:

Directorio del cliente (antes de compilar)	*Directorio del servidor (antes de compilar)*
`Client.java` **`Hello.java`**	**`Server.java`** **`Hello.java`**

A continuación, y dentro de cada directorio en cada paso, compilamos con la opción de despliegue del compilador "**`-d`**" indicando la ruta de despliegue de las clases del paquete. Es decir, en el "cliente":

```
javac -d . Hello.java
javac -d . Client.java
```

Y dentro del "servidor":

```
javac -d . Hello.java
javac -d . Server.java
```

El resultado es una estructura de carpetas que reproduce la estructura de paquetes. Si todo va bien, y no hay errores de compilación, con lo cual, tenemos algo así:

Directorio del cliente (después de compilar)	*Directorio del servidor (después de compilar)*

`Client.java`
`Hello.java`
`directorio helloClient`
`directorio helloServer`
	`Server.java`
	`Hello.java`
	`directorio helloServer`

El directorio `cliente/helloClient` debe contener: `Client.class`, y el directorio `cliente/helloServer` debe contener `Hello.class`. Mientras tanto, el directorio `servidor/helloServer` debe contener `Server.class` y `Hello.class`. Todas estas clases son fruto de la compilación con estructura de paquetes.

A continuación, debemos emplear el terminal donde ejecutaremos el servidor, y arrancar el registro rmi en background dentro del directorio del servidor así:

```
rmiregistry &
```

de este modo tendrá accesible las clases `helloServer.Server` y `HelloServer.Hello,` pues el **`rmiregistry`** es en el fondo un programa Java y necesita conocer dónde se alojan estas clases. Después, iniciaremos el servidor propiamente dicho así:

```
java helloServer.Server
```

Y finalmente, en otro terminal distinto, arrancaremos el cliente **dentro** del directorio cliente, de la siguiente forma:

```
java helloClient.Client
```

Como ves, el cliente también necesita `Hello.class`, pero en su paquete correspondiente, en definitiva: `helloServer.Hello.class.`

5 Recapitulación de la mini-práctica

Espero que este preámbulo te haya sido útil. Si no comprendes todo, no te desesperes, aunque trata de esforzarte, pues es camino que llevas andado hasta ver el capítulo teórico correspondiente sobre Java RMI. De cualquier manera debes esforzarte en dominar la mecánica de despliegue sencilla que te he presentado, pues es necesaria para completar la práctica de laboratorio completa que veremos a continuación.

Te preguntarás, sin embargo, porqué no hemos incluido esta discusión para Netbeans y Eclipse. Muy sencillo, la experiencia de unos años me ha mostrado que la forma de conseguir este resultado cambia de una versión a otra, pero no te desesperes, es preciso hacerlo primero en modo terminal para comprender como lo hará tu IDE preferido. Más adelante incluiré algunas pistas de qué debes hacer para automatizar este despliegue en el entorno gráfico, si es que no resuelves este desafío por tí mismo!!