PROGRAMACION CONCURRENTE Y DISTRIBUIDA

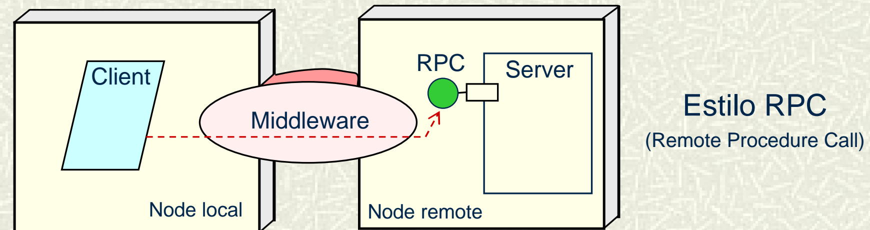
VII.1: RMI: Remote Method Invocation

José M. Drake

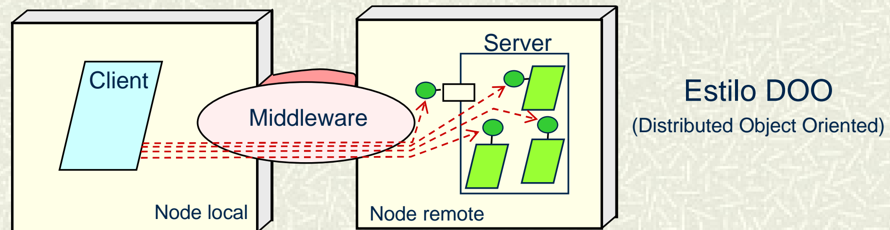
Notas:

Posibilidades que ofrece Java para la comunicación en red: Socket,RMI y URL.

Sistemas distribuidos basados en middleware



Estilo RPC
(Remote Procedure Call)



Estilo DOO
(Distributed Object Oriented)

Notas:

Modelos de infraestructura para sistemas distribuidos

- Las aplicaciones distribuidas requieren que componentes que se ejecutan en diferentes procesadores se comuniquen entre sí.
- Modelos:
  - Sockets: Facilitan la generación dinámica de canales de comunicación. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación.
  - Remote Procedure Call (RPC): Abstrae la comunicación a nivel de invocación de procedimientos. Es adecuada para programación estructurada basada en librerías.
  - Invocación remota de objetos: Abstrae la comunicación a la invocación de métodos de objetos que se encuentran distribuidos por el sistema distribuido. Los objetos se localizan por su identidad. Es adecuada para aplicaciones basadas en el paradigma OO.
- RMI (Remote Method Invocation) es la solución Java para la comunicación de objetos Java distribuidos.
  - Solución simple (Fácil de uso)
  - Solución natural (Se comporta como se espera que haga)

Notas:

Objetivos de RMI

- Proporcionar un middelware para el desarrollo de aplicaciones distribuida manteniendo un estilo Java puro y ortodoxo:
  - Facilita la interacción de objetos instanciados en diferente JVM mediante el paradigma de invocación de métodos de los objetos.
  - Integra el modelo de objetos distribuidos en el lenguaje Java de una forma natural y manteniendo la semántica que le es propia.
  - Capacita para escribir aplicaciones distribuidas tan simple como sea posible.
  - Mantiene y preserva en aplicaciones distribuidas el tipado fuerte propio de Java.
  - Proporciona diferentes modelos de persistencia de objetos distribuidos (objetos vivos, objetos persistentes, objetos con activación débil) para conseguir la escalabilidad de las aplicaciones.
  - Introduce los niveles de seguridad necesarios para garantizar la integridad de las aplicaciones distribuidas.

Notas:

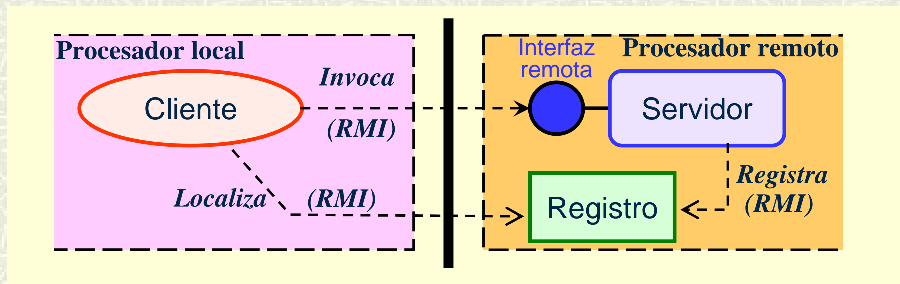
Ventajas e inconvenientes de RMI

- Ventaja:
  - Permite distribuir una aplicación de forma muy transparente, es decir, sin que el programador tenga que modificar apenas el código.
  - Las invocaciones remotas son más eficientes que las peticiones vía http que se usan con los CGIs o los Servlets.
- Inconvenientes:
  - El paso de parámetros por valor implica tiempo para hacer la serialización, enviar los objetos serializados a través de la red y luego volver a recomponer los objetos en el destino.

Notas:

Componentes de aplicaciones distribuidas RMI

- Las aplicaciones RMI se componen de:
  - Clientes: Conducen el flujo de la aplicación. Localizan e invocan métodos ofertados como remotos por los servidores.
  - Servidores: Conjunto de objetos que ofrecen interfaces remotas públicas cuyos métodos pueden ser invocados clientes de cualquier procesador de la plataforma.
  - Registro: Servicio estático que se establece en cada nudo, en el que se registran los servidores con un nombre, y donde los clientes los localizan por él.



Notas:

Servicios que ofrece RMI

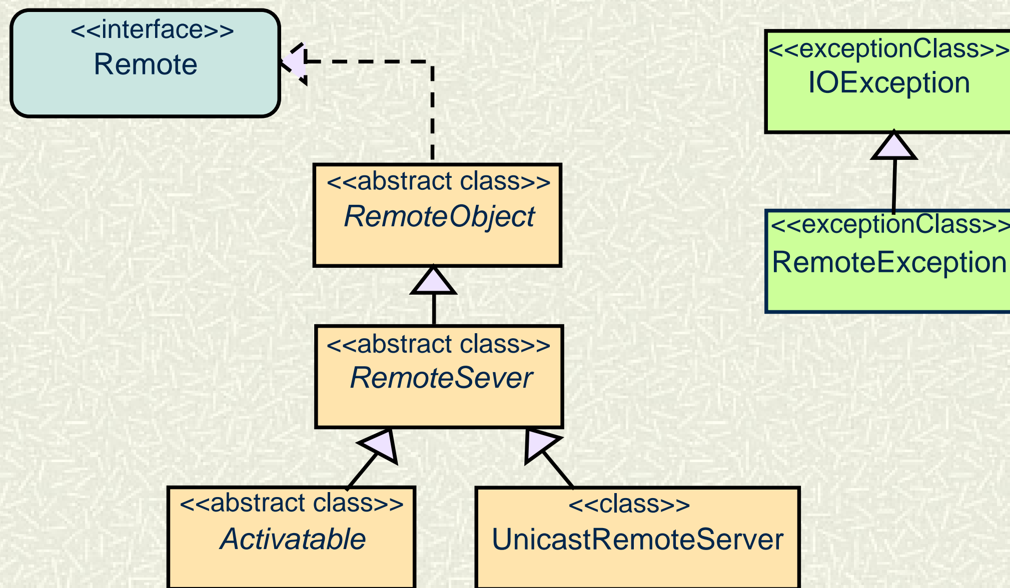
- **Localizar objetos remotos**: Hay dos metodos de localizar un objeto remoto:
  - En el ***rmiRegistry*** se pueden registrar objetos con un nombren. Otros pueden buscar su referencia a través del mismo nombre.
  - Un objeto puede pasar como parámetro u obtener como retorno de la invocación de un método repoto, la referencia a un tercer objeto, y luego usarlo para comunicarse con él.
- **Comunicar con objetos remotos**: Un objeto que disponga de la referencia remota (stub) de un objeto que ofrezca una interfaz remota, puede invocar los métodos remotos sobre él, en una forma similar a la invocación de cualquier método públicos.
- **Descargar objetos remotos**: RMI ofrece mecanismos para transferir por valor los objetos que se pasan como parámetro de los métodos que se invocan. Esta transferencia incluye la transferencia de su código (bytecodes) y de su estado.

Notas:

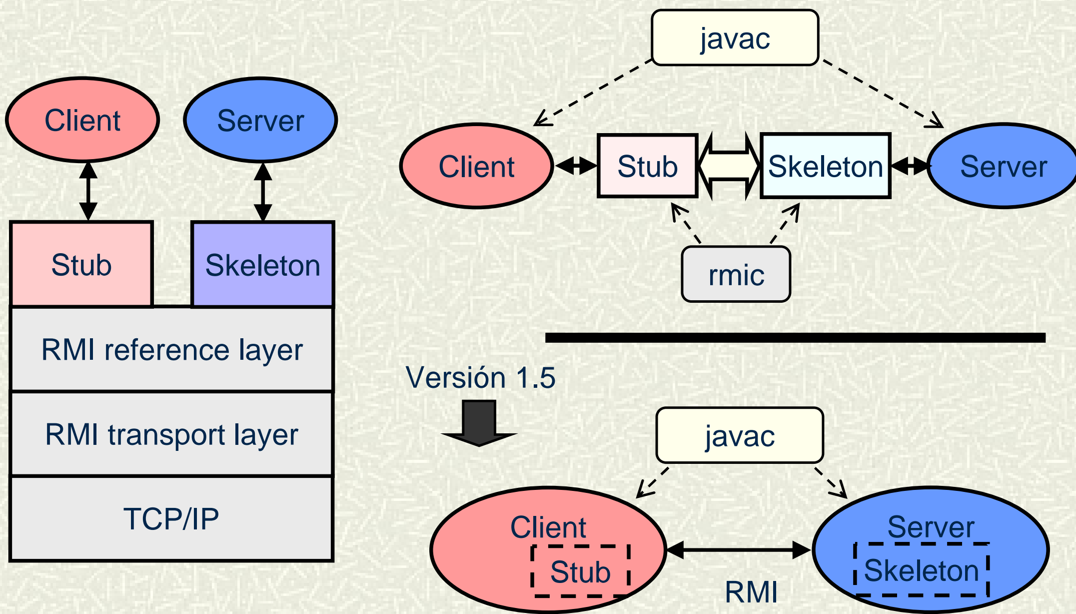
Invocaciones distribuidas y localizadas

- **Semejanzas**:
  - Una referencia a un objeto remoto puede ser pasado como parámetro de un método, y devuelto como resultados de los métodos.
  - El tipo de una referencia a un objeto remoto puede ser transformado por operaciones de *casting* siempre que sean compatibles con sus relaciones de herencias.
  - A las referencias remotas se les puede aplicar el método instanceof() para identificar dinámicamente las interfaces que soporta.
- **Diferencias**:
  - Los clientes interaccionan con los objetos remotos a través de las interfaces remotas, no a través de implementaciones o interfaces estándares.
  - Los objetos que se pasan como parámetros de métodos que no son remotos se pasan por copia (valor) y nunca por referencia.
  - Los objetos que se pasan como parámetros de métodos que se referencian por sus interfaces remotas se pasan por referencia y nunca por valor.
  - La semántica de los métodos heredados de Object es especializada para los objetos remotos.
  - Las invocaciones de objetos remoto pueden lanzar excepciones adicionales que son propias de los mecanismos de comunicación,

Notas:



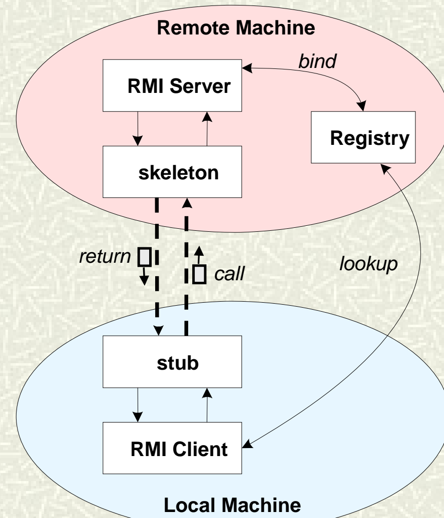
Notas:



Notas:

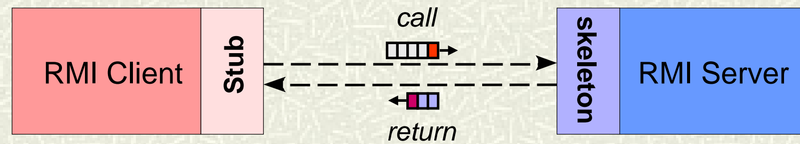
Arquitectura básica de RMI

1. El servidor debe registrarse (*bind*) en eñ *registry* bajo un nombre.
2. El cliente localiza (*lookup*) la referencia de servidor en el *registry* por su nombre.
3. El cliente crea un stub que referencia al skeleton
4. El cliente invoca localmente el *stub*.
5. El *stub* transfiere como un mensaje la invocación al *skeleton*.
6. El *skeleton* invocamente localmente el método del servidor.
7. El *skeleton* transfiere al *stub* como mensaje los resultados obtenidos de la invocación.
8. El *stub* finaliza la invocación del cliente retornándole los resultados.



Notas:

Stub y skeleton



1. Uncliente invoca un método remoto invocando localmente el mismo método en el stub.
2. The stub genera un mensaje que contiene: la referencia al método y un stream de bytes que resulta de secuencializar los parámetros del método.
3. El stub crea dinámicamente un socket y establece la conexión con el skeleton.
4. El skeleton recibe el mensaje los decodifica y delega en un thread la invocación del método del servidor. Quedando dispuesto de nuevo a la recepción de un nuevo mensaje (no siempre es multithread).
5. El thread genera un mensaje con el stream de bytes que corresponde a la secuencilización de los resultados de la invocación.
6. El thread envía por el socket abierto el mensaje de retorno.
7. El stub decodifica el mensaje y concluye la invocación inicial retornando los resultados al cliente.

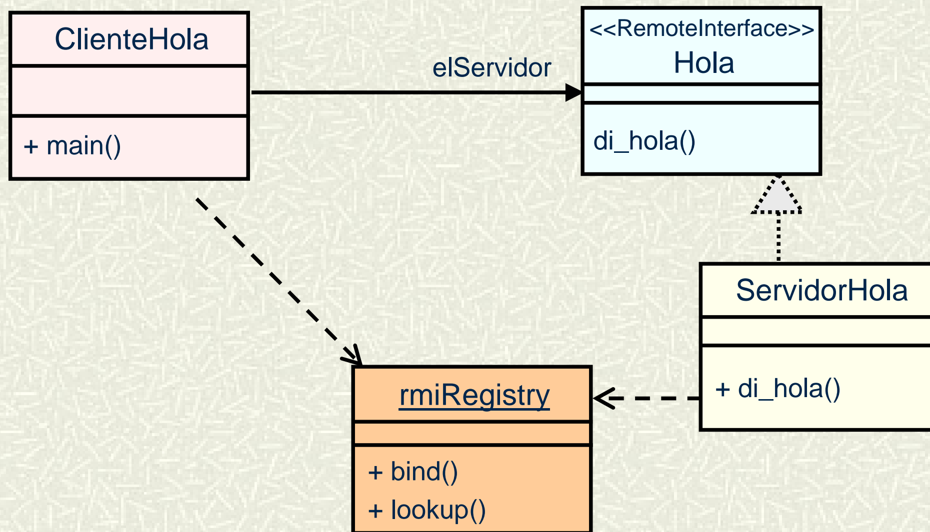
Notas:

Pasos para desarrollar una aplicación distribuida RMI

1. Se define la interfaz remota
2. Se desarrolla el servidor que implementa la interfaz remota.
3. Se desarrolla el cliente.
4. Se compilan los ficheros Java fuentes.
5. Se ejecuta el RMI Registry en el procesador remoto.
6. Se ejecuta el servidor en el procesador remoto.
7. Se ejecuta el cliente en el procesador local.

Notas:

Ejemplo HolaMundo



Notas:

1- HolaMundo: Interfaz remota fichero Hola.java

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

public interface Hola extends Remote {
		String di_hola() throws RemoteException;
}
```

- Es una interfaz remota porque extiende a la interfaz **Remote**
- Todos los métodos de la interfaz remota debe lanzar explícitamente la excepción **RemoteException**

Notas:

2- HolaMundo: Servidor remoto ServidorHola.java

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class ServidorHola implements Hola {
    // Constructor
    public ServidorHola() {}
    // Implementación del metodo remoto
    public String di_hola() { return "Hola, Mundo!"; }

    public static void main(String args[]) {
        try {
            //Instancia del servidor
            ServidorHola obj = new ServidorHola();
            // Casting a la interfaz declarada como remota
            Hola stub = (Hola) UnicastRemoteObject.exportObject(obj, 0);
            // Se autoregistra en el RMI Registry
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind("Servidor_Hola", stub);
           System.err.println("Server ready");
        } catch (Exception e) { System.err.println("Server exception: " + e.toString())}
    }
}
```


Notas:

3. HolaMundo: Fichero ClienteHola.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class ClienteHola {
      private ClienteHola() {}    // Constructor
      public static void main(String[] args) {
              // Si no hay parámetro de entrada el servidor es local
              String elHost=null;
              if (args.length >= 1) elHost= args[0];
              try {
                     // Se localiza el servidor en el registro por su nombre "Servidor_Hola" .
                     Registry registry = LocateRegistry.getRegistry(elHost);
                     Hola elServidor = (Hola) registry.lookup("Servidor_Hola");
                     // Se invoca el servicio remoto
                    String respuesta = elServidor.di_hola();
                    System.out.println("Respuesta:  " + respuesta);
              } catch (Exception e) {System.err.println("Excepción del cliente: " + e.toString());}
      }
}
```


Notas:

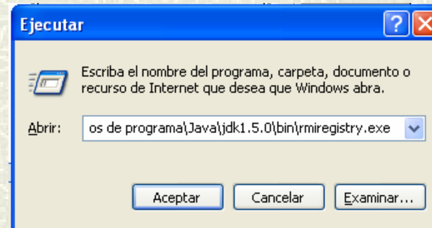
4. HolaMundo: Compilación de los ficheros Java

- Se compila de forma ordinaria (con **javac**) los ficheros :
  - Ficheros del servidor:
    - ServidorHola.java
    - Hola.java
  - Ficheros del cliente:
    - ClienteHola.java
    - Hola.java

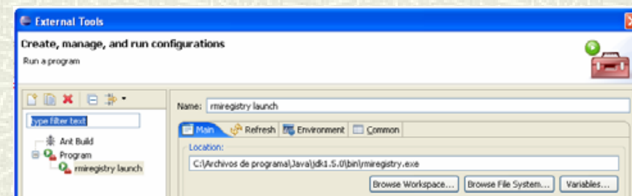
Notas:

5. HolaMundo: Instanciacion del RMI Registry

- Se lanza la ejecución del ***rmiregistry*** como un proceso en el procesador del servicio.
  - Hay que ejecutar independientemente el programa **rmiregistry.exe** que existe en el jdk de java.
- Desde Windows:



- O desde Eclipse:

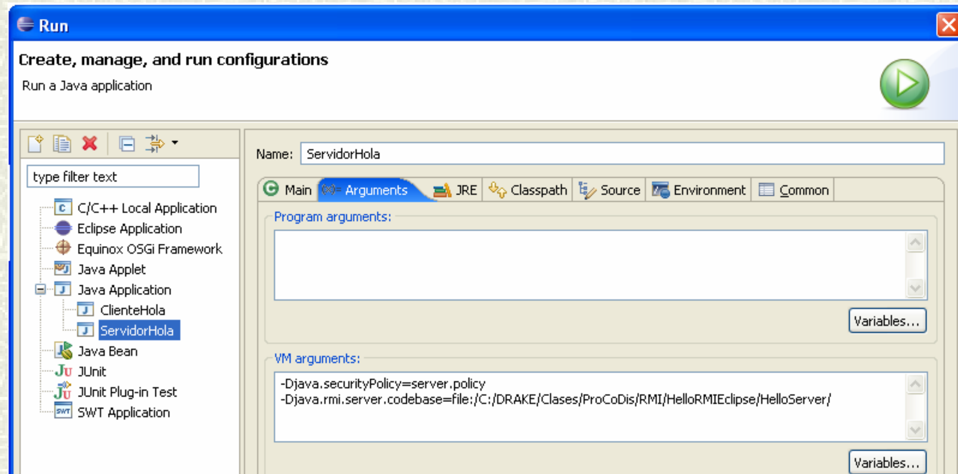


Notas:

6. HolaMundo: Ejecución del servidor

El servidor se ejecuta en el procesador remoto

- Hay que establecer propiedades de JVM



- Hay que establecer una política de seguridad adecuada.

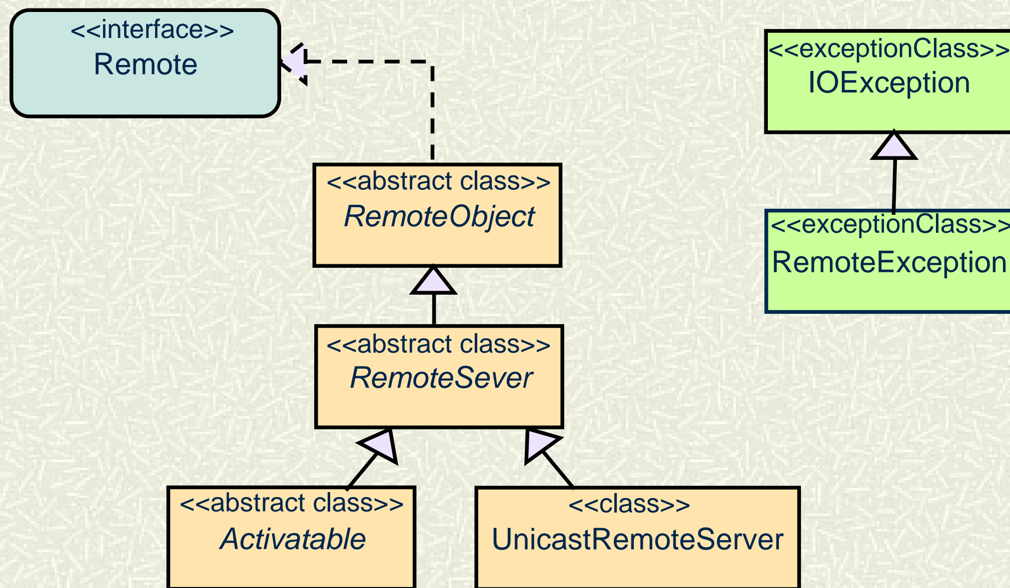
```
grant {
        permission java.security.AllPermission;
};
```


Notas:

7. HolaMundo: Ejecución del Cliente

- El cliente se ejecuta en el procesador local.

Notas:



Notas:

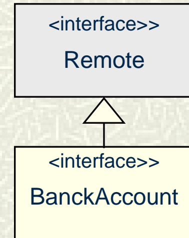
Interfaz java.rmi.Remote

- Una interfaz remota es aquella que ofrece métodos que pueden ser invocado por clientes instanciados en otra JVM.
- Para que una interfaz sea remota se requiere:
  - Debe extender directa o indirectamente a la interfaz java.rmi.Remote
  - Todos los métodos que se declaran en ella deben satisfacer los siguientes requisitos:
    - Los métodos remotos deben incluir la declaración de la excepción java.rmi.RemoteException.
    - Los parámetros o valores de retorno de los métodos remotos, que hagan referencia a objetos remotos deben ser declarados por su interfaz remota.
- La interfaz java.rmi.Remote es la interfaz raíz que debe ser extendida por cualquier interfaz que se declara como remota.
  - No declara ningún método.
- Una interfaz remota puede extender otras muchas interfaces, sean o no sean remotas, siempre que los métodos definidos en ellas satisfagan los criterios de los métodos remotos.

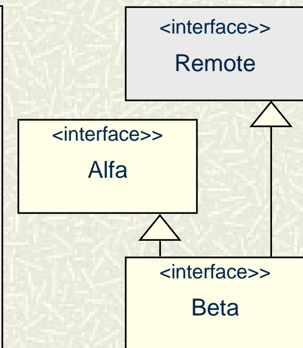
Notas:

Ejemplos de interfaces remotas

```
// Ejemplo de interfaz remota
public interface BankAccount extends java.rmi.Remote {
    public void deposit(float amount) throws java.rmi.RemoteException;
    public void withdraw(float amount)
                    throws OverdrawnException, java.rmi.RemoteException;
    public float getBalance() throws java.rmi.RemoteException;
}
```




```
// Ejemplo de interfaz remota derivada de otra interfaz no remota
public interface Alfa{
    public final String okay="Constante es Okey tambien";
    public Object foo(Object object) throws java.rmi.RemoteException;
    public void bar() throws java.rmi.RemoteException;
    public int baz() throws java.rmi.RemoteException;
}
public interface Beta extends Alfa, java.rmi.Remote {
   public void ping() throws java.rmi.RemoteException;}
```



Notas:

Clase java.rmi.RemoteException

- Es la clase raíz de las excepciones que puede lanzar el middleware RMI durante una invocación de un método remoto.
- Es lanzada cuando la invocación de un método remoto falla por alguna razón propia del mecanismo RMI (no de la lógica de negocio del servidor), tales como:
  - Fallos en la comunicación.
  - Fallos en los procesos de serialización o recomposición de los parámetros o resultados.
  - Errores de protocolo.
- Es una excepción chequeada y no una RuntimeException, y por tanto su lanzamiento debe ser declarado en todos los métodos remotos, y su gestión es chequeada por el compilador.

Notas:

Clase abstracta RemoteObject

- Representa la clase raíz de los objetos remotos. Aporta la semántica de las referencias remotas.
- Adecua algunos métodos de Object para la semántica de los objetos remotos:

boolean **equals** (Object obj)

// Compares two remote objects for equality.

int **hashCode**()

// Returns a hashcode for a remote object. Two remote object stubs that
// refer to the same remote object will have the same hash code

String **toString**()

// Returns a String that represents the value of this remote object.

static Remote **toStub**(Remote obj)

// Returns the stub for the remote object obj passed as a parameter.
// This operation is only valid after the object has been exported.

Notas:

Clases especializadas java.rmi.Server

- Clase abstracta Java.rmi.Server: Aporta a RemoteObject el framework de las referencias a objetos remotos. Specifically, the functions needed to create and export remote objects (i.e. to make them remotely available) :

static String **getClientHost**()

// Returns a string representation of the client host for the remote

// method invocation being processed in the current thread.

static PrintStream **getLog**()

// Returns stream for the RMI call log.

log.static void **setLog**(OutputStream out)

// Log RMI calls to the output stream out.

Notas:

Clase java.rmi.server.UnicastRemoteObject

- Define un objeto remoto concreto cuya referencia sólo es válida mientras que el thread del server está vivo (*alive*)
- Sirve para implementar un objeto remoto, esto es un objeto que implementa una o varias interfaces remotas.
- La clase UnicastRemote Objectr proporciona los métodos para crear objetos remotos y exportarlos (Hacerlos disponibles para los clientes remotos).
- Exportar un objeto remoto es declararlo en RMI para que quede a la escucha de un determinado socket TCP .
- Varios objetos remotos pueden quedar a la espera de un mismo socket TCP.
- Un objeto remoto puede definir nuevos métodos públicos, pero esto no son remotos y sólo pueden ser invocados localmente.

Notas:

Exportación de un objeto remoto mediante herencia

- Un objeto remoto puede extender la clase UnicastRemoteObject para haciendo uso de su constructor exportar sus objetos.
- Constructores:

protected **UnicastRemoteObject**()

// Creates and exports a new UnicastRemoteObject object using
// an anonymous port.

protected **UnicastRemoteObject**(int port)

// Creates and exports a new UnicastRemoteObject object using the
// supplied port.

protected **UnicastRemoteObject**(int port, RMIClientSocketFactory csf,
RMIServerSocketFactory ssf)

//Creates and exports a new UnicastRemoteObject object using the
// supplied port and socket factories.

Notas:

Exportación de un objeto remoto sin herencia

- Se utilizan los métodos estáticos exportObject(...) que exportan un objeto remoto, y retornan un objeto stub a un RemoteObject.

```
static RemoteStub exportObject( Remote obj)
    // Exports the remote object to make it available to receive incoming calls
    // using an anonymous port.
static Remote exportObject( Remote obj, int port)
    // Exports the remote object to make it available to receive incoming calls,
    // using the particular supplied port.
static Remote exportObject( Remote obj, int port,
            RMIClientSocketFactory csf,
            RMIServerSorcketFactory ssf)
    // Exports the remote object to make it available to receive incoming calls,
    // using a transport specified by the given socket factory.
```


Notas:

Otros métodos de UnicastRemoteObject

- Métodos:

clone()

// Returns a clone of the remote object that is distinct from the original.

static unexportObject(Remote obj, boolean force)

// Removes the remote object, obj, from the RMI runtime.

// If successful, the object can no longer accept incoming RMI calls.

*// If the **force** parameter **is true**, the object is forcibly unexported even if*

// there are pending calls to the remote object or the remote object still

// has calls in progress.

*// If the **force** parameter **is false**, the object is only unexported if there are*

// no pending or in progress calls to the object.

Notas:

Clase abstracta java.rmi.server.activation.Activatable

- Clase abstracta que define un objeto remoto que se activa cuando es invocado algún método remoto, y que se desactiva cuando se le requiere.

static Remote **exportObject**(Remote obj, ActivationID id, int port)
  //Export the activatable remote object to the RMI runtime to make the
  // object available to receive incoming calls.

protected ActivationID **getID**()
  //Returns the object's activation identifier.static

boolean **inactive**(ActivationID id)
  // Informs the system that the object with the corresponding id is inactive.

static Remote **register**(ActivationDesc desc)
  // Register an object descriptor for an activatable remote object so that is can be
  // activated on demand.

static boolean **unexportObject**(Remote obj, boolean force)
  // Remove the remote object, obj, from the RMI runtime.

static void **unregister**(ActivationID id)
  // Revokes previous registration for the activation descriptor associated with id.

Notas:

Ejemplo de implementación de un servidor remoto.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class BankAccountImpl extends UnicastRemoteObject
                                                    implements BankAccount {
    private float balance = 0.0;
    public BankAccountImpl(float initialBalance) throws RemoteException {
        balance = initialBalance;
    }
    public void deposit(float amount) throws RemoteException {  ...  }
    public void withdraw(float amount) throws OverdrawnException,
                                                    RemoteException {... }
    public float getBalance() throws RemoteException {...}
}
```


Notas:

Paso de objetos no remotos en invocaciones de métodos remotos

- Los métodos remotos pueden tener como parámetro o como valor de retorno cualquier clase de objeto siempre que sea serializable. Esto es, o es primitivo o implementa la interfaz *java.io.Serializable.*
- Un objeto no remoto que es pasado como parámetro o resultado en la invocación de un método es pasado por copia.
- Cuando un objeto no remoto es pasado como parámetro, es primero serializado, luego es transferido a la JVM remota y luego se invoca el método haciendo referencia a la copia.
- Cuando un objeto no remoto es retornado como resultado por un método, se serializa el objeto, se transfiere a la JVM local y luego se retorna la referencia de la copia al thread que invocó.
- Cuando un objeto es transferido de una JVM a otra, también transfiere la anotación de la clase que implementa el objeto, así que la clase y sus métodos pueden ser cargado en la JVM que lo recibe.

Notas:

Paso de objetos remotos en invocaciones de métodos remotos

- Cuando se pasa un objeto remoto como un parámetro, o se retorna como resultado, sólo se transfiere el stub al objeto. A través de él, el que lo posee puede invocar remotamente el objeto pasado.
- Un objeto remoto no es pasado nunca por copia.

Notas:

Localización de objetos remotos

- RMI proporciona una aplicación denominada *rmiregistery* que permite almacenar las referencia a los objetos remotos que se han instanciado en la JVM con un nombre que lo identifica como clave.
- Un objeto remoto sólo puede registrarse en un registro existente en el propio procesador.
- Un cliente que quiere invocar un objeto primero debe disponer de su referencia.
  - La referencia puede haberse recibido como un parámetro pasado o como un valor retornado en la invocación de un método remoto.
  - El cliente puede localizar la referencia en el rmiregistry si conoce el identificador con el que el objeto buscado fue registrado.

Notas:

Interfaz Registry

- La interfaz registry define los métodos que ofrece un objeto de tipo registry para almacenar y recuperar objetos remotos identificados con nombres como claves.

  void **bind**(String name, Remote obj)

  // Binds a remote reference to the specified name in this registry.

  String[] **list**()

  // Returns an array of the names bound in this registry.

  Remote **lookup**(String name)

  // Returns the remote reference bound to the specified name in the registry.

  void **rebind**(String name, Remote obj)

  // Replaces the binding for the specified name in this registry with the

  //supplied remote reference.

  void **unbind**(String name)

  // Removes the binding for the specified name in this registry.

Notas:

Clase LocateRegistry

- Se utiliza para obtener la referencia (obtener el stub) de un registro sobre un procesador determinado

static Registry **createRegistry**(int port)
// Creates and exports a Registry instance on the local host that accepts requests on the port.
static Registry **createRegistry**(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
// Creates and exports a Registry instance on the local host that uses custom socket factories
// for communication with that instance.
static Registry **getRegistry**()
// Returns a reference to the the remote object Registry for the local host on the port 1099.
static Registry **getRegistry**(int port)
// Returns a reference to the the remote object Registry for the local host on the port.
static Registry **getRegistry**(String host)
// Returns a reference to the remote object Registry on the specified host on the port 1099.
static Registry **getRegistry**(String host, int port)
// Returns a reference to the remote object Registry on the specified host and port.
static Registry **getRegistry**(String host, int port, RMIClientSocketFactory csf)
// Returns a locally created remote reference to the remote object Registry on the specified
// host and port.

Notas: