

Desarrollo de Aplicaciones Distribuidas

Índice

Presentación	5
Red de contenidos	6
Sesiones de aprendizaje	
SEMANA 1 : Java Sockets	7
SEMANA 2 : Threads	17
SEMANA 3 : Java Input / Output Streams I	29
SEMANA 4 : Java Input / Output Streams II	39
SEMANA 5 : Serialización de Objetos	49
SEMANA 6 : RMI: Remote Method Invocation	57
SEMANA 7 : EXAMEN PARCIAL DE TEORÍA	
SEMANA 8 : EXAMEN PARCIAL DE LABORATORIO	
SEMANAS 9 y 10 : Java Web Services	69
SEMANAS 11 y 12 : JMS: Java Message Services	79
SEMANA 13 : Asesoría Trabajo Práctico	
SEMANA 14 : Asesoría Trabajo Práctico	
SEMANA 15 : EXAMEN FINAL DE LABORATORIO	
SEMANA 16 : SUSTENTACIÓN DE PROYECTOS	
SEMANA 17 : EXAMEN FINAL DE TEORÍA	

Presentación

El presente material tiene por objeto apoyar el proceso de enseñanza-aprendizaje a los alumnos participantes del curso de Desarrollo de Aplicaciones Distribuidas.

La documentación contenida en este volumen brinda a los alumnos del curso las herramientas y conocimientos necesarios para implementar una solución de Java Networking, en la cual diferentes objetos pertenecientes a una misma aplicación podrán encontrarse físicamente en diferentes servidores y Java Virtual Machines.

Para llegar al objetivo trazado, el trabajo se dividirá en 2 fases:

Fase 1: Programación Java en entornos de Red y Distribuidos

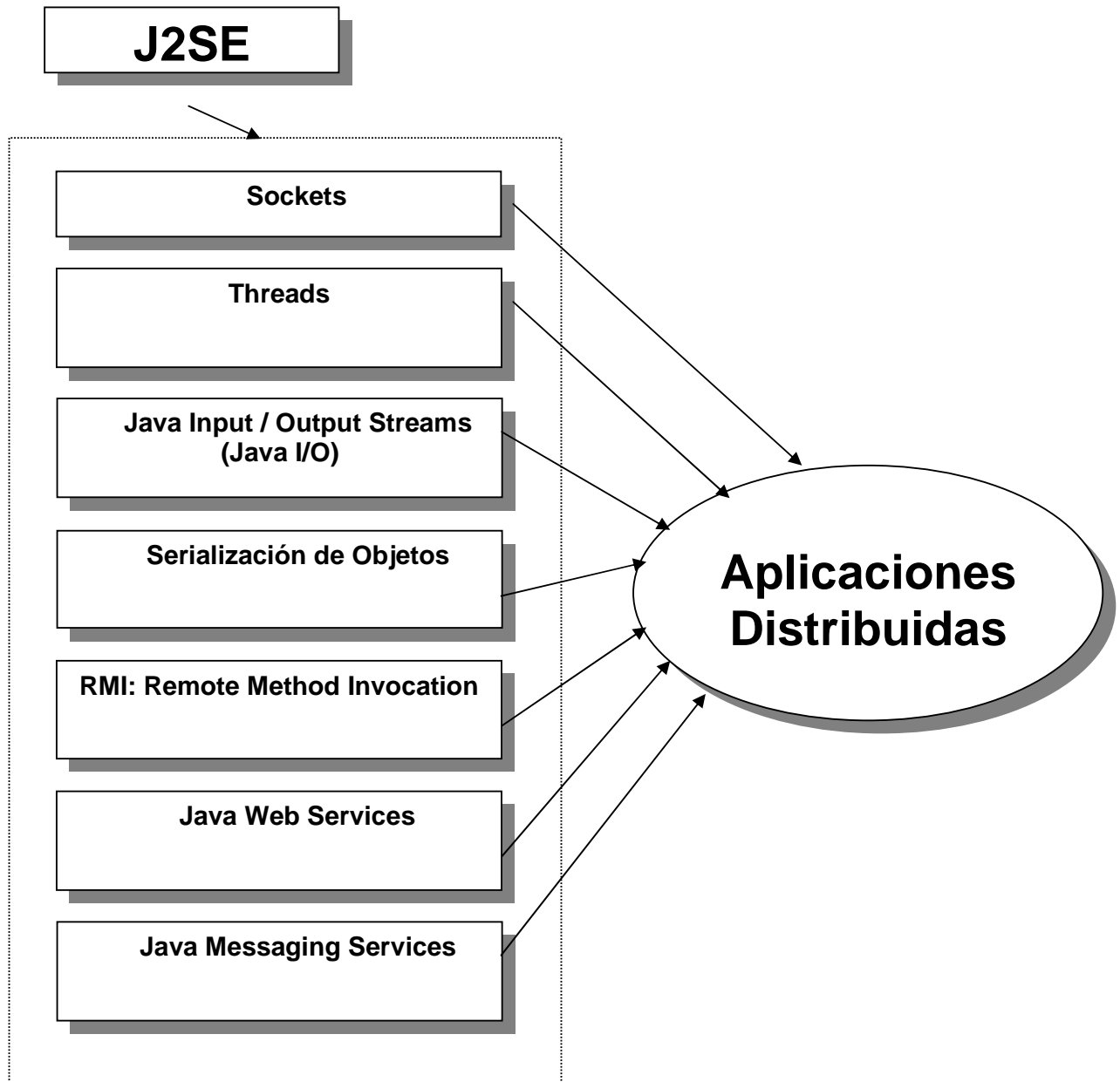
- ❖ Se definen conceptos básicos de Programación Java orientada a entornos de red, pre-requisito para el desarrollo y comprensión de los casos prácticos desarrollados en la segunda etapa.
- ❖ Los alumnos dominarán los conceptos y aplicaciones puntuales para la creación de Client Sockets, Server Sockets, Java RMI, Manejo de Flujos en Java (I/O), Web Services y Servicios de Mensajería.

Fase 2: Casos Prácticos

- ❖ Implementación de Casos Prácticos de aplicación directa de los conceptos aprendidos en la Fase 1. Se desarrollarán plantillas de utilidad general que puedan ser empleadas en cualquier desarrollo e-business.

Sin más que mencionar, el autor, profesor responsable del curso, con mucho gusto atenderá y agradecerá todo comentario o sugerencia al presente material.

Red de contenidos





Sockets

OBJETIVOS ESPECÍFICOS

- Reconocer la estructura básica de un Socket
- Determinar la utilidad de los objetos Socket y ServerSocket

CONTENIDOS

- Definición
- La clase ServerSocket
- Ciclo de vida de un servidor

ACTIVIDADES

- Intercambia información entre un Socket Cliente y un Socket Servidor

1. DEFINICION

Un Socket es una conexión entre dos host. Bajo esta conexión, se pueden ejecutar las siguientes operaciones básicas:

- conectarse a una máquina remota;
- enviar datos;
- Recibir datos;
- cerrar una conexión;
- escuchar datos entrantes;
- aceptar conexiones de máquinas remotas.

La clase java Socket, la cual es usada tanto por aplicaciones cliente como aplicaciones servidor, tiene métodos que permiten implementar las cuatros primeras operaciones; las restantes, son exclusivas de aplicaciones servidor. Las aplicaciones servidor siempre esperan por aplicaciones cliente que se conecten a ellas y son implementadas por la clase ServerSocket.

Los programas java utilizan normalmente sockets cliente de la siguiente manera:

1. El programa crea un nuevo socket utilizando el constructor Socket()
2. El socket intenta conectarse a un host remoto.
3. Una vez establecida la conexión, las máquinas local y remota obtienen flujos de entrada y salida desde el socket y usan estos flujos para intercambiar información.
4. Cuando la transmisión de datos ha sido completada, una de las dos partes, o ambas, cierran la conexión.

Actividad

1. Cree la clase **ScannerPuertos** dentro del paquete `aprendamos.java.socket` e implemente la funcionalidad mostrada a continuación:

```
import java.io.IOException;
import java.net.*;

public class ScannerPuertos {

    String host = "localhost";

    ScannerPuertos(){

        // --- Verificando puertos del host
        for(int i=1;i<1024;i++){
            System.out.println("iteracion nro: "+i);
            try {
                Socket s = new Socket(host,i);
                System.out.println("Hay un servidor en el puerto "+i);
                System.out.println(s.getLocalAddress());
                System.out.println(s.getLocalPort());
                System.out.println(s.getPort());
                System.out.println(s.getInetAddress());

                // El puerto local es asignado automaticamente en
                // tiempo de ejecucion
            } catch (IOException e) {
                // No hay servidor en este puerto
            }
        }
    }
}
```

```
        } catch (UnknownHostException e) {  
            // TODO Auto-generated catch block  
            System.out.println("Excepcion: "+e);  
            break;  
        } catch (IOException e) {  
            // No debe existir un servidor en este puerto  
        }  
    }  
}  
  
public static void main(String args[]){  
    ScannerPuertos ps = new ScannerPuertos();  
}
```

Ahora modifique la aplicación previa para que reciba como parámetros el puerto de inicio y el puerto de fin a verificar. Indique si los puertos 7,8 y 25 están disponibles en el servidor de prueba.

2. Cree la clase **ClienteDiaHora** dentro del paquete `aprendamos.java.socket` e implemente la funcionalidad mostrada a continuación:

```
import java.io.IOException;
import java.net.*;
import java.io.*;
public class ClienteDiaHora {

    String host = "localhost";
    int puerto = 13;

    public ClienteDiaHora() {
        try {
            // --- creando el socket
            Socket s = new Socket(host,puerto);

            // --- Obteniendo el flujo de entrada
            BufferedReader entrada=

            new BufferedReader(new InputStreamReader(s.getInputStream()));
            System.out.println(entrada.readLine());
            // -- cerramos el socket
            s.close();

        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (IOException e) {
            // No existe un servidor en este puerto
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ClienteDiaHora ce= new ClienteDiaHora();
    }
}
```

Pruebe la funcionalidad. ¿Se genera alguna una excepción ? De ser afirmativa la respuesta, determine si falta algún componente para su correcto funcionamiento.

2. LA CLASE SERVERSOCKET

La clase `ServerSocket` contiene todo lo necesario para poder escribir un servidor en Java. Por ejemplo:

- ✓ Tiene constructores que crean objetos `ServerSocket`.
- ✓ Métodos que escuchan por conexiones en un puerto específico.
- ✓ Métodos que retornan un objeto `Socket` (cliente) cuando una conexión es realizada, de modo que se puedan enviar y recibir datos.

El ciclo de vida de un servidor básico es el siguiente:

1. Un nuevo `ServerSocket` es creado utilizando un puerto específico.
- 2.El `ServerSocket` escucha por intentos de conexión entrantes usando su método `accept()`.
- 3.`accept()` bloquea hasta que un cliente intenta hacer una conexión, en ese caso se retorna un objeto `Socket` conectándose el cliente y el servidor.
- 4.Dependiendo del tipo de servidor se invocan los métodos `getInputStream` y `getOutputStream`.
- 5.El servidor y el cliente interactúan de acuerdo a un protocolo preestablecido hasta que se debe cerrar la conexión.
- 6.El servidor, el cliente, o ambos cierran la conexión.
- 7.El servidor retorna al paso 2 y espera por la siguiente conexión.

Actividad

1. Cree la clase **ServidorDiaHora** dentro del paquete `aprendamos.java.socket` e implemente la funcionalidad mostrada a continuación:

```
import java.io.IOException;
import java.io.*;
import java.net.*;
public class ServidorDiaHora {
    int portdiahora=13;
    Socket conexion;
    public ServidorDiaHora() {
        try {
            ServerSocket ss = new ServerSocket(portdiahora);
            while(true){
                conexion=ss.accept();

                // --- obteniendo flujo de salida
                PrintWriter out= new
                PrintWriter(conexion.getOutputStream(),true);
                // --- retornando mensaje
                out.println("La fecha y hora es "+new
                java.util.Date());
                conexion.close();

            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        ServidorDiaHora se= new ServidorDiaHora();

    }
}
```

Ejecute la aplicación **ServidorDiaHora** y luego pruebe nuevamente la aplicación **ClienteDiaHora**. Verifique la correcta ejecución de la aplicación cliente.

2. Cree la clases **ClienteEcho** y **ServidorEcho** mostradas a continuación. Luego, ejecute la aplicación y verifique su correcto funcionamiento (Retorno desde el servidor del mensaje enviado por el cliente).

ClienteEcho

```
import java.io.IOException;
import java.net.*;
import java.io.*;
public class ClienteEcho {
    String host = "localhost";
    int puerto = 8;
    public ClienteEcho() {
        try {
            // --- creando el socket
            Socket s = new Socket(host,puerto);
            // --- Obteniendo el flujo de entrada
            BufferedReader entrada=
                new BufferedReader(new
InputStreamReader(s.getInputStream()));

            // --- Obteniendo el flujo de salida
            PrintWriter salida=new PrintWriter( s.getOutputStream(),true);

            salida.println("Hola, esta es una prueba");
            System.out.println(entrada.readLine());

            // -- cerramos el socket
            s.close();
        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // No existe un servidor en este puerto
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ClienteEcho ce= new ClienteEcho();
    }
}
```

ServidorEcho

```
import java.io.IOException;
import java.io.*;
import java.net.*;

public class ServidorEcho {

    int portecho=8;
    Socket conexion;

    public ServidorEcho() {
        try {
            ServerSocket ss = new ServerSocket(portecho);

            while(true){
                conexion=ss.accept();

                // --- obteniendo data del cliente
                BufferedReader entrada=new BufferedReader(new
InputStreamReader(conexion.getInputStream()));

                // --- obteniendo flujo de salida
                PrintWriter out= new
PrintWriter(conexion.getOutputStream(),true);

                // --- retornando mensaje
                out.println("Respuesta de servidor
"+entrada.readLine());

                conexion.close();
            }

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ServidorEcho se= new ServidorEcho();
    }
}
```


Para recordar

- La clase `java.net.ServerSocket` tiene todo lo necesario para escribir servidores en Java. Se pueden implementar servidores web, servidores de archivos, servidores de correo, etc.
- La clase `java.net.Socket` permite implementar conexiones cliente hacia un servidor.

Autoevaluación

- Implemente un servidor de “consulta de empleados”, el cual recibirá código de empleado y retornará el nombre completo obteniéndolo de una base de datos. Usted deberá definir la estructura de la tabla y los registros de prueba respectivos.



Threads

TEMA

Threads : Hilos de ejecución en Java

OBJETIVOS ESPECÍFICOS

- Reconocer la utilidad de la clase Thread y la interface Runnable
- Implementar aplicaciones que definan, instancien e inicien nuevos hilos usando la clase Thread y la interface Runnable

CONTENIDOS

- Definición
 - Thread Principal
- Creación de Threads
 - Interface Runnable
 - Clase Thread

ACTIVIDADES

- Se implementa un servidor de Echo utilizando threads

1. DEFINICIÓN

Los Threads son hilos de ejecución que simulan ser ejecutados en paralelo dentro de un programa principal. Un thread comparte memoria y datos con el resto del programa Principal.

El uso de threads en java nos permite contar con un entorno de ejecución asíncrono, es decir, varias tareas pueden ser ejecutadas de manera simultánea.

1.1 El Thread Principal

El entorno de ejecución de java distingue entre threads de usuario y threads demonios (daemon threads). Mientras un thread de usuario siga activo, la ejecución del programa no podrá ser terminada. Por otro lado, los threads demonios están a merced del sistema de ejecución: estos son detenidos si no hay más threads de usuario ejecutándose.

Los threads demonios sólo existen para servir a los threads de usuario.

Cuando una aplicación standalone es ejecutada, un thread de usuario es automáticamente creado para ejecutar el método main. Este thread es llamado el thread principal (main thread). Si ningún otro thread es lanzado, el programa termina cuando el método main termina de ser ejecutado. Todos los otros threads, llamados threads hijos (child threads) son lanzados desde el thread principal, heredando su estatus. En este caso, el thread principal puede terminar, pero el programa continuará ejecutándose hasta que todos los threads de usuario hayan concluido.

El estatus de un thread (demonio o usuario) puede ser cambiado utilizando el método:

setDaemon(boolean)

La invocación a este método sólo puede ser hecha cuando el thread aún no ha sido iniciado (no se ha invocado su método start).

2. CREACIÓN DE THREADS

Un thread en java es representado por un objeto de la clase Thread. Podemos implementar threads en java de dos maneras:

Implementando la interface Runnable

```
class MyClass implements Runnable {  
    public void run() {  
        //Cuerpo del método en blanco  
    }  
}
```

Heredando de la clase Thread:

```
class MyClass extends Thread {  
  
}
```

El método run() de la interface Runnable es utilizado para definir una unidad de código de programa. Un thread creado, basado en un objeto que implementa la interface Runnable, ejecutará el código definido en el método run(). En otras palabras, el código en el método run () define un hilo de ejecución independiente y por lo tanto la **entrada** y la **salida** del thread.

2.1 Creación de Threads con la interface Runnable

El procedimiento para crear threads basados en la interface Runnable es el siguiente:

- ✓ Una clase implementa la interface runnable proporcionando el método run que será ejecutado por el thread.

```
class MyClass implements Runnable {  
    public void run() {  
        //Cuerpo del método en blanco  
    }  
}
```

- ✓ Un objeto de la clase thread es creado. Un objeto de una clase implementando la interface runnable es pasado como argumento al constructor del Thread.

```
MyClass mc= new MyClass();  
Thread th= new Thread (mc);
```

- ✓ Se invoca al método start del objeto de la clase Thread.

2.2 Instanciando e iniciando un Thread

Aunque el código que se ejecuta en un thread está en el método run, no necesitamos llamar directamente a este método para ejecutarlo, en vez de ello, invocamos al método start de la clase Thread.

```
MyClass mc = new MyClass();  
Thread t = new Thread(mc);  
t.start();
```

Un hilo se inicia a través del método start.

Actividad

1. Implemente las clases **Fabula** y **Animal** mostradas a continuación.

Fabula

```
public class Fabula {  
  
    public Fabula(){  
        // --- creando una liebre  
        Animal laliebre = new Animal("L",5);  
        Animal latortuga = new Animal("T",1);  
        latortuga.run();  
        laliebre.run();  
  
    }  
  
    public static void main(String[] args) {  
        Fabula f = new Fabula();  
    }  
  
}
```

Animal

```
public class Animal {

    String nombre;
    int velocidad;

    Animal(String nombre, int velocidad){
        this.nombre=nombre;
        this.velocidad=velocidad;
    }

    public void sleep(int tiempo){
        for(int i=0;i<tiempo;i++){
            ;
        }
    }

    public void run(){

        for(int i=0;i<10;i++){
            System.out.print(nombre);
            sleep(1000/velocidad);
        }
        System.out.println("\n"+nombre+" ha llegado!!");
    }

}
}
```

Si se ejecuta la clase Fabula, obtendrá algo similar a lo siguiente:

```
TTTTTTTTTT
T ha llegado
LLLLLLLLLL
L ha llegado
```

En el ejemplo, la liebre no tiene ninguna oportunidad de llegar en primer lugar, porque el programa no se interesa por ella hasta que ha llegado la Tortuga.

Sin embargo, si transformamos las clases Fabula y Animal a threads, si podremos obtener ejecuciones simultáneas:

FabulaTh

```
public class FabulaTh {  
  
    public FabulaTh(){  
        // --- creando una liebre  
        AnimalTh laliebre = new AnimalTh("L",5);  
        AnimalTh latortuga = new AnimalTh("T",1);  
        //latortuga.run();  
        //laliebre.run();  
        latortuga.start();  
        laliebre.start();  
  
    }  
  
    public static void main(String[] args) {  
        FabulaTh f = new FabulaTh();  
    }  
  
}
```

AnimalTh

```

public class AnimalTh extends Thread{

    String nombre;
    int velocidad;

    AnimalTh(String nombre, int velocidad){
        this.nombre=nombre;
        this.velocidad=velocidad;
    }

    /*public void sleep(int tiempo){
        for(int i=0;i<tiempo;i++){
            ;
        }
    }*/

    public void run(){

        for(int i=0;i<10;i++){
            System.out.print(nombre);
            try {
                Thread.sleep(1000/velocidad);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        System.out.println("\n"+nombre+" ha llegado!!");
    }

}

```

A diferencia del ejemplo anterior, la clase Animal hereda de Thread y la llamada al método run (), método de la clase Animal, ha sido reemplazada por la llamada al método start (), método de la clase Thread.

La ejecución de este programa da como resultado:

```

TLLLLLTLLLLL
L ha llegado
TTTTTTTTT
T ha llegado

```

1. Utilizando Sockets y Threads, implemente las clases ClienteEcho y ServidorEcho mostradas a continuación:

ClienteEcho

```
import java.io.IOException;
import java.net.*;
import java.io.*;

public class ClienteEcho extends Thread{
    String host = "localhost";
    int puerto = 8;
    String mensaje;
    int veces;

    public ClienteEcho(String mensaje, int veces) {
        this.mensaje=mensaje;
        this.veces=veces;
    }

    public static void main(String[] args) {
        ClienteEcho c1= new ClienteEcho("Hola que tal",5);
        ClienteEcho c2= new ClienteEcho("Como estas",8);
        c1.start();
        c2.start();
    }

    public void run() {
        // TODO Auto-generated method stub
        try {
            // --- creando el socket
            Socket s = new Socket(host,puerto);
            // --- Obteniendo el flujo de entrada
            BufferedReader entrada=
                new BufferedReader(new
                    InputStreamReader(s.getInputStream()));
            // --- Obteniendo el flujo de salida
            PrintWriter salida=new PrintWriter( s.getOutputStream(),true);
            for (int i=0;i<veces;i++){
                salida.println(mensaje);
                try {
                    Thread.sleep(veces*100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```

for (int i=0;i<veces;i++){
    System.out.println(entrada.readLine());
}
// -- cerramos el socket
s.close();

} catch (UnknownHostException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (IOException e) {
    // No existe un servidor en este puerto
    e.printStackTrace();
}

}

}

```

ServidorEcho

```

import java.io.*;
import java.net.*;
public class ServidorEcho {
    int portecho=8;
    Socket conexion;
    public ServidorEcho() {
        try {
            ServerSocket ss = new ServerSocket(portecho);
            while(true){
                System.out.println("Servidor: escuchando por
clientes");
                conexion=ss.accept();
                Coneccion c=new Coneccion(conexion);
                c.start();
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    class Coneccion extends Thread{
        Socket s;
    }
}

```

```
        public Coneccion(Socket s){
            this.s=s;
        }
        public void run() {

            BufferedReader entrada;
            try {

                entrada = new BufferedReader(new
InputStreamReader(s.getInputStream()));
                // --- obteniendo flujo de salida
                PrintWriter out= new PrintWriter(s.getOutputStream(),true);
                String linea=null;
                // --- retornando mensaje
                while((linea=entrada.readLine())!=null){
                    out.println(linea);
                    System.out.println("retornando "+linea+"
"+this.getName());
                }

                s.close();

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }

    }

    public static void main(String[] args) {
        ServidorEcho se= new ServidorEcho();
    }
}
```

¿ Cómo puede verificar en el lado del servidor, que se están ejecutando peticiones de manera simultáneas?

Autoevaluación

Implemente el thread de la clase Animal utilizando la interface Runnable.

Para recordar

- Los Threads son hilos de ejecución que simulan ser ejecutados en paralelo dentro de un programa principal.
- El método run() de la interface Runnable es utilizado para definir una unidad de código de programa.



Java Input / Output Streams I

TEMA

Flujos de entrada y de salida

OBJETIVOS ESPECÍFICOS

- Implementar aplicaciones que utilicen objetos de la clase File
- Implementar aplicaciones que utilicen objetos de las clases FileInputStream y FileOutputStream

CONTENIDOS

- Definición
 - La clase File
- File Streams

ACTIVIDADES

- Se implementan funcionalidades utilitarias de manejo de archivos y directorios

1. DEFINICIÓN

El paquete `java.io` proporciona un conjunto de clases para manejar inputs y outputs dentro de nuestras aplicaciones. Java utiliza flujos (streams) como un mecanismo general para administrar el ingreso y salida de datos. Existen dos tipos básicos de flujos:

- ✓ flujos de bytes;
- ✓ flujos de caracteres.

Un flujo de entrada es un objeto que una aplicación puede utilizar para leer una secuencia de datos. Un flujo de salida es un objeto que una aplicación puede utilizar para escribir una secuencia de datos. Un flujo de entrada actúa como una **fuentes** de datos, y un flujo de salida actúa como un **destino** de datos.

Las siguientes entidades pueden actuar como flujos de entrada y / o flujos de salida:

- ✓ un arreglo de bytes y caracteres;
- ✓ un archivo;
- ✓ una conexión de red.

Los flujos pueden ser encadenados a filtros para proporcionarnos funcionalidad adicional. Adicionalmente a los bytes y caracteres los flujos permiten manejar la entrada y salida de valores java primitivos o de objetos.

1.1 La clase File

La clase `File` proporciona una interface general al sistema de archivos de la plataforma en la que nos encontremos. Un objeto `File` representa el nombre de la ruta de un archivo o carpeta en el sistema de archivos del host. Una aplicación puede usar la funcionalidad proporcionada por la clase `File` para administrar archivos y carpetas en el sistema de archivos, más no puede manejar el contenido de un archivo. Para este propósito existen las clases:

```
FileInputStream;  
FileOutputStream;  
RandomAccessFile.
```


El nombre de ruta para un archivo o carpeta es especificado usando las convenciones de nombrado del sistema en el que nos encontremos. Algunos ejemplos de nombres de rutas son los siguientes:

/book/capitulo1	en Unix
c:/book/capitulo1	en Windows
HD:book:capitulo1	en Macintosh

2. FILE STREAMS

Las clases `FileInputStream` y `FileOutputStream` definen flujos de entrada y salida de bytes que están conectados a archivos. Los datos sólo pueden ser leídos o escritos como una secuencia de bytes.

Un flujo de entrada para leer bytes puede ser creado utilizando los siguientes constructores:

```
FileInputStream(String name) throws FileNotFoundException  
FileInputStream(File file) throws FileNotFoundException  
FileInputStream(FileDescriptor fd)
```

El archivo puede ser especificado por su nombre, a través de un objeto `File` o utilizando un objeto `FileDescriptor`.

Si el archivo no existe, se lanza la excepción `FileNotFoundException`.

Un flujo de salida para escribir bytes puede ser creado usando los siguientes constructores:

```
FileOutputStream(String name) throws FileNotFoundException  
  
FileInputStream(String name, boolean append) throws  
FileNotFoundException  
  
FileOutputStream(File file) throws IOException  
  
FileOutputStream(FileDescriptor fd)
```

El archivo puede ser especificado por su nombre, a través de un objeto File o utilizando un objeto FileDescriptor.

Si el archivo no existe, éste es creado. Si el archivo existe, su contenido es eliminado, a menos que se haya utilizado el constructor adecuado para indicar que la salida debe ser **agregada** al archivo.

La clase FileInputStream implementa los métodos **read()** definidos en la superclase InputStream. De manera similar, la clase FileOutputStream implementa los métodos **write()** definidos en la superclase OutputStream.

Actividad

1. Implemente la clase **ListandoDirectorio** mostrada a continuación.

ListandoDirectorio

```
import java.io.*;
public class ListandoDirectorio {
    String dirname="c:/";
    ListandoDirectorio(){
        File directorio = new File(dirname);
        if(!directorio.exists()){
            System.out.println("error no existe "+dirname );
            return;
        }
        if(!directorio.isDirectory()){
            System.out.println("error "+dirname+" no es un directorio" );
            return;
        }
        String[] nombres=directorio.list();

        try {
            System.out.println("abspath: "+directorio.getAbsolutePath());
            System.out.println("canpath: "+directorio.getCanonicalPath());
            System.out.println("name: "+directorio.getAbsolutePath());
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        for(int i=0;i<nombres.length;i++){
            System.out.println(nombres[i]);
        }
    }

    public static void main(String[] args) {
        ListandoDirectorio ld= new ListandoDirectorio();
    }
}
```

2. Implemente la clase **InformaFile** mostrada a continuación.

InformaFile

```
/**
 * Ejemplo que presenta información sobre el fichero que se pasa como
 * parámetro en el momento de la ejecución
 */
import java.io.*;

class InformaFile {
    public static void main( String args[] ) throws IOException {
        // Se comprueba que nos han indicado algún fichero
        if( args.length > 0 ) {
            // Vamos comprobando cada uno de los ficheros que se hayan pasado
            // en la línea de comandos
            for( int i=0; i < args.length; i++ ) {
                // Se crea un objeto File para tener una referencia al fichero
                // físico del disco
                File f = new File( args[i] );

                // Se presenta el nombre y directorio donde se encuentra
                System.out.println( "Nombre: "+f.getName() );
                System.out.println( "Camino: "+f.getPath() );
                // Si el fichero existe se presentan los permisos de lectura y
                // escritura y su longitud en bytes
                if( f.exists() ) {
                    System.out.print( "Fichero existente" );
                    System.out.print( (f.canRead() ? " y se puede Leer" : "" ) );
                    System.out.print( (f.canWrite() ? " y se puese Escribir" : "" ) );
                    System.out.println( "." );
                    System.out.println( "La longitud del fichero es de "+
                        f.length()+" bytes" );
                }
                else {
                    System.out.println( "El fichero no existe." );
                }
            }
        }
        else {
            System.out.println( "Debe indicar un fichero." );
        }
    }
}
```

4. Implemente la clase **GeneraFile** mostrada a continuación.

GeneraFile

```
/**
 * Este ejemplo muestra la utilización de canales de entrada y salida
 * formados por arrays de bytes hacia fichero.
 * Presenta un campo de entrada de texto cuyo contenido es grabado en un
 * fichero cuando se pulsa el botón.
 * Una vez pulsado el botón se presenta el contenido de otro fichero
 * y vuelve a empezar.
 */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

class GeneraFile extends JFrame implements ActionListener {
    JLabel texto;
    JLabel pulsado;
    JButton boton;
    JButton botonClick;
    JPanel panel;
    JTextField campoTexto;

    // Constructor
    GeneraFile() {
        // Etiquetas informativas a colocar en la parte superior, indicando
        // Cuando se va a salvar a fichero y cuando se ha leído de fichero
        setFont( new Font("Helvetica",Font.PLAIN,14) );
        texto = new JLabel( "Texto a grabar en fichero:" );
        pulsado = new JLabel( "Texto recuperado de fichero:" );
        // Boton que se presenta cuando se va a grabar el contenido del
        // campo de texto en el fichero
        boton = new JButton( "Grabar" );
        boton.addActionListener( this );

        // Botón que se presenta una vez recuperado el texto de fichero,
        // para volver a comenzar el ciclo
        botonClick = new JButton( "Reiniciar" );
        botonClick.addActionListener( this );

        // Campo de texto cuyo contenido se va a grabar a fichero
        campoTexto = new JTextField( 20 );

        // Contenedor de los componentes que se han creado
        panel = new JPanel();
    }
}
```

```

panel.setLayout( new BorderLayout() );
panel.setBackground( Color.white );
getContentPane().add( panel );
// Se incorporan los componentes al panel
panel.add( "North",texto );
panel.add( "Center",campoTexto );
panel.add( "South",boton );
}

// Este es el método en que se realizan todas las operaciones que
// involucran a los ficheros
public void actionPerformed((ActionEvent evt) {
    // Obtenemos el origen del evento, ya que sólo tratamos los que
    // provengan del click del botón
    Object source = evt.getSource();
    // Directorio raíz del usuario
    String dir = System.getProperty( "user.home" );

    // Si se trata del botón Inicial
    if( source == boton ) {
        JLabel etiqueta = new JLabel();

        try {
            // Porción de código encargada de grabar en fichero el contenido
            // que se introduzca en el campo de texto
            // Primero se recupera el texto del campo de texto y se convierte
            // a un array de bytes
            String texto = campoTexto.getText();
            byte b[] = texto.getBytes();

            // Se crea un objeto File correspondiente al fichero donde se va a
            // grabar el texto
            File fichSalida = new File( dir +File.separatorChar+ "textoe.txt" );
            // Se crea el canal de salida conectado a ese fichero
            FileOutputStream canalSalida = new FileOutputStream( fichSalida );
            // Se escribe el contenido del array de bytes en el fichero
            canalSalida.write( b );
            // Se cierra el canal
            canalSalida.close();

            // Porción de código encargada de recuperar del fichero el contenido
            // que se presenta en la zona correspondiente al campo de texto,
            // una vez que se ha grabado en fichero el array de bytes
            // Se crea un objeto File para referenciar al fichero del que se va
            // a recuperar el contenido
            File fichEntrada = new File( dir +File.separatorChar+ "textol.txt" );
            // Se crea el canal de entrada para leer el texto
            FileInputStream canalEntrada = new FileInputStream( fichEntrada );
            // Creamos un array de bytes para almacenar el contenido del fichero
            byte bt[] = new byte[(int)fichEntrada.length()];

```

```
// Se lee el fichero
int numBytes = canalEntrada.read( bt );
// Se convierte el array de bytes a cadena que se presenta en la
// ventana
String cadena = new String( bt );
etiqueta.setText( cadena );
// Se cierra el canal de comunicación con el fichero
canalEntrada.close();
} catch( IOException e ) {
    e.printStackTrace();
}

// Se presenta el texto recuperado del fichero en la ventana y se
// cambia el botón para poder realizar otro ciclo del proceso
panel.removeAll();
panel.add( "North", pulsado );
panel.add( "Center", etiqueta );
panel.add( "South", botonClick );
panel.validate();
panel.repaint();
}

// Si se trata del botón de reinicio, volvemos a presentar en la
// ventana los componentes iniciales
if( source == botonClick ) {
    panel.removeAll();
    panel.add( "North", texto );
    // Texto en blanco
    campoTexto.setText( "" );
    panel.add( "Center", campoTexto );
    panel.add( "South", boton );
    panel.validate();
    panel.repaint();
}
}

public static void main( String[] args ) {
    GeneraFile frame = new GeneraFile();
    frame.setTitle( "Sistemas Distribuidos, Archivos" );

    WindowListener wl = new WindowAdapter() {
        public void windowClosing( WindowEvent evt ) {
            System.exit( 0 );
        }
    };

    frame.addWindowListener( wl );
    frame.pack();
    frame.setVisible( true );
}
}
```

Autoevaluación

Implemente una aplicación cliente / servidor que determine si un parámetro ingresado es un archivo o un directorio. De ser directorio, deberá mostrar todos los archivos dentro de ese directorio.

Para recordar

- El paquete `java.io` proporciona un conjunto de clases para manejar inputs y outputs dentro de nuestras aplicaciones.
- Las clases `FileInputStream` y `FileOutputStream` definen flujos de entrada y salida de bytes que están conectados a archivos.



Java Input / Output Streams II

TEMA

Flujos de entrada y de salida

OBJETIVOS ESPECÍFICOS

- Implementar aplicaciones que utilicen objetos de las clases `DataInputStream` y `DataOutputStream`
- Implementar aplicaciones que utilicen objetos de las clases `InputStreamReader` e `OutputStreamWriter`

CONTENIDOS

- Input / Output de valores primitivos java
 - Filtros
 - Input / Output de valores primitivos
- Flujos de caracteres: readers y writers
 - Métodos `close` y `flush`

ACTIVIDADES

- Implementan funcionalidades de lectura y escritura de archivos en formato binario y texto.

1. INPUT / OUTPUT DE VALORES PRIMITIVOS JAVA

1.1 Filtros

Un filtro es un flujo de alto nivel que proporciona funcionalidad adicional sobre el flujo base al que se encuentre encadenado. Los datos provenientes del flujo base son manipulados en realidad por el filtro asociado.

1.2 Input / Output de valores primitivos Java

El paquete `java.io` contiene dos interfaces: `DataInput` y `DataOutput`. Estos flujos pueden ser implementados por una aplicación para leer y escribir representaciones binarias de valores primitivos java (`boolean`, `char`, `byte`, `short`, `int`, `Long`, `float`, `double`). Los métodos para escribir representaciones binarias de valores primitivos java son llamados `writeX()`, donde X es cualquier tipo de dato primitivo java.

Los métodos para leer representaciones binarias de valores primitivos java son de manera similar llamados `readX()`.

Los filtros `DataOutputStream` y `DataInputStream` implementan las interfaces `DataOutput` y `DataInput` respectivamente y pueden ser usados para escribir y leer representaciones binarias de valores primitivos java.

Los siguientes constructores pueden ser utilizados para configurar un filtro y leer o escribir valores primitivos desde un flujo: objetos.

`DataInputStream(InputStream in)`

`DataOutputStream(OutputStream out)`

Para manejar flujos de caracteres, java proporciona flujos especiales llamados **readers** y **writers**.

Actividad

1. Implemente las clases **EnviaArchivo** y **RecibeArchivo** mostradas a continuación.

EnviaArchivo

```
import java.io.*;
import java.net.*;
public class EnviaArchivo {
    String host="localhost";
    int puertofileserver = 5678;
    public EnviaArchivo(){

        File input = new File("c:/Luis/LuisGarcia.jpg");
        try {
            FileInputStream inputs=new FileInputStream(input);
            Socket socket = new Socket(host, puertofileserver);
            DataOutputStream salida=new
DataOutputStream(socket.getOutputStream());
            int leido;
            while((leido=inputs.read())!=-1){
                salida.write(leido);
            }
            socket.close();
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();

        } catch (UnknownHostException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    public static void main(String[] args){
        EnviaArchivo f = new EnviaArchivo();
    }
}
```

RecibeArchivo

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;

public class RecibeArchivo {

    Socket hostcliente;
    int puertofileserv = 5678;

    public RecibeArchivo(){

        try {

            ServerSocket ssocket = new ServerSocket(
puertofileserv);

            while(true){
                hostcliente=ssocket.accept();

                File output = new File("c:/LuisGarcia.jpg");

                FileOutputStream outputs=new
FileOutputStream(output);

                DataInputStream entrada=new
DataInputStream(hostcliente.getInputStream());
                int leido;
                while((leido=entrada.read())!=-1){
                    outputs.write(leido);
                }

                outputs.close();
                hostcliente.close();

            }

        }

    }

}
```

```
        } catch (FileNotFoundException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
  
        } catch (UnknownHostException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args){  
        RecibeArchivo f = new RecibeArchivo();  
    }  
}
```

2. FLUJOS DE CARACTERES: READERS Y WRITERS

Un programa java representa caracteres internamente bajo la codificación Unicode de 16 bits; sin embargo, la plataforma en la que dicho programa puede estar siendo ejecutado podría usar otra codificación para representar los caracteres externamente. Por ejemplo, la codificación ASCII es ampliamente usada para representar caracteres sobre muchas plataformas, a pesar de que éste es sólo un subconjunto pequeño del estandar Unicode.

Las clases abstractas Reader y Writer son las raíces principales dentro de la jerarquía de flujos que leen y escriben caracteres Unicode.

Un reader es un flujo de **entrada** de caracteres que lee una secuencia de caracteres Unicode.

Un writer es un flujo de **salida** de caracteres que escribe una secuencia de caracteres Unicode.

Los readers usan los siguientes métodos para leer caracteres Unicode:

- ✓ `int read() throws IOException;`
- ✓ `int read(char cbuf[]) throws IOException;`
- ✓ `int read(char cbuf[], int off, int len) throws IOException.`

Es importante destacar que el método `read` lee un valor `int` dentro del rango de 0 a 65535. El valor `-1` es retornado si se ha alcanzado el final del archivo.

Los writers usan los siguientes métodos para escribir caracteres Unicode:

- ✓ `void write(char[] cbuf) throws IOException;`
- ✓ `void write(String str) throws IOException;`
- ✓ `void write(char[] cbuf, int off, int len) throws IOException;`
- ✓ `void write(String str, int off, int len) throws IOException.`

Importante:

El método `write` toma como argumento un valor de tipo `int`, pero solo escribe los 16 bits menos significativos.

2.1 Métodos `close` y `flush`

`void close() throws IOException`

`void flush() throws IOException`

Al igual que los flujos de bytes, un flujo de caracteres debe ser cerrado para liberar recursos cuando ya no es necesario. Al cerrar un flujo de caracteres de salida, automáticamente, se hace un flush del flujo. También, podemos hacer un flush de un flujo de caracteres de manera manual.

Actividad

1. Implemente las clases **LeeTexto** y **LeeTextoReloaded** mostradas a continuación.

LeeTexto

```
/**
 * Lectura de un fichero de texto carácter a carácter. Incorpora un
 * contador para cuantificar el tiempo que se tarda en leer el archivo.
 */
import java.io.*;
import java.util.prefs.*;
public class LeeTexto{

    public static void main( String args[] ) {
        long intervalo = System.currentTimeMillis();
        FileReader fReader = null;
        int c;
        try {
            fReader = new FileReader("prueba.txt");
            while( (c = fReader.read()) != -1 ) {
            }
        } catch( Exception e ) {
            e.printStackTrace();
        } finally {
            try {
                if( fReader != null )
                    fReader.close();
            } catch( Exception e ) {
                e.printStackTrace();
            }
        }
        System.out.println( "Tiempo: "+
            (System.currentTimeMillis()-intervalo)+ " msgs." );
    }
}
```

LeeTextoReloaded

```
/**
 * Esta es una versión modificada de la clase LeeTexto.
 * Lectura de un fichero de texto a través de un buffer de lectura,
 * lo cual acelerará la lectura del fichero. También se incorpora un
 * contador para cuantificar el tiempo que se tarda en leer el
 * archivo.
 */

import java.io.*;
import java.util.prefs.*;

public class LeeTextoReloaded {
    public static void main( String args[] ) {
        long intervalo = System.currentTimeMillis();
        BufferedReader bReader = null;
        int c;

        try {
            bReader = new BufferedReader( new FileReader("prueba.txt") );
            while( (c = bReader.read()) != -1 ) {
            }
        } catch( Exception e ) {
            e.printStackTrace();
        } finally {
            try {
                if( bReader != null )
                    bReader.close();
            } catch( Exception e ) {
                e.printStackTrace();
            }
        }
        System.out.println( "Tiempo: "+
            (System.currentTimeMillis()-intervalo)+ " msgs." );
    }
}
```


Investigación

Investigue el funcionamiento de las clases `java.util.zip.ZipInputStream` y `java.util.zip.ZipOutputStream`. Implemente luego una aplicación que permita empaquetar varios archivos dentro de un archivo `.zip`

Autoevaluación

¿Se debe usar una clase `DataInputStream` para leer el contenido de un archivo de texto?

Para recordar

- Un filtro es un flujo de alto nivel que proporciona funcionalidad adicional sobre el flujo base al que se encuentre encadenado.
- Las clases abstractas `Reader` y `Writer` son las raíces principales dentro de la jerarquía de flujos que leen y escriben caracteres Unicode.



Serialización de Objetos

TEMA

Serialización de Objetos

OBJETIVOS ESPECÍFICOS

- Implementar aplicaciones que utilicen objetos de las clases `ObjectInputStream` y `ObjectOutputStream` para leer y escribir objetos

CONTENIDOS

- Serialización de Objetos
 - Definición
 - La clase `ObjectInputStream`
 - La clase `ObjectOutputStream`

ACTIVIDADES

- Implementan una aplicación que escribe y lee un objeto serializado

1. SERIALIZACIÓN DE OBJETOS

1.1 Definición

La serialización de objetos permite que un objeto sea transformado en una secuencia de bytes que posteriormente pueda ser recreada (deserializada) en el objeto original. Después de una deserialización, el objeto tiene el mismo estado que poseía en el momento de ser serializado.

Java proporciona esta facilidad a través de las interfaces

ObjectInput y ObjectOutput.

Estas interfaces permiten leer y escribir objetos desde y hacia flujos de entrada y salida respectivamente.

La clase **ObjectOutputStream** implementa la interface **ObjectOutput**. Esto significa que la clase **ObjectOutputStream** implementa métodos como bytes, texto y valores primitivos de java. De manera similar la clase **ObjectInputStream** implementa métodos para leer objetos como bytes, texto y valores primitivos java.

1.2 La clase ObjectOutputStream

La clase **ObjectOutputStream** puede escribir objetos sobre cualquier flujo que sea una subclase de **OutputStream**, por ejemplo, sobre un archivo o sobre una conexión de red (socket). Un objeto **ObjectOutputStream** debe ser encadenado a un **OutputStream** usando el siguiente constructor:

ObjectOutputStream(OutputStream out) throws IOException

Por ejemplo, para almacenar objetos en un archivo y proporcionarles un almacenamiento persistente, un **ObjectOutputStream** puede ser encadenado a un **FileOutputStream**:

```
FileOutputStream outputFile= new FileOutputStream("objfacilito.dat");  
ObjectOutputStream opSt=new ObjectOutputStream(outputFile);
```

Los objetos pueden ser escritos sobre el flujo, usando el método `writeObject()` de la clase `ObjectOutputStream`:

```
final void writeObject(Object obj) throws IOException
```

1.3 La clase `ObjectInputStream`

Un objeto `ObjectInputStream` es usado para restaurar (deserializar) objetos que han sido previamente serializados usando un `ObjectOutputStream`. Un `ObjectInputStream` debe ser encadenado a un `InputStream`, usando el siguiente constructor:

```
ObjectInputStream(InputStream in) throws  
IOException, StreamCorruptedException
```

Por ejemplo, para restaurar objetos de un archivo, un `ObjectInputStream` puede ser encadenado a un `FileInputStream`:

```
FileInputStream inputFile= new FileInputStream("objfacilito.dat");  
ObjectInputStream opSt=new ObjectInputStream(inputFile);
```

El método `readObject()` de la clase `ObjectInputStream` es usado para leer objetos desde un flujo:

```
final void readObject( ) throws IOException,  
    ClassNotFoundException, OptionalDataException
```

Actividad

1. Implemente las clases **Arbol** y **SeriaSeria** mostradas en el programa listado a continuación.

SeriaSeria

```
/**
 * Este programa declara una clase Arbol, que implementa un vector y es una
 * clase serializable, se puede almacenar en un stream.
 * En la ejecución se crea un objeto de este tipo que se graba en fichero
 * y se imprime en pantalla. Seguidamente, se recupera el objeto del fichero
 * y se vuelve a imprimir el contenido del objeto recuperado del fichero
 */
import java.io.*;
import java.util.Date;
import java.util.Vector;
import java.util.Enumeraion;

class Arbol implements Serializable {
    Vector<Arbol> hijos;
    Arbol padre;
    String nombre;

    // Constructor de la clase, que implementa un Vector de
    // cinco elementos
```



```
public Arbol( String s ) {
    hijos = new Vector<Arbol>( 5 );
    nombre = s;
}

// Método para incorporar elementos al Vector
public void addRama( Arbol n ) {
    hijos.addElement( n );
    n.padre = this;
}

// Este es el método que vuelca en contenido del vector
// a un formato legible en pantalla, imprimiendo el contenido
// de los elementos del Arbol
public String toString() {
    Enumeration enu = hijos.elements();
    StringBuffer buffer = new StringBuffer( 100 );

    buffer.append( "[" +nombre+ ": " );
    while( enu.hasMoreElements() ) {
        buffer.append( enu.nextElement().toString() );
    }
    buffer.append("] ");
    return( buffer.toString() );
}

public class SeriaSeria {
    public static void main(String[] args) {
        // En primer lugar construimos el árbol
        Arbol raiz = new Arbol("raiz" );
        raiz.addRama( new Arbol("izqda" ) );
        raiz.addRama( new Arbol("drcha" ) );

        // Lo imprimimos para ver lo que aparece
        System.out.println( "Arbol Original: \n" +raiz.toString() );

        try {
            // Ahora grabamos el árbol completo en un fichero
            FileOutputStream fOut = new FileOutputStream( "prueba.rmi" );
            ObjectOutputStream out = new ObjectOutputStream( fOut );
            out.writeObject( raiz );
            out.flush();
            out.close();
        }
    }
}
```

```
// Recuperamos el árbol del fichero
FileInputStream fln = new FileInputStream( "prueba.rmi" );
ObjectInputStream in = new ObjectInputStream( fln );
Arbol n = (Arbol)in.readObject();
in.close();
// Imprimimos el resultado de la lectura del fichero
System.out.println( "Arbol Leido: \n" +n.toString() );
} catch( Exception e ) {
    e.printStackTrace();
}
}
}
```

Autoevaluación

¿Cuál es la interface que debe implementar un objeto para poder ser serializado?

Para recordar

- La serialización de objetos permite que un objeto sea transformado en una secuencia de bytes que posteriormente pueda ser recreada (deserializada) en el objeto original.
- La clase ObjectOutputStream puede escribir objetos sobre cualquier flujo que sea una subclase de OutputStream, por ejemplo, sobre un archivo o sobre una conexión de red (socket).



RMI: Remote Method Invocation

TEMA

Invocación Remota de Métodos

OBJETIVOS ESPECÍFICOS

- Implementar aplicaciones que utilicen objetos Remotos

CONTENIDOS

- RMI: Remote Method Invocation
 - Definición
 - Arquitectura básica RMI
- Pasos para implementar una aplicación RMI

ACTIVIDADES

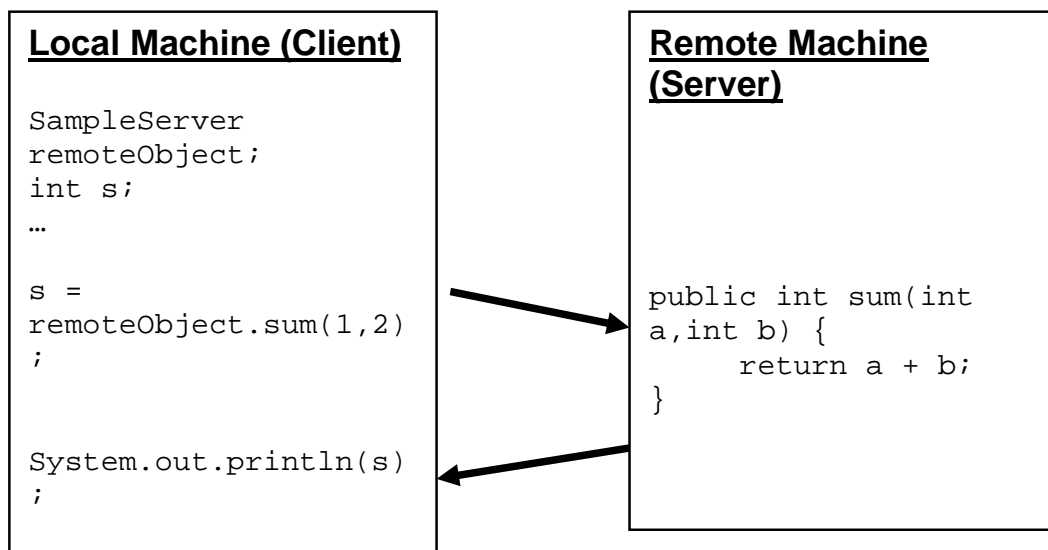
- Implementan una aplicación cliente que invoca métodos de una aplicación remota

1. RMI: REMOTE METHOD INVOCATION

1.1 Definición

En java, la invocación de métodos remotos (RMI), permite que un objeto que se ejecuta bajo el control de una máquina virtual java pueda invocar métodos de un objeto que se encuentra en ejecución bajo el control de una máquina virtual java diferente. Estas dos máquinas virtuales pueden estar ejecutándose como dos procesos independientes en el mismo ordenador o en ordenadores distintos conectados a través de una red TCP/IP.

Remote Method Invocation



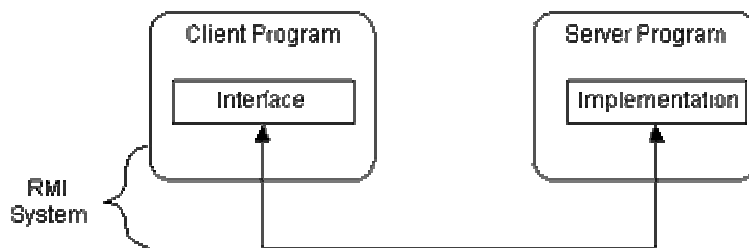
Por lo tanto, dado que internet es una red TCP/IP, una máquina cliente en cualquier parte del mundo es capaz de invocar métodos de un objeto que se encuentre en ejecución sobre un servidor en cualquier otra parte del mundo.

La máquina que contiene objetos cuyos métodos se pueden invocar se llama **servidor**.

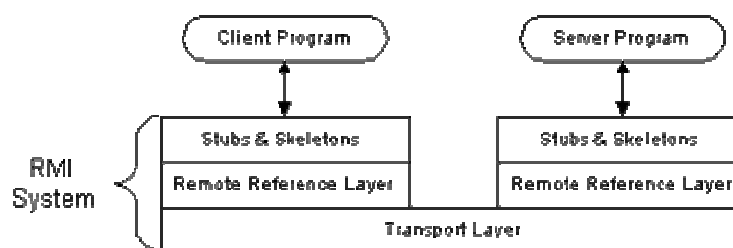
La máquina que invoca métodos sobre el objeto remoto se llama **cliente**.

Cuando se invocan métodos remotos sobre objetos remotos, el cliente puede pasar objetos como parámetros y los métodos de los objetos remotos pueden devolver objetos. Esto es posible gracias a la capacidad de **serialización** de objetos en java.

1.2 Arquitectura básica RMI



El programa cliente interactúa con el servidor a través de una interfaz en la cual se exponen los métodos remotos que pueden ser invocados.



El **stub** es un componente que:

- ✓ tiene información de donde está el objeto remoto;
- ✓ funciona como un Proxy;
- ✓ intercepta las invocaciones de la referencia remota (en el cliente);
- ✓ delega la invocación al objeto remoto;

- ✓ recibe la respuesta del objeto remoto y se la entrega al cliente.

Importante: A partir de la version 1.5 del jdk, ya no es necesario generar explícitamente el stub.

2. PASOS PARA IMPLEMENTAR UNA APLICACIÓN RMI

Los pasos típicos para implementar una aplicación RMI son los siguientes:

- *implementar los siguientes componentes:*
 - ✓ la interfaz remota
 - ✓ la clase remota
 - ✓ clase Registradora (registra el objeto remoto en el Registro de Objetos)
 - ✓ el cliente
- *compilar los componentes;*
- *generar el stub;*
- *arrancar el Registro RMI;*
- *ejecutar la clase registradora;*
- *ejecutar el cliente.*

2.1 La interface Remota

Declara métodos de negocio.

```
public interface Calculadora extends java.rmi.Remote {  
    public long suma(long a, long b) throws java.rmi.RemoteException;  
    public long resta(long a, long b) throws java.rmi.RemoteException;  
}
```

2.2 La clase Remota

Implementa la interface Remota.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class CalculadoraImpl extends UnicastRemoteObject
implements Calculadora {
    public CalculadoraImpl() throws RemoteException {
        super();
    }
    public long suma(long a, long b) throws RemoteException {
        return a + b;
    }
    public long resta(long a, long b) throws RemoteException {
        return a - b;
    }
}
```

2.3 La clase Registradora o Servidor

- ✓ Crea un objeto remoto
- ✓ Exporta el objeto
- ✓ Registra la referencia del objeto remoto en el registro de objetos

```
import java.rmi.Naming;
public class CalculadoraServer {
    public static void main(String[] args) {
        try {
            Calculadora c = new CalculadoraImpl();
            Naming.rebind("rmi://localhost:1099/CalculatorService", c);
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
}
```


2.4 La clase Cliente

- ✓ Busca el objeto remoto en el registro de objetos.
- ✓ Invoca los métodos de negocio del objeto referenciado.

```
import java.rmi.Naming;
public class CalculadoraClient {
    public static void main(String[] args) {
        try {
            Calculadora c = (Calculadora)Naming.lookup
"rmi://remotehost:1099/CalculatorService");
            System.out.println(c.suma(4,3));
            System.out.println(c.resta(4,5));
        } catch (Exception e){
            System.out.println("Error:" + e);
        }
    }
}
```

2.5 Compilar y generar el stub

Podemos compilar las clases utilizando el comando `javac`. En el ejemplo, la sentencia sería la siguiente:

```
javac -d bin -cp bin src\rmi\calculadora\*.java
```

La generación del stub la hacemos a través del comando `rmic`. Para el ejemplo, la sentencia sería la siguiente:

```
rmic -d bin -classpath bin rmi.calculadora.CalculadoraImpl
```

Una vez aplicado el comando **rmic**, se obtendría el siguiente archivo compilado:

CalculadoraImpl_Stub.class

2.6 Iniciar el registro RMI

Se inicia el registro ejecutando el comando `rmiregistry`. Se sugiere ejecutar el comando en donde se encuentran los componentes compilados.

Sintaxis:

`rmiregistry` <opciones> <puerto>

1099 es el puerto por defecto

2.7 Ejecutar el Servidor y el cliente

Ejecutamos el servidor de la siguiente manera:

```
java rmi.calculadora.CalculadoraServer
```

Ejecutamos el cliente de la siguiente manera:

```
java rmi.calculadora.CalculadoraClient
```

Actividad

1. Implemente la aplicación HolaMundoRMI. Para ello, contará con la siguiente estructura de componentes:

```
src\  
  hello\  
    HelloInterface.java  
    Hello.java  
  server\  
    HelloServer.java  
  client\  
    HelloClient.java
```

2. Implemente los componentes mostradas a continuación:

HelloInterface

```
package hello;  
import java.rmi.*;  
public interface HelloInterface extends Remote {  
    public String say() throws RemoteException;  
}
```

Hello

```
package hello;
import java.rmi.*;
import java.rmi.server.*;
public class Hello extends UnicastRemoteObject implements
HelloInterface {
    private String message;
    public Hello (String msg) throws RemoteException {
        message = msg;
    }
    public String say() throws RemoteException {
        return message;
    }
}
```

HelloServer

```
package server;
import java.rmi.*;
import hello.*;
public class HelloServer{
    public static void main (String[] argv) {
        try {
            Naming.rebind ("Hello", new Hello ("Hello, world!"));
            System.out.println ("Hello Server is ready.");
        } catch (Exception e) {
            System.out.println ("Error ocurrido en registrar: " + e);
        }
    }
}
```

HelloClient

```
package client;
import java.rmi.*;
import hello.HelloInterface;
public class HelloClient{
    public static void main (String[] argv) {
        try {
            HelloInterface hello =
                (HelloInterface) Naming.lookup ("//localhost:1099/Hello");
            System.out.println (hello.say());
            System.out.println ("Bye");
        } catch (Exception e) {
            System.out.println ("HelloClient exception" + e);
        }
    }
}
```

3. Compilar las clases:

```
javac -d classes -cp classes src\hello\*.java
```

4. Generar stub para la interfaz remota usado el commando rmic:

```
rmic -d classes -classpath classes hello.Hello
```

Debería haber generado un nuevo archivo en la carpeta classes, ¿Dónde se encuentra exactamente? ¿Qué nombre tiene?

5. Compilar la clase servidor:

```
javac -d classes -cp classes src\server\*.java
```

6. Compilar la clase cliente:

```
javac -d classes -cp classes src\client\*.java
```

7. Situarse en la carpeta classes, e iniciar el registro de objetos:

```
start rmiregistry
```

8. Ejecutar el servidor para registrar el objeto remoto:

```
java server.HelloServer
```

9. Ejecutar el cliente:

```
java client.HelloClient
```

Autoevaluación

Utilizando RMI, implemente aplicación de “consulta de productos”, la cual recibirá la descripción del producto y retornará la relación de productos que cumplen con la descripción ingresada. Usted deberá definir la estructura de la tabla y los registros de prueba respectivos.

Para recordar

- En java, la invocación de métodos remotos (RMI), permite que un objeto que se ejecuta bajo el control de una máquina virtual java pueda invocar métodos de un objeto que se encuentra en ejecución bajo el control de una máquina virtual java diferente.
- Cuando se invocan métodos remotos sobre objetos remotos, el cliente puede pasar objetos como parámetros y los métodos de los objetos remotos pueden devolver objetos.



JAX-WS: Java Web Services

TEMA

Java Web Services

OBJETIVOS ESPECÍFICOS

- Implementar aplicaciones que utilicen Servicios web usando la API JAX-WS

CONTENIDOS

- JAX-WS: Web Services
 - Definición
 - Web Services: Cliente - Servidor

ACTIVIDADES

- Implementan una aplicación que expone un servicio web

1. JAX-WS: WEB SERVICES

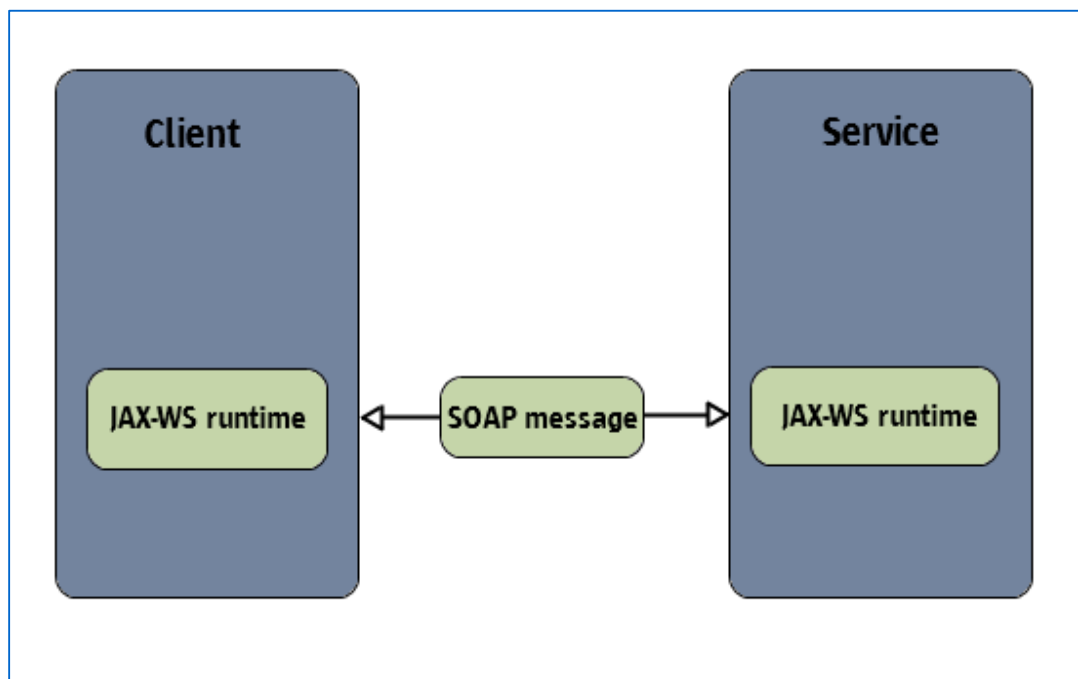
1.1 Definición

JAX-WS (Java Api for XML WebServices) es una tecnología desarrollada a partir de Java Enterprise Edition 5 (JEE5), que permite construir servicios web y clientes que se comunican utilizando xml.

En jax-ws una invocación a un servicio web es representada por un protocolo basado en xml: SOAP. La especificación SOAP define la estructura de la trama, las reglas de codificación y las convenciones para representar las invocaciones a los servicios web, y las respuestas respectivas.

Todas estas llamadas y respuestas son transmitidas como mensajes SOAP (archivos XML) sobre http. Aunque los mensajes SOAP son complejos, la API jax-ws oculta esta complejidad.

Comunicación entre un Servicio Web y un Cliente



1. Web Services: Cliente - Servidor

Lado servidor

- ✓ El desarrollador especifica las operaciones de un servicio web definiendo los métodos en una interface.
- ✓ El desarrollador codifica una o más clases que implementan dichos métodos.

Lado cliente

- ✓ El cliente crea un proxy (un objeto local representando el servicio) y simplemente invoca los métodos sobre el proxy.
- ✓ El desarrollador codifica una o más clases que implementan dichos métodos.

Importante: En jax-ws el desarrollador no genera ni interpreta mensajes SOAP. Es el entorno de ejecución jax-ws el que interpreta las llamadas y respuestas "de" y "a" mensajes SOAP.

JAX-WS no es restrictivo: un cliente puede acceder a un servicio web que no esté corriendo sobre la plataforma java y viceversa. Esta flexibilidad es posible gracias a que JAX-WS utiliza tecnologías definidas por el World Wide Web Consortium (W3C) :

- ✓ Http
- ✓ Soap
- ✓ Web Services Description Language (WSDL)

WSDL especifica un formato XML para describir la interface pública de un servicio web.

Actividad

1. Implemente el servicio Web Hola mostrado a continuación:

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Hola {

    @WebMethod
    public String saluda(String cliente){
        return "Hola "+cliente+" bienvenido a jax-ws :)";
    }

    @WebMethod
    public String suma(int a,int b){
        return "La suma es : "+(a+b);
    }

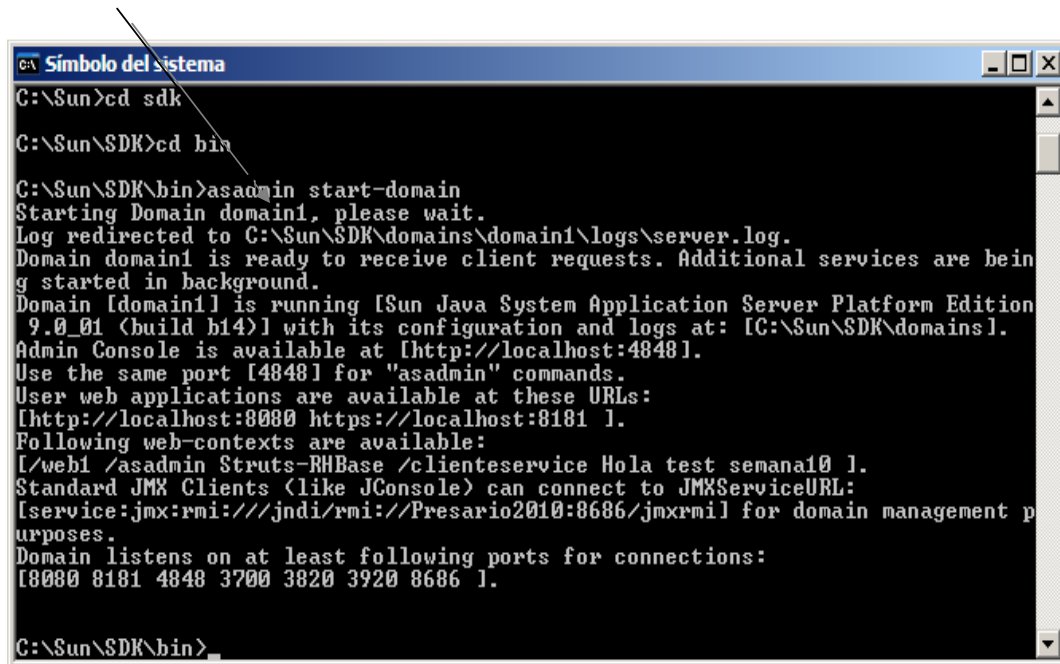
}
```

2. Copie el archivo compilado dentro de la carpeta autodeploy del servidor Sun Java System Application Server Platform Edition 9.0

<Directorio de Instalación>\Sun\SDK\domains\domain1\autodeploy

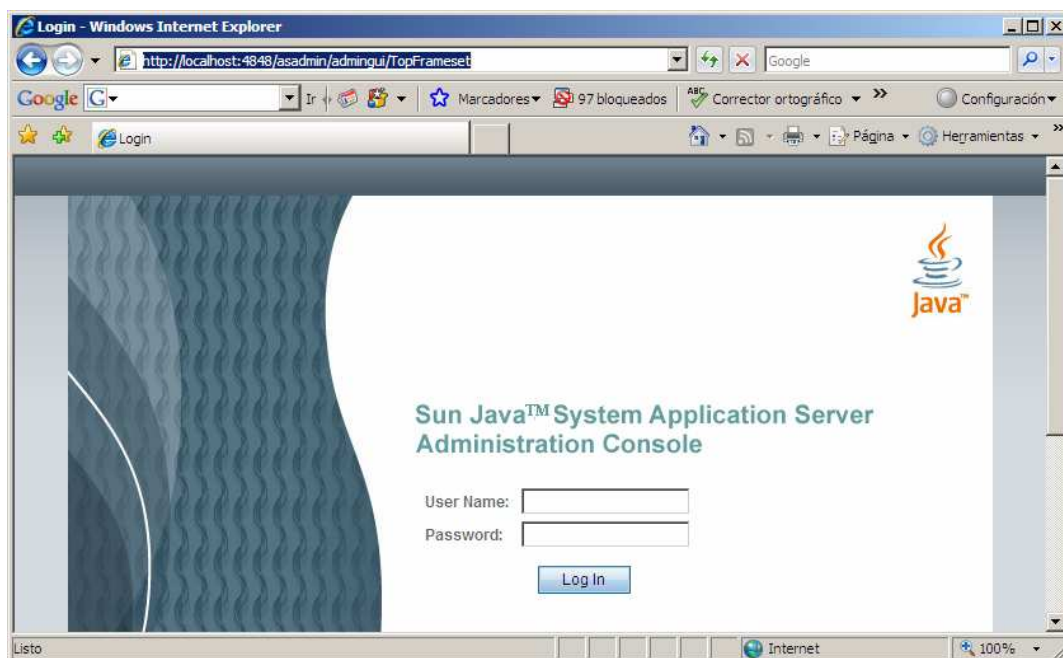
3. Inicie el servidor de aplicaciones ejecutando el comando:

asadmin start-domain

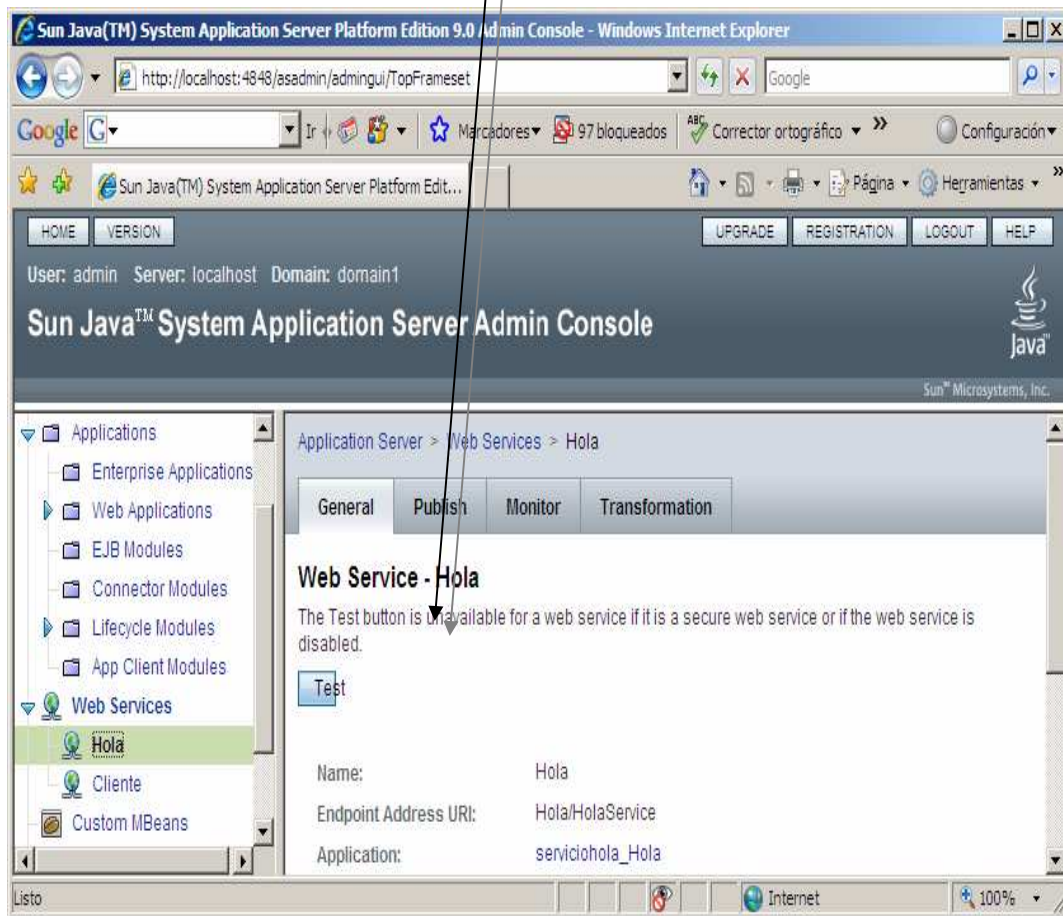


```
C:\Sun>cd sdk
C:\Sun\SDK>cd bin
C:\Sun\SDK\bin>asadmin start-domain
Starting Domain domain1, please wait.
Log redirected to C:\Sun\SDK\domains\domain1\logs\server.log.
Domain domain1 is ready to receive client requests. Additional services are being started in background.
Domain [domain1] is running [Sun Java System Application Server Platform Edition 9.0_01 (build b14)] with its configuration and logs at: [C:\Sun\SDK\domains].
Admin Console is available at [http://localhost:4848].
Use the same port [4848] for "asadmin" commands.
User web applications are available at these URLs:
[http://localhost:8080 https://localhost:8181 ].
Following web-contexts are available:
[/web1 /asadmin Struts-RHBase /clienteservice Hola test semana10 ].
Standard JMX Clients (like JConsole) can connect to JMXServiceURL:
[service:jmx:rmi:///jndi/rmi://Presario2010:8686/jmxrmi for domain management purposes.
Domain listens on at least following ports for connections:
[8080 8181 4848 3700 3820 3920 8686 ].
C:\Sun\SDK\bin>
```

4. Ingrese a la consola administrativa del servidor de aplicaciones:



5. Pruebe el servicio web seleccionando el botón test.



Actividad

1. Implemente y pruebe el correcto funcionamiento del servicio Web Cliente mostrado a continuación:

Cliente

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class Cliente {

    @WebMethod
    public BeanCliente buscaPorNombre(String nombre){
        BeanCliente cliente = new BeanCliente();
        if(nombre.equals("defa")){
            cliente.setNombre(nombre);
            cliente.setEdad(25);
            cliente.setDireccion("La direccion de
.."+nombre);
        }

        return cliente;
    }
}
```

BeanCliente

```
import java.io.Serializable;

public class BeanCliente implements Serializable {

    String nombre;
    int edad;
    String direccion;
    public String getDireccion() {
        return direccion;
    }
    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Actividad

Modifique el ejercicio anterior e implemente la funcionalidad de búsqueda utilizando acceso a una base de datos.

Autoevaluación

¿Se puede acceder a un servicio web creado en Java desde otro servicio web creado en .Net?

¿Cuáles son las principales ventajas del uso de jax-ws para la creación de servicios web en Java?

Para recordar

- En jax-ws una invocación a un servicio web es representada por un protocolo basado en xml: SOAP.
- En jax-ws el desarrollador no genera ni interpreta mensajes SOAP. Es el entorno de ejecución jax-ws el que interpreta las llamadas y respuestas “de” y “a” mensajes SOAP.



JMS: Java Message Service

OBJETIVOS ESPECÍFICOS

- Identificar las principales características de un Messaging Service y la API Java JMS

CONTENIDO

1. Introducción a la Mensajería
2. La API JMS
3. Dominios de Mensajes JMS
4. Modelo de Objetos JMS

ACTIVIDAD

- Implementan una aplicación que cargue y consulte una cola JMS

1. INTRODUCCIÓN A LA MENSAJERÍA

Un servicio de mensajería es aquel que proporciona comunicación entre aplicaciones o entre componentes de software. Cualquier aplicación o componente de software que use un servicio de mensajería es llamado un “messaging client”. En un servicio típico de mensajería, un messaging client puede enviar y recibir mensajes.

Un chat es un claro ejemplo de aplicaciones de mensajería. Normalmente el servidor de chat proporciona dos métodos de comunicación para sus usuarios. Estos son los siguientes:

- Un usuario puede enviar un mensaje que es replicado a todos los usuarios del chat.

- Un usuario puede enviar un mensaje privado a otro usuario. Este mensaje no será visible para ningún otro usuario.

JMS proporciona justamente dos servicios similares a los mencionados. En realidad, un messaging service es más sofisticado que una aplicación de chat. Para poder intercambiar mensajes en un chat, tanto el emisor como el receptor deben estar conectados. En un messaging service el emisor puede enviar un mensaje sin que el receptor esté conectado al mismo tiempo. El emisor envía un mensaje a un destino y el receptor lo recogerá de dicho destino.

JMS también desacopla clientes, en el sentido de que el emisor no necesita saber nada acerca del receptor y viceversa.

2. LA API JMS

Sun y otras compañías se unieron para crear la API JMS. De esta manera, se posibilitaba que las aplicaciones java puedan crear, enviar, recibir y leer mensajes, así como comunicarse también con otras implementaciones de mensajería. La versión original de JMS fue liberada en agosto de 1998 y se creó básicamente para permitir que las aplicaciones java se conecten a sistemas existentes, tales como IBM's MQ Series.

Las aplicaciones Java que usan JMS son llamadas clientes JMS y el sistema de mensajería, que administra el enrutamiento y entrega de los mensajes, es llamado el proveedor JMS (JMS provider). Un cliente JMS que envía un mensaje es llamado un productor y un cliente JMS que recibe un mensaje es llamado un consumidor. Un mismo cliente JMS puede ser a la vez productor y consumidor. Una aplicación JMS consiste de muchos clientes JMS y uno o más proveedores JMS.

Son dos las características básicas inherentes a JMS:

Asíncrono: un cliente JMS no tiene que solicitar mensajes para poder recibirlos. El proveedor JMS entrega los mensajes al cliente en el orden en que estos van llegando.

Seguro: un sistema JMS puede asegurarse de que un mensaje sea entregado sólo una vez.

3. DOMINIOS DE MENSAJES JMS

Los dominios de mensajes no son más que modelos de mensajería. La API JMS nos proporciona los siguientes dos dominios de mensajes:

Publish/Subscribe (pub/sub)
Point-to-Point (PTP)

3.1 Publish/Subscribe

En este modelo, el cliente envía un mensaje a un tópico específico. El mensaje es recibido por cualquier cliente interesado que está inscrito a ese tópico. Este es un modelo de uno - a muchos, donde el emisor es llamado el publisher y el receptor es llamado el subscriber. El hecho de enviar un mensaje se denomina publicar.

Los tópicos envían los mensajes a todos los clientes inscritos y no se quedan con ninguno. Un cliente sólo puede consumir mensajes después de haberse inscrito al tópico. El cliente inscrito debe continuar activo para poder seguir consumiendo mensajes. Esta restricción es, en realidad, bastante flexible con JMS, ya que habilita a un cliente a poder crear inscripciones durareras.

3.2 Point to Point

Bajo este modelo un cliente envía un mensaje que es recibido sólo por un cliente. El emisor envía un mensaje a una cola y el receptor extrae el mensaje de dicha cola en el momento en que lo estime conveniente, es decir, cuando se conecte al sistema. Una cola retiene todos los mensajes que recibe hasta que el mensaje es consumido o hasta que el mensaje expire.

4. MODELO DE OBJETOS JMS

Los objetos más importantes en el modelo de objetos JMS son representados por las siguientes interfaces en el paquete javax.jms:

ConnectionFactory
Destination
Connection
Session
MessageProducer
MessageConsumer
Message

4.1 ConnectionFactory

Un objeto ConnectionFactory es usado para crear una conexión con un proveedor JMS. Este objeto soporta concurrencia y contiene parámetros de configuración de la conexión que han sido previamente establecidos por un administrador.

La interface ConnectionFactory tiene dos subinterfaces directas:
TopicConnectionFactory y
QueueConnectionFactory

La interface QueueConnectionFactory es utilizada para crear una conexión punto a punto con un proveedor JMS. Por otro lado, la interface TopicConnectionFactory es usada para crear una conexión pub/sub con un proveedor JMS. A continuación, tenemos un ejemplo para obtener una conexión Point to Point:

```
Context context = new InitialContext();  
QueueConnectionFactory queue=  
    (QueueConnectionFactory) context.lookup("QueueConnectionFactory");
```

4.2 Destination

Un objeto Destination encapsula la dirección de un proveedor específico. Estos objetos soportan accesos concurrentes y están representados en java, dentro del paquete javax.jms por la interface Destination.

Dependiendo del dominio en el que nos encontremos, un Destination podrá ser una cola o un tópico.

4.3 Connection

Un objeto Connection representa una conexión activa de un cliente JMS hacia un proveedor. Típicamente es un socket TCP/IP entre un cliente JMS y su proveedor.

En JMS, existen dos tipos de objetos Connection representados por las siguientes interfaces:

QueueConnection y TopicConnection.

Cuando una conexión es creada por primera vez, esta se encuentra en modo detenido (stopped mode), lo que significa que ningún mensaje ha sido entregado.

Es el cliente el que inicia la conexión invocando al método start. De esta manera, habilitamos a nuestra aplicación JMS para poder consumir mensajes. Debemos recordar que es una buena práctica siempre liberar recursos, por lo que al terminar de usar la conexión es una buena práctica invocar al método close del Connection.

4.4 Session

Un objeto Session es un contexto único creado para consumir o producir mensajes. Las interfaces típicas a ser usadas son las siguientes:

Topic Session y
Queue Session.

4.5 MessageProducer

Es un objeto usado por un cliente JMS para enviar mensajes a un destino en particular. La interface MessageProducer tiene dos subinterfaces directas:

QueueSender
TopicPublisher

Un objeto MessageProducer es creado cuando se pasa un objeto Destination al método message-producer del correspondiente objeto Session. A continuación, se muestra un ejemplo de creación del objeto QueueSender.

```
Context context = new InitialContext();
QueueConnectionFactory
queue=(QueueConnectionFactory)context.lookup("QueueConnection
Factory");
Queue queue=context.lookup(queueName);
QueueConnection
queueconnection=queueConnectionFactory.createQueueConnection
();
QueueSession queueSession =
    queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
QueueSender queueSender =
queueSession.createSender(queue);
```

4.6 MessageConsumer

Es un objeto usado por un cliente JMS para recibir mensajes de un destino. La interface MessageConsumer tiene dos subinterfaces directas:

QueueReceiver
TopicSubscriber

Un objeto MessageConsumer, de manera equivalente a un MessageProducer, es creado al pasar un objeto Destination al método message-consumer del correspondiente objeto Session.

4.7 Message

Existen muchos tipos de mensajes en JMS:

TextMessage. Es un mensaje que contiene parejas nombre/valor. Cada nombre es un objeto String y cada valor en tipo primitivo de Java. una cadena.

MapMessage. Es un mensaje conteniendo una cadena.

BytesMessage. Es un flujo de bytes no interpretados.

StreamMessage. Es un flujo de tipos primitivos Java.

ObjectMessage. Es un mensaje que contiene un objeto serializable.

Todo mensaje JMS está compuesto de las siguientes partes:

Header: todos los mensajes soportan el mismo conjunto de campos de cabecera. Los valores de estos campos son usados por clientes y proveedores para identificar y direccionar mensajes.

Properties: cada mensaje soporta propiedades definidas por la aplicación.

Body: la API JMS define varios tipos de cuerpo de mensajes, los cuales determinan el tipo de mensaje.

Para enviar un mensaje, primero debemos crear un objeto Message llamando o invocando a uno de los métodos create message de la interface Session. Podemos crear diferentes tipos de mensajes invocando a diferentes métodos de la interface Session, tales como:

createTextMessage,
createObjectMessage,
createTextMessage, etc.

Para recibir un mensaje, usamos el método receive de la interface MessageConsumer.

```
Message message=queueReceiver.receive();
```


Actividad

Implemente dos aplicaciones JMS. La primera debe enviar un mensaje a una cola denominada MyQueue. La segunda aplicación recibirá el mensaje y lo mostrará en consola.

MessageSender

```
import javax.jms.*;
import javax.naming.*;

public class MessageSender {

    public static void main(String[] args) {
        QueueConnection queueConnection = null;

        try {
            Context context = new InitialContext();
            QueueConnectionFactory queueConnectionFactory =
                (QueueConnectionFactory) context.lookup("QueueConnectionFactory");
            String queueName = "MyQueue";
            Queue queue = (Queue) context.lookup(queueName);
            queueConnection =
                queueConnectionFactory.createQueueConnection();
            QueueSession queueSession =
                queueConnection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
            QueueSender queueSender = queueSession.createSender(queue);
            TextMessage message = queueSession.createTextMessage();
            message.setText("This is a TextMessage");
            queueSender.send(message);
            System.out.println("Message sent.");
        }
        catch (NamingException e) {
            System.out.println("Naming Exception");
        }
    }
}
```

```

catch (JMSEException e) {
    System.out.println("JMS Exception");
}
finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        }
        catch (JMSEException e) {}
    }
}
}
}
}

```

MessageReceiver

```

import javax.jms.*;
import javax.naming.*;
public class MessageReceiver {
    public static void main(String[] args) {
        QueueConnection queueConnection = null;
        try {
            Context context = new InitialContext();
            QueueConnectionFactory queueConnectionFactory =
                (QueueConnectionFactory) context.lookup("QueueConnectionFactory");
            String queueName = "MyQueue";
            Queue queue = (Queue) context.lookup(queueName);
            queueConnection =
                queueConnectionFactory.createQueueConnection();
            QueueSession queueSession =
                queueConnection.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
            QueueReceiver queueReceiver = queueSession.createReceiver(queue);
            queueConnection.start();
            Message message = queueReceiver.receive(1);
            if (message != null) {
                if (message instanceof TextMessage) {

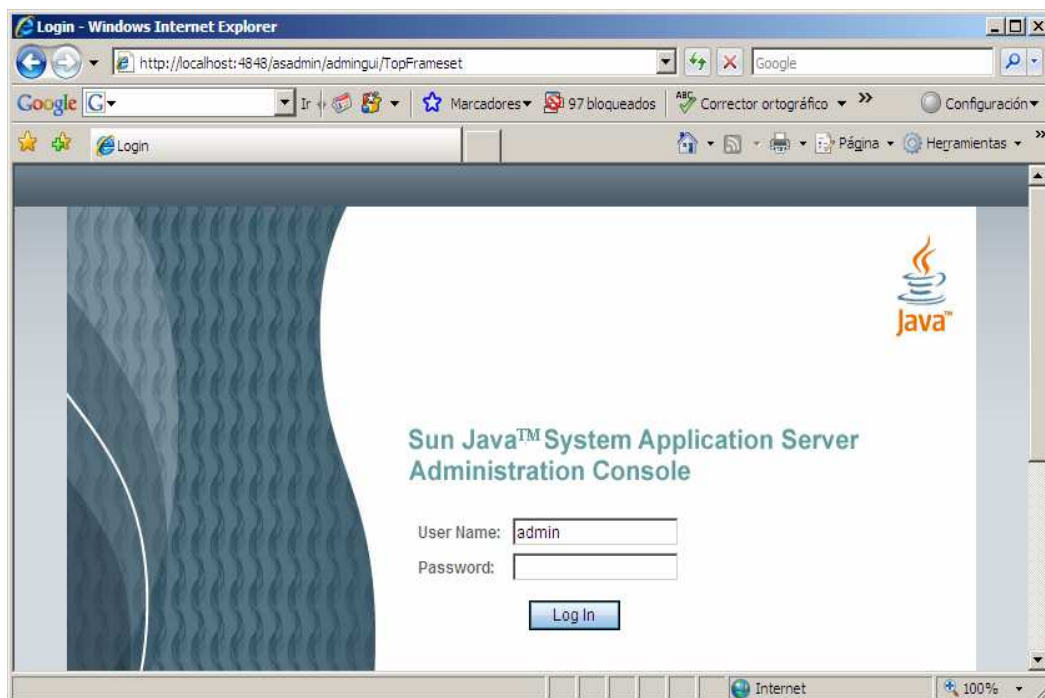
```

```
TextMessage textMessage = (TextMessage) message;
    System.out.println(textMessage.getText());
}
}
}
catch (NamingException e) {
    System.out.println("Naming Exception");
}
catch (JMSEException e) {
    System.out.println("JMS Exception");
}
finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        }
        catch (JMSEException e) {}
    }
}
}
```

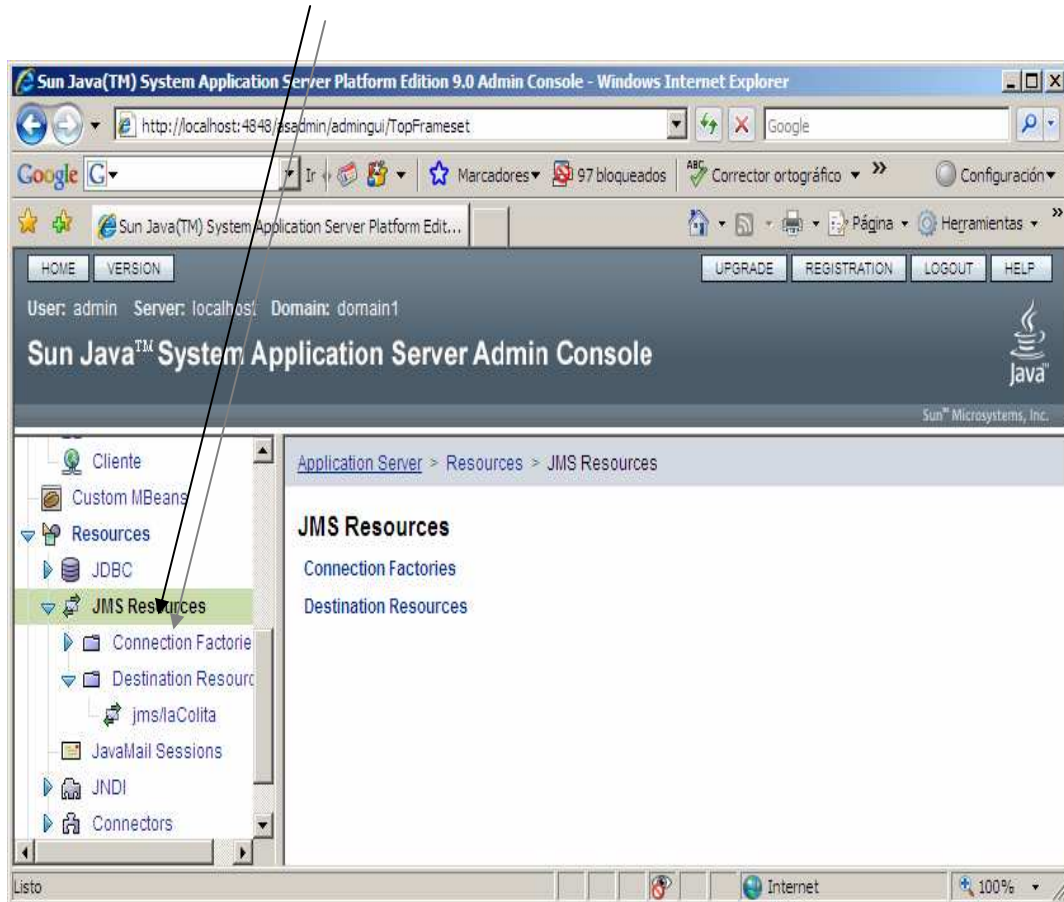
Actividad

Implemente un aplicación que permita consultar los mensajes de una cola. Para ello deberá de seguir los siguientes pasos:

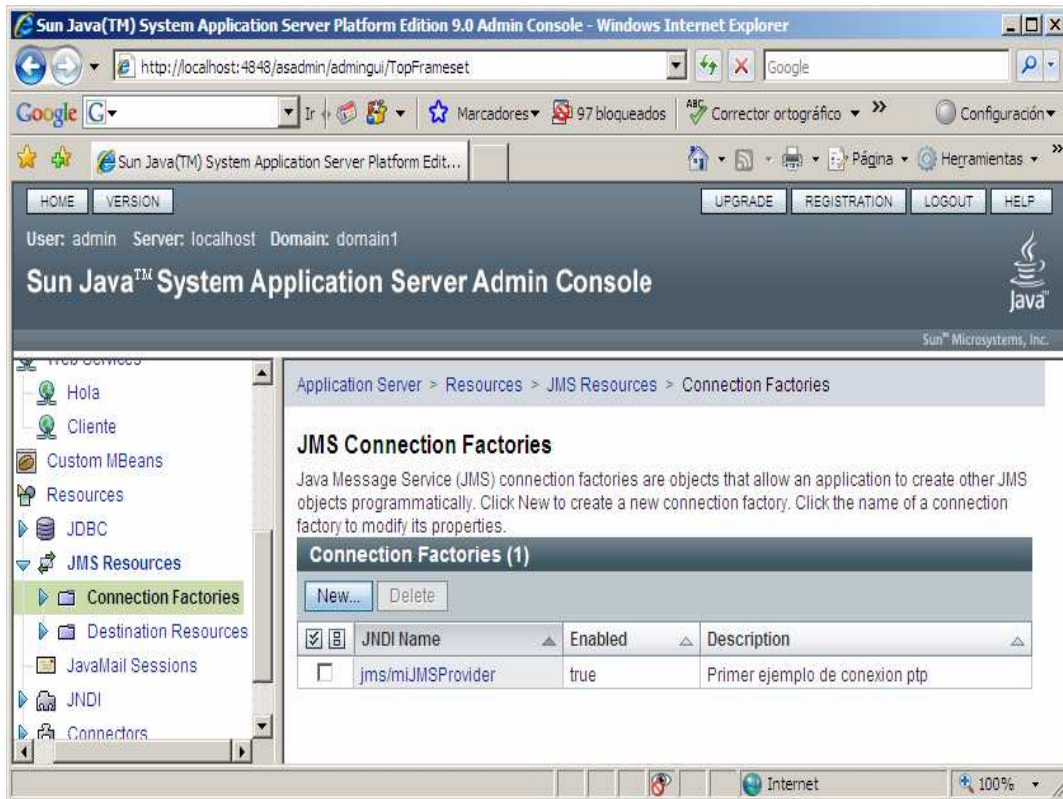
1. Ingrese a la consola administrativa del servidor de aplicaciones:



2. Seleccione la entrada JMS Resources:

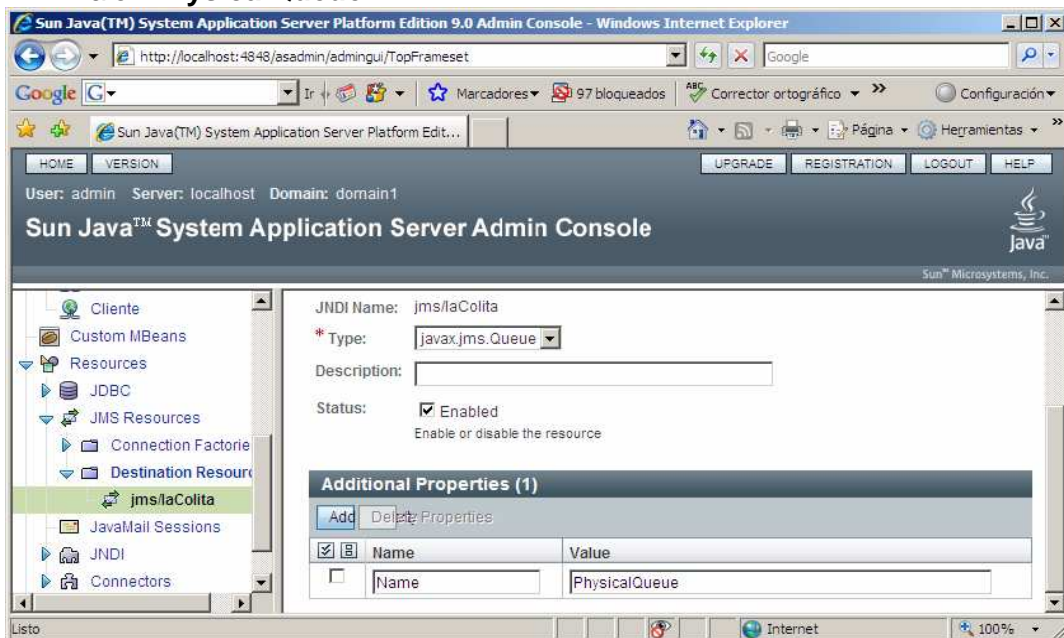


3. Cree una fábrica de conexiones a Colas:



4. Cree una cola de Destino:

Al crear la cola de Destino, debe crear también el atributo **Name** con el valor **PhysicalQueue**



5. Cree las clases EnviaMensaje y ConsultaCola:

EnviaMensaje

```
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.*;

import javax.jms.*;
import javax.naming.*;

public class EnviaMensaje extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req,
                          HttpServletResponse res)
        throws ServletException, IOException {

        // --- recuperando parametro con el mensaje
        String msg=req.getParameter("msg");

        // --- referenciando objetos en el servidor
        try {
            InitialContext ctx=new InitialContext();
            QueueConnectionFactory fabrica=
                (QueueConnectionFactory)ctx.lookup("jms/miJMSPProvider");

            Queue cola=
                (Queue)ctx.lookup("jms/laColita");

            // -- creando una conexion
            QueueConnection cn=
                fabrica.createQueueConnection();
```

```
// --- creamos la sesion
    QueueSession sesion=

    cn.createQueueSession(false,QueueSession.AUTO_ACKNOWLEDG
E);

        // -- enviando mensaje

        MessageProducer producer=
            sesion.createProducer(cola);

        TextMessage mensaje=
            sesion.createTextMessage();

        mensaje.setText(msg);

        producer.send(mensaje);

        System.out.println("exito si se pudo!:"+msg);


    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}

}
```


ConsultaCola

```
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.*;

import javax.jms.*;
import javax.naming.*;
import java.util.Enumuration;
import java.util.Date;

public class ConsultaCola extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req,
                          HttpServletResponse res)
        throws ServletException, IOException {

        // --- referenciando objetos en el servidor
        try {
            InitialContext ctx=new InitialContext();
            QueueConnectionFactory fabrica=
                (QueueConnectionFactory)ctx.lookup("jms/miJMSProvider");

            Queue cola=
                (Queue)ctx.lookup("jms/laColita");

            // -- creando una conexion
            QueueConnection cn=
                fabrica.createQueueConnection();

            // --- creamos la sesion
            QueueSession sesion=
                cn.createQueueSession(false,QueueSession.AUTO_ACKNOWLEDG
E);
```

```
// -- consultando cola
        QueueBrowser browser=
            sesion.createBrowser(cola);

        Enumeration enu=
            browser.getEnumeration();

        System.out.println("mensajes a la fecha: "+new Date());
        while(enu.hasMoreElements()){
            TextMessage m=
                (TextMessage)enu.nextElement();
            System.out.println(m.getText());

        }

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
```

6. Ejecute las clases **EnviaMensaje** (cargue al menos 5 mensajes) y luego consulte la cola ejecutando la clase **ConsultaCola**.

Actividad

Modifique la aplicación previamente implementada y cree una aplicación web en la que usando JSPs:

- ✓ Se ingresen los mensajes a la cola
- ✓ Se muestre la consulta de la cola

Autoevaluación

¿Cuál es la principal diferencia entre el uso de un servicio de mensajería basado en Colas y otro basado en Tópicos?

¿Qué modelo de mensajería debemos utilizar si el requerimiento de un cliente es implementar una lista de interés con suscriptores?

Para recordar

- Un servicio de mensajería es aquel que proporciona comunicación entre aplicaciones o entre componentes de software.
- En un messaging service el emisor puede enviar un mensaje sin que el receptor esté conectado al mismo tiempo.