# Ejercicios de Introducción a JMS

## Índice

## 1. Dominio JMS

Para que todas las factorias sean independientes del resto de módulos, vamos a crear un nuevo dominio. Para ello, crearemos desde Netbeans un nuevo servidor al que denominaremos **'GlassFish Server 3 - JMS'**. Al crear el servidor crearemos un nuevo dominio personal que llamaremos `domain-jms` el cual situaremos dentro de la carpeta de dominios de Glassfish, es decir, en `/home/especialista/glassfish-v2.1/domains/domain-jms`.

Todos los ejercicios de este módulo utilizarán este dominio.

## 2. Primeros Ejemplos (1p)

Se plantea como ejercicio la creación de los ejemplos PTP y Pub/Sub vistos en los apuntes.

Antes de nada habrá que crear en *Glassfish* tanto la factoría de conexiones como los recursos de cola y tópico vistos en los apuntes.

En cuanto a los clientes, crearemos un proyecto de aplicación cliente para cada elemento de cada dominio. Así pues, tendremos los proyectos `jms-productor`, `jms-consumidor-sync`, `jms-consumidor-async`, `jms-publicador`, `jms-subscriptor-sync` y `jms-subscriptor-async`.

## 3. Ejemplos con Clases Específicas (1p)

Los ejemplos que habéis hecho en el primer ejercicio se basan en los interfaces JMS de más alto nivel. Así, cuando estamos accediendo a una cola, hemos utilizado las clases `ConnectionFactory`, `Connection`, `Session`, `MessageConsumer`, etc...

Se plantea como ejercicio rescribir los 6 proyectos y que cada clase utilice las clases específicas de cada dominio. Es decir, en el caso de la cola, utilizaremos `QueueConnectionFactory`, `QueueConnection`, `QueueSession`, `QueueReceiver`, etc...

Las nuevas clases se situarán en los mismos proyectos, pero nombrando a las clases con el sufijo `Especifico`. Por ejemplo, la nueva clase `Consumidor` se llamará `ConsumidorEspecifico`.

Para tener más claro las relaciones de las clases se recomienda consultar los javadocs del API de JMS: java.sun.com/javaee/6/docs/api/javax/jms/package-summary.html

## 4. Semáforo Hipotecario (2p)

Vamos a hacer un pequeño proyecto para este módulo que va a simular un semáforo hipotecario.

El semáforo va a recibir peticiones sobre hipotecas, y dependiendo del riesgo que conllevan, responderá con Rojo (denegar), Naranja (previo análisis de un consultor financiero) o Verde (aceptar).

El semáforo recibirá los siguientes parámetros:

- Nombre de la entidad financiera
- Cuantía de la hipoteca
- Años de la hipoteca
- Tipo de interés anual hipotecario
- Nómina mensual del cliente (neta, sin retenciones). Suponemos que cobra 12 nominas al año.

Las reglas asociadas al semáforo para responder con uno u otro color dependiendo de la cuota mensual de la hipoteca serán:

- Si la cuota mensual supone menos del 30% de la nomina del cliente, devolverá verde.
- Si la cuota mensual supone entre el 30% y el 40% de la nomina, devolverá naranja.
- Si la cuota mensual supone más del 40% de la nomina, devolverá rojo.

Para calcular la cuota mensual, tenéis la formula en la wikipedia : es.wikipedia.org/wiki/Contrato_de_Hipoteca.

## 4.1. jms-banco

Para realizar esta aplicación necesitaréis crear 2 proyectos. Un proyecto productor de mensajes (`jms-banco`), el cual enviará mensajes a una cola, la cual llamaremos `SemaforoHipotecarioRequestQueue`.

Todo esto lo haremos dentro de una clase `Banco` dentro de `org.especialistajee.jms`.

El mensaje se enviará desde un método con la siguiente firma:

```
void enviaHipoteca(String banco, double cuantia, int anyos, double
interes, double nomina);
```

## 4.2. jms-semaforo

El segundo proyecto (`jms-semaforo`) se encarga de escuchar los mensajes de forma asíncrona y devolver uno de los tres colores comentados anteriormente en la cola de respuesta. La consumición del mensaje se realizará en un método con la siguiente firma:

```
void analizaHipoteca();
```

Este método obtendrá el color de la hipoteca (y de momento, lo sacará por consola).

Para facilitaros el trabajo, se os da comenzada una clase de prueba para comprobar la **lógica de negocio** del semáforo (debéis completar la prueba para que abarque la mayor casuística posible):

```
package es.ua.jtech.jms;

// imports
public class SemaforoBRTest {

    private SemaforoBR sbr = null;

    public SemaforoBRTest() {
        sbr = SemaforoBR.getInstance();
    }

    @Test
    public void devuelveVerde() {
       String color = "Verde";

       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("BBVA", 100000, 30, 5,
1800).equals(color));
       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("BBVA", 50000, 20, 5,
1800).equals(color));

       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("Banesto", 100000, 40, 4,
1500).equals(color));
       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("Banesto", 100000, 35, 3,
1500).equals(color));

    // Resto de métodos de prueba....
}
```

Además, también tenéis el esqueleto para crear la lógica de negocio del semáforo:

```
package es.ua.jtech.jms;

public class SemaforoBR {
    private static SemaforoBR me = new SemaforoBR();

    public final static String ROJO = "Rojo";
    public final static String VERDE = "Verde";
    public final static String NARANJA = "Naranja";

    private SemaforoBR() { }

    public static SemaforoBR getInstance() {
        return me;
    }

    public String obtenerColorHipoteca(
            String banco, double cuantia, int anyos, double
interes, double nomina) {
        String result = null;
```

```
        double interesMensual = interes / 12;
        int plazo = anyos * 12;

        double cuota = (cuantia * interesMensual) /
            (100 * (1 - Math.pow(1 + (interesMensual / 100),
-plazo)));
        double ratio = (cuota / nomina) * 100;

        // TODO Completar logica de negocio
    }
}
```

## 4.3. Enviando y parseando la información

En cuanto al envío y recepción de la información mediante un mensaje de texto, tendremos que agrupar la información para luego parsearla. Para ello, vamos a utilizar el carácter '#' para unir cada campo. Para construir el mensaje haremos algo similar a:

```
String miMensaje = banco + "#" + cuantia + "#" + anyos + "#" +
interes + "#" + nomina;
```

Y para parsearlo, utilizaremos el método `split` de `String`, tal que así:

```
String[] tokens = miMensaje.split("#");

String banco = tokens[0];
double cuantia = Double.parseDouble(tokens[1]);
int anyos = Integer.parseInt(tokens[2]);
double interes = Double.parseDouble(tokens[3]);
double nomina = Double.parseDouble(tokens[4]);
```