Desarrollo de un mecanismo de serialización en J2ME

Celeste Campo Vázquez, Rosa Mª García Rioja, Guillermo Diez-Andino Sancho
Departamento. Ingeniería Telemática - Universidad Carlos III de Madrid
Avda. Universidad 30 Leganés (Madrid), España
{celeste, rgrioja, gdandino}@it.uc3m.es

Abstract

En 1999 aparece Java 2 Micro Edition (J2ME) con el propósito de permitir que aplicaciones Java se ejecuten en dispositivos con capacidad de proceso limitada, como teléfonos móviles, pagers, palm pilots, set-top boxes, etc. Debido a limitaciones de recursos y al modelo de seguridad en J2ME, parte de las características propias de Java han desaparecido en la versión de Java para dispositivos limitados. Entre las características desaparecidas se pueden destacar aquéllas que nos permitían implementar movilidad de objetos: carga dinámica de clases, serialización y reflexión.

En este artículo se plantea el reto de implementar un mecanismo genérico de serialización de modo que los objetos en J2ME puedan convertirse automáticamente en un flujo de bytes, serialización, para posteriormente poder se reconstruidos, deserialización.

I. Introducción

La necesidad de la serialización se ha venido dando implícitamente mediante la utilización de protocolos como RMI, no obstante, existen múltiples campos, como es el de la programación de agentes móviles[1], en el que resulta de vital importancia.

Debido a la ausencia en J2ME del mecanismo conocido como **reflexión**, mediante el cual los propios objetos son capaces de inspeccionarse a sí mismos, averiguando sus métodos, atributos, parámetros, constructores, etc. la serialización que se puede llevar a cabo va a tener ciertas limitaciones, no pudiéndose conseguir una serialización totalmente automatizada, en la que el programador no tenga que intervenir en este proceso.

Se puede decir que lo que se busca conseguir es el soporte básico para que mediante éste y el conocimiento del programador sobre las clases que implementa se disponga de un mecanismo genérico y extensible para transformar y recuperar objetos, en un flujo de bytes y a un flujo de bytes, respectivamente.

Una vez introducidos los conceptos básicos sobre serialización, se va a describir en los siguientes apartados el sistema básico de serialización llevado a cabo. Tras plantear los problemas existentes, apartado II se procede a enumerar los elementos necesarios, apartado III, para llevar a cabo la solución desarrollada en detalle, apartado IV, en este último apartado se concretan los interfaces y las clases necesarias para poder llevar a cabo el proceso de serialización y deserialización así como las decisiones de diseño adoptadas y el formato de los datos establecido.

Finalmente se incluyen una serie de ejemplos descriptivos del potencial que un mecanismo de serialización en J2ME puede ofrecernos, tras lo cual se concluye con un breve apartado en donde se exponen tanto las conclusiones obtenidas con este desarrollo como las posibles líneas futuras de trabajo.

II. Problemas existentes

El principal problema a afrontar a la hora de realizar un mecanismo de serialización reside en la imposibilidad de conocer en tiempo de ejecución los métodos y atributos de las clases. Al contrario que J2ME, la versión estándar de Java J2SE dispone del mecanismo conocido como reflexión, mediante

[1]Programas que son capaces de viajar de forma autónoma a través de la red, serializándose y deserializándose a medida que pasan de una máquina a otra.

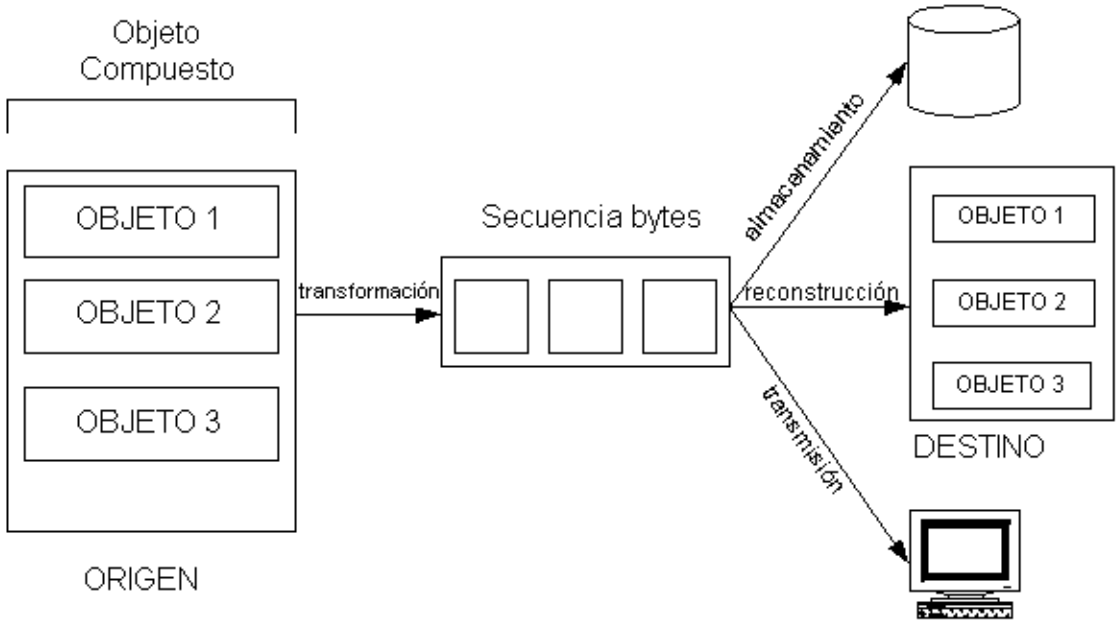

Fig. 1. Proceso de serializacion

el cual se pueden conocer las características de las clases (métodos, atributos, ...) en tiempo de ejecución.

Esta carencia va a suponer que no pueda llevarse a cabo la serialización automática, siendo necesaria la actuación del programador a la hora de especificar los atributos de las clases a serializar en un flujo de bytes. La serialización llevada a cabo facilitará y se encargará de que el programador se despreocupe de cómo se lleva a cabo ésta escritura, del formato interno de los datos así como del proceso de **recuperación** de las clases serializadas, necesitando únicamente su participación a la hora de seleccionar los atributos de las clases a serializar.

Un segundo inconveniente encontrado es el hecho de que en J2SE la gran mayoría de las clases base de que se dispone son serializables y disponen de los mecanismos necesarios para que se puedan serializar/deserializar. Este hecho ofrece la ventaja de que cualquier programador que desee crear una clase serializable puede incluir a su vez multitud de clases ya serializables sin necesidad de preocuparse de la manera en que se serializan/deserializan. Por el contrario en J2ME no existe este concepto ya que ninguna clase es serializable. Debido a esto habrá que buscar alguna manera con la que en caso de que el programador incluya alguna de estas clases en su nueva clase serializable, todas puedan ser serializadas de modo transparente.

III. Elementos necesarios

Si se descarta la reflexión como elemento principal a la hora de serializar (el perfil MIDP no proporciona los mecanismos necesarios para examinar las clases en tiempo de ejecución), lo siguiente más importante es disponer de algún modo de escribir y leer tipos básicos en un flujo de bytes.

Para ello J2ME proporciona las clases `DataInputStream` y `DataOutputStream` que permiten a una aplicación leer y escribir tipos primitivos Java en y de un flujo de bytes.

La clase `DataOutputStream` tiene métodos como `writeBoolean`, `writeByte`, `writeInt`, `writeLong`, `writeShort` y `writeUTF` que escriben datos primitivos en un flujo de bytes de manera independiente y portable. Por otro lado la clase **DataInputStream** permite leer estos datos de un flujo de bytes mediante llamadas a `readBoolean`, `readByte`, `readInt`, `readLong`, `readShort` y `readUTF`.

Basándose en estas dos clases y creando los interfaces y clases necesarias se podrá lograr un mecan-

ismo básico de serialización/deserialización que si bien necesitará de la ayuda del programador, facilitará en la medida de lo posible su funcionamiento.

IV. Solución planteada

La solución planteada pasa en primer lugar por la creación de un interfaz `Serializable` con el fin de *marcar* aquéllas clases que sí saben cómo serializarse/deserializarse[2]. Estas clases deberán implementar sus dos métodos (`readObject` y `writeObject`) de modo que cuando se requiera, sean capaces de llevar a cabo cualquiera de estas dos acciones.

El siguiente paso es la creación de dos nuevas clases **ObjInputStream** y **ObjOutputStream**. Estas clases son las que van a controlar el proceso completo realizando las llamadas necesarias a los métodos `readObject` y `writeObject` que todas las clases serializables deben poseer.

Mediante el diseño planteado se van a poder crear objetos serializables compuestos a su vez por otros objetos, de modo que se forme una estructura de grafo. Esta posibilidad no plantea ningún problema a la hora de realizar el proceso de serialización/deserialización, ya que será el objeto de nivel superior el encargado de realizar las sucesivas llamadas a los objetos que se encuentren por debajo de él.

Finalmente se desea incluir la implementación serializable de alguna clase estándar (como por ejemplo `Vector`) así como un ejemplo de serialización que la utilice.

V. Implementación de la serialización en J2ME

En este apartado se describe el formato de los datos internos del proceso de serialización así como el algoritmo empleado para este fin.

A. Formato de los datos

El formato de los datos es un aspecto muy a tener en cuenta. Existen diversas situaciones en las que el tamaño de las clases serializadas es crítico. La serialización fue originariamente pensada para transmitir objetos por la red, siendo el ancho de banda un recurso limitado que hay que optimizar en todo momento.

Al llevar a cabo el proceso de serialización los diferentes objetos son convertidos en un flujo lineal de bytes, que si bien conservan un formato interno, mediante el cual posteriormente se podrán reconocer sus partes, externamente no conservan ninguna estructura.

A grandes rasgos el formato serializado de una clase sería el siguiente:

1. Nombre de la clase
2. Datos primer atributo
3. ...
4. Datos último atributo

Debido a que cualquiera de los atributos puede ser a su vez un objeto podría darse la siguiente situación:

1. Nombre de la clase
   (a) Nombre clase primer atributo
   (b) Datos primer atributo
   (c) ...
   (d) Datos último atributo
2. ...
3. Datos último atributo

Si nos centramos en la clase VectorS[3] el formato sería:

1. Nombre de la clase: `VectorS`

[2]A partir de ahora se va a emplear serializar tanto para la serialización como la deserialización

[3]Esta clase corresponde a la implementación serializable de la Vector en J2ME ya que se le han añadido los métodos writeObject y readObject definidos por el interfaz Serializable

2. Número de elementos
3. Tipo primer elemento (entero, cadena, boolean, serializable, ...)
4. Dato primer elemento
5. ...
6. Tipo último elemento
7. Dato último elemento

El tipo establecido para los diferentes tipos que un vector pueda contener son los siguientes:

```
NULL         = 0;
ENTERO       = 1;
LONG         = 2;
BOOLEAN      = 3;
CADENA       = 4;
BYTE         = 5;
CHAR         = 6;
SHORT        = 7;
SERIALIZABLE = 9;
```

El tipo NULL representa un elemento del vector vacío, si el tipo es SERIALIZABLE quiere decir que en esa posición del vector hay un objeto de tipo serializable. Con este ejemplo se puede ver la potencia que un mecanismo de serialización transparente puede tener, ya que a la hora de serializar un objeto vector, todos los objetos serializables que éste contenga serán a su vez serializados de manera automática.

B. Algoritmo de serialización y deserialización

A continuación se describe el algoritmo de serialización y deserialización diseñado. Para los objetos serializables el método `writeObject` permite a una clase controlar la serialización de sus propios campos. Cada subclase de un objeto serializable debe definir su propio método `writeObject`.

El método de la clase `writeObject`, si está implementado, es el responsable de salvar el estado de la clase. El formato y estructura es responsabilidad completa de la clase.

B.1 Serialización

Es la clase `ObjOutputStream` la encargada del proceso de serialización. Una vez creada una instancia de ésta, a la cual se le ha asociado un flujo de bytes de escritura está todo listo para comenzar el proceso.

Cuando la clase `ObjOutputStream` recibe una llamada al método `writeObject` obtiene mediante un parámetro el objeto a serializar.

El primer paso consiste en inspeccionar la clase del objeto a serializar, para ello se realiza una llamada al método `object.getClass().getName()` con el que se obtiene el nombre de la clase. Seguidamente se escribe el nombre de la clase en el flujo de bytes de escritura y se llama al método `writeObject` del objeto recibido para que lleve a cabo la serialización de sus atributos.

Cabe destacar que es un proceso recursivo, por lo que no habrá problemas a la hora de serializar objetos anidados, ya que son las propias llamadas a sucesivos métodos `writeObject` las que se encargarán de aplanar esta recursividad.

B.2 Deserialización

Este proceso está íntimamente relacionado con el de la serialización, ya que dependiendo del orden que éste haya seguido habrá que leer una cosa u otra.

Si se parte de un flujo de bytes que alberga uno o varios objetos serializados, el primer paso consiste en leer el nombre de la clase que se encuentra serializada. A continuación se crea un objeto de este

tipo mediante `object = (Serializable)Class.forName(className).newInstance()` tras lo cual bastará con llamar al método `readObject` para que todos los valores de sus atributos sean inicializados.

La llamada al método `readObject` simplemente hace que se vayan leyendo los diferentes valores de los atributos que componen el objeto en el orden escritos, ya que es el propio objeto el que anteriormente se ha serializado, con lo que conocerá cómo deserializarse.

C. Clases serializables

A continuación se enumeran a modo de ejemplo una serie de clases serializables ya implementadas a fin de comprobar el correcto funcionamiento del mecanismo desarrollado.

- **VectorS** Esta clase es capaz de serializarse y deserializarse tal y como lo haría un vector de J2SE mediante sus respectivos métodos `writeObject` y `readObject`.
Representa un vector que puede almacenar cualquier tipo de objetos básicos (cadenas, enteros, booleanos, etc) así como cualquier tipo de objeto complejo siempre que éste sea serializable.
- **Prueba** Representa una clase que contiene a su vez otras clases también serializables (VectorS).
- **Config** Representa una configuración de un sistema así como las operaciones para escribirse y leerse en y de un flujo de bytes.

D. Diagrama de clases

En la figura 2 se muestra el diagrama de casos de uso de la serialización. En éste se indican las dos operaciones que se pueden llevar a cabo, *serialización* y *deserialización*.

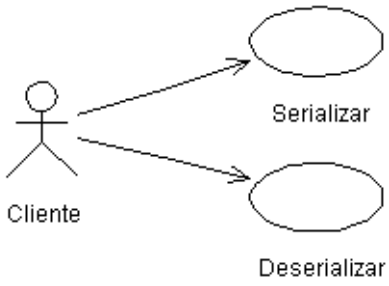

Fig. 2. Diagrama de casos de uso serialización

En la figura 3 se muestra el diagrama de clases de la serialización.

Cualquier aplicación que desee serializar una clase deberá crear un `ObjOutputStream`. Si lo que se desea es deserializar habrá que emplear un `ObjInputStream`.

D.1 Interfaces

El único interfaz existente es el siguiente:

- **Serializable** Sirve para **marcar** que una clase es serializable, ya que al implementarlo tendrá definidos los métodos necesarios para poderse escribir en flujo de bytes con un formato dato y posteriormente recuperarse de ese mismo flujo.

E. Clases

Aquí se incluyen únicamente las clases necesarias para llevar a cabo el proceso de la serialización.

- **ObjInputStream** Representa un objeto serializado del cual se van a poder leer los diferentes objetos que éste alberga. Esta clase está asociada a un flujo de bytes que es realmente la información. Hereda de la clase `DataInputStream`, con lo que todos sus métodos estarán disponibles en esta clase.
- **ObjOutputStream** Esta clase se va a encargar de serializar los objetos recibidos en su método `writeObject` en el flujo de bytes obtenido en su constructor. Se apoya en los métodos de `DataOutputStream`.

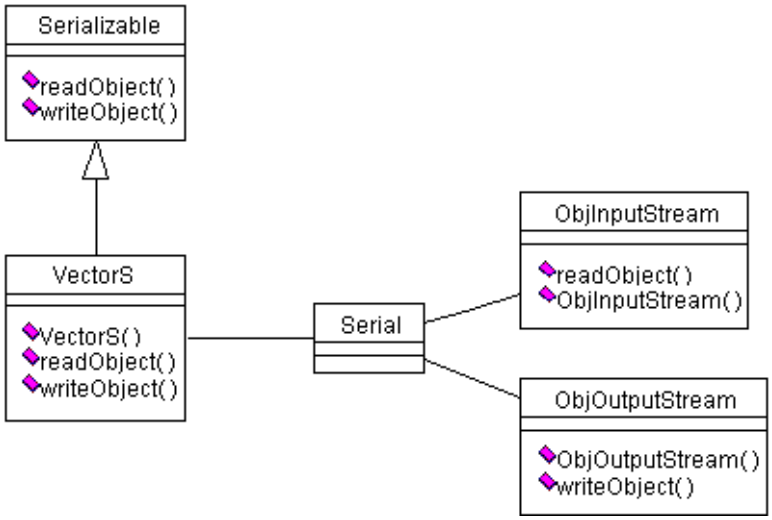

Fig. 3. Diagrama de clses serialización.

A medida que se realicen llamadas a su método `writeObject` los diferentes objetos se irán convirtiendo en un flujo lineal de bytes con una estructura predefinida mediante la cual podrán ser posteriormente reconstruidos a través de la clase `ObjInputStream`.

VI. Transporte de objetos a través de sockets

Como se verá a continuación una de las utilidades de la serialización es el transporte de objetos a través de sockets.

Si consideramos el caso del protocolo de fechas, ampliamente utilizado en sistemas UNIX, basta con conectarnos al puerto 13 de un servidor para obtener la fecha y la hora. El problema aquí es la necesidad de utilizar caracteres ASCII estándar. Por el contrario, si se emplea la serialización para enviar un objeto **Date**, este problema dejará de existir.

Otra utilidad de los sockets en cuanto a la serialización se refiere es la posibilidad de construir un servidor que proporcione objetos serializados que los diferentes clientes reconstruirán previamente a su utilización.

Un último ejemplo de la utilidad de la serialización y los sockets es la creación de una plataforma de agentes, programas capaces de serializarse y viajar a otra máquina a través de la red empleando para ello los sockets.

Para llevar a cabo cualquiera de las posibilidades mencionadas bastaría asociar como flujo de bytes de las clases `ObjInputStream` y `ObjOutputStream` los correspondientes sockets de entrada y salida.

A. Serialización y RMS

Centrándose en J2ME la serialización no sólo puede resultar de gran utilidad en el transporte mediante sockets, al contrario, caben innumerables posibilidades mediante el almacenamiento persistente que J2ME proporciona con el paquete `javax.microedition.rms`.

Podría utilizarse RMS para almacenar los diferentes objetos de una aplicación sin necesidad de una transformación en un formato externo de representación, ya que la aplicación sería capaz de escribir y recuperar sus propios objetos independientemente del sistema de almacenaje empleado.

VII. Ejemplo de funcionamiento

En este apartado se describen los pasos a seguir para llevar a cabo un caso básico de serialización.

A. Descripción

Se pretende implementar una clase compuesta por diversos objetos, de modo que ésta sea serializable.

Una vez inicializados los valores de esta clase, se procederá a su serialización, obteniendo un flujo de bytes que represente el estado de este objeto.

Seguidamente se creará un nuevo objeto del mismo tipo que será inicializado a partir del resultado obtenido con la deserialización del flujo de bytes anteriormente mencionado.

B. Cómo hacer una clase serializable

El primer paso para hacer una clase serializable consiste en hacer que implemente el interfaz `Serializable`, con lo cual se obliga al programador a proporcionar los mecanismos de serialización(`writeObject`) y deserialización(`readObject`).

```
public class Prueba implements Serializable
```

Supongamos que la clase se ha llamado `Prueba`. El siguiente paso consiste en implementar el método para convertir la clase en un flujo de bytes **writeObject** tal y como se muestra a continuación.

```
public void writeObject(ObjOutputStream out) throws IOException{
out.writeLong(puerto);
out.writeLong(bb);
out.writeInt(cc);
out.writeUTF(ruta);
out.writeObject(vect);
}
```

En el código anterior se observa que el programador debe encargarse de escribir los atributos de la clase en el flujo de bytes recibido por parámetro. Para el caso de objetos que ya disponen de serialización (como la clase VectorS), basta con realizar una llamada a su método de escritura.

Una vez implementado el método de serialización hay que proceder al de la deserialización, **readObject**, que deberá ser en todo momento el proceso inverso al de la escritura.

```
public void readObject(ObjInputStream in) throws IOException{
puerto = in.readLong();
bb = in.readLong();
cc = in.readInt();
ruta = in.readUTF();
vect2 = (VectorS)in.readObject();
}
```

Con este método la clase tomaría los valores obtenidos del resultado de deserializar el flujo de bytes incluido en el objeto `ObjInputStream` recibido por parámetro.

C. Funcionamiento

Tras haber creado una clase serializable, llega el momento de utilizarla. En este caso se va a emplear la serialización con dos fines diferentes.

El primero de ellos va a ser el de inicializar un objeto con el resultado de otro ya serializado. La segunda funcionalidad será la de almacenar el objeto serializado en un soporte permanente.

Los pasos a seguir son los siguientes:

1. Crear el flujo de datos en el que se va a serializar

```
ByteArrayOutputStream datos = new ByteArrayOutputStream();
ObjOutputStream out = new ObjOutputStream(datos);
```

2. Inicializar el objeto a serializar

```
Prueba origen = new Prueba();
origen.setParametros(11,22,33);
```

3. Serializar el objeto

```
try{
  out.writeObject(serializo);
  out.flush();
  out.close();
}
catch(IOException e){
  System.out.println(e.getMessage());
}
```

4. Almacenar el objeto en el sistema de archivos

```
FileSystem fs = FileSystem.getInstance();
fs.addFolder("/objects/");
fs.addFile("/objects/","object1.rms","r",datos.toByteArray());
```

5. Inicializar un segundo objeto deserializando el primero. Para ello hay que crear en primer lugar el `ObjInputStream` necesario, así como el objeto a inicializar.

```
ByteArrayInputStream leo =
new ByteArrayInputStream(datos.toByteArray());
ObjInputStream sal = new ObjInputStream(leo);

try{
  sal = new ObjInputStream(leo);
  Prueba leido = (Prueba) sal.readObject();
}
catch(IOException e){
  System.out.println(e.getMessage());
}
```

En este apartado se he mostrado un sencillo ejemplo, aunque las posibilidades de la serialización son mucho más amplias y dependen de la utilidad que se le quiera dar.

VIII. Conclusiones

La serialización es un mecanismo ampliamente utilizado en la programación. Generalmente no somos conscientes de su utilización, ya que su funcionamiento suele resultar transparente al usuario. Con el surgimiento de la versión de la plataforma de Sun J2ME, orientada a dispositivos limitados este mecanismo ha sido suprimido por diversas razones (limitaciones de memoria, modelo de seguridad sandbox, ...). Debido a esto, van a existir multitud de aplicaciones no realizables en J2ME así como otras muchas a las que les va a resultar imposible interoperar con aplicaciones residentes en PCs que hagan uso de la serialización como mecanismo de intercambio de información.

La realización de un mecanismo de serialización/deserialización en J2ME resulta innovador y de gran interés, ya que supone un acercamiento cada vez mayor a la funcionalidad proporcionada por la versión estándar de Java, permitiendo realizar cada vez un sinfín de aplicaciones hasta ahora impensables en los dispositivo limitados.

Agradecimientos

Este trabajo ha sido parcialmente subvencionado por el marco de la cátedra Nokia de la Universidad Carlos III de Madrid.

References

[1] E. Davids. Java object serialization in parsable ascii, 1998. AUUG-Vic/CAUUG: Summer Chapter Technical Conference 98.
[2] R. Dragomirescu. Dynamic classloading in the kvm, Apr. 2000.
[3] E. Giguere. Simple store and forward messaging for midp applications, Mar. 2002. http://www.ericgiguere.com.
[4] W. Grosso. Serialization, Oct. 2001. http://www.onjava.com.
[5] S. Halloway. Serialization in the real world, Feb. 2000. http://developer.java.sun.com/developer/TechTips/2002/tt0229.html.
[6] A. Kaminsky. Jinime: Jini connection technology for mobile devices, Aug. 2000. Rochester Institute of Technology.
[7] B. Kurniawan. Learning polymorphism and object serialization, Aug. 2001. http://www.oreillynet.com.
[8] G. McCluskey. Using java reflection, Jan. 1998. http://developer.java.sun.com.
[9] S. Microsystems. System architecture, 1997. Descripción detallada de todo el proceso de serialización/deserialización en J2SE.
[10] S. MicroSystems. Validating deserialized objects, Mar. 2002. http://java.sun.com/j2se/1.4/docs/guide/serialization.
[11] E. Miedes. Serialización de objetos en java.
[12] L. Suarez. El paquete java.io, July 2001. http://www.javahispano.com.