JSR-82: Bluetooth desde Java

Alberto Gimeno Brieba

JSR-82: Bluetooth desde Java™

por Alberto Gimeno Brieba

El presente tutorial trata sobre la programación de aplicaciones Java™ que hagan uso de Bluetooth. Más concretamente trata sobre las APIs definidas en el JSR-82.

Tabla de contenidos

Lista de tablas

Capítulo 1. Introducción

Sobre la tecnología Bluetooth

Bluetooth es una tecnología de comunicación inalámbrica omnidireccional. Se ideó pensando en dispositivos de bajo consumo y comunicaciones a corta distancia (10 metros). Se trata de una tecnología barata con un ancho de banda reducido: hasta 11 Mbit/s. Es ideal para periféricos de ordenador (ratón, teclado, manos libres,...) y dispositivos móviles (teléfonos móviles, PDAs, Pocket PCs,...).

Mediante Bluetooth es posible formar pequeñas redes de dispositivos conectados denominadas piconets. Se pueden conectar varias piconets formando lo que se denomina una scatternet.

Las principales aplicaciones de Bluetooth son: transferencia de archivos, sincronización de dispositivos y conectividad de periféricos.

Sobre el API JSR-82

Este API está dividida en dos partes: el paquete javax.bluetooth y el paquete javax.obex. Los dos paquetes son totalmente independientes. El primero de ellos define clases e interfaces básicas para el descubrimiento de dispositivos, descubrimiento de servicios, conexión y comunicación. La comunicación a través de javax.bluetooth es a bajo nivel: mediante flujos de datos o mediante la transmisión de arrays de bytes.

Por el contrario el paquete javax.obex permite manejar el protocolo de alto nivel OBEX (OBject EXchange). Este protocolo es muy similar a HTTP y es utilizado sobre todo para el intercambio de archivos. El protocolo OBEX es un estándar desarrollado por IrDA y es utilizado también sobre otras tecnologías inalámbricas distintas de Bluetooth.

La plataforma principal de desarrollo del API JSR-82 es J2ME. El API ha sido diseñada para depender de la configuración CLDC. Sin embargo existen implementaciones para poder hacer uso de este API en J2SE. Al final del tutorial se listan la mayoría de las implementaciones del JSR-82 existentes.

Sobre este tutorial

La finalidad de este tutorial es la de abordar el API definida en la especificación JSR-82 en su totalidad a través de ejemplos. Los ejemplos serán MIDlets que han sido probados usando el Java Wireless Toolkit 2.2Beta para Windows.

Es muy recomendable tener la documentación javadoc del API a mano. Se puede descargar de la página del JSR-82 en jcp.org [http://jcp.org/en/jsr/detail?id=82]

Capítulo 2. El paquete javax.bluetooth

En una comunicación Bluetooth existe un dispositivo que ofrece un servicio (servidor) y otros dispositivos acceden a él (clientes). Dependiendo de qué parte de la comunicación debamos programar deberemos realizar una serie de acciones diferentes.

Un cliente Bluetooth deberá realizar las siguientes:

- Búsqueda de dispositivos. La aplicación realizará una búsqueda de los dispositivos Bluetooth a su alcance que estén en modo conectable.
- Búsqueda de servicios. La aplicación realizará una búsqueda de servicios por cada dispositivo.
- Establecimiento de la conexión. Una vez encontrado un dispositivo que ofrece el servicio deseado nos conectaremos a él.
- Comunicación. Ya establecida la conexión podremos leer y escribir en ella.

Por otro lado, un servidor Bluetooth deberá hacer las siguientes operaciones:

- Crear una conexión servidora
- Especificar los atributos de servicio
- Abrir las conexiones clientes

Algunas clases básicas

Clase `LocalDevice`

Un objeto `LocalDevice` representa al dispositivo local. Este objeto será el punto de partida de prácticamente cualquier operación que vayamos a llevar a cabo en este API.

Alguna información de interés que podemos obtener a través de este objeto es, por ejemplo, la dirección Bluetooth de nuestro dispositivo, el apodo o "friendly-name" (también llamado "Bluetooth device name" o "user-friendly name"). A través de este objeto también podemos obtener y establecer el modo de conectividad: la forma en que nuestro dispositivo está o no visible para otros dispositivos. Esta clase es un "singleton"; para obtener la única instancia existente de esta clase llamaremos al método `getLocalDevice()` de la clase `LocalDevice`. Veamos un ejemplo:

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;

public class Ejemplo1 extends MIDlet {

        public void startApp() {
                LocalDevice localDevice = null;
                try {
                        localDevice = LocalDevice.getLocalDevice();
                } catch(BluetoothStateException e) {
                        System.out.println("Error al iniciar"+
                                " el sistema Bluetooth");
```

```
        return;
}
String address = localDevice.getBluetoothAddress();
System.out.println("Dirección bluetooth del"+
                        " dispositivo local: "+address);
String friendlyName = localDevice.getFriendlyName();
if(friendlyName == null)
        System.err.println("El dispositivo "+
                "local no soporta un 'friendly-name'"+
                " o no ha sido establecido");
else
        System.out.println("El 'friendly-name'"+
                " del dispositivo local es:"+
                friendlyName);
try {
        if(!localDevice.setDiscoverable(DiscoveryAgent.GIAC))
                System.out.println("El dispositivo no"+
                        " soporta el modo de "+
                        "conectividad 'GIAC'");
} catch(BluetoothStateException e) {
        System.err.println("No se pudo establecer"+
                        " la propiedad 'discoverable'"+
                        " a 'GIAC'");
}
DeviceClass deviceClass = localDevice.getDeviceClass();

int discoverable = localDevice.getDiscoverable();
String discoverableString = null;
switch(discoverable) {
        case DiscoveryAgent.GIAC:
        discoverableString =
                "General / Unlimited Inquiry Access";
        break;
        case DiscoveryAgent.LIAC:
        discoverableString =
                "Limited Dedicated Inquiry Access";
        break;
        case DiscoveryAgent.NOT_DISCOVERABLE:
        discoverableString = "Not discoverable";
        break;
        default:
        discoverableString = "Desconocido";
}

StringBuffer device = new StringBuffer("0x");
device.append(Integer.toHexString(
        deviceClass.getMajorDeviceClass()));
device.append(", 0x");
device.append(Integer.toHexString(
        deviceClass.getMinorDeviceClass()));
device.append(", 0x");
device.append(Integer.toHexString(
        deviceClass.getServiceClasses()));

//interfaz de usuario
Form main = new Form("Ejemplo LocalDevice");
main.append(new TextField("Dirección bluetooth",
        address, 12, TextField.UNEDITABLE));
main.append(new TextField("Nombre del dispositivo",
        friendlyName,
        friendlyName.length(), TextField.UNEDITABLE));
main.append(new TextField("Modo de conectividad",
        discoverableString,
        discoverableString.length(),
        TextField.UNEDITABLE));
main.append(new TextField("Tipo de dispositivo",
        device.toString(),
        device.length(),
        TextField.UNEDITABLE));
```

```
                Display.getDisplay(this).setCurrent(main);
        }

        public void pauseApp() {
        }

        public void destroyApp(boolean unconditional) {
        }
}
```

Los posibles valores que admite `setDiscoverable()` están definidos en la clase `DiscoveryAgent` como campos estáticos. Estos son:

- DiscoveryAgent.GIAC (General/Unlimited Inquiry Access Code) : Conectividad ilimitada.
- DiscoveryAgent.LIAC (Limited Dedicated Inquiry Access Code) : Conectividad limitada.
- DiscoveryAgent.NOT_DISCOVERABLE : El dispositivo no será visible para el resto de dispositivos.

Excepción `BluetoothStateException`

Al llamar al método `getLocalDevice()` se puede producir una excepción del tipo `BluetoothStateException`. Esto significa que no se pudo inicializar el sistema Bluetooth. La excepción `BluetoothStateException` extiende de `java.io.IOException` y no añade ningún método adicional.

En el paquete javax.bluetooth se definen dos tipos más de excepciones que veremos más adelante.

Clase `DeviceClass`

En el ejemplo hemos usado el método `getDeviceClass()` que devuelve un objeto de tipo `DeviceClass`. Este tipo de objeto describe el tipo de dispositivo. A través de sus métodos podremos saber, por ejemplo, si se trata de un teléfono, de un ordenador,... En la documentación del API hay una pequeña tabla con los posibles valores que devuelven los métodos de esta clase y su significado.

Clase `UUID`

La clase `UUID` (*universally unique identifier*) representa indentificadores únicos universales. Se trata de enteros de 128 bits que identifican protocolos y servicios. Como veremos más adelante un dispositivo puede ofrecer varios servicios. Los UUID servirán para identificarlos. En la documentación de la clase `UUID` se puede encontrar una tabla con los protocolos y servicios más comunes y sus UUIDs asignadas.

Del mismo modo que cuando hacemos aplicaciones de ejemplo con sockets usamos números de puertos "inventados", que no estén asociados a ningún protocolo existente, en los ejemplos de este tutorial usaremos una UUID cualquiera.

Podremos crear un objeto `UUID` a partir de un String o de un entero largo. En los ejemplos del tutorial crearemos `UUIDs` mediante enteros que representaremos en hexadecimal, como se acostumbra a hacer en los documentos relacionados con Bluetooth.

Búsqueda de dispositivos y servicios

Las búsquedas de dispositivos y servicios son tareas que solamente realizarán los dispositivos clientes.

Clase `DiscoveryAgent`

Las búsquedas de dispositivos y servicios Bluetooth las realizaremos a través del objeto `DiscoveryAgent`. Este objeto es único y lo obtendremos a través del método `getDiscoveryAgent()` del objeto `LocalDevice`:

```
DiscoveryAgent discoveryAgent =
        LocalDevice.getLocalDevice().getDiscoveryAgent();
```

A través de `DiscoveryAgent` tenemos la posibilidad de obtener un array de dispositivos que el usuario haya especificado como "ya conocidos" (PREKNOWN, quizá una lista de dispositivos "favoritos") o un array de dispositivos descubiertos en búsquedas anteriores (CACHED). Esto lo haremos llamando al método `retrieveDevices()` pasándole DiscoveryAgent.PREKNOWN o DiscoveryAgent.CACHED respectivamente. Este método no devolverá un array vacío si no encuentra dispositivos que coincidan con el criterio especificado, en vez de ello devolverá `null`.

El array devuelto por este método es de objetos `RemoteDevice`, los cuales, representan dispositivos remotos. La clase `RemoteDevice` permite obtener la dirección Bluetooth del dispositivo que representa a través del método `getBluetoothAddress()` en forma de `String`. También podremos obtener el "friendly-name" del dispositivo a través del método `getFriendlyName()`. Este último método requiere un argumento de tipo booleano. Este argumento permite especificar si se debe forzar a que se contacte con el dispositivo para preguntar su "friendly-name" (`true`) o bien se puede obtener un "friendly-name" que tuvo en una ocasión previa (`false`). El método `getFriendlyName()` puede lanzar una `java.io.IOException` en caso de que no se pueda contactar con el dispositivo o este no provea un "friendly-name".

```
import javax.microedition.midlet.MIDlet;
import javax.bluetooth.*;
import java.io.IOException;

public class Ejemplo2 extends MIDlet {

        public void startApp() {
                LocalDevice localDevice = null;
                try {
                        localDevice = LocalDevice.getLocalDevice();
                } catch(BluetoothStateException e) {
                        System.out.println("Error al iniciar"+
                                " el sistema Bluetooth");
                        return;
                }
                DiscoveryAgent discoveryAgent =
                        localDevice.getDiscoveryAgent();

                RemoteDevice[] preknown =
                        discoveryAgent.retrieveDevices(
                        DiscoveryAgent.PREKNOWN);
                if(preknown == null) {
                        System.out.println("No hay"+
                                " dispositivos conocidos");
                } else {
                        System.out.println("Dispositivos conocidos:");
```

```
                        for(int i=0; i<preknown.length; i++) {
                                String address =
                                        preknown[i].getBluetoothAddress();
                                String friendlyName = null;
                                try {
                                        preknown[i].getFriendlyName(false);
                                } catch(IOException e) {
                                        System.err.println(e);
                                }
                                System.out.println(i+": "+
                                        friendlyName+"; "+address);
                        }
                }

                RemoteDevice[] cached =
                        discoveryAgent.retrieveDevices(
                        DiscoveryAgent.CACHED);
                if(cached == null) {
                        System.out.println("No hay dispositivos"+
                                " encontrados en búsquedas previas");
                } else {
                        System.out.println("Dispositivos"+
                                " encontrados en búsquedas previas:");
                        for(int i=0; i<cached.length; i++) {
                                String address =
                                        cached[i].getBluetoothAddress();
                                String friendlyName = null;
                                try {
                                        cached[i].getFriendlyName(false);
                                } catch(IOException e) {
                                        System.err.println(e);
                                }
                                System.out.println(i+": "+
                                        friendlyName+"; "+address);
                        }
                }
        }

        public void pauseApp() {
        }

        public void destroyApp(boolean unconditional) {
        }
}
```

Antes hemos visto que el método `retrieveDevices()` no realiza una búsqueda, sino que nos devuelve un array de dispositivos remotos ya conocidos o encontrados en búsquedas anteriores. Para comenzar una nueva búsqueda de dispositivos llamaremos al método `startInquiry()`. Este método requiere dos argumentos. El primer argumento es un entero que especifica el modo de conectividad que deben tener los dispositivos a buscar. Este valor deberá ser DiscoveryAgent.GIAC o bien DiscoveryAgent.LIAC. El segundo argumento es un objeto que implemente `DiscoveryListener`. A través de este último objeto serán notificados los dispositivos que se vayan descubriendo. Para cancelar la búsqueda usaremos el método `cancelInquiry()`.

La interfaz `DiscoveryListener`

La interfaz `DiscoveryListener` tiene los siguientes métodos:

public void deviceDiscovered(RemoteDevice rd, DeviceClass c)

public void inquiryCompleted(int c)

public void servicesDiscovered(int transID, ServiceRecord[] sr)

public void serviceSearchCompleted(int transID, int respCode)

Los dos primeros métodos serán llamados durante el proceso de búsqueda de dispositivos. Los otros dos métodos serán llamados durante un proceso de búsqueda de servicios. El proceso de búsqueda de servicios se verá más adelante. Pasemos a investigar más profundamente los dos primeros métodos que son los usados durante una búsqueda de dispositivos.

public void deviceDiscovered(RemoteDevice rd, DeviceClass c)

Cada vez que se descubre un dispositivo se llama a este método. Nos pasa dos argumentos. El primero es un objeto de la clase `RemoteDevice` que representa el dispositivo encontrado. El segundo argumento nos permitirá determinar el tipo de dispositivo encontrado.

public void inquiryCompleted(int c)

Este método es llamado cuando la búsqueda de dispositivos ha finalizado. Nos pasa un argumento entero indicando el motivo de la finalización. Este argumento podrá tomar los valores: DiscoveryListener.INQUIRY_COMPLETED si la búsqueda ha concluido con normalidad, DiscoveryListener.INQUIRY_ERROR si se ha producido un error en el proceso de búsqueda, o DiscoveryListener.INQUIRY_TERMINATED si la búsqueda fue cancelada.

Veamos un ejemplo de descubrimiento de dispositivos. Este ejemplo se puede ejecutar varias veces en el Java Wireless Toolkit y cada instancia del emulador de móviles podrá encontrar al resto de instancias.

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import java.io.IOException;

public class Ejemplo3 extends MIDlet implements
        CommandListener, DiscoveryListener {

        private Command comenzar, cancelar;
        private List dispositivos;
        private Form busqueda;

        public void startApp() {
                dispositivos = new List("Ejemplo descubrimiento"+
                        " de dispositivos", List.IMPLICIT);
                comenzar = new Command("Buscar", Command.ITEM, 1);
                cancelar = new Command("Cancelar", Command.ITEM, 1);

                busqueda = new Form("Busqueda de dispositivos");
                busqueda.append(new Gauge("Buscando dispositivos...",
                        false, Gauge.INDEFINITE,
                        Gauge.CONTINUOUS_RUNNING));
                busqueda.addCommand(cancelar);
                busqueda.setCommandListener(this);

                dispositivos.addCommand(comenzar);
                dispositivos.setCommandListener(this);

                LocalDevice localDevice = null;
                try {
                        localDevice = LocalDevice.getLocalDevice();
                        localDevice.setDiscoverable(DiscoveryAgent.GIAC);
                        Display.getDisplay(this).setCurrent(dispositivos);
                } catch(Exception e) {
                        e.printStackTrace();
                        Alert alert = new Alert("Error",
                                "No se puede hacer uso de Bluetooth",
```

```
                        null, AlertType.ERROR);
                Display.getDisplay(this).setCurrent(alert);
        }
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {
        if (c == comenzar) {
                comenzar();
        } else if( c == cancelar) {
                cancelar();
        }
}

private void comenzar() {
        dispositivos.deleteAll();
        try {
                DiscoveryAgent discoveryAgent =
                LocalDevice.getLocalDevice().getDiscoveryAgent();
                discoveryAgent.startInquiry(DiscoveryAgent.GIAC,
                        this);
                Display.getDisplay(this).setCurrent(busqueda);
        } catch(BluetoothStateException e) {
                e.printStackTrace();
                Alert alert = new Alert("Error",
                        "No se pudo comenzar la busqueda",
                        null, AlertType.ERROR);
                Display.getDisplay(this).setCurrent(alert);
        }
}

private void cancelar() {
        try {
                DiscoveryAgent discoveryAgent =
                LocalDevice.getLocalDevice().getDiscoveryAgent();
                discoveryAgent.cancelInquiry(this);
        } catch(BluetoothStateException e) {
                e.printStackTrace();
                Alert alert = new Alert("Error",
                        "No se pudo cancelar la busqueda",
                        null, AlertType.ERROR);
                Display.getDisplay(this).setCurrent(alert);
        }
}

//metodos de la interfaz DiscoveryListener
public void deviceDiscovered(RemoteDevice remoteDevice,
        DeviceClass deviceClass) {
        String address = remoteDevice.getBluetoothAddress();
        String friendlyName = null;
        try {
                friendlyName = remoteDevice.getFriendlyName(true);
        } catch(IOException e) { }

        String device = null;
        if(friendlyName == null) {
                device = address;
        } else {
                device = friendlyName + " ("+address+")";
        }

        dispositivos.append(device, null);
}
```

```
        public void inquiryCompleted(int discType) {
                switch(discType) {
                        case DiscoveryListener.INQUIRY_COMPLETED:
                                System.out.println("Busqueda"+
                                " concluida con normalidad");
                        break;
                        case DiscoveryListener.INQUIRY_TERMINATED:
                                System.out.println("Busqueda cancelada");
                        break;
                        case DiscoveryListener.INQUIRY_ERROR:
                                System.out.println("Busqueda"+
                                " finalizada debido a un error");
                        break;
                }
                Display.getDisplay(this).setCurrent(dispositivos);
        }

        public void servicesDiscovered(int transID,
                ServiceRecord[] servRecord) {
        }

        public void serviceSearchCompleted(int transID, int respCode) {
        }
}
```

Para realizar una búsqueda de servicios también usaremos la clase `DiscoveryAgent` e implementaremos la interfaz `DiscoveryListener`. En este caso nos interesarán los métodos `servicesDiscovered()` y `serviceSearchCompleted()` de la interfaz `DiscoveryListener`. Para comenzar la búsqueda usaremos `searchServices()` de la clase `DiscoveryAgent`. Este método es un poco complejo así que lo vamos a ver con más detalle:

public int searchServices(int[] attrSet, UUID[] uuidSet, RemoteDevice btDev, DiscoveryListener discListener) throws BluetoothStateException;

El primer argumento es un array de enteros con el que especificaremos los atributos de servicio en los que estamos interesados. Como veremos más adelante, cuando hablemos de la interfaz `ServiceRecord`, los servicios son descritos a través de atributos que son identificados numéricamente. Pues bien, este array contendrá los identificadores de estos atributos.

El segundo argumento es un array de identificadores de servicio. Nos permite especificar los servicios en los que estamos interesados.

El tercer argumento es el dispositivo remoto sobre el que vamos a realizar la búsqueda.

Por último argumento pasaremos un objeto que implemente `DiscoveryListener` que será usado para notificar los eventos de búsqueda de servicios. Ahora veremos estos eventos.

Es posible que queramos hacer diversas búsquedas de servicios al mismo tiempo. El método `searchServices()` nos devolverá un entero que identificará la búsqueda. Este valor entero nos servirá para saber a qué búsqueda pertenecen los eventos `servicesDiscovered()` y `serviceSearchCompleted()`. Ahora mismo comenzamos a estudiar estos eventos.

public void serviceSearchCompleted(int transID, int respCode)

Este método es llamado cuando se finaliza un proceso de búsqueda. El primer argumento identifica el proceso de búsqueda (que recordemos que es el valor devuelto al invocar el método `searchServices()` de la clase `DiscoveryAgent`). El segundo argumento indica el motivo de finalización de la búsqueda. Este último argumento puede tomar los si-

guientes valores:

- DiscoveryListener.SERVICE_SEARCH_COMPLETED: El proceso de búsqueda ha finalizado con normalidad
- DiscoveryListener.SERVICE_SEARCH_TERMINATED: El proceso de búsqueda ha sido cancelado
- DiscoveryListener.SERVICE_SEARCH_NO_RECORDS: No se han encontrado registros en el proceso de búsqueda
- DiscoveryListener.SERVICE_SEARCH_ERROR: Hubo un error durante el proceso de búsqueda
- DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHABLE: No se pudo conectar al dispositivo sobre el que se quería realizar la búsqueda.

Podemos cancelar un proceso de búsqueda de servicios llamando al método `cancelServiceSearch()` pasándole como argumento el identificador de proceso de búsqueda, que es el número entero devuelto cuando se comenzó la búsqueda con `searchServices()`.

public void servicesDiscovered(int transID, ServiceRecord[] sr)

Este método nos notifica que se han encontrado servicios. El primer argumento es un entero que es el que identifica el proceso de búsqueda. Este entero es el mismo que devolvió `searchDevices()` cuando se comenzó la búsqueda.

El segundo argumento es un array de objetos `ServiceRecord`. Un objeto `ServiceRecord` describe las características de un servicio bluetooth. Un servicio Bluetooth se describe mediante atributos, los cuales se identifican numéricamente, es decir, un servicio Bluetooth tiene una lista de pares identificador-valor que lo describen. El objeto `ServiceRecord` se encarga de almacenar estos pares.

Los identificadores son números enteros y los valores son objetos de tipo `DataElement`. Los objetos `DataElement` encapsulan los posibles tipos de datos mediante los cuales se pueden describir los servicios bluetooth. Estos tipos de datos son: valor nulo, enteros de diferente longitud, arrays de bytes, URLs, UUIDs, booleanos, `String`s o enumeraciones (`java.util.Enumeration`) de los tipos anteriores.

Para entender mejor para qué sirven los atributos de servicio nada mejor que un ejemplo. Supongamos que queremos crear un servicio de impresión, es decir, supongamos que deseamos hacer una aplicación cliente-servidor en la cual el cliente envía información al servidor para imprimirla. En este caso podríamos indicar en los atributos de servicio qué tipos de impresora hay disponibles en el servicio de impresión (impresora láser blanco y negro tamaño papel din A-4, impresora de tinta color tamaño del papel din A-4,...). El cliente realizará una búsqueda de los dispositivos que ofrezcan un servicio de impresión y los atributos le permitirán elegir el mejor servicio de impresión encontrado para imprimir la información seleccionada por el usuario. Por ejemplo para imprimir una imagen buscaremos un servicio de impresión que disponga de impresora a color, sin embargo para imprimir un texto nos será indiferente si las impresoras disponibles son a color o en blanco y negro.

Ahora que hemos entendido la utilidad de los atributos de servicio veamos cómo podemos acceder a ellos a través de un objeto `ServiceRecord`. A través del método `getAttributeValue()` de un objeto `ServiceRecord` podemos obtener el valor de un atributo de servicio pasándole su identificador como argumento. También podemos obtener un array de todos los identificadores de atributo de servicios disponibles mediante el método `getAttributeIDs()`.

...

```
ServiceRecord serviceRecord = ...;
DataElement value = serviceRecord.getAttributeValue(0x54321);
...
```

La clase `DataElement`

Como hemos visto la clase `DataElement` se encarga de encapsular los tipos de datos disponibles para describir un atributo de servicio. Estos tipos de datos son: valor nulo, enteros de diferente longitud, arrays de bytes, URLs, UUIDs, booleanos, `Strings` o enumeraciones (`java.util.Enumeration`) de los tipos anteriores.

Para saber qué tipo de dato está almacenando un `DataElement` usaremos el método `getDataType()`. Según el tipo de dato que esté almacenando usaremos un método diferente para obtenerlo. Si está almacenano un valor booleano usaremos el método `getBoolean()`, si se trata de un entero usaremos el método `getLong()` y en otro caso usaremos el método `getValue()` que devolverá un `java.lang.String` si está almacenando una URL o un `String`, o bien devolverá un `javax.bluetooth.UUID` si se trata de un UUID, o bien un `java.util.Enumeration`. Para aclarar esto veamos la siguiente tabla que relaciona el valor devuelto por `getDataType()`, el valor almacenado, y el método que debemos usar para almacenar dicho valor:

Tabla 2.1. Tipos de datos DataElemnet

Tipo de dato DataElement	Descripción	Tipo de dato en java	Método
DataElement.NULL	Representa un valor nulo	null	
U_INT_1, U_INT_2, U_INT_4, INT_1, INT_2, INT_4, e INT_8.	Almacena valores enteros de 1, 2, 4 u 8 bytes con o sin signo	long	getLong()
DataElement.BOOL	Representa un valor booleano	boolean	getBoolean()
DataElement.STRING	Almacena una cadena de texto.	java.lang.String	getValue()
DataElement.URL	Almacena una URL.	java.lang.String	getValue()
DataElement.UUID	Almacena un UUID	javax.bluetooth.UUID	getValue()
DataElement.DATSEQ	Almacena una secuencia de objetos DataElement.	java.util.Enumeration	getValue()
DataElement.DATALT	Almacena una secuencia de objetos alternativos DataElement.	java.util.Enumeration	getValue()

Es sencillo: exceptuando los valores enteros y los tipos booleanos que necesitan un método propio (`getLong()` y `getBoolean()` respectivamente) porque son tipos primitivos y no objetos, el resto de tipos de datos se obtienen mediante `getValue()` que devuelve un `java.lang.Object`, de modo que deberemos hacer el "cast" apropiado según se ha indicado en la tabla. Por ejemplo:

```
DataElement element = ...;
int type = element.getDataType();
if(type == DataElement.DATSEQ)
```

```
        Enumeration e = (Enumeration) element.getValue();
```

La diferencia entre un dato de tipo DATSEQ y uno de tipo DATALT es unicamente su significado. El primero representa una "secuencia", es decir, representa una colección de datos secuenciales. Sin embargo el tipo DATALT representa una colección de objetos "alternativos", es decir, una colección de alternativas entre las que el usuario debe elegir.

Veamos un ejemplo completo de una aplicación que accede a los atributos de un servicio. Antes de probar este ejemplo se debe ejecutar el servidor Ejemplo5.java

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import java.io.IOException;
import java.util.*;

public class Ejemplo4 extends MIDlet implements
        CommandListener, DiscoveryListener {

        public static final UUID SERVICIO_MUSEO = new UUID(0x1234);
        public static final int ATRIBUTO_LISTADO_OBRAS = 0x4321;

        public static final UUID[] SERVICIOS =
                new UUID[]{ SERVICIO_MUSEO };
        public static final int[] ATRIBUTOS =
                new int[]{ ATRIBUTO_LISTADO_OBRAS };

        private Command comenzar, cancelar;
        private List obras;
        private Form busqueda;
        private Vector busquedas;

        private DiscoveryAgent discoveryAgent;

        public void startApp() {
                busquedas = new Vector();

                obras = new List("Ejemplo descubrimiento de servicios",
                        List.IMPLICIT);
                comenzar = new Command("Buscar", Command.ITEM, 1);
                cancelar = new Command("Cancelar", Command.ITEM, 1);

                busqueda = new Form("Busqueda de servicio");
                busqueda.append(new Gauge("Buscando servicio...",
                        false, Gauge.INDEFINITE,
                        Gauge.CONTINUOUS_RUNNING));
                busqueda.addCommand(cancelar);
                busqueda.setCommandListener(this);

                obras.addCommand(comenzar);
                obras.setCommandListener(this);

                LocalDevice localDevice = null;
                try {
                        localDevice = LocalDevice.getLocalDevice();
                        localDevice.setDiscoverable(DiscoveryAgent.GIAC);
                        discoveryAgent = localDevice.getDiscoveryAgent();
                        Display.getDisplay(this).setCurrent(obras);
                } catch(Exception e) {
                        e.printStackTrace();
                        Alert alert = new Alert("Error",
                                "No se puede hacer uso de Bluetooth",
                                null, AlertType.ERROR);
                        Display.getDisplay(this).setCurrent(alert);
                }
```

```
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
}

public void commandAction(Command c, Displayable s) {
        if (c == comenzar) {
                comenzar();
        } else if( c == cancelar) {
                cancelar();
        }
}

private void comenzar() {
        obras.deleteAll();
        try {
                discoveryAgent.startInquiry(
                        DiscoveryAgent.GIAC, this);
                Display.getDisplay(this).setCurrent(busqueda);
        } catch(BluetoothStateException e) {
                e.printStackTrace();
                Alert alert = new Alert("Error",
                        "No se pudo comenzar la busqueda",
                        null, AlertType.ERROR);
                Display.getDisplay(this).setCurrent(alert);
        }
}

private void cancelar() {
        discoveryAgent.cancelInquiry(this);

        Enumeration en = busquedas.elements();
        Integer i;
        while(en.hasMoreElements()) {
                i = (Integer) en.nextElement();
                discoveryAgent.cancelServiceSearch(i.intValue());
        }
}

//metodos de la interfaz DiscoveryListener
public void deviceDiscovered(RemoteDevice remoteDevice,
        DeviceClass deviceClass) {
        String address = remoteDevice.getBluetoothAddress();
        String friendlyName = null;
        try {
                friendlyName = remoteDevice.getFriendlyName(true);
        } catch(IOException e) { }


        String device = null;
        if(friendlyName == null) {
                device = address;
        } else {
                device = friendlyName + " ("+address+")";
        }

        try {
                int transId =
                        discoveryAgent.searchServices(ATRIBUTOS,
                        SERVICIOS, remoteDevice, this);
                System.out.println("Comenzada busqueda"+
                        " de serivicios en: "+device+"; "+transId);

                busquedas.addElement(new Integer(transId));
        } catch(BluetoothStateException e) {
                e.printStackTrace();
```

```
                        System.err.println("No se pudo "+
                                "comenzar la busqueda");
                }
        }

        public void inquiryCompleted(int discType) {
                switch(discType) {
                        case DiscoveryListener.INQUIRY_COMPLETED:
                                System.out.println("Busqueda "+
                                "de dispositivos concluida "+
                                "con normalidad");
                        break;
                        case DiscoveryListener.INQUIRY_TERMINATED:
                                System.out.println("Busqueda "+
                                "de dispositivos cancelada");
                        break;
                        case DiscoveryListener.INQUIRY_ERROR:
                                System.out.println("Busqueda"+
                                " de dispositivos finalizada "+
                                "debido a un error");
                        break;
                }
                //Display.getDisplay(this).setCurrent(obras);
        }

        public void servicesDiscovered(int transID,
                ServiceRecord[] servRecord) {
                ServiceRecord service = null;
                for(int i=0; i<servRecord.length; i++){
                        service = servRecord[i];
                        DataElement element = service.getAttributeValue(
                                ATRIBUTO_LISTADO_OBRAS);
                        Enumeration e = (Enumeration) element.getValue();

                        while(e.hasMoreElements()) {
                                element = (DataElement) e.nextElement();
                                obras.append(element.getValue().toString(), nul
                        }
                }
        }

        public void serviceSearchCompleted(int transID, int respCode) {
                System.out.println("Busqueda de servicios "+transID+
                        " completada");
                switch(respCode) {
                        case DiscoveryListener.SERVICE_SEARCH_COMPLETED:
                        System.out.println("Busqueda completada "+
                                "con normalidad");
                        break;
                        case DiscoveryListener.SERVICE_SEARCH_TERMINATED:
                        System.out.println("Busqueda cancelada");
                        break;
                        case DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHA
                        System.out.println("Dispositivo no alcanzable");
                        break;
                        case DiscoveryListener.SERVICE_SEARCH_NO_RECORDS:
                        System.out.println("No se encontraron registros"+
                                " de servicio");
                        break;
                        case DiscoveryListener.SERVICE_SEARCH_ERROR:
                        System.out.println("Error en la b#squeda");
                        break;
                }
        }
}
```

Comunicación

El API javax.bluetooth permite usar dos mecanismos de conexión: SPP y L2CAP. Mediante SPP obtendremos un InputStream y un OutputStream. Mediante L2CAP enviaremos y recibiremos arrays de bytes.

Para abrir cualquier tipo de conexión haremos uso de la clase `javax.microedition.io.Connector`. En concreto usaremos el método estático `open()` que está sobrecargado; su versión más sencilla requiere un parámetro que es un `String` que contendrá la URL con los datos necesarios para realizar la conexión. La URL será diferente dependiendo si queremos ser cliente o servidor de una conexión L2CAP o SPP.

Comunicación cliente

Obtendremos la URL necesaria para realizar la conexión a través del método `getConnectionURL()` de un objeto `ServiceRecord`. Recordemos que un objeto `ServiceRecord` representa un servicio, es decir una vez hayamos encontrado el servicio deseado (un objeto `ServiceRecord`), él mismo nos proveerá la URL necesaria para conectarnos a él.

Este método requiere dos argumentos, el primero de los cuales indica si se debe autenticar y/o cifrar la conexión. Los posibles valores de este primer argumento son:

- ServiceRecord.NOAUTHENTICATE_NOENCRYPT: No se requiere ni autenticación, ni cifrado.
- ServiceRecord.AUTHENTICATE_NOENCRYPT: Se requiere autenticación, pero no cifrado
- ServiceRecord.AUTHENTICATE_ENCRYPT: Se requiere tanto autenticación como cifrado.

El segundo argumento del método `getConnectionURL()` es un booleano que especifica si nuestro dispositivo debe hacer de maestro (`true`) o bien no importa si es maestro o esclavo (`false`).

```
ServiceRecord sr = ...;
String url = sr.getConnectionURL(ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
```

Comunicación cliente SPP

Una vez tenemos la URL con la información necesaria usaremos el método `Connector.open()` para realizar la conexión. Este método devuelve un objeto distinto según el tipo de protocolo usado. En el caso de un cliente SPP devolverá un `StreamConnection`. A partir del `StreamConnection` podremos obtener los flujos de entrada y de salida:

```
StreamConnection con = (StreamConnection) Connector.open(url);
OutputStream out = con.openOutputStream();
InputStream in = con.openInputStream();
```

o bien

```
StreamConnection con = (StreamConnection) Connector.open(url);
DataOutputStream out = con.openDataOutputStream();
DataInputStream in = con.openDataInputStream();
```

A partir de aquí ya podemos escribir y leer de la conexión. Veamos un ejemplo completo. Este ejemplo requiere que esté ejecutándose el servidor Ejemplo7.java.

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import javax.microedition.io.*;

public class Ejemplo6 extends MIDlet implements
        CommandListener, DiscoveryListener {

        public static final UUID SERVICIO_CHAT = new UUID(0x2345);

        public static final UUID[] SERVICIOS =
                new UUID[]{ SERVICIO_CHAT };
        public static final int[] ATRIBUTOS = null;
                //no necesitamos ningun atributo de servicio

        private Command comenzar, cancelar;
        private TextBox texto;
        private Form principal;
        private Form busqueda;
        private Vector busquedas;

        private DiscoveryAgent discoveryAgent;

        public void startApp() {
                busquedas = new Vector();
                principal = new Form("Ejemplo cliente SPP");

                texto = new TextBox("El servidor te dice...",
                        "", 50, TextField.UNEDITABLE);
                comenzar = new Command("Buscar", Command.ITEM, 1);
                cancelar = new Command("Cancelar", Command.ITEM, 1);

                busqueda = new Form("Busqueda de servicio");
                busqueda.append(new Gauge("Buscando servicio...",
                        false, Gauge.INDEFINITE,
                        Gauge.CONTINUOUS_RUNNING));
                busqueda.addCommand(cancelar);
                busqueda.setCommandListener(this);

                principal.addCommand(comenzar);
                principal.setCommandListener(this);

                LocalDevice localDevice = null;
                try {
                        localDevice = LocalDevice.getLocalDevice();
                        localDevice.setDiscoverable(DiscoveryAgent.GIAC);
                        discoveryAgent = localDevice.getDiscoveryAgent();
                        Display.getDisplay(this).setCurrent(principal);
                } catch(Exception e) {
                        e.printStackTrace();
                        Alert alert = new Alert("Error",
                                "No se puede hacer uso de Bluetooth",
                                null, AlertType.ERROR);
                        Display.getDisplay(this).setCurrent(alert);
                }
        }

        public void pauseApp() {
        }

        public void destroyApp(boolean unconditional) {
        }
```

```
public void commandAction(Command c, Displayable s) {
        if (c == comenzar) {
                comenzar();
        } else if( c == cancelar) {
                cancelar();
        }
}

private void comenzar() {
        try {
                discoveryAgent.startInquiry(
                        DiscoveryAgent.GIAC, this);
                Display.getDisplay(this).setCurrent(busqueda);
        } catch(BluetoothStateException e) {
                e.printStackTrace();
                Alert alert = new Alert("Error",
                        "No se pudo comenzar la busqueda",
                        null, AlertType.ERROR);
                Display.getDisplay(this).setCurrent(alert);
        }
}

private void cancelar() {
        discoveryAgent.cancelInquiry(this);

        Enumeration en = busquedas.elements();
        Integer i;
        while(en.hasMoreElements()) {
                i = (Integer) en.nextElement();
                discoveryAgent.cancelServiceSearch(i.intValue());
        }
}

//metodos de la interfaz DiscoveryListener
public void deviceDiscovered(RemoteDevice remoteDevice,
        DeviceClass deviceClass) {
        String address = remoteDevice.getBluetoothAddress();
        String friendlyName = null;
        try {
                friendlyName = remoteDevice.getFriendlyName(true);
        } catch(IOException e) { }


        String device = null;
        if(friendlyName == null) {
                device = address;
        } else {
                device = friendlyName + " ("+address+")";
        }

        try {
                int transId =
                        discoveryAgent.searchServices(ATRIBUTOS,
                        SERVICIOS, remoteDevice, this);
                System.out.println("Comenzada busqueda"+
                        " de serivicios en: "+device+"; "+transId);

                busquedas.addElement(new Integer(transId));
        } catch(BluetoothStateException e) {
                e.printStackTrace();
                System.err.println("No se pudo comenzar"+
                " la busqueda");
        }
}

public void inquiryCompleted(int discType) {
        switch(discType) {
                case DiscoveryListener.INQUIRY_COMPLETED:
```

```
                        System.out.println("Busqueda "+
                                "de dispositivos concluida "+
                                "con normalidad");
                break;
                case DiscoveryListener.INQUIRY_TERMINATED:
                        System.out.println("Busqueda de"+
                        "dispositivos cancelada");
                break;
                case DiscoveryListener.INQUIRY_ERROR:
                        System.out.println("Busqueda de"+
                        "dispositivos finalizada debido a "+
                        "un error");
                break;
        }
}

public void servicesDiscovered(int transID,
        ServiceRecord[] servRecord) {
        ServiceRecord service = null;
        for(int i=0; i<servRecord.length; i++){
                service = servRecord[i];

                String url =
                service.getConnectionURL(
                ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);

                StreamConnection connection = null;
                DataInputStream in = null;
                DataOutputStream out = null;

                try {
                        connection =
                        (StreamConnection) Connector.open(url);

                        in = connection.openDataInputStream();
                        out = connection.openDataOutputStream();

                        out.writeUTF("saludos desde el cliente!");
                        out.flush();

                        String s = in.readUTF();
                        texto.setString(s);
                        Display.getDisplay(this).setCurrent(texto);

                        cancelar(); //cancelamos las busquedas
                } catch(IOException e) {
                        e.printStackTrace();
                } finally {
                        try {
                                if(in != null)
                                        in.close();
                                if(out != null)
                                        out.close();
                                if(connection != null)
                                        connection.close();
                        } catch(IOException e) {}
                }
        }
}

public void serviceSearchCompleted(int transID, int respCode) {
        System.out.println("Busqueda de servicios "+
                transID+ " completada");
        switch(respCode) {
                case DiscoveryListener.SERVICE_SEARCH_COMPLETED:
                System.out.println("Busqueda completada "+
                        "con normalidad");
                break;
                case DiscoveryListener.SERVICE_SEARCH_TERMINATED:
```

```
                        System.out.println("Busqueda cancelada");
                        break;
                        case DiscoveryListener.SERVICE_SEARCH_DEVICE_NOT_REACHA
                        System.out.println("Dispositivo no alcanzable");
                        break;
                        case DiscoveryListener.SERVICE_SEARCH_NO_RECORDS:
                        System.out.println("No se encontraron registros"+
                                " de servicio");
                        break;
                        case DiscoveryListener.SERVICE_SEARCH_ERROR:
                        System.out.println("Error en la busqueda");
                        break;
                }
        }
}
```

Comunicación cliente L2CAP

A diferencia de las conexiones SPP, con L2CAP no nos comunicaremos a través de flujos de datos, sino a través de arrays de bytes.

En una conexión L2CAP al llamar al método `Connector.open()` se nos devolverá un objeto `L2CAPConnection`. Con este objeto podremos leer bytes a través del método `receive()` o enviar bytes mediante `send()`. Ambos métodos requieren como parámetro un array de bytes. En el caso de `send()` el parámetro serán los bytes a enviar, y en el caso de `receive()` el parámetro será un array en el que se guardarán los bytes leídos.

En una conexión L2CAP el tamaño de los arrays de bytes leídos y los recibidos están limitados a un tamaño máximo. Los tamaños límite se establecen mediante parámetros en la URL. Para saber el tamaño máximo de los arrays recibidos usaremos `getRecieveMTU()` y para saber el tamaño máximo de los arrays que se envían usaremos `getTransmitMTU()`

Para evitar pérdida de información el array de bytes pasado al método `receive()` deberá ser igual o mayor que `getRecieveMTU()`.

Comunicación del lado del servidor

Lo primero de todo, para ofrecer un servicio a través de Bluetooth necesitaremos poner nuestro dispositivo en modo visible. Esto se hace, recordemos, a través de la clase `LocalDevice`:

```
LocalDevice localdevice = LocalDevice.getLocalDevice();
if(!localdevice.setDiscoverable(DiscoveryAgent.GIAC)) {
        System.err.println("No se ha podido poner "+
                "en modo visible el dispositivo!");
}
```

Para crear una conexión servidora necesitaremos pasarle una URL al método `Connector.open()` del mismo modo como hemos hecho para realizar conexiones clientes. La diferencia está en que deberemos crearla nosotros y para indicar que queremos ser servidores indicaremos "localhost" como host en la URL. De este modo la URL deberá comenzar por "btspp://localhost:" o por "btl2cap://localhost:".

Además del host de la URL deberemos indicar el UUID que identifica el servicio. Posteriormente indicaremos el nombre del servicio y otros parámetros, como por ejemplo el requerimiento o no de autenticación.

Al pasar una URL de este tipo al método `Connector.open()`, éste nos devolverá un "no-

tifier", que en el caso de SPP será un objeto `StreamConnectionNotifier` y en el caso de L2CAP será un objeto `L2CAPConnectionNotifier`. Estos objetos nos permitirán escuchar conexiones entrantes de los clientes.

Veamos un ejemplo de apertura de una conexión servidora mediante SPP:

```
StringBuffer url = new StringBuffer("btspp://localhost:");
url.append((new UUID(0x12345)).toString());
url.append(";name=Mi servicio;authorize=true");

StreamConnectionNotifier notifier =
        (StreamConnectionNotifier) Connector.open(url.toString());
```

Ahora veamos un ejemplo de apertura de una conexión servidora mediante l2CAP:

```
StringBuffer url = new StringBuffer("btl2cap://localhost:");
url.append((new UUID(0x12345)).toString());
url.append(";name=Mi servicio;RecieveMTU=512;TransmitMTU=512");

L2CAPConnectionNotifier notifier =
        (L2CAPConnectionNotifier) Connector.open(url.toString());
```

Como se puede observar en los parámetros de la URL hemos especificado los tamaños máximos de los arrays de envío y recepción.

Una vez hemos creado el "notifier" que escuchará las conexiones clientes debemos especificar los atributos de nuestro servicio. Los atributos de nuestro servicio están almacenados en un `ServiceRecord`. El `ServiceRecord` se obtiene a través de la clase `LocalDevice` con el método `getRecord()`. Una vez tenemos el `ServiceRecord` podremos establecer sus atributos mediante el método `setAttributeValue()` al que le pasaremos un identificador numérico y un objeto `DataElement` que representará el valor del atributo de servicio.

Anteriormente hemos visto cómo utilizar objetos `DataElement`, pero no hemos visto cómo crearlos. Los constructores de la clase `DataElmenet` son cuatro.

Para crear un `DataElement` que almacene un booleano simplemente le pasaremos un valor booleano:

```
DataElement element = new DataElement(true);
```

Para crear un `DataElement` que almacene un valor entero le pasaremos el tipo de valor entero como primer argumento, y su valor como segundo argumento:

```
DataElement element = new DataElement(DataElemnet.U_INT_1, 123);
```

Para crear un DataElement que almacene una URL o un `String` usaremos el siguiente constructor:

```
DataElement element1 =
        new DataElement(DataElement.STRING, "Hola, esto es un String");
DataElement element2 =
        new DataElement(DataElement.URL, "http://esto.es.una.url");
```

Si lo que queremos crear es un `DataElement` que represente una colección de objetos `DataElement` usaremos el constructor:

```
DataElement element = new DataElement(DataElement.DATSEQ); // o bien DATALT
element.insertElementAt(new DataElement(...), 0);
element.insertElementAt(new DataElement(...), 1);
element.insertElementAt(new DataElement(...), 2);
```

Como vemos, para insertar elementos en una colección usaremos el método `insertElementAt()`. Para eliminar un elemento usaremos el método `removeElement()` al que le pasaremos el objeto que queremos eliminar y que devolverá true o false según la eliminación se ha llevado a cabo con éxito o no respectivamente. Para obtener el tamaño de una colección (DATSEQ o DATALT) usaremos el método `getSize()`.

Para crear un objeto `DataElement` que represente un valor nulo haremos lo siguiente:

```
DataElement element = new DataElement(DataElement.NULL);
```

Veamos un ejemplo de un servidor que establece atributos de servicio. Este ejemplo trabaja con el cliente Ejemplo4.java visto anteriormente.

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import javax.microedition.io.*;
import java.io.IOException;

public class Ejemplo5 extends MIDlet implements
        CommandListener, Runnable {

        public static final UUID SERVICIO_MUSEO = new UUID(0x1234);
        public static final int ATRIBUTO_LISTADO_OBRAS = 0x4321;

        private Command comenzar;
        private Form principal;

        private Thread thread;

        public void startApp() {
                thread = new Thread(this);

                comenzar = new Command("Comenzar", Command.ITEM, 1);

                principal = new Form("Ejemplo servidor Bluetooth");
                principal.addCommand(comenzar);
                principal.setCommandListener(this);

                Display.getDisplay(this).setCurrent(principal);
        }

        public void pauseApp() {
        }

        public void destroyApp(boolean unconditional) {
        }

        public void commandAction(Command c, Displayable s) {
                if (c == comenzar) {
                        thread.start();
                }
        }

        public void run() {
                try {
                        LocalDevice localDevice =
```

```
                        LocalDevice.getLocalDevice();
                        if(!localDevice.setDiscoverable(
                                DiscoveryAgent.GIAC)) {
                                Display.getDisplay(this).setCurrent(
                                        new Alert("Imposible"+
                                        " ofrecer un servicio"));
                        }

                        StringBuffer url =
                                new StringBuffer("btspp://localhost:");
                        url.append(SERVICIO_MUSEO.toString());
                        url.append(";name=Servicio museo;authorize=true");
                        StreamConnectionNotifier notifier =
                                (StreamConnectionNotifier)
                                Connector.open(url.toString());

                        ServiceRecord record =
                                localDevice.getRecord(notifier);
                        DataElement listado =
                                new DataElement(DataElement.DATSEQ);
                        listado.insertElementAt(
                                new DataElement(DataElement.STRING,
                                "Las hilanderas"), 0);
                        listado.insertElementAt(
                                new DataElement(DataElement.STRING,
                                "Las meninas"), 1);
                        listado.insertElementAt(
                                new DataElement(DataElement.STRING,
                                "La rendicion de Breda"), 2);
                        listado.insertElementAt(
                                new DataElement(DataElement.STRING,
                                "Gernica"), 3);

                        record.setAttributeValue(
                                ATRIBUTO_LISTADO_OBRAS, listado);

                        principal.append("Servidor en marcha...");
                        while(true) {
                                StreamConnection con =
                                notifier.acceptAndOpen();
                        }
                } catch(IOException e) {
                        e.printStackTrace();
                        Display.getDisplay(this).setCurrent(
                                new Alert("Error: "+e));
                }
        }
}
```

Llegados a este punto ya podemos escuchar conexiones clientes a través de `acceptAndOpen()`. En una conexión SPP lo haremos del siguiente modo:

```
StreamConnection conn = notifier.acceptAndOpen();
```

Y en una conexión L2CAP lo haremos del siguiente modo:

```
L2CAPConnection conn = notifier.acceptAndOpen();
```

Una vez que tenemos un objeto `StreamConnection` o `L2CAPConnection` ya podemos comunicarnos del mismo modo que hemos visto en las aplicaciones cliente.

Para terminar: un ejemplo de comunicación por parte del servidor. Este ejemplo trabaja con el cliente Ejemplo6.java

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;
import javax.microedition.io.*;
import java.io.*;

public class Ejemplo7 extends MIDlet implements
        CommandListener, Runnable {

        public static final UUID SERVICIO_CHAT = new UUID(0x2345);

        private Command comenzar;
        private Form principal;

        private Thread thread;

        public void startApp() {
                thread = new Thread(this);

                comenzar = new Command("Comenzar", Command.ITEM, 1);

                principal = new Form("Ejemplo servidor Bluetooth SPP");
                principal.addCommand(comenzar);
                principal.setCommandListener(this);

                Display.getDisplay(this).setCurrent(principal);
        }

        public void pauseApp() {
        }

        public void destroyApp(boolean unconditional) {
        }

        public void commandAction(Command c, Displayable s) {
                if (c == comenzar) {
                        thread.start();
                }
        }

        public void run() {
                try {
                        LocalDevice localDevice =
                                LocalDevice.getLocalDevice();
                        if(!localDevice.setDiscoverable(
                                DiscoveryAgent.GIAC)) {
                                Display.getDisplay(this).setCurrent(
                                        new Alert("Imposible ofrecer"+
                                        " un servicio"));
                        }

                        StringBuffer url =
                                new StringBuffer("btspp://localhost:");
                        url.append(SERVICIO_CHAT.toString());
                        url.append(";name=Servicio chat;authorize=true");
                        StreamConnectionNotifier notifier =
                                (StreamConnectionNotifier)
                                Connector.open(url.toString());


                        principal.append("Servidor en marcha...");

                        StreamConnection connection = null;
                        DataInputStream in = null;
```

```
                        DataOutputStream out = null;
                        while(true) {
                                try {
                                        connection =
                                                notifier.acceptAndOpen();
                                        in =
                                        connection.openDataInputStream();
                                        out =
                                        connection.openDataOutputStream();

                                        out.writeUTF("El servidor "+
                                                "dice HOLA!");
                                        out.flush();
                                        String s = in.readUTF();
                                        System.out.println("Un "+
                                                "cliente dice...");
                                        System.out.println(s);
                                } catch(IOException e) {
                                        e.printStackTrace();
                                } finally {
                                        try {
                                                if(in != null)
                                                        in.close();
                                                if(out != null)
                                                        out.close();
                                                if(connection != null)
                                                        connection.close();
                                        } catch(IOException e) {}
                                }
                        }
                } catch(IOException e) {
                        e.printStackTrace();
                        Display.getDisplay(this).setCurrent(
                                new Alert("Error: "+e));
                }
        }
}
```

Capítulo 3. El paquete javax.obex

Este paquete es totalmente independiente del paquete javax.bluetooth, es decir, cuando hagamos una aplicación utilizando el protocolo OBEX no usaremos ninguna de las clases del paquete javax.bluetooth.

OBEX es un protocolo que se usa generalmente para la transferencia de archivos. Es muy similar a HTTP: consiste en el intercambio de mensajes entre el cliente y el servidor. Tales mensajes consisten en un conjunto de cabeceras de mensaje y opcionalmente un cuerpo de mensaje. En este protocolo el cliente envía comandos al servidor (CONNECT, PUT, GET, DELETE, SETPATH, DISCONNECT) junto con algunas cabeceras de mensaje y en ocasiones (comando PUT) un cuerpo de mensaje. Las cabeceras de mensaje están encapsuladas en un objeto `HeaderSet` y el cuerpo de mensaje se lee/escribe mediante un Input/OutputStream.

El servidor por su parte recibirá los comandos del cliente y responderá con un código de respuesta indicando el éxito o no de la petición. Enviará además una serie de cabeceras de mensaje con información adicional y un cuerpo de mensaje en caso de tratarse de una respuesta al comando GET.

El comando CONNECT es necesario para completar el inicio de la sesión. El comando PUT envía datos del cliente al servidor y el comando GET envía datos del servidor al cliente. El comando DELETE sirve para eliminar un recurso del servidor. El comando SETPATH sirve para crear directorios y navegar por ellos, y finalmente el comando DISCONNECT sirve para cerrar la conexión.

Clases básicas

La clase `HeaderSet`

La clase `HeaderSet` representa las cabeceras de mensaje tanto de los mensajes enviados por un cliente como los enviados por un servidor.

Las cabeceras de mensaje se guardan como pares clave-valor en los que la clave es un número entero; es decir, las cabeceras de mensaje se identifican numéricamente. Estos identificadores numéricos serán utilizados en los métodos `setHeader()` y `getHeader()` para establecer y obtener respectivamente una cabecera de mensaje. También podemos obtener un array con todos los identificadores de las cabeceras que guarda un objeto `HeaderSet` mediante el método `getHeaderList()`.

Existen una serie de cabeceras de mensaje predefinidas. Como ejemplo, la cabecera "LENGTH" será muy utilizada, ya que informa de la longitud del cuerpo del mensaje. Otro ejemplo es la cabecera "NAME" que especifica el nombre del recurso (generalmente el nombre de un archivo). Los identificadores numéricos de estas cabeceras predefinidas se guardan como variables estáticas de la clase `HeaderSet`. Por ejemplo HeaderSet.LENGTH o HeaderSet.NAME.

El método `getHeader()` devuelve un objeto de diferente tipo según el tipo de cabecera. Por ejemplo para `getHeader(HeaderSet.LENGTH)` devuelve un `java.lang.Long`, y para `getHeader(HeaderSet.NAME)` deuvelve un objeto del tipo `java.lang.String`.

Tabla 3.1. Cabeceras y tipos de datos relacionados

Cabecera	Descripción	Tipo de dato en java
COUNT	Entero sin signo de 4 bytes	java.lang.Long
NAME	Texto Unicode	java.lang.String

Cabecera	Descripción	Tipo de dato en java
TYPE	Texto ASCII	java.lang.String
LENGTH	Entero sin signo de 4 bytes	java.lang.Long
TIME_ISO_8601	Texto ASCII de la forma YYYYMMDDTHHMMSS[Z] donde [Z] indica hora Zulu	java.util.Calendar
TIME_4_BYTE	Entero sin signo de 4 bytes	java.util.Calendar
DESCRIPTION	Texto Unicode	java.lang.String
TARGET	Secuencia de bytes	byte[]
HTTP	Secuencia de bytes	byte[]
WHO	Secuencia de bytes	byte[]
OBJECT_CLASS	Secuencia de bytes	byte[]
APPLICA-TION_PARAMETER	Secuencia de bytes	byte[]

También se pueden usar identificadores propios para establecer cabeceras definidas por el usuario. Sin embargo, habrá que tener en cuenta que no se podrá utilizar cualquier valor para identificarlas. Se deberán utilizar valores dentro de un rango específico según el tipo de dato que deberá almacenar la cabecera de mensaje. Estos rengos de valores están reflejados en la siguiente tabla:

Tabla 3.2. Rangos y tipos de datos asociados para identificadores creados por el usuario

Identificador	Rango decimal	Descripción	Tipo de dato en java
0x30 a 0x3F	48 a 63	Texto Unicode	java.lang.String
0x70 a 0x7F	112 a 127	Secuencia de bytes	byte[]
0xB0 a 0xBF	176 a 191	Un byte	java.lang.Byte
0xF0 a 0xFF	240 a 255	Entero de 4 bytes sin signo	java.lang.Long

Los mensajes enviados por el servidor tienen adicionalmente un código de respuesta que indica si tuvo éxito la petición o en caso de no tenerlo cuál fue el motivo. Este código es almacenado también en el objeto `HeaderSet`. El código de respuesta es un número entero que se obtiene a través del método `getResponseCode()`; los posibles valores que puede tomar están reflejados como variables estáticas de la clase `ResponseCodes`

La clase `Operation`

Comom se verá más adelante en los comandos GET y PUT necesitaremos enviar o recibir, además de las cabeceras, el cuerpo del mensaje. Pues bien, un objeto `Operation` encapsula las cabeceras y el cuerpo del mensaje. Las cabeceras de mensaje se guardan en un objeto `HeaderSet` que se obtiene mediante `getRecivedHeaders()`. Para enviar datos usaremos `OutputStream` que obtendremos con `openOutputStream()` o bien usaremos un `DataOutputStream` que obtendremos con `openDataOutputStream`. En caso de que estemos recibiendo datos usaremos un `InputStream` que obtendremos con `openInputStream()` o bien usaremos un `DataInputStream` que obtendremos con `openDataInputStream`.

Adicionalmente un cliente puede obtener el código de respuesta enviado por el servidor con `getResponseCode()`.

Conexión cliente

Una conexión cliente OBEX viene encapsulada en forma de un objeto `ClientSession`. Para obtener un objeto `ClientSession` usaremos el método `Connector.open()`. A este método le pasaremos una URL del tipo "irdaobex://discover.0210;ias=NombreServicio" Una vez obtenido el objeto `ClientSession` llamaremos al método `connect()`. Veamos un ejemplo:

```
String url = "irdaobex://discover.0210;ias=NombreServicio";
ClientSession session = (ClientSession) Connector.open(url);

HeaderSet response = session.connect(session.createHeaderSet());
int respnoseCode = response.getResponseCode();
if(responseCode != ResponseCodes.OBEX_HTTP_OK)
        System.err.println("No se pudo completar la conexion");
```

Nota: Como se aprecia en el ejemplo, se usa el método `createHeaderSet()` de la clase `ClientSession` para crear un objeto HeaderSet.

A partir de ahora podemos efectuar las operaciones DELETE, PUT, GET, y SETPATH a través de los métodos `delete()`, `put()`, `get()` y `setPath()` respectivamente. Todos estos métodos requieren un parámetro de tipo `HeaderSet`.

Los métodos `get()` y `put()` devuelven un objeto de tipo `Operation`, el cual encapsula un mensaje completo: código de respuesta, cabeceras de mensaje y cuerpo del mensaje. Cuando ejecutemos una llamada al método `get()` nos interesará leer el cuerpo del mensaje usando `openInputStream()` u `openDataInputStream()`; mientras que cuando ejecutemos una llamada al método `put()` nos interesará escribir en el cuerpo del mensaje usando `openOutpuStream()` u `openDataOutputStream()`.

Veamos un ejemplo de envío de un array de bytes a través del método `put()`.

```
ClientSession session = ...;
byte[] data = ....; //datos a enviar
HeaderSet headers = session.createHeaderSet();
headers.setHeader(HeaderSet.LENGHT, new Long(data.length);
Operation operation = session.put(headers);
DataOutputStream out = operation.openDataOutputStream();
out.write(imageData);
out.close();

int code = operation.getResponseCode();

if( code != ResponseCodes.OBEX_HTTP_OK)
        System.err.println("La operacion no concluyo con exito");
```

Ahora veamos un ejemplo de lectura de datos a través del método `get()`. Supongamos que queremos leer un archivo que se llama "imagen.jpg".

```
ClientSession session = ...;
HeaderSet headers = session.createHeaderSet();
headers.setHeader(headers.NAME, "imagen.jpg");
Operation operation = session.get(headers);
int length = operation.getLength();
byte[] data = new byte[length];
```

```
DataInputStream in = operation.openDataInputStream();
in.read(data);
in.close();

int code = operation.getResponseCode();

if( code != ResponseCodes.OBEX_HTTP_OK)
        System.err.println("La operacion no concluyo con exito");
```

Como se ha hecho en el ejemplo, para especificar el recurso que se desea obtener o eliminar estableceremos la cabecera HeaderSet.NAME indicando el nombre del recurso.

El método `setPath()` requiere adicionalmente dos argumentos de tipo booleano. El primer argumento si es `true` indica que se quiere navegar al directorio padre del directorio indicado por la cabecera HeaderSet.NAME (similar a hacer "cd .."). El segundo argumento indica si se debe crear o no el directorio en caso de que no exista.

Para finalizar la conexión llamaremos al método `disconnect()` que también requiere un argumento de tipo `HeaderSet`.

Conexión servidor

Una conexión servidora viene encapsulada en un objeto `SessionNotifier`. Para obtenerlo utilizaremos también el método `Connection.open()` pasándole una URL del tipo "irdaobex://localhost.0010;ias=NombreServicio". A través del objeto `SessionNotifier` podremos escuchar las conexiones cliente mediante `acceptAndOpen()`. El método `acceptAndOpen()` requiere que se le pase un parámetro de tipo `ServerRequestHandler`. Este objeto nos servirá como "listener". Cada vez que un cliente nos envíe un comando CONNECT, GET, PUT, DELETE o DISCONECT se llamará a los métodos `onConnect()`, `onGet()`, `onPut()`, `onDelete()` u `onDisconnect()` respectivamente. En su semántica son métodos muy similares a los métodos `doGet()` y `doPost()` de un servlet. A excepción de los métodos `onGet()` y `onPut()` el resto de estos métodos tienen dos argumentos de tipo HeaderSet. El primero representa las cabeceras que envió el cliente y el segundo representa las cabeceras que serán enviadas al cliente.

Los métodos `onGet()` y `onPut()` son especiales ya que requieren un cuerpo de mensaje. Por ello a estos métodos se les pasa un único argumento de tipo `Operation`. Como se ha descrito anteriormente, este objeto encapsula las cabeceras de mensaje y el cuerpo del mensaje.

Todos estos métodos a excepción de `onDisconnect()` deben devolver un código de respuesta representado por un valor entero que deberá ser uno de los valores definidos en la clase `ResponseCodes`.

Capítulo 4. Implementaciones de APIs Bluetooth para Java

Implementaciones del JSR-82

- El Java™ Wireless Toolkit 2.2 Beta [http://java.sun.com/products/j2mewtoolkit/download-2_2.html] se trata de un entorno de desarrollo de aplicaciones J2ME. Este entorno implementa una gran cantidad de JSRs, entre ellos el JSR-82. La implementación del JSR-82 del Java™ Wireless Toolkit es virtual, es decir, no usa hardware bluetooth, sino que lo emula. Para simular un entorno de dispositivos Bluetooth simplemente tendremos que ejecutar nuevas instancias del simulador de móviles, y cada instancia se podrá comunicar con las demás a través del API Bluetooth como si de un entorno real se tratara.

  El Java™ Wireless Toolkit 2.2 Beta es gratuito pero sólo está disponible por el momento para plataformas Windows.

- El Nokia Developer's Suite [http://www.forum.nokia.com/main/0,,034-2,00.html] se trata de un simulador de móviles que, al igual que el Java™ Wireless Toolkit permite simular un entorno Bluetooth sin necesidad de tener hardware Bluetooth. Está disponible para Windows y para Linux. Es gratuito.

- Impronto [http://www.rococosoft.com/products/impronto.html] es un simulador de un entorno Bluetooth. A través de su interfaz gráfica podemos añadir y configurar dispositivos virtuales. Podemos usarlo tanto para hacer aplicaciones J2ME como para hacer aplicaciones J2SE.

  Está disponible para Windows y Linux. Para Linux es gratuito para uso no comercial y universitario.

- Atinav [http://www.atinav.com/bluetooth/bt_javaj2se.htm] es quizá la implementación más popular para J2SE del JSR-82. Se trata de una implementación 100% Java.

- Avetana [http://www.avetana-gmbh.de/avetana-gmbh/jsr82.xml] es una implementación OpenSource del JSR-82 con la que podemos hacer uso de Bluetooth en nuestras aplicaciones J2SE.

  Está en desarrollo y pretende estar disponible para Windows, Mac OS X, y Linux. Por el momento sólo funciona en Linux.

- Blue Cove [http://sourceforge.net/projects/bluecove/] es una implementación OpenSource del JSR-82 que utiliza la implementación de Bluetooth de Windows XP SP2. En la página del proyecto afirman que desean soportar otros sistemas operativos en adelante.

- javaBluetooth [http://www.javabluetooth.org/] se trata de una implementación 100% Java™ Open Source que accede a los dispositivos Bluetooth a través del API javax.comm.

- Possio es un fabricante de dispositivos Bluetooth. Ofrece una implementación del JSR-82 [http://devzone.possio.com/Documents/Java_Bluetooth.html] para sus dispositivos.

APIs no-JSR-82

Además del API javax.bluetooth existen otras APIs para la programación de aplicaciones

Java™ que hagan uso de Bluetooth. A pesar de no tratarlas en este tutorial he creído interesante al menos mencionarlas.

- Harald [http://www.control.lth.se/~johane/harald/] es una implementación de la pila Bluetooth de reducido tamaño, 100% Java. Requiere la disponibilidad del paquete javax.comm.
- JBlueZ [http://jbluez.sourceforge.net/] Se trata de un API que accede a la pila Bluetooth de Linux (BlueZ). Es Open Source.

Capítulo 5. Documentación de interés

A continuación se lista una serie de enlaces relacionados con Java™ y Bluetooth.

- JSR-82 [http://jcp.org/en/jsr/detail?id=82]: Todo lo referente al JSR-82 en la web del JCP. Documentación javadoc, especificación y más.
- JavaBluetooth.com [http://www.javabluetooth.com/]: Página web del libro "Bluetooth for Java".
- Bluetooth.com [http://bluetooth.com/]: La página oficial de Bluetooth. En ella entre otras cosas podrás encontrar la especificación de Bluetooth, noticias y eventos relacionados.
- "Wireless Application Programming with J2ME and Bluetooth" La primera parte [http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth1/] trata sobre las características técnicas y la arquitectura de Bluetooth; muy, muy útil para hacerse una idea de los aspectos técnicos de esta tecnología y sus diferencias respecto a otras tecnologías inalámbricas. La segunda parte [http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth2/] trata sobre la arquitectura del API y las tareas más habituales con diversos ejemplos.
- "Getting Started with Java™ and Bluetooth" [http://today.java.net/pub/a/today/2004/07/27/bluetooth.html]: Artículo que trata muy brevemente las diferentes fases necesarias para establecer una conexión bluetooth.
- Benhui.net [http://benhui.net/bluetooth/]: Sección sobre Bluetooth de esta página sobre desarrollo para teléfonos móviles. Contiene sobre todo ejemplos de todo tipo.