Excepciones e hilos. Acceso a la red

Índice

Las excepciones e hilos son imprescindibles para las aplicaiones que se comunican por red y que al mismo tiempo deben responder a la interfaz gráfica de usuario. Ambos mecanismos están presentes tanto en Java como en JavaME. Sin embargo la librería de comunicación por red es un poco más reducida en JavaME, y es la que se estudia en esta sesión.

1. Excepciones

1.1. Introducción

Las excepciones son eventos que ocurren durante la ejecución de un programa y hacen que éste salga de su flujo normal de instrucciones. Este mecanismo permite tratar los errores de una forma elegante, ya que separa el código para el tratamiento de errores del código normal del programa. Se dice que una excepción es *lanzada* cuando se produce un error, y esta excepción puede ser *capturada* para tratar dicho error.

1.2. Tipos de excepciones

Tenemos diferentes tipos de excepciones dependiendo del tipo de error que representen. Todas ellas descienden de la clase `Throwable`, la cual tiene dos descendientes directos:

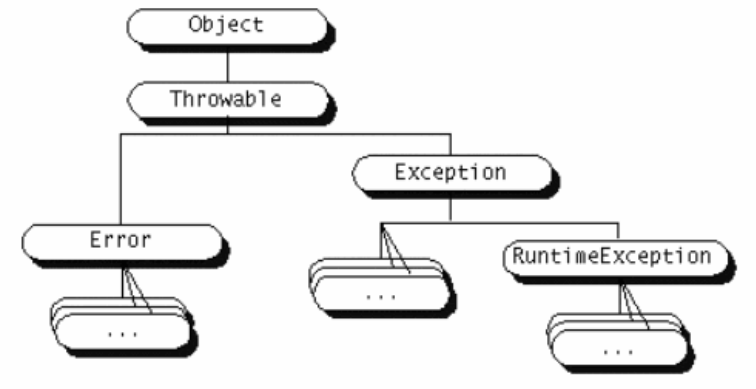
- **`Error`**: Se refiere a errores graves en la máquina virtual de Java, como por ejemplo fallos al enlazar con alguna librería. Normalmente en los programas Java no se tratarán este tipo de errores.
- **`Exception`**: Representa errores que no son críticos y por lo tanto pueden ser tratados y continuar la ejecución de la aplicación. La mayoría de los programas Java utilizan estas excepciones para el tratamiento de los errores que puedan ocurrir durante la ejecución del código.

Dentro de `Exception`, cabe destacar una subclase especial de excepciones denominada `RuntimeException`, de la cual derivarán todas aquellas excepciones referidas a los errores que comúnmente se pueden producir dentro de cualquier fragmento de código, como por ejemplo hacer una referencia a un puntero `null`, o acceder fuera de los límites de un *array*.

Estas `RuntimeException` se diferencian del resto de excepciones en que no son de tipo *checked*. Una excepción de tipo *checked* debe ser capturada o bien especificar que puede ser lanzada de forma obligatoria, y si no lo hacemos obtendremos un error de compilación. Dado que las `RuntimeException` pueden producirse en cualquier fragmento de código, sería impensable tener que añadir manejadores de excepciones y declarar que éstas pueden ser lanzadas en todo nuestro código. Deberemos:

- Utilizar excepciones *unchecked* (no predecibles) para indicar errores graves en la lógica del programa, que normalmente no deberían ocurrir. Se utilizarán para

comprobar la consistencia interna del programa.
- Utilizar excepciones *checked* para mostrar errores que pueden ocurrir durante la ejecución de la aplicación, normalmente debidos a factores externos como por ejemplo la lectura de un fichero con formato incorrecto, un fallo en la conexión, o la entrada de datos por parte del usuario.



Tipos de excepciones

Dentro de estos grupos principales de excepciones podremos encontrar tipos concretos de excepciones o bien otros grupos que a su vez pueden contener más subgrupos de excepciones, hasta llegar a tipos concretos de ellas. Cada tipo de excepción guardará información relativa al tipo de error al que se refiera, además de la información común a todas las excepciones. Por ejemplo, una `ParseException` se suele utilizar al procesar un fichero. Además de almacenar un mensaje de error, guardará la línea en la que el *parser* encontró el error.

1.3. Captura de excepciones

Cuando un fragmento de código sea susceptible de lanzar una excepción y queramos tratar el error producido o bien por ser una excepción de tipo *checked* debamos capturarla, podremos hacerlo mediante la estructura `try-catch-finally`, que consta de tres bloques de código:

- Bloque `try`: Contiene el código regular de nuestro programa que puede producir una excepción en caso de error.
- Bloque `catch`: Contiene el código con el que trataremos el error en caso de producirse.
- Bloque `finally`: Este bloque contiene el código que se ejecutará al final tanto si se ha producido una excepción como si no lo ha hecho. Este bloque se utiliza para, por ejemplo, cerrar algún fichero que haya podido ser abierto dentro del código regular del programa, de manera que nos aseguremos que tanto si se ha producido un error como si no este fichero se cierre. El bloque `finally` no es obligatorio ponerlo.

Para el bloque `catch` además deberemos especificar el tipo o grupo de excepciones que

tratamos en dicho bloque, pudiendo incluir varios bloques `catch`, cada uno de ellos para un tipo/grupo de excepciones distinto. La forma de hacer esto será la siguiente:

```
try {
                // Código regular del programa
                // Puede producir excepciones
        } catch(TipoDeExcepcion1 e1) {
                // Código que trata las excepciones de tipo
                // TipoDeExcepcion1 o subclases de ella.
                // Los datos sobre la excepción los encontraremos
                // en el objeto e1.
        } catch(TipoDeExcepcion2 e2) {
                // Código que trata las excepciones de tipo
                // TipoDeExcepcion2 o subclases de ella.
                // Los datos sobre la excepción los encontraremos
                // en el objeto e2.
        ...
        } catch(TipoDeExcepcionN eN) {
                // Código que trata las excepciones de tipo
                // TipoDeExcepcionN o subclases de ella.
                // Los datos sobre la excepción los encontraremos
                // en el objeto eN.
        } finally {
                // Código de finalización (opcional)
        }
```

Si como tipo de excepción especificamos un grupo de excepciones este bloque se encargará de la captura de todos los subtipos de excepciones de este grupo. Por lo tanto, si especificamos `Exception` capturaremos cualquier excepción, ya que está es la superclase común de todas las excepciones.

En el bloque `catch` pueden ser útiles algunos métodos de la excepción (que podemos ver en la API de la clase padre `Exception`):

```
String getMessage()
        void printStackTrace()
```

con `getMessage` obtenemos una cadena descriptiva del error (si la hay). Con `printStackTrace` se muestra por la salida estándar la traza de errores que se han producido (en ocasiones la traza es muy larga y no puede seguirse toda en pantalla con algunos sistemas operativos).

Un ejemplo de uso:

```
try {
                ... // Aqui va el codigo que puede lanzar una excepcion
        } catch (Exception e) {
                System.out.println ("El error es: " + e.getMessage());
                e.printStackTrace();
        }
```

Nunca deberemos dejar vacío el cuerpo del `catch`, porque si se produce el error, nadie se va a dar cuenta de que se ha producido. En especial, cuando estemos con excepciones *no-checked*.

1.4. Lanzamiento de excepciones

Hemos visto cómo capturar excepciones que se produzcan en el código, pero en lugar de capturarlas también podemos hacer que se propaguen al método de nivel superior (desde el cual se ha llamado al método actual). Para esto, en el método donde se vaya a lanzar la excepción, se siguen 2 pasos:

- Indicar en el método que determinados tipos de excepciones o grupos de ellas pueden ser lanzados, cosa que haremos de la siguiente forma, por ejemplo:

```
public void lee_fichero()
                throws IOException, FileNotFoundException
        {
                // Cuerpo de la función
        }
```

Podremos indicar tantos tipos de excepciones como queramos en la claúsula `throws`. Si alguna de estas clases de excepciones tiene subclases, también se considerará que puede lanzar todas estas subclases.

- Para lanzar la excepción utilizamos la instrucción `throw`, proporcionándole un objeto correspondiente al tipo de excepción que deseamos lanzar. Por ejemplo:

```
throw new IOException(mensaje_error);
```

- Juntando estos dos pasos:

```
public void lee_fichero()
                throws IOException, FileNotFoundException
        {
                ...
                throw new IOException(mensaje_error);
                ...
        }
```

Podremos lanzar así excepciones en nuestras funciones para indicar que algo no es como debiera ser a las funciones llamadoras. Por ejemplo, si estamos procesando un fichero que debe tener un determinado formato, sería buena idea lanzar excepciones de tipo `ParseException` en caso de que la sintaxis del fichero de entrada no sea correcta.

NOTA: para las excepciones que no son de tipo *checked* no hará falta la cláusula *throws* en la declaración del método, pero seguirán el mismo comportamiento que el resto, si no son capturadas pasarán al método de nivel superior, y seguirán así hasta llegar a la función principal, momento en el que si no se captura provocará la salida de nuestro programa mostrando el error correspondiente.

1.5. Creación de nuevas excepciones

Además de utilizar los tipos de excepciones contenidos en la distribución de Java, podremos crear nuevos tipos que se adapten a nuestros problemas.

Para crear un nuevo tipo de excepciones simplemente deberemos crear una clase que herede de `Exception` o cualquier otro subgrupo de excepciones existente. En esta clase podremos añadir métodos y propiedades para almacenar información relativa a nuestro

tipo de error. Por ejemplo:

```
public class MiExcepcion extends Exception
        {
                public MiExcepcion (String mensaje)
                {
                        super(mensaje);
                }
        }
```

Además podremos crear subclases de nuestro nuevo tipo de excepción, creando de esta forma grupos de excepciones. Para utilizar estas excepciones (capturarlas y/o lanzarlas) hacemos lo mismo que lo explicado antes para las excepciones que se tienen definidas en Java.

1.6. Nested exceptions

Cuando dentro de un método de una librería se produce una excepción, normalmente se propagará dicha excepción al llamador en lugar de gestionar el error dentro de la librería, para que de esta forma el llamador tenga constancia de que se ha producido un determinado error y pueda tomar las medidas que crea oportunas en cada momento. Para pasar esta excepción al nivel superior puede optar por propagar la misma excepción que le ha llegado, o bien crear y lanzar una nueva excepción. En este segundo caso la nueva excepción deberá contener la excepción anterior, ya que de no ser así perderíamos la información sobre la causa que ha producido el error dentro de la librería, que podría sernos de utilidad para depurar la aplicación. Para hacer esto deberemos proporcionar la excepción que ha causado el error como parámetro del constructor de nuestra nueva excepción:

```
public class MiExcepcion extends Exception
        {
                public MiExcepcion (String mensaje, Throwable causa)
                {
                        super(mensaje, causa);
                }
        }
```

En el método de nuestra librería en el que se produzca el error deberemos capturar la excepción que ha causado el error y lanzar nuestra propia excepción al llamador:

```
try {
            ...
        } catch(IOException e) {
            throw new MiExcepcion("Mensaje de error", e);
        }
```

Cuando capturemos una excepción, podemos consultar la excepción previa que la ha causado (si existe) con el método:

```
Exception causa = (Exception)e.getCause();
```

Las *nested exceptions* son útiles para:

- Encadenar errores producidos en la secuencia de métodos a los que se ha llamado.

- Facilitan la depuración de la aplicación, ya que nos permite conocer de dónde viene el error y por qué métodos ha pasado.
- El lanzar una excepción propia de cada método permite ofrecer información más detallada que si utilizásemos una única excepción genérica. Por ejemplo, aunque en varios casos el origen del error puede ser una `IOException`, nos será de utilidad saber si ésta se ha producido al guardar un fichero de datos, al guardar datos de la configuración de la aplicación, al intentar obtener datos de la red, etc.
- Aislar al llamador de la implementación concreta de una librería. Por ejemplo, cuando utilicemos los objetos de acceso a datos de nuestra aplicación, en caso de error recibiremos una excepción propia de nuestra capa de acceso a datos, en lugar de una excepción propia de la implementación concreta de esta capa, como pudiera ser `SQLException` si estamos utilizando una BD SQL o `IOException` si estamos accediendo a ficheros.

2. Hilos

Un hilo es un flujo de control dentro de un programa. Creando varios hilos podremos realizar varias tareas simultáneamente. Cada hilo tendrá sólo un contexto de ejecución (contador de programa, pila de ejecución). Es decir, a diferencia de los procesos UNIX, no tienen su propio espacio de memoria sino que acceden todos al mismo espacio de memoria común, por lo que será importante su sincronización cuando tengamos varios hilos accediendo a los mismos objetos.

2.1. Creación de hilos

En Java los hilos están encapsulados en la clase **Thread**. Para crear un hilo tenemos dos posibilidades:

- Heredar de **Thread** redefiniendo el método *run()*.
- Crear una clase que implemente la interfaz **Runnable** que nos obliga a definir el método *run()*.

En ambos casos debemos definir un método *run()* que será el que contenga el código del hilo. Desde dentro de este método podremos llamar a cualquier otro método de cualquier objeto, pero este método *run()* será el método que se invoque cuando iniciemos la ejecución de un hilo. El hilo terminará su ejecución cuando termine de ejecutarse este método *run()*.

Para crear nuestro hilo mediante herencia haremos lo siguiente:

```
public class EjemploHilo extends Thread
                {
                        public void run()
                        {
                                // Código del hilo
                        }
                }
```

Una vez definida la clase de nuestro hilo deberemos instanciarlo y ejecutarlo de la siguiente forma:

```
Thread t = new EjemploHilo();
                t.start();
```

Al llamar al método *start* del hilo, comenzará ejecutarse su método *run*. Crear un hilo heredando de **Thread** tiene el problema de que al no haber herencia múltiple en Java, si heredamos de **Thread** no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

Este problema desaparece si utilizamos la interfaz **Runnable** para crear el hilo, ya que una clase puede implementar varios interfaces. Definiremos la clase que contenga el hilo como se muestra a continuación:

```
public class EjemploHilo implements Runnable
                {
                        public void run()
                        {
                                // Código del hilo
                        }
                }
```

Para instanciar y ejecutar un hilo de este tipo deberemos hacer lo siguiente:

```
Thread t = new Thread(new EjemploHilo());
                t.start();
```

Esto es así debido a que en este caso **EjemploHilo** no deriva de una clase **Thread**, por lo que no se puede considerar un hilo, lo único que estamos haciendo implementando la interfaz es asegurar que vamos a tener definido el método *run()*. Con esto lo que haremos será proporcionar esta clase al constructor de la clase **Thread**, para que el objeto **Thread** que creemos llame al método *run()* de la clase que hemos definido al iniciarse la ejecución del hilo, ya que implementando la interfaz le aseguramos que esta función existe.

2.2. Estado y propiedades de los hilos

Un hilo pasará por varios estados durante su ciclo de vida.

```
Thread t = new Thread(this);
```

Una vez se ha instanciado el objeto del hilo, diremos que está en estado de *Nuevo hilo*.

```
t.start();
```

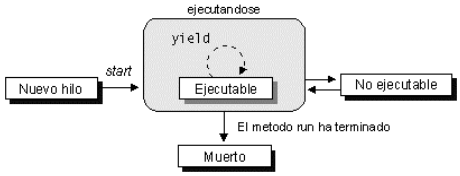
Cuando invoquemos su método *start()* el hilo pasará a ser un hilo *vivo*, comenzándose a ejecutar su método *run()*. Una vez haya salido de este método pasará a ser un hilo *muerto*.

La única forma de parar un hilo es hacer que salga del método *run()* de forma natural. Podremos conseguir esto haciendo que se cumpla una condición de salida de *run()* (lógicamente, la condición que se nos ocurra dependerá del tipo de programa que estemos

haciendo). Las funciones para parar, pausar y reanudar hilos están desaprobadas en las versiones actuales de Java.

Mientras el hilo esté *vivo*, podrá encontrarse en dos estados: *Ejecutable* y *No ejecutable*. El hilo pasará de *Ejecutable* a *No ejecutable* en los siguientes casos:

- Cuando se encuentre dormido por haberse llamado al método *sleep()*, permanecerá *No ejecutable* hasta haber transcurrido el número de milisegundos especificados.
- Cuando se encuentre bloqueado en una llamada al método *wait()* esperando que otro hilo lo desbloquee llamando a *notify()* o *notifyAll()*. Veremos cómo utilizar estos métodos más adelante.
- Cuando se encuentre bloqueado en una petición de E/S, hasta que se complete la operación de E/S.



Ciclo de vida de los hilos

Lo único que podremos saber es si un hilo se encuentra vivo o no, llamando a su método *isAlive()*.

Prioridades de los hilos

Además, una propiedad importante de los hilos será su prioridad. Mientras el hilo se encuentre vivo, el *scheduler* de la máquina virtual Java le asignará o lo sacará de la CPU, coordinando así el uso de la CPU por parte de todos los hilos activos basándose en su prioridad. Se puede forzar la salida de un hilo de la CPU llamando a su método *yield()*. También se sacará un hilo de la CPU cuando un hilo de mayor prioridad se haga *Ejecutable*, o cuando el tiempo que se le haya asignado expire.

Para cambiar la prioridad de un hilo se utiliza el método *setPriority()*, al que deberemos proporcionar un valor de prioridad entre *MIN_PRIORITY* y *MAX_PRIORITY* (tenéis constantes de prioridad disponibles dentro de la clase *Thread*, consultad el API de Java para ver qué valores de constantes hay).

Hilo actual

En cualquier parte de nuestro código Java podemos llamar al método *currentThread* de la clase *Thread*, que nos devuelve un objeto hilo con el hilo que se encuentra actualmente ejecutando el código donde está introducido ese método. Por ejemplo, si tenemos un código como:

```
public class EjemploHilo implements Runnable
                {
                        public EjemploHilo()
                        {
                                ...
```

```
                                int i = 0;
                                Thread t = Thread.currentThread();
                                t.sleep(1000);
                        }
                }
```

La llamada a *currentThread* dentro del constructor de la clase nos devolverá el hilo que corresponde con el programa principal (puesto que no hemos creado ningún otro hilo, y si lo creáramos, no ejecutaría nada que no estuviese dentro de un método *run.*

Sin embargo, en este otro caso:

```
public class EjemploHilo implements Runnable
                {
                        public EjemploHilo()
                        {
                                Thread t1 = new Thread(this);
                                Thread t2 = new Thread(this);
                                t1.start();
                                t2.start();
                        }

                        public void run()
                        {
                                int i = 0;
                                Thread t = Thread.currentThread();
                                t.sleep(1000);
                        }
                }
```

Lo que hacemos es crear dos hilos auxiliares, y la llamada a *currentThread* se produce dentro del *run*, con lo que se aplica a los hilos auxiliares, que son los que ejecutan el *run:* primero devolverá un hilo auxiliar (el que primero entre, t1 o t2), y luego el otro (t2 o t1).

Dormir hilos

Como hemos visto en los ejemplos anteriores, una vez obtenemos el hilo que queremos, el método *sleep* nos sirve para dormirlo, durante los milisegundos que le pasemos como parámetro (en los casos anteriores, dormían durante 1 segundo). El tiempo que duerme el hilo, deja libre el procesador para que lo ocupen otros hilos. Es una forma de no sobrecargar mucho de trabajo a la CPU con muchos hilos intentando entrar sin descanso.

2.3. Sincronización de hilos

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

Un primer mecanismo de comunicación es la variable cerrojo incluida en todo objeto **Object**, que permitirá evitar que más de un hilo entre en la sección crítica para un objeto determinado. Los métodos declarados como *synchronized* utilizan el cerrojo del objeto al que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

```
public synchronized void seccion_critica()
                { // Código sección crítica }
```

Todos los métodos *synchronized* de un mismo objeto (no clase, sino objeto de esa clase), comparten el mismo cerrojo, y es distinto al cerrojo de otros objetos (de la misma clase, o de otras).

También podemos utilizar cualquier otro objeto para la sincronización dentro de nuestro método de la siguiente forma:

```
synchronized (objeto_con_cerrojo)
                { // Código sección crítica }
```

de esta forma sincronizaríamos el código que escribiésemos dentro, con el código *synchronized* del objeto *objeto_con_cerrojo*.

Además podemos hacer que un hilo quede bloqueado a la espera de que otro hilo lo desbloquee cuando suceda un determinado evento. Para bloquear un hilo usaremos la función *wait()*, para lo cual el hilo que llama a esta función debe estar en posesión del monitor, cosa que ocurre dentro de un método *synchronized*, por lo que sólo podremos bloquear a un proceso dentro de estos métodos.

Para desbloquear a los hilos que haya bloqueados se utilizará *notifyAll()*, o bien *notify()* para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor.

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la sección crítica del objeto y desbloquearlo.

Por último, puede ser necesario esperar a que un determinado hilo haya finalizado su tarea para continuar. Esto lo podremos hacer llamando al método *join()* de dicho hilo, que nos bloqueará hasta que el hilo haya finalizado.

3. Conexiones de red

En J2SE tenemos una gran cantidad de clases en el paquete `java.net` para permitir establecer distintos tipos de conexiones en red. Sin embargo, el soportar esta gran API no es viable en la configuración CLDC dedicada a dispositivos muy limitados. Por lo tanto en CLDC se sustituye esta API por el marco de conexiones genéricas (GCF, Generic Connection Framework), con el que se pretenden cubrir todas las necesidades de conectividad de estos dispositivos a través de una API sencilla.

3.1. Marco de conexiones genéricas

Los distintos dispositivos móviles pueden utilizar distintos tipos de redes para conectarse. Algunos utilizan redes de conmutación de circuitos, orientadas a conexión, que necesitarán protocolos como TCP. Otros utilizan redes de transmisión de paquetes en las que no se establece una conexión permanente, y con las que deberemos trabajar con protocolos como por ejemplo UDP. Incluso otros dispositivos podrían utilizar otras redes

distintas en las que debamos utilizar otro tipo de protocolos.

El marco de conexiones genéricas (GFC) hará que esta red móvil subyacente sea transparente para el usuario, proporcionando a éste protocolos estándar de comunicaciones. La API de GFC se encuentra en el paquete `javax.microedition.io`. Esta API utilizará un único método que nos servirá para establecer cualquier tipo de conexión que queramos, por esta razón recibe el nombre de marco de conexiones genéricas, lo cuál además lo hace extensible para incorporar nuevos tipos de conexiones. Para crear la conexión utilizaremos el siguiente método:

```
Connection con = Connector.open(url);
```

En el que deberemos especificar una URL como parámetro con el siguiente formato:

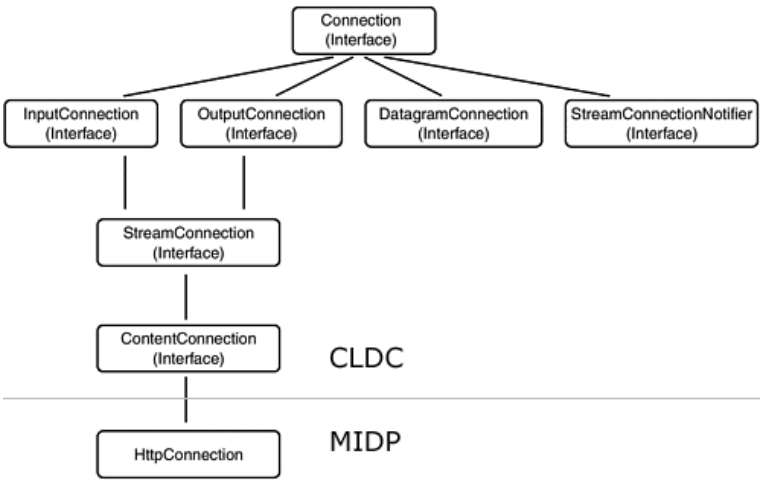
`protocolo:direccion;parámetros`

Cambiando el protocolo podremos especificar distintos tipos de conexiones. Por ejemplo, podríamos utilizar las siguientes URLs:

`"http://jtech.ua.es/pdm"`	Abre una conexión HTTP.
`"datagram://192.168.0.4:6666"`	Abre una conexión por datagramas.
`"socket://192.168.0.4:4444"`	Abre una conexión por *sockets*.
`"comm:0;baudrate=9600"`	Abre una conexión a través de un puerto de comunicaciones.
`"file:/fichero.txt"`	Abre un fichero.

Cuando especifiquemos uno de estos protocolos, la clase `Connector` buscará en tiempo de ejecución la clase que implemente dicho tipo de conexión, y si la encuentra nos devolverá un objeto que implemente la interfaz `Connection` que nos permitirá comunicarnos a través de dicha conexión.

CLDC nos proporciona interfaces para cada tipo genérico de conexión, pero las implementaciones reales de los protocolos pertenecen a los perfiles.

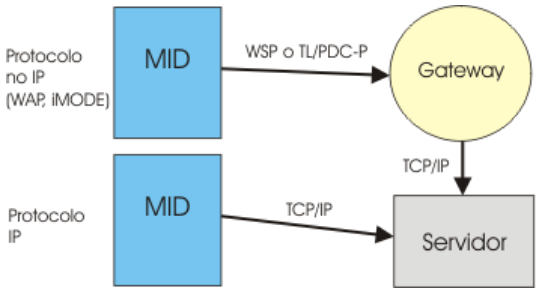


Componentes de GCF

El único protocolo que la especificación de MIDP exige que se implemente es el protocolo HTTP. Este protocolo pertenece a MIDP, y no a CLDC como era el caso de las clases genéricas anteriores. Distintos modelos de dispositivos pueden soportar otro tipo de conexiones, pero si queremos hacer aplicaciones portables deberemos utilizar HTTP.

3.2. Conexión HTTP

La conexión mediante el protocolo HTTP es el único tipo de conexión que sabemos que va a estar soportado por todos los dispositivos MIDP. Este protocolo podrá ser implementado en cada modelo de móvil bien utilizando protocolos IP como TCP/IP o bien protocolos no IP como WAP o i-Mode.



Gateway para protocolos no IP

De esta forma nosotros podremos utilizar directamente HTTP de una forma estándar sin importarnos el tipo de red que el móvil tenga por debajo.

Cuando establezcamos una conexión mediante protocolo HTTP, podemos hacer una conversión *cast* del objeto `Connection` devuelto a un subtipo `HttpConnection` especializado en conexiones HTTP:

```
        HttpConnection con =
            (HttpConnection)Connector.open("http://jtech.ua.es/datos.txt");
```

Este objeto `HttpConnection` contiene gran cantidad de métodos dedicados a trabajar con el protocolo HTTP, lo cuál facilitará en gran medida el trabajo de los desarrolladores.

HTTP es un protocolo de petición/respuesta. El cliente crea un mensaje de petición y lo envía a una determinada URL. El servidor analizará esta petición y le devolverá una respuesta al cliente. Estos mensajes de petición y respuesta se compondrán de una serie de cabeceras y del bloque de contenido. Cada cabecera tendrá un nombre y un valor. El contenido podrá contener cualquier tipo de información (texto, HTML, imágenes, mensajes codificados en binario, etc). Tendremos una serie de cabeceras estándar con las que podremos intercambiar datos sobre el cliente o el servidor, o bien sobre la información que estamos transmitiendo. También podremos añadir nuestras propias cabeceras para intercambiar datos propios.

Una vez creada la conexión, ésta pasará por tres estados:

- **Configuración**: No se ha establecido la conexión, todavía no se ha enviado el mensaje de petición. Este será el momento en el que deberemos añadir la información necesaria a las cabeceras del mensaje de petición.
- **Conectada**: El mensaje de petición ya se ha enviado, y se espera recibir una respuesta. En este momento podremos leer las cabeceras o el contenido de la respuesta.
- **Cerrada**: La conexión se ha cerrado y ya no podemos hacer nada con ella.

La conexión nada más crearse se encuentra en estado de configuración. Pasará automáticamente a estado conectada cuando solicitemos cualquier información sobre la respuesta.

3.2.1. Lectura de la respuesta

Vamos a comenzar viendo cómo leer el contenido de una URL. En este caso no vamos a añadir ninguna información al mensaje de petición, ya que no es necesario. Sólo queremos obtener el contenido del recurso solicitado en la URL.

Imaginemos que queremos leer el fichero en la URL `http://jtech.ua.es/datos.txt`. Como primer paso deberemos crear una conexión con dicha URL como hemos visto anteriormente. Una vez tengamos este objeto `HttpConnection` abriremos un flujo de entrada para leer su contenido de la siguiente forma:

```
        InputStream in = con.openInputStream();
```

Una vez hecho esto, la conexión pasará a estado conectada, ya que estamos solicitando leer su contenido. Por lo tanto en este momento será cuando envíe el mensaje de petición al servidor, y se quede esperando a recibir la respuesta. Con el flujo de datos obtenido

podremos leer el contenido de la misma, al igual que leemos cualquier otro flujo de datos en Java.

Dado que en este momento ya se ha enviado el mensaje de petición, ya no tendrá sentido realizar modificaciones en la petición. Es por esta razón por lo que la creación del mensaje de petición debe hacerse en el estado de configuración.

Una vez hayamos terminado de leer la respuesta, deberemos cerrar el flujo y la conexión:

```
        in.close();
        con.close();
```

Con esto la conexión pasará a estado cerrada, liberando todos los recursos.

3.2.2. Mensaje de petición

En muchos casos podemos necesitar enviar información al servidor, como por ejemplo el login y el password del usuario para autentificarse en la aplicación web. Esta información deberemos incluirla en el mensaje de petición. Existen distintas formas de enviar información en la petición.

Encontramos los diferentes tipos de mensajes de petición soportados por MIDP:

`HttpConnection.GET`	Los parámetros que se envían al servidor se incluyen en la misma URL. Por ejemplo, podemos mandar un parámetro `login` en la petición de la siguiente forma: `http://www.jtech.ua.es/dadm?login=miguel`
`HttpConnection.POST`	Los parámetros que se envían al servidor se incluyen como contenido del mensaje. Tiene la ventaja de que se puede enviar la cantidad de datos que queramos, a diferencia del método GET en el que esta cantidad puede estar limitada. Además los datos no serán visibles en la misma URL, ya que se incluyen como contenido del mensaje.
`HttpConnection.HEAD`	No se solicita el contenido del recurso al servidor, sólo información sobre éste, es decir, las cabeceras HTTP.

Podemos establecer uno de estos tipos utilizando el método `setRequestMethod`, por ejemplo para utilizar una petición POST haremos lo siguiente:

```
        con.setRequestMethod(HttpConnection.POST);
```

Además podremos añadir cabeceras a la petición con el siguiente método:

```
        con.setRequestProperty(nombre, valor);
```

Por ejemplo, podemos mandar las siguiente cabeceras:

```
        c.setRequestProperty("IF-Modified-Since",
            "22 Sep 2002 08:00:00 GMT");
        c.setRequestProperty("User-Agent",
            "Profile/MIDP-1.0 Configuration/CLDC-1.0");
        c.setRequestProperty("Content-Language", "es-ES");
```

Con esto estaremos diciendo al servidor que queremos que nos devuelva una respuesta sólo si ha sido modificada desde la fecha indicada, y además le estamos comunicando datos sobre el cliente. Indicamos mediante estas cabeceras estándar que el cliente es una aplicación MIDP, y que el lenguaje es español de España.

3.2.3. Envío de datos en la petición

Cuando necesitemos enviar datos al servidor mediante HTTP mediante nuestra aplicación Java, podemos simular el envío de datos que realiza un formulario HTML. Podremos simular tanto el comportamiento de un formulario que utilice método GET como uno que utilice método POST.

En el caso del método GET, simplemente utilizaremos una petición de tipo `HttpConnection.GET` e incluiremos estos datos codificados en la URL. Por ejemplo, si estamos registrando los datos de un usuario (nombre, apellidos y edad) podemos incluir estos parámetros en la URL de la siguiente forma:

```
        HttpConnection con =
            (HttpConnection)Connector.open("http://www.jtech.ua.es/aplic"
+
"/registraUsuario?nombre=Pedro&apellidos=Lopez+Garcia&edad=25");
```

Cada parámetro tiene la forma `nombre=valor`, pudiendo incluir varios parámetros separados por el carácter `'&'`. Como en la URL no puede haber espacios, estos caracteres se sustituyen por el carácter `'+'` como podemos ver en el ejemplo.

En el caso de que queramos simular un formulario con método POST, utilizamos una petición de tipo `HttpConnection.POST` y deberemos incluir los parámetros que enviemos al servidor como contenido del mensaje. Para ello deberemos indicar que el tipo de contenido de la petición es `application/x-www-form-urlencoded`, y como contenido codificaremos los parámetros de la misma forma que se utiliza para codificarlos en la URL cuando se hace una petición GET:

```
        nombre=Pedro&apellidos=Lopez+Garcia&edad=25
```

De esta forma podemos enviar al servidor datos en forma de una serie de parámetros que toman como valor cadenas de texto. Sin embargo, puede que necesitemos intercambiar datos más complejos con el servidor. Por ejemplo, podemos querer serializar objetos Java y enviarlos al servidor, o enviar documentos XML.

Para enviar estos tipos de información podemos utilizar también el bloque de contenido, debiendo especificar en cada caso el tipo MIME del contenido que vamos a añadir. Ejemplos de tipos MIME que podemos utilizar para el bloque de contenido son:

`application/x-www-form-urlencoded`	Se envían los datos codificados de la misma forma en la que son codificados por un formulario HTML con método POST.
`text/plain`	Se envía como contenido texto ASCII.
`application/octet-stream`	Se envía como contenido datos binarios. Dentro de la secuencia de bytes podremos codificar la información como queramos. Por ejemplo, podemos codificar de forma binaria un objeto serializado, utilizando un `DataOutputStream`.

Para establecer el tipo de contenido la cabecera estándar de HTTP `Content-Type`. Por ejemplo, si añadimos texto ASCII, podemos establecer esta cabecera de la siguiente forma:

```
con.setRequestProperty("Content-Type", "text/plain");
```

Para escribir en el contenido del mensaje de petición deberemos abrir un flujo de salida como se muestra a continuación:

```
OutputStream out = con.openOutputStream();
```

Podremos escribir en este flujo de salida igual que lo hacemos en cualquier otro flujo de salida, con lo que de esta forma podremos escribir cualquier contenido en el mensaje de petición.

Al abrir el flujo para escribir en la petición provocaremos que se pase a estado conectado. Por lo tanto deberemos haber establecido el tipo de petición y todas las cabeceras previamente a la apertura de este flujo, cuando todavía estábamos en estado de configuración.

3.2.4. Tipo y cabeceras de la respuesta

En estado conectado, además del contenido del mensaje de la respuesta, podemos obtener el estado de la respuesta y sus cabeceras. Los estados de respuesta se componen de un código y un mensaje y nos permitirán saber si la petición ha podido atenderse

correctamente o si por el contrario ha habido algún tipo de error. Por ejemplo, posibles estados son:

`HttpConnection.HTTP_OK`	200	OK
`HttpConnection.HTTP_BAD_`	400	Bad Request
`HttpConnection.HTTP_INTE`	500	Internal Server Error

Este mensaje de estado encabeza el mensaje de respuesta. Si el servidor nos devuelve un mensaje con código 200 como el siguiente:

```
HTTP/1.1 200 OK
```

Es que se ha procesado correctamente la petición y nos devuelve su respuesta. Si ha ocurrido un error, nos mandará el código y mensaje de error correspondiente. Por ejemplo, el error 400 indica que el servidor no ha entendido la petición que hemos hecho, posiblemente porque la hemos escrito incorrectamente. El error 500 nos dice que se trata de un error interno del servidor, no de la petición realizada.

Podemos obtener tanto el código como el mensaje de estado con los siguientes métodos:

```
int cod = con.getResponseCode();
String msg = con.getResponseMessage();
```

Los códigos de estado podemos encontrarlos como constantes de la clase `HttpConnection` como hemos visto para los tres códigos anteriores.

También podemos utilizar este objeto para leer las cabeceras que nos ha devuelto la respuesta. Nos ofrece métodos para leer una serie de cabeceras estándar de HTTP como los siguientes:

`getLength`	`content-length`	Longitud del contenido, o `-1` si la longitud es desconocida
`getType`	`content-type`	Tipo MIME del contenido devuelto
`getEncoding`	`content-encoding`	Codificación del contenido
`getExpiration`	`expires`	Fecha de expiración del recurso
`getDate`	`date`	Fecha de envío del recurso
`getLastModified`	`last-modified`	Fecha de última modificación del recurso

Puede ser que queramos obtener otras cabeceras, como por ejemplo cabeceras propias no estándar. Para ello tendremos una serie de métodos que obtendrán las cabeceras

directamente por su nombre:

```
String valor = con.getHeaderField(nombre);
int valor = con.getHeaderFieldInt(nombre);
long valor = con.getHeaderFieldDate(nombre);
```

De esta forma podemos obtener el valor de la cabecera o bien como una cadena, o en los datos que sean de tipo fecha (valor `long`) o enteros también podremos obtener su valor directamente en estos tipos de datos.

Podremos acceder a las cabeceras también a partir de su índice:

```
String valor = con.getHeaderField(int indice);
String nombre = con.getHeaderFieldKey(int indice);
```

Podemos obtener de esta forma tanto el nombre como el valor de la cabecera que ocupa un determinado índice.

Esta respuesta HTTP, además de un estado y una serie de cabeceras, tendrá un bloque que contenido que podremos leer abriendo un flujo de entrada en la conexión como hemos visto anteriormente. Normalmente cuando hacemos una petición a una URL de una aplicación web nos devuelve como contenido un documento HTML. Sin embargo, en el caso de nuestra aplicación MIDP este tipo de contenido no es apropiado. En su lugar podremos utilizar como contenido de la respuesta cualquier otro tipo MIME, que vendrá indicado en la cabecera `content-type` de la respuesta. Por ejemplo, podremos devolver una respuesta codificada de forma binaria que sea leída y descodificada por nuestra aplicación MIDP.

Tanto los métodos que obtienen un flujo para leer o escribir en la conexión, como estos métodos que acabamos de ver para obtener información sobre la respuesta producirán una transición al estado conectado.

3.3. Acceso a la red a bajo nivel

Como hemos comentado, el único tipo de conexión especificada en MIDP 1.0 es HTTP, la cual es suficiente y adecuada para acceder a aplicaciones corporativas. Sin embargo, con las redes 2.5G y 3G tendremos una mayor capacidad en las conexiones, nos permitirán realizar cualquier tipo de conexión TCP y UDP (no sólo HTTP) y además la comunicación podrá ser más fluida.

Para poder acceder a estas mejoras desde nuestra aplicación Java, surge la necesidad de que en MIDP 2.0 se incorpore soporte para tipos de conexiones a bajo nivel: sockets (TCP) y datagramas (UDP). Estos tipos de conexiones de MIDP 2.0 son optativos, de forma que aunque se encuentran definidos en la especificación de MIDP 2.0, no se obliga a que los fabricantes ni los operadores de telefonía lo soporten. Es decir, que la

posibilidad de utilizar estas conexiones dependerá de que la red de telefonía de nuestro operador y el modelo de nuestro móvil las soporte.

Si intentamos utilizar un tipo de conexión no soportada por nuestro sistema, se producirá una excepción de tipo `ConnectionNotFoundException`.

3.3.1. Sockets

Los sockets nos permiten crear conexiones TCP. En este tipo de conexiones se establece un circuito virtual de forma permanente entre los dispositivos que se comunican. Se nos asegura que los datos enviados han llegado al servidor y que han llegado en el mismo orden en el que los enviamos. El inconveniente que tienen es que el tener un canal de comunicación abierto permanentemente consume una mayor cantidad de recursos.

Para abrir una conexión mediante sockets utilizaremos una URL como la siguiente:

```
SocketConnection sc =
    (SocketConnection) Connector.open("socket://host:puerto");
```

Una vez abierta la conexión, podremos abrir sus correspondientes flujos de entrada y salida para enviar y recibir datos a través de ella:

```
InputStream in = sc.openInputStream();
OutputStream out = sc.openOutputStream();
```

Es posible también hacer que nuestro dispositivos actúe como servidor. En este caso utilizaremos una URL como las siguientes para crear el socket servidor:

```
ServerSocketConnection ssc =
  (ServerSocketConnection) Connector.open("socket://:puerto");
ServerSocketConnection ssc =
  (ServerSocketConnection) Connector.open("socket://");
```

En el primer caso indicamos el puerto en el que queremos que escuche nuestro servidor. En el segundo caso este puerto será asignado automáticamente por el sistema. Para conocer la dirección y el puerto donde escucha nuestro servidor podremos utilizar los siguientes métodos:

```
int puerto = ssc.getLocalPort();
String host = ssc.getLocalAddress();
```

Para hacer que el servidor comience a escuchar y aceptar conexiones utilizaremos el siguiente método:

```
SocketConnection sc = (SocketConnection) ssc.acceptAndOpen();
```

Obtendremos un objeto `SocketConnection` con el que podremos comunicarnos con el cliente que acaba de conectarse a nuestro servidor.

Debemos tener en cuenta que normalmente los móviles realizan conexiones puntuales cuando necesitan acceder a la red, y cada vez que se conecta se le asigna una nueva IP de forma dinámica. Esto hace difícil que un móvil pueda comportarse como servidor, ya que no podremos conocer a priori la dirección en la que está atendiendo para poder conectarnos a ella desde un cliente.

3.3.2. Datagramas

Cuando trabajemos con datagramas estaremos utilizando una conexión UDP. En ella no se establece un circuito virtual permanente, sino que cada paquete (datagrama) es enrutado de forma independiente. Esto produce que los paquetes puedan perderse o llegar desordenados al destino. Cuando la pérdida de paquetes o su ordenación no sea críticos, convendrá utilizar este tipo de conexiones, ya que consume menos recursos que los circuitos virtuales.

Para trabajar con datagramas utilizaremos una URL como la siguiente:

```
DatagramConnection dc =
   (DatagramConnection) Connector.open("datagram://host:puerto");
```

En este caso no hemos abierto una conexión, ya que sólo se establecerá una conexión cuando se envíe un datagrama, simplemente hemos creado el objeto que nos permitirá intercambiar estos paquetes. Podemos crear un datagrama que contenga datos codificados en binario de la siguiente forma:

```
byte[] datos = obtenerDatos();
Datagram dg = dc.newDatagram(datos, datos.length);
```

Una vez hemos creado el datagrama, podemos enviarlo al destinatario utilizando la conexión:

```
dc.send(dg);
```

En el caso del servidor, crearemos la conexión de datagramas de forma similar, pero sin especificar la dirección a la que conectar, ya que dependiendo del cliente deberemos enviar los datagramas a diferentes direcciones.

```
DatagramConnection dc =
   (DatagramConnection) Connector.open("datagram://:puerto");
```

El servidor no conocerá las direcciones de sus clientes hasta que haya recibido algún datagrama de ellos. Para recibir un datagrama crearemos un datagrama vacío indicando su capacidad (en bytes) y lo utilizaremos para recibir en él la información que se nos envía desde el cliente de la siguiente forma:

```
Datagram dg = dc.newDatagram(longitud);
dc.receive(dg);
```

Una vez obtenido el datagrama, podremos obtener la dirección desde la cual se nos envía:

```
String direccion = dg.getAddress();
```

Ahora podremos crear un nuevo datagrama con la respuesta indicando la dirección a la que vamos a enviarlo. En este caso en cada datagrama se deberá especificar la dirección a la que se envía:

```
Datagram dg = dc.newDatagram(datos, datos.length, direccion);
```

El datagrama será enviado de la misma forma en la que se hacía en el cliente. Posteriormente el cliente podrá recibir este datagrama de la misma forma en que hemos visto que el servidor recibía su primer datagrama. De esta forma podremos establecer una conversación entre cliente y servidor, intercambiando estos datagramas.

3.4. Envío y recepción de mensajes

Podemos utilizar la API adicional WMA para enviar o recibir mensajes cortos (SMS, Short Message Service) a través del teléfono móvil. Esta API extiende GFC, permitiendo establecer conexiones para recibir o enviar mensajes. Cuando queramos enviar mensajes nos comportaremos como clientes en la conexión, mientras que para recibirlos actuaremos como servidor. La URL para establecer una conexión con el sistema de mensajes para ser enviados o recibidos a través de una portadora SMS sobre GSM tendrá el siguiente formato:

```
sms://telefono:puerto
```

Las clases de esta API se encuentran en el paquete `javax.wireless.messaging`. Aquí se definen una serie de interfaces para trabajar con los mensajes y con la conexión.

3.4.1. Envio de mensajes

Si queremos enviar mensajes, deberemos crear una conexión cliente proporcionando en la URL el número del teléfono al que vamos a enviar el mensaje y el puerto al que lo

enviaremos de forma opcional:

```
sms://+34555000000
sms://+34555000000:4444
```

Si no especificamos el puerto se utilizará el puerto que se use por defecto para los mensajes del usuario en el teléfono móvil. Deberemos abrir una conexión con una de estas URLs utilizando GFC, con lo que nos devolverá una conexión de tipo `MessageConnection`

```
MessageConnection mc =
    (MessageConnection)Connector.open("sms://+34555000000");
```

Una vez creada la conexión podremos utilizarla para enviar mensajes cortos. Podremos mandar tanto mensajes de texto como binarios. Estos mensajes tienen un tamaño limitado a un máximo de 140 bytes. Si el mensaje es de texto el número de caracteres dependerá de la codificación de éstos. Por ejemplo si los codificamos con 7 bits tendremos una longitud de 160 caracteres, mientras que con una codificación de 8 bits tendremos un juego de caracteres más amplio pero los mensajes estarán limitados a 140 caracteres.

WMA permite encadenar mensajes, de forma que esta longitud podrá ser por lo menos 3 veces mayor. El encadenamiento consiste en que si el mensaje supera la longitud máxima de 140 bytes que puede transportar SMS, entonces se fracciona en varios fragmentos que serán enviados independientemente a través de SMS y serán unidos al llegar a su destino para formar el mensaje completo. Esto tiene el inconveniente de que realmente por la red están circulando varios mensajes, por lo que se nos cobrará por el número de fragmentos que haya enviado.

Podremos crear el mensaje a enviar a partir de la conexión. Los mensajes de texto los crearemos de la siguiente forma:

```
String texto = "Este es un mensaje corto de texto";
TextMessage msg = mc.newMessage(mc.TEXT_MESSAGE);
msg.setPayloadText(texto);
```

Para el caso de un mensaje binario, lo crearemos de la siguiente forma:

```
byte [] datos = codificarDatos();
BinaryMessage msg = mc.newMessage(mc.BINARY_MESSAGE);
msg.setPayloadData(datos);
```

Antes de enviar el mensaje, podemos ver en cuántos fragmentos deberá ser dividido para poder ser enviado utilizando la red subyacente con el siguiente método:

```
int num_segmentos = mc.numberOfSegments(msg);
```

Esto nos devolverá el número de segmentos en los que se fraccionará el mensaje, ó 0 si el mensaje no puede ser enviado utilizando la red subyacente.

Independientemente de si se trata de un mensaje de texto o de un mensaje binario, podremos enviarlo utilizando el siguiente método:

```
            mc.send(msg);
```

3.4.2. Recepción de mensajes

Para recibir mensajes deberemos crear una conexión de tipo servidor. Para ello en la URL sólo especificaremos el puerto en el que queremos recibir los mensajes:

```
sms://:4444
```

Crearemos una conexión utilizando una URL como esta, en la que no se especifique el número de teléfono destino.

```
MessageConnection mc =
    (MessageConnection)Connector.open("sms://:4444");
```

Para recibir un mensaje utilizaremos el método:

```
Message msg = mc.receive();
```

Si hemos recibido un mensaje que todavía no hay sido leido este método obtendrá dicho mensaje. Si todavía no se ha recibido ningún mensaje, este método se quedará bloqueado hasta que se reciba un mensaje, momento en el que lo leerá y nos lo devolverá.

Podemos determinar en tiempo de ejecución si se trata de un mensaje de texto o de un mensaje binario. Para ello deberemos comprobar de qué tipo es realmente el objeto devuelto, y según este tipo leer sus datos como texto o como array de bytes:

```
if(msg instanceof TextMessage) {
    String texto = ((TextMessage)msg).getPayloadText();
    // Procesar texto
} else if(msg instanceof TextMessage) {
    byte [] datos = ((BinaryMessage)msg).getPayloadData();
    // Procesar datos
}
```

Hemos visto que el método `receive` se queda bloqueado hasta que se reciba un mensaje. No debemos hacer que la aplicación se quede bloqueada esperando un mensaje, ya que

éste puede tardar bastante, o incluso no llegar nunca. Podemos solucionar este problema realizando la lectura de los mensajes mediante un hilo en segundo plano. Otra solución es utilizar un listener.

3.4.3. Listener de mensajes

Estos *listeners* nos servirán para que se nos notifique el momento en el que se recibe un mensaje corto. De esta forma no tendremos que quedarnos bloqueados esperando recibir el mensaje, sino que podemos invocar `receive` directamente cuando sepamos que se ha recibido el mensaje.

Para crear un *listener* de este tipo deberemos crear una clase que implemente la interfaz `MessageListener`:

```
public MiListener implements MessageListener {
    public void notifyIncomingMessage(MessageConnection mc) {
        // Se ha recibido un mensaje a través de la conexion mc
    }
}
```

Dentro del método `notifyIncomingMessage` deberemos introducir el código a ejecutar cuando se reciba un mensaje. No debemos ejecutar la operación `receive` directamente dentro de este método, ya que es una operación costosa que no debe ser ejecutada dentro de los *callbacks* que deben devolver el control lo antes posible para no entorpecer el procesamiento de eventos de la aplicación. Deberemos hacer que la recepción del mensaje la realice un hilo independiente.

Para que la recepción de mensajes le sea notificada a nuestro listener deberemos registrarlo como listener de la conexión con:

```
mc.setMessageListener(new MiListener());
```

En WTK 2.0 tenemos disponible una consola WMA con la que podremos simular el envío y la recepción de mensajes cortos que se intercambien entre los emuladores, de forma que podremos probar estas aplicaciones sin tener que enviar realmente los mensajes y pagar por ellos.