

CURSO DE J2ME

(JAVA 2 MICROEDITION)

Manuel J. Prieto
(vitike@canal21.com)
(Abril. 2003)

Cualquier comentario, sugerencia o errata, puede ser remitida a vitike@canal21.com.

Todos los mensajes serán bienvenidos y sin duda ayudarán a mejorar este documento y las posteriores ampliaciones del mismo. Cualquier otra cuestión o tema puede ser remitida también a la misma dirección.

1	PRESENTACIÓN DE J2ME	6
1.1	Nuevos Conceptos	6
1.1.1	Configuración.....	6
1.1.2	Perfiles.....	7
1.2	Primer MIDlet	9
1.2.1	Descripción del MIDlet.....	9
1.2.2	Compilación.....	12
2	LAS APIS DE CLDC Y DE MIDP	15
2.1	El API de CLDC.....	16
2.1.1	El paquete java.lang	16
2.1.2	El paquete java.util	17
2.1.3	El paquete java.io	17
2.2	El GCF (Generic Connection Framework) de CLDC.....	18
2.3	El API de MIDP	19
2.3.1	Las clases heredadas de J2SE	20
2.3.2	Clases e interfaces propios de MIDP	20
2.3.3	El paquete javax.microedition.midlet	20
2.3.4	El paquete javax.microedition.lcdui.....	20
2.3.5	El paquete javax.microedition.io	23
2.3.6	El paquete javax.microedition.rms	23
2.4	Ejemplo de uso de APIs de MIDP y J2SE.....	24
3	MIDLETS GRÁFICOS	28
3.1	La clase Graphics	28
3.2	Primitivas gráficas	29
3.3	Escribiendo texto	30
3.4	Dibujando imágenes	33
3.5	Ejemplo de uso de los métodos gráficos	34
4	COMPONENTES DE INTERFAZ DE USUARIO.....	39
4.1	Screens y Forms	40
4.2	La clase Alert.....	44

4.3	La clase List.....	45
4.4	La clase TextBox	47
4.5	La clase Ticker	48
4.6	La clase StringItem	49
4.7	La clase ImageItem	50
4.8	La clase TextField	54
4.9	La clase DateField.....	55
4.10	La clase ChoiceGroup	55
4.11	La clase Gauge.....	56
5	GESTIÓN DE COMANDOS.....	58
6	CONEXIÓN A REDES.....	61
6.1	Entrada/Salida desde el MIDlet.....	64
6.2	La clase InputStream	65
6.3	La clase OutputStream	67
6.4	Ejemplo de conexión.....	68
7	PERSISTENCIA DE DATOS (RMS).....	74
7.1	El paquete RMS	74
7.2	La clase RecordStore	75
7.3	Las interfaces de RMS.....	75
7.4	Abriendo un almacén de registros	75
7.5	Añadiendo nuevos registros	76
7.6	Recuperando registros	76
7.7	Borrando registros.....	77
7.8	Enumeración de registros.....	77
7.9	Cerrando un almacén de datos.....	79

8	MIDP 2.0.....	80
8.1	El API de juegos de MIDP 2.0	82
8.2	El Push Registry de MIDP 2.0.....	87
9	MEJORANDO LA EXPERIENCIA DEL USUARIO EN CONEXIONES J2ME.....	92
10	ALGUNOS PATRONES DE DISEÑO PARA J2ME.....	98
10.1	Patrón para la generación de menús en cascada.....	98
10.2	Patrón para la generación de diálogos.....	100
10.3	Patrón de la paginación	102
10.4	Patrón para la creación de aplicaciones portables.....	103
11	OPTIMIZACIÓN DE CÓDIGO.....	107
11.1	Optimización para la mantenibilidad	107
11.2	Optimización del tamaño	107
11.3	Optimización de velocidad	108
11.4	Eliminar subexpresiones comunes.....	109
11.5	Aprovechar las variables locales.....	109
11.6	Expandir los bucles.....	109
11.7	Mejora de las operaciones gráficas	110
11.8	Recolector de basura	111

1 PRESENTACIÓN DE J2ME

J2ME es el acrónimo de Java 2 *Micro Edition*. J2ME es la versión de Java orientada a los dispositivos móviles. Debido a que los dispositivos móviles tienen una potencia de cálculo baja e interfaces de usuario pobres, es necesaria una versión específica de Java destinada a estos dispositivos, ya que el resto de versiones de Java, J2SE o J2EE, no encajan dentro de este esquema. J2ME es por tanto, una versión “reducida” de J2SE.

1.1 Nuevos Conceptos

1.1.1 Configuración

La configuración es un mínimo grupo de APIs (Application Program Interface), útiles para desarrollar las aplicaciones destinadas a un amplio rango de dispositivos. La configuración estándar para los dispositivos inalámbricos es conocida como CLDC (*Connected Limited Device Configuration*). El CLDC proporciona un nivel mínimo de funcionalidades para desarrollar aplicaciones para un determinado conjunto de dispositivos inalámbricos. Se puede decir que CLDC es el conjunto de clases esenciales para construir aplicaciones. Hoy por hoy, sólo tenemos una configuración, pero es de esperar que en el futuro aparezcan distintas configuraciones orientadas a determinados grupos de dispositivos.

Los requisitos mínimos de hardware que contempla CLDC son:

- 160KB de memoria disponible para Java
- Procesador de 16 bits
- Consumo bajo de batería
- Conexión a red

Los dispositivos que claramente encajan dentro de este grupo, son los teléfono móviles, los PDA (*Personal Digital Assistant*), los “Pocket PC”...

En cuanto a los requisitos de memoria, según CLDC, los 160KB se utilizan de la siguiente forma:

- 128KB de memoria no volátil para la máquina virtual Java y para las librerías del API de CLDC
- 32KB de memoria volátil, para sistema de ejecución (*Java Runtime System*).

En cuanto a las limitaciones impuestas por CLDC, tenemos por ejemplo las operaciones en coma flotante. CLDC no proporciona soporte para matemática en coma flotante. Otra limitación es la eliminación del método *Object.finalize*. Este método es invocado cuando un objeto es eliminado de la memoria, para optimizar los recursos. También se limita el manejo de las excepciones. Es complicado definir una serie de clases de error estándar, que se ajuste a todos los dispositivos contemplados dentro de CLDC. La solución es soportar un grupo limitado de clases de error y permitir que el API específico de cada dispositivo defina su propio conjunto de errores y excepciones.

La seguridad dentro de CLDC es sencilla, sigue el famoso modelo *sandbox*. Las líneas básicas del modelo de seguridad *sandbox* en CLDC son:

- Los ficheros de clases, deben ser verificados como aplicaciones válidas.
- Sólo las APIs predefinidas dentro de CLDC están disponibles.
- No se permite cargadores de clases definidos por el usuario.
- Sólo las capacidades nativas proporcionadas por CLDC son accesibles.

1.1.2 Perfiles

En la arquitectura de J2ME, por encima de la configuración, tenemos el perfil (*profile*). El perfil es un grupo más específico de APIs, desde el punto de vista del dispositivo. Es decir, la configuración se ajusta a

una familia de dispositivos, y el perfil se orienta hacia un grupo determinado de dispositivos dentro de dicha familia. El perfil, añade funcionalidades adicionales a las proporcionadas por la configuración. La especificación MIDP (*Mobile Information Device Profile*), describe un dispositivo MIDP como un dispositivo, pequeño, de recursos limitados, móvil y con una conexión “inalámbrica”.

MIDLet

Las aplicaciones J2ME desarrolladas bajo la especificación MIDP, se denominan MIDlets. Las clases de un MIDlet, son almacenadas en *bytecodes* java, dentro de un fichero *.class*. Estas clases, deben ser verificadas antes de su “puesta en marcha”, para garantizar que no realizan ninguna operación no permitida. Este preverificación, se debe hacer debido a las limitaciones de la máquina virtual usada en estos dispositivos. Esta máquina virtual se denomina KVM. Para mantener esta máquina virtual lo más sencilla y pequeña posible, se elimina esta verificación, y se realiza antes de la entrada en producción. La preverificación se realiza después de la compilación, y el resultado es una nueva clase, lista para ser puesta en producción.

Los MIDlets, son empaquetados en ficheros “.jar”. Se requiere alguna información extra, para la puesta en marcha de las aplicaciones. Esta información se almacena en el fichero de “manifiesto”, que va incluido en el fichero “.jar” y en un fichero descriptor, con extensión “.jad”. Un fichero “.jar” típico, por tanto, se compondrá de:

- Clases del MIDlet
- Clases de soporte
- Recursos (imágenes, sonidos...)
- Manifiesto (fichero “.mf”)
- Descriptor (fichero “.jad”)

Un fichero “.jar” puede contener varios MIDlets. Esta colección de MIDlets, se suele llamar “MIDlet Suite”. Esta unión de varios

MIDlets en una distribución, permite compartir recursos (imágenes, sonidos...), y por tanto optimizar los recursos del dispositivo.

1.2 Primer MIDlet

1.2.1 Descripción del MIDlet

Todos los MIDlets, deben heredar de la clase *javax.microedition.midlet.MIDlet*, contenida en el API MIDP estándar. Esta clase define varios métodos, de los cuales destacaremos los siguientes:

- *startApp()* – Lanza el MIDlet
- *pauseApp()* – Para el MIDlet
- *destroyApp()* – Destruye el MIDlet

1.2.1.1 CICLO DE VIDA

Los MIDlets tienen tres posibles estados que determinan su comportamiento: activo, parado y destruido. Estos estados se relacionan directamente con los métodos antes enumerados. Estos métodos son llamados directamente por el entorno de ejecución de aplicaciones, pero también pueden ser invocados desde código.

Los métodos indicados anteriormente, serán normalmente utilizados para gestionar los recursos (solicitar y liberar). Un MIDlet puede ser lanzado y parado varias veces, pero sólo será destruido una vez.

1.2.1.2 COMANDOS MIDLET

La mayoría de los MIDlets, implementan el método *commandAction()*, un método de respuesta a eventos, definido en la interfaz *javax.microedition.lcdui.CommandListener*. La forma de funcionar de este método, es similar al control de eventos típico de java.

1.2.1.3 DISPLAY, SCREEN Y CANVAS

La clase *javax.microedition.lcdui.Display*, representa el controlador de pantalla del dispositivo. Esta clase es responsable de controlar la pantalla y la interacción con el usuario. Nunca se debe crear un objeto *Display*, normalmente se obtiene una referencia al objeto *Display* en el constructor del MIDlet, y se usa para crear la interfaz de usuario en el método *startApp()*. Hay una instancia de *Display* por cada MIDlet que se está ejecutando en el dispositivo.

Mientras que del objeto *Display* sólo hay una instancia, del objeto *javax.microedition.lcdui.Screen*, pueden existir varias. El objeto *Screen*, es un componente GUI genérico, que sirve como base para otros componentes. Estos objetos representan una pantalla entera de información. Sólo se puede mostrar una pantalla cada vez. La mayoría de los MIDlets, usan subclases de la clase *Screen*, como *Form*, *TextBox* o *List*, ya que proporcionan una funcionalidad mucho más específica.

Una clase similar en cierta medida a *Screen*, es la clase *Canvas*, perteneciente al mismo paquete. Los objetos *Canvas* son usados para realizar operaciones gráficas directamente, como puede ser pintar líneas o imágenes. No se puede mostrar un objeto *Canvas* y un objeto *Screen* a la vez, pero se pueden alternar en la misma aplicación.

1.2.1.4 MANOS A LA OBRA

```
package cursoj2me;

/**
 * Curso de J2ME
 * Manuel J. Prieto
 */
```

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

public class MIDLet1 extends MIDlet implements CommandListener {
    private Command salir;
    private Display display;
    private Form pantalla;

    public MIDLet1(){
        // Recogemos el objeto Display
        display = Display.getDisplay(this);

        // Creamos el comando de salida
        salir = new Command("Salir", Command.EXIT, 2);

        // Creamos el "Form" de la pantalla principal
        pantalla = new Form("Primer MIDLet");

        // Creamos una cadena y la ponemos en pantalla
        StringItem cad = new StringItem("", "Este es mi primer MIDLet");
        pantalla.append(cad);

        // Añadimos y configuramos el comando de salida
        pantalla.addCommand(salir);
        pantalla.setCommandListener(this);
    }

    public void startApp() throws MIDletStateChangeException{
        // Establecemos el "Display" actual a nuestra pantalla
        display.setCurrent(pantalla);
    }

    public void pauseApp(){
    }

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

```

public void commandAction(Command c, Displayable s){
    if (c == salir){
        destroyApp(false);
        notifyDestroyed();
    }
}
}
}

```

1.2.2 Compilación

Para compilar el código fuente con las utilidades de línea de comando del *Java Wireless Toolkit*, se usa el comando *javac*. Se debe usar la opción *-bootclasspath*, para que el código fuente se compile contra las clases del CLDC y del MIDP.

```

javac      -bootclasspath      [j_w_toolkit_home]\lib\midpapi.zip
MIDLet1.java

```

El comando anterior produce el fichero MIDLet1.class.

1.2.2.1 Preverificación de la clase

El siguiente paso es preverificar la clase, usando el comando *preverify*.

```

preverify -classpath [ruta_clase];[j_w_toolkit_home]\lib\midpapi.zip
MIDLet1

```

El comando anterior crea un directorio dentro de la ruta en la que nos encontremos y crea un nuevo fichero MIDLet1.class. Esta es la clase preverificada que la máquina virtual (KVM) puede ejecutar.

1.2.2.2 Empaquetamiento de la clase en un fichero jar

Para permitir la descarga de aplicaciones MIDP, la aplicación debe estar empaquetada dentro de un fichero jar.

```
jar cvf midlet1.jar MIDLet1.class
```

1.2.2.3 Creación del fichero jad

El fichero jad es necesario para ejecutar la aplicación. Un fichero jad podría ser:

```
MIDLet-1: midlet1, ,MIDLet1
MIDLet-Name: Prueba1
MIDLet-Version: 1.0
MIDLet-Vendor: Manuel J. Prieto
MIDLet-Jar-URL: midlet1.jar
MIDLet-Jar-Size: 981
```

1.2.2.4 Lanzamiento del emulador

Una vez que tenemos el fichero jad, podemos probar el midlet desarrollado.

```
Emulator -Xdescriptor:[ruta]\midlet1.jad
```

1.2.2.5 KToolBar

Como hemos visto, el proceso completo de construcción de un midlet desde la línea de comandos, es un poco tedioso. Para agilizar este trabajo, tenemos varias alternativas:

- Desarrollar una serie de scripts que hagan este trabajo.
- Usar la aplicación KToolBar, disponible en el *J2ME Wireless Toolkit*.
- Usar la herramienta ant del proyecto Jakarta

KToolBar no incluye un editor de código, pero cubre el resto del proceso. Es decir, una vez que tenemos el código fuente escrito, con esta aplicación podemos compilar, preverificar, empaquetar y comprobar el funcionamiento de los MIDLet.

Para que la herramienta KToolBar funcione correctamente, los MIDLets deben estar bajo el directorio "apps" que está en el directorio de instalación del *J2ME Wireless Toolkit*. KToolBar también asume que el directorio principal del MIDLet tiene el mismo nombre que este en el fichero JAD. Siguiendo con la organización dentro de KToolBar, este espera que el código fuente del MIDLet esté dentro del directorio "src", que debe colgar del directorio principal del MIDLet. Debajo del directorio principal del MIDLet, también debe ir un directorio denominado "bin", que contendrá el fichero jar y el fichero jad. Una vez que tenemos estos requerimientos cumplidos, podemos lanzar la aplicación KToolBar y realizar los procesos antes descritos (compilar, preverificar...)

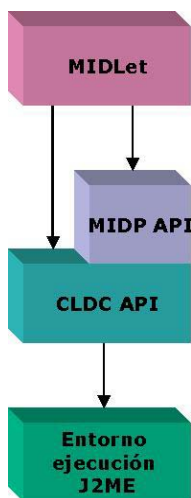
2 LAS APIs DE CLDC Y DE MIDP

Los elementos principales involucrados en el proceso de desarrollo con Java, son el lenguaje Java propiamente dicho y el grupo de APIs (*Application Programming Interface*) que proporcionan el soporte para el software desarrollado.

Los APIs específicos para el desarrollo de MIDlets, están descritos en las especificaciones de CLDC y MIDP, que definen la configuración y el perfil para los dispositivos inalámbricos móviles. Estas APIs constan de clases e interfaces ya presentes en el estándar J2SE así como con clases e interfaces únicos del desarrollo de MIDlets.

Como ya se ha mencionado, los MIDlets son aplicaciones especiales, diseñadas bajo los requerimientos de la especificación MIDP. Esta especificación son una serie de normas que indican las capacidades y restricciones de Java respecto a los dispositivos móviles. Un aspecto importante de estas capacidades y limitaciones, es el conjunto de clases e interfaces disponibles para afrontar el desarrollo de aplicaciones.

La especificación MIDP provee una descripción detallada del API disponible para el desarrollo de MIDlets. El CLDC proporciona un API adicional. De hecho, el API de MIDP, se basa en el API de CLDC, para construir clases e interfaces más específicos.



Relación entre el API de CLDC y MIDP

2.1 El API de CLDC

El API de CLDC es un pequeño subgrupo del API de J2SE. A parte de estas clases e interfaces, el API de CLDC contiene una serie de interfaces propias, dedicadas a los servicios de red.

2.1.1 El paquete `java.lang`

Las clases e interfaces del paquete *java.lang*, están relacionadas con el núcleo del lenguaje Java. Es decir, estas clases incluyen soporte para las capacidades del lenguaje como los recubrimientos de los tipos primitivos de variables, las cadenas, las excepciones y los *threads*, entre otras.

Las clases e interfaces del lenguaje `java.lang` son :

- Boolean – Encapsula el tipo primitivo boolean
- Byte – Encapsula el tipo primitivo byte
- Character – Encapsula el tipo primitivo char
- Class – Proporciona información sobre la ejecución de una clase
- Integer – Encapsula el tipo primitivo int
- Long – Encapsula el tipo primitivo long
- Math – Proporciona acceso a varias operaciones y constantes matemáticas
- Object – La superclase del resto de clases en Java
- Runnable – Interfaz que proporciona un significado a la creación de threads (hilos), sin heredar la clase Thread
- Runtime – Proporciona acceso al entorno de ejecución
- Short – Encapsula el tipo primitivo short
- String – Representa una cadena de texto constante
- StringBuffer – Representa una cadena de texto, de longitud y valor variable

- System – Proporciona acceso a los recursos del sistema
- Thread – Se usa para crear un “thread” (hilo) de ejecución dentro de un programa
- Throwable – Proporciona soporte para el control de excepciones

2.1.2 El paquete java.util

El paquete *java.util*, como en J2SE, incluye clases e interfaces con utilidades variadas, como puede ser el manejo de fechas, estructuras de datos...

Las clases e interfaces de *java.util* son:

- Calendar – Proporciona funciones para manejar fechas y convertir valores numéricos en fechas
- Date – Representa un instante de tiempo
- Enumeration – Es una interfaz que describe como manejar la iteración entre un grupo de valores
- Hashtable – Una colección que asocia valores y claves
- Random – Generador de números pseudoaleatorios
- Stack – Una colección con gestión LIFO
- TimeZone – Representa la zona horaria
- Vector – Una colección en forma de matriz dinámica

2.1.3 El paquete java.io

Este paquete proporciona clases e interfaces de apoyo para leer y escribir datos. Aunque la funciones de persistencia, recaen sobre los perfiles.

Las clases e interfaces de *java.io* son:

- ByteArrayInputStream – Un flujo (stream) de entrada que se gestiona internamente como una matriz de bytes
- ByteArrayOutputStream – Un flujo (stream) de salida que se gestiona internamente como una matriz de bytes

- `DataInput` – Una interfaz que define los métodos para leer datos desde un flujo (stream) binario a tipos primitivos
- `DataInputStream` – Un flujo (stream) desde el cual se leen datos como tipos primitivos
- `DataOutput` – Una interfaz que define métodos para escribir datos en forma de tipos primitivos en un flujo (stream) binario
- `DataOutputStream` – Escribe datos en tipos primitivos en un flujo (stream) en formato binario
- `InputStream` – La clase base para todos los flujos (streams) de entrada
- `InputStreamReader` – Un flujo (stream) desde el que se pueden leer caracteres de texto
- `OutputStream` – La clase base para todos los flujos (streams) de salida
- `OutputStreamWriter` – Un flujo (stream) en el que se pueden escribir caracteres de texto
- `PrintStream` – Un flujo (stream) de escritura que facilita el “envío” de datos en forma de tipos primitivos
- `Reader` – Una clase abstracta para leer flujos (streams) de lectura
- `Writer` – Una clase abstracta para leer flujos (streams) de escritura

2.2 El GCF (*Generic Connection Framework*) de CLDC

Debido a las dificultades para proveer soporte para funciones de red a nivel de configuración, por la variedad en los dispositivos, el CLDC delega esta parte del API a los perfiles. Para realizar esta delegación de forma satisfactoria, el CLDC ofrece un marco general de trabajo en red, conocido como el GCF (*Generic Connection Framework*). El GCF está compuesto básicamente por una serie de interfaces de conexión, junto con una clase “Conector” que es usada para establecer las

diferentes conexiones. Todo esto, está dentro del paquete *javax.microedition.io*.

Las interfaces del paquete *javax.microedition.io* son:

- Connection – Una conexión básica que sólo puede ser abierta y cerrada
- ContentConnection – Un flujo (stream) de conexión que proporciona acceso a datos web
- DatagramConnection – Una conexión para manejar comunicaciones orientadas a paquetes
- InputConnection – Una conexión de entrada para las comunicaciones del dispositivo
- OutputConnection – Una conexión de salida para las comunicaciones del dispositivo
- StreamConnection – Una conexión en ambas direcciones para las comunicaciones del dispositivo
- StreamConnectionNotifier – Una conexión especial para notificaciones, que es usada para esperar que se establezca una conexión

2.3 El API de MIDP

El perfil de dispositivo, comienza donde la configuración para, en lo que se refiere a proveer funciones para llevar a cabo importantes tareas en un determinado tipo de dispositivo.

De forma similar a lo que hicimos con el API de CLDC, el API de MIDP se puede dividir en dos partes. Dos clases heredadas directamente del API de J2SE y una serie de paquetes que incluyen clases e interfaces únicas para el desarrollo de MIDP.

2.3.1 Las clases heredadas de J2SE

Sólo dos clases del API de MIDP, provienen directamente del API de J2SE. Estas clases están dentro del paquete *java.util*, dentro del API de MIDP.

Timer – Proporciona funcionalidad para crear tareas programadas temporalmente

TimerTask – Representa una tarea que es temporizada a través de la clase Timer

2.3.2 Clases e interfaces propios de MIDP

La gran parte del API de MIDP, son una serie de clases e interfaces diseñadas explícitamente para la programación de MIDlets. Aunque estas clases e interfaces son similares a algunas clases del API de J2SE, son totalmente exclusivas del API de MIDP. Esta parte del API se divide en varios paquetes:

- javax.microedition.midlet
- javax.microedition.lcdui
- javax.microedition.io
- javax.microedition.rms

2.3.3 El paquete javax.microedition.midlet

Este es el paquete central del API de MIDP y contiene una sola clase: MIDlet. Esta clase provee la funcionalidad básica para que una aplicación se puede ejecutar dentro de un dispositivo con soporte para MIDlets.

2.3.4 El paquete javax.microedition.lcdui

Este paquete contiene clases e interfaces que soportan componentes de interfaz de usuario (GUI), específicos para las pantallas de los

dispositivo móviles. La funcionalidad de este paquete es similar a la del *Abstract Windowing Toolkit* (AWT) de J2SE, aunque bastante más reducida, como es obvio.

Lcdui, corresponde al texto *Liquid Crystal Displays User Interfaces*. Las pantallas de cristal líquido son comunes en los dispositivos móviles.

Un concepto básico dentro de la programación de MIDlets, es la pantalla (*screen*), que es un componente GUI genérico, que sirve de clase base para otros componentes. Las interfaces de usuario, se compondrán añadiendo componentes a la clase base (*screen*). A parte de la creación de interfaces de usuario mediante esta aproximación de alto nivel, también se puede acceder directamente a primitivas de dibujo, sobre la pantalla. En este caso, la superficie de la pantalla del dispositivo es como un lienzo (*canvas*).

Las interfaces del paquete *javax.microedition.lcdui* son:

- Choice – Una interfaz que describe una serie de elementos sobre los que el usuario puede escoger
- CommandListener – Una interfaz de monitorización de eventos (*listener*), para gestionar comandos a alto nivel
- ItemStateListener – Una interfaz de monitorización de eventos (*listener*) para gestionar los eventos sobre el estado de los elementos

Además de las interfaces antes enumeradas, el paquete lcdui, contiene también las siguientes clases:

- Alert – Una pantalla que muestra información al usuario y después desaparece.
- AlertType – Representa diferentes tipos de alertas, usadas junto con la clase Alert
- Canvas – Una superficie (lienzo) para dibujar a bajo nivel. Permite dibujar las pantallas que mostrará el dispositivo, a bajo nivel

- ChoiceGroup – Presenta un grupo de elementos seleccionables. Se usa junto con el interfaz Choice
- Command – Representa un comando a alto nivel, que puede ser generado desde el MIDlet
- DateField – Representa una fecha y una hora que pueden ser editadas
- Display – Representa la pantalla del dispositivo y acoge la recuperación de las acciones del usuario
- Displayable – Es un componente abstracto que se puede mostrar por pantalla. Es una superclase para otros componentes.
- Font – Representa un tipo de letra y las métricas asociadas al mismo
- Form – Es una pantalla que sirve como contenedor para otros componentes gráficos de usuario
- Gauge – Muestra un valor, como un porcentaje dentro de una barra
- Graphics – Encapsula operaciones gráficas bidimensionales, como son el dibujo de líneas, elipses, texto e imágenes
- Image – Representa una imagen
- ImageItem – Es un componente que soporta la presentación (*layout*) de una imagen
- Item – Es un componente que representa un elemento con una etiqueta
- List – Es un componente que consiste en una lista de opciones para seleccionar
- Screen – Representa una pantalla completa a alto nivel, y sirve como clase base para todos los componentes del interfaz de usuario de MIDP
- StringItem – Un componente que representa un elemento consistente en una etiqueta y una cadena de texto asociada

- TextBox – Un tipo de pantalla que soporta la visualización y edición de texto
- TextField – Un componente que soporta la visualización y edición de texto. A diferencia de un TextBox, este componente puede ser añadido a un form, junto con otros componentes
- Ticker – Un componente que muestra texto moviéndose horizontalmente, como una marquesina

2.3.5 El paquete javax.microedition.io

El CLDC, descarga el trabajo con la red y la entrada/salida en el paquete *java.io* y en el *Generic Connection Framework* (GCF). El API de MIDP parte de esta base, añadiendo la interfaz *HttpConnection*, que pertenece al paquete *javax.microedition.io*.

2.3.6 El paquete javax.microedition.rms

El API de MIDP, presenta un sistema de persistencia basado en registros para almacenar información. Este sistema, conocido como *Record Management System* (RMS). Las interfaces del paquete *javax.microedition.rms* son:

- RecordComparator – Para comparar dos registros
- RecordEnumeration – Para iterar sobre los registros
- RecordFilter – Para filtrar registros de acuerdo a un registro
- RecordListener – Un monitorizador de eventos usado para controlar los cambios en los registros

A parte de estas interfaces, tenemos una serie de clases, de las que debemos destacar la clase *RecordStore*, que representa un *record store* (almacén de registros). Las clases del paquete *javax.microedition.rms* son:

- InvalidRecordException – Se lanza cuando una operación falla porque el identificador del registro es invalido

- RecordStore – Representa un “almacén de registros”
- RecordStoreException – Se lanza cuando una operación falla por un error general
- RecordStoreFullException – Se lanza cuando una operación falla porque el *record store* está completo
- RecordStoreNotFoundException – Se lanza cuando una operación falla porque el *record store* no se ha podido localizar
- RecordStoreNotOpenException – Se lanza cuando se realiza una operación sobre un *record store* cerrado

2.4 Ejemplo de uso de APIs de MIDP y J2SE

A continuación vamos a ver un ejemplo de un MIDlet que usa parte del API de MIDP y algunas clases de J2SE. En concreto, de J2SE vamos a utilizar las clases de envoltura de tipos, la clase de entorno de ejecución (*Runtime*) y una clase para el manejo de fechas.

La aplicación es un MIDlet que muestra por pantalla información general sobre el dispositivo en el que se ejecuta.

```
package com.vitike.cursoj2me;

/**
 * <p>Title: CursoJ2ME</p>
 * <p>Description: Clases del Curso de J2ME</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Manuel J. Prieto
 * @version 1.0
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
```

```
public class InfoDispositivo extends MIDlet implements CommandListener{

    private Command salir;
    private Display display;
    private Form pantalla;

    public InfoDispositivo() {
        // Recuperamos el objeto Display
        display = Display.getDisplay(this);

        // Creamos el comando de salida
        salir = new Command("Salir", Command.EXIT, 2);

        // Creamos el "Form" de la pantalla principal
        pantalla = new Form("InfoDispositivo");

        // Obtenemos la hora
        Calendar calendar = Calendar.getInstance();
        String hora = Integer.toString(calendar.get(Calendar.HOUR_OF_DAY)) + ":" +
            Integer.toString(calendar.get(Calendar.MINUTE)) + ":" +
            Integer.toString(calendar.get(Calendar.SECOND));

        // Obtenemos la memoria total y la libre
        Runtime runtime = Runtime.getRuntime();
        String memTotal = Long.toString(runtime.totalMemory());
        String memLibre = Long.toString(runtime.freeMemory());

        // Obtenemos las propiedades de la pantalla
        String color = display.isColor() ? "Sí" : "No";
        String numColores = Integer.toString(display.numColors());

        // Creamos las cadenas de información y las añadimos a la pantalla
        pantalla.append(new StringItem("", "Hora: " + hora + "\n"));
        pantalla.append(new StringItem("", "Mem Total: " + memTotal + " b\n"));
        pantalla.append(new StringItem("", "Mem Libre: " + memLibre + " b\n"));
        pantalla.append(new StringItem("", "Color: " + color + "\n"));
        pantalla.append(new StringItem("", "Colores: " + numColores + "\n"));
```

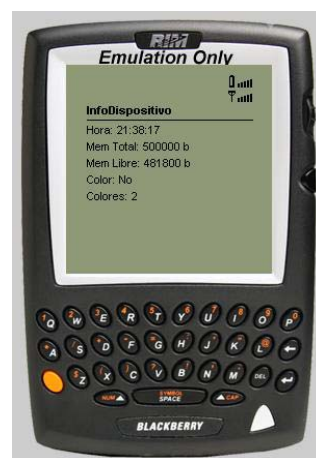
```
// Establecemos el comando de salida
pantalla.addCommand(salir);
pantalla.setCommandListener(this);
}

public void startApp() throws MIDletStateChangeException{
    // Establecemos el "Display" actual a nuestra pantalla
    display.setCurrent(pantalla);
}

public void pauseApp(){
}

public void destroyApp(boolean incondicional){
}

public void commandAction (Command c, Displayable s){
    if (c == salir){
        destroyApp(false);
        notifyDestroyed();
    }
}
}
```



3 MIDlets GRÁFICOS

A pesar de las limitaciones de las pantallas de los dispositivos móviles, el uso de gráficos en las aplicaciones suele ser necesario, útil y mejora las mismas. Los juegos, por ejemplo, son uno de los principales grupos de aplicaciones que se desarrollan para dispositivos móviles y habitualmente necesitan programación gráfica a bajo nivel. Hablamos de gráficos a bajo nivel para diferenciar este tipo de gráficos, de los componentes estándar del GUI de MIDP.

El sistema de coordenadas para los gráficos de MIDP, sitúa el punto (0,0) en la esquina superior-izquierda de la pantalla. La coordenada *x*, crecen hacia la derecha y la coordenada *y* crece hacia abajo. Los valores de las coordenadas siempre son enteros positivos.

3.1 La clase *Graphics*

Esta clase es conocida de los programadores en Java. Contiene la mayoría de funcionalidad para dibujar a *bajo nivel*. La clase *Graphics* proporciona la capacidad de dibujar gráficas primitivas (líneas, rectángulos...), texto e imágenes tanto en la pantalla como en un buffer en memoria. El método *paint()* de los MIDlet, tiene como parámetro un objeto de tipo *Graphics* y este será usado para dibujar. Ya que el método *paint* proporciona un objeto *Graphics*, nunca deberemos crearlo explícitamente.

La clase *Graphics* tiene una serie de atributos que determinan cómo se llevan a cabo las diferentes operaciones. El más importante de estos atributos es el atributo *color*, que determina el color usado en el dibujo del resto de elementos. Este atributo se puede modificar con el método *setColor()*. Este método recibe tres parámetros enteros, que especifican el color en función de los tres colores primarios. De forma análoga a la función del método *setColor()*, tenemos el método *setGrayScale()*, que toma como parámetro un único valor entero, que establece el grado de gris en un rango que va desde 0 hasta 255.

Los objetos de tipo *Graphics*, contienen también un atributo denominado *font* y que determina el tamaño y la apariencia del texto. Este atributo se modifica a través del método *setFont()*.

3.2 Primitivas gráficas

Las primitivas gráficas que podemos dibujar son: líneas, rectángulos y arcos. La clase *Graphics* proporciona métodos para realizar estas operaciones. Además, nos proporciona métodos para rellenar las áreas.

La línea es el ejemplo más simple de primitiva gráfica. El método que nos permite dibujar una línea es:

```
void drawLine (int x1, int y1, int x2, int y2)
```

Los parámetros *x1* e *y1*, indican el punto de comienzo de la línea y los parámetros *x2* e *y2*, indican el punto final. El API de MIDP nos permite cambiar el estilo de la línea a través del método *setStrokeStyle()*. Este método acepta un valor de los siguientes:

- *Graphics.SOLID* – Línea sólida
- *Graphics.DOTTED* – Línea de puntos

El método para dibujar rectángulos es:

```
void drawRect (int x, int y, int width, int height)
```

Los parámetros *x* e *y* especifican la localización de la esquina superior-izquierda del rectángulo y los parámetros *width* y *height* especifican el ancho y el alto respectivamente del rectángulo.

También existe un método para dibujar rectángulos con las esquinas redondeadas:

```
void drawRoundRect (int x, int y, int width, int height, int arcWidth,
```

```
int arcHeight)
```

Este método, contiene también los parámetros *arcWidth* y *arcHeight*, que configuran el arco de las esquinas del rectángulo.

La clase *Graphics* también tiene métodos para dibujar rectángulos rellenos, es decir, rectángulos de un color determinado, en lugar de dibujar sólo su contorno. El color de relleno será el color configurado en el atributo *color*. Los métodos para hacer esto son *fillRect()* y *fillRoundRect()*.

Otra primitiva gráfica son los arcos. Un arco es una sección de un óvalo. El método que dibuja un óvalo es el siguiente:

```
void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)
```

Los primeros cuatro parámetros definen el óvalo del que el arco forma parte y los últimos dos parámetros definen el arco como una sección del óvalo. Para dibujar un óvalo, tendremos que utilizar la clase *drawArc*, especificando un ángulo de 360°. El método para dibujar un arco relleno de color es *fillArch()*.

3.3 Escribiendo texto

El texto que se escribe usando la clase *Graphics*, estará configurado por el atributo *font* antes mencionado. Este atributo se cambia con el método:

```
void setFont (Font font)
```

El objeto *Font* indica el tipo de fuente, el estilo y el tamaño del texto. Los estilos pueden ser:

- STYLE_PLAIN – Texto normal

- STYLE_BOLD – Texto en negrita
- STYLE_ITALIC – Texto en cursiva
- STYLE_UNDERLINED – Texto subrayado

Los estilos pueden usarse combinados, salvo el primero de ellos, que especifica la ausencia de los otros tres.

Nunca crearemos un objeto de tipo *Font* usando un constructor. En lugar de hacer esto, se usa el método *getFont()*, que tiene el siguiente formato:

```
static Font getFont(int face, int style, int size)
```

Los valores que configuran la fuente de texto son constantes. Para el tipo de letra, podemos especificar:

- FACE_SYSTEM
- FACE_MONOSPACE
- FACE_PROPORTIONAL

El tamaño del texto es indicado con una de las siguientes constantes:

- SIZE_SMALL
- SIZE_MÉDIUM
- SIZE_LARGE

Un ejemplo de configuración de un tipo de texto sería:

```
Font f = Font.getFont(Font.FACE_MONOSPACE,  
Font.STYLE_BOLD|Font.STYLE_UNDERLINED, Font.SIZE_LARGE);
```

Una vez que tenemos la fuente usamos el método *setFont()* para que se use la misma en las operaciones sucesivas.

El método para dibujar el texto es el siguiente:

```
void drawstring (String str, int x, int y, int anchor)
```

El primer parámetro es el texto a escribir. Los siguientes dos parámetros (x e y) especifican la localización del texto. El significado específico de esta localización es determinada por el último parámetro, *anchor*. Para ayudar a posicionar el texto y las imágenes, el API de MIDP introduce *anchor points* (puntos de anclaje), que ayudan a posicionar texto e imágenes fácilmente. Un punto de anclaje está asociado con una constante vertical y con otra horizontal, que especifican la posición vertical y horizontal del texto con respecto a dicho punto de anclaje. Las constantes horizontales usadas para describir el punto de anclaje son:

- LEFT
- HCENTER
- RIGHT

Las constantes disponibles para el punto vertical del anclaje son:

- TOP
- BASELINE
- BOTTOM

Por ejemplo, para escribir un texto en el centro y en la parte de arriba de la pantalla, el comando sería:

```
g.drawstring ("Este es el texto", getWidth()/2, 0, Graphics.HCENTER  
| Graphics.TOP);
```

A parte del método *drawString()*, tenemos algunos métodos más para dibujar texto. Los métodos *drawChar()* y *drawChars()* son usados para dibujar caracteres de texto individuales:

```
void drawChar (char character, int x, int y, int anchor)
void drawChars (char[] data, int offset, int length, int x, int y, int
anchor)
```

También podemos usar el método *drawSubstring*, que nos permite escribir una parte de un cadena:

```
void drawSubstring (String str, int offset, int len, int x, int y, int
anchor)
```

3.4 Dibujando imágenes

Las imágenes son objetos gráficos rectangulares compuestas de *píxels* de color o grises. Antes de dibujar una imagen, debemos cargarla. Típicamente las imágenes están almacenadas en ficheros externos. Las imágenes son cargadas y creadas usando un método especial de la clase *Image*, denominada *createImage()*:

```
Public static Image createImage (String name) throws IOException
```

El parámetro indica el nombre del fichero que contiene la imagen. El método *createImage()* retorna un objeto de tipo *Image* que puede ser usado dentro del MIDP. También es posible crear un objeto *Image* en “blanco”, usando una versión diferente del método *createImage* que acepta el ancho y el alto de la imagen. La clase *Image* representa una imagen gráfica, como puede ser un fichero PNG, GIF o JPEG y proporciona método para recuperar el ancho y el alto de dicha imagen. La clase *Image* también incluye un método para recuperar un objeto *Graphics*, que nos permite dibujar directamente sobre la imagen.

La clase *Graphics* proporciona un método para dibujar las imágenes:

```
boolean drawImage (Image img, int x, int y, int anchor)
```

Un ejemplo del proceso sería:

```
public void paint(Graphics g){
    // Limpiamos la pantalla
    g.setColor(255, 255, 255); // Blanco
    g.fillRect(0, 0, getWidth(), getHeight());

    // Creamos y cargamos la imagen
    Image img = Image.createImage("imagen.png");

    // Dibujamos la imagen
    g.drawImage(img,                getWidth()/2,                getHeight()/2,
Graphics.HCENTER|Graphics.VCENTER);
}
```

3.5 Ejemplo de uso de los métodos gráficos

A continuación vamos a desarrollar un sencillo MIDlet que utiliza los métodos que hemos descrito, para realizar un pequeño dibujo sobre la pantalla. El MIDlet se compondrá de dos clases. Una clase será el MIDlet propiamente dicho y la otra será el objeto *Canvas* que nos permite dibujar.

```
package com.vitike.cursoj2me;

/**
 * <p>Title: CursoJ2ME</p>
 * <p>Description: Clases del Curso de J2ME</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
```

```
* @author Manuel J. Prieto
* @version 1.0
*/

import javax.microedition.lcdui.*;
import java.io.*;

class CanvasUsal extends Canvas{

    public void paint(Graphics g){
        // Cargamos la imagen
        int ancho = 0;
        int alto = 0;
        try{
            Image img = Image.createImage("/usal.png");

            // Limpiamos la pantalla
            g.setColor(255, 255, 255);
            g.fillRect(0, 0, getWidth(), getHeight());

            // Ponemos la imagen en el centro de la pantalla
            ancho = getWidth();
            alto = getHeight();
            g.drawImage(img,    ancho/2,    4,    Graphics.HCENTER    |
Graphics.TOP);

        } catch (IOException ex){
            System.err.println("Error cargando imágenes");
        }
        g.setColor(0, 0 ,0);
    }
}
```

```
// Creamos un recuadro para la pantalla
g.drawRect(1, 1, ancho-3, alto-3);

g.setColor(255, 0 ,0);
// Escribimos la url de USAL
Font f = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
Font.SIZE_LARGE);
g.setFont(f);
g.drawString("www.usal.es", ancho/2, alto, Graphics.HCENTER |
Graphics.BOTTOM);
}
}
```

```
package com.vitike.cursoj2me;

/**
 * <p>Title: CursoJ2ME</p>
 * <p>Description: Clases del Curso de J2ME</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Manuel J. Prieto
 * @version 1.0
 */

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class Usal extends MIDlet implements CommandListener{

    private Command salir;
```

```
private Display display;
private CanvasUsal pantalla;

public Usal() {
    // Recuperamos el objeto Display
    display = Display.getDisplay(this);

    // Creamos el comando de salida
    salir = new Command("Salir", Command.EXIT, 2);

    // Creamos la pantalla principal
    pantalla = new CanvasUsal();

    // Establecemos el comando de salida
    pantalla.addCommand(salir);
    pantalla.setCommandListener(this);
}

public void startApp() throws MIDletStateChangeException {
    // Establecemos el "Display" actual a nuestra pantalla
    display.setCurrent(pantalla);
}

public void pauseApp(){
}

public void destroyApp(boolean incondicional){
}

public void commandAction(Command c, Displayable s){
    if (c == salir){
```

```
destroyApp(false);  
notifyDestroyed();  
}  
}  
}
```



4 COMPONENTES DE INTERFAZ DE USUARIO

El API de MIDP nos proporciona una serie de componentes que nos permitirán construir las interfaces de usuario de forma sencilla. Por supuesto, aunque estos componentes son potentes para el entorno que nos ocupa, siempre hay que tener presente las limitaciones de los dispositivos móviles en cuanto a pantalla y en cuanto a interacción con el usuario.

Como hemos visto en el código presentado hasta el momento, siempre debemos recoger el objeto de tipo *Display* que gestiona lo que muestra la pantalla del dispositivo:

```
display = Display.getDisplay(this)
```

La clase *Display* se define dentro del paquete *javax.microedition.lcdui* e incluye una serie de métodos útiles para gestionar que queremos presentar en la pantalla del dispositivo y la interacción con el usuario. La visualización de un MIDlet es controlado por un objeto *displayable* (visible), que es de tipo *Displayable*. Un objeto *displayable* es un componente GUI especial que puede presentarse en la pantalla del dispositivo.

Sólo hay dos clases que hereden directamente de la clase *Displayable*:

- Canvas – Representa una superficie general de dibujo
- Screen – Clase base para otros componentes GUI, como la clase *Form*

La clase *Form*, que se ha visto anteriormente en algunos fragmentos de código, actúa como contenedor para otros componentes.

Cada MIDlet tiene su instancia de la clase *Display* y esta instancia proporciona acceso a la pantalla y a los controles de interacción con el usuario, típicamente el teclado. Este objeto, estará disponible desde el comienzo del método *startApp()*, que es llamado cuando el MIDlet es lanzado. El método estático *Display.getDisplay()* es

utilizado para recuperar el objeto *Display* del MIDlet. El objeto *Display* es válido hasta el final de la ejecución del método *destroyApp()*.

Para ser visible por pantalla, un MIDlet debe crear un objeto que derive de la clase *Displayable*. Esta clase será entonces la responsable de dibujar la pantalla correspondiente. La clase *Displayable* es una clase abstracta, por lo que no puede ser usada directamente. En su lugar, deberemos usar una de sus clase derivadas (*Canvas* o *Screen*).

Para que un objeto *Displayable* sea visible, deberemos invocar al siguiente método:

```
Void setCurrent(Displayable next)
```

Lo habitual será cambiar de pantalla a menudo en los MIDlets, por lo que usaremos este método con cierta asiduidad. También tenemos una función para recuperar el objeto *Displayable* activo en cada momento:

```
Displayable getCurrent()
```

La clase *Display* posee una serie de métodos para recoger la información sobre las características del dispositivo. Los métodos *isColor()* y *numColors()*, nos sirven para recuperar la información sobre las capacidades de color del dispositivo.

4.1 Screens y Forms

Como se ha mencionado anteriormente, la clase *Displayable* posee dos subclases:

- Canvas – Representa una superficie general de dibujo
- Screen – Clase base para otros componentes GUI, como la clase *Form*

La clase *Canvas* está pensada para usar directamente, es decir, dibujar sobre ella y utilizarlas, sin más componentes. En cambio, la clase *Screen* esta diseñada para representar una pantalla que pueda interactuar con el usuario.

Las pantallas de los MIDlets suelen estar modeladas por la clase *Screen*, que es una clase abstracta. Esta clase proporciona la funcionalidad básica para una pantalla, que principalmente consiste en un título que aparecerá en la parte de arriba de la pantalla del dispositivo. Se puede modificar este título con los siguientes métodos:

```
String getTitle()  
Void setTitle (String s)
```

Además del atributo de título, las pantallas también pueden tener una línea de texto en movimiento, que aparecerá sobre la pantalla. Este texto, se configura a través de la clase *Ticker*, que es un componente GUI de MIDP. Este elemento se puede manejar con los siguientes métodos:

```
Ticker getTicker()  
void setTicker (Ticker ticker)
```

Hay cuatro clases que heredan de la clase *Screen* que como hemos dicho es una clase abstracta. Estas clases son:

- Form – Sirve como contenedor para construir interfaces de usuario con otros componentes
- Alert – Muestra un mensaje (y opcionalmente una imagen) al usuario
- List – Muestra una lista de elementos sobre los que el usuario puede seleccionar

- TextBox – Proporciona un simple editor de texto

Estas clases son componentes de interfaz de usuario, que están diseñados para ocupar toda la pantalla del dispositivo.

Como se ha mencionado, un *Form* es capaz de contener otros componentes GUI. Sólo los componentes que son heredados de la clase *Item*, pueden ser añadidos a un *Form*. La clase *Item* es un componente que a diferencia de los anteriores, no ocupa toda la pantalla. Lo habitual es crear un *Form* y añadir varios elementos de tipo *Item* para componer una interfaz de usuario personalizada. Para añadir los *Items*, se utiliza el siguiente método:

```
int append (Item item)
```

Cada *Item* añadido a un *Form* posee un índice dentro del *Form* y este índice es el número que retorna el método *append*. Para hacer referencia a un *Item* dentro de un *Form*, usaremos este índice.

Para eliminar un *Item* de un *Form* podemos usar el siguiente método:

```
void delete(int index)
```

Se puede insertar un *Item* en un punto concreto del *Form* con el siguiente método:

```
void insert(int index, Item item)
```

Para modificar un *Item*, debemos usar el siguiente método:

```
void set(int index, Item item)
```

Para recuperar un determinado *Item* del *Form*, disponemos del método:

```
Item get(int index)
```

Finalmente, para saber el número total de *Items* en un *Form* tenemos el método:

```
int size()
```

Otro punto interesante en el trabajo con *Forms* es la detección de los cambios en los *Items*. Para controlar estos cambios, debemos crear un *listener* del estado del *Item*. La interfaz *ItemStateListener* describe un solo método: *itemStateChanged()*, que es usado para responder a los cambios. El prototipo de este método es:

```
void itemStateChanged(Item item)
```

Para responder al evento de cambio de un *Item*, debemos implementar la interfaz *ItemStateListener* en nuestro MIDlet e implementar el método *itemStateChanged()*. También debemos registrar nuestro MIDlet como *listener* usando el método *setItemStateListener()* que define la clase *Form*. Un ejemplo de este código sería:

```
Form.setItemStateListener(this);
```

Cualquier componente GUI que deseemos usar dentro de un *Form* debe heredar de la clase *Item*. Varias clases dentro del API de MIDP derivan de esta clase, lo que nos da una cierta flexibilidad para diseñar *Form* e interfaces de usuario. Estas clases son:

- StringItem
- ImageItem
- TextField

- DateField
- ChoiceGroup
- Gauge

4.2 La clase *Alert*

Como ya se ha mencionado, la clase *Alert* hereda de la clase *Screen* e implementa una pantalla que muestra un texto informativo al usuario. Esta información se muestra durante un determinado espacio de tiempo o hasta que el usuario seleccione un comando, según se configure. El constructor de la clase *Alert* es el siguiente:

```
Alert(String title, String alertText, Image alertImage, AlertType  
alertType)
```

El título de la alerta es un texto que aparece en la parte superior de la pantalla, mientras que el texto de la alerta actúa como el cuerpo del mensaje de alerta. Con el tercer parámetro podemos indicar una imagen que se muestre en la alerta o *null* si no queremos mostrar ninguna imagen. El último parámetro indica el tipo de alerta. Los tipos de alertas se describen en la clase *AlertType* y puede ser:

- ALARM
- CONFIRMATION
- ERROR
- INFO
- WARNING

Los tipos de alertas son usados normalmente junto con la clase *Alert*, pero también se puede usar como elemento independiente:

```
AlertType.ERROR.playSound(display);
```

Este código usa el objeto incluido dentro de *AlertType* para hacer sonar un sonido de error.

Por defecto, las alertas se mostrarán durante unos segundos y desaparecerán automáticamente. Este periodo de visualización de la alerta se puede configurar con el siguiente método:

```
void setTimeout(int time)
```

También podemos mostrar una alerta sin *timeout* y configurada para que desaparezca cuando el usuario seleccione un comando. Para hacer esto, debemos pasar la constante *Alert.FOREVER* al método *setTimeout()* que acabamos de ver.

Para mostrar una alerta al usuario, debemos hacer que la pantalla actual del MIDlet sea esta. Esto se hace con el siguiente código:

```
display.setCurrent(alert);
```

4.3 La clase *List*

La clase *List* hereda de la clase *Screen*, pero presenta una funcionalidad más amplia que la clase *Alert*. La clase *List* proporciona una pantalla que contiene una lista de elementos sobre los que el usuario puede seleccionar. Esta clase implementa la interfaz *Choice*, que define constantes que describen tres tipos básicos de listas de opciones:

- EXCLUSIVE – Una lista que permite seleccionar un solo elemento a la vez
- IMPLICIT – Un lista *EXCLUSIVE* en la que el elemento seleccionado como respuesta a un comando
- MÚLTIPLE – Una lista que permite seleccionar uno o más elementos a la vez

Los constructores de la clase *List* son:

```
List(String title, int listType)  
List(String title, int listType, String[] stringElements, Image[]  
imageElements)
```

El primer constructor es el más sencillo y acepta un título para la lista y el tipo de la lista. El segundo constructor va un poco más lejos y nos permite inicializar los elementos de la lista.

Una vez creada la lista se puede interactuar con ella usando métodos de la clase *List* que son implementados según al interfaz *Choice*. Para recuperar el elemento seleccionado en una lista *exclusive* o *implicit*, podemos usar el método *getSelectedIndex()*, que devuelve el índice del elemento seleccionado dentro de la lista. Para listas de tipo *múltiple*, debemos usar el método *getSelectedFlags()*, que devuelve una matriz cuyo tamaño es el tamaño de la lista. Los valores de esta matriz son de tipo *boolean* e indican si el elemento en correspondiente está seleccionado (*true*) o no (*false*). El prototipo de este método es:

```
int getSelectedFlags(Boolean[] selectedArray)
```

Se pueden activar elementos como seleccionados dentro de una lista con los siguientes métodos:

```
void setSelectedIndex (int index, boolean selected)  
void setSelectedFlags (Boolean[] selectedArray)
```

El índice de un elemento en la lista es la clave para obtener su valor y determinar su estado. Por ejemplo, el método *getString()* toma como parámetro del índice del elemento en la lista y devuelve el texto del elemento. Podemos determinar el estado de un elemento usando el

método *isSelected()*, que recibe como parámetro el índice del elemento.

La clase *List* también incluye varios métodos para manipular la lista añadiendo y eliminando elementos. El método *append()* inserta un nuevo elemento al final de la lista. El método *insert()* inserta un elemento en la lista en un punto determinado de esta. Finalmente, el método *delete()* borra un determinado elemento de lista. Para recuperar el número de elementos de la lista, disponemos del método *size()*.

4.4 La clase *TextBox*

La clase *TextBox* implementa un componente de edición de texto, que ocupa toda la pantalla. El constructor de la clase es:

```
TextBox(String title, String text, int maxSize, int constraints)
```

El parámetro título es un texto que aparecerá en la parte superior de la pantalla, mientras que el parámetro texto es usado para inicializar el texto que contendrá el *TextBox*, si es necesario. El parámetro *maxSize* especifica el número máximo de caracteres de texto que pueden ser introducidos en el *TextBox*. Por último, el parámetro *constraints* describe las limitaciones a aplicar sobre el texto. Estas limitaciones son especificadas según las constantes definidas en la clase *TextField*:

- ANY – No hay limitaciones en el texto
- EMAILADDR – Sólo se puede introducir una dirección de correo electrónico
- NUMERIC – Sólo se puede introducir un valor entero
- PASSWORD – El texto es protegido para que no sea visible
- PHONENUMBER – Sólo se puede introducir un número de teléfono

- URL – Sólo se puede introducir una URL

Un ejemplo de uso sería:

```
TextBox box = new TextBox("NOTAS", "Nota: ", 256, TextField.ANY);
```

Una vez que la caja de texto ha sido creada, podemos trabajar con el texto que contiene. Para recuperar el texto, podemos usar el método *getString()*, que devuelve un *String*. También podemos recuperar el texto como una matriz de caracteres a través del siguiente método:

```
int getChars(char[] data);
```

El valor retornado es el número de caracteres copiados. Con los siguientes métodos podemos establecer el texto del *TextBox*:

```
void setString(String text)
void setChars(char[] data, int offset, int length)
void insert(String src, int position)
void insert(char[] data, int offset, int length, int position)
```

Hay varios métodos más para manipular el texto. El método *delete()* nos permite borrar el texto seleccionado. El método *size()* nos devuelve el número de caracteres que hay actualmente en el *TextBox*. También podemos saber el número máximo de caracteres que puede almacenar el *TextBox* con el método *getMaxSize()*.

4.5 La clase *Ticker*

La clase *Ticker*, muestra un texto desplazándose por la pantalla. Esta clase es única dentro de las clases del GUI de MIDP, porque no hereda ni de la clase *Screen* ni de la clase *Item*. El *Ticker* está

diseñado para usarlo dentro de un *Screen*. La clase *Screen* coloca el *ticker* en la parte superior de la pantalla. El constructor de la clase es:

```
Ticker(String str)
```

Para introducir un *ticker* en una pantalla, primero creamos el *ticker* y luego lo incluimos en la pantalla con el método *setTicker()*:

```
String str = Calendar.getInstance().toString();
Ticker ticker = new Ticker(str);
TextBox box = new TextBox("NOTAS", "Nota: ", 256, TextField.ANY);
box.setTicker(ticker);
```



Se puede recuperar y establecer el texto del *ticker* a través de los métodos *getString()* y *setString()*.

4.6 La clase *StringItem*

La clase *StringItem* hereda de la clase *Item*. Esta clase es significativa porque proporciona el soporte necesario para mostrar un componente dentro de un *Form*. La clase *StringItem* representa un elemento que contiene una cadena de texto. Esta clase es usada

principalmente para mostrar texto a un usuario en un *Form*. Está compuesto de dos partes, una etiqueta y un texto. A pesar de que la etiqueta (que es texto) y el texto son almacenados y mostrados por separado, el usuario no puede editar el mismo. El constructor de esta clase es:

```
StringItem(String label, String text)
```

Para mostrar un solo texto, podemos indicar el parámetro *label* como una cadena vacía. La clase *StringItem* sólo contiene dos métodos, usados para recuperar y establecer el texto:

```
String getText()  
void setText(String text)
```

4.7 La clase *ImageItem*

La clase *ImageItem* es similar a la clase *StringItem* pero está diseñada para mostrar imágenes en lugar de texto. El constructor de esta clase es el siguiente:

```
ImageItem(String label, Image img, int layout, String altText)
```

Como vemos en el constructor, tenemos un texto, que podemos indicar como una cadena vacía si no queremos que muestre ningún texto. La imagen es el segundo parámetro y el parámetro *layout* determina cómo la imagen es posicionada en el *Form* respecto a otros elementos. El último parámetro, especifica un texto para mostrar en lugar de la imagen.

El *layout* de la imagen es determinado por constantes definidas en la clase *ImageItem*:

- LAYOUT_DEFAULT – Posiciona la imagen usando lo especificado por defecto en el *form*.
- LAYOUT_LEFT – Alinea la imagen en el borde izquierdo del *form*
- LAYOUT_RIGHT – Alinea la imagen en el borde derecho del *form*
- LAYOUT_CENTER – Centra la imagen horizontalmente
- LAYOUT_NEWLINE_BEFORE – Empieza una nueva línea para la imagen
- LAYOUT_NEWLINE_AFTER – Empieza una nueva línea de items después de la imagen

Para comprender cómo funciona el *layout*, hay que comprender que los elementos del *form* son mostrados en líneas en la pantalla. El valor *LAYOUT_DEFAULT* no puede usarse con cualquiera de los otros valores e indica que la imagen será dispuesta como cualquier otro elemento del *form*. Los valores *LAYOUT_LEFT*, *LAYOUT_RIGHT* y *LAYOUT_CENTER*, indican cómo se posiciona la imagen horizontalmente. Estos tres últimos valores pueden ser combinados con las constantes *LAYOUT_NEWLINE_BEFORE* y *LAYOUT_NEWLINE_AFTER* para indicar una posición más exacta.

Antes de crear el objeto *ImageItem* debemos crear el objeto *Image* que se mostrará. El método estático *createImage()* de la clase *Image* gestiona esta tarea. Al crear la imagen, será necesario controlar la excepción *IOException* que puede ser lanzada.

```
package com.vitike.cursoj2me;

/**
 * <p>Title: Curso de J2ME</p>
 * <p>Description: Clases del Curso de J2ME</p>
 * <p>Copyright: Copyright (c) 2002</p>
```

```
* <p>Company: </p>
* @author Manuel J. Prieto
* @version 1.0
*/

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class ImgItem extends MIDlet implements CommandListener {
    private Command salir;
    private Display display;
    private Form pantalla;

    public ImgItem(){

        // Recogemos el objeto Display
        display = Display.getDisplay(this);

        // Creamos el comando de salida
        salir = new Command("Salir", Command.EXIT, 2);

        // Creamos el "Form" de la pantalla principal
        pantalla = new Form("J2ME");

        // Configuramos la pantalla
        Image duke = null;
        try{
            duke = Image.createImage("/duke.png");
        }catch (java.io.IOException ex){
            System.err.println("Excepción: " + ex);
        }
    }
}
```

```
ImageItem    imgIt    =    new    ImageItem("",    duke,
ImageItem.LAYOUT_DEFAULT, "Duke");
String txt = "Duke";
String txt2 = "Curso J2ME";
pantalla.append(imgIt);
pantalla.append(txt + "\n");
pantalla.append(txt2);

// Añadimos y configuramos el comando de salida
pantalla.addCommand(salir);
pantalla.setCommandListener(this);
}

public void startApp() throws MIDletStateChangeException{
    // Establecemos el "Display" actual a nuestra pantalla
    display.setCurrent(pantalla);
}

public void pauseApp(){
}

public void destroyApp (boolean incondicional){
}

public void commandAction(Command c, Displayable s){
    if (c == salir){
        destroyApp(false);
        notifyDestroyed();
    }
}
}
```



4.8 La clase *TextField*

La clase *TextField* proporciona un editor de texto diseñado para ser usado dentro de los *forms*. Esta es la principal diferencia con respecto a la clase *TextBox*. A pesar de esta diferencia, estas dos clases tienen su parecido. De hecho, se puede interactuar con el texto en la clase *TextField* usando los mismo métodos que se especificaron anteriormente en la clase *TextBox*. El constructor de la clase *TextField* es:

```
TextField(String label, String text, int maxSize, int constraints)
```

El primer parámetro establece la etiqueta que se muestra junto al componente y el segundo el texto utilizado para inicializar el elemento. El parámetro *maxSize* indica el máximo número de caracteres que pueden ser introducidos. El último parámetro, de forma similar a lo indicado para la clase *TextBox*, indica las restricciones del texto a introducir. Como ya se indicó, los valores pueden ser:

- ANY – No hay limitaciones en el texto
- EMAILADDR – Sólo se puede introducir una dirección de correo electrónico

- NUMERIC – Sólo se puede introducir un valor entero
- PASSWORD – El texto es protegido para que no sea visible
- PHONENUMBER – Sólo se puede introducir un número de teléfono
- URL – Sólo se puede introducir una URL

4.9 La clase *DateField*

La clase *DateField* presenta una interfaz intuitiva para introducir fechas y horas. El constructor de esta clase es el siguiente:

```
DateField (String label, int mode)
```

A parte de este constructor, tenemos otro que nos permite indicar también la zona horaria. Si no indicamos dicho dato, se usará la configurada en el dispositivo. El constructor presentado, como es habitual, recoge en su primer parámetro la etiqueta a mostrar con el elemento. El parámetro *mode* indica el modo de funcionar del componente. Estos modos de funcionamiento son:

- DATE – Se introduce solo una fecha (día, mes y año)
- TIME – Se introduce solo una hora (horas y minutos)
- DATE_TIME – Se introduce el día y la hora

Estas constantes están definidas en la clase *DateField*. Esta clase también incluye los métodos *getDate()* y *setDate()* que permiten recoger la información almacenada en el componente.

4.10 La clase *ChoiceGroup*

Esta clase presenta una lista de elementos sobre los que el usuario puede seleccionar. Esta clase es similar a la clase *List*, pero la clase *ChoiceGroup* está pensada para ser usada dentro de un *Form*. A parte

de esto, no hay muchas más diferencias entre ambas. Los constructores de la clase *ChoiceGroup* son:

```
ChoiceGroup(String label, int choiceType)
ChoiceGroup(String label, int choiceType, String[] stringElements,
Image[] imageElements)
```

El primer parámetro de ambos constructores, indica la etiqueta que se mostrará para el grupo de opciones. El segundo parámetro es el tipo de la lista. El segundo constructor nos permite además inicializar las opciones y las imágenes asociadas a estas. El tipo de la lista, puede tener los mismos valores que lo indico para la clase *List*:

- EXCLUSIVE – Una lista que permite seleccionar un solo elemento a la vez
- IMPLICIT – Un lista *EXCLUSIVE* en la que el elemento seleccionado como respuesta a un comando
- MÚLTIPLE – Una lista que permite seleccionar uno o más elementos a la vez

4.11 La clase *Gauge*

La clase *Gauge* implementa una barra gráfica que puede ser usada para visualizar un valor dentro de un rango. Un objeto de tipo *Gauge* tiene un valor máximo que define el rango del objeto (de 0 al máximo) y un valor actual, que determina la configuración de la barra. La clase *Gauge* puede funcionar interactivamente y no interactivamente. Si funciona de forma no interactiva, simplemente se muestra la barra, mientras que si trabaja de forma interactiva, se usará la barra como método para introducir un valor por parte del usuario, manipulando la barra.

Para crear un objeto de tipo *Gauge*, tendremos que usar el siguiente constructor:

```
Gauge(String labe, boolean interactive, int maxValue, int initialValue)
```

El primer parámetro es la etiqueta asociada al componente, como es habitual. El segundo parámetro (*interactive*) indica si el objeto *Gauge* es interactivo o no interactivo. Los dos últimos parámetros son obvios.

Tenemos dos parámetros para trabajar con el *Gauge*. Podemos acceder al valor actual y cambiar dicho valor:

```
int getValue()  
void setValue(int value)
```

También es posible manipular el valor máximo con los siguientes métodos:

```
int getMaxValue()  
void setMaxValue(int value)
```



5 GESTIÓN DE COMANDOS

Los comandos proporcionan al usuario la capacidad de interactuar con el MIDlet seleccionando funcionalidades. Es el método por el que el usuario introduce “órdenes” en el MIDlet. Los comandos son modelados por la clase *Command*, que proporciona el siguiente constructor:

```
Command(String label, int commandType, int priority)
```

El primer parámetro del constructor es la etiqueta del comando. La etiqueta es importante porque será lo que el usuario verá para seleccionar el comando. El tipo del comando (*commandType*) se especifica a través de una serie de constantes y el último parámetro indica la prioridad del comando. Esta prioridad es importante porque determina cómo se muestran los comandos al usuario.

Las constantes que se pueden usar para definir el tipo del comando están definidas en la clase *Command* y son las siguientes:

- OK – Verifica que se puede comenzar una acción
- CANCEL – Cancela la acción
- STOP – Detiene la acción
- EXIT – Sale del MIDlet
- BACK – Envía al usuario a la pantalla anterior
- HELP – Solicita la ayuda
- ITEM – Un comando específico de la aplicación que es relativo a un determinado *item* de la pantalla actual
- SCREEN – Un comando específico de la aplicación que es relativo a la pantalla actual

Los tipos de comandos son necesarios porque los dispositivos proporcionan botones especiales para algunas funciones como el botón de retorno (*back*). Si está disponible en el dispositivo un botón

especial para la operación “volver”, un comando de tipo *BACK* será automáticamente asociado con este botón.

La prioridad del comando, como se ha dicho, es importante porque se usa para determinar la situación del comando para el acceso del usuario. Para comprender esta necesidad, basta saber que la mayoría de los teléfonos móviles tienen un número muy pequeño de botones disponibles para el uso en las aplicaciones.

Consecuentemente, sólo los comandos más importantes son mapeados a los botones del dispositivo directamente. Si un MIDlet tiene comando con una prioridad menor, será accesible, pero a través de un menú, no directamente desde un botón del dispositivo. El valor de prioridad de un comando disminuye con el nivel de importancia. Por ejemplo, el valor 1 es asignado con los comandos de máxima prioridad. La gestión de la posición de los comandos es realizada por el dispositivo, por lo que no tenemos un control directo sobre cómo se presentarán los comandos.

Una vez que el comando ha sido creado y añadido a la pantalla, este es visible para el usuario. Ahora vamos a ver cómo responder a estos comandos por parte del MIDlet. Para hacer esto, debemos configurar un *listener* (monitorizador) de comandos para la pantalla que contiene el comando, que típicamente será la propia clase del MIDlet. Esta clase debe implementar la interfaz *CommandListener*, que contiene un sólo método: *commandAction()*. El siguiente ejemplo muestra dos acciones necesarias para configurar el *listener*:

```
public class MIDlet1 extends MIDlet implements CommandListener{
...
pantalla.setCommandListener(this);
...
}
```

El método *commandAction()* tiene el siguiente prototipo:

```
void commandAction(Command c, Displayable s)
```

Como vemos, este método recibe un objeto de tipo *Command* y otro de tipo *Displayable*, que será la pantalla que contiene el comando. Ya que sólo existe un método *commandAction()* en cada MIDlet, es necesario comprobar cada comando en dicho método.

6 CONEXIÓN A REDES

Uno de los aspectos más interesantes de los MIDlets es su acceso a Internet a través de una conexión inalámbrica. Las clases e interfaces usadas en MIDP para acceso a red son muy similares a las usadas en J2SE y J2EE.

Como vimos anteriormente, el CLDC delega las funciones de red en el GCF. El propósito del GCF es proporcionar una abstracción de los servicios de red. En lugar de forzar a todos los dispositivos a soportar todos los protocolos de red, el GCF describe un marco de trabajo en red del que el dispositivo seleccionará los protocolos y servicios que es capaz de soportar.

En realidad, no es el dispositivo el que selecciona las capacidades de red que soportará, sino que esto lo hará el perfil del dispositivo. La responsabilidad del GCF es describir las capacidades de red disponibles en todos los dispositivos así como un sistema extensible en el que los diferentes perfiles de dispositivo pueden soportar un subconjunto de dichas capacidades.

El GCF describe una clase fundamental denominada *Connector*, que será usada para establecer todas las conexiones de red. Los tipos específicos de conexiones son modelados por las interfaces del GCF que son obtenidos a través de la clase *Connector*. Todas estas clases e interfaces están dentro del paquete *javax.microedition.io*. Tal y como vimos anteriormente, las interfaces son las siguientes:

- *Connection* – Una conexión básica que sólo puede ser abierta y cerrada
- *ContentConnection* – Un flujo (stream) de conexión que proporciona acceso a datos web
- *DatagramConnection* – Una conexión para manejar comunicaciones orientadas a paquetes
- *InputConnection* – Una conexión de entrada para las comunicaciones del dispositivo

- `OutputConnection` – Una conexión de salida para las comunicaciones del dispositivo
- `StreamConnection` – Una conexión en ambas direcciones para las comunicaciones del dispositivo
- `StreamConnectionNotifier` – Una conexión especial para notificaciones, que es usada para esperar que se establezca una conexión

Como se puede ver, estas interfaces son muy generales y no se acercan mucho a los protocolos reales. Ofrecen una arquitectura básica que es capaz de soportar un amplio rango de capacidades de conexión. Si bien los diferentes protocolos de red requieren código distinto, las interfaces del CLDC proporcionan una visión uniforme de todos ellos.

La descripción de las capacidades específicas en cada caso, la tenemos en los perfiles. Por ejemplo, el API de MIDP construye sobre estas interfaces el soporte para el trabajo en red de los MIDlets. El API de MIDP añade la interfaz *HttpConnection*, que proporciona un componente nuevo en el GCF para conexiones HTTP. Estas conexiones permiten a los MIDlets conectarse con páginas web. La especificación MIDP indica que las conexiones HTTP son el único tipo de conexiones obligatorio en las implementaciones de MIDP.

Siempre usaremos la clase *Connector* para establecer conexiones, independientemente del tipo de conexión. Todos los métodos de la clase *Connector* son estáticos. El más importante de ellos es el método *open()*. Las tres versiones de este método son:

```
static Connection open (String name) throws IOException
static Connection open (String name, int mode) throws IOException
static Connection open (String name, int mode, boolean timeouts)
throws IOException
```

El primer parámetro de estos métodos es la cadena de conexión. Este es el parámetro más importante porque determina el tipo de conexión que se realizará. La cadena de conexión tendrá el siguiente formato:

- Protocolo:Destino[;Parámetros]

La primera parte define el protocolo de red, como http o ftp. El destino es típicamente el nombre de la dirección de red. El último parámetro es una lista de parámetros asociados a la conexión. Algunos ejemplos de diferentes tipos de cadenas de conexión:

- HTTP – <http://www.usal.es/>
- Socket – socket://www.usal.es:80
- Datagram – datagram://:9000
- File – file:/datos.txt
- Port – comm:0;baudrate=9600

Debemos tener siempre presente que estos ejemplos son correctos, pero que el único que cuyo soporte está garantizado por la implementación de MIDP es el primero.

La segunda y la tercera versión del método *open()* esperan un segundo parámetro denominado modo (*mode*), que describe el modo de conexión. Este modo indica si la conexión es abierta para leer, escribir o para ambas cosas. Las siguientes constantes, definidas en la clase *Conector*, pueden ser usadas para describir el tipo de conexión:

- READ
- WRITE
- READ_WRITE

La primera versión del método *open()*, que no tiene el parámetro *mode*, usa el modo READ_WRITE.

La última versión del método *open()* acepta un tercer parámetro, que es un flag que indica si el código que ha llamado el método es capaz de controlar una excepción por tiempo excedido (*timeout*). Esta excepción es lanzada si el periodo de *timeout* para establecer la conexión falla. Los detalles de este periodo y cómo es gestionado, es labor del protocolo específico, por lo que no tenemos control sobre el mismo.

El método *open()* devuelve un objeto de tipo *Connection*, que es la interface básica para el resto de interfaces de conexión. Para usar un determinado tipo de interface de conexión, debemos convertir la interfaz *Connection* que devuelve el método *open()* al tipo apropiado:

```
StreamConnection          conn          =
(StreamConnection)Connector.open(http://www.usal.es/);
```

6.1 Entrada/Salida desde el MIDlet

La clase *Connector* y las interfaces asociadas a esta son usadas para obtener una conexión a la red. Una vez que la conexión está establecida, debemos usar las clases de entrada/salida para leer y escribir datos a través de la conexión. Estas clases son aportadas por el paquete *java.io*:

- *ByteArrayInputStream* – Un flujo (stream) de entrada que se gestiona internamente como una matriz de bytes
- *ByteArrayOutputStream* – Un flujo (stream) de salida que se gestiona internamente como una matriz de bytes
- *DataInputStream* - Un flujo (stream) desde el cual se leen datos como tipos primitivos
- *DataOutputStream* – Escribe datos en tipos primitivos en un flujo (stream) en formato binario
- *InputStream* – La clase base para todos los flujos (streams) de entrada

- `InputStreamReader` – Un flujo (stream) desde el que se pueden leer caracteres de texto
- `OutputStream` – La clase base para todos los flujos (streams) de salida
- `OutputStreamWriter` – Un flujo (stream) en el que se pueden escribir caracteres de texto
- `PrintStream` – Un flujo (stream) de escritura que facilita el “envío” de datos en forma de tipos primitivos
- `Reader` – Una clase abstracta para leer flujos (streams) de lectura
- `Writer` – Una clase abstracta para leer flujos (streams) de escritura

Las dos clases más importantes de esta lista son *InputStream* y *OutputStream*, que proporcionan la funcionalidad básica para recibir y enviar datos.

6.2 La clase *InputStream*

La clase *InputStream* es una clase abstracta que sirve como base para otras clases de *streams* de entrada en MIDP. Esta clase define un interfaz básico para leer bytes en modo de *stream*. El uso normal será crear un objeto de tipo *InputStream* desde una conexión y entonces recoger información a través del método *read()*. Si no hay información disponible en este momento, el *stream* de entrada usa una técnica como conocida como bloqueo (*bloquing*) para esperar hasta que haya datos disponibles.

La clase *InputStream* define los siguientes métodos:

- `read()`
- `read(byte b[])`
- `read(byte b[], int off, int len)`
- `skip(long n)`
- `available()`

- `mark(int readlimit)`
- `reset()`
- `markSupported()`
- `close()`

Como podemos ver, tenemos tres métodos para leer datos. El primer método *read* no recibe parámetros y simplemente lee un byte que nos retorna como un entero. Cuando el final del *stream* es alcanzado, el valor retornado será -1. La segunda versión de *read* recibe una matriz de bytes como parámetros y nos permite leer varios bytes de una vez. Los datos leídos son almacenados en la matriz. Esta versión retorna el número de bytes leídos o -1 si se ha llegado al final del stream. La última versión de *read*, tiene tres parámetros. Esta versión es similar a la segunda, pero nos permite especificar la posición de la matriz en la que serán insertados los datos.

El método *skip* se usa para “saltar” un determinado número de bytes en el *stream*. Este método retorna el número de bytes saltados o -1 si hemos alcanzado el final del *stream*.

El método *available* es usado para determinar el número de bytes del *stream* que pueden ser leídos sin bloquear la lectura. Este método devuelve el número de bytes disponibles en el *stream*. Este método es útil cuando queremos asegurarnos de que hay datos disponibles antes del llamar al método *read* y así evitar el bloqueo.

El método *mark* marca la posición actual en el *stream*. Más tarde, podremos volver a esta posición usando el método *reset*. Estos métodos son útiles para aquellas situaciones en las que queremos leer datos del *stream* sin perder la posición original. El método *mark* recibe como parámetro un entero (*readlimit*) que indica cuántos bytes se pueden leer antes de que la marca quede invalidada. El método *markSupported* nos devuelve un *boolean* que indica si el *stream* soporta las marcas.

Finalmente, el método *close* cierra el *stream* y libera los recursos asociados al mismo. No es necesario llamar explícitamente a este método ya que el *stream* será automáticamente cerrado cuando el objeto de tipo *InputStream* sea destruido.

6.3 La clase *OutputStream*

La clase *OutputStream* es el homólogo de salida de la clase *InputStream* y sirve como base para todas las clases de *streams* de salida de MIDP. La clase *OutputStream* define el protocolo básico para escribir datos a través de un *stream* de salida. Normalmente crearemos un objeto de tipo *OutputStream* en una conexión y entonces llamaremos al método *write*. La clase *OutputStream* usa la técnica de bloqueo de forma similar a lo visto para la clase *InputStream*, es decir, se bloquea hasta que los datos son escritos en el *stream*. Durante el bloqueo la clase *OutputStream* no permite que más datos sean escritos.

Los métodos de la clase son:

- *write*(int b)
- *write*(byte b[])
- *write*(byte b[], int off, int len)
- *flush*()
- *close*()

De forma similar a los métodos *read* de la clase *InputStream*, la clase *OutputStream* define tres métodos *write* diferentes. La primera versión de *write* escribe un sólo byte en el *stream*. La segunda versión toma una matriz de bytes y los escribe en el *stream*. La última versión de *write* es similar a la segunda versión, salvo que nos permite especificar la parte del matriz a escribir.

El método *flush* es usado para vaciar el *stream*. Llamando a este método se fuerza al objeto *OutputStream* a enviar los datos pendientes.

El método *close* cierra el *stream* y libera los recursos asociados. Como en los objetos de tipo *InputStream*, no es necesario invocar a este método, ya que al destruir al objeto de tipo *OutputStream* el *stream* es cerrado automáticamente.

6.4 Ejemplo de conexión

Ahora vamos a ver un MIDlet de ejemplo que nos mostrará cómo realizar conexiones. Tenemos un servlet que recibe como parámetro un texto y lo devuelve dado la vuelta:

- Entrada – Salamanca
- Salida – acnamalaS

El MIDlet muestra en primer lugar una pantalla de presentación. A continuación, al pulsar un botón dentro de la pantalla de presentación, pasamos la pantalla principal. La pantalla principal la configuran dos *TextField*. El primero de ellos, será el que debemos editar. Una vez insertado un texto, seleccionando el botón de "ok", hacemos la petición al servlet y la respuesta de este será mostrada en el segundo *TextField*. Si a la hora de hacer el envío el primer *TextField* está vacío, se mostrará un mensaje de error en una alerta (*Alert*).

```
package com.vitike.cursoj2me;

/**
 * <p>Title: Curso de J2ME</p>
 * <p>Description: Clases del Curso de J2ME</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Manuel J. Prieto
 * @version 1.0
```

```
*/
```

```
import javax.microedition.midlet.*;
```

```
import javax.microedition.lcdui.*;
```

```
import javax.microedition.io.*;
```

```
import java.io.*;
```

```
import java.util.*;
```

```
public class ServletConexion extends MIDlet implements  
CommandListener {
```

```
    private Command salir;
```

```
    private Command enviar;
```

```
    private Display display;
```

```
    private Form pantalla;
```

```
    private Form presentacion;
```

```
    private TextField txt1;
```

```
    private TextField txt2;
```

```
    public ServletConexion(){
```

```
        // Recogemos el objeto Display
```

```
        display = Display.getDisplay(this);
```

```
        // Creamos los comandos
```

```
        salir = new Command("Salir", Command.EXIT, 2);
```

```
        enviar = new Command("Ok", Command.OK, 1);
```

```
        // Creamos el "Form" de la pantalla principal
```

```
        pantalla = new Form("J2ME");
```

```
        presentacion = new Form("Presentacion");
```

```
// Configuramos la pantalla de presentacion
String txt = "MIDlet que prueba las conexiones con un servlet.";
presentacion.append(txt);
presentacion.addCommand(enviar);
presentacion.setCommandListener(this);

// Configuramos la pantalla principal
txt1 = new TextField("", "", 20, TextField.ANY);
txt2 = new TextField("", "", 20, TextField.ANY);
pantalla.append(txt1);
pantalla.append(txt2);
pantalla.addCommand(enviar);
pantalla.addCommand(salir);
}

public void startApp() throws MIDletStateChangeException{
    // Establecemos el "Display" actual a la pantalla de presentacion
    display.setCurrent(presentacion);
}

public void pauseApp(){
}

public void destroyApp (boolean incondicional){
}

public void commandAction(Command c, Displayable s){
    if (c == enviar){
        System.out.println(s.toString());
        if (s == presentacion){
            // Estoy en la pantalla de presentacion. Muestro la principal
```

```
        display.setCurrent(pantalla);
        pantalla.setCommandListener(this);
    } else {
        // Estoy en la pantalla principal. Hago la petición al servlet
        hacerPetición();
    }

}

if (c == salir){
    destroyApp(false);
    notifyDestroyed();
}
}

/**
 * Este método hace la petición al servlet
 */
private void hacerPetición(){
    // Si el texto a enviar está vacío, muestro un alert
    if (txt1.getString().length() == 0){
        Alert alert = new Alert("Error",
            "Cadena de texto vacía", null, AlertType.ERROR);
        alert.setTimeout(Alert.FOREVER);
        // Muestro la alerta para que luego vaya a la pantalla principal.
        display.setCurrent(alert, pantalla);
        return;
    }

    // Realizo la conexión al servidor
    StreamConnection conn = null;
    InputStream in = null;
```

```
OutputStream out = null;
StringBuffer data = new StringBuffer();
try{
    // Abrimos la conexión http
    String url = "http://localhost:8080/servlet/" +
        pruebagraph.GiraPalabra?txt=" + txt1.getString();
    conn = (StreamConnection)Connector.open(url);

    // Obtenemos el stream de salida
    out = conn.openOutputStream();

    // Abrimos el stream de entrada
    in = conn.openInputStream();

    // Leemos del stream
    int ch;
    boolean fin = false;
    while ((ch = in.read()) != -1){
        System.out.println("- " + (char)ch);
        data.append((char)ch);
    }
    txt2.setString(data.toString());
}catch (IOException ex){
    System.err.println("Error: No se puede conectar..." );
    ex.printStackTrace();
}

}
```



7 PERSISTENCIA DE DATOS (RMS)

El sistema de gestión de registros (*Record Management System, RMS*) se compone de una serie de clases e interfaces que proporcionan soporte a un sistema simple de base de datos que es usado para almacenar información. Es de esperar que la información que maneje un MIDlet no sea excesivamente complicada.

El objetivo del RMS es almacenar datos de tal forma que estén disponibles una vez que el MIDlet pare su ejecución. La unidad básica de almacenamiento es el registro (*record*) que será almacenado en un base de datos especial, denominada almacén de registros (*record store*). Cuando un MIDlet usa un almacén de registros, primero debe crearlo y luego añadir los registros. Cuando un registro es añadido a un almacén de registros, se le asigna un identificador único (*record id*).

7.1 El paquete RMS

El API de MIDP incluye un paquete para el RMS, *javax.microedition.rms*. Este paquete incluye clases e interfaces que proporcionan un marco de trabajo para los registros, los almacenes y otras características. Básicamente tenemos:

- Capacidad para añadir y borrar registros de un almacén
- Capacidad para compartir almacenes por parte de todos los MIDlets de un MIDlet *suite*

Para representar el *record store* tenemos la clase *RecordStore*. A parte de esta clase, tenemos varias interfaces que presentan la funcionalidad para enumerar registros y filtrarlos, por ejemplo.

7.2 La clase *RecordStore*

Esta clase nos permite abrir, cerrar y borrar almacenes de registros. También podemos añadir, recuperar y borrar registros, así como a enumerar los registros de un almacén. Los métodos son:

- `openRecordStore` – Abre el almacén de registros
- `closeRecordStore` – Cierra el almacén de registros
- `deleteRecordStore` – Borra el almacén de registros
- `getName` – Recupera el nombre del almacén de registros
- `getNumRecords` – Recuperar el número de registros del almacén
- `addRecord` – Añade un registro al almacén de registros
- `getRecord` – Recupera un registro del almacén de registros
- `deleteRecord` – Borra un registro del almacén de registros
- `enumerateRecord` – Obtiene un *enumeration* del almacén de registros

7.3 Las interfaces de RMS

A parte de la clase *RecordStore*, tenemos las siguientes interfaces:

- `RecordEnumeration` – Describe una enumeración del almacén de registros
- `RecordComparator` – Describe la comparación de registros
- `RecordFilters` – Describe como usar filtros sobre los registros
- `RecordListener` – Describe un *listener* que recibe notificaciones cuando un registro es añadido, modificado o borrado del almacén de registros

7.4 Abriendo un almacén de registros

El primer paso para trabajar con RMS es abrir el almacén de registros usando el método estático `openRecordStore` de la clase *RecordStore*. Con este método también podemos crear un almacén nuevo y abrirlo. El siguiente código muestra como crear un almacén y abrirlo:

```
RecordStore rs = RecordStore.openRecordStore("data", true);
```

El primer parámetro indica el nombre del almacén y el segundo indicará si el almacén debe ser creado o si ya existe. El nombre del almacén debe ser menor de 32 caracteres.

7.5 Añadiendo nuevos registros

Para añadir un nuevo registro a un almacén, debemos tener antes la información en el formato correcto, es decir, como una matriz de bytes. El siguiente código muestra como añadir un registro:

```
int id = 0;
try{
    id = recordStore.addRecord(bytes, 0, bytes.length);
}catch (RecordStoreException e){
    e.printStackTrace();
}
```

El primer parámetro es la matriz de bytes, el segundo es el desplazamiento dentro de la matriz y el tercero es el número de bytes a añadir. En el ejemplo anterior, añadimos toda la matriz de datos al almacén de registros. El método *addRecord* devuelve el identificador del registro añadido, que lo identifica unívocamente en el almacén.

7.6 Recuperando registros

Para recuperar un registro de un almacén, debemos utilizar el método *getRecord*, que recupera el registro del almacén a través de su

identificador. La información devuelta por este método es una matriz de bytes. Un ejemplo de uso sería:

```
byte[] recordData = null;
try{
    recordData = recordStore.getRecord(id);
}catch (RecordStoreException ex){
    ex.printStackTrace();
}
```

El código anterior asume que conocemos el identificador del registro.

7.7 Borrando registros

De forma similar a cómo se recuperan registros, se pueden borrar. Para borrar un registro, debemos conocer su identificador. El método para borrar un registro es *deleteRecord*. Un ejemplo de uso sería:

```
try{
    recordStore.deleteRecord(id);
}catch (RecordStoreException ex){
    ex.printStackTrace();
}
```

Cuando se borra un registro, se elimina definitivamente del almacén.

7.8 Enumeración de registros

Para enumerar los registros que hay en un almacén, disponemos del método *enumerateRecords*. Este método nos devuelve un objeto que implementa la interfaz *RecordEnumeration*, que es lo que usaremos para enumerar los registros. El siguiente ejemplo nos muestra cómo podemos obtener una enumeración de registros:

```
RecordEnumeration records =
    recordStore.enumerateRecords(null, null, false);
```

Los parámetros del método son usados para personalizar el proceso. El primer parámetro es un objeto de tipo *RecordFilter* que es usado para filtrar los registros de la enumeración. El segundo parámetro es un objeto *RecordComparator* que determina el orden en que los registros son enumerados. El último parámetro es de tipo *boolean* y determina si la enumeración debe ser actualizada con el almacén. Si este último valor es *true*, la enumeración será automáticamente actualizada para reflejar los cambios en el almacén, así como las inserciones y eliminaciones en el mismo.

La interfaz *RecordEnumeration* define algunos métodos necesarios para el manejo de la enumeración. Por ejemplo, el método *hasNextElement* comprueba si hay más registros disponibles. Para movernos por los registros, podemos usar los métodos *nextRecord* y *nextRecordId*. El primero de ellos retorna una matriz de bytes y el segundo retorna el identificador del registro. El siguiente ejemplo recorre el almacén, mostrando los identificadores de los registros:

```
RecordEnumeration records = null;
try{
    records = recordStore.enumerateRecords(null, null, false);
    while(records.hasNextElement()){
        int id = records.getRecordId();
        System.out.println(id);
    }
}catch (Exception ex){
    ex.printStackTrace();
}
```

7.9 Cerrando un almacén de datos

Es importante cerrar el almacén una vez que hemos acabado de trabajar con él. La clase *RecordStore* proporciona el método *closeRecordStore* con este fin. Este método tiene la siguiente forma:

```
recordStore.closeRecordStore();
```

8 MIDP 2.0

En Noviembre de 2002, Sun Microsystems presentó la última versión 2.0 de MIDP. Esta nueva versión proporciona una serie de nuevas funcionalidades que permiten realizar aplicaciones más ambiciosas. Muchas de las nuevas características están orientadas al desarrollo de aplicaciones para el gran público (juegos, por ejemplo).

Una de las nuevas importantes aportaciones en MIDP 2.0 es la seguridad. En MIDP 2.0 podemos realizar firmado de aplicaciones y manejar dominios privilegiados. Con el firmado de aplicaciones podemos comprobar la identidad del proveedor de la aplicación y por lo tanto confiar en la misma. MIDP 2.0 proporciona soporte para WAPCERT (WAP Certificate Profile), que está basado en certificados de clave pública (X.509) y en listas de revocación de certificados. A través de los privilegios de dominio los proveedores de aplicaciones y los vendedores de dispositivos, pueden definir qué APIs son consideradas como restringidas. Estas nuevas características protegen al dispositivo frente contra accesos no autorizados por parte de las aplicaciones. Los protocolos seguros estándar como HTTPS, TLS/SSL y WTLS permiten la transmisión segura de datos a través de la encriptación de los mismos.

MIDP 2.0 también incluye soporte para interfaces de usuario mejoradas, que permiten a los desarrolladores la creación de aplicaciones más atractivas. La nueva versión de MIDP incluye un nuevo API destinado al desarrollo de juegos y al aprovechamiento de las capacidades multimedia de los dispositivos. La especificación 2.0 introduce el soporte para sonido, compatible con los expuesto en el *Mobile Media API* (JSR-135). Este API soporta generación de tonos, wav files... El nuevo API pensado para el desarrollo de aplicaciones de

entretenimiento, incluye un Canvas pensado especialmente para juegos y clases para el manejo de *sprites*.

La versión 2.0 extiende la conectividad de red, para soportar HTTP (único tipo de conexión en MIDP 1.0), datagramas UDP, sockets TCP y comunicación a través del puerto serie. Una de las nuevas características más potentes es lo que se ha denominado "Push registry" que activa aplicaciones dormidas, cuando está disponible cierta información. Introduciendo una entrada en el descriptor de la aplicación o a través de la clase *PushRegistry*, un midlet puede registrar un *listener*, de tal forma que cuando el midlet no está activo, el software de gestión de aplicaciones de la plataforma MIDP (AMS – *Application Management Software*) escucha peticiones entrantes. Cuando una de estas peticiones es recibida, el AMS lanza el midlet asociado usando el método *startApp*. A través de este sistema, por ejemplo, un proceso servidor, puede activar una aplicación en el dispositivo para notificar un nuevo evento, aunque la aplicación no esté activa en este preciso momento.

La provisión OTA de las aplicaciones, también ha sido mejorada. La provisión OTA permite a los usuarios descargar las aplicaciones al teléfono e instalarlas, sin necesidad de conexiones directas (con cables) a ninguna otra máquina. La versión original de MIDP no incluía provisión OTA y esta fue más tarde recomendada a través de un documento separado de la especificación. Dentro de MIDP 2.0, OTA es expuesto como el medio ideal para la descarga de aplicaciones.

La especificación MIDP 2.0 presenta varias mejoras en lo referente a interfaz de usuario, de tal forma que esta presenta una mayor extensibilidad y portabilidad. Algunas de estas mejoras son:

Una clase *CustomItem* que puede ser heredada para desarrollar elementos de interfaz de usuario no incluidas en el estándar

Una clase *ItemCommandListener* para mejorar las interacciones con los items y una clase *Spacer* para realizar *layouts* más precisos

Los elementos (*items*) de los *Form* tienen ahora un tamaño mínimo y un tamaño recomendado para adaptarse mejor a las capacidades de los distintos dispositivos.

8.1 El API de juegos de MIDP 2.0

Como se ha mencionado anteriormente, una de las mejoras más importantes que presenta MIDP 2.0 es su API destinado al desarrollo de aplicaciones de entretenimiento. A través de este API, podemos desarrollar interfaces de usuario más rápidas con mejor usabilidad y con un uso de los recursos más eficiente. El API de juegos también ayuda a reducir el tamaño del jar. Con MIDP 1.0, los desarrolladores debían proveer sus propias rutinas de gestión de gráficos para realizar los procesos y esto provoca el aumento del tamaño de los jar, al tener que añadir estas clases en la distribución.

La idea básica del API de juegos, es que la pantalla del juego se compone de capas. Las pantallas a mostrar se componen a través de capas. Todas estas capas pueden ser manejadas por separado y el API se encarga de dibujar las distintas capas. El área de trabajo puede ser mayor que el tamaño de la pantalla del dispositivo y el movimiento o *scroll* de esta área puede resultar costoso. El API de juegos proporciona una vista de una parte del área de trabajo que compone una pantalla del juego (no una pantalla en el dispositivo). El API de juegos se encuentra en el paquete *javax.microedition.lcdui.game*. Hay cinco clases nuevas en el API:

GameCanvas

Layer

LayerManager

Sprite

TiledLayer

La clase *GameCanvas* es abstracta y proporciona la funcionalidad básica para la realización de la interfaz de usuario. Esta clase presenta dos beneficios sobre la versión básica de *Canvas*. Por una parte tiene un *buffer off-screen* y permite recoger el estado de las teclas del dispositivo.

La clase *Layer* es una clase abstracta que representa un elemento del juego. Las clases *Sprite* y *TiledLayer* heredan de esta clase. El gestor de capas (*LayerManager*) maneja los objetos *Layer* y los dibuja en la pantalla en el orden específico. La clase *Sprite* es una capa (*Layer*) que contiene varios *fotogramas* almacenados en una imagen (*Image*). Con la clase *Sprite* se pueden usar partes de la imagen como fotogramas y a través de una secuencia de fotogramas, crear movimiento. La clase *Sprite* también contiene funcionalidad para comprobar colisiones con otros *Sprites* o con *TiledLayers*.

La clase *TiledLayer* es similar a *Sprite* pero está pensada para ser usada en la construcción de fondos. *TiledLayer* consiste en una serie de celdas, que pueden ser rellenas con imágenes o *tiles*. Así, el fondo se compondrá a través de pequeñas imágenes.

En MIDP 2.0, el manejo de las ordenes del usuario se realiza de forma distinta a como se hacía con MIDP 1.0. En la versión 1.0 se usaba el método *GameAction* de la clase *Canvas* para recoger la acción. En la nueva versión, se puede recuperar el estado de las teclas a través del método *getKeyStates* de la clase *GameCanvas*. El

siguiente código muestra un ejemplo de cómo se puede controlar el estado de las teclas:

```
protected void keypressed (int keyCode){
    int move = 0;
    int keyState = getKeyStates();
    if (keyState & LEFT_PRESSED) != 0){
    }
    if (keyState & RIGHT_PRESSED) != 0){
    }
    if (keyState & UP_PRESSED) != 0){
    }
    if (keyState & DOWN_PRESSED) != 0){
    }
}
```

Los *buffer off-screen* nos permiten crear animaciones (o movimiento) sin parpadeos. La idea del *buffer off-screen* en la clase *GameCanvas* es permitir un objeto *Graphics off-screen* que pueda ser usado para dibujar y cuando tengamos el dibujo completado, mostrarlo. El siguiente código tenemos un ejemplo del uso del *buffer off-screen*:

```
public void run(){
    Graphics g = getGraphics();

    while(play){
        try{
            // Primero dibujamos todo
            layers.paint(g,0,0);

            // Si el juego está activo, mostramos los gráficos
            if (play){
```

```

        flushGraphics();
    }

    try{
        mythread.sleep(sleepTime);
    }catch (java.lang.InterruptedException ex){}
    } catch (Exception ex) {}
}
}

```

Dentro de un juego (o de cualquier aplicación gráfica) la pantalla suele consistir en una serie de elementos que pueden estar interrelacionados o no. Las capas (*Layers*) de MIDP 2.0 proporcionan una forma de manejar objetos en la pantalla. Como hemos mencionado anteriormente, los elementos *Sprite* y *TiledLayer* son capas. La clase *Sprite* contiene varias imágenes del mismo tamaño. La clase *TiledLayer* usa estas imágenes colocándolas dentro de una *rejilla*. Cuando una instancia de *TiledLayer* es creada, tenemos que indicar cinco parámetros al constructor. Estos parámetros son el número de filas y columnas, la imagen y el ancho y alto de la celda. Una vez creada la instancia, las celdas pueden ser rellenas (*fillCells* o *fillCell*). El siguiente código muestra un ejemplo de cómo usar estos elementos:

```

private TiledLayer tiles;
private LayerManager layers;
private Image tilesImage;

public final int TILE_GROUND = 1;
public final int TILE_WATER = 2;
public final int TILE_SHORE_LEFT = 3;

```

```

public final int TILE_FOREST = 4;
public final int TILE_AIR = 5;
public final int TILE_SHORE_RIGHT = 6;

layers = new LayerManager();
try{
    tilesImage = Image.createImage("/res/Tiles.png");
}catch (IOException ex){}
tiles = new TiledLayer(40, 16, tilesImage, 7, 7);

tiles.fillCells(0, 0, 40, 16, TILE_AIR);
tiles.fillCells(14, 12, 12, 4, TILE_WATER);
tiles.fillCells(0, 10, 14, 6, TILE_GROUND);

layers.append(tiles);

```

Como se ha mencionado anteriormente, los *sprites* representan objetos en la pantalla. La clase *Sprite* contiene varias imágenes del mismo tamaño. Estas imágenes son los distintos fotogramas que nos permiten configurar una animación.

```

try{
    spriteImage = Image.createImage("/res/sprite.png");
} catch (Exception ex){}
sprite = new Sprite(spriteImage, 7, 7);

```

Otro elemento importante dentro de la programación de juegos es la detección de colisiones. La clase *Sprite* contiene cuatro métodos relacionados con la detección de colisiones:

- *collidesWith(Image image, int x, int y, boolean pixelLevel)*
- *collidesWith(Sprite s, boolean pixelLevel)*
- *collidesWith(TiledLayer t, boolean pixelLevel)*

- *defineCollisionRectangle(int x, int y, int width, int height)*

Como podemos ver, se pueden detectar colisiones entre dos *sprites*, entre un *sprite* y un *TiledLayer* y entre un *sprite* y una imagen

8.2 El Push Registry de MIDP 2.0

El concepto *Push* (sea cual sea el contexto) indica la capacidad o el mecanismo para recibir información de forma asíncrona, es decir, cuando la información está disponible. Es decir, nuestra aplicación estará en modo *pasivo* y en un determinado momento será avisada de que dispone de nueva información y por lo tanto actuará en consecuencia.

Dentro de MIDP, la tecnología *Push Registry* permite que los MIDlets sean lanzados automáticamente, sin interacción directa del usuario. Tendremos dos posibles formas de lanzar las aplicaciones J2ME, tendremos activación por red (mediante una conexión externa) o por tiempo (pasado un determinado tiempo). La tecnología *Push Registry* forma parte del sistema de gestión de aplicaciones (*Application Management System – AMS*), que es el software que se encarga en el dispositivo del ciclo de vidas de las aplicaciones (instalación, activación, ejecución y eliminación).

Push Registry forma parte del Marco Genérico de Conexión (*Generic Connection Framework – GCF*) y es encapsulado dentro de una única clase, la clase *javax.microedition.io.PushRegistry*. Los métodos de esta clase son:

- *getFilter()* – Retorna el filtro de emisores permitidos para una determinada conexión *push*
- *getMidlet()* – Retorna el Midlet responsable de gestionar la conexión *push* especificada

- *listConnections()* – Retorna la lista de conexiones registradas para un midlet o *midlet suite*
- *registerAlarm()* – Registra una alarma basada en tiempo para lanzar el midlet. Si se le pasa 0 como argumento, deshabilitará las alarmas que haya configuradas
- *registerConnection()* – Registra una conexión *push*
- *unregisterConnection()* – Elimina el registro correspondiente a una conexión *push*

El API de *Push Registry* también define una serie de excepciones que nos servirán para controlar los posibles errores o anomalías que se presenten. Estas excepciones son:

- *ClassNotFoundException*
- *ConnectionNotFoundException*
- *IllegalArgumentException*
- *IOException*
- *SecurityException*

La tecnología *Push Registry* no cambia el ciclo de vida de los midlets, simplemente introduce nuevas situaciones en el proceso de activación de estos:

- Cuando se produce una nueva conexión entrante
- Cuando se dispara una alarma temporal

MIDP 2.0 define nuevos tipos de conexión con respecto a la versión anterior de MIDP, donde sólo se aseguraba la disponibilidad de conexiones HTTP. Entre estos nuevos tipos de conexiones tenemos *sockets* TCP y datagramas UDP, que nos permitirán configurar conexiones entrantes y por lo tanto poderlas usar como disparadores de *Push Registry*. Además el API de mensajería para J2ME (*Wireless Messaging API – WMA*) permite la activación de *Push* a través de mensajes SMS. La especificación de MIDP 2.0 no indica qué tipo de

conexiones son validadas como disparadores de *Push*, esta labor recae sobre los vendedores de terminales, aunque típicamente tendremos las antes mencionadas.

Para que un midlet sea capaz de recibir activaciones mediante *Push*, debe registrarse anteriormente de uno de los siguientes modos:

Registro estático – Este tipo de registros se realizan durante la instalación del midlet. A través de los atributos *MIDlet-Push* del fichero jad, podremos configurar los registros.

Registro dinámico – Se pueden configurar registros de forma dinámica y alarmas temporales en tiempo de ejecución, usando el API de *Push Registry*.

Como acabamos de decir, podemos configurar registros *Push* de forma estática a través del fichero jad. Cuando se instala el midlet, el AMS realiza el registro y cuando el midlet se desinstala, automáticamente el registro es también borrado. El formato del fichero jad es el siguiente:

MIDlet-Push-[n]: [url], [clase], [emisor permitido]

Dentro de esta línea, los datos son:

- MIDlet-Push-[n] – Identifica el registro *Push* y n es un número secuencial que comienza en 1.
- url – Es una cadena que identifica el punto de conexión entrante. Este valor será el mismo que tendrá el midlet dentro de *Conector.open()* para recoger la conexión. Por ejemplo, podremos tener *socket://5000*
- clase – Es el nombre completo de la clase que será activada cuando se detecte la conexión entrante.
- emisor permitido – Es un filtro usado para restringir los servidores que pueden activar el midlet. Se pueden usar

comodines, como * (1 o más caracteres) y ¿ (un solo carácter).

El registro dinámico nos permite configurar tanto activaciones por conexiones entrantes como por el paso de un determinado tiempo.

Si necesitamos que nuestra aplicación realice una determinada operación cada cierto tiempo, podemos configurar una alarma temporal, de tal forma que nuestro midlet se active cada cierto tiempo. En la versión anterior de MIDP, podíamos tener también esta funcionalidad, configurando un *TimerTask* que se ejecute cada cierto tiempo, pero esta solución nos obliga a tener el midlet en ejecución constantemente. Con *Push Registry* esto no es necesario y podemos tener el midlet parado y este será arrancada cuando llegue el momento de realizar la operación correspondiente. Para configurar los *Push Registry* temporales, tendremos el método *PushRegistry.registerAlarm()*, que recibirá como argumentos la clase a ejecutar y el tiempo que configurar las alarmas. Si este último valor es 0, la alarma será deshabilitada. El siguiente código muestra como utilizar este tipo de alarmas:

```
private void programaMIDlet (long milisegundos)
    throws ClassNotFoundException,
        ConnectionNotFoundException,
        SecurityException {

    String clase = this.getClass().getName();
    Date alarma = new Date();
    long t = PushRegistry.registerAlarm(class,
        alarma.getTime() + milisegundos);
}
```

Si el método *registerAlarm* sobrescribe una alarma anterior, retorna el momento para el que se ha configurado la alarma y de lo contrario, devuelve 0.

Como ya hemos dicho, *Push Registry* nos permite configurar la activación de midlets cuando se produzca una determinada conexión entrante. Un midlet debe utilizar el método *registerConnection* para registrar la activación. El siguiente código muestra un ejemplo:

```
....  
String clase = this.getClass().getName();  
String url = "socket://:6909";  
String filter = "*";  
  
try{  
    ServerSocketConnection ssc =  
(ServerSocketConnection) Connector.open(url);  
PushRegistry.registerConnection(url, clase, filter);  
SocketConnection sc =  
    (SocketConnection)ssc.acceptAndOpen();  
InputStream is = sc.openInputStream();  
}catch (SecurityException ex){  
}catch (ClassNotFoundException ex){  
}catch (IOException ex){  
}  
....
```

A través del método *PushRegistry.listConnections()* podemos averiguar las conexiones entrantes registradas por el midlet.

9 MEJORANDO LA EXPERIENCIA DEL USUARIO EN CONEXIONES J2ME

Uno de los objetivos de cualquier aplicación debe ser el proporcionar una interfaz de usuario lo mejor posible, para que la experiencia de este sea satisfactoria usando dicha aplicación. En el caso de aplicaciones inalámbricas se presentan dos circunstancias que hacen mucho más importante el proceso de diseño de la interfaz de usuario. Por un lado, los entornos móviles de ejecución tienen unas importantes limitaciones en cuanto a la posibilidades de construcción de dichas interfaces, debido básicamente al reducido tamaño de las pantallas. Por otro lado, la utilización de los terminales móviles no suele ser sencilla y los elementos destinados a la indicación de comandos por parte del usuario no son demasiado sofisticados.

Dentro de versión MIDP de J2ME las conexiones de red son habituales, debido a la familia de dispositivos para la que está pensada dicha versión. Sin embargo, las conexiones existentes hoy en día suelen ser lentas, lo que nos obliga a programar cuidadosamente las conexiones de red para que la experiencia del usuario sea lo más satisfactoria posible y que esté informado en todo momento de qué está realizando la aplicación y por lo tanto sepa como actuar en cada instante.

La versión más sencilla de un método de conexión a la red puede ser:

```
package com.vitike.j2me;
/**
 * <p>Title: Pruebas J2ME </p>
 * <p>Copyright: Copyright (c) 2003</p>
 * @author Manuel J. Prieto
 * @version 1.0
 */
```

```

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

public class MIDletV1 extends MIDlet implements
CommandListener{
    private Display display;
    private Form pantalla;
    private Command salir;
    private Command conectar;
    private StringItem msg;
    public void startApp(){
        display = Display.getDisplay(this);
        if (pantalla == null){
            pantalla = new Form("Version 1.0");
            msg = new StringItem("", "Test de conexi\u{f3}n");
            pantalla.append(msg);
            salir = new Command("Salir", Command.EXIT, 0);
            conectar = new Command("Conectar", Command.SCREEN, 0);
            pantalla.addCommand(salir);
            pantalla.addCommand(conectar);
            pantalla.setCommandListener(this);
        }
        display.setCurrent(pantalla);
    }
    public void commandAction(Command c, Displayable d){
        if (c == salir){
            notifyDestroyed();
        }
        if (c == conectar){
            msg.setText(conexion());
        }
    }
}

```

```
}  
private String conexion(){  
    String url = "";  
    url = getAppProperty("url");  
    try{  
        HttpURLConnection conn = (HttpURLConnection)Connector.open(url);  
        InputStream in = conn.openInputStream();  
        int contentLength = (int)conn.getLength();  
        if (contentLength == -1){  
            contentLength = 255;  
        }  
        byte[] datos = new byte[contentLength];  
        int leidos = in.read(datos);  
        String dat = new String(datos, 0, leidos);  
        in.close();  
        conn.close();  
        return dat;  
    }catch (IOException ioe){  
        return ioe.toString();  
    }  
}  
  
public void pauseApp(){}  
public void destroyApp(boolean incondicional){}  
}
```

Esta versión es muy sencilla, pero tiene varios problemas:

El usuario puede seguir interactuando con la aplicación, a pesar de que la aplicación está realizando la conexión

El usuario no tiene constancia de la evolución del proceso de conexión

Si probamos la aplicación anterior en el emulador, aparentemente no tendrá problemas debido a que la velocidad de proceso y conexión

son altas. Sin embargo, al pasar la aplicación al teléfono, la experiencia del usuario será totalmente distinta ya que tanto la capacidad de proceso del terminal como la calidad de las conexiones es mucho menor. En el ejemplo anterior, un *Thread* del sistema está siendo usado para realizar la conexión y por lo tanto, si esta lleva un periodo de tiempo considerable, el *Thread* estará ocupado durante un periodo de tiempo y la aplicación permanecerá bloqueada.

La aplicación de conexión que acabamos de ver, puede ser mejorada, incluyendo dentro de la misma un *Thread* propio de la aplicación que se encargue del proceso de conexión. Una posible solución podría ser la siguiente:

```
public void commandAction(Command c, Displayable d){
    if (c == salir){
        notifyDestroyed();
    }
    if (c == conectar){
        Thread t = new Thread(){
            public void run(){
                msg.setText(conexion());
            }
        };
        t.start();
    }
}
```

Ahora tenemos un *Thread* que no bloquea el sistema para realizar la conexión, ya que crea un *Thread* propio que se encargará de dicho proceso. Si implementamos esta solución, se nos presenta un problema en la interacción con el usuario. Debido a que el sistema sigue completamente operativo, la interfaz de usuario permanece

normal y por lo tanto este no está informado de la evolución de la conexión y por lo tanto puede creer que el sistema no está realizando la conexión y seleccionar el comando "conectar" de nuevo, lo que provocará que la creación de un nuevo *Thread*. Este hecho se puede repetir varias veces, lo que agrava el problema.

La solución a todos estos problemas pasa por informar al usuario en cada momento de lo que está haciendo la aplicación. Si mostramos una pantalla de espera al usuario, este sabrá que la aplicación está realizando la conexión y además impedimos que intente reconectar. Una forma sencilla de hacer esto, sería mostrar al usuario una pantalla sencilla en la que se indicara que el sistema está conectando. Al iniciar el proceso de conexión se activa como pantalla actual la pantalla informativa de conexión y al finalizar la conexión, se vuelve a la pantalla principal. Esta solución parece buena, pero aún hay ciertas cosas que se pueden mejorar.

Debemos dar al usuario la posibilidad de cancelar el proceso de conexión, ya debido a la naturaleza de las conexiones que se realizan desde teléfonos móviles, es posible que la conexión tarde en realizarse. Para realizar esto, pondremos un botón (*Command*) de cancelación en la pantalla de espera.

Otra posible mejora para la aplicación, es la reutilización del *Thread* de conexión y así evitar la creación de un nuevo *Thread* para cada conexión. Para hacer esto, crearemos un método *run()* en el *Thread* que se un bucle "infinito". Así, pararemos y relanzaremos el *Thread* según nuestras necesidades y tendremos una mejor gestión de los recursos y por lo tanto una aplicación mucho más eficiente. Un ejemplo de esto sería:

```
public synchronized void run(){
```

```
while (corriendo){  
    try{  
        wait();  
    }catch (InterruptedException ie){}  
    if (corriendo){  
        conexión();  
    }  
}  
}  
public synchronized void go(){  
    notify();  
}
```

En el código anterior, la variable *corriendo* indicará si la instancia del *Thread* está en ejecución o no. Así, el *Thread* será rearrancado cuando el usuario seleccione el comando conectar y el *Thread* será parado cuando la conexión finalice o cuando el usuario cancele la conexión.

Por último, podemos realizar una mejora puramente estética pero que hará de nuestra aplicación una solución más profesional y más agradable para el usuario. Lo único que debemos hacer es realizar una pantalla de espera para la conexión más bonita. Por ejemplo, podemos usar un *Canvas* para realizar dicha pantalla de espera y hacer que esta muestre una pequeña animación. Desde el punto de vista funcional, nuestra aplicación es la misma, pero desde el punto de vista de la interfaz de usuario, nuestra aplicación es mucho mejor.

10 ALGUNOS PATRONES DE DISEÑO PARA J2ME

En el desarrollo de aplicaciones para J2ME, hay que tener muy en cuenta las restricciones del terminal y del propio lenguaje. Además de esto, si queremos que nuestras aplicaciones sean realmente profesionales y expresen al máximo las capacidades del dispositivo en concreto en el que se ejecuten, tendremos que tener en cuenta las características de cada uno de estos dispositivos. Esto nos obliga en cierta medida a hacer una versión de la aplicación para cada terminal o grupo de terminales con características similares (tamaño pantalla, número de colores...). También es cierto que las modificaciones entre las distintas versiones son mínimas. Por lo tanto, tendremos que tener mucho cuidado a la hora de diseñar nuestras aplicaciones, para que el esfuerzo de adaptación a los distintos terminales sea lo más pequeño posible. A continuación, veremos una serie de patrones de diseño o modos de actuación, que nos facilitarán el desarrollo de aplicaciones J2ME de calidad y que serán más sencillas de adaptar a cada terminal, por lo que el resultado final será mucho más profesional.

10.1 Patrón para la generación de menús en cascada

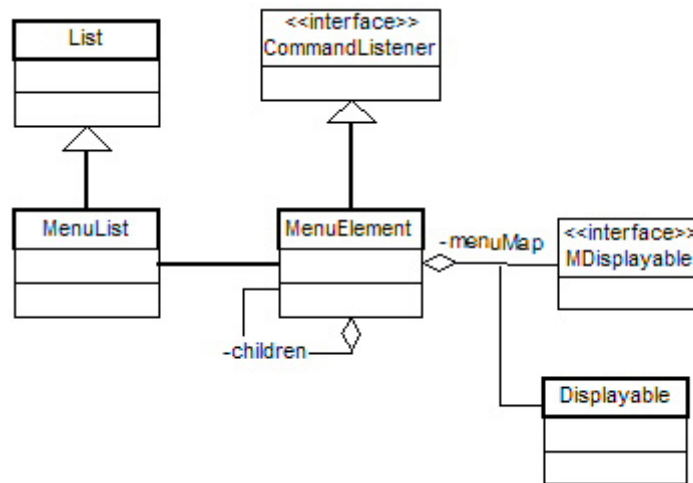
Este patrón nos muestra como realizar menús en cascada dentro de las aplicaciones. Este patrón es útil en la realización de los procesos de navegación por las aplicaciones. Los menús en cascada organizan las distintas funcionalidades del sistema dentro una arquitectura de varios niveles. Dentro de esta organización, cada elemento de un menú puede ser el enlace a otro menú o el acceso a una funcionalidad del sistema. La siguiente imagen muestra un ejemplo de esto:



Este tipo de interfaz de usuario es común dentro de los entornos WAP. Si usamos elementos de interfaz de usuario de alto nivel, podremos implementar fácilmente estos menús dentro de nuestra aplicación a través de elementos *List*. Cada objeto de tipo *List* contendrá las distintas opciones disponibles en un menú o pantalla y cuando un elemento de la lista sea seleccionado el sistema presentará la lista de sub-opciones correspondiente o ejecutará la acción correspondiente.

Cuando el número de listas o pantallas es muy grande, la gestión de los comandos empieza a ser un poco complicada si cada objeto de tipo *List* gestiona sus propios comandos. Para solucionar esto podemos optar por un patrón MVC, lo que nos permitirá crear separar las responsabilidades de mejor forma y hacer la aplicación más sencilla y potente. El patrón MVC separa el Modelo, de la Vista y el Controlador. El modelo es la estructura de datos que representa el contenido de la aplicación. La vista es el componente visual que muestra los datos del modelo al usuario y por último, el controlador es el puente entre el modelo y la vista y gestiona la lógica o el flujo

de información entre los otros dos componentes. El diagrama de clases de la solución MVC al menú en cascada sería:



La clase *MenuElement* es la implementación del modelo de datos. Cada instancia de esta clase representa un nodo dentro del árbol de menús. El siguiente código muestra como construir parte de la estructura de menús vista anteriormente según este patrón:

```

MenuElement menu3 = new MenuElement("Atracciones");
menu3.addChild(new MenuElement("Museos"),new MenuAction());
menu3.addChild(new MenuElement("Teatros"),new MenuAction());
menu3.addChild(new MenuElement("Cines"),new MenuAction());

MenuElement menuPrincipal = new MenuElement("MenuPrincipal");
menuPrincipal.addChild(new MenuElement("Atracciones"), menu3);

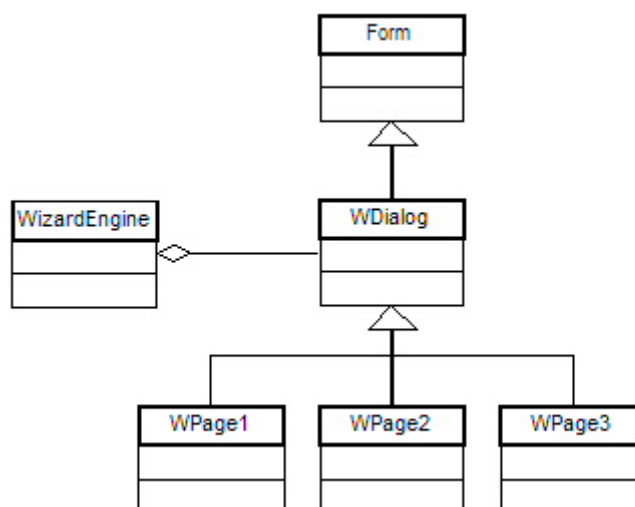
menuList.showMenu(menuPrincipal);

```

10.2 Patrón para la generación de diálogos

Muchas aplicaciones usan agentes de ayuda (*wizards*) para guiar al usuario a través de un determinado proceso, como puede ser un proceso de instalación. Típicamente, en este tipo de interacción con el usuario, se le presentan una serie de pantallas en las que este debe proporcionar cierta información y se le permite pasar a la siguiente pantalla y volver a la anterior. Este modo de interacción con el usuario, puede ser una buena forma de evitar los formularios demasiados largos (como los de muchas páginas web), ya que las limitaciones de pantalla de los terminales hacen del uso de los formularios extensos una mala práctica desde el punto de vista de la usabilidad.

La solución propuesta en el párrafo anterior es buena, por lo tanto, merece la pena realizar un buen diseño software para permitir al desarrollador el crear este tipo de secuencias de pantallas de forma sencilla y flexible. Si implementamos el patrón conocido como Mediador (*Mediator*), tendremos una forma elegante y limpia de desarrollar este tipo de soluciones. El siguiente diagrama UML muestra las clases necesarias en la realización de una secuencia de diálogos:



La clase *WDialog* tendrá una serie de métodos para poder configurar el *workflow* de pantalla. Las clases que heredan de estas, implementan un método abstracto de la clase *WDialog*, el método *initDialog*. Este método permite configurar el contenido de la pantalla. Un posible ejemplo de este método sería:

```
public class WPage2 extends WDialog{
    ChoiceGroup cuestion1;
    public void initDialog(){
        setTitle("Paso 2");
        cuestion1 = new ChoiceGroup....
        append(cuestion1);
    }
}
```

Podremos tener métodos que se ejecuten al entrar y salir de cada una de las pantallas, de tal forma que controlemos las acciones del usuario y podamos informarle de posibles errores al salir y configurar cada pantalla al entrar en la misma.

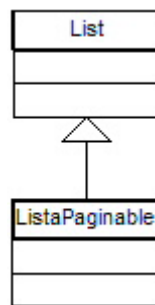
10.3 Patrón de la paginación

Este patrón nos vuelve a ayudar en la creación de interfaces de usuario mucho más eficientes en cuanto a la usabilidad. Las limitaciones de pantalla de las que ya hemos hablado anteriormente hacen que el proceso de paginación de las interfaces (como las listas) sea muy tedioso para el usuario.

El uso de la paginación en las aplicaciones en las que se debe presentar una lista de elementos larga, es una buena política de

usabilidad ya que requieren menos *clicks* por parte del usuario que la presentación de todos los elementos en una única pantalla. La paginación también es más eficiente desde el punto de vista del uso de los recursos, como la memoria.

El patrón de paginación nos permite automatizar en cierta medida el proceso de paginado. En este caso también tenemos una separación clara entre el modelo de datos y la vista. Los datos son la lista de elementos y esta estructura no tiene conocimiento sobre como va a ser mostrada en pantalla. El diagrama de clases para este patrón sería:



La clase *ListaPaginable* contiene internamente todos los elementos de la lista y un enlace al índice de la lista que se está mostrando en cada momento. Tendrá también dos posibles comandos, siguiente y anterior, que nos permitirán avanzar y retroceder en la lista. Debido a que la clase *ListaPaginable* hereda de *List*, tendremos los métodos de esta última para añadir la información. La clase *ListaPaginable* implementa un patrón *Proxy*.

10.4 Patrón para la creación de aplicaciones portables

Hoy por hoy, las diferencias entre los distintos dispositivos que disponen de J2ME (incluso dentro de un mismo perfil) son muy

grandes, en cuanto a la interfaz de usuario. Es decir, tendremos terminales con pantalla pequeñas con dos colores y tendremos terminales con pantallas grandes y gran cantidad de colores. De forma similar, en algunos terminales tendremos un pequeño *joystick* que el usuario utilizará para interaccionar con el sistema y en otros casos tendremos simplemente dos teclas. Por otra parte, si queremos que nuestras aplicaciones sean totalmente profesionales y aprovechen al máximo las normas de cada terminal, tendremos que personalizar las interfaces de usuario para cada terminal.

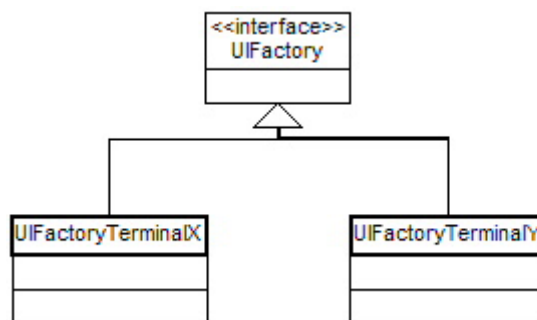
Una de las virtudes de J2ME es que nos permite desarrollar aplicaciones que cubran un amplio espectro de terminales. Para hacer esto de forma automática y que las aplicaciones corran sin problemas en todos los terminales, tendremos que utilizar los componentes de interfaz de usuario a alto nivel que nos proporciona de forma nativa J2ME. Esto parece una buena solución, pero a cambio el resultado de las mismas no suele ser muy vistoso y por supuesto, no aprovechamos al máximo las capacidades de los terminales ligeramente avanzados. Otra cuestión que nos obliga a realizar interfaces de usuario adaptadas al terminal, es la necesidad de hacer los menús y demás elementos de nuestra aplicación lo más parecidos posible a los menús o demás elementos nativos del terminal. De esta forma, la usabilidad de la aplicación será mucho mayor debido a que el usuario interactúa con un entorno que ya conoce. Además, de esta forma, el usuario nota menos la diferencia entre aplicaciones nativas del terminal y aplicaciones J2ME descargadas, lo que reduce en gran medida la posible reticencia del usuario al uso de las aplicaciones externas al terminal.

Bien, para conseguir soluciones con este tipo de características, tendremos que diseñar nuestra aplicación con sumo cuidado, de tal forma que el esfuerzo necesario para portar la aplicación entre

terminales sea mínima. Es decir, por una parte tendremos que conseguir que los distintos terminales compartan la mayor parte de código y que la aplicación este bien diseñada, para que el reparto de responsabilidades sea correcto y por lo tanto la adaptación al terminal requiera la menor cantidad de código posible.

Una buena forma de empezar a diseñar aplicaciones con las características anteriormente descritas, es la separación entre el modelos de datos o la lógica de aplicación y la interfaz de usuario, de forma similar a lo expuesto en alguno de los patrones anteriores. Si optamos por esta separación de responsabilidades, podremos compartir totalmente el modelo de datos entre todas las versiones de la aplicación y sólo tendremos que adaptar la interfaz de usuario. Una vez conseguido esto, lo único que tendremos que hacer será mejorar el diseño para que la generación de interfaces de usuario sea lo más sencilla y directa posible.

Un patrón de diseño de software conocido para soluciones de este tipo, son los denominados *Factory*. Es decir, tendremos lo que podemos denominar una fábrica o creador (*factory*) de interfaces de usuario, de tal forma, que el creador genere una interfaz de usuario totalmente adaptada al terminal o familia de terminales. El diseño de una solución de este tipo, sería algo así:



En esta arquitectura de clases, tendremos que la interfaz define todos los métodos necesarios para crear las distintas pantallas que se mostrarán al usuario en la aplicación. Las clases que implementan esta interfaz, serán las encargadas de construir la pantalla adaptada al terminal y así conseguiremos nuestro objetivo.

El siguiente código ilustra la forma de hacer esto:

```
public interface UIFactory{
    public Displayable getPresentacion(int terminal);
    public Displayable getMenuPrincipal(int terminal);
    ....
}
public class UIFactoryTerminalX implements UIFactory{
    public Displayable getPresentacion(int terminal){
        // Creamos un Canvas personalizado
        ....
    }
}
```

Esta solución es buena, pero para completarla, tendremos que incluir en la fichero jar sólo aquellas clases que van a ser utilizadas en el terminal o familia de terminales en concreto. Así, tendremos una distribución para cada una de las familias de terminales y esta distribución sólo tendrá las clases que se utilizan en tiempo de ejecución en dicha aplicación, mientras que lo que podríamos considerar como el proyecto global de desarrollo, tiene un número de clases superior.

11 OPTIMIZACIÓN DE CÓDIGO

La mayoría de los teléfonos móviles tienen importantes restricciones en cuanto a memoria y capacidad de proceso. Esto nos obliga a optimizar en la medida de lo posible los MIDlets. Hay tres importantes facetas en las que optimizar un MIDlet, entre otras:

- Mantenibilidad
- Tamaño
- Velocidad

Lo mejor que se puede hacer de cara a la optimización, es hacer los MIDlets lo más simple posible. Es decir, debemos hacer los interfaces de usuario sencillos, para reducir el número de componentes y comandos.

11.1 Optimización para la mantenibilidad

Esta optimización hará que el código sea manejable en el futuro y más sencillo de comprender. Depende básicamente de la estructura y la organización del código. Esta optimización, está reñida en cierta forma con los otros dos tipos de optimización. A pesar de que la optimización orientada a la mantenibilidad es importante, la optimización del tamaño y la velocidad es aún más importante.

11.2 Optimización del tamaño

Para optimización el tamaño del MIDlet lo que debemos hacer es que el tamaño de las clases resultantes sea lo más pequeño posible. La piedra angular de esta optimización es la reutilización del código,

cuya base es la herencia. Otro punto a considerar son los recursos usados por el MIDlets y los datos que gestiona.

Para reducir el uso de memoria, podemos usar alguna de las siguientes técnicas:

Evitar el uso de objetos siempre que sea posible

Cuando usamos objetos, debemos reciclarlos siempre que sea posible

Limpiar los objetos explícitamente cuando finalicemos de usarlos. El *garbage collector* de J2ME no es del todo eficiente, ya que su prioridad es muy baja

Usar un ofuscador. Los ofuscadores reducen de forma considerable el tamaño del jar a distribuir

Tener los recursos optimizados para el terminal. No debemos olvidar que los recursos son incluidos dentro del jar de distribución

11.3 Optimización de velocidad

La optimización de la velocidad de ejecución del MIDlet es la más importante y es lo que debemos tener presente en primer lugar.

Eliminación de Evaluaciones innecesarias

El siguiente código:

```
for (int i=0; i<size(); i++)  
    a = (b + c) / i ;
```

Puede ser optimizado de la siguiente forma:

```
int tmp = b + c;  
int s = size();  
for (int i=0; i<s; i++)  
    a = tmp / i;
```

11.4 Eliminar subexpresiones comunes

El siguiente código:

```
b = Math.abs(a) * c;  
d = e / (Math.abs(a) + b);
```

Puede ser optimizado de la siguiente forma:

```
int tmp = Math.abs(a);  
b = tmp * c;  
d = e / (tmp + b);
```

11.5 Aprovechar las variables locales

El siguiente código:

```
for (int i=0; i<1000; i++)  
    a = obj.b * i;
```

Puede ser optimizado de la siguiente forma:

```
int localb = obj.b;  
for (int i=0; i<1000; i++)  
    a = localb * i;
```

11.6 Expandir los bucles

El siguiente código:

```
for (int i=0; i<1000; i++)  
    a[i] = 25;
```

Puede ser optimizado de la siguiente forma:

```
for (int i=0; i<100; i++){  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
    a[i++] = 25;  
}
```

11.7 Mejora de las operaciones gráficas

Cuando usamos gráficas de bajo nivel, debemos tener mucho cuidado con la forma en que se realizan estas operaciones, ya que un funcionamiento lento, echará a perder cualquier aplicación.

Una buena política es repintar sólo aquella parte de la pantalla que ha de cambiar, en lugar de repintar toda la pantalla. El siguiente método nos permite hacer esto de forma sencilla:

```
Canvas.repaint(int x, int y, int width, int height);
```

Otro buen sistema de optimización de procesos gráficos es la utilización de imágenes *off-screen*. Este sistema es especialmente adecuado cuando nuestras pantallas cambian poco entre repintados. En este caso podemos dibujar sobre una imagen en memoria y luego mostrar en el método *paint* esta imagen.

Por último, sólo decir que es muy útil la utilización del método *serviceRepaints*. Este método fuerza el repintado de la pantalla de manera inmediata. Cuando tenemos una interacción constante con el usuario, este método es esencial para que el usuario perciba que el sistema reacciona a sus comandos. Sin este método, el repintado podría tardar el tiempo suficiente como para que se sobrepongan varios comandos del usuario y este no tenga una buena experiencia en la utilización de la aplicación.

11.8 Recolector de basura

Debemos evitar la creación de objetos innecesarios, ya que el proceso que se encarga de recoger los objetos no útiles y destruirlos, tiene una prioridad muy baja. Debemos reutilizar los objetos siempre que sea posible.

Debemos evitar en la medida de lo posible los objetos inmutables, es decir, aquellos objetos cuyo estado no puede ser modificado después de su creación. Este tipo de objetos suelen ser inútiles una vez que su valor inicial ya no es necesario y cuando se necesita un nuevo valor, habrá que crear un nuevo objeto. El ejemplo más típico de este tipo de problemas lo tenemos con *java.util.String*. Muchos de los usos normales de *String* conllevan la creación de objetos intermedios. Por ejemplo:

```
static String reverse (String s){  
    String t = "";
```

```
    for (int i=s.length()-1; i >=0; --i)
        t += s.charAt(i);
    return t;
}
```

La línea 4 no modifica la variable t, ya que esta es inmutable, sino que crea una nueva cadena cada vez. Un posible código optimizado para la no creación de objetos innecesarios sería:

```
static String reverse(String s){
    StringBuffer t = new StringBuffer(s.length());
    for (int i=s.length()-1; i>=0; --i)
        t.append(s.charAt(i));
    return t.toString();
}
```