Java a tope:

J2ME

(Java 2 Micro Edition)

Sergio Gálvez Rojas
Lucas Ortega Díaz

Universidad de Málaga

JAVA A TOPE: J2ME (JAVA 2 MICRO EDITION). EDICIÓN ELECTRÓNICA

AUTORES: SERGIO GÁLVEZ ROJAS
LUCAS ORTEGA DÍAZ
ILUSTRACIÓN DE PORTADA: JOSÉ MIGUEL GÁLVEZ ROJAS

Depósito Legal: MA-1769-2003
ISBN: 84-688-4704-6

Java a tope:

J2ME

(Java 2 Micro Edition)

Sergio Gálvez Rojas
Doctor Ingeniero en Informática

Lucas Ortega Díaz
Ingeniero Técnico en Informática de Sistemas

Dpto. de Lenguajes y Ciencias de la Computación
E.T.S. de Ingeniería Informática
Universidad de Málaga

Índice

Prólogo v

1. Introducción 1
1.1. Descripción del capítulo 1
1.2. Análisis comparativo 2
1.3. Nociones básicas de J2ME 5
1.3.1. Máquinas Virtuales J2ME 5
1.3.2. Configuraciones 8
1.3.3. Perfiles 11
1.4. J2ME y las comunicaciones 14
1.5. OTA 15
1.5.1. Requerimientos Funcionales 16
1.5.2. Localización de la Aplicación 16
1.5.3. Instalación de *MIDlets* 17
1.5.4. Actualización de *MIDlets* 18
1.5.5. Ejecución de *MIDlets* 18
1.5.6. Eliminación de *MIDlets* 18

2 Herramientas de desarrollo 19
2.1 Descripción del capítulo 19
2.2 Desarrollo en línea de comandos 20
2.2.1 Instalación de Componentes 20
2.2.2 Fases de Desarrollo 21
2.2.3 Creación del archivo manifiesto 22
2.2.4 Creación del archivo JAR 23
2.2.5 Creación del archivo JAD 23
2.3 Desarrollo en entornos visuales 24
2.3.1 Instalación del *Sun One Studio Mobile Edition* 25
2.3.2 Instalación del *J2ME Wireless Toolkit 2.0* 25
2.3.3 Desarrollo de aplicaciones en el *Sun One Studio Mobile Edition* 25
2.3.4 Desarrollo con el *J2ME Wireless Toolkit 2.0*. 26
2.4 Uso de otros emuladores 28

3. Los *MIDlets* 31
3.1. Descripción del capítulo 31
3.2. El Gestor de Aplicaciones 31

3.2.1. Ciclo de vida de un *MIDlet* 32
3.2.2. Estados de un *MIDlet* en fase de ejecución 33
3.2.3. Estados de un *MIDlet* 34
3.3. El paquete javax.microedition.midlet 35
3.3.1. Clase MIDlet 35
3.3.2. Clase MIDletChangeStateException 37
3.4. Estructura de los *MIDlets* 38
3.5. Ejemplo práctico 40

4. La configuración CLDC 41
4.1. Introducción 41
4.2. Objetivos y requerimientos 42
4.2.1. Objetivos 42
4.2.2. Requerimientos 43
4.3. Diferencias de CLDC con J2SE 44
4.4. Seguridad en CLDC 46
4.5. Librerías CLDC 46
4.5.1. Objetivos generales 46
4.5.2. Compatibilidad 47
4.5.3. Clases heredadas de J2SE 47
4.5.4. Clases propias de CLDC 48

5. Interfaces gráficas de usuario 51
5.1. Descripción del capítulo 51
5.2. Introducción a las interfaces de usuario 51
5.2.1. La clase Display 53
5.2.2. La clase Displayable 54
5.2.3. Las clases Command y CommandListener 55
5.3. La interfaz de usuario de alto nivel 56
5.3.1. La clase Alert 57
5.3.2. La clase List 59
5.3.3. La clase TextBox 64
5.3.4. La clase Form 66
5.3.4.1. Manejo de Eventos 67
5.3.4.2. La Clase StringItem 68
5.3.4.3. La clase ImageItem 69
5.3.4.4. La clase TextField 70
5.3.4.5. La clase DateField 71
5.3.4.6. La clase ChoiceGroup 72
5.3.4.7. La clase Gauge 73
5.3.5. Creación de *MIDlets* usando la API de alto nivel 75
5.4. La interfaz de usuario de bajo nivel 89
5.4.1. Eventos de bajo nivel 91

5.4.2. Manipulación de elementos en una pantalla Canvas.....93
5.4.2.1. El método paint().....93
5.4.2.2. La clase Graphics.....94
5.4.2.3. Sistema de coordenadas.....94
5.4.2.4. Manejo de colores.....95
5.4.2.5. Manejo de texto.....95
5.4.2.6. Posicionamiento del texto.....96
5.4.2.7. Figuras geométricas.....97
5.4.3. Conceptos básicos para la creación de juegos en MIDP.....114
5.4.3.1. Eventos de teclado para juegos.....114
5.4.3.2. El paquete javax.microedition.lcdui.Game.....116
5.4.3.3. Técnicas útiles.....118
5.4.3.3.1. Double Buffering.....118
5.4.3.3.2. Clipping.....119
5.4.4. Diseño de un juego usando la clase javax.microedition.lcdui.game.....119
5.5. Internacionalización.....129
5.5.1. Aspectos a internacionalizar.....130
5.5.2. Limitaciones.....130
5.5.3. Soluciones para la internacionalización.....130
5.5.3.1. Uso de Atributos en el JAD.....131
5.5.3.2. Uso de ficheros con recursos de localización.....132

6. *Record management system*..... 135
6.1. Descripción del capítulo.....135
6.2. Conceptos Básicos.....135
6.2.1. Modelo de datos.....135
6.2.2. *Record Stores*.....137
6.3. Operaciones con *Record Stores*.....138
6.3.1. Creación de un *Record Store*.....138
6.3.2. Manipulación de registros.....139
6.4. Operaciones avanzadas con *Record Stores*.....144
6.4.1. Navegación a través de un *Record Store*.....144
6.4.2. Búsqueda de registros.....146
6.4.3. Ordenación de registros.....147
6.5. Manejo de eventos en los *Record Stores*.....148
6.6. Ejemplo práctico.....149

7. Comunicaciones..... 153
7.1. Descripción del capítulo.....153
7.2. Conceptos básicos.....153
7.3. Clases y conexiones del *Generic Connection Framework*.....155
7.3.1. Clase Connector.....155
7.3.2. Interfaz Connection.....156

7.3.3. Interfaz InputConnection 156
7.3.4. Interfaz OutputConnection 157
7.3.5. Interfaz StreamConnection 157
7.3.6. Interfaz ContentConnection 158
7.3.7. Interfaz StreamConnectionNotifier 159
7.3.8. Interfaz DatagramConnection 159
7.4. Comunicaciones HTTP 160
7.4.1. Estado de Establecimiento 160
7.4.1.1. Peticiones GET 161
7.4.1.2. Peticiones POST 162
7.4.2. Estado de Conexión 162
7.4.2.1. Respuesta del servidor 163
7.4.3. Estado de Cierre 167
7.5. Otras Conexiones 167
7.5.1. Interfaz HttpsConnection 168
7.5.2. Interfaz UDPDatagramConnection 168
7.5.3. Interfaz CommConnection 169
7.5.4. Interfaz SocketConnection 169
7.5.5. Interfaz SecureConnection 170
7.5.6. Interfaz ServerSocketConnection 171
7.6. Ejemplo práctico 171
7.6.1. Código del *MIDlet* 173
7.6.2. Código del *servlet* 183

Prólogo

l éxito del lenguaje de programación Java y de los diversos estándares que orbitan a su alrededor es, hoy por hoy, un hecho indiscutible. Los programadores en Java son los profesionales más demandados en el área de Desarrollo de Software que, a su vez, se enmarca en la más general disciplina de las Tecnologías de la Información y las Comunicaciones. Tener conocimientos de Java se ha convertido en una necesidad a la hora de entrar en este mercado laboral, y es nuestro deseo que, con esta serie de documentos electrónicos que se inicia con el presente volumen, los lectores vean facilitada su labor en el aprendizaje de tan extenso y completo lenguaje. No en vano el nombre del lenguaje, Java, alude a como se conoce popularmente al café de alta calidad en EE.UU.

Los constructores de Java optaron en su momento por dotar al lenguaje de un conjunto rico de capacidades orientadas a objetos que permitiese añadir librerías de apoyo a medida que fuera necesario y en función de las necesidades tecnológicas del momento. Es por ello que Java supone un núcleo de sorprendente consistencia teórica en torno al cual giran una multitud de bibliotecas con las más diversas orientaciones: desde bibliotecas dedicadas a la gestión de interfaces de usuario hasta aquéllas que se dedican al tratamiento de imágenes bidimensionales, pasando por las que se centran en difundir información multimedia, envío y recepción de correo electrónico, distribución de componentes software en Internet, etc.

Algunas de estas bibliotecas, a pesar de su indudable utilidad, no se suministran acompañadas de una documentación clara y concisa acerca de sus características y funcionamiento. En otros casos, la documentación existente es demasiado específica y no proporciona una visión de conjunto que permita apreciar desde una perspectiva general la filosofía con que fueron creadas dificultando, de esta manera, su difusión.

Con el objetivo de facilitar la comprensión, en la medida de nuestras posibilidades, de las bibliotecas estándares de este lenguaje, se presenta al lector el volumen que tiene ante sí. En este caso trataremos los mecanismos disponibles para desarrollar, instalar y ejecutar software basado en Java en dispositivos de pequeña capacidad con acceso a redes de información: principalmente teléfonos móviles.

Las características concretas de este tipo de dispositivos ha obligado a los desarrolladores de Java a construir un subconjunto del lenguaje y a reconfigurar sus principales bibliotecas para permitir su adaptación a un entorno con poca capacidad de memoria, poca velocidad de proceso, y pantallas de reducidas dimensiones. Todo esto hace que necesitemos una nueva plataforma de desarrollo y ejecución sobre la que centraremos nuestro estudio: Java 2 Micro Edition, o de forma abreviada: J2ME.

Capítulo 1:

Introducción

1.1 Descripción del capítulo

La empresa Sun Microsystems lanzó a mediados de los años 90 el lenguaje de programación Java que, aunque en un principio fue diseñado para generar aplicaciones que controlaran electrodomésticos como lavadoras, frigoríficos, etc, debido a su gran robustez e independencia de la plataforma donde se ejecutase el código, desde sus comienzos se utilizó para la creación de componentes interactivos integrados en páginas Web y programación de aplicaciones independientes. Estos componentes se denominaron applets y casi todo el trabajo de los programadores se dedicó al desarrollo de éstos. Con los años, Java ha progresado enormemente en varios ámbitos como servicios HTTP, servidores de aplicaciones, acceso a bases de datos (JDBC)... Como vemos Java se ha ido adaptando a las necesidades tanto de los usuarios como de las empresas ofreciendo soluciones y servicios tanto a unos como a otros. Debido a la explosión tecnológica de estos últimos años Java ha desarrollado soluciones personalizadas para cada ámbito tecnológico. Sun ha agrupado cada uno de esos ámbitos en una edición distinta de su lenguaje Java. Estas ediciones son Java 2 Stantard Edition, orientada al desarrollo de aplicaciones independientes y de applets, Java 2 Enterprise Edition, enfocada al entorno empresarial y Java 2 Micro Edition, orientada a la programación de aplicaciones para pequeños dispositivos. En esta última edición de Java es en la que vamos a centrar todo nuestro estudio de ahora en adelante.

La edición Java 2 Micro Edition fue presentada en 1999 por Sun Microsystems con el propósito de habilitar aplicaciones Java para pequeños dispositivos. En esta presentación, lo que realmente se enseñó fue una primera versión de una nueva Java Virtual Machine (JVM) que podía ejecutarse en dispositivos Palm. Para empezar podemos decir que Java Micro Edition es la versión del lenguaje Java que está orientada al desarrollo de aplicaciones para dispositivos pequeños con capacidades restringidas tanto en pantalla gráfica, como de procesamiento y memoria (teléfonos móviles, PDA`s, Handhelds, Pagers, etc). La tardía aparición de esta tecnología, (hemos visto que la tecnología Java nació a mediados de los 90 y Java Micro Edition apareció a finales), puede ser debido a que las necesidades de los usuarios de telefonía móvil ha cambiado mucho en estos últimos años y cada vez demandan más servicios y prestaciones por parte tanto de los terminales como de las compañías. Además el uso de esta tecnología depende del asentamiento en el mercado de otras, como GPRS, íntimamente asociada a J2ME y que no ha estado a nuestro alcance hasta hace poco. J2ME es la tecnología del

futuro para la industria de los dispositivos móviles. Actualmente las compañías telefónicas y los fabricantes de móviles están implantando los protocolos y dispositivos necesarios para soportarla.

Los objetivos de este documento son que el lector conozca los pilares sobre los que se asienta la plataforma J2ME y domine las clases que componen esta edición de Java para que pueda realizar rápidamente sus propias aplicaciones para teléfonos que dispongan de esta tecnología. Para la realización de dichas aplicaciones usaremos software de libre distribución disponible en la página oficial de Sun Microsystems y que cualquier programador puede descargar libremente. Usaremos también unos emuladores dónde podremos ejecutar y depurar nuestros programas, evitando así la descarga insatisfactoria de nuestra aplicación en un verdadero terminal. Antes de comenzar con los conceptos necesarios de Java Micro Edition vamos a realizar un estudio de las distintas versiones de Java.

1.2 Análisis comparativo

Sun, dispuesto a proporcionar las herramientas necesarias para cubrir las necesidades de todos los usuarios, creó distintas versiones de Java de acuerdo a las necesidades de cada uno. Según esto nos encontramos con que el paquete Java 2 lo podemos dividir en 3 ediciones distintas. J2SE (Java Standard Edition) orientada al desarrollo de aplicaciones independientes de la plataforma, J2EE (Java Enterprise Edition) orientada al entorno empresarial y J2ME (Java Micro Edition) orientada a dispositivos con capacidades restringidas. Veamos cuáles son las características de cada una de las versiones:

1. **Java 2 Platform, Standard Edition (J2SE)**: Esta edición de Java es la que en cierta forma recoge la iniciativa original del lenguaje Java. Tiene las siguientes características:

    - Inspirado inicialmente en C++, pero con componentes de alto nivel, como soporte nativo de strings y recolector de basura.
    - Código independiente de la plataforma, precompilado a bytecodes intermedio y ejecutado en el cliente por una JVM (Java Virtual Machine).
    - Modelo de seguridad tipo *sandbox* proporcionado por la JVM.
    - Abstracción del sistema operativo subyacente mediante un juego completo de APIs de programación.

    Esta versión de Java contiene el conjunto básico de herramientas usadas para desarrollar Java Applets, así cómo las APIs orientadas a la programación de aplicaciones de usuario final: Interfaz gráfica de usuario, multimedia, redes de comunicación, etc.

2. **Java 2 Platform, Enterprise Edition (J2EE)**: Esta versión está orientada al entorno empresarial. El software empresarial tiene unas características propias marcadas: está pensado no para ser ejecutado en un equipo, sino para ejecutarse sobre una red de ordenadores de manera distribuida y remota mediante EJBs (Enterprise Java Beans). De hecho, el sistema se monta sobre varias unidades o aplicaciones. En muchos casos, además, el software empresarial requiere que se sea capaz de integrar datos provenientes de entornos heterogéneos. Esta edición está orientada especialmente al desarrollo de servicios web, servicios de nombres, persistencia de objetos, XML, autenticación, APIs para la gestión de transacciones, etc. El cometido de esta especificación es ampliar la J2SE para dar soporte a los requisitos de las aplicaciones de empresa.
3. **Java 2 Platform, Micro Edition (J2ME)**: Esta versión de Java está enfocada a la aplicación de la tecnología Java en dispositivos electrónicos con capacidades computacionales y gráficas muy reducidas, tales como teléfonos móviles, PDAs o electrodomésticos inteligentes. Esta edición tiene unos componentes básicos que la diferencian de las otras versiones, como el uso de una máquina virtual denominada KVM (Kilo Virtual Machine, debido a que requiere sólo unos pocos Kilobytes de memoria para funcionar) en vez del uso de la JVM clásica, inclusión de un pequeño y rápido recolector de basura y otras diferencias que ya iremos viendo más adelante.

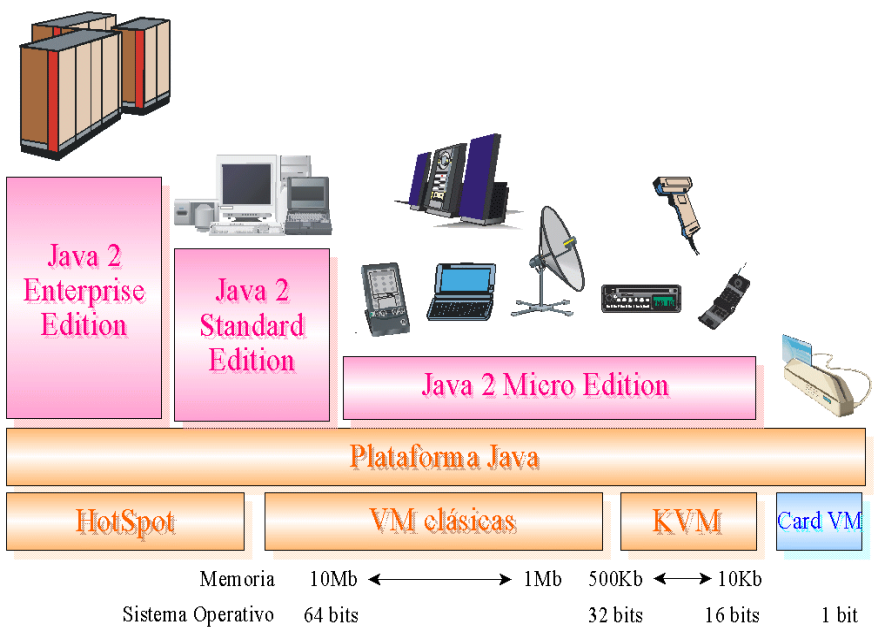

Figura 1.1 Arquitectura de la plataforma Java 2 de Sun

La Figura 1.1 nos muestra la arquitectura de la plataforma Java 2.

En la actualidad no es realista ver Java como un simple lenguaje de programación, si no como un conjunto de tecnologías que abarca a todos los ámbitos de la computación con dos elementos en común:

- El código fuente en lenguaje Java es compilado a código intermedio interpretado por una Java Virtual Machine (JVM), por lo que el código ya compilado es independiente de la plataforma.
- Todas las tecnologías comparten un conjunto más o menos amplio de APIs básicas del lenguaje, agrupadas principalmente en los paquetes java.lang y java.io.

Un claro ejemplo de éste último punto es que J2ME contiene una mínima parte de las APIs de Java. Esto es debido a que la edición estándar de APIs de Java ocupa 20 Mb, y los dispositivos pequeños disponen de una cantidad de memoria mucho más reducida. En concreto, J2ME usa 37 clases de la plataforma J2SE provenientes de los paquetes java.lang, java.io, java.util. Esta parte de la API que se mantiene fija forma parte de lo que se denomina "configuración" y ya hablaremos de ella más extensamente en el siguiente apartado. Otras diferencias con la plataforma J2SE vienen dadas por el uso de una máquina virtual distinta de la clásica JVM denominada KVM. Esta KVM tiene unas restricciones que hacen que no posea todas las capacidades incluidas en la JVM. Estas diferencias las veremos más detenidamente cuándo analicemos las capacidades de la KVM en el siguiente apartado.

Como vemos, J2ME representa una versión simplificada de J2SE. Sun separó estas dos versiones ya que J2ME estaba pensada para dispositivos con limitaciones de proceso y capacidad gráfica. También separó J2SE de J2EE porque este último exigía unas características muy pesadas o especializadas de E/S, trabajo en red, etc. Por tanto, separó ambos productos por razones de eficiencia. Hoy, J2EE es un superconjunto de J2SE pues contiene toda la funcionalidad de éste y más características, así como J2ME es un subconjunto de J2SE (excepto por el paquete javax.microedition) ya que, como se ha mencionado, contiene varias limitaciones con respecto a J2SE.

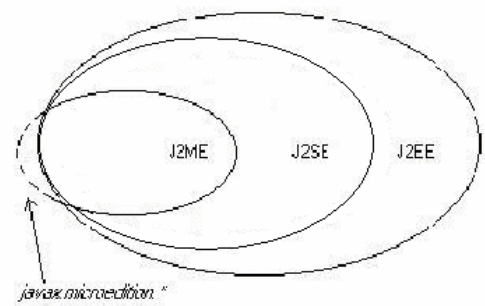

Figura 1.2 Relación entre las APIs de la plataforma Java.

Sólo de manera muy simplista se puede considerar a J2ME y J2EE como versiones reducidas y ampliadas de J2SE respectivamente (ver Figura 1.2): en realidad cada una de las ediciones está enfocada a ámbitos de aplicación muy distintos. Las necesidades computacionales y APIs de programación requeridas para un juego ejecutándose en un móvil difieren bastante con las de un servidor distribuido de aplicaciones basado en EJB.

1.3 Nociones básicas de J2ME

Ya hemos visto qué es Java Micro Edition y la hemos enmarcado dentro de la plataforma Java2. En este apartado vamos a ver cuales son los componentes que forman parte de esta tecnología.

- Por un lado tenemos una serie de máquinas virtuales Java con diferentes requisitos, cada una para diferentes tipos de pequeños dispositivos.
- Configuraciones, que son un conjunto de clases básicas orientadas a conformar el corazón de las implementaciones para dispositivos de características específicas. Existen 2 configuraciones definidas en J2ME: Connected Limited Device Configuration (CLDC) enfocada a dispositivos con restricciones de procesamiento y memoria, y Connected Device Configuration (CDC) enfocada a dispositivos con más recursos.
- Perfiles, que son unas bibliotecas Java de clases específicas orientadas a implementar funcionalidades de más alto nivel para familias específicas de dispositivos.

Un entorno de ejecución determinado de J2ME se compone entonces de una selección de:

a) Máquina virtual.
b) Configuración.
c) Perfil.
d) Paquetes Opcionales.

La arquitectura de un entorno de ejecución la podemos ver en la Figura 1.3. A continuación estudiaremos en profundidad cada uno de estos tres componentes.

1.3.1 Máquinas Virtuales J2ME

Una máquina virtual de Java (JVM) es un programa encargado de interpretar código intermedio (bytecode) de los programas Java precompilados a código máquina ejecutable por la plataforma, efectuar las llamadas pertinentes al sistema operativo subyacente y observar las reglas de seguridad y corrección de código definidas para el lenguaje Java. De esta forma, la JVM proporciona al programa Java independencia de la plataforma con respecto al hardware y al sistema operativo subyacente. Las

implementaciones tradicionales de JVM son, en general, muy pesadas en cuanto a memoria ocupada y requerimientos computacionales. J2ME define varias JVMs de referencia adecuadas al ámbito de los dispositivos electrónicos que, en algunos casos, suprimen algunas características con el fin de obtener una implementación menos exigente.

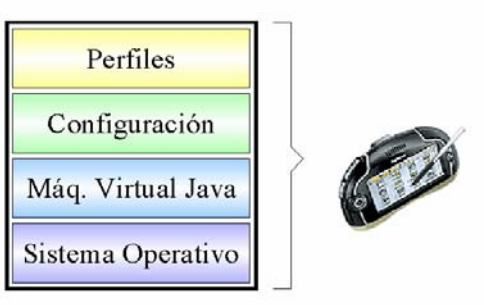

Figura 1.3 Entorno de ejecución.

Ya hemos visto que existen 2 configuraciones CLDC y CDC, cada una con unas características propias que veremos en profundidad más adelante. Como consecuencia, cada una requiere su propia máquina virtual. La VM (Virtual Machine) de la configuración CLDC se denomina KVM y la de la configuración CDC se denomina CVM. Veremos a continuación las características principales de cada una de ellas:

- **KVM**

Se corresponde con la Máquina Virtual más pequeña desarrollada por Sun. Su nombre KVM proviene de Kilobyte (haciendo referencia a la baja ocupación de memoria, entre 40Kb y 80Kb). Se trata de una implementación de Máquina Virtual reducida y especialmente orientada a dispositivos con bajas capacidades computacionales y de memoria. La KVM está escrita en lenguaje C, aproximadamente unas 24000 líneas de código, y fue diseñada para ser:

- Pequeña, con una carga de memoria entre los 40Kb y los 80 Kb, dependiendo de la plataforma y las opciones de compilación.
- Alta portabilidad.
- Modulable.
- Lo más completa y rápida posible y sin sacrificar características para las que fue diseñada.

Sin embargo, esta baja ocupación de memoria hace que posea algunas limitaciones con respecto a la clásica *Java Virtual Machine* (JVM):

1. No hay soporte para tipos en coma flotante. No existen por tanto los tipos double ni float. Esta limitación está presente porque los dispositivos carecen del hardware necesario para estas operaciones.
2. No existe soporte para JNI (*Java Native Interface*) debido a los recursos limitados de memoria.

3. No existen cargadores de clases (*class loaders*) definidos por el usuario. Sólo existen los predefinidos.
4. No se permiten los grupos de hilos o hilos *daemon*. Cuándo queramos utilizar grupos de hilos utilizaremos los objetos *Colección* para almacenar cada hilo en el ámbito de la aplicación.
5. No existe la finalización de instancias de clases. No existe el método Object.finalize().
6. No hay referencias débiles.[1]
7. Limitada capacidad para el manejo de excepciones debido a que el manejo de éstas depende en gran parte de las APIs de cada dispositivo por lo que son éstos los que controlan la mayoría de las excepciones.
8. Reflexión[2].

Aparte de la no inclusión de estas características, la verificación de clases merece un comentario aparte. El verificador de clases estándar de Java es demasiado grande para la KVM. De hecho es más grande que la propia KVM y el consumo de memoria es excesivo, más de 100Kb para las aplicaciones típicas. Este verificador de clases es el encargado de rechazar las clases no válidas en tiempo de ejecución. Este mecanismo verifica los *bytecodes* de las clases Java realizando las siguientes comprobaciones:

- Ver que el código no sobrepase los límites de la pila de la VM.
- Comprobar que no se utilizan las variables locales antes de ser inicializadas.
- Comprobar que se respetan los campos, métodos y los modificadores de control de acceso a clases.

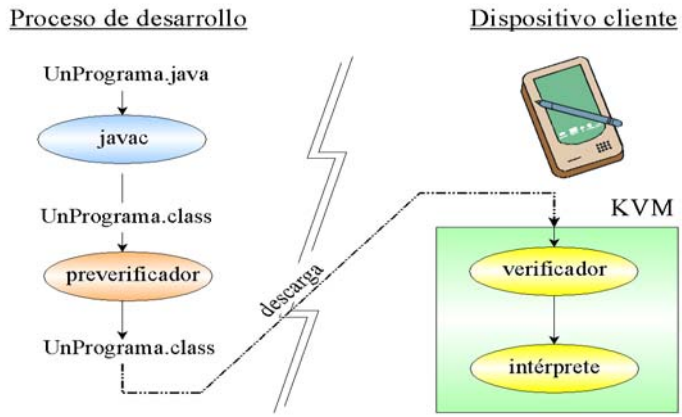

Figura 1.4 Preverificación de clases en CDLC/KVM.

[1] Un objeto que está siendo apuntado mediante una referencia débil es un candidato para la recolección de basura. Estas referencias están permitidas en J2SE, pero no en J2ME.

[2] La reflexión es el mecanismo por el cual los objetos pueden obtener información de otros objetos en tiempo de ejecución como, por ejemplo, los archivos de clases cargados o sus campos y métodos.

Por esta razón los dispositivos que usen la configuración CLDC y KVM introducen un algoritmo de verificación de clases en dos pasos. Este proceso puede apreciarse gráficamente en la Figura 1.4.

La KVM puede ser compilada y probada en 3 plataformas distintas:

1. Solaris Operating Environment.
2. Windows
3. PalmOs

- **CVM**

La CVM (Compact Virtual Machine) ha sido tomada como Máquina Virtual Java de referencia para la configuración CDC y soporta las mismas características que la Máquina Virtual de J2SE. Está orientada a dispositivos electrónicos con procesadores de 32 bits de gama alta y en torno a 2Mb o más de memoria RAM. Las características que presenta esta Máquina Virtual son:

1. Sistema de memoria avanzado.
2. Tiempo de espera bajo para el recolector de basura.
3. Separación completa de la VM del sistema de memoria.
4. Recolector de basura modularizado.
5. Portabilidad.
6. Rápida sincronización.
7. Ejecución de las clases Java fuera de la memoria de sólo lectura (ROM).
8. Soporte nativo de hilos.
9. Baja ocupación en memoria de las clases.
10. Proporciona soporte e interfaces para servicios en Sistemas Operativos de Tiempo Real.
11. Conversión de hilos Java a hilos nativos.
12. Soporte para todas las características de Java2 v1.3 y librerías de seguridad, referencias débiles, Interfaz Nativa de Java (JNI), invocación remota de métodos (RMI), Interfaz de depuración de la Máquina Virtual (JVMDI).

1.3.2 Configuraciones

Ya hemos mencionado algo anteriormente relacionado con las configuraciones. Para tenerlo bien claro diremos que una configuración es el conjunto mínimo de APIs Java que permiten desarrollar aplicaciones para un grupo de dispositivos. Éstas APIs describen las características básicas, comunes a todos los dispositivos:

- Características soportadas del lenguaje de programación Java.
- Características soportadas por la Máquina Virtual Java.
- Bibliotecas básicas de Java y APIs soportadas.

Como ya hemos visto con anterioridad, existen dos configuraciones en J2ME: CLDC, orientada a dispositivos con limitaciones computacionales y de memoria y

CDC, orientada a dispositivos con no tantas limitaciones. Ahora veremos un poco más en profundidad cada una de estas configuraciones.

- **Configuración de dispositivos con conexión, CDC** (*Connected Limited Configuration*)

La CDC está orientada a dispositivos con cierta capacidad computacional y de memoria. Por ejemplo, decodificadores de televisión digital, televisores con internet, algunos electrodomésticos y sistemas de navegación en automóviles. CDC usa una Máquina Virtual Java similar en sus características a una de J2SE, pero con limitaciones en el apartado gráfico y de memoria del dispositivo. Ésta Máquina Virtual es la que hemos visto como CVM (Compact Virtual Machine). La CDC está enfocada a dispositivos con las siguientes capacidades:

- Procesador de 32 bits.
- Disponer de 2 Mb o más de memoria total, incluyendo memoria RAM y ROM.
- Poseer la funcionalidad completa de la Máquina Virtual Java2.
- Conectividad a algún tipo de red.

La CDC está basada en J2SE v1.3 e incluye varios paquetes Java de la edición estándar. Las peculiaridades de la CDC están contenidas principalmente en el paquete javax.microedition.io, que incluye soporte para comunicaciones http y basadas en datagramas. La Tabla 1.1 nos muestra las librerías incluidas en la CDC.

Nombre de Paquete CDC	Descripción
java.io	Clases e interfaces estándar de E/S.
java.lang	Clases básicas del lenguaje.
java.lang.ref	Clases de referencia.
java.lang.reflect	Clases e interfaces de reflection.
java.math	Paquete de matemáticas.
java.net	Clases e interfaces de red.
java.security	Clases e interfaces de seguridad
java.security.cert	Clases de certificados de seguridad.
java.text	Paquete de texto.
java.util	Clases de utilidades estándar.
java.util.jar	Clases y utilidades para archivos JAR.
java.util.zip	Clases y utilidades para archivos ZIP y comprimidos.
javax.microedition.io	Clases e interfaces para conexión genérica CDC.

Tabla 1.1 Librerías de configuración CDC.

- **Configuración de dispositivos limitados con conexión, CLDC** (*Connected Limited Device Configuration*).

La CLDC está orientada a dispositivos dotados de conexión y con limitaciones en cuanto a capacidad gráfica, cómputo y memoria. Un ejemplo de éstos dispositivos

son: teléfonos móviles, buscapersonas (pagers), PDAs, organizadores personales, etc. Ya hemos dicho que CLDC está orientado a dispositivos con ciertas restricciones. Algunas de éstas restricciones vienen dadas por el uso de la KVM, necesaria al trabajar con la CLDC debido a su pequeño tamaño. Los dispositivos que usan CLDC deben cumplir los siguientes requisitos:

- Disponer entre 160 Kb y 512 Kb de memoria total disponible. Como mínimo se debe disponer de 128 Kb de memoria no volátil para la Máquina Virtual Java y las bibliotecas CLDC, y 32 Kb de memoria volátil para la Máquina Virtual en tiempo de ejecución.
- Procesador de 16 o 32 bits con al menos 25 Mhz de velocidad.
- Ofrecer bajo consumo, debido a que éstos dispositivos trabajan con suministro de energía limitado, normalmente baterías.
- Tener conexión a algún tipo de red, normalmente sin cable, con conexión intermitente y ancho de banda limitado (unos 9600 bps).

La CLDC aporta las siguientes funcionalidades a los dispositivos:

- Un subconjunto del lenguaje Java y todas las restricciones de su Máquina Virtual (KVM).
- Un subconjunto de las bibliotecas Java del núcleo.
- Soporte para E/S básica.
- Soporte para acceso a redes.
- Seguridad.

La Tabla 1.2 nos muestra las librerías incluidas en la CLDC.

Nombre de paquete CLDC	Descripción
java.io	Clases y paquetes estándar de E/S. Subconjunto de J2SE.
java.lang	Clases e interfaces de la Máquina Virtual. Subconj. de J2SE.
java.util	Clases, interfaces y utilidades estándar. Subconj. de J2SE.
javax.microedition.io	Clases e interfaces de conexión genérica CLDC

Tabla 1.2 Librerías de configuración CLDC.

Un aspecto muy a tener en cuenta es la seguridad en CLDC. Esta configuración posee un modelo de seguridad *sandbox* al igual que ocurre con los applets. En el capítulo 4 dedicado exclusivamente a la configuración CLDC se explica totalmente este modelo.

En cualquier caso, una determinada Configuración no se encarga del mantenimiento del ciclo de vida de la aplicación, interfaces de usuario o manejo de eventos, sino que estas responsabilidades caen en manos de los **perfiles**.

1.3.3 Perfiles

Acabamos de decir que el perfil es el que define las APIs que controlan el ciclo de vida de la aplicación, interfaz de usuario, etc. Más concretamente, un perfil es un conjunto de APIs orientado a un ámbito de aplicación determinado. Los perfiles identifican un grupo de dispositivos por la funcionalidad que proporcionan (electrodomésticos, teléfonos móviles, etc.) y el tipo de aplicaciones que se ejecutarán en ellos. Las librerías de la interfaz gráfica son un componente muy importante en la definición de un perfil. Aquí nos podemos encontrar grandes diferencias entre interfaces, desde el menú textual de los teléfonos móviles hasta los táctiles de los PDAs.

El perfil establece unas APIs que definen las características de un dispositivo, mientras que la configuración hace lo propio con una familia de ellos. Esto hace que a la hora de construir una aplicación se cuente tanto con las APIs del perfil como de la configuración. Tenemos que tener en cuenta que un perfil siempre se construye sobre una configuración determinada. De este modo, podemos pensar en un perfil como un conjunto de APIs que dotan a una configuración de funcionalidad específica.

Ya hemos visto los conceptos necesarios para entender cómo es un entorno de ejecución en Java Micro Edition. Este entorno de ejecución se estructura en capas, una construida sobre la otra como veíamos en la figura 3.

Anteriormente vimos que para una configuración determinada se usaba una Máquina Virtual Java específica. Teníamos que con la configuración CDC usábamos la CVM y que con la configuración CLDC usábamos la KVM. Con los perfiles ocurre lo mismo. Existen unos perfiles que construiremos sobre la configuración CDC y otros que construiremos sobre la CLDC. Para la configuración CDC tenemos los siguientes perfiles:

- *Foundation Profile*.
- *Personal Profile*.
- *RMI Profile*.

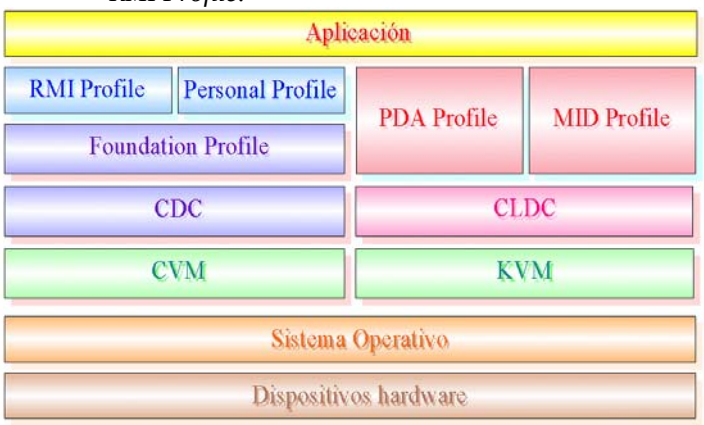

Figura 1.5 Arquitectura del entorno de ejecución de J2ME.

y para la configuración CLDC tenemos los siguientes:

- *PDA Profile*.
- *Mobile Information Device Profile* (MIDP).

En la Figura 1.5 se puede ver como quedaría el esquema del entorno de ejecución al completo.

Un perfil puede ser construido sobre cualquier otro. Sin embargo, una plataforma J2ME sólo puede contener una configuración.

A continuación vamos a ver con detenimiento cada uno de estos perfiles:

- ***Foundation Profile***: Este perfil define una serie de APIs sobre la CDC orientadas a dispositivos que carecen de interfaz gráfica como, por ejemplo, decodificadores de televisión digital. Este perfil incluye gran parte de los paquetes de la J2SE, pero excluye totalmente los paquetes "java.awt" *Abstract Windows Toolkit* (AWT) y "java.swing" que conforman la interfaz gráfica de usuario (GUI) de J2SE. Si una aplicación requiriera una GUI, entonces sería necesario un perfil adicional. Los paquetes que forman parte del *Foundation Profile* se muestran en la Tabla 1.3.

Paq. del Foundation Profile	Descripción
java.lang	Soporte del lenguaje Java
java.util	Añade soporte completo para zip y otras funcionalidades (java.util.Timer)
java.net	Incluye sockets TCP/IP y conexiones HTTP
java.io	Clases Reader y Writer de J2SE
java.text	Incluye soporte para internacionalización
java.security	Incluye códigos y certificados

Tabla 1.3 Librerías del *Foundation Profile*.

- ***Personal Profile***: El *Personal Profile* es un subconjunto de la plataforma J2SE v1.3, y proporciona un entorno con un completo soporte gráfico AWT. El objetivo es el de dotar a la configuración CDC de una interfaz gráfica completa, con capacidades web y soporte de *applets* Java. Este perfil requiere una implementación del *Foundation Profile*. La Tabla 1.4 nos muestra los paquetes que conforman el *Personal Profile* v1.0.

Paq. del Personal Profile	Descripción
java.applet	Clases necesitadas para crear applets o que son usadas por ellos
java.awt	Clases para crear GUIs con AWT
java.awt.datatransfer	Clases e interfaces para transmitir datos entre aplicaciones
java.awt.event	Clases e interfaces para manejar eventos AWT
java.awt.font	Clases e interfaces para la manipulación de fuentes

java.awt.im	Clases e interfaces para definir métodos editores de entrada
java.awt.im.spi	Interfaces que añaden el desarrollo de métodos editores de entrada para cualquier entorno de ejecución Java
java.awt.image	Clases para crear y modificar imágenes
java.beans	Clases que soportan JavaBeans
javax.microedition.xlet	Interfaces que usa el Personal Profile para la comunicación.

Tabla 1.4 Librerías del *Personal Profile*

- ***RMI Profile*:** Este perfil requiere una implementación del *Foundation Profile* se construye encima de él. El perfil RMI soporta un subconjunto de las APIs J2SE v1.3 RMI. Algunas características de estas APIs se han eliminado del perfil RMI debido a las limitaciones de cómputo y memoria de los dispositivos. Las siguientes propiedades se han eliminado del J2SE RMI v1.3:
  - Java.rmi.server.disableHTTP.
  - Java.rmi.activation.port.
  - Java.rmi.loader.packagePrefix.
  - Java.rmi.registry.packagePrefix.
  - Java.rmi.server.packagePrefix.

- ***PDA Profile*:** El PDA Profile está construido sobre CLDC. Pretende abarcar PDAs de gama baja, tipo Palm, con una pantalla y algún tipo de puntero (ratón o lápiz) y una resolución de al menos 20000 pixels (al menos 200x100 pixels) con un factor 2:1. No es posible dar mucha más información porque en este momento este perfil se encuentra en fase de definición.

- ***Mobile Information Device Profile* (MIDP):** Este perfil está construido sobre la configuración CLDC. Al igual que CLDC fue la primera configuración definida para J2ME, MIDP fue el primer perfil definido para esta plataforma. Este perfil está orientado para dispositivos con las siguientes características:
  - Reducida capacidad computacional y de memoria.
  - Conectividad limitada (en torno a 9600 bps).
  - Capacidad gráfica muy reducida (mínimo un display de 96x54 pixels monocromo).
  - Entrada de datos alfanumérica reducida.
  - 128 Kb de memoria no volátil para componentes MIDP.
  - 8 Kb de memoria no volátil para datos persistentes de aplicaciones.
  - 32 Kb de memoria volátil en tiempo de ejecución para la pila Java.

  Los tipos de dispositivos que se adaptan a estas características son: teléfonos móviles, buscapersonas (pagers) o PDAs de gama baja con conectividad.

  El perfil MIDP establece las capacidades del dispositivo, por lo tanto, especifica las APIs relacionadas con:
  - La aplicación (semántica y control de la aplicación MIDP).

- Interfaz de usuario.
- Almacenamiento persistente.
- Trabajo en red.
- Temporizadores.

En la Tabla 1.5 podemos ver cuáles son los paquetes que están incluidos en el perfil MIDP.

Paquetes del MIDP	Descripción
javax.microedition.lcdui	Clases e interfaces para GUIs
javax.microedition.rms	*Record Management Storage*. Soporte para el almacenamiento persistente del dispositivo
javax.microedition.midlet	Clases de definición de la aplicación
javax.microedition.io	Clases e interfaces de conexión genérica
java.io	Clases e interfaces de E/S básica
java.lang	Clases e interfaces de la Máquina Virtual
java.util	Clases e interfaces de utilidades estándar

Tabla 1.5 Librerías del perfil MIDP.

Las aplicaciones que realizamos utilizando MIDP reciben el nombre de *MIDlets* (por simpatía con *APPlets*). **Decimos así que un *MIDlet* es una aplicación Java realizada con el perfil MIDP sobre la configuración CLDC**. En los temas siguientes nos centraremos en la creación de estos *MIDlets* ya que es un punto de referencia para cualquier programador de J2ME. Además, desde un punto de vista práctico MIDP es el único perfil actualmente disponible.

1.4 J2ME y las comunicaciones

Ya hemos visto que una característica importante que tienen que tener los dispositivos que hagan uso de J2ME, más específicamente CLDC/MIDP (a partir de ahora nuestro estudio de J2ME se va a basar exclusivamente en el entorno CLDC/MIDP) es que necesitan poseer conexión a algún tipo de red, por lo que la comunicación de estos dispositivos cobra una gran importancia. En este apartado vamos a ver cómo participan las distintas tecnologías en estos dispositivos y cómo influyen en el uso de la tecnología J2ME. Para ello vamos a centrarnos en un dispositivo en especial: los teléfonos móviles. De aquí en adelante todo el estudio y la creación de aplicaciones se realizarán para este dispositivo. Esto es así debido a la rápida evolución que han tenido los teléfonos móviles en el sector de las comunicaciones, lo que ha facilitado el desarrollo, por parte de algunas empresas, de herramientas que usaremos para crear las aplicaciones.

Uno de los primeros avances de la telefonía móvil en el sector de las comunicaciones se dio con la aparición de la tecnología WAP. WAP proviene de *Wireless Application Protocol* o Protocolo de Aplicación Inalámbrica. Es un protocolo con el que se ha tratado de dotar a los dispositivos móviles de un pequeño y limitado

navegador web. WAP exige la presencia de una puerta de enlace encargado de actuar cómo intermediario entre Internet y el terminal. Esta puerta de enlace o *gateway* es la que se encarga de convertir las peticiones WAP a peticiones web habituales y viceversa. Las páginas que se transfieren en una petición usando WAP no están escritas en HTML, si no que están escritas en WML, un subconjunto de éste. WAP ha sido un gran avance, pero no ha resultado ser la herramienta que se prometía. La navegación es muy engorrosa (la introducción de URLs largas por teclado es muy pesada, además de que cualquier error en su introducción requiere que se vuelva a escribir la dirección completa por el teclado del móvil). Además su coste es bastante elevado ya que el pago de uso de esta tecnología se realiza en base al tiempo de conexión a una velocidad, que no es digamos, muy buena.

Otra tecnología relacionada con los móviles es SMS. SMS son las siglas de *Short Message System* (Sistema de Mensajes Cortos). Actualmente este sistema nos permite comunicarnos de una manera rápida y barata con quien queramos sin tener que establecer una comunicación con el receptor del mensaje. Con ayuda de J2ME, sin embargo, podemos realizar aplicaciones de chat o mensajería instantánea.

Los últimos avances de telefonía móvil nos llevan a las conocidas cómo generación 2 y 2´5 que hacen uso de las tecnologías GSM y GPRS respectivamente. GSM es una conexión telefónica que soporta una circulación de datos, mientras que GPRS es estrictamente una red de datos que mantiene una conexión abierta en la que el usuario paga por la cantidad de información intercambiada y no por el tiempo que permanezca conectado. La aparición de la tecnología GPRS no hace más que favorecer el uso de J2ME y es, además, uno de los pilares sobre los que se asienta J2ME, ya que podemos decir que es el vehículo sobre el que circularán las futuras aplicaciones J2ME.

Otras tecnologías que favorecen la comunicación son Bluetooth y las redes inalámbricas que dan conectividad a ordenadores, PDAs y teléfonos móviles. De hecho, una gran variedad de estos dispositivos disponen de soporte bluetooth. Esto nos facilita la creación de redes con un elevado ancho de banda en distancias pequeñas (hasta 100 metros).

Es de esperar que todas estas tecnologías favorezcan el uso de J2ME en el mercado de la telefonía móvil.

1.5 OTA

Las aplicaciones realizadas con J2ME están pensadas para que puedan ser descargadas a través de una conexión a internet. El medio empleado para garantizar esta descarga recibe el nombre de OTA (*Over the Air*), y viene totalmente reflejado en un documento denominado *«Over The Air User Initiater Provisioning Recommended Practice»*, Sun Microsystems, 2002. Antes de nada hemos de decir que una aplicación J2ME está formada por un archivo JAR que es el que contiene a la aplicación en sí y un archivo JAD (*Java Archive Descriptor*) que contiene diversa información sobre la aplicación. El propósito de este documento es describir como se pueden descargar los

MIDlets "*over the air*" y establecer cuáles son los requerimientos impuestos a los dispositivos que realizan estas descargas.

1.5.1 Requerimientos Funcionales

Los dispositivos deben proporcionar mecanismos mediante los cuales podamos encontrar los *MIDlets* que deseemos descargar. En algunos casos, encontraremos los *MIDlets* a través de un navegador WAP o a través de una aplicación residente escrita específicamente para identificar *MIDlets*. Otros mecanismos como Bluetooth, cable serie, etc, pueden ser soportados por el dispositivo.

El programa encargado de manejar la descarga y ciclo de vida de los *MIDlets* en el dispositivo se llama Gestor de Aplicaciones o AMS (*Application Management Software*).

Un dispositivo que posea la especificación MIDP debe ser capaz de:

- Localizar archivos JAD vinculados a un *MIDlet* en la red.
- Descargar el MIDlet y el archivo JAD al dispositivo desde un servidor usando el protocolo HTTP 1.1 u otro que posea su funcionalidad.
- Enviar el nombre de usuario y contraseña cuando se produzca una respuesta HTTP por parte del servidor 401 (*Unauthorized*) o 407 (*Proxy Authentication Required*).
- Instalar el *MIDlet* en el dispositivo.
- Ejecutar *MIDlets*.
- Permitir al usuario borrar *MIDlets* instalados.

1.5.2 Localización de la Aplicación

El descubrimiento de una aplicación es el proceso por el cual un usuario a través de su dispositivo localiza un *MIDlet*. El usuario debe ser capaz de ver la descripción del *MIDlet* a través de un enlace que, una vez seleccionado, inicializa la instalación del *MIDlet*. Si éste enlace se refiere a un archivo JAR, el archivo y su URL son enviados al AMS del dispositivo para empezar el proceso de instalación. Si el enlace se refiere a un archivo JAD se realizan los siguientes pasos:

1. El descriptor de la aplicación (archivo JAD) y su URL son transferidos al AMS para empezar la instalación. Este descriptor es usado por el AMS para determinar si el *MIDlet* asociado puede ser instalado y ejecutado satisfactoriamente.

2. Este archivo JAD debe ser convertido al formato *Unicode* antes de ser usado. Los atributos del JAD deben ser comprensibles, acorde con la sintaxis de la especificación MIDP, y todos los atributos requeridos por la especificación MIDP deben estar presentes en el JAD.

3. El usuario debería de tener la oportunidad de confirmar que desea instalar el *MIDlet*. Asimismo debería de ser informado si se intenta instalar una versión anterior del *MIDlet* o si la versión es la misma que ya está instalada. Si existen problemas de memoria con la ejecución del *MIDlet* se intentarían solucionar liberando componentes de memoria para dejar espacio suficiente.

1.5.3 Instalación de *MIDlets*

La instalación de la aplicación es el proceso por el cual el *MIDlet* es descargado al dispositivo y puede ser utilizado por el usuario.

Cuando existan múltiples *MIDlets* en la aplicación que deseamos descargar, el usuario debe ser avisado de que existen más de uno.

Durante la instalación, el usuario debe ser informado del progreso de ésta y se le debe de dar la oportunidad de cancelarla. La interrupción de la instalación debe dejar al dispositivo con el mismo estado que cuando se inició ésta. Veamos cuáles son los pasos que el AMS sigue para la instalación de un *MIDlet*:

1. Si el JAD fue lo primero que descargó el AMS, el *MIDlet* debe tener exactamente la misma URL especificada en el descriptor.

2. Si el servidor responde a la petición del *MIDlet* con un código 401 (*Unauthorized*) o un 407 (*Proxy Authentication Required*), el dispositivo debe enviar al servidor las correspondientes credenciales.

3. El MIDlet y las cabeceras recibidas deben ser chequeadas para verificar que el MIDlet descargado puede ser instalado en el dispositivo. El usuario debe ser avisado de los siguientes problemas durante la instalación:

- Si no existe suficiente memoria para almacenar el *MIDlet*, el dispositivo debe retornar el Código de Estado (*Status Code*) 901.
- Si el JAR no está disponible en la URL del JAD, el dispositivo debe retornar el Código 907.
- Si el JAR recibido no coincide con el descrito por el JAD, el dispositivo debe retornar el Código 904.
- Si el archivo *manifest* o cualquier otro no puede ser extraído del JAR, o existe algún error al extraerlo, el dispositivo debe retornar el Código 907.
- Si los atributos “MIDlet-Name”, “MIDlet-Version” y “MIDlet Vendor” del archivo JAD, no coinciden con los extraidos del archivo *manifes*t del JAR, el dispositivo debe retornar el Código 905.
- Si la aplicación falla en la autenticación, el dispositivo debe retornar el Código 909.
- Si falla por otro motivo distinto del anterior, debe retornar el Código 911.
- Si los servicios de conexión se pierden durante la instalación, se debe retornar el Código 903 si es posible.

La instalación se da por completa cuando el *MIDlet* esté a nuestra disposición en el dispositivo, o no haya ocurrido un error irrecuperable.

1.5.4 Actualización de *MIDlets*

La actualización se realiza cuando instalamos un *MIDlet* sobre un dispositivo que ya contenía una versión anterior de éste. El dispositivo debe ser capaz de informar al usuario cual es la versión de la aplicación que tiene instalada.

Cuando comienza la actualización, el dispositivo debe informar si la versión que va a instalar es más nueva, más vieja o la misma de la ya instalada y debe obtener verificación por parte del usuario antes de continuar con el proceso.

En cualquier caso, un *MIDlet* que no posea firma no debe de reemplazar de ninguna manera a otro que sí la tenga.

1.5.5 Ejecución de *MIDlets*

Cuando un usuario comienza a ejecutar un *MIDlet*, el dispositivo debe invocar a las clases CLDC y MIDP requeridas por la especificación MIDP. Si existen varios *MIDlets* presentes, la interfaz de usuario debe permitir al usuario seleccionar el *MIDlet* que desea ejecutar.

1.5.6 Eliminación de *MIDlets*

Los dispositivos deben permitir al usuario eliminar *MIDlets*. Antes de eliminar una aplicación el usuario debe dar su confirmación. El dispositivo debería avisar al usuario si ocurriese alguna circunstancia especial durante la eliminación del *MIDlet*. Por ejemplo, el *MIDlet* a borrar podría contener a otros *MIDlets*, y el usuario debería de ser alertado ya que todos ellos quedarían eliminados.

Capítulo 2:
Herramientas de desarrollo

2.1 Descripción del capítulo

En este capítulo vamos a ver las herramientas que se necesitan para construir nuestros *MIDlets*. El proceso de creación de éstos se puede realizar básicamente de dos formas:

- A través de la línea de comandos: en este caso no haremos uso de ninguna herramienta especial para el desarrollo de nuestra aplicación.
- A través de un entorno visual: el uso de este tipo de herramientas nos facilitará mucho, como veremos, el proceso de desarrollo de los *MIDlets*.

Los *MIDlets* que vamos a crear serán ejecutados en dispositivos MID (*Mobile Information Device*) y no en la máquina donde los desarrollamos. Por esta razón, sea cual sea el método de creación que usemos, tendremos que hacer uso de algún emulador para realizar las pruebas de nuestra aplicación.

Este emulador puede representar a un dispositivo genérico o puede ser de algún modelo de MID específico. El uso de estos emuladores ya lo veremos más adelante.

Antes de empezar a explicar los pasos a seguir para instalar las herramientas necesarias que usaremos para la construcción de los *MIDlets*, vamos a ver las etapas básicas que han de realizarse con este objetivo:

1. Desarrollo: En esta fase vamos a escribir el código que conforma nuestro *MIDlet*.
2. Compilación: Se compilará nuestra aplicación haciendo uso de un compilador J2SE.
3. Preverificación: Antes de empaquetar nuestro *MIDlet* es necesario realizar un proceso de preverificación de las clases Java. En esta fase se realiza un examen del código del *MIDlet* para ver que no viola ninguna restricción de seguridad de la plataforma J2ME.
4. Empaquetamiento: En esta fase crearemos un archivo JAR que contiene los recursos que usa nuestra aplicación, y crearemos también un archivo descriptor JAD.
5. Ejecución: Para esta fase haremos uso de los emuladores que nos permitirán ejecutar nuestro *MIDlet*.
6. Depuración: Esta última fase nos permitirá depurar los fallos detectados en la fase anterior de nuestro *MIDlet*.

Básicamente cualquier aplicación en Java sigue este proceso de desarrollo excepto por las etapas de empaquetamiento y preverificación que es exclusivo de las aplicaciones desarrolladas usando la plataforma J2ME.

2.2 Desarrollo en línea de comandos

Para empezar vamos a ver cómo se realizaría el desarrollo de aplicaciones MIDP usando únicamente la línea de comandos. Aquí veremos muy claramente los pasos de desarrollo anteriores ya que vamos a tener que realizar manualmente cada uno de ellos.

2.2.1 Instalación de Componentes

Las herramientas que vamos a usar para el desarrollo de *MIDlets* serán las siguientes:

- Un editor de texto cualquiera como, por ejemplo, el Bloc de Notas o vid para escribir el código del *MIDlet*.
- Un compilador estándar de Java. Haremos uso del SDK de J2SE que puede ser descargado desde la dirección http://java.sun.com/j2se/1.4.1/download.html
- Las APIs de la configuración CLDC y del perfil MIDP que pueden ser descargadas desde http://java.sun.com/j2me/download.html.

Lo primero que haremos será instalar el SDK de J2SE. Una vez descargado el archivo j2sdk-1_4_1_04-windows-i586.exe solo tendremos que hacer doble *click* sobre él e instalarlo en una carpeta en nuestro disco duro (p.e. c:\jdk1.4.1). Una vez instalado añadiremos a nuestro *path* la carpeta anterior. Para ello modificaremos la variable de entorno *path* a la que le añadiremos la carpeta c:\jdk1.4.1\bin y crearemos una nueva variable de entorno llamada JAVA_HOME con el valor c:\jdk1.4.1.

A continuación instalaremos las APIs de CLDC y de MIDP. Con descargarnos las APIs del perfil MIDP tenemos suficiente. Para ello solo tendremos que descomprimir el archivo .zip descargado anteriormente en el lugar que deseemos (p.e. c:\midp2.0fcs). Al igual que hicimos con anterioridad tendremos que añadir la dirección c:\midp2.0fcs\bin a la variable *path* y además tendremos que crear una nueva variable de entorno MIDP_HOME con el valor c:\midp2.0fcs.

Para comprobar que hemos realizado correctamente la instalación nos iremos a la línea de comandos y escribiremos lo siguiente:

```
java –version
```

Por la pantalla deberá aparecer algo similar a la Figura 2.1.

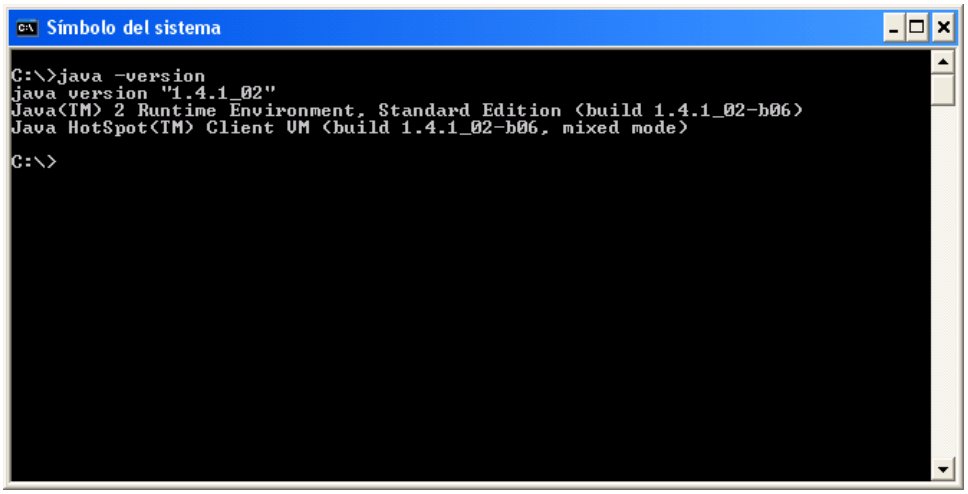

Figura 2.1 Comprobación de la instalación del JSDK.

A continuación escribiremos (ver Figura 2.2):

midp –versión

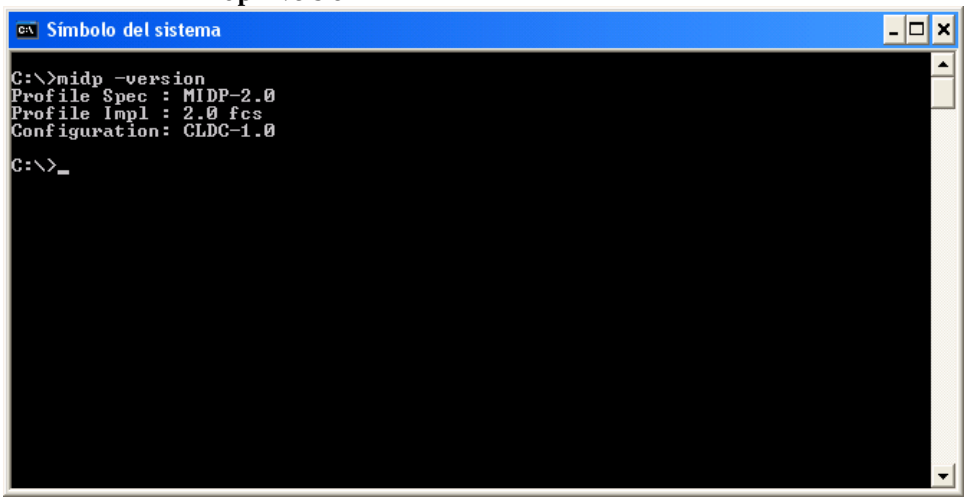

Figura 2.2 Comprobación de la instalación del perfil MIDP.

2.2.2 Fases de Desarrollo

El desarrollo y preparación de una aplicación puede dividirse en las siguientes cuatro fases:

1. Desarrollo del código: En esta fase como hemos dicho con anterioridad haremos uso de un editor de texto cualquiera. Una vez que terminemos de escribir el código que conformará nuestro *MIDlet* deberemos guardar el fichero con el mismo nombre de la clase principal y con la extensión .java.

2. Compilación: En este paso vamos a generar el archivo .class asociado a la clase .java creada en la fase anterior. Para realizar la compilación nos situaremos en la línea de comandos, y después de situarnos en el directorio donde se encuentra el fichero <fuente>.java escribiremos lo siguiente:

   javac –bootclasspath c:\midp2.0fcs\classes <fuente>.java

3. Preverificación: En este paso realizaremos la preverificación de clases. Para ello tendremos que movernos al directorio donde se encuentre la clase ya compilada y escribiremos lo siguiente:

   preverify –classpath c:\midp2.0fcs\classes <fuente>.java

   Por defecto, la preverificación nos generará un fichero .class en el directorio ./output/.

4. Empaquetamiento: En esta fase vamos a empaquetar nuestro *MIDlet* y dejarlo totalmente preparado para su descarga sobre el dispositivo MID. Para ello vamos a tener que construir dos archivos:
   - Un archivo JAR con los ficheros que forman el *MIDlet*.
   - Un archivo descriptor de la aplicación que es opcional.

   Normalmente el empaquetamiento involucra a varios *MIDlets* que conformarían lo que se denomina una *suite* de *MIDlets*. Para simplificar un poco, realizaremos el proceso completo de empaquetado para un solo *MIDlet*, aunque se explicará los pasos a seguir para formar esta *suite* de *MIDlets*.

   Un archivo JAR está formado por los siguientes elementos:
   - Un archivo manifiesto que describe el contenido del archivo JAR.
   - Las clases Java que forman el *MIDlet*
   - Los archivos de recursos usados por el *MIDlet*.

2.2.3 Creación del archivo manifiesto

El archivo de manifiesto es opcional y como hemos dicho, describe el contenido del archivo JAR. Este fichero contiene atributos de la forma atributo: valor. La creación de éste puede realizarse desde cualquier editor de texto y tiene la siguiente apariencia:

```
MIDlet-1: Prueba, prueba.png, Prueba
MIDlet-Name: Prueba
MIDlet-Vendor: Francisco García
MIDlet-Version: 1.0
Microedition-Configuration: CLDC-1.0
Microedition-Profile: MIDP-1.0
```

La Tabla 2.1 nos muestra los atributos que deben formar parte del archivo manifiesto.

Atributo	Descripción
MIDlet-Name	Nombre de la *MIDlet suite*.
MIDlet-Version	Versión de la *MIDlet suite*.
MIDlet-Vendor	Desarrollador del *MIDlet*.
MIDlet-n	Contiene una lista con el nombre de la *MIDlet* suite, icono y nombre del *MIDlet* en la *suite*.
Microedition-Configuration	Configuración necesitada para ejecutar el *MIDlet*.
Microedition-Profile	Perfil necesitado para ejecutar el *MIDlet*.

Tabla 2.1 Atributos requeridos para el archivo de manifiesto

Existen la Tabla 2.2 ilustra otros atributos opcionales que se pueden definir:

Atributo	Descripción
MIDlet-Description	Descripción de la *MIDlet suite*
MIDlet-Icon	Nombre del archivo png incluido en el JAR.
MIDlet-Info-URL	URL con información sobre el *MIDlet*.
MIDlet-Data-Size	Número de bytes requeridos por el *MIDlet*.

Tabla 2.2 Atributos opcionales del archivo de manifiesto.

En el caso de que se vaya a crear una *suite* de *MIDlets* con varios *MIDlets* habría que definir cada uno de ellos usando el siguiente atributo:

MIDlet-1: Prueba, prueba1.png, Prueba1
MIDlet-2: Prueba, prueba2.png, Prueba2
…

2.2.4 Creación del archivo JAR

Una vez creado este archivo solo nos queda crear el fichero JAR. Para ello nos iremos a la línea de comandos y escribiremos:

jar cmf <archivo manifiesto> <nombrearchivo>.jar -C <clases java> . –C <recursos>

2.2.5 Creación del archivo JAD

El AMS o Gestor de Aplicaciones del que hablaremos en el siguiente capítulo, puede usar este archivo para obtener información útil sobre el *MIDlet*. Este archivo al igual que el manifiesto define una serie de atributos. El archivo JAD es opcional, pero si lo creamos debe poseer los atributos de la Tabla 2.3.

Atributo	Descripción
MIDlet-Name	Nombre de la MIDlet suite.
MIDlet-Vendor	Nombre del desarrollador.
MIDlet-Version	Versión del MIDlet.
MIDlet-Configuration	Configuración necesitada para ejecutar el *MIDlet*.
MIDlet-Profile	Perfil necesitado para ejecutar el *MIDlet*.
MIDlet-Jar-URL	URL del archivo JAR de la MIDlet suite.

MIDlet-Jar-Size	Tamaño en bytes del archivo JAR.

Tabla 2.3 Atributos requeridos por el archivo JAD.

Opcionalmente también puede poseer los atributos de la Tabla 2.4.

Atributo	Descripción
MIDlet-Data-Size	Mínimo número de bytes de almacenamiento persistente usado por el *MIDlet*.
MIDlet-Delete-Confirm	Confirmación a la hora de eliminar el *MIDlet*.
MIDlet-Description	Descripción de la *MIDlet suite*.
MIDlet-Icon	Archivo .png incluido en el JAR.
MIDlet-Info-URL	URL con información de la *MIDlet* suite.
MIDlet-Install-Notify	Indica que el AMS notifique al usuario de la instalación del nuevo *MIDlet*.

Tabla 2.4 Atributos opcionales del archivo JAD.

Además de los anteriores, el desarrollador de la *MIDlet* suite puede definir atributos adicionales útiles para el *MIDlet* durante su ejecución. En el capítulo 5 veremos la utilidad que se le puede dar a estos atributos definidos por el desarrollador de la suite.

Las últimas fases que quedan por realizar son:

5 y 6. Ejecución y depuración: En esta fase tendremos que ejecutar el *MIDlet* sobre un emulador. Para ello haremos uso de la herramienta Wireless Toolkit 2.0. Su funcionamiento será explicado en el siguiente apartado.

Llegados a este punto ya conocemos el proceso de desarrollo de *MIDlets*. En el siguiente apartado simplemente veremos algunas herramientas que podemos usar y que nos facilitará bastante este desarrollo.

2.3 Desarrollo en entornos visuales

En el mercado existen varias herramientas que nos pueden ayudar a la hora de crear nuestros *MIDlets*. En este tutorial vamos a hacer uso de un par de ellas que explicaremos a continuación:

1. La primera de ellas es un entorno de desarrollo de Java con un emulador integrado, el *Sun One Studio Mobile Edition*. Este entorno es exactamente igual al *Sun One Studio*, pero incluye un emulador con el que podemos ver la ejecución de nuestros *MIDlets*, además de incluir las APIs propias de la configuración CLDC y el perfil MIDP (*Mobile Edition*).
2. La segunda herramienta es el *J2ME Wireless Toolkit* 2.0 que es simplemente un emulador al que le proporcionamos las clases java ya creadas y podemos ver el *MIDlet* en ejecución.

2.3.1 Instalación del *Sun One Studio Mobile Edition*

Sun nos proporciona una herramienta muy útil para la creación de *MIDlets*. Este software puede ser descargado desde el *site* oficial de Sun http://java.sun.com/.

Una vez descargado el fichero de instalación, tan solo tendremos que proceder con ésta. Para ello debemos tener instalado el SDK de J2SE en nuestro equipo ya que durante el proceso de instalación se nos preguntará por la ubicación de éste.

2.3.2 Instalación del *J2ME Wireless Toolkit 2.0*

La herramienta *J2ME Wireless Toolkit 2.0* puede ser descargada desde la dirección http://java.sun.com/products/j2mewtoolkit/download-2_0.html. La instalación de esta herramienta es bastante sencilla ya que solo hay que iniciar el proceso de instalación y seguir los pasos que te indica el instalador.

2.3.3 Desarrollo de aplicaciones en el *Sun One Studio Mobile Edition*

Una vez instalado el *Sun One Studio Mobile Edition*, nos aparecerá un entorno basado en ventanas donde podremos desarrollar y compilar nuestro *MIDlet* (Figura 2.3).

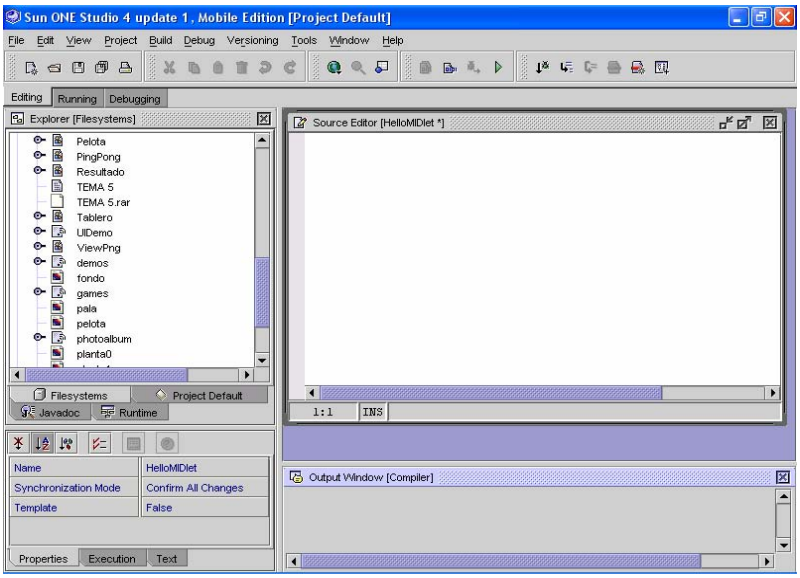

Figura 2.3 Aspecto del *Sun One Studio Mobile Edition*.

En esta herramienta es posible realizar todas las fases del desarrollo de aplicaciones MIDP.

- Disponemos de un editor de texto totalmente integrado donde crear el código fuente.

- Una vez creado el código del *MIDlet* es posible su compilación ya que el entorno dispone de todas las librerías necesarias para ello.
- El proceso de preverificación se realiza automáticamente después de la compilación.
- El entorno también nos da la posibilidad de empaquetar el *MIDlet* por separado o dentro de una *MIDlet* suite.
- Por último, las fases de ejecución y depuración también la podemos realizar con esta herramienta ya que nos permitirá ejecutar el *MIDlet* sobre un emulador, ya que trae integrada la herramienta *J2ME Wireless Toolkit 1.0.4*, que nosotros vamos a sustituir por la 2.0 con objeto de utilizar ciertas extensiones que esta última incorpora.

Como vemos, esta herramienta engloba todas las fases de desarrollo en un mismo entorno.

2.3.4 Desarrollo con el *J2ME Wireless Toolkit 2.0*.

Anteriormente hemos visto que el entorno *Sun One Studio Mobile Edition* tiene integrado el emulador *Wireless Toolkit 1.0.4*. Este emulador soporta la especificación MIDP 1.0. Nosotros, sin embargo vamos a trabajar sobre la especificación MIDP 2.0 con lo que debemos actualizar este emulador para que soporte los *MIDlets* que se vayan a crear en este tutorial.

Es posible trabajar independientemente con esta herramienta. A continuación vamos a ver como sería el proceso de desarrollo de un *MIDlet* usando tan solo el *J2ME Wireless Toolkit.*

El módulo principal de esta herramienta es la llamada **KToolBar**. A través de este módulo vamos a poder realizar distintos proyectos y ejecutarlos sobre un emulador. El proceso que se realizaría sería el siguiente:

- En primer lugar habría que crear un nuevo proyecto al que le daríamos un nombre en concreto. A la hora de crear este proyecto se nos da la oportunidad de definir ciertos atributos en el archivo JAD.

- Una vez creado el proyecto, el entorno nos crea un sistema de directorios dentro de la carpeta **apps**. Aquí nos crearía una carpeta con el nombre del proyecto y dentro de ella un conjunto de subdirectorios cuya estructura puede apreciarse en la Figura 2.4.

- En el subdirectorio **src** es donde se deben guardar los ficheros .java con el código fuente. En el directorio **res** guardaremos los recursos que utilice nuestro *MIDlet*. Si usamos alguna librería adicional, debemos guardarla en el subdirectorio **lib** (Figura 2.5).

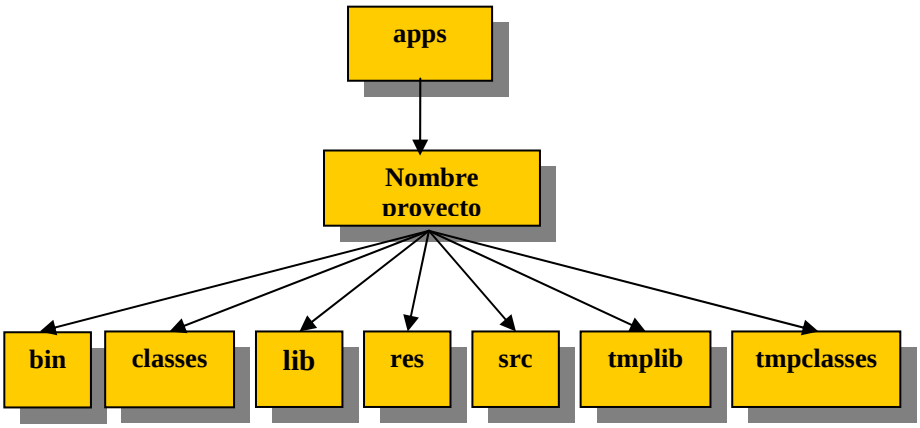

Figura 2.4 Jerarquía de directorios.

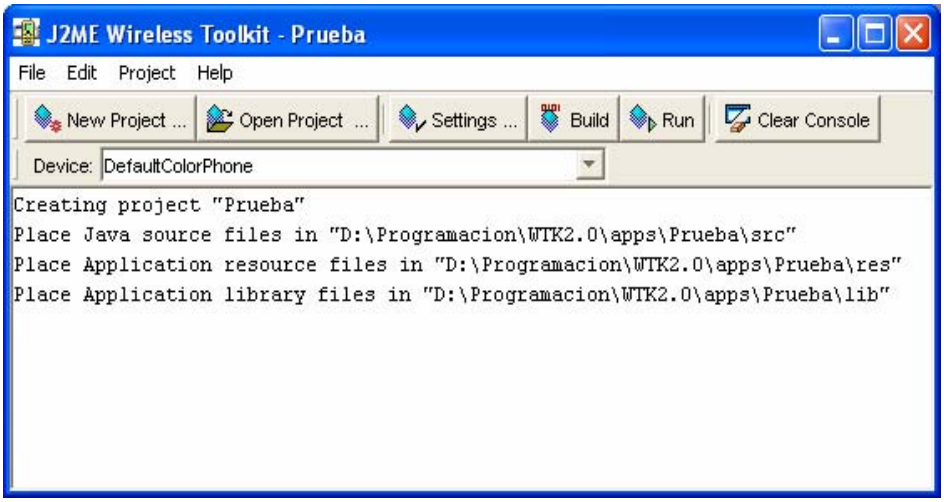

Figura 2.5 Aspecto de la KToolBar del J2ME Wireless Toolkit.

- Una vez situados los archivos en los subdirectorios correspondientes tan sólo tendremos que compilar nuestro proyecto y ejecutarlo sobre el emulador. Al igual que con el *Sun One Studio Mobile Edition*, el proceso de preverificación se realiza automáticamente después de la compilación.

- Una vez que hayamos hecho todas las pruebas necesarias con nuestro *MIDlet*, esta herramienta también nos da la posibilidad de empaquetar

nuestro *MIDlet* y dejarlo de esta manera, preparado para la descarga en un dispositivo real.

Es posible integrar el emulador *Wireless Toolkit* 2.0 dentro del entorno de desarrollo *Sun One Studio M.E*. Para ello tan sólo tendremos que seguir los siguientes pasos:

1. En primer lugar iniciaremos el *Sun One Studio M.E.* y dentro de la ventana del explorador, pincharemos en la pestaña *Runtime*.
2. A continuación escogeremos la secuencia de menús *Device Emulator Registry -> Installed Emulators* y con el botón derecho pulsaremos y seleccionaremos *Add Emulator*.

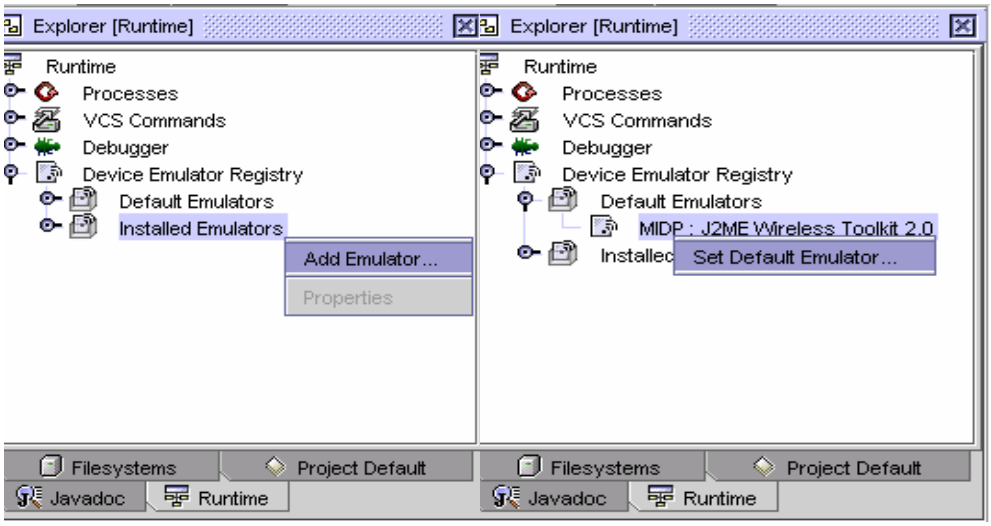

Figura 2.6 Integración del *J2ME Wireless Toolkit* en el *Sun One Studio ME*.

3. A continuación seleccionaremos la carpeta donde hayamos instalado el emulador *J2ME Wireless Toolkit 2.0.*
4. Ahora nos moveremos hasta *Default Emulators* y pincharemos con el botón derecho sobre el *Wireless Toolkit 2.0* y lo seleccionaremos por defecto (Figura 2.6).

2.4 Uso de otros emuladores

Uno de los principales objetivos de un programador de *MIDlets* es conseguir que las aplicaciones puedan ser soportadas por un amplio número de dispositivos MID. Para ello existen en el mercado un extenso número de emuladores donde se pueden hacer

pruebas con el *MIDlet* y así depurar el código. Estos emuladores normalmente se pueden descargar desde las páginas web de sus fabricantes.

Capítulo 3:

Los *MIDlets*

3.1. Descripción del capítulo

En este capítulo vamos a dar unas nociones básicas sobre los *MIDlets*. Aunque ya hemos hablado de ellos y sabemos qué son, en este capítulo vamos a profundizar más en su estudio. Veremos cuáles son sus propiedades, conoceremos su ciclo de vida y estados por los que pasa. Vamos a hablar también del gestor de aplicaciones y la relación de éste con los *MIDlets*. La última parte del capítulo está dedicada al estudio de las clases que contiene el paquete javax.microedition.midlet y estudiaremos el código de nuestra primera aplicación creada usando la especificación MIDP. Suponemos que el lector ha programado anteriormente en Java y dispone de unos conocimientos básicos sobre esta tecnología. Así que no nos vamos a parar a explicar conceptos básicos como herencia, excepciones, métodos de clase, etc.…

Los *MIDlets* son aplicaciones creadas usando la especificación MIDP. Están diseñados para ser ejecutados, como ya sabemos, en dispositivos con poca capacidad gráfica, de cómputo y de memoria. En estos dispositivos no disponemos de líneas de comandos donde poder ejecutar las aplicaciones que queramos, si no que reside en él un *software* que es el encargado de ejecutar los *MIDlets* y gestionar los recursos que éstos ocupan. En el siguiente apartado vamos a hablar mas profundamente de este *software* que no es otro que el gestor de aplicaciones.

3.2. El Gestor de Aplicaciones

El gestor de aplicaciones o AMS (*Application Management System*) es el *software* encargado de gestionar los *MIDlets*. Este *software* reside en el dispositivo y es el que nos permite ejecutar, pausar o destruir nuestras aplicaciones J2ME. A partir de ahora nos referiremos a él con las siglas de sus iniciales en inglés AMS. Ya hemos hablado anteriormente de él en el tema 1cuando explicábamos el proceso de descarga de *MIDlets*. El AMS realiza dos grandes funciones:

- Por un lado gestiona el ciclo de vida de los *MIDlets*.
- Por otro, es el encargado de controlar los estados por los que pasa el *MIDlet* mientras está en la memoria del dispositivo, es decir, en ejecución.

3.2.1. Ciclo de vida de un *MIDlet*

El ciclo de vida de un MIDlet pasa por 5 fases (ver Figura 3.1): descubrimiento, instalación, ejecución, actualización y borrado. En el tema 1 ya hablamos de ellas cuando explicábamos el proceso de descarga de *MIDlets* en un dispositivo.

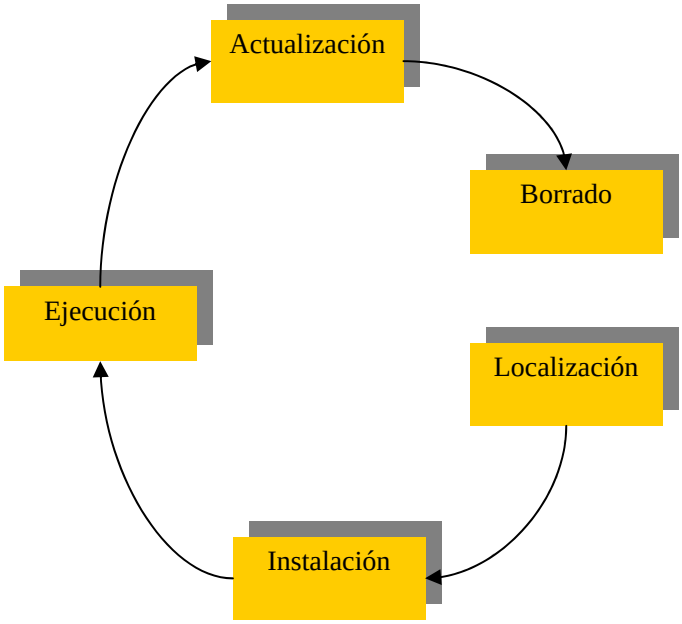

Figura 3.1 Ciclo de vida de un *MIDlet*.

El AMS es el encargado de gestionar cada una de estas fases de la siguiente manera:

1. Descubrimiento: Esta fase es la etapa previa a la instalación del MIDlet y es dónde seleccionamos a través del gestor de aplicaciones la aplicación a descargar. Por tanto, el gestor de aplicaciones nos tiene que proporcionar los mecanismos necesarios para realizar la elección del MIDlet a descargar. El AMS puede ser capaz de realizar la descarga de aplicaciones de diferentes maneras, dependiendo de las capacidades del dispositivo. Por ejemplo, esta descarga la podemos realizar mediante un cable conectado a un ordenador o mediante una conexión inalámbrica.
2. Instalación: Una vez descargado el MIDlet en el dispositivo, comienza el proceso de instalación. En esta fase el gestor de aplicaciones controla todo el proceso informando al usuario tanto de la evolución de la instalación como de si existiese algún problema durante ésta. Cuándo un MIDlet está

instalado en el dispositivo, todas sus clases, archivos y almacenamiento persistente están preparados y listos para su uso.

3. Ejecución: Mediante el gestor de aplicaciones vamos a ser capaces de iniciar la ejecución de los *MIDlets*. En esta fase, el AMS tiene la función de gestionar los estados del MIDlet en función de los eventos que se produzcan durante esta ejecución. Esto lo veremos un poco más en profundidad más adelante.

4. Actualización: El AMS tiene que ser capaz de detectar después de una descarga si el MIDlet descargado es una actualización de un MIDlet ya presente en el dispositivo. Si es así, nos tiene que informar de ello, además de darnos la oportunidad de decidir si queremos realizar la actualización pertinente o no.

5. Borrado: En esta fase el AMS es el encargado de borrar el MIDlet seleccionado del dispositivo. El AMS nos pedirá confirmación antes de proceder a su borrado y nos informará de cualquier circunstancia que se produzca.

Hay que indicar que el MIDlet puede permanecer en el dispositivo todo el tiempo que queramos. Después de la fase de instalación, el MIDlet queda almacenado en una zona de memoria persistente del dispositivo MID. El usuario de éste dispositivo es el encargado de decidir en qué momento quiere eliminar la aplicación y así se lo hará saber al AMS mediante alguna opción que éste nos suministre.

3.2.2. Estados de un *MIDlet* en fase de ejecución

Además de gestionar el ciclo de vida de los *MIDlets*, como ya hemos visto, el AMS es el encargado de controlar los estados del MIDlet durante su ejecución. Durante ésta el *MIDlet* es cargado en la memoria del dispositivo y es aquí donde puede transitar entre 3 estados diferentes: Activo, en pausa y destruido.

Cuándo un *MIDlet* comienza su ejecución, está en el estado “Activo” pero, ¿qué ocurre si durante su ejecución recibimos una llamada o un mensaje? El gestor de aplicaciones debe ser capaz de cambiar el estado de la aplicación en función de los eventos externos al ámbito de ejecución de la aplicación que se vayan produciendo. En este caso, el gestor de aplicaciones interrumpiría la ejecución del *MIDlet* sin que se viese afectada la ejecución de éste y lo pasaría al estado de “Pausa” para atender la llamada o leer el mensaje. Una vez que terminemos de trabajar con el *MIDlet* y salgamos de él, éste pasaría al estado de “Destruido” dónde sería eliminado de la memoria del dispositivo. Cuándo decimos que el *MIDlet* pasa al estado “Destruido” y es eliminado de memoria, nos referimos a la memoria volátil del dispositivo que es usada para la ejecución de aplicaciones. Una vez finalizada la ejecución del *MIDlet* podemos volver a invocarlo las veces que queramos ya que éste permanece en la zona de memoria persistente hasta el momento que deseemos desinstalarlo.

3.2.3. Estados de un *MIDlet*

Un *MIDlet* durante su ejecución pasa por 3 estados diferentes. Como ya hemos visto en el apartado anterior, estos tres estados son:

- Activo: El MIDlet está actualmente en ejecución.
- Pausa: El *MIDlet* no está actualmente en ejecución. En este estado el *MIDlet* no debe usar ningún recurso compartido. Para volver a pasar a ejecución tiene que cambiar su estado a Activo.
- Destruido: El *MIDlet* no está en ejecución ni puede transitar a otro estado. Además se liberan todos los recursos ocupados por el *MIDlet*.

La Figura 3.2 nos muestra el diagrama de estados de un *MIDlet* en ejecución:

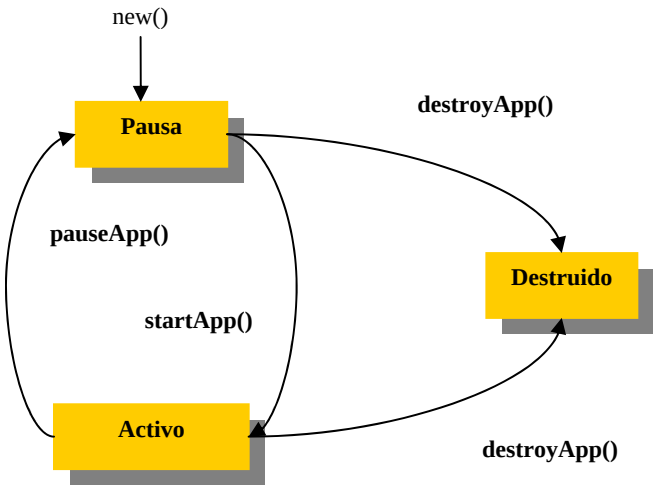

Figura 3.2 Estados de un *MIDlet*.

Como vemos en el diagrama, un *MIDlet* puede cambiar de estado mediante una llamada a los métodos MIDlet.startApp(), MIDlet.pauseApp() o MIDlet.destroyApp(). El gestor de aplicaciones cambia el estado de los *MIDlets* haciendo una llamada a cualquiera de los métodos anteriores. Un *MIDlet* también puede cambiar de estado por sí mismo.

Ahora vamos a ver por los estados que pasa un *MIDlet* durante una ejecución típica y cuáles son las acciones que realiza tanto el AMS como el *MIDlet*. En primer lugar, se realiza la llamada al constructor del MIDlet pasando éste al estado de "Pausa" durante un corto período de tiempo. El AMS por su parte crea una nueva instancia del *MIDlet*. Cuándo el dispositivo está preparado para ejecutar el MIDlet, el AMS invoca al

método MIDlet.startApp() para entrar en el estado de "Activo". El *MIDlet* entonces, ocupa todos los recursos que necesita para su ejecución. Durante este estado, el MIDlet puede pasar al estado de "Pausa" por una acción del usuario, o bien, por el AMS que reduciría en todo lo posible el uso de los recursos del dispositivo por parte del MIDlet. Tanto en el estado "Activo" como en el de "Pausa", el *MIDlet* puede pasar al estado "Destruido" realizando una llamada al método MIDlet.destroyApp(). Esto puede ocurrir porque el *MIDlet* haya finalizado su ejecución o porque una aplicación prioritaria necesite ser ejecutada en memoria en lugar del *MIDlet*. Una vez destruido el *MIDlet*, éste libera todos los recursos ocupados.

3.3. El paquete javax.microedition.midlet

El paquete javax.microedition.midlet define las aplicaciones MIDP y su comportamiento con respecto al entorno de ejecución. Como ya sabemos, una aplicación creada usando MIDP es un *MIDlet*. En la Tabla 3.1 podemos ver cuáles son las clases que están incluidas en este paquete:

Clases	Descripción
MIDlet	Aplicación MIDP.
*MIDlet*stateChangeException	Indica que el cambio de estado ha fallado.

Tabla 3.1 Clases del paquete javax.microedition.midlet

Veamos en profundidad cada una de estas clases con sus correspondientes métodos.

3.3.1. Clase MIDlet

```
public abstract class MIDlet
```

Un MIDlet es una aplicación realizada usando el perfil MIDP como ya sabemos. La aplicación debe extender a esta clase para que el AMS pueda gestionar sus estados y tener acceso a sus propiedades. El MIDlet puede por sí mismo realizar cambios de estado invocando a los métodos apropiados. Los métodos de los que dispone esta clase son los siguientes:

```
protected MIDlet()
```

Constructor de clase sin argumentos. Si la llamada a este constructor falla, se lanzaría la excepción SecurityException.

```
public final int checkPermission(String permiso)
```

Consigue el estado del permiso especificado. Este permiso está descrito en el atributo MIDlet-Permission del archivo JAD. En caso de no existir el permiso por el que se pregunta, el método devolverá un 0. En caso de no conocer el estado del permiso en ese momento debido a que sea necesaria alguna acción por parte del usuario, el método

devolverá un -1. Los valores devueltos por el método se corresponden con la siguiente descripción:

- 0 si el permiso es denegado
- 1 si el permiso es permitido
- -1 si el estado es desconocido

```
protected abstract void destroyApp(boolean incondicional) throws
MIDletstateChangeException
```

Indica la terminación del *MIDlet* y su paso al estado de "Destruido". En el estado de "Destruido" el *MIDlet* debe liberar todos los recursos y salvar cualquier dato en el almacenamiento persistente que deba ser guardado. Este método puede ser llamado desde los estados "Pausa" o "Activo".

Si el parámetro 'incondicional' es *false*, el MIDlet puede lanzar la excepción *MIDlets*tateChangeException para indicar que no puede ser destruido en este momento. Si es *true*, el *MIDlet* asume su estado de destruido independientemente de como finalice el método.

```
public final String getAppProperty(String key)
```

Este método proporciona al *MIDlet* un mecanismo que le permite recuperar el valor de las propiedades desde el AMS. Las propiedades se consiguen por medio de los archivos *manifest* y *JAD*. El nombre de la propiedad a recuperar debe ir indicado en el parámetro **key**. El método nos devuelve un String con el valor de la propiedad o *null* si no existe ningún valor asociado al parámetro **key**. Si **key** es **null** se lanzará la excepción NullPointerException.

```
public final void notifyDestroyed()
```

Este método es utilizado por un MIDlet para indicar al AMS que ha entrado en el estado de "Destruido". En este caso, todos los recursos ocupados por el MIDlet deben ser liberados por éste de la misma forma que si se hubiera llamado al método MIDlet.destroyApp(). El AMS considerará que todos los recursos que ocupaba el MIDlet están libres para su uso.

```
public final void notifyPaused()
```

Se notifica al AMS que el MIDlet no quiere estar "Activo" y que ha entrado en el estado de "Pausa". Este método sólo debe ser invocado cuándo el MIDlet esté en el estado "Activo". Una vez invocado este método, el MIDlet puede volver al estado "Activo" llamando al método MIDlet.startApp(), o ser destruido llamando al método MIDlet.destroyApp(). Si la aplicación es pausada por sí misma, es necesario llamar al método MIDlet.resumeRequest() para volver al estado "Activo".

```
protected abstract void pauseApp()
```

Indica al MIDlet que entre en el estado de "Pausa". Este método sólo debe ser llamado cuándo el MIDlet esté en estado "Activo". Si ocurre una excepción RuntimeException durante la llamada a MIDlet.pauseApp(), el MIDlet será destruido

inmediatamente. Se llamará a su método MIDlet.destroyApp() para liberar los recursos ocupados.

public final boolean platformRequest(String url)

Establece una conexión entre el MIDlet y la dirección URL. Dependiendo del contenido de la URL, nuestro dispositivo ejecutará una determinada aplicación que sea capaz de leer el contenido y dejar al usuario que interactúe con él.

Si, por ejemplo, la URL hace referencia a un archivo JAD o JAR, nuestro dispositivo entenderá que se desea instalar la aplicación asociada a este archivo y comenzará el proceso de descarga. Si, por el contrario, la URL tiene el formato tel:<número>, nuestro dispositivo entenderá que se desea realizar una llamada telefónica.

public final void resumeRequest()

Este método proporciona un mecanismo a los *MIDlets* mediante el cual pueden indicar al AMS su interés en pasar al estado de "Activo". El AMS, en consecuencia, es el encargado de determinar qué aplicaciones han de pasar a este estado llamando al método MIDlet.startApp().

protected abstract void startApp() **throws** *MIDlets*tateChangeException

Este método indica al MIDlet que ha entrado en el estado "Activo". Este método sólo puede ser invocado cuándo el MIDlet está en el estado de "Pausa". En el caso de que el MIDlet no pueda pasar al estado "Activo" en este momento pero si pueda hacerlo en un momento posterior, se lanzaría la excepción *MIDlets*tateChangeException.

A través de los métodos anteriores se establece una comunicación entre el AMS y el MIDlet. Por un lado tenemos que los métodos startApp(), pauseApp() y destroyApp() los utiliza el AMS para comunicarse con el MIDlet, mientras que los métodos resumeRequest(), notifyPaused() y notifyDestroyed() los utiliza el MIDlet para comunicarse con el AMS.

En la Tabla 3.2 podemos ver un resumen de los métodos que dispone la clase MIDlet.

3.3.2. Clase MIDletChangeStateException

public class *MIDlets*tateChangeException **extends** Exception

Esta excepción es lanzada cuando ocurre un fallo en el cambio de estado de un MIDlet.

Resumen	
Constructores	
protected	MIDlet()
Métodos	
int	checkPermission(String permiso)
protected abstract void	destroyApp(olean unconditional)
String	getAppProperty(String key)
void	notifyDestroyed()
void	notifyPaused()
protected abstract void	pauseApp()
boolean	platformRequest()
void	resumeRequest()
protected abstract void	startApp()

Tabla 3.2 Métodos de la clase MIDlet.

3.4. Estructura de los *MIDlets*

En este punto ya sabemos cuales son los estados de un *MIDlet*, conocemos su ciclo de vida y hemos estudiado todas sus clases y métodos. Ahora vamos a ver cuál es la estructura que comparten todos los *MIDlets* y posteriormente estudiaremos el código de nuestro primer *MIDlet*.

Hemos de decir que los *MIDlets*, al igual que los *applets* carecen de la función main(). Aunque existiese, el gestor de aplicaciones la ignoraría por completo. Un *MIDlet* tampoco puede realizar una llamada a System.exit(). Una llamada a este método lanzaría la excepción SecurityException.

Los *MIDlets* tienen la siguiente estructura:

```
import javax.microedition.midlet.*
public class MiMidlet extends MIDlet
        public MiMidlet() {
                /* Éste es el constructor de clase. Aquí debemos
                inicializar nuestras variables.
                */
        }
        public startApp(){
                /* Aquí incluiremos el código que queremos que el
                MIDlet ejecute cuándo se active.
                */
        }
        public pauseApp(){
                /* Aquí incluiremos el código que queremos que el
                MIDlet ejecute cuándo entre en el estado de pausa
                (Opcional)
                */
        }
```

```
                public destroyApp(){
                        /* Aquí incluiremos el código que queremos que el
                        MIDlet ejecute cuándo sea destruido. Normalmente
                        aquí se liberaran los recursos ocupados por el
                        MIDlet como memoria, etc. (Opcional)
                        */
                }
        }
```

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

public class HolaMundo extends MIDlet{

    private Display pantalla;
    private Form formulario = null;

    public HolaMundo(){
        pantalla = Display.getDisplay(this);
        formulario = new Form("Hola Mundo");
    }

    public void startApp(){
        pantalla.setCurrent(formulario);
    }

    public void pauseApp(){
    }

    public void destroyApp(boolean unconditional){
        pantalla = null;
        formulario = null;
        notifyDestroyed();
    }
}
```

Figura 3.3 Visualización del programa HolaMundo.java

Estos métodos son los que obligatoriamente tienen que poseer todos los *MIDlets* ya que, como hemos visto, la clase que hemos creado tiene que heredar de la clase MIDlet y ésta posee tres métodos abstractos: startApp(), pauseApp() y destroyApp() que han de ser implementados por cualquier *MIDlet*.

3.5. Ejemplo práctico

En este apartado vamos a crear nuestro primer *MIDlet*. Ya hemos visto cuál es la estructura que van a compartir todos ellos y ahora veremos cómo se refleja en la creación de nuestra primera aplicación. Vamos a centrarnos especialmente en la estructura de la aplicación y, no tanto en el código, ya que todavía no disponemos de los conocimientos necesarios para su total compresión. Con este ejemplo, lo que se intenta conseguir es una primera toma de contacto con los *MIDlets*, en los que de ahora en adelante vamos a centrar todo nuestro estudio.

Como primer ejemplo vamos a realizar la aplicación HolaMundo, cuyo código se muestra a continuación, siendo su salida la mostrada por la Figura 3.3.

En este ejemplo se ve claramente la estructura interna de un *MIDlet*. El constructor lo vamos a usar para crear los objetos que poseerá nuestra aplicación y crear la referencia a la pantalla del dispositivo. En el método startApp() pondremos activa la pantalla inicial de nuestro programa. El método pauseApp() está vacío ya que no necesitaremos su uso, aunque podríamos emitir si quisiéramos un mensaje al usuario informándole de que la aplicación está en estado de pausa. En el método destroyApp() hemos liberado recursos de memoria y hemos notificado al AMS que el *MIDlet* ha sido finalizado.

Capítulo 4:

La Configuración CLDC

4.1 Introducción

En este capítulo vamos a estudiar la configuración CLDC de J2ME. Vamos a ver con qué propósito se creó CLDC y cuales son sus objetivos y metas. En primer lugar veamos en qué aspectos se centra la configuración CLDC, y posteriormente estudiaremos con más detalle cada uno de ellos.

La configuración CLDC se ocupa de las siguientes áreas:

- Lenguaje Java y características de la máquina virtual.
- Librerías del núcleo de Java (java.lang.* y java.util.*).
- Entrada / Salida.
- Comunicaciones.
- Seguridad.
- Internacionalización.

Para estudiar el primer punto haremos una comparación donde veremos las principales características que se han tenido que omitir en la especificación de la configuración CLDC con respecto la plataforma J2SE. Todo lo concerniente con las librerías CLDC se verán en un apartado común que comprenderá desde las clases que se heredan de la plataforma J2SE hasta las clases propias de la configuración CLDC. Dentro de este punto veremos qué clases relacionadas con la E/S se han mantenido y qué novedades encontramos en el tema de las comunicaciones. En puntos separados veremos el modelo de seguridad que nos proporciona CLDC y dedicaremos un punto también a la internacionalización.

Sin embargo, dentro de este punto no vamos a ver aspectos como:

- Manejo del ciclo de vida de la aplicación.
- Interfaz de usuario.
- Manejo de eventos.
- Interacción entre el usuario y la aplicación.

Estas características no entran dentro del ámbito de la configuración CLDC, pero sí están incluidas en los perfiles e implementadas en una capa justo por encima del CLDC.

Lo que se intenta con la configuración CLDC es mantener lo más pequeño posible el número de áreas abarcadas. Es mejor restringir el alcance de CLDC para no

exceder las limitaciones de memoria o no excluir ningún dispositivo en particular. En el futuro, la configuración CLDC podría incluir otras áreas adicionales.

4.2 Objetivos y requerimientos

Una configuración de J2ME especifica un subconjunto de características soportadas por el lenguaje de programación Java, un subconjunto de funciones de la configuración para la máquina virtual de Java, el trabajo en red, seguridad, instalación y, posiblemente, otras APIs de programación, todo lo necesario para soportar un cierto tipo de productos.

CLDC es la base para uno o más perfiles. Un perfil de J2ME define un conjunto adicional de APIs y características para un mercado concreto, dispositivo determinado o industria. Las configuraciones y los perfiles están mas definidos exactamente en la publicación *Configurations and Profiles Architecture Specification, Java 2 Platform Micro Edition (J2ME), Sun Microsystems, Inc*.

4.2.1 Objetivos

El objetivo principal de la especificación CLDC es definir un estándar, para pequeños dispositivos conectados con recursos limitados con las siguientes características:

- 160 Kb a 512 Kb de memoria total disponible para la plataforma Java.
- Procesador de 16 bits o 32 bits.
- Bajo consumo, normalmente el dispositivo usa una batería.
- Conectividad a algún tipo de red, normalmente inalámbrica, con conexión intermitente y ancho de banda limitado (unos 9600 bps).

Teléfonos móviles, PDAs y terminales de venta, son algunos de los dispositivos que podrían ser soportados por esta especificación. Esta configuración J2ME define los componentes mínimos y librerías requeridas por dispositivos conectados pequeños.

Aunque el principal objetivo de la especificación CLDC es definir un estándar para dispositivos pequeños y conectados con recursos limitados, existen otros objetivos que son los siguientes:

- Extensibilidad: Uno de los grandes beneficios de la tecnología Java en los pequeños dispositivos es la distribución dinámica y de forma segura de contenido interactivo y aplicaciones sobre diferentes tipos de red. Hace unos años, estos dispositivos se fabricaban con unas características fuertemente definidas y sin capacidad apenas de extensibilidad. Los desarrolladores de dispositivos empezaron a buscar soluciones que permitieran construir dispositivos extensibles que soportaran una gran variedad de aplicaciones provenientes de terceras partes. Con la reciente introducción de teléfonos conectados a internet, comunicadores, etc. La transición está actualmente en

marcha. Uno de los principales objetivos de CLDC es tomar parte en esta transición permitiendo el uso del lenguaje de programación Java como base para la distribución de contenido dinámico para esta próxima generación de dispositivos.

- Desarrollo de aplicaciones por terceras partes: La especificación CLDC solo deberá incluir librerías de alto nivel que proporcionen suficiente capacidad de programación para desarrollar aplicaciones por terceras partes. Por esta razón, las APIs de red incluidas en CLDC deberían proporcionar al programador una abstracción de alto nivel como, por ejemplo, la capacidad de transferir archivos enteros, aplicaciones o páginas web, en vez de requerir que el programador conozca los detalles de los protocolos de transmisión para una red específica.

4.2.2 Requerimientos

Podemos clasificar los requerimientos en *hardware*, *software* y requerimientos basados en las características Java de la plataforma J2ME.

- Requerimientos hardware: CLDC está diseñado para ejecutarse en una gran variedad de dispositivos, desde aparatos de comunicación inalámbricos como teléfonos móviles, buscapersonas, hasta organizadores personales, terminales de venta, etc. Las capacidades del hardware de estos dispositivos varían considerablemente y por esta razón, los únicos requerimientos que impone la configuración CLDC son los de memoria. La configuración CLDC asume que la máquina virtual, librerías de configuración, librerías de perfil y la aplicación debe ocupar una memoria entre 160 Kb y 512 Kb. Más concretamente:
  - 128 Kb de memoria no volátil para la JVM y las librerías CLDC.
  - Al menos 32 Kb de memoria volátil para el entorno de ejecución Java y objetos en memoria.

  Este rango de memoria varía considerablemente dependiendo del dispositivo al que hagamos referencia.

- Requerimientos software: Al igual que las capacidades hardware, el software incluido en los dispositivos CLDC varía considerablemente. Por ejemplo, algunos dispositivos pueden poseer un sistema operativo (S.O.) completo que soporta múltiples procesos ejecutándose a la vez y con sistema de archivos jerárquico. Otros muchos dispositivos pueden poseer un software muy limitado sin noción alguna de lo que es un sistema de ficheros. Dentro de esta variedad, CLDC define unas mínimas características que deben poseer el software de los dispositivos CLDC. Generalmente, la configuración CLDC asume que el dispositivo contiene un mínimo Sistema Operativo encargado del manejo del hardware de éste. Este S.O. debe proporcionar al menos una entidad de planificación para ejecutar la JVM. El S.O. no necesita soportar espacios de memoria separados o procesos, ni debe garantizar la planificación de procesos en tiempo real o comportamiento latente.

- Requerimientos J2ME: CLDC es definida como la configuración de J2ME. Esto tiene importantes implicaciones para la especificación CLDC:
    - Una configuración J2ME solo deber definir un complemento mínimo de la tecnología Java. Todas las características incluidas en una configuración deben ser generalmente aplicables a una gran variedad de dispositivos. Características específicas para un dispositivo, mercado o industria deben ser definidas en un perfil en vez de en el CLDC. Esto significa que el alcance de CLDC está limitado y debe ser complementado generalmente por perfiles.
    - El objetivo de la configuración es garantizar portabilidad e interoperabilidad entre varios tipos de dispositivos con recursos limitados. Una configuración no debe definir ninguna característica opcional. Esta limitación tiene un impacto significativo en lo que se puede incluir en una configuración y lo que no. La funcionalidad más específica debe ser definida en perfiles.

4.3 Diferencias de CLDC con J2SE

El objetivo general para una JVM/CLDC es ser lo más compatible posible con la especificación del lenguaje Java, pero dentro de las restricciones de memoria que hemos visto en el apartado anterior. Las diferencias entre CLDC y la plataforma J2SE ya las hemos visto anteriormente en el primer capítulo. Estas diferencias se deben principalmente a dos aspectos: Diferencias entre el lenguaje Java y el subconjunto de éste que conforman las APIs de CLDC y diferencias entre la JVM y la JVM que usa CLDC. Ahora vamos a profundizar un poco en estas diferencias y a explicarlas con más detenimiento:

No existe soporte para operaciones en punto flotante

La principal diferencia entre los dos lenguajes es que la JVM que soporta CLDC no permite operaciones en punto flotante. Esta capacidad ha sido eliminada porque la mayoría de los dispositivos CLDC no poseen el hardware que les permita realizar estas operaciones, y el coste de realizar operaciones en punto flotante a través de software es muy alto.

No existe finalización de objetos

Las librerías CLDC no admiten la finalización de objetos ya que no incluyen el método Object.finalize().

Limitaciones en el manejo de errores

Una JVM/CLDC deberá soportar el manejo de excepciones. Sin embargo, el conjunto de las clases de error incluidas en las librerías CLDC ha sido limitado:

- Por un lado, CLDC sólo cuenta con un subconjunto de las excepciones disponibles para J2SE, por lo que el manejo de éstas queda limitado.

- La clase java.lang.Error y todas sus subclases han sido completamente eliminadas de las librerías CLDC.

Estas limitaciones se pueden explicar debido a que muchos de los sistemas que hacen uso de CLDC poseen su propio manejo interno de errores.

Estas diferencias que hemos visto son a nivel del lenguaje. Las diferencias que veremos a continuación se deben a la especificación de las máquinas virtuales. Un cierto número de características han sido eliminadas de la JVM/CLDC porque las librerías incluidas en CLDC son bastante más limitadas que las librerías incluidas en J2SE. Además la presencia de algunas de estas características podría provocar problemas de seguridad en los dispositivos al no disponer del modelo de seguridad de Java al completo. Las características eliminadas son:

Java Native Interface (JNI)

Una JVM/CLDC no implementa la interfaz nativo de Java (JNI). El soporte de JNI fue eliminado principalmente debido a dos razones:

- El modelo de seguridad limitado proporcionado por CLDC no soporta ni la invocación de métodos nativos ni la de APIs de otros lenguajes de programación.
- Debido a las restricciones de memoria de los dispositivos que soportan CLDC, la implementación completa de JNI se considera como muy costosa.

Cargadores de clase definidos por el usuario

Una JVM que soporta CLDC no permite cargadores de clase definidos por el usuario. Una JVM/CLDC posee un cargador de clase que no puede ser suprimido, sustituido o reconfigurado por el usuario. La eliminación de los cargadores de clase definidos por el usuario es parte de las restricciones de seguridad del modelo *sandbox*.

Reflexión

Una JVM/CLDC no posee características de reflexión, por ejemplo, características que permitan a una aplicación Java inspeccionar el número y contenido de clases, objetos, métodos, campos, pilas e hilos de ejecución y otras estructuras que se encuentren en el interior de la máquina virtual.

Grupos de *threads* o *daemon threads*

Una JVM/CLDC es capaz de implementar múltiples hilos o *multithreading*, pero no es capaz de soportar grupos de *threads* o *daemon threads*. Las operaciones con *threads*, desde su ejecución a su parada sólo pueden ser aplicadas a un *thread*. Cualquier programador que quiera implementar operaciones para grupos de *threads* deberá hacer uso de los objetos Collection para almacenar cada uno de ellos.

Referencias débiles

Una JVM/CLDC no soporta referencias débiles, entendiendo a éstas como referencias a objetos que pueden ser eliminadas por el recolector de basura si el sistema necesita memoria (véase el paquete java.lang.ref).

4.4 Seguridad en CLDC

Debido a las características de los dispositivos englobados bajo CLDC, en los que se hace necesaria la descarga de aplicaciones y la ejecución de éstas en dispositivos que almacenan información muy personal, se hace imprescindible hacer un gran hincapié en la seguridad. Hay que asegurar la integridad de los datos transmitidos y de las aplicaciones. Éste modelo de seguridad no es nuevo, ya que la ejecución de applets (programas Java que se ejecutan en un navegador web) se realiza en una zona de seguridad denominada *sandbox*. Los dispositivos englobados en CLDC se encuentran ante un modelo similar al de los applets.

Este modelo establece que sólo se pueden ejecutar algunas acciones que se consideran seguras. Existen, entonces, algunas funcionalidades críticas que están fuera del alcance de las aplicaciones. De esta forma, las aplicaciones ejecutadas en estos dispositivos deben cumplir unas condiciones previas:

- Los ficheros de clases Java deben ser verificados como aplicaciones Java válidas.
- Sólo se permite el uso de APIs autorizadas por CLDC.
- No está permitido cargar clases definidas por el usuario.
- Sólo se puede acceder a características nativas que entren dentro del CLDC.
- Una aplicación ejecutada bajo KVM no debe ser capaz de dañar el dispositivo dónde se encuentra. De esto se encarga el verificador de clases que se asegura que no haya referencias a posiciones no válidas de memoria. También comprueba que las clases cargadas no se ejecuten de una manera no permitida por las especificaciones de la Máquina Virtual.

4.5 Librerías CLDC

Las versiones de Java J2EE y J2SE proporcionan un gran conjunto de librerías para el desarrollo de aplicaciones empresariales en servidores y ordenadores de sobremesa respectivamente. Desafortunadamente, estas librerías requieren varios megabytes de memoria para ser ejecutadas y es impracticable el almacenamiento de todas estas librerías en pequeños dispositivos con recursos limitados.

4.5.1 Objetivos generales

Un objetivo general para el diseño de librerías Java para CLDC es proporcionar un conjunto mínimo de librerías útiles para el desarrollo de aplicaciones y definición de perfiles para una variedad de pequeños dispositivos. Dadas las estrictas restricciones de

memoria y diferencias en las características de estos dispositivos, es completamente imposible ofrecer un conjunto de librerías que satisfagan a todo el mundo.

4.5.2 Compatibilidad

La mayoría de las librerías incluidas en CLDC son un subconjunto de las incluidas en las ediciones J2SE y J2EE de Java. Esto es así para asegurar la compatibilidad y portabilidad de aplicaciones. El mantenimiento de la compatibilidad es un objetivo muy deseable. Las librerías incluidas en J2SE y J2EE tienen fuertes dependencias internas que hacen que la construcción de un subconjunto de ellas sea muy difícil en áreas como la seguridad, E/S, interfaz de usuario, trabajo en red y almacenamiento de datos. Desafortunadamente, estas dependencias de las que hemos hablado hacen muy difícil tomar partes de una librería sin incluir otras. Por esta razón, se han rediseñado algunas librerías, especialmente en las áreas de trabajo en red y Entrada/Salida.

Las librerías CLDC pueden ser divididas en dos categorías:

- Clases que son un subconjunto de las librerías de J2SE.
- Clases específicas de CLDC.

4.5.3 Clases heredadas de J2SE

CLDC proporciona un conjunto de clases heredadas de la plataforma J2SE. En total, usa unas 37 clases provenientes de los paquetes java.lang, java.util y java.io. Cada una de estas clases debe ser idéntica o ser un subconjunto de la correspondiente clase de J2SE. Tanto los métodos como la semántica de cada clase deben permanecer invariables. De la Tabla 4.1 a la Tabla 4.4 podemos ver cada una de estas clases agrupadas según su funcionalidad, o sea, por paquetes.

Clases de sistema (Subconjunto de java.lang)
java.lang.Class
java.lang.Object
java.lang.Runnable
java.lang.Runtime
java.lang.String
java.lang.Stringbuffer
java.lang.System
java.lang.Thread
java.lang.Throwable

Tabla 4.1 Clases de sistema heredadas de J2SE

Clases de Datos (Subconjunto de java.lang)
java.lang.Boolean
java.lang.Byte
java.lang.Character
java.lang.Integer
java.lang.Long
java.lang.Short

Tabla 4.2 Clases de datos heredadas de J2SE

Clases de Utilidades (Subconjunto de java.util)
java.util.Calendar
java.util.Date
java.util.Enumeration
java.util.Hashtable
java.util.Random
java.util.Stack
java.util.TimeZone
java.util.Vector

Tabla 4.3 Clases de utilidades heredadas de J2SE

Clases de E/S (Subconjunto de java.io)
java.io.ByteArrayInputStream
java.io.ByteArrayOutputStream
java.io.DataInput
java.io.DataOutput
java.io.DataInputStream
java.io.DataOutputStream
java.io.InputStream
java.io.InputStreamReader
java.io.OutputStream
java.io.OutputStreamWriter
java.io.PrintStream
java.io.Reader
java.io.Writer

Tabla 4.4 Clases de E/S heredadas de J2SE

4.5.4 Clases propias de CLDC

La plataforma J2SE contiene a los paquetes java.io y java.net encargados de las operaciones de E/S. Debido a las limitaciones de memoria de CLDC no es posible

incluir dentro de él a todas las clases de estos paquetes. Ya hemos visto que CLDC hereda algunas clases del paquete java.io, pero no hereda ninguna clase relacionada con la E/S de ficheros, por ejemplo. Esto es debido a la gran variedad de dispositivos que abarca CLDC, ya que, para éstos puede resultar innecesario manejar ficheros. No se han incluido tampoco las clases del paquete java.net, basado en comunicaciones TCP/IP ya que los dispositivos CLDC no tienen por qué basarse en este protocolo de comunicación. Para suplir estas "carencias" CLDC posee un conjunto de clases más genérico para la E/S y la conexión en red que recibe el nombre de "*Generic Connection Framework*". Estas clases están incluidas en el paquete javax.microedition.io y son las que aparecen en la Tabla 4.5.

Clase	Descripción
Connector	Clase genérica que puede crear cualquier tipo de conexión.
Connection	Interfaz que define el tipo de conexión más genérica.
InputConnection	Interfaz que define una conexión de *streams* de entrada.
OutputConnection	Interfaz que define una conexión de *streams* de salida.
StreamConnection	Interfaz que define una conexión basada en *streams.*
ContentConnection	Extensión a StreamConnection para trabajar con datos.
Datagram	Interfaz genérico de datagramas.
DatagramConnection	Interfaz que define una conexión basada en datagramas.
StreamConnectionNotifier	Interfaz que notifica una conexión. Permite crear una conexión en el lado del servidor.

Tabla 4.5 Clases e interfaces incluidos en el paquete javax.microedition.io

Estas clases e interfaces las veremos con más detenimiento en el capítulo 7 que estará dedicado en su totalidad a las comunicaciones. Aquí sólo las hemos descrito ya que pertenecen a la configuración CLDC, pero aún nos faltan algunos conocimientos para llegar a manejarlas. En el capítulo 7 veremos en profundidad cada una de ellas y estudiaremos los métodos que nos proporcionan.

Capítulo 5:
Interfaces gráficas de usuario

5.1. Descripción del capítulo

En este punto vamos a entrar de lleno en la programación de MIDlets usando el perfil MIDP. Este perfil es el encargado de definir, como ya sabemos, peculiaridades de un tipo de dispositivos, en nuestro caso, teléfonos móviles. Este perfil se ocupa de definir aspectos como interfaces de usuario, sonidos, almacenamiento persistente, etc. Estos aspectos los veremos en profundidad en los siguientes capítulos. En este concretamente, nos vamos a centrar en el paquete javax.microedition.lcdui (*Interfaz de usuario con pantalla LCD*), que es donde se encuentran los elementos gráficos que vamos a utilizar en nuestras aplicaciones.

En primer lugar realizaremos una introducción a las interfaces de usuario en los dispositivos MIDP. En este punto realizaremos una división entre interfaces de usuario de alto nivel e interfaces de usuario de bajo nivel. Esta división continuará hasta el final del capítulo ya que veremos por separado cada una de estas interfaces y los elementos que están presentes en cada una de ellas.

Conforme avancemos en el capítulo, se irán describiendo las clases que componen el paquete javax.microedition.lcdui y sus correspondientes métodos. Hay que recordar que el objetivo de este proyecto no es ser una guía de referencia, por lo que no vamos a explicar exhaustivamente cada clase y método. En cambio, realizaremos un gran número de MIDlets con los que tendremos la oportunidad de ver como funciona cada una de estas clases. Al final del capítulo, seremos capaces de crear interfaces de usuario fáciles de usar, controlar eventos y comunicarnos con el usuario a través de un entorno amigable.

5.2. Introducción a las interfaces de usuario

Teniendo en cuenta la diversidad de aplicaciones que podemos realizar para los dispositivos MID (llamaremos así a los dispositivos que soportan MIDP), y los elementos que nos proporcionan la configuración CLDC y el perfil MIDP, vamos a dividir a estos elementos en dos grupos:

- Por un lado estudiaremos los elementos que componen la interfaz de usuario de alto nivel. Esta interfaz usa componentes tales como botones, cajas de texto, formularios, etc. Estos elementos son implementados por cada dispositivo y la finalidad de usar las APIs de alto nivel es su portabilidad. Al

usar estos elementos, perdemos el control del aspecto de nuestra aplicación ya que la estética de estos componentes depende exclusivamente del dispositivo donde se ejecute. En cambio, usando estas APIs de alto nivel ganaremos un alto grado de portabilidad de la misma aplicación entre distintos dispositivos. Fundamentalmente, se usan estas APIs cuando queremos construir aplicaciones de negocios.

- Por otro lado tenemos las interfaces de usuario de bajo nivel. Al crear una aplicación usando las APIs de bajo nivel, tendremos un control total de lo que aparecerá por pantalla. Estas APIs nos darán un control completo sobre los recursos del dispositivo y podremos controlar eventos de bajo nivel como, por ejemplo, el rastreo de pulsaciones de teclas. Generalmente, estas APIs se utilizan para la creación de juegos donde el control sobre lo que aparece por pantalla y las acciones del usuario juegan un papel fundamental.

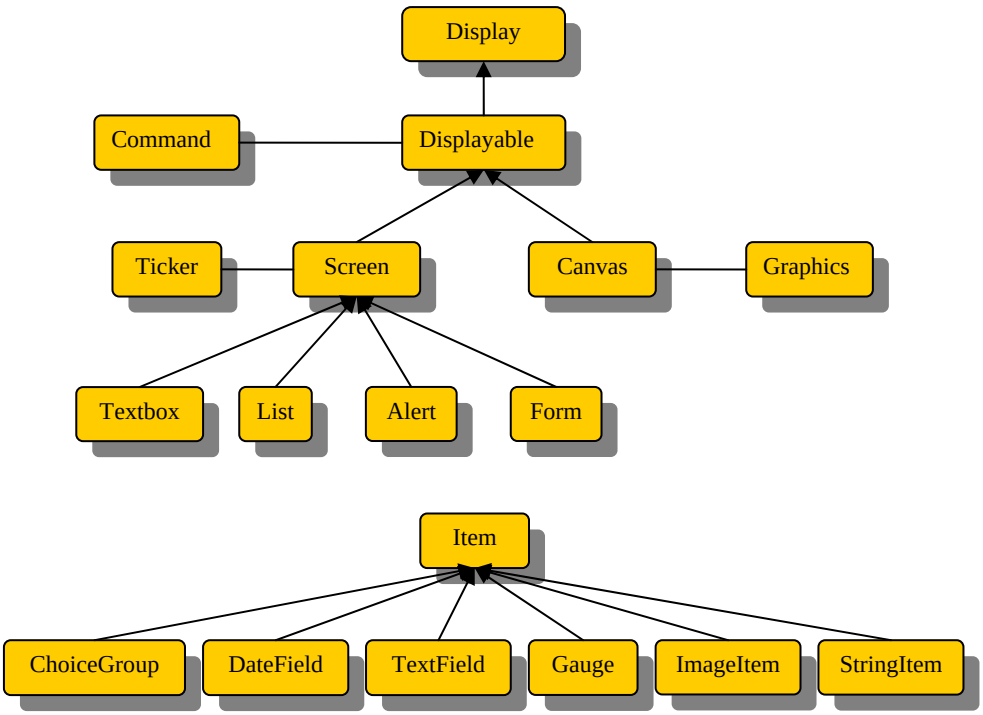

Figura 5.1 Jerarquía de clases derivadas de Display e Item

Una vez hecha la anterior clasificación, veremos cada una de estos grupos con mayor detalle estudiando los elementos que los integran. En el caso de las interfaces de alto nivel, crearemos una aplicación con la que desde nuestro teléfono móvil vamos a poder enviar órdenes a un ordenador que controlará los electrodomésticos de una casa. De esta manera, desde nuestro móvil podremos, por ejemplo, ver el estado de nuestra alarma, encender la calefacción, programar el vídeo o incluso calentar la cena que antes

de salir de casa habíamos dejado en el microondas para que cuando lleguemos esté preparada.

Veremos también los mecanismos que nos proporcionan las APIs de bajo nivel para la creación de juegos. Antes de adentrarnos en cada uno de las interfaces es necesario estudiar algunos conceptos generales que son comunes a ambas.

El paquete javax.microedition.lcdui incluye las clases necesarias para crear interfaces de usuario, tanto de alto nivel como de bajo nivel. En la Figura 5.1 podemos ver la organización de estas clases, que son las que estudiaremos en este capítulo.

Vamos a empezar nuestro estudio de estas clases por la clase Display, y así iremos descendiendo en la jerarquía de clases hasta estudiar cada una en concreto.

5.2.1. La clase Display

public class Display

La clase Display representa el manejador de la pantalla y los dispositivos de entrada. Todo *MIDlet* debe poseer por lo menos un objeto Display. En este objeto Display podemos incluir tantos objetos Displayable como queramos. La clase Display puede obtener información sobre las características de la pantalla del dispositivo donde se ejecute el *MIDlet*, además de ser capaz de mostrar los objetos que componen nuestras interfaces.

En la Tabla 5.1 podemos ver los métodos incluidos en esta clase

Métodos	Descripción
void callSerially(Runnable r)	Retrasa la ejecución del método run() del objeto *r* para no interferir con los eventos de usuario.
boolean flashBacklight(int duracion)	Provoca un efecto de *flash* en la pantalla.
int getBestImageHeight(int imagen)	Devuelve el mejor alto de imagen para un tipo dado.
int getBestImageWidth(int imagen)	Devuelve el mejor ancho de imagen para un tipo dado.
int getBorderStyle(bolean luminosidad)	Devuelve el estilo de borde actual.
int getColor(int color)	Devuelve un color basado en el parámetro pasado.
Displayable getCurrent()	Devuelve la pantalla actual.
static Display getDisplay(MIDlet m)	Devuelve una referencia a la pantalla del MIDlet *m.*
boolean isColor()	Devuelve *true* o *false* si la pantalla es de color o b/n.
int numAlphaLevels()	Devuelve el número de niveles *alpha* soportados.
int numColors()	Devuelve el número de colores aceptados por el MID.
void setCurrent(Alert a, Displayable d)	Establece la pantalla *d* despues de la alerta *a*
void setCurrent(Displayable d)	Establece la pantalla actual
void setCurrent(Item item)	Establece la pantalla en la zona dónde se encuentre el *item*
boolean vibrate(int duracion)	Realiza la operación de vibración del dispositivo.

Tabla 5.1 Métodos de la clase Display

Como hemos dicho al principio de este punto, todo *MIDlet* debe poseer al menos una instancia del objeto Display. Para obtenerla emplearemos el siguiente código:

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

La llamada a este método la realizaremos dentro del constructor del *MIDlet*. De esta forma nos aseguramos que el objeto Display esté a nuestra disposición durante toda la ejecución de éste. Además, dentro del método startApp tendremos que hacer referencia a la pantalla que queramos que esté activa haciendo uso del método setCurrent(). Hay que tener en cuenta que cada vez que salimos del método pauseApp, entramos en el método startApp, por lo que la construcción de las pantallas y demás elementos que formarán parte de nuestro *MIDlet* la tendremos que hacer en el método constructor.

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

public class MiMIDlet extends MIDlet{
    Display pantalla;
    public MiMIDlet{
     pantalla = Display.getDisplay(this);
     // Construir las pantallas que vayamos a utilizar en el MIDlet,
     // es decir, crear los objetos Displayable.
    }
    public startApp{
     if (pantalla == null)
              pantalla.setCurrent(Displayable d);
     // d tiene que ser un objeto que derive de la clase Displayable:
     // Form, Textbox, …
     ...
    }
    public pauseApp{
     ...
    }
    public destroyApp{
     ...
    }
}
```

En el caso de que se entre por primera vez en el método startApp, el valor de *pantalla* después de la llamada al método getDisplay() será *null*. Si no es así, es porque volvemos del estado de pausa por lo que debemos de dejar la pantalla tal como está.

5.2.2. La clase Displayable

```
public abstract class Displayable
```

La clase Displayable representa a las pantallas de nuestra aplicación. Como hemos dicho, cada objeto Display puede tener tantos objetos Displayable como

quiera. Como veremos más adelante, nuestras aplicaciones estarán formadas por varias pantallas que crearemos dentro del método constructor. Mediante los métodos getCurrent y setCurrent controlamos qué pantalla queremos que sea visible y accesible en cada momento.

La clase abstracta Displayable incluye los métodos encargados de manejar los eventos de pantalla y añadir o eliminar comandos. Estos métodos aparecen en la Tabla 5.2.

Métodos	Descripción
void addComand(Command cmd)	Añade el Command *cmd.*
int getHeight()	Devuelve el alto de la pantalla.
Ticker getTicker()	Devuelve el T*icker* (cadena de texto que se desplaza) asignado a la pantalla.
String getTitle()	Devuelve el título de la pantalla.
int getWidth()	Devuelve el ancho de la pantalla.
bolean isShown()	Devuelve *true* si la pantalla está activa.
void removeCommand(Command cmd)	Elimina el Command *cmd*.
void setCommandListener(CommandListener l)	Establece un *listener* para la captura de eventos.
void setTicker(Ticker ticker)	Establece un *Ticker* a la pantalla.
void setTitle(String s)	Establece un título a la pantalla.
protected void sizeChanged(int w, int h)	El AMS llama a este método cuándo el área disponible para el objeto Displayable es modificada.

Tabla 5.2 Métodos de la clase Displayable

5.2.3. Las clases Command y CommandListener

public class Command

Un objeto de la clase Command mantiene información sobre un evento. Podemos pensar en él como un botón de Windows, por establecer una analogía. Generalmente, los implementaremos en nuestros *MIDlets* cuando queramos detectar y ejecutar una acción simple.

Existen tres parámetros que hay que definir cuando construimos un objeto Command:

- Etiqueta: La etiqueta es la cadena de texto que aparecerá en la pantalla del dispositivo que identificará a nuestro Command.
- Tipo: Indica el tipo de objeto Command que queremos crear. Los tipos que podemos asignarle aparecen en la Tabla 5.3.

Tipo	Descripción
BACK	Petición para volver a la pantalla anterior
CANCEL	Petición para cancelar la acción en curso
EXIT	Petición para salir de la aplicación
HELP	Petición para mostrar información de ayuda

ITEM	Petición para introducir el comando en un "item" en la pantalla
OK	Aceptación de una acción por parte del usuario
SCREEN	Para Commands de propósito más general
STOP	Petición para parar una operación

Tabla 5.3 Tipos de los objetos Command

La declaración del tipo sirve para que el dispositivo identifique el Command y le dé una apariencia específica acorde con el resto de aplicaciones existentes en el dispositivo.

- Prioridad: Es posible asignar una prioridad específica a un objeto Command. Esto puede servirle al AMS para establecer un orden de aparición de los Command en pantalla. A mayor número, menor prioridad.

Por ejemplo, si queremos crear un objeto Command con la etiqueta "Atras", de tipo BACK y prioridad 1 lo haremos de la siguiente manera:

```
new Command("Atras",Command.BACK,1)
```

La Tabla 5.4 muestra los métodos de la clase Command.

Métodos	Descripción
public int getCommandType()	Devuelve el tipo del Command.
public String getLabel()	Devuelva la etiqueta del Command.
public String getLongLabel()	Devuelve la etiqueta larga del Command.
public int getPriority()	Devuelve la prioridad del Command.

Tabla 5.4 Métodos de la clase Command

No basta sólo con crear un objeto Command de un determinado tipo para que realice la acción que nosotros deseamos, de acuerdo a su tipo. Para ello tenemos que implementar la interfaz CommandListener.

En cualquier *MIDlet* que incluyamos Commands, tendremos además que implementar la interfaz CommandListener. Como sabemos, una interfaz es una clase donde todos sus métodos son declarados como abstract. Es misión nuestra implementar sus correspondientes métodos. En este caso, la interfaz CommandListener sólo incluye un método commandAction(Command c, Displayable d) en donde indicaremos la acción que queremos que se realice cuando se produzca un evento en el Command *c* que se encuentra en el objeto Displayable *d*.

En el siguiente punto veremos más elementos que nos ayudarán a crear nuestras interfaces de usuario y estudiaremos ejemplos donde podremos ver el uso de los Commands y de CommandListener.

5.3. La interfaz de usuario de alto nivel

Ya tenemos un concepto general de cómo manejar y controlar las acciones del usuario a través de objetos Command y de cómo insertar éstos en la pantalla del dispositivo MID. En este punto vamos a profundizar un poco más en la jerarquía de clases de la figura 5.1 y vamos a estudiar la clase Screen y todas las subclases

derivadas de ella. Todas estas clases conforman la interfaz de usuario de alto nivel. Por tanto, vamos a estudiar las APIs de alto nivel y realizaremos al final un ejemplo recopilatorio donde usaremos todos los elementos que veamos en este punto.

Como decíamos, la clase Screen es la superclase de todas las clases que conforman la interfaz de usuario de alto nivel:

```
public abstract class Screen extends Displayable
```

En la especificación MIDP 1.0 esta clase contenía cuatro métodos que le permitían definir y obtener el título y el *ticker*: setTitle(String s), getTitle(), setTicker(Ticket ticker) y getTicker(). El *ticker* es una cadena de texto que se desplaza por la pantalla de derecha a izquierda. En la especificación MIDP 2.0 que es la más reciente en este momento, estos cuatro métodos han sido incluidos en la clase Displayable.

5.3.1. La clase Alert

```
public class Alert extends Screen
```

El objeto Alert representa una pantalla de aviso. Normalmente se usa cuando queremos avisar al usuario de una situación especial como, por ejemplo, un error. Un Alert está formado por un título, texto e imágenes si queremos. Vamos a ver como crear una pantalla de alerta. Para ello contamos con dos constructores:

```
Alert(String titulo)
Alert(String titulo, String textoalerta, Image imagen, AlertType tipo)
```

Además podemos definir el tiempo que queremos que el aviso permanezca en pantalla, diferenciando de esta manera dos tipos de Alert:

1. Modal: La pantalla de aviso permanece un tiempo indeterminado hasta que es cancelada por el usuario. Esto lo conseguimos invocando al método Alert.setTimeOut(Alert.FOREVER).
2. No Modal: La pantalla de aviso permanece un tiempo definido por nosotros. Para ello indicaremos el tiempo en el método setTimeOut(tiempo). Una vez finalizado el tiempo, la pantalla de aviso se eliminará de pantalla y aparecerá el objeto Displayable que nosotros definamos.

Podemos elegir el tipo de alerta que vamos a mostrar. Cada tipo de alerta tiene asociado un sonido. Los tipos que podemos definir aparecen en la Tabla 5.5.

Tipo	Descripción
ALARM	Aviso de una petición previa
CONFIRMATION	Indica la aceptación de una acción
ERROR	Indica que ha ocurrido un error
INFO	Indica algún tipo de información
WARNING	Indica que puede ocurrir algún probl

Tabla 5.5 Tipos de alerta

Es posible ejecutar el sonido sin tener que crear un objeto Alert, invocando al método playSound(Display) de la clase AlertType, por ejemplo:

```
AlertType.CONFIRMATION.playSound(display)
```

En el siguiente ejemplo vamos a ver los dos tipos de alerta: modal y no modal. Vamos a crear un objeto Form donde insertaremos dos comandos. Cada uno de ellos activará una pantalla de alerta. El código de nuestro ejemplo aparece e acontinuación, y los resultados visuales en la Figura 5.2.

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

public class EjemploAlerta extends MIDlet implements CommandListener{
   Alert alerta1,alerta2;
   Command salir, aler1, aler2;
   Displayable temp;
   Display pantalla;
   Form pantallainicial;
   public EjemploAlerta(){
      // Obtengo la referencia a la pantalla del MIDlet
      pantalla = Display.getDisplay(this);
      // Creo los objetos que forman las pantallas del MIDlet
      salir = new Command("Salir",Command.EXIT,1);
      aler1 = new Command("Alerta Modal",Command.SCREEN,1);
      aler2 = new Command("Alerta No Modal",Command.SCREEN,1);
      // Creo la pantalla de alerta 1
      alerta1 = new Alert("Alerta Modal","Esta alerta desaparecerá"+
            "cuando pulses el botón de aceptar", null, AlertType.INFO);
      // Creo la pantalla de alerta 2
      alerta2 = new Alert("Alerta No Modal","Esta alerta desaparecera"+
            " cuando pasen 5 segundos",null,AlertType.INFO);
      alerta1.setTimeout(Alert.FOREVER);
      alerta2.setTimeout(5000);
      // Creo la pantalla principal del MIDlet
      pantallainicial = new Form("Programa principal");
      // Inserto objetos en la pantalla
      pantallainicial.addCommand(salir);
      pantallainicial.addCommand(aler1);
      pantallainicial.addCommand(aler2);
      pantallainicial.setCommandListener(this);
   }
   public void startApp() {
      pantalla.setCurrent(pantallainicial);
   }

   public void pauseApp() {
   }

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

```
    public void commandAction(Command c, Displayable d){
        if (c == salir){
            destroyApp(false);
            notifyDestroyed();
        }
        else if (c == aler1){
            pantalla.setCurrent(alerta1,pantallainicial);    //Método sobrecargado de
        }                       // la clase Display. Primero muestra un objeto del
        else{                   //tipo Alert y a continuación un Displayable.
            Pantalla.setCurrent(alerta2,pantallainicial);
        }
       }
}
```

Figura 5.2 Ejemplo de uso de alerta modal y no modal

5.3.2. La clase List

public class List **extends** Screen **implements** Choice

La clase List nos va a permitir construir pantallas que poseen una lista de opciones. Esto nos será muy útil para crear menús de manera independiente. La clase List implementa la interfaz Choice y esto nos va a dar la posibilidad de crear 3 tipos distintos de listas cuyo tipo están definidos en esta interfaz (Tabla 5.6).

Existen dos constructores que nos permiten construir listas: el primero de ellos nos crea una lista vacía y el segundo nos proporciona una lista con un conjunto inicial de opciones y de imágenes asociadas si queremos:

List(String titulo, int listType)
List(String titulo, int listType, String[] elementos, Image[] imagenes)

Tipo	Descripción
EXCLUSIVE	Lista en la que un sólo elemento puede ser seleccionado a la vez.
IMPLICIT	Lista en la que la selección de un elemento provoca un evento.
MULTIPLE	Lista en la que cualquier número de elementos pueden ser seleccionados al mismo tiempo.

Tabla 5.6 Tipos de Listas

Hay que tener en cuenta que el *array* de elementos y de imágenes tiene que tener el mismo número de componentes.

Veamos los distintos tipos de listas y como se produce la selección de opciones en cada una de ellas (la Figura 5.3 muestra las diferencias visuales):

1. Listas implícitas: Como hemos dicho, en este tipo de listas cuando seleccionamos un elemento provocamos una acción. Éste mecanismo nos va a ser muy útil para crear el clásico menú de opciones. La acción a ejecutar cuándo seleccionemos una opción la tenemos que implementar en el método commandAction(Command c, Displayable d)
   Vamos a ver un ejemplo de cómo crear una lista implícita y así veremos su comportamiento de una manera más clara:

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

public class ListaImplicita extends MIDlet implements CommandListener{
   Command atras, salir;          //Declaracion de Variables
   Display pantalla;
   List menu;
   Form formu1, formu2, formu3;

   public ListaImplicita(){
      pantalla = Display.getDisplay(this); //Creacion de pantallas
      menu = new List("Menú",List.IMPLICIT);
      menu.insert(0,"Opcion3",null);
      menu.insert(0,"Opcion2",null);
      menu.insert(0,"Opcion1",null);
      atras = new Command("Atras",Command.BACK,1);
      salir = new Command("Salir",Command.EXIT,1);
      menu.addCommand(salir);
      formu1 = new Form("Formulario 1");
      formu2 = new Form("Formulario 2");
      formu3 = new Form("Formulario 3");
      formu1.addCommand(atras);
      formu2.addCommand(atras);
```

```
        formu3.addCommand(atras);
        menu.setCommandListener(this);
    }
    public void startApp() {
        pantalla.setCurrent(menu);          // Pongo el menu en pantalla
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void commandAction(Command c, Displayable d){
        if (c == menu.SELECT_COMMAND){                  // Si selecciono
            switch(menu.getSelectedIndex()){            //opcion del menu
                case 0:{ pantalla.setCurrent(formu1);break;}
                case 1:{ pantalla.setCurrent(formu2);break;}
                case 2:{ pantalla.setCurrent(formu3);break;}
            }
        }
        else if (c == atras){                   //Selecciono comando “Atrás”
            pantalla.setCurrent(menu);
        }
        else if (c == salir){                   // Selecciono salir de la aplicacion
            destroyApp(false);
            notifyDestroyed();
        }
    }
}
```

En este ejemplo hemos creado una lista con el constructor simple y luego le hemos ido añadiendo opciones a través del método insert(int posición, String opcion, Image imagen). En este caso la inserción la hemos hecho siempre en la primera posición, por esta razón hemos ido insertando las opciones de la última a la primera.

2. Listas exclusivas: En este tipo de listas sólo podemos seleccionar un elemento a la vez. La selección de uno de ellos implica la deselección del resto. La implementación en el dispositivo de este tipo de listas se realiza a través de Radio Buttons.
   En este caso, al contrario que ocurre con las listas implícitas, la selección de una opción no provoca ningún tipo de evento, por lo que es necesario incluir un Command para salvar el estado del menú de opciones y ver cuál de éstas ha sido seleccionada. El siguiente ejemplo nos va a servir para ver el comportamiento de este tipo de listas.

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

```
public class ListaExclusiva extends MIDlet implements CommandListener {

   Display pantalla;
   Command salir, salvar;
   List menu;
   public ListaExclusiva(){
      pantalla = Display.getDisplay(this);
      salir = new Command("Salir",Command.EXIT,1);
      salvar = new Command("Salvar",Command.ITEM,1);
      String opciones[] = {"Opcion1","Opcion2","Opcion3"};
      menu = new List("Lista exclusiva",List.EXCLUSIVE,opciones,null);
      menu.addCommand(salvar);
      menu.addCommand(salir);
      menu.setCommandListener(this);
   }

   public void startApp() {
       pantalla.setCurrent(menu);
   }

   public void pauseApp() {
   }

   public void destroyApp(boolean unconditional) {
   }

   public void commandAction(Command c, Displayable d){
      if (c == salvar){
         int opcionelegida = menu.getSelectedIndex();
         //salvar opciones en memoria persistente.
         System.out.println("Opcion elegida nº"+(opcionelegida+1));
      }
      else{
          destroyApp(false);
          notifyDestroyed();
      }
   }
}
```

En este ejemplo hemos utilizado el constructor ampliado de la clase List, dándole todos los parámetros (excepto las imágenes). Como vemos, cuándo seleccionamos una opción, las demás quedan deseleccionadas. Es misión nuestra salvar la opción que queremos pulsando sobre el comando "Salvar".

3. Listas múltiples: En estas listas podemos seleccionar los elementos que queramos al mismo tiempo. Como las listas exclusivas también es necesario incluir un Command para salvar el contexto.

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

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

public class ListaMultiple extends MIDlet implements CommandListener {
   Display pantalla;
   List menu;
   Command salir, salvar;
   public ListaMultiple(){
pantalla = Display.getDisplay(this);
      salir = new Command("Salir",Command.EXIT,1);
      salvar = new Command("Salvar",Command.ITEM,1);
      menu = new List("Lista Multiple",List.MULTIPLE);
      menu.insert(menu.size(),"Opcion1",null);
      menu.insert(menu.size(),"Opcion2",null);
      menu.insert(menu.size(),"Opcion3",null);
      menu.addCommand(salir);
      menu.addCommand(salvar);
      menu.setCommandListener(this);
   }
   public void startApp() {
      pantalla.setCurrent(menu);
   }

   public void pauseApp() {
   }

   public void destroyApp(boolean unconditional) {
   }

   public void commandAction(Command c, Displayable d){
      if (c == salvar){
         boolean seleccionados[] = new boolean[menu.size()];
         menu.getSelectedFlags(seleccionados);
         for(int i = 0;i<menu.size();i++){
            System.out.println(menu.getString(i) +
            (seleccionados[i] ? " seleccionada" : " no seleccionada"));
         }
      }
      else{
         destroyApp(false);
         notifyDestroyed();
      }
   }
}
```

En este caso hemos vuelto a utilizar el constructor simple de la clase List. Esta vez hemos ido añadiendo las opciones del menú al final, haciendo uso del método size(), de ésta manera podemos añadir los elementos en el orden correcto. Como podemos ver, el Command "Salvar" es el que determina que opciones tenemos seleccionadas.

Figura 5.3 Distintos tipos de listas

En la Tabla 5.7 podemos ver los métodos de la clase List.

Métodos	Descripción
int append(String texto,Image imagen)	Añade un elemento al final de la lista
void delete(int posición)	Elimina el elemento de la posición especificada
void deleteAll()	Elimina todas las entradas de la lista.
void insert(int pos, String texto, Image im)	Inserta un elemento en la posición especificada
int getFitPolicy()	Devuelve el modo en el que se muestran las entradas de la lista por pantalla
Font getFont(int pos)	Devuelve la fuente del elemento *pos*.
Image getImage(int pos)	Obtiene la imagen de una posición determinada.
int getSelectedFlags(bolean[] array)	Almacena el estado de selección en un array.
int getSelectedIndex()	Obtiene el índice del elemento seleccionado.
String getString(int pos)	Obtiene el texto del elemento indicado por *pos*.
boolean isSelected(int pos)	Determina si está seleccionado el elemento
void removeCommand(Command cmd)	Elimina el Command *cmd*.
void set(int pos, String texto, Image im)	Reemplaza el elemento de la posición *pos*
void setFitPolicy(int modo)	Establece el modo de posicionar las entradas de la lista por pantalla.
void setFont(int pos, Font fuente)	Establece la fuente de la entrada indicada en *pos*
void setSelectCommand(Command cmd)	Selecciona el Command a usar.
int setSelectedFlags(bolean[] array)	Reemplaza el estado de selección por el de *array*
int setSelectedIndex(int pos, bolean selec)	Reemplaza el estado de selección
int size()	Obtiene el número de elementos

Tabla 5.7 Métodos de la clase List

5.3.3. La clase TextBox

public class TextBox **extends** Screen

Una TextBox es una pantalla que nos permite editar texto en ella. Cuando creamos una TextBox, tenemos que especificar su capacidad, es decir, el número de

caracteres que queremos que albergue cómo máximo. Esta capacidad puede ser mayor que la que el dispositivo puede mostrar a la vez. En este caso, la implementación proporciona un mecanismo de *scroll* que permite visualizar todo el texto. Hemos de tener en cuenta que la capacidad devuelta por la llamada al constructor de clase puede ser distinta a la que habíamos solicitado. De cualquier forma, el método getMaxSize() devuelve la capacidad máxima que permite un TextBox ya creado.

Podemos también poner restricciones al texto que se puede incluir en una TextBox. Estas restricciones se encuentran en la clase TextField, íntimamente relacionada con Textbox como veremos más adelante, y se detallan en la Tabla 5.8.

Valor	Descripción
ANY	Permite la inserción de cualquier carácter.
CONSTRAINT_MASK	Se usa cuándo necesitamos determinar el valor actual de las restricciones
EMAILADDR	Permite caracteres válidos para direcciones de correo electrónico.
NUMERIC	Permite sólo números.
PASSWORD	Oculta los caracteres introducidos mediante una máscara para proporcionar privacidad.
PHONENUMBER	Permite caracteres válidos sólo para números de teléfono
URL	Permite caracteres válidos sólo para direcciones URL.

Tabla 5.8 Restricciones de entrada de caracteres

Ya tenemos todo lo necesario para crear cualquier tipo de TextBox. La llamada al constructor la realizaríamos de la siguiente manera:

```
TextBox(String titulo, String texto, int tamaño, int restricciones)
```

Figura 5.4 TextBox con contraseña (*password*)

Por ejemplo, si invocamos al constructor de la siguiente forma:

```
TextBox cajatexto = new TextBox("Contraseña","",30,
                    TextField.NUMERIC | TextField.PASSWORD)
```

obtendremos una TextBox que sólo acepta números y además no los muestra por pantalla al ponerle la restricción PASSWORD (ver Figura 5.4).

La clase TextBox contiene los métodos que se indican en la Tabla 5.9.

Métodos	Descripción
void delete(int desplazamiento, int longitud)	Borra caracteres del TextBox.
int getCaretPosition()	Devuelve la posición del cursor en pantalla.
int getChars(char[] datos)	Copia el contenido del TextBox en *datos*.
int getConstraints()	Devuelve las restricciones de entrada.
int getMaxSize()	Devuelve el tamaño máximo del TextBox.
String getString()	Devuelve el contenido del TextBox.
void insert(char[] datos, int des, int long, int pos)	Inserta un subrango de caracteres de *datos* en el TextBox.
void insert(char[] datos, int pos)	Inserta la cadena de caracteres *datos* en una posición determinada.
void setChars(char[] datos, int des, int long)	Reemplaza el contenido del TextBox por un subconjunto de caracteres de *datos*.
void setConstraints(int restricciones)	Establece las restricciones de entrada.
void setInitialInputMode(String caracteres)	Establece un tipo de entrada inicial.
int setMaxSize(int capacidad)	Establece el tamaño máximo del TextBox.
void setString(String texto)	Establece el contenido del TextBox.
int size()	Devuelve el número de caracteres.

Tabla 5.9 Métodos de la clase TextBox

5.3.4. La clase Form

public class Form **extends** Screen

Un formulario (clase Form) es un componente que actúa como contenedor de un número indeterminado de objetos. Todos los objetos que puede contener un formulario derivan de la clase Item.

El número de objetos que podemos insertar en un formulario es variable pero, teniendo en cuenta el tamaño de las pantallas de los dispositivos MID, se recomienda que se número sea pequeño para evitar así el *scroll* que se produciría si insertáramos demasiados objetos en un formulario.

Para referirnos a los Items o componentes de un formulario usaremos unos índices, siendo 0 el índice del primer Item y Form.size()-1 el del último. El método size() nos devuelve el número de Items que contiene un formulario.

Un mismo Item no puede estar en más de un formulario a la vez. Si, por ejemplo, deseamos usar una misma imagen en más de un formulario, deberemos borrar esa imagen de un formulario antes de insertarla en el que vamos a mostrar por pantalla. Si no cumplimos esta regla, se lanzaría la excepción IllegalStateException.

La Tabla 5.10 muestra los métodos de la clase Form.

Métodos	Descripción
Int append(Image imagen)	Añade una imagen al formulario.
int append(Item item)	Añade un Item al formulario.
int append(String texto)	Añade un String al formulario.
void delete(int num)	Elimina el Item que ocupa lo posición *num*.
void deleteAll()	Elimina todos los Items del formulario.
Item get(int num)	Devuelve el Item que se encuentra en la posición *num*.
int getHeight()	Devuelve la altura en pixels del área disponible (sin realizar *scroll*) para los Items.
int getWidth()	Devuelve la anchura en pixels del área disponible (sin realizar *scroll*) para los Items.
void insert(int num, Item item)	Inserta un Item justo antes del que ocupa la posición *num*.
void set(int num, Item item)	Reemplaza el Item que ocupa la posición *num*.
void setItemStateListener(ItemStateListener listener)	Establece un 'listener' que captura los eventos que produzcan cualquier Item del formulario.
int size()	Devuelve el número de Items del formulario.

Tabla 5.10 Métodos de la clase Form

5.3.4.1. Manejo de Eventos

El manejo de eventos de un formulario se hace de manera muy similar al de los Commands. Es necesario implementar la interfaz ItemStateListener que contiene un solo método abstracto itemStateChanged(Item item). Cuando realizamos algún tipo de acción en un Item de un formulario, ejecutamos el código asociado a ese Item que definamos en el método itemStateChanged(Item item) de igual forma que hacíamos con el método commandAction(Command c, Displayable d) cuando manejamos Commands.

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

public class ManejoItems extends MIDlet implements ItemStateListener,
CommandListener{

    Display pantalla;
    Form formulario;
    TextField txt;
    Command salir;

    public ManejoItems(){
       pantalla = Display.getDisplay(this);
       formulario = new Form("");
       txt = new TextField("Introduce datos","",70,TextField.ANY);
       salir = new Command("Salir",Command.EXIT,1);
```

```
        formulario.append(txt);
        formulario.addCommand(salir);
        formulario.setItemStateListener(this);
        formulario.setCommandListener(this);
    }

    public void startApp() {
        pantalla.setCurrent(formulario);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

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

    public void itemStateChanged(Item i){
        if (i == txt){
            System.out.println("Evento detectado en el TextBox");
        }
    }
}
```

En este ejemplo hemos insertado un objeto TextField que deriva de la clase Item, y cada vez que insertamos texto en el TextField mostramos un mensaje por la consola de salida en la que informamos que se ha producido un evento. Además de TextField los Item que podemos insertar en un formulario son: StringItem, ImageItem, DateField, ChoiceGroup y Gauge. Todos ellos los veremos a continuación.

5.3.4.2. La Clase StringItem

```
public class StringItem extends Item
```

La clase StringItem es la clase más simple que deriva de Item. Simplemente es una cadena no modificable de texto, es decir, una cadena de texto con la que el usuario no puede interactuar de ninguna manera. Para construir un StringItem haremos uso de cualquiera de sus dos constructores:

```
StringItem(String etiqueta, String texto)
StringItem(String etiqueta, String texto, int apariencia)
```

donde:

- etiqueta – es la etiqueta del Item.

- texto – es el texto que contiene el Item
- apariencia – es la apariencia del texto: Item.PLAIN, Item.HYPERLINK, Item.BUTTON.

Invocar al primer constructor es lo mismo que hacer la llamada:
StringItem(etiqueta, texto, Item.PLAIN).

Los métodos que posee la clase StringItem aparecen en la Tabla 5.11, y su apariencia en la Figura 5.5.

Métodos	Descripción
int getAppearanceMode()	Nos devuelve la apariencia del texto
Font getFont()	Nos devuelve la Fuente del texto
String getText()	Nos devuelve el texto del StringItem
void setFont(Font fuente)	Establece la Fuente del texto.
void setText(String texto)	Establece el texto del StringItem

Tabla 5.11 Métodos de la clase StringItem

Figura 5.5 Apariencia de un StringItem

5.3.4.3. La clase ImageItem

public class ImageItem **extends** Item

La clase ImageItem nos da la posibilidad de incluir imágenes en un formulario. Al igual que la clase StringItem, el usuario no podrá interactuar con la imagen.

Para crear un objeto ImageItem haremos uso de uno de sus dos constructores:
ImageItem(String etiqueta, Image imagen, int layout, String textoalt)
ImageItem(String etiqueta, Image imagen, int layout, String textoalt, int apariencia)

El parámetro *textoalt* especifica una cadena de texto alternativa a la imagen en caso de que ésta exceda la capacidad de la pantalla. Por su parte, el parámetro *layout* indica la posición de la imagen en la pantalla. Los valores que puede tomar este parámetro aparecen en la Tabla 5.12.

Valor	Descripción
LAYOUT_LEFT	Imagen posicionada a la izquierda

LAYOUT_RIGHT	Imagen posicionada a la derecha
LAYOUT_CENTER	Imagen centrada
LAYOUT_DEFAULT	Posición por defecto
LAYOUT_NEWLINE_AFTER	Imagen posicionada tras un salto de línea
LAYOUT_NEWLINE_BEFORE	Imagen posicionada antes de un salto de línea

Tabla 5.12 Valores que puede tomar el parámetro *layout*

Los métodos de la clase ImageItem podemos verlos en la Tabla 5.13.

Métodos	Descripción
String getAltText()	Nos devuelve la cadena de texto alternativa.
Int getAppearanceMode()	Nos devuelve la apariencia
Image getImage()	Nos devuelve la imagen
Int getLayout()	Nos devuelve el posicionado de la imagen
void setAltText(String textoalt)	Establece un texto alternativo
void setImage(Image imagen)	Establece una nueva imagen
void setLayout(int layout)	Establece un nuevo posicionado en pantalla

Tabla 5.13 Métodos de la clase ImageItem

5.3.4.4. La clase TextField

```
public class TextField extends Item
```

Un TextField es un campo de texto que podemos insertar en un formulario y dónde podemos editar texto. Ya hemos visto algo muy parecido cuándo estudiamos la clase TextBox. Las diferencias entre ambas son:

- Un TextField tiene que ser insertado en un formulario, mientras que un TextBox puede implementarse por sí mismo.
- TextField deriva de la clase Item, mientras que TextBox deriva directamente de Screen, y sus eventos los controlamos a través de Commands. Por esta razón, los eventos que produce un TextField los controlamos a través del método itemStateChanged(Item item), mientras que en un TextBox los controlamos en el método commandAction(Command c, Displayable d).

Sin embargo, ambas clases comparten las restricciones de entrada que veíamos en la Tabla 5.8.

Para crear un TextField sólo hay que invocar al constructor con los siguientes parámetros:

```
TextField(String etiqueta, String texto, int capacidad, int restricciones)
```

Otros métodos de esta clase pueden verse en la Tabla 5.14.

Métodos	Descripción
void delete(int desplazamiento, int longitud)	Borra caracteres del TextField.
int getCaretPosition()	Devuelve la posición del cursor en pantalla.
int getChars(char[] datos)	Copia el contenido del TextField en *datos*.

int getConstraints()	Devuelve las restricciones de entrada.
int getMaxSize()	Devuelve el tamaño máximo del TextField.
String getString()	Devuelve el contenido del TextField.
void insert(char[] datos, int des, int long, int pos)	Inserta un subrango de caracteres de *datos* en el TextField.
void insert(char[] datos, int pos)	Inserta la cadena de caracteres *datos* en una posición determinada.
void setChars(char[] datos, int des, int long)	Reemplaza el contenido del TextField por un subconjunto de caracteres de *datos*.
void setConstraints(int restricciones)	Establece las restricciones de entrada.
void setInitialInputMode(String caracteres)	Establece un tipo de entrada inicial.
int setMaxSize(int capacidad)	Establece el tamaño máximo del TextField.
void setString(String texto)	Establece el contenido del TextField.
int size()	Devuelve el número de caracteres.

Tabla 5.14 Métodos de la clase TextField

5.3.4.5. La clase DateField

```
public class DateField extends Item
```

El componente DateField nos permite manejar fechas y horas en nuestro formulario. Para ello, hace uso de la clase java.util.Date ya que es con este objeto con el que trabaja.

Cuando creamos un objeto DateField, podemos definir si queremos que el usuario edite la fecha, la hora o ambas. Esto lo hacemos de la siguiente manera:

```
DateField(String etiqueta, int modo)
```

donde modo puede ser DATE, TIME o DATE_TIME.

También podemos realizar la invocación al constructor de la siguiente manera:

```
DateField(String etiqueta, int modo, java.util.TimeZone zonahoraria)
```

El siguiente código nos muestra como crear un componente DateField que, a partir de la hora actual nos permita insertar la fecha y hora que deseemos. El aspecto del objeto DateField depende del dispositivo MID dónde lo ejecutemos, aunque suele ser muy vistoso:

```
Date fechaactual = new Date();  // Creo un objeto Date con fecha actual
DateField fecha = new DateField("Fecha",DateField.DATE_TIME);
fecha.setDate(fechaactual);   // Establezco en el objeto la fecha actual
```

Ya sólo nos queda insertar el componente fecha en un formulario y podremos seleccionar la fecha y hora que deseemos. El resultado visual puede apreciarse en la Figura 5.6.

Figura 5.6 Apariencia del objeto DateField

5.3.4.6. La clase ChoiceGroup

public class ChoiceGroup **extends** Item **implements** Choice

Un componente ChoiceGroup es un grupo de elementos que podemos seleccionar. Es prácticamente lo mismo que el componente List, pero dentro de un formulario.

Para construir un objeto ChoiceGroup realizaremos una llamada a su constructor con los siguientes parámetros:

ChoiceGroup(String etiqueta, int tipo)

ChoiceGroup(String etiq, int tipo, String[] elementos, Image[] imagenes)

La Tabla 5.15 muestra el resto de sus métodos.

Podemos manejar los eventos de una ChoiceGroup de dos formas distintas:

- A través del método itemStateChanged(). En este caso, cuándo un usuario selecciona una opción, el formulario registra un ItemStateListener y se realiza una llamada al método anterior. En este método podemos ver qué elemento ha sido seleccionado y cuál no y realizar las acciones oportunas. Hay que tener en cuenta que cada vez que cambiemos nuestra selección, realizaremos una llamada a este método.
- A través de commandAction(). En este caso podemos añadir un componente Command al formulario que nos valide nuestra selección de opciones. Este modo de operación ya lo hemos visto cuándo estudiábamos la clase List.

Métodos	Descripción
int append(String texto,Image imagen)	Añade un elemento al final de la lista
void delete(int posición)	Elimina el elemento de la posición especificada
void deleteAll()	Elimina todas las entradas de la lista.
void insert(int pos, String texto, Image im)	Inserta un elemento en la posición especificada

int getFitPolicy()	Devuelve el modo en el que se muestran las entradas de la lista por pantala
Font getFont(int pos)	Devuelve la fuente del elemento *pos*.
Image getImage(int pos)	Obtiene la imagen de una posición determinada.
int getSelectedFlags(bolean[] array)	Almacena el estado de selección en un array.
int getSelectedIndex()	Obtiene el índice del elemento seleccionado.
String getString(int pos)	Obtiene el texto del elemento indicado por *pos*.
boolean isSelected(int pos)	Determina si está seleccionado el elemento
void set(int pos, String texto, Image im)	Reemplaza el elemento de la posición *pos*
void setFitPolicy(int modo)	Establece el modo de posicionar las entradas de la lista por pantalla.
void setFont(int pos, Font fuente)	Establece la fuente de la entrada indicada en *pos*
int setSelectedFlags(bolean[] array)	Reemplaza el estado de selección por el de *array*
int setSelectedIndex(int pos, bolean selec)	Reemplaza el estado de selección
int size()	Obtiene el número de elementos

Tabla 5.15 Métodos de la clase ChoiceGroup

5.3.4.7. La clase Gauge

public class Gauge **extends** Item

La clase Gauge implementa un indicador de progresos a través de un gráfico de barras. El componente Gauge representa un valor entero que va desde 0 hasta un valor máximo que definimos al crearlo:

Gauge(String etiqueta, boolean interactivo, int valormax, int valorinicial)

Un Gauge puede ser:

- Interactivo: En este modo, el usuario puede modificar el valor actual del Gauge. En la mayoría de los casos, el usuario dispondrá de unos botones con los cuales podrá modificar el valor a su antojo, pero siempre dentro del rango que se estableció al crear el componente Gauge.
- No Interactivo: En este modo, el usuario no puede modificar el valor del Gauge. Esta tarea es realizada por la aplicación que actualiza su valor a través del método Gauge.setValor(int valor). Este modo es muy útil cuándo queremos indicar el progreso en el tiempo de una tarea determinada.

El código siguiente nos muestra como crear y manejar un Gauge interactivo. La Figura 5.7 muestra su aspecto visual. Este componente lo utilizaremos en nuestra aplicación **Hogar** para seleccionar la temperatura de la calefacción.

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

public class GaugeInt extends MIDlet implements
            CommandListener, ItemStateListener {

   Display pantalla;
   Form formulario;
   Gauge temp;
   Command salir;

   public GaugeInt(){
      pantalla = Display.getDisplay(this);
      formulario = new Form("Temperatura");
      temp = new Gauge("",true,15,0);
      temp.setLabel("Min");
      salir = new
            Command("Salir",Command.EXIT,1);
      formulario.append(temp);
      formulario.addCommand(salir);
      formulario.setCommandListener(this);
      formulario.setItemStateListener(this);
   }

   public void startApp() {
      pantalla.setCurrent(formulario);
   }

   public void pauseApp() {
   }

   public void destroyApp(boolean unconditional) {
   }

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

   public void itemStateChanged(Item i){
      if (i == temp){
         if (temp.getValue() == 0)
            temp.setLabel("Min");
         else if (temp.getValue() == 15) temp.setLabel("Ma
         else temp.setLabel(temp.getValue()+15+" º");
      }
   }
}
```

Figura 5.7 Aspecto de un Gauge

5.3.5. Creación de *MIDlets* usando la API de alto nivel

En este punto ya hemos visto completamente la API de alto nivel. Sabemos crear formularios, insertar ítems en ellos, incluir Commands, etc. Ya tenemos a nuestra disposición las herramientas necesarias para crear aplicaciones profesionales.

Hasta ahora, el método de construcción de aplicaciones ha sido bastante simple. Hemos creado las pantallas que conformaban nuestra aplicación en el método constructor, hemos implementado las interfaces CommandListener e ItemStateListener y hemos controlado las acciones que el usuario haya podido realizar en cualquiera de las pantallas a través de los métodos asociados a las interfaces anteriores. Cuando hablamos de pantallas nos referimos a las clases heredadas de Displayable (Form, List, TextBox, …) Este método es bastante válido para aplicaciones pequeñas como las que hemos creado hasta ahora, pero no tanto cuando queremos crear un MIDlet de mayor tamaño. Imaginemos una aplicación que usa en total 15 pantallas distintas, cada una de ellas con Commands que nos permitan salvar acciones y movernos por las distintas pantallas. Tanto el método constructor, como el método commandAction() llevarían toda la carga del código de la aplicación, y la depuración en este caso sería casi impracticable ya que tendríamos que revisar un gran número de líneas de código fuente.

Para resolver este problema lo que vamos a hacer es modularizar nuestra aplicación. Vamos a crear cada pantalla por separado, donde además de crear los elementos que van a formar parte de ella, vamos a capturar los eventos que produzcan los Commands que insertemos y escribiremos el código de la acción asociada a ese evento. Con este método, podremos crear aplicaciones con el número de pantallas que deseemos, ya que al encontrarse el código de cada pantalla en un archivo *.java* diferente, su depuración y mantenimiento es mucho más factible. Hay que pensar también en las posibles futuras actualizaciones o modificaciones de nuestro MIDlet. Al estar modularizada nuestra aplicación, el trabajo de modificación de pantallas es mucho más sencillo y se realizaría de una manera muy rápida.

Este método de trabajo es el que vamos a seguir para crear nuestra aplicación Hogar con la que vamos a poder controlar distintos dispositivos de nuestra casa a través del teléfono móvil. En este punto sólo vamos a crear la interfaz de usuario de la aplicación, ya que no tenemos aún los conocimientos necesarios para establecer la comunicación entre el móvil y los dispositivos. Una vez que veamos el tema 8, completaremos nuestra aplicación para que sea “totalmente” operativa.

Nuestro MIDlet nos va a permitir comunicarnos con varios dispositivos de nuestra casa: La alarma, el climatizador, las luces, el microondas y el vídeo. Cada uno de estos dispositivos los controlaremos por separado, por lo que crearemos una clase para cada uno de ellos. Nuestro MIDlet básicamente se ocupará de darnos a elegir qué dispositivo queremos controlar a través de un objeto List y comunicarse con cada uno de los formularios de cada dispositivo. En este punto presentaremos las pantallas de

nuestro MIDlet que están implementadas usando la interfaz de usuario de alto nivel. Más adelante presentaremos nuestra aplicación al completo cuando estudiemos la interfaz de bajo nivel.

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

public class Hogar extends MIDlet implements CommandListener {
   public Display pantalla;
   public Edificio edificio;
   private Falarma alarma;
   private Fcalefaccion calefaccion;
   private Fvideo video;
   private Fmicroondas microondas;
   private Fluces luces;
   private CargaEstados cargaest;
   private Alert alerta;
   private List menu;
   private Command salir;
   private Ticker tick;

   public Hogar(){
         pantalla = Display.getDisplay(this);
         edificio = new Edificio();
         cargaest = new CargaEstados(this);
         alarma = new Falarma(this);
         calefaccion = new Fcalefaccion(this);
         video = new Fvideo(this);
         microondas = new Fmicroondas(this);

         luces = new Fluces(this);
         alerta = new Alert("¡Atencion!","",null,AlertType.INFO);
         alerta.setTimeout(Alert.FOREVER);
         tick = new Ticker("Bienvenido a Mi Hogar");
         String opciones[] = {"Alarma", "Climatizacion", "Luces", "Video", "Microondas"};
         menu = new List("Menu",Choice.IMPLICIT,opciones,null);
         menu.setTicker(tick);
         salir = new Command("Salir",Command.EXIT,1);
         menu.addCommand(salir);
         menu.setCommandListener(this);
     }

     public void startApp() {
         pantalla.setCurrent(cargaest);
     }

     public void pauseApp() {
     }

     public void destroyApp(boolean unconditional) {
```

```
        notifyDestroyed();
    }

    // Comentario 1
    public void commandAction(Command c, Displayable d){
        if (c == salir){
            destroyApp(false);
            notifyDestroyed();
        }
        else{//Si selecciono una opción del menú
            switch (menu.getSelectedIndex()){
                case 0: {
                        alarma.setCoord(edificio.getCoordHabActiva());
                        pantalla.setCurrent(alarma);break;
                    }
                case 1: {
                        calefaccion.setCoord(edificio.getCoordHabActiva());
                        pantalla.setCurrent(calefaccion);break;
                    }
                case 2: {
                        pantalla.setCurrent(luces);
                        break;
                    }
                case 3: {
                        if(!edificio.getVideo())    pantalla.setCurrent(video);
                        else {
                            alerta.setString("Video funcionando");
                            pantalla.setCurrent(alerta,menu);
                        }
                        break;
                    }
                case 4:{
                        if (!edificio.getMicroondas())
                            pantalla.setCurrent(microondas);
                        else {
                            alerta.setString("Microondas     funcionando");
                            pantalla.setCurrent(alerta,menu);
                        }
                        break;
                    }
            }
        }
    }
    // Fin comentario 1

    // Comentario 2
    public void verOpciones(){
        pantalla.setCurrent(menu);
        menu.setCommandListener(this);
    }
    // Fin comentario 2
```

```
        // Comentario 3
        public void setCalefaccion(){
            pantalla.setCurrent(calefaccion);
        }

        public void setAlarma(){
            pantalla.setCurrent(alarma);
        }

        public void setInfoLuz1(boolean[] est){
            luces.setInfo1(est);
        }

        public void setInfoLuz2(int[][] coord){
            luces.setInfo2(coord);
        }

        public void setInfoAlarma1(int[][] coord){
            alarma.setInfo1(coord);
        }

        public void setInfoAlarma2(boolean[] est){
            alarma.setInfo2(est);
        }
    // Fin comentario 3
}
```

- Comentario 1

Este código corresponde al menú de nuestro *MIDlet*. En él se nos da la oportunidad de seleccionar el dispositivo de nuestra casa que queramos modificar o ver su estado. Podemos navegar entre las distintas opciones del menú a través de los botones Arriba/Abajo, y seleccionar cualquier dispositivo pulsando Select. Cuándo realicemos esta pulsación aparecerá la pantalla seleccionada.

- Comentario 2

El método verOpciones() lo utilizaremos desde otras pantallas para volver al menú principal.

- Comentario 3

El *MIDlet* también posee algunos métodos que envían información adicional a las distintas pantallas que controlan algunos dispositivos. De esta forma cuándo se cargue la pantalla de ese dispositivo, toda la información aparecerá correctamente.

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

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

public class Fvideo extends Form implements CommandListener, ItemStateListener{
   private Form fecha;
   private ChoiceGroup elec;
   private TextField canal;
   private DateField fechahora;
   private Ticker tick;
   private Command atras, siguiente, aceptarfecha, aceptar;
   private boolean estado, estasig;
   private Date fechaactual;
   private Hogar midlet;
```

// Comentario 4

```
   public Fvideo(Hogar midlet){
          super("Video");
          this.midlet = midlet; //Guardo referencia del MIDlet
          estado = false;//Guardo estado actual
          fechaactual = new Date();//Guardo fecha actual
          estasig = false;
          fecha = new Form("Introduzca fecha y hora");
          fechahora = new DateField("Introduce la fecha y la hora",DateField.DATE_TIME);
          fechahora.setDate(fechaactual);
          canal = new TextField("Seleccione el canal a grabar", "",2,TextField.NUMERIC);
          String opc[] = {"Gabar","Programar Hora"};
          elec = new ChoiceGroup("",Choice.EXCLUSIVE,opc,null);
          tick = new Ticker("Seleccione canal y hora");
          siguiente = new Command("Siguiente",Command.OK,1);
          aceptarfecha = new  Command("Aceptar",Command.OK,1);
          aceptar = new Command("Aceptar",Command.OK,1);
          atras = new Command("Atras",Command.BACK,1);
          fecha.addCommand(aceptar);
          fecha.append(fechahora);
          this.append(canal);
          this.append(elec);
          this.setTicker(tick);
          this.addCommand(atras); this.addCommand(aceptar);
          this.setItemStateListener(this);
          this.setCommandListener(this);
   }
```

// Fin comentario 4

// Comentario 5

```
   public void itemStateChanged(Item item){
          if (item == elec){
               if (elec.getSelectedIndex() == 1){
                    this.removeCommand(aceptar);
                    this.addCommand(siguiente);
                    estasig = true;
               }
```

```
                else{
                    if (estasig){
                        this.removeCommand(siguiente);
                        this.addCommand(aceptar);
                        estasig = false;
                    }
                }
            }
    }
// Fin comentario 5

// Comentario 6
    public void commandAction(Command c, Displayable d){
            if (c == atras){
                midlet.verOpciones();
            }
            else if (c == siguiente){
                midlet.pantalla.setCurrent(fecha);
                fecha.setCommandListener(this);
                fecha.setItemStateListener(this);
            }
            else if (c == aceptar){
                midlet.edificio.setVideo(true);
                midlet.edificio.setCanalVideo(Integer.parseInt(canal.getString()));
                midlet.verOpciones();
            }
        }
// Fin comentario 6
}
```

- Comentario 4

La pantalla que controla el Vídeo nos da la oportunidad de seleccionar el canal a grabar. Una vez seleccionado podemos, o bien grabar en ese momento el canal o seleccionar una hora determinada.

- Comentario 5

En el caso de que seleccionemos una hora determinada, pincharemos a continuación en siguiente y nos aparecerá un objeto DateField en el cual podremos seleccionar tanto el día como la hora a la que deseamos se realice la grabación.

- Comentario 6

Una vez que todo esté a nuestro gusto pulsaremos aceptar y actualizaremos la clase *Edificio* que guardará el nuevo estado del vídeo.

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

```
public class Fmicroondas extends Form implements CommandListener, ItemStateListener{
   private ChoiceGroup potencia, tiempo;
   private TextField txt;
   private Ticker tick;
   private Command atras, aceptar;
   private boolean estado;
   private int indicetxt, nivel, tiem;
   private Hogar midlet;

// Comentario 7
   public Fmicroondas(Hogar midlet){
         super("Microondas");
         this.midlet = midlet;
         estado = false;
         indicetxt = -1;
         String potcad[] = {"Nivel1","Nivel2","Nivel3","Nivel4"};
         String tiempocad[] = {"5 min","10 min","15 min","20 min","Seleccionar tiempo"};
         potencia = new ChoiceGroup("Seleccione potencia",Choice.EXCLUSIVE,potcad,null);
         tiempo = new ChoiceGroup("Seleccione tiempo",Choice.EXCLUSIVE,tiempocad,null);
         txt = new TextField("Tiempo","",2,TextField.NUMERIC);
         tick = new Ticker("Seleccione potencia y tiempo");
         atras = new Command("Atras",Command.BACK,1);
         aceptar = new Command("Aceptar",Command.OK,1);
         this.append(potencia);
         this.append(tiempo);
         this.setTicker(tick);
         this.addCommand(atras);
         this.addCommand(aceptar);
         this.setCommandListener(this);
         this.setItemStateListener(this);
   }
// Fin comentario 7

   public void itemStateChanged(Item item){
         if (item == potencia){
               nivel = potencia.getSelectedIndex()+1;
         }
         else if (item == tiempo){
               if (this.tiempo.getSelectedIndex() == 4){
                     indicetxt = this.append(txt);
                     nivel = 0;
               }
               else{
                     tiem = (tiempo.getSelectedIndex()+1)*5;
                     if (indicetxt != -1){
                           this.delete(indicetxt);
                           indicetxt = -1;
                     }
               }
         }
```

```
    }
// Comentario 8
    public void commandAction(Command c, Displayable d){
        if (c == aceptar){
            midlet.edificio.setMicroondas(true);
            if (nivel != 0)
                midlet.edificio.setNivelesMic(nivel, tiem);
            else  midlet.edificio.setNivelesMic(nivel, Integer.parseInt(txt.getString()));
        }
        if (indicetxt != -1){
            this.delete(indicetxt);
            indicetxt = -1;
        }
        this.potencia.setSelectedIndex(0,true);
        this.tiempo.setSelectedIndex(0,true);
        this.txt.setString("");
        midlet.verOpciones();
    }
// Fin comentario 8
}
```

- Comentario 7

La pantalla que controla el microondas está formada por dos *ChoiceGroup*. El primero de ellos nos va a dar la oportuniad de seleccionar la potencia del microondas, y el segundo nos permitirá seleccionar el tiempo de duración de éste.

- Comentario 8

Una vez que seleccionemos tanto la potencia como el tiempo de duración, enviaremos esta información a la clase Edificio para que actualice el estado del microondas.

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

public class Temper extends Form implements CommandListener, ItemStateListener{

    private Command aceptar;
    private Gauge nivel;
    private Hogar midlet;
    private int temperatura;
// Comentario 9
    public Temper(Hogar m){
        super("Seleccione temperatura");
        midlet = m;
        nivel = new Gauge("18 ºC",true,15,3);
        temperatura = 18;
```

```
            aceptar = new Command("Aceptar",Command.OK,1);
            this.append(nivel);
            this.addCommand(aceptar);
            this.setCommandListener(this);
            this.setItemStateListener(this);
    }
// Fin comentario 9

    public void itemStateChanged(Item item){
            if (item == nivel){//Controlo el Gauge
                temperatura = (nivel.getValue()+15);
                nivel.setLabel(temperatura+" ºC");
            }
    }

    public void commandAction(Command c, Displayable d){
            if (c == aceptar){
                midlet.edificio.setTempHab(temperatura);
                midlet.setCalefaccion();
            }
    }
    public int getTemperatura(){
            return temperatura;
    }
}
```

- Comentario 9

La clase **Temper** es la que se encarga de seleccionar la temperatura de cada habitación. Está formada por un Gauge que nos permite seleccionar la temperatura adecuada dentro de un rango de valores. Una vez que tengamos seleccionada la temperatura deseada, pulsaremos Aceptar y se guardará la nueva temperatura en la clase **Edificio**.

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

public class SelectAlarm extends Form implements CommandListener, ItemStateListener{
    private StringItem msg;
    private ChoiceGroup elec;
    private TextField txt;
    private Ticker tick;
    private Command atras,aceptar;
    private int numtxt;
    private boolean estado;
    private int habitacionActiva = 0;
    private Hogar midlet;

// Comentario 10
    public SelectAlarm(Hogar midlet){
```

```
        super("");
        this.midlet = midlet;     //Guardo la referencia del MIDlet
        numtxt = -1;      //Indice del TextField en el formulario
        tick = new Ticker("");
        txt = new TextField("Introduzca clave","",10,TextField.PASSWORD);
        String opc[] = {"No","Si"};
        elec = new ChoiceGroup("",Choice.EXCLUSIVE,opc,null);
        elec.setSelectedIndex(0,true);
        atras = new Command("Atras",Command.BACK,1);
        aceptar = new Command("Aceptar",Command.OK,1);
        msg = new StringItem("","");
        this.setTicker(tick);
        this.append(msg);
        this.append(elec);
        this.addCommand(atras);
        this.setItemStateListener(this);
        this.setCommandListener(this);
    }
// Fin comentario 10

    public void itemStateChanged(Item item){
        if (item == elec){//Miro qué opción he elegido
            if (elec.getSelectedIndex() == 1){
                numtxt = this.append(txt);
                this.addCommand(aceptar);
            }
            else if (numtxt != -1){
                this.delete(numtxt);
                numtxt = -1;
                this.removeCommand(aceptar);
            }
        }
    }
// Fin comentario 10

// Comentario 11
    public void commandAction(Command c, Displayable d){
        if (c == aceptar){
            //Comprobar contraseña
            estado = !estado;
            midlet.edificio.setAlarHab(estado);
            midlet.setAlarma();
        }
        if (numtxt != -1){
            this.delete(numtxt);
            numtxt = -1;
            this.removeCommand(aceptar);
        }
        this.txt.setString("");
        this.elec.setSelectedIndex(0,true);
        if (c == atras){
```

```
                midlet.setAlarma();
            }
    }
```

```
// Comentario 12
    public void setEstado(boolean est){
            estado = est;
            if (estado){
                msg.setText("¿Desea desactivar la alarma?");
                tick.setString("Alarma activada");
            }else{
                msg.setText("¿Desea activar la alarma?");
                tick.setString("Alarma desactivada");
            }
    }
    // Fin comentario 12

    public boolean getestado(){
            return estado;
    }

    public void setHabitacion(int n){
            habitacionActiva = n;
    }
}
```

- Comentario 10

La clase SelectAlarm se encarga de darnos a elegir entre activar o desactivar la alarma dependiendo del estado de ésta. En cualquier caso tendremos que escribir una contraseña para poder realizar cualquier acción.

- Comentario 11

Una vez que pulsemos en el botón *Aceptar*, se enviará toda la información a la clase Edificio que actualizará el estado de la alarma. Si pulsamos sobre *Atrás*, volveremos al menú principal.

- Comentario 12

Este método me inicializa los textos de la pantalla de alarma dependiendo del estado en el que se encuentre ésta.

```
import java.io.*;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.Canvas;

public class Edificio {

    // Comentario 13
```

```
    private Image planta;
    private int habActiva = 1;
    private int numHabitaciones;
    private int[][] adyacenciaHab;
    private int[][] coordenadasHab;
    private int[][] coordenadasTemp;
    private int[] temperatura;
    private int numLuces;
    private boolean[] estadoLuz;
    private int[][] coordenadasLuz;
    private boolean[] alarma;
    private boolean video = false;
    private boolean microondas = false;
    // Fin comentario 13

        public Edificio() {
            numHabitaciones = getNumHabitaciones();
            try{
                planta = Image.createImage("/planta0.png");
                temperatura = new int[numHabitaciones];
                alarma = new boolean[numHabitaciones];
                adyacenciaHab = new int[numHabitaciones][4];
                coordenadasHab = new int[numHabitaciones][4];
                coordenadasTemp = new int[numHabitaciones][2];
            }
            catch (Exception e){
                System.out.println("No se pudo crear galería de imágenes");
            }
    }

        public int[] mostrarPlanta(int mov){
            int hab = adyacenciaHab[habActiva-1][mov];
            if (hab != 0){
                habActiva = hab;
                return coordenadasHab[hab-1];
            }
            else return coordenadasHab[habActiva-1];
    }


    public void establecerTemperatura(int[] temp){
            for (int i=0;i<numHabitaciones;i++){
                temperatura[i] = temp[i];
            }
    }

    public int getTemperatura(int i){
            return temperatura[i];
    }

    public void setNumLuces(int n){
```

```
        numLuces = n;
        estadoLuz = new boolean[numLuces];
        coordenadasLuz = new int[numLuces][2];
}

public int getNumLuces(){
        return numLuces;
}

public void setEstadoLuz(boolean[] l){
        for (int i=0;i<numHabitaciones;i++){
              estadoLuz[i] = l[i];
        }
}

public void setCoordLuz(int[][] l){
        for (int i=0;i<numLuces;i++){
              for (int j=0;j<=1;j++){
                    coordenadasLuz[i][j] = l[i][j];
              }
        }
}

public void setAlarma(boolean[] alar){
        for (int i=0;i<numHabitaciones;i++){
              alarma[i] = alar[i];
        }
}

public void setVideo(boolean est){
        video = est;
}

public boolean getVideo(){
        return video;
}

public void setMicroondas(boolean est){
        microondas = est;
}

   public boolean getMicroondas(){
        return microondas;
}

public void establecerCoord(int[][] coord){
        for (int i=0;i<numHabitaciones;i++)
              for (int j=0;j<4;j++)
                    coordenadasHab[i][j] = coord[i][j];
}
```

```
    public void establecerAdyacencia(int[][] ady){
            for (int i=0;i<numHabitaciones;i++)
                for (int j=0;j<4;j++)
                    adyacenciaHab[i][j] = ady[i][j];
    }

    public void setCoordTemp(int[][] c){
            for (int i=0;i<numHabitaciones;i++){
                for (int j=0;j<2;j++){
                    coordenadasTemp[i][j] = c[i][j];
                }
            }
    }

    public int[][] getCoordTemp(){
            return coordenadasTemp;
    }

    public int getNumHabitaciones(){
            return 4;
    }

    public Image getPlanta(){
            return planta;
    }

    public int[] getCoordHabActiva(){
            return coordenadasHab[habActiva-1];
    }

    public void setTempHab(int t){
            // Enviar datos al servlet
            temperatura[habActiva-1] = t;
    }

    public boolean getAlarmaHab(){
            return alarma[habActiva-1];
    }

    public boolean[] getEstAlarma(){
            return alarma;
    }

    public void setAlarHab(boolean est){
            alarma[habActiva-1] = est;
    }
}
```

- Comentario 13

Por último, la clase Edificio está formada por varios arrays y matrices que guardan el estado de cada dispositivo y coordenada de dónde se encuentre éste. Guarda información sobre la imagen que forma la planta del edificio y las coordenadas de cada habitación de éste. El resto de la clase está formada por métodos que simplemente actualizan el estado de algún dispositivo o devuelve el valor de éste.

El aspecto de las pantallas que conforman el código anterior puede apreciarse en la Figura 5.8.

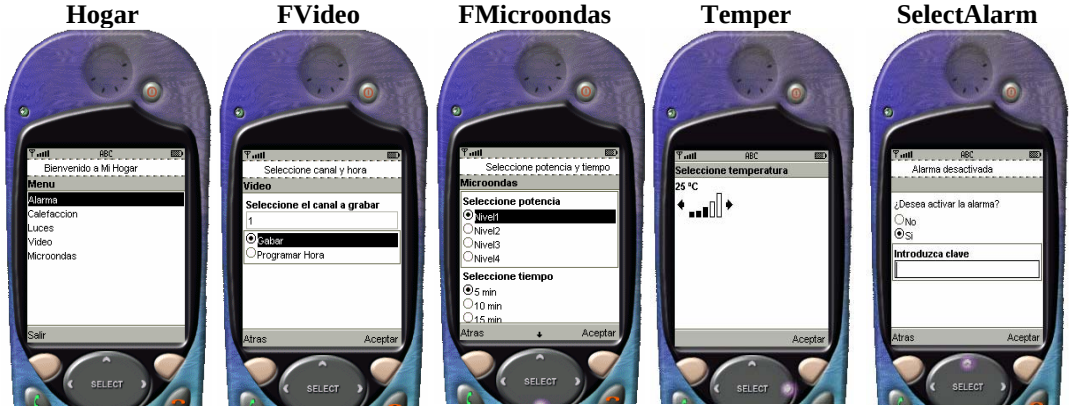

Figura 5.8 Aspecto de diversas pantallas de la aplicación **Hogar**

Los objetos Temper y SelectAlarm no están definidos dentro del *MIDlet* ya que son pantallas definidas dentro de los módulos de climatización y de alarma respectivamente.

Por su parte, la clase Edificio es la que mantiene la información sobre todos los dispositivos, así como las coordenadas de los distintos elementos.

Al código anterior tan solo habría que añadirle las pantallas donde manejaremos la alarma, el climatizador y las luces. Estas pantallas las vamos a implementar usando la interfaz de bajo nivel que vamos a pasar a explicar a continuación.

5.4. La interfaz de usuario de bajo nivel

Todas las pantallas que vayamos a crear usando las APIs de bajo nivel heredan de la clase Canvas. Por esta razón, lo primero de todo es conocer a fondo esta clase y luego iremos profundizando en cada uno de los elementos que la componen.

La clase Canvas es la superclase de todas las pantallas que usan las APIs de bajo nivel, al igual que Screen lo era para las pantallas que usaban las APIs de alto nivel. No existe ningún impedimento que nos permita usar en el mismo MIDlet pantallas tanto derivadas de Canvas como de Screen. Por ejemplo, en la aplicación Hogar, las pantallas de cada dispositivo podrían haberse creado usando las APIs de bajo nivel, con

lo que habríamos conseguido un aspecto mucho más llamativo, mientras que el menú principal podría haber seguido siendo un objeto List.

La clase Canvas permite manejar eventos de bajo nivel y dibujar cualquier cosa por pantalla. Por esta razón se usa como base para la realización de juegos. Esta clase posee un método abstracto paint() que debemos implementar obligatoriamente y es el que se encarga de dibujar en la pantalla del dispositivo MID. En el siguiente ejemplo podemos ver como sería el programa **HolaMundo** creado en una pantalla derivada de Canvas (Figura 5.9).

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class HolaMundo extends MIDlet {
   private HolaCanvas panCanvas;
   private Display pantalla;
   public HolaMundo(){
      pantalla = Display.getDisplay(this);
      panCanvas = new HolaCanvas(this);
   }
   public void startApp() {
      pantalla.setCurrent(panCanvas);
   }
   public void pauseApp() {
   }
   public void destroyApp(boolean unconditional) {
   }
   public void salir(){
      destroyApp(false);
      notifyDestroyed();
   }
}

import javax.microedition.lcdui.*;
public class HolaCanvas extends Canvas implements CommandListener {
   private HolaMundo midlet;
   private Command salir;
   public HolaCanvas(HolaMundo mid){
      salir = new Command("Salir",  Command.EXIT,1);
      this.midlet = mid;
      this.addCommand(salir);
      this.setCommandListener(this);
   }
   public void paint(Graphics g) {
      g.setColor(255,255,255);
      g.fillRect(0,0,getWidth(),getHeight());
      g.setColor(0,0,0);
      g.drawString("Hola Mundo", (getWidth()/2), (getHeight()/2),
           Graphics.BASELINE|Graphics.HCENTER);
   }

   public void commandAction(Command c, Displayable d){
```

```
        if (c == salir){
            midlet.salir();
        }
    }
}
```

Figura 5.9 Ejecución del *MIDlet* **HolaMundo** (Canvas)

En la primera parte de este punto veremos los mecanismos que proporciona la clase Canvas para manejar los eventos de bajo nivel, lo que nos permitirá detectar pulsaciones de teclas o eventos del puntero (si el MID posee este mecanismo). Posteriormente estudiaremos el funcionamiento del método paint() y de la clase Graphics lo que nos permitirá manejar gráficos en nuestras aplicaciones. En la parte final del capítulo veremos algunas clases que nos facilitarán la labor de creación de juegos.

5.4.1. Eventos de bajo nivel

Los eventos dentro de la clase Canvas podemos manejarlos principalmente de dos formas distintas:

- A través de Commands como hemos visto en el ejemplo anterior. Este sistema es el que hemos utilizado hasta el momento, por lo que su funcionamiento es sobradamente conocido.
- A través de códigos de teclas. Este método es el que Canvas proporciona para detectar eventos de bajo nivel. Estos códigos son valores numéricos que están asociados a las diferentes teclas de un MID. Estos códigos se corresponden con las teclas de un teclado convencional de un teléfono móvil (0-9,*,#). La clase Canvas proporciona estos códigos a través de constantes que tienen asociados valores enteros (Tabla 5.16).

Nombre	Valor
KEY_NUM0	48
KEY_NUM1	49

KEY_NUM2	50
KEY_NUM3	51
KEY_NUM4	52
KEY_NUM5	53
KEY_NUM6	54
KEY_NUM7	55
KEY_NUM8	56
KEY_NUM9	57
KEY_STAR	42
KEY_POUND	35

Tabla 5.16 Códigos de teclas

Con estos códigos anteriores ya podemos conocer cual es la tecla que ha pulsado el usuario. Canvas, además proporciona unos métodos que permitirán manejar estos eventos con facilidad. La implementación de estos métodos es vacía, por lo que es misión nuestra implementar los que necesitemos en nuestra aplicación de acuerdo con el propósito de ésta. Los métodos para el manejo de códigos de teclas aparecen en la Tabla 5.17.

Métodos	Descripción
boolean hasRepeatEvents()	Indica si el MID es capaz de detectar la repetición de teclas
String getKeyName(int codigo)	Devuelve una cadena de texto con el nombre del código de tecla asociado
void keyPressed(int codigo)	Se invoca cuando pulsamos una tecla
void keyReleased(int codigo)	Se invoca cuando soltamos una tecla
void keyRepeated(int codigo)	Se invoca cuando se deja pulsada una tecla

Tabla 5.17 Métodos para manejar códigos de teclas

El perfil MIDP nos permite detectar eventos producidos por dispositivos equipados con algún tipo de puntero como un ratón o una pantalla táctil. Para ello, nos proporciona un conjunto de métodos cuya implementación es vacía. Estos métodos en particular, detectan las tres acciones básicas que aparecen en la Tabla 5.18.

Métodos	Descripción
void pointerDragged()	Invocado cuando arrastramos el puntero
void pointerPressed()	Invocado cuando hacemos un ‘click’
void pointerReleased()	Invocado cuando dejamos de pulsar el puntero

Tabla 5.18 Métodos para detectar eventos de puntero

Al igual que hacíamos con los eventos de teclado, es nuestro deber implementar estos métodos si queremos que nuestro *MIDlet* detecte este tipo de eventos. Para saber si el dispositivo MID está equipado con algún tipo de puntero podemos hacer uso de los métodos de la Tabla 5.19.

Método	Descripción
boolean hasPointerEvents()	Devuelve *true* si el dispositivo posee algún

	puntero
boolean has PointerMotionEvents()	Devuelve *true* si el dispositivo puede detectar acciones como pulsar, arrastrar y soltar el puntero.

Tabla 5.19 Métodos para ver si el MID permite eventos de puntero

5.4.2. Manipulación de elementos en una pantalla Canvas

La API de bajo nivel tiene dos funciones principalmente: controlar los eventos de bajo nivel tal y como hemos visto en el apartado anterior y controlar qué aparece en la pantalla del dispositivo MID. En este punto veremos los métodos que nos proporciona Canvas para manipular elementos en pantalla y especialmente la clase Graphics. Antes de empezar a ver éstos métodos vamos a conocer cual es el mecanismo que usa la clase Canvas para dibujar por pantalla.

5.4.2.1. El método paint()

La clase Canvas posee un método abstracto paint(Graphics g) que se ocupa de dibujar el contenido de la pantalla. Para ello, se usa un objeto de la clase Graphics que es el que contiene las herramientas gráficas necesarias y que se pasa como parámetro al método paint(). Cuando vayamos a implementar este método tendremos que tener en cuenta lo siguiente:

- El método paint() nunca debe ser llamado desde el programa. El gestor de aplicaciones es el que se encarga de realizar la llamada a éste método cuando sea necesario.
- Cuando deseemos que se redibuje la pantalla actual debido a alguna acción en concreto del usuario o como parte de alguna animación, deberemos realizar una llamada al método repaint(). Al igual que ocurre con los eventos de teclado, la petición se encolará y será servida cuando retornen todas las peticiones anteriores a ella.
- La implementación del MID no se encarga de limpiar la pantalla antes de cada llamada al método paint(). Por esta razón, éste método debe pintar cada pixel de la pantalla para, de esta forma, evitar que se vean porciones no deseadas de pantallas anteriores.

Hemos dicho que el gestor de aplicaciones se encarga de invocar al método paint() cuando sea necesario. Esta llamada la realiza normalmente cuando se pone la pantalla Canvas como pantalla activa a través del método de la clase Display setCurrent(Canvas) o después de invocar al método showNotify(). Hemos de tener en cuenta que una pantalla Canvas no posee la capacidad por sí misma de restaurar su estado en caso de que el AMS interrumpa la ejecución normal de la aplicación para, por ejemplo, avisar de una llamada entrante. Para resolver este inconveniente, Canvas nos

proporciona dos métodos cuya implementación es vacía: hideNotify() y showNotify(). El primero de ellos es llamado justo antes de interrumpir a la aplicación y borrar la pantalla actual. En él podríamos incluir el código necesario para salvar el estado actual de la pantalla, variables, etc, y así posteriormente poder restaurar correctamente la aplicación. El método showNotify() es invocado por el gestor de aplicaciones justo antes de devolver éste el control a la aplicación. En él podemos insertar el código necesario para restaurar correctamente la pantalla de la aplicación.

5.4.2.2. La clase Graphics

```
public class Graphics
```

Hemos visto que cuando invocamos al método paint(Graphics g) tenemos que pasarle como parámetro un objeto Graphics. Este objeto nos proporciona la capacidad de dibujar en una pantalla Canvas. Un objeto Graphics lo podemos obtener sólo de dos maneras:

- Dentro del método paint() de la clase Canvas. Aquí podemos usar el objeto Graphics para pintar en la pantalla del dispositivo.
- A través de una imagen usando el siguiente código:

```
Image imgtemp = Image.createImage(ancho,alto);
Graphics g = imgtemp.getGraphics();
```

La clase Graphics posee multitud de métodos que nos permitirán seleccionar colores, dibujar texto, figuras geométricas, etc. En los siguientes puntos vamos a ver cada uno de estos métodos según su función.

5.4.2.3. Sistema de coordenadas

Vamos a comenzar a partir de este punto a explicar las nociones necesarias para poder dibujar en la pantalla de un MID.

Dado que la clase Graphics nos va a proporcionar bastantes recursos para dibujar en una pantalla Canvas, hemos de comprender como se organiza esta pantalla en base a los pixeles, ya que al programar en bajo nivel hemos de trabajar a nivel de pixel.

La clase Canvas nos proporciona los métodos necesarios para obtener el ancho y el alto de la pantalla a través de getWidth() y getHeight() respectivamente. Una posición en la pantalla estará definida por dos coordenadas x e y que definirán el desplazamiento lateral y vertical, siendo la posición (0,0) el pixel situado en la esquina superior izquierda. Incrementando los valores de x e y, nos moveremos hacia la derecha y hacia abajo respectivamente.

Es posible cambiar el origen de coordenadas de la pantalla mediante el método de la clase Graphics translate(int x, int y). Éste método cambia el origen de coordenadas al punto definido por los parámetros x e y provocando un desplazamiento de todos los objetos en pantalla. Piénsese en lo útil que resulta este método para provocar *scrolls*.

5.4.2.4. Manejo de colores

La clase Graphics nos proporciona métodos con los que podremos seleccionar colores, pintar la pantalla de un color determinado o zonas de ellas, dibujar líneas, rectángulos, etc. Como programadores de *MIDlets* es misión nuestra adaptar nuestra aplicación a las capacidades gráficas del dispositivo donde se vaya a ejecutar. Disponemos para ello de varios métodos que nos suministra la clase Display con los que podremos saber si el dispositivo en cuestión cuenta con pantalla a color o escala de grises (Display.isColor()), y también podremos conocer el número de colores soportados (Display.numColors()). Para crear aplicaciones de calidad deberíamos usar constantemente estos métodos en ellas.

En particular, la clase Graphics nos proporciona un modelo de color de 24 bits, con 8 bits para cada componente de color: rojo, verde y azul.

Podemos seleccionar un color invocando al método Graphics.setColor(int RGB) o Graphics.setColor(int rojo, int verde, int azul) donde podemos indicar los niveles de los componentes que conforman el color que deseamos usar. Veamos como se realizaría la selección de algunos colores básicos:

```
Graphics g;
g.setColor(0,0,0)              //Selecciono el color negro
g.setColor(255,255,255)    //Selecciono el color blanco
g.setColor(0,0,255)            //Selecciono el color azul
g.setColor(#00FF00)            //Selecciono el color verde
g.setColor(255,0,0)            //Selecciono el color rojo
```

Si después de seleccionar un color se escribe lo siguiente:

```
g.fillRect(0,0,getWidth(),getHeight())
```

se pintaría toda la pantalla del color seleccionado con anterioridad.

5.4.2.5. Manejo de texto

Es posible incluir texto en una pantalla Canvas. Al igual que hacíamos en el punto anterior con los colores, es posible seleccionar un estilo de letra con el que posteriormente podremos escribir texto por pantalla. Para ello nos ayudaremos de la clase Font que nos permitirá seleccionar el tipo de letra y almacenarlo en un objeto de este tipo para posteriormente usarlo en nuestro Canvas.

Para seleccionar un tipo de letra tenemos que tener en cuenta los tres atributos que definen nuestra fuente (ver Tabla 5.20).

	Atributos
Aspecto	FACE_SYSTEM
	FACE_MONOSPACE
	FACE_PROPORTIONAL
Estilo	STYLE_PLAIN

	STYLE_BOLD
	STYLE_ITALIC
	STYLE_UNDERLINED
Tamaño	SIZE_SMALL
	SIZE_MEDIUM
	SIZE_LARGE

Tabla 5.20 Fuentes disponibles

Realizando combinaciones de atributos tendremos bastantes tipos de fuentes donde elegir. Para crear una determinada tendremos que invocar al método Font.getFont(int aspecto, int estilo, int tamaño) que nos devuelve el objeto Font deseado. Por ejemplo:

```
Font fuente = Font.getFont(FACE_SYSTEM,STYLE_PLAIN,SIZE_MEDIUM);
```

Una vez obtenido el objeto Font, deberemos asociarlo al objeto Graphics que es el que realmente puede escribir texto en pantalla. Esto lo haremos de la siguiente manera:

```
g.setFont(fuente);          //Seleccionamos la fuente activa.
g.drawString("Cadena de texto", getWidth()/2, getHeight()/2,
        BASELINE|HCENTER);
```

Este código selecciona un tipo de fuente con la que posteriormente se escribe la cadena "Cadena de texto" posicionado en la pantalla en el punto medio (getWidth()/2, getHeight()/2). El parámetro BASELINE|HCENTER indica el posicionamiento del texto con respecto al punto medio.

5.4.2.6. Posicionamiento del texto

Ya hemos visto que al mostrar una cadena de texto por pantalla hemos de indicar la posición en la que queremos situarla. Esta posición viene indicada por el punto de anclaje o *anchorpoint* (en inglés). En la Figura 5.10 podemos ver el valor y situación dentro de la cadena de texto de los distintos *anchorpoints*.

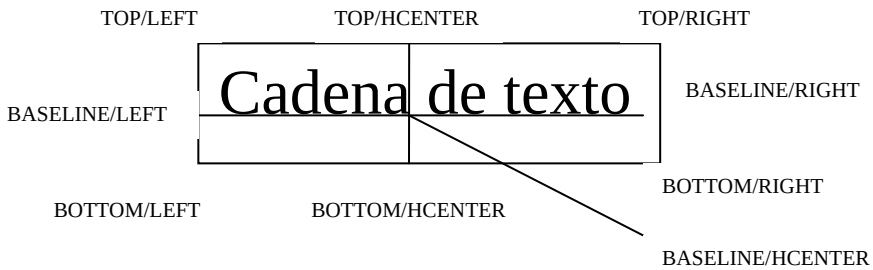

Figura 5.10 Opciones de anclaje disponibles para una cadena de texto

En el ejemplo anterior lo que hicimos fue poner el punto de anclaje BASELINE|HCENTER de la cadena "Cadena de texto" en la posición central definida por getWidth()/2, getHeight()/2.

5.4.2.7. Figuras geométricas

Ya decíamos anteriormente que la clase Graphics nos proporcionaba métodos capaces de mostrar figuras geométricas por pantalla. Estas figuras pueden ser: líneas, rectángulos y arcos.

- **Líneas**

Esta es la figura más simple que podemos representar. Una línea queda definida por un punto inicial que representaremos por (x1,y1) y un punto final representado por (x2,y2). Pues esa es toda la información que necesitamos para dibujar cualquier línea en una pantalla Canvas, simplemente llamando a su constructor de la siguiente forma:

```
g.setColor(0,0,0)          //Selecciono el color negro
g.drawLine(x1,y1,x2,y2)    //Dibujo una línea desde (x1,y1) a (x2,y2)
```

El ancho de la línea dibujada será de 1 pixel. Además podremos definir si la traza de la línea queremos que sea continua o discontinua. Esto lo realizaremos a través de los métodos getStrokeStyle() y setStrokeStyle(int estilo) que devuelve el estilo de la línea o lo selecciona respectivamente. Es estilo puede ser SOLID (líneas continuas) o DOTTED (líneas discontinuas). El estilo que seleccionemos afecta también a cualquier figura geométrica que dibujemos.

- **Rectángulos**

Aquí se nos da la posibilidad de dibujar hasta 4 tipos diferentes de rectángulos.

1. Plano
2. Plano con color de relleno
3. Redondeado
4. Redondeado con color de relleno

Para crear cualquier tipo de rectángulo tenemos que definir cuatro parámetros. Los dos primeros corresponden al punto inicial (x,y), y los dos últimos al ancho y alto del rectángulo:

```
g.setColor(0,0,0);              //Selecciono el color negro
g.drawRect(x,y,ancho,alto);     //Dibujo un rectángulo
```

Si queremos dibujar un rectángulo con color de relleno tendremos que escribir:

```
g.fillRect(x,y,ancho,alto);
```

Cuando vayamos a construir rectángulos con las esquinas redondeadas, es necesario definir dos parámetros más: *arcWidth* y *arcHeight* que especifican el grado de redondez, tal y como aparece en la Figura 5.11.

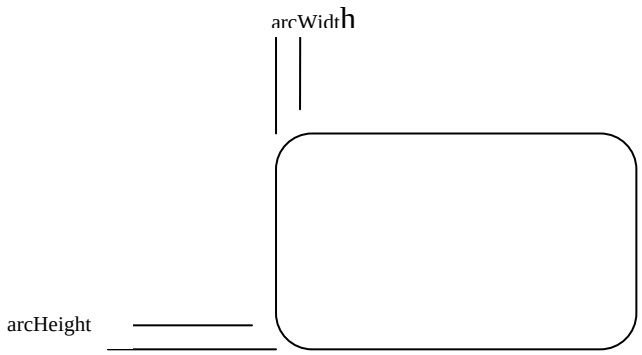

Figura 5.11 Rectángulos de esquinas redondeadas

La creación de un rectángulo de este tipo se realizaría de una de las siguientes maneras:

```
g.drawRoundRect(x,y,ancho,alto,arcWidth,arcHeight)
g.fillRoundRect(x,y,ancho,alto,arcWidth,arcHeight)
```

- **Arcos**

En este caso podemos dibujar dos tipos de arcos: Arco simple o arco con color de relleno. Para dibujar un arco tenemos que imaginarnos que ese arco va dentro de una "caja" (ver Figura 5.12).

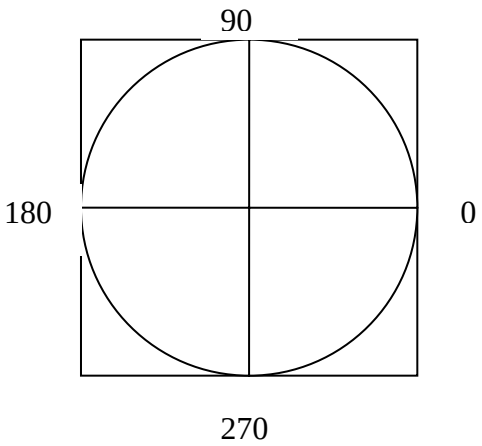

Figura 5.12 Gradación de un arco

Aquí podemos ver un círculo y también el cuadrado que delimita a ese círculo. Cuándo vayamos a crear un arco tenemos que tener en mente la "caja" que lo va a delimitar. Esto es así porque a la hora de crear un arco hemos de crear primero esa caja, especificando los parámetros como hacíamos cuando creábamos un rectángulo, además del ángulo inicial y ángulo total. El ángulo inicial es desde donde empezaremos a

construir el arco, siendo 0 el punto situado a las 3 en punto, 90 el situado a las 12, 180 el situado a las 9 y 270 el situado a las 6. El ángulo total especifica la amplitud que tendrá el arco. Si por ejemplo construimos un arco de la siguiente manera:

```
g.drawArc(1,1,getWidth()-1,getHeight()-1,0,270);
```

ó

```
g.fillArc(1,1,getWidth()-1,getHeight()-1,90,290);
```

podemos ver el resultado en la Figura 5.13.

Figura 5.13 Arco sin relleno y arco con color de relleno

- **Imágenes**

Ya veíamos cuando estudiábamos el interfaz de alto nivel que era posible insertar imágenes en nuestros formularios. Estas imágenes que podíamos insertar provenían del archivo .jar ya que estaban almacenadas en un archivo gráfico .png. Pues bien, en una pantalla Canvas también es posible insertar imágenes. Además, aquí es posible usar dos tipos distintos de objetos Image:

- Inmutables: Son imágenes que, como las que usábamos en la API de alto nivel, provienen de un archivo con formato gráfico. Se llaman inmutables ya que no es posible variar su aspecto.

```
Image im = Image.createImage("\imagen.png");
```

- Mutables: Cuando tratamos con imágenes mutables, realmente estamos trabajando sobre un bloque de memoria en el que podemos crear la imagen que deseemos, modificarla, etc.

```
Image im = Image.createImage(75,25);
im = crearImagen();
```

Una vez creado el objeto Image, sea del tipo que sea, sólo nos queda mostrarlo por la pantalla del dispositivo MID. La clase Graphics nos proporciona el método drawImage(imagen,x,y,anchorpoint) que nos permite mostrar la imagen especificada en el primer parámetro situando el punto de anclaje *anchorpoint* que indiquemos en la posición (x,y).

```
public void paint(Graphics g){
        …
        g.drawImage(imagen,x,y,anchorpoint);
        …
}
```

Al igual que ocurría con las cadenas de texto, las imágenes las posicionaremos en pantalla ayudándonos de los puntos de anclaje. En este caso poseemos los puntos que ilustra la Figura 5.14.

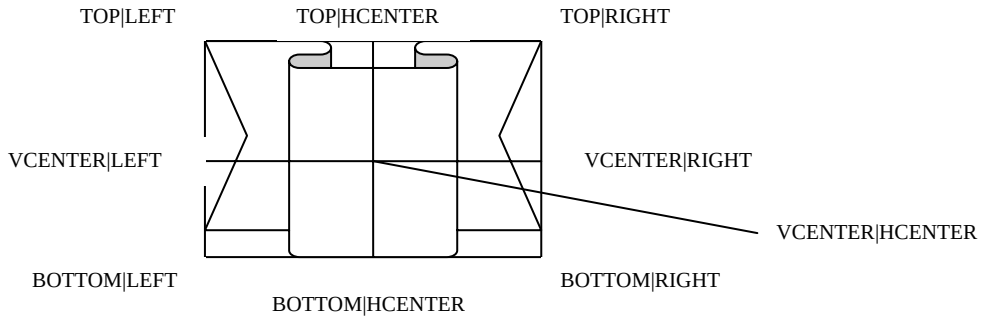

Figura 5.14 Opciones de anclaje disponibles para una imagen

En el siguiente ejemplo vamos a mostrar una imagen por pantalla, la cual podremos mover a nuestro gusto usando el teclado del dispositivo MID (Figura 5.15).

```
import javax.microedition.lcdui.*;
public class Imagen extends Canvas implements CommandListener {
   private MoverImagen midlet;
   private Image im;
   private Command salir;
   private int x, y;
   public Imagen(MoverImagen mid){
      salir = new Command("Salir",Command.EXIT,1);
      this.midlet = mid;
      this.addCommand(salir);
      this.setCommandListener(this);
      try{
         im = Image.createImage("/dibujo.png");
      }
      catch (Exception e){
         System.out.println("Error al cargar archivo de imagen");
      }
      x = 0;
      y = 0;
   }
   public void paint(Graphics g) {
      g.setColor(255,255,255);
      g.fillRect(0,0,getWidth(),getHeight());
      g.setColor(0,0,0);
```

```
        g.drawImage(im,x,y,Graphics.TOP|Graphics.LEFT);
    }
    protected void keyPressed(int keyCode) {
      switch(getGameAction(keyCode)){
        case Canvas.DOWN: { if ((y+20)<getHeight()) y = y+1;
                    break; }
        case Canvas.LEFT: { if (x > 0) x = x-1;
                    break;}
        case Canvas.UP:   { if (y > 0) y = y-1;
                    break;}
        case Canvas.RIGHT:{ if ((x+20) < getWidth()) x = x+1;
                    break;}
      }
      this.repaint();
    }
    public void commandAction(Command c, Displayable d){
      if (c == salir){
        midlet.salir();
      }
    }
}
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class MoverImagen extends MIDlet {
    private Imagen panCanvas;
    private Display pantalla;
    public MoverImagen(){
      pantalla = Display.getDisplay(this);
      panCanvas = new Imagen(this);
    }
    public void startApp() {
      pantalla.setCurrent(panCanvas);
    }
    public void pauseApp() {
    }
    public void destroyApp(boolean uncond) {
    }
    public void salir(){
      destroyApp(false);
      notifyDestroyed();
    }
}
```

Figura 5.15 Ejecución de MoverImagen.java

Antes de adentrarnos en el siguiente punto, parece un buen momento para ver el código que faltaba de nuestra aplicación **Hogar**. Con este código ya tenemos completa y "totalmente operativa" nuestra aplicación. Decimos "totalmente operativa" porque como veremos a continuación, vamos a realizar una simulación sobre los estados del dispositivo ya que lo cargaremos de forma local. La clase CargaEstados como veremos se encargará de inicializar los estados y coordenadas de cada dispositivo, con lo que el *MIDlet* cada vez que se ejecute siempre se iniciará con los mismos estados. En el capítulo 8 modificaremos esta clase para que toda la información la recibamos desde un servlet y así las modificaciones que efectuemos tengan efecto para posteriores ejecuciones de nuestro *MIDlet*.

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

public class Falarma extends Canvas implements CommandListener{
   //Declaración de variables
   private SelectAlarm temp;
   private Ticker tick;
   private Command atras, aceptar;
   private int[][] coordTemp;
   private boolean[] estAlarma;
   private Image planta, conectada, noConectada;
   private int[] coordRect;
   private Hogar midlet;

   public Falarma(Hogar midlet){
      this.midlet = midlet; //Guardo la referencia del MIDlet
      temp = new SelectAlarm(midlet);
      coordRect = new int[4];
      planta = midlet.edificio.getPlanta();
      tick = new Ticker("Alarma");
      //Creo los elementos del formulario principal
      atras = new Command("Atras",Command.BACK,1);
      aceptar = new Command("Aceptar",Command.OK,1);
```

```
        //Inserto los elementos en el formulario principal
        this.setTicker(tick);
        this.addCommand(atras);
        this.setCommandListener(this);
        try{
            conectada = Image.createImage("/conec.png");
            noConectada = Image.createImage("/nocon.png");
        }catch (Exception e){
            System.out.println(e);
        }
    }

    // Comentario 1
    public void paint(Graphics g){
        estAlarma = midlet.edificio.getEstAlarma();
        g.drawImage(planta,0,0,Graphics.TOP|Graphics.LEFT);
        dibujarAlarmas(g);
        g.setColor(255,0,0);
        g.drawRect(coordRect[0],coordRect[1],coordRect[2],coordRect[3]);
    }
    // Fin comentario 1

    // Comentario 2
    public void dibujarAlarmas(Graphics g){
        g.setColor(255,255,255);
        for (int i=0;i<midlet.edificio.getNumHabitaciones();i++){
            if (estAlarma[i]){
                g.drawImage(conectada,coordTemp[i][0],coordTemp[i][1],
                            Graphics.TOP|Graphics.LEFT);
            }else g.drawImage(noConectada,coordTemp[i][0],coordTemp[i][1],
                              Graphics.TOP|Graphics.LEFT);
        }
    }
    // Fin comentario 2

public void commandAction(Command c, Displayable d){
        if (c == atras)
            midlet.verOpciones();
    }

    // Comentario 3
    public void keyPressed(int codigo){
        int ncod = getGameAction(codigo);
        int[] ncoord;
        switch (ncod){
            case Canvas.FIRE:{
                    temp.setEstado(midlet.edificio.getAlarmaHab());
                    midlet.pantalla.setCurrent(temp);
                    break;
                }
```

```
            case Canvas.UP:{
                    ncoord = midlet.edificio.mostrarPlanta(0);
                    for (int i=0;i<4;i++)
                        coordRect[i] = ncoord[i];
                    break;
                }
            case Canvas.DOWN:{
                    ncoord = midlet.edificio.mostrarPlanta(1);
                    for (int i=0;i<4;i++)
                        coordRect[i] = ncoord[i];
                    break;
                }
            case Canvas.RIGHT:{
                    ncoord = midlet.edificio.mostrarPlanta(2);
                    for (int i=0;i<4;i++)
                        coordRect[i] = ncoord[i];
                    break;
                }
            case Canvas.LEFT:{
                    ncoord = midlet.edificio.mostrarPlanta(3);
                    for (int i=0;i<4;i++)
                        coordRect[i] = ncoord[i];
                    break;
                }
        }
        ncoord = null;
        repaint();
    }
        // Fin comentario 3

    public void setCoord(int[] c){
        for (int i=0;i<midlet.edificio.getNumHabitaciones();i++)
            coordRect[i] = c[i];
    }

    public void setInfo1(int[][] c){
        coordTemp = c;
    }

    public void setInfo2(boolean[] est){
        estAlarma = est;
    }
}
```

- Comentario 1

La pantalla FAlarma deriva de Canvas y, por tanto, debemos implementar el método paint() que será el encargado de dibujar lo que vemos por pantalla. En este caso, dibujamos la planta del edificio, seguido de las alarmas dentro de las coordenadas

pasadas por la clase CargaEstados. Por último dibujamos un rectángulo que nos marcará una de las habitaciones dejando a ésta como habitación activa.

- Comentario 2

El método dibujarAlarmas() simplemente recorre el array de estados de la alarma y dibuja en pantalla un círculo rojo si la alarma está desconectada o verde en caso contrario

- Comentario 3

Este método controla las acciones que realice el usuario. Aquí se controla el movimiento del rectángulo entre las distintas habitaciones. Para ello nos ayudamos con un método de la clase *Edificio* que nos devuelve las nuevas coordenadas del rectángulo dependiendo de la tecla pulsada por el usuario. Si pulsamos el botón *Select* accederemos a la pantalla implementada por la clase *SelectAlarm*.

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

public class Fcalefaccion extends Canvas implements CommandListener{
   private Temper temp;
   private Gauge nivel;
   private Ticker tick;
   private Command atras, aceptar, acptemp;
   private int[][] coordTemp;
   private boolean estado;
   private Image planta;
   private int[] coordRect;
   private Hogar midlet;

   public Fcalefaccion(Hogar midlet){
      this.midlet = midlet; //Guardo la referencia del MIDlet
      temp = new Temper(midlet);
      coordRect = new int[4];
      planta = midlet.edificio.getPlanta();
      coordTemp = new int[midlet.edificio.getNumHabitaciones()][2];
      tick = new Ticker("Climatización");
      atras = new Command("Atras",Command.BACK,1);
      aceptar = new Command("Aceptar",Command.OK,1);
      this.setTicker(tick);
      this.addCommand(atras);
      this.setCommandListener(this);
   }

   // Comentario 4
   public void paint(Graphics g){
      coordTemp = midlet.edificio.getCoordTemp();
      g.drawImage(planta,0,0,Graphics.TOP|Graphics.LEFT);
```

```
        dibujarTemperaturas(g);
        g.setColor(255,0,0);
        g.drawRect(coordRect[0],coordRect[1],
                   coordRect[2],coordRect[3]);
    }
    // Fin comentario 4

    public void dibujarTemperaturas(Graphics g){
        g.setColor(255,255,255);
        for (int i=0;i<midlet.edificio.getNumHabitaciones();i++){
            g.drawString(""+midlet.edificio.getTemperatura(i),
                         coordTemp[i][0],coordTemp[i][1],
                         Graphics.TOP|Graphics.LEFT);
        }
    }

    public void commandAction(Command c, Displayable d){
        if (c == atras){
            midlet.verOpciones();
        }else if (c == acptemp){
            midlet.pantalla.setCurrent(this);
            repaint();
        }
    }
    // Comentario 5
    public void keyPressed(int codigo){
        int ncod = getGameAction(codigo);
        int[] ncoord;
        switch (ncod){
            case Canvas.FIRE:{
                    midlet.pantalla.setCurrent(temp);
                    break;
                }
            case Canvas.UP:{
                    ncoord = midlet.edificio.mostrarPlanta(0);
                    for (int i=0;i<4;i++)
                        coordRect[i] = ncoord[i];
                    break;
                    }
            case Canvas.DOWN:{
                    ncoord = midlet.edificio.mostrarPlanta(1);
                    for (int i=0;i<4;i++)
                        coordRect[i] = ncoord[i];
                    break;
                }
            case Canvas.RIGHT:{
                    ncoord = midlet.edificio.mostrarPlanta(2);
                    for (int i=0;i<4;i++)
                        coordRect[i] = ncoord[i];
                    break;
                }
```

```
            case Canvas.LEFT:{
                        ncoord = midlet.edificio.mostrarPlanta(3);
                        for (int i=0;i<4;i++)
                              coordRect[i] = ncoord[i];
                        break;
                  }
      }
      ncoord = null;
      repaint();
   }
   // Fin comentario 5

   public void setCoord(int[] c){
      for (int i=0;i<4;i++)
            coordRect[i] = c[i];
   }

   public void setCoordTemp(int[][] c){
      coordTemp = c;
   }
}
```

- Comentario 4

La pantalla **FCalefacción** también hereda de Canvas por lo que es necesario implementar el método paint() que se encargará de dibujar los elementos por pantalla.

- Comentario 5

Este trozo de código es casi igual que el mostrado en el comentario 3. En este caso accedemos a la clase **Temper** si pulsamos el botón Select donde podremos seleccionar la temperatura deseada de la habitación.

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

   // Comentario 6
public class Fluces extends Canvas implements CommandListener{
   private Ticker tick;
   private Command atras;
   private int[][] coordLuz;
   private boolean[] estLuz;
   private Image planta, encendida, apagada, selec1, selec2;
   private Hogar midlet;
   private int luzSeleccionada;
   // Fin comentario 6

   public Fluces(Hogar midlet){
```

```
        this.midlet = midlet; //Guardo la referencia del MIDlet
        planta = midlet.edificio.getPlanta();
        luzSeleccionada = 0;
        tick = new Ticker("Iluminacion");
        atras = new Command("Atras",Command.BACK,1);
        this.setTicker(tick);
        this.addCommand(atras);
        this.setCommandListener(this);
        try{
            encendida = Image.createImage("/luzencendida.png");
            apagada = Image.createImage("/luzapagada.png");
            selec1 = Image.createImage("/luzselec1.png");
            selec2 = Image.createImage("/luzselec2.png");
        }catch (Exception e){
            System.out.println(e);
        }
    }

    public void paint(Graphics g){
        g.drawImage(planta,0,0,Graphics.TOP|Graphics.LEFT);
        dibujarLuces(g);
    }

public void dibujarLuces(Graphics g){
        for (int i=0;i<midlet.edificio.getNumLuces();i++){
            if (luzSeleccionada == i){
                if (estLuz[i]) g.drawImage(selec2,coordLuz[i][0],coordLuz[i][1],
                                           Graphics.TOP|Graphics.LEFT);
                else g.drawImage(selec1,coordLuz[i][0],coordLuz[i][1],
                                 Graphics.TOP|Graphics.LEFT);
            }else{
                if (estLuz[i]) g.drawImage(encendida,coordLuz[i][0],coordLuz[i][1],
                                           Graphics.TOP|Graphics.LEFT);
                else g.drawImage(apagada,coordLuz[i][0],coordLuz[i][1],
                                 Graphics.TOP|Graphics.LEFT);
            }
        }
    }

    public void commandAction(Command c, Displayable d){
        if (c == atras)
            midlet.verOpciones();
    }

    public void keyPressed(int codigo){
        int ncod = getGameAction(codigo);
        switch (ncod){
            case Canvas.FIRE:{
                    estLuz[luzSeleccionada] = !estLuz[luzSeleccionada];
                    midlet.edificio.setEstadoLuz(estLuz);
                    break;
```

```
                }
            case Canvas.RIGHT:{
                    if (luzSeleccionada < (midlet.edificio.getNumLuces()-1))
                        luzSeleccionada++;
                    break;
                }
            case Canvas.LEFT:{
                    if (luzSeleccionada > 0)
                        luzSeleccionada--;
                    break;
                }
        }
        repaint();
    }

    public void setInfo1(boolean[] est){
        estLuz = est;
    }

    public void setInfo2(int[][] coord){
        coordLuz = coord;
    }
}
```

- Comentario 6

La clase **FLuces** no aporta nada nuevo al código de las otras dos pantallas. En este caso lo único que varía es que no tenemos ningún rectángulo que se mueva entre las habitaciones, si no que la navegación se realiza entre las distintas luces de la casa. En el método dibujarLuces() hacemos algo muy parecido a lo del comentario 2.

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

public class CargaEstados extends Form implements CommandListener, Runnable{
    private Gauge carga;
    private int indice, numluces;
    private Command cancelar, aceptar, salir;
    private Thread t;
    private Hogar midlet;

    public CargaEstados(Hogar midlet){
        super("Cargando información");
        this.midlet = midlet;
        carga = new Gauge("Pulse Aceptar",false,15,0);
        cancelar = new Command("Cancelar",Command.CANCEL,1);
        aceptar = new Command("Aceptar",Command.OK,1);
```

```
    salir = new Command("Salir",Command.EXIT,1);
    this.append(carga);
    this.addCommand(aceptar);
    this.setCommandListener(this);
  }

  public void start(){
    t = new Thread(this);
    t.start();
  }

  // Comentario 7
  public void run(){
    System.out.println("Carga en pantalla");
    carga.setLabel("Comprobando alarma");
    System.out.println("Comprobando alarma");
    verEstadoAlarma();
    System.out.println("Estado alarma comprobado correctamente");
    carga.setValue(3);
    carga.setLabel("Comprobando calefacción");
    verEstadoCalefaccion();
    System.out.println("Estado calefacción comprobado correctamente");
    carga.setValue(6);
    carga.setLabel("Comprobando luces");
    verEstadoLuz();
    System.out.println("Estado luces comprobado correctamente");
    carga.setValue(9);
    carga.setLabel("Comprobando microondas");
    verEstadoMicroondas();
    System.out.println("Estado microondas comprobado correctamente");
    carga.setValue(12);
    carga.setLabel("Comprobando vídeo");
    verEstadoVideo();
    System.out.println("Estado video comprobado correctamente");
    carga.setValue(15);
    carga.setLabel("Estableciendo coordenadas");
    establecerCoordenadas();
    carga.setLabel("Carga Completa");
    try{
          t.sleep(1000);
    }catch (Exception e){
          System.out.println(e);
    }
    midlet.verOpciones();
  }
    // Fin comentario 7

  public void commandAction(Command c, Displayable d){
    if (c==cancelar){
          //
    }else if (c==aceptar){
```

```
            this.removeCommand(aceptar);
            this.addCommand(cancelar);
            this.start();
        }
    }

    // Comentario 8
    private void verEstadoAlarma(){
        boolean[] est;
        int numHab = midlet.edificio.getNumHabitaciones();
        est = new boolean[numHab];
        for (int i=0;i<numHab;i++)
            est[i] = false;
        midlet.edificio.setAlarma(est);
        midlet.setInfoAlarma2(est);
        try{
            t.sleep(1000);
        }catch(Exception e){
            System.out.println("Error en estado alarma");
        }
}

    private void verEstadoCalefaccion(){
        int[] est;
        int numHab = midlet.edificio.getNumHabitaciones();
        est = new int[numHab];
        for (int i=0;i<numHab;i++)
            est[i] = 0;
        midlet.edificio.establecerTemperatura(est);
        try{
            t.sleep(1000);
        }catch(Exception e){
            System.out.println("Error en estado calefaccion");
        }
    }

    private void verEstadoMicroondas(){
        boolean estado = false;
        midlet.edificio.setMicroondas(estado);
        try{
            t.sleep(1000);
        }catch(Exception e){
            System.out.println("Error en estado microondas");
        }
    }

    private void verEstadoVideo(){
        boolean estado = false;
        midlet.edificio.setVideo(estado);
        try{
            t.sleep(1000);
```

```
    }catch(Exception e){
        System.out.println("Error en estado video");
    }
}

public void verEstadoLuz(){
    numluces = 6;
    midlet.edificio.setNumLuces(numluces);
    boolean[] est = new boolean[6];
    for (int i=0;i<6;i++)
        est[i] = false;
    midlet.edificio.setEstadoLuz(est);
    midlet.setInfoLuz1(est);
}

public void establecerCoordenadas(){
    try{
        t.sleep(1000);
        int numHab = midlet.edificio.getNumHabitaciones();
        int[][] coord = new int[numHab][4];
        coord[0][0] = 9;coord[0][2] = 86;
        coord[0][1] = 12;coord[0][3] = 47;
        coord[1][0] = 98;coord[1][1] = 14;
        coord[1][2] = 77;coord[1][3] = 83;
        coord[2][0] = 11;coord[2][1] = 65;
        coord[2][2] = 84;coord[2][3] = 93;
        coord[3][0] = 100;coord[3][1] = 102;
        coord[3][2] = 73;coord[3][3] = 56;
        int[][] ady = new int[numHab][4];
        ady[0][0] = 0;ady[0][1] = 3;
        ady[0][2] = 2;ady[0][3] = 0;
        ady[1][0] = 0;ady[1][1] = 4;
        ady[1][2] = 0;ady[1][3] = 1;
        ady[2][0] = 1;ady[2][1] = 0;
        ady[2][2] = 4;ady[2][3] = 0;
        ady[3][0] = 2;ady[3][1] = 0;
        ady[3][2] = 0;ady[3][3] = 3;
        int[][] coordTemp = new int[numHab][2];
        coordTemp[0][0] = 45;coordTemp[0][1] = 44;
        coordTemp[1][0] = 128;coordTemp[1][1] = 80;
        coordTemp[2][0] = 47;coordTemp[2][1] = 72;
        coordTemp[3][0] = 130;coordTemp[3][1] = 106;
        int[][] coordluz = new int[numluces][2];
        coordluz[0][0] = 45;coordluz[0][1] = 36;
        coordluz[1][0] = 103;coordluz[1][1] = 20;
        coordluz[2][0] = 132;coordluz[2][1] = 49;
        coordluz[3][0] = 64;coordluz[3][1] = 73;
        coordluz[4][0] = 55;coordluz[4][1] = 105;
        coordluz[5][0] = 150;coordluz[5][1] = 116;
        midlet.edificio.establecerAdyacencia(ady);
        midlet.edificio.establecerCoord(coord);
```

```
            midlet.edificio.setCoordTemp(coordTemp);
            midlet.edificio.setCoordLuz(coordluz);
            midlet.setInfoLuz2(coordluz);
            midlet.setInfoAlarma1(coordTemp);
            coord = null;
            ady = null;
            coordluz = null;
            coordTemp = null;
        }catch (Exception e){
            System.out.println("Error al establecer coordenadas");
        }
    }
    // Fin comentario 8
}
```

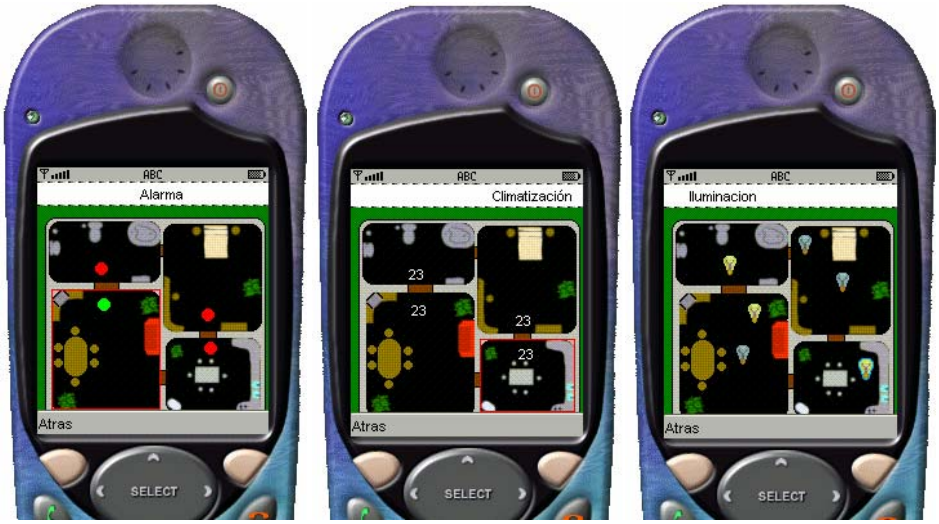

Figura 5.16 Visualización de Alarma, Climatización y Luces, respectivamente

La clase **CargaEstados** es la que se encarga de suministrar a la clase **Edificio** y a los distintos módulos la información sobre los estados de cada dispositivo y las coordenadas de cada elemento.

- Comentario 7

Este es el cuerpo principal de la clase **CargaEstados**. Aquí vemos el estado de cada dispositivo que inicializamos con un valor por defecto. A la vez que vamos inicializando estos valores vamos viendo el nivel de carga por pantalla y actualizamos la clase *Edificio* con estos valores.

- Comentario 8

Estos métodos se encargan de inicializar los estados de cada dispositivo como hemos dicho con anterioridad. Estos estados tienen un valor que nosotros le damos por defecto y en cada ejecución del *MIDlet* utilizará estos estados.

En la Figura 5.16 ver el aspecto que tienen las pantallas anteriores usando la interfaz de bajo nivel.

5.4.3. Conceptos básicos para la creación de juegos en MIDP

Uno de los primeros campos que ha hecho uso de la tecnología MIDP es el de los videojuegos. Desde hace unos años, la mayoría de los teléfonos móviles traen incorporados algún que otro juego, la mayoría de ellos bastante simples y, por supuesto en blanco y negro. Dado el 'boom' tecnológico que ha dado la telefonía móvil en estos últimos años, podemos encontrar actualmente en el mercado teléfonos con pantalla a color de tamaño considerable, teléfonos que soportan GPRS, etc. Con este panorama no es de extrañar que los teléfonos móviles se conviertan en dispositivos de ocio, además de simples dispositivos de comunicación. No podemos terminar este punto sin dejar un hueco para explicar las facilidades que nos proporciona MIDP para la creación de juegos.

En este apartado sólo veremos unos conceptos básicos que, unidos a lo que ya sabemos sobre las APIs de bajo nivel, nos permitirán crear nuestros propios juegos.

5.4.3.1. Eventos de teclado para juegos

Los eventos de bajo nivel son controlados en las pantallas Canvas a través de códigos. MIDP nos proporciona además de los códigos genéricos de teclado, un conjunto de constantes que se refieren a acciones específicas de un juego. Cada una de estas constantes está definidas según la Tabla 5.21.

Constante	Código
UP	1
DOWN	6
LEFT	2
RIGHT	5
FIRE	8
GAME_A	9
GAME_B	10
GAME_C	11
GAME_D	12

Tabla 5.21 Códigos de teclas para juegos

Dependiendo del dispositivo, cada uno de los códigos anteriores puede estar asignado a una tecla diferente. Pueden existir dispositivos MID que tengan botones especiales que hagan la función de movimiento y disparo: UP, DOWN, LEFT, RIGHT y FIRE, o puede que estas acciones estén asociadas a los botones 2, 8, 4, 6 y 5 respectivamente. En cualquier caso, el programador no tiene por qué conocer a que tecla específica está asociada una acción en concreto, lo que nos facilita bastante el trabajo.

MIDP nos proporciona algunos métodos que nos permiten realizar conversiones entre los códigos generales de teclado (keyCodes) y los códigos de juegos (ver Tabla 5.22).

Métodos	Descripción
int getKeyCode(int gameAction)	Devuelve el código genérico asociado *gameAction*.
int getGameAction(int keyCode)	Devuelve si existe el código de juego asociado a *keyCode*.
string getKeyName(int keyCode)	Obtiene el nombre del *keyCode*.

Tabla 5.22 Métodos para obtener los códigos de juegos

Los métodos proporcionados por Canvas para controlar los eventos de bajo nivel keyPressed(int keyCode), keyRepeated(int keyCode) y keyReleased(int keyCode) trabajan con códigos genéricos de teclado (keyCodes). Para poder trabajar con códigos de juegos tenemos que usar el método getGameAction(int keyCode) dentro de cada uno de los métodos anteriores:

```
protected void keyPressed(int codigo){
        switch (getGameAction(codigo)){
                case Canvas.FIRE: disparar(); break;
                case Canvas.UP: moverArriba();break;
                case Canvas.DOWN: moverAbajo();break;
                        ...
        }
}
```

Podemos también inicializar en variables cada código de juego y luego utilizar estas variables en los métodos anteriores:

```
// En el constructor cuando se inicializan las variables
…
disparo = getKeyCode(Canvas.FIRE);
arriba = getKeyCode(Canvas.UP);
abajo = getKeyCode(Canvas.DOWN);
…
protected void keyPressed(int codigo){
        if (codigo == disparo) disparar();
        else if (codigo == arriba) moverArriba();
        else if (codigo == abajo) moverAbajo();
                ...
}
```

5.4.3.2. El paquete javax.microedition.lcdui.Game

Este paquete contiene un determinado número de clases que nos van a proporcionar una gran ayuda a la hora de crear juegos. Dado que estamos programando aplicaciones orientadas a dispositivos con ciertas restricciones computacionales, estas clases básicamente lo que pretenden es aumentar el rendimiento de nuestra aplicación, minimizando en lo posible la carga de trabajo del programador y así, reducir el tamaño final de la aplicación.

Estas clases que vamos a ver a continuación pueden usarse conjuntamente con las clases gráficas de bajo nivel que hemos estudiado anteriormente (Graphics, Image, …) y deben entenderse como un complemento a éstas para ayudarnos a crear juegos.

Este paquete se compone de 5 clases de las que veremos solamente una breve descripción de su función. Para más información sobre sus campos y métodos se puede consultar la especificación MIDP 2.0 que se encuentra en la página oficial de Sun (http://java.sun.com).

- **Clase GameCanvas**

La clase GameCanvas es una clase abstracta que hereda de la clase Canvas y representa una pantalla básica para desarrollar un juego. Además de los métodos que posee Canvas, esta clase proporciona características especiales para juegos que nos permitirán por ejemplo, preguntar por el estado actual de las teclas del juego, además de sincronizar la aparición de gráficos por pantalla. Veamos como sería un bucle típico de un juego usando algunos métodos que nos proporciona GameCanvas:

```
Graphics g = getGraphics();
        while (true){
        int estadoteclas = getKeyStates();
        if ((estadoteclas & LEFT_PRESSED) != 0){
                sprite.move(-1,0);
        }else if ((estadoteclas & RIGHT_PRESSED) != 0){
                sprite.move(1,0);
        }else if ((estadoteclas & UP_PRESSED) != 0){
                sprite.move(0,-1);
        }else if((estadoteclas & DOWN_PRESSED) != 0){
                sprite.move(0,1);
        }else if((estadoteclas & FIRE_PRESSED) != 0){
                disparar();
        }
        g.setColor(255,255,255);
        g.fillRect(0,0,getWidth(),getHeight());
        sprite.paint(g);
        flushGraphics();
}
```

- **Clase Layer**

La clase Layer (capa) representa un elemento visual en un juego. Este elemento puede ser un objeto Sprite o TiledLayer. La clase Layer es una clase abstracta que se usa como clase básica para trabajar con capas (de gráficos) y proporciona atributos a estos elementos tales como localización, tamaño y visibilidad.

Los métodos de la clase Layer nos permitirán trabajar con los atributos anteriormente nombrados. Cualquier clase que creemos que herede de Layer debe implementar obligatoriamente el método paint(Graphics).

- **Clase LayerManager**

Esta clase nos facilitará el trabajar con varios objetos de la clase Layer automatizando el proceso de dibujo de cada una de las capas. Además, esta clase nos permitirá crear vistas que representen la vista del usuario en los juegos.

El manejo de las distintas capas se realiza de la siguiente manera: esta clase mantiene una lista ordenada de objetos Layer con los que va a trabajar, permitiendo acciones cómo insertar o borrar capas de la lista. Podemos acceder a cada una de las capas mediante un índice que puede recorrer la lista. Los valores más cercanos al cero representan las capas que están más cerca de la vista del usuario, mientras que los valores más cercanos al final de la lista representan a las capas más lejanas de la vista del usuario. El dibujo de cada capa se realiza de forma automática lo que nos quita una gran cantidad de trabajo a los programadores.

- **Clase Sprite**

La clase Sprite representa a un Layer animado. Este Sprite suele estar formado por varios fotogramas gráficos que pueden representar desde una animación del movimiento del Sprite en pantalla a distintos puntos de vista de éste. Estos fotogramas tienen que tener el mismo tamaño y estar representados por objetos Image.

La clase Sprite contiene métodos que nos permitirán realizar transformaciones cómo rotaciones y detección de colisiones.

Piénsese en un juego dónde controlemos a un caballero que debe ir avanzando por pantallas dónde se encuentra con multitud de enemigos. Tanto el caballero cómo los enemigos estarían representados por un objeto Sprite. Estos objetos ya tendrían la información suficiente para realizar las animaciones de los movimientos de los personajes y dispondríamos de métodos que nos darían la posición de cada uno de ellos y evitaríamos de esta manera, que dos personajes ocuparan la misma posición en pantalla (detección de colisiones).

- **Clase TiledLayer.**

Esta clase permitirá crear grandes áreas de contenido gráfico sin el problema que nos podría ocasionar tener una imagen de grandes dimensiones en memoria. Esto es

posible debido a que un objeto TiledLayer está compuesto por una malla de celdas, las cuales pueden ser rellenadas con partes de una imagen.

5.4.3.3. Técnicas útiles

Para terminar este apartado dedicado a los juegos en MIDP veremos un par de técnicas que nos permitirán optimizar el código de nuestro juego.

5.4.3.3.1. *Double Buffering*

La técnica de pantalla en segundo plano consiste en lo siguiente: Muchos dispositivos que no poseen una gran capacidad computacional provocan un parpadeo a la hora de actualizar los gráficos en pantalla. En estos casos, lo ideal sería hacer todas las actualizaciones gráficas en memoria y posteriormente volcar el contenido de memoria en pantalla. Esta técnica es la que se conoce como *double buffering.* Algunos de estos dispositivos implementan esta técnica sin que tengamos que escribir ni una sola línea de código. Para saber si el MID donde ejecutamos nuestra aplicación aplica esta técnica, existe un método de la clase Canvas que nos devuelve true o false en caso de que aplique esta técnica o no: boolean isDoubleBuffered().

En caso de que el método nos devuelva true, no tenemos que hacer absolutamente nada especial ya que la implementación del MID se encarga de hacer todas las actualizaciones gráficas en memoria para posteriormente, volcarlo en pantalla.

En el caso de que el método retorne false, nos indicaría que el dispositivo no es capaz de implementar esta técnica. En este caso es posible implementarla por nosotros mismos. Para ello, lo que haremos será crear una imagen mutable del tamaño de la pantalla del MID y efectuar todas las operaciones gráficas en el objeto Image para, posteriormente volcar el contenido de esta imagen por pantalla.

1.- Crear la imagen mutable del tamaño de la pantalla.

```
Image displaysecundario;
If (! isDoubleBuffered()){
    Displaysecundario = Image.createImage(getWidth(),getHeight());
}
```

2.- Realizar todas las actualizaciones en el objeto Image dentro del método paint(Graphics).

```
protected void paint(Graphics g){
    Graphics displayprimario = g;
    g = displaysecundario.getGraphics();
   // Realizamos todas las operaciones para actualizar la pantalla en g.
    ...
    //Realizamos el volcado en pantalla
    displayprimario.drawImage(displaysecundario,
                                0,0,Graphics.LEFT | Graphics.TOP);
}
```

5.4.3.3.2. *Clipping*

Como hemos visto anteriormente, el hecho de actualizar los gráficos en pantalla puede causar algún problema en algunos dispositivos, si éste no dispone de doble *buffer*. Incluso implementando el d*ouble buffering* por nosotros mismos no aseguramos que esa actualización se produzca correctamente ya que algunos dispositivos no son suficientemente rápidos en las operaciones de lectura de memoria. Sería entonces interesante a la hora de actualizar una pantalla volver a dibujar sólo la parte que haya que modificarse dejando el resto en el mismo estado. Cada vez que llamamos al método paint(Graphics) dibujamos completamente la pantalla. Podemos usar entonces la técnica de *clipping* para definir una región que limite lo que se va a pintar de nuevo por pantalla en la llamada al método paint():

```
void setClip(int x, int y, int ancho, int alto)
```

Esta llamada formará un rectángulo imaginario en pantalla que comenzará en el punto (x,y) y tendrá una determinada anchura y altura que definirá la zona que se verá afectada tras las operaciones gráficas. La Tabla 5.23 muestra otros métodos de la clase Graphics relacionados con el *clipping*.

Sólo puede existir una zona de *Clip* por cada objeto Graphics. Cualquier operación que realicemos en este objeto y que esté fuera de la zona de *clipping* no tendrá ningún efecto sobre la pantalla.

Como vemos, esta técnica nos permite realizar optimizaciones en el código ya que reduce la carga computacional relacionada con las operaciones gráficas al reducir a un rectángulo la zona de dibujo, en vez de dibujar la pantalla completa.

Método	Descripción
void clipRect(int x, int y, int ancho, int alto)	Une el rectángulo definido a la región de *Clipping*.
int getClipX()	Devuelve la coordenada X.
int getClipY()	Devuelve la coordenada Y.
int getClipHeight()	Devuelve la altura.
int getClipWidth()	Devuelve la anchura.

Tabla 5.23 Métodos relacionados con el *Clipping*

5.4.4. Diseño de un juego usando la clase javax.microedition.lcdui.game

En el punto anterior hemos dado los conceptos necesarios para poder crear casi cualquier tipo de juego. A partir de aquí, la imaginación es lo que cuenta ya que disponemos de las herramientas adecuadas para crear lo que se nos ocurra. De todas formas, vamos a crear un pequeño videojuego que haga uso de las clases anteriormente

explicadas y que puede servir de guía para que el lector cree su propio juego partiendo desde un ejemplo práctico y operativo.

En este caso, vamos a desarrollar el juego del PingPong. La mecánica del juego es bien sencilla. Nosotros manejamos una especie de raqueta que debe golpear una pelota para marcar un tanto en el lado contrario del terreno de juego. Como contrincante tenemos a otra raqueta controlada por la máquina y que intentará hacer lo mismo que nosotros para marcarse ella el tanto. El primero de los dos que llegue a 10 tantos gana el partido.

A continuación veremos el código que forma nuestro juego. La clase principal es la clase **PingPong**. Esta clase hereda de *MIDlet* y es la encargada de crear y manejar todos los elementos que forman nuestro juego.

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

public class PingPong extends MIDlet {

   private Display pantalla;
   private Menu m;
   private Tablero t;
   private Resultado r;
   private Dificultad d;
   private int nivel = 1;
   private NuevoRecord nr;

   public PingPong(){
      pantalla = Display.getDisplay(this);
      m = new Menu(this);
      t = new Tablero(this);
      r = new Resultado(this);
      d = new Dificultad(this);
      nr = new NuevoRecord(this);
   }
   public void startApp() {
      pantalla.setCurrent(m);
   }
   public void pauseApp() {
   }
public void destroyApp(boolean unconditional) {
      pantalla = null;
      m = null;
      t = null;
      r = null;
      d = null;
      nr = null;
   }
   public void salir(){
      destroyApp(false);
      notifyDestroyed();
```

```
    }
    public void setTablero(){
      pantalla.setCurrent(t);
    }
public void setDificultad(){
      pantalla.setCurrent(d);
    }
    public void setResultado(){
      pantalla.setCurrent(r);
    }
    public void setMenu(){
      pantalla.setCurrent(m);
    }
    public void setNivel(int n){
      nivel = n+1;
    }
    public int getNivel(){
      return nivel;
    }
    public void iniciarTablero(){
      t.inicializar();
    }
    public void cargarResultado(){
      r.verResultado();
    }
    public boolean esRecord(long punt){
      boolean record = r.esRecord(punt);
      return record;
    }
    public void setRecord(long punt){
      nr.setPuntuacion(punt);
      pantalla.setCurrent(nr);
    }
    public void insertarRecord(String nombre, long puntuacion){
      r.insertarRegistro(nombre,puntuacion);
    }
}
```

La clase **Menu** hereda de la clase List y únicamente nos muestra el menú de opciones disponibles. No suele ser buena idea llamar "Menu" a una clase de nuestro programa, ya que ello puede dar lugar a conflictos (java.awt.Menu). No obstante, dado que en nuestro caso no se producen, hemos optado por este nombre.

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

public class Menu extends List implements CommandListener{

   private PingPong midlet;
   private Command salir;
```

```
    public Menu(PingPong m){
        super("Menu",List.IMPLICIT);
        this.append("Jugar",null);
        this.append("Dificultad",null);
        this.append("Resultados",null);
        this.midlet = m;
        salir = new Command("Salir",Command.EXIT,1);
        this.addCommand(salir);
        this.setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d){
        if (c == List.SELECT_COMMAND){
            switch(this.getSelectedIndex()){
                case 0:{
                        midlet.iniciarTablero();
                        midlet.setTablero();
                        break;
                    }
                case 1:{
                        midlet.setDificultad();
                        break;
                    }
                case 2:{
                        midlet.cargarResultado();
                        midlet.setResultado();
                        break;
                    }
            }
        }else{
            midlet.salir();
        }
    }
}
```

La clase **Dificultad** que veremos a continuación también hereda de la clase List (al igual que **Menu**) y nos permite seleccionar el nivel de dificultad.

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

public class Dificultad extends List implements CommandListener{

    private PingPong midlet;
    private Command atras, aceptar;

    public Dificultad(PingPong m){
        super("Dificultad",List.EXCLUSIVE);
        this.append("Baja",null);
        this.append("Normal",null);
        this.append("Alta",null);
```

```
      this.setSelectedIndex(0,true);
      midlet = m;
      atras = new Command("Atras",Command.BACK,1);
      aceptar = new Command("Aceptar",Command.OK,1);
      this.addCommand(aceptar);
      this.addCommand(atras);
      this.setCommandListener(this);
   }

   public void commandAction(Command c, Displayable d){
      if (c == aceptar){
            midlet.setNivel(this.getSelectedIndex());
            System.out.println("Nivel "+this.getSelectedIndex());
            midlet.setMenu();
      }
   }
}
```

La clase **NuevoRecord** permite al jugador introducir su nombre una vez que haya conseguido un nuevo record.

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

public class NuevoRecord extends Form implements CommandListener{

   private TextField txtNombre;
   private Command aceptar;
   private PingPong midlet;
   private long puntuacion;

   public NuevoRecord(PingPong m){
      super("Nuevo Record");
      midlet = m;
      puntuacion = 0;
      txtNombre = new TextField("Nombre","",3,TextField.ANY);
      aceptar = new Command("Aceptar",Command.OK,1);
      this.append(txtNombre);
      this.addCommand(aceptar);
      this.setCommandListener(this);
   }

   public void setPuntuacion(long punt){
      puntuacion = punt;
   }

   public void commandAction(Command c, Displayable d){
      midlet.insertarRecord(txtNombre.getString(),puntuacion);
      midlet.cargarResultado();
      midlet.setResultado();
   }
```

```
}
```

La clase **Tablero** es, sin duda, la más importante de este ejemplo. Con ella se desarrolla toda la ejecución del juego en sí. Como veremos, el método run() contiene el cuerpo principal de ejecución. Si prestamos un poco de atención podemos ver el típico bucle que posee un juego y que ya vimos anteriormente. Además, esta clase posee los siguientes métodos:

- crearFondo(): Nos crea el fondo de pantalla.
- crearPelota(): Nos crea el objeto pelota.
- crearPaleta(): Nos crea el objeto paleta.
- inicializar(): Nos inicializa los elementos del juego.
- start(): Comenzamos el juego.
- run(): Posee el cuerpo principal del juego.
- verColisiones(): Detecta cualquier colisión que produzca la pelota y actúa en consecuencia.
- calcularTrayectoria(Paleta pal): Calcula la nueva trayectoria de la pelota tras ocurrir una colisión.
- verEntrada(): Aquí se detecta cualquier pulsación de teclas que efectúe el usuario.
- moverPelota(): Movemos la pelota según la dirección que posea.
- dibujar(Graphics g): Dibujamos todos los elementos por pantalla.
- mostrarResultados(int r1, int r2): Nos muestra el resultado por pantalla tras un tanto.

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;
import java.lang.Math.*;

public class Tablero extends GameCanvas
implements CommandListener, Runnable{

   private PingPong midlet;
   private Command atras, comenzar, seguir, pausa, continuar;
   private Pelota p;
   private Paleta jug, maq;
   private boolean ejecucion, pausar;
   private Image fondo;
   private Thread t;
   private int velocidad, nivel, res1, res2;

   public Tablero(PingPong m){
      super(true);
      midlet = m;
      velocidad = 50;
      res1 = 0;
      res2 = 0;
      atras = new Command("Atras",Command.BACK,1);
      comenzar = new Command("Comenzar",Command.OK,1);
```

```
    seguir = new Command("Seguir",Command.OK,1);
    pausa = new Command("Pausa",Command.OK,1);
    continuar = new Command("Continuar",Command.OK,1);
    this.addCommand(atras);
    this.addCommand(comenzar);
    this.setCommandListener(this);
    p = crearPelota();
    jug = crearPaleta();
    maq = crearPaleta();
    crearFondo();
}

public void crearFondo(){
    fondo = Image.createImage(getWidth(),getHeight());
    Graphics gf = fondo.getGraphics();
    gf.setColor(114,165,154);
    gf.fillRect(0,0,getWidth(),getHeight());
}

public Pelota crearPelota(){
    try{
        Image im = Image.createImage("/pelota.png");
        return (new Pelota(im));
    }catch (Exception e){
        System.out.println("No se pudo abrir pelota.png");
        return null;
    }
}

public Paleta crearPaleta(){
    try{
        Image im = Image.createImage("/pala.png");
        return (new Paleta(im));
    }catch (Exception e){
        System.out.println("No se pudo abrir pala.png");
        return null;
    }
}

public void inicializar(){
    p.setPosition((getWidth()/2)-5,(getHeight()/2)-5);
    p.setDireccion(-1,-1);
    jug.setPosition(5,(getHeight()/2)-25);
    maq.setPosition(165,(getHeight()/2)-25);
    nivel = midlet.getNivel();
    Graphics g = getGraphics();
    dibujar(g);
}

public void start(){
    t = new Thread(this);
```

```
        ejecucion = true;
        pausar = false;
        t.start();
    }

    public void run(){
        Graphics g = getGraphics();
        while (ejecucion)
            while (!pausar){
                verColisiones();
                verEntrada();
                moverPelota();
                dibujar(g);
                try{
                    Thread.sleep(velocidad);
                }catch (Exception e){
                    System.out.println("Error de ejecucion");
                }
            }
    }

    public void verColisiones(){
        if ((p.getX() <= 0)||(p.getX()+10 >= getWidth())) {
            this.removeCommand(pausa);
            ejecucion = false;
            if (p.getX() <= 0) res2++;
            else res1++;
            mostrarResultado(res1,res2);
            if (res2 == 10){
                t = null;
                res2 = 0;res1 = 0;
                midlet.setMenu();
            }else if (res1 == 10){
                t = null;
                if (midlet.esRecord(res1-res2)){
                    midlet.setRecord(res1-res2);
                    res1 = 0;res2 = 0;
                }else{
                    res1 = 0;res2 = 0;
                    midlet.setMenu();
                }
            }
        }else if (p.collidesWith(jug,true)){
            p.setPosition(jug.getX()+10,p.getY());
            calcularTrayectoria(jug);
        }else if (p.collidesWith(maq,true)){
            p.setPosition(maq.getX()-10,p.getY());
            calcularTrayectoria(maq);
        }else if (p.getY() <= 0){
            p.setPosition(p.getX(),0);
            p.setDireccion(p.getDirX(),Math.abs(p.getDirY()));
```

```
    }else if (p.getY() >= getHeight()-10){
        p.setPosition(p.getX(),getHeight()-10);
        p.setDireccion(p.getDirX(),p.getDirY()-(2*(p.getDirY())));
    }
}

public void calcularTrayectoria(Paleta pal){
    int ypal = pal.getY();
    int yp = p.getY();
    int dy = yp-ypal;
    int signo;
    if (p.getDirX() < 0) signo = 1;
    else signo = -1;
    if ((dy >= 20)&&(dy <= 30)) p.setDireccion(signo*p.getDirX(),p.getDirY());
    else if ((dy >= 15)&&(dy < 20)) p.setDireccion(signo*3,-1);
    else if ((dy >= 10)&&(dy < 15)) p.setDireccion(signo*2,-2);
    else if ((dy >= -10)&&(dy < 10)) p.setDireccion(signo,-3);
    else if ((dy > 30)&&(dy <=35)) p.setDireccion(signo*3,1);
    else if ((dy > 35)&&(dy <=40)) p.setDireccion(signo*2,2);
    else if ((dy > 40)&&(dy <=50)) p.setDireccion(signo,3);
}

public void verEntrada(){
    int keyStates = getKeyStates();
    if ((keyStates & UP_PRESSED) != 0){
        if (jug.getY() >= 3) jug.move(0,-3);
        else jug.move(0,0-jug.getY());
    }else if ((keyStates & DOWN_PRESSED) != 0){
        if ((jug.getY()+50) <= (getHeight()-3)) jug.move(0,3);
        else jug.move(0,getHeight()-(jug.getY()+50));
    }
}

public void moverPelota(){
    p.move(p.getDirX()*nivel,p.getDirY()*nivel);
    maq.move(0,p.getDirY());
    if (maq.getY() < 0)
        maq.setPosition(maq.getX(),0);
    else if ((maq.getY()+50) > getHeight())
        maq.setPosition(maq.getX(),getHeight()-50);
}

public void dibujar(Graphics g){
    g.drawImage(fondo,0,0,Graphics.LEFT|Graphics.TOP);
    g.setColor(0,0,0);
    p.paint(g);
    jug.paint(g);
    maq.paint(g);
    flushGraphics();
}
```

```
    public void mostrarResultado(int r1, int r2){
        Graphics g = getGraphics();
        Font fuente =
              Font.getFont(Font.FACE_SYSTEM,Font.STYLE_PLAIN,Font.SIZE_LARGE);
        g.setFont(fuente);
        g.drawString(""+r1,getWidth()/4,getHeight()/2,
                        Graphics.BASELINE|Graphics.HCENTER);
        g.drawString(""+r2,3*getWidth()/4,getHeight()/2,
                        Graphics.BASELINE|Graphics.HCENTER);
        flushGraphics();
        try{
              t.sleep(3000);
              this.addCommand(seguir);
              pausar = true;
        }catch (Exception e){
              System.out.println(e);
        }
    }

    public void commandAction(Command c, Displayable d){
        if (c == comenzar){
              this.removeCommand(comenzar);
              this.removeCommand(atras);
              this.addCommand(pausa);
              start();
        }else if (c == atras){
              ejecucion = false;
              t = null;
              this.removeCommand(continuar);
              this.addCommand(comenzar);
              midlet.setMenu();
        }else if (c == seguir){
              inicializar();
              this.removeCommand(seguir);
              this.addCommand(pausa);
              t = null;
              start();
        }else if (c==pausa){
              pausar = true;
              this.removeCommand(pausa);
              this.addCommand(continuar);
              this.addCommand(atras);
        }else if (c==continuar){
              pausar = false;
              this.removeCommand(continuar);
              this.removeCommand(atras);
              this.addCommand(pausa);
        }
    }
}
```

La clase **Pelota** hereda de Sprite y contiene la imagen de la pelota y una dirección determinada.

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

public class Pelota extends Sprite {
   private int dx, dy;
   public Pelota(Image im){
      super(im);
      dx = 0; //Parada
      dy = 0;
   }
   public int getDirX(){
      return dx;
   }
   public int getDirY(){
      return dy;
   }
   public void setDireccion(int x, int y){
      dx = x;
      dy = y;
   }
}
```

Al igual que **Pelota**, la clase **Paleta** también hereda de Sprite y tan sólo contiene la imagen que forma la raqueta.

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

public class Paleta extends Sprite {
   public Paleta(Image im){
      super(im);
   }
}
```

5.5. Internacionalización

Los desarrolladores de aplicaciones MIDP deben conocer las herramientas que nos proporciona Java para emigrar programas escritos en un determinado idioma a otro diferente. Las características lingüísticas, geográficas y culturales de un determinado idioma son aspectos que debemos tener en cuenta al desarrollar un MIDlet. Estas características forman lo que denominados *internacionalización*. En vez de este término se usa normalmente *i18n*, ya que la palabra internacionalización posee 18 letras entre la i y la n.

Para poder llevar a cabo este proceso, Java hace uso de los llamados *Locale* que es una clase que almacena información sobre un determinado idioma.

5.5.1. Aspectos a internacionalizar

La internacionalización involucra a varias partes del desarrollo del software:

- Mensajes: Aquí entra todo lo relacionado con la presentación del texto visible al usuario. Mensajes de error, de alerta, cadenas de texto, etc.
- Formato de datos: Hay que tener en cuenta el formato de datos que usa un determinado país. Dentro de esta área forman parte el formato de fechas, horas, numeración y valores monetarios.
- Calendarios y zonas horarias: Hay que usar un calendario que se adecue a la localidad donde se ejecute la aplicación.
- Aspectos relacionados con la UI de la aplicación: En este punto hay que tener en cuenta los iconos, imágenes, colores y elementos visuales que usemos en nuestra aplicación.

5.5.2. Limitaciones

Como es bien sabido, los dispositivos MID poseen bastantes restricciones. En este caso, como en la mayoría, solo poseemos un subconjunto de las clases orientadas a la internacionalización que posee J2SE. Veamos cuales son las limitaciones con las que nos vamos a encontrar:

- En primer lugar, la plataforma MIDP no proporciona soporte para la internacionalización de mensajes. MIDP no incluye la clase MessageFormat de J2SE con lo que el formato de mensajes debe ser implementado por el programador.
- Tampoco se incluye el soporte necesario para dar formato a las fechas, horas y números ya que se han excluido las clases DateFormat, NumberFormat y DecimalFormat.
- Tampoco se soporta ningún tipo de ordenación de cadenas. Si nuestra aplicación necesita algún tipo de ordenación lexicográfica, ésta debe ser proporcionada por el programador.
- Por último nos encontramos con ciertas restricciones para el soporte de calendarios. La clase java.util.Calendar es abstracta, pero la plataforma MIDP nos asegura al menos una implementación concreta.

5.5.3. Soluciones para la internacionalización

Para lograr la internacionalización de aplicaciones basadas en MIDP 2.0 podemos poner en práctica alguna de las siguientes aproximaciones:

- Uso de atributos en el fichero descriptor del MIDlet.
- Uso de ficheros para definir los recursos de localización.

5.5.3.1. Uso de Atributos en el JAD

En el capítulo 2 vimos que el usuario podía definir atributos en el archivo descriptor JAD. Pues bien, nuestra primera solución para la internacionalización será hacer uso de estos atributos para definir recursos de localización que usará nuestro MIDlet.

Una vez que se definan todos los atributos que se vayan a utilzar podemos recuperarlos durante la ejecución del MIDlet a través del método getAppProperty(String).

Un archivo JAD que defina estos recursos puede tener la siguiente apariencia:

```
PruebaInt-saludo-es-Es: ¡Hola!
PruebaInt-saludo-en-En: Hello!
PruebaInt-salir-es-Es: Salir
PruebaInt-salir-en-En: Exit
Midlet-Name: PruebaInt
…
```

Como vemos, los atributos que definimos tienen la forma:

<NombreMidlet>-<clave>-<localización>

Una vez definidos todos los atributos y sabiendo que podemos recuperarlos durante la ejecución del MIDlet, lo único que tenemos que hacer es formar el nombre del atributo y recuperar su valor. Para ello en el método startApp() haremos lo siguiente:

```
String locale = System.getProperty("microedition.locale");
String tituloform = getAtributo("saludo");
Form formPrincipal = new Form(tituloform);
String titulosalir = getAtributo("salir");
Command salir = new Command(titulosalir,Command.EXIT,1);
formPrincipal.addCommand(salir);
```

El método getAtributo(String) es el que se encarga de recuperar el valor del atributo requerido:

```
public String getAtributo(String s){
    StringBuffer valor = new StringBuffer("PruebaInt-");
    valor.append(s);
    valor.append("-");
    valor.append(locale);
    String atr = getAppProperty(valor.toString());
    return atr;
}
```

Este método nos resuelve algunos de los problemas que nos plantea la internacionalización, pero posee algunos problemas y restricciones:

1. En primer lugar, tendríamos que tener definidos atributos para todos los idiomas existentes si queremos que nuestro MIDlet sea totalmente portable. Esto supone un gran problema en el momento que queramos soportar más de 3 o 4 idiomas ya que el archivo JAD alcanzaría un gran tamaño.
2. Además, estamos usando el archivo JAD para algo que, por definición, no es su cometido. Este archivo define características del MIDlet usando para ello unos atributos ya definidos por la implementación. Por tanto, el acceso a este archivo debería estar limitado tan solo al AMS para obtener información sobre el MIDlet cuando la necesite.
3. Por último, el uso de esta técnica no nos resuelve el problema del formato de datos de nuestro MIDlet.

5.5.3.2. Uso de ficheros con recursos de localización

Esta solución evita el uso del archivo JAD para definir atributos que representen recursos de localización. Esta técnica consiste en hacer uso de ficheros de texto para definir estos atributos. Por ejemplo, podríamos crear un fichero con el nombre es-ES.txt para definir los recursos del idioma español y un fichero con el nombre en-US.txt para definir el inglés (ver Figura 5.17).

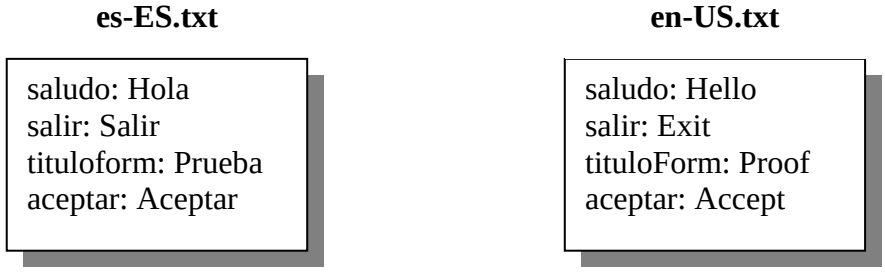

Figura 5.17 Ficheros de recursos

Estos ficheros estarían ubicados en el archivo JAR y podríamos acceder a ellos a través del método getResourceAsStream(String) que devuelve un InputStream asociado al fichero que le pasemos como parámetro al método anterior. Es misión del programador acceder al fichero de recursos correcto y tratar correctamente la información que contiene.

Como vemos, este método no aporta nada nuevo al anterior excepto que aquí ya no se hace un uso indebido del archivo JAD. Es más, aumentamos el tamaño de nuestra aplicación al incluir en el archivo JAR los ficheros de recursos de cada idioma soportado e incrementamos complejidad a la hora de programar el *MIDlet* al tener que acceder a un fichero y recuperar de él la información buscada.

La solución más elegante y a su vez, más compleja sería implementar por nosotros mismos las clases ResourceBundle y ListResourceBundle. Estas clases

están definidas en el perfil MIDP como abstractas, por lo que sería misión nuestra reescribirlas o hacer unas nuevas a la medida de nuestras necesidades.

Capítulo 6:
Record Management System

6.1. Descripción del capítulo

Hasta ahora nuestras aplicaciones perdían toda la información tras su ejecución. Es cierto también, que no hemos necesitado guardar ningún tipo de datos, pero piénsese en la realización de, por ejemplo, una agenda telefónica en la que guardaríamos los números de teléfonos y direcciones de unas determinadas personas. Aquí se haría imprescindible el poder guardar toda la información introducida (nombres, teléfonos y direcciones) en el dispositivo MID para posteriormente poder consultarla o modificarla.

Pues bien, MIDP proporciona un mecanismo a los MIDlets que les permite almacenar datos de forma persistente para su futura recuperación. Este mecanismo está implementado sobre una base de datos basada en registros que llamaremos *Record Management System* o RMS (Sistema de gestión de registros).

Comenzaremos viendo en este capítulo conceptos básicos sobre cómo se realiza el almacenamiento persistente o sobre cómo es el modelo de datos usado por el RMS. Veremos qué operaciones básicas podemos realizar a través de la clase RecordStore y también qué otras operaciones más avanzadas podemos realizar con el resto de clases que están dentro del paquete javax.microedition.rms.

6.2. Conceptos Básicos

El sistema de gestión de registros o RMS, como lo llamaremos de ahora en adelante, nos permite almacenar información entre cada ejecución de nuestro MIDlet.

Esta información será guardada en el dispositivo en una zona de memoria dedicada para este propósito. La cantidad de memoria y la zona asignada para ello dependerá de cada dispositivo.

6.2.1. Modelo de datos

Como ya se ha dicho, el RMS está implementado en una base de datos basada en registros (ver Figura 6.1).

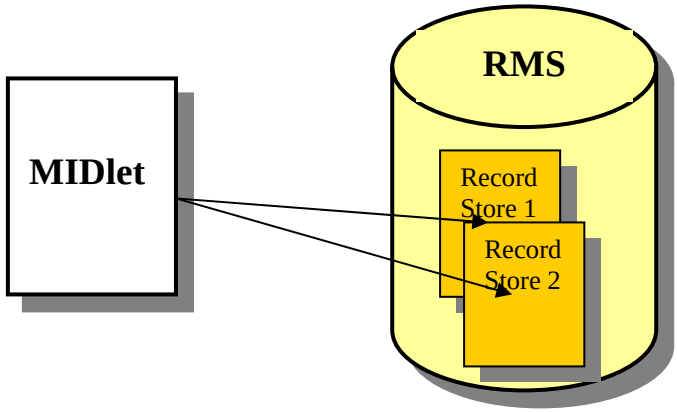

Figura 6.1 Comunicación entre un MIDlet y el RMS

Los *MIDlets* son los encargados de crear los *Record Stores* para comunicarse con ellos. Estos *Record Stores* quedan almacenados en el dispositivo y pueden ser accedidos por cualquier *MIDlet* que pertenezca a la misma *suite* (ver Figura 6.2).

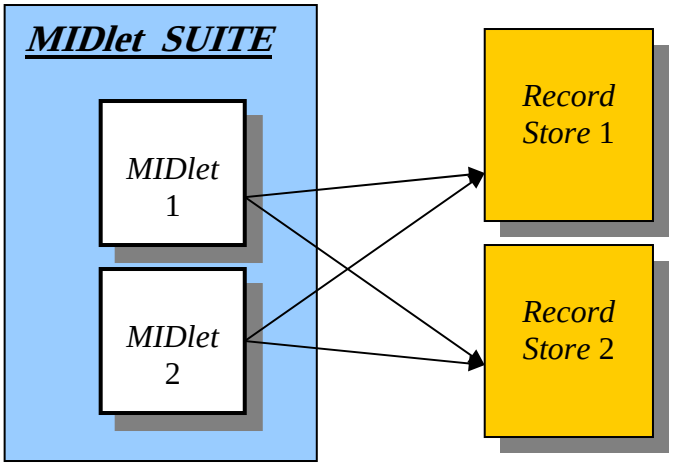

Figura 6.2 Acceso a un RMS a través de una *MIDlet suite*

6.2.2. *Record Stores*

Las propiedades de estos almacenes de registros son:

1. Cada *Record Store* está compuesto por cero o más registros.
2. Un nombre de *Record Store* es sensible a mayúsculas y minúsculas y está formado por un máximo de 32 caracteres UNICODE.
3. Dentro de una *suite* no pueden coexistir dos *Record Stores* con el mismo nombre.
4. Si una *suite* de *MIDlets* es borrada del dispositivo MID, todos los *Record Stores* pertenecientes a esa *suite* se borrarán.
5. Es posible que un *MIDlet* acceda a un *Record Store* creado por otra *suite*, siempre que ésta de permiso para ello.

Un *Record Store* tal como su nombre indica es un almacén de registros. Estos registros son la unidad básica de información que utiliza la clase RecordStore para almacenar datos.

Cada uno de estos registros está formado por dos unidades:

- Un número identificador de registro (*Record* ID) que es un valor entero que realiza la función de clave primaria en la base de datos.
- Un *array* de bytes que es utilizado para almacenar la información deseada.

En la Figura 6.3 se ilustra la estructura de un *Record Store*:

Record Store	
***Record* ID**	**Datos**
1	byte [] arrayDatos
2	byte [] arrayDatos
…	…

Figura 6.3 Estructura de un *Record Store*

Además de un nombre, cada *Record Store* también posee otros dos atributos:

- Número de versión: Es un valor entero que se actualiza conforme vayamos insertando, modificando o borrando registros en el *Record Store*. Podemos consultar este valor invocando al método RecordStore.getVersion().
- Marca temporal: Es un entero de tipo long que representa el número de milisegundos desde el 1 de enero de 1970 hasta el momento de realizar la última modificación en el *Record Store*. Este valor lo podemos obtener invocando al método RecordStore.getLastModified().

Así, la estructura de un *Record Store* se aproxima más a la Figura 6.4.

Nombre: Record Store 1
Version: 1.0
TimeStamp: 2909884894049
Registros:

Record ID	Datos
1	byte [] arrayDatos
2	byte [] arrayDatos
…	…

Figura 6.4 Estructura completa de un *Record Store*

6.3. Operaciones con *Record Stores*

Una vez vista la teoría, pasemos a la práctica. Ya sabemos qué es un *Record* Store y como está formado. En este punto veremos la clase javax.microedition.rms.RecordStore y todas las operaciones que nos permitan realizar sus métodos.

6.3.1. Creación de un *Record Store*

La clase RecordStore no dispone de ningún constructor, pero posee el método estático:

static RecordStore **openRecordStore**(String name, boolean createIfNeccesary)

Este método nos abre el *Record Store* con el nombre pasado como parámetro o nos crea uno si no existe cuando el parámetro createIfNeccesary es *true*. Además, existen otrasdos versiones alternativas de este método:

- static RecordStore **openRecordStore**(String name, boolean createIfNeccesary, int autorización, boolean writable)
- static RecordStore **openRecordStore**(String name, String vendorName, String suiteName)

El primero de ellos usa los siguientes parámetros:

- autorización:
  - AUTHMODE_PRIVATE: Sólo permite el acceso al *Record Store* a la MIDlet *suite* que lo creó.
  - AUTHMODE_ANY: Permite el acceso a cualquier MIDlet del dispositivo. Este modo hay que usarlo con mucho cuidado ya que podría provocar problemas de privacidad y seguridad.
- writable: Indicamos si el *Record Store* puede ser modificado por cualquier *MIDlet* que pueda acceder a él.

Estos parámetros sólo tienen efecto si estamos creando un *Record Store*. Si éste ya estaba creado, estos parámetros se ignorarán.

El segundo método lo usaremos para abrir un *Record Store* que está asociado a alguna *MIDlet suite* especificada por los parámetros vendorName y suiteName. El acceso vendrá limitado por el tipo de autorización del *Record Store* cuando fue creado (véase método anterior).

Cuándo terminemos de usar el *Record Store*, hay que cerrar la comunicación con él. Esto lo haremos mediante el método:

public void closeRecordStore() throws RecordStoreNotFoundException, RecordStoreException

Para cerrar correctamente la comunicación con un *Record Store*, es necesario invocar este método tantas veces como llamadas se haya realizado al método openRecordStore().

En la Tabla 6.1 podemos ver algunos métodos que nos proporcionan operaciones generales con los *Record Stores*.

Métodos	Descripción
String getName()	Devuelve el nombre del Record Store.
int getVersion()	Devuelve la versión del Record Store.
long getLastModified()	Devuelve la marca temporal.
int getNumRecords()	Devuelve el número de registros.
int getSize()	Devuelve el número de bytes ocupado por el Record Store.
int getSizeAvailable()	Devuelve el tamaño disponible para añadir registros.
String[] listRecordStores()	Devuelve una lista con los nombres de los Record Stores que existen en la MIDlet suite.
void deleteRecordStore(String name)	Elimina del dispositivo al Record Store especificado por el parámetro 'name'.
RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean actualizado)	Nos devuelve un objeto RecordEnumeration (vease punto 6.4 Operaciones avanzadas con Record Stores).
void addRecordListener(RecordListener listener)	Añade un 'listener' para detectar cambios en el Record Store.
void removeRecordListener (RecordListener listener)	Elimina un 'listener'.

Tabla 6.1 Métodos generales de la clase RecordStore

6.3.2. Manipulación de registros

Una vez creado o abierta la comunicación con el *Record Store*, podemos leer, escribir, modificar o borrar registros a nuestro gusto. Para ello, usaremos los métodos de la clase RecordStore que se ven en la Tabla 6.2.

Método	Descripción
int addRecord(byte[] datos, int offset, int numBytes)	Añade un registro al Record Store
void deleteRecord(int id)	Borra el registro 'id' del Record Store
Int getNextRecordId()	Devuelve el siguiente 'id' del registro que se vaya a insertar

byte[] getRecord(int id)	Devuelve el registro con identificador ‘id’
int getRecord(int id, byte[] buffer, int offset)	Devuelve el registro con identificador ‘id’ en ‘buffer’ a partir de ‘offset’
Int getRecordSize(int id)	Devuelve el tamaño del registro ‘id’
void setRecord(int id, byte[] datonuevo, int offset, int tamaño)	Sustituye el registro ‘id’ con el valor de ‘datonuevo’

Tabla 6.2 Métodos para el manejo de registros

Las operaciones que más vamos a realizar a la hora de trabajar con registros serán, sin duda, las de lectura y escritura. Para empezar, veremos un ejemplo donde cada registro almacenará un tipo básico, en este caso cadenas de caracteres que representarán una entrada en una agenda. Iremos guardando en los registros el nombre de las personas incluidas en la agenda. Este ejemplo nos dará una visión global de los pasos que debemos realizar para comunicarnos en el RMS. Crearemos un *Record Store* en el que guardaremos un par de registros que posteriormente recuperaremos y mostraremos por pantalla.

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

public class Agenda extends MIDlet {
   private RecordStore rs;

   public Agenda(){
      abrirRecordStore();
      escribirDatos();
      leerRegistros();
      cerrarRecordStore();
      destruirRecordStore();
   }

   public void startApp() {
      //No realizamos ninguna acción ya que no usamos ningún GUI
       //Podríamos crear un formulario para introducir datos en el Record //Store
      destroyApp(true);
      notifyDestroyed();
   }

   public void pauseApp() {
   }

   public void destroyApp(boolean unconditional) {
   }

   public void abrirRecordStore(){
      try{
         rs = RecordStore.openRecordStore(“Agenda”,true);
      }
      catch (RecordStoreException e){
```

```
            System.out.println("Error al abrir el Record Store");
        }
    }

    public void escribirDatos(){
        escribirRegistro("Antonio");
        escribirRegistro("Manolo");
    }

    public void escribirRegistro(String entrada){
        byte[] registro;
        registro = entrada.getBytes();
        try{
            rs.addRecord(registro,0,registro.length);
        }
        catch (RecordStoreException e){
            System.out.println("Error al insertar registro");
        }
    }

    public void leerRegistros(){
        byte[] registro = new byte[75];
        int longitud;
        try{
            for (int i=1;i<=rs.getNumRecords();i++){
             longitud = rs.getRecordSize(i);
             registro = rs.getRecord(i);
             System.out.println("Registro "+i+": "+ new String(registro,0,longitud));
            }
        }
        catch (RecordStoreException e){
            System.out.println("Error al leer los registros");
        }
        registro = null;
    }

    public void cerrarRecordStore(){
        try{
            rs.closeRecordStore();
        }
        catch (RecordStoreException e){
            System.out.println("Error al cerrar el Record Store");
        }
    }

    public void destruirRecordStore(){
        try{
            RecordStore.deleteRecordStore("Agenda");
        }
        catch (RecordStoreException e){
            System.out.println("Error al eliminar el Record Store");
```

```
        }
    }
}
```

Este ejemplo no es muy útil en el sentido práctico. De nada nos sirve guardar sólo el nombre de alguien si no guardamos también alguna información de interés como, por ejemplo, su número de teléfono. Llegados a este punto habría que hacerse una pregunta: ¿es posible guardar en el mismo registro el nombre y teléfono de cada persona? La respuesta es sí, es posible guardar en el mismo registro el nombre, teléfono e incluso dirección o cualquier otro dato de interés, pero no del modo que hemos hecho anteriormente, ya que para ello hay que recurrir al concepto de *stream*. Siguiendo el ejemplo que nos ocupa, lo que mejor podemos hacer es construir una segunda versión de nuestra agenda donde se guarde en cada registro el nombre y teléfono de cada persona. Aquí se verá como se realizan las operaciones de lectura y escritura a través de *streams* de bytes.

```
import java.io.*;
import javax.microedition.midlet.* ;
import javax.microedition.rms.* ;

public class Agenda2 extends MIDlet {
   private RecordStore rs;

   public Agenda2(){
      abrirRecordStore();
      escribirDatos();
      leerRegistros();
      cerrarRecordStore();
      destruirRecordStore();
   }

   public void startApp() {
      //No realizamos ninguna acción ya que no usamos ningún GUI
       //Podríamos crear un formulario para introducir datos en el Record //Store
      destroyApp(true);
      notifyDestroyed();
   }

   public void pauseApp() {
   }

   public void destroyApp(boolean unconditional) {
   }

   public void abrirRecordStore(){
      try{
         rs = RecordStore.openRecordStore("Agenda",true);
      }
      catch (RecordStoreException e){
         System.out.println("Error al abrir el Record Store");
```

```
        }
    }

    public void escribirDatos(){
        escribirRegistro("Antonio",555987654);
        escribirRegistro("Manolo",555123456);
    }

    public void escribirRegistro(String entrada, long tel){
        byte[] registro;
        ByteArrayOutputStream baos;
        DataOutputStream dos;
        try{
            baos = new ByteArrayOutputStream();
            dos = new DataOutputStream(baos);
            dos.writeUTF(entrada);
            dos.writeLong(tel);
            dos.flush();
            registro = baos.toByteArray();
            rs.addRecord(registro,0,registro.length);
            baos.close();
            dos.close();
        }
        catch (Exception e){
            System.out.println("Error al insertar registro");
        }
    }

    public void leerRegistros(){
        ByteArrayInputStream bais;
        DataInputStream dis;
        byte[] registro = new byte[75];
        try{
            bais = new ByteArrayInputStream(registro);
            dis = new DataInputStream(bais);
            for (int i=1;i<=rs.getNumRecords();i++){
             rs.getRecord(i,registro,0);
             System.out.println("Registro "+i);
              System.out.println("Nombre: "+dis.readUTF()+" Telefono:
"+dis.readLong());
             bais.reset();
            }
            bais.close();
            dis.close();
        }
        catch (Exception e){
            System.out.println("Error al leer los registros");
        }
        registro = null;
    }
```

```
    public void cerrarRecordStore(){
        try{
            rs.closeRecordStore();
        }
        catch (RecordStoreException e){
            System.out.println("Error al cerrar el Record Store");
        }
    }
    public void destruirRecordStore(){
        try{
            RecordStore.deleteRecordStore("Agenda");
        }
        catch (RecordStoreException e){
            System.out.println("Error al eliminar el Record Store");
        }
    }
}
```

En este ejemplo hemos utilizado *streams* para leer y escribir registros en el *Record Store*. Con ello hemos conseguido almacenar información de distinto tipo en el mismo registro. La estructura del programa es la misma que antes, pero cambiando el contenido de los métodos encargados de escribir y leer registros: escribirDatos(), escribirRegistro(String nombre, long telefono) y leerRegistros(). En estos dos últimos métodos es dónde usamos los *streams* para leer y escribir datos del Record Store. También hemos cambiado la forma de manejar las excepciones ya que capturamos una excepción genérica de tipo Exception para simplificar el programa en vez de una RecordStoreException. Esto es así porque al usar *streams* se puede lanzar una excepción de tipo IOException que también ha de ser capturada. En la práctica deberíamos usar un par de bloques catch que capturaran ambos tipos de excepciones por separado y así informar al usuario de una forma más clara que tipo de error ha ocurrido.

6.4. Operaciones avanzadas con *Record Stores*

La clase RecordStore nos proporciona algunas interfaces que nos facilitan el trabajo a la hora de manipular registros. Básicamente, vamos a ver como estas interfaces nos ayudan a la hora de navegar por los registros de un *Record Store*, realizar búsquedas en ellos y ordenaciones.

6.4.1. Navegación a través de un *Record Store*

En los ejemplos anteriores hemos usado un simple bucle para movernos entre los distintos registros de nuestro *Record Store*:

```
for (int i=1;i<=rs.getNumRecords();i++){
      longitud = rs.getRecordSize(i);
      registro = rs.getRecord(i);
      System.out.println("Registro "+i+": "+ new
      String(registro,0,longitud));
      }
```

Sin embargo, la clase RecordStore nos proporciona la interfaz RecordEnumeration que nos facilita esta tarea. Esta interfaz posee los métodos que aparecen en la Tabla 6.3.

Método	Descripción
int numRecords()	Devuelve el número de registros
Byte[] nextRecord()	Devuelve el siguiente registro
int nextRecordId()	Devuelve el siguiente 'id' a devolver
Byte[] previousRecord()	Devuelve el registro anterior
int previousRecordId()	Devuelve el 'id' del registro anterior
boolean hasNextElement()	Devuelve true si existen más registros en adelante
boolean hasPreviousElement()	Devuelve true si existen más registros anteriores
void keepUpdated()	Provoca que los indices se actualicen cuándo se produzca algún cambio en el Record Store
boolean isKeptUpdated()	Devuelve true si los indices se actualizan al producirse algún cambio en el Record Store
void rebuild()	Actualiza los índices del RecordEnumeration
void reset()	Actualiza los índices a su estado inicial
void destroy()	Libera todos los recursos ocupados

Tabla 6.3 Métodos de RecordEnumeration

Haciendo uso de esta interfaz podemos sustituir el bucle anterior por algo como lo siguiente:

```
RecordEnumeration re = rs.enumerateRecords(null,null,false);
while (re.hasNextElement()){
      registro = re.nextRecord();
      //Realizo las operaciones que quiera
      ...
}
```

Como vemos, la navegación por los registros usando RecordEnumeration es mucho más intuitiva y nos permite realizar acciones como movernos hacia delante o hacia atrás de una manera muy sencilla.

Vamos a fijarnos en los parámetros que le pasamos al método enumerateRecords(). Como podemos ver en la tabla 6.1, hemos de pasarle como primer parámetro una referencia a un RecordFilter y como segundo, otra a un RecordComparator. Nosotros hemos sustituido ambas referencias por null con lo que conseguiremos un objeto RecordEnumeration con la misma estructura de registros que el *Record Store* original. El tercer parámetro indica si queremos que los índices se actualicen en el caso de que se produzca alguna acción en el *Record Store* por cualquier otro MIDlet mientras trabajamos con él.

6.4.2. Búsqueda de registros

Para realizar una búsqueda eficiente de registros en un Record Store vamos a utilizar la interfaz RecordFilter. Esta interfaz se encarga de devolver al RecordEnumeration únicamente los registros que coincidan con un determinado patrón de búsqueda.

Para usar esta interfaz hemos de implementar necesariamente el método:

public boolean matches(byte [] candidato)

que se encarga de comparar el registro candidato pasado como parámetro con el valor que queremos buscar y devolvemos *true* en caso de que coincidan.

En el siguiente ejemplo vamos a crear un RecordFilter y así podremos ver su funcionamiento

```
public class Filtro implements RecordFilter{
	private String cadenaabuscar = null;

	public Filtro(String cadena){
		this.cadenaabuscar = cadena.toLowerCase();
}

public boolean matches(byte[] candidato){
	boolean resultado = false;
	String cadenacandidata;
	ByteArrayInputStream bais;
	DataInputStream dis;
	try{
		bais = new ByteArrayInputStream(candidato);
		dis = new DataInputStream(bais);
		cadenacandidata = dis.readUTF().toLowerCase();
	}
	catch (Exception e){
		return false;
	}
	if ((cadenacandidata != null) && (cadenacandidata.indexOf(cadenaabuscar)
	!= -1))
		return true;
	else
		return false;
```

```
        }
    }
```

Si usamos este RecordFilter a la hora de invocar a enumerateRecords():

```
//Creo un objeto de tipo RecordFilter
Filtro buscar = new Filtro("Antonio");
//Obtengo el RecordEnumeration usando el filtro anterior
RecordEnumeration re = rs.enumerateRecords(buscar,null,false);
//Si encuentro algún registro dado el patrón de búsqueda
if (re.numRecords() > 0)
    System.out.println("Patron 'Antonio' encontrado");
```

el resultado será que el RecordEnumeration devuelto sólo contendrá los registros en los que se haya encontrado el patrón de búsqueda.

6.4.3. Ordenación de registros

La ordenación de registros se realiza a través del interfaz RecordComparator. Esta interfaz funciona básicamente igual que el RecordFilter. Existe un método public int compare(byte[] reg1, byte[] reg2) que ha de ser implementado obligatoriamente. Este método es el encargado de realizar la comparación entre los campos que deseemos de los registros y el entero que nos devuelve nos indica si reg1 va antes o después que reg2. El valor devuelto por este método puede ser:

- EQUIVALENT: Los registros pasados al método compare son equivalentes.
- FOLLOWS: El primer registro sigue al segundo.
- PRECEDES: El primer registro precede al segundo.

Al igual que hicimos con el RecordFilter vamos a crear una clase que implemente la interfaz RecordComparator y veamos cómo se realiza la comparación entre los registros:

```
public class Compara implements RecordComparator{

    public boolean compare(byte[] reg1, byte[] reg2){
    ByteArrayInputStream bais;
    DataInputStream dis;
    String cad1, cad2;
      try{
            bais = new ByteArrayInputStream(reg1);
            dis = new DataInputStream(bais);
            cad1 = dis.readUTF();
            bais = new ByteArrayInputStream(reg2);
            dis = new DataInputStream(bais);
            cad2 = dis.readUTF();
            int resultado = cad1.compareTo(cad2);
            if (resultado == 0)
                  return RecordComparator.EQUIVALENT;
            else if (result < 0)
                  return RecordComparator.PRECEDES;
```

```
                else
                        return RecordComparator.FOLLOWS;
            }
            catch (Exception e){
                return RecordComparator.EQUIVALENT;
            }
        }
    }
```

Esta clase puede usarse a la hora de crear el RecordEnumeration para recorrer los elementos de manera ordenada:

```
//Creo un objeto de tipo RecordComparator
Compara comp = new Compara();
//Obtengo el RecordEnumeration usando el objeto anterior, y
// con los registros ordenados
RecordEnumeration re = rs.enumerateRecords(null,comp,false);
```

6.5. Manejo de eventos en los *Record Stores*

Existe además otra interfaz que nos permite capturar eventos que se produzcan a la hora de realizar alguna acción en un *Record Store*. La interfaz RecordListener es la encargada de recoger estas acciones. Esta interfaz funciona como cualquier otro *listener*. Cuándo ocurre algún evento, un método es llamado para notificarnos este cambio. Los tres métodos de la Tabla 6.4 se encargan de esta tarea.

Método	Descripción
void recordAdded(RecordStore rs, int id)	Invocado cuándo un registro es añadido
void recordChanged(RecordStore rs, int id)	Invocado cuándo un registro es modificado
void recordDeleted(RecordStore rs, int id)	Invocado cuándo un registro es borrado

Tabla 6.4 Métodos de la interfaz RecordListener

La implementación de estos métodos es vacía por lo que debe ser proporcionada por nosotros. En el siguiente ejemplo vamos a crear una clase que implemente esta interfaz. Los métodos anteriores lo único que harán será informar por pantalla que se ha producido algún cambio en el *Record Store*.

```
public class PruebaListener implements RecordListener{

public void recordAdded(RecordStore rs, int id){
    try{    String nombre = rs.getName();
    System.out.println("Registro "+id+" añadido al Record Store: "+nombre);
    }
    catch (Exception e){
            System.err.println(e);
    }
}
```

```
public void recordDeleted(RecordStore rs, int id){
        try{        String nombre = rs.getName();
        System.out.println("Registro "+id+" borrado del Record Store: "+ nombre);
        }
        catch (Exception e){
                System.err.println(e);
        }
}

public void recordChanged(RecordStore rs, int id){
        try{String nombre = rs.getName();
        System.out.println("Registro "+id+" modificado del Record Store: "+nombre);
        }
        catch (Exception e){
                System.err.println(e);
        }
}
}
```

Añadir este RecordListener a un *Record Store* es muy sencillo. Sólo tenemos que incluir el siguiente código en el *MIDlet*:

```
RecordStore rs = openRecordStore("Almacen1",true);
rs.addRecordListener( new PruebaListener());
```

Ya hemos visto todos los elementos que nos proporciona el paquete javax.microedition.rms. A partir de ahora ya podemos almacenar cualquier tipo de información en nuestro dispositivo MID de manera persistente. Lo siguiente que podríamos hacer es dotar de una interfaz de usuario a los ejemplos que hemos visto en este capítulo para dotar de funcionalidad a la agenda que hemos creado.

6.6. Ejemplo práctico

En este punto vamos a dotar a nuestra aplicación **PingPong** de capacidad para almacenar las mejores puntuaciones en el MID. El código de esta aplicación ya lo conocemos excepto por la clase Resultado que es la que posee los métodos necesarios para almacenar la información en los *Record Stores*. Aquí podemos ver el código de esta clase:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;
import java.io.*;

public class Resultado extends List implements CommandListener{

   private PingPong midlet;
   private Command aceptar;
   private RecordStore rs;
```

```
public Resultado(PingPong m){
   super("Records",List.IMPLICIT);
   initPuntuacion("Clasificacion1");
   initPuntuacion("Clasificacion2");
   initPuntuacion("Clasificacion3");
   midlet = m;
   aceptar = new Command("Aceptar",Command.OK,1);
   this.addCommand(aceptar);
   this.setCommandListener(this);
}

public void commandAction(Command c, Displayable d){
   midlet.setMenu();
}

public void verResultado(){
   this.setTitle("Records Nivel "+midlet.getNivel());
   try{
      for (int i=this.size();i>0;i--){
         this.delete(i-1);
      }
      this.append("Puntuacion   Jugador",null);
      rs = RecordStore.openRecordStore("Clasificacion"+midlet.getNivel(),true);
      for (int i=1;i<=rs.getNumRecords();i++){
         byte[] registro = rs.getRecord(i);
         ByteArrayInputStream bais = new ByteArrayInputStream(registro);
         DataInputStream dis = new DataInputStream(bais);
         String nombre = dis.readUTF();
         long puntuacion = dis.readLong();
         this.append("    "+puntuacion+"            "+nombre,null);
         bais.close();
         dis.close();
         registro = null;
      }
      rs.closeRecordStore();
   }
   catch (Exception e){
      System.out.println("Excepción detectada");
   }
}

public void insertarRegistro(String nombre, long punt){
  try{
     ByteArrayOutputStream baos = new ByteArrayOutputStream();
     DataOutputStream dos = new DataOutputStream(baos);
     dos.writeUTF(nombre);
     dos.writeLong(punt);
     dos.flush();
     byte[] nreg = baos.toByteArray();
     baos.close();
```

```
      dos.close();
      rs = RecordStore.openRecordStore("Clasificacion"+midlet.getNivel(),true);
      byte[] reg3 = rs.getRecord(3);
      if (getPuntuacion(reg3) < punt){
          byte[] reg2 = rs.getRecord(2);
          byte[] reg1 = rs.getRecord(1);
          if (getPuntuacion(reg2) < punt){
              if (getPuntuacion(reg1) < punt){
                  rs.setRecord(1,nreg,0,nreg.length);
                  rs.setRecord(2,reg1,0,reg1.length);
                  rs.setRecord(3,reg2,0,reg2.length);
              }
              else{
                  rs.setRecord(2,nreg,0,nreg.length);
                  rs.setRecord(3,reg2,0,reg2.length);
              }
          }
          else{
              rs.setRecord(3,nreg,0,nreg.length);
          }
      }
      rs.closeRecordStore();
    }
    catch (Exception e){
        System.out.println(e);
    }
}

private long getPuntuacion(byte[] registro){
    try{
      ByteArrayInputStream bais = new ByteArrayInputStream(registro);
      DataInputStream dis = new DataInputStream(bais);
      String nombre = dis.readUTF();
      long puntuacion = dis.readLong();
      bais.close();
      dis.close();
      return puntuacion;
    }
    catch (Exception e){
        System.out.println(e);
        return -1;
    }
}

private void initPuntuacion(String nombre){
    try{
   rs = RecordStore.openRecordStore(nombre,true);
        if (rs.getNumRecords() == 0){
            for (int i=0;i<3;i++){
                byte[] registro;
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
                DataOutputStream dos = new DataOutputStream(baos);
                dos.writeUTF("AAA");
                dos.writeLong(0);
                dos.flush();
                registro = baos.toByteArray();
                rs.addRecord(registro,0,registro.length);
                baos.close();
                dos.close();
                registro = null;
              }
            }
            rs.closeRecordStore();
        }
        catch (Exception e){
            System.out.println(e);
        }
    }
    public boolean esRecord(long punt){
        try{
            rs = RecordStore.openRecordStore("Clasificacion"+midlet.getNivel(),true);
            byte[] reg = rs.getRecord(3);
            long p3 = getPuntuacion(reg);
            rs.closeRecordStore();
            reg = null;
            return ((p3 >= punt)?false:true);
        }
        catch (Exception e){
            System.out.println(e);
            return false;
        }
    }
}
```

Esta clase es la encargada de almacenar los registros con las mejores puntuaciones. Dispone de los siguientes métodos:

- verResultado(): Nos muestra los records almacenados en el *Record Store*.
- insertarRegistro(String n, long p): Nos inserta en el *Record Store* un nuevo record en el caso de que sea así en la posición correcta.
- getPuntuacion(byte[] registro): Nos devuelve la puntuación almacenada en un registro dado.
- initPuntuacion(String nombre): Nos inicializa las puntuaciones en el caso de que sea la primera ejecución del MIDlet y no existan datos en el Record Store.
- esRecord(long puntuacion): Nos devuelve *true* o *false* dependiendo de si la puntuación pasada como parámetro sea un record nuevo o no.

Capítulo 7: Comunicaciones

7.1. Descripción del capítulo

Hasta ahora hemos realizado multitud de aplicaciones que podríamos haber hecho con una mayor calidad y sin perder ninguna funcionalidad usando la plataforma J2SE en un ordenador de sobremesa. Hemos creado interfaces de usuario limitadas que podríamos haber mejorado enormemente usando las clases que nos proporciona J2SE para ello: java.awt y/o java.swing. También hemos sido capaces de almacenar información en nuestro móvil usando una estructura de registros cuando podríamos haber usado un potente SGBD (Sistema Gestor de Bases de Datos). Sin duda, hasta ahora hemos perdido numerosas características debido a cantidad de restricciones que soportan los dispositivos MID y restricciones propias del lenguaje.

Sin embargo, la gran ventaja que poseen estos dispositivos MID es su posibilidad de estar siempre 'conectados'. La posibilidad de llevar un dispositivo que ocupa poco espacio y nos permita además comunicarnos en cualquier momento y lugar nos abre un abanico de posibles aplicaciones que no podríamos disfrutar en un PC de sobremesa, ni tan siquiera en un ordenador portátil. Piénsese en la realización de aplicaciones de mensajería instantánea o en la posibilidad de leer e-mails de nuestra cuenta de correo o enviarlos.

Este capítulo, sin duda alguna, es el más importante de todos los dados hasta ahora ya que, el gran potencial de los dispositivos MID es la posibilidad de conexión en cualquier momento y transferir cualquier tipo de información mientras dure esa conexión. En este capítulo veremos cómo realizar esa transferencia de información con nuestro dispositivo MID y las herramientas que la plataforma J2ME nos proporciona para ese propósito. Para ello, a la vez que se va explicando cada una de estas herramientas, vamos a ir dotando de funcionalidad a nuestra aplicación **Hogar**. En el caso de nuestra aplicación, vamos a establecer una comunicación con un *servlet* que será el encargado de responder a nuestras peticiones y consultas y que se encargará "virtualmente" de aceptar nuestras órdenes y llevarlas a cabo.

7.2. Conceptos básicos

Para empezar, veamos las diferencias y semejanzas que nos vamos a encontrar a la hora de trabajar en J2ME con respecto a J2SE.

En ambas plataformas se usan *streams* para escribir y leer datos. Estos *streams* ya los hemos usado en el capítulo anterior a la hora de almacenar o recuperar información en un *Record Store*. Estas clases están en el paquete java.io por lo que no forman parte del perfil MIDP. En cambio, disponemos del paquete javax.microedition.io, el cual contiene las clases que nos dan soporte para el trabajo en red y comunicaciones en las aplicaciones MIDP. Este paquete existe en contraposición con el paquete java.net de J2SE.

Las aplicaciones MIDP usan los paquetes javax.microedition.io y java.io de la siguiente manera. El primer paquete contiene numerosas clases que nos permitirán crear y manejar diferentes conexiones de red. Estas conexiones podrán usar diferentes formas de comunicación: HTTP, datagramas, sockets,… Pues bien, el paquete java.io se encargará de proporcionarnos las clases necesarias para leer y escribir en estas conexiones.

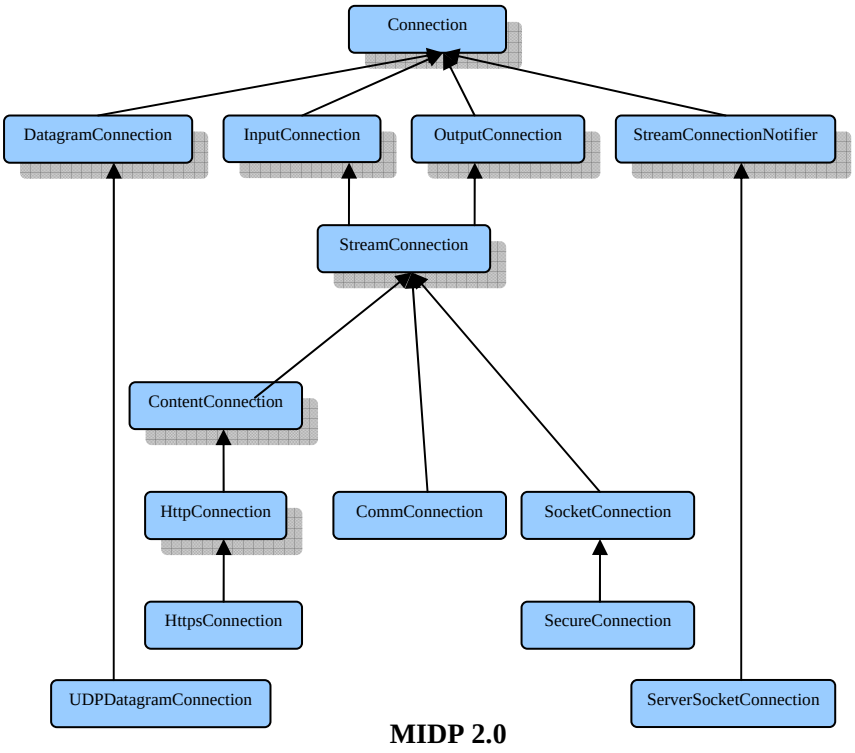

Figura 7.1 Jerarquía de interfaces

Como ya vimos en el capítulo 4, estas clases orientadas a la conexión en red y comunicaciones reciben el nombre de *Generic Connection Framwork*. En la Figura 7.1 podemos ver cómo se organizan jerárquicamente. Es más, también vamos a ver cómo se

organizan las interfaces que nos suministra la especificación MIDP y que están en un nivel superior al GCF ya que implementan los detalles de diversos protocolos de comunicación.

Como vemos, la raíz del árbol es la interfaz Connection que representa la conexión más genérica y abstracta que podemos crear. El resto de interfaces que derivan de Connection representan los distintos tipos de conexiones que podemos crear. En los siguientes puntos veremos más detenidamente cada una de estas conexiones.

7.3. Clases y conexiones del *Generic Connection Framework*

La anterior jerarquía de clases nos puede dar una idea de la filosofía que sigue el *Generic Connection Framework*. Los dispositivos MID, como ya sabemos, poseen bastantes restricciones. Por esta razón, se hace imposible poseer la implementación de los distintos protocolos existentes. Lo que nos proporciona el *Generic Connection Framework* es una sola clase Connector que nos esconde los detalles de la conexión. Esta clase puede por sí misma crear cualquier tipo de conexión: Archivos, Http, socket, etc.

7.3.1. Clase Connector

public class Connector

Como hemos dicho anteriormente, el GCF proporciona la clase Connector que nos esconde los detalles de la conexión. De esta forma podemos realizar cualquier tipo de conexión usando sólo esta clase y sin preocuparnos de cómo se implementa el protocolo requerido

La conexión se realiza mediante el siguiente mensaje genérico:
Connector.open(“protocolo:dirección;parámetros”);

Algunos ejemplo de invocación son:

```
Connector.open(“http://direccionquesea.es”);
Connector.open(“file://autoexec.bat”);
Connector.open(“socket://direccion:0000”);
```

La clase Connector se encarga de buscar la clase específica que implemente el protocolo requerido. Si esta clase se encuentra, el método open() devuelve un objeto que implementa la interfaz Connection. La clase Connector posee los métodos de la Tabla 7.1.

Método	Descripción
public static Connection open(String dir)	Crea y abre una conexión.
public static Connection open(String dir,	Crea y abre una conexión con permisos.

int modo)	
public static Connection open(String dir, int mode, boolean tespera)	Crea y abre una conexión especificando el permiso y tiempo de espera.
public static DataInputStream openDataInputStream(String dir)	Crea y abre una conexión de entrada devolviendo para ello un DataInputStream.
public static DataOutputStream openDataOutputStream(String dir)	Crea y abre una conexión de salida a través de un DataOutputStream.
public static InputStream openInputStream(String dir)	Crea y abre una conexión de entrada usando un InputStream.
public static OutputStream openOutputStream(String dir)	Crea y abre una conexión de salida devolviendo para ello un OutputStream.

Tabla 7.1 Métodos de la clase Connector

Los permisos para realizar una conexión aparecen en la Tabla 7.2**.**

Modo	Descripción
READ	Permiso de sólo lectura.
READ_WRITE	Permiso tanto de lectura como de escritura.
WRITE	Permiso de sólo escritura.

Tabla 7.2 Permisos que se pueden usar al abrir una conexión

7.3.2. Interfaz Connection

public abstract interface Connection

La interfaz Connection se encuentra en lo más alto de la jerarquía de interfaces del *Generic Connection Framework*, por lo que cualquier otra interfaz deriva de él. Una conexión de tipo Connection se crea después de que un objeto Connector invoque al método open(). Como sabemos, esta interfaz representa a la conexión más abstracta y genérica posible. Por esta razón, el único método que posee esta interfaz es el de la Tabla 7.3.

Método	Descripción
public void close()	Cierra la conexión.

Tabla 7.3 Métodos de la interfaz Connection

Conforme avancemos en la jerarquía de clases del *Generic Connection Framework* veremos que cada una de ellas van añadiendo más capacidades a la conexión.

7.3.3. Interfaz InputConnection

public abstract interface InputConnection **extends** Connection

La interfaz InputConnection representa una conexión basada en *streams* de entrada. Esta interfaz sólo posee dos métodos que devuelven objetos de tipo InputStreams (ver Tabla 7.4).

Método	Descripción
DataInputStream openDataInputStream()	Devuelve un DataInputStream asociado a la conexión.
InputStream openInputStream()	Devuelve un InputStream asociado a la conexión

Tabla 7.4 Métodos de la interfaz InputConnection

El siguiente ejemplo ilustra cómo se realiza una conexión a través de esta interfaz:

```
String url = "www.midireccion.com";
InputConnection conexión = (InputConnection)Connector.open(url);
DataInputStream dis = conexion.openDataInputStream();
```

7.3.4. Interfaz OutputConnection

```
public abstract interface OutputConnection extends Connection
```

La interfaz OutputConnection representa una conexión basada en *streams* de salida. Esta interfaz sólo posee dos métodos que devuelven objetos de tipo OutputStreams (ver Tabla 7.5).

Método	Descripción
DataOutputStream openDataOutputStream()	Devuelve un DataOutputStream asociado a la conexión.
OutputStream openOutputStream()	Devuelve un OutputStream asociado a la conexión

Tabla 7.5 Métodos de la interfaz OutputConnection

La conexión a través de esta interfaz se realiza de forma análoga a la interfaz InputConnection.

7.3.5. Interfaz StreamConnection

```
public abstract interface StreamConnection extends InputConnection,
OutputConnection
```

Esta interfaz representa una conexión basada en *streams* tanto de entrada como de salida. No añade ningún método nuevo, si no que hereda los métodos de los interfaces que están por encima de él. Su única misión en la jerarquía del GCF es representar un tipo de conexión cuyos datos pueden ser tratados como *streams* de *bytes* y en la que es posible leer y escribir.

El siguiente ejemplo ilustra cómo se crea una conexión de este tipo en la que se va a crear un *stream* de entrada del que se leerá la información:

```
StreamConnection sc = (StreamConnection)Connector.open(url);
InputStream is = sc.openInputStream();
```

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
int c;
while((c = is.read()) != -1){
      baos.write(c);
}
```

Téngase en cuenta que la dirección *url* que se usa al crear la conexión puede apuntar a un fichero (url = "http://www.direccion.com/fichero1.txt") o a una imagen (url = "http://www.direccion.com/imagen.png"); en cualquier caso, esa información la guardamos en un ByteArrayOutputStream que podemos tratar de la manera que más convenga.

7.3.6. Interfaz ContentConnection

public abstract interface ContentConnection **extends** StreamConnection

La interfaz ContentConnection extiende a la interfaz StreamConnection. Esta interfaz representa conexiones que pueden describir su contenido de alguna forma. En las conexiones anteriores transmitíamos *bytes* sin importarnos su composición, pero en estas conexiones la estructura de *bytes* a transmitir debe ser conocida de antemano.

Concretamente, esta interfaz representa a familias de protocolos en los que se definen atributos los cuales describen los datos que se transportan. Esta interfaz añade varios métodos que pueden ser usados en esta familia de protocolos (Tabla 7.6).

Método	Descripción
public String getEncoding()	Devuelve la codificación empleada para representar el contenido de la información.
public long getLength()	Devuelve la longitud de datos.
public String getType()	Devuelve el tipo de datos.

Tabla 7.6 Métodos de la interfaz ContentConnection

Con esta interfaz podemos conocer de antemano la longitud de datos que recibimos, con lo que las operaciones de lectura de datos se pueden simplificar mucho haciendo uso de esta información. Este mecanismo de lectura se puede apreciar en el siguiente ejemplo donde vamos a sustituir el bucle usado para realizar la lectura de datos por dos simples líneas de código:

```
ContentConnection cc = (ContentConnection)Connector.open(url);
is = cc.openInputStream();
byte[] datos;
int long = (int)cc.getLength();
if (long != -1) {
datos = new byte[long];
is.read(datos);
}
else // Realizamos bucle de lectura de datos
```

Este algoritmo mejora al bucle del apartado 7.3.5 ya que realizamos la lectura de datos de una sola vez, en vez de byte a byte.

7.3.7. Interfaz StreamConnectionNotifier

public abstract interface StreamConnectionNotifier **extends** Connection

Esta interfaz deriva directamente de Connection. Representa al establecimiento de conexiones lanzadas por clientes remotos. Si la conexión se realiza con éxito, se devuelve un StreamConnection para establecer la comunicación.

Esta interfaz sólo posee el método de la Tabla 7.7.

Método	Descripción
public StreamConnection acceptAndOpen()	Devuelve un StreamConnection que representa un socket por parte del servidor.

Tabla 7.7 Métodos de la interfaz StreamConnectionNotifier

7.3.8. Interfaz DatagramConnection

public abstract interface DatagramConnection **extends** Connection

Esta interfaz define las capacidades que debe tener una conexión basada en datagramas. A partir de esta interfaz se pueden definir distintos protocolos basados en datagramas, pero su implementación habría que realizarla a nivel del perfil.

Los métodos que posee esta interfaz son los de la Tabla 7.8.

Método	Descripción
public int getMaximumLength()	Devuelve la longitud máxima que puede tener un datagrama.
public int getNominalLength()	Devuelve la longitud nominal que puede tener un datagrama
public Datagram newDatagram(byte[] buf, int tam)	Crea un datagrama.
public Datagram newDatagram(byte[] buf, int tam, String dir)	Crea un datagrama.
public Datagram newDatagram(int tam)	Crea un datagrama automáticamente.
public Datagram newDatagram(int tam, String dir)	Crea un datagrama
public void receive(Datagram dat)	Recive un datagrama.
public void send(Datagram dat)	Envía un datagrama.

Tabla 7.8 Métodos de la interfaz DatagramConnection

7.4. Comunicaciones HTTP

Ya hemos estudiado las clases e interfaces que conforman el *Generic Connection Framework*. Estas interfaces nos dan un mecanismo por el cual podemos establecer un gran número de conexiones de distinto tipo. Como hemos dicho, el GCF nos esconde los detalles de la conexión y además no implementa ningún protocolo en concreto ya que esta implementación debe estar en el nivel de los perfiles. Pues bien, en este punto vamos a ver en detalle la interfaz HttpConnection que implementa el protocolo HTTP 1.1. En el siguiente punto estudiaremos el resto de interfaces implementadas por el perfil MIDP 2.0.

El protocolo HTTP es un protocolo de tipo petición/respuesta. El funcionamiento de este protocolo es el siguiente: El cliente realiza una petición al servidor y espera a que éste le envíe una respuesta. Normalmente, esta comunicación es la que suele realizarse entre un navegador web (cliente) y un servidor web (servidor). En nuestra aplicación **Hogar**, esta comunicación la vamos a realizar entre nuestro MIDlet (cliente) y un servlet (servidor) que recibirá nuestras peticiones y, dependiendo del caso, nos devolverá un resultado.

Una conexión HTTP pasa por tres estados, como ilustra la Figura 7.2.

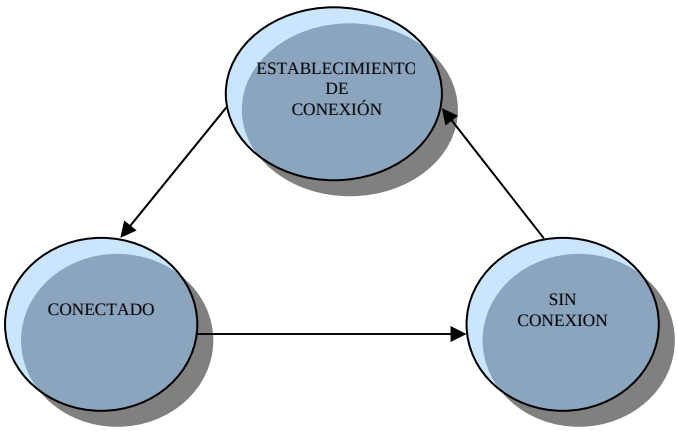

Figura 7.2 Estados de una conexión HTTP

Ahora veremos más detenidamente cada uno de estos estados y las operaciones que podemos realizar en cada uno de ellos:

7.4.1. Estado de Establecimiento

En este estado es dónde vamos a establecer los parámetros de la comunicación. El cliente prepara la petición que va a realizar al servidor, además de negociar con él una serie de parámetros como el formato, idioma, etc…

Existen dos métodos que sólo pueden ser invocados en este estado (ver Tabla 7.9).

Método	Descripción
public void setRequestMethod(String tipo)	Establece el tipo de petición.
public void setRequestProperty(String clave, String valor)	Establece una propiedad de la petición.

Tabla 7.9 Métodos relacionados con la etapa de establecimiento

El primer método establece el tipo de petición que vamos a realizar. Esta petición puede ser de los tipos indicados en la Tabla 7.10.

Tipo	Descripción
GET	Petición de información en la que los datos se envían como parte del URL.
POST	Petición de información en la que los datos se envían aparte en un *stream*.
HEAD	Petición de metainformación.

Tabla 7.10 Tipos de peticiones

El segundo método establece cierta información adicional a la petición. Esta información permite negociar entre el cliente y el servidor detalles de la petición como, por ejemplo, idioma, formato, etc. Estos campos forman parte de la cabecera de la petición y aunque existen más de 40 distintos, los más interesantes podemos verlos en la tabla Tabla 7.11.

Campo	Descripción
User-Agent	Tipo de contenido que devuelve el servidor.
Content-Language	Pais e idioma que usa el cliente.
Content-Length	Longitud de la petición.
Accept	Formatos que acepta el cliente.
Connection	Indica al servidor si se quiere cerrar la conexión después de la petición o se quiere dejar abierta.
Cache-Control	Sirve para controlar el almacenamiento de información.
Expires	Tiempo máximo para respuesta del servidor.
If-Modified-Since	Pregunta si el contenido solicitado se ha modificado desde una fecha dada.

Tabla 7.11 Campos de cabecera

Ya tenemos la información necesaria para establecer una comunicación entre un cliente y un servidor. Ya que existen principalmente dos tipos de petición vamos a ver cada una por separada.

7.4.1.1. Peticiones GET

A continuación se muestra un ejemplo en el que se prepara una conexión mediante la interfaz HttpConnection usando una petición de tipo GET.

```
//Creamos la conexión
String url = “http://www.midireccion.com/local?sala=1&luz=apagada”;
HttpConnection hc = (HttpConnection)Connector.open(url);
```

```
// Informamos del tipo de petición que vamos a efectuar
hc.setRequestMethod(HttpConnection.GET);
//Establecemos algunos campos de cabecera
hc.setRequestProperty("User-Agent","Profile/MIDP-2.0
Configuration/CLDC-1.0");
hc.setRequestProperty("Content-Language","es-ES");
```

El código anterior establece la comunicación con el servidor y deja preparada una conexión para intercambiar información con éste. Como puede apreciarse, la información sobre la petición va incluida en la cabecera de la dirección URL. El cuerpo de la petición lo forma la cadena "sala=1&luz=apagada". Esta información va detrás del símbolo '?' situado al final de la dirección URL. Cada parámetro de la petición va separado del siguiente por el símbolo '&'.

7.4.1.2. Peticiones POST

En una petición de este tipo, el cuerpo de la petición se envía en un *stream* después de iniciar la conexión. El siguiente ejemplo muestra cómo se realiza el envío del cuerpo de la petición:

```
//Creamos la conexión
String url = "http://www.midireccion.com";
HttpConnection hc = (HttpConnection)Connector.open(url);
// Informamos del tipo de petición que vamos a efectuar
hc.setRequestMethod(HttpConnection.POST);
//Establecemos algunos campos de cabecera
hc.setRequestProperty("User-Agent","Profile/MIDP-2.0
Configuration/CLDC-1.0");
hc.setRequestProperty("Content-Language","es-ES");
// Enviamos el cuerpo de la petición
OutputStream os = hc.openOutputStream();
os.write("sala=1".getBytes());
os.write("&luz=apagada".getBytes());
os.flush();
```

7.4.2. Estado de Conexión

En este estado se realiza el intercambio de información entre el cliente y el servidor. En los ejemplos anteriores hemos visto la manera de enviar nuestra petición al servidor. Este es un buen momento para ver cómo se realiza la respuesta del servidor hacia el cliente

7.4.2.1. Respuesta del servidor

Al igual que la petición del cliente poseía distintas partes, la respuesta del servidor se compone de:

- Línea de estado
- Cabecera
- Cuerpo de la respuesta.

Para conocer la respuesta del servidor, la interfaz HttpConnection nos proporciona diversos métodos que nos permitirán conocer las distintas partes de ésta (ver Tabla 7.12).

Método	Descripción
public int getResponseCode()	Devuelve el código de estado.
public String getResponseMessage()	Devuelve el mensaje de respuesta.

Tabla 7.12 Métodos para obtener el estado de la respuesta

La interfaz HttpConnection dispone de 35 códigos de estado diferentes. Básicamente podemos dividirlos en cinco clases de la siguiente manera:

- 1xx – Código de información.
- 2xx – Código de éxito.
- 3xx – Código de redirección.
- 4xx – Código de error del cliente.
- 5xx – Código de error del servidor.

Por ejemplo, el código 400 corresponde a la constante HTTP_BAD_REQUEST. En realidad la respuesta del servidor posee el siguiente formato:

HTTP/1.1 400 Bad Request
HTTP/1.1 200 OK

En la respuesta se incluye el protocolo usado, seguido del código de estado y de un mensaje de respuesta. Este mensaje es el devuelto al invocar el método getResponseMessage().

Al igual que el cliente puede mandar información de cabecera adicional a la petición, el servidor también puede mandar esta información al cliente. Los métodos de la interfaz HttpConnection que aparecen en la Tabla 7.13 se usan para conseguir esta respuesta.

Método	Descripción
String getHeaderField(int n)	Devuelve el valor del campo de cabecera número n.
String getHeaderField(String nombre)	Devuelve el valor del campo de cabecera especificado por el nombre.
long getHeaderFieldDate(String nombre, long def)	Devuelve un long que representa una fecha.
int getHeaderFieldInt(String nombre, int def)	Devuelve el campo nombrado como un entero.
int getHeaderFieldKey(int n)	Devuelve la clave del campo de cabecera usando un índice.

public long getDate()	Devuelve el campo de cabecera “fecha”.
public long getExpiration()	Devuelve el campo de cabecera “expires”.
public long getLastMoified()	Devuelve el campo de cabecera “last-modified”.

Tabla 7.13 Métodos usados para obtener información de la cabecera

Existen también otros métodos que nos permiten obtener diversa información sobre la conexión (ver Tabla 7.14).

Método	Descripción
String getFile()	Devuelve el nombre del archivo de la URL.
String getHost()	Devuelve el host de la URL.
Int getPort()	Devuelve el puerto de la URL.
String getProtocol()	Devuelve el protocolo de la URL.
String getQuery()	Devuelve la cadena de petición(respuestas GET).
String getRef()	Devuelve la referencia.
String getURL()	Devuelve la cadena de conexión URL.

Tabla 7.14 Métodos que proporcionan información sobre la conexión

En este punto vamos a realizar un ejemplo recopilatorio donde a través de un *MIDlet* realizaremos una petición a un servlet y mostraremos por pantalla la respuesta recibida. Aquí se verá como se realiza la lectura del cuerpo de la respuesta y veremos el código tanto del *MIDlet* como del servlet. El del *MIDlet* es:

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

public class PruebaServlet extends MIDlet implements CommandListener{

   private Display pantalla;
   private Command salir, conectar;
   private Form fprincipal;
   private String respuesta = null;
   private StringItem txt;
   private Comunicacion com;

   public PruebaServlet(){
      pantalla = Display.getDisplay(this);
      com = new Comunicacion(this);
      txt = new StringItem("","");
      salir = new Command("Salir",Command.EXIT,1);
      conectar = new Command("Conectar",Command.OK,1);
      fprincipal = new Form("Servlet");
      fprincipal.addCommand(salir);
      fprincipal.addCommand(conectar);
      fprincipal.setCommandListener(this);
   }
   public void startApp() {
      pantalla.setCurrent(fprincipal);
   }
```

```
    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
        notifyDestroyed();
    }

    public void commandAction(Command c, Displayable d){
        if (c == salir){
            destroyApp(false);
        }
        else{
            com.conectar();
            fprincipal.removeCommand(conectar);
        }
    }

    public void setRespuesta(String res){
        txt.setText(res);
        fprincipal.append(txt);
    }

}


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

public class Comunicacion implements Runnable{

    private Thread t;
    private String respuesta;
    private HttpConnection c;
    private PruebaServlet midlet;

    public Comunicacion(PruebaServlet m) {
        respuesta = null;
        midlet = m;
    }

    public void conectar(){
        t = new Thread(this);
        t.start();
    }

    public void run(){
        try{
            String url =
"http://localhost:8090/Hogar/InfoHogar?nombre=yo&password=Java";
            c = (HttpConnection)Connector.open(url);
```

```
            c.setRequestProperty("Content-Language","es-ES");
            c.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
            c.setRequestProperty("Connection","close");
            c.setRequestMethod(HttpConnection.GET);
            InputStream is = c.openInputStream();
            procesarRespuesta(c,is);
         }
         catch (Exception e){
            System.out.println(e);
         }
      }

      public void procesarRespuesta(HttpConnection http, InputStream inst)
      throws IOException{
         if (http.getResponseCode() == HttpConnection.HTTP_OK){
            System.out.println("OK");
            int lon = (int) http.getLength();
            if (lon != -1){
               byte datos[] = new byte[lon];
               inst.read(datos,0,datos.length);
               respuesta = new String(datos);
            }
            else{
               ByteArrayOutputStream baos = new ByteArrayOutputStream();
               int ch;
               while ((ch = inst.read()) != -1)
                  baos.write(ch);
               respuesta = new String(baos.toByteArray());
               baos.close();
            }
            midlet.setRespuesta(respuesta);
         }
         else{
            System.out.println(http.getResponseMessage());
         }
      }

      public String getRespuesta(){
         return respuesta;
      }
}
```

Por su parte el *servlet* tiene la siguiente apariencia:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class Hogar extends HttpServlet{

public void doPost(HttpServletRequest req, HttpServletResponse res){
```

```
        try{    res.setContentType("text/plain");
                String nombre = req.getParameter("nombre");
                String contr = req.getParameter("password");
                String res1 = "Hola Java";
                String res2 = "Usuario desconocido";
                PrintWriter out;
                if (contr.compareTo("Java") == 0){
                        res.setContentLength(res1.length());
                        out = res.getWriter();
                        out.println(res1);
                }
                else{ res.setContentLength(res2.length());
                      out = res.getWriter();
                      out.println(res2);
                }
                out.close();
        }
        catch (Exception e){
                System.out.println(e);
        }
}

public void doGet(HttpServletRequest req, HttpServletResponse res){
        doPost(req,res);
}
}
```

El funcionamiento de este ejemplo es muy sencillo. El *servlet* únicamente comprueba si la contraseña (*password*) coincide con una predeterminada y en caso afirmativo devuelve un mensaje de bienvenida. En caso de que no coincida devuelve otro mensaje distinto.

7.4.3. Estado de Cierre

La conexión entra en este estado una vez que se termina la comunicación entre el cliente y el servidor invocando al método close().

7.5. Otras Conexiones

Como hemos visto en la Figura 7.1 el perfil MIDP implementa varias interfaces de comunicación además del HTTP. Estas interfaces no las vamos a tratar tan profundamente como hicimos con HttpConnection, pero sí daremos unas pequeñas nociones sobre las funcionalidades que nos proporcionan.

7.5.1. Interfaz HttpsConnection

```
public interface HttpsConnection extends HttpConnection
```

HTTPS es la versión segura del protocolo HTTP. Esta interfaz define los métodos necesarios para establecer una conexión de este tipo. En esta conexión, los parámetros de la petición se deben establecer antes de que ésta se envíe.

Un objeto de tipo HttpsConnection es devuelto invocando al método Connector.open(url) de la siguiente manera:

```
String url = "https://www.direccion.com";
HttpsConnection conexion = (HttpsConnection)Connector.open(url);
```

Algunos métodos de esta interfaz pueden lanzar la excepción CertificateException para indicar distintos fallos mientras se establece una conexión segura.

Los métodos que añade esta interfaz a los ya vistos en HttpConnection aparecen en la Tabla 7.15.

Método	Descripción
public int getPort()	Devuelve el número de puerto.
public SecurityInfo getSecurityInfo()	Devuelve la información de seguridad asociada con la conexión.

Tabla 7.15 Métodos de la interfaz HttpsConnection

7.5.2. Interfaz UDPDatagramConnection

```
public interface UDPDatagramConnection extends DatagramConnection
```

Esta interfaz representa una conexión basada en datagramas en los que se conoce su dirección final. Esta interfaz se usa cuando el parámetro *url* del método Connector.open(url) tiene el siguiente formato:

```
url = "datagram://<host>:<port>";
```

y la conexión se realiza de la siguiente manera :

```
UDPDatagramConnection dc =
UDPDatagramConnection)Connector.open(url);
```

Si la cadena de conexión omite el *host* y el *port*, el sistema deberá de localizar un puerto libre. La dirección y el puerto local pueden conocerse usando los métodos que proporciona esta interfaz (ver Tabla 7.16).

Método	Descripción
public String getLocalAddress()	Devuelve la dirección local.
public int getLocalPort()	Devuelve el puerto local.

Tabla 7.16 Métodos de la interfaz UDPDatagramConnection

7.5.3. Interfaz CommConnection

```
public interface CommConnection extends StreamConnection
```

Esta interfaz representa una conexión mediante un puerto serie. Tal y como su nombre indica, en esta conexión los *bits* de datos se transmiten secuencialmente, en serie. Esta conexión se establece cuando el parámetro *url* que le pasamos al método Connector.open(url) tiene el siguiente formato:

```
url = "comm:<port><parámetros>";
```

y la conexión se realiza de la siguiente manera :

```
CommConnection cc = (CommConnection)Connector.open(url);
```

Los parámetros van separados por ';' y pueden ser los que aparecen en la Tabla 7.17.

Parámetro	Descripción
baudrate	Velocidad de la conexión.
bitsperchar	Número de bits por carácter (7 u 8).
stopbits	Número de bits de parada por carácter (1 o 2).
parity	Paridad.
blocking	Estado on u off.
autocts	Estado on u off
autorts	Estado on u off

Tabla 7.17 Parámetros de una conexión mediante CommConnection

Por otro lado el nombre del puerto se debe indicar según el siguiente criterio:

- COM# para puertos RS-232.
- IR# para puertos IrDA IRCOMM.

donde '#' indica el número asignado al puerto.

Los métodos que posee esta interfaz aparecen en la Tabla 7.18.

Método	Descripción
public int getBaudRate()	Devuelve la velocidad de conexión del puerto serie.
public int setBaudeRate(int baudrate)	Establece la velocidad de conexión.

Tabla 7.18 Métodos de la interfaz CommConnection

7.5.4. Interfaz SocketConnection

```
public interface SocketConnection extends StreamConnection
```

Esta interfaz define una conexión entre *sockets* basados en *streams*. La conexión con el *socket* de destino tiene el siguiente formato:

```
url = "socket://<host>:<port>";
```

y la conexión se realiza de la siguiente manera :

```
SocketConnection sc = (SocketConnection)Connector.open(url);
```

Esta conexión está basada en la interfaz StreamConnection. Como sabemos, un StreamConnection puede ser tanto de entrada como de salida. Por otro lado si, por ejemplo, el sistema nos proporciona un sistema de comunicación *duplex*, para cerrar la comunicación a través del *socket* es necesario cerrar los *streams* de entrada y de salida. Si tan sólo cerramos el canal de entrada, podemos seguir mandando información por el de salida. Incluso una vez cerrados ambos, es posible volverlos a abrir si aún no se ha cerrado el *socket*.

Esta interfaz nos proporciona los métodos de la Tabla 7.19.

Método	Descripción
public String getAddress()	Devuelve la dirección remota del socket.
public String getLocalAddress()	Devuelve la dirección local del socket.
public int getLocalPort()	Devuelve el puerto local del socket.
public int getPort()	Devuelve el puerto remoto del socket.
public int getSocketOption(byte opcion)	Devuelve una opción de la conexión.
public void setSocketOption(byte opcion, int valor)	Establece una opción de la conexión.

Tabla 7.19 Métodos de la interfaz SocketConnection

Las opciones disponibles get/setSocketOption() aparecen en la Tabla 7.20.

Opción	Descripción
byte DELAY	Opción que habilita o inhabilita el algoritmo para operaciones con buffer pequeño.
byte KEEPALIVE	Opción que habilita o inhabilita la característica de mantener despierto el socket
byte LINGER	Opción que define el tiempo de espera para cerrar una conexión con datos pendientes en el stream de salida.
byte RCVBUF	Opción que define el tamaño del buffer de entrada.
byte SNDBUF	Opción que define el tamaño del buffer de salida.

Tabla 7.20 Opciones disponibles para get/setSocketOption()

7.5.5. Interfaz SecureConnection

```
public interface SecureConnection extends SocketConnection
```

Esta interfaz representa una conexión segura entre *sockets*. Esta conexión se consigue invocando al método Connector.open(url) de la siguiente forma:

```
String url = "ssl://<host>:<port>";
```

y la conexión se realiza de la siguiente manera :

```
SecureConnection ssc = (SecureConnection)Connector.open(url);
```

Si el establecimiento de la conexión falla, se lanzaría una excepción del tipo CertificateException.

Esta interfaz sólo añade el método de la Tabla 7.21.

Método	Descripción
public SecurityInfo getSecurityInfo()	Devuelve la información de seguridad asociada a la conexión.

Tabla 7.21 Métodos de la interfaz SecureConnection

7.5.6. Interfaz ServerSocketConnection

```
public interface ServerSocketConnection extends
StreamConnectionNotifier
```

Esta interfaz representa una conexión con un servidor de *sockets*. Este tipo de conexión se establece cuando invocamos al método Connector.open(url) sin especificar el *host*:

```
ServerSocketConnection ssc =
(ServerSocketConnection)Connector.open("socket://:00");
```

7.6. Ejemplo práctico

En este punto vamos a terminar de desarrollar la aplicación **Hogar**. Se van a aplicar los conocimientos adquiridos en este capítulo para modificar el *MIDlet* de manera que a través de la comunicación con un *servlet* se cargue toda la información relacionada con el estado de los dispositivos y las coordenadas de éstos en pantalla.

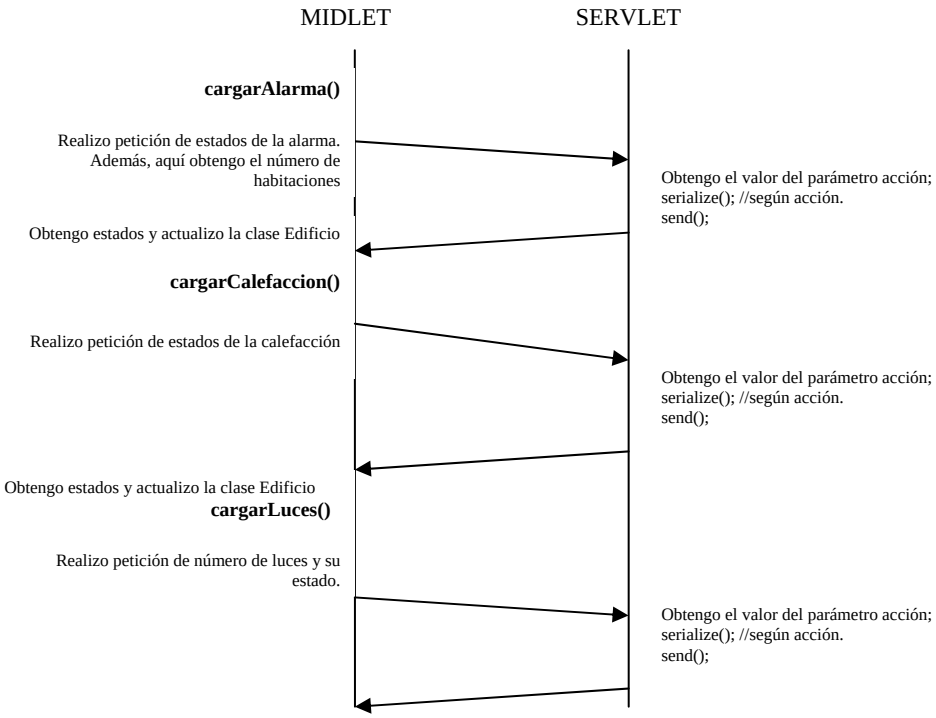

Diagrama 7.1 Fase de carga de estados

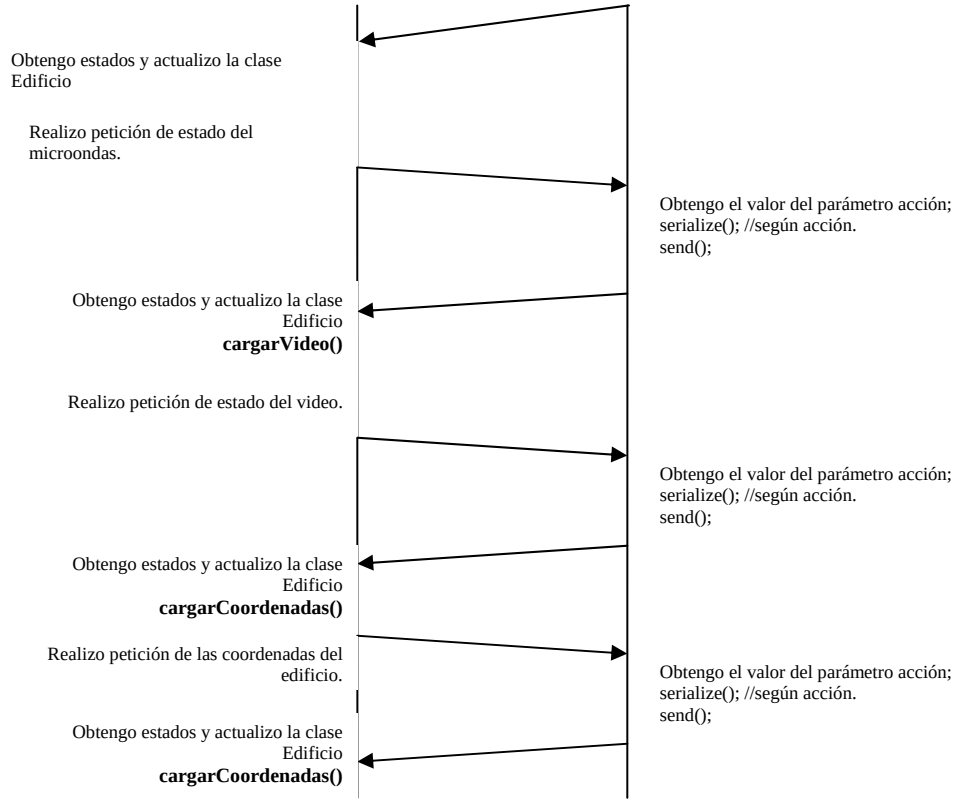

Diagrama 7.2 Continuación de la fase de carga de estados

Para ello vamos a tener que modificar un poco el código de nuestro *MIDlet*, sobre todo la parte encargada de cargar los estados y coordenadas: **CargaEstados**. Además vamos a crear una nueva clase llamada **EnviarEstados** que se encargará de enviar al *servlet* los cambios que se efectúen en cualquiera de los dispositivos. Antes de ver el código tanto del *MIDlet* como del *servlet*, veamos en el Diagrama 7.1 y continuación (Diagrama 7.2) la manera en que se realiza la comunicación entre ambos.

En esta fase de carga de estados, se establece como vemos, una comunicación entre el *MIDlet* y el servlet. El *MIDlet* informa al servlet de la respuesta que desea a través de un parámetro que le envía junto con la URL. Según el valor de éste parámetro, el servlet se encarga de serializar (pasar a un array de bytes) la información deseada y de enviarla al *MIDlet*. Éste, cuando recibe la información solicitada, actualiza la clase Edificio y realiza la siguiente petición.

También se establece una comunicación entre el *MIDlet* y el *servlet* cuando el usuario efectúa algún cambio en el estado de cualquier dispositivo. En este caso, el *MIDlet* envía junto con la URL y el parámetro que indica que se va a realizar alguna modificación de estados (parámetro accion = 6), otros parámetros adicionales indicando el dispositivo modificado y la información correspondiente al nuevo estado.

7.6.1. Código del *MIDlet*

El código de las clases encargadas de la comunicación con el *servlet* lo podemos ver a continuación:

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

public class EnviarEstados {

    private HttpConnection hc;
    private StringBuffer cad;

    public EnviarEstados() {
        hc = null;
        cad = null;
    }

    public void sendTemperatura(int hab, int temp) {
        try{  cad = new
                StringBuffer("http://localhost:8090/proyecto/Hogar?accion=6&tipo=1&");
            String chab = "p1="+hab;
            String ctemp = "&p2="+temp;
            cad.append(chab);
            cad.append(ctemp);
            String url = cad.toString();
            hc = (HttpConnection)Connector.open(url);
            hc.setRequestProperty("Content-Language","es-ES");
            hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
            hc.setRequestMethod(HttpConnection.GET);
            if (hc.getResponseCode() == HttpConnection.HTTP_OK)
                System.out.println("Peticion enviada correctamente");
            else  System.out.println("Peticion no recibida");
        }
```

```
        catch (Exception e) {
            System.out.println(e);
        }
    }

    public void sendEstAlarma(int hab, boolean alar) {
        try{  cad = new StringBuffer("http://localhost:8090/proyecto/Hogar?accion=6&tipo=2&");
            String chab = "p1="+hab;
            String calar = "&p2="+alar;
            cad.append(chab);
            cad.append(calar);
            String url = cad.toString();
            hc = (HttpConnection)Connector.open(url);
            hc.setRequestProperty("Content-Language","es-ES");
            hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
            hc.setRequestMethod(HttpConnection.GET);
            if (hc.getResponseCode() == HttpConnection.HTTP_OK)
                System.out.println("Peticion enviada correctamente");
            else  System.out.println("Peticion no recibida");
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }

    public void sendEstLuz(int luz, boolean est) {
        try{  cad = new
    StringBuffer("http://localhost:8090/proyecto/Hogar?accion=6&tipo=3&");
            String cluz = "p1="+luz;
            String cest = "&p2="+est;
            cad.append(cluz);
            cad.append(cest);
            String url = cad.toString();
            hc = (HttpConnection)Connector.open(url);
            hc.setRequestProperty("Content-Language","es-ES");
            hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
            hc.setRequestMethod(HttpConnection.GET);
            if (hc.getResponseCode() == HttpConnection.HTTP_OK)
                System.out.println("Peticion enviada correctamente");
            else  System.out.println("Peticion no recibida");
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }

    public void sendEstMic(int nivel, int tiempo) {
        try{  cad = new
    StringBuffer("http://localhost:8090/proyecto/Hogar?accion=6&tipo=4&");
            String cnivel = "p1="+nivel;
            String cpot = "&p2="+tiempo;
```

```
                cad.append(cnivel);
                cad.append(cpot);
                String url = cad.toString();
                hc = (HttpConnection)Connector.open(url);
                hc.setRequestProperty("Content-Language","es-ES");
                hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
                hc.setRequestMethod(HttpConnection.GET);
                if (hc.getResponseCode() == HttpConnection.HTTP_OK)
                    System.out.println("Peticion enviada correctamente");
                else  System.out.println("Peticion no recibida");
            }
            catch (Exception e) {
                System.out.println(e);
            }
        }

        public void sendEstVideo(int cadena) {
            try{
                cad = new StringBuffer("http://localhost:8090/proyecto/Hogar?accion=6&tipo=5&");
                String ccad = "p1="+cadena;
                cad.append(ccad);
                String url = cad.toString();
                hc = (HttpConnection)Connector.open(url);
                hc.setRequestProperty("Content-Language","es-ES");
                hc.setRequestProperty("User-Agent","Profile/MIDP-2.0
                    Configuration/CLDC-1.0");
                hc.setRequestMethod(HttpConnection.GET);
                if (hc.getResponseCode() == HttpConnection.HTTP_OK)
                    System.out.println("Peticion enviada correctamente");
                else  System.out.println("Peticion no recibida");
            }catch (Exception e) {
                System.out.println(e);
            }
        }
}

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

// Comentario 1
public class CargaEstados extends Form implements
        CommandListener, Runnable {
    private Gauge carga;
    private int indice, numluces, numHabitaciones;
    private boolean[] estluces, estAlarma;
    private int[] estCalefaccion;
    private int[][] coord, ady, coordTemp, coordLuz;
```

```
   private boolean mic, video;
   private Command cancelar, aceptar, salir;
   private Thread t;
   private Hogar midlet;
// Fin comentario 1

      public CargaEstados(Hogar midlet) {
            super("Cargando información");
            this.midlet = midlet;
            carga = new Gauge("Pulse Aceptar",false,15,0);
            cancelar = new Command("Cancelar",Command.CANCEL,1);
            aceptar = new Command("Aceptar",Command.OK,1);
            salir = new Command("Salir",Command.EXIT,1);
            this.append(carga);
            this.addCommand(aceptar);
            this.setCommandListener(this);
   }

   public void start() {
            t = new Thread(this);
            t.start();
   }

   // Comentario 2
   public void run() {
            carga.setLabel("Comprobando alarma");
            verEstadoAlarma();
            carga.setValue(3);
            carga.setLabel("Comprobando calefacción");
            verEstadoCalefaccion();
            carga.setValue(6);
            carga.setLabel("Comprobando luces");
            verEstadoLuz();
            carga.setValue(9);
            carga.setLabel("Comprobando microondas");
            verEstadoMicroondas();
            carga.setValue(12);
            carga.setLabel("Comprobando vídeo");
            verEstadoVideo();
            carga.setValue(15);
            carga.setLabel("Estableciendo coordenadas");
            setCoordenadas();
            carga.setLabel("Carga Completa");
            try{ t.sleep(1000);
            }
            catch (Exception e) {
                  System.out.println(e);
            }
            midlet.veropciones();
   }
   // Fin comentario 2
```

```
public void commandAction(Command c, Displayable d) {
        if (c==cancelar) {

        }
        else if (c==aceptar) {
            this.removeCommand(aceptar);
            this.addCommand(cancelar);
            this.start();
        }
}
```

```
 // Comentario 3
private void verEstadoAlarma() {
        cargarAlarma();
        midlet.edificio.setNumHabitaciones(numHabitaciones);
        midlet.edificio.setAlarma(estAlarma);
        midlet.setInfoAlarma2(estAlarma);
        try {
            t.sleep(1000);
        }
        catch(Exception e) {
            System.out.println("Error en estado alarma");
        }
   }
// Fin comentario 3

private void verEstadoCalefaccion() {
        cargarCalefaccion();
        midlet.edificio.establecerTemperatura(estCalefaccion);
        try{ t.sleep(1000);
        }
        catch(Exception e) {
            System.out.println("Error en estado calefaccion");
        }
}

private void verEstadoMicroondas() {
        cargarMic();
        midlet.edificio.setMicroondas(mic);
        try{ t.sleep(1000);
        }
        catch(Exception e) {
            System.out.println("Error en estado microondas");
        }
}

private void verEstadoVideo() {
        cargarVideo();
        midlet.edificio.setVideo(video);
        try{ t.sleep(1000);
```

```
            }
            catch(Exception e) {
                System.out.println("Error en estado video");
            }
    }

    public void verEstadoLuz() {
            cargarNumLuces();
            midlet.edificio.setNumLuces(numluces);//Envio el número de luces;
            midlet.edificio.setEstadoLuz(estluces);
            midlet.setInfoLuz1(estluces);
    }

    public void setCoordenadas() {
            try{cargarCoordenadas();
                midlet.edificio.establecerAdyacencia(ady);
                midlet.edificio.establecerCoord(coord);
                midlet.edificio.setCoordTemp(coordTemp);
                midlet.edificio.setCoordLuz(coordLuz);
                midlet.setInfoLuz2(coordLuz);
                midlet.setInfoAlarma1(coordTemp);
                coord = null;
                ady = null;
                coordLuz = null;
                coordTemp = null;
            }
            catch (Exception e) {
                System.out.println("Error al establecer coordenadas");
            }
    }
```

```
// Comentario 4
    public void cargarNumLuces() {
            try {
                String url = "http://localhost:8090/proyecto/Hogar?accion=0";
                HttpConnection hc = (HttpConnection)Connector.open(url);
                hc.setRequestProperty("Content-Language","es-ES");
                hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
                hc.setRequestProperty("Content-Type","application/octect-stream");
                hc.setRequestProperty("Connection","close");
                hc.setRequestMethod(HttpConnection.GET);
                System.out.println("Conexion establecida y peticion enviada");
                InputStream is = hc.openInputStream();
                procesar(0,hc,is);
            }
            catch (Exception e) {
                System.out.println(e);
            }
    }
// Fin comentario 4
```

```
   public void cargarMic() {
        try {
             String url = "http://localhost:8090/proyecto/Hogar?accion=1";
             HttpConnection hc = (HttpConnection)Connector.open(url);
             hc.setRequestProperty("Content-Language","es-ES");
             hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
             hc.setRequestProperty("Content-Type","application/octect-stream");
             hc.setRequestProperty("Connection","close");
             hc.setRequestMethod(HttpConnection.GET);
             System.out.println("Conexion establecida y peticion enviada");
             InputStream is = hc.openInputStream();
             procesar(1,hc,is);
        }
        catch (Exception e) {
             System.out.println(e);
        }
}

public void cargarVideo() {
        try {
             String url = "http://localhost:8090/proyecto/Hogar?accion=2";
             HttpConnection hc = (HttpConnection)Connector.open(url);
             hc.setRequestProperty("Content-Language","es-ES");
             hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
             hc.setRequestProperty("Content-Type","application/octect-stream");
             hc.setRequestProperty("Connection","close");
             hc.setRequestMethod(HttpConnection.GET);
             System.out.println("Conexion establecida y peticion enviada");
             InputStream is = hc.openInputStream();
             procesar(2,hc,is);
        }
        catch (Exception e) {
             System.out.println(e);
        }
}

public void cargarAlarma() {
        try {
             String url = "http://localhost:8090/proyecto/Hogar?accion=3";
             HttpConnection hc = (HttpConnection)Connector.open(url);
             hc.setRequestProperty("Content-Language","es-ES");
             hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
             hc.setRequestProperty("Content-Type","application/octect-stream");
             hc.setRequestProperty("Connection","close");
             hc.setRequestMethod(HttpConnection.GET);
             System.out.println("Conexion establecida y peticion enviada");
             InputStream is = hc.openInputStream();
             procesar(3,hc,is);
        }
        catch (Exception e) {
             System.out.println(e);
```

```
        }
    }

    public void cargarCalefaccion() {
        try {
            String url = "http://localhost:8090/proyecto/Hogar?accion=4";
            HttpConnection hc = (HttpConnection)Connector.open(url);
            hc.setRequestProperty("Content-Language","es-ES");
            hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
            hc.setRequestProperty("Content-Type","application/octect-stream");
            hc.setRequestProperty("Connection","close");
            hc.setRequestMethod(HttpConnection.GET);
            System.out.println("Conexion establecida y peticion enviada");
            InputStream is = hc.openInputStream();
            procesar(4,hc,is);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }

    public void cargarCoordenadas() {
        try {
            String url = "http://localhost:8090/proyecto/Hogar?accion=5";
            HttpConnection hc = (HttpConnection)Connector.open(url);
            hc.setRequestProperty("Content-Language","es-ES");
            hc.setRequestProperty("User-Agent","Profile/MIDP-2.0 Configuration/CLDC-1.0");
            hc.setRequestProperty("Content-Type","application/octect-stream");
            hc.setRequestProperty("Connection","close");
            hc.setRequestMethod(HttpConnection.GET);
            System.out.println("Conexion establecida y peticion enviada");
            InputStream is = hc.openInputStream();
            procesar(5,hc,is);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
```

// Comentario 5

```
    public void procesar(int tipoPeticion, HttpConnection http, InputStream inst)
        throws IOException {
        try{if (http.getResponseCode() == HttpConnection.HTTP_OK) {
                System.out.println("OK");
                int lon = (int) http.getLength();
                byte datos[];
                if (lon != -1) {
                    System.out.println("Longitud conocida");
                    datos = new byte[lon];
                    System.out.println("Creado array de lon "+lon);
                    inst.read(datos,0,lon);
```

```
            System.out.println("Datos leidos");
        }
        else {
            System.out.println("Longitud no conocida");
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int ch;
            while ((ch = inst.read()) != -1)
                baos.write(ch);
            datos = baos.toByteArray();
            baos.close();
        }
// Fin comentario 5

// Comentario 6
        ByteArrayInputStream bais = new ByteArrayInputStream(datos);
        DataInputStream dis = new DataInputStream(bais);
        switch(tipoPeticion) {
            case 0:{ numluces = dis.readInt();
                     estluces = new boolean[numluces];
                     for (int i=0;i<numluces;i++) {
                         estluces[i] = dis.readBoolean();
                     }
                     break;}
            case 1:{ mic = dis.readBoolean();
                     break;}
            case 2:{ video = dis.readBoolean();
                     break;}
            case 3:{ numHabitaciones = dis.readInt();
                     estAlarma = new boolean[numHabitaciones];
                     for (int i=0;i<numHabitaciones;i++) {
                         estAlarma[i] = dis.readBoolean();
                     }
                     break;}
            case 4:{ estCalefaccion = new int[numHabitaciones];
                     for (int i=0;i<numHabitaciones;i++) {
                         estCalefaccion[i] = dis.readInt();
                     }
                     break;}
            case 5:{ coord = new int[numHabitaciones][4];
                     for (int i=0;i<numHabitaciones;i++) {
                         for (int j=0;j<4;j++) {
                             coord[i][j] = dis.readInt();
                         }
                     }
                     ady = new int[numHabitaciones][4];
                     for (int i=0;i<numHabitaciones;i++) {
                         for (int j=0;j<4;j++) {
                             ady[i][j] = dis.readInt();
                         }
                     }
                     coordTemp = new int[numHabitaciones][2];
```

```
                                    for (int i=0;i<numHabitaciones;i++) {
                                        for (int j=0;j<2;j++) {
                                            coordTemp[i][j] = dis.readInt();
                                        }
                                    }
                                    coordLuz = new int[numluces][2];
                                    for (int i=0;i<numluces;i++) {
                                        for (int j=0;j<2;j++) {
                                            coordLuz[i][j] = dis.readInt();
                                        }
                                    }
                                    break;}
                            }
                            dis.close();
                            bais.close();
                        }

                    }
                    catch (Exception e) {
                    }
            }
// Fin comentario 6

}
```

La clase EnviarEstados posee métodos que se encargan de enviar los estados de los distintos dispositivos al *servlet*. Estos métodos tienen la nomenclatura send*NombreDispositivo*. Lo único que hace cada método es conectarse al *servlet* usando distintos parámetros en la URL que son los que poseen la información sobre el dispositivo.

Por su parte, la clase CargarEstados ha sido modificada para que sea capaz de recibir por parte del *servlet*, el estado actual de los dispositivos y las coordenadas correspondientes. Para ello se ha cambiado el código donde creábamos ficticiamente todos los estados y coordenadas por métodos que realizan peticiones al *servlet* y, de esta manera, se va creando dinámicamente la clase Edificio con toda la información necesaria.

- Comentario 1

Este código representa al constructor de la clase CargaEstados. Aquí creamos la pantalla de carga del dispositivo. Esta pantalla está formada por un gauge no interactivo que irá mostrando el nivel de la carga.

- Comentario 2

Aquí podemos observar el cuerpo principal de la clase CargaEstados. En este cuerpo vemos el estado de cada dispositivo y vamos actualizando el nivel del gauge de carga. Una vez que se finaliza la carga de estados, se invoca al método midlet.veropciones() para ir al menú principal.

- Comentario 3

Los métodos que comprueban el estado de cada dispositivo son prácticamente iguales. Vamos a ver cómo es uno de ello. En este caso comprobaremos el estado de la alarma. Para ello se invoca al método cargarAlarma() que es el que realiza la conexión con el *servlet* y le realiza la petición correcta. Al finalizar el método cargarAlarma() ya tenemos los estados cargados del servidor en el array estAlarma[], además del número de habitaciones en la variable numHabitaciones. Lo único que hay que hacer es enviar esta información a la clase Edificio mediante los métodos correspondientes.

- Comentario 4

Los métodos cargarDispositivo*()* tienen también la misma apariencia. Aquí se realiza la conexión con el servidor y se envía con la URL la petición adecuada. Se inicializan los campos de cabecera y se procesa la petición a través del método procesar(). El primer parámetro que recibe este método indica el tipo de petición que se ha enviado y, de esta manera, procesa correctamente la información recibida.

- Comentario 5

Esta parte de código del método procesar(int, HttpConnection, InputStream) realiza la lectura de la información recibida y la guarda en el array de bytes datos.

- Comentario 6

Aquí se realiza la deserialización de la información recibida. Dependiendo del tipo de petición enviada se recibe un tipo de información u otro. Aquí tratamos correctamente la información recibida y actualizamos las variables de los dispositivos correctamente.

Tanto la clase Edificio como el resto de clases que conforman el *MIDlet* han sufrido pocas variaciones. A la clase Edificio simplemente se le ha añadido la clase EnviarEstados. Ahora, a la vez que actualizamos el estado de un dispositivo, enviamos la información de ese dispositivo al servlet para que también lo actualice.

7.6.2. Código del *servlet*

Por último vamos a ver el código del *servlet*, que es el que posee toda la información sobre nuestro edificio y estudiaremos cómo envía toda esa información al *MIDlet*. Como puede observarse, el *servlet* lleva codificadas las coordenadas de posicionamiento de los componentes; indudablemente, una buena práctica de programación nos llevaría a introducir dichos datos en un fichero de propiedades, con el fin de hacer nuestro *servlet* más parametrizable. No obstante, nuestro objetivo se centra en estudiar las técnicas de programación con J2ME, motivo por el que se ha optado por este mecanismo de programación.

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
```

```
public class Hogar extends HttpServlet {

    private int numLuces, numHabitaciones;
    private int[] estadoCalefaccion;
    private boolean[] estadoLuces, estadoAlarma;
    private boolean estadoMic, estadoVideo;
    private int[][] coord, ady, coordTemp, coordluz;

    public void init(ServletConfig c){
        try{
            super.init(c);
            numLuces = 6;
            numHabitaciones = 4;

            estadoLuces = new boolean[numLuces];
            for (int i=0;i<numLuces;i++){
                estadoLuces[i]=true;
            }
            estadoMic = true;
            estadoVideo = true;

            estadoAlarma = new boolean[numHabitaciones];
            estadoCalefaccion = new int[numHabitaciones];
            for (int i=0;i<numHabitaciones;i++){
                estadoAlarma[i]=true;
                estadoCalefaccion[i]=19;
            }
            coord = new int[numHabitaciones][4];
            coord[0][0] = 9;coord[0][2] = 86;
            coord[0][1] = 12;coord[0][3] = 47;
            coord[1][0] = 98;coord[1][1] = 14;
            coord[1][2] = 77;coord[1][3] = 83;
            coord[2][0] = 11;coord[2][1] = 65;
            coord[2][2] = 84;coord[2][3] = 93;
            coord[3][0] = 100;coord[3][1] = 102;
            coord[3][2] = 73;coord[3][3] = 56;
            //
            ady = new int[numHabitaciones][4];
            ady[0][0] = 0;ady[0][1] = 3;
            ady[0][2] = 2;ady[0][3] = 0;
            ady[1][0] = 0;ady[1][1] = 4;
            ady[1][2] = 0;ady[1][3] = 1;
            ady[2][0] = 1;ady[2][1] = 0;
            ady[2][2] = 4;ady[2][3] = 0;
            ady[3][0] = 2;ady[3][1] = 0;
            ady[3][2] = 0;ady[3][3] = 3;
            //
            coordTemp = new int[numHabitaciones][2];
            coordTemp[0][0] = 45;coordTemp[0][1] = 44;
            coordTemp[1][0] = 128;coordTemp[1][1] = 80;
```

```
                coordTemp[2][0] = 47;coordTemp[2][1] = 72;
                coordTemp[3][0] = 130;coordTemp[3][1] = 106;
                //
                coordluz = new int[numLuces][2];
                coordluz[0][0] = 45;coordluz[0][1] = 36;
                coordluz[1][0] = 103;coordluz[1][1] = 20;
                coordluz[2][0] = 132;coordluz[2][1] = 49;
                coordluz[3][0] = 64;coordluz[3][1] = 73;
                coordluz[4][0] = 55;coordluz[4][1] = 105;
                coordluz[5][0] = 150;coordluz[5][1] = 116;
        }
        catch(Exception e){
        }
    }

    public void doGet(HttpServletRequest request,HttpServletResponse response)
                throws ServletException, IOException {
        try{String parametro = request.getParameter("accion");
            if (parametro.compareTo("6") == 0){
                recibir(request);
                response.setStatus(response.SC_OK);
            }
            else send(response,parametro);
        }catch(Exception e){
        }
    }

    public void doPost(HttpServletRequest request,HttpServletResponse response)
throws ServletException, IOException {
    }

    private void send(HttpServletResponse response, String parametro)
                throws Exception{
        byte[] data;// = new byte[0];
        data = serialize(parametro);
        response.setStatus(response.SC_OK);
        response.setContentLength(data.length);
        response.setContentType("application/octect-stream");
        OutputStream os = response.getOutputStream();
        os.write(data);
        os.close();
    }

    private byte[] serialize(String parametro) throws IOException{
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream(bout);
        if (parametro.compareTo("0") == 0){
            dout.writeInt(numLuces);
            for (int i=0;i<numLuces;i++){
                dout.writeBoolean(estadoLuces[i]);
            }
```

```
            }
            if (parametro.compareTo("1") == 0){
                dout.writeBoolean(estadoMic);
            }
            if (parametro.compareTo("2") == 0){
                dout.writeBoolean(estadoVideo);
            }
            if (parametro.compareTo("3") == 0){
                dout.writeInt(numHabitaciones);
                for (int i=0;i<numHabitaciones;i++){
                    dout.writeBoolean(estadoAlarma[i]);
                }
            }
            if (parametro.compareTo("4") == 0){
                for (int i=0;i<numHabitaciones;i++){
                    dout.writeInt(estadoCalefaccion[i]);
                }
            }
            if (parametro.compareTo("5") == 0){
                for (int i=0;i<numHabitaciones;i++){
                    for (int j=0;j<4; j++){
                        dout.writeInt(coord[i][j]);
                    }
                }
                for (int i=0;i<numHabitaciones;i++){
                    for (int j=0;j<4; j++){
                        dout.writeInt(ady[i][j]);
                    }
                }
                for (int i=0;i<numHabitaciones;i++){
                    for (int j=0;j<2; j++){
                        dout.writeInt(coordTemp[i][j]);
                    }
                }
                for (int i=0;i<numLuces;i++){
                    for (int j=0;j<2; j++){
                        dout.writeInt(coordluz[i][j]);
                    }
                }

            }
            dout.flush();
            return bout.toByteArray();
    }

        public void recibir(HttpServletRequest req){
            String tipoPet = req.getParameter("tipo");
            if (tipoPet.compareTo("1") == 0){
                String hab = req.getParameter("p1");
                String temp = req.getParameter("p2");
                estadoCalefaccion[Integer.parseInt(hab)] = Integer.parseInt(temp);
```

```
            }
            if (tipoPet.compareTo("2") == 0){
                String hab = req.getParameter("p1");
                String est = req.getParameter("p2");
                estadoAlarma[Integer.parseInt(hab)] = (Boolean.valueOf(est)).booleanValue();
            }
            if (tipoPet.compareTo("3") == 0){
                String luz = req.getParameter("p1");
                String est = req.getParameter("p2");
                estadoLuces[Integer.parseInt(luz)] = (Boolean.valueOf(est)).booleanValue();
            }
            if (tipoPet.compareTo("4") == 0){
                String nivel = req.getParameter("p1");
                String tiempo = req.getParameter("p2");
                estadoMic = true;
                //Establecer potencia y tiempo del microondas
            }
            if (tipoPet.compareTo("5") == 0){
                String cadena = req.getParameter("p1");
                estadoVideo = true;
                //Establecer grabación de la cadena elegida
            }
        }
}
```

Java a Tope: J2ME

(Java 2 Micro Edition)

El presente volumen introduce al lector en uno de los aspectos más fascinantes y potentes del lenguaje de programación Java: J2ME (Java 2 Micro Edition). La plataforma J2ME proporciona al desarrollador los medios necesarios para construir aplicaciones Java destinadas a ejecutarse en dispositivos con pocos recursos, principalmente teléfonos móviles y PDAs.

Con una descriptiva introducción en la que se estudian las diferencias fundamentales entre J2ME y sus «hermanas mayores» J2SE y J2EE, se profundiza a continuación sobre los conceptos teóricos que subyacen bajo esta plataforma, introduciéndolos paulatinamente mediante ejemplos prácticos diversos. A lo largo del texto se van creando dos aplicaciones que ilustran la práctica totalidad de los aspectos básicos en la programación de dispositivos de pequeña envergadura: arquitectura, configuraciones, perfiles, entornos de desarrollo, paquetes específicos, limitaciones, etc.

La primera de estas aplicaciones (un juego interactivo de ping-pong) nos enseña el manejo de *sprites* y las posibilidades interactivas en la construcción de juegos de destreza y *arcade*. La segunda (control remoto de los electrodomésticos de una casa) muestra las posibilidades de comunicación entre los teléfonos móviles y un servidor central que recibe las órdenes y actúa sobre los dispositivos electromecánicos del hogar.

Universidad de Málaga

ISBN 84-688-4704-6

9 788468 847047 90000