INTRODUCCIÓN A J2ME

Javier Navarro Melchor

Javier San Pablo Hernández

Departamento de Informática y Automática

Universidad de Salamanca

Información de los autores:

Javier Navarro Melchor

Estudiante de 3º de Ingeniería Técnica en Informática de Sistemas

Facultad de Ciencias – Universidad de Salamanca

Plaza de los Caídos S/N – 37008 – Salamanca

chaky_navarro@hotmail.com

Javier San Pablo Hernández

Estudiante de 3º de Ingeniería Técnica en Informática de Sistemas

Facultad de Ciencias – Universidad de Salamanca

Plaza de los Caídos S/N – 37008 – Salamanca

javier_san_pablo@hotmail.com

Resumen

Este documento realiza una presentación del lenguaje de programación J2ME. En él, se inicia al lenguaje a través de unas nociones básicas y la descripción de un MIDlet, haciendo hincapié en el uso de interfaces de usuario, manejo de la memoria persistente y el acceso a redes a través de esta tecnología.

No se trata de una guía para aprender a programar en J2ME, sino introducción básica de las características que presenta el lenguaje J2ME.

Abstract

This document makes a presentation of programming language J2ME. It starts with few basic notions of the language and the description of a MIDlet, insisting on the use of user's interfaces, managing of persistent memory and the access to networks using this technology.

It is not a guide to learn to program in J2ME. It is a basic introduction to features that J2ME language offers.

Tabla de Contenidos

Índice de Figuras

Índice de Contenidos

1. Introducción

Al principio de los 90, Sun Microsystems creó un nuevo lenguaje de programación llamado Oak como parte de un proyecto de investigación para construir productos electrónicos que dependan principalmente del software. El primer prototipo para Oak fue un controlador portable llamado Star7, un pequeño dispositivo *handheld* con una pantalla *touchscreen* LCD que tenía incorporado soporte a redes inalámbricas y comunicaciones infrarrojas. El software para este tipo de dispositivos necesitaba ser extremadamente confiable y no debía hacer excesivo uso de memoria ni requerir demasiada potencia en el procesador. Oak fue desarrollado como resultado de la experiencia del equipo de desarrollo con el lenguaje C++, el cual, a pesar de tener muchas grandes características, demostró que era un lenguaje complejo y ocasionaba que los programadores comentan fácilmente errores y eso afectaba la confiabilidad del software.

Oak fue diseñado para quitar o reducir la posibilidad de que los programadores comentan errores, ¿cómo? detectando la mayoría de errores en tiempo de compilación y quitando algunas de las características del lenguaje C++ (como punteros y la administración de memoria controlada por el programador) que eran los problemas más comunes.

Desafortunadamente, el mercado para el tipo de dispositivos que el nuevo lenguaje fue creado no se desarrolló tanto como Sun Microsystems esperaba, y al final ningún dispositivo basado en Oak fue vendido a los clientes. Sin embargo, al mismo tiempo, el inicio del conocimiento público de Internet produjo un mercado para el software de navegación para Internet (los navegadores *Web*). En respuesta a esto, Sun Microsystems renombró el lenguaje de programación Oak a Java y lo usó para desarrollar un navegador multiplataforma llamado *HotJava*. También le dio la licencia de Java a Netscape, quienes lo incorporaron en su navegador, que por entonces era el más popular en el mercado, luego fueron incorporados los Java *applets*.

En un par de años, las capacidades multiplataforma del lenguaje de programación Java y su potencia como plataforma de desarrollo para aplicaciones que podían ser escritas una vez y ejecutadas en diversos sistemas Windows y Unix, había despertado el interés de usuarios finales, porque vieron en ella una manera de reducir los costos del desarrollo de software.

Sun Microsystems rápidamente expandió el alcance y tamaño de la plataforma Java. Esta plataforma extendida incluyó un conjunto más complejo de librerías de interfaces de usuario que aquellas que usaran para construir applets, además con un conjunto de características de computación distribuida y seguridad mejorada.

Con el tiempo, Sun Microsystems liberó la primera versión de la plataforma Java 2, había sido necesario dividirla en varias piezas. La funcionalidad principal, estimado como el mínimo soporte requerido para cualquier ambiente Java, estaba empaquetada en el Java 2 *Standard Edition* (J2SE).

Muchos paquetes opcionales pueden ser agregados al J2SE para satisfacer requerimientos específicos para aplicaciones particulares, como extensiones seguras de *sockets* que permitan el comercio electrónico. Sun Microsystems también respondió al incremento del interés de usar Java para el desarrollo a un nivel empresarial, y ambientes de servidores de aplicaciones con la plataforma Java 2 *Enterprise Edition* (J2EE), el cual incorpora nuevas tecnologías como *servlets*, *Enterprise JavaBeans*, *JavaServer pages*, etc.

Irónicamente, mientras Sun Microsystems estaba desarrollando Java para Internet y para la programación comercial, la demanda empezó a crecer en los dispositivos pequeños e incluso en tarjetas inteligentes, retornando Java a sus raíces.

Sun Microsystems respondió a esta demanda creando varias plataformas Java con funcionalidades reducidas, cada una hecha a la medida de un segmento vertical y específico del mercado.

Estas plataformas reducidas están todas basadas en el JDK 1.1, el predecesor de la plataforma Java 2, y cada una tiene una estrategia diferente al problema de reducir la plataforma para acomodarla a los recursos disponibles. Por lo tanto, cada una de estas plataformas de funcionalidad reducida representan una solución *ad hoc* al problema. Por ello es que aparece la plataforma J2ME, para reemplazar todas esas plataformas reducidas basadas en el JDK 1.1 y crear una sola solución basada en Java 2.

Este trabajo aborda una introducción a la programación en J2ME comenzando con unas nociones básicas sobre los componentes que forman esta tecnología (máquinas virtuales, configuraciones y perfiles). Se hará una introducción al concepto de MIDlet así como a su ciclo de vida y su estructura básica. Se exponen los métodos de desarrollo de las interfaces de usuario, de las posibilidades que ofrece para el manejo de la memoria persistente y el soporte de la tecnología de conectividad que ofrecen estos dispositivos. Todo ello orientado a los servicios que ofrece CLCD/MIDP en la que se basan los dispositivos de más baja capacidad como son los teléfonos móviles, apartándose de otras configuraciones y perfiles existentes.

2. Nociones básicas de J2ME

Los componentes que forman parte de esta tecnología son los siguientes:

- Por un lado 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.
- Perfiles, que son unas bibliotecas Java de clases específicas orientadas a implementar funcionalidades de más alto nivel para familias específicas de dispositivos.

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.

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

a) Máquina virtual.

b) Configuración.

c) Perfil.

d) Paquetes Opcionales.

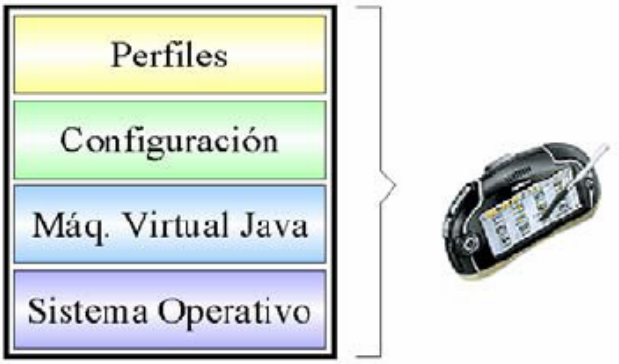

Figura 2.1 Entorno de ejecución

2.1. 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.

Cada una de las configuraciones CLDC y CDC 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. 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 80Kb, 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.
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.

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.

Por esta razón los dispositivos que usen la configuración CLDC y KVM introducen un algoritmo de verificación de clases en dos pasos.

La KVM puede ser compilada y probada en 3 plataformas distintas: Solaris Operating Environment, Windows y 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 biblioteca de seguridad, referencias débiles, Interfaz Nativa de Java (JNI), invocación remota de métodos (RMI), Interfaz de depuración de la Máquina Virtual (JVMDI).

2.2. Configuraciones. CLDC

Una configuración es, básicamente, el conjunto mínimo de APIs Java que permiten desarrollar aplicaciones para un grupo de dispositivos. Estas APIs describen las características básicas, comunes a todos los dispositivos.

La configuración conocida por las siglas CLDC *(Connectec Limited Device Configuration,* Configuración de Dispositivo Conectado Limitado) es la única definida a día de hoy por Sun para J2ME contiene las clases e interfaces Java necesarios para desarrollar aplicaciones sobre dispositivos móviles con conexión inalámbrica *(wireless),* básicamente teléfonos móviles y dispositivos equivalentes (por ejemplo, los sistemas bidireccionales de mensajes, conocidos como *pagers* en Estados Unidos). En el futuro aparecerán más configuraciones a medida que nuevos tipos de dispositivos adopten el soporte nativo a Java como una de sus funcionalidades.

La configuración CLDC contiene cuatro los elementos básicos:

- Un subconjunto de los elementos básicos del lenguaje Java.
- Una parte de la funcionalidad de la máquina virtual Java.
- Las APIs básicas para el desarrollo de aplicaciones.
- Requisitos hardware de los dispositivos englobados en el CLDC.

Las limitaciones en el uso del lenguaje Java se centran esencialmente en la ausencia de soporte a operaciones con coma flotante (números reales) debido a que el hardware soportado por CLDC no cuenta con la capacidad de realizar estas operaciones. La única forma de realizarlas sería vía software, lo que supondría obtener un rendimiento muy pobre, por lo que es mejor evitar, en la medida de lo posible este tipo de operaciones.

Otra limitación en el lenguaje es la ausencia del método `Object.finalize()`, que se invoca cada vez que se elimina un objeto de memoria, con objeto de facilitar la liberación de los recursos asociados. Dado que CLDC no precisa que la máquina virtual de soporte a la finalización de objetos, este elemento no es necesario.

También hay cambios en el manejo de excepciones. Esto se debe a que buena parte de las excepciones dependen de cada dispositivo, por lo que deben ser definidas en cada caso y no de forma genérica. Esto hace que se reduzca mucho el número de errores y excepciones soportados de forma estándar sea bastante reducido.

Un elemento especialmente cuidado en la redefinición que CLDC hace de Java es la seguridad. Resulta obvio que el esquema de descarga y ejecución de aplicaciones en dispositivos tan personales como puedan ser teléfonos móviles, implica tomar precauciones para garantizar la integridad y confidencialidad de dispositivo, aplicaciones y datos. En realidad no es algo especialmente novedoso, ya el uso de *applets* (pequeños programas escritos en Java que se descargan y ejecutan sobre un navegador *Web*) supuso imponer serias restricciones a la funcionalidad disponible en la máquina virtual Java encargada de ejecutarlos, por medio de la creación de una zona segura para hacerlo o *sandbox.*

Básicamente, hay una serie de funcionalidades sensibles o críticas que quedan fuera del alcance de los programas y una serie de condiciones previas que debe cumplir cualquier aplicación para poder ser utilizada dentro de un dispositivo:

- Los ficheros que contienen las clases deben haber sido verificados como una aplicación válida Java en un paso adicional dentro del proceso de desarrollo.
- Sólo se puede hacer uso del API CLDC predefinido. Por lo que sólo está accesible el código nativo de las funcionalidades de ese API.
- No se permite el uso de cargadores de clases definidos por el usuario.

Todas estas restricciones de seguridad son de muy bajo nivel, ya que CLDC es en realidad una especificación muy básica. En la especificación de perfil, que veremos más adelante, se incorporan algunas restricciones adicionales.

Por último, los requisitos mínimos hardware que definen a un dispositivo englobado en la configuración CLDC son disponer de:

- 160Kb de memoria disponible para Java. Esta memoria está dividida en dos áreas: 128Kb de memoria no volátil para la máquina virtual Java y las librerías del API CLDC, y 32Kb de memoria volátil para el entorno de ejecución.
- Un procesador de 16 *bits*.
- Bajo consumo, por regla general por medio del uso de baterías.
- Conexión a red, normalmente con un ancho de banda de 9.600 bps o inferior (típico de conexiones GSM).

2.3. Perfiles. MIDP

En la jerarquía definida en J2ME, los perfiles *(profiles)* se apoyan sobre las configuraciones, CLDC en este caso. Un perfil es una concreción aún más restrictiva de las APIs para centrarse en un conjunto muy reducido de dispositivos (incluso uno solo). Es decir, el perfil añade 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 con las de la configuración.

CLDC es la primera configuración definida en J2ME. MIDP (de *Mobile Information Device Profile*, o "Perfil de Dispositivo Móvil de información") es el primer perfil, hecho para describir dispositivos similares a teléfonos móviles o *pagers*. Estos dispositivos se describen a través de unos requisitos mínimos que se aplican a algunas de sus características:

Memoria:

- Al menos 128Kb de memoria no volátil para almacenar el API MIDP.
- Al menos 32Kb de memoria volátil para el sistema de ejecución Java.
- Un mínimo de 8Kb de memoria persistente para el almacenamiento de información por parte de los programas.

Este último es el principal requisito añadido en este capítulo por el perfil.

Entrada de datos: un dispositivo MIDP debe tener un teclado o una pantalla táctil, o ambos. Aunque el ratón no forma parte de los medios de entrada reconocidos, un puntero (como en las PDAs) sí es un sistema de entrada válido ya que se aplica sobre una pantalla táctil.

Pantalla: al referirse a un tipo de dispositivos con grandes limitaciones de visualización, el perfil MIDP es muy restrictivo en este capítulo. La pantalla mínima es de 96x54 píxeles, con dos colores (blanco y negro, o activado y desactivado). Además con una relación entre altura y anchura de 1:1, es decir que la pantalla debe parecer cuadrada, lo que supone usar unos píxeles alargados (en un monitor son generalmente cuadrados, no rectangulares). Ello no impide que la mayor parte de los dispositivos rebasen ampliamente estas restricciones tanto en dimensiones como en colores.

Red: deben contar con acceso a una red bidireccional (para enviar y recibir) inalámbrica. Las exigencias mínimas de ancho banda son muy humildes: 9.600 bps.

Plataforma software: los dispositivos MIDP, no importa el hardware o el sistema operativo en el que se basen (uno de los grandes beneficios de Java) requieren la presencia de una serie de elementos en la gestión de la plataforma como son:

- La presencial de un *kernel* (núcleo de sistema operativo) capaz de hacerse cargo de tareas de bajo nivel como la gestión de interrupciones, excepciones y temporizadores.
- Un mecanismo para leer y escribir en memoria no volátil.
- Gestión del tiempo para fijar temporizadores y añadir marcas temporales a la información persistente.
- Acceso de lectura escrita a la conexión inalámbrica del dispositivo.
- Medios para obtener la información introducida por el usuario a través de los dispositivos de entrada.

- Soporte a gráficos basados en mapas de bits.
- Medios para gestionar el ciclo de vida de una aplicación (inicio, interrupción, reactivación y destrucción).

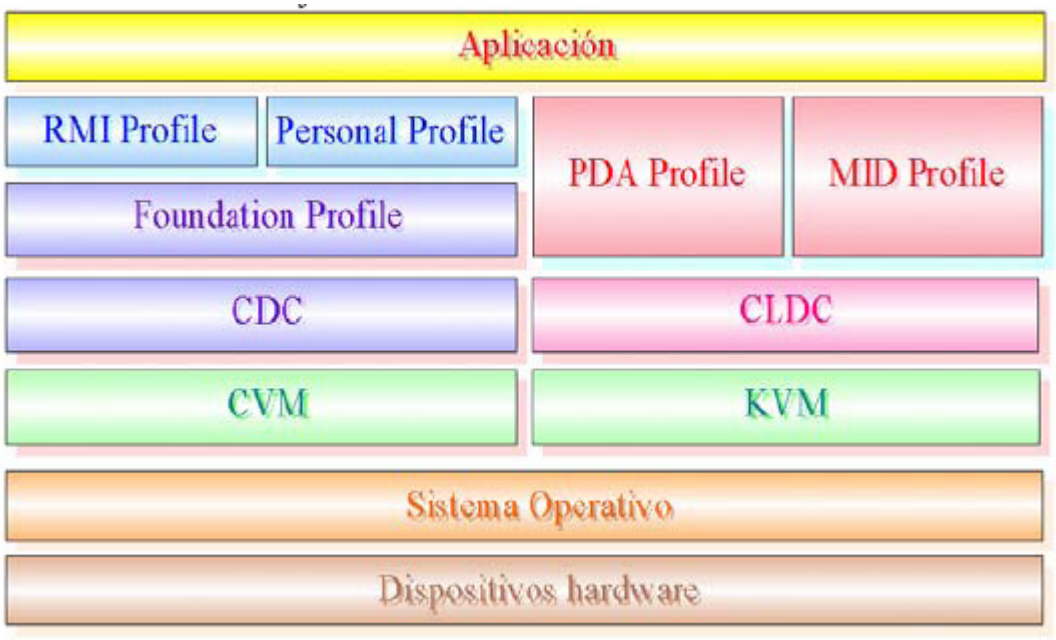

Figura 2.2 Arquitectura del entorno de ejecución de J2ME

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.

3. MIDlet

3.1. ¿Que es un MIDlet?

Un MIDlet es una aplicación J2ME desarrollada sobre el perfil MID, es decir, una aplicación para dispositivos móviles cuyas limitaciones caen dentro de la especificación MIDP. Gracias a la filosofía Java ("*write one, run anywhere*") podemos ejecutarlas sobre un amplio rango de dispositivos sin realizar ninguna modificación. Para que esta portabilidad sea realidad la especificación MIDP ha definido los siguientes requisitos:

- Todos los dispositivos de información móviles deben contar con un módulo software encargado de la gestión de los MIDlets (cargarlos, ejecutarlos…). Este software es denominado gestor de aplicaciones[1].
- Todos los MIDlet deben ofrecer la misma interfaz a todos los gestores de aplicaciones. Así, independientemente de la funcionalidad interna que implementen (un juego, una agenda,…), a los dispositivos pueden identificar a los MIDlet y realizar acciones sobre ellos. Este comportamiento se consigue mediante el mecanismo de herencia: todos los MIDlets heredan de la misma clase, javax.microedition.midlet.MIDlet.

3.2. Gestor de aplicaciones

El gestor de aplicaciones 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.

En la especificación no se índica que implementación concreta debe tener un gestor de aplicaciones. En su lugar, se describen las principales funciones que debe realizar, siendo finalmente los fabricantes quienes se encarguen de desarrollar gestores de aplicaciones específicos para sus dispositivos.

El gestor de aplicaciones realiza dos grandes funciones:

- Gestión de las aplicaciones.
- Gestión de estados de las aplicaciones.

3.2.1 Gestión de las aplicaciones

Las principales funciones que realiza un gestor de aplicaciones son la carga, instalación, ejecución, actualización y desinstalación del MIDlet.

1.**Carga**

Los gestores de aplicaciones proporcionan mecanismos para obtener MIDlet de distintas fuentes, bien sea Internet o desde un sistema de ficheros local. Sin esta funcionalidad solo tendríamos los MIDlet incluidos por el fabricante.

Entre los posibles mecanismos de carga que podemos encontrar tenemos:

- El cable.
- La tecnología inalámbrica.

[1] El gestor de aplicaciones recibe diversos acrónimos, entre ellos: AMS (*Application Management System*), JAM (*Java Application Management*) o GAJ (Gestor de Aplicaciones Java).

Se puede tener un dispositivo con más de un método para cargar MIDlet. En este caso, el proceso de carga incluye una fase intermedia de selección, donde podemos elegir cual usar.

2.**Instalación**
Una vez obtenido el MIDlet, el gestor de aplicaciones se encarga de su instalación. Este proceso puede variar dependiendo del tipo de dispositivo aunque suele incluir comprobaciones de seguridad y adaptaciones del modelo de almacenamiento concreto del dispositivo.

3.**Ejecución**
El gestor de aplicaciones es el encargado de iniciar la ejecución de los MIDlet. Además cuenta con facilidades para interrumpir y reanudar esta ejecución en función del estado del dispositivo.

4.**Actualización**
Para poder disfrutar de nuevas versiones de un MIDlet el gestor de aplicaciones proporciona facilidades de actualización y gestión de las versiones. Antes de instalar una nueva versión preguntara al usuario si este desea actualizar recuperando la antigua.

5.**Desinstalación**
Los gestores de aplicaciones permiten desinstalar los MIDlet, liberando los recursos reservados (memoria ocupada por el MIDlet y datos almacenados durante su utilización).

3.2.2 Gestión de estados del MIDlet

Los gestores de aplicaciones adaptan el funcionamiento de los MIDlet a los entornos concretos de ejecución de los dispositivos móviles. Los dispositivos se dedican a otras actividades además de ejecutar aplicaciones (por ejemplo por una llamada telefónica, enviar sms…). La especificación MIDP ha tenido en cuenta esta situación y ha dotado a los MIDlet de la capacidad de ser interrumpidos y lanzados repetidas veces sin que se vea afectado su comportamiento.

La interacción del entorno de ejecución con los MIDlet se realiza mediante llamadas que el gestor de aplicaciones realiza a unas funciones predeterminadas comunes a todos los MIDlet. Estas llamadas permiten que el MIDlet pase de un estado de ejecución a otro de forma controlada, manteniendo la integridad de la información y siempre de acuerdo a unas transiciones establecidas en el denominado ciclo de vida de los MIDlet. En cada una de estas transiciones el gestor de aplicaciones es el encargado de almacenar el estado de los MIDlet.

3.3. Ciclo de vida de un MIDlet

Los MIDlet pasan por distintos estados a lo largo de su ejecución, desde su inicio hasta su finalización. El ciclo de vida de un MIDlet describe todos sus estados, así como los eventos que disparan las transiciones entre ellos.

Los tres estados por los que pasa un MIDlet durante un ciclo de vida son:

1. Estado de Espera: Un MIDlet se encuentra en estado de espera cuando es interrumpido por el gestor de aplicaciones y también durante un breve periodo de tiempo al inicio de la aplicación, entre la llamada al método de creación del MIDlet y el inicio de la ejecución.

2. Estado Activado: Un MIDlet se encuentra activado durante su ejecución normal. Se puede llegar aquí tanto al inicio de una ejecución como después de una interrupción.
3. Estado Destruido: Cuando un MIDlet termina su ejecución pasa al estado de destruido.

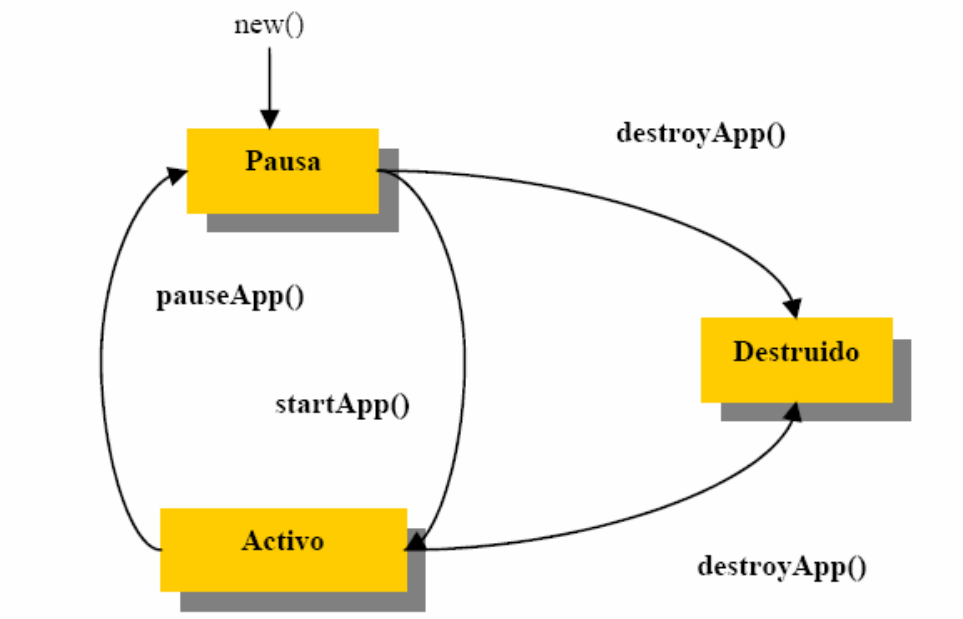

Figura 3.1 Ciclo de vida del MIDlet

Todos los cambios de estado de un MIDlet son realizados por el gestor de aplicaciones. Sin embargo, también los MIDlet pueden solicitar un cambio de estado al gestor de aplicaciones como consecuencia, por ejemplo, de una acción de usuario. Para facilitar estos cambios de estado los MIDlet cuentan con los siguientes métodos.

Estado	Llamar antes de pasar a…	Solicitud de cambio al estado…
Activo	`startApp()`	`resumeRequest()`
Espera	`pauseApp()`	`notifyPause()`
Destruido	`destroyApp()`	`notifyDestroy()`

Tabla 3.1 Funciones de cambio de estado

Es responsabilidad del programador incluir en estos métodos el código necesario para mantener la consistencia de la aplicación (borrar registros, inicializar variables, abrir o cerrar conexiones…).

3.3.1 Pasos que constituyen el ciclo de vida habitual de un MIDlet.

- Paso 1: La aplicación es creada por el gestor de aplicaciones, y tras pasar por el estado de espera, es movido inmediatamente al estado activado al retornar correctamente la llamada al método `startApp()`.
- Paso 2: El MIDlet se ejecuta normalmente hasta que surge un evento externo que obliga al gestor de aplicaciones a pararlo, justo después del retorno sin problemas de la función `pauseApp()`.

- Paso 3: El gestor de aplicaciones decide continuar con la ejecución de la aplicación por lo que llama de nuevo al método `startApp()`, y cambia el estado del MIDlet a activado.
- Paso 4: El usuario selecciona la opción de salir del programa por lo que el MIDlet libera sus recursos e informa al gestor de aplicaciones que desea pasar al estado de destruido, mediante una llamada a `notifyDestroyed()`. Inmediatamente después el gestor finaliza la ejecución del MIDlet.

3.3.2 Estructura de un MIDlet

Una vez comprendidos los conceptos del gestor de aplicaciones y del ciclo de vida se puede ver el aspecto que tiene un MIDlet real.

Los MIDlet 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) */

      }

}
```

Del código anterior, se puede destacar los siguientes elementos de código, comunes a todo MIDlet:

- Se importan todas las clases del paquete `javax.microedition.midlet` (`MIDlet` y `MIDletStateChangeExeption`)
- Todos los MIDlet heredan de la clase `javax.microedition.midlet.MIDlet`. Gracias a esta herencia, todos presenta un mismo interfaz que permite su gestión independientemente del entorno de ejecución.
- 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`.
- Debemos implementar obligatoriamente las funciones abstractas `startApp()`, `pauseApp()` y `destroyApp()`.

Método	Descripción
`protected MIDlet()`	Constructor de la clase
`Protected abstract void destroyApp(boolean unconditional)`	Llamada previa a la finalización de la ejecución
`Public final String getAppProperty(String key)`	Recupera el valor de una variable del fichero descriptor
`public final void notifyDestroy()`	Solicita al gestor de aplicaciones pasar al estado Destruido
`public void notifyPaused()`	Solicita al gestor de aplicaciones pasar al estado de Espera
`protected abstract void pauseApp()`	Llamada previa al paso de Espera
`public final void resumeRequest()`	Solicita al gestor de aplicaciones volver al estado Activado
`protected abstract void starApp()`	Llamada previa al paso al estado Activado

Tabla 3.2 Métodos de la clase MIDlet

3.3.3 Descripción de los métodos más importantes de la clase MIDlet.

- `protected abstract void startApp() throws MIDletstateChangeException`

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

- `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". Este método se puede aprovechar para guardar información que pudiera perderse durante la interrupción del MIDlet. 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.

- `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 `MIDletstateChangeException` 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.

3.3.4 Otros métodos de la clase MIDlet.

Estos métodos que proporcionan mecanismos adicionales se pueden dividir en dos grupos:

- Peticiones de cambio de estado: estos métodos permiten a los MIDlet solicitar al gestor de aplicaciones un cambio de estado. Son: `notifyPaused()`, `notifyDestroyed()` y `resumeRequest()`.
- Acceso a variables de entorno: gracias a la función `getAppProperty()` los MIDlet pueden recuperar variables almacenadas en el fichero descriptor (.jad). La gran utilidad que nos aporta este método es poder configurar el comportamiento de un MIDlet sin necesidad de recompilarlo y empaquetarlo de nuevo. Basta con sacar al fichero descriptor las variables que determinan el comportamiento y recuperarlas al inicio de la ejecución. Se puede cambiar el funcionamiento del MIDlet, cambiando el valor de las variables editando el fichero descriptor.

4. Interfaces de usuario

4.1. Introducción a las interfaces de usuario

Los dispositivos MIDP tienen unas características que hace que sus interfaces gráficas difieran de los ordenadores de sobremesa:

- Cuentan con grades limitaciones en capacidad de proceso, tamaño de ventana y memoria.
- Los MIDlet deben de ser fáciles de usar. No se puede suponer que todos los usuarios están habituados al manejo de ordenadores. En general un MIDlet no debería ser muy distinto a las aplicaciones presentes en un teléfono móvil.
- Los MIDlet conviven en los dispositivos con otras aplicaciones. Para no confundir al usuario, deben ofrecer un aspecto similar tanto en el interfaz como en el modo de interacción.
- Los dispositivos móviles se utilizan en entornos donde no es posible mantener toda la atención sobre ellos. Por ello los programas deben facilitar su uso en estas situaciones.

Durante el proceso de especificación del MIDP, el lenguaje Java ya contaba (en su versión estándar, J2SE) con una completa API para generar interfaces de usuario, AWT (*Application Windows Toolkit*), además de las clases *Swing*. Sin embargo el MIDP *Expert Group* decidió definir una API especifica, ya que AWT fue ideada para su uso en ordenadores de sobremesa. Para ofrecer máxima flexibilidad a los desarrolladores esta API se dividió en dos partes: una de alto nivel (*High Level* API) y otra de bajo nivel (*Low Level* API).

4.1.1 API de alto nivel

Esta API fue diseñada pensando en aplicaciones de negocio, donde el MIDlet actuaría como cliente de una aplicación servidora, y teniendo como principal objeto la portabilidad. Por este motivo, incluye un conjunto de elementos genéricos (formularios, etiquetas…) que se encuentran presentes en todos los dispositivos MIDP. El precio a pagar por esta compatibilidad es la imposibilidad de controlar aspectos estéticos, el modo de funcionamiento interno y el acceso a los mecanismos de entrada de bajo nivel.

4.1.2 API de bajo nivel

Esta API esta diseñada para aplicaciones, principalmente juegos, que requieren de un control total de lo que aparece por pantalla y de los mecanismos de entrada de bajo nivel del dispositivo (teclas, coordenadas del cursor…). Exprimir al máximo las capacidades de los MIDP acarrea en muchos casos problemas de compatibilidad. La API de bajo nivel nos ofrece un control casi absoluto sobre los recursos de la interfaz, con un nivel de abstracción mínimo. A cambio recae sobre el programador asegurar la compatibilidad de los programas, haciendo uso de las facilidades que la API ofrece para identificar las capacidades de cada dispositivo.

4.1.3 Estructura general de la API de interfaz de usuario

Las principales clases para crear interfaces de usuario en MIDP, tanto de alto como de bajo nivel, se agrupan dentro del paquete `javax.microedition.lcdui` y se recogen en la siguiente tabla.

Clase / Interfaz	Descripción
`Display`	Gestor de recursos gráficos
`Displayable`	Pantalla genérica de la interfaz de usuario
`Screen`	Pantalla genérica de la API de alto nivel
`TextBox`	Pantalla de introducción de texto
`List`	Pantalla de selección de opciones
`Alert`	Pantalla de aviso
`Form`	Pantalla de formulario
`Item`	Elemento genérico de un formulario
`ChoiceGroup`	Elemento de formulario para seleccionar opciones
`DataField`	Elemento de formulario para introducir fechas
`TextField`	Elemento de formulario para introducir texto
`Gauge`	Elemento de formulario con forma de diagrama de barras, para introducir valores enteros pequeños
`ImageItem`	Elemento de formulario que representa una imagen descriptiva
`StringItem`	Elemento de formulario que representa un texto descriptivo
`Canvas`	Pantalla genérica de la API de bajo nivel
`Command`	Acción genérica realizada sobre la interfaz por el usuario
`Ticket`	Texto deslizante disponible en todas las pantallas de la API de alto nivel
`Graphics`	Conjunto de herramientas para dibujar dentro de una pantalla de tipo canvas
`Choice`	Interfaz genérico que implementan las clases que permiten seleccionar opciones

Tabla 4.1 Clases para crear interfaces de usuario en MIDP

La Figura 4.1 muestra la relación y jerarquía existente en entre las clases de la tabla anterior.

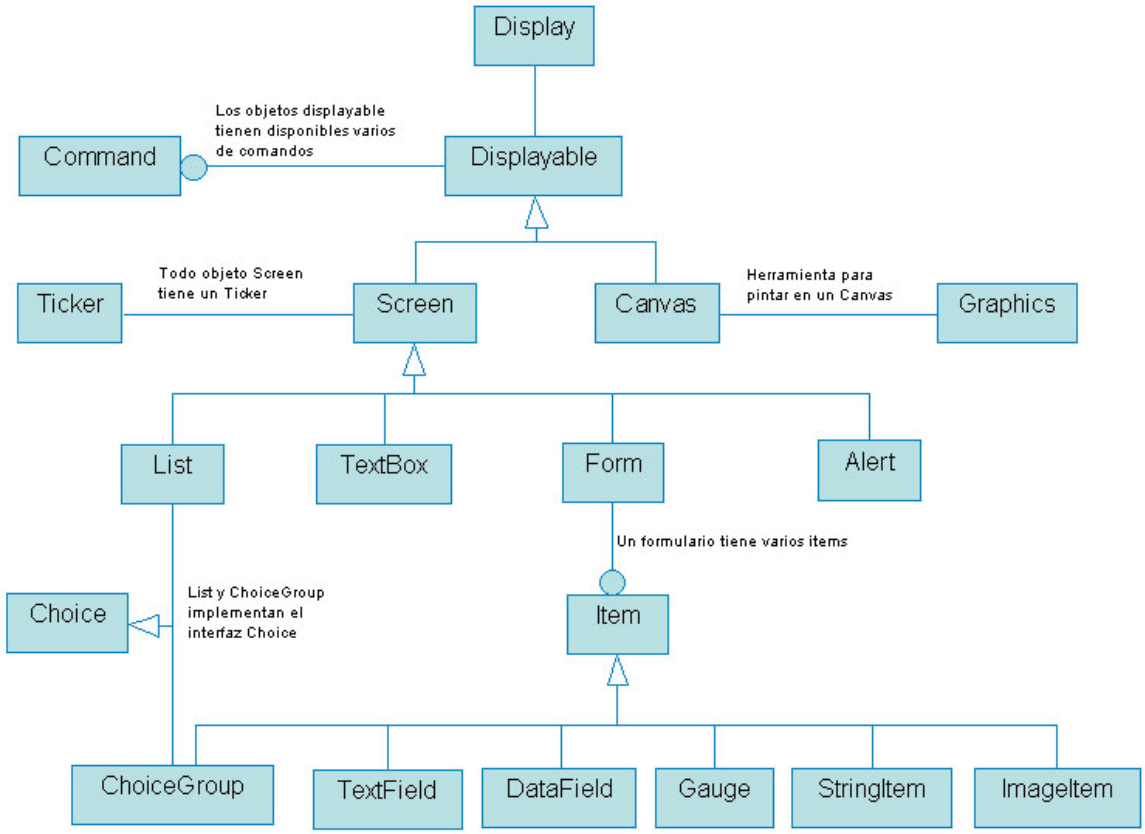

Figura 4.1 Jerarquía de clases de interfaz de usuario en MIDP

4.2. Funcionamiento de la interfaz de usuario

4.2.1 Una interfaz compuesta de pantallas

Las interfaces de usuario en MIDP se componen de pantallas, cada una de las cuales agrupa elementos gráficos que permite la interacción con el usuario (listas, formularios…). Existen dos tipos de pantallas en MIDP, todas ellas descendientes de la clase `Displayable`:

- Pantallas de la API de alto nivel. Todas estas clases descienden de la clase `Screen`. No proporcionan ningún tipo de control sobre su apariencia, que es responsabilidad concreta de cada dispositivo. Además, encapsulan toda la gestión de eventos de bajo nivel. Se pueden dividir en predefinidas (`TextBox`, `List` y `Alert`) a las que no se le pueden añadir componentes adicionales y genéricas (`Form`) que permiten la adición de componentes como imágenes, etiquetas de texto…
- Pantallas de la API de bajo nivel. Todas las pantallas de la API de bajo nivel descienden de la clase `Canvas`. Ofrecen al programador un control casi absoluto sobre lo que aparece en la ventana del MID. Además, dan acceso al manejo de eventos de bajo nivel.

4.2.2 Navegación guiada por eventos

Las interfaces de usuario en MIDP funcionan guiadas por eventos, asociadas a las acciones de los usuarios.

Figura 4.2 Funcionamiento de la interfaz de usuario de un MIDlet

El funcionamiento se divide en varias fases:

1. Creación e iniciación: en esta fase se crean y se inicializan los elementos de la interfaz de usuario. Entre las acciones realizadas destacan las siguientes:
   a. Creación de pantallas: estas pantallas pueden ser tanto de la interfaz de alto nivel como de la de bajo nivel.
   b. Creación de comandos y asignación a pantallas: se crean comandos que representan acciones de usuario, mediante llamadas al constructor de la clase `Command`. Estos comandos se asignan a las diferentes pantallas. Finalmente, se designa un manejador de eventos para cada pantalla, al que se enviarán notificaciones cuando el usuario ejecute un comando.
2. Visualización de la primera pantalla: en esta segunda fase, se selecciona la primera pantalla a mostrar al usuario, mediante una llamada al método `setCurrent()` del objeto `Display`. Esta selección se realiza dentro del método `startApp()` mostrándose la pantalla inmediatamente después de que `startApp()` retorne correctamente.
3. Captura de eventos y toma de decisiones: después de que se muestre la pantalla inicial, los MIDlet quedan a la espera de las acciones que realice el usuario. Cuando esto sucede, el gestor de aplicaciones llama al manejador de eventos asociado a la pantalla. Este manejador identifica la acción del usuario y actúa en consecuencia. En otros, casos los eventos pueden dar paso a otra pantalla.

5. Introducción a RMS

RMS (*Record Management System o* Sistema de Gestión de Registros) es el mecanismo que proporciona la especificación MIDP para que los MIDlet almacenen información persistente entre distintas ejecuciones. 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. RMS se ha diseñado pensando en la importancia de un sistema de almacenamiento para los MIDlet, pero teniendo en cuenta las limitaciones propias de un MID.

El concepto fundamental en RMS es el de `RecordStore`, que puede verse como un almacén de registros de información. Estableciendo una analogía con los sistemas de bases de datos, un `RecordStore` equivaldría a una tabla dentro de un modelo de datos. Algunas de sus principales características son:

- Persistencia entre ejecuciones: Un `RecordStore` proporciona a los MIDlet la posibilidad de almacenar información persistente entre distintas ejecuciones. Es responsabilidad del MID guardar esta información y mantener su integridad durante las operaciones de uso normales (arranque, descarga de batería, llamadas,...). También depende del MID proporcionar el lugar de almacenamiento, que es transparente para las aplicaciones.
- Asociados a *suite*s del MIDlet: Una *suite* de MIDlet puede tener asociados varios `RecordStore` (varias tablas), los cuales pueden ser compartidos por todos los MIDlet incluidos en ella. La eliminación de una *suite* produce el borrado de los `RecordStore` asociados (no tiene sentido mantenerlos cuando no son accesibles desde ninguna otra *suite)*. En cierto modo, el conjunto de `RecordStore` de una *suite* equivaldría a su modelo de datos.
- Identificador único y control de versiones: Cada `RecordStore` tiene un nombre único dentro de la *suite* (aunque *suites* distintas pueden tener `RecordStore` del mismo nombre). Adicionalmente, un `RecordStore` tiene asociados fecha y hora de la última actualización (lo que conoce como marca temporal o *time stamp*), además de un número de versión, actualizado con cada cambio efectuado. Esta información facilita la correcta gestión de la información almacenada tanto por los MIDlet como por la implementación.

El otro concepto clave al hablar de RMS es el de registro. Un registro en RMS es un vector de *bytes* de longitud variable, con un identificador numérico asociado, que se almacena en un `RecordStore`. Siguiendo con la comparación con una base de datos, un registro RMS equivaldría a un registro dentro de una tabla, aunque con una estructura bastante sencilla: un campo identificador y un campo de información con formato de vector de *bytes*.

Las propiedades de estos almacenes de registros son:

1. Cada `RecordStore` está compuesto por cero o más registros.
2. Un nombre de `RecordStore` 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 `RecordStore` con el mismo nombre.
4. Si una *suite* de MIDlets es borrada del dispositivo MID, todos los `RecordStore` pertenecientes a esa *suite* se borrarán.
5. Es posible que un *MIDlet* acceda a un `RecordStore` creado por otra *suite*, siempre que ésta de permiso para ello.

5.1. Estructura de la API de RMS

La API de RMS cuenta con apenas diez componentes, entre clases e interfaces, que se agrupan en el paquete `javax.microedition.rms`.

Clase/ Interfaz	Descripción
`RecordStore`	Clase que representa un `RecordStore`
`RecordEnumeration`	Interfaz que define métodos para desplazarse por los registros de un `RecordStore`
`RecordComparator`	Interfaz que define un método para comparar registros
`RecordFilter`	Interfaz que define un método para seleccionar registros
`RecordListener`	Interfaz para crear un manejador de eventos asociados a las modificaciones en un `RecordStore`
`RecordStoreException`	Excepción de la que derivan todas las demás de RMS
`InvalidRecordIDException`	Excepción lanzada cuando un identificador de registro no existe
`RecordStoreFullException`	Excepción lanzada cuando se llena el espacio de almacenamiento
`RecordStoreNotFoundException`	Excepción lanzada cuando un `RecordStore` no se encuentra
`RecordStoreNotOpenException`	Excepción lanzada cuando se intenta realizar una operación sobre un `RecordStore` que este cerrado

Tabla 5.1 Clases que componen el paquete javax.microedition.rms

Se pueden distinguir tres tipos de clases dentro de RMS:

- La clase `RecordStore`: Es la clase más importante del paquete y proporciona todos los métodos necesarios para manejar un `RecordStore` y sus registros asociados.
- Interfaces de soporte: `RecordEnumeration`, `RecordComparator`, `RecordFilter` y `RecordListener` son interfaces que definen métodos de ayuda al manejo de `RecordStore`. Los tres primeros facilitan el recorrido por los registros, permitiendo establecer ordenaciones y filtros de selección entre ellos. Por su parte, `RecordListener` permite crear manejadores de eventos, asociados a las modificaciones realizadas sobre un `RecordStore`.
- Excepciones: RMS proporciona un conjunto de excepciones específicas para tratar las situaciones anómalas que se puedan presentar durante la ejecución de un programa, desde el acceso a un `RecordStore` cerrado a la falta de espacio de almacenamiento.

5.1.1 Métodos generales de la `RecordStore`

La clase RecordStore representa un conjunto de registros de información, que sirven a los MIDlet de mecanismo de almacenamiento persistente entre distintas ejecuciones. Para facilitar el acceso y la gestión de toda esta información la clase RecordStore proporciona diversos métodos que permite desde la creación de registros hasta su eliminación, pasando por operaciones de lectura y modificación. Estos métodos se pueden dividir en cuatro grupos: métodos generales, operaciones básicas con registros, desplazamiento entre registros y gestión de eventos.

Los métodos generales permiten crear, cerrar y obtener información de un `RecordStore`. Además, se incluyen métodos de clase (no asociados a ningún objeto) para obtener la lista de los `RecordStore` existentes en el MID y permitir el borrado de estos almacenes de información.

La clase `RecordStore` no tiene un método constructor. En su lugar se proporciona un método de clase `openRecordStore()`, que abre un `RecordStore` o lo crea en caso de que no exista. Una vez abierto un `RecordStore`, se pueden realizar sobre él varias operaciones relacionadas con registros (operaciones básicas con registros) finalizando con una llamada a `closeRecordStore()`, que lo cierra.

Existen métodos que devuelven información sobre el `RecordStore`. La API proporciona métodos para las siguientes propiedades:

- Nombre: Los `RecordStore` tienen asociados un nombre único dentro de cada *suite*. Este puede obtenerse con el método `getName()`.
- Versión: Los `RecordStore` tienen asociado un número de versión (un entero positivo), que se actualiza con cada operación de modificación realizada sobre él. Este valor cobra especial importancia en entornos donde varios MIDlet de una misma *suite* accedan a un mismo `RecordStore`, y necesiten saber si se han producido o no modificaciones. El acceso al número de versión se realiza con el método `getVersion()`.
- Fecha y hora de la última modificación: Esta propiedad almacena la fecha y hora de la última modificación del `RecordStore`, en formato de entero largo (milisegundos transcurridos desde el 1 de enero de 1970). Este valor es accesible mediante el método `getLastModified()`.
- Tamaño: Número de *bytes* del espacio de almacenamiento ocupados por el `RecordStore`. Este valor incluye tanto el espacio ocupado por los registros como por las estructuras de datos asociadas, obteniéndose mediante el método `getSize()`.
- Espacio libre: Espacio libre disponible (en *bytes*) para añadir más registros al `RecordStore`, obtenido mediante una llamada a `getSizeAvailable()`.

5.2. Operaciones básicas con registros

La principal utilidad de un `RecordStore` reside en las posibilidades que ofrece para almacenar y gestionar registros de información. Entre las operaciones que proporciona están la inserción y eliminación de registros y su modificación y lectura.

Para trabajar con registros hay que tener en cuenta las siguientes observaciones:

- Cuando se añade un registro a un `RecordStore`, recibe un identificador único (un entero positivo), que es devuelto por el método `addRecord()`. El primer registro recibe como identificador el valor 1, y los sucesivos aumentan en una unidad cada vez que se insertan, no pudiendo reutilizarse cuando se producen operaciones de borrado. Gracias a este identificador, es posible acceder de forma directa a cualquier registro.
- Los `RecordStore` no proporcionan ningún mecanismo de bloqueo para acceder a los registros. La especificación simplemente asegura que las operaciones se realizan de forma secuencial, síncrona y atómica. Es responsabilidad del programador establecer los mecanismos de bloqueo necesarios para mantener la consistencia de los datos cuando varios *threads* accedan simultáneamente a un mismo `RecordStore`.

Los métodos más importantes son aquellos que permiten manejar registros directamente, utilizando vectores de *bytes* para almacenar la información.

Más factible que trabajar con vectores de *bytes* es trabajar con la clases `Stream`, estas clases se encargan de la conversión de los formatos, permitiendo al programador trabajar directamente con los tipos originales (`string`, `int`, `long`…) utilizados en el programa.

5.3. Desplazamiento entre registros de un `RecordStore`

Esta es la funcionalidad más potente que proporciona el RMS. Para ello, la especificación ha definido la interfaz `RecordEnumeration` cuyos métodos proporcionan facilidades para moverse entre los registros de un `RecordStore`.

Para poder utilizar los métodos de `RecordEnumeration` es necesario obtener previamente un objeto que implemente esta interfaz. Para ello, la clase `RecordStore` nos ofrece un método específico (`enumerateRecord`) para devolver objetos de este tipo. Un objeto `RecordEnumeration` obtenido de esta forma mantiene internamente una lista de identificadores de los registros del `RecordStore`, incluyendo un puntero que permite un rápido desplazamiento bidireccional.

Se puede decidir que registros del `RecordStore` aparecen en `RecordEnumeration` usando la interfaz de `RecorFilter`, filtrando los registros no deseados. Por otro lado, es posible establecer criterios de ordenación sobre los registros del `RecordStore` que aparecen en `RecordEnumeration`. Para tal efecto se hace uso de la interfaz `RecordComparator`.

5.4. Gestión de eventos de un `RecordStore`

Hay ocasiones en que interesa realizar acciones asociadas a los cambios producidos en un `RecordStore`. Para ello RMS nos proporcional una interfaz, `RecordListener`, que permite crear manejadores de eventos asociados a un `RecordStore`. Es posible añadir y eliminar manejadores del `RecordStore`. Los manejadores asociados a un `RecordStore` facilitan un mantenimiento de la consistencia de la información entre distintas operaciones.

Estas son los elementos que ofrecen el paquete `javax.microedition.rms` para la gestión de la información persistente de un MIDlet.

6. Acceso a redes desde MIDP

6.1. Introducción

La posibilidad de llevar un dispositivo que ocupa poco espacio y que permite además comunicarse en cualquier momento y lugar abre un abanico de posibles aplicaciones que no se podrían disfrutar en un PC de sobremesa, ni tan siquiera en un ordenador portátil.

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.

6.2. Generic Connection Framework

Proporcionar conectividad en un entorno de recursos tan limitados como son los dispositivos con configuración CLDC no es una tarea sencilla. Destacan dos problemas:

- El tamaño de la API de conectividad de J2SE (`java.io.*` y `java.net.*`): es demasiado grande para ajustarse a las limitaciones de memoria de los dispositivos abarcados en la especificación CLDC.
- Hay una gran variedad de dispositivos móviles con capacidades y protocolos de conectividad distintos, dependiendo en gran medida de las infraestructuras inalámbricas disponibles. Además este número no deja de aumentar.

Además hay que añadir ciertas restricciones propias de la filosofía de diseño seguida en J2ME, como con la definición de interfaces de programación sencillas y estables y la eliminación de ambigüedades que abran la puerta a incompatibilidades entre los programas. El resultado final ha sido la definición de una nueva API, el *Generic Connection Framework* (GCF), basado en:

- En lugar de múltiples clases adaptadas a diferentes protocolos y mecanismos de comunicación, el GCF define una abstracción del concepto de conexión, que oculta al programador los detalles concretos de los distintos protocolos.
- El núcleo del GCF se localiza en la capa de configuración y se compone de elementos genéricos, sin sugerir ni proporcionar protocolos de comunicación específicos. Son los perfiles quienes extienden el GCF con interfaces adicionales y proporcionan implementaciones concretas de protocolos.

6.3. Estructura de la Generic Conection Framework

El GCF se basa en el concepto de conexión, que puede definirse como un canal de comunicación, independientemente del protocolo de bajo nivel utilizado. Las conexiones se crean mediante una única llamada al método estático `open()` de la clase `Connector`. Como resultado de esta llamada, la clase `Connector` devuelve un objeto que implementa una de las interfaces genéricas de conexión definidas en el GCF, con capacidad de manejar el protocolo seleccionado.

A continuación, se muestra la jerarquía de las interfaces de conexión del GCF incluidas en el CLDC

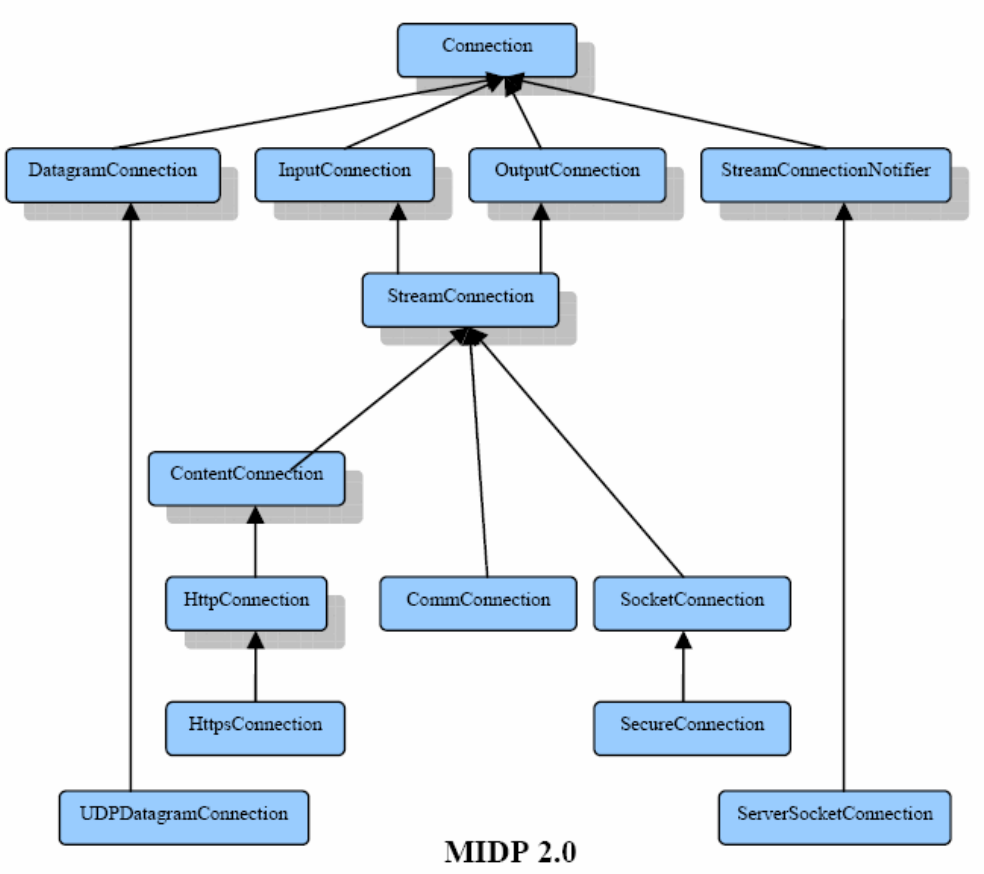

Figura 6.1 Jerarquía de interfaces de GCF

6.4. Conectividad en MIDP

MIDP amplia el GCF con la interfaz `HttpConnection`. Además, MIDP incluye en su especificación la exigencia de que todas las implementaciones incluyan soporte al protocolo HTTP 1.1 abriendo las puertas a la conectividad en los MIDlet.

La disponibilidad de implementaciones de HTTP en todos los MID compatibles MIDP tiene las siguientes consecuencias:

- Cualquier MIDlet que utilice las conexiones basadas en HTTP y las interfaces definidas en GCF es portable entre distintos dispositivos MIDP. La disponibilidad del protocolo HTTP no impide que los fabricantes añadan otros adicionales (como *sockets* o *datagram*) que pueden ser utilizados en programas a costa de sacrificar portabilidad.
- Si se desea mantener compatibilidad y, al mismo tiempo, acceder a servicios que utilicen protocolos distintos de HTTP (como el correo electrónico), es necesario colocar una pasarela en la parte servidora (*gateway*)

7. Conclusión

Para finalizar el trabajo se pueden obtener diversas conclusiones.

J2ME es un lenguaje de programación para unificar las diversas soluciones que existían para el desarrollo de aplicaciones en dispositivos de baja capacidad. Este sigue una estructura por capas asentada sobre una máquina virtual, que le da independencia de plataforma, una configuración, la cual aporta características comunes a un conjunto de dispositivos, y uno o varios perfiles, que concretiza las APIs para centrarse en un conjunto muy reducido de dispositivos. La configuración y el perfil más usados son el CLDC/MIDP, respectivamente, y las aplicaciones que los usan se denominan MIDlet

En la generación de interfaces de usuario se pueden usar dos APIs: una de alto nivel que proporciona portabilidad de los MIDlet y un conjunto de elementos genéricos y una de bajo nivel que ofrece un control total de lo que aparece en pantalla. Por otro lado también se dispone de la posibilidad de almacenar en memoria persistente con el uso de RMS y el posible realizar conexiones a redes mediante GCF.

Los dispositivos móviles y de baja capacidad han tenido una constante evolución, en la que se han visto incrementadas sus posibilidades. J2ME ha sabido aprovechar las posibilidades que ofrecían consiguiendo una gran aceptación y difusión. Actualmente se desarrollan aplicaciones y servicios de gran utilidad para estos dispositivos que se han convertido en una parte imprescindible en la sociedad actual.

8. Referencias

- Alonso Álvarez García y José Ángel Morales Grela. “J2ME”. Anaya multimedia 2002

- José María Gutiérrez Martínez, Roberto Barchino Plata, Jaime Antonio Gutierrez de Mesa, Carmen Pages Arévalo, Luís Miguel Vindel Berendel. “Programación de videojuegos para teléfonos móviles en Java con J2ME”. Ediversitas multimedia 2003.

- Sergio Gálvez Rojas y Lucas Ortega Díaz. “Java a tope” http://www.lcc.uma.es/~galvez/J2ME.html

- Tecnologías y componentes para el desarrollo http://java.sun.com/j2me/

- Publicación de ayuda al desarrollo de J2MEhttp://www.microjava.com/

- Curso orientado al desarrollo de juegos para móviles con J2ME http://www.mailxmail.com/curso/informatica/j2me