



Programación de Dispositivos Móviles

Miguel Ángel Lozano Ortega

Índice

1. INTRODUCCIÓN A LOS DISPOSITIVOS MÓVILES	6
1.1. EVOLUCIÓN DE INTERNET	6
1.2. CARACTERÍSTICAS DE LOS DISPOSITIVOS	10
1.3. APLICACIONES J2ME	21
1.4. J2WTI	28
2. ENTORNO DE DESARROLLO	30
2.1. APLICACIONES J2ME	30
2.2. CONSTRUCCIÓN DE APLICACIONES	34
2.3. KITS DE DESARROLLO	39
2.4. ANTENNA	53
2.5. ENTORNOS DE DESARROLLO INTEGRADOS	57
3. INTRODUCCIÓN A JAVA PARA MIDs	82
3.1. INTRODUCCIÓN A JAVA	82
3.2. INTRODUCCIÓN A LA POO	82
3.3. CONCEPTOS BÁSICOS DE JAVA	84
3.4. CARACTERÍSTICAS BÁSICAS DE CLDC	93
4. MIDLETS	117
4.1. COMPONENTES Y CONTENEDORES	117
4.2. CICLO DE VIDA	118
4.3. CERRAR LA APLICACIÓN	120
4.4. PARAMETRIZACIÓN DE LOS MIDLETS	121
4.5. PETICIONES AL DISPOSITIVO	121

5. INTERFAZ DE USUARIO	123
5.1. ACCESO AL VISOR	123
5.2. COMPONENTES DISPONIBLES	123
5.3. COMPONENTES DE ALTO NIVEL	125
5.4. IMÁGENES	133
5.5. COMANDOS DE ENTRADA	135
5.6. TRANSICIONES ENTRE PANTALLAS	140
6. GRÁFICOS AVANZADOS	144
6.1. GRÁFICOS EN LCDUI	144
6.2. CONTEXTO GRÁFICO	147
6.3. ANIMACIÓN	154
6.4. EVENTOS DE ENTRADA	161
6.5. APIS PROPIETARIAS	164
6.6. GRÁFICOS 3D	166
7. JUEGOS	185
7.1. TIPOS DE JUEGOS	185
7.2. DESARROLLO DE JUEGOS PARA MÓVILES	187
7.3. COMPONENTES DE UN VIDEOJUEGO	194
7.4. SPRITES	196
7.5. FONDO	200
7.6. PANTALLA	203
7.7. MOTOR DEL JUEGO	204
7.8. ENTRADA DE USUARIO EN JUEGOS	207
7.9. EJEMPLO DE MOTOR DE JUEGO	211

8. SONIDO Y MULTIMEDIA	218
8.1. REPRODUCTOR DE MEDIOS	218
8.2. REPRODUCCIÓN DE TONOS	223
8.3. MÚSICA Y SONIDO	224
8.4. REPRODUCCIÓN DE VIDEO	225
8.5. CAPTURA	226
9. ALMACENAMIENTO PERSISTENTE	229
9.1. ALMACENES DE REGISTROS	229
9.2. REGISTROS	232
9.3. NAVEGAR EN EL ALMACÉN DE REGISTROS	234
9.4. NOTIFICACIÓN DE CAMBIOS	237
9.5. OPTIMIZACIÓN DE CONSULTAS	237
9.6. PATRÓN DE DISEÑO ADAPTADOR	239
10. RED Y E/S	244
10.1. MARCO DE CONEXIONES GENÉRICAS	244
10.2. CONEXIÓN HTTP	245
10.3. ACCESO A LA RED A BAJO NIVEL	251
10.4. ENVÍO Y RECEPCIÓN DE MENSAJES	254
10.5. SERVICIOS WEB	257
10.6. CONEXIONES BLUETOOTH	262
11. REGISTRO PUSH	272
11.1. APLICACIONES ACTIVADAS POR PUSH	273
11.2. TEMPORIZADORES	274
11.3. CONEXIONES PUSH	277

12. SEGURIDAD	281
12.1. SANDBOX.....	281
12.2. SOLICITUD DE PERMISOS	282
12.3. DOMINIOS.....	284
12.4. FIRMAR MIDLETS	287
10.5. SERVICIOS WEB	257
10.6. CONEXIONES BLUETOOTH	262
13. APLICACIONES CORPORATIVAS	290
13.1. FRONT-ENDS DE APLICACIONES.....	290
13.2. INTEGRACIÓN CON APLICACIONES.....	292
13.3. ARQUITECTURA MVC.....	299
13.4. MODO SIN CONEXIÓN	312
A. EJERCICIOS	317
A.1. INTRODUCCIÓN Y ENTORNO DE DESARROLLO.....	317
A.2. JAVA PARA MIDS. MIDLETS.....	317
A.3. INTERFAZ GRÁFICA DE ALTO NIVEL.....	320
A.4. PROYECTO: APLICACIÓN BÁSICA.....	321
A.5. INTERFAZ GRÁFICA DE BAJO NIVEL	322
A.6. JUEGOS.....	324
A.7. MULTIMEDIA.....	324
A.8. PROYECTO: JUEGOS.....	325
A.9. RMS.....	327
A.10. RED.....	327
A.11. APLICACIONES CORPORATIVAS.....	329
A.12. PROYECTO: APLICACIÓN CORPORATIVA.....	330

1. Introducción a los dispositivos móviles

Durante los últimos años hemos visto como han ido evolucionando los teléfonos móviles, pasando de ser simplemente teléfonos a ser pequeños ordenadores en los que tenemos disponibles una serie de aplicaciones preinstaladas (mensajería, registro de llamadas, agenda, calculadora, juegos, etc).

Un paso importante en esta evolución ha sido la posibilidad de que estos dispositivos se conecten a Internet. Esto permitirá que accedan a información disponible en la red, a aplicaciones corporativas utilizando algún *front-end*, o descargar nuevas aplicaciones para nuestro móvil.

De esta forma vemos que se podrán incluir nuevas aplicaciones en cada dispositivo, adecuándose a las necesidades de cada usuario. Estas aplicaciones podrán ser tanto *front-ends* que nos permitan acceder a aplicaciones de empresa residentes en algún servidor de Internet, como aplicaciones locales para utilizar en nuestro móvil.

Nos interesará contar con un amplio repertorio de aplicaciones disponibles, y con la posibilidad de que las empresas puedan desarrollar fácilmente software para los móviles que se adapte a sus aplicaciones. Por lo tanto surge la necesidad de que los desarrolladores puedan realizar estas aplicaciones de forma rápida y con el menor coste posible, sin tener que conocer los detalles de la especificación de cada modelo concreto de móvil.

Vamos primero a estudiar las distintas eras en la evolución de Internet, según el tipo de dispositivos que se conectan a esta red, para ver el momento en el que se introducen este tipo de dispositivos y la problemática que plantean. A continuación estudiaremos las características de este tipo de dispositivos, y por último analizaremos de forma general las distintas APIs ofrecidas por la plataforma J2ME para la programación de dispositivos móviles.

1.1. Evolución de Internet

Hasta hace poco hemos visto la red Internet como una red en la que se encuentran conectados ordenadores de todo el mundo. Sin embargo, durante los últimos años estamos entrando en una segunda era de Internet, en la que se están conectando a la red diferentes tipos de dispositivos heterogéneos como por ejemplo teléfonos móviles o PDAs.

1.1.1. Primera era: El Internet de los ordenadores

En esta primera era tenemos conectados a Internet ordenadores que cuentan con una interfaz común y una capacidad mínima suficiente. La información que obtenemos de la red se obtiene normalmente en forma de páginas web escritas en HTML, diseñadas para mostrarse en dispositivos gráficos con resoluciones por encima de 640x480 *pixels* y con monitores de por lo menos 14".

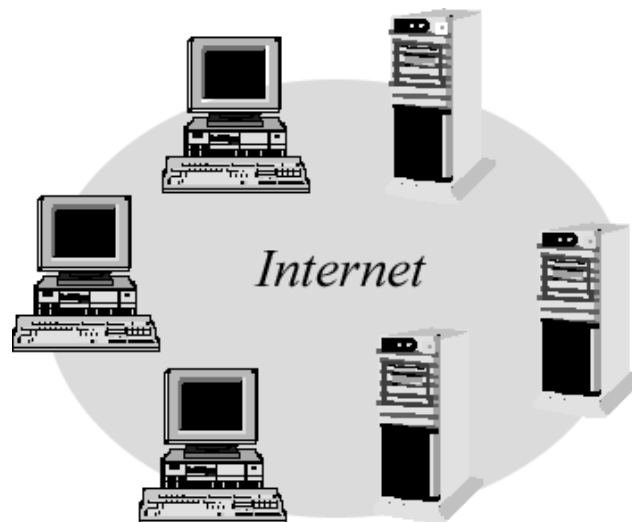


Figura 1. El Internet de los ordenadores

Asumiendo que los clientes cuentan con esta configuración mínima, podemos elaborar páginas que se visualicen correctamente en ellos. Es decir, parte de la capa de presentación residirá en el servidor, ya que estas páginas HTML definirán el formato con el que se mostrará la información. Los navegadores en las máquinas cliente deberán ser capaces de interpretar el código HTML y mostrar en pantalla las páginas web.

1.1.2. Segunda era: El Internet de los dispositivos

Al incorporarse distintos tipos de dispositivos a Internet, cada uno de ellos con interfaces y capacidad totalmente distinta, ya no va a ser posible definir una presentación única en el lado del servidor, ya que ésta no va a poder ser visualizada correctamente por todos los clientes.

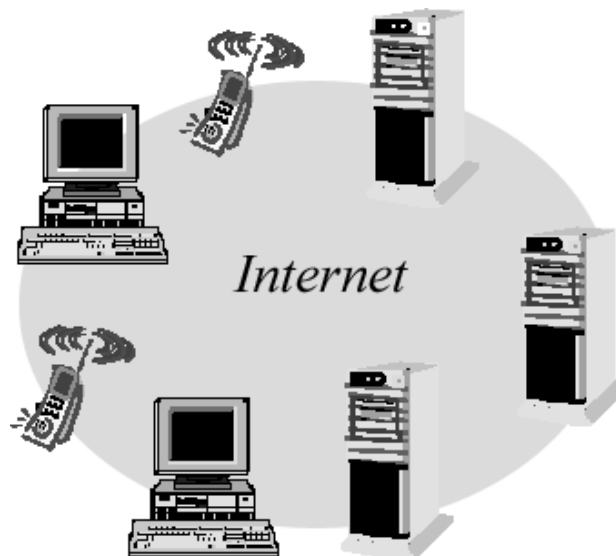


Figura 2. El Internet de los dispositivos

Estos dispositivos llevan ordenadores embebidos, en muchos casos con grandes limitaciones en su interfaz y baja capacidad, debido a su reducido tamaño o a la escasa necesidad de incorporar un ordenador mayor que encarecería el producto. Deberán conectarse a Internet utilizando estos ordenadores, por lo que el acceso será distinto dependiendo del dispositivo del que se trate.

Una posible solución es definir un nuevo tipo de documento, adaptado a la reducida interfaz de estos dispositivos. Este es el caso por ejemplo de los documentos WML para los teléfonos móviles WAP. El problema es que si definimos esta presentación en el servidor para todos los móviles, tendremos que definirla de forma que sea compatible con todos los modelos existentes que soporten esta tecnología. Conforme mejoran los modelos de móviles, podríamos visualizar en ellos aplicaciones más ricas, pero el tener que utilizar un formato compatible para todos los móviles nos obligará a definir un documento bastante más pobre que sea soportado por todos ellos.

Por ello en esta segunda era surgen nuevos paradigmas para la programación web. La información se empezará a ofrecer en forma de servicios, en lugar de documentos. Es decir, obtendremos de Internet únicamente la información, pero no la presentación. Por ejemplo, esta información puede estar codificada en lenguaje XML siguiendo un determinado estándar, de forma que cualquier aplicación sea capaz de entenderla, o bien utilizar nuestra propia codificación. De esta forma, cada cliente (móvil) tendrá aplicaciones adaptadas a sus características, que obtendrán de Internet únicamente la información a la que necesitan acceder.

Nos vamos a centrar en la programación de este tipo de aplicaciones para dispositivos móviles. Estas aplicaciones se alojarán y se ejecutarán de forma local en el dispositivo. Será deseable que la API (Interfaz de Programación de Aplicaciones) que ofrezca cada dispositivo para programar en él nos permita utilizar todas las características del mismo, y establecer conexiones de red para obtener la información necesaria de Internet.

1.1.3. Tercera era: El Internet de las cosas

En una futura era de Internet, podrá formar parte de Internet cualquier cosa. Podremos de esta forma tener conectada a la red ropa, productos de alimentación o cualquier otra cosa que incorpore un chip capaz de conectarse y ofrecer información sobre el producto. Con esto se disparará el número de elementos conectados a la red.

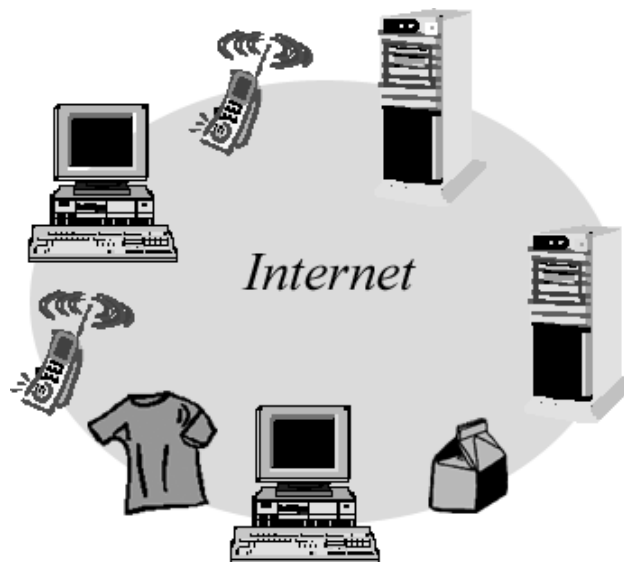


Figura 3. El Internet de las cosas

Por ejemplo, la tecnología *Auto ID* consiste en un pequeño chip transmisor (del tamaño de la cabeza de un alfiler) que podrá ser incluido en los artículos. Este chip vendrá a sustituir en el futuro al código de barras, ya que además de ofrecernos toda la información sobre el producto, nos permitirá conocer su localización. Para comunicarse emitirán una señal de radio, que será recibida por un receptor conectado a un ordenador que nos ofrezca la información sobre estas "cosas" conectadas.

De esta forma podremos saber fácilmente si en nuestra tienda tenemos un determinado tipo de productos, o si se han agotado y tenemos que realizar un pedido. También podremos saber si alguien se lleva un producto de nuestra tienda, lo cuál será útil para evitar robos. O incluso ver si en nuestra tienda tenemos productos que hayan caducado, y descubrir dónde están estos productos para poder retirarlos de la venta.

Vemos que esta tecnología tiene multitud de posibles aplicaciones. Como inconveniente, tenemos la presunta violación de la intimidad de los consumidores al poder estos chips ofrecer información sobre los artículos que han adquirido.

1.1.4. Internet como un conjunto de servicios

Hemos visto como en la segunda era de Internet surgen los Servicios Web como alternativa a los documentos web (como las páginas HTML y WML por ejemplo). Podemos definir un servicio como la interfaz que nos da acceso a un módulo de funcionalidad.

Por ejemplo, podemos tener un servicio que nos permita acceder a información sobre cambio de monedas. Este servicio nos ofrecerá una operación `cambioVenta(moneda_origen, moneda_destino)` que nos devolverá el cambio actual entre dos monedas, para la venta, y otra operación análoga

`cambioCompra(moneda_origen, moneda_destino)` para obtener esta información en el caso de la compra.

Un Servicio Web será un servicio al que podamos acceder mediante protocolos web estándar. Se utilizará lenguaje XML para codificar el mensaje de invocación de las operaciones del servicio, y la respuesta que nos haya devuelto la operación. De esta forma podremos invocar servicios a través de Internet.

Los documentos web mezclan la información y la presentación. Normalmente la información que ofrecen está escrita en lenguaje natural y formateada de forma que los humanos la podamos entender fácilmente. Sin embargo, esto será difícilmente entendible por una aplicación. Los Servicios Web nos ofrecen únicamente la información que nos ha devuelto la operación invocada. Por ello podemos ver estos Servicios Web como la web para aplicaciones, frente a los documentos web que serían la web para humanos. Serán las aplicaciones instaladas en el cliente, que invocan los Servicios Web, las encargadas de dar formato y presentar al usuario la información obtenida.

En la actualidad podemos encontrar algunos Servicios Web ofrecidos por una serie de proveedores. En el futuro Internet será como un gran Sistema Operativo distribuido, en el que haya disponibles un gran número de servicios que puedan ser utilizados por cualquier máquina conectada a la red. De esta forma las aplicaciones se encontrarán distribuidas en Internet, es decir, estarán formadas por módulos que podrán residir en distintos lugares del mundo. La cuestión es quién se hará con el control de este gran Sistema Operativo, ¿Microsoft con .NET o Sun con las tecnologías Java?

1.2. Características de los dispositivos

Vamos a ver los distintos tipos de dispositivos con ordenadores embebidos que podemos encontrar, así como sus características, para luego centrarnos en los dispositivos móviles de información (*Mobile Information Devices*, MIDs) que son los que trataremos con mayor detalle.

1.2.1. Tipos de dispositivos

Podemos encontrar distintos dispositivos con ordenadores embebidos que van introduciéndose en nuestros hogares. Por un lado tenemos los teléfonos móviles y las agendas electrónicas o PDAs. Estos dispositivos son conocidos como **dispositivos móviles de información (MIDs)**. Incorporan un reducido Sistema Operativo y una serie de aplicaciones que se ejecutan sobre él (Agenda, Notas, Mensajes, etc) adaptadas a la interfaz de estos dispositivos, de forma que se puedan visualizar correctamente en su pequeña pantalla y que se puedan manejar usando el teclado del teléfono o el puntero del PDA por ejemplo.

A parte de los MIDs encontramos más tipos de dispositivos, cuyo número crecerá conforme pase el tiempo. Tenemos por ejemplo los descodificadores de televisión digital (*set top boxes*). Las plataformas de televisión digital, tanto

por satélite, por cable, o terrestre, ofrecen una serie de servicios accesibles desde estos descodificadores, como por ejemplo servicios de teleguía, juegos, etc. Para ello deberemos tener aplicaciones que se ejecuten en estos dispositivos.

Los electrodomésticos también siguen esta tendencia y ya podemos encontrar por ejemplo frigoríficos o lavadoras con un pequeño computador capaz de conectarse a la red doméstica. La idea es integrar todos estos dispositivos en nuestra red para construir un sistema domótico. Para ello todos los dispositivos que queramos integrar deberán ser capaces de conectarse a la red.

Hablamos de **dispositivos conectados** para referirnos a cualquier dispositivo capaz de conectarse a la red. Podemos tener distintos tipos de dispositivos conectados, con grandes diferencias de capacidad. Un subgrupo de estos dispositivos conectados son los MIDs de los que hemos hablado previamente.

1.2.2. Dispositivos móviles de información (MIDs)

Vamos a centrarnos en el estudio de los MIDs. Para ello vamos a ver las características de una serie de modelos actuales con lo que podremos hacernos una idea del rango de memoria y los tipos de interfaz con los que vamos a tener que trabajar.



	Nokia 8310	Nokia 3410	Nokia 6100	Nokia 3650
	-Serie 20	-Serie 30	-Serie 40	-Serie 60
	-Similar a 3330	-Similar a 6610, 7210, 7250, 5100, Gage 3300	-Similar a 6610, 7210, 7250, 5100, Gage 3300	-Similar a 7650, N-Gage
	-Sin Java			
Interfaz	84x48 Monocromo	96x65 Monocromo	128x128 12bits (4096 colores)	176x208 12 bits (4096 colores)
Memoria	-	Heap 164 KB Shared 150 KB JAR 50 KB	Heap 200 KB Shared 725 KB (625 KB en 6610, 7210) (4,6 MB en 7250) (4,5 MB en 3300) JAR 64 KB	Heap 1,4 MB (2,4 MB en N-Gage) Shared 3,4 MB (4 MB en 7650, N-Gage) JAR 4 MB CLDC 1.0 MIDP 1.0
APIs	-	CLDC 1.0 MIDP 1.0 WMA Nokia UI Nokia SMS	CLDC 1.0 MIDP 1.0 Nokia UI WMA (3300)	WMA (N-Gage, 3650) MMAPI (N-Gage, 3650) Nokia UI SO Symbian

Otros	HSCSD GPRS (8310) CSD (3330) WAP 1.2.1 (8310) WAP 1.1 (3330) IrDA (8310)	CSD WAP 1.1	HSCSD GPRS WAP 1.2.1 USB IrDA Cámara (7250)	HSCSD GPRS WAP 1.2.1 XHTML Camara (3650, 7650) Bluetooth IrDA (3650, 7650) USB (N-Gage)
--------------	---	----------------	--	--



SL45i



C55
-Similar a M50



SL55
-Similar a M55, S55

Interfaz	101x80 Monocromo	101x64 Monocromo	101x80 12 bits (4096 colores) (256 colores en S55)
Memoria	Heap 147 KB Flex 0,2 MB JAR 70 KB	Heap 150 KB Flex 0,5 MB (0,2 MB en M50) JAR 70 KB	Heap 365 KB (560 KB en M55) Flex 1,6 MB (1 MB en S55) JAR 70 KB
APIs	MIDP 1.0	MIDP 1.0	MIDP 1.0
Otros	WAP 1.1 IrDA	GPRS WAP 1.2.1	GPRS WAP 1.2.1 IrDA (S55, SL55) Bluetooth (S55) Cámara (SL55)



Nokia
-Serie
-Comunicador



9210 Sharp
80 5500



Zaurus SL- Palm Tungsten



HP iPAQ Pocket
PC H1930

Interfaz	640x200 12 bits (4096 colores)	240x320 16 bits (65536 colores)	240x320 Color	240x320 Color
Memoria	8 MB SD-RAM 18 MB disponibles 14 MB aplicaciones	64 MB RAM	16 MB RAM	64 MB SDRAM

APIs	PersonalJava JavaPhone SO Symbian	SO Linux 2.4	SO Palm	SO Windows Pocket PC 2003 Professional
Otros	HSCSD WAP 1.1 HTML 3.2 IrDA Cable serie	IrDA USB	IrDA Bluetooth USB	IrDA USB

Podemos ver que las características de los móviles varían bastante, tanto a nivel de memoria como de interfaz entre los diferentes modelos disponibles. Además, a medida que aparezcan modelos nuevos encontraremos mayor diferencia entre los modelos antiguos y los más recientes. Es difícil realizar una aplicación que se adapte perfectamente al modelo utilizado. Por eso normalmente lo que se hará será realizar distintas versiones de una misma aplicación para distintos conjuntos de móviles con similares características. El coste de hacer una aplicación capaz de adaptarse sería mayor que el de hacer varias versiones de la aplicación, intentando siempre hacerlo de forma que tenga que modificarse la menor cantidad de código posible.

Tenemos además un problema adicional, y es que cada móvil puede tener su propia API. Distintos modelos de móviles, tanto de marcas distintas como de la misma marca, pueden incluir Sistemas Operativos diferentes y por lo tanto una interfaz diferente para programar aplicaciones (API). Esto nos obligará a tener que modificar gran parte del código para portar nuestras aplicaciones a diferentes modelos de móviles. Dado el creciente número de dispositivos disponibles esto será un grave problema ya que será difícil que una aplicación esté disponible para todos los modelos.

Aquí es donde aparece la conveniencia de utilizar tecnologías Java para programar este tipo de dispositivos tan heterogéneos. Sun ha apostado desde hace tiempo por las aplicaciones independientes de la plataforma con su tecnología Java. Esta independencia de la plataforma cobra un gran interés en este caso, en el que tenemos un gran número de dispositivos heterogéneos.

Con la edición *Micro Edition* de Java 2, se introduce esta plataforma en los dispositivos. De esta forma podremos desarrollar aplicaciones Java que puedan correr en todos los dispositivos que soporten las APIs de J2ME. Al ser estas APIs un estándar, los programas que escribamos en Java que las utilicen serán portables a todos estos dispositivos.

Es más, no será necesario recompilar la aplicación para cada tipo de dispositivo. Las aplicaciones Java corren sobre una máquina virtual Java, que deberá estar instalada en todos los dispositivos que soporten Java. Las aplicaciones Java se compilan a un tipo de código intermedio (conocido como *bytecodes*) que es capaz de interpretar la máquina virtual. De esta forma, nuestra aplicación una vez compilada podrá ejecutarse en cualquier dispositivo que tenga una máquina virtual Java, ya que esta máquina virtual será capaz de interpretar el código compilado de la aplicación sin necesidad de compilarla otra vez. Incluso si aparece un nuevo modelo de dispositivo con soporte para

Java después de que nosotros hayamos publicado una aplicación, esta aplicación funcionará en este modelo.

Es por esta razón que las tecnologías Java se han impuesto en el mercado de programación de dispositivos móviles, teniendo prácticamente todos los modelos de móviles que aparecen en la actualidad soporte para Java.

1.2.3. Redes móviles

Centrándonos en los teléfonos móviles celulares, vamos a ver las distintas generaciones de estos dispositivos según la red de comunicaciones utilizada.

Se denominan celulares porque la zona de cobertura se divide en zonas de menor tamaño llamadas células. Cada célula tendrá un transmisor que se comunicará con los dispositivos dentro del área de dicha célula mediante señales de radio, operando en una determinada banda de frecuencias. Dentro de esta banda de frecuencias, tendremos un determinado número de canales a través de los cuales se podrán comunicar los móviles de dicha zona.

Este número de canales estará limitado por la banda de frecuencias utilizada, lo que limitará el número de usuarios que puedan estar conectados simultáneamente a la red. Al dividir la cobertura en células permitimos que en distintas células se reutilicen las mismas frecuencias, por lo que este número limitará sólo los usuarios conectados dentro de una misma célula, que es un área pequeña donde habrá menos usuarios, y no de toda la zona global de cobertura.

Primera generación (1G): Red analógica

Los primeros teléfonos móviles que aparecieron utilizaban una red analógica para comunicarse. La información analógica se transfiere por ondas de radio sin ninguna codificación, por frecuencia modulada (FM). Este tipo de redes permiten únicamente comunicaciones de voz. Un ejemplo de red analógica es la red TACS, que opera en la banda de frecuencias entorno a los 900MHz. Al ser analógica hace un uso poco eficiente del espectro, por lo que tendremos menos canales disponibles y por lo tanto será más fácil que la red se sature.

Esta es la red analógica que se utiliza en España para telefonía móvil, basada en la Estadounidense AMPS (*Advanced Mobile Phone System*). En EEUU además de esta podemos encontrar bastantes más tipos de redes analógicas.

Un gran inconveniente de estas redes analógicas es que existen numerosos tipos de redes, y cada país adoptó una distinta. Esto dificulta la itinerancia (o *roaming*), ya que no posibilita que utilicemos nuestro dispositivo en otros países con redes diferentes.

Segunda generación (2G): Red digital

La segunda generación de móviles consiste en aquellos que utilizan una red digital para comunicarse. En Europa se adoptó como estándar la red GSM

(*Global System for Mobile communications*). Se trata de una red inicialmente diseñada para voz, pero que también puede transferir datos, aunque es más apropiada para voz. Digitaliza tanto la voz como los datos para su transmisión. Esta red es la que ha producido un acercamiento entre la telefonía móvil y la informática. En Japón se utiliza una red diferente con características similares a GSM, llamada PDC.

La red GSM se implanta como estándar en Europa y desplaza rápidamente a los analógicos. En una primera fase opera en los 900MHz. Una vez saturada la banda de los 900MHz empieza a funcionar en 1800MHz. Los móviles capaces de funcionar en ambas bandas (*dualband*) tendrán una mayor cobertura, ya que si una banda está saturada podrán utilizar la otra. En EEUU se utiliza la banda 1900MHz. Hay móviles tribanda que nos permitirán conectarnos en cualquiera de estas 3 bandas, por lo que con ellos tendremos cobertura en EEUU, Europa y Asia.

Una característica de los dispositivos GSM es que los datos relevantes del usuario se almacenan en una tarjeta de plástico extraíble (SIM, *Suscriber Identification Module*), lo cual independiza la identificación del usuario (número de teléfono y otros datos) del terminal móvil, permitiendo llevar esta tarjeta de un terminal a otro sin tener que cambiar el número.

Opera por conmutación de circuitos (CSD, *Circuit Switched Data*), es decir, se establece un circuito permanente de comunicación. Con esta tecnología se consigue una velocidad de 9,6kbps, siendo el tiempo de establecimiento de conexión de unos 10 segundos. Los inconvenientes de esta tecnología son:

- Debido a que se establece un circuito de comunicación de forma permanente, se tarifica por tiempo, por lo que se cobrará al usuario el tiempo que esté conectado aunque no esté descargando nada.
- Además, tiene una velocidad de transmisión demasiado lenta, lo cual es bastante negativo para el usuario sobretodo al tarificarse por tiempo de conexión.
- Otro inconveniente es que no se trata de una red IP, por lo que el móvil no podrá conectarse directamente a Internet. Tendrá que hacerlo a través de algún intermediario (*gateway*) que traduzca los protocolos propios de la red móvil a los protocolos TCP/IP de Internet.

Más adelante se desarrolla la tecnología HSCSD (*High Speed Circuit Switched Data*), que consigue una velocidad de 56Kbps sobre la misma red GSM. Para conseguir esta alta velocidad, esta tecnología utiliza varios canales de comunicación simultáneamente, cada uno de los cuales funciona a una velocidad de 9'6Kbps físicos de forma similar a como se hacía con CSD (se puede conseguir aumentar a 14'4kbps utilizando métodos de compresión por software). Al transmitir por 4 canales se consigue esta mayor velocidad, pero tenemos el gran inconveniente de tener 4 circuitos establecidos de forma permanente, por lo que el coste de la conexión se multiplicará por 4, tarificándose por tiempo de conexión igual que ocurría en el caso anterior. Otro inconveniente es que sigue sin ser compatible con IP.

Paso intermedio (2,5G): GPRS

Hemos visto que con las tecnologías anteriores de portadoras de datos (CSD y HSCSD) tenemos los problemas de la baja velocidad, tarificación por tiempo y el no ser una red IP. Surge aquí la necesidad de implantar un método de transmisión por paquetes que no requiera ocupar un canal de forma permanente, sino que envíe los datos fraccionados en paquetes IP independientes. De esta forma surge GPRS, que se considera como un paso intermedio entre la segunda (2G) y tercera (3G) generación.

GPRS (*General Packet Radio Service*) es una evolución de las redes GSM y funciona sobre estas mismas redes, por lo que no será necesario realizar una gran inversión en infraestructuras, simplemente se deberá actualizar la red GSM para que soporte la transmisión de paquetes.

Esta tecnología realiza una transmisión inalámbrica de datos basada en paquetes. Puede alcanzar una velocidad de hasta 144kbps teóricamente, aunque en la práctica no suele pasar de los 40Kbps por limitaciones de los dispositivos. La información se fragmenta en paquetes que se envían mediante el protocolo IP por distintos canales de forma independiente y se reconstruye en destino. Al seguir el protocolo IP, podremos acceder a todos los recursos de la red Internet.

Un canal no tendrá que estar dedicado a la comunicación exclusivamente de un punto a otro, es decir, no mantiene una conexión abierta de forma permanente, conecta solo para transmitir datos (paquetes). Por esta razón se aprovecharán mejor los canales de comunicación, ya que sólo se ocupan cuando es necesario y se optimiza así el uso de la red. El tiempo de establecimiento de conexión es de aproximadamente 1 segundo. De esta forma se paga por información transmitida, y no por tiempo de conexión. Esto nos permite que podamos estar siempre conectados a Internet, ya que mientras no se transfieran paquetes no estaremos ocupando los canales de comunicación y por lo tanto no se nos estará cobrando.

Los dispositivos cuentan con varios canales para transmitir (a 10Kbps cada uno), tanto para enviar como para recibir. Por ejemplo podemos tener dispositivos (2+1), con 2 canales para recibir y 1 para enviar, por lo que recibirán datos a 20Kbps y enviarán a 10Kbps. Se pueden tener hasta un máximo de 8 canales. Esto nos permite tener simultaneidad de voz y datos.

En Japón, de forma similar, se tiene la red PDC-P que es una extensión de la anterior red PDC para trabajar mediante transmisión por paquetes a una velocidad de 28.8Kbps.

Tercera generación (3G): Banda ancha

La tercera generación de telefonía móvil viene con la red UMTS (*Universal Mobile Telephony System*). Con esta nueva tecnología de radio se pretende buscar un estándar mundial para las redes de telefonía móvil, de forma que

podemos movernos a cualquier parte del mundo sin que nuestro móvil deje de funcionar.

Además con ella se pretende dar acceso a servicios multimedia, como videoconferencia, ver la TV a través del móvil, oír música, etc. Para ello esta red proporciona un ancho de banda mucho mayor que las redes de segunda generación, teniendo velocidades de transferencia entre 384kbps y 2Mbps. Al igual que ocurría con GPRS la información se enviará por paquetes, por lo que se cobrará por información transmitida, lo que nos permitirá estar conectados permanentemente.

Esta red tiene el inconveniente de que para implantarla es necesario realizar una fuerte inversión en infraestructuras. Es compatible con GSM y funciona en la frecuencia 2GHz.

1.2.4. Portadores

En la arquitectura de capas de las redes móviles, denominamos capa portadora (*bearer*) a aquella que se va a utilizar para transferir los datos a través de la red. Distintas tecnologías que podemos utilizar como portadoras para enviar o recibir datos a través de la red son:

- **CSD**: Conmutación de circuitos sobre una red GSM.
- **HSCSD**: Conmutación de circuitos de alta velocidad sobre una red GSM.
- **GPRS**: Envío de paquetes a través de una red GSM.
- **PDC**: Red de características similares a GSM utilizada en Japón.
- **PDC-P**: Extensión de PDC para trabajar por transmisión de paquetes.
- **SMS**: Mensajes de texto cortos. Se envían por paquetes, limitando su contenido a 140 *bytes*.
- **MMS**: Mensajes multimedia. Permiten incorporar elementos multimedia (audio, imágenes, video) al mensaje.

Debemos distinguir claramente las tecnologías portadoras de las tecnologías para desarrollar aplicaciones. Podemos desarrollar aplicaciones para dispositivos móviles utilizando diferentes tecnologías, y para acceder a una aplicación podremos utilizar diferentes portadores.

1.2.5. Aplicaciones para móviles

Las aplicaciones web normalmente se desarrollan pensando en ser vistas en las pantallas de nuestro PCs, con una resolución de unos 800x600 *pixels* y navegar en ellas mediante el puntero del ratón. Por lo tanto, estas aplicaciones no se podrán mostrar correctamente en las reducidas pantallas de los dispositivos móviles. Además, en gran parte de los dispositivos móviles (como por ejemplo los teléfonos) no tenemos ningún dispositivo de tipo puntero, por lo que deberemos realizar páginas en las que se pueda navegar fácilmente utilizando el reducido teclado de estos dispositivos.

Han surgido diferentes tecnologías diseñadas para ofrecer contenidos aptos para este tipo de dispositivos. Entre ellas destacamos las siguientes:

- **WAP (Wireless Application Protocol):** Se compone de un conjunto de protocolos que se sitúan por encima de la capa portadora. Puede funcionar sobre cualquier tecnología portadora existente (CSD, HSCSD, GRPS, SMS, UMTS, etc).

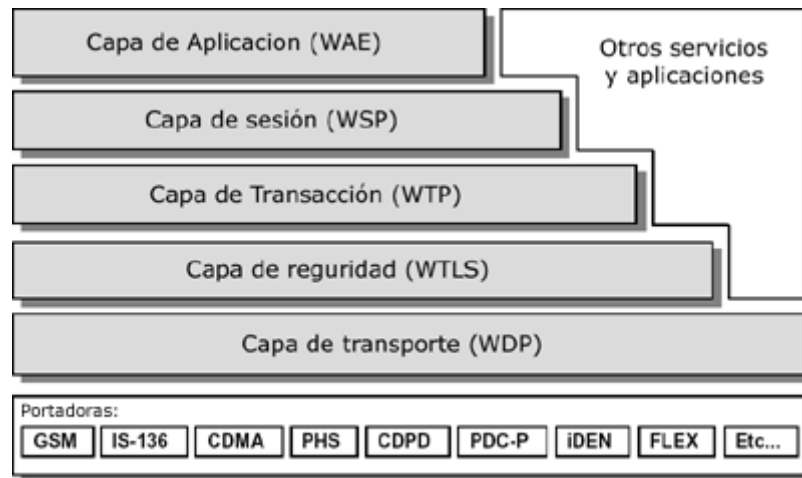


Figura 3. Arquitectura WAP

Debido a que la red móvil puede no ser una red IP, se introduce un elemento intermediario: el *gateway* WAP. Este *gateway* traduce entre el protocolo WSP (perteneciente a WAP) de la red móvil y el protocolo TCP/IP de Internet.

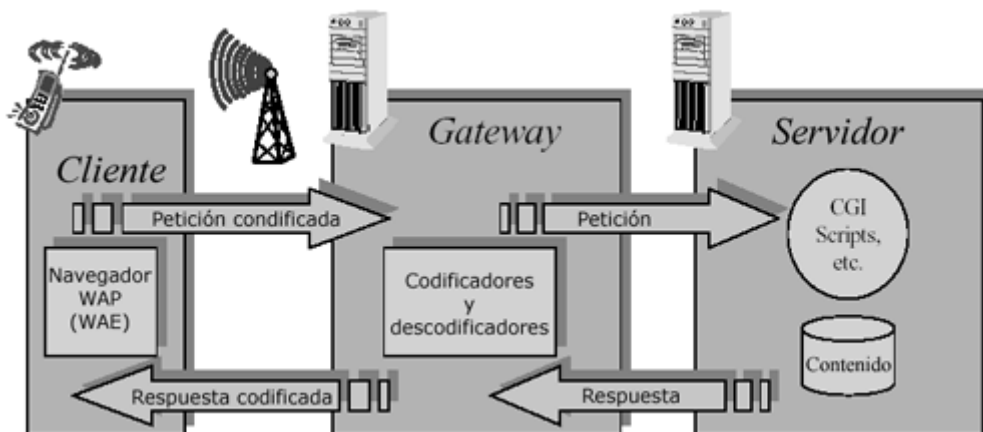


Figura 4. Gateway WAP

Los documentos web se escriben en lenguaje WML (*Wireless Markup Language*), que forma parte del protocolo WAP. Se trata de un lenguaje de marcado para definir documentos web que puedan ser visualizados en pantallas pequeñas, usando navegadores WML. Como inconvenientes de este lenguaje encontramos la pobreza de los documentos que podemos generar con él, ya que para asegurarse de funcionar en todos los dispositivos debe ser muy restrictivo, y también el ser un lenguaje totalmente distinto a HTML, que obligará a los desarrolladores a

tener que aprender otro lenguaje y escribir desde cero una nueva versión de las aplicaciones.

- **iMode:** Esta tecnología fue lanzada por la empresa NTT en Japón. Para escribir los documentos se utiliza cHTML, un subconjunto del HTML (compacto). En Japón esta tecnología se ofreció desde el principio con una velocidad similar a la de las red GSM, con conexión permanente y tarifando por información transmitida. Se lanzó con gran cantidad de servicios, y cuenta con un gran éxito en Japón. En Europa también se introduce esta tecnología, en este caso sobre GPRS. El inconveniente que tiene es que es propietario, mientras que WAP es abierto.
- **XHTML:** Se trata de una versión reducida de lenguaje HTML ideado para crear aplicaciones para dispositivos móviles con interfaces reducidas. Es similar a cHTML, pero a diferencia de este último, se ha desarrollado como estándar

Con estas tecnologías podemos desarrollar aplicaciones web para acceder desde dispositivos móviles. Sin embargo, en este tipo de dispositivos donde muchas veces la conexión es lenta, cara e inestable es conveniente poder trabajar con las aplicaciones de forma local. Además las aplicaciones que instalemos en el dispositivo podrán estar hechas a medida para nuestro modelo de dispositivo concreto y de esta manera adaptarse mejor a las posibilidades que ofrece.

Vamos a ver qué tecnologías podemos utilizar para desarrollar estas aplicaciones. Los dispositivos tendrán instalado un sistema operativo. Existen diferentes sistemas operativos disponibles para este tipo de dispositivos, entre los que destacamos los siguientes:

- **Windows Pocket PC:** Se trata de una versión del sistema operativo Windows de Microsoft para PDAs. Es una evolución de Windows CE, destinado también a este tipo de dispositivos. Windows CE tenía un aspecto similar al Windows 9X, pero no se adaptaba bien a las reducidas interfaces de estos dispositivos. Pocket PC soluciona este problema y tiene un aspecto similar a Windows XP. Una ventaja de Pocket PC es que mantiene la compatibilidad con los sistemas Windows de escritorio, ya que maneja los mismos formatos de ficheros, por lo tanto podremos transferir fácilmente nuestros datos entre PCs y PDAs.
- **Palm OS:** Sistema operativo para los PDAs Palm. Se adapta mejor a los dispositivos que Windows CE.
- **Symbian OS:** Se trata de un Sistema Operativo incluido en distintos modelos de móviles, como por ejemplo en la serie 60 de Nokia.

Podemos programar aplicaciones utilizando la API de estos SO, pero estas aplicaciones sólo funcionarán en dispositivos que cuenten con dicho SO. Si además accedemos directamente al hardware del dispositivo en nuestros programas, entonces sólo podemos confiar en que sea compatible con el modelo concreto para el que lo hayamos desarrollado.

Debido al gran número de dispositivos distintos existentes con diferentes sistemas operativos, programar a bajo nivel (es decir, utilizando directamente la API del SO) no será conveniente ya que será muy complicado portar nuestros programas a otros dispositivos distintos. Por ello adquiere especial interés en el campo de la programación de dispositivos móviles el tener aplicaciones independientes de la plataforma.

Para tener esta independencia de la plataforma deberemos tener instalado en los dispositivos un entorno de ejecución que sea capaz de interpretar estas aplicaciones multiplataforma. Existen diferentes tecnologías que nos permitirán crear este tipo de aplicaciones, como son las siguientes:

- **BREW** (*Binary Runtime Environment for Wireless*): El lenguaje de programación nativo es C/C++. También puede usarse Java y otros lenguajes. El inconveniente de esta tecnología es que es desconocida por los desarrolladores, y que está soportada por un número reducido de dispositivos.
- **J2ME** (*Java 2 Micro Edition*): Edición de la plataforma Java para dispositivos móviles. Se trata de una versión reducida de Java que nos permitirá ejecutar aplicaciones escritas en este lenguaje en estos dispositivos. Con Java desde el principio se apostó por la multiplataforma, por lo que la filosofía seguida con esta tecnología es muy adecuada a este tipo de dispositivos.

Tiene la ventaja de que existe una gran comunidad de desarrolladores Java, a los que no les costará aprender a programar para estos dispositivos, ya que se usa prácticamente la misma API, y además la mayoría de modelos de móviles que aparecen en el mercado soportan esta tecnología. Podemos encontrar gran número de páginas que nos ofrecen aplicaciones y juegos Java para descargar en nuestros móviles.

1.2.6. Conectividad de los MIDs

Hemos hablado de que estos dispositivos son capaces de conectarse. Vamos a ver las posibles formas de conectar estos dispositivos móviles para obtener datos, aplicaciones o intercambiar cualquier otra información. Una primera forma de conectarlos consiste en establecer una conexión a Internet desde el móvil a través de la red GSM. Sin embargo, esta conexión cuesta dinero, por lo que será interesante tener otros mecanismos de conexión directa de nuestro móvil para poder copiar en él las aplicaciones que estemos desarrollando para hacer pruebas, gestionar nuestros datos, o intercambiar información con otros usuarios.

- **OTA (Over The Air)**: Se conecta directamente a Internet de forma inalámbrica utilizando la red móvil (GSM o en el futuro UMTS). Esto nos permitirá acceder a los recursos que haya en Internet. Por ejemplo, podemos publicar nuestras aplicaciones en una página WML y descargarlas desde ahí para instalarlas en el móvil. El inconveniente de este método es que el tiempo de conexión tiene un coste elevado.

- **Cable serie/USB:** Algunos dispositivos permiten ser conectados a un PC mediante cable serie o USB. Con ello podremos copiar los datos del móvil al PC, o al revés. De este forma podremos subir nuestras aplicaciones al móvil para probarlas. El problema es que tendremos que conectar el móvil físicamente mediante un cable.
- **Infrarrojos (IrDA):** Un gran número de modelos nos permiten establecer conexiones vía infrarrojos. Podemos de esta manera conectar varios dispositivos entre si, o bien conectarlos con un PC. Para ello el PC deberá contar con puerto de infrarrojos. Muchos portátiles incorporan este puerto, o también podemos añadir este puerto a cualquier otro ordenador por ejemplo a través de interfaz USB. Tenemos la ventaja de que podemos conectar todo tipo de dispositivos que cuenten con este puerto, y además no es necesario contar con un cable físico. El inconveniente es que los dispositivos deberán verse entre si para poder comunicarse por infrarrojos.
- **Bluetooth:** *Bluetooth* es una tecnología que nos permite conectar distintos tipos de dispositivos utilizando ondas de radio para comunicarse. Podremos de esta forma conectar distintos dispositivos *bluetooth* entre ellos. Podemos incorporar un adaptador *bluetooth* a nuestro PC (a través de USB por ejemplo) de forma que nuestro ordenador se comporte como un dispositivo *bluetooth* más y de esta forma podamos conectarlo a nuestro móvil. Al comunicarse por ondas de radio no hará falta que los dispositivos estén visibles entre si, teniendo estas ondas un alcance de unos 10 metros. Es el sustituto de alta velocidad de los infrarrojos, pudiendo alcanzar velocidades de 723Kbit/s.

1.3 Aplicaciones J2ME

La plataforma J2ME nos ofrece una serie de APIs con las que desarrollar las aplicaciones en lenguaje Java. Una vez tengamos la aplicación podremos descargarla en cualquier dispositivo con soporte para J2ME y ejecutarla en él.

J2ME soporta una gran variedad de dispositivos, no únicamente MIDs. Actualmente define APIs para dar soporte a los dispositivos conectados en general, tanto aquellos con una gran capacidad como a tipos más limitados de estos dispositivos.

1.3.1. Arquitectura de J2ME

Hemos visto que existen dispositivos de tipos muy distintos, cada uno de ellos con sus propias necesidades, y muchos con grandes limitaciones de capacidad. Si obtenemos el máximo común denominador de todos ellos nos quedamos prácticamente con nada, por lo que es imposible definir una única API en J2ME que nos sirva para todos.

Por ello en J2ME existirán diferentes APIs cada una de ellas diseñada para una familia de dispositivos distinta. Estas APIs se encuentran arquitecturadas en dos capas: configuraciones y perfiles.

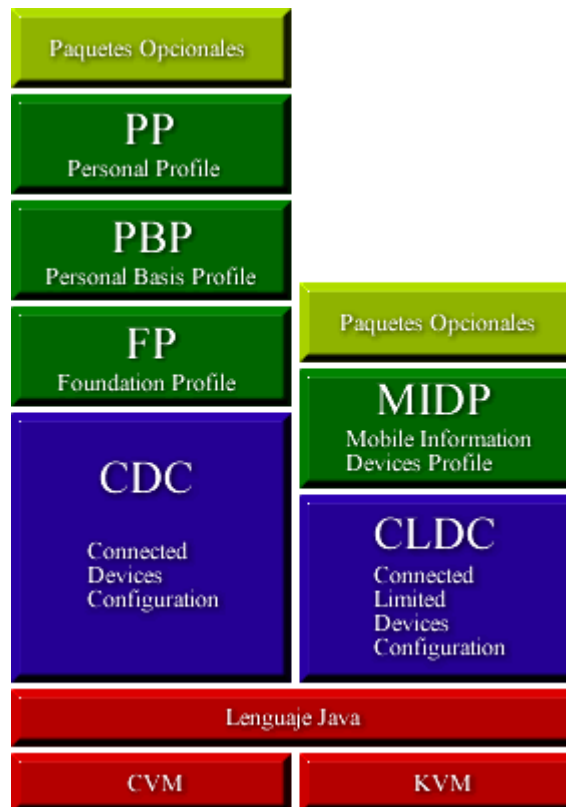


Figura 5. Arquitectura de J2ME

Configuraciones

Las configuraciones son las capas de la API de bajo nivel, que residen sobre la máquina virtual y que están altamente vinculadas con ella, ofrecen las características básicas de todo un gran conjunto de dispositivos. En esta API ofrecen lo que sería el máximo denominador común de todos ellos, la API de programación básica en lenguaje Java.

Encontramos distintas configuraciones para adaptarse a la capacidad de los dispositivos. La configuración CDC (*Connected Devices Configuration*) contiene la API común para todos los dispositivos conectados, soportada por la máquina virtual Java.

Sin embargo, para algunos dispositivos con grandes limitaciones de capacidad esta máquina virtual Java puede resultar demasiado compleja. Hemos de pensar en dispositivos que pueden tener 128 KB de memoria. Es evidente que la máquina virtual de Java (JVM) pensada para ordenadores con varias megas de RAM instaladas no podrá funcionar en estos dispositivos.

Por lo tanto aparece una segunda configuración llamada CLDL (*Connected Limited Devices Configuration*) pensada para estos dispositivos con grandes limitaciones. En ella se ofrece una API muy reducida, en la que tenemos un menor número de funcionalidades, adaptándose a las posibilidades de estos dispositivos. Esta configuración está soportada por una máquina virtual mucho

más reducida, la KVM (*Kilobyte Virtual Machine*), que necesitará muchos menos recursos por lo que podrá instalarse en dispositivos muy limitados.

Vemos que tenemos distintas configuraciones para adaptarnos a dispositivos con distinta capacidad. La configuración CDC soportada por la JVM (*Java Virtual Machine*) funcionará sólo con dispositivos con memoria superior a 1 MB, mientras que para los dispositivos con memoria del orden de los KB tenemos la configuración CLDC soportada por la KVM, de ahí viene el nombre de *Kilobyte Virtual Machine*.

Perfiles

Como ya hemos dicho, las configuraciones nos ofrecen sólo la parte básica de la API para programar en los dispositivos, aquella parte que será común para todos ellos. El problema es que esta parte común será muy reducida, y no nos permitirá acceder a todas las características de cada tipo de dispositivo específico. Por lo tanto, deberemos extender la API de programación para cada familia concreta de dispositivos, de forma que podamos acceder a las características propias de cada familia.

Esta extensión de las configuraciones es lo que se denomina perfiles. Los perfiles son una capa por encima de las configuraciones que extienden la API definida en la configuración subyacente añadiendo las operaciones adecuadas para programar para una determinada familia de dispositivos.

Por ejemplo, tenemos un perfil MIDP (*Mobile Information Devices Profile*) para programar los dispositivos móviles de información. Este perfil MIDP reside sobre CLDC, ya que estos son dispositivos bastante limitados a la mayoría de las ocasiones.

Paquetes opcionales

Además podemos incluir paquetes adicionales, como una tercera capa por encima de las anteriores, para dar soporte a funciones concretas de determinados modelos de dispositivos. Por ejemplo, los móviles que incorporen cámara podrán utilizar una API multimedia para acceder a ella.

1.3.2. Configuración CDC

La configuración CDC se utilizará para dispositivos conectados con una memoria de por lo menos 1 MB (se recomiendan al menos 2 MB para un funcionamiento correcto). Se utilizará en dispositivos como PDAs de gama alta, comunicadores, decodificadores de televisión, impresoras de red, *routers*, etc.

CDC se ha diseñado de forma que se mantenga la máxima compatibilidad posible con J2SE, permitiendo de este modo portar fácilmente las aplicaciones con las que ya contamos en nuestros ordenadores a CDC.

La máquina virtual utilizada, la CVM, cumple con la misma especificación que la JVM, por lo que podremos programar las aplicaciones de la misma forma

que lo hacíamos en J2SE. Existe una nueva máquina virtual para soportar CDC, llamada CDC *Hotspot*, que incluye diversas optimizaciones utilizando la tecnología *Hotspot*.

La API que ofrece CDC es un subconjunto de la API que ofrecía J2SE, optimizada para este tipo de dispositivos que tienen menos recursos que los PCs en los que utilizamos J2SE. Se mantienen las clases principales de la API, que ofrecerán la misma interfaz que en su versión de J2SE.

CDC viene a sustituir a la antigua API *PersonalJava*, que se aplicaba al mismo tipo de dispositivos. La API CDC, a diferencia de *PersonalJava*, está integrada dentro de la arquitectura de J2ME.

Tenemos diferentes perfiles disponibles según el tipo de dispositivo que estemos programando: *Foundation Profile* (FP), *Personal Basis Profile* (PBP) y *Personal Profile* (PP).

Foundation Profile

Este es el perfil básico para la programación con CDC. No incluye ninguna API para la creación de una interfaz gráfica de usuario, por lo que se utilizará para dispositivos sin interfaz, como por ejemplo impresoras de red o *routers*.

Los paquetes que se incluyen en este perfil son:

```
java.io
java.lang
java.lang.ref
java.lang.reflect
java.net
java.security
java.security.acl
java.security.cert
java.security.interfaces
java.security.spec
java.text
java.util
java.util.jar
java.util.zip
```

Vemos que incluye todos los paquetes del núcleo de Java, para la programación básica en el lenguaje (`java.lang`), para manejar la entrada/salida (`java.io`), para establecer conexiones de red (`java.net`), para seguridad (`java.security`), manejo de texto (`java.text`) y clases útiles (`java.util`). Estos paquetes se incluyen en su versión integra, igual que en J2SE. Además incluye un paquete adicional que no pertenece a J2SE:

```
javax.microedition.io
```

Este paquete pertenece está presente para mantener la compatibilidad con CLDC, ya que pertenece a esta configuración, como veremos más adelante.

Personal Basis Profile

Este perfil incluye una API para la programación de la interfaz gráfica de las aplicaciones. Lo utilizaremos para dispositivos en los que necesitemos aplicaciones con interfaz gráfica. Este es el caso de los descodificadores de televisión por ejemplo.

Además de los paquetes incluidos en FP, añade los siguientes:

```
java.awt  
java.awt.color  
java.awt.event  
java.awt.image  
java.beans  
java.rmi  
java.rmi.registry
```

Estos paquetes incluyen un subconjunto de las clases que contenían en J2SE. Tenemos el paquete AWT (`java.awt`) para la creación de la interfaz gráfica de las aplicaciones. Este paquete sólo incluye soporte para componentes ligeros (aquellos que definen mediante código Java la forma de dibujarse), y no incluye ningún componente de alto nivel (como botones, campos de texto, etc). Tendremos que crear nosotros nuestros propios componentes, definiendo la forma en la que se dibujará cada uno.

También incluye un soporte limitado para *beans* (`java.beans`) y objetos distribuidos RMI (`java.rmi`).

Además podemos encontrar un nuevo tipo de componente no existente en J2SE, que son los **Xlets**.

```
javax.microedition.xlet  
javax.microedition.xlet.ixc
```

Estos *xlets* son similares a los *applets*, son componentes que se ejecutan dentro de un contenedor que controla su ciclo de vida. En el caso de los *applets* este contenedor era normalmente el navegador donde se cargaba el *applet*. Los *xlets* se ejecutan dentro del *xlet manager*. Los *xlets* pueden comunicarse entre ellos mediante RMI. De hecho, la parte de RMI incluida en PBP es únicamente la dedicada a la comunicación entre *xlets*.

Los *xlets* se diferencian también de los *applets* en que tienen un ciclo de vida definido más claramente, y que no están tan vinculados a la interfaz (AWT) como los *applets*. Por lo tanto podremos utilizar tanto *xlets* con interfaz gráfica, como sin ella.

Estos *xlets* se suelen utilizar en aplicaciones de televisión interactiva, instaladas en los descodificadores (*set top boxes*).

Personal Profile

Este perfil incluye soporte para *applets* e incluye la API de AWT íntegra. De esta forma podremos utilizar los componentes pesados de alto nivel definidos en AWT (botones, menús, campos de texto, etc). Estos componentes pesado

utilizan la interfaz gráfica nativa del dispositivo donde se ejecutan. De esta forma, utilizaremos este perfil cuando trabajemos con dispositivos que disponen de su propia interfaz gráfica de usuario (GUI) nativa.

Incluye los siguientes paquetes:

```
java.applet
java.awt
java.awt.datatransfer
```

En este caso ya se incluye íntegra la API de AWT (`java.awt`) y el soporte para applets (`java.applet`). Este paquete es el más parecido al desaparecido *PersonalJava*, por lo que será el más adecuado para migrar las aplicaciones *PersonalJava* a J2ME.

Paquetes opcionales

En CDC se incluyen como paquetes opcionales subconjuntos de otras APIs presentes en J2SE: la API **JDBC** para conexión a bases de datos, y la API de **RMI** para utilizar esta tecnología de objetos distribuidos.

Además también podremos utilizar como paquete opcional la API **Java TV**, adecuada para aplicaciones de televisión interactiva (iTV), que pueden ser instaladas en descodificadores de televisión digital. Incluye la extensión JMF (*Java Media Framework*) para controlar los flujos de video.

Podremos utilizar estas APIs para programar todos aquellos dispositivos que las soporten.

1.3.3. Configuración CLDC

La configuración CLDC se utilizará para dispositivos conectados con poca memoria, pudiendo funcionar correctamente con sólo 128 KB de memoria. Normalmente la utilizaremos para los dispositivos con menos de 1 ó 2 MB de memoria, en los que CDC no funcionará correctamente. Esta configuración se utilizará para teléfonos móviles (celulares) y PDAs de gama baja.

Esta configuración se ejecutará sobre la KVM, una máquina virtual con una serie de limitaciones para ser capaz de funcionar en estas configuraciones de baja memoria. Por ejemplo, no tiene soporte para tipos de datos `float` y `double`, ya que estos dispositivos normalmente no tienen unidad de punto flotante.

La API que ofrece esta configuración consiste en un subconjunto de los paquetes principales del núcleo de Java, adaptados para funcionar en este tipo de dispositivos. Los paquetes que ofrece son los siguientes:

```
java.lang
java.io
java.util
```

Tenemos las clases básicas del lenguaje (`java.lang`), algunas clases útiles (`java.util`), y soporte para flujos de entrada/salida (`java.io`). Sin embargo vemos que no se ha incluido la API de red (`java.net`). Esto es debido a que esta API es demasiado compleja para estos dispositivos, por lo que se sustituirá por una API de red propia más reducida, adaptada a sus necesidades de conectividad:

```
javax.microedition.io
```

En la actualidad encontramos únicamente un perfil que se ejecuta sobre CLDC. Este perfil es MIDP (*Mobile Information Devices Profile*), y corresponde a la familia de dispositivos móviles de información (teléfonos móviles y PDAs).

Los dispositivos iMode utilizan una API de Java propietaria de NTT DoCoMo, llamada DoJa. Esta API se construye sobre CLDC, pero no es un perfil perteneciente a J2ME. Simplemente es una extensión de CLDC para teléfonos iMode.

Mobile Information Devices Profile

Utilizaremos este perfil para programar aplicaciones para MIDs. En los siguientes capítulos nos centraremos en la programación de aplicaciones para móviles utilizando este perfil, que es el que más protagonismo ha cobrado tras la aparición de los últimos modelos de móviles que incluyen soporte para esta API.

La API que nos ofrece MIDP consiste, además de los paquetes ofrecidos en CLDL, en los siguientes paquetes:

```
javax.microedition.lcdui  
javax.microedition.lcdui.game  
javax.microedition.media  
javax.microedition.media.control  
javax.microedition.midlet  
javax.microedition.pki  
javax.microedition.rms
```

Las aplicaciones que desarrollaremos para estos dispositivos se llaman **MIDlets**. El móvil actuará como contenedor de este tipo de aplicaciones, controlando su ciclo de vida. Tenemos un paquete con las clases correspondientes a este tipo de componentes (`javax.microedition.midlet`). Además tendremos otro paquete con los elementos necesarios para crear la interfaz gráfica en la pantalla de los móviles (`javax.microedition.lcdui`), que además nos ofrece facilidades para la programación de juegos. Tenemos también un paquete con clases para reproducción de músicas y tonos (`javax.microedition.media`), para creación de certificados por clave pública para controlar la seguridad de las aplicaciones (`javax.microedition.pki`), y para almacenamiento persistente de información (`javax.microedition.rms`).

Paquetes opcionales

Como paquetes opcionales tenemos:

- **Wireless Messaging API (WMA)** (JSR-120): Nos permitirá trabajar con mensajes en el móvil. De esta forma podremos por ejemplo enviar o recibir mensajes de texto SMS.
- **Mobile Media API (MMAPI)** (JSR-135): Proporciona controles para la reproducción y captura de audio y video. Permite reproducir ficheros de audio y video, generar y secuenciar tonos, trabajar con *streams* de estos medios, e incluso capturar audio y video si nuestro móvil está equipado con una cámara.
- **J2ME Web Services API (WSA)** (JSR-172): Nos permitirá invocar Servicios Web desde nuestro cliente móvil. Muchos fabricantes de servidores de aplicaciones J2EE, con soporte para desplegar Servicios Web, nos ofrecen sus propias APIs para invocar estos servicios desde los móviles J2ME, como es el caso de Weblogic por ejemplo.
- **Bluetooth API** (JSR-82): Con esta API podremos establecer comunicaciones con otros dispositivos de forma inalámbrica y local vía *bluetooth*.
- **Security and Trust Services API for J2ME** (JSR-177): Ofrece servicios de seguridad para proteger los datos privados que tenga almacenados el usuario, encriptar la información que circula por la red, y otros servicios como identificación y autenticación.
- **Location API for J2ME** (JSR-179): Proporciona información acerca de la localización física del dispositivo (por ejemplo mediante GPS o E-OTD).
- **SIP API for J2ME** (JSR-180): Permite utilizar SIP (*Session Initiation Protocol*) desde aplicaciones MIDP. Este protocolo se utiliza para establecer y gestionar conexiones IP multimedia. Este protocolo puede usarse en aplicaciones como juegos, videoconferencias y servicios de mensajería instantánea.
- **Mobile 3D Graphics (M3G)** (JSR-184): Permite mostrar gráficos 3D en el móvil. Se podrá utilizar tanto para realizar juegos 3D como para representar datos.
- **PDA Optional Packages for J2ME** (JSR-75): Contiene dos paquetes independientes para acceder a funcionalidades características de muchas PDAs y algunos teléfonos móviles. Estos paquetes son PIM (*Personal Information Management*) y FC (*FileConnection*). PIM nos permitirá acceder a información personal que tengamos almacenada de forma nativa en nuestro dispositivo, como por ejemplo nuestra libreta de direcciones. FC nos permitirá abrir ficheros del sistema de ficheros nativo de nuestro dispositivo.

1.4 JWTI

JWTI (*Java Technology for Wireless Industry*) es una especificación que trata de definir una plataforma estándar para el desarrollo de aplicaciones para móviles. En ella se especifican las tecnologías que deben ser soportadas por los dispositivos móviles:

- CLDC 1.0
- MIDP 2.0
- WMA 1.1
- Opcionalmente CLDC 1.1 y MMAPI

De esta forma se pretende evitar la fragmentación de APIs, proporcionando un estándar que sea aceptado por la gran mayoría de los dispositivos existentes. El objetivo principal de esta especificación es aumentar la compatibilidad e interoperabilidad, de forma que cualquier aplicación que desarrollemos que cumpla con JTWI pueda ser utilizada en cualquier dispositivo que soporte esta especificación.

En ella se especifica el conjunto de APIs que deben incorporar los teléfonos JTWI, de forma que podremos confiar en que cuando utilicemos estas APIs, la aplicación va a ser soportada por todos ellos. Además con esta especificación se pretende evitar el uso de APIs opcionales no estándar que producen aplicaciones incompatibles con la mayoría de dispositivos.

Hay aspectos en los que las especificaciones de las diferentes APIs (MIDP, CLDC, etc) no son claras del todo. Con JTWI también se pretende aclarar todos estos puntos, para crear un estándar que se cumpla al 100% por todos los fabricantes, consiguiendo de esta forma una interoperabilidad total.

También se incluyen recomendaciones sobre las características mínimas que deberían tener todos los dispositivos JTWI.

2. Entorno de desarrollo

En este tema vamos a ver cómo construir aplicaciones J2ME a partir del código fuente de forma que estén listas para ser instaladas directamente en cualquier dispositivo con soporte para esta tecnología.

Vamos a estudiar la creación de aplicaciones para MIDs, que serán normalmente teléfonos móviles o algunos PDAs. Por lo tanto nos centraremos en el perfil MIDP.

Para comenzar vamos a ver de qué se componen las aplicaciones MIDP que podemos instalar en los móviles (ficheros JAD y JAR), y cómo se realiza este proceso de instalación. A continuación veremos como crear paso a paso estos ficheros de los que se componen estas aplicaciones, y como probarlas en emuladores para no tener que transferirlas a un dispositivo real cada vez que queramos hacer una prueba. En el siguiente punto se verá cómo podremos facilitar esta tarea utilizando los kits de desarrollo que hay disponibles, y algunos entornos integrados (IDEs) para hacer más cómodo todavía el desarrollo de aplicaciones para móviles.

2.1. Aplicaciones J2ME

Para distribuir e instalar las aplicaciones J2ME en los dispositivos utilizaremos ficheros de tipos JAR y JAD. Las aplicaciones estarán compuestas por un fichero JAR y un fichero JAD.

2.1.1. Suite de MIDlets

Los MIDlets son las aplicaciones Java desarrolladas con MIDP que se pueden ejecutar en los MIDs. Los ficheros JAD y JAR contienen un conjunto de MIDlets, lo que se conoce como *suite*. Una *suite* es un conjunto de uno o más MIDlets empaquetados en un mismo fichero. De esta forma cuando dicha *suite* sea instalada en el móvil se instalarán todas las aplicaciones (MIDlets) que contenga.

El fichero JAR será el que contendrá las aplicaciones de la *suite*. En él tendremos tanto el código compilado como los recursos que necesite para ejecutarse (imágenes, sonidos, etc). Estos ficheros JAR son un estándar de la plataforma Java, disponibles en todas las ediciones de esta plataforma, que nos permitirán empaquetar una aplicación Java en un solo fichero. Al ser un estándar de la plataforma Java será portable a cualquier sistema donde contemos con esta plataforma.

Por otro lado, el fichero JAD (*Java Application Descriptor*) contendrá una descripción de la *suite*. En él podremos encontrar datos sobre su nombre, el tamaño del fichero, la versión, su autor, MIDlets que contiene, etc. Además también tendrá una referencia al fichero JAR donde se encuentra la aplicación.

2.1.2. Instalación de aplicaciones

De esta forma cuando queramos instalar una aplicación deberemos localizar su fichero JAD. Una vez localizado el fichero JAD, deberemos indicar que deseamos instalar la aplicación, de forma que se descargue e instale en nuestro dispositivo el fichero JAR correspondiente. Además el fichero JAD localizado nos permitirá saber si una aplicación ya está instalada en nuestro dispositivo, y de ser así comprobar si hay disponible una versión superior y dar la opción al usuario de actualizarla. De esta forma no será necesario descargar el fichero JAR entero, cuyo tamaño será mayor debido a que contiene toda la aplicación, para conocer los datos de la aplicación y si la tenemos ya instalada en nuestro móvil.

Un posible escenario de uso es el siguiente. Podemos navegar con nuestro móvil mediante WAP por una página WML. En esa página puede haber publicadas una serie de aplicaciones Java para descargar. En la página tendremos los enlaces a los ficheros JAD de cada aplicación disponible. Seleccionando con nuestro móvil uno de estos enlaces, accederá al fichero JAD y nos dará una descripción de la aplicación que estamos solicitando, preguntándonos si deseamos instalarla. Si decimos que sí, descargará el fichero JAR asociado en nuestro móvil e instalará la aplicación de forma que podemos usarla. Si accedemos posteriormente a la página WML y pinchamos sobre el enlace al JAD de la aplicación, lo comparará con las aplicaciones que tenemos instaladas y nos dirá que la aplicación ya está instalada en nuestro móvil. Además, al incluir la información sobre la versión podrá saber si la versión que hay actualmente en la página es más nueva que la que tenemos instalada, y en ese caso nos dará la opción de actualizarla.

2.1.3. Software gestor de aplicaciones

Los dispositivos móviles contienen lo que se denomina AMS (*Application Management Software*), o software gestor de aplicaciones en castellano. Este software será el encargado de realizar el proceso de instalación de aplicaciones que hemos visto en el punto anterior. Será el que controle el ciclo de vida de las *suítes*:

- Obtendrá información de las *suítes* a partir de un fichero JAD mostrándosela al usuario y permitiendo que éste instale la aplicación.
- Comprobará si la aplicación está ya instalada en el móvil, y en ese caso comparará las versiones para ver si la versión disponible es más reciente que la instalada y por lo tanto puede ser actualizada.
- Instalará o actualizará las aplicaciones cuando se requiera, de forma que el usuario tenga la aplicación disponible en el móvil para ser utilizada.
- Ejecutará los MIDlets instalados, controlando el ciclo de vida de estos MIDlets como veremos en el capítulo 4.
- Permitirá desinstalar las aplicaciones, liberando así el espacio que ocupan en el móvil.

2.1.4. Fichero JAD

Los ficheros JAD son ficheros ASCII que contienen una descripción de la *suite*. En él se le dará valor a una serie de propiedades (parámetros de configuración) de la *suite*. Tenemos una serie de propiedades que deberemos especificar de forma obligatoria en el fichero:

MIDlet-Name	Nombre de la suite.
MIDlet-Version	Versión de la suite. La versión se compone de 3 número separados por puntos: <mayor>.<menor>.<micro>, como por ejemplo 1.0.0
MIDlet-Vendor	Autor (Proveedor) de la suite.
MIDlet-Jar-URL	Dirección (URL) de donde obtener el fichero JAR con la suite.
MIDlet-Jar-Size	Tamaño del fichero JAR en <i>bytes</i> .

Además podemos incluir una serie de propiedades adicionales de forma optativa:

MIDlet-Icon	Icono para la suite. Si especificamos un icono éste se mostrará junto al nombre de la suite, por lo que nos servirá para identificarla. Este icono será un fichero con formato PNG que deberá estar contenido en el fichero JAR.
MIDlet-Description	Descripción de la suite.
MIDlet-Info-URL	Dirección URL donde podemos encontrar información sobre la suite.
MIDlet-Data-Size	Número mínimo de <i>bytes</i> que necesita la suite para almacenar datos de forma persistente. Por defecto este número mínimo se considera 0.
MIDlet-Delete-Confirm	Mensaje de texto con el que se le preguntará al usuario si desea desinstalar la aplicación.
MIDlet-Delete-Notify	URL a la que se enviará una notificación de que el usuario ha desinstalado nuestra aplicación de su móvil.
MIDlet-Install-Notify	URL a la que se enviará una notificación de que el usuario ha instalado nuestra aplicación en su móvil.

Estas son propiedades que reconocerá el AMS y de las que obtendrá la información necesaria sobre la *suite*. Sin embargo, como desarrolladores puede interesarnos incluir una serie de parámetros de configuración propios de nuestra aplicación. Podremos hacer eso simplemente añadiendo nuevas propiedades con nombres distintos a los anteriores al fichero JAR. En el capítulo 4 veremos como acceder a estas propiedades desde nuestras aplicaciones.

Un ejemplo de fichero JAD para una *suite* de MIDlets es el siguiente:


```

MIDlet-Name: SuiteEjemplos
MIDlet-Version: 1.0.0
MIDlet-Vendor: Universidad de Alicante
MIDlet-Description: Aplicaciones de ejemplo para moviles.
MIDlet-Jar-Size: 16342
MIDlet-Jar-URL: ejemplos.jar

```

2.1.5. Fichero JAR

En el fichero JAR empaquetaremos los ficheros `.class` resultado de compilar las clases que componen nuestra aplicación, así como todos los recursos que necesite la aplicación, como pueden ser imágenes, sonidos, músicas, videos, ficheros de datos, etc.

Para empaquetar estos ficheros en un fichero JAR, podemos utilizar la herramienta `jar` incluida en J2SE. Más adelante veremos como hacer esto.

Además de estos contenidos, dentro del JAR tendremos un fichero `MANIFEST.MF` que contendrá una serie de parámetros de configuración de la aplicación. Se repiten algunos de los parámetros especificados en el fichero JAD, y se introducen algunos nuevos. Los parámetros requeridos son:

<code>MIDlet-Name</code>	Nombre de la <i>suite</i> .
<code>MIDlet-Version</code>	Versión de la <i>suite</i> .
<code>MIDlet-Vendor</code>	Autor (Proveedor) de la <i>suite</i> .
<code>MicroEdition-Profile</code>	Perfil requerido para ejecutar la <i>suite</i> . Podrá tomar el valor <code>MIDP-1.0</code> ó <code>MIDP-2.0</code> , según la versión de MIDP que utilicen las aplicaciones incluidas.
<code>MicroEdition-Configuration</code>	Configuración requerida para ejecutar la <i>suite</i> . Tomará el valor <code>CLDC-1.0</code> para las aplicaciones que utilicen esta configuración.

Deberemos incluir también información referente a cada MIDlet contenido en la *suite*. Esto lo haremos con la siguiente propiedad:

`MIDlet-<n>` Nombre, icono y clase principal del MIDlet número *n*

Los MIDlets se empezarán a numerar a partir del número 1, y deberemos incluir una línea de este tipo para cada MIDlet disponible en la *suite*. Daremos a cada MIDlet un nombre para que lo identifique el usuario, un icono de forma optativa, y el nombre de la clase principal que contiene dicho MIDlet.

Si especificamos un icono, deberá ser un fichero con formato PNG contenido dentro del JAR de la *suite*. Por ejemplo, para una *suite* con 3 MIDlets podemos tener la siguiente información:

```
MIDlet-Name: SuiteEjemplos
MIDlet-Version: 1.0.0
MIDlet-Vendor: Universidad de Alicante
MIDlet-Description: Aplicaciones de ejemplo para moviles.
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
MIDlet-1: Snake, /icons/snake.png,
es.ua.j2ee.serpiente.SerpMIDlet
MIDlet-2: TeleSketch, /icons/ts.png,
es.ua.j2ee.ts.TeleSketchMIDlet
MIDlet-3: Panj, /icons/panj.png, es.ua.j2ee.panj.PanjMIDlet
```

Además tenemos las mismas propiedades optativas que en el fichero JAD:

MIDlet-Icon	Icono para la <i>suite</i> .
MIDlet-Description	Descripción de la <i>suite</i> .
MIDlet-Info-URL	Dirección URL con información
MIDlet-Data-Size	Número mínimo de <i>bytes</i> para datos persistentes.

En este fichero, a diferencia del fichero JAD, no podremos introducir propiedades propias del usuario, ya que desde dentro de la aplicación no podremos acceder a los propiedades contenidas en este fichero.

2.2. Construcción de aplicaciones

Vamos a ver los pasos necesarios para construir una aplicación con J2ME a partir del código fuente, obteniendo finalmente los ficheros JAD y JAR con los que podremos instalar la aplicación en dispositivos móviles.

El primer paso será compilar las clases, obteniendo así el código intermedio que podrá ser ejecutado en una máquina virtual de Java. El problema es que este código intermedio es demasiado complejo para la KVM, por lo que deberemos realizar una preverificación del código, que simplifique el código intermedio de las clases y compruebe que no utiliza ninguna característica no soportada por la KVM. Una vez preverificado, deberemos empaquetar todos los ficheros de nuestra aplicación en un fichero JAR, y crear el fichero JAD correspondiente. En este momento podremos probar la aplicación en un emulador o en un dispositivo real. Los emuladores nos permitirán probar las aplicaciones directamente en nuestro ordenador sin tener que transferirlas a un dispositivo móvil real.

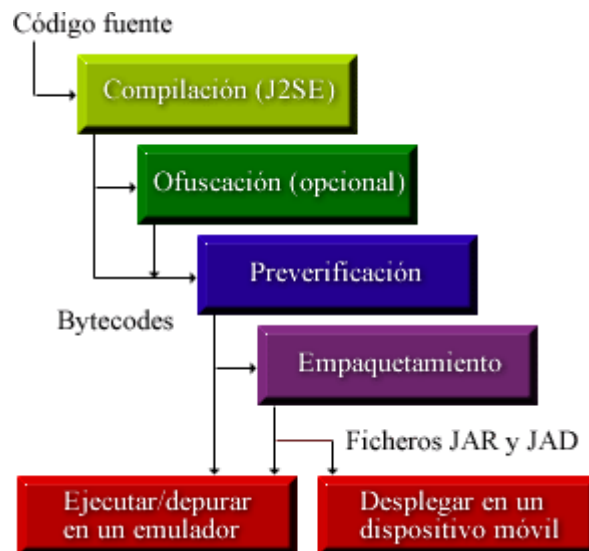


Figura 1. Proceso de construcción de aplicaciones

Necesitaremos tener instalado J2SE, ya que utilizaremos las mismas herramientas para compilar y empaquetar las clases. Además necesitaremos herramientas adicionales, ya que la máquina virtual reducida de los dispositivos CLDC necesita un código intermedio simplificado.

2.2.1. Compilación

Lo primero que deberemos hacer es compilar las clases de nuestra aplicación. Para ello utilizaremos el compilador incluido en J2SE, `javac`, por lo que deberemos tener instalada esta edición de Java.

Al compilar, el compilador buscará las clases que utilizamos dentro de nuestros programas para comprobar que estamos utilizándolas correctamente, y si utilizamos una clase que no existe, o bien llamamos a un método o accedemos a una propiedad que no pertenece a dicha clase nos dará un error de compilación. Java busca las clases en el siguiente orden:

- Clases de núcleo de Java (*bootstrap*)
- Extensiones instaladas
- *Classpath*

Si estamos compilando con el compilador de J2SE, por defecto considerará que las clases del núcleo de Java son las clases de la API de J2SE. Debemos evitar que esto ocurra, ya que estas clases no van a estar disponibles en los dispositivos MIDP que cuentan con una API reducida. Debemos hacer que tome como clases del núcleo las clases de la API de MIDP, esto lo haremos mediante el parámetro `bootclasspath` del compilador:

```
javac -bootclasspath ${ruta_midp}/midpapi.zip <ficheros .java>
```

Con esto estaremos compilando nuestras clases utilizando como API del núcleo de Java la API de MIDP. De esta forma, si dentro de nuestro programa

utilizásemos una clase que no pertenece a MIDP, aunque pertenezca a J2SE nos dará un error de compilación.

2.2.2. Ofuscación

Este es un paso opcional, pero recomendable. El código intermedio de Java incluye información sobre los nombres de los constructores, de los métodos y de los atributos de las clases e interfaces para poder acceder a esta información utilizando la API de *reflection* en tiempo de ejecución.

El contar con esta información nos permite descompilar fácilmente las aplicaciones, obteniendo a partir del código compilado unos fuentes muy parecidos a los originales. Lo único que se pierde son los comentarios y los nombres de las variables locales y de los parámetros de los métodos.

Esto será un problema si no queremos que se tenga acceso al código fuente de nuestra aplicación. Además incluir esta información en los ficheros compilados de nuestra aplicación harán que crezca el tamaño de estos ficheros ocupando más espacio, un espacio muy preciado en el caso de los dispositivos móviles con baja capacidad. Hemos de recordar que el tamaño de los ficheros JAR que soportan está limitado en muchos casos a 64kb o menos.

El proceso de ofuscación del código consiste en simplificar esta información, asignándoles nombres tan cortos como se pueda a las clases e interfaces y a sus constructores, métodos y atributos. De esta forma al descompilar obtendremos un código nada legible con nombres sin ninguna significancia.

Además conseguiremos que los ficheros ocupen menos espacio en disco, lo cuál será muy conveniente para las aplicaciones para dispositivos móviles con baja capacidad y reducida velocidad de descarga.

La ofuscación de código deberemos hacerla antes de la preverificación, dejando la preverificación para el final, y asegurándonos así de que el código final de nuestra aplicación funcionará correctamente en la KVM. Podemos utilizar para ello diferentes ofusadores, como ProGuard, RetroGuard o JODE. Deberemos obtener alguno de estos ofusadores por separado, ya que no se incluyen en J2SE ni en los kits de desarrollo para MIDP que veremos más adelante.

2.2.3. Preverificación

Con la compilación que acabamos de realizar hemos generado código intermedio que serán capaces de interpretar las máquinas virtuales Java. Sin embargo, máquina virtual de los dispositivos CLDC, la KVM, es un caso especial ya que las limitaciones de estos dispositivos hacen que tenga que ser bastante más reducida que otras máquinas virtuales para poder funcionar correctamente.

La máquina virtual de Java hace una verificación de las clases que ejecuta en ella. Este proceso de verificación es bastante complejo para la KVM, por lo que

deberemos reorganizar el código intermedio generado para facilitar esta tarea de verificación. En esto consiste la fase de preverificación que deberemos realizar antes de llevar la aplicación a un dispositivo real.

Además la KVM tiene una serie de limitaciones en cuanto al código que puede ejecutar en ella, como por ejemplo la falta de soporte para tipos de datos `float` y `double`. Con la compilación hemos comprobado que no estamos utilizando clases que no sean de la API de MIDP, pero se puede estar permitiendo utilizar características del lenguaje no soportada por la KVM. Es el proceso de preverificación el que deberá detectar el error en este caso.

Para realizar la preverificación necesitaremos la herramienta `preverify`. Esta herramienta no se incluye en J2SE, por lo que deberemos obtenerla por separado. Podemos encontrarla en diferentes kits de desarrollo o en implementaciones de referencia de MIDP, como veremos más adelante. Deberemos especificar como `classpath` la API que estemos utilizando para nuestra aplicación, como por ejemplo MIDP:

```
preverify -classpath ${ruta_midp}/midpapi.zip -d <directorio
destino>
<ficheros .class>
```

Preverificará los ficheros `.class` especificados y guardará el resultado de la preverificación en el directorio destino que indiquemos. Las clases generadas en este directorio destino serán las que tendremos que empaquetar en nuestra *suite*.

2.2.4. Creación de la suite

Una vez tenemos el código compilado preverificado, deberemos empaquetarlo todo en un fichero JAR para crear la *suite* con nuestra aplicación. En este fichero JAR deberemos empaquetar todos los ficheros `.class` generados, así como todos los recursos que nuestra aplicación necesite para funcionar, como pueden ser iconos, imágenes, sonidos, ficheros de datos, videos, etc.

Para empaquetar un conjunto de ficheros en un fichero JAR utilizaremos la herramienta `jar` incluida en J2SE. Además de las clases y los recursos, deberemos añadir al fichero `MANIFEST.MF` del JAR los parámetros de configuración que hemos visto en el punto anterior. Para ello crearemos un fichero de texto ASCII con esta información, y utilizaremos dicho fichero a la hora de crear el JAR. Utilizaremos la herramienta `jar` de la siguiente forma:

```
jar cmf <fichero manifest> <fichero jar> <ficheros a incluir>
```

Una vez hecho esto tendremos construido el fichero JAR con nuestra aplicación. Ahora deberemos crear el fichero JAD. Para ello podemos utilizar cualquier editor ASCII e incluir las propiedades necesarias. Como ya hemos generado el fichero JAR podremos indicar su tamaño dentro del JAD.

2.2.5. Prueba en emuladores

Una vez tengamos los ficheros JAR y JAD ya podremos probar la aplicación transfiriéndola a un dispositivo que soporte MIDP e instalándola en él. Sin embargo, hacer esto para cada prueba que queramos hacer es una tarea tediosa. Tendremos que limitarnos a hacer pruebas de tarde en tarde porque si no se perdería demasiado tiempo. Además no podemos contar con que todos los desarrolladores tengan un móvil con el que probar las aplicaciones.

Si queremos ir probando con frecuencia los avances que hacemos en nuestro programa lo más inmediato será utilizar un emulador. Un emulador es una aplicación que se ejecuta en nuestro ordenador e imita (emula) el comportamiento del móvil. Entonces podremos ejecutar nuestras aplicaciones dentro de un emulador y de esta forma para la aplicación será prácticamente como si se estuviese ejecutando en un móvil con soporte para MIDP. Así podremos probar las aplicaciones en nuestro mismo ordenador sin necesitar tener que llevarla a otro dispositivo.

Además podremos encontrar emuladores que imitan distintos modelos de móviles, tanto existentes como ficticios. Esta es una ventaja más de tener emuladores, ya que si probamos en dispositivos reales necesitaríamos o bien disponer de varios de ellos, o probar la aplicación sólo con el que tenemos y arriesgarnos a que no vaya en otros modelos. Será interesante probar emuladores de teléfonos móviles con distintas características (distinto tamaño de pantalla, colores, memoria) para comprobar que nuestra aplicación funciona correctamente en todos ellos.

Podemos encontrar emuladores proporcionados por distintos fabricantes, como Nokia, Siemens o Sun entre otros. De esta forma tendremos emuladores que imitan distintos modelos de teléfonos Nokia o Siemens existentes. Sun proporciona una serie de emuladores genéricos que podremos personalizar dentro dentro de su kit de desarrollo que veremos en el próximo apartado.

2.2.6. Prueba de la aplicación en dispositivos reales

Será importante también, una vez hayamos probado la aplicación en emuladores, probarla en un dispositivo real, ya que puede haber cosas que funcionen bien en emuladores pero no lo hagan cuando lo llevamos a un dispositivo móvil de verdad. Los emuladores pretenden imitar en la medida de lo posible el comportamiento de los dispositivos reales, pero siempre hay diferencias, por lo que será importante probar las aplicaciones en móviles de verdad antes de distribuir la aplicación.

La forma más directa de probar la aplicación en dispositivos móviles es conectarlos al PC mediante alguna de las tecnologías disponibles (*bluetooth*, IrDA, cable serie o USB) y copiar la aplicación del PC al dispositivo. Una vez copiada, podremos instalarla desde el mismo dispositivo, y una vez hecho esto ya podremos ejecutarla.

Por ejemplo, los emuladores funcionan bien con código no preverificado, o incluso muchos de ellos funcionan con los ficheros una vez compilados sin necesidad de empaquetarlos en un JAR.

2.2.7. Despliegue

Entendemos por despliegue de la aplicación la puesta en marcha de la misma, permitiendo que el público acceda a ella y la utilice. Para desplegar una aplicación MIDP deberemos ponerla en algún lugar accesible, al que podamos conectarnos desde los móviles y descargarla.

Podremos utilizar cualquier servidor web para ofrecer la aplicación en Internet, como puede ser por ejemplo el Tomcat. Deberemos configurar el servidor de forma que reconozca correctamente los tipos de los ficheros JAR y JAD. Para ello asociaremos estas extensiones a los tipos MIME:

```
.jad    text/vnd.sun.j2me.app-descriptor
.jar    application/java-archive
```

Además en el fichero JAD, deberemos especificar como URL la dirección de Internet donde finalmente hemos ubicado el fichero JAR.

2.3. Kits de desarrollo

Para simplificar la tarea de desarrollar aplicaciones MIDP, tenemos disponibles distintos kits de desarrollo proporcionados por distintos fabricantes, como Sun o Nokia. Antes de instalar estos kits de desarrollo deberemos tener instalado J2SE. Estos kits de desarrollo contienen todos los elementos necesarios para, junto a J2SE, crear aplicaciones MIDP:

- **API de MIDP.** Librería de clases que componen la API de MIDP necesaria para poder compilar las aplicaciones que utilicen esta API.
- **Preverificador.** Herramienta necesaria para realizar la fase de preverificación del código.
- **Emuladores.** Nos servirán para probar la aplicación en nuestro propio PC, sin necesidad de llevarla a un dispositivo real.
- **Entorno para la creación de aplicaciones.** Estos kits normalmente proporcionarán una herramienta que nos permita automatizar el proceso de construcción de aplicaciones MIDP que hemos visto en el punto anterior.
- **Herramientas adicionales.** Podemos encontrar herramientas adicionales, de configuración, personalización de los emuladores, despliegue de aplicaciones, conversores de formatos de ficheros al formato reconocido por MIDP, etc.

Vamos a centrarnos en estudiar cómo trabajar con el kit de desarrollo de Sun, ya que es el más utilizado por ser genérico y el que mejor se integra con otros entornos y herramientas. Este kit recibe el nombre de *Wireless Toolkit* (WTK). Existen diferentes versiones de WTK, cada una de ellas adecuada para un determinado tipo de aplicaciones:

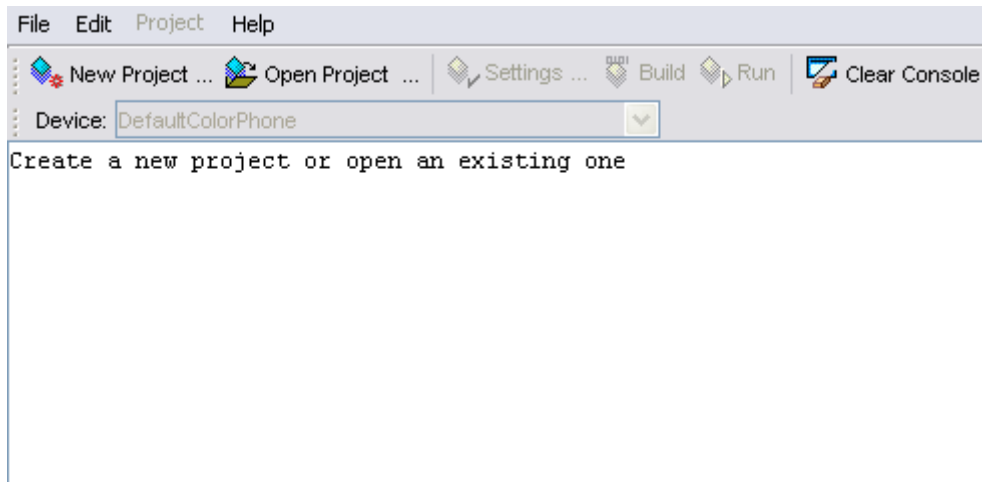
- **WTK 1.0.4:** Soporta MIDP 1.0 y CLDC 1.0. Será adecuado para desarrollar aplicaciones para móviles que soporten sólo esta versión de MIDP, aunque también funcionarán con modelos que soporten versiones posteriores de MIDP, aunque en estos casos no estaremos aprovechando al máximo las posibilidades del dispositivo.
- **WTK 2.0:** Soporta MIDP 2.0, CLDC 1.0 y las APIs opcionales WMA y MMAPI. Será adecuado para realizar aplicaciones para móviles MIDP 2.0, pero no para aquellos que sólo soporten MIDP 1.0, ya que las aplicaciones que hagamos con este kit pueden utilizar elementos que no estén soportados por MIDP 1.0 y por lo tanto es posible que no funcionen cuando las desplaguemos en este tipo de dispositivos. Además en esta versión se incluyen mejoras como la posibilidad de probar las aplicaciones vía OTA.
- **WTK 2.1:** Soporta MIDP 2.0, MIDP 1.0, CLDC 1.1 (con soporte para punto flotante), CLDC 1.0, y las APIs opcionales WMA, MMAPI y WSA. En este caso podemos configurar cuál es la plataforma para la que desarrollamos cada aplicación. Por lo tanto, esta versión será adecuada para desarrollar para cualquier tipo de móviles. Puede generar aplicaciones totalmente compatibles con JTWI.
- **WTK 2.2:** Aparte de todo lo soportado por WTK 2.1, incorpora las APIs para gráficos 3D (JSR-184) y bluetooth (JSR-82). Nos centraremos en el estudio de esta versión por ser la que incorpora un mayor número de APIs en el momento de la escritura de este texto, además de ser genérica (se puede utilizar para cualquier versión de MIDP).

2.3.1. Creación de aplicaciones con WTK

Hemos visto en el punto anterior los pasos que deberemos seguir para probar nuestras aplicaciones MIDP: compilar, preverificar, empaquetar, crear el archivo JAD y ejecutar en un emulador.

Normalmente, mientras escribimos el programa querremos probarlo numerosas veces para comprobar que lo que llevamos hecho funciona correctamente. Si cada vez que queremos probar el programa tuviésemos que realizar todos los pasos vistos anteriormente de forma manual programar aplicaciones MIDP sería una tarea tediosa. Además requeriría aprender a manejar todas las herramientas necesarias para realizar cada paso en la línea de comando.

Por ello los kits de desarrollo, y concretamente WTK, proporcionan entornos para crear aplicaciones de forma automatizada, sin tener que trabajar directamente con las herramientas en línea de comando. En el caso de WTK, esta herramienta recibe el nombre de `ktoolbar`:



Este entorno nos permitirá construir la aplicación a partir del código fuente, pero no proporciona ningún editor de código fuente, por lo que tendremos que escribir el código fuente utilizando cualquier editor externo. Otra posibilidad es integrar WTK en algún entorno de desarrollo integrado (IDE) de forma que tengamos integrado el editor con todas las herramientas para construir las aplicaciones facilitando más aun la tarea del desarrollador. En el siguiente punto veremos como desarrollar aplicaciones utilizando un IDE.

Directorio de aplicaciones

Este entorno de desarrollo guarda todas las aplicaciones dentro de un mismo directorio de aplicaciones. Cada aplicación estará dentro de un subdirectorio dentro de este directorio de aplicaciones, cuyo nombre corresponderá al nombre de la aplicación.

Por defecto, este directorio de aplicaciones es el directorio `${WTK_HOME}/apps`, pero podemos modificarlo añadiendo al fichero `ktools.properties` la siguiente línea:

```
kvem.apps.dir: <directorio de aplicaciones>
```

Además, dentro de este directorio hay un directorio `lib`, donde se pueden poner las librerías externas que queremos que utilicen todas las aplicaciones. Estas librerías serán ficheros JAR cuyo contenido será incorporado a las aplicaciones MIDP que creemos, de forma que podamos utilizar esta librería dentro de ellas.

Por ejemplo, después de instalar WTK podemos encontrar a parte del directorio de librerías una serie de aplicaciones de demostración instaladas. El directorio de aplicaciones puede contener por ejemplo los siguientes directorios (en el caso de WTK 2.1):

```
audiodemo  
demos  
FPDemo  
games  
JSR172Demo
```

```
lib  
mmademo  
NetworkDemo  
photoalbum  
SMSDemo  
tmplib  
UIDemo
```

Tendremos por lo tanto las aplicaciones `games`, `demos`, `photoalbum`, y `UIDemo`. El directorio `tmplib` lo utiliza el entorno para trabajar de forma temporal con las librerías del directorio `lib`.

NOTA: Dado que se manejan gran cantidad de herramientas y emuladores independientes en el desarrollo de las aplicaciones MIDP, es recomendable que el directorio donde está instalada la aplicación (ni ninguno de sus ascendientes) contenga espacios en blanco, ya que algunas aplicaciones puede fallar en estos casos.

Estructura de las aplicaciones

Dentro del directorio de cada aplicación, se organizarán los distintos ficheros de los que se compone utilizando la siguiente estructura de directorios:

```
bin  
lib  
res  
src  
classes  
tmpclasses  
tmplib
```

Deberemos crear el código fuente de la aplicación dentro del directorio `src`, creando dentro de este directorio la estructura de directorios correspondiente a los paquetes a los que pertenezcan nuestras clases.

En `res` guardaremos todos los recursos que nuestra aplicación necesite, pudiendo crear dentro de este directorio la estructura de directorios que queramos para organizar estos recursos.

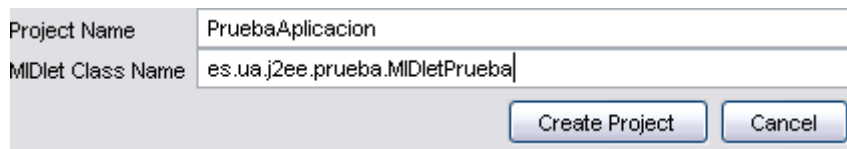
Por último, en `lib` deberemos poner las librerías adicionales que queramos incorporar a nuestra aplicación. Pondremos en este directorio el fichero JAR con la librería de clases que queramos añadir. Lo que se hará será añadir todas las clases contenidas en estas librerías, así como las contenidas en las librerías globales que hemos visto anteriormente, al fichero JAR que creemos para nuestra aplicación.

NOTA: Si lo que queremos es utilizar en nuestra aplicación una API opcional soportada por el móvil, no debemos introducirla en este directorio. En ese caso sólo deberemos añadirla al `classpath` a la hora de compilar, pero no introducirla en este directorio ya que el móvil ya cuenta con su propia implementación de dicha librería y no deberemos añadir la implementación de referencia que tenemos en el ordenador al paquete de nuestra aplicación.

Esto es todo lo que tendremos que introducir nosotros. Todo lo demás será generado automáticamente por la herramienta `ktoolbar` como veremos a continuación. En el directorio `classes` se generarán las clases compiladas y preverificadas de nuestra aplicación, y en `bin` tendremos finalmente los ficheros JAR y JAD para desplegar nuestra aplicación.

Creación de una nueva aplicación

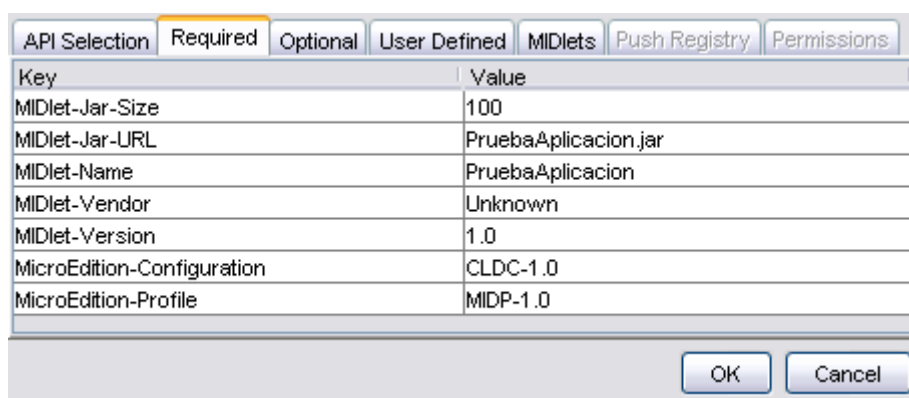
Cuando queramos crear una nueva aplicación, lo primero que haremos será pulsar el botón **"New Project ..."** para abrir el asistente de creación de aplicaciones. Lo primero que nos pedirá es el nombre que queremos darla a la aplicación, y el nombre de la clase principal (MIDlet) que vamos a crear:



Project Name	PruebaAplicacion
MIDlet Class Name	es.ua.j2ee.prueba.MIDletPrueba
<div>Create Project Cancel</div>	

Debemos indicar aquí un nombre para la aplicación (*Project Name*), que será el nombre del directorio donde se guardará la aplicación. Además deberemos indicar el nombre de la clase correspondiente al MIDlet principal de la *suite* (*MIDlet Class Name*). Es posible que nosotros todavía no hayamos creado esta clase, por lo que deberemos indicar el nombre que le asignaremos cuando la creamos. De todas formas este dato puede ser modificado más adelante.

Una vez hayamos introducido estos datos, pulsamos **"Create Project"** y nos aparecerá una ficha para introducir todos los datos necesarios para crear el fichero JAD y el `MANIFEST.MF` del JAR. Con los datos introducidos en la ventana anterior habrá rellenado todos los datos necesarios, pero nosotros podemos modificarlos manualmente si queremos personalizarlo más. La primera ficha nos muestra los datos obligatorios:



Key	Value
MIDlet-Jar-Size	100
MIDlet-Jar-URL	PruebaAplicacion.jar
MIDlet-Name	PruebaAplicacion
MIDlet-Vendor	Unknown
MIDlet-Version	1.0
MicroEdition-Configuration	CLDC-1.0
MicroEdition-Profile	MIDP-1.0

OK Cancel

Como nombre de la *suite* y del JAR habrá tomado por defecto el nombre del proyecto que hayamos especificado. Será conveniente modificar los datos del fabricante y de la versión, para adaptarlos a nuestra aplicación. No debemos preocuparnos por especificar el tamaño del JAR, ya que este dato será actualizado de forma automática cuando se genere el JAR de la aplicación.

En la segunda pestaña tenemos los datos opcionales que podemos introducir en estos ficheros:

Key	Value
MIDlet-Data-Size	
MIDlet-Delete-Confirm	
MIDlet-Delete-Notify	
MIDlet-Description	
MIDlet-Icon	
MIDlet-Info-URL	
MIDlet-Install-Notify	

Estos datos están vacíos por defecto, ya que no son necesarios, pero podemos darles algún valor si lo deseamos. Estas son las propiedades opcionales que reconoce el AMS. Si queremos añadir propiedades propias de nuestra aplicación, podemos utilizar la tercera pestaña:

Key	Value
msg.bienvenida	Hola mundo!

Aquí podemos añadir o eliminar cualquier otra propiedad que queramos definir para nuestra aplicación. De esta forma podemos parametrizarlas. En el ejemplo de la figura hemos creado una propiedad `msg.bienvenida` que contendrá el texto de bienvenida que mostrará nuestra aplicación. De esta forma podremos modificar este texto simplemente modificando el valor de la propiedad en el JAD, sin tener que recompilar el código.

En la última pestaña tenemos los datos de los MIDlets que contiene la *suite*. Por defecto nos habrá creado un único MIDlet:

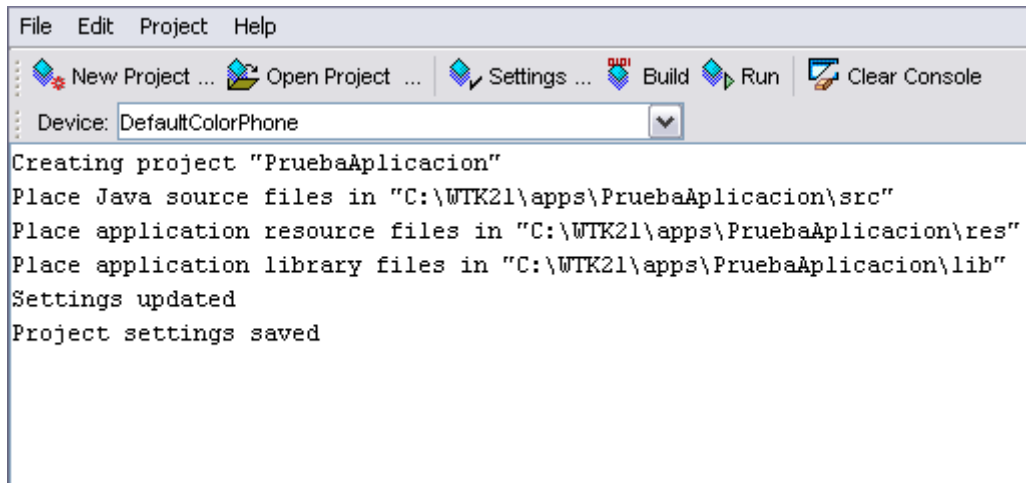
Key	Name	Icon	Class
MIDlet-1	PruebaAplicacion	PruebaAplicacion.png	es.ua.j2ee.prueba.MI...

Por defecto le habrá dado a este MIDlet el mismo nombre que a la aplicación, es decir, el nombre del proyecto que hemos especificado, al igual que ocurre

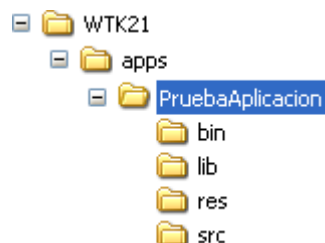
con el nombre del icono. Como clase correspondiente al MIDlet habrá introducido el nombre de la clase que hemos especificado anteriormente.

Dado que una *suite* puede contener más de un MIDlet, desde esta pestaña podremos añadir tantos MIDlets como queramos, especificando para cada uno de ellos su nombre, icono (de forma opcional) y clase.

Una vez terminemos de introducir todos estos datos, pulsamos **"OK"** y en la ventana principal nos mostrará el siguiente mensaje:



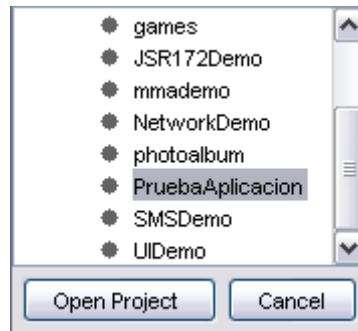
Con este mensaje nos notifica el directorio donde se ha creado la aplicación, y los subdirectorios donde debemos introducir el código fuente, recursos y librerías externas de nuestra aplicación. Se habrá creado la siguiente estructura de directorios en el disco:



En el directorio `bin` se habrán creado los ficheros `JAD` y `MANIFEST.MF` provisionales con los datos que hayamos introducido. Los demás directorios estarán vacíos, deberemos introducir en ellos todos los componentes de nuestra aplicación.

Abrir una aplicación ya existente

Si tenemos una aplicación ya creada, podemos abrirla desde el entorno para continuar trabajando con ella. Para abrir una aplicación pulsamos **"Open Project ..."** y nos mostrará la siguiente ventana con las aplicaciones disponibles:



Podemos seleccionar cualquiera de ellas y abrirla pulsando **"Open Project"**. Una vez abierta podremos modificar todos los datos que hemos visto anteriormente correspondientes a los ficheros JAD y `MANIFEST.MF` pulsando sobre el botón **"Settings ..."**.

Además podremos compilarla, empaquetarla y probarla en cualquier emulador instalado como veremos a continuación.

2.3.2. Compilación y empaquetamiento

Una vez hemos escrito el código fuente de nuestra aplicación MIDP (en el directorio `src`) y hemos añadido los recursos y las librerías necesarias para ejecutarse dicha aplicación (en los directorios `res` y `lib` respectivamente) podremos utilizar la herramienta `ktoolbar` para realizar de forma automatizada todos los pasos para la construcción de la aplicación. Vamos a ver ahora como realizar este proceso.

Compilación

Para compilar el código fuente de la aplicación simplemente deberemos pulsar el botón **"Build"** o ir a la opción del menú **Project > Build**. Con esto compilará y preverificará de forma automática todas las clases de nuestra aplicación, guardando el resultado en el directorio `classes` de nuestro proyecto.

Para compilar las clases utilizará como *classpath* la API proporcionada por el emulador seleccionado actualmente. Para los emuladores distribuidos con WTK estas clases serán las API básica de MIDP (1.0 ó 2.0 según la versión de WTK instalada). Sin embargo, podemos incorporar emuladores que soporten APIs adicionales, como por ejemplo MMAPi para dar soporte a elementos multimedia, o APIs propietarias de distintas compañías como Nokia. En caso de tener seleccionado un emulador con alguna de estas APIs adicionales, estas APIs también estarán incluidas en el *classpath*, por lo que podremos compilar correctamente programas que las utilicen. El emulador seleccionado aparece en el desplegable **Device**.

Ofuscación

El entorno de desarrollo de WTK también nos permitirá ofuscar el código de forma automática. Este paso es opcional, y si queremos que WTK sea capaz de utilizar la ofuscación deberemos descargar alguno de los ofusadores

soportados por este entorno, como *ProGuard* (en WTK 2.X) o *RetroGuard* (en WTK 1.0). Estos ofuscadores son proporcionados por terceros.

Una vez tenemos uno de estos ofuscadores, tendremos un fichero JAR con las clases del ofuscador. Lo que deberemos hacer para instalarlo es copiar este fichero JAR al directorio `${WTK_HOME}/bin`. Una vez tengamos el fichero JAR del ofuscador en este directorio, WTK podrá utilizarlo de forma automática para ofuscar el código.

La ofuscación la realizará WTK en el mismo paso de la creación del paquete JAR, en caso de disponer de un ofuscador instalado, como veremos a continuación.

Empaquetamiento

Para poder instalar una aplicación en el móvil y distribuirla, deberemos generar el fichero JAR con todo el contenido de la aplicación. Para hacer esto de forma automática deberemos ir al menú **Project > Package**. Dentro de este menú tenemos dos opciones:

- **Create Package**
- **Create Obfuscated Package**

Ambas realizan todo el proceso necesario para crear el paquete de forma automática: compilan los fuentes, ofuscan (sólo en el segundo caso), preverifican y empaquetan las clases resultantes en un fichero JAR. Por lo tanto no será necesario utilizar la opción **Build** previamente, ya que el mismo proceso de creación del paquete ya realiza la compilación y la preverificación.

Una vez construido el fichero JAR lo podremos encontrar en el directorio `bin` de la aplicación. Además este proceso actualizará de forma automática el fichero JAD, para establecer el tamaño correcto del fichero JAR que acabamos de crear en la propiedad correspondiente.

2.3.3. Ejecución en emuladores

Dentro del mismo entorno de desarrollo de WTK podemos ejecutar la aplicación en diferentes emuladores que haya instalados para probarla. Podemos seleccionar el emulador a utilizar en el cuadro desplegable **Device** de la ventana principal de `ktoolbar`.

Para ejecutar la aplicación en el emulador seleccionado solo debemos pulsar el botón **"Run"** o la opción del menú **Project > Run**. Normalmente, para probar la aplicación en un emulador no es necesario haber creado el fichero JAR, simplemente con las clases compiladas es suficiente. En caso de ejecutarse sin haber compilado las clases, el entorno las compilará de forma automática.

Sin embargo, hay algunos emuladores que sólo funcionan con el fichero JAR, por lo que en este caso deberemos crear el paquete antes de ejecutar el

emulador. Esto ocurre por ejemplo con algún emulador proporcionado por Nokia.

Por ejemplo, los emuladores de teléfonos móviles proporcionados con WTK 2.2 son:

- **DefaultColorPhone**. Dispositivo con pantalla a color.
- **DefaultGrayPhone**. Dispositivo con pantalla monocroma.
- **MediaControlSkin**. Dispositivo con teclado orientado a la reproducción de elementos multimedia.
- **QwertyDevice**. Dispositivo con teclado de tipo QWERTY.

Además de estos, podemos incorporar otros emuladores al kit de desarrollo. Por ejemplo, los emuladores proporcionados por Nokia, imitando diversos modelos de teléfonos móviles de dicha compañía, pueden ser integrados fácilmente en WTK.

Para integrar los emuladores de teléfonos Nokia en WTK simplemente tendremos que instalar estos emuladores en el directorio `${WTK_HOME}/wtklib/devices`. Una vez instalados en este directorio, estos emuladores estarán disponibles dentro del kit de desarrollo, de forma que podremos seleccionarlos en el cuadro desplegable como cualquier otro emulador.



MinimumPhone



DefaultColorPhone



Motorola i85s



Nokia 6310



Nokia 7210

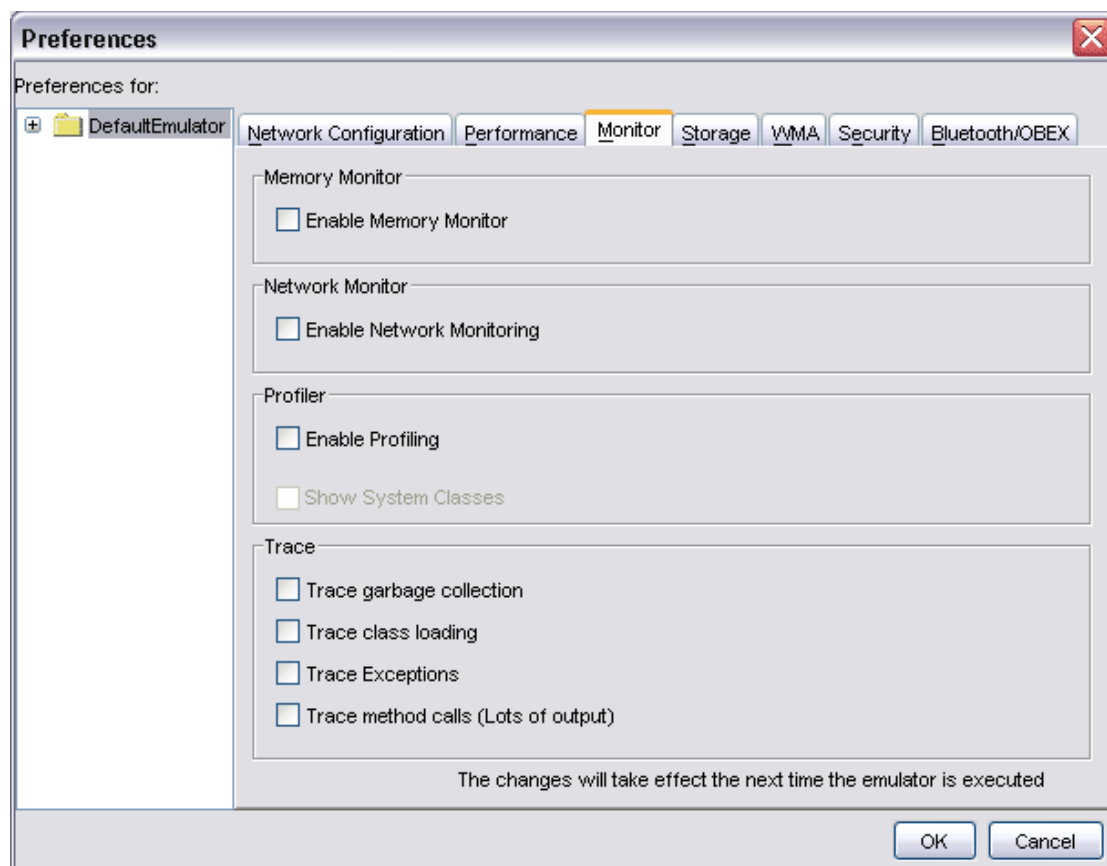


Nokia Series 60

Podemos encontrar además emuladores proporcionados por otras compañías. WTK también nos permite personalizar los emuladores, cambiando su aspecto y características para adaptarlos a nuestras necesidades.

Optimización

En WTK, además de los emuladores, contamos con herramientas adicionales que nos ayudarán a optimizar nuestras aplicaciones. Desde la ventana de preferencias podemos activar distintos monitores que nos permitirán monitorizar la ocupación de memoria y el tráfico en la red:



Será conveniente utilizar estos monitores para medir el consumo de recursos de nuestra aplicación e intentar reducirlo al mínimo.

En cuanto a la memoria, deberemos intentar que el consumo sea lo menor posible y que nunca llegue a pasar de un determinado umbral. Si la memoria creciese sin parar en algún momento la aplicación fallaría por falta de memoria al llevarla a nuestro dispositivo real.

Es importante también intentar minimizar el tráfico en la red, ya que en los dispositivos reales este tipo de comunicaciones serán lentas y caras.

Desde esta ventana de preferencias podemos cambiar ciertas características de los emuladores, como el tamaño máximo de la memoria o la velocidad de su procesador. Es conveniente intentar utilizar los parámetros más parecidos a los dispositivos para los cuales estemos desarrollando, sobretodo en cuanto a consumo de memoria, para asegurarnos de que la aplicación seguirá funcionando cuando la llevamos al dispositivo real.

2.3.4. Provisionamiento OTA

Hemos visto como probar la aplicación directamente utilizando emuladores. Una vez generados los ficheros JAR y JAD también podremos copiarlos a un dispositivo real y probarlos ahí.

Sin embargo, cuando un usuario quiera utilizar nuestra aplicación, normalmente lo hará vía OTA (Over The Air), es decir, se conectará a la dirección donde hayamos publicado nuestra aplicación y la descargará utilizando la red de nuestro móvil.

Para desplegar una aplicación de forma que sea accesible vía OTA, simplemente deberemos:

- Publicar los ficheros JAR y JAD de nuestra aplicación en un servidor web, que sea accesible a través de Internet.
- Crear un documento web que tenga un enlace al fichero JAD de nuestra aplicación. Este documento puede ser por ejemplo WML, XHTML o XHTML.
- Configurar el servidor web para que asocie los ficheros JAD y JAR al tipo MIME adecuado, tal como hemos visto anteriormente.
- Editar el fichero JAD. En la línea donde hace referencia a la URL del fichero JAR deberemos indicar la URL donde hemos desplegado realmente el fichero JAR.

Una vez está desplegada la aplicación vía OTA, el provisionamiento OTA consistirá en los siguientes pasos:

- El usuario accede con su móvil a nuestra dirección de Internet utilizando un navegador web.
- Selecciona el enlace que lleva al fichero JAD de nuestra aplicación
- El navegador descarga el fichero JAD

- El fichero JAD será abierto por el AMS del móvil, que nos mostrará sus datos y nos preguntará si queremos instalar la aplicación.
- Si respondemos afirmativamente, se descargará el fichero JAR utilizando la URL que se indica en el fichero JAD.
- Instalará la aplicación en el móvil.
- Una vez instalada, se añadirá la aplicación a la lista de aplicaciones instaladas en nuestro móvil. Desde esta lista el usuario del móvil podrá ejecutar la aplicación cada vez que quiera utilizarla. Cuando el usuario no necesite la aplicación, podrá desinstalarla para liberar espacio en el medio de almacenamiento del móvil.

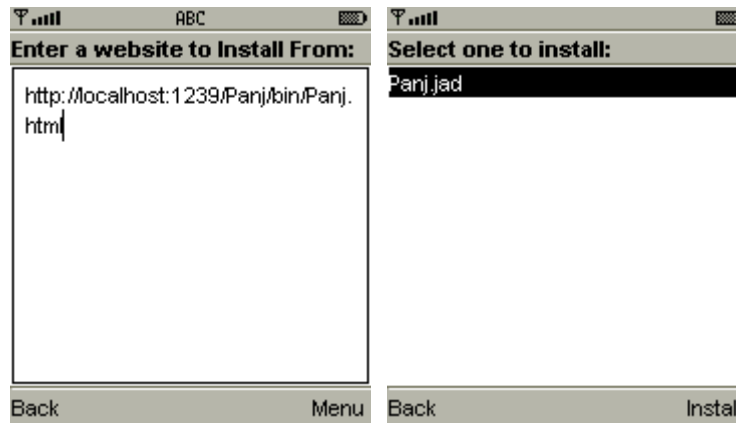
A partir de WTK 2.0 podemos simular en los emuladores el provisionamiento OTA de aplicaciones. Ejecutando la aplicación *OTA Provisioning* se nos abrirá el emulador que tengamos configurado por defecto y nos dará la opción de instalar aplicaciones (*Install Application*) vía OTA. Si pulsamos sobre esta opción nos pedirá la URL donde hayamos publicado nuestra aplicación.



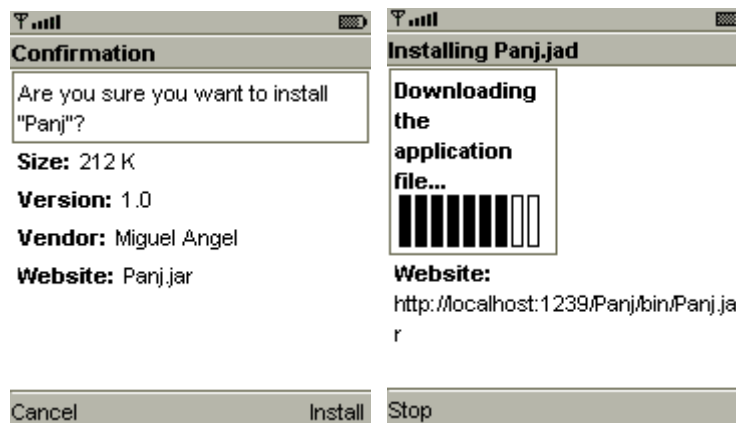
De esta forma podremos probar aplicaciones publicadas en algún servidor de Internet. Para probar nuestras aplicaciones utilizando este procedimiento deberemos desplegar previamente nuestra aplicación en un servidor web y utilizar la dirección donde la hayamos desplegados para instalar la aplicación desde ese lugar.

Este procedimiento puede ser demasiado costoso si queremos probar la aplicación repetidas veces utilizando este procedimiento, ya que nos obligaría, para cada nueva prueba que quisiésemos hacer, a volver a desplegar la aplicación en el servidor web.

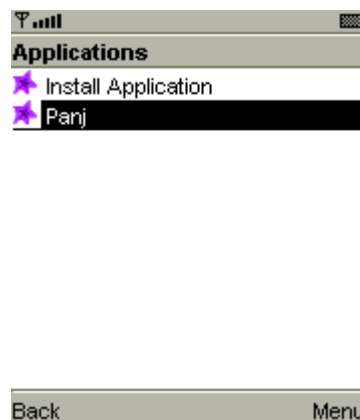
La aplicación `ktoolbar` nos ofrece una facilidad con el que simular el provisionamiento OTA utilizando un servidor web interno, de forma que no tendremos que publicar la aplicación manualmente para probarla. Para ello, abriremos nuestro proyecto en `ktoolbar` y seleccionaremos la opción **Project > Run via OTA**. Con esto, automáticamente nos rellenará la dirección de donde queremos instalar la aplicación con la dirección interna donde está desplegada:



Una vez introducida la dirección del documento web donde tenemos publicada nuestra aplicación, nos mostrará la lista de enlaces a ficheros JAD que tengamos en esa página. Podremos seleccionar uno de estos enlaces para instalar la aplicación. En ese momento descargará el fichero JAD y nos mostrará la información contenida en él, preguntándonos si queremos instalar la aplicación:



Si aceptamos la instalación de la aplicación, pulsando sobre *Install*, descargará el fichero JAR con la aplicación y lo instalará. Ahora veremos esta aplicación en la lista de aplicaciones instaladas:



Desde esta lista podremos ejecutar la aplicación e instalar nuevas aplicaciones que se vayan añadiendo a esta lista. Cuando no necesitemos esta aplicación desde aquí también podremos desinstalarla.

2.4. Antenna

La herramienta `ant` nos permite automatizar tareas como la compilación, empaquetamiento, despliegue o ejecución de aplicaciones. Es similar a la herramienta `make`, pero con la ventaja de que es totalmente independiente de la plataforma, ya que en lugar de utilizar comandos nativos utiliza clases Java para realizar las tareas.

Tiene una serie de tareas definidas, que servirán para compilar clases, empaquetar en ficheros JAR, ejecutar aplicaciones, etc. Todas estas tareas están implementadas mediante clases Java. Además, nos permitirá añadir nuevas tareas, incorporando una librería de clases Java que las implemente.

Antenna es una librería de tareas para *ant* que nos permitirán trabajar con aplicaciones MIDP. Entre estas tareas encontramos la compilación y el empaquetamiento (con preverificación y ofuscación), la creación de los ficheros JAD y `MANIFEST.MF`, y la ejecución de aplicaciones en emuladores.

Para realizar estas tareas utiliza WTK, por lo que necesitaremos tener este kit de desarrollo instalado. Los emuladores que podremos utilizar para ejecutar las aplicaciones serán todos aquellos emuladores instalados en WTK.

Para utilizar estas tareas deberemos copiar el JAR de *Antenna* al directorio de librerías de *ant*, o bien introducir este JAR en el *classpath* para que esté localizable.

2.4.1. Declaraciones

Dentro del fichero `build.xml` de *ant* deberemos especificar como propiedad `wtk.home` el directorio donde tenemos instalado WTK:

```
<property name="wtk.home" value="c:\WTK104"/>
```

Además, debemos declarar las tareas de *Antenna* para poder utilizarlas dentro de dicho fichero:

```
<taskdef name="wtkjad"
  classname="de.pleumann.antenna.WtkJad"/>
<taskdef name="wtkbuild"
  classname="de.pleumann.antenna.WtkBuild"/>
<taskdef name="wtkpackage"
  classname="de.pleumann.antenna.WtkPackage"/>
<taskdef name="wtkrun"
  classname="de.pleumann.antenna.WtkRun"/>
<taskdef name="wtkpreverify"
  classname="de.pleumann.antenna.WtkPreverify"/>
<taskdef name="wtkobfuscate"
  classname="de.pleumann.antenna.WtkObfuscate"/>
```

2.4.2. Tareas

Para crear el fichero JAD utilizaremos la tarea `wtkjad`, a la que debemos proporcionar la siguiente información:

```
<wtkjad jadfile="${jad.file}"
  jarfile="${jar.file}"
  name="${midlet.name}"
  vendor="${vendor.name}"
  version="${midlet.version}">
  <midlet name="${midlet.name}"
    class="${midlet.class}"/>
  <attribute name="msg.bienvenida" value="Hola mundo"/>
</wtkjad>
```

Para compilar utilizaremos `wtkbuild`. Esta tarea nos permite preverificar, pero podemos dejar esto para el paso siguiente:

```
<wtkbuild srcdir="${src.home}"
  destdir="${build.home}"
  preverify="false"/>
```

Para empaquetar utilizaremos `wtkpackage`. En este paso podremos ofuscar y preverificar el código:

```
<wtkpackage jarfile="${jar.file}"
  jadfile="${jad.file}"
  obfuscate="${obfuscate}"
  preverify="true">
  <fileset dir="${build.home}"/>
  <fileset dir="${res.home}"/>
</wtkpackage>
```

Aquí debemos especificar los ficheros que vamos a incluir en el paquete JAR mediante las etiquetas `fileset`. En este ejemplo estamos incluyendo todos los ficheros que haya en el directorio de clases compiladas y en el directorio de recursos. Si queremos incluir librerías JAR (o ZIP) podemos utilizar las siguientes etiquetas:

```
<zipfileset src="libreria.zip"/>
<zipgroupfileset dir="lib"/>
```

La primera de ellas (`zipfileset`) nos permite especificar una única librería para incluir. Con la segunda (`zipgroupfileset`) se incluirán todas las librerías del directorio especificado.

Por último, para ejecutar la aplicación en un emulador utilizaremos la tarea `wtkrun`:

```
<wtkrun jadfile="${jad.file}"
  heapsize="0"
  device="${emulator.name}"
  wait="true"/>
```

Con esto podremos crear nuestras aplicaciones MIDP utilizando *ant*, en lugar de utilizar la aplicación `ktoolbar` de WTK.

2.4.3. Ejemplo completo

Vamos a ver un ejemplo completo genérico. Este código podrá ser aprovechado para la mayoría de aplicaciones J2ME, simplemente cambiando las propiedades declaradas al principio del fichero para introducir los datos correspondientes a cada aplicación:

```
<?xml version="1.0"?>

<project name="Prueba" default="run" basedir=".">

  <!-- Propiedades del MIDlet -->
  <property name="jad.file" value="aplic.jad"/>
  <property name="jar.file" value="aplic.jar"/>
  <property name="vendor.name" value="Miguel Angel"/>
  <property name="midlet.version" value="1.0.0"/>
  <property name="midlet.name" value="Prueba"/>
  <property name="midlet.class"
    value="es.ua.j2ee.prueba.PrimeroMIDlet"/>

  <!-- Propiedades del entorno -->
  <property name="emulator.name"
    value="DefaultColorPhoneEmulator"/>
  <property name="base.home" value="."/>
  <property name="src.home" value="${base.home}/src"/>
  <property name="build.home" value="${base.home}/classes"/>
  <property name="res.home" value="${base.home}/res"/>
  <property name="obfuscate" value="true"/>

  <!-- Establece el directorio de WTK (requerido por Antenna). -
  ->
  <property name="wtk.home" value="c:\WTK104"/>

  <!-- Define las tareas de Antenna. -->
  <taskdef name="wtkjad"
    classname="de.pleumann.antenna.WtkJad"/>
  <taskdef name="wtkbuild"
    classname="de.pleumann.antenna.WtkBuild"/>
  <taskdef name="wtkpackage"
    classname="de.pleumann.antenna.WtkPackage"/>
  <taskdef name="wtkrun"
    classname="de.pleumann.antenna.WtkRun"/>
  <taskdef name="wtkpreverify"
    classname="de.pleumann.antenna.WtkPreverify"/>
  <taskdef name="wtkobfuscate"
    classname="de.pleumann.antenna.WtkObfuscate"/>

  <!-- Crea el fichero JAD. -->
  <target name="jad">
    <wtkjad jadfile="${jad.file}"
      jarfile="${jar.file}"
      name="${midlet.name}"
      vendor="${vendor.name}"
      version="${midlet.version}">
      <midlet name="${midlet.name}" class="${midlet.class}"/>
    </wtkjad>
```

```

</target>

<!-- Limpia el directorio de compilacion. -->
<target name="clean">
  <delete dir="${build.home}"/>
  <mkdir dir="${build.home}"/>
</target>

<!-- Compila las clases. -->
<target name="compile" depends="clean">
  <wtkbuidl srcdir="${src.home}"
    destdir="${build.home}"
    preverify="false"/>
</target>

<!-- Empaqueta la aplicacion. -->
<target name="package" depends="jad,compile">
  <wtkpackage jarfile="${jar.file}"
    jadfile="${jad.file}"
    obfuscate="${obfuscate}"
    preverify="true">
    <fileset dir="${build.home}"/>
    <fileset dir="${res.home}"/>
  </wtkpackage>
</target>

<!-- Ejecuta el MIDlet en un emulador. -->
<target name="run" depends="package">
  <wtkrun jadfile="${jad.file}" heapsize="0"
    device="${emulator.name}" wait="true"/>
</target>

</project>

```

2.4.4. APIs opcionales

Puede ocurrir que una aplicación necesite APIs adicionales para compilarse, como puede ser MMAPAPI, WMA u otras APIs opcionales que podamos encontrar en algunos modelos de teléfonos móviles. Es este caso deberemos especificar de forma explícita el *classpath* que vamos a utilizar para la compilación (*wtkbuidl*) y el empaquetamiento (*wtkpackage*). Para ello deberemos añadir dentro de cada una de estas tareas el siguiente atributo:

```
bootclasspath="<classpath>"
```

Por ejemplo, si queremos utilizar la API multimedia MMAPAPI, podemos hacerlo de la siguiente forma:

```

<!-- Compila las clases. -->
<target name="compile" depends="clean">
  <wtkbuidl srcdir="${src.home}"
    destdir="${build.home}"

bootclasspath="${wtk.home}/lib/midpapi.zip;${wtk.home}/lib/mma
pi.jar"
    preverify="false">
  </wtkbuidl>
</target>

```



```
<!-- Empaqueta la aplicacion. -->
<target name="package" depends="jad,compile">
  <wtkpackage jarfile="${jar.file}"
    jadfile="${jad.file}"
    obfuscate="${obfuscate}"

  bootclasspath="${wtk.home}/lib/midpapi.zip;${wtk.home}/lib/mmapi.jar"
    preverify="true">
    <fileset dir="${build.home}"/>
    <fileset dir="${res.home}"/>
  </wtkpackage>
</target>
```

2.5. Entornos de Desarrollo Integrados (IDEs)

Hemos visto que los kits de desarrollo como WTK nos permiten construir la aplicación pero no tienen ningún editor integrado donde podamos escribir el código. Por lo tanto tendríamos que escribir el código fuente utilizando cualquier editor de texto externo, y una vez escrito utilizar WTK para construir la aplicación.

Vamos a ver ahora como facilitar el desarrollo de la aplicación utilizando distintos entornos integrados de desarrollo (IDEs) que integran un editor de código con las herramientas de desarrollo de aplicaciones MIDP. Estos editores además nos facilitarán la escritura del código coloreando la sintaxis, revisando la corrección del código escrito, autocompletando los nombres, formateando el código, etc.

Para desarrollar aplicaciones J2ME podremos utilizar la mayoría de los IDEs existentes para Java, añadiendo alguna extensión para permitirnos trabajar con este tipo de aplicaciones. También podemos encontrar entornos dedicados exclusivamente a la creación de aplicaciones J2ME.

Vamos a centrarnos en dos entornos que tienen la ventaja de ser de libre distribución, y que son utilizados por una gran cantidad de usuarios dadas sus buenas prestaciones. Luego comentaremos más brevemente otros entornos disponibles para trabajar con aplicaciones J2ME.

2.5.1. Eclipse

Eclipse es un entorno de desarrollo de libre distribución altamente modular. Una de sus ventajas es que no necesita demasiados recursos para ejecutarse correctamente, por lo que será adecuado para máquinas poco potentes. Vamos a utilizar como referencia la versión 2.1.1 de este entorno. Algunas características pueden variar si se utiliza una versión distinta.

Este entorno nos permite crear proyectos en Java. Nos ofrece un editor, en el que podemos escribir el código, viendo la sintaxis coloreada para mayor claridad, y notificándonos de los errores que hayamos cometido al escribir el código, como por ejemplo haber escrito mal el nombre de un método, o usar un

tipo o número incorrecto de parámetros. Además nos permitirá autocompletar los nombres de los métodos o las propiedades de las clases conforme los escribimos. Si el código ha quedado desordenado, nos permite darle formato automáticamente, poniendo la sangría adecuada para cada línea de código.

Esto nos facilitará bastante la escritura del código fuente. Sin embargo, no nos permitirá crear visualmente la GUI de las aplicaciones, ni el diseño, ni manejará conexiones con BDs ni con servidores de aplicaciones. Esto hace que el entorno sea bastante más ligero que otros entornos, por lo que será más cómodo de manejar si no necesitamos todas estas características. Incluso podemos añadirle muchas de estas funcionalidades que se echan en falta añadiendo módulos (*plugins*) al entorno.

Podremos compilar las clases del proyecto desde el mismo entorno, y ejecutar la aplicación para probarla utilizando la máquina virtual de Java. Esto será suficiente para aplicaciones J2SE, pero en principio no ofrece soporte directo para J2ME. Podemos optar por diferentes soluciones para crear aplicaciones J2ME con este entorno:

- **Editor de código.** Lo que podemos hacer es utilizarlo únicamente como editor de código, ya que es un editor bastante cómodo y rápido, y utilizar de forma externa WTK para compilar y ejecutar la aplicación.
- **Integración con Antenna.** Dado que el entorno viene integrado con la herramienta *ant*, podemos utilizar *Antenna* para compilar y ejecutar las aplicaciones desde el mismo entorno. Esta solución es bastante versátil, ya que desde el fichero de *ant* podemos personalizar la forma en la que se realizarán los distintos pasos del proceso. El inconveniente es que es más complicado escribir el fichero de *ant* que usar un entorno que realiza ese proceso automáticamente, y requerirá que los usuarios conozcan dicha herramienta.
- **EclipseME.** Otra solución es utilizar un *plugin* que nos permita desarrollar aplicaciones J2ME desde Eclipse. Tenemos disponible el plugin **EclipseME** que realizará esta tarea.

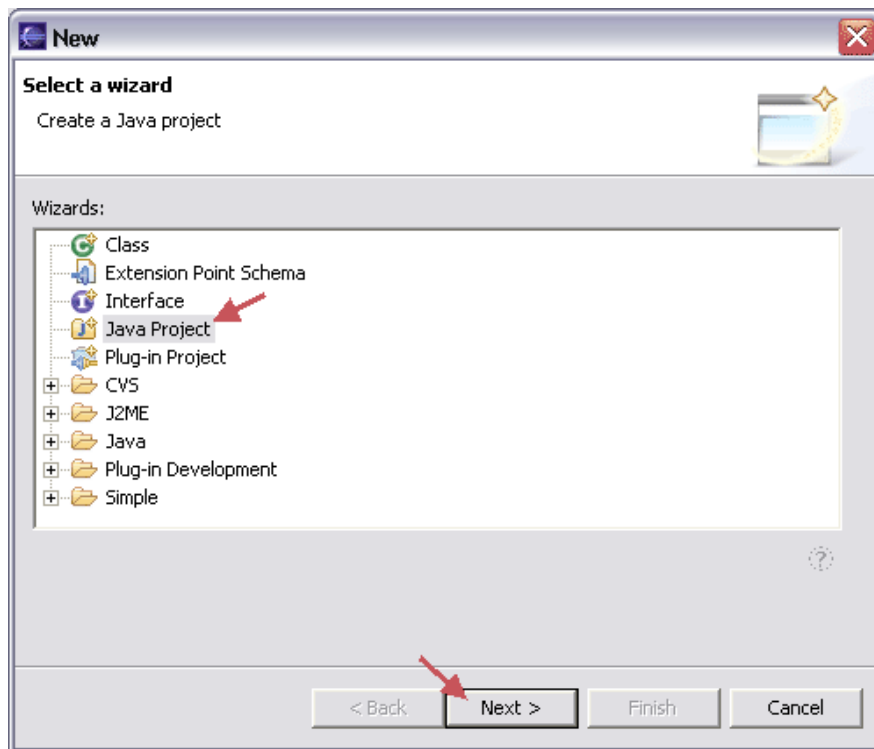
A continuación veremos como crear aplicaciones J2ME paso a paso siguiendo cada uno de estos tres métodos.

Editor de código

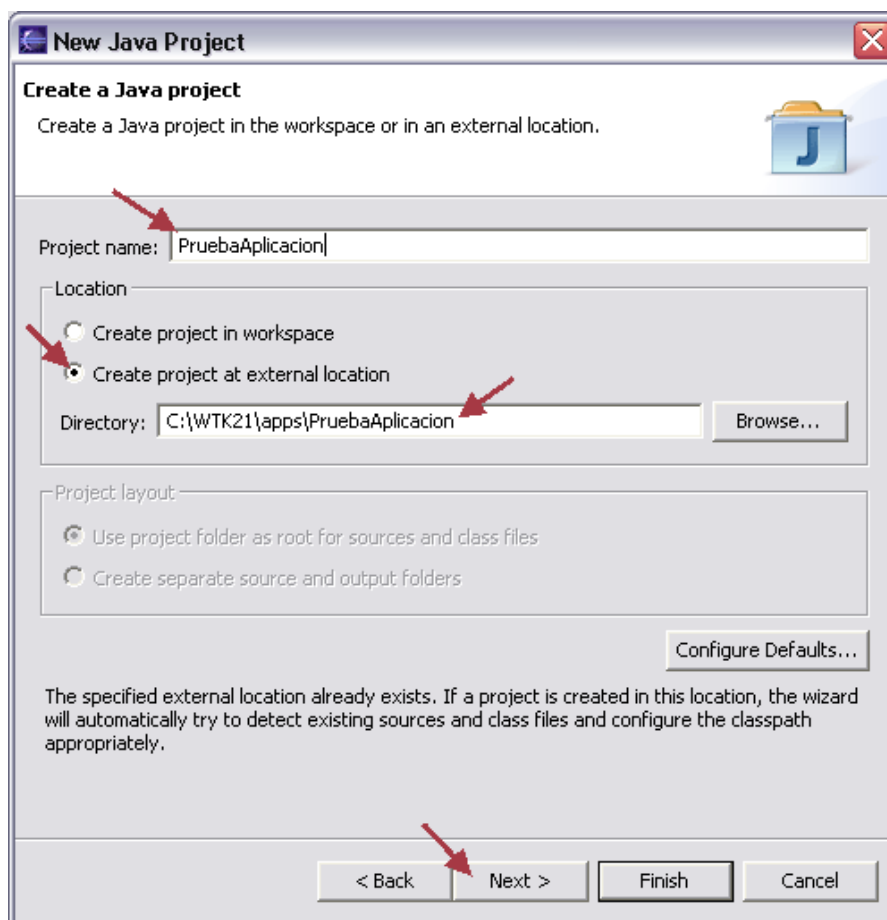
Vamos a ver como utilizar Eclipse simplemente para editar el código de las aplicaciones J2ME, dejando las tareas de compilación, empaquetamiento y ejecución para realizarlas de forma externa con WTK.

Lo primero que tenemos que hacer es crear una nueva aplicación utilizando WTK, como hemos visto en el punto anterior, de forma que nos cree la estructura de directorios necesaria en nuestro directorio de proyectos.

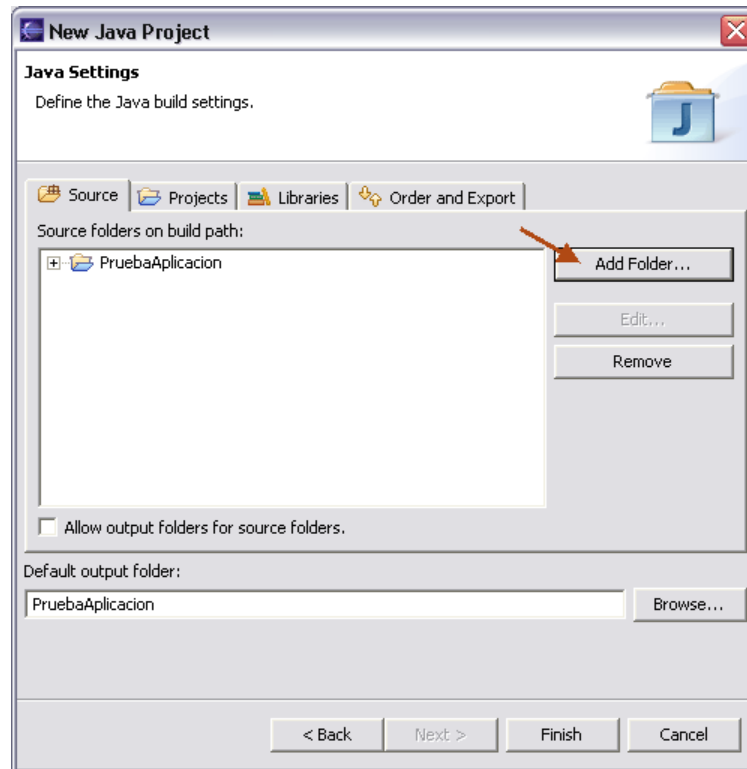
Una vez hecho esto, podemos entrar ya en Eclipse para comenzar a escribir código. Crearemos un nuevo proyecto Java, utilizando el comando **New**:



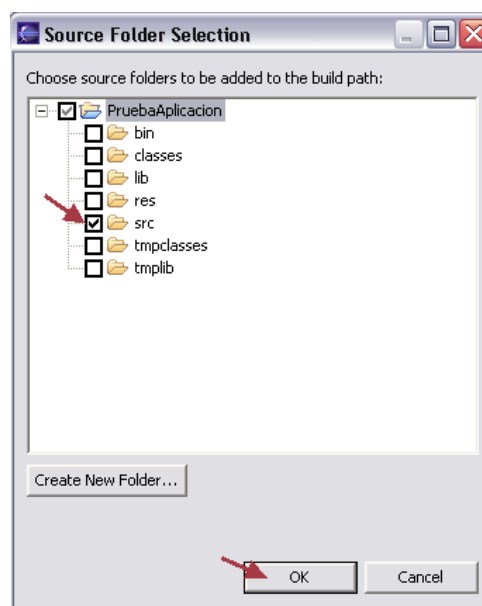
Elegimos **Java Project** y pulsamos **Next** para comenzar el asistente de creación del proyecto:



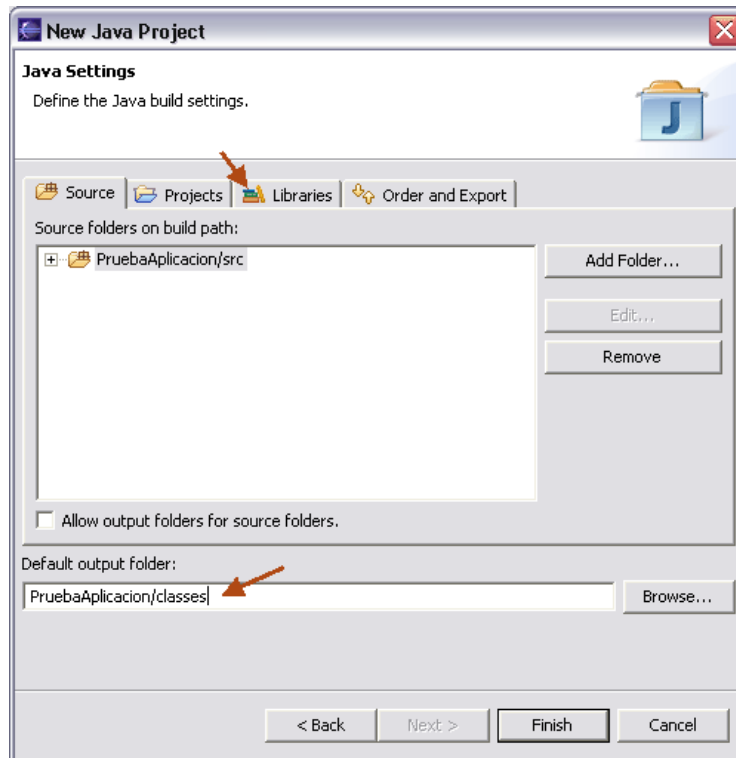
Debemos darle un nombre al proyecto, y un directorio donde guardar su contenido. Elegiremos la opción **Create project at external location** para poder elegir como directorio del proyecto el directorio que queramos, y seleccionaremos como tal el directorio que hemos creado previamente con WTK. Pulsamos **Next** para continuar con el asistente.



Debemos especificar los directorios donde guardar los fuentes del proyecto, y donde se guardarán las clases compiladas. Pulsamos sobre **Add Folder ...** para seleccionar el directorio donde se encontrarán los fuentes de nuestro proyecto.

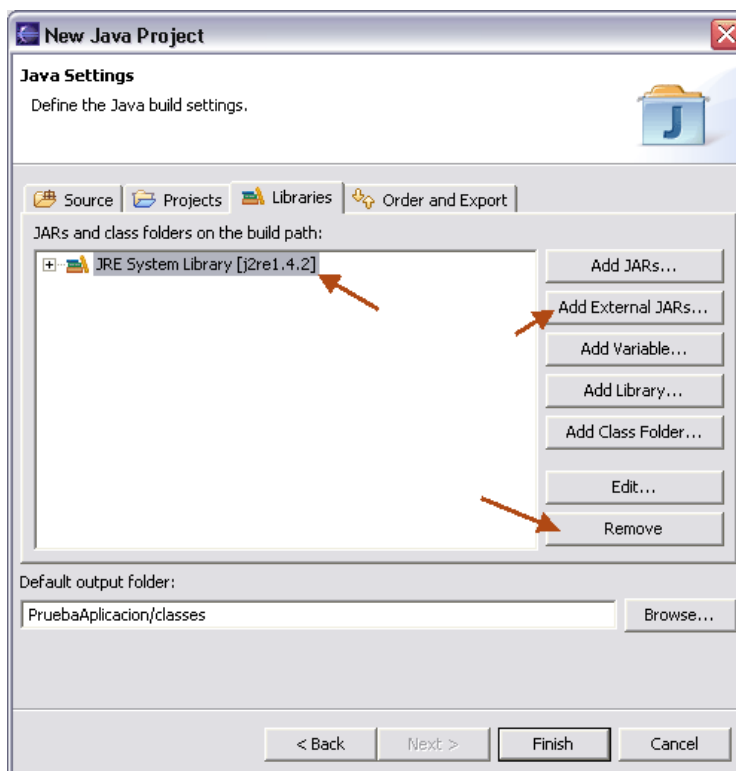


Como directorio de fuentes seleccionaremos el subdirectorio `src` del directorio de nuestro proyecto, si no estuviese seleccionado ya.

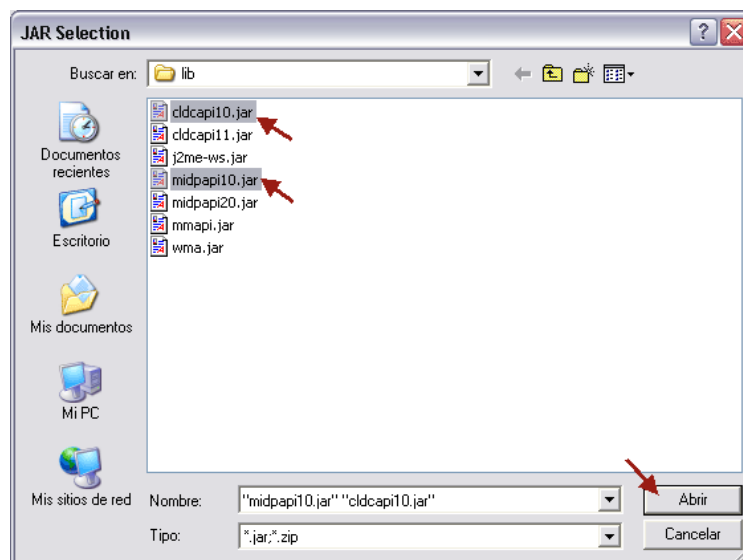


Como directorio de salida (**Default output folder**) podemos especificar el directorio `classes`.

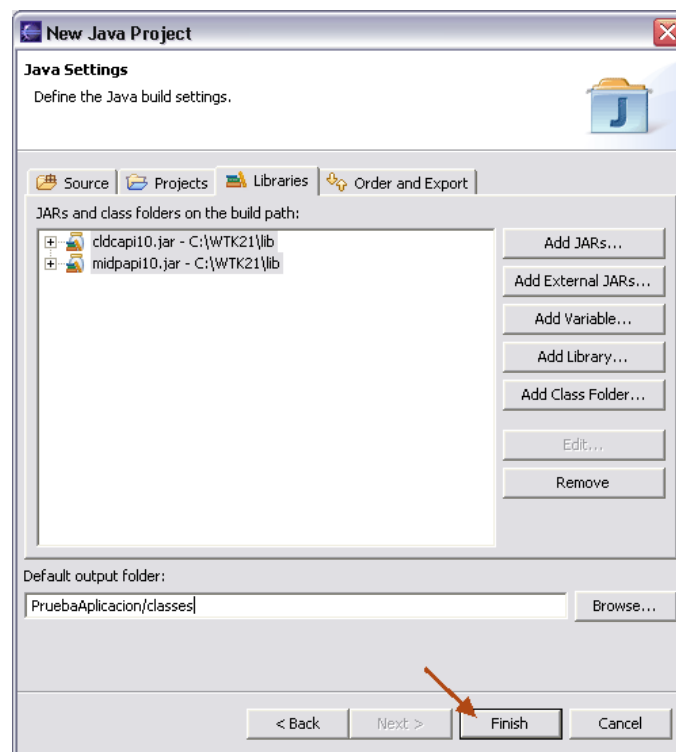
Ahora pasamos a la pestaña **Libraries** de esta misma ventana.



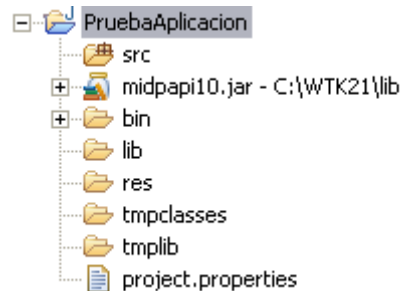
En ella por defecto tendremos las librerías de J2SE. Como no queremos que se utilicen estas librerías en nuestra aplicación, las eliminaremos de la lista con **Remove**, y añadiremos la librería de la API MIDP. Para ello pulsaremos el botón **Add External JARs** y seleccionaremos el JAR de MIDP, ubicado normalmente en el directorio `${WTK_HOME}/lib/midpapi.zip`. Si quisiéramos utilizar otras APIs en nuestra aplicación, como MMAPAPI o APIs propietarias, seguiremos el mismo proceso para añadir sus correspondientes ficheros JAR a la lista.



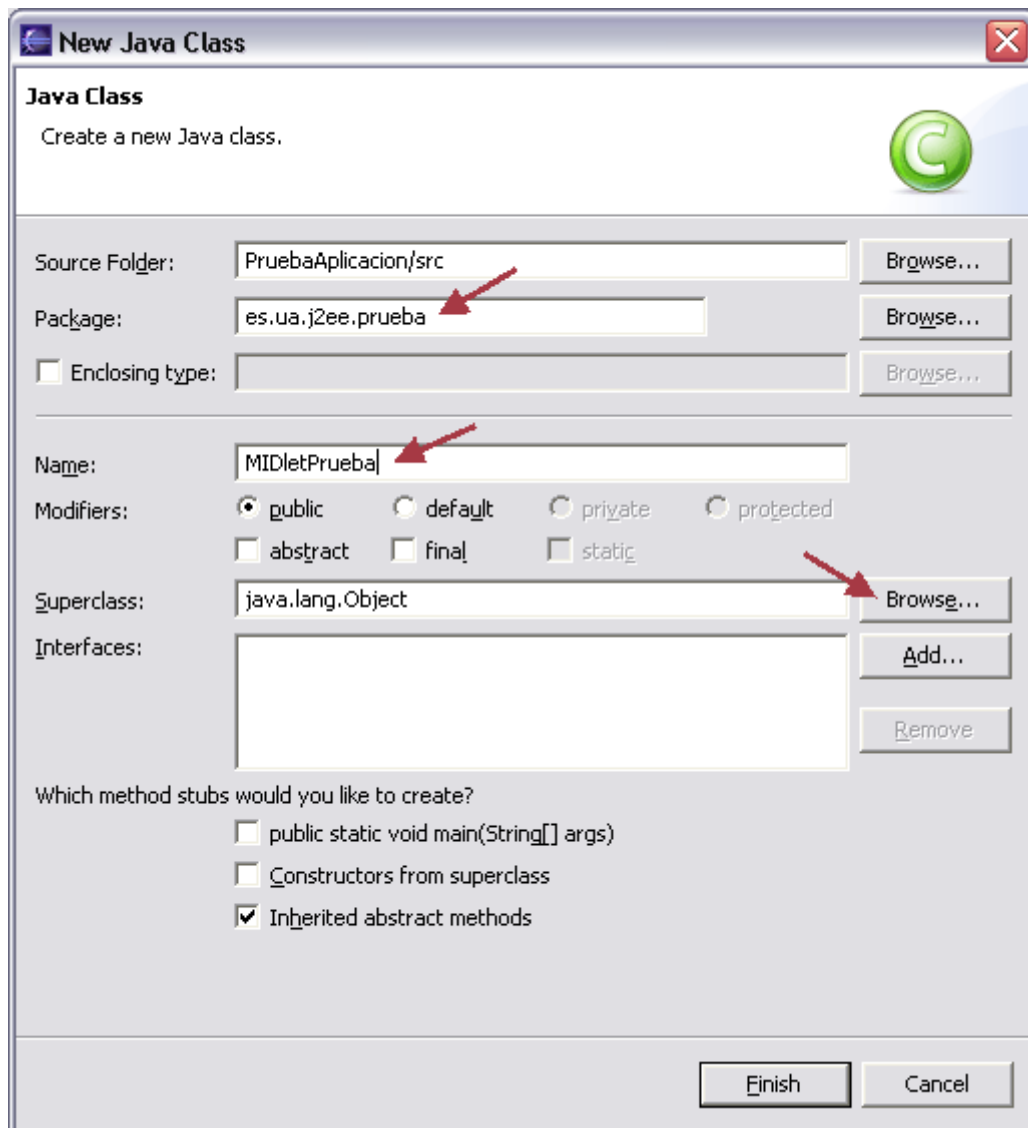
Como no vamos a utilizar Eclipse para compilar, estas librerías nos servirán simplemente para que Eclipse pueda autocompletar el código que escribamos y detectar errores.



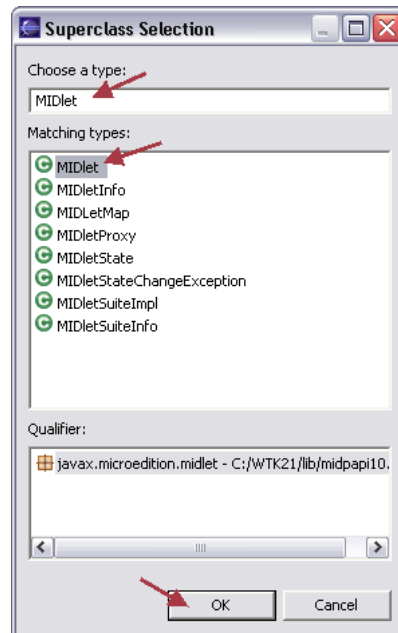
Una vez hemos terminado pulsaremos **Finish** con lo que terminaremos de configurar el proyecto en Eclipse. Una vez hecho esto, en la ventana del explorador de paquete de Eclipse (**Package Explorer**) veremos nuestro proyecto ya creado:



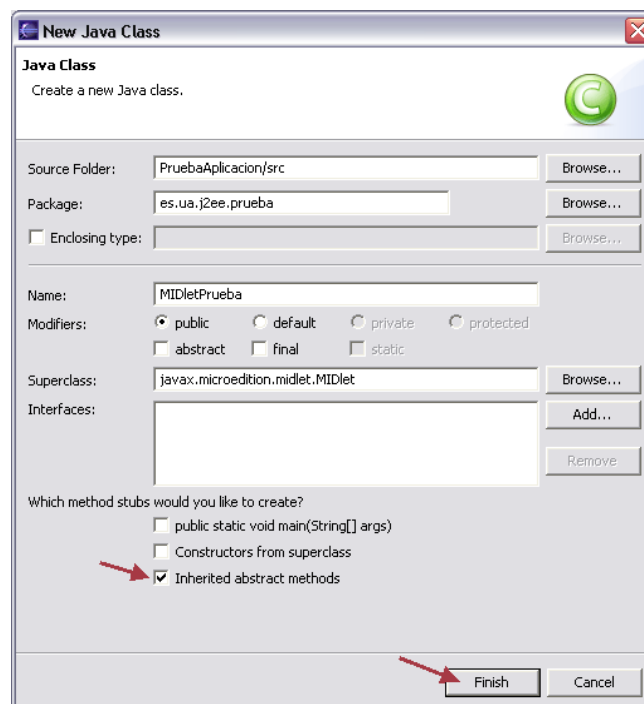
Ahora podemos empezar a crear las clases de nuestra aplicación. Para ello pulsaremos sobre **New** y elegiremos crear una nueva clase Java, con lo que se abrirá la siguiente ventana de creación de la clase:



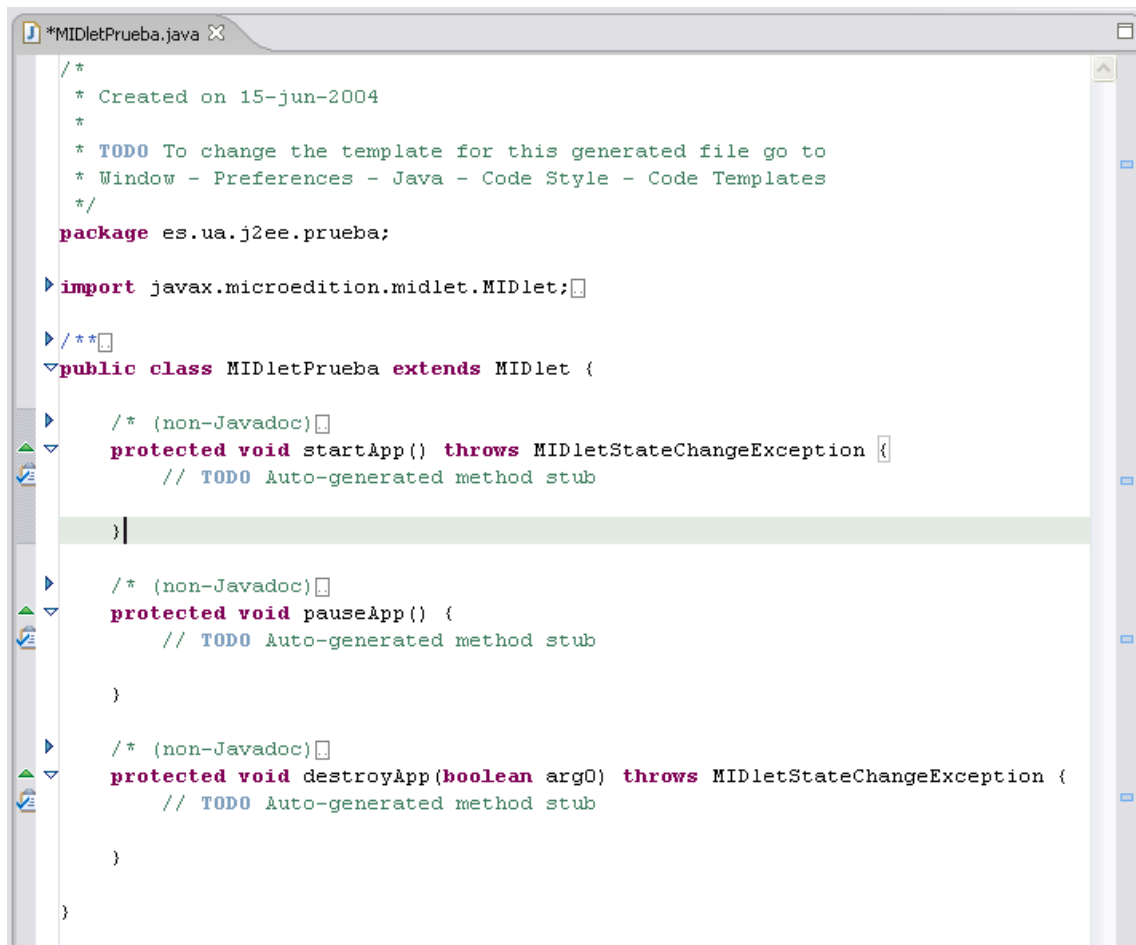
Aquí deberemos introducir el paquete al que pertenecerá la clase, y el nombre de la misma. También debemos indicar la superclase y las interfaces que implementa la clase que vayamos a crear. En caso de querer crear un MIDlet, utilizaremos como superclase la clase `MIDlet`. Para añadir una superclase podemos pulsar sobre el botón **Browse...**, de forma que nos mostrará la siguiente ventana desde la que podremos buscar clases de las que heredar:



En el campo de texto superior, donde nos dice **Choose a type**, podremos empezar a escribir el nombre de la clase de la que queramos heredar, y el explorador nos mostrará todas las clases cuyo nombre coincida total o parcialmente con el texto escrito. Seleccionaremos la clase deseada y pulsamos **OK**.



Si dejamos marcada la casilla **Inherited abstract methods**, nos creará el esqueleto de la clase con los métodos definidos como abstractos en la superclase que debemos rellenar como vemos a continuación:



```
/*
 * Created on 15-jun-2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package es.ua.j2ee.prueba;

import javax.microedition.midlet.MIDlet;

/**
 *
 */
public class MIDletPrueba extends MIDlet {

    /** (non-Javadoc)
     * @throws MIDletStateChangeException
     * // TODO Auto-generated method stub
     */
    protected void startApp() throws MIDletStateChangeException {

    }

    /** (non-Javadoc)
     * @throws MIDletStateChangeException
     * // TODO Auto-generated method stub
     */
    protected void pauseApp() {

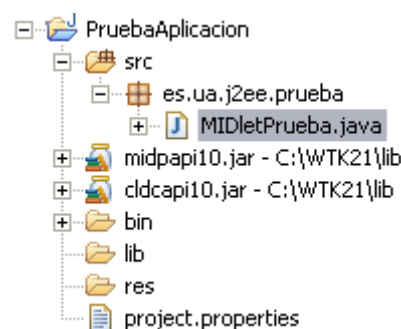
    }

    /** (non-Javadoc)
     * @throws MIDletStateChangeException
     * // TODO Auto-generated method stub
     */
    protected void destroyApp(boolean arg0) throws MIDletStateChangeException {

    }

}
```

Aquí podremos introducir el código necesario en los métodos que nos ha creado. Junto al editor de código en el explorador de paquetes veremos la clase que acabamos de crear:



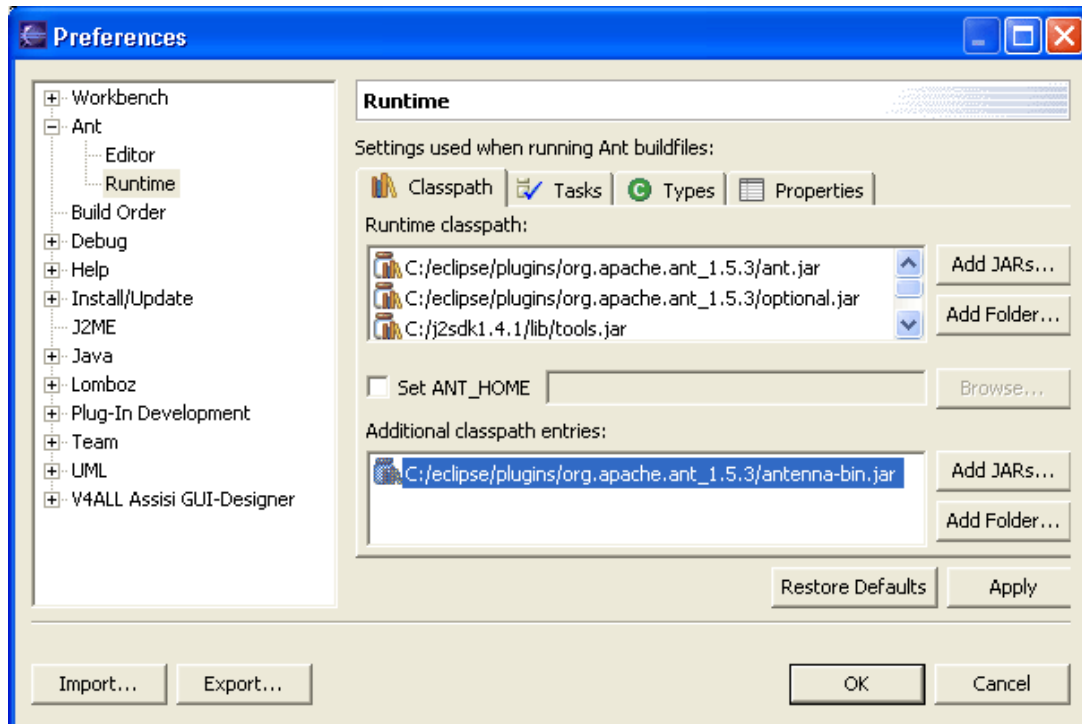
Aquí podemos ver la estructura de directorios de nuestro proyecto, los paquetes y las clases de nuestra aplicación, y las librerías utilizadas.

Una vez hayamos creado todas las clases necesarias desde Eclipse, y hayamos escrito el código fuente, deberemos volver a WTK para compilar y ejecutar nuestra aplicación.

Integración con Antenna

Para no tener que utilizar dos herramientas por separado (WTK y Eclipse), podemos aprovechar la integración de *ant* con Eclipse para compilar y ejecutar las aplicaciones J2ME utilizando las tareas de *Antenna*.

Para poder utilizar estas tareas deberemos configurar *Antenna* dentro de Eclipse, para lo cual debemos ir a **Window > Preferences**, y dentro de la ventana de preferencias seleccionar las preferencias de *Runtime* de Ant:



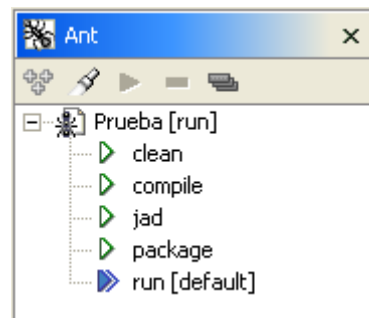
Aquí deberemos añadir como entrada adicional de *classpath* el fichero JAR de *Antenna*. Una vez hecho esto podremos utilizar las tareas de *Antenna* dentro de los ficheros de *ant*.

NOTA: Para que *ant* funcione correctamente desde dentro de Eclipse es necesario añadir al classpath de *ant* (**Runtime classpath** de la ventana anterior) la librería `tools.jar` que podemos encontrar dentro del directorio `${JAVA_HOME}/lib`.

Ahora tenemos que crear el fichero de *ant*. Para ello seleccionamos **New > File**, para crear un fichero genérico. Llamaremos al fichero `build.xml`, y escribiremos en él todas las tareas necesarias para compilar y ejecutar la aplicación, como vimos en el punto de *Antenna*. Una vez escrito este fichero lo grabaremos.

Ahora debemos ir al panel de **Ant** dentro de Eclipse. Si no tenemos este panel iremos a **Window > Show view** para mostrarlo. Dentro de este panel pulsaremos sobre el botón para añadir un *buildfile*, seleccionando el fichero que acabamos de crear, y una vez añadido veremos en ese panel la lista de los

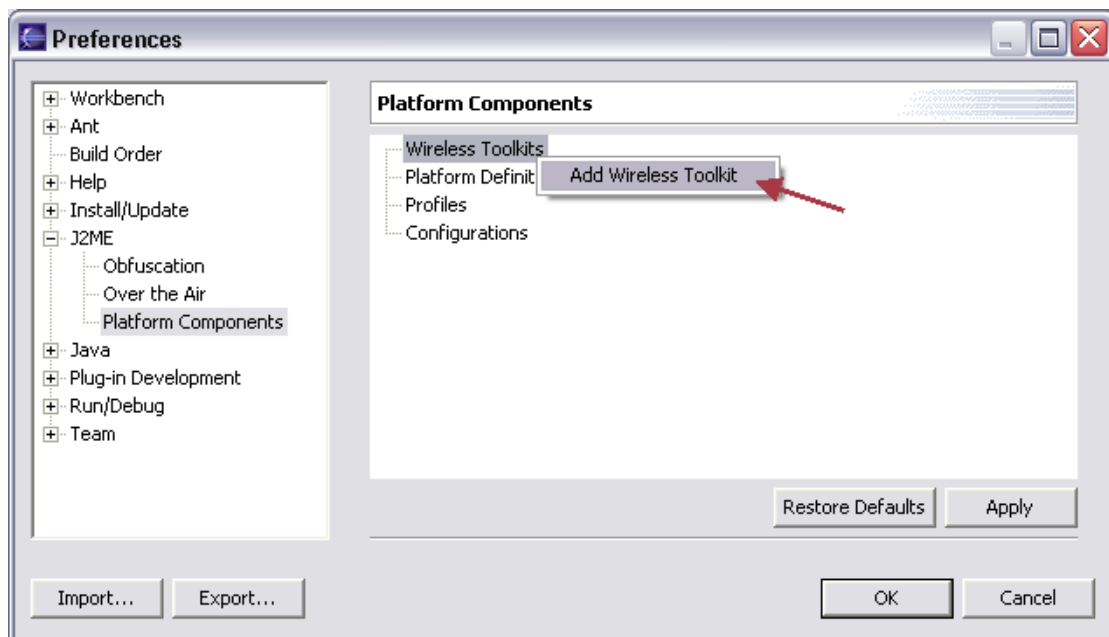
posibles objetivos que hay definidos en el fichero. Podremos desde este mismo panel ejecutar cualquiera de los objetivos, pudiendo de esta forma compilar y ejecutar la aplicación directamente desde el entorno.



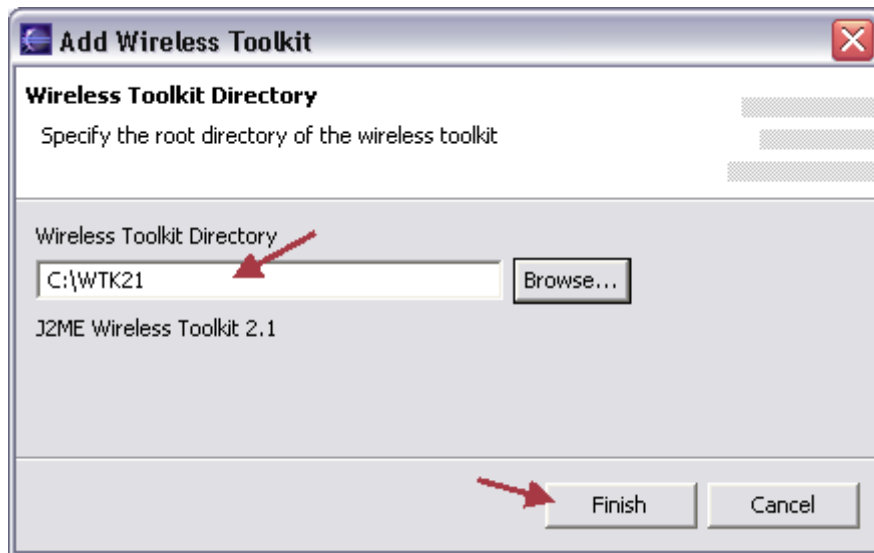
EclipseME

EclipseME es un *plugin* realizado por terceros que nos facilitará la creación de aplicaciones J2ME desde Eclipse.

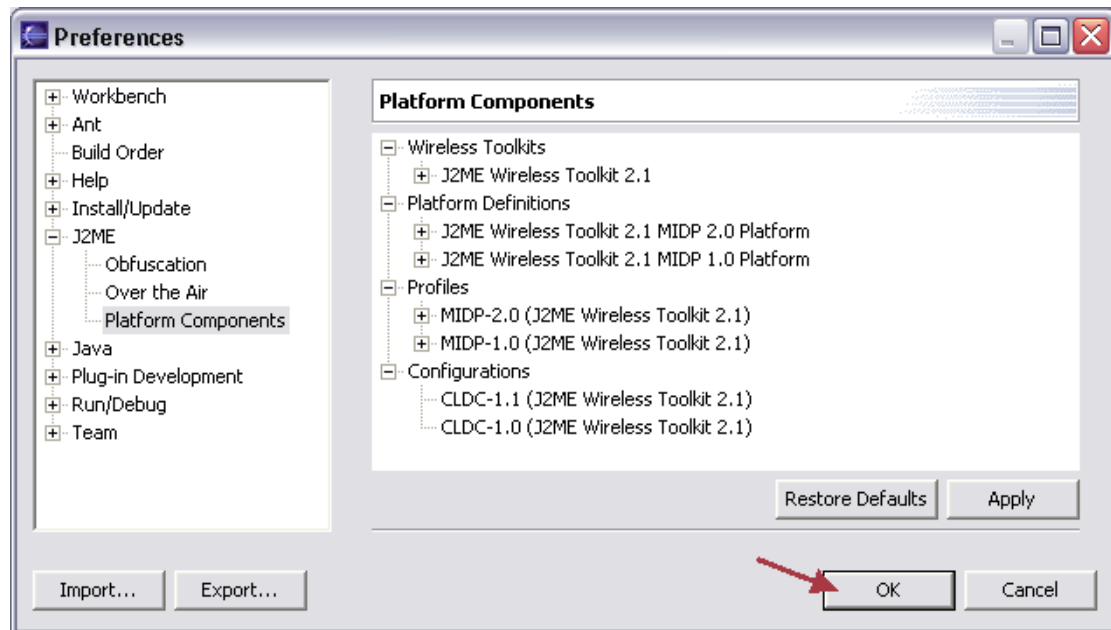
Lo primero que debemos hacer es instalar el *plugin*, descomprimiéndolo en el directorio `${ECLIPSE_HOME}/plugins`. Una vez hecho esto, deberemos reiniciar el entorno, y entonces deberemos ir a **Window > Preferences** para configurar el *plugin*:



En el apartado de configuración de J2ME, dentro del subapartado **Platform Components**, deberemos especificar el directorio donde tenemos instalado WTK. Para ello pulsamos con el botón derecho del ratón sobre **Wireless Toolkits** y seleccionamos la opción **Add Wireless Toolkit**. Nos mostrará la siguiente ventana, en la que deberemos seleccionar el directorio donde se encuentra WTK:

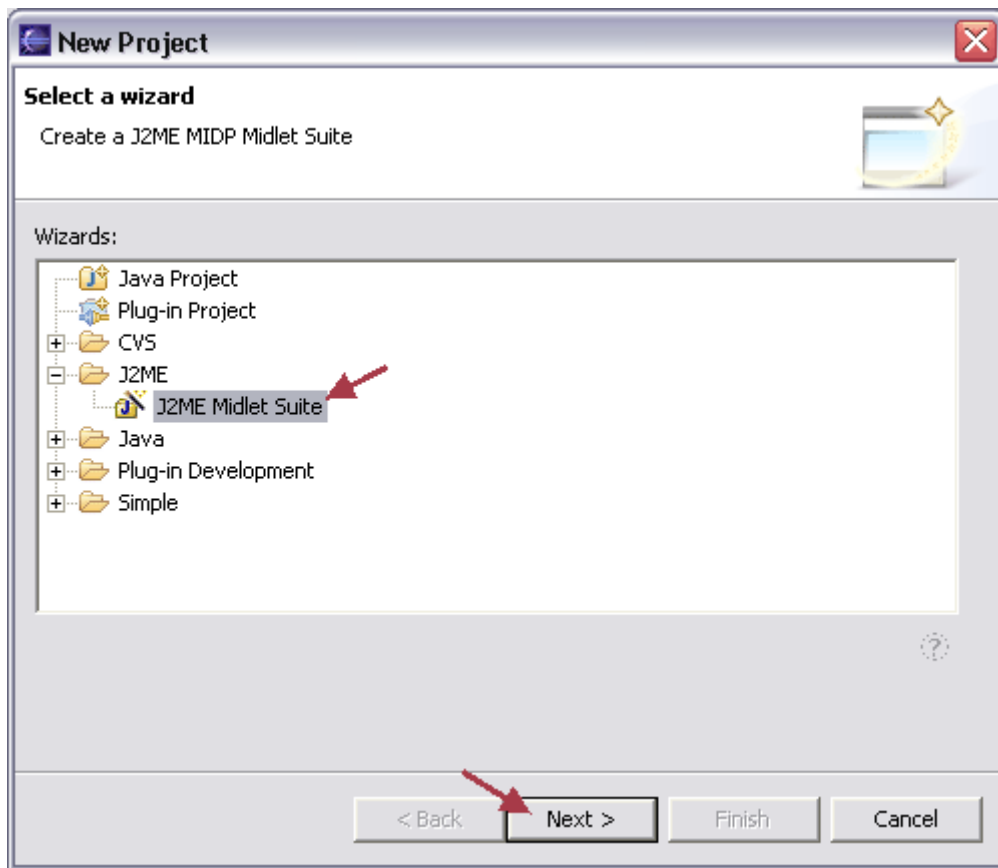


Una vez añadido un *toolkit*, se mostrarán los componentes añadidos en la ventana de configuración:

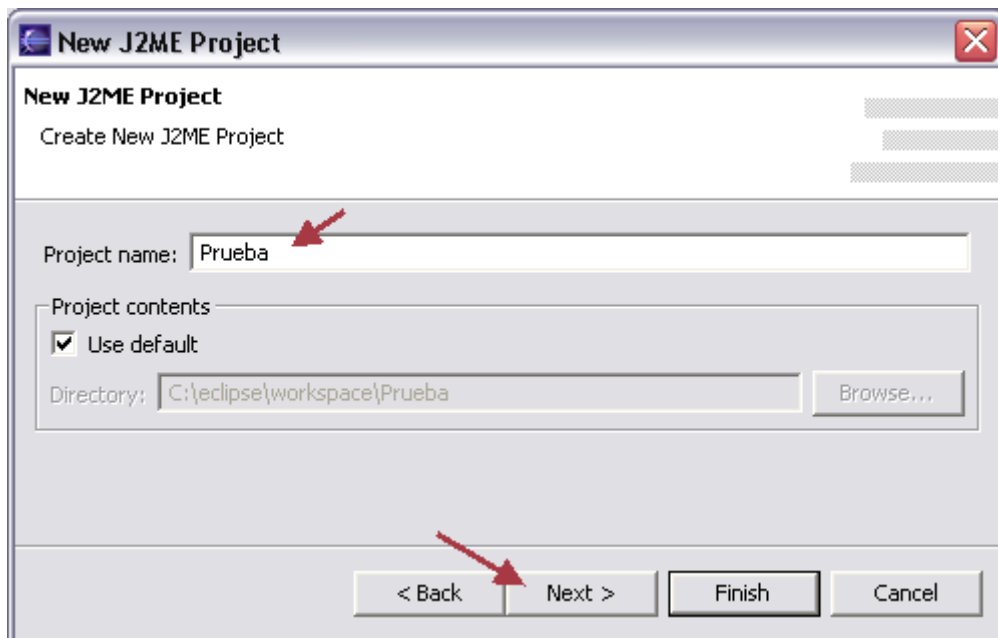


De esta forma vemos que al añadir WTK 2.1 hemos añadido soporte para los perfiles MIDP 1.0 y 2.0, y para las configuraciones CLDC 1.0 y 1.1. Podremos configurar varios *toolkits*. Por ejemplo, podemos tener configuradas las distintas versiones de WTK (1.0, 2.0, 2.1 y 2.2) para utilizar la que convenga en cada momento. Una vez hayamos terminado de configurar los *toolkits*, pulsaremos **OK** para cerrar esta ventana.

Una vez configurado, podremos pulsar sobre **New**, donde encontraremos disponibles asistentes para crear aplicaciones J2ME:

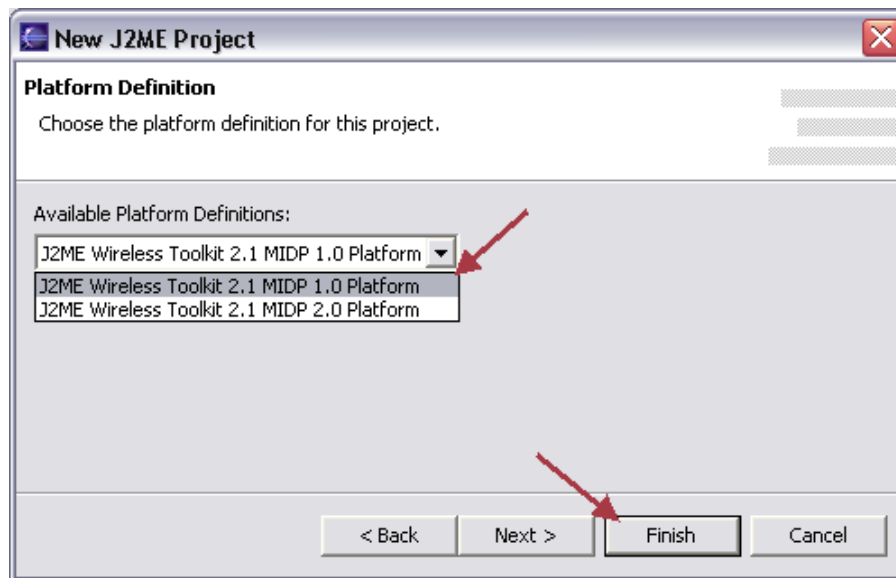


Lo primero que haremos será crear la *suite* (proyecto). Seleccionamos **J2ME Midlet Suite** y pulsamos **Next** para comenzar con el asistente de creación de la *suite* J2ME:



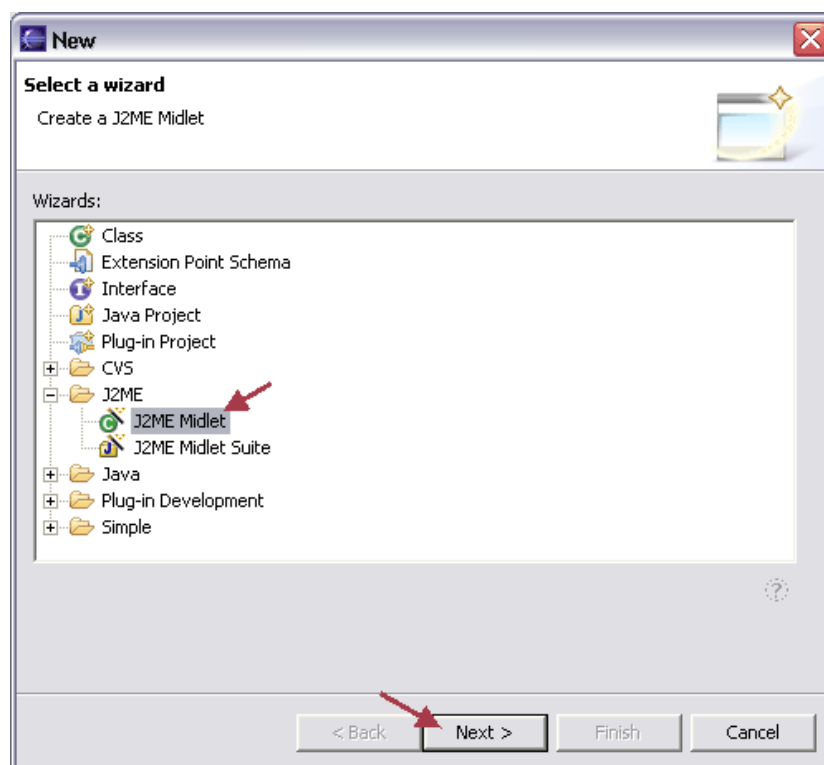
Deberemos darle un nombre al proyecto que estamos creando. En este caso podemos utilizar el directorio por defecto, ya que no vamos a utilizar WTK para

construir la aplicación, la construiremos directamente desde Eclipse. Una vez asignado el nombre pulsamos sobre **Next** para ir a la siguiente ventana:

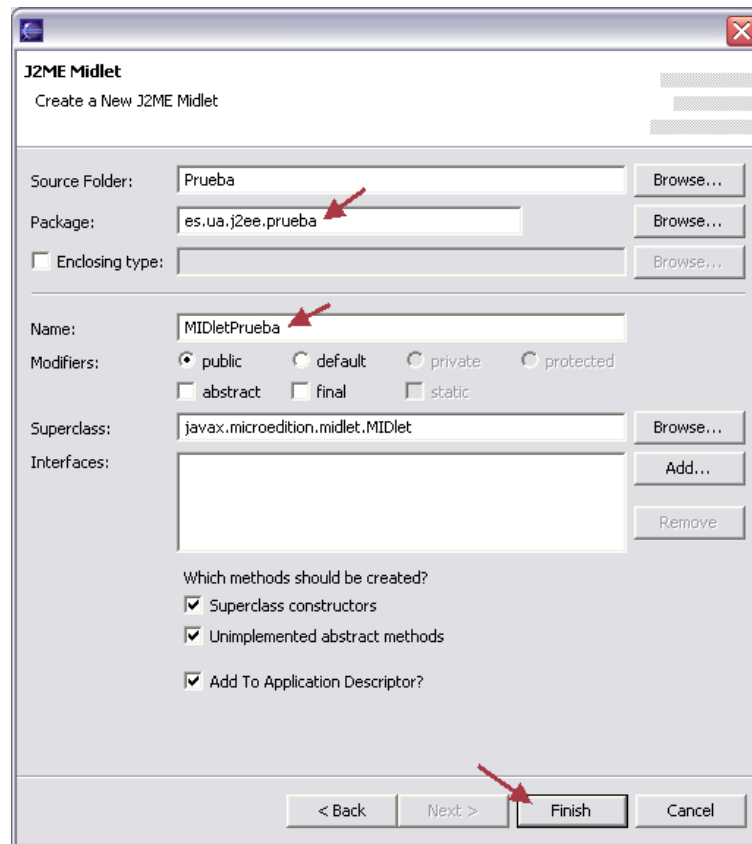


Aquí podemos elegir la versión de MIDP para la que queremos programar, siempre que tengamos instalado el WTK correspondiente para cada una de ellas. Una vez elegida la versión para la que queremos desarrollar pulsamos **Finish**, con lo que habremos terminado de configurar nuestro proyecto. En este caso no hace falta que especifiquemos las librerías de forma manual, ya que el asistente las habrá configurado de forma automática.

Una vez creado el proyecto, podremos añadir MIDlets u otras clases Java. Pulsando sobre **New** veremos los elementos que podemos añadir a la *suite*:

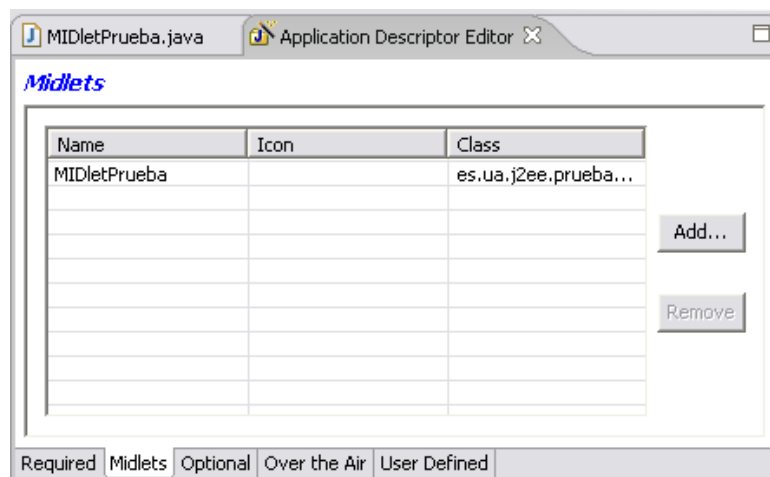


Si queremos crear un MIDlet, podremos utilizar la opción **J2ME Midlet** y pulsar **Next**, con lo que se mostrará la siguiente ventana para introducir los datos del MIDlet:



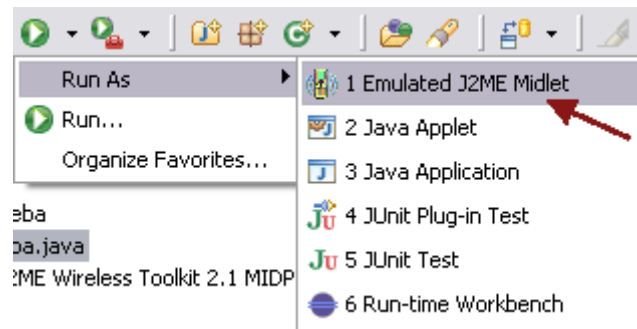
Aquí deberemos dar el nombre del paquete y el nombre de la clase de nuestro MIDlet. Pulsando sobre **Finish** creará el esqueleto de la clase correspondiente al MIDlet, donde nosotros podremos insertar el código necesario.

En el explorador de paquetes podemos ver las clases creadas, la librería utilizada y el fichero JAD del proyecto. Pinchando sobre el fichero JAD se mostrará en el editor la siguiente ficha con los datos de la *suíte*:

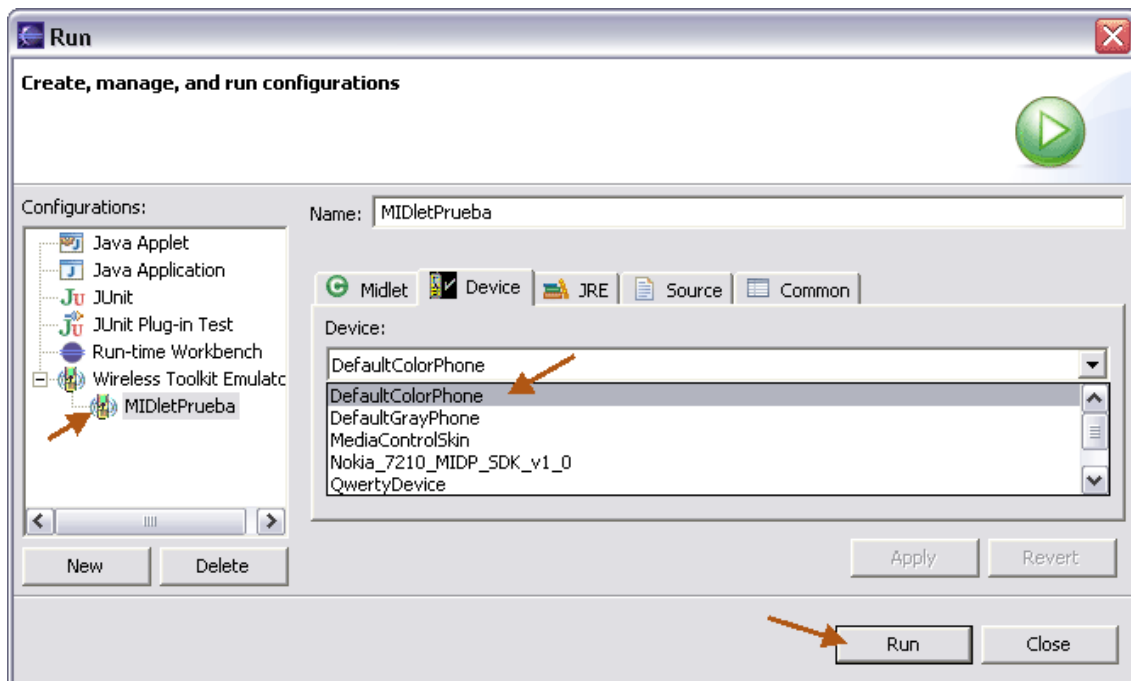


Aquí deberemos introducir la información necesaria, sobre los datos de la *suite* (**Required**) y los MIDlets que hayamos creado en ella (en la pestaña **Midlets**). Podemos ver que, cuando creemos un MIDlet mediante el asistente que acabamos de utilizar para la creación de MIDlets, los datos del MIDlet creado se añaden automáticamente al JAD.

No es necesario compilar el proyecto manualmente, ya que Eclipse se ocupará de ello. Cuando queramos ejecutarlo, podemos seleccionar en el explorador de paquetes el MIDlet que queramos probar y pulsar sobre el botón **Run > Emulated J2ME Midlet**:



Esto abrirá nuestro MIDlet en el emulador que se haya establecido como emulador por defecto del *toolkit* utilizado. Si queremos tener un mayor control sobre cómo se ejecuta nuestra aplicación, podemos utilizar la opción **Run...** que nos mostrará la siguiente ventana:



En esta ventana pulsaremos sobre **Wireless Toolkit Emulator** y sobre **New** para crear una nueva configuración de ejecución sobre los emuladores de J2ME. Dentro de esta configuración podremos seleccionar el emulador dentro de la pestaña **Device**, y una vez seleccionado ya podremos pulsar sobre **Run** para ejecutar la aplicación.

2.5.2. NetBeans

Con Eclipse hemos visto un entorno bastante ligero para la escritura del código. Vamos a ver ahora un entorno más completo también de libre distribución. Se trata de NetBeans, versión de libre distribución del entorno de Sun Forte for Java, también conocido como Sun One Studio.

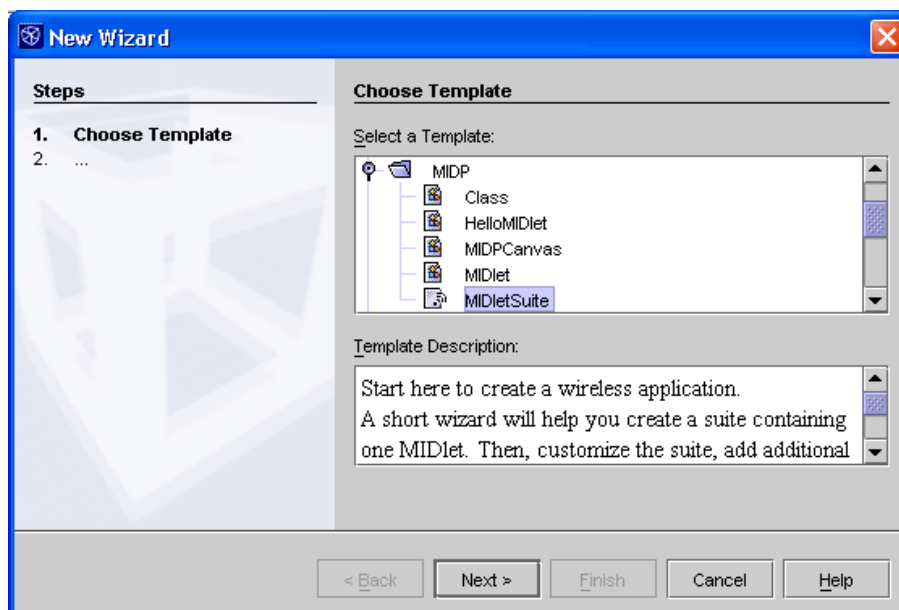
NetBeans además del editor integrado de código, nos permitirá crear la GUI de las aplicaciones de forma visual, crear elementos para aplicaciones J2EE, como *servlets*, JSPs y *beans*, manejar conexiones a BDs, e integra su propio servidor de aplicaciones para poder probar las aplicaciones web entre otras cosas. El contar con todas estas características le hace ser un entorno bastante más pesado que el anterior, que necesitará un mayor número de recursos para ejecutarse correctamente.

Al igual que Eclipse, el editor también nos permite autocompletar el código, colorea la sintaxis para una mayor claridad y detecta algunos fallos conforme vamos escribiendo.

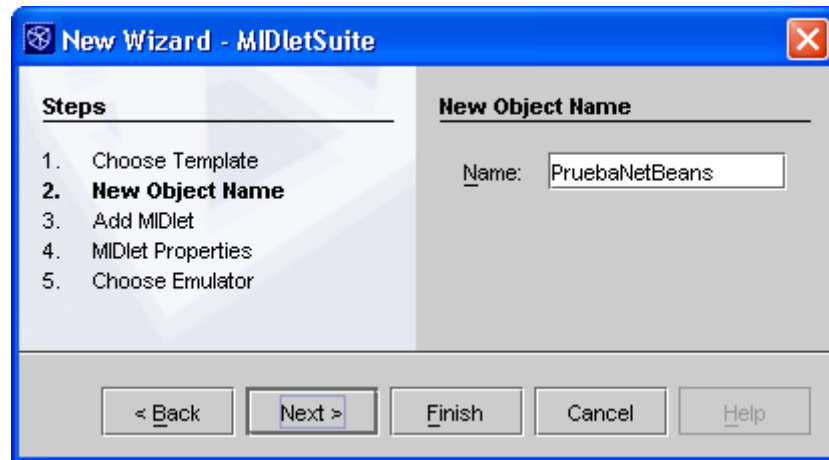
Respecto a las aplicaciones MIDP, podemos encontrar *plugins* oficiales para desarrollar este tipo de aplicaciones desde el entorno. Además, incluirá un depurador (*debugger*) con el que podremos depurar las aplicaciones para móviles, cosa que no podemos hacer simplemente con WTK o con Eclipse.

Tenemos una serie de *plugins* para añadir los asistentes y soporte necesario para los componentes MIDP y para instalar una versión de WTK integrada en el mismo entorno, por lo que no necesitaríamos instalar una versión externa. También disponemos de *plugins* con distintos ofuscadores, que podemos instalar de forma opcional, de forma que podamos ofuscar el código desde el mismo entorno.

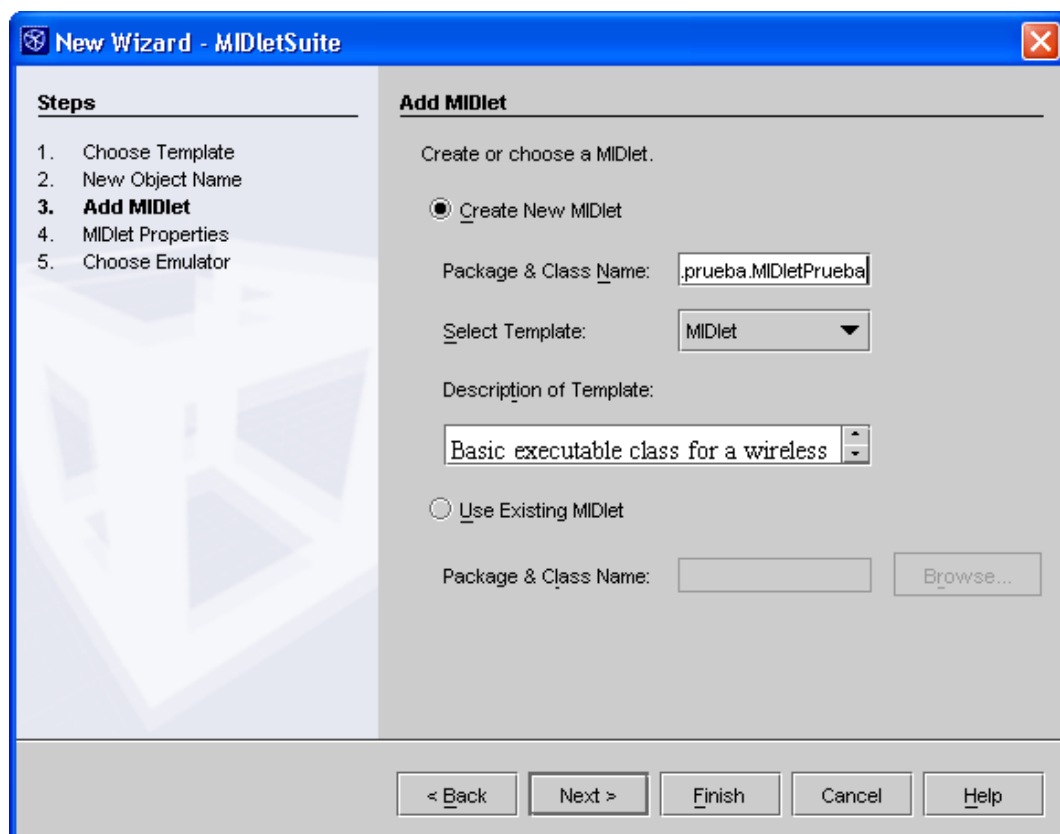
Una vez instalados estos *plugins*, pulsando sobre **New...** podemos crear diferentes elementos para las aplicaciones MIDP:



Vamos a comenzar creando la *suite*. Para ello seleccionamos **MIDletSuite** y pulsamos sobre **Next** para continuar con el asistente de creación de la *suite*:

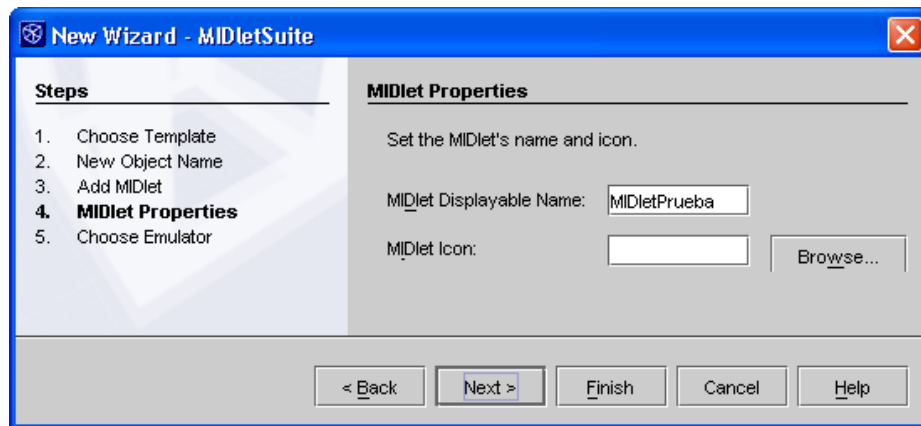


Debemos especificar un nombre para la *suite*. Escribiremos el nombre que queramos y pulsamos sobre **Next** para pasar a la siguiente ficha:



Aquí podremos crear nuestro MIDlet principal para incluir en la *suite* si no tenemos ningún MIDlet creado todavía. Existen diferentes plantillas para MIDlets, que introducen cierta parte del código por nosotros. Podemos seleccionar la plantilla **MIDlet** si queremos que se genere el esqueleto vacío de un MIDlet, o **HelloMIDlet** si queremos que se genere un MIDlet de ejemplo que contenga el código para mostrar el mensaje "*Hola mundo*", cosa que nos puede servir para probar estas aplicaciones sin tener que introducir código

fuentes nosotros. Debemos además darle un nombre al MIDlet que creemos, que debe constar del nombre del paquete y nombre de la clase. Pulsamos sobre **Next** para continuar:



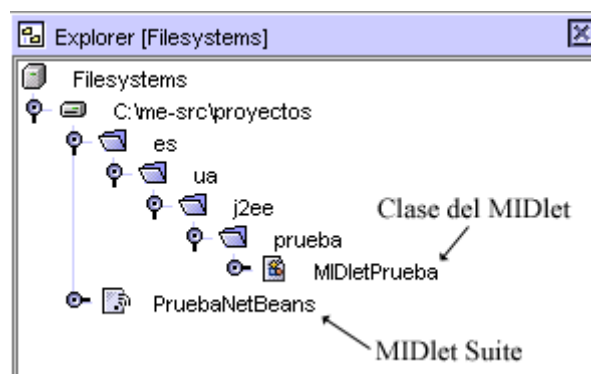
Ahora deberemos introducir el nombre que queremos darle al MIDlet, y de forma opcional el icono con el que se identificará el MIDlet. Una vez hecho esto ya podremos pulsar sobre **Finish** con lo que habremos terminado de crear la *suite*. Podremos ver en el explorador de NetBeans los elementos que se han creado.

Dentro del entorno tenemos tres pestañas como las siguientes:

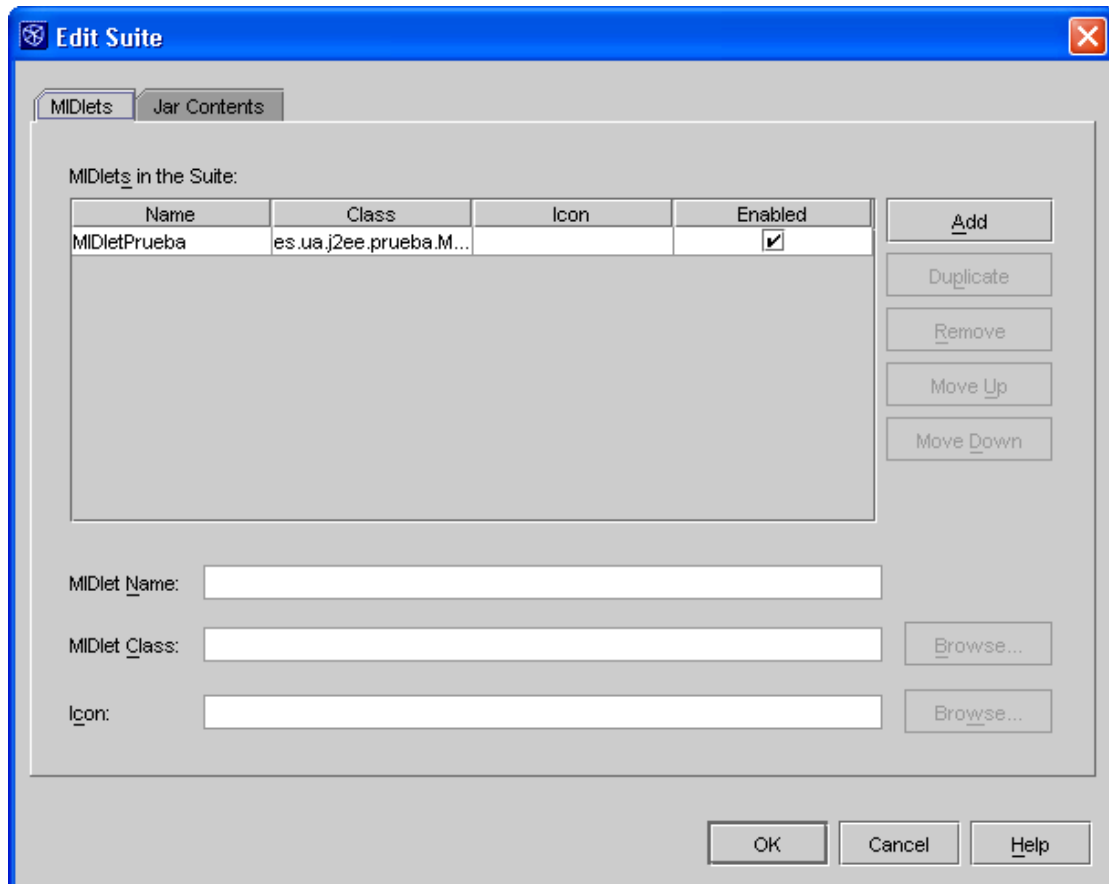


Para editar el código utilizaremos la vista de edición, teniendo seleccionada la primera pestaña (**Editing**). La segunda (**GUI Editing**) nos servirá para crear de forma visual la GUI de las aplicaciones AWT y Swing, por lo que no nos servirá para el desarrollo de aplicaciones J2ME. La tercera (**Debugging**) la utilizaremos cuando estemos depurando el código, tal como veremos más adelante.

Vamos a ver como trabajar en vista de edición para editar y probar nuestra aplicación. En esta vista se mostrará en la parte izquierda de la pantalla el explorador, donde podemos ver los elementos que hemos creado:

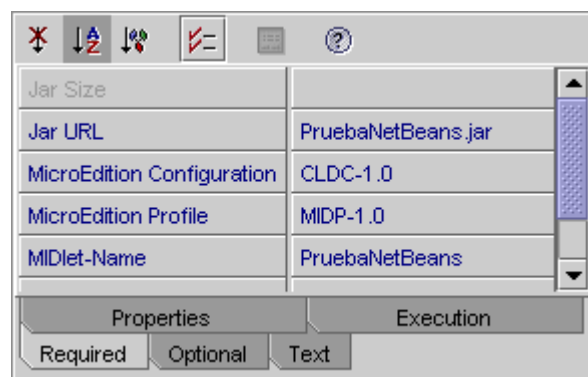


Haciendo doble *click* sobre el elemento correspondiente a la *suite* podremos modificar sus propiedades. Se abrirá la siguiente ventana:



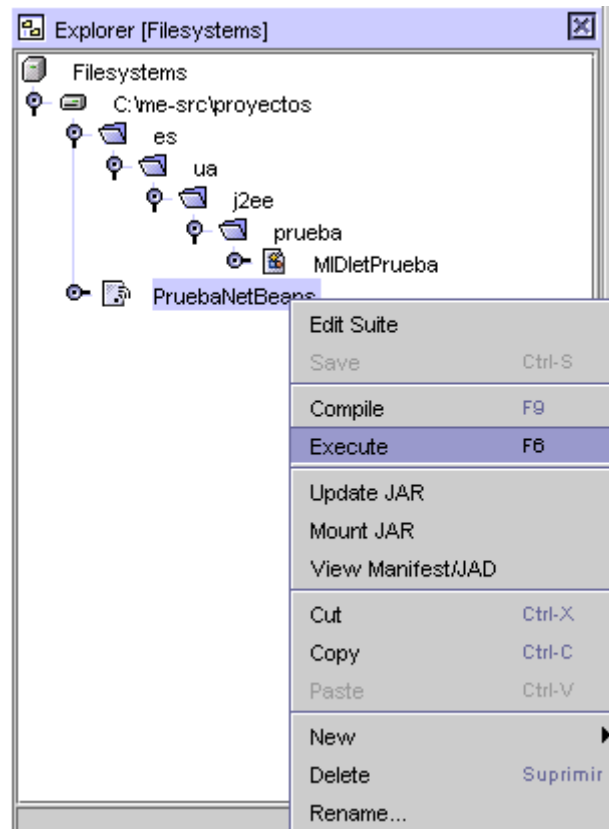
Aquí podremos modificar la lista de MIDlets que vamos a incluir en la *suite*. En la pestaña **Jar Contents** podremos seleccionar todos los elementos que vamos a introducir en el JAR de la *suite*, como recursos, clases y librerías externas.

En la parte inferior el explorador tenemos el inspector de propiedades, donde podemos consultar o modificar las propiedades del elemento seleccionado actualmente en el explorador. Si tenemos seleccionado el elemento correspondientes a la *suite*, veremos las siguientes propiedades:



Aquí podremos modificar distintas propiedades de la *suite*, correspondientes a los datos que se incluirán en los ficheros JAD y `MANIFEST.MF`. Además, en la pestaña **Execution** podremos seleccionar el emulador en el que se va a ejecutar esta *suite* cuando la probemos. Tendremos disponibles los mismos emuladores que contenga el WTK, y podremos especificar la versión de WTK de la que queremos que coja los emuladores.

Para ejecutar la *suite* en el emulador pulsaremos con el botón derecho sobre el elemento correspondiente a dicha *suite* en el explorador, y seleccionamos la opción **Execute**, con lo que la ejecutará en el emulador seleccionado:



Otra forma de ejecutar la *suite* es, teniendo seleccionada la *suite* en el explorador, pulsar el botón de ejecución (ó F6):



Depuración

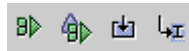
En lugar de simplemente ejecutar la aplicación para probarla, si queremos localizar fallos en ella, podemos utilizar el depurador que lleva integrado NetBeans. Podemos establecer puntos de ruptura (*breakpoints*) en el código para que cuando la ejecución llegue a ese lugar se detenga, permitiéndonos ejecutar paso a paso y ver detenidamente lo que ocurre. Para establecer un punto de ruptura pincharemos sobre la banda gris a la izquierda de la línea donde queremos que se detenga, quedando marcada de la siguiente forma:

```

public class MIDletPrueba extends javax.microedition.midlet.MIDlet {
    public void startApp() {
        Display d = Display.getDisplay(this);
        Form f = new Form("Hola mundo!");
        d.setCurrent(f);
    }
}

```

Para ejecutar la aplicación en modo depuración utilizaremos los siguientes botones:



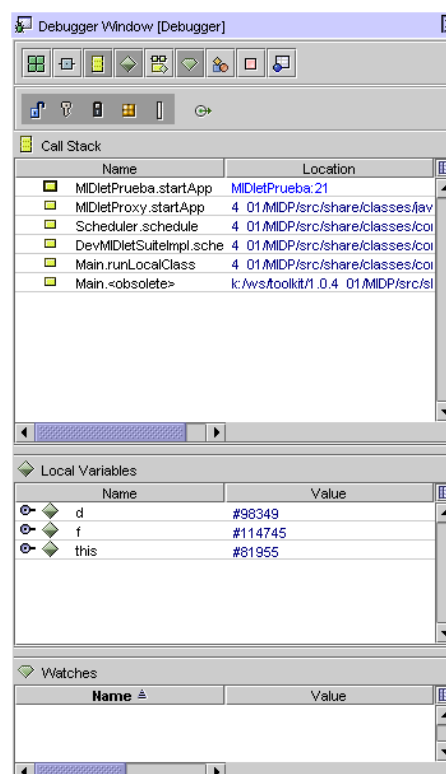
El primero de ellos nos servirá para comenzar la ejecución hasta que llegue un punto de ruptura. Una vez se produzca se detendrá el programa y podremos ir ejecutando instrucciones paso a paso utilizando el botón correspondiente en la barra de botones anterior. Se mostrará con una flecha verde la línea que se va a ejecutar en cada momento, como se muestra a continuación:

```

public class MIDletPrueba extends javax.microedition.midlet.MIDlet {
    public void startApp() {
        Display d = Display.getDisplay(this);
        Form f = new Form("Hola mundo!");
        d.setCurrent(f);
    }
}

```

Mientras se ejecuta el programa podemos ver el estado de la memoria y de las llamadas a métodos en la ventana del depurador. Para ello tendremos que estar en vista de depuración (pestaña **Debugger** del entorno). Veremos la siguiente información:

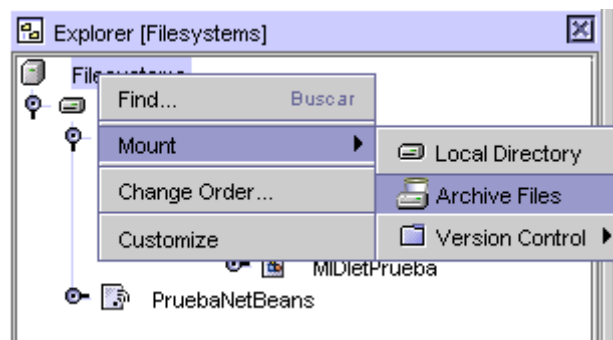


Aquí podremos ver los valores que toma cada variable conforme se ejecuta el código, lo cual nos facilitará la detección de fallos en nuestro programas.

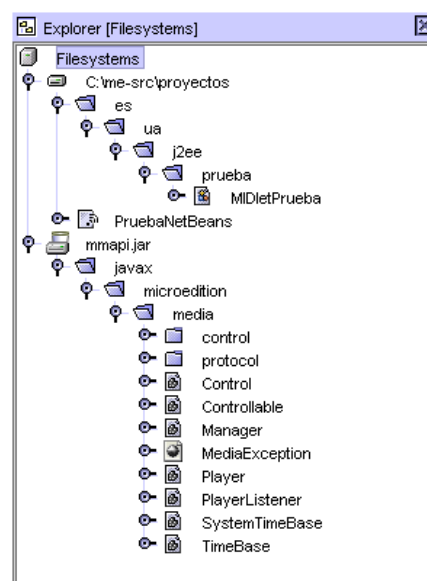
Librerías adicionales

Las librerías que se utilizan al compilar y ejecutar las aplicaciones MIDP son las librerías que aporte el emulador seleccionado, al igual que ocurriría con WTK. Sin embargo, conforme editamos el código sólo cuenta con que estemos utilizando la API de MIDP básica, por lo que todos los elementos que incluyamos de librerías adicionales nos los marcará como erróneos, y no nos permitirá autocompletar los nombres para ellos.

Para que reconozca estos elementos correctamente deberemos añadir estas librerías a los sistemas de ficheros montados en el entorno. Para ello seleccionamos montar un sistema de ficheros desde un archivo, como vemos en la siguiente figura, de forma que nos permita seleccionar el fichero JAR correspondiente a la librería que queremos añadir.



Una vez montada la librería JAR, podremos verla en el explorador. Ahora considerará que esa librería está en el *classpath* y nos permitirá utilizar sus elementos en el editor de código sin mostrar errores. Podremos navegar por los elementos de la librería dentro de explorador:



Esto será suficiente si la librería corresponde a una API disponible en el teléfono móvil, como puede ser MMAPI, WMA y APIs propietarias de Nokia.

Si lo que queremos es añadir una librería al fichero JAR de la aplicación para introducirla en el móvil junto a la aplicación, lo primero que haremos es montarla como acabamos de ver. Una vez montada, podemos ir a la ventana de edición de la *suite* como hemos visto antes (haciendo doble *click* sobre la *suite*), y en la pestaña **Jar Contents** podremos añadir las librerías montadas a nuestro JAR.

2.5.3. Otros entornos

A parte de los entornos que hemos visto, existen numerosos IDEs para desarrollo con J2ME, la mayoría de ellos de pago. A continuación vamos a ver brevemente los más destacados.

Sun One Studio ME

Se trata de la versión ME (*Micro Edition*) del entorno de desarrollo de Sun, Sun One Studio, anteriormente llamado Forte for Java. Esta versión ME está dirigida a crear aplicaciones J2ME, e incluye todo el software necesario para realizar esta tarea, no hace falta instalar por separado el WTK ni otras herramientas.

El entorno es muy parecido a NetBeans. Podemos descargar una versión de prueba sin ninguna limitación. Una ventaja de este entorno es que podemos integrarlo con otros kits de desarrollo como por ejemplo el kit de desarrollo de Nokia.

JBuilder y MobileSet

Podemos utilizar también el entorno de Borland, JBuilder, con la extensión MobileSet. A partir de la versión 9 de JBuilder tenemos una edición Mobile para trabajar con aplicaciones J2ME directamente sin tener que instalar ninguna extensión. Podemos descargar de forma gratuita la versión personal del entorno JBuilder, pero tiene el inconveniente de estar bastante más limitada que las versiones de pago.

Este entorno puede también integrarse con el kit de desarrollo de Nokia. Además como característica adicional podremos crear de forma visual la GUI de las aplicaciones móviles. Esta característica no está muy extendida por este tipo de entornos debido a la simplicidad de las GUIs para móviles.

JDeveloper y J2ME Plugin

El entorno de desarrollo de Oracle, JDeveloper, está dedicado principalmente a la creación de aplicaciones J2EE, permitiéndonos crear un gran número de componentes Java, como *servlets*, JSPs, EJBs, servicios web, etc. Para facilitar la tarea de creación de estos componentes, automatizando todo lo posible, utiliza APIs propietarias de Oracle.

Podemos trabajar directamente en vista de diseño, utilizar distintos patrones de diseño para desarrollar las aplicaciones web, etc. Tiene integrado un servidor de aplicaciones propio para probar las aplicaciones en modo local, y nos permite establecer conexiones a BDs y a servidores de aplicaciones para realizar el despliegue de estas aplicaciones.

Aunque está principalmente dedicado para aplicaciones web con J2EE, también podemos utilizarlo para aplicaciones J2SE. Además también podemos encontrar un *plugin* para realizar aplicaciones J2ME, permitiéndonos crear MIDlets y *suites* mediante asistentes, y ejecutar las aplicaciones directamente en emuladores.

Podemos descargar de forma gratuita una versión de prueba de este entorno de la web sin limitaciones.

Websphere Studio Device Developer

Se trata de un entorno de IBM basado en Eclipse, por lo que tiene una interfaz similar. Este entorno está dedicado a la programación de aplicaciones para dispositivos móviles. Integra los asistentes necesarios para la creación de los componentes de aplicaciones MIDP, así como las herramientas de desarrollo necesarias y nos permite probar la aplicación directamente en emuladores desde el mismo entorno.

Podemos encontrar en la web una versión de prueba sin limitaciones para descargar.

Codewarrior Wireless Studio

Este es otro entorno bastante utilizado también para el desarrollo de aplicaciones para móviles. Está desarrollado por Metrowerks y se puede encontrar disponible para un gran número de plataformas distintas. Existe una versión de evaluación limitada a 30 días de uso que puede ser encargada desde la web.

3. Introducción a Java para MIDs

3.1. Introducción a Java

Java es un lenguaje de programación creado por *Sun Microsystems* para poder funcionar en distintos tipos de procesadores. Su sintaxis es muy parecida a la de C o C++, e incorpora como propias algunas características que en otros lenguajes son extensiones: gestión de hilos, ejecución remota, etc.

El código Java, una vez compilado, puede llevarse sin modificación alguna sobre cualquier máquina, y ejecutarlo. Esto se debe a que el código se ejecuta sobre una máquina hipotética o virtual, la **Java Virtual Machine**, que se encarga de interpretar el código (ficheros compilados `.class`) y convertirlo a código particular de la CPU que se esté utilizando (siempre que se soporte dicha máquina virtual).

En el caso de los MIDs, este código intermedio Java se ejecutará sobre una versión reducida de la máquina virtual, la **KVM (Kilobyte Virtual Machine)**.

Cuando se programa con Java, se dispone de antemano de un conjunto de clases ya implementadas. Estas clases (aparte de las que pueda hacer el usuario) forman parte del propio lenguaje (lo que se conoce como **API (Application Programming Interface)** de Java).

La API que se utilizará para programar las aplicaciones para MIDs será la API de MIDP, que contendrá un conjunto reducido de clases que nos permitan realizar las tareas fundamentales en estas aplicaciones. La implementación de esta API estará optimizada para ejecutarse en este tipo de dispositivos.

3.2. Introducción a la Programación Orientada a Objetos (POO)

3.2.1. Objetos y clases

- **Objeto:** conjunto de variables junto con los métodos relacionados con éstas. Contiene la **información** (las variables) y la forma de manipular la información (los métodos).
- **Clase:** prototipo que define las variables y métodos que va a emplear un determinado tipo de objeto.

3.2.2. Campos, métodos y constructores

- **Campos:** contienen la información relativa a la clase
- **Métodos:** permiten manipular dicha información.
- **Constructores:** reservan memoria para almacenar un objeto de esa clase.

3.2.3. Herencia y polimorfismo

Con la **herencia** podemos definir una clase a partir de otra que ya exista, de forma que la nueva clase tendrá todas las variables y métodos de la clase a partir de la que se crea, más las variables y métodos nuevos que necesite. A la clase base a partir de la cual se crea la nueva clase se le llama **superclase**.

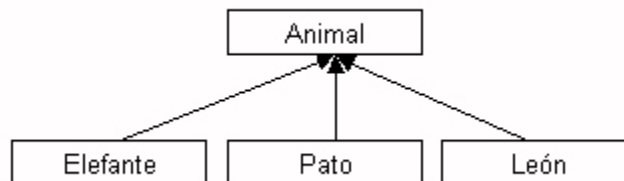


Figura 1. Ejemplo de herencia

Por ejemplo, tenemos una clase genérica `Animal`, y heredamos de ella para formar clases más específicas, como `Pato`, `Elefante`, o `León`. Si tenemos por ejemplo el método `dibuja(Animal a)`, podremos pasarle a este método como parámetro tanto un `Animal` como un `Pato`, `Elefante`, etc. Esto se conoce como **polimorfismo**.

3.2.4. Clases abstractas e interfaces

Mediante las **clases abstractas** y los **interfaces** podemos definir el esqueleto de una familia de clases, de forma que los subtipos de la clase abstracta o la interfaz implementen ese esqueleto para dicho subtipo concreto. Por ejemplo, podemos definir en la clase `Animal` el método `dibuja()` y el método `imprime()`, y que `Animal` sea una clase abstracta o un interfaz.

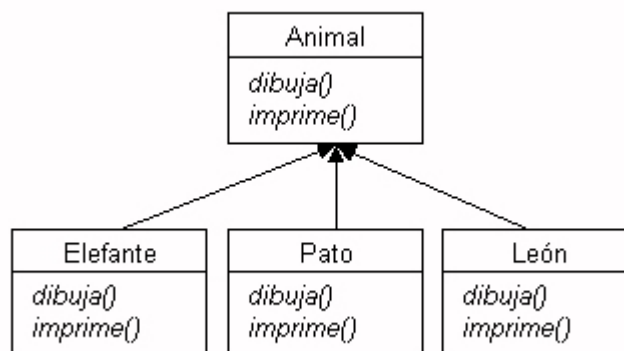


Figura 2. Ejemplo de interfaz y clase abstracta

Vemos la diferencia entre clase, clase abstracta e interfaz con este esquema:

- En una **clase**, al definir `Animal` tendríamos que implementar los métodos `dibuja()` e `imprime()`. Las clases hijas no tendrían por qué implementar

los métodos, a no ser que quieran adaptarlos a sus propias necesidades.

- En una **clase abstracta** podríamos implementar los métodos que nos interese, dejando sin implementar los demás (dejándolos como métodos abstractos). Dichos métodos tendrían que implementarse en las clases hijas.
- En un **interfaz** no podemos implementar ningún método en la clase padre, y cada clase hija tiene que hacer sus propias implementaciones de los métodos. Además, las clases hija podrían implementar otros interfaces.

3.3. Conceptos Básicos de Java

3.3.1. Componentes de un programa Java

En un programa Java podemos distinguir varios elementos:

- **Paquetes:** equivalentes a los "include" de C, permiten utilizar clases en otras, y llamarlas de forma abreviada:

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

- **Clases:**

```
public class  
MiClase  
{  
    ...
```

- **Campos:** Constantes, variables y en general elementos de información.

```
public int a;  
Vector v;
```

- **Métodos:** Para las funciones que devuelvan algún tipo de valor, es imprescindible colocar una sentencia `return` en la función.

```
public void imprimirA()  
public void insertarVector(String cadena)
```

- **Constructores:** Un tipo de método que siempre tiene el mismo nombre que la clase. Se pueden definir uno o varios.

```
public MiClase()
```

Así, podemos definir una **instancia** con **new**:

```
MiClase mc;  
mc = new MiClase ();  
mc.a++;  
mc.insertarVector("hola");
```

No tenemos que preocuparnos de liberar la memoria del objeto al dejar de utilizarlo. Esto lo hace automáticamente el **garbage collector**. A diferencia de J2SE, en MIDP los objetos no tienen el método `finalize`.

3.3.2. Otras posibilidades

- **Herencia:** Se utiliza la palabra `extends` para decir de qué clase se hereda. Para hacer que `Pato` herede de `Animal`:

```
class Pato extends Animal
```

- `this` se usa para hacer referencia a los miembros de la propia clase. Se utiliza cuando hay otros elementos con el mismo nombre, para distinguir:

```
public class MiClase {  
    int i;  
    public MiClase (int i) {  
        this.i = i;    // i de la clase = parametro i  
    }  
}
```

- `super` se usa para llamar al mismo elemento en la clase padre. Si la clase `MiClase` tiene un método `Suma_a_i(...)`, lo llamamos con:

```
public class MiNuevaClase extends MiClase {  
    public void Suma_a_i (int j) {  
        i = i + (j / 2);  
        super.Suma_a_i (j);  
    }  
}
```

- **Modificadores:** en algunos elementos (campos, métodos, clases, etc) se utilizan algunos de estos modificadores al declararlos:
 - `public`: cualquier objeto puede acceder al elemento
 - `protected`: sólo pueden acceder las subclases de la clase.
 - `private`: sólo pueden ser accedidos desde dentro de la clase.
 - `abstract`: elemento base para la herencia (los objetos subtipo deberán definir este elemento).
 - `static`: elemento compartido por todos los objetos de la misma clase.
 - `final`: objeto final, no modificable ni heredable.
 - `synchronized`: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución.
- **Clases abstractas e Interfaces:** si queremos definir una clase (por ejemplo, `Animal`), como clase abstracta o como interfaz, se declara como sigue (respectivamente). También se indica cómo hacer una clase o interfaz subtipo de la clase o interfaz padre (en este caso, la subclase es `Pato`):

```
public interface Animal
{
    void dibujar ();
    void imprimir ();
}
```

```
public class Pato
implements Animal
{
    void dibujar() {
        codigo; }
    void imprimir() {
        codigo; }
}
```

```
public abstract class
Animal
{
    abstract void dibujar
();
    void imprimir () {
        codigo; }
}
```

```
public class Pato extends
Animal
{
    void dibujar() {
        codigo; }
}
```

- **Paquetes:** la palabra `package` permite agrupar clases e interfaces. Los nombres de los paquetes son palabras separadas por puntos, y se almacenan en directorios que coinciden con esos nombres. Así, si definimos la clase:

```
package paquetel.subpaquetel;
public class MiClase1_1
...
```

haremos que la clase `MiClase1_1` pertenezca al subpaquete `subpaquetel` del paquete `paquetel`.

Para utilizar las clases de un paquete utilizamos `import`:

```
import java.Date;
import paquetel.subpaquetel.*;
import java.awt.*;
```

Para importar todas las clases del paquete se utiliza el asterisco `*` (aunque no vayamos a usarlas todas, si utilizamos varias de ellas puede ser útil simplificar con un asterisco). Si sólo queremos importar una o algunas pocas, se pone un `import` por cada una, terminando el paquete con el nombre de la clase en lugar del asterisco (como pasa con `Date` en el ejemplo).

Al poner `import` podemos utilizar el nombre corto de la clase. Es decir, si ponemos:

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

Podemos hacer referencia a un objeto `Date` o a un objeto `Vector` (una clase del paquete `java.util`) con:

```
Date d = ...
Vector v = ...
```

Si no pusiéramos los `import`, deberíamos hacer referencia a los objetos con:

```
java.Date d = ...
java.util.Vector v = ...
```

Es decir, cada vez que queramos poner el nombre de la clase, deberíamos colocar todo el nombre, con los paquetes y subpaquetes.

Cuando no especificamos el paquete al que pertenece una clase, esa clase será incluida en un paquete *sin nombre*. Al no tener nombre, no podremos importar este paquete desde otras clases pertenecientes a paquetes distintos, por lo que no podrán utilizar esta clase.

Cuando realicemos aplicaciones en Java es importante asignar un nombre de paquete a cada clase, de forma que puedan ser localizadas. El utilizar clases en paquetes *sin nombre* nos servirá únicamente si queremos hacer un programa de forma rápida para hacer alguna prueba, pero no se debe hacer en ningún otro caso.

La forma recomendada de asignar nombre a los paquetes de las aplicaciones que desarrollemos será similar a las DNS de Internet pero al revés, es decir, comenzaremos por el dominio, compañía, subunidad y nombre de la aplicación. Por ejemplo, si tenemos la URL `j2ee.ua.es` y vamos a realizar una aplicación llamada `prueba`, pondremos las clases en un paquete `es.ua.j2ee.prueba` o subpaquetes del mismo.

3.3.3. Sintaxis de Java

Tipos de datos

Se tienen los siguientes tipos de datos simples. Además, se pueden crear complejos, todos los cuales serán subtipos de `Object`

Tipo	Tamaño/Formato	Descripción	Ejemplos
byte	8 bits, complemento a 2	Entero de 1 byte	210, 0x456
short	16 bits, complemento a 2	Entero corto	"

int	32 bits, complemento a 2	Entero	"
long	64 bits, complemento a 2	Entero largo	"
char	16 bits, carácter	Carácter simple	'a'
boolean	true / false	verdadero / falso	true, false

Aquí desaparecen los tipos `float` y `double` que podíamos usar en otras ediciones de Java. Esto es debido a que la **KVM** no tiene soporte para estos tipos, ya que las operaciones con números reales son complejas y estos dispositivos muchas veces no tienen unidad de punto flotante.

Las aplicaciones J2ME para dispositivos CDC si que podrán usar estos tipos de datos, ya que funcionarán con la máquina virtual **CVM** que si que soporta estos tipos. Por lo tanto la limitación no es de J2ME, sino de la máquina virtual **KVM** en la que se basan las aplicaciones CLDC.

NOTA: En CLDC 1.1 se incorporan los tipos de datos `double` y `float`. En los dispositivos que soporten esta versión de la API podremos utilizar estos tipos de datos.

Cadenas

Para trabajar con cadenas de caracteres se utiliza la clase `String`. Un valor posible para este tipo de datos es "Hola mundo". Cuando escribamos una cadena de este tipo dentro del código Java, se creará un objeto `String` encapsulando dicha cadena.

Si trabajamos con cadenas largas, o vamos a realizar bastantes operaciones que modifiquen la cadena, será conveniente utilizar `StringBuffer`, ya que se trata de una implementación más eficiente. La clase `String` no permite modificar el contenido de la cadena, por lo que cualquier modificación implicará reservar más memoria. `StringBuffer` si que nos permite modificar el *buffer* interno donde almacena la cadena, de forma que podremos hacer modificaciones sin tener que instanciar nuevos objetos.

Arrays

Se definen *arrays* o conjuntos de elementos de forma similar a como se hace en C. Hay 2 métodos:

```
int a[] = new int [10];
String s[] = {"Hola", "Adios"};
```

No pueden crearse *arrays* estáticos en tiempo de compilación (`int a[8];`), ni rellenar un *array* sin definir previamente su tamaño con el operador `new`. La función miembro `length` se puede utilizar para conocer la longitud del *array*:

```
int a [][] = new int [10] [3];
a.length;           // Devolvería 10
a[0].length;        // Devolvería 3
```


Los *arrays* empiezan a numerarse desde 0, hasta el tope definido menos uno (como en C).

Identificadores

Nombran variables, funciones, clases y objetos. Comienzan por una letra, carácter de subrayado `'_'` o símbolo `'$'`. El resto de caracteres pueden ser letras o dígitos (o `'_'`). Se distinguen mayúsculas de minúsculas, y no hay longitud máxima. Las variables en Java sólo son válidas desde el punto donde se declaran hasta el final de la sentencia compuesta (las llaves) que la engloba. No se puede declarar una variable con igual nombre que una del mismo ámbito.

En Java se tiene también un término NULL, pero si bien el de C es con mayúsculas (`NULL`), éste es con minúsculas (`null`):

```
String a = null;
...
if (a == null)...
```

Referencias

En Java no existen punteros, simplemente se crea otro objeto que referencie al que queremos "apuntar".

```
MiClase mc = new
MiClase();
MiClase mc2 = mc;
```

```
MiClase mc = new
MiClase();
MiClase mc2 = new
MiClase();
```

`mc2` y `mc` apuntan a la misma variable (al cambiar una cambiará la otra).

Tendremos dos objetos apuntando a elementos diferentes en memoria.

Comentarios

```
// comentarios para una sola línea

/* comentarios de
una o más líneas */

/** comentarios de documentación para javadoc,
de una o más líneas */
```

Operadores

Se muestra una tabla con los operadores en orden de precedencia:

Operador	Ejemplo	Descripción
.	a.length	Campo o método de objeto
[]	a[6]	Referencia a elemento de array
()	(a + b)	Agrupación de operaciones
++, --	a++; b--	Autoincremento / Autodecremento de 1 unidad
!, ~	!a ; ~b	Negación / Complemento
instanceof	a instanceof TipoDato	Indica si <i>a</i> es del tipo <i>TipoDato</i>
*, /, %	a*b; b/c; c%a	Multipliación, división y resto de división entera
+, -	a+b; b-c	Suma y resta
<<, >>	a>>2; b<<1	Desplazamiento de bits a izquierda y derecha
<, >, <=, >=, ==, !=	a>b; b==c; c!=a	Comparaciones (mayor, menor, igual, distinto...)
&, , ^	a&b; b c	AND, OR y XOR lógicas
&&,	a&&b; b c	AND y OR condicionales
?:	a?b:c	Condicional: si <i>a</i> entonces <i>b</i> , si no <i>c</i>
=, +=, -=, *=, /= ...	a=b; b*=c	Asignación. a += b equivale a (a = a + b)

Puesto que con la KVM no tenemos soporte para números reales, la operación de división será entera. Nos devolverá un valor entero.

Control de flujo

TOMA DE DECISIONES

Este tipo de sentencias definen el código que debe ejecutarse si se cumple una determinada condición. Se dispone de sentencias `if` y de sentencias `switch`:

Sintaxis

```
if (condicion1) {
    sentencias;
} else if
(condicion2) {
```

Ejemplos

```
if
(a == 1) {
    b++;
} else if (b
```

```

    sentencias;
    ...
} else
if(condicionN) {
    sentencias;
} else {
    sentencias;
}

switch (condicion)
{
    case caso1:
sentencias;
    case caso2:
sentencias;
    case casoN:
sentencias;
    default:
sentencias;
}

```

```

== 1) {
    c++;
} else if (c
== 1) {
    d++;
}

switch (a) {
    case 1:
b++;

    break;
    case 2:
c++;

    break;
    default:b-
-;

    break;
}

```

BUCLES

Para repetir un conjunto de sentencias durante un determinado número de iteraciones se tienen las sentencias `for`, `while` y `do...while`:

Sintaxis

```

for(inicio;condicion;
    incremento)
{
    sentencias;
}

```

```

while (condicion){
    sentencias;
}

```

```

do{
    sentencias;
} while (condicion);

```

Ejemplo

```

for
(i=1;i<10;
i++)
{
    b =
b+i;
}

```

```

while (i <
10) {
    b += i;
    i++;
}

```

```

do {
    b += i;
    i++;
} while (i
< 10);

```

SENTENCIAS DE RUPTURA

Se tienen las sentencias `break` (para terminar la ejecución de un bloque o saltar a una etiqueta), `continue` (para forzar una ejecución más de un bloque o saltar a una etiqueta) y `return` (para salir de una función devolviendo o sin devolver un valor):

```
public int miFuncion(int n)
{
    int i = 0;
    while (i < n)
    {
        i++;
        if (i > 10)
            // Sale del while
            break;
        if (i < 5)
            // Fuerza una iteracion mas
            continue;
    }
    // Devuelve lo que valga i al llegar aquí
    return i;
}
```

Números reales

En CLDC 1.0 echamos en falta el soporte para número de coma flotante (`float` y `double`). En principio podemos pensar que esto es una gran limitación, sobretodo para aplicaciones que necesiten trabajar con valores de este tipo. Por ejemplo, si estamos trabajando con información monetaria para mostrar el precio de los productos necesitaremos utilizar números como 13.95€.

Sin embargo, en muchos casos podremos valernos de los números enteros para representar estos números reales. Vamos a ver un truco con el que implementar soporte para números reales de coma fija mediante datos de tipo entero (`int`).

Este truco consiste en considerar un número *N* fijo de decimales, por ejemplo en el caso de los precios podemos considerar que van a tener 2 decimales. Entonces lo que haremos será trabajar con números enteros, considerando que las *N* últimas cifras son los decimales. Por ejemplo, si un producto cuesta 13.95€, lo guardaremos en una variable entera con valor 1395, es decir, en este caso es como si estuviésemos guardando la información en céntimos.

Cuando queramos mostrar este valor, deberemos separar la parte entera y la fraccionaria para imprimirlo con el formato correspondiente a un número real. Haremos la siguiente transformación:

```
public String imprimeReal(int numero) {
    int entero = numero / 100;
    int fraccion = numero % 100;
    return entero + "." + fraccion;
}
```

Cuando el usuario introduzca un número con formato real, y queramos leerlo y guardarlo en una variable de tipo entero (`int`) deberemos hacer la transformación contraria:

```
public int leeReal(String numero) {
    int pos_coma = numero.indexOf('.');
}
```

```
String entero = numero.substring(0, pos_coma - 1);

String fraccion = numero.substring(pos_coma + 1,
                                   pos_coma + 2);

return Integer.parseInt(entero)*100 +
       Integer.parseInt(fraccion);
}
```

Es posible que necesitemos realizar operaciones básicas con estos números reales. Podremos realizar operaciones como suma, resta, multiplicación y división utilizando la representación como enteros de estos números.

El caso de la suma y de la resta es sencillo. Si sumamos o restamos dos números con N decimales cada uno, podremos sumarlos como si fuesen enteros y sabremos que las últimas N cifras del resultado son decimales. Por ejemplo, si queremos añadir dos productos a la cesta de la compra, cuyos precios son 13.95€ y 5.20€ respectivamente, deberemos sumar estas cantidades para obtener el importe total. Para ello las trataremos como enteros y hacemos la siguiente suma:

$$1395 + 520 = 1915$$

Por lo tanto, el resultado de la suma de los números reales será 19.15€.

El caso de la multiplicación es algo más complejo. Si queremos multiplicar dos números, con N y M decimales respectivamente, podremos hacer la multiplicación como si fuesen enteros sabiendo que el resultado tendrá $N+M$ decimales. Por ejemplo, si al importe anterior de 19.15€ queremos añadirle el IVA, tendremos que multiplicarlo por 1.16. Haremos la siguiente operación entera:

$$1915 * 116 = 222140$$

El resultado real será 22.2140€, ya que si cada operando tenía 2 decimales, el resultado tendrá 4.

Si estas operaciones básicas no son suficiente podemos utilizar una librería como **MathFP**, que nos permitirá realizar operaciones más complejas con números de coma fija representados como enteros. Entre ellas tenemos disponibles operaciones trigonométricas, logarítmicas, exponenciales, potencias, etc. Podemos descargar esta librería de <http://www.jscience.net/> e incluirla libremente en nuestra aplicaciones J2ME.

3.4. Características básicas de CLDC

Vamos a ver las características básicas del lenguaje Java (plataforma J2SE) que tenemos disponibles en la API CLDC de los dispositivos móviles. Dentro de esta API tenemos la parte básica del lenguaje que debe estar disponible en cualquier dispositivo conectado limitado.

Esta API básica se ha tomado directamente de J2SE, de forma que los programadores que conozcan el lenguaje Java podrán programar de forma sencilla aplicaciones para dispositivos móviles sin tener que aprender a manejar una API totalmente distinta. Sólo tendrán que aprender a utilizar la parte de la API propia de estos dispositivos móviles, que se utiliza para características que sólo están presentes en estos dispositivos.

Dado que estos dispositivos tienen una capacidad muy limitada, en CLDC sólo está disponible una parte reducida de esta API de Java. Vamos a ver en este punto qué características de las que ya conocemos del lenguaje Java están presentes en CLDC para programar en dispositivos móviles.

3.4.1. Excepciones

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

Tipos de excepciones

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

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

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

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

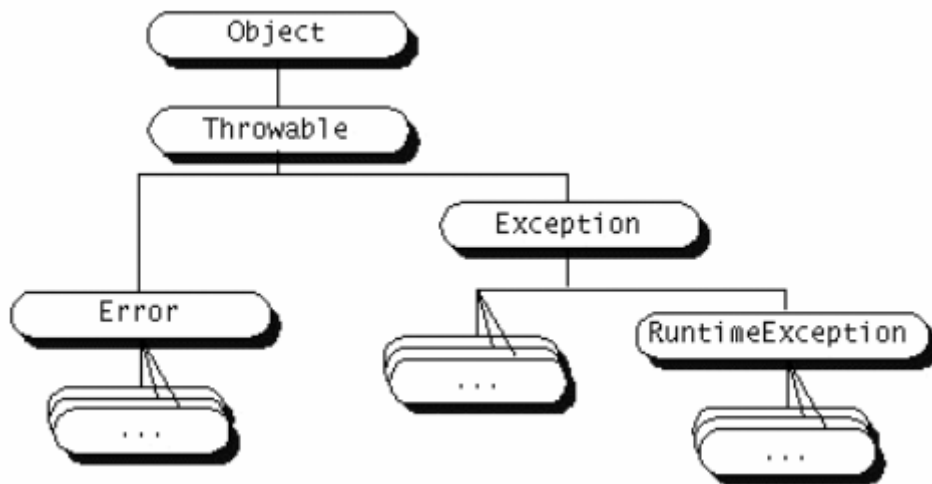


Figura 3. Tipos de excepciones

Dentro de estos grupos principales de excepciones podremos encontrar tipos concretos de excepciones o bien otros grupos que a su vez pueden contener más subgrupos de excepciones, hasta llegar a tipos concretos de ellas. Cada tipo de excepción guardará información relativa al tipo de error al que se refiera, además de la información común a todas las excepciones.

Captura de excepciones

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

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

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

```
try {
    // Código regular del programa
    // Puede producir excepciones
} catch (TipoDeExcepcion1 e1) {
```

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

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

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

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

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

Normalmente en los dispositivos móviles cuando imprimimos por la salida estándar se ignorará lo que estamos imprimiendo (se envía a un dispositivo *null*), por lo que imprimir esta traza en el dispositivo no tiene mucho sentido. Puede resultar útil para depurar la aplicación mientras la estemos probando en emuladores, ya que en este caso cuando imprimamos por la salida estándar veremos los mensajes en la consola.

Un ejemplo de uso:

```
try
{
    ... // Aquí va el código que puede lanzar una excepción
} catch (Exception e) {
    muestraAlerta("El error es: " + e.getMessage());
    e.printStackTrace();
}
```


Lanzamiento de excepciones

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

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

```
public void lee_datos()  
throws IOException, ClassNotFoundException  
{  
    // Cuerpo de la función  
}
```

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

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

```
throw new ClassNotFoundException(mensaje_error);
```

- Juntando estos dos pasos:

```
public void lee_datos()  
throws IOException, ClassNotFoundException  
{  
    ...  
    throw new ClassNotFoundException(mensaje_error);  
    ...  
}
```

Podremos lanzar así excepciones en nuestras funciones para indicar que algo no es como debiera ser a las funciones llamadas.

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

Creación de nuevas excepciones

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

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

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

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

3.4.2. Hilos

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

Creación de hilos

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

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

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

Para crear nuestro hilo mediante herencia haremos lo siguiente:

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

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

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

Crear un hilo heredando de `Thread` tiene el problema de que al no haber herencia múltiple en Java, si heredamos de `Thread` no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

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

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

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

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

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

Estado y propiedades de los hilos

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

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

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

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

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

La única forma de parar un hilo es hacer que salga del método `run` de forma natural. Podremos conseguir esto haciendo que se cumpla la condición de

salida del bucle principal definido dentro del `run`. Las funciones para parar, pausar y reanudar hilos están desaprobadadas en las versiones actuales de Java.

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

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

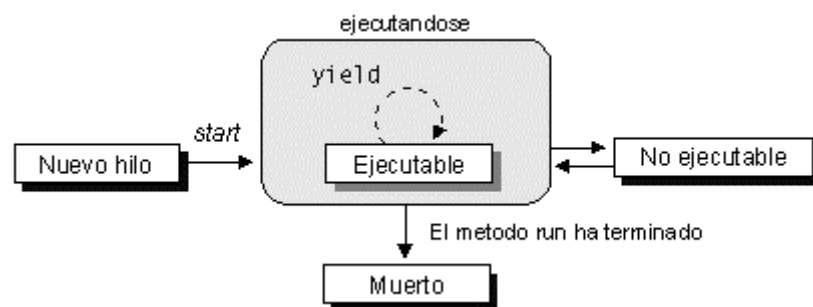


Figura 4. Ciclo de vida de los hilos

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

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

Para cambiar la prioridad de un hilo se utiliza el método `setPriority`, al que deberemos proporcionar un valor de prioridad entre `MIN_PRIORITY` y `MAX_PRIORITY`.

Sincronización de hilos

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

Un primer mecanismo de comunicación es la variable cerrojo incluida en todo objeto `Object`, que permitirá evitar que más de un hilo entre en la sección crítica. Los métodos declarados como `synchronized` utilizan el cerrojo de la

clase a la que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

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

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

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

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

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

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

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

En la API de CLDC no están presentes los grupos de hilos. La clase `ThreadGroup` de la API de J2SE no existe en la API de CLDC, por lo que no podremos utilizar esta característica desde los MIDs. Tampoco podemos ejecutar hilos como demonios (*daemon*).

3.4.3. Tipos de datos

En J2SE existe lo que se conoce como marco de colecciones, que comprende una serie de tipos de datos. Estos tipos de datos se denominan colecciones por ser una colección de elementos, tenemos distintos subtipos de colecciones como las listas (secuencias de elementos), conjuntos (colecciones sin elementos repetidos) y mapas (conjunto de parejas *<clave, valor>*). Tendremos varias implementaciones de estos tipos de datos, siendo sus operadores polimórficos, es decir, se utilizan los mismos operadores para distintos tipos de datos. Para ello se definen interfaces que deben implementar estos tipos de datos, una serie de implementaciones de estas interfaces y algoritmos para trabajar con ellos.

Sin embargo, en CLDC no tenemos este marco de colecciones. Al tener que utilizar una API tan reducida como sea posible, tenemos solamente las clases `Vector` (tipo lista), `Stack` (tipo pila) y `Hashtable` (mapa) tal como ocurría en las primeras versiones de Java. Estas clases son independientes, no implementan ninguna interfaz común.

Enumeraciones

Para consultar las colecciones de elementos que contienen estos tipos de datos, podemos utilizar las enumeraciones. En J2SE teníamos la posibilidad de utilizar también iteradores, pero la clase `Iterator` no está disponible en CLDC.

Las enumeraciones, definidas mediante la interfaz `Enumeration`, nos permiten consultar los elementos que contiene una colección de datos. Muchos métodos de clases Java que deben devolver múltiples valores, lo que hacen es devolvernos una enumeración que podremos consultar mediante los métodos que ofrece dicha interfaz.

La enumeración irá recorriendo secuencialmente los elementos de la colección. Para leer cada elemento de la enumeración deberemos llamar al método:

```
Object item = enum.nextElement();
```

Que nos proporcionará en cada momento el siguiente elemento de la enumeración a leer. Además necesitaremos saber si quedan elementos por leer, para ello tenemos el método:

```
enum.hasMoreElements()
```

Normalmente, el bucle para la lectura de una enumeración será el siguiente:

```
while (enum.hasMoreElements()) {  
    Object item = enum.nextElement();  
    // Hacer algo con el item leído  
}
```

Vemos como en este bucle se van leyendo y procesando elementos de la enumeración uno a uno mientras queden elementos por leer en ella.

Vector

Implementa una lista de elementos mediante un *array* de tamaño variable. Conforme se añaden elementos el tamaño del *array* irá creciendo si es necesario. El *array* tendrá una capacidad inicial, y en el momento en el que se rebase dicha capacidad, se aumentará el tamaño del *array*.

Las operaciones de añadir un elemento al final del *array* (*add*), y de establecer u obtener el elemento en una determinada posición (*get/set*) tienen un coste temporal constante. Las inserciones y borrados tienen un coste lineal $O(n)$, donde n es el número de elementos del *array*.

Los métodos que tenemos para trabajar con el `Vector` son los métodos que tenía en las primeras versiones de Java:

```
void addElement(Object obj)
```

Añade un elemento al final del vector.

```
Object elementAt(int indice)
```

Devuelve el elemento de la posición del vector indicada por el índice.

```
void insertElementAt(Object obj, int indice)
```

Inserta un elemento en la posición indicada.

```
boolean removeElement(Object obj)
```

Elimina el elemento indicado del vector, devolviendo `true` si dicho elemento estaba contenido en el vector, y `false` en caso contrario.

```
void removeElementAt(int indice)
```

Elimina el elemento de la posición indicada en el índice.

```
void setElementAt(Object obj, int indice)
```

Sobrescribe el elemento de la posición indicada con el objeto especificado.

```
int size()
```

Devuelve el número de elementos del vector.

Stack

Sobre el vector se construye el tipo pila (`Stack`), que apoyándose en el tipo vector ofrece métodos para trabajar con dicho vector como si se tratase de una pila, apilando y desapilando elementos (operaciones `push` y `pop` respectivamente). La clase `Stack` hereda de `Vector`, por lo que en realidad será un vector que ofrece métodos adicionales para trabajar con él como si fuese una pila.

Hashtable

Relaciona una clave (*key*) con un valor. Contendrá un conjunto de claves, y a cada clave se le asociará un determinado valor. Tanto la clave como el valor puede ser cualquier objeto. Lo que contendrá este tipo de dato será una colección de parejas *<clave, valor>*.

Los métodos básicos para trabajar con estos elementos son los siguientes:

```
Object get(Object clave)
```

Nos devuelve el valor asociado a la clave indicada

```
Object put(Object clave, Object valor)
```

Inserta una nueva clave con el valor especificado. Nos devuelve el valor que tenía antes dicha clave, o `null` si la clave no estaba en la tabla todavía.

```
Object remove(Object clave)
```

Elimina una clave, devolviéndonos el valor que tenía dicha clave.

```
Enumeration keys()
```

Este método nos devolverá una enumeración de todas las claves registradas en la tabla.

```
int size()
```

Nos devuelve el número de parejas *<clave,valor>* registradas.

Wrappers de tipos básicos

Hemos visto que en Java cualquier tipo de datos es un objeto, excepto los tipos de datos básicos: `boolean`, `int`, `long`, `byte`, `short`, `char`.

Cuando trabajamos con estos tipos de datos los elementos que contienen éstos son siempre objetos, por lo que en un principio no podríamos insertar elementos de estos tipos básicos. Para hacer esto posible tenemos una serie de objetos que se encargarán de envolver a estos tipos básicos, permitiéndonos tratarlos como objetos y por lo tanto insertarlos como elementos de colecciones. Estos objetos son los llamados *wrappers*, y las clases en las que se definen tienen nombres similares al del tipo básico que encapsulan, con la diferencia de que comienzan con mayúscula: `Boolean`, `Integer`, `Long`, `Byte`, `Short`, `Character`.

Estas clases, además de servirnos para encapsular estos datos básicos en forma de objetos, nos proporcionan una serie de métodos e información útiles para trabajar con estos datos. Nos proporcionarán métodos por ejemplo para convertir cadenas a datos numéricos de distintos tipos y viceversa, así como información acerca del valor mínimo y máximo que se puede representar con cada tipo numérico.

NOTA: Dado que a partir de CLDC 1.1 se incorporan los tipos de datos `float` y `double`, aparecerán también sus correspondientes *wrappers*: `Float` y `Double`.

3.4.4. Clases útiles

Vamos a ver ahora una serie de clases básicas del lenguaje Java que siguen estando en CLDC. La versión de CLDC de estas clases estará normalmente más limitada, a continuación veremos las diferencias existentes entre la versión de J2SE y la de CLDC.

Object

Esta es la clase base de todas las clases en Java, toda clase hereda en última instancia de la clase `Object`, por lo que los métodos que ofrece estarán disponibles en cualquier objeto Java, sea de la clase que sea.

En Java es importante distinguir claramente entre lo que es una variable, y lo que es un objeto. Las variables simplemente son referencias a objetos, mientras que los objetos son las entidades instanciadas en memoria que podrán ser manipulados mediante las referencias que tenemos a ellos (mediante variable que apunten a ellos) dentro de nuestro programa. Cuando hacemos lo siguiente:

```
new MiClase()
```

Se está instanciando en memoria un nuevo objeto de clase `MiClase` y nos devuelve una referencia a dicho objeto. Nosotros deberemos guardarnos dicha referencia en alguna variable con el fin de poder acceder al objeto creado desde nuestro programa:

```
MiClase mc = new MiClase();
```

Es importante declarar la referencia del tipo adecuado (en este caso tipo `MiClase`) para manipular el objeto, ya que el tipo de la referencia será el que indicará al compilador las operaciones que podremos realizar con dicho objeto. El tipo de esta referencia podrá ser tanto el mismo tipo del objeto al que vayamos a apuntar, o bien el de cualquier clase de la que herede o interfaz que implemente nuestro objeto. Por ejemplo, si `MiClase` se define de la siguiente forma:

```
public class MiClase extends Thread implements List {  
    ...  
}
```

Podremos hacer referencia a ella de diferentes formas:

```
MiClase mc = new MiClase();  
Thread t = new MiClase();  
List l = new MiClase();  
Object o = new MiClase();
```

Esto es así ya que al heredar tanto de `Thread` como de `Object`, sabemos que el objeto tendrá todo lo que tienen estas clases más lo que añade `MiClase`, por lo que podrá comportarse como cualquiera de las clases anteriores. Lo mismo ocurre al implementar una interfaz, al forzar a que se implementen sus métodos podremos hacer referencia al objeto mediante la interfaz ya que sabemos que va a contener todos esos métodos. Siempre vamos a poder hacer esta asignación 'ascendente' a clases o interfaces de las que deriva nuestro objeto.

Si hacemos referencia a un objeto `MiClase` mediante una referencia `Object` por ejemplo, sólo podremos acceder a los métodos de `Object`, aunque el objeto contenga métodos adicionales definidos en `MiClase`. Si conocemos que nuestro objeto es de tipo `MiClase`, y queremos poder utilizarlo como tal, podremos hacer una asignación 'descendente' aplicando una conversión `cast` al tipo concreto de objeto:

```
Object o = new MiClase();
...
MiClase mc = (MiClase) o;
```

Si resultase que nuestro objeto no es de la clase a la que hacemos `cast`, ni hereda de ella ni la implementa, esta llamada resultará en un `ClassCastException` indicando que no podemos hacer referencia a dicho objeto mediante esa interfaz debido a que el objeto no la cumple, y por lo tanto podrán no estar disponibles los métodos que se definen en ella.

Una vez hemos visto la diferencia entre las variables (referencias) y objetos (entidades) vamos a ver como se hará la asignación y comparación de objetos. Si hiciésemos lo siguiente:

```
MiClase mc1 = new MiClase();
MiClase mc2 = mc1;
```

Puesto que hemos dicho que las variables simplemente son referencias a objetos, la asignación estará copiando una referencia, no el objeto. Es decir, tanto la variable `mc1` como `mc2` apuntarán a un mismo objeto.

En J2SE la clase `Object` tiene un método `clone` que podemos utilizar para realizar una copia del objeto, de forma que tengamos dos objetos independientes en memoria con el mismo contenido. Este método no existe en CLDC, por lo que si queremos realizar una copia de un objeto deberemos definir un constructor de copia, es decir, un constructor que construya un nuevo objeto copiando todas las propiedades de otro objeto de la misma clase.

Por ejemplo, si tenemos una clase `Punto2D`, cuyas propiedades sean las coordenadas (x,y) del punto, podemos definir un constructor de copia como se muestra a continuación:

```
public class Punto2D {
    public int x, y;

    ...

    public Punto2D(Punto2D p) {
        this.x = p.x;
        this.y = p.y
    }
}
```

Por otro lado, para la comparación, si hacemos lo siguiente:

```
mc1 == mc2
```

Estaremos comparando referencias, por lo que estaremos viendo si las dos referencias apuntan a un mismo objeto, y no si los objetos a los que apuntan son iguales. Para ver si los objetos son iguales, aunque sean entidades distintas, tenemos:

```
mc1.equals(mc2)
```

Este método también es propio de la clase `Object`, y será el que se utilice para comparar internamente los objetos.

El método `equals`, deberá ser redefinido en nuestras clases para adaptarse a éstas. Deberemos especificar dentro de él como se compara si dos objetos de esta clase son iguales:

```
public class Punto2D {  
    public int x, y;  
  
    ...  
  
    public boolean equals(Object o) {  
        Punto2D p = (Punto2D)o;  
        // Compara objeto this con objeto p  
        return (x == p.x && y == p.y);  
    }  
}
```

Un último método interesante de la clase `Object` es `toString`. Este método nos devuelve una cadena (`String`) que representa dicho objeto. Por defecto nos dará un identificador del objeto, pero nosotros podemos sobrescribirla en nuestras propias clases para que genere la cadena que queramos. De esta manera podremos imprimir el objeto en forma de cadena de texto, mostrándose los datos con el formato que nosotros les hayamos dado en `toString`. Por ejemplo, si tenemos una clase `Punto2D`, sería buena idea hacer que su conversión a cadena muestre las coordenadas (x,y) del punto:

```
public class Punto2D {  
    public int x,y;  
  
    ...  
  
    public String toString() {  
        String s = "(" + x + "," + y + ")";  
        return s;  
    }  
}
```

System

Esta clase nos ofrece una serie de métodos y campos útiles del sistema. Esta clase no se debe instanciar, todos estos métodos y campos son estáticos.

Podemos encontrar los objetos que encapsulan la salida y salida de error estándar como veremos con más detalle en el apartado de entrada/salida. A diferencia de J2SE, en CLDC no tenemos entrada estándar.

Tampoco nos permite instalar un gestor de seguridad para la aplicación. La API de CLDC y MIDP ya cuenta con las limitaciones suficientes para que las aplicaciones sean seguras.

Otros métodos útiles que encontramos son:

```
void exit(int estado)
```

Finaliza la ejecución de la aplicación, devolviendo un código de estado. Normalmente el código 0 significa que ha salido de forma normal, mientras que con otros códigos indicaremos que se ha producido algún error. Este método produce que se cierre la máquina virtual de Java, normalmente no utilizaremos este método directamente en las aplicaciones MIDP, haremos que el AMS sea quien cierre la aplicación, como veremos más adelante.

```
void gc()
```

Fuerza una llamada al colector de basura para limpiar la memoria. Esta es una operación costosa. Normalmente no lo llamaremos explícitamente, sino que dejaremos que Java lo invoque cuando sea necesario.

```
long currentTimeMillis()
```

Nos devuelve el tiempo medido en el número de milisegundos transcurridos desde el 1 de Enero de 1970 a las 0:00.

```
void arraycopy(Object fuente, int pos_fuente,  
               Object destino, int pos_dest, int n)
```

Copia *n* elementos del *array* *fuente*, desde la posición *pos_fuente*, al *array* *destino* a partir de la posición *pos_dest*.

```
String getProperty(String key)
```

En CLDC no tenemos una clase `Properties` con una colección de propiedades. Por esta razón, cuando leamos propiedades del sistema no podremos obtenerlas en un objeto `Properties`, sino que tendremos que leerlas individualmente. Estas son propiedades del sistema, no son las propiedades del usuario que aparecen en el fichero JAD. En el próximo tema veremos cómo leer estas propiedades del usuario.

Runtime

Toda aplicación Java tiene una instancia de la clase `Runtime` que se encargará de hacer de interfaz con el entorno en el que se está ejecutando. Para obtener este objeto debemos utilizar el siguiente método estático:

```
Runtime rt = Runtime.getRuntime();
```

En J2SE podemos utilizar esta clase para ejecutar comandos del sistema con `exec`. En CLDC no disponemos de esta característica. Lo que si podremos hacer con este objeto es obtener la memoria del sistema, y la memoria libre.

Math

La clase `Math` nos será de gran utilidad cuando necesitemos realizar operaciones matemáticas. Esta clase no necesita ser instanciada, ya que todos sus métodos son estáticos. En CLDC 1.0, al no contar con soporte para números reales, esta clase contendrá muy pocos métodos, sólo tendrá aquellas operaciones que trabajan con números enteros, como las operaciones de valor absoluto, máximo y mínimo.

Random

La clase `Random` nos permitirá generar números aleatorios. En CLDC 1.0 sólo nos permitirá generar números enteros de forma aleatoria, ya que no tenemos soporte para reales.

Fechas y horas

Si miramos dentro del paquete `java.util`, podremos encontrar una serie de clases que nos podrán resultar útiles para determinadas aplicaciones.

Entre ellas tenemos la clase `Calendar`, que junto a `Date` nos servirá cuando trabajemos con fechas y horas. La clase `Date` representará un determinado instante de tiempo, en tiempo absoluto. Esta clase trabaja con el tiempo medido en milisegundos desde el 1 de enero de 1970 a las 0:00, por lo que será difícil trabajar con esta información directamente.

Podremos utilizar la clase `Calendar` para obtener un determinado instante de tiempo encapsulado en un objeto `Date`, proporcionando información de alto nivel como el año, mes, día, hora, minuto y segundo.

Con `TimeZone` podemos representar una determinada zona horaria, con lo que podremos utilizarla junto a las clases anteriores para obtener diferencias horarias.

Temporizadores

Los temporizadores nos permitirán planificar tareas para ser ejecutadas por un hilo en segundo plano. Para trabajar con temporizadores tenemos las clases `Timer` y `TimerTask`.

Lo primero que deberemos hacer es crear las tareas que queramos planificar. Para crear una tarea crearemos una clase que herede de `TimerTask`, y que defina un método `run` donde incluiremos el código que implemente la tarea.

```
public class MiTarea extends TimerTask {  
    public void run() {  
        // Código de la tarea  
    }  
}
```

Una vez definida la tarea, utilizaremos un objeto `Timer` para planificarla. Para ello deberemos establecer el tiempo de comienzo de dicha tarea, cosa que puede hacerse de dos formas diferentes:

- **Retardo** (*delay*): Nos permitirá planificar la tarea para que comience a ejecutarse transcurrido un tiempo dado. Por ejemplo, podemos hacer que una determinada tarea comience a ejecutarse dentro de 10 segundos.
- **Fecha y hora**: Podemos hacer que la tarea comience a una determinada hora y fecha dada en tiempo absoluto. Por ejemplo, podemos hacer que a las 8:00 se ejecute una tarea que haga de despertador.

Tenemos diferentes formas de planificación de tareas, según el número de veces y la periodicidad con la que se ejecutan:

- **Una sola ejecución**: Se ejecuta en el tiempo de inicio especificado y no se vuelve a ejecutar a no ser que la volvamos a planificar.
- **Repetida con retardo fijo**: Se ejecuta repetidas veces, con un determinado retardo entre cada dos ejecuciones consecutivas. Este retardo podremos especificarlo nosotros. La tarea se volverá a ejecutar siempre transcurrido este tiempo desde la última vez que se ejecutó, hasta que detengamos el temporizador.
- **Repetida con frecuencia constante**: Se ejecuta repetidas veces con una frecuencia dada. Deberemos especificar el retardo que queremos entre dos ejecuciones consecutivas. A diferencia del caso anterior, no se toma como referencia el tiempo de ejecución de la tarea anterior, sino el tiempo de inicio de la primera ejecución. De esta forma, si una ejecución se retrasa por alguna razón, como por ejemplo por tener demasiada carga el procesador, la siguiente tarea comenzará transcurrido un tiempo menor, para mantener la frecuencia deseada.

Deberemos como primer paso crear el temporizador y la tarea que vamos a planificar:

```
Timer t = new Timer();  
TimerTask tarea = new MiTarea();
```

Ahora podemos planificarla para comenzar con un retardo, o bien a una determinada fecha y hora. Si vamos a hacerlo por retardo, utilizaremos uno de los siguientes métodos, según la periodicidad:

```
t.schedule(tarea, retardo); // Una vez  
t.schedule(tarea, retardo, periodo); // Retardo  
fijo  
t.scheduleAtFixedRate(tarea, retardo, periodo); // Frecuencia  
constante
```

Si queremos comenzar a una determinada fecha y hora, deberemos utilizar un objeto `Date` para especificar este tiempo de comienzo:

```
Calendar calendario = Calendar.getInstance();  
calendario.set(Calendar.HOUR_OF_DAY, 8);  
calendario.set(Calendar.MINUTE, 0);  
calendario.set(Calendar.SECOND, 0);  
calendario.set(Calendar.MONTH, Calendar.SEPTEMBER);  
calendario.set(Calendar.DAY_OF_MONTH, 22);  
Date fecha = calendario.getTime();
```

Una vez obtenido este objeto con la fecha a la que queremos comenzar la tarea (en nuestro ejemplo el día 22 de septiembre a las 8:00), podemos planificarla con el temporizador igual que en el caso anterior:

```
t.schedule(tarea, fecha); // Una vez  
t.schedule(tarea, fecha, periodo); // Retardo fijo  
t.scheduleAtFixedRate(tarea, fecha, periodo); // Frecuencia  
constante
```

Los temporizadores nos serán útiles en las aplicaciones móviles para realizar aplicaciones como por ejemplo agendas o alarmas. La planificación por retardo nos permitirá mostrar ventanas de transición en nuestras aplicaciones durante un número determinado de segundos.

Si queremos que un temporizador no vuelva a ejecutar la tarea planificada, utilizaremos su método `cancel` para cancelarlo.

```
t.cancel();
```

Una vez cancelado el temporizador, no podrá volverse a poner en marcha de nuevo. Si queremos volver a planificar la tarea deberemos crear un temporizador nuevo.

3.4.5. Flujos de entrada/salida

Los programas muy a menudo necesitan enviar datos a un determinado destino, o bien leerlos de una determinada fuente externa, como por ejemplo puede ser un fichero para almacenar datos de forma permanente, o bien enviar datos a través de la red, a memoria, o a otros programas. Esta entrada/salida

de datos en Java la realizaremos por medio de *flujos (streams)* de datos, a través de los cuales un programa podrá recibir o enviar datos en serie.

En las aplicaciones CLDC, normalmente utilizaremos flujos para enviar o recibir datos a través de la red, o para leer o escribir datos en algún *buffer* de memoria.

Existen varios objetos que hacen de flujos de datos, y que se distinguen por la finalidad del flujo de datos y por el tipo de datos que viajen a través de ellos. Según el tipo de datos que transporten podemos distinguir:

- Flujos de caracteres
- Flujos de *bytes*

Dentro de cada uno de estos grupos tenemos varios pares de objetos, de los cuales uno nos servirá para leer del flujo y el otro para escribir en él. Cada par de objetos será utilizado para comunicarse con distintos elementos (memoria, red, etc). Estas clases, según sean de entrada o salida y según sean de caracteres o de *bytes* llevarán distintos sufijos, según se muestra en la siguiente tabla:

	Flujo de entrada / lector	Flujo de salida / escritor
Caractéres	<code>_Reader</code>	<code>_Writer</code>
Bytes	<code>_InputStream</code>	<code>_OutputStream</code>

Además podemos distinguir los flujos de datos según su propósito, pudiendo ser:

- Canales de datos, simplemente para leer o escribir datos directamente en una fuente o sumidero externo.
- Flujos de procesamiento, que además de enviar o recibir datos realizan algún procesamiento con ellos. Tenemos por ejemplo flujos conversores datos (con prefijo `Data`) que permiten escribir distintos tipos de datos (numéricos, *booleanos*, *bytes*, caracteres), y flujos preparados para la impresión de elementos (con prefijo `Print`) que ofrecen métodos para imprimir distintos tipos de datos en forma de cadena de texto.

Un tipo de filtros de procesamiento a destacar son aquellos que nos permiten convertir un flujo de *bytes* a flujo de caracteres. Estos objetos son `InputStreamReader` y `OutputStreamWriter`. Como podemos ver en su sufijo, son flujos de caracteres, pero se construyen a partir de flujos de *bytes*, permitiendo de esta manera acceder a nuestro flujo de *bytes* como si fuese un flujo de caracteres.

Para cada uno de los tipos básicos de flujo que hemos visto existe una superclase, de la que heredaran todos sus subtipos, y que contienen una serie de métodos que serán comunes a todos ellos. Entre estos métodos encontramos los métodos básicos para leer o escribir caracteres o *bytes* en el

flujo a bajo nivel. En la siguiente tabla se muestran los métodos más importantes de cada objeto:

InputStream	read() , reset() , available() , close()
OutputStream	write(int b) , flush() , close()
Reader	read() , reset() , close()
Writer	write(int c) , flush() , close()

En CLDC no encontramos flujos para acceder directamente a ficheros, ya que no podemos contar con poder acceder al sistema de ficheros de los dispositivos móviles, esta característica será opcional. Tampoco tenemos disponible ningún *tokenizer*, por lo que la lectura y escritura deberá hacerse a bajo nivel como acabamos de ver, e implementar nuestro propio analizador léxico en caso necesario.

Serialización de objetos

Otra característica que no está disponible en CLDC es la serialización automática de objetos, por lo que no podremos enviar directamente objetos a través de los flujos de datos. No existe ninguna forma de serializar cualquier objeto arbitrario automáticamente en CLDC, ya que no soporta *reflection*.

Sin embargo, podemos hacerlo de una forma más sencilla, y es haciendo que cada objeto particular proporcione métodos para serializarse y deserializarse. Estos métodos los deberemos escribir nosotros, adaptándolos a las características de los objetos.

Por ejemplo, supongamos que tenemos una clase `Punto2D` como la siguiente:

```
public class Punto2D {  
    int x;  
    int y;  
    String etiqueta;  
    ...  
}
```

Los datos que contiene cada objeto de esta clase son las coordenadas (x,y) del punto y una etiqueta para identificar este punto. Si queremos serializar un objeto de esta clase esta será la información que deberemos codificar en forma de serie de *bytes*.

Podemos crear dos métodos manualmente para codificar y decodificar esta información en forma de *array* de *bytes*, como se muestra a continuación:

```
public class Punto2D {  
    int x;  
    int y;  
    String etiqueta;  
    ...  
    public void serialize(OutputStream out) throws IOException  
{
```

```
        DataOutputStream dos = new DataOutputStream( out );

        dos.writeInt(x);
        dos.writeInt(y);
        dos.writeUTF(etiqueta);
        dos.flush();
    }

    public static Punto2D deserialize(InputStream in)
        throws IOException {
        DataInputStream dis = new DataInputStream( in );

        Punto2D p = new Punto2D();
        p.x = dis.readInt();
        p.y = dis.readInt();
        p.etiqueta = dis.readUTF();

        return p;
    }
}
```

Hemos visto como los flujos de procesamiento `DataOutputStream` y `DataInputStream` nos facilitan la codificación de distintos tipos de datos para ser enviados a través de un flujo de datos.

Acceso a los recursos

Hemos visto que no podemos acceder al sistema de ficheros directamente como hacíamos en J2SE. Sin embargo, con las aplicaciones MIDP podemos incluir una serie de recursos a los que deberemos poder acceder. Estos recursos son ficheros incluidos en el fichero JAR de la aplicación, como por ejemplo sonidos, imágenes o ficheros de datos.

Para acceder a estos recursos deberemos abrir un flujo de entrada que se encargue de leer su contenido. Para ello utilizaremos el método `getResourceAsStream` de la clase `Class`:

```
InputStream in = getClass().getResourceAsStream("datos.txt");
```

De esta forma podremos utilizar el flujo de entrada obtenido para leer el contenido del fichero que hayamos indicado. Este fichero deberá estar contenido en el JAR de la aplicación.

Salida y salida de error estándar

Al igual que en C, en Java también existen los conceptos de entrada, salida, y salida de error estándar. En J2SE la entrada estándar normalmente se refiere a lo que el usuario escribe en la consola, aunque el sistema operativo puede hacer que se tome de otra fuente. De la misma forma la salida y la salida de error estándar lo que hacen normalmente es mostrar los mensajes y los errores del programa respectivamente en la consola, aunque el sistema operativo también podrá redirigirlas a otro destino.

En los MIDs no tenemos consola, por lo que los mensajes que imprimamos por la salida estándar normalmente serán ignorados. Esta salida estará dirigida a un dispositivo *null* en los teléfonos móviles. Sin embargo, imprimir por la salida estándar puede resultarnos útil mientras estemos probando la aplicaciones en emuladores, ya que al ejecutarse en el ordenador estos emuladores, estos mensajes si que se mostrarán por la consola, por lo que podremos imprimir en ellos información que nos sirva para depurar las aplicaciones.

En MIDP no existe la entrada estándar. La salida y salida de error estándar se tratan de la misma forma que cualquier otro flujo de datos, estando estos dos elementos encapsulados en dos objetos de flujo de datos que se encuentran como propiedades estáticas de la clase `System`:

	Tipo	Objeto
Salida estándar	<code>PrintStream</code>	<code>System.out</code>
Salida de error estándar	<code>PrintStream</code>	<code>System.err</code>

Se utilizan objetos `PrintWriter` que facilitan la impresión de texto ofreciendo a parte del método común de bajo nivel `write(int b)` para escribir *bytes*, dos métodos más: `print(s)` y `println(s)`. Estas funciones nos permitirán escribir cualquier cadena, tipo básico, o bien cualquier objeto que defina el método `toString()` que devuelva una representación del objeto en forma de cadena. La única diferencia entre los dos métodos es que el segundo añade automáticamente un salto de línea al final del texto impreso, mientras que en el primero deberemos especificar explícitamente este salto.

Para escribir texto en la consola normalmente utilizaremos:

```
System.out.println("Hola mundo");
```

En el caso de la impresión de errores por la salida de error de estándar, deberemos utilizar:

```
System.err.println("Error: Se ha producido un error");
```

3.4.6. Características ausentes

Además de las diferencias que hemos visto en los puntos anteriores, tenemos APIs que han desaparecido en su totalidad, o prácticamente en su totalidad.

Reflection

En CLDC no está presente la API de *reflection*. Sólo está presente la clase `Class` con la que podremos cargar clases dinámicamente y comprobar la clase a la que pertenece un objeto en tiempo de ejecución. Tenemos además en esta clase el método `getResourceAsStream` que hemos visto anteriormente, que nos servirá para acceder a los recursos dentro del JAR de la aplicación.

Red

La API para el acceso a la red de J2SE es demasiado compleja para los MIDs. Por esta razón se ha sustituido por una nueva API totalmente distinta, adaptada a las necesidades de conectividad de estos dispositivos. Desaparece la API `java.net`, para acceder a la red ahora deberemos utilizar la API `javax.microedition.io` incluida en CLDC que veremos en detalle en el próximo tema.

AWT/Swing

Las librerías para la creación de interfaces gráficas, AWT y Swing, desaparecen totalmente ya que estas interfaces no son adecuadas para las pantallas de los MIDs. Para crear la interfaz gráfica de las aplicaciones para móviles tendremos la API `javax.microedition.lcdui` perteneciente a MIDP.

4. MIDlets

Hasta ahora hemos visto la parte básica del lenguaje Java que podemos utilizar en los dispositivos móviles. Esta parte de la API está basada en la API básica de J2SE, reducida y optimizada para su utilización en dispositivos de baja capacidad. Esta es la base que necesitaremos para programar cualquier tipo de dispositivo, sin embargo con ella por si sola no podemos acceder a las características propias de los móviles, como su pantalla, su teclado, reproducir tonos, etc.

Vamos a ver ahora las APIs propias para el desarrollo de aplicaciones móviles. Estas APIs ya no están basadas en APIs existentes en J2SE, sino que se han desarrollado específicamente para la programación en estos dispositivos. Todas ellas pertenecen al paquete `javax.microedition`.

Los MIDlets son las aplicaciones para MIDs, realizadas con la API de MIDP. La clase principal de cualquier aplicación MIDP deberá ser un MIDlet. Ese MIDlet podrá utilizar cualquier otra clase Java y la API de MIDP para realizar sus funciones.

Para crear un MIDlet deberemos heredar de la clase `MIDlet`. Esta clase define una serie de métodos abstractos que deberemos definir en nuestros MIDlets, introduciendo en ellos el código propio de nuestra aplicación:

```
protected abstract void startApp();  
protected abstract void pauseApp();  
protected abstract void destroyApp(boolean incondicional);
```

A continuación veremos con más detalle qué deberemos introducir en cada uno de estos métodos.

4.1. Componentes y contenedores

Numerosas veces encontramos dentro de las tecnologías Java el concepto de componentes y contenedores. Los componentes son elementos que tienen una determinada interfaz, y los contenedores son la infraestructura que da soporte a estos componentes.

Por ejemplo, podemos ver los *applets* como un tipo de componente, que para poderse ejecutar necesita un navegador web que haga de contenedor y que lo soporte. De la misma forma, los *servlets* son componentes que encapsulan el mecanismo petición/respuesta de la web, y el servidor web tendrá un contenedor que de soporte a estos componentes, para ejecutarlos cuando se produzca una petición desde un cliente. De esta forma nosotros podemos deberemos definir sólo el componente, con su correspondiente interfaz, y será el contenedor quien se encargue de controlar su ciclo de vida (instanciarlo, ejecutarlo, destruirlo).

Cuando desarrollamos componentes, no deberemos crear el método `main`, ya que estos componentes no se ejecutan como una aplicación independiente

(*stand-alone*), sino que son ejecutados dentro de una aplicación ya existente, que será el contenedor.

El contenedor que da soporte a los MIDlets recibe el nombre de *Application Management Software* (AMS). El AMS además de controlar el ciclo de vida de la ejecución MIDlets (inicio, pausa, destrucción), controlará el ciclo de vida de las aplicaciones que se instalen en el móvil (instalación, actualización, ejecución, desinstalación).

4.2. Ciclo de vida

Durante su ciclo de vida un MIDlet puede estar en los siguientes estados:

- **Activo:** El MIDlet se está ejecutando actualmente.
- **Pausado:** El MIDlet se encuentra a mitad de una ejecución pero está pausado. La ejecución podrá reanudarse, pasando de nuevo a estado activo.
- **Destruído:** El MIDlet ha terminado su ejecución y ha liberado todos los recursos, por lo que ya no se puede volver a estado activo. La aplicación está cerrada, por lo que para volver a ponerla en marcha tendríamos que volver a ejecutarla.

Será el AMS quién se encargue de controlar este ciclo de vida, es decir, quién realice las transiciones de un estado a otro. Nosotros podremos saber cuando hemos entrado en cada uno de estos estados porque el AMS invocará al método correspondiente dentro de la clase del MIDlet. Estos métodos son los que se muestran en el siguiente esqueleto de un MIDlet:

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

public class MiMIDlet extends MIDlet {

    protected void startApp()
        throws MIDletStateChangeException {
        // Estado activo -> comenzar
    }

    protected void pauseApp() {
        // Estado pausa -> detener hilos
    }

    protected void destroyApp(boolean incondicional)
        throws MIDletStateChangeException {
        // Estado destruido -> liberar recursos
    }

}
```

Deberemos definir los siguientes métodos para controlar el ciclo de vida del MIDlet:

- **startApp():** Este método se invocará cuando el MIDlet pase a estado activo. Es aquí donde insertaremos el código correspondiente a la tarea que debe realizar dicho MIDlet.

Si ocurre un error que impida que el MIDlet empiece a ejecutarse deberemos notificarlo. Podemos distinguir entre errores pasajeros o errores permanentes. Los errores pasajeros impiden que el MIDlet se empiece a ejecutar ahora, pero podría hacerlo más tarde. Los permanentes se dan cuando el MIDlet no podrá ejecutarse nunca.

Pasajero: En el caso de que el error sea pasajero, lo notificaremos lanzando una excepción de tipo `MIDletStateChangeException`, de modo que el MIDlet pasará a estado pausado, y se volverá intentar activar más tarde.

Permanente: Si por el contrario el error es permanente, entonces deberemos destruir el MIDlet llamando a `notifyDestroyed` porque sabemos que nunca podrá ejecutarse correctamente. Si se lanza una excepción de tipo `RuntimeException` dentro del método `startApp` tendremos el mismo efecto, se destruirá el MIDlet.

- `pauseApp()`: Se invocará cuando se pause el MIDlet. En él deberemos detener las actividades que esté realizando nuestra aplicación.

Igual que en el caso anterior, si se produce una excepción de tipo `RuntimeException` durante la ejecución de este método, el MIDlet se destruirá.

- `destroyApp(boolean incondicional)`: Se invocará cuando se vaya a destruir la aplicación. En él deberemos incluir el código para liberar todos los recursos que estuviese usando el MIDlet. Con el *flag* que nos proporciona como parámetro indica si la destrucción es incondicional o no. Es decir, si `incondicional` es `true`, entonces se destruirá siempre. En caso de que sea `false`, podemos hacer que no se destruya lanzando la excepción `MIDletStateChangeException` desde dentro de este método.

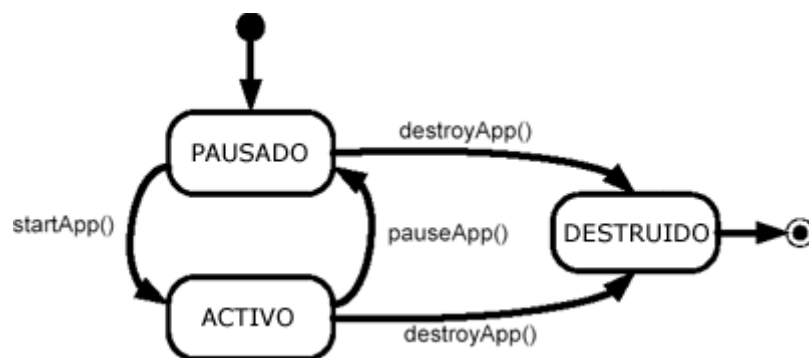


Figura 1. Ciclo de vida de un MIDlet

Hemos visto que el AMS es quien realiza las transiciones entre distintos estados. Sin embargo, nosotros podremos forzar a que se produzcan transiciones a los estados pausado o destruido:

- `notifyDestroyed()`: Destruye el MIDlet. Utilizaremos este método cuando queramos finalizar la aplicación. Por ejemplo, podemos ejecutar

este método como respuesta a la pulsación del botón "Salir" por parte del usuario.

NOTA: La llamada a este método notifica que el MIDlet ha sido destruido, pero no invoca el método `destroyApp` para liberar los recursos, por lo que tendremos que invocarlo nosotros manualmente antes de llamar a `notifyDestroyed`.

- `notifyPause()`: Notifica al AMS de que el MIDlet ha entrado en modo pausa. Después de esto, el AMS podrá realizar una llamada a `startApp` para volverlo a poner en estado activo.
- `resumeRequest()`: Solicita al AMS que el MIDlet vuelva a ponerse activo. De esta forma, si el AMS tiene varios MIDlets candidatos para activar, elegirá alguno de aquellos que lo hayan solicitado. Este método no fuerza a que se produzca la transición como en los anteriores, simplemente lo solicita al AMS y será éste quién decida.

4.3. Cerrar la aplicación

La aplicación puede ser cerrada por el AMS, por ejemplo si desde el sistema operativo del móvil hemos forzado a que se cierre. En ese caso, el AMS invocará el método `destroyApp` que nosotros habremos definido para liberar los recursos, y pasará a estado **destruido**.

Si queremos hacer que la aplicación termine de ejecutarse desde dentro del código, nunca utilizaremos el método `System.exit` (o `Runtime.exit`), ya que estos métodos se utilizan para salir de la máquina virtual. En este caso, como se trata de un componente, si ejecutásemos este método cerraríamos toda la aplicación, es decir, el AMS. Por esta razón esto no se permite, si intentásemos hacerlo obtendríamos una excepción de seguridad.

La única forma de salir de una aplicación MIDP es haciendo pasar el componente a estado destruido, como hemos visto en el punto anterior, para que el contenedor pueda eliminarlo. Esto lo haremos invocando `notifyDestroyed` para cambiar el estado a destruido. Sin embargo, si hacemos esto no se invocará automáticamente el método `destroyApp` para liberar los recursos, por lo que deberemos ejecutarlo nosotros manualmente antes de marcar la aplicación como destruida:

```
public void salir() {
    try {
        destroyApp(true);
    } catch(MIDletStateChangeException e) {
    }

    notifyDestroyed();
}
```

Si queremos implementar una salida condicional, para que el método `destroyApp` pueda decidir si permitir que se cierre o no la aplicación, podemos hacerlo de la siguiente forma:


```
public void salir_cond() {  
    try {  
  
        destroyApp(false);  
  
        notifyDestroyed();  
    } catch (MIDletStateChangeException e) {  
    }  
}
```

4.4. Parametrización de los MIDlets

Podemos añadir una serie de propiedades en el fichero descriptor de la aplicación (JAD), que podrán ser leídas desde el MIDlet. De esta forma, podremos cambiar el valor de estas propiedades sin tener que rehacer el fichero JAR.

Cada propiedad consistirá en una clave (*key*) y en un valor. La clave será el nombre de la propiedad. De esta forma tendremos un conjunto de parámetros de configuración (claves) con un valor asignado a cada una. Podremos cambiar fácilmente estos valores editando el fichero JAD con cualquier editor de texto.

Para leer estas propiedades desde el MIDlet utilizaremos el método:

```
String valor = getAppProperty(String key)
```

Que nos devolverá el valor asignado a la clave con nombre *key*.

4.5. Peticiones al dispositivo

A partir de MIDP 2.0 se incorpora una nueva función que nos permite realizar peticiones que se encargará de gestionar el dispositivo, de forma externa a nuestra aplicación. Por ejemplo, con esta función podremos realizar una llamada a un número telefónico o abrir el navegador web instalado para mostrar un determinado documento.

Para realizar este tipo de peticiones utilizaremos el siguiente método:

```
boolean debeSalir = platformRequest(url);
```

Esto proporcionará una URL al AMS, que determinará, según el tipo de la URL, qué servicio debe invocar. Además nos devolverá un valor *booleano* que indicará si para que este servicio sea ejecutado debemos cerrar el MIDlet antes. Algunos servicios de determinados dispositivos no pueden ejecutarse concurrentemente con nuestra aplicación, por lo que en estos casos hasta que no la cerremos no se ejecutará el servicio.

Los tipos servicios que se pueden solicitar dependen de las características del móvil en el que se ejecute. Cada fabricante puede ofrecer un serie de servicios accesibles mediante determinados tipos de URLs. Sin intentamos acceder a un

servicio que no está disponible en el móvil, se producirá una excepción de tipo `ConnectionNotFoundException`.

En el estándar de MIDP 2.0 sólo se definen URLs para dos tipos de servicios:

- Iniciar una llamada de voz. Para ello se utilizará una URL como la siguiente:

```
tel:<numero>
```

Por ejemplo, podríamos poner:

```
tel:+34-965-123-456.
```

- Instalar una *suite* de MIDlets. Se proporciona la URL donde está el fichero JAD de la suite que queramos instalar. Por ejemplo:

```
http://www.j2ee.ua.es/prueba/aplic.jad
```

Si como URL proporcionamos una cadena vacía (no `null`), se cancelarán todas las peticiones de servicios anteriores.

5. Interfaz de usuario

Vamos a ver ahora como crear la interfaz de las aplicaciones MIDP. En la reducida pantalla de los móviles no tendremos una consola en la que imprimir utilizando la salida estándar, por lo que toda la salida la tendremos que mostrar utilizando una API propia que nos permita crear componentes adecuados para ser mostrados en este tipo de pantallas.

Esta API propia para crear la interfaz gráfica de usuario de los MIDlets se denomina LCDUI (*Limited Connected Devices User Interface*), y se encuentra en el paquete `javax.microedition.lcdui`.

5.1. Acceso al visor

El visor del dispositivo está representado por un objeto `Display`. Este objeto nos permitirá acceder a este visor y a los dispositivos de entrada (normalmente el teclado) del móvil.

Tendremos asociado un único *display* a cada aplicación (MIDlet). Para obtener el *display* asociado a nuestro MIDlet deberemos utilizar el siguiente método estático:

```
Display mi_display = Display.getDisplay(mi_midlet);
```

Donde *mi_midlet* será una referencia al MIDlet del cual queremos obtener el `Display`. Podremos acceder a este *display* desde el momento en que `startApp` es invocado por primera vez (no podremos hacerlo en el constructor del MIDlet), y una vez se haya terminado de ejecutar `destroyApp` ya no podremos volver a acceder al *display* del MIDlet.

Cada MIDlet tiene un *display* y sólo uno. Si el MIDlet ha pasado a segundo plano (pausado), seguirá asociado al mismo *display*, pero en ese momento no se mostrará su contenido en la pantalla del dispositivo ni será capaz de leer las teclas que pulse el usuario.

Podemos utilizar este objeto para obtener propiedades del visor como el número de colores que soporta:

```
boolean color = mi_display.isColor();  
int num_color = mi_display.numColors();
```

5.2. Componentes disponibles

Una vez hemos accedido al *display*, deberemos mostrar algo en él. Tenemos una serie de elementos que podemos mostrar en el *display*, estos son conocidos como elementos *displayables*.

En el *display* podremos mostrar a lo sumo un elemento *displayable*. Para obtener el elemento que se está mostrando actualmente en el visor utilizaremos el siguiente método:

```
Displayable elemento = mi_display.getCurrent();
```

Nos devolverá el objeto *Displayable* correspondiente al objeto que se está mostrando en la pantalla, o *null* en el caso de que no se esté mostrando ningún elemento. Esto ocurrirá al comienzo de la ejecución de la aplicación cuando todavía no se ha asignado ningún elemento al *Display*. Podemos establecer el elemento que queremos mostrar en pantalla con:

```
mi_display.setCurrent(nuevo_elemento);
```

Como sólo podemos mostrar simultáneamente un elemento *displayable* en el *display*, este elemento ocupará todo el visor. Además será este elemento el que recibirá la entrada del usuario.

Entre estos elementos *displayables* podemos distinguir una API de bajo nivel, y una API de alto nivel.

5.2.1. API de alto nivel

Consiste en una serie de elementos predefinidos: *Form*, *List*, *Alert* y *TextBox* que son extensiones de la clase abstracta *Screen*. Estos son elementos comunes que podemos encontrar en la interfaz de todos los dispositivos, por lo que el tenerlos predefinidos nos permitirá utilizarlos de forma sencilla sin tenerlos que crear nosotros a mano en nuestras aplicaciones. Se implementan de forma nativa por cada dispositivo concreto, por lo que pueden variar de unos dispositivos a otros. Estos componentes hacen que las aplicaciones sean más sencillas y portables, pero nos limita a una serie de controles predefinidos.

Este tipo de componentes serán adecuados para realizar *front-ends* de aplicaciones corporativas. De esta forma obtendremos aplicaciones totalmente portables, en las que la implementación nativa será la que se deberá encargar de dibujar estos componentes. Por lo tanto, en cada dispositivo podrán mostrarse de una forma distinta. Además no se permitirá acceder directamente a los eventos de entrada del teclado.

5.2.2. API de bajo nivel

Consiste en la clase *Canvas*, que nos permitirá dibujar lo que queramos en la pantalla. Tendremos que dibujarlo todo nosotros a mano. Esto nos permitirá tener un mayor control sobre lo que dibujamos, y podremos recibir eventos del teclado a bajo nivel. Esto provocará que las aplicaciones sean menos portables. Esta API será conveniente para las aplicaciones que necesitan tener control total sobre lo que se dibuja y sobre la entrada, como por ejemplo los juegos.

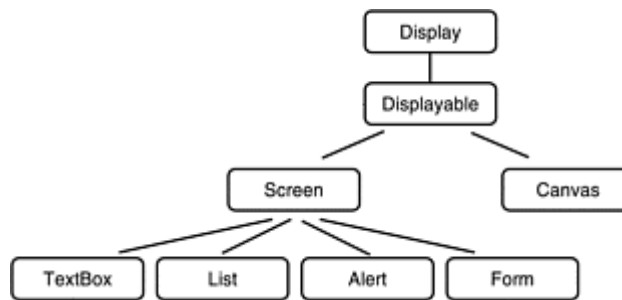


Figura 1. Jerarquía de elementos displayables

5.3. Componentes de alto nivel

Todos los componentes de alto nivel derivan de la clase `Screen`. Se llama así debido a que cada uno de estos componentes será una pantalla de nuestra aplicación, ya que no puede haber más de un componente en la pantalla al mismo tiempo. Esta clase contiene las propiedades comunes a todos los elementos de alto nivel:

Título: Es el título que se mostrará en la pantalla correspondiente al componente. Podemos leer o asignar el título con los métodos:

```
String titulo = componente.getTitle();  
componente.setTitle(titulo);
```

Ticker: Podemos mostrar un *ticker* en la pantalla. El *ticker* consiste en un texto que irá desplazándose de derecha a izquierda. Podemos asignar o obtener el *ticker* con:

```
Ticker ticker = componente.getTicker();  
componente.setTicker(ticker);
```

A continuación podemos ver cómo se muestra el título y el *ticker* en distintos modelos de móviles:



Figura 2. Título y ticker de las pantallas

Los componentes de alto nivel disponibles son cuadros de texto (`TextBox`), listas (`List`), formularios (`Form`) y alertas (`Alert`).

5.3.1. Cuadros de texto

Este componente muestra un cuadro donde el usuario puede introducir texto. La forma en la que se introduce el texto es dependiente del dispositivo. Por ejemplo, los teléfonos que soporten texto predictivo podrán introducir texto de esta forma. Esto se hace de forma totalmente nativa, por lo que desde Java no podremos modificar este método de introducción del texto.

Para crear un campo de texto deberemos crear un objeto de la clase `TextBox`, utilizando el siguiente constructor:

```
TextBox tb = new TextBox(titulo, texto, capacidad,
restricciones);
```

Donde `titulo` será el título que se mostrará en la pantalla, `texto` será el texto que se muestre inicialmente dentro del cuadro, y `capacidad` será el número de caracteres máximo que puede tener el texto. Además podemos añadir una serie de restricciones, definidas como constantes de la clase `TextField`, que limitarán el tipo de texto que se permita escribir en el cuadro. Puede tomar los siguientes valores:

<code>TextField.ANY</code>	Cualquier texto
<code>TextField.NUMERIC</code>	Números enteros
<code>TextField.PHONENUMBER</code>	Números de teléfono
<code>TextField.EMAILADDR</code>	Direcciones de e-mail
<code>TextField.URL</code>	URLs
<code>TextField.PASSWORD</code>	Se ocultan los caracteres escritos utilizando, por ejemplo utilizando asteriscos (*). Puede combinarse con los valores anteriores utilizando el operador OR (<code> </code>).

Una vez creado, para que se muestre en la pantalla debemos establecerlo como el componente actual del *display*:

```
mi_display.setCurrent(tb);
```

Una vez hecho esto este será el componente que se muestre en el *display*, y el que recibirá los eventos y comandos de entrada, de forma que cuando el usuario escriba utilizando el teclado del móvil estará escribiendo en este cuadro de texto.

Podemos obtener el texto que haya escrito el usuario en este cuadro de texto utilizando el método:

```
String texto = tb.getString();
```

Esto lo haremos cuando ocurra un determinado evento, que nos indique que el usuario ya ha introducido el texto, como por ejemplo cuando pulse sobre la opción OK.

Además tiene métodos con los que podremos modificar el contenido del cuadro de texto, insertando, modificando o borrando caracteres o bien cambiando todo el texto, así como para obtener información sobre el mismo, como el número de caracteres que se han escrito, la capacidad máxima o las restricciones impuestas.

Por ejemplo, podemos crear y mostrar un campo de texto para introducir una contraseña de 8 caracteres de la siguiente forma:

```
TextBox tb = new TextBox("Contraseña", "", 8,  
                        TextField.ANY | TextField.PASSWORD);  
Display d = Display.getDisplay(this);  
d.setCurrent(tb);
```

El aspecto que mostrará esta pantalla en distintos modelos de móviles será el siguiente:



Figura 3. Cuadros de texto

5.3.2. Listas

Este componente muestra una lista de elementos en la pantalla. Las listas pueden ser de distintos tipos:

- **Implícita:** Este tipo de listas nos servirán por ejemplo para hacer menús. Cuando pulsemos sobre un elemento de la lista se le notificará inmediatamente a la aplicación el elemento sobre el que hemos pulsado, para que ésta pueda realizar la acción correspondiente.
- **Exclusiva:** A diferencia de la anterior, en esta lista cuando se pulsa sobre un elemento no se notifica a la aplicación, sino que simplemente lo que hace es marcar el elemento como seleccionado. En esta lista podremos tener sólo un elemento marcado, si previamente ya tuviésemos uno marcado, cuando pulsemos sobre uno nuevo se desmarcará el anterior.

- **Múltiple:** Es similar a la exclusiva, pero podemos marcar varios elementos simultáneamente. Pulsando sobre un elemento lo marcaremos o lo desmarcaremos, pudiendo de esta forma marcar tantos como queramos.

Las listas se definen mediante la clase `List`, y para crear una lista podemos utilizar el siguiente constructor:

```
List l = new List(titulo, tipo);
```

Donde `titulo` será el título de la pantalla correspondiente a nuestra lista, y `tipo` será uno de los tipos vistos anteriormente, definidos como constantes de la clase `Choice`:

<code>Choice.IMPLICIT</code>	Lista implícita
<code>Choice.EXCLUSIVE</code>	Lista exclusiva
<code>Choice.MULTIPLE</code>	Lista múltiple

También tenemos otro constructor en el que podemos especificar un *array* de elementos a mostrar en la lista, para añadir toda esa lista de elementos en el momento de su construcción. Si no lo hacemos en este momento, podremos añadir elementos posteriormente utilizando el método:

```
l.append(texto, imagen);
```

Donde `texto` será la cadena de texto que se muestre, e `imagen` será una imagen que podremos poner a dicho elemento de la lista de forma opcional. Si no queremos poner ninguna imagen podemos especificar `null`.

Podremos conocer desde el código los elementos que están marcados en la lista en un momento dado. También tendremos métodos para insertar, modificar o borrar elementos de la lista, así como para marcarlos o desmarcarlos.

Por ejemplo, podemos crear un menú para nuestra aplicación de la siguiente forma:

```
List l = new List("Menu", Choice.IMPLICIT);  
l.append("Nuevo juego", null);  
l.append("Continuar", null);  
l.append("Instrucciones", null);  
l.append("Hi-score", null);  
l.append("Salir", null);  
Display d = Display.getDisplay(this);  
d.setCurrent(l);
```

A continuación se muestra el aspecto de los distintos tipos de listas existentes:



Figura 4. Tipos de listas

5.3.3. Formularios

Este componente es más complejo, permitiéndonos mostrar varios elementos en una misma pantalla. Los formularios se encapsulan en la clase `Form`, y los elementos que podemos incluir en ellos son todos derivados de la clase `Item`. Tenemos disponibles los siguientes elementos:

- **Etiquetas** (`StringItem`): Muestra una etiqueta de texto estático, es decir, que no podrá ser modificado por el usuario. Se compone de un título del campo y de un texto como contenido.
- **Imágenes** (`ImageItem`): Muestra una imagen en el formulario. Esta imagen también es estática. Se compone de un título, la imagen, y un texto alternativo en el caso de que el dispositivo no pueda mostrar imágenes.
- **Campo de texto** (`TextField`): Muestra un cuadro donde el usuario podrá introducir texto. Se trabaja con él de forma similar al componente `TextBox` visto anteriormente.
- **Campo de fecha** (`DateField`): Permite al usuario introducir una fecha. La forma de introducir la fecha variará de un modelo de móvil a otro. Por ejemplo, puede introducirse directamente introduciendo numéricamente la fecha, o mostrar un calendario donde el usuario pueda seleccionar el día.
- **Cuadro de opciones** (`ChoiceGroup`): Muestra un grupo de opciones para que el usuario marque una o varias de ellas. Se trabaja con él de forma similar al componente `List` visto anteriormente, pudiendo en este caso ser de tipo exclusivo o múltiple.
- **Barra de nivel** (`Gauge`): Muestra una barra para seleccionar un nivel, como por ejemplo podría ser el nivel de volumen. Cada posición de esta barra corresponderá a un valor entero. Este valor irá de cero a un valor máximo que podremos especificar nosotros. La barra podrá ser interactiva o fija.

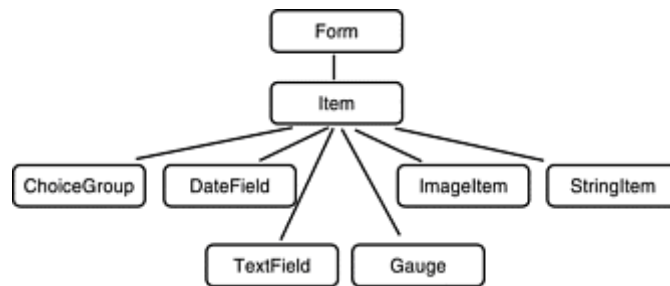


Figura 5. Jerarquía de los elementos de los formularios

Para crear el formulario podemos utilizar el siguiente constructor, en el que especificamos el título de la pantalla:

```
Form f = new Form(titulo);
```

También podemos crear el formulario proporcionando el *array* de elementos (items) que tiene en el constructor. Si no lo hemos hecho en el constructor, podemos añadir items al formulario con:

```
f.append(item);
```

Podremos añadir como item o bien cualquiera de los items vistos anteriormente, derivados de la clase `Item`, o una cadena de texto o una imagen. También podremos insertar, modificar o borrar los items del formulario.

A continuación mostramos un ejemplo de formulario:

```

Form f = new Form("Formulario");

Item itemEtiqueta = new StringItem("Etiqueta:",
                                   "Texto de la etiqueta");
Item itemTexto = new TextField("Telefono:", "", 8,
                               TextField.PHONENUMBER);
Item itemFecha = new DateField("Fecha", DateField.DATE_TIME);
Item itemBarra = new Gauge("Volumen", true, 10, 8);
ChoiceGroup itemOpcion = new ChoiceGroup("Opcion",
                                         Choice.EXCLUSIVE);

itemOpcion.append("Si", null);
itemOpcion.append("No", null);

f.append(itemEtiqueta);
f.append(itemTexto);
f.append(itemFecha);
f.append(itemBarra);
f.append(itemOpcion);

Display d = Display.getDisplay(this);
d.setCurrent(f);
  
```

El aspecto de este formulario es el siguiente:



Figura 6. Aspecto de los formularios

En MIDP 2.0 aparecen dos nuevos tipos de items que podremos añadir a los formularios. Estos items son:

- **Spacer:** Se trata de un item vacío, al que se le asigna un tamaño mínimo, que nos servirá para introducir un espacio en blanco en el formulario. El item tendrá una altura y anchura mínima, y al insertarlo en el formulario se creará un espacio en blanco con este tamaño. El siguiente item que añadamos se posicionará después de este espacio.
- **CustomItem:** Este es un item personalizable, en el que podremos definir totalmente su aspecto y su forma de interactuar con el usuario. La forma en la que se definen estos items es similar a la forma en la que se define el `Canvas`, que estudiaremos en temas posteriores. Al igual que el `Canvas`, este componente pertenece a la API de bajo nivel, ya que permite al usuario dibujar los gráficos y leer la entrada del usuario a bajo nivel.

5.3.4. Alertas

Las alertas son un tipo especial de pantallas, que servirán normalmente de transición entre dos pantallas. En ellas normalmente se muestra un mensaje de información, error o advertencia y se pasa automáticamente a la siguiente pantalla.

Las alertas se encapsulan en la clase `Alert`, y se crearán normalmente con el siguiente constructor:

```
Alert a = new Alert(titulo, texto, imagen, tipo);
```

Donde `titulo` es el título de la pantalla y `texto` será el texto que se muestre en la alerta. Podemos mostrar una imagen de forma opcional. Si no queremos usar ninguna imagen pondremos `null` en el campo correspondiente. Además debemos dar un tipo de alerta. Estos tipos se definen como constantes de la clase `AlertType`:

<code>AlertType.ERROR</code>	Muestran un mensaje de error de la aplicación.
<code>AlertType.WARNING</code>	Muestran un mensaje de advertencia.
<code>AlertType.INFO</code>	Muestran un mensaje de información.
<code>AlertType.CONFIRMATION</code>	Muestran un mensaje de confirmación de alguna acción realizada.
<code>AlertType.ALARM</code>	Notifican de un evento en el que está interesado el usuario.

A estas alertas se les puede asignar un tiempo límite (*timeout*), de forma que transcurrido este tiempo desde que se mostró la alerta se pase automáticamente a la siguiente pantalla.

Para mostrar una alerta lo haremos de forma distinta a los componentes que hemos visto anteriormente. En este caso utilizaremos el siguiente método:

```
mi_display.setCurrent(alerta, siguiente_pantalla);
```

Debemos especificar además de la alerta, la siguiente pantalla a la que iremos tras mostrar la alerta, ya que como hemos dicho anteriormente la alerta es sólo una pantalla de transición.

Por ejemplo, podemos crear una alerta que muestre un mensaje de error al usuario y que vuelva a la misma pantalla en la que estamos:

```
Alert a = new Alert("Error", "No hay ninguna nota  
seleccionada",  
                    null, AlertType.ERROR);  
Display d = Display.getDisplay(midlet);  
d.setCurrent(a, d.getCurrent());
```

A continuación podemos ver dos alertas distintas, mostrando mensajes de error, de información y de alarma respectivamente:



Figura 7. Alertas de error, de información y de alarma

Puede ser interesante combinar las alertas con temporizadores para implementar agendas en las que el móvil nos recuerde diferentes eventos

mostrando una alerta a una hora determinada, o hacer sonar una alarma, ya que estas alertas nos permiten incorporar sonido.

Por ejemplo podemos implementar una tarea que dispare una alarma, mostrando una alerta y reproduciendo sonido. La tarea puede contener el siguiente código:

```
class Alarma extends TimerTask {  
    public void run() {  
        Alert a = new Alert("Alarma",  
            "Se ha disparado la alarma", null,  
            AlertType.ALARM);  
        a.setTimeout(Alert.FOREVER);  
  
        Display d = Display.getDisplay(midlet);  
        AlertType.ALARM.playSound(d);  
  
        d.setCurrent(a, d.getCurrent());  
    }  
}
```

Una vez definida la tarea que implementa la alarma, podemos utilizar un temporizador para planificar el comienzo de la alarma a una hora determinada:

```
Timer temp = new Timer();  
Alarma a = new Alarma();  
temp.schedule(a, tiempo);
```

Como tiempo de comienzo podremos especificar un retardo en milisegundos o una hora absoluta a la que queremos que se dispare la alarma.

5.4. Imágenes

En muchos de los componentes anteriores hemos visto que podemos incorporar imágenes. Estas imágenes se encapsularán en la clase `Image`, que contendrá el *raster* (matriz de *pixels*) correspondiente a dicha imagen en memoria. Según si este *raster* puede ser modificado o no, podemos clasificar las imágenes en mutables e inmutables.

5.4.1. Imágenes mutables

Nos permitirán modificar su contenido dentro del código de nuestra aplicación. En la API de interfaz gráfica de bajo nivel veremos cómo modificar estas imágenes. Las imágenes mutables se crean como una imagen en blanco con unas determinadas dimensiones, utilizando el siguiente método:

```
Image img = Image.createImage(ancho, alto);
```

Nada más crearla estará vacía. A partir de este momento podremos dibujar en ella cualquier contenido, utilizando la API de bajo nivel.

5.4.2. Imágenes inmutables

Las imágenes inmutables una vez creadas no pueden ser modificadas. Las imágenes que nos permiten añadir los componentes de alto nivel vistos previamente deben ser inmutables, ya que estos componentes no están preparados para que la imagen pueda cambiar en cualquier momento.

Para crear una imagen inmutable deberemos proporcionar el contenido de la imagen en el momento de su creación, ya que no se podrá modificar más adelante. Lo normal será utilizar ficheros de imágenes. Las aplicaciones MIDP soportan el formato PNG, por lo que deberemos utilizar este formato.

- Carga de imágenes desde recursos:

Podemos cargar una imagen de un fichero PNG incluido dentro del JAR de nuestra aplicación utilizando el siguiente método:

```
Image img = Image.createImage(nombre_fichero);
```

De esta forma buscará dentro del JAR un recurso con el nombre que hayamos proporcionado, utilizando internamente el método `Class.getResourceAsStream` que vimos en el capítulo anterior.

NOTA: Las imágenes son el único tipo de recurso que proporcionan su propio método para cargarlas desde un fichero dentro del JAR. Para cualquier otro tipo de recurso, como por ejemplo ficheros de texto, deberemos utilizar `Class.getResourceAsStream` para abrir un flujo de entrada que lea de él y leerlo manualmente.

- Carga desde otra ubicación:

Si la imagen no está dentro del fichero JAR, como por ejemplo en el caso de que queramos leerla de la web, no podremos utilizar el método anterior. Encontramos un método más genérico para la creación de una imagen inmutable que crea la imagen a partir de la secuencia de *bytes* del fichero PNG de la misma:

```
Image img = Image.createImage(datos, offset, longitud);
```

Donde `datos` es un *array* de *bytes*, `offset` la posición del *array* donde comienza la imagen, y `longitud` el número de *bytes* que ocupa la imagen.

Por ejemplo, si queremos cargar una imagen desde la red podemos hacer lo siguiente:

```
// Abre una conexion en red con la URL de la imagen
String url = "http://j2ee.ua.es/imagenes/logo.png";
URLConnection con =
    (URLConnection) Connector.open(url);
InputStream in = con.getInputStream();
```

```
// Lee bytes de la imagen
int c;
ByteArrayOutputStream baos = new
ByteArrayOutputStream();
while( (c=in.read()) != -1 ) {
    baos.write(c);
}

// Crea imagen a partir de array de bytes
byte [] datos = baos.toByteArray();
Image img = Image.createImage(datos,0,datos.length);
```

- Conversión de mutable a inmutable:

Es posible que queramos mostrar una imagen que hemos modificado desde dentro de nuestro programa en alguno de los componentes de alto nivel anteriores. Sin embargo ya hemos visto que sólo se pueden mostrar en estos componentes imágenes inmutables.

Lo que podemos hacer es convertir la imagen de mutable a inmutable, de forma que crearemos una versión no modificable de nuestra imagen mutable, que pueda ser utilizada en estos componentes. Si hemos creado una imagen mutable `img_mutable`, podemos crear una versión inmutable de esta imagen de la siguiente forma:

```
Image img_inmutable = Image.createImage(img_mutable);
```

Una vez tenemos creada la imagen inmutable, podremos mostrarla en distintos componentes de alto nivel, como alertas, listas y algunos ítems dentro de los formularios (cuadro de opciones e ítem de tipo imagen).

En las alertas, listas y cuadros de opciones de los formularios simplemente especificaremos la imagen que queremos mostrar, y éste se mostrará en la pantalla de alerta o junto a uno de los elementos de la lista. En los ítems de tipo imagen (`ImageItem`) de los formularios, podremos controlar la disposición (*layout*) de la imagen, permitiéndonos por ejemplo mostrarla centrada, a la izquierda o a la derecha.

5.5. Comandos de entrada

Hemos visto como crear una serie de componentes de alto nivel para mostrar en nuestra aplicación. Sin embargo no hemos visto como interactuar con las acciones que realice el usuario, para poderles dar una respuesta desde nuestra aplicación.

En estos componentes de alto nivel el usuario podrá interactuar mediante una serie de comandos que podrá ejecutar. Para cada pantalla podremos definir una lista de comandos, de forma que el usuario pueda seleccionar y ejecutar uno de ellos. Esta es una forma de interacción de alto nivel, que se implementará a nivel nativo y que será totalmente portable.

En el móvil estos comandos se encontrarán normalmente en una o en las dos esquinas inferiores, y se podrán activar pulsando sobre el botón situado justo bajo dicha esquina:

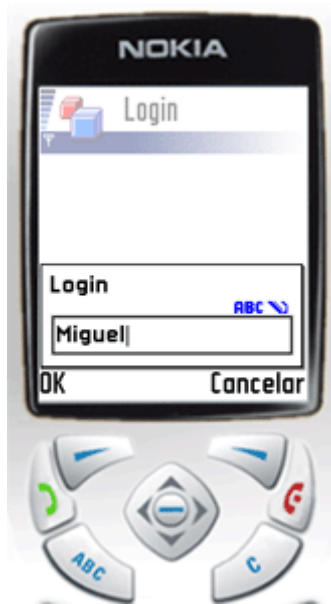


Figura 8. Comandos de las pantallas

Según el dispositivo tendremos uno o dos botones de este tipo. Si tenemos varios comandos, al pulsar sobre el botón de la esquina correspondiente se abrirá un menú con todos los comandos disponibles para seleccionar uno de ellos.

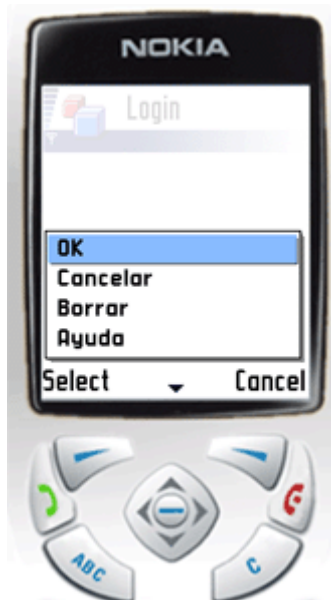


Figura 9. Despliegue del menú de comandos

5.5.1. Creación de comandos

Estos comandos se definen mediante la clase `Command`, y pueden ser creados utilizando el siguiente constructor:

```
Command c = new Command(etiqueta, tipo, prioridad);
```

En etiqueta especificaremos el texto que se mostrará en el comando. Los otros dos parámetros se utilizarán para mejorar la portabilidad entre dispositivos. En tipo podremos definir el tipo del comando, pudiendo ser:

<code>Command.OK</code>	Dar una respuesta positiva
<code>Command.BACK</code>	Volver a la pantalla anterior
<code>Command.CANCEL</code>	Dar una respuesta negativa
<code>Command.EXIT</code>	Salir de la aplicación
<code>Command.HELP</code>	Mostrar una pantalla de ayuda
<code>Command.STOP</code>	Detener algún proceso que se esté realizando
<code>Command.SCREEN</code>	Comando propio de nuestra aplicación para la pantalla actual.
<code>Command.ITEM</code>	Comando específico para ser aplicado al ítem seleccionado actualmente. De esta forma se comportará como un menú contextual.

El asignar uno de estos tipos no servirá para que el comando realice una de estas acciones. Las acciones que se realicen al ejecutar el comando las deberemos implementar siempre nosotros. El asignar estos tipos simplemente sirve para que la implementación nativa del dispositivo conozca qué función desempeña cada comando, de forma que los sitúe en el lugar adecuado para dicho dispositivo. Cada dispositivo podrá distribuir los distintos tipos de comandos utilizando diferentes criterios.

Por ejemplo, si en nuestro dispositivo la acción de volver atrás suele asignarse al botón de la esquina derecha, si añadimos un comando de este tipo intentará situarlo en este lugar.

Además les daremos una prioridad con la que establecemos la importancia de los comandos. Esta prioridad es un valor entero, que cuanto menor sea más importancia tendrá el comando. Un comando con prioridad `1` tiene importancia máxima. Primero situará los comandos utilizando el tipo como criterio, y para los comandos con el mismo tipo utilizará la prioridad para poner más accesibles aquellos con mayor prioridad.

Una vez hemos creado los comandos, podemos añadirlos a la pantalla actual utilizando el método:

```
pantalla.addCommand(c);
```

Esta pantalla podrá ser cualquier elemento *displayable* de los que hemos visto anteriormente excepto `Alarm`, ya que no está permitido añadir comandos a las alarmas. De esta forma añadiremos todos los comandos necesarios.

Por ejemplo, podemos añadir una serie de comandos a la pantalla de *login* de nuestra aplicación de la siguiente forma:

```
TextBox tb = new TextBox("Login", "", 8, TextField.ANY);

Command cmdOK = new Command("OK", Command.OK, 1);
Command cmdAyuda = new Command("Ayuda", Command.HELP, 1);
Command cmdSalir = new Command("Salir", Command.EXIT, 1);
Command cmdBorrar = new Command("Borrar", Command.SCREEN, 1);
Command cmdCancelar = new Command("Cancelar", Command.CANCEL, 1);

tb.addCommand(cmdOK);
tb.addCommand(cmdAyuda);
tb.addCommand(cmdSalir);
tb.addCommand(cmdBorrar);
tb.addCommand(cmdCancelar);

Display d = Display.getDisplay(this);
d.setCurrent(tb);
```

5.5.2. Listener de comandos

Una vez añadidos los comandos a la pantalla, deberemos definir el código para dar respuesta a cada uno de ellos. Para ello deberemos crear un *listener*, que es un objeto que escucha las acciones del usuario para darles una respuesta.

El *listener* será una clase en la que introduciremos el código que queremos que se ejecute cuando el usuario selecciona uno de los comandos. Cuando se pulse sobre uno de estos comandos, se invocará dicho código.

Para crear el *listener* debemos crear una clase que implemente la interfaz `commandListener`. El implementar esta interfaz nos obligará a definir el método `commandAction`, que será donde deberemos introducir el código que dé respuesta al evento de selección de un comando.

```
class MiListener implements CommandListener {

    public void commandAction(Command c, Displayable d) {

        // Código de respuesta al comando

    }

}
```

Cuando se produzca un evento de este tipo, conoceremos qué comando se ha seleccionado y en qué *displayable* estaba, ya que esta información se proporciona como parámetros. Según el comando que se haya ejecutado, dentro de este método deberemos decidir qué acción realizar.

Por ejemplo, podemos crear un listener para los comandos añadidos a la pantalla de *login* del ejemplo anterior:

```
class ListenerLogin implements CommandListener {

    public void commandAction(Command c, Displayable d) {

        if(c == cmdOK) {
            // Aceptar
        } else if(c == cmdCancelar) {
            // Cancelar
        } else if(c == cmdSalir) {
            // Salir
        } else if(c == cmdAyuda) {
            // Ayuda
        } else if(c == cmdBorrar) {
            // Borrar
        }
    }
}
```

Una vez creado el *listener* tendremos registrarlo en el *displayable* que contiene los comandos para ser notificado de los comandos que ejecute el usuario. Para establecerlo como *listener* utilizaremos el método `setCommandListener` del *displayable*.

Por ejemplo, en el caso del campo de texto de la pantalla de *login* lo registraremos de la siguiente forma:

```
tb.setCommandListener(new ListenerLogin());
```

Una vez hecho esto, cada vez que el usuario ejecute un comando se invocará el método `commandAction` del *listener* que hemos definido, indicándonos el comando que se ha invocado.

5.5.3. Listas implícitas

En las listas implícitas dijimos que cuando se pulsa sobre un elemento de la lista se notifica inmediatamente a la aplicación para que se realice la acción correspondiente, de forma que se comporta como un menú.

La forma que tiene de notificarse la selección de un elemento de este tipo de listas es invocando un comando. En este caso se invocará un tipo especial de comando definido como constante en la clase `List`, se trata de `List.SELECT_COMMAND`.

Dentro de `commandAction` podemos comprobar si se ha ejecutado un comando de este tipo para saber si se ha seleccionado un elemento de la lista. En este caso, podremos saber el elemento del que se trata viendo el índice que se ha seleccionado:

```
class ListenerLogin implements CommandListener {

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

```
if(c == List.SELECT_COMMAND) {  
    int indice = l.getSelectedIndex();  
    if(indice == 0) {  
        // Nuevo juego  
    } else if(indice == 1) {  
        // Continuar  
    } else if(indice == 2) {  
        // Instrucciones  
    } else if(indice == 3) {  
        // Hi-score  
    } else if(indice == 4) {  
        // Salir  
    }  
}  
}  
}
```

5.5.4. Listener de items

En el caso de los formularios, podremos tener constancia de cualquier cambio que el usuario haya introducido en alguno de sus campos antes de que se ejecute algún comando para realizar alguna acción.

Por ejemplo, esto nos puede servir para validar los datos introducidos. En el momento que el usuario cambie algún campo, se nos notificará dicho cambio pudiendo comprobar de esta forma si el valor introducido es correcto o no. Además, de esta forma sabremos si ha habido cambios, por lo que podremos volver a grabar los datos del formulario de forma persistente sólo en caso necesario.

Para recibir la notificación de cambio de algún item del formulario, utilizaremos un *listener* de tipo `ItemStateListener`, en el que deberemos definir el método `itemStateChanged` donde introduciremos el código a ejecutar en caso de que el usuario modifique alguno de los campos modificables (cuadros de opciones, campo de texto, campo de fecha o barra de nivel). El esqueleto de un *listener* de este tipo será el siguiente:

```
class MiListener implements ItemStateListener {  
    public void itemStateChanged(Item i) {  
        // Se ha modificado el item i  
    }  
}
```

5.6. Transiciones entre pantallas

Hemos visto que cada uno de los componentes *displayables* que tenemos disponibles representa una pantalla, y podemos cambiar esta pantalla utilizando el método `setCurrent` del *display*.

De esta forma podremos pasar de una pantalla a otra de la aplicación cuando ocurra un determinado evento, como puede ser por ejemplo que el usuario

ejecute un determinado comando o que se ejecute alguna tarea planificada por un temporizador.

Cuando tengamos una aplicación con un número elevado de pantallas, será recomendable hacer previamente un diseño de esta aplicación. Definiremos un diagrama de navegación, en el que cada bloque representará una pantalla, y las flechas que unen dichos bloques serán las transiciones entre pantallas.

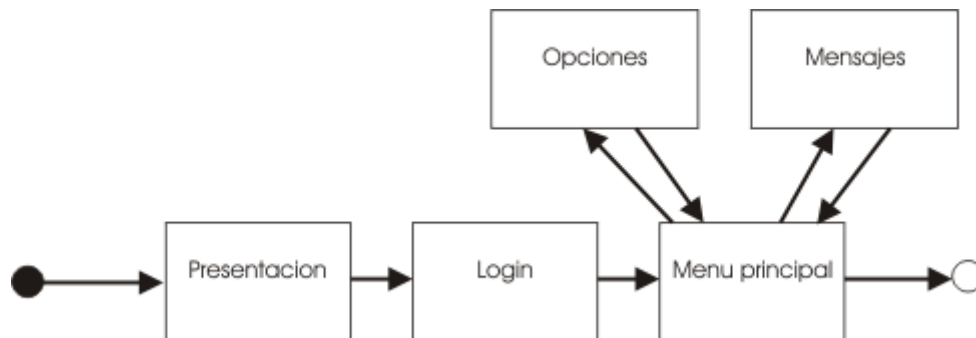


Figura 10. Mapa de pantallas

Debemos asegurarnos en este mapa de pantallas que el usuario en todo momento puede volver atrás y que hemos definido todos los enlaces necesarios para acceder a todas las pantallas de la aplicación.

5.6.1. Vuelta atrás

Normalmente las aplicaciones tendrán una opción que nos permitirá volver a la pantalla visitada anteriormente. Para implementar esto podemos utilizar una pila (*Stack*), en la que iremos apilando todas las pantallas conforme las visitamos. Cuando pulsemos el botón para ir atrás desapilaremos la ultima pantalla y la mostraremos en el *display* utilizando `setCurrent`.

5.6.2. Diseño de pantallas

Es conveniente tomar algún determinado patrón de diseño para implementar las pantallas de nuestra aplicación. Podemos crear una clase por cada pantalla, donde encapsularemos todo el contenido que se debe mostrar en la pantalla, los comandos disponibles, y los *listeners* que den respuesta a estos comandos.

Las clases implementadas según este patrón de diseño cumplirán lo siguiente:

- Heredan del tipo de *displayable* al que pertenecen. De esta forma nuestra pantalla será un tipo especializado de este *displayable*.
- Implementan la interfaz `CommandListener` para encapsular la respuesta a los comandos. Esto nos forzará a definir dentro de esta clase el método `commandAction` para dar respuesta a los comandos. Podemos implementar también la interfaz `ItemStateListener` en caso necesario.
- Al constructor se le proporciona como parámetro el `MIDlet` de la aplicación, además de cualquier otro parámetro que necesitemos añadir.

Esto será necesario para poder obtener una referencia al *display*, y de esa forma poder provocar la transición a otra pantalla. Así podremos hacer que sea dentro de la clase de cada pantalla donde se definan las posibles transiciones a otras pantallas.

Por ejemplo, podemos implementar el menú principal de nuestra aplicación de la siguiente forma:

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

public class MenuPrincipal extends List implements
CommandListener {

    MiMIDlet owner;
    Command selec;
    int itemNuevo;
    int itemSalir;

    public MenuPrincipal(MiMIDlet owner) {
        super("Menu", List.IMPLICIT);
        this.owner = owner;

        // Añade opciones al menu
        itemNuevo = this.append("Nuevo juego", null);
        itemSalir = this.append("Salir", null);

        // Crea comandos
        selec = new Command("Seleccionar", Command.SCREEN, 1);
        this.addCommand(selec);
        this.setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d) {
        if(c == selec || c == List.SELECT_COMMAND) {
            if(getSelectedIndex() == itemNuevo) {
                // Nuevo juego
                Display display = Display.getDisplay(owner);
                PantallaJuego pj =
                    new PantallaJuego(owner, this);
                display.setCurrent(pj);
            } else if(getSelectedIndex() == itemSalir) {
                // Salir de la aplicación
                owner.salir();
            }
        }
    }
}
```

Si esta es la pantalla principal de nuestra aplicación, la podremos mostrar desde nuestro MIDlet de la siguiente forma:

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

public class MiMIDlet extends MIDlet {
    protected void startApp()
        throws MIDletStateChangeException {
        Display d = Display.getDisplay(this);
    }
}
```

```
        MenuPrincipal mp = new MenuPrincipal(this);
        d.setCurrent(mp);
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean incondicional)
        throws MIDletStateChangeException {
    }

    public void salir() {
        try {
            destroyApp(false);
            notifyDestroyed();
        } catch (MIDletStateChangeException e) {
            // Evitamos salir de la aplicacion
        }
    }
}
```

Este patrón de diseño encapsula el comportamiento de cada pantalla en clases independientes, lo cual hará más legible y reutilizable el código.

Con este diseño, si queremos permitir volver a una pantalla anterior podemos pasar como parámetro del constructor, además del MIDlet, el elemento *displayable* correspondiente a esta pantalla anterior. De esta forma cuando pulsemos *Atrás* sólo tendremos que mostrar este elemento en el *display*.

6. Gráficos avanzados

Hasta ahora hemos visto la creación de aplicaciones con una interfaz gráfica creada a partir de una serie de componentes de alto nivel definidos en la API LCDUI (alertas, campos de texto, listas, formularios).

En este punto veremos como dibujar nuestros propios gráficos directamente en pantalla. Para ello Java nos proporciona acceso a bajo nivel al contexto gráfico del área donde vayamos a dibujar, permitiéndonos a través de éste modificar los *pixels* de este área, dibujar una serie de figuras geométricas, así como volcar imágenes en ella.

También podremos acceder a la entrada del usuario a bajo nivel, conociendo en todo momento cuándo el usuario pulsa o suelta cualquier tecla del móvil.

Este acceso a bajo nivel será necesario en aplicaciones como juegos, donde debemos tener un control absoluto sobre la entrada y sobre lo que dibujamos en pantalla en cada momento. El tener este mayor control tiene el inconveniente de que las aplicaciones serán menos portables, ya que dibujaremos los gráficos pensando en una determinada resolución de pantalla y un determinado tipo de teclado, pero cuando la queramos llevar a otro dispositivo en el que estos componentes sean distintos deberemos hacer cambios en el código.

Por esta razón para las aplicaciones que utilizan esta API a bajo nivel, como los juegos Java para móviles, encontramos distintas versiones para cada modelo de dispositivo. Dada la heterogeneidad de estos dispositivos, resulta más sencillo rehacer la aplicación para cada modelo distinto que realizar una aplicación adaptable a las características de cada modelo.

Al programar las aplicaciones deberemos facilitar en la medida de lo posible futuros cambios para adaptarla a otros modelos, permitiendo reutilizar la máxima cantidad de código posible.

6.1. Gráficos en LCDUI

La API de gráficos a bajo nivel de LCDUI es muy parecida a la existente en AWT, por lo que el aprendizaje de esta API para programadores que conozcan la de AWT va a ser casi inmediato.

Las clases que implementan la API de bajo nivel en LCDUI son `Canvas` y `Graphics`. Estas clases reciben el mismo nombre que las de AWT, y se utilizan de una forma muy parecida. Tienen alguna diferencia en cuanto a su interfaz para adaptarse a las necesidades de los dispositivos móviles.

El `Canvas` es un tipo de elemento *displayable* correspondiente a una pantalla vacía en la que nosotros podremos dibujar a bajo nivel el contenido que queramos. Además este componente nos permitirá leer los eventos de entrada del usuario a bajo nivel.

Esta pantalla del móvil tiene un contexto gráfico asociado que nosotros podremos utilizar para dibujar en ella. Este objeto encapsula el *raster* de pantalla (la matriz de *pixels* de los que se compone la pantalla) y además tiene una serie de atributos con los que podremos modificar la forma en la que se dibuja en este *raster*. Este contexto gráfico se definirá en un objeto de la clase `Graphics`. Este objeto nos ofrece una serie de métodos que nos permiten dibujar distintos elementos en pantalla. Más adelante veremos con detalle los métodos más importantes.

Este objeto `Graphics` para dibujar en la pantalla del dispositivo nos lo deberá proporcionar el sistema en el momento en que se vaya a dibujar, no podremos obtenerlo nosotros por nuestra cuenta de ninguna otra forma. Esto es lo que se conoce como *render* pasivo, definimos la forma en la que se dibuja pero es el sistema el que decidirá cuándo hacerlo.

6.1.1. Creación de un Canvas

Para definir la forma en la que se va a dibujar nuestro componente deberemos extender la clase `Canvas` redefiniendo su método `paint`. Dentro de este método es donde definiremos cómo se realiza el dibujo de la pantalla. Esto lo haremos de la siguiente forma:

```
public class MiCanvas extends Canvas {  
    public void paint(Graphics g) {  
        // Dibujamos en la pantalla  
        // usando el objeto g proporcionado  
    }  
}
```

Con esto en la clase `MiCanvas` hemos creado una pantalla en la que nosotros controlamos lo que se dibuja. Este método `paint` nunca debemos invocarlo nosotros, será el sistema el que se encargue de invocarlo cuando necesite dibujar el contenido de la pantalla. En ese momento se proporcionará como parámetro el objeto correspondiente al contexto gráfico de la pantalla del dispositivo, que podremos utilizar para dibujar en ella. Dentro de este método es donde definiremos cómo dibujar en la pantalla, utilizando para ello el objeto de contexto gráfico `Graphics`.

Siempre deberemos dibujar utilizando el objeto `Graphics` dentro del método `paint`. Guardarnos este objeto y utilizarlo después de haberse terminado de ejecutar `paint` puede producir un comportamiento indeterminado, y por lo tanto no debe hacerse nunca.

6.1.2. Propiedades del Canvas

Las pantallas de los dispositivos pueden tener distintas resoluciones. Además normalmente el área donde podemos dibujar no ocupa toda la pantalla, ya que el móvil utiliza una franja superior para mostrar información como la cobertura o el título de la aplicación, y en la franja inferior para mostrar los comandos disponibles.

Sin embargo, a partir de MIDP 2.0 aparece la posibilidad de utilizar modo a pantalla completa, de forma que controlaremos el contenido de toda la pantalla. Para activar el modo a pantalla completa utilizaremos el siguiente método:

```
setFullScreenMode(true); // Solo disponible a partir de MIDP 2.0
```

Es probable que nos interese conocer desde dentro de nuestra aplicación el tamaño real del área del `Canvas` en la que podemos dibujar. Para ello tenemos los métodos `getWidth` y `getHeight` de la clase `Canvas`, que nos devolverán el ancho y el alto del área de dibujo respectivamente.

Para obtener información sobre el número de colores soportados deberemos utilizar la clase `Display` tal como vimos anteriormente, ya que el número de colores es propio de todo el visor y no sólo del área de dibujo.

6.1.3. Mostrar el Canvas

Podemos mostrar este componente en la pantalla del dispositivo igual que mostramos cualquier otro *displayable*:

```
MiCanvas mc = new MiCanvas();  
mi_display.setCurrent(mc);
```

Es posible que queramos hacer que cuando se muestre este *canvas* se realice alguna acción, como por ejemplo poner en marcha alguna animación que se muestre en la pantalla. De la misma forma, cuando el *canvas* se deje de ver deberemos detener la animación. Para hacer esto deberemos tener constancia del momento en el que el *canvas* se muestra y se oculta.

Podremos saber esto debido a que los métodos `showNotify` y `hideNotify` de la clase `Canvas` serán invocados por el sistema cuando dicho componente se muestra o se oculta respectivamente. Nosotros podremos en nuestra subclase de `Canvas` redefinir estos métodos, que por defecto están vacíos, para definir en ellos el código que se debe ejecutar al mostrarse u ocultarse nuestro componente. Por ejemplo, si queremos poner en marcha o detener una animación, podemos redefinir los métodos como se muestra a continuación:

```
public class MiCanvas extends Canvas {  
    public void paint(Graphics g) {  
        // Dibujamos en la pantalla  
        // usando el objeto g proporcionado  
    }  
  
    public void showNotify() {  
        // El Canvas se muestra  
        comenzarAnimacion();  
    }  
  
    public void hideNotify() {  
        // El Canvas se oculta  
    }  
}
```

```
        detenerAnimacion();  
    }  
}
```

De esta forma podemos utilizar estos dos métodos como respuesta a los eventos de aparición y ocultación del *canvas*.

6.2. Contexto gráfico

El objeto `Graphics` nos permitirá acceder al contexto gráfico de un determinado componente, en nuestro caso el *canvas*, y a través de él dibujar en el *raster* de este componente. En el caso del contexto gráfico del *canvas* de LCDUI este *raster* corresponderá a la pantalla del dispositivo móvil. Vamos a ver ahora como dibujar utilizando dicho objeto `Graphics`. Este objeto nos permitirá dibujar distintas primitivas geométricas, texto e imágenes.

6.2.1. Atributos

El contexto gráfico tendrá asociados una serie de atributos que indicarán cómo se va a dibujar en cada momento, como por ejemplo el color o el tipo del lápiz que usamos para dibujar. El objeto `Graphics` proporciona una serie de métodos para consultar o modificar estos atributos. Podemos encontrar los siguientes atributos en el contexto gráfico de LCDUI:

- **Color del lápiz:** Indica el color que se utilizará para dibujar la primitivas geométricas y el texto. MIDP trabaja con color de 24 bits (*truecolor*), que codificaremos en modelo RGB. Dentro de estos 24 bits tendremos 8 bits para cada uno de los tres componentes: rojo (R), verde (G) y azul (B). No tenemos canal *alpha*, por lo que no soportará transparencia. No podemos contar con que todos los dispositivos soporten color de 24 bits. Lo que hará cada implementación concreta de MIDP será convertir los colores solicitados en las aplicaciones al color más cercano soportado por el dispositivo.

Podemos trabajar con los colores de dos formas distintas: tratando los componentes R, G y B por separado, o de forma conjunta. En MIDP desaparece la clase `Color` que teníamos en AWT, por lo que deberemos asignar los colores proporcionando directamente los valores numéricos del color.

Si preferimos tratar los componentes de forma separada, tenemos los siguientes métodos para obtener o establecer el color actual del lápiz:

```
g.setColor(rojo, verde, azul);  
int rojo = g.getRedComponent();  
int green = g.getGreenComponent();  
int blue = g.getBlueComponent();
```

Donde `g` es el objeto `Graphics` del contexto donde vamos a dibujar. Estos componentes rojo, verde y azul tomarán valores entre 0 y 255.

Podemos tratar estos componentes de forma conjunta empaquetándolos en un único entero. En hexadecimal se codifica de la siguiente forma:

0x00RRGGBB

Podremos leer o establecer el color utilizando este formato empaquetado con los siguientes métodos:

```
g.setColor(rgb);  
int rgb = g.getColor();
```

Tenemos también métodos para trabajar con valores en escala de grises. Estos métodos nos pueden resultar útiles cuando trabajemos con dispositivos monocromos.

```
int gris = g.getGrayScale();  
g.setGrayScale(gris);
```

Con estos métodos podemos establecer como color actual distintos tonos en la escala de grises. El valor de gris se moverá en el intervalo de 0 a 255. Si utilizamos `getGrayScale` teniendo establecido un color fuera de la escala de grises, convertirá este color a escala de grises obteniendo su brillo.

- **Tipo del lápiz:** Además del color del lápiz, también podemos establecer su tipo. El tipo del lápiz indicará cómo se dibujan las líneas de las primitivas geométricas. Podemos encontrar dos estilos:

<code>Graphics.SOLID</code>	Línea sólida (se dibujan todos los <i>pixels</i>)
<code>Graphics.DOTTED</code>	Línea punteada (se salta algunos <i>pixels</i> sin dibujarlos)

Podemos establecer el tipo del lápiz o consultarlo con los siguientes métodos:

```
int tipo = g.getStrokeStyle();  
g.setStrokeStyle(tipo);
```

- **Fuente:** Indica la fuente que se utilizará para dibujar texto. Utilizaremos la clase `Font` para especificar la fuente de texto que vamos a utilizar, al igual que en AWT. Podemos obtener o establecer la fuente con los siguientes métodos:

```
Font fuente = g.getFont();  
g.setFont(fuente);
```

- **Área de recorte:** Podemos definir un rectángulo de recorte. Cuando definimos un área de recorte en el contexto gráfico, sólo se dibujarán en pantalla los *pixels* que caigan dentro de este área. Nunca se dibujarán los *pixels* que escribamos fuera de este espacio. Para establecer el área de recorte podemos usar el siguiente método:

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

También tenemos disponible el siguiente método:

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

Este método establece un recorte en el área de recorte anterior. Si ya existía un rectángulo de recorte, el nuevo rectángulo de recorte será la intersección de ambos. Si queremos eliminar el área de recorte anterior deberemos usar el método `setClip`.

- **Origen de coordenadas:** Indica el punto que se tomará como origen en el sistema de coordenadas del área de dibujo. Por defecto este sistema de coordenadas tendrá la coordenada (0,0) en su esquina superior izquierda, y las coordenadas serán positivas hacia la derecha (coordenada x) y hacia abajo (coordenada y), tal como se muestra a continuación:

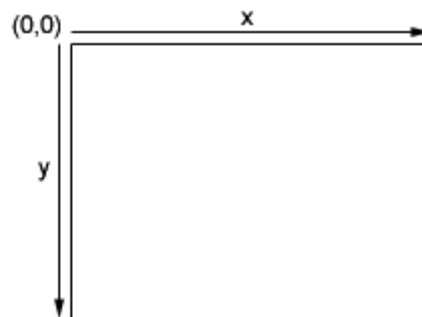


Figura 1. Sistema de coordenadas del área de dibujo

Podemos trasladar el origen de coordenadas utilizando el método `translate`. También tenemos métodos para obtener la traslación del origen de coordenadas.

```
int x = g.getTranslateX();  
int y = g.getTranslateY();  
g.translate(x, y);
```

Estas coordenadas no corresponden a *pixels*, sino a los límites de los *pixels*. De esta forma, el *píxel* de la esquina superior izquierda de la imagen se encontrará entre las coordenadas (0,0), (0,1), (1,0) y (1,1).

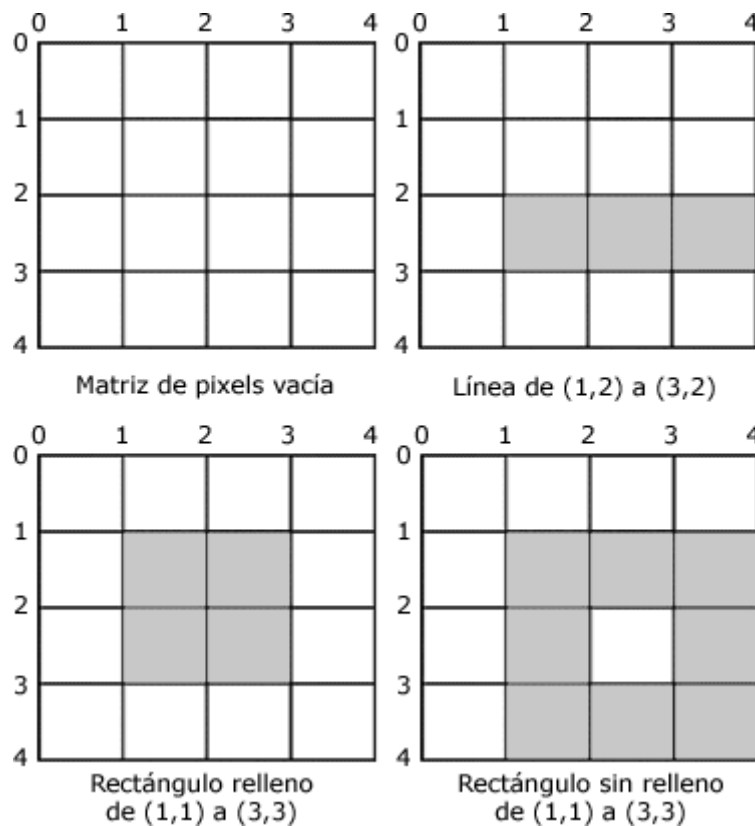


Figura 2. Coordenadas de los límites de los pixels

6.2.2. Dibujar primitivas geométricas

Una vez establecidos estos atributos en el contexto gráfico, podremos dibujar en él una serie de elementos utilizando una serie de métodos de `Graphics`. Vamos a ver en primer lugar cómo dibujar una serie de primitivas geométricas. Para ello tenemos una serie de métodos que comienzan por `draw_` para dibujar el contorno de una determinada figura, o `fill_` para dibujar dicha figura con relleno.

- **Líneas:** Dibuja una línea desde un punto $(x1,y1)$ hasta $(x2,y2)$. Dibujaremos la línea con:

```
g.drawLine(x1, y1, x2, y2);
```

En este caso no encontramos ningún método `fill` ya que las líneas no pueden tener relleno. Al dibujar una línea se dibujarán los *pixels* situados inmediatamente abajo y a la derecha de las coordenadas indicadas. Por ejemplo, si dibujamos con `drawLine(0, 0, 0, 0)` se dibujará el *pixel* de la esquina superior izquierda.

- **Rectángulos:** Podemos dibujar rectángulos especificando sus coordenadas y su altura y anchura. Podemos dibujar el rectángulo relleno o sólo el contorno:

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

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

En el caso de `fillRect`, lo que hará será rellenar con el color actual los *pixels* situados entre las coordenadas limítrofes. En el caso de `drawRect`, la línea inferior y derecha se dibujarán justo debajo y a la derecha respectivamente de las coordenadas de dichos límites, es decir, se dibuja con un *pixel* más de ancho y de alto que en el caso relleno. Esto es poco intuitivo, pero se hace así para mantener la coherencia con el comportamiento de `drawLine`.

Al menos, lo que siempre se nos asegura es que cuando utilizamos las mismas dimensiones no quede ningún hueco entre el dibujo del relleno y el del contorno.

Podemos también dibujar rectángulos con las esquinas redondeadas, utilizando los métodos:

```
g.drawRoundRect(x, y, ancho, alto, ancho_arco,
alto_arco);
g.fillRoundRect(x, y, ancho, alto, ancho_arco,
alto_arco)
```

- **Arcos:** A diferencia de AWT, no tenemos un método para dibujar directamente elipses, sino que tenemos uno más genérico que nos permite dibujar arcos de cualquier tipo. Nos servirá tanto para dibujar elipses y círculos como para cualquier otro tipo de arco.

```
g.drawArc(x, y, ancho, alto, angulo_inicio, angulo_arco);
g.fillArc(x, y, ancho, alto, angulo_inicio, angulo_arco);
```

Los ángulos especificados deben estar en grados. Por ejemplo, si queremos dibujar un círculo o una elipse en `angulo_arco` pondremos un valor de 360 grados para que se cierre el arco. En el caso del círculo los valores de `ancho` y `alto` serán iguales, y en el caso de la elipse serán diferentes.

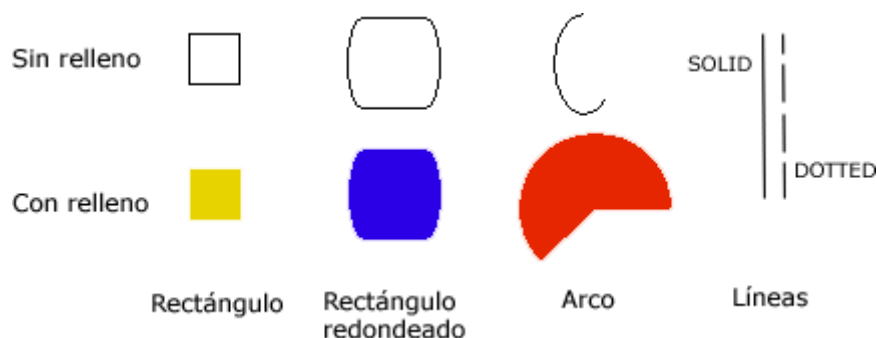


Figura 3. Ejemplos de diferentes primitivas

Por ejemplo, el siguiente *canvas* aparecerá con un dibujo de un círculo rojo y un cuadrado verde:

```
public class MiCanvas extends Canvas {
    public void paint(Graphics g) {
        g.setColor(0x00FF0000);
        g.fillArc(10,10,50,50,0,360);
        g.setColor(0x0000FF00);
        g.fillRect(60,60,50,50);
    }
}
```

6.2.3. Puntos anchor

En MIDP se introduce una característica no existente en AWT que son los puntos *anchor*. Estos puntos nos facilitarán el posicionamiento del texto y de las imágenes en la pantalla. Con los puntos *anchor*, además de dar una coordenada para posicionar estos elementos, diremos qué punto del elemento vamos a posicionar en dicha posición.

Para el posicionamiento horizontal tenemos las siguientes posibilidades:

Graphics.LEFT	En las coordenadas especificadas se posiciona la parte izquierda del texto o de la imagen.
Graphics.HCENTER	En las coordenadas especificadas se posiciona el centro del texto o de la imagen.
Graphics.RIGHT	En las coordenadas especificadas se posiciona la parte derecha del texto o de la imagen.

Para el posicionamiento vertical tenemos:

Graphics.TOP	En las coordenadas especificadas se posiciona la parte superior del texto o de la imagen.
Graphics.VCENTER	En las coordenadas especificadas se posiciona el centro de la imagen. No se aplica a texto.
Graphics.BASELINE	En las coordenadas especificadas se posiciona la línea de base del texto. No se aplica a imágenes.
Graphics.BOTTOM	En las coordenadas especificadas se posiciona la parte inferior del texto o de la imagen.

6.2.4. Cadenas de texto

Podemos dibujar una cadena de texto utilizando el método `drawString`. Debemos proporcionar la cadena de texto de dibujar y el punto *anchor* donde dibujarla.

```
g.drawString(cadena, x, y, anchor);
```

Por ejemplo, si dibujamos la cadena con:

```
g.drawString("Texto de prueba", 0, 0,
Graphics.LEFT|Graphics.BASELINE);
```


Este punto corresponderá al inicio de la cadena (lado izquierdo), en la línea de base del texto como se muestra a continuación:

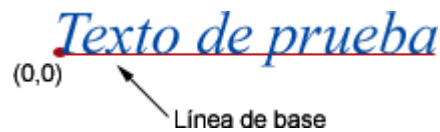


Figura 4. Línea de base del texto

Con esto dibujaremos un texto en pantalla, pero es posible que nos interese conocer las coordenadas que limitan el texto, para saber exactamente el espacio que ocupa en el área de dibujo. En AWT podíamos usar para esto un objeto `FontMetrics`, pero este objeto no existe en MIDP. En MIDP la información sobre las métricas de la fuente está encapsulada en la misma clase `Font` por lo que será más sencillo acceder a esta información. Podemos obtener esta información utilizando los siguientes métodos de la clase `Font`:

- `stringWidth(cadena)`: Nos devuelve el ancho que tendrá la cadena *cadena* en *pixels*.
- `getHeight()`: Nos devuelve la altura de la fuente, es decir, la distancia entre las líneas de base de dos líneas consecutivas de texto. Llamamos ascenso (*ascent*) a la altura típica que suelen subir los caracteres desde la línea de base, y descenso (*descent*) a lo que suelen bajar desde esta línea. La altura será la suma del ascenso y el descenso de la fuente, más un margen para evitar que se junten los caracteres de las dos líneas. Es la distancia existente entre el punto superior (`TOP`) y el punto inferior (`BOTTOM`) de la cadena de texto.
- `getBaselinePosition()`: Nos devuelve el ascenso de la fuente, es decir, la altura típica desde la línea de base hasta la parte superior de la fuente.

Con estas medidas podremos conocer exactamente los límites de una cadena de texto, tal como se muestra a continuación:

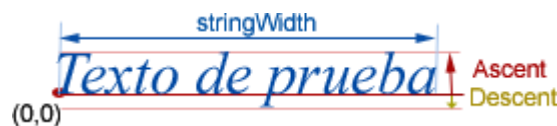


Figura 5. Métricas del texto

6.2.5. Imágenes

Hemos visto como crear imágenes y como utilizarlas en componentes de alto nivel. Estas mismas imágenes encapsuladas en la clase `Image`, podrán ser mostradas también en cualquier posición de nuestro área de dibujo.

Para ello utilizaremos el método:

```
g.drawImage(img, x, y, anchor);
```

En este caso podremos dibujar tanto imágenes mutables como inmutables.

Vimos que las imágenes mutables son aquellas cuyo contenido puede ser modificado. Vamos a ver ahora como hacer esta modificación. Las imágenes mutables, al igual que el *canvas*, tienen un contexto gráfico asociado. En el caso de las imágenes, este contexto gráfico representa el contenido de la imagen que es un *raster* en memoria, pero podremos dibujar en él igual que lo hacíamos en el *canvas*. Esto es así debido a que dibujaremos también mediante un objeto de la clase `Graphics`. Podemos obtener este objeto de contexto gráfico en cualquier momento invocando el método `getGraphics` de la imagen:

```
Graphics offg = img.getGraphics();
```

Si queremos modificar una imagen que hemos cargado de un fichero o de la red, y que por lo tanto es inmutable, podemos crear una copia mutable de la imagen para poder modificarla. Para hacer esto lo primero que deberemos hacer es crear la imagen mutable con el mismo tamaño que la inmutable que queremos copiar. Una vez creada podremos obtener su contexto gráfico, y dibujar en él la imagen inmutable, con lo que habremos hecho la copia de la imagen inmutable a una imagen mutable, que podrá ser modificada más adelante.

```
Image img_mut = Image.createImage(img.getWidth(),  
img.getHeight());  
Graphics offg = img_mut.getGraphics();  
offg.drawImage(img, 0, 0, Graphics.TOP|Graphics.LEFT);
```

6.3. Animación

Hasta ahora hemos visto como dibujar gráficos en pantalla, pero lo único que hacemos es definir un método que se encargue de dibujar el contenido del componente, y ese método será invocado cuando el sistema necesite dibujar la ventana.

Sin embargo puede interesarnos cambiar dinámicamente los gráficos de la pantalla para realizar una animación. Para ello deberemos indicar el momento en el que queremos que se redibujen los gráficos.

6.3.1. Redibujado del área

Para forzar que se redibuje el área de la pantalla deberemos llamar al método `repaint` del *canvas*. Con eso estamos solicitando al sistema que se repinte el contenido, pero no lo repinta en el mismo momento en el que se llama. El sistema introducirá esta solicitud en la cola de eventos pendientes y cuando tenga tiempo repintará su contenido.

```
MiCanvas mc = new MiCanvas();  
...  
mc.repaint();
```

En MIDP podemos forzar a que se realicen todos los repintados pendientes llamando al método `serviceRepaints`. La llamada a este método nos bloqueará hasta que se hayan realizado todos los repintados pendientes. Por esta razón deberemos tener cuidado de no causar un interbloqueo invocando a este método.

```
mc.serviceRepaints();
```

Para repintar el contenido de la pantalla el sistema llamará al método `paint`, en MIDP no existe el método `update` de AWT. Por lo tanto, deberemos definir dentro de `paint` qué se va a dibujar en la pantalla en cada instante, de forma que el contenido de la pantalla varíe con el tiempo y eso produzca el efecto de la animación.

Podemos optimizar el redibujado repintando únicamente el área de la pantalla que haya cambiado. Para ello en MIDP tenemos una variante del método `repaint` que nos permitirá hacer esto.

```
repaint(x, y, ancho, alto);
```

Utilizando este método, la próxima vez que se redibuje se invocará `paint` pero se proporcionará un objeto de contexto gráfico con un área de recorte establecida, correspondiente a la zona de la pantalla que hemos solicitado que se redibuje.

Al dibujar cada *frame* de la animación deberemos borrar el contenido del *frame* anterior para evitar que quede el rastro, o al menos borrar la zona de la pantalla donde haya cambios.

Imaginemos que estamos moviendo un rectángulo por pantalla. El rectángulo irá cambiando de posición, y en cada momento lo dibujaremos en la posición en la que se encuentre. Pero si no borramos el contenido de la pantalla en el instante anterior, el rectángulo aparecerá en todos los lugares donde ha estado en instantes anteriores produciendo este efecto indeseable de dejar rastro. Por ello será necesario borrar el contenido anterior de la pantalla.

Sin embargo, el borrar la pantalla y volver a dibujar en cada *frame* muchas veces puede producir un efecto de parpadeo de los gráficos. Si además en el proceso de dibujado se deben dibujar varios componentes, y vamos dibujando uno detrás de otro directamente en la pantalla, en cada *frame* veremos como se va construyendo poco a poco la escena, cosa que también es un efecto poco deseable.

Para evitar que esto ocurra y conseguir unas animaciones limpias utilizaremos la técnica del *doble buffer*.

6.3.2. Técnica del doble buffer

La técnica del *doble buffer* consiste en dibujar todos los elementos que queremos mostrar en una imagen en memoria, que denominaremos

backbuffer, y una vez se ha dibujado todo volcarlo a pantalla como una unidad. De esta forma, mientras se va dibujando la imagen, como no se hace directamente en pantalla no veremos efectos de parpadeo al borrar el contenido anterior, ni veremos como se va creando la imagen, en pantalla se volcará la imagen como una unidad cuando esté completa.

Para utilizar esta técnica lo primero que deberemos hacer es crearnos el *backbuffer*. Para implementarlo en Java utilizaremos una imagen (objeto `Image`) con lo que tendremos un *raster* en memoria sobre el que dibujar el contenido que queramos mostrar. Deberemos crear una imagen del mismo tamaño de la pantalla en la que vamos a dibujar.

Crearemos para ello una imagen mutable en blanco, como hemos visto anteriormente, con las dimensiones del *canvas* donde vayamos a volcarla:

```
Image backbuffer = Image.createImage(getWidth(), getHeight());
```

Obtenemos su contexto gráfico para poder dibujar en su *raster* en memoria:

```
Graphics offScreen = backbuffer.getGraphics();
```

Una vez obtenido este contexto gráfico, dibujaremos todo lo que queremos mostrar en él, en lugar de hacerlo en pantalla. Una vez hemos dibujado todo el contenido en este contexto gráfico, deberemos volcar la imagen a pantalla (al contexto gráfico del *canvas*) para que ésta se haga visible:

```
g.drawImage(backbuffer, 0, 0, Graphics.TOP|Graphics.LEFT);
```

La imagen conviene crearla una única vez, ya que la animación puede redibujar frecuentemente, y si cada vez que lo hacemos creamos un nuevo objeto imagen estaremos malgastando memoria inútilmente. Es buena práctica de programación en Java instanciar nuevos objetos las mínimas veces posibles, intentando reutilizar los que ya tenemos.

Podemos ver como quedaría nuestra clase ahora:

```
public MiCanvas extends Canvas {  
  
    // Backbuffer  
    Image backbuffer = null;  
  
    // Ancho y alto del backbuffer  
    int width, height;  
  
    // Coordenadas del rectangulo dibujado  
    int x, y;  
  
    public void paint(Graphics g) {  
        // Solo creamos la imagen la primera vez  
        // o si el componente ha cambiado de tamaño  
        if( backbuffer == null ||  
            width != getWidth() ||  
            height != getHeight() )  
        {  

```

```

        width = getWidth();
        height = getHeight();
        backbuffer = Image.createImage(
            width, height);
    }

    Graphics offScreen = backbuffer.getGraphics();

    // Vaciamos el área de dibujo
    offScreen.clearRect(0,0,getWidth(), getHeight());

    // Dibujamos el contenido en offScreen
    offScreen.setColor(0xFF0000);
    offScreen.fillRect(x,y,50,50);

    // Volcamos el back buffer a pantalla
    g.drawImage(backbuffer,0,0,Graphics.TOP|Graphics.LEFT);
}

```

En ese ejemplo se dibuja un rectángulo rojo en la posición (x,y) de la pantalla que podrá ser variable, tal como veremos a continuación añadiendo a este ejemplo métodos para realizar la animación.

Algunas implementaciones de MIDP ya realizan internamente el doble *buffer*, por lo que en esos casos no será necesario que lo hagamos nosotros. Es más, convendrá que no lo hagamos para no malgastar innecesariamente el tiempo. Podemos saber si implementa el doble *buffer* o no llamando al método `isDoubleBuffered` del `Canvas`.

Podemos modificar el ejemplo anterior para en caso de realizar el doble *buffer* la implementación de MIDP, no hacerla nosotros:

```

public MiCanvas extends Canvas {
    ...
    public void paint(Graphics gScreen) {

        boolean doblebuffer = isDoubleBuffered();

        // Solo creamos el backbuffer si no hay doble
        // buffer
        if( !doblebuffer ) {
            if ( backbuffer == null ||
                width != getWidth() ||
                height != getHeight() )
            {
                width = getWidth();
                height = getHeight();
                backbuffer = Image.createImage(width, height);
            }
        }

        // g sera la pantalla o nuestro backbuffer segun si
        // el doble buffer está ya implementado o no

        Graphics g = null;
    }
}

```

```

        if(doblebuffer) {
            g = gScreen;
        } else {
            g = backbuffer.getGraphics();
        }

        // Vaciamos el área de dibujo

        g.clearRect(0,0,getWidth(), getHeight());

        // Dibujamos el contenido en g
        g.setColor(0x00FF0000);
        g.fillRect(x,y,50,50);

        // Volcamos si no hay doble buffer implementado
        if(!doblebuffer) {
            gScreen.drawImage(backbuffer,0,0,
                             Graphics.TOP|Graphics.LEFT);
        }
    }
}

```

6.3.3. Código para la animación

Si queremos hacer una animación tendremos que ir cambiando ciertas propiedades de los objetos de la imagen (por ejemplo su posición) y solicitar que se redibuje tras cada cambio. Esta tarea deberá realizarla un hilo que se ejecute en segundo plano. El bucle para la animación podría ser el siguiente:

```

public class MiCanvas extends Canvas {
    ...
    public void run() {
        // El rectángulo comienza en (10,10)
        x = 10;
        y = 10;

        while(x < 100) {
            x++;
            repaint();

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {}
        }
    }
}

```

Con este código de ejemplo veremos una animación en la que el rectángulo que dibujamos partirá de la posición (10,10) y cada 100ms se moverá un *pixel* hacia la derecha, hasta llegar a la coordenada (100,10).

Si queremos que la animación se ponga en marcha nada más mostrarse la pantalla del canvas, podremos hacer que este hilo comience a ejecutarse en el método `showNotify` como hemos visto anteriormente.

```

public class MiCanvas extends Canvas implements Runnable {

```

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

Para implementar estas animaciones podemos utilizar un hilo que duerma un determinado período tras cada iteración, como en el ejemplo anterior, o bien utilizar temporizadores que realicen tareas cada cierto periodo de tiempo. Los temporizadores nos pueden facilitar bastante la tarea de realizar animaciones, ya que simplemente deberemos crear una tarea que actualice los objetos de la escena en cada iteración, y será el temporizador el que se encargue de ejecutar cíclicamente dicha tarea.

6.3.4. Hilo de eventos

Hemos visto que existen una serie de métodos que se invocan cuando se produce algún determinado evento, y nosotros podemos redefinir estos métodos para indicar cómo dar respuesta a estos eventos. Estos métodos que definimos para que sean invocados cuando se produce un evento son denominados *callbacks*. Tenemos los siguientes *callbacks*:

- `showNotify` y `hideNotify`, para los eventos de aparición y ocultación del canvas.
- `paint` para el evento de dibujado.
- `commandAction` para el evento de ejecución de un comando.
- `keyPressed`, `keyRepeated`, `keyReleased`, `pointerPressed`, `pointerDragged` y `pointerReleased` para los eventos de teclado y de puntero, que veremos más adelante.

Estos eventos son ejecutados por el sistema de forma secuencial, desde un mismo hilo de eventos. Por lo tanto, estos *callbacks* deberán devolver el control cuanto antes, de forma que bloqueen al hilo de eventos el mínimo tiempo posible.

Si dentro de uno de estos *callbacks* tenemos que realizar una tarea que requiera tiempo, deberemos crear un hilo que realice la tarea en segundo plano, para que el hilo de eventos siga ejecutándose mientras tanto.

En algunas ocasiones puede interesarnos ejecutar alguna tarea de forma secuencial dentro de este hilo de eventos. Por ejemplo esto será útil si queremos ejecutar el código de nuestra animación sin que interfiera con el método `paint`. Podemos hacer esto con el método `callSerially` del objeto `Display`. Deberemos proporcionar un objeto `Runnable` para ejecutar su método `run` en serie dentro del hilo de eventos. La tarea que definamos dentro de este `run` deberá terminar pronto, al igual que ocurre con el código definido en los *callbacks*, para no bloquear el hilo de eventos.

Podemos utilizar `callSerially` para ejecutar el código de la animación de la siguiente forma:

```
public class MiCanvas extends Canvas implements Runnable {  
    ...  
    public void anima() {  
        // Inicia la animación  
        repaint();  
        mi_display.callSerially(this);  
    }  
  
    public void run() {  
        // Actualiza la animación  
        ...  
        repaint();  
        mi_display.callSerially(this);  
    }  
}
```

La llamada a `callSerially` nos devuelve el control inmediatamente, no espera a que el método `run` sea ejecutado.

6.3.5. Optimización de imágenes

Si tenemos varias imágenes correspondientes a varios *frames* de una animación, podemos optimizar nuestra aplicación guardando todas estas imágenes como una única imagen. Las guardaremos en forma de mosaico dentro de un mismo fichero de tipo imagen, y en cada momento deberemos mostrar por pantalla sólo una de las imágenes dentro de este mosaico.



Figura 6. Imagen con los frames de una animación

De esta forma estamos reduciendo el número de ficheros que incluimos en el JAR de la aplicación, por lo que por una lado reduciremos el espacio de este fichero, y por otro lado tendremos que abrir sólo un fichero, y no varios.

Para mostrar sólo una de las imágenes del mosaico, lo que podemos hacer es establecer un área de recorte del tamaño de un elemento del mosaico (*frame*) en la posición donde queramos dibujar esta imagen. Una vez hecho esto, ajustaremos las coordenadas donde dibujar la imagen de forma que dentro del área de recorte caiga el elemento del mosaico que queremos mostrar en este momento. De esta forma, sólo será dibujado este elemento, ignorándose el resto.

Podemos ver esto ilustrado en la Figura 18. En ella podemos ver a la izquierda cómo mostrar el segundo frame del reloj, y a la derecha cómo mostrar el tercer frame. Queremos dibujar el reloj en la posición (x, y) de la imagen. Cada frame de este reloj tiene un tamaño `anchoFrame` x `altoFrame`. Por lo tanto, el área de recorte será un rectángulo cuya esquina superior izquierda estará en las coordenadas (x, y) y tendrá una altura y una anchura de `altoFrame` y `anchoFrame` respectivamente, para de esta manera poder dibujar en esa región de la pantalla cada *frame* de nuestra imagen.

Para dibujar cada uno de los *frames* deberemos desplazar la imagen de forma que el *frame* que queramos mostrar caiga justo bajo el área de recorte establecida. De esta forma al dibujar la imagen, se volcará a la pantalla sólo el *frame* deseado, y el resto, al estar fuera del área de recorte, no se mostrará. En la figura podemos ver los *frames* que quedan fuera del área de recorte representados con un color más claro, al volcar la imagen estos *frames* no se dibujarán.

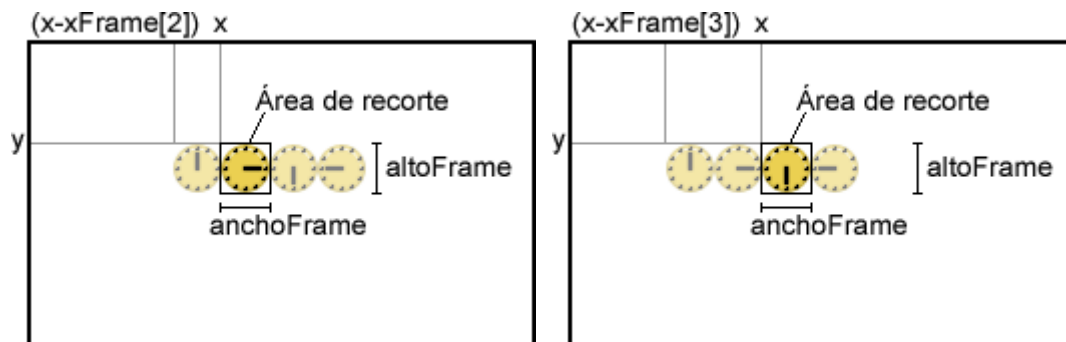


Figura 7. Ejemplo de dibujo de diferentes frames de una imagen

A continuación se muestra el código fuente del ejemplo anterior, con el que podremos dibujar cada *frame* de la imagen.

```
// Guardamos el área de recorte anterior
int clipX = g.getClipX();
int clipY = g.getClipY();
int clipW = g.getClipWidth();
int clipH = g.getClipHeight();

// Establecemos nuevo área de recorte
g.clipRect(x, y, anchoFrame, altoFrame);

// Dibujamos la imagen con el desplazamiento adecuado
g.drawImage(
    imagen,
    x - xFrame[frameActual],
    y - yFrame[frameActual],
    Graphics.TOP | Graphics.LEFT);

// Reestablecemos el área de recorte anterior
g.setClip(clipX, clipY, clipW, clipH);
```

6.4. Eventos de entrada

La clase `Canvas` nos permite acceder a los eventos de entrada del usuario a bajo nivel. De esta forma podremos saber cuando el usuario pulsa o suelta cualquier tecla del dispositivo. Cuando ocurra un evento en el teclado se invocará uno de los siguientes métodos de la clase `Canvas`:

<code>keyPressed(int cod)</code>	Se ha presionado la tecla con código <code>cod</code>
<code>keyRepeated(int cod)</code>	Se mantiene presionada la tecla con código <code>cod</code>
<code>keyReleased(int cod)</code>	Se ha soltado la tecla con código <code>cod</code>

Estos dispositivos, además de generar eventos cuando presionamos o soltamos una tecla, son capaces de generar eventos de repetición. Estos eventos se producirán cada cierto período de tiempo mientras mantengamos pulsada una tecla.

Al realizar aplicaciones para móviles debemos tener en cuenta que en la mayoría de estos dispositivos no se puede presionar más de una tecla al mismo tiempo. Hasta que no hayamos soltado la tecla que estemos pulsando, no se podrán recibir eventos de pulsación de ninguna otra tecla.

Para dar respuesta a estos eventos del teclado deberemos redefinir estos métodos en nuestra subclase de `Canvas`:

```
public class MiCanvas extends Canvas {
    ...
    public void keyPressed(int cod) {
        // Se ha presionado la tecla con código cod
    }

    public void keyRepeated(int cod) {
        // Se mantiene pulsada la tecla con código cod
    }

    public void keyReleased(int cod) {
        // Se ha soltado la tecla con código cod
    }
}
```

6.4.1. Códigos del teclado

Cada tecla del teclado del dispositivo tiene asociado un código identificativo que será el parámetro que se le proporcione a estos métodos al presionarse o soltarse. Tenemos una serie de constantes en la clase `Canvas` que representan los códigos de las teclas estándar:

<code>Canvas.KEY_NUM0</code>	0
<code>Canvas.KEY_NUM1</code>	1
<code>Canvas.KEY_NUM2</code>	2
<code>Canvas.KEY_NUM3</code>	3
<code>Canvas.KEY_NUM4</code>	4
<code>Canvas.KEY_NUM5</code>	5
<code>Canvas.KEY_NUM6</code>	6
<code>Canvas.KEY_NUM7</code>	7
<code>Canvas.KEY_NUM8</code>	8

<code>Canvas.KEY_NUM9</code>	<code>9</code>
<code>Canvas.KEY_POUND</code>	<code>#</code>
<code>Canvas.KEY_STAR</code>	<code>*</code>

Los teclados, además de estas teclas estándar, normalmente tendrán otras teclas, cada una con su propio código numérico. Es recomendable utilizar únicamente estas teclas definidas como constantes para asegurar la portabilidad de la aplicación, ya que si utilizamos cualquier otro código de tecla no podremos asegurar que esté disponible en todos los modelos de teléfonos.

Los códigos de tecla corresponden al código Unicode del carácter correspondiente a dicha tecla. Si la tecla no corresponde a ningún carácter Unicode entonces su código será negativo. De esta forma podremos obtener fácilmente el carácter correspondiente a cada tecla. Sin embargo, esto no será suficiente para realizar entrada de texto, ya que hay caracteres que corresponden a múltiples pulsaciones de una misma tecla, y a bajo nivel sólo tenemos constancia de que una misma tecla se ha pulsado varias veces, pero no sabemos a qué carácter corresponde ese número de pulsaciones. Si necesitamos que el usuario escriba texto, lo más sencillo será utilizar uno de los componentes de alto nivel.

Podemos obtener el nombre de la tecla correspondiente a un código dado con el método `getKeyName` de la clase `Canvas`.

6.4.2. Acciones de juegos

Tenemos también definidas lo que se conoce como acciones de juegos (*game actions*) con las que representaremos las teclas que se utilizan normalmente para controlar los juegos, a modo de *joystick*. Las acciones de juegos principales son:

<code>Canvas.LEFT</code>	Movimiento a la izquierda
<code>Canvas.RIGHT</code>	Movimiento a la derecha
<code>Canvas.UP</code>	Movimiento hacia arriba
<code>Canvas.DOWN</code>	Movimiento hacia abajo
<code>Canvas.FIRE</code>	Fuego

Una misma acción puede estar asociada a varias teclas del teléfono, de forma que el usuario pueda elegir la que le resulte más cómoda. Las teclas asociadas a cada acción de juego serán dependientes de la implementación, cada modelo de teléfono puede asociar a las teclas las acciones de juego que considere más apropiadas según la distribución del teclado, para que el manejo sea cómodo. Por lo tanto, el utilizar estas acciones hará la aplicación más portable, ya que no tendremos que adaptar los controles del juego para cada modelo de móvil.

Para conocer la acción de juego asociada a un código de tecla dado utilizaremos el siguiente método:

```
int accion = getGameAction(keyCode);
```

De esta forma podremos realizar de una forma sencilla y portable aplicaciones que deban controlarse utilizando este tipo de acciones.

Podemos hacer la transformación inversa con:

```
int codigo = getKeyCode(accion);
```

Hemos de resaltar que una acción de código puede estar asociada a más de una tecla, pero con este método sólo podremos obtener la tecla principal que realiza dicha acción.

6.4.3. Punteros

Algunos dispositivos tienen punteros como dispositivos de entrada. Esto es común en los PDAs, pero no en los teléfonos móviles. Los *callbacks* que deberemos redefinir para dar respuesta a los eventos del puntero son los siguientes:

<code>pointerPressed(int x, int y)</code>	Se ha pinchado con el puntero en (x,y)
<code>pointerDragged(int x, int y)</code>	Se ha arrastrado el puntero a (x,y)
<code>pointerReleased(int x, int y)</code>	Se ha soltado el puntero en (x,y)

En todos estos métodos se proporcionarán las coordenadas (x,y) donde se ha producido el evento del puntero.

6.5. APIs propietarias

Existen APIs propietarias de diferentes vendedores, que añaden funcionalidades no soportadas por la especificación de MIDP. Los desarrolladores de estas APIs propietarias no deben incluir en ellas nada que pueda hacerse con MIDP. Estas APIs deben ser únicamente para permitir acceder a funcionalidades que MIDP no ofrece.

Es recomendable no utilizar estas APIs propietarias siempre que sea posible, para hacer aplicaciones que se ajusten al estándar de MIDP. Lo que podemos hacer es desarrollar aplicaciones que cumplan con el estándar MIDP, y en el caso que detecten que hay disponible una determinada API propietaria la utilicen para obtener alguna mejora. A continuación veremos como detectar en tiempo de ejecución si tenemos disponible una determinada API.

Vamos a ver la API Nokia UI, disponible en gran parte de los modelos de teléfonos Nokia, que incorpora nuevas funcionalidades para la programación de la interfaz de usuario no disponibles en MIDP 1.0. Esta API está contenida en el paquete `com.nokia.mid`.

6.5.1. Gráficos

En cuanto a los gráficos, tenemos disponibles una serie de mejoras respecto a MIDP.

Añade soporte para crear un *canvas* a pantalla completa. Para crear este *canvas* utilizaremos la clase `FullCanvas` de la misma forma que utilizábamos `Canvas`.

Define una extensión de la clase `Graphics`, en la clase `DirectGraphics`. Para obtener este contexto gráfico extendido utilizaremos el siguiente método:

```
DirectGraphics dg = DirectUtils.getDirectGraphics(g);
```

Siendo `g` el objeto de contexto gráfico `Graphics` en el que queremos dibujar. Este nuevo contexto gráfico añade:

- Soporte para nuevos tipos de primitivas geométricas (triángulos y polígonos).
- Soporte para transparencia, incorporando un canal *alpha* al color. Ahora tenemos colores de 32 bits, cuya forma empaquetada se codifica como `0xAARRGGBB`.
- Acceso directo a los *pixels* del *raster* de pantalla. Podremos dibujar *pixels* en pantalla proporcionando directamente el *array* de *pixels* a dibujar, o bien obtener los *pixels* de la pantalla en forma de un *array* de *pixels*. Cada *pixel* de este *array* será un valor `int`, `short` o `byte` que codificará el color de dicho *pixel*.
- Permite transformar las imágenes a dibujar. Podremos hacer rotaciones de 90, 180 o 270 grados y transformaciones de espejo con las imágenes al mostrarlas.

6.5.2. Sonido

Una limitación de MIDP 1.0 es que no soporta sonido. Por ello para incluir sonido en las aplicaciones de dispositivos que sólo soporten MIDP 1.0 como API estándar deberemos recurrir a APIs propietarias para tener estas funcionalidades. La API Nokia UI nos permitirá solucionar esta carencia.

Nos permitirá reproducir sonidos como tonos o ficheros de onda (WAV). Los tipos de formatos soportados serán dependientes de cada dispositivo.

6.5.3. Control del dispositivo

Además de las características anteriores, esta API nos permitirá utilizar funciones propias de los dispositivos. En la clase `DeviceControl` tendremos métodos para controlar la vibración del móvil y el parpadeo de las luces de la pantalla.

6.5.4. Detección de la API propietaria

Si utilizamos una API propietaria reduciremos la portabilidad de la aplicación. Por ejemplo, si usamos la API Nokia UI la aplicación sólo funcionará en algunos dispositivos de Nokia. Hay una forma de utilizar estas APIs propietarias sin afectar a la portabilidad de la aplicación. Podemos detectar en tiempo de ejecución si la API propietaria está disponible de la siguiente forma:

```
boolean hayNokiaUI = false;

try {
    Class.forName("com.nokia.mid.sound.Sound");
    hayNokiaUI = true;
} catch (ClassNotFoundException e) {}
```

De esta forma, si la API propietaria está disponible podremos utilizarla para incorporar más funcionalidades a la aplicación. En caso contrario, no deberemos ejecutar nunca ninguna instrucción que acceda a esta API propietaria.

6.6. Gráficos 3D

Podemos crear aplicaciones que muestren gráficos en 3D utilizando la API opcional Mobile 3D Graphics for J2ME (JSR-184).

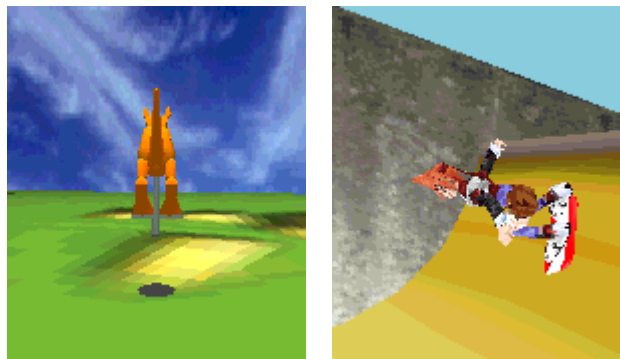


Figura 8. Ejemplos de aplicaciones 3D para dispositivos móviles

Esta API nos permitirá añadir contenido 3D a las aplicaciones de dos modos distintos:

- **Modo inmediato:** Este modo nos servirá para crear los gráficos 3D a bajo nivel, creando directamente los polígonos que va a tener nuestro mundo 3D. Este modo nos permitirá tener un control absoluto sobre los polígonos que se dibujan, pero el tener que definir manualmente estos polígonos hace que este modo no sea adecuado cuando tengamos que mostrar objetos complejos (por ejemplo personajes de un juego). Este modo resultará útil por ejemplo para generar gráficos 3D para representar datos.

- **Modo *retained*:** En este modo contaremos con una serie de objetos ya creados que podremos añadir a la escena. Estos objetos que utilizamos los podremos cargar de ficheros en los que tendremos almacenados los distintos objetos 3D que vayamos a mostrar. La escena se representará como un grafo, del que colgarán todos los objetos que queramos mostrar en ella. Este modo será útil para aplicaciones como juegos, en las que tenemos que mostrar objetos complejos, como por ejemplo los personajes del juego.

Más adelante estudiaremos con más detalle cómo mostrar gráficos 3D utilizando cada uno de estos modos.

6.6.1. Renderización 3D

Para renderizar gráficos 3D en el móvil utilizaremos un objeto de tipo `Graphics3D`. Para obtener este objeto utilizaremos el siguiente método estático:

```
Graphics3D g3d;  
...  
g3d = Graphics3D.getInstance();
```

Podremos declarar este objeto como campo global de la clase y obtenerlo una única vez durante la inicialización de la misma.

Para poder utilizar este objeto, deberemos decirle en qué contexto gráfico queremos que vuelque el contenido 3D que genere. Normalmente estableceremos como objetivo donde volcar el contexto gráfico asociado a la pantalla del móvil.

Esto lo haremos en el método `paint`, como se muestra a continuación:

```
protected void paint(Graphics g) {  
    try {  
        g3d.bindTarget(g);  
        ...  
        g3d.render(escena);  
    } finally {  
        g3d.releaseTarget();  
    }  
}
```

Con `bindTarget` establecemos en qué contexto gráfico vamos a volver los gráficos 3D. Una vez hecho esto podremos renderizar la escena que hayamos definido utilizando el método `render` del objeto `Graphics3D`. Más adelante veremos cómo definir esta escena utilizando tanto el modo inmediato como el modo *retained*. Por último, con `releaseTarget` liberamos el contexto gráfico que estábamos utilizando para volcar los gráficos.

6.6.2. Transformaciones geométricas

Cuando queramos establecer la posición de un objeto 3D en el mundo, deberemos transformar sus coordenadas para que éstas estén en la posición

deseada. Para representar estas transformaciones geométricas en el espacio tenemos la clase `Transform`.

Utilizaremos objetos `Transform` para indicar la posición y el tamaño que van a tener los objetos en el mundo. En la transformación podemos establecer traslaciones del objeto, rotaciones, y cambios de escala.

Normalmente asociaremos a cada objeto que tengamos en el mundo 3D una transformación, donde se indicará la posición de dicho objeto. Además, esto nos servirá para hacer animaciones, ya que simplemente cambiando los valores de la transformación en el tiempo podremos hacer que el objeto cambie de posición o tamaño.

Utilizando el constructor de `Transform` podemos crear una transformación identidad:

```
Transform trans = new Transform();
```

Con esta transformación, el objeto al que se aplique permanecerá en sus coordenadas iniciales. Podremos modificar esta transformación para cambiar la posición o el tamaño del objeto mediante:

- **Traslación:** Especificamos la traslación en (x,y,z) que queremos hacer con el objeto.

```
trans.postTranslate(tx, ty, tz);
```

- **Rotación:** Especificamos un eje de rotación del objeto, dado con el vector (x,y,z) que representa el eje, un ángulo de rotación alrededor de dicho eje dado en grados.

```
trans.postRotate(angulo, vx, vy, vz);
```

- **Escala:** Especificamos el factor de escala a aplicar en cada una de las tres coordenadas. Con factor 1.0 mantendrá su tamaño actual, y reduciendo o aumentando este valor conseguiremos reducir o agrandar el objeto en dicha coordenada.

```
trans.postScale(sx, sy, sz);
```

Podemos hacer que la transformación vuelva a ser la identidad para volver a la posición original del objeto:

```
trans.setIdentity();
```

6.6.3. Cámara

Para poder renderizar una escena es imprescindible establecer previamente el punto de vista desde el que la queremos visualizar. Para hacer esto tendremos que definir una cámara, encapsulada en el objeto `Camera`.

Crearemos la cámara utilizando el constructor vacío de esta clase:

```
Camera cam;  
...  
cam = new Camera();
```

Con esto tendremos la cámara, que siempre estará apuntando hacia la coordenada Z negativa (0, 0, -1). Si queremos mover o rotar la cámara, deberemos hacerlo mediante un objeto `Transform` que aplicaremos a la cámara:

```
Transform tCam;  
...  
tCam = new Transform();  
tCam.postTranslate(0.0f, 0.0f, 2.0f);
```

De la cámara deberemos especificar el tipo de proyección que queremos utilizar y una serie de parámetros. Podemos establecer dos tipos de proyecciones:

- **Paralela:** En este tipo de proyecciones, el centro de proyección está en el infinito. Las líneas que son paralelas en el espacio, se mantienen paralelas en la imagen proyectada. Estableceremos este tipo de proyección con:

```
cam.setParallel(campoDeVision, relacionAspecto,  
               planoFrontal, planoPosterior);
```

- **Perspectiva:** El centro de proyección es un punto finito. En este caso, las líneas que son paralelas en el espacio, en la imagen proyectada convergen en algún punto. Estableceremos este tipo de proyección con:

```
cam.setPerspective(campoDeVision, relacionAspecto,  
                  planoFrontal, planoPosterior);
```

En el parámetro `campoDeVision` indicaremos el campo visual de la cámara en grados. Este valor normalmente será 45° o 60°. Valores superiores a 60° distorsionan la imagen. En `relacionAspecto` indicaremos la proporción entre el ancho y el alto de la pantalla donde se van a mostrar los gráficos. Además deberemos establecer dos planos de recorte: uno cercano (`planoFrontal`) y uno lejano (`planoPosterior`), de forma que sólo se verán los objetos que estén entre estos dos planos. Todo aquello más cercano al plano frontal, y más lejano al plano posterior será recortado.

Por ejemplo, podemos utilizar una proyección perspectiva como la siguiente:

```
cam.setPerspective(60.0f,  
                  (float)getWidth() / (float)getHeight(),  
                  1.0f, 1000.0f);
```

Una vez definida la cámara, deberemos establecerla en el objeto `Graphics3D` de la siguiente forma:

```
g3d.setCamera(cam, tCam);
```

Vemos que al establecer la cámara, además de el objeto `Camera`, deberemos especificar la transformación que se va a aplicar sobre la misma. Si queremos hacer movimientos de cámara no tendremos más que modificar la transformación que utilizamos para posicionarla en la escena.

Si vamos a dejar la cámara fija, podemos establecer la cámara en el objeto `Graphics3D` una única vez durante la inicialización. Sin embargo, si queremos mover la cámara, deberemos establecerla dentro del método `paint`, para que cada vez que se redibuje se sitúe la cámara en su posición actual.

6.6.4. Luces

Deberemos también definir la iluminación del mundo. Para crear luces utilizaremos el objeto `Light`. Podemos crear una luz utilizando su constructor vacío:

```
Light luz;  
...  
luz = new Light();
```

La luz tendrá un color y una intensidad, que podremos establecer utilizando los siguientes métodos:

```
luz.setColor(0xffffffff);  
luz.setIntensity(1.0f);
```

Como color podremos especificar cualquier color RGB, y como intensidad deberemos introducir un valor dentro del rango [0.0, 1.0]. Estos atributos son comunes para cualquier tipo de luces.

Además, podemos utilizar diferentes tipos de luces con distintas propiedades. Estos tipos son:

- **Ambiente** (`Light.AMBIENT`): Esta luz afectará a todos los objetos por igual en todas las direcciones. Es la luz mínima del ambiente, que incide en todas las partes de los objetos. Simula la reflexión de la luz sobre los objetos que produce dicha luz que está "en todas partes".
- **Direccional** (`Light.DIRECTIONAL`): Esta luz incide en todos los objetos en una misma dirección. Con esta luz podemos modelar una fuente de luz muy lejana (en el infinito). Por ejemplo, nos servirá para modelar la iluminación del sol, que incide en todos los objetos en la misma dirección en cada momento. La luz estará apuntando en la dirección del eje de las Z negativo (0, 0, -1).
- **Omnidireccional** (`Light.OMNI`): Se trata de una fuente de luz que ilumina en todas las direcciones. Por ejemplo con esta luz podemos modelar una bombilla, que es un punto que emana luz en todas las direcciones.

- **Foco** (`Light.SPOT`): Este tipo produce un cono de luz. El foco ilumina en una determinada dirección, produciendo un cono de luz. El foco estará orientado en la dirección del eje de las Z negativo (0, 0, -1).

Para establecer el tipo de luz que queremos utilizar tenemos el método `setMode`:

```
luz.setMode(Light.DIRECTIONAL);
```

Dentro de los tipos de luces anteriores, podemos distinguir dos grupos según la posición de la fuente. Tenemos luces sin posición (ambiente y direccional) y luces con posición (omnidireccional y foco). En el caso de las luces con posición, podemos utilizar el método `setAttenuation` para dar una atenuación a la luz en función a la distancia de la fuente. Con esto haremos que los objetos que estén más lejos de la fuente de luz estén menos iluminados que los que estén más cerca. Esto no se puede hacer en el caso de las fuentes de luz sin posición, ya que en ese caso la distancia de cualquier objeto a la fuente es infinito o no existe.

Otra clasificación que podemos hacer es entre fuentes de luz sin dirección (ambiente y omnidireccional) y con dirección (direccional y foco). Hemos visto que las fuentes con dirección siempre están apuntando en la dirección del eje de las Z negativo. Si queremos modificar esta dirección podemos aplicar una rotación a la luz mediante un objeto `Transform`. De la misma forma, en las fuentes con posición podremos aplicar una traslación para modificar la posición de la fuente de luz con este objeto.

```
Transform tLuz;  
...  
tLuz = new Transform();
```

Para añadir las luces al mundo utilizaremos el método `addLight` sobre el objeto de contexto gráfico 3D, en el que especificaremos la luz y la transformación que se va a realizar:

```
g3d.addLight(luz, tLuz);
```

Si como transformación especificamos `null`, se aplicará la identidad. Por ejemplo en el caso de una luz ambiente no tiene sentido aplicar ninguna transformación a la luz, ya que no tiene ni posición ni dirección.

```
g3d.addLight(luzAmbiente, null);
```

Al igual que ocurría en el caso de la cámara, si las luces van a permanecer fijas sólo hace falta que las añadamos una vez durante la inicialización. Sin embargo, si se van a mover, tendremos que establecer la posición de la luz cada vez que se renderiza para que se apliquen los cambios. En este caso si añadimos las luces dentro del método `paint` deberemos llevar cuidado, ya que el método `addLight` añade luces sobre las que ya tiene definidas el mundo. Para evitar que las luces se vayan acumulando, antes de volver a añadirlas

tendremos que eliminar las que teníamos en la iteración anterior con `resetLights`. Una vez hecho esto podremos añadir las demás luces:

```
g3d.resetLights();
g3d.addLight(luz, tLuz);
g3d.addLight(luzAmbiente, null);
```

6.6.5. Fondo

También deberemos establecer un fondo para el visor antes de renderizar en él los gráficos 3D. Como fondo podemos poner un color sólido o una imagen. El fondo lo representaremos mediante un objeto de la clase `Background`:

```
Background fondo;
...
fondo = new Background();
```

Con el constructor vacío construiremos el fondo por defecto que consiste en un fondo negro. Podemos cambiar el color de fondo con el método `setColor`, o establecer una imagen de fondo con `setImage`.

Cada vez que vayamos a renderizar los gráficos en `paint` es importante que vaciemos todo el área de dibujo con el color (o la imagen) de fondo. Para ello utilizaremos el método `clear`:

```
g3d.clear(fondo);
```

Si no hiciésemos esto, es posible que tampoco se visualizasen los gráficos 3D que se rendericen posteriormente. Si no queremos crear ningún fondo y utilizar el fondo negro por defecto, podremos llamar a este método proporcionando `null` como parámetro:

```
g3d.clear(null);
```

Vamos a recapitular todo lo visto hasta ahora, y a ver cómo quedará el método `paint` añadiendo luces, cámara y fondo:

```
protected void paint(Graphics g) {
    try {
        g3d.bindTarget(g);

        g3d.setCamera(cam, tCam);

        g3d.resetLights();
        g3d.addLight(luz, tLuz);
        g3d.addLight(luzAmbiente, null);

        g3d.clear(fondo);

        g3d.render(vb, tsa, ap, tCubo);
    } finally {
        g3d.releaseTarget();
    }
}
```

6.6.6. Modo inmediato

Para crear gráficos utilizando este modo deberemos definir la lista de vértices (x, y, z) de nuestro gráfico, y una lista de índices en la que definimos las caras (polígonos) que se van a mostrar. Cada cara se construirá a partir de un conjunto de vértices. Los índices con los que se define cada cara en la lista de caras, harán referencia a los índices de los vértices que la componen en la lista de vértices.

A continuación se muestra un ejemplo de cómo crear un cubo utilizando modo inmediato:

```
// Lista de vertices
short [] vertexValues = {
    0, 0, 0, // 0
    0, 0, 1, // 1
    0, 1, 0, // 2
    0, 1, 1, // 3
    1, 0, 0, // 4
    1, 0, 1, // 5
    1, 1, 0, // 6
    1, 1, 1 // 7
};

// Lista de caras
int [] faceIndices = {
    0, 1, 2, 3,
    7, 5, 6, 4,
    4, 5, 0, 1,
    3, 7, 2, 6,
    0, 2, 4, 6,
    1, 5, 3, 7
};
```

En la lista de vértices podemos ver que tenemos definidas las coordenadas de las 8 esquinas del cubo. En la lista de caras se definen las 6 caras del cubo. Cada cara se define a partir de los 4 vértices que la forman. De estos vértices especificaremos su índice en la lista de vértices.

En M3G representaremos las caras y vértices de nuestros objetos mediante los objetos `VertexBuffer` e `IndexBuffer` respectivamente. Estos objetos los crearemos a partir de los arrays de vértices y caras definidos anteriormente.

Para crear el objeto `VertexBuffer` es necesario que le pasemos los arrays de datos necesarios en forma de objetos `VertexArray`. Para construir el objeto `VertexArray` como parámetros deberemos proporcionar:

```
VertexArray va = new VertexArray(numVertices,
                                numComponentes,
                                bytesComponente);
```

Debemos decir en `numVertices` el número de vértices que hemos definido, en `numComponentes` el número de componentes de cada vértice (al tratarse de coordenadas 3D en este caso será siempre 3: x, y ,z) y el número de bytes por

componente. Es decir, si se trata de una array de tipo `byte` tendrá un byte por componente, mientras que si es de tipo `short` tendrá 2 bytes por componente.

Una vez creado el array, deberemos llenarlo de datos. Para ello utilizaremos el siguiente método:

```
va.set(vertexInicial, numVertices, listaVertices);
```

Proporcionaremos un array de vértices (`listaVertices`) como el visto en el ejemplo anterior para insertar dichos vértices en el objeto `VertexArray`. Además diremos, dentro de este array de vértices, cual es el vértice inicial a partir del cual queremos empezar a insertar (`vertexInicial`) y el número de vértices, a partir de dicho vértice, que vamos a insertar (`numVertices`).

En el caso del array de vértices de nuestro ejemplo, utilizaremos los siguientes datos:

```
VertexArray va = new VertexArray(8, 3, 2);  
va.set(0, 8, vertexValues);
```

Con esto tendremos construido un objeto `VertexArray` en el que tendremos el array de posiciones de los vértices.

Ahora podremos crear un objeto `VertexBuffer` y utilizar el objeto anterior para establecer su lista de coordenadas de los vértices.

```
VertexBuffer vb;  
...  
vb = new VertexBuffer();  
vb.setPositions(va, 1.0f, null);
```

Al método `setPositions` le podemos proporcionar como segundo y tercer parámetro un factor de escala y una traslación respectivamente, para transformar las coordenadas de los puntos que hemos proporcionado. Si no queremos aplicar ninguna transformación geométrica a estos puntos, simplemente proporcionaremos como escala `1.0f` y como traslación `null`.

Una vez creado este objeto, necesitaremos el objeto de índices en el que se definen las caras. Este objeto será de tipo `IndexBuffer`, sin embargo esta clase es abstracta, por lo que deberemos utilizar una de sus subclases. La única subclase disponible por el momento es `TriangleStripArray` que representa las caras a partir de tiras (strips) de triángulos.

Para definir cada tira de triángulos podremos especificar 3 o más vértices. Si indicamos más de 3 vértices se irán tomando como triángulos cada trio adyacente de vértices. Por ejemplo, si utilizamos la tira `{ 1,2,3,4 }`, se crearán los triángulos `{ 1,2,3 }` y `{ 2,3,4 }`.

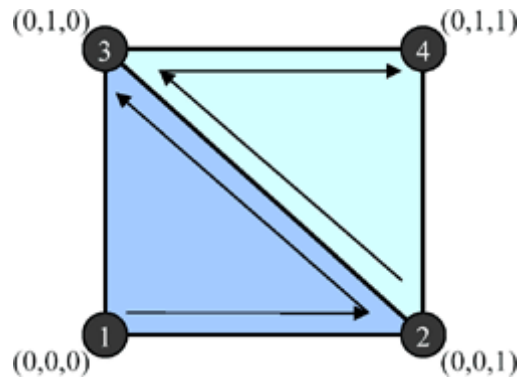


Figura 9. Tira de triángulos para una cara del cubo.

El orden en el que indiquemos los vértices de cada polígono será importante. Según el sentido de giro en el que se defina su cara frontal y cara trasera será una u otra. La cara frontal será aquella para la que el sentido de giro de los vértices vaya en el sentido de las agujas del reloj.

Cuando creemos este objeto deberemos proporcionar, a parte de la lista de índices, un array con el tamaño de cada strip.

Por ejemplo, en el caso del ejemplo del cubo, como cada strip se compone de 4 índices, el array de tamaños tendrá el valor 4 para cada uno de los 6 strips que teníamos definidos:

```
int [] stripSizes = {
    4, 4, 4, 4, 4, 4
};
```

Con esta información ya podremos crear el objeto `TriangleStripArray`:

```
TriangleStripArray tsa;
...
tsa = new TriangleStripArray(faceIndices, stripSizes);
```

Además de la información de vértices y caras, deberemos darle una apariencia al objeto creado. Para ello podremos utilizar el objeto `Appearance`. Podemos crear un objeto con la apariencia por defecto:

```
Appearance ap;
...
ap = new Appearance();
```

Sobre este objeto podremos cambiar el material o la textura de nuestro objeto 3D.

Para renderizar el gráfico 3D que hemos construido, utilizaremos la siguiente variante del método `render`:

```
g3d.render(VertexBuffer vertices, IndexBuffer indices,
            Appearance apariencia, Transform transformacion);
```

En ella especificaremos los distintos componentes de nuestra figura 3D, conocida como *submesh*: lista de vértices (`VertexBuffer`), lista de caras (`IndexBuffer`) y apariencia (`Appaerance`). Además especificaremos también un objeto `Transform` con la transformación geométrica que se le aplicará a nuestro objeto al añadirlo a la escena 3D. Podemos crear una transformación identidad utilizando el constructor vacío de esta clase:

```
Transform tCubo;  
...  
tCubo = new Transform();
```

Aplicando esta transformación se dibujará el cubo en sus coordenadas originales. Si posteriormente queremos moverlo en el espacio (trasladarlo o rotarlo) o cambiar su tamaño, podremos hacerlo simplemente modificando esta transformación.

Para renderizar nuestro ejemplo del cubo utilizaremos el método `render` como se muestra a continuación:

```
g3d.render(vb, tsa, ap, tCubo);
```

Con lo que hemos visto hasta ahora, si mostramos el cubo que hemos creado aparecerá con el siguiente aspecto:



Esta es la apariencia (`Appaerance`) definida por defecto. En ella no hay ningún material ni textura definidos para el objeto. Al no haber ningún material establecido (el material es `null`), la iluminación no afecta al objeto.

Vamos ahora a establecer un material para el objeto. Creamos un material (objeto `Material`):

```
Material mat = new Material();
```

Esto nos crea un material blanco por defecto. Podremos modificar en este objeto el color de ambiente, el color difuso, el color especular, y el color emitido del material.

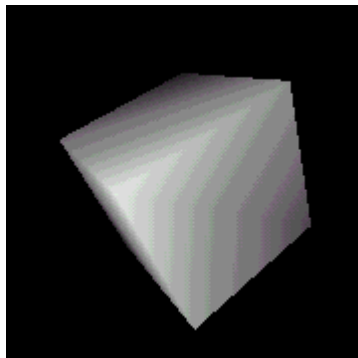
Cuando hayamos establecido un material para el objeto, la iluminación afectará sobre él. En este caso será necesario definir las normales de los vértices del

objeto. Si estas normales no estuviesen definidas, se producirá un error al renderizar.

Para asignar las normales a los vértices del objeto utilizaremos un objeto `VertexArray` que añadiremos al `VertexBuffer` de nuestro objeto utilizando el método `setNormals`.

```
byte [] normalValues = {  
    -1, -1, -1,  
    -1, -1, 1,  
    -1, 1, -1,  
    -1, 1, 1,  
    1, -1, -1,  
    1, -1, 1,  
    1, 1, -1,  
    1, 1, 1  
};  
  
...  
  
VertexArray na = new VertexArray(8, 3, 1);  
na.set(0, 8, normalValues);  
  
...  
  
vb.setNormals(na);
```

Con esto, al visualizar nuestro cubo con una luz direccional apuntando desde nuestro punto de vista hacia el cubo, se mostrará con el siguiente aspecto:



Vamos a ver ahora como añadir textura al objeto. Para ello lo primero que debemos hacer es añadir coordenadas de textura a cada vértice. Para añadir estas coordenadas utilizaremos también un objeto `VertexArray` que añadiremos al `VertexBuffer` mediante el método `setTexCoords`.

```
short [] tex = {  
    0,0, 0,0, 0,1, 0,1,  
    1,0, 1,0, 1,1, 1,1  
};  
  
VertexArray ta = new VertexArray(8, 2, 2);  
ta.set(0, 8, tex);  
  
vb.setTexCoords(0, ta, 1.0f, null);
```

El primer parámetro que toma `setTexCoords` es el índice de la unidad de textura a la que se van a asignar esas coordenadas. Como segundo parámetro se proporcionan las coordenadas en forma de `VertexArray`. El tercer y cuarto parámetro nos permiten realizar escalados y traslaciones de estas coordenadas respectivamente.

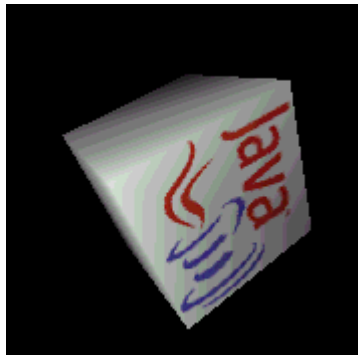
Una vez hecho esto podemos crear la textura a partir de una imagen y añadirla a la apariencia:

```
try {
    Image img = Image.createImage("/texture.png");
    Image2D img2d = new Image2D(Image2D.RGB, img);

    Texture2D tex2d = new Texture2D(img2d);
    ap.setTexture(0, tex2d);
} catch (IOException e) { }
```

La textura se establece en el objeto `Appaerance` mediante el método `setTexture`. A este método le proporcionamos como parámetros el índice de la unidad de textura, y la textura que vamos a establecer en dicha unidad de textura. De esta forma se podrán definir varias unidades de textura, teniendo cada una de ellas unas coordenadas y una imagen.

Con esto el aspecto del cubo será el siguiente:



Podemos añadir animación al objeto modificando su transformación a lo largo del tiempo. Para ello podemos crear un hilo como el siguiente:

```
public void run() {
    while(true) {
        tCubo.postTranslate(0.5f, 0.5f, 0.5f);
        tCubo.postRotate(1.0f, 1.0f, 1.0f, 1.0f);
        tCubo.postTranslate(-0.5f, -0.5f, -0.5f);
        repaint();
        try {
            Thread.sleep(25);
        } catch (InterruptedException e) { }
    }
}
```

A continuación mostramos el código completo del canvas de este ejemplo:

```
package es.ua.j2ee.m3d;

import java.io.IOException;

import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;

public class Visor3DImmediate extends Canvas implements
Runnable {

    MIDlet3D owner;

    Graphics3D g3d;
    Transform tCam;
    Transform tLuz;
    Transform tCubo;
    VertexBuffer vb;
    IndexBuffer tsa;
    Appearance ap;
    Camera cam;
    Light luz;
    Light luzAmbiente;
    Background fondo;

    short [] vertexValues = {
        0, 0, 0, // 0
        0, 0, 1, // 1
        0, 1, 0, // 2
        0, 1, 1, // 3
        1, 0, 0, // 4
        1, 0, 1, // 5
        1, 1, 0, // 6
        1, 1, 1 // 7
    };

    byte [] normalValues = {
        -1, -1, -1,
        -1, -1, 1,
        -1, 1, -1,
        -1, 1, 1,
        1, -1, -1,
        1, -1, 1,
        1, 1, -1,
        1, 1, 1
    };

    int [] faceIndices = {
        0, 1, 2, 3,
        7, 5, 6, 4,
        4, 5, 0, 1,
        3, 7, 2, 6,
        0, 2, 4, 6,
        1, 5, 3, 7
    };

    int [] stripSizes = {
        4, 4, 4, 4, 4, 4
    };

    short [] tex = {
        0,0, 0,0, 0,1, 0,1,
```

```
1,0, 1,0, 1,1, 1,1
};

public Visor3DImmediate(MIDlet3D owner) {
    this.owner = owner;
    init3D();
}

public void init3D() {

    g3d = Graphics3D.getInstance();
    tCubo = new Transform();
    tCam = new Transform();
    tLuz = new Transform();

    cam = new Camera();
    cam.setPerspective(60.0f,
(float)getWidth()/(float)getHeight(), 1.0f, 10.0f);
    tCam.postTranslate(0.0f, 0.0f, 2.0f);

    luz = new Light();
    luz.setColor(0x0ffffff);
    luz.setIntensity(1.0f);
    luz.setMode(Light.DIRECTIONAL);
    tLuz.postTranslate(0.0f, 0.0f, 5.0f);

    luzAmbiente = new Light();
    luzAmbiente.setColor(0x0ffffff);
    luzAmbiente.setIntensity(0.5f);
    luzAmbiente.setMode(Light.AMBIENT);

    fondo = new Background();

    VertexArray va = new VertexArray(8, 3, 2);
    va.set(0, 8, vertexValues);

    VertexArray na = new VertexArray(8, 3, 1);
    na.set(0, 8, normalValues);

    VertexArray ta = new VertexArray(8, 2, 2);
    ta.set(0, 8, tex);

    vb = new VertexBuffer();
    vb.setPositions(va, 1.0f, null);
    vb.setNormals(na);
    vb.setTexCoords(0, ta, 1.0f, null);

    tsa = new TriangleStripArray(faceIndices, stripSizes);

    ap = new Appearance();
    Material mat = new Material();
    ap.setMaterial(mat);

    try {
        Image img = Image.createImage("/texture.png");
        Image2D img2d = new Image2D(Image2D.RGB, img);

        Texture2D tex2d = new Texture2D(img2d);
        ap.setTexture(0, tex2d);
    } catch (IOException e) { }
```

```

        tCubo.postTranslate(-0.5f, -0.5f, -0.5f);
    }

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

    public void run() {
        while(true) {
            tCubo.postTranslate(0.5f, 0.5f, 0.5f);
            tCubo.postRotate(1.0f, 1.0f, 1.0f, 1.0f);
            tCubo.postTranslate(-0.5f, -0.5f, -0.5f);
            repaint();
            try {
                Thread.sleep(25);
            } catch (InterruptedException e) { }
        }
    }

    protected void paint(Graphics g) {
        try {
            g3d.bindTarget(g);

            g3d.setCamera(cam, tCam);
            g3d.resetLights();
            g3d.addLight(luz, tLuz);
            g3d.addLight(luzAmbiente, null);
            g3d.clear(fondo);

            g3d.render(vb, tsa, ap, tCubo);
        } finally {
            g3d.releaseTarget();
        }
    }
}

```

6.6.7. Modo retained

Utilizando este modo trabajaremos con un grafo en el que tendremos los distintos elementos de la escena 3D: objetos 3D, luces, cámaras, etc. Este grafo estará compuesto por objetos derivados de `Node` (nodos). Tenemos los siguientes tipos de nodos disponibles:

- **world:** El nodo raíz de este grafo es del tipo `World`, que representa nuestro mundo 3D completo. De él colgaremos todos los objetos del mundo, las cámaras y las luces. Además, el nodo `World` tendrá asociado el fondo (`Background`) de la escena.
- **group:** Grupo de nodos. Nos permite crear grupos de nodos dentro del grafo. De él podremos colgar varios nodos. El tener los nodos agrupados de esta forma nos permitirá, aplicando transformaciones geométricas sobre el grupo, mover todos estos nodos como un único bloque. Podemos crear un nuevo grupo creando un nuevo objeto `Group`, y añadir nodos hijos mediante su método `addChild`.
- **camera:** Define una cámara (punto de vista) en la escena. Las cámaras definen la posición del espectador en la escena. Podemos tener varias

cámaras en el mundo, pero en un momento dado sólo puede haber una cámara activa, que será la que se utilice para renderizar. Se crean como hemos visto en apartados anteriores. El objeto correspondiente al mundo (`World`) tiene un método `setActiveCamera` con el que podremos establecer cuál será la cámara activa en cada momento.

- **Light**: Define las luces de la escena. Se crean como hemos visto en apartados anteriores.
- **Mesh**: Define un objeto 3D. Este objeto se puede crear a partir de sus vértices y caras como vimos en el apartado anterior.

```
Mesh obj = new Mesh(vb, tsa, ap);
```

- **Sprite3D**: Representa una imagen 2D posicionada dentro de nuestro mundo 3D y alineada con la pantalla.

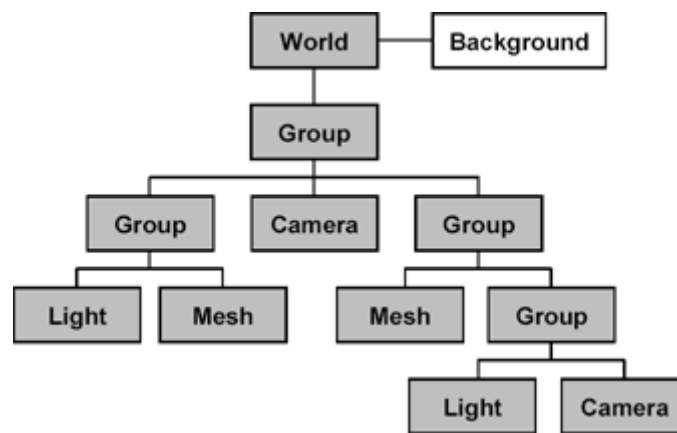


Figura 10. Ejemplo de grafo de la escena.

En este modo normalmente cargaremos estos componentes de la escena 3D de un fichero con formato M3G. Para crear mundo en este formato podremos utilizar herramientas como Swerve Studio. De este modo podremos modelar objetos 3D complejos utilizando una herramienta adecuada, y posteriormente importarlos en nuestra aplicación.

Para cargar objetos 3D de un fichero M3G utilizaremos un objeto `Loader` de la siguiente forma:

```
World mundo;
...
try {
    Object3D [] objs = Loader.load("/mundo.m3g");
    mundo = (World)objs[0];
} catch (IOException e) {
    System.out.println("Error al cargar modelo 3D: " +
e.getMessage());
}
```

Con `load` cargaremos del fichero M3G especificado los objetos 3D que contenga. La clase `Object3D` es la superclase de todos los tipos de objetos que podemos utilizar para definir nuestra escena. De esta forma permitimos que

esta función nos devuelva cualquier tipo de objeto, según lo que haya almacenado en el fichero. Estos objetos pueden ser nodos como los vistos anteriormente, animaciones, imágenes, etc.

Por ejemplo, si tenemos un fichero M3G que contiene un mundo 3D completo (objeto `World`), cogeremos el primer objeto devuelto y haremos una conversión cast al tipo adecuado (`World`).

Como en este objeto se define la escena completa, no hará falta añadir nada más, podemos renderizarlo directamente con:

```
protected void paint(Graphics g) {
    try {
        g3d.bindTarget(g);
        g3d.render(mundo);
    } finally {
        g3d.releaseTarget();
    }
}
```

En el mundo 3D del fichero, a parte de los modelos 3D de los objetos, luces y cámaras, podemos almacenar animaciones predefinidas sobre elementos del mundo. Para utilizar esta animación llamaremos al método `animate` sobre el mundo que queremos animar:

```
protected void paint(Graphics g) {
    try {
        g3d.bindTarget(g);
        mundo.animate(tiempo);
        g3d.render(mundo);
    } finally {
        g3d.releaseTarget();
    }
}
```

Le deberemos proporcionar un valor de tiempo, que deberemos ir incrementando cada vez que se pinta. Podemos crear un hilo que cada cierto periodo incremente este valor y llame a `repaint` para volver a renderizar la escena.

A continuación vemos el código completo el ejemplo de mundo que utiliza modo *retained*:

```
package es.ua.j2ee.m3d;

import java.io.IOException;

import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;

public class Visor3DRetained extends Canvas implements
Runnable {

    MIDlet3D owner;
```

```
Graphics3D g3d;
World mundo;

int tiempo = 0;

public Visor3DRetained(MIDlet3D owner) {
    this.owner = owner;
    init3D();
}

public void init3D() {
    g3d = Graphics3D.getInstance();

    try {
        Object3D [] objs = Loader.load("/mundo.m3g");
        mundo = (World)objs[0];
    } catch (IOException e) {
        System.out.println("Error al cargar modelo 3D: " +
e.getMessage());
    }
}

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

public void run() {
    while(true) {
        tiempo+=10;
        repaint();
        try {
            Thread.sleep(25);
        } catch (InterruptedException e) { }
    }
}

protected void paint(Graphics g) {
    try {
        g3d.bindTarget(g);

        mundo.animate(tiempo);
        g3d.render(mundo);
    } finally {
        g3d.releaseTarget();
    }
}
}
```


7. Juegos

Sin duda una de las aplicaciones que más famosas se han hecho con el surgimiento de los teléfonos móviles MIDP son los juegos Java. Con estos teléfonos los usuarios pueden descargar estos juegos de Internet, normalmente previo pago de una tarifa, e instalarlos en el teléfono. De esta forma podrán añadir fácilmente al móvil cualquier juego realizado en Java, sin limitarse así a tener únicamente el típico juego de "la serpiente" y similares que vienen preinstalados en determinados teléfonos.

Vamos a ver en esta sección los conceptos básicos de la programación de videojuegos y las APIs de MIDP dedicadas a esta tarea. Comenzaremos viendo una introducción a los tipos de videojuegos que normalmente encontraremos en los teléfonos móviles.

7.1. Tipos de juegos

Podemos distinguir diferentes tipos de juegos:

- **Juegos de mesa:** Podemos encontrar juegos de mesa como por ejemplo juegos de cartas, ajedrez, reversi, etc. Estos juegos suelen ser muy sencillos, y la velocidad con la que se ejecuten no es crítica como ocurre en el caso de los juegos de acción.
- **Puzzles:** Son juegos de inteligencia, normalmente con una mecánica bastante sencilla. Lo fundamental en estos juegos es superar las pruebas propuestas, sin necesitar unos gráficos complejos ni tener ninguna componente de acción.
- **Juegos de acción 2D:** Consideremos juegos de acción 2D aquellos juegos en los que debemos manejar a un personaje u objeto. Podemos encontrar multitud de géneros dentro de este tipo de juegos: *shoot'em ups*, simuladores, plataformas, lucha, etc.
- **Juegos de acción 3D:** Con la evolución de los procesadores gráficos los juegos de acción han ido pasando de desarrollarse con gráficos 2D a desarrollarse con gráficos en 3D. Podemos encontrar prácticamente los mismos géneros que en el caso de los juegos 2D. Este tipo de juegos todavía es demasiado complejo para los modelos de móviles actuales, por lo que no lo tendremos en consideración.





Figura 1. Capturas de juegos para diferentes modelos de móviles

Los primeros juegos que podíamos encontrar en los móviles eran normalmente juegos muy sencillos tipo puzzle o de mesa, o en todo caso juegos de acción muy simples similares a los primeros videojuegos aparecidos antes de los 80. En los móviles con soporte para Java podremos tener juegos más complejos, como los que se podían ver en los ordenadores y consolas de 8 bits, y estos juegos irán mejorando conforme evolucionen los teléfonos móviles.

Además tenemos las ventajas de que existe una gran comunidad de programadores en Java, a los que no les costará aprender a desarrollar este tipo de juegos para móviles, por lo que el número de juegos disponible crecerá rápidamente. El poder descargar y añadir estos juegos al móvil de forma sencilla, como cualquier otra aplicación Java, hará estos juegos especialmente atractivos para los usuarios, ya que de esta forma podrán estar disponiendo continuamente de nuevos juegos en su móvil.

Los juegos que se ejecutan en un móvil tendrán distintas características que los juegos para ordenador o videoconsolas, debido a las peculiaridades de estos dispositivos.

- **Escasa memoria.** No podremos crear demasiados objetos. Además el tamaño del JAR con todos los datos del juego también suele estar limitado, muchas veces deberemos hacer un juego en 64KB. Esto nos obligará a rescatar viejas técnicas de programación de videojuegos de los tiempos de los 8 bits a mediados/finales de los 80.
- **CPU lenta.** La CPU de los móviles es bastante lenta comparada con la de los ordenadores de sobremesa y las videoconsolas. Es importante que los juegos vayan de forma fluida, por lo que antes de distribuir nuestra aplicación deberemos probarla en móviles reales para asegurarnos de que funcione bien, ya que muchas veces los emuladores funcionarán a velocidades distintas.
- **Pantalla reducida.** Deberemos tener esto en cuenta en los juegos, y hacer que todos los objetos se vean correctamente. Podemos utilizar *zoom* en determinadas zonas para poder visualizar mejor los objetos de la escena.
- **Almacenamiento limitado.** En el móvil el espacio para guardar datos sobre la partida también está bastante limitado. Será interesante permitir guardar la partida, para que el usuario puede continuar más adelante donde se quedó. Esto es especialmente importante en los móviles, ya que muchas veces se utilizan estos juegos mientras el usuario viaja en autobús, o está esperando, de forma que puede tener que finalizar la partida en cualquier momento. Deberemos hacer esto utilizando la mínima cantidad de espacio posible.
- **Poco ancho de banda.** Si desarrollamos juegos en red deberemos tener en cuenta la baja velocidad y la latencia de la red. Deberemos minimizar el tráfico que circula por la red.
- **Teclado pequeño.** El teclado de los móviles es muy pequeño y si hacemos que deban utilizarse muchas teclas puede resultar incómodo y complicado de manejar. Deberemos intentar proporcionar un manejo cómodo, haciendo que el control sea lo más sencillo posible, con un número reducido de posibles acciones.
- **Posibles interrupciones.** En el móvil es muy probable que se produzca una interrupción involuntaria de la partida, por ejemplo cuando recibimos una llamada entrante. Deberemos permitir que esto ocurra. Además también es conveniente que el usuario pueda pausar la partida fácilmente.

Ante todo, estos videojuegos deben ser atractivos para los jugadores, ya que su única finalidad es entretener.

7.2. Desarrollo de juegos para móviles

Hemos visto que los juegos son aplicaciones que deben resultar atractivas para los usuarios. Por lo tanto deben tener gráficos personalizados e innovadores. La API de bajo nivel de LCDUI nos ofrece el control suficiente sobre lo que dibujamos en pantalla para poder crear cualquier interfaz gráfica que queramos

que tengan nuestros juegos. Esto, junto al control a bajo nivel que nos ofrece sobre los eventos de entrada, hace que esta API sea suficiente para desarrollar videojuegos para móviles.

Además, en MIDP 2.0 se incluyen una serie de clases adicionales en el paquete `javax.microedition.lcdui.game` que basándose en la API a bajo nivel de LCDUI nos ofrecerán facilidades para el desarrollo de juegos. En estas clases tendremos implementados los objetos genéricos que solemos encontrar en todos los juegos. Si queremos desarrollar juegos con MIDP 1.0, deberemos implementar nosotros manualmente todos estos objetos.

Los juegos se instalarán en el móvil como cualquier otra aplicación Java. En el caso concreto de los teléfonos Nokia, podemos establecer en el fichero JAD un atributo propio de esta marca con el que especificamos el tipo de aplicación:

```
Nokia-MIDlet-Category: Game
```

Especificando que se trata de un juego, cuando lo instalemos en el teléfono los guardará directamente en la carpeta *juegos*, y no en la carpeta *aplicaciones* como lo haría por defecto.

Para ilustrar las explicaciones, utilizaremos como ejemplo un clon del clásico *Frogger*. El mecanismo de este juego es muy sencillo:

- Tenemos una carretera con varios carriles por los que circulan coches.
- Por cada carril los coches circulan a distinta velocidad y con distinta separación entre ellos.
- Debemos cruzar la carretera pasando entre los coches sin ser atropellados.
- Cuando consigamos llegar al otro lado de la carretera habremos completado el nivel.
- Si un coche nos atropella, perderemos una vida.

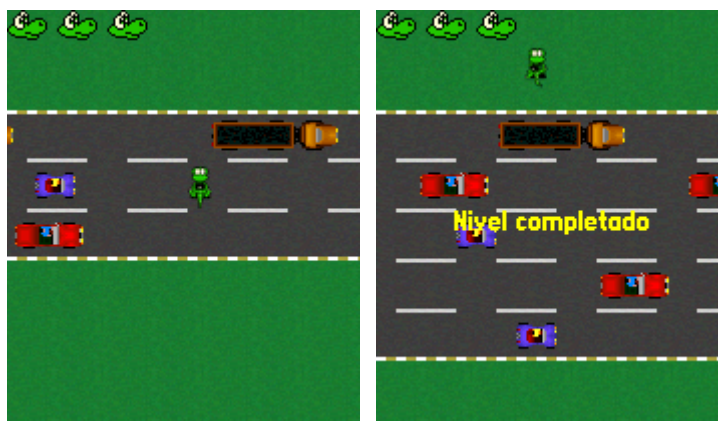


Figura 2. Ejemplo de juego

7.2.1. Aplicación conducida por los datos

Cuando desarrollamos juegos, será conveniente llevar a la capa de datos todo lo que podamos, dejando la parte del código lo más simple y genérica posible.

Por ejemplo, podemos crear ficheros de datos donde se especifiquen las características de cada nivel del juego, el tipo y el comportamiento de los enemigos, los textos, etc.

Normalmente los juegos consisten en una serie de niveles. Cada vez que superemos un nivel, entraremos en uno nuevo en el que se habrá incrementado la dificultad, pero la mecánica del juego en esencia será la misma. Por esta razón es conveniente que el código del programa se encargue de implementar esta mecánica genérica, que conoceremos como motor del juego, y lea de ficheros de datos todas las características de cada nivel concreto.

De esta forma, si queremos añadir o modificar niveles del juego, cambiar la inteligencia artificial de los enemigos, añadir nuevos tipos de enemigos, o cualquier otro cambio de este tipo, no tendremos que modificar el código fuente, simplemente bastará con cambiar los ficheros de datos.

Es recomendable también centralizar la carga y la gestión de los recursos en una única clase. De esta forma quedará más claro qué recursos carga la aplicación, ya que no tendremos la carga de recursos dispersa por todo el código de las clases del juego. En este mismo fichero podemos tener los textos que se muestren en pantalla, lo que nos facilitará realizar traducciones del juego, ya que sólo tendremos que modificar este fichero.

Por ejemplo, podemos tener un clase `Resources` donde se centralice la carga de los recursos como la siguiente:

```
public class Resources {  
  
    // Identificadores de las imagenes  
    public static final int IMG_TIT_TITULO = 0;  
    public static final int IMG_SPR_CROC = 1;  
    public static final int IMG_BG_STAGE_1 = 2;  
    public static final int IMG_CAR_TYPE_1 = 3;  
    public static final int IMG_CAR_TYPE_2 = 4;  
    public static final int IMG_CAR_TYPE_3 = 5;  
    public static final int IMG_FACE_LIVES = 6;  
  
    // Imagenes y datos del juego  
    public static Image[] img;  
    public static Image splashImage;  
    public static StageData[] stageData;  
  
    // MIDlet principal del juego  
    public static MIDletJuego midlet;  
  
    // Nombres de los ficheros de las imagenes  
    private static String[] imgNames =  
    {
```

```
        "/title.png",
        "/sprite.png",
        "/stage01.png",
        "/car01.png",
        "/car02.png",
        "/car03.png",
        "/face.png"
    };
    private static String splashImageFile = "/logojava.png";
    private static String stageFile = "/stages.dat";

    // Obtiene una imagen
    public static Image getImage(int imgIndex) {
        return img[imgIndex];
    }

    // Obtiene los datos de una fase
    public static StageData getStage(int stageIndex) {
        return stageData[stageIndex];
    }

    // Carga los recursos
    public static void init() throws IOException {
        // Carga las imagenes
        loadCommonImages();

        // Carga datos de niveles
        InputStream in =
stageFile.getClass().getResourceAsStream(stageFile);
        stageData = loadStageData(in);
    }

    // Inicializa los recursos para la pantalla splash
    public static void initSplash(MIDletJuego pMidlet)
        throws IOException {
        midlet = pMidlet;
        splashImage = Image.createImage(splashImageFile);
    }

    // Carga imagenes
    private static void loadCommonImages() throws IOException {
        int nImg = imgNames.length;

        img = new Image[nImg];

        for (int i = 0; i < nImg; i++) {
            img[i] = Image.createImage(imgNames[i]);
        }
    }

    // Carga los datos de los niveles
    public static StageData[] loadStageData(InputStream in)
        throws IOException {

        StageData[] stages = null;

        DataInputStream dis = new DataInputStream(in);

        int stageNum = dis.readInt();
        stages = new StageData[stageNum];
    }
}
```



```
for (int i = 0; i < stageNum; i++) {  
    stages[i] = StageData.deserialize(dis);  
}  
  
return stages;  
}
```

Cuando se inicie el juego, llamaremos al método `init` de esta clase para que se inicialicen los recursos, en este momento se cargarán todas las imágenes y los datos que necesitemos. Una vez inicializado, podremos utilizar el método `getImage` para obtener las imágenes que vayamos a utilizar desde cualquier parte del código.

Como la carga de recursos suele ser lenta, normalmente mostraremos una pantalla de presentación, llamada *Splash Screen*, mientras éstos se cargan. En esta pantalla se suele mostrar el logo de nuestra compañía e información sobre el *copyright* de la aplicación.

Por esta razón existe en la clase `Resources` anterior un método `initSplash`, que hace una inicialización previa de los recursos que necesita esta pantalla *splash*, que normalmente será el logo que se vaya a mostrar. De esta forma, cuando carguemos la aplicación, se cargarán estos recursos básicos para la pantalla de *splash*, se mostrará la pantalla *splash*, y mientras ésta se muestra se cargarán el resto de recursos.

También debemos destacar que proporcionamos en la inicialización el `MIDlet` principal de nuestra aplicación. De esta forma, a través de la clase de recursos desde cualquier lugar de la aplicación tendremos acceso a este `MIDlet`, que será necesario para poder cambiar de pantalla o salir de la aplicación.

Por último, como hemos comentado, es recomendable llevar a la capa de datos toda la información posible. Esta clase `Resources` tiene un método `loadStageData` que la función que realiza es cargar desde un fichero la información sobre las fases del juego.

En el ejemplo de nuestro clon de *Frogger*, para cada fase consideraremos los siguientes datos:

- Título de la fase
- Carriles de la carretera. Para cada carril tendremos los siguientes datos:
 - Velocidad de los coches que circulan por él.
 - Separación que hay entre los coches del carril.
 - Tipo de coche que circula por el carril.

Toda esta información será conveniente tenerla almacenada en un fichero, de forma que simplemente editando el fichero, podremos editar las fases del juego. Por ejemplo, podemos utilizar un objeto `DataOutputStream` para codificar esta información de forma binaria con la siguiente estructura:

```
<int> Numero de fases  
Para cada fase
```

```
<UTF> Título
<byte> Número de carriles
Para cada carril
  <byte> Velocidad
  <short> Separación
  <byte> Tipo de coche
```

Para leer en nuestro juego este tipo de información, podemos crear los métodos de deserialización oportunos en las clases que encapsulen estos datos.

En el método `loadStageData` de la clase `Resources` se lee el primer valor con el número de fases, y para cada fase deserializaremos la información utilizando el objeto que encapsula los datos de las fases:

```
public class StageData {

    public String title;
    public TrackData [] tracks;

    public static StageData deserialize(DataInputStream dis)
                                   throws IOException {
        StageData data = new StageData();

        data.title = dis.readUTF();
        byte nTracks = dis.readByte();

        data.tracks = new TrackData[nTracks];
        for(byte i=0;i<nTracks;i++) {
            data.tracks[i] = TrackData.deserialize(dis);
        }

        return data;
    }
}
```

De la misma forma, utilizaremos la clase que encapsula los datos de los carriles para deserializar la información de los mismos:

```
public class TrackData {

    public byte velocity;
    public short distance;
    public byte carType;

    public static TrackData deserialize(DataInputStream dis)
                                   throws IOException {
        TrackData data = new TrackData();

        data.velocity = dis.readByte();
        data.distance = dis.readShort();
        data.carType = dis.readByte();

        return data;
    }
}
```


7.2.2. Portabilidad

Al utilizar la API de bajo nivel para desarrollar el juego, la portabilidad de la aplicación se reducirá, ya que la implementación será dependiente de parámetros como el tamaño de la pantalla, el número de colores, el teclado del dispositivo, etc.

Para poder adaptar nuestra aplicación fácilmente a distintos tipos de dispositivos, conviene definir toda esta información como constantes centralizadas en una única clase, de forma que simplemente cambiando la información sobre las dimensiones de los distintos elementos del juego podremos adaptarlo a distintos tamaños de pantalla.

Por ejemplo, podríamos definir una clase como la siguiente:

```
public class CommonData {  
  
    // Numero de vidas total  
    public static final int NUM_LIVES = 3;  
  
    // Dimensiones de la pantalla  
    public final static int SCREEN_WIDTH = 176;  
    public final static int SCREEN_HEIGHT = 208;  
  
    // Dimensiones del sprite  
    public final static int SPRITE_WIDTH = 16;  
    public final static int SPRITE_HEIGHT = 22;  
  
    // Posicion inicial y velocidad del personaje  
    public final static int SPRITE_STEP = 2;  
    public final static int SPRITE_INI_X = 80;  
    public final static int SPRITE_INI_Y = 180;  
    public final static int SPRITE_END_Y = 20;  
  
    // Datos del texto de la pantalla de titulo  
    public static final int GAME_START_X = 88;  
    public static final int GAME_START_Y = 150;  
    public static final String GAME_START_TEXT =  
        "PULSA START PARA COMENZAR";  
    public static final int GAME_START_COLOR = 0x0FFFF00;  
    public static final Font GAME_START_FONT =  
        Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,  
            Font.SIZE_SMALL);  
  
    ...  
}
```

En ella definimos el tamaño de la pantalla, el tamaño de nuestro personaje, las coordenadas (x,y) donde ubicar distintos elementos del juego, etc. De esta forma, para hacer una nueva versión sólo tendremos que editar este fichero y dar a cada elemento el tamaño y las coordenadas adecuadas.

7.2.3. Optimización

Los juegos deben funcionar de manera fluida y dar una respuesta rápida al usuario para que estos resulten jugables y atractivos. Por lo tanto, será

conveniente optimizar el código todo lo posible, sobretodo en el caso de los dispositivos móviles en el que trabajamos con poca memoria y CPUs lentas.

No debemos cometer el error de intentar escribir un código optimizado desde el principio. Es mejor comenzar con una implementación clara, y una vez funcione esta implementación, intentar optimizar todo lo que sea posible.

Para optimizar el código deberemos detectar primero en qué lugar se está invirtiendo más tiempo. Por ejemplo, si lo que está ralentizando el juego es el volcado de los gráficos, deberemos optimizar esta parte, mientras que si es la lógica interna del juego la que requiere un tiempo alto, deberemos fijarnos en cómo optimizar este código.

El dibujado de los gráficos suele ser bastante costoso. Para optimizar este proceso deberemos intentar dibujar sólo aquello que sea necesario, es decir, lo que haya cambiado de un fotograma al siguiente. Muchas veces en la pantalla se está moviendo sólo un personaje pequeño, sería una pérdida de tiempo redibujar toda la pantalla cuando podemos repintar únicamente la zona en la que se ha movido este personaje. Esto lo podemos hacer con una variante del método `repaint` que nos permite redibujar sólo el área de pantalla indicada.

Por otro lado, dentro del código del juego deberemos utilizar las técnicas de optimización que conocemos, propias del lenguaje Java. Es importante intentar crear el mínimo número de objetos posibles, reutilizando objetos siempre que podamos. Esto evita el tiempo que requiere la instanciación de objetos y su posterior eliminación por parte del colector de basura.

También deberemos tener en cuenta que la memoria del móvil es muy limitada, por lo que deberemos permitir que se desechen todos los objetos que ya no necesitamos. Para que un objeto pueda ser eliminado por el colector de basura deberemos poner todas las referencias que tengamos a dicho objeto a `null`, para que el colector de basura sepa que ya nadie va a poder usar ese objeto por lo que puede eliminarlo.

7.3. Componentes de un videojuego

Cuando diseñemos un juego deberemos identificar las distintas entidades que encontraremos en él. Normalmente en los juegos de acción en 2D tendremos una pantalla del juego, que tendrá un fondo y una serie de personajes u objetos que se mueven en este escenario. Estos objetos que se mueven en el escenario se conocen como *sprites*. Además, tendremos un motor que se encargará de conducir la lógica interna del juego. Podemos abstraer los siguientes componentes:

- **Sprites:** Objetos o personajes que pueden moverse por la pantalla y/o con los que podemos interactuar.
- **Fondo:** Escenario de fondo, normalmente estático, sobre el que se desarrolla el juego. Muchas veces tendremos un escenario más grande que la pantalla, por lo que tendrá *scroll* para que la pantalla se desplace a la posición donde se encuentra nuestro personaje.

- **Pantalla:** En la pantalla se muestra la escena del juego. Aquí es donde se deberá dibujar todo el contenido, tanto el fondo como los distintos *sprites* que aparezcan en la escena y otros datos que se quieran mostrar.
- **Motor del juego:** Es el código que implementará la lógica del juego. En él se leerá la entrada del usuario, actualizará la posición de cada elemento en la escena, comprobando las posibles interacciones entre ellos, y dibujará todo este contenido en la pantalla.

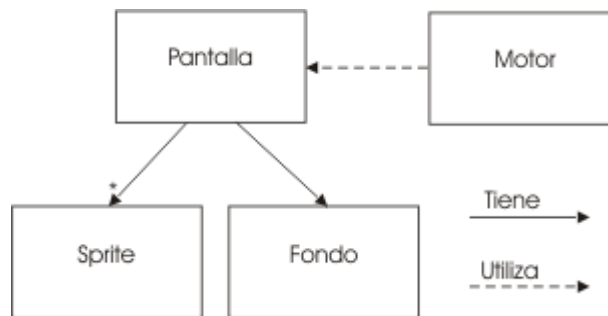


Figura 3. Componentes de un juego

A continuación veremos con más detalle cada uno de estos componentes, viendo como ejemplo las clases que MIDP 2.0 nos proporciona para crear cada uno de ellos.

Esto es lo que encontraremos en la pantalla de juego, mientras dure la partida. Sin embargo los juegos normalmente constarán de varias pantallas. Las más usuales son las siguientes:

- **Pantalla de presentación (*Splash screen*).** Pantalla que se muestra cuando cargamos el juego, con el logo de la compañía que lo ha desarrollado y los créditos. Aparece durante un tiempo breve (se puede aprovechar para cargar los recursos necesarios en este tiempo), y pasa automáticamente a la pantalla de título.
- **Título y menú.** Normalmente tendremos una pantalla de título principal del juego donde tendremos el menú con las distintas opciones que tenemos. Normalmente podremos comenzar una nueva partida, reanudar una partida anterior, ver las puntuaciones más altas, ver las instrucciones, o bien salir del juego.
- **Hi-score.** Pantalla de puntuaciones más altas obtenidas. Se mostrará el *ranking* de puntuaciones, donde aparecerá el nombre o iniciales de los jugadores junto a su puntuación obtenida.
- **Instrucciones.** Nos mostrará un texto con las instrucciones del juego.
- **Juego.** Será la pantalla donde se desarrolle el juego, que tendrá los componentes que hemos visto anteriormente.

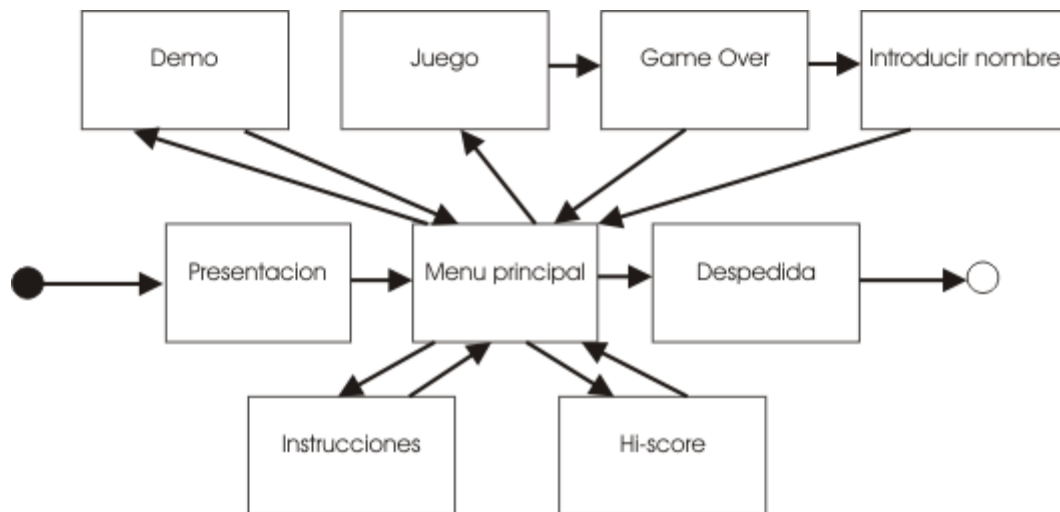


Figura 4. Mapa de pantallas típico de un juego

Hemos de decidir qué API utilizar para desarrollar cada pantalla. La pantalla de juego claramente debe realizarse utilizando la API de bajo nivel. Sin embargo con esta API hemos visto que no podemos introducir texto de una forma sencilla. Si utilizamos la API de alto nivel para la pantalla de puntuaciones más altas se nos facilitará bastante la tarea. El inconveniente es que siempre tendrá un aspecto más atractivo y propio del juego si utilizamos la API de bajo nivel, aunque nos cueste más programarla.

Es importante poder salir en todo momento del juego, cuando el usuario quiera de terminar de jugar. Durante la partida se deberá permitir volver al menú principal, o incluso salir directamente del juego. Para las acciones de salir y volver al menú se deben utilizar las teclas asociadas a las esquinas de la pantalla (*soft-keys*). No es recomendable utilizar estas teclas para acciones del juego.

7.4. Sprites

Los *sprites* hemos dicho que son todos aquellos objetos que aparecen en la escena que se mueven y/o podemos interactuar con ellos de alguna forma.

7.4.1. Animación

Estos objetos pueden estar animados. Para ello deberemos definir los distintos fotogramas (o *frames*) de la animación. Podemos definir varias animaciones para cada *sprite*, según las acciones que pueda hacer. Por ejemplo, si tenemos un personaje podemos tener una animación para andar hacia la derecha y otra para andar hacia la izquierda.

El *sprite* tendrá un determinado tamaño (ancho y alto), y cada fotograma será una imagen de este tamaño. Para no tener un número demasiado elevado de imágenes lo que haremos será juntar todos los fotogramas del *sprite* en una misma imagen, dispuestos como un mosaico.



Figura 5. Mosaico con los frames de un *sprite*

Cambiando el fotograma que se muestra del *sprite* en cada momento podremos animarlo. En MIDP 2.0 se proporciona la clase `Sprite` que nos permite manejar este tipo de mosaicos para definir los fotogramas del *sprite* y animarlo. Podemos crear el *sprite* de la siguiente forma:

```
Sprite personaje = new Sprite(imagen, ancho_fotograma,  
alto_fotograma);
```

Proporcionamos la imagen donde tenemos el mosaico de fotogramas, y definimos las dimensiones de cada fotograma. De esta forma esta clase se encargará de separar los fotogramas que hay dentro de esta imagen.

Cada fotograma tendrá un índice que se empezará a numerar a partir de cero. La ordenación de los *frames* en la imagen se realiza por filas y de izquierda a derecha, por lo que el *frame* de la esquina superior izquierda será el *frame* 0. Podemos establecer el *frame* a mostrar actualmente con:

```
personaje.setFrame(indice);
```

Podremos definir determinadas secuencias de *frames* para crear animaciones.

7.4.2. Desplazamiento

Además el *sprite* se podrá desplazar por la pantalla, por lo que deberemos tener algún método para moverlo. El *sprite* tendrá una cierta localización, dada en coordenadas (x,y) de la pantalla, y podremos o bien establecer una nuevas coordenadas o desplazar el *sprite* respecto a su posición actual:

```
personaje.setPosition(x, y);  
personaje.move(dx, dy);
```

Con el primer método damos la posición absoluta donde queremos posicionar el *sprite*. En el segundo caso indicamos un desplazamiento, para desplazarlo desde su posición actual. Normalmente utilizaremos el primer método para posicionarlo por primera vez en su posición inicial al inicio de la partida, y el segundo para moverlo durante el transcurso de la misma.

7.4.3. Colisiones

Otro aspecto de los *sprites* es la interacción entre ellos. Nos interesará saber cuando somos tocados por un enemigo o una bala para disminuir la vida, o cuando alcanzamos nosotros a nuestro enemigo. Para ello deberemos detectar las colisiones entre *sprites*. La colisión con *sprites* de formas complejas puede resultar costosa de calcular. Por ello se suele realizar el cálculo de colisiones con una forma aproximada de los *sprites* con la que esta operación resulte más

sencilla. Para ello solemos utilizar el *bounding box*, es decir, un rectángulo que englobe el *sprite*. La intersección de rectángulos es una operación muy sencilla, podremos comprobarlo simplemente con una serie de comparaciones menor que < y mayor que >.

La clase `Sprite` nos permite realizar esta comprobación de colisiones utilizando un *bounding box*. Podemos comprobar si nuestro personaje está tocando a un enemigo con:

```
personaje.collidesWith(enemigo, false);
```

Con el segundo parámetro a `true` le podemos decir que compruebe la colisión a nivel de *pixels*, es decir, que en lugar de usar el *bounding box* compruebe *pixel a pixel* si ambos *sprites* colisionan. Esto será bastante más costoso. Si necesitamos hacer la comprobación a este nivel, podemos comprobar primero si colisionan sus *bounding boxes* para descartar así de forma eficiente bastantes casos, y en caso de que los *bounding boxes* si que intersecten, hacer la comprobación a nivel de *pixels* para comprobar si realmente colisionan o no. Normalmente las implementaciones harán esto internamente cuando comprobemos la colisión a nivel de *pixels*.

7.4.4. Ejemplo

Vamos a ver como ejemplo la creación del *sprite* de nuestro personaje para el clon del *Frogger*. Para cada *sprite* podemos crear una subclase de `Sprite` que se especialice en el tipo de *sprite* concreto del que se trate.

```
public class CrocSprite extends Sprite {

    public final static int MOVE_UP = 0;
    public final static int MOVE_DOWN = 1;
    public final static int MOVE_LEFT = 2;
    public final static int MOVE_RIGHT = 3;
    public final static int MOVE_STAY = 4;

    int lastMove;

    public CrocSprite() {
        super(Resources.getImage(Resources.IMG_SPR_CROC),
              CommonData.SPRITE_WIDTH, CommonData.SPRITE_HEIGHT);
        this.defineCollisionRectangle(CommonData.SPRITE_CROP_X,
              CommonData.SPRITE_CROP_Y,
              CommonData.SPRITE_CROP_WIDTH,
              CommonData.SPRITE_CROP_HEIGHT);
        reset();
    }

    public void reset() {
        // Inicializa frame y movimiento actual
        lastMove = MOVE_STAY;
        this.setFrameSequence(null);
        this.setFrame(CommonData.SPRITE_STAY_UP);

        // Inicializa posición
```

```
        this.setPosition(CommonData.SPRITE_INI_X,
CommonData.SPRITE_INI_Y);
    }

    public void stay() {
        switch(lastMove) {
            case MOVE_UP:
                this.setFrameSequence(null);
                this.setFrame(CommonData.SPRITE_STAY_UP);
                break;
            case MOVE_DOWN:
                this.setFrameSequence(null);
                this.setFrame(CommonData.SPRITE_STAY_DOWN);
                break;
            case MOVE_LEFT:
                this.setFrameSequence(null);
                this.setFrame(CommonData.SPRITE_STAY_LEFT);
                break;
            case MOVE_RIGHT:
                this.setFrameSequence(null);
                this.setFrame(CommonData.SPRITE_STAY_RIGHT);
                break;
        }
        lastMove = MOVE_STAY;
    }

    public void die() {
        this.setFrameSequence(null);
        this.setFrame(CommonData.SPRITE_STAY_DIED);
        lastMove = MOVE_STAY;
    }

    public void moveUp() {
        if(lastMove != MOVE_UP) {
            this.setFrameSequence(CommonData.SPRITE_MOVE_UP);
        }
        lastMove = MOVE_UP;
        this.move(0, -CommonData.SPRITE_STEP);
        this.nextFrame();
    }

    public void moveDown() {
        if(lastMove != MOVE_DOWN) {
            this.setFrameSequence(CommonData.SPRITE_MOVE_DOWN);
        }
        lastMove = MOVE_DOWN;
        this.move(0, CommonData.SPRITE_STEP);
        this.nextFrame();
    }

    public void moveLeft() {
        if(lastMove != MOVE_LEFT) {
            this.setFrameSequence(CommonData.SPRITE_MOVE_LEFT);
        }
        lastMove = MOVE_LEFT;
        this.move(-CommonData.SPRITE_STEP, 0);
        this.nextFrame();
    }

    public void moveRight() {
        if(lastMove != MOVE_RIGHT) {
```

```
        this.setFrameSequence(CommonData.SPRITE_MOVE_RIGHT);
    }
    lastMove = MOVE_RIGHT;
    this.move(CommonData.SPRITE_STEP, 0);
    this.nextFrame();
}
}
```

En este ejemplo vemos como para los movimientos que requieran animación (como por ejemplo andar en las distintas direcciones) podemos indicar una secuencia de frames que componen esta animación con `setFrameSequence`, y mediante el método `nextFrame` ir pasando al siguiente.

Si queremos volver a disponer del conjunto de frames completo deberemos llamar al método `setFrameSequence` pasándole como parámetro `null`.

Otro problema que nos puede surgir es por ejemplo que no queramos que las colisiones se calculen con el rectángulo completo de nuestro sprite, sino sólo con un subrectángulo de éste. Por ejemplo, si en nuestro juego tenemos a un cocodrilo que debe cruzar la calle, no queremos que cuando un coche pise su cola se considere que lo han atropellado, el jugador se sentirá frustrado si cada vez que un coche roza su cola pierde una vida.

La parte inferior de la imagen del cocodrilo sólo contiene la cola, de forma que por defecto, como el rectángulo de colisión es del tamaño de la imagen, cuando haya un coche que colisione con esta parte inferior se considerará que nos han atropellado. Podemos establecer un nuevo rectángulo para el cálculo de colisiones con el método `defineCollisionRectangle`, de forma que ya no consideremos toda la imagen, sino sólo el área de la misma en la que está el cuerpo del cocodrilo.

7.5. Fondo

En los juegos normalmente tendremos un fondo sobre el que se mueven los personajes. Muchas veces los escenarios del juego son muy extensos y no caben enteros en la pantalla. De esta forma lo que se hace es ver sólo la parte del escenario donde está nuestro personaje, y conforme nos movamos se irá desplazando esta zona visible para enfocar en todo momento el lugar donde está nuestro personaje. Esto es lo que se conoce como *scroll*.

El tener un fondo con *scroll* será más costoso computacionalmente, ya que siempre que nos desplazemos se deberá redibujar toda la pantalla, debido a que se está moviendo todo el fondo. Además para poder dibujar este fondo deberemos tener una imagen con el dibujo del fondo para poder volcarlo en pantalla. Si tenemos un escenario extenso, sería totalmente prohibitivo hacer una imagen que contenga todo el fondo. Esta imagen podría llegar a ocupar el tamaño máximo permitido para los ficheros JAR es muchos móviles.

Para evitar este problema lo que haremos normalmente en este tipo de juegos es construir el fondo como un mosaico. Nos crearemos una imagen con los

elementos básicos que vamos a necesitar para nuestro fondo, y construiremos el fondo como un mosaico en el que se utilizan estos elementos.



Figura 6. Mosaico de elementos del fondo

Al igual que hacíamos con los *sprites*, tomaremos estos distintos elementos de una misma imagen. A cada elemento se le asociará un índice con el que lo referenciaremos posteriormente. El fondo lo crearemos como un mosaico del tamaño que queramos, en el que cada celda contendrá el índice del elemento que se quiere mostrar en ella.

La clase `TiledLayer` de MIDP 2.0 nos permite realizar esto. Debemos especificar el número de filas y columnas que va a tener el mosaico, y después la imagen que contiene los elementos y el ancho y alto de cada elemento.

```
TiledLayer fondo = new TiledLayer(columnas, filas, imagen, ancho, alto);
```

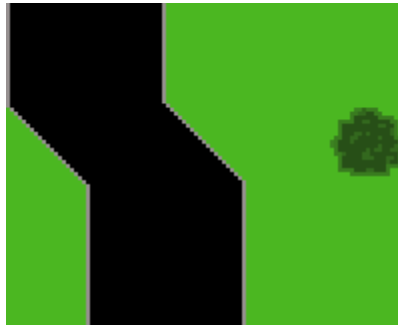


Figura 7. Índices de los elementos del mosaico

De esta forma el fondo tendrá un tamaño en *pixels* de $(columnas \times ancho) \times (filas \times alto)$. Ahora podemos establecer el elemento que contendrá cada celda del mosaico con el método:

```
fondo.setCell(columna, fila, indice);
```

Como `indice` especificaremos el índice del elemento en la imagen que queremos incluir en esa posición del mosaico, 0 si queremos dejar esa posición vacía con el color del fondo, o un número negativo para crear celdas con animación.



*Figura 8. Ejemplo de fondo
construido con los elementos
anteriores*

A continuación mostramos el código que podríamos utilizar para generar la carretera de nuestro juego:

```
public class Background extends TiledLayer {

    public Background() {
        super(CommonData.BG_H_TILES, CommonData.BG_V_TILES,
            Resources.getImage(Resources.IMG_BG_STAGE_1),
            CommonData.BG_TILE_WIDTH,
            CommonData.BG_TILE_HEIGHT);
    }

    public void reset(StageData stage) {
        TrackData [] tracks = stage.tracks;
        int nTracks = tracks.length;

        int row;

        // Filas superiores de césped
        for(row=0;row<CommonData.BG_V_TOP_TILES;row++) {
            for(int col=0;col<CommonData.BG_H_TILES; col++) {
                this.setCell(col, row, CommonData.BG_TILE_GRASS);
            }
        }

        // Margen superior de la carretera
        for(int col=0;col<CommonData.BG_H_TILES; col++) {
            this.setCell(col, row, CommonData.BG_TILE_TOP);
        }
        row++;

        // Parte interna de la carreteta
        for(;row<CommonData.BG_V_TOP_TILES + nTracks - 1;row++) {
            for(int col=0;col<CommonData.BG_H_TILES; col++) {
                this.setCell(col, row, CommonData.BG_TILE_CENTER);
            }
        }

        // Margen inferior de la carretera
        for(int col=0;col<CommonData.BG_H_TILES; col++) {
            this.setCell(col, row, CommonData.BG_TILE_BOTTOM);
        }
        row++;

        // Hierba en la zona inferior
```

```
for(;row<CommonData.BG_V_TILES;row++) {  
    for(int col=0;col<CommonData.BG_H_TILES; col++) {  
        this.setCell(col, row, CommonData.BG_TILE_GRASS);  
    }  
}  
}
```

7.6. Pantalla

En la pantalla se dibujarán todos los elementos anteriores para construir la escena del juego. De esta manera tendremos el fondo, nuestro personaje, los enemigos y otros objetos que aparezcan durante el juego, además de marcadores con el número de vidas, puntuación, etc.

La pantalla la vamos a dibujar por capas. Cada *sprite* y cada fondo que incluyamos será una capa, de esta forma poniendo una capa sobre otra construiremos la escena. Tanto la clase `Sprite` como la clase `TiledLayer` heredan de `Layer`, que es la clase que define de forma genérica las capas, por lo que podrán comportarse como tales. Todas las capas podrán moverse o cambiar de posición, para mover de esta forma su contenido en la pantalla.

Construiremos el contenido de la pantalla superponiendo una capa sobre otra. Tenemos la clase `LayerManager` en MIDP 2.0 que nos permitirá construir esta superposición de capas. Este objeto contendrá todo lo que se vaya a mostrar en pantalla, encargándose él internamente de gestionar y dibujar esta superposición de capas. Podemos crear este objeto con:

```
LayerManager escena = new LayerManager();
```

Ahora deberemos añadir por orden las capas que queramos que se muestren. El orden en el que las añadamos indicará el orden z, es decir, la profundidad de esta capa en la escena. La primera capa será la más cercana al punto de vista del usuario, mientras que la última será la más lejana. Por lo tanto, las primeras capas que añadamos quedarán por delante de las siguientes capas. Para añadir capas utilizaremos:

```
escena.append(personaje);  
escena.append(enemigo);  
escena.append(fondo);
```

También podremos insertar capas en una determinada posición de la lista, o eliminar capas de la lista con los métodos `insert` y `remove` respectivamente.

El área dibujada del `LayerManager` puede ser bastante extensa, ya que abarcará por lo menos toda la extensión del fondo que hayamos puesto. Deberemos indicar qué porción de esta escena se va a mostrar en la pantalla, especificando la ventana de vista. Moviendo esta ventana podremos implementar *scroll*. Podemos establecer la posición y tamaño de esta ventana con:

```
escena.setViewWindow(x, y, ancho, alto);
```

La posición de esta ventana de vista es referente al sistema de coordenadas de la escena (de la clase `LayerManager`).

Debemos tener en cuenta al especificar el tamaño del visor que diferentes modelos de móviles tendrán pantallas de diferente tamaño. Podemos hacer varias cosas para que el juego sea lo más portable posible. Podríamos crear un visor del tamaño mínimo de pantalla que vayamos a considerar, y en el caso de que la pantalla sea de mayor tamaño mostrar este visor centrado. Otra posibilidad es establecer el tamaño de la ventana de vista según la pantalla del móvil, haciendo que en los móviles con pantalla más grande se vea un mayor trozo del escenario.

Una vez hemos establecido esta ventana de vista, podemos dibujarla en el contexto gráfico `g` con:

```
escena.paint(g, x, y);
```

Donde daremos las coordenadas donde dibujaremos esta vista, dentro del espacio de coordenadas del contexto gráfico indicado.

7.7. Motor del juego

Hemos visto que los juegos suelen constar de varias pantallas. En la mayoría de los casos tenemos una pantalla de título, la pantalla en la que se desarrolla la partida, la pantalla de *game over*, una pantalla de demo en la que vemos el juego funcionar automáticamente como ejemplo, etc.

Normalmente lo primero que veremos será el título, de aquí podremos pasar al juego o al modo demo. Si transcurre un determinado tiempo sin que el usuario pulse ninguna tecla pasará a demo, mientras que si el usuario pulsa la tecla *start* comienza el juego. La demo finalizará pasado un tiempo determinado, tras lo cual volverá al título. El juego finalizará cuando el jugador pierda todas sus vidas, pasando a la pantalla de *game over*, y de ahí volverá al título.

7.7.1. Ciclo del juego

Vamos a centrarnos en cómo desarrollar la pantalla en la que se desarrolla la partida. Aquí tendremos lo que se conoce como ciclo del juego (o *game loop*). Se trata de un bucle infinito en el que tendremos el código fuente que implementa el funcionamiento del juego. Dentro de este bucle se efectúan las siguientes tareas básicas:

- **Leer la entrada:** Lee la entrada del usuario para conocer si el usuario ha pulsado alguna tecla desde la última iteración.
- **Actualizar escena:** Actualiza las posiciones de los *sprites* y su fotograma actual, en caso de que estén siendo animados, la posición del fondo si se haya producido *scroll*, y cualquier otro elemento del juego que deba cambiar. Para hacer esta actualización se pueden tomar diferentes criterios. Podemos mover el personaje según la entrada del usuario, la de los enemigos según su inteligencia artificial, o según las

interacciones producidas entre ellos y cualquier otro objeto (por ejemplo al ser alcanzados por un disparo, colisionando el *sprite* del disparo con el del enemigo), etc.

- **Redibujar:** Tras actualizar todos los elementos del juego, deberemos redibujar la pantalla para mostrar la escena tal como ha quedado en el instante actual.
- **Dormir:** Normalmente tras cada iteración dormiremos un determinado número de milisegundos para controlar la velocidad a la que se desarrolla el juego. De esta forma podemos establecer a cuantos fotogramas por segundo (*fps*) queremos que funcione el juego, siempre que la CPU sea capaz de funcionar a esta velocidad.

La clase `GameCanvas` es un tipo especial de *canvas* para el desarrollo de juegos presente en MIDP 2.0. Esta clase nos ofrecerá los métodos necesarios para realizar este ciclo del juego, ofreciéndonos un acceso a la entrada del usuario y un método de *render* adecuados para este tipo de aplicaciones. Para hacer la pantalla del juego crearemos una subclase de `GameCanvas` en la que introduciremos el ciclo del juego.

La forma en la que dibujaremos los gráficos utilizando esta clase será distinta a la que hemos visto para la clase `Canvas`. En el caso del `Canvas` utilizamos *render* pasivo, es decir, nosotros definimos en el método `paint` la forma en la que se dibuja y es el sistema el que llama a este método. Ahora nos interesa poder dibujar en cada iteración los cambios que se hayan producido en la escena directamente desde el bucle del ciclo del juego. Es decir, seremos nosotros los que decidamos el momento en el que dibujar, utilizaremos *render* activo. Para ello en cualquier momento podremos obtener un contexto gráfico asociado al `GameCanvas` desde dentro de este mismo objeto con:

```
Graphics g = getGraphics();
```

Este contexto gráfico estará asociado a un *backbuffer* del *canvas* de juegos. De esta forma durante el ciclo del juego podremos dibujar el contenido de la pantalla en este *backbuffer*. Cuando queramos que este *backbuffer* se vuelque a pantalla, llamaremos al método:

```
flushGraphics();
```

Existe también una versión de este método en la que especificamos la región que queremos que se vuelque, de forma que sólo tenga que volcar la parte de la pantalla que haya cambiado.

Vamos a ver ahora el esqueleto de un ciclo del juego básico que podemos realizar:

```
Graphics g = getGraphics();

while(true) {
    leeEntrada();
    actualizaEscena();
    dibujaGraficos(g);
}
```

```
flushGraphics();
}
```

Será conveniente dormir un determinado tiempo tras cada iteración para controlar así la velocidad del juego. Vamos a considerar que `CICLO` es el tiempo que debe durar cada iteración del juego. Lo que podremos hacer es obtener el tiempo que ha durado realmente la iteración, y dormir el tiempo restante hasta completar el tiempo de ciclo. Esto podemos hacerlo de la siguiente forma:

```
Graphics g = getGraphics();
long t1, t2, td;

while(true) {
    t1 = System.currentTimeMillis();

    leeEntrada();
    actualizaEscena();
    dibujaGraficos();

    flushGraphics();

    t2 = System.currentTimeMillis();
    td = t2 - t1;
    td = td < CICLO ? td : CICLO;

    try {
        Thread.sleep(CICLO - td);
    } catch (InterruptedException e) { }
}
```

Todo este bucle donde se desarrolla el ciclo del juego lo ejecutaremos como un hilo independiente. Un buen momento para poner en marcha este hilo será el momento en el que se muestre el *canvas*:

```
public class Juego extends GameCanvas implements Runnable {
    public void showNotify() {
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        // Ciclo del juego
        ...
    }
}
```

7.7.2. Máquina de estados

Hemos visto cómo implementar el ciclo del juego en el que en cada momento se lea la entrada del usuario, se actualice la escena, y se vuelquen los gráficos a la pantalla.

Sin embargo, este ciclo no siempre deberá comportarse de la misma forma. El juego podrá pasar por distintos estados, y en cada uno de ellos deberán realizarse una tareas e ignorarse otras.

Podemos modelar esto como una máquina de estados, en la que en cada momento, según el estado actual, se realicen unas funciones u otras, y cuando suceda un determinado evento, se pasará a otro estado.

Por ejemplo, en un juego podremos encontrar comúnmente los siguientes estados durante el desarrollo del mismo.

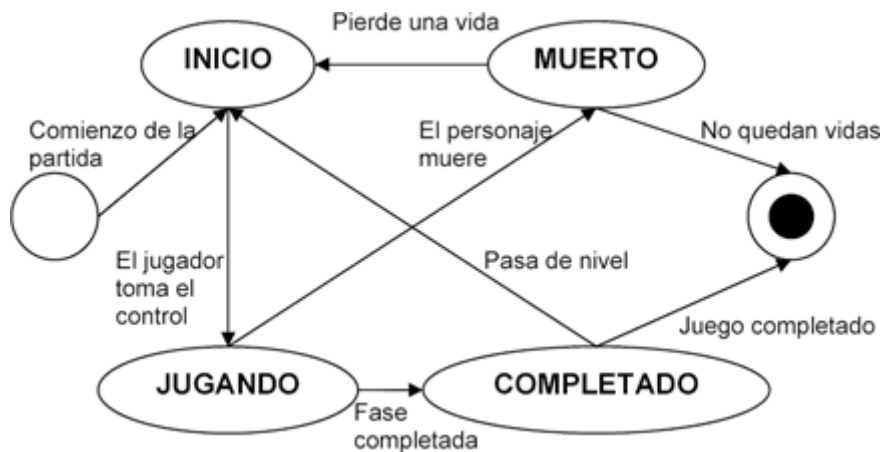


Figura 9. Diagrama de estados del juego

- **INICIO:** Al comienzo de la fase, se muestra durante un instante la fase en la que se va a jugar y el título de la misma para que el jugador se vaya preparando. En este momento todavía no se deja al usuario manejar al personaje. Pasado un determinado tiempo, pasará a estado **JUGANDO**.
- **JUGANDO:** En este estado el jugador tiene el control sobre el personaje. Si el jugador muere, pasará a estado **MUERTO**, mientras que si consigue completar el nivel pasará a estado **COMPLETADO**.
- **MUERTO:** Se muestra la muerte del personaje. Aquí el usuario ya no tendrá el control sobre el mismo. Una vez pasado un tiempo, nos restará una vida y pasará a estado de **INICIO**, o si no nos quedan vidas finalizará el juego.
- **COMPLETADO:** Se muestra un letrero en el que se indique se el jugador ha conseguido completar este nivel y posiblemente un resumen de lo conseguido en él, como por ejemplo la puntuación, el tiempo que ha sobrado, etc. Una vez pasado un tiempo, pasará a estado **INICIO** para la siguiente fase del juego, o bien si estábamos en la última fase finalizará el juego.

7.8. Entrada de usuario en juegos

7.8.1. Acciones de juegos en MIDP 1.0

La clase `Canvas` en MIDP 1.0 ofrece facilidades para leer la entrada de usuario para juegos. Hemos visto en el tema anterior que asocia a los códigos de las teclas lo que se conoce como acciones de juegos, lo cual nos facilitará el

desarrollo de aplicaciones que se controlan mediante estas acciones (arriba, abajo, izquierda, derecha, fuego), que principalmente son juegos.

Cuando se lee una tecla en cualquier evento de pulsación de teclas (`keyPressed`, `keyRepeated` o `keyReleased`), se nos proporciona como parámetro el código de dicha tecla. Podemos comprobar a qué acción de juego corresponde dicha tecla de forma independiente a la plataforma con el método `getGameAction` como se muestra en el siguiente ejemplo:

```
public void keyPressed(int keyCode) {  
  
    int action = getGameAction(keyCode);  
  
    if (action == LEFT) {  
        moverIzquierda();  
    } else if (action == RIGHT) {  
        moverDerecha();  
    } else if (action == FIRE) {  
        disparar();  
    }  
}
```

También podemos obtener la tecla principal asociada a una acción de juego con el método `getKeyCode`, pero dado que una acción puede estar asociada a varias teclas, si usamos este método no estaremos considerando las teclas secundarias asociadas a esa misma acción. A pesar de que en algunos ejemplos podamos ver código como el que se muestra a continuación, esto no debe hacerse:

```
public class MiCanvas extends Canvas {  
  
    // NO HACER ESTO!  
  
    int izq, der, fuego;  
  
    public MiCanvas() {  
        izq = getKeyCode(LEFT);  
        der = getKeyCode(RIGHT);  
        fuego = getKeyCode(FIRE);  
    }  
  
    public void keyPressed(int keyCode) {  
        if (keyCode == izq) {  
            moverIzquierda();  
        } else if (keyCode == der) {  
            moverDerecha();  
        } else if (keyCode == fuego) {  
            disparar();  
        }  
    }  
}
```

Hemos de tener en cuenta que muchos modelos de móviles no nos permiten mantener pulsadas más de una tecla al mismo tiempo. Hay otros que, aunque esto se permita, el *joystick* no puede mantener posiciones diagonales, sólo se puede pulsar una de las cuatro direcciones básicas al mismo tiempo.

Esto provoca que si no tenemos suficiente con estas cuatro direcciones básicas y necesitamos realizar movimientos en diagonal, tendremos que definir nosotros manualmente los códigos de las teclas para cada una de estas acciones. Esto reducirá portabilidad a la aplicación, será el precio que tendremos que pagar para poder establecer una serie de teclas para movimiento diagonal.

Si definimos nosotros las acciones directamente a partir de los códigos de teclas a bajo nivel deberemos intentar respetar en la medida de lo posible el comportamiento que suele tener cada tecla en los móviles. Por ejemplo, las teclas asociadas a las esquinas de la pantalla (*soft keys*) deben utilizarse para terminar el juego y volver al menú principal o bien salir directamente del juego.

7.8.2. Acceso al teclado con MIDP 2.0

Hasta ahora hemos visto las facilidades que nos ofrece MIDP para leer la entrada del usuario en los juegos desde su versión 1.0. Sin embargo, MIDP 2.0 incluye facilidades adicionales en la clase `GameCanvas`.

Podremos leer el estado de las teclas en cualquier momento, en lugar de tener que definir *callbacks* para ser notificados de las pulsaciones de las teclas. Esto nos facilitará la escritura del código, pudiendo obtener directamente esta información desde el ciclo del juego.

Podemos obtener el estado de las teclas con el método `getKeyStates` como se muestra a continuación:

```
int keyState = getKeyStates();
```

Esto nos devolverá un número entero en el que cada uno de sus *bits* representa una tecla. Si el *bit* vale 0 la tecla está sin pulsar, y si vale 1 la tecla estará pulsada. Tenemos definidos los *bits* asociados a cada tecla como constantes, que podremos utilizar como máscaras *booleanas* para extraer el estado de cada tecla a partir del número entero de estado obtenido:

<code>GameCanvas.LEFT_PRESSED</code>	Movimiento a la izquierda
<code>GameCanvas.RIGHT_PRESSED</code>	Movimiento a la derecha
<code>GameCanvas.UP_PRESSED</code>	Movimiento hacia arriba
<code>GameCanvas.DOWN_PRESSED</code>	Movimiento hacia abajo
<code>GameCanvas.FIRE_PRESSED</code>	Fuego

Por ejemplo, para saber si están pulsadas las teclas izquierda o derecha haremos la siguiente comprobación:

```
if ((keyState & LEFT_PRESSED) != 0) {  
    moverIzquierda();  
}  
  
if ((keyState & RIGHT_PRESSED) != 0) {  
    moverDerecha();  
}
```

```
}
```

Vamos a ver ahora un ejemplo de ciclo de juego sencillo completo:

```
public class CanvasJuego extends GameCanvas
                                implements Runnable {
    private final static int CICLO = 50;

    public CanvasJuego() {
        super(true);
    }

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

    public void run() {
        Graphics g = getGraphics();
        long t1, t2, td;

        // Carga sprites
        CrocSprite personaje = new CrocSprite();
        personaje.reset();

        // Crea fondo
        Background fondo = new
            Background(Resources.getStage(0));
        fondo.reset();

        // Crea escena
        LayerManager escena = new LayerManager();
        escena.append(personaje);
        escena.append(fondo);

        while(true) {
            t1 = System.currentTimeMillis();

            // Lee entrada del teclado
            int keyState = getKeyStates();

            if ((keyState & LEFT_PRESSED) != 0) {
                personaje.moveLeft();
            } else if ((keyState & RIGHT_PRESSED) != 0) {
                personaje.moveRight();
            } else if ((keyState & UP_PRESSED) != 0) {
                personaje.moveUp();
            } else if ((keyState & DOWN_PRESSED) != 0) {
                personaje.moveDown();
            }

            escena.paint(g);

            flushGraphics();

            t2 = System.currentTimeMillis();
            td = t2 - t1;
            td = td < CICLO ? td : CICLO;

            try {
                Thread.sleep(CICLO - td);
            }
        }
    }
}
```

```
        } catch (InterruptedException e) {}  
    }  
}
```

7.9. Ejemplo de motor de juego

Una vez hemos visto todos los elementos necesarios para desarrollar nuestro juego, vamos a ver un ejemplo de motor de juego completo que utilizaremos para nuestro clon de *Frogger*.

La siguiente clase será la encargada de realizar el ciclo básico del juego. En ella implementaremos la posibilidad de realizar pausas en el juego, lo cual ya hemos comentado que es muy importante en juegos para móviles:

```
public class GameEngine extends GameCanvas implements Runnable  
{  
  
    // Milisegundos que transcurren entre dos frames  
    // consecutivos  
    public final static int CICLO = 50;  
  
    // Codigos de las teclas soft  
    public final static int LEFT_SOFTKEY = -6;  
    public final static int RIGHT_SOFTKEY = -7;  
  
    // Escena actual  
    Scene escena;  
  
    // Estado de pausa  
    boolean isPaused;  
  
    // Hilo del juego  
    Thread t;  
  
    // Fuente pausa  
    Font font;  
  
    public GameEngine(Scene escena) {  
        super(false);  
        this.setFullScreenMode(true);  
  
        // Establece la escena actual  
        this.escena = escena;  
  
        // Inicializa fuente para el texto de la pausa  
        font = Font.getFont(Font.FACE_SYSTEM,  
                             Font.STYLE_BOLD, Font.SIZE_MEDIUM);  
    }  
  
    // Cambia la escena  
    public void setScene(Scene escena) {  
        this.escena = escena;  
    }  
  
    public void showNotify() {  
        start();  
    }  
}
```

```
public void hideNotify() {
    stop();
}

public void keyPressed(int keyCode) {

    if(isPaused) {
        if(keyCode == LEFT_SOFTKEY) {
            start();
        } else if(keyCode == RIGHT_SOFTKEY) {
            Resources.midlet.exitGame();
        }
    } else {
        if(keyCode == LEFT_SOFTKEY || keyCode == RIGHT_SOFTKEY)
        {
            stop();
        }
    }
}

// Pausa el juego
public synchronized void stop() {
    t = null;
    isPaused = true;

    Graphics g = getGraphics();
    render(g);
    flushGraphics();
}

// Reanuda el juego
public synchronized void start() {
    isPaused = false;

    t = new Thread(this);
    t.start();
}

// Ciclo del juego
public void run() {

    // Obtiene contexto gráfico
    Graphics g = getGraphics();

    while (t == Thread.currentThread()) {

        long t_ini, t_dif;
        t_ini = System.currentTimeMillis();

        // Lee las teclas
        int keyState = this.getKeyStates();

        // Actualiza la escena
        escena.tick(keyState);

        // Vuelca los graficos
        render(g);
        flushGraphics();

        t_dif = System.currentTimeMillis() - t_ini;
```

```

        // Duerme hasta el siguiente frame
        if (t_dif < CICLO) {
            try {
                Thread.sleep(CICLO - t_dif);
            } catch (InterruptedException e) { }
        }
    }
}

public void render(Graphics g) {

    escena.render(g);

    if(isPaused) {
        g.setColor(0x000000);
        for(int i=0;i<CommonData.SCREEN_HEIGHT;i+=2) {
            g.drawLine(0,i,CommonData.SCREEN_WIDTH,i);
        }

        g.setColor(0xFFFF00);
        g.setFont(font);
        g.drawString("Reanudar", 0, CommonData.SCREEN_HEIGHT,
                    Graphics.LEFT | Graphics.BOTTOM);
        g.drawString("Salir", CommonData.SCREEN_WIDTH,
                    CommonData.SCREEN_HEIGHT,
                    Graphics.RIGHT | Graphics.BOTTOM);
        g.drawString("PAUSADO", CommonData.SCREEN_WIDTH/2,
                    CommonData.SCREEN_HEIGHT/2,
                    Graphics.HCENTER | Graphics.BOTTOM);
    }
}
}

```

Con los métodos `stop` y `start` definidos en esta clase podremos pausar y reanudar el juego. En el estado pausado se detiene el hilo del ciclo del juego y en la pantalla se muestra el mensaje de que el juego ha sido pausado.

Podemos ver también que este ciclo de juego es genérico, deberemos proporcionar una clase que implemente la interfaz `Scene` en la que se especificará el comportamiento y el aspecto de la escena. Esta interfaz tiene los siguientes métodos:

```

public interface Scene
{
    public void tick(int k);
    public void render(Graphics g);
}

```

Con `tick` nos referimos a cada actualización instantánea de la escena. En cada ciclo se invocará una vez este método `tick` para que la escena se actualice según la entrada del usuario y las interacciones entre los distintos objetos que haya en ella. Por otro lado, en `render` definiremos la forma de dibujar la escena.

La escena para nuestro clon de *Frogger* será la siguiente:

```
public class GameScene implements Scene {

    public static int E_INICIO = 0;
    public static int E_JUGANDO = 1;
    public static int E_MUERTO = 2;
    public static int E_CONSEGUIDO = 3;

    int numLives;
    int state;
    int timer;
    int stage;

    CrocSprite croc;
    Background bg;
    Tracks tracks;
    Image face;

    Vector cars;

    public GameScene() {

        croc = new CrocSprite();
        bg = new Background();
        tracks = new Tracks();
        face = Resources.getImage(Resources.IMG_FACE_LIVES);

        cars = new Vector();

        reset();
    }

    // Reiniciado de la partida
    public void reset() {
        numLives = CommonData.NUM_LIVES;
        stage = 0;

        reset(stage);
    }

    // Reiniciado de una fase
    public void reset(int nStage) {
        StageData stage = Resources.stageData[nStage];
        croc.reset();
        bg.reset(stage);
        tracks.reset(stage);

        cars.removeAllElements();

        this.setState(E_INICIO);
    }

    // Cambia el estado actual del juego
    public void setState(int state) {
        this.state = state;

        if(state==E_INICIO) {
            timer = 50;
        } else if(state==E_MUERTO) {
            timer = 30;
            croc.die();
        } else if(state==E_CONSEGUIDO) {
```

```
        timer = 50;
    }
}

public void tick(int keyState) {

    // Decrementa el tiempo de espera
    if(state==E_INICIO || state==E_MUERTO ||
state==E_CONSEGUIDO) {
        timer--;
    }

    // Comienza el juego
    if(state==E_INICIO && timer <= 0) {
        this.setState(E_JUGANDO);
    }

    // Reinicia el nivel o termina el juego
    if(state==E_MUERTO && timer <= 0) {
        numLives--;
        if(numLives<0) {
            Resources.midlet.showTitle();
        } else {
            this.reset(stage);
            return;
        }
    }

    // Pasa de fase
    if(state==E_CONSEGUIDO && timer <= 0) {
        stage++;
        if(stage >= Resources.stageData.length) {
            stage = 0;
        }

        this.reset(stage);
        return;
    }

    // Permite controlar el personaje si el estado es JUGANDO
    if(state==E_JUGANDO) {

        // Control del sprite
        if( (keyState & GameCanvas.UP_PRESSED) !=0
            && croc.getY() > 0 ) {
            croc.moveUp();
        } else if( (keyState & GameCanvas.DOWN_PRESSED) !=0
            && croc.getY() < CommonData.SCREEN_HEIGHT
            - CommonData.SPRITE_HEIGHT) {
            croc.moveDown();
        } else if( (keyState & GameCanvas.LEFT_PRESSED) !=0
            && croc.getX() > 0) {
            croc.moveLeft();
        } else if( (keyState & GameCanvas.RIGHT_PRESSED) !=0
            && croc.getX() < CommonData.SCREEN_WIDTH
            - CommonData.SPRITE_WIDTH) {
            croc.moveRight();
        } else {
            croc.stay();
        }
    }
}
```

```
// Crea nuevos coches en los carriles si ha llegado el
momento
tracks.checkTracks(cars);

// Actualiza coches y comprueba colisiones
Enumeration enum = cars.elements();
while(enum.hasMoreElements()) {
    CarSprite car = (CarSprite)enum.nextElement();
    car.tick();

    if(state == E_JUGANDO && car.collidesWith(croc, false))
    {
        this.setState(E_MUERTO);
    }
}

// Ha conseguido cruzar la carretera
if(state == E_JUGANDO &&
croc.getY() < CommonData.SPRITE_END_Y) {
    this.setState(E_CONSEGUIDO);
}

return;
}

public void render(Graphics g) {

    bg.paint(g);
    croc.paint(g);

    Enumeration enum = cars.elements();
    while(enum.hasMoreElements()) {
        CarSprite car = (CarSprite)enum.nextElement();
        car.paint(g);
    }

    for(int i=0;i<this.numLives;i++) {
        g.drawImage(face, i*CommonData.FACE_WIDTH, 0,
            Graphics.TOP | Graphics.LEFT);
    }

    if(state==E_INICIO) {
        g.setFont(CommonData.STAGE_TITLE_FONT);
        g.setColor(CommonData.STAGE_TITLE_COLOR);
        g.drawString(Resources.stageData[stage].title,
            CommonData.STAGE_TITLE_X, CommonData.STAGE_TITLE_Y,
            Graphics.HCENTER | Graphics.TOP);
    }

    if(state==E_CONSEGUIDO) {
        g.setFont(CommonData.STAGE_TITLE_FONT);
        g.setColor(CommonData.STAGE_TITLE_COLOR);
        g.drawString(CommonData.STAGE_COMPLETED_TEXT,
            CommonData.STAGE_TITLE_X, CommonData.STAGE_TITLE_Y,
            Graphics.HCENTER | Graphics.TOP);
    }
}
}
```


Los métodos `reset` nos permiten reiniciar el juego. Si no proporcionamos ningún parámetro, se reiniciará el juego desde el principio, nos servirá para empezar una nueva partida. Si proporcionamos como parámetro un número de fase, comenzaremos a jugar desde esa fase con las vidas que nos queden actualmente. Esta segunda forma nos servirá para empezar de nuevo una fase cuando nos hayan matado, o cuando hayamos pasado al siguiente nivel.

De esta forma no tendremos que instanciar nuevos objetos, sino que simplemente llamaremos a este método para reiniciar los valores del objeto actual.

Por otro lado podemos observar en este ejemplo cómo se ha modelado la máquina de estados para el juego. Tenemos los estados `E_INICIO`, `E_JUGANDO`, `E_MUERTO` y `E_CONSEGUIDO`. El estado `E_JUGANDO` durará hasta que nos maten o hayamos pasado a la siguiente fase. Los demás estados durarán un tiempo fijo durante el cual se mostrará algún gráfico o mensaje en pantalla, y una vez pasado este tiempo se pasará a otro estado.

Podemos ver que en `tick` es donde se implementa toda la lógica del juego. Dentro de este método podemos ver las distintas comprobaciones y acciones que se realizan, que variarán según el estado actual.

En `render` podemos ver también algunos elementos que se dibujarán sólo en determinados estados.

8. Sonido y multimedia

Hemos visto que MIDP 1.0 carece de soporte para la reproducción de sonido. Esta carencia se soluciona en MIDP 2.0 que proporciona soporte para la reproducción de distintos tipos de sonidos y músicas. Esta API que incorpora MIDP 2.0 es un subconjunto de la API multimedia MMAPI, que se ofrece como API opcional para los dispositivos MIDP. Esta API multimedia, a parte de sonido, nos permite reproducir video, y además con ella podremos capturar sonido, imágenes o video utilizando el micrófono y la cámara del móvil.

Vamos a comenzar viendo cómo reproducir sonido, cosa que podremos hacer tanto en los dispositivos MIDP 2.0 como en los dispositivos MIDP 1.0 que incorporen la API opcional MMAPI. Después de esto veremos como reproducir y capturar otros medios utilizando las características que sólo están disponibles en MMAPI, y que podrán implementar los modelos de teléfonos móviles multimedia.

Esta API multimedia se encuentra en el paquete `javax.microedition.media` y subpaquetes. En MIDP 2.0 tendremos sólo el subconjunto de clases de la API MMAPI necesarias para la reproducción de sonido.

8.1. Reproductor de medios

Para poder reproducir los distintos tipos de medios (sonidos, músicas, videos) deberemos utilizar un objeto reproductor de medios, al que referenciaremos mediante la interfaz `Player`. Para obtener el reproductor adecuado para un determinado medio utilizaremos la clase `Manager`. Esta clase ofrece una serie de métodos estáticos que nos permiten crear objetos reproductores para los medios que queramos reproducir.

Para crear un reproductor utilizaremos el método estático `createPlayer` de la clase `Manager`. Podemos crear el reproductor a partir de un flujo de datos abierto de donde leer el medio, o a partir de una URL que nos sirva para localizar este medio:

```
// A partir de un flujo de datos y un tipo
InputStream in = abrirFlujo();
Player player = Manager.createPlayer(in, tipo);

// A partir de una URL
Player player = Manager.createPlayer(url);
```

Cuando creamos el reproductor a partir del flujo de datos deberemos indicar explícitamente el tipo de medio del que se trata, ya que del flujo podrá leer el contenido pero no conocerá qué tipo de medio está leyendo. En el caso de la URL no será necesario, ya que podrá obtener información sobre el tipo de contenido conectando a dicha URL.

Si queremos abrir un medio almacenado en un fichero dentro del JAR de la aplicación podremos obtener un flujo para leer de dicho recurso con

`getResourceAsStream`, y crear un reproductor para reproducir el contenido leído a través de este flujo. En este caso deberemos especificar el tipo MIME del medio que estamos abriendo. Cada implementación de MMAPI puede soportar tipos de datos distintos. Por ejemplo, posibles tipos que podemos encontrar son:

<code>audio/x-wav</code>	Ficheros WAV
<code>audio/basic</code>	Ficheros AU
<code>audio/mpeg</code>	Ficheros MP3
<code>audio/midi</code>	Ficheros MIDI
<code>audio/x-tone-seq</code>	Secuencias de tonos
<code>video/mpeg</code>	Videos MPEG
<code>video/3gpp</code>	Videos 3GPP

Utilizando URLs podremos, además de abrir estos tipos de ficheros de medios proporcionando la URL donde se localizan, acceder a dispositivos de captura y reproducir *streams* de audio y video en tiempo real. Estas URLs tendrán la siguiente forma:

`protocolo:dirección`

Por ejemplo, podemos reproducir un sonido que haya disponible en Internet proporcionando su URL:

```
Player player =  
    Manager.createPlayer("http://j2ee.ua.es/pdm/sonido.wav");
```

Normalmente en este caso descargará el medio desde Internet y lo almacenará en un buffer temporal para reproducirlo. Podemos utilizar otras URLs, como por ejemplo `capture://dispositivo` para acceder a un dispositivo de captura (micrófono o cámara), o `rtp://direccion` para conectarnos a un servidor de *streaming* de audio o video. El soporte para la captura, *streaming*, o diferentes tipos de medios dependerá de cada modelo concreto de móvil.

8.1.1. Estados

La reproducción de muchos de estos tipos de medios consume gran cantidad de recursos. Deberemos almacenar el medio a reproducir en memoria, y cuando vayamos a reproducirlo necesitamos reservar una serie de recursos de forma exclusiva, como puede ser por ejemplo el altavoz del dispositivo. Dado que estos recursos son escasos, los reproductores nos permitirán decidir cuando los reservamos y cuando los liberamos.

Para controlar esta asignación de recursos, los reproductores pasarán por una serie de estados. En cada uno de ellos se habrán reservado una serie de recursos. Cambiando el estado actual podremos controlar este proceso de reserva de recursos por parte del reproductor. Hemos de tener en cuenta que si hemos reservado todos los recursos necesarios podremos empezar a reproducir el medio de forma casi inmediata, mientras que si todavía no se han

reservado y queremos reproducir el sonido, el comienzo de la reproducción puede tener un cierto retardo debido a que previamente deberá adquirir los recursos necesarios, lo cuál llevará un tiempo.

Los estados por los que pasará el reproductor son los siguientes:

- **Unrealized:** Nada más crear el reproductor se encontrará en este estado. En este estado todavía no se ha reservado ningún recurso. En este estado no podremos obtener ninguna información sobre los medios a reproducir. Podremos pasar al siguiente estado invocando el método `realize`.
- **Realized:** Normalmente en este estado el reproductor habrá adquirido todos los recursos que necesita para la reproducción del medio excepto aquellos que sean de acceso exclusivo, como puede ser el altavoz del móvil. Se habrá leído ya el medio a reproducir, por lo que a partir de este momento podremos obtener información sobre él. Para pasar el siguiente estado invocaremos el método `prefetch`.
- **Prefetched:** En el estado *realized* todavía no tenemos todos los recursos necesarios para empezar a reproducir el medio. Será al pasar a estado *prefetched* cuando obtengamos estos recursos. Esto implica por ejemplo reservar los recursos de uso exclusivo necesario, crear los *bufferes* de datos necesarios, etc. Cuando estemos en estado *prefetched* podremos empezar a reproducir el medio en cualquier momento, ya que tenemos todos los recursos necesarios. Si queremos empezar a reproducir el medio utilizaremos el método `start`. Si por el contrario, queremos liberar los recursos de uso exclusivo y volver a estado *realized*, utilizaremos `deallocate`.
- **Reproduciendo:** Nos encontraremos en este estado mientras se esté reproduciendo el medio. Cuando termine la reproducción se volverá automáticamente al estado *prefetched*. También podemos detener manualmente la reproducción invocando el método `stop`.
- **Cerrado:** Desde cualquiera de los estados anteriores podemos cerrar el reproductor invocando el método `close`. Con esto liberaremos todos los recursos y ya no podremos utilizar más el reproductor.

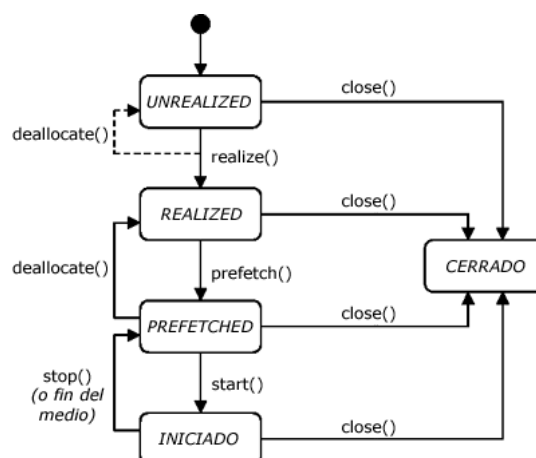


Figura 1. Diagrama de estados del reproductor

No tendremos que pasar por todos estos estados manualmente. Podremos llamar a `start` sin haber llamado antes a `realize` y `prefetch`. De esta forma en la misma llamada a `start` se pasará implícitamente por los estados *realized* y *prefetched*.

Podremos obtener el estado en el que se encuentra el reproductor en cada momento con el método:

```
int estado = player.getState();
```

8.1.2. Información y control del medio

Una vez en estado *realized*, podremos obtener información sobre el medio y controlar la forma en la que se va a reproducir. El objeto `Player` nos ofrecerá una serie de métodos con los que podremos acceder a esta información. Podemos obtener el tipo MIME del contenido del medio con:

```
String tipo = player.getContentType();
```

Podemos obtener la duración del medio en milisegundos cuando se reproduce a velocidad normal con:

```
long tiempo = player.getDuration();
```

Si no es posible obtener el tiempo del medio nos devolverá `Player.TIME_UNKNOWN`. Podemos modificar el tiempo de reproducción, de forma que el medio se reproduzca más lentamente o más rápidamente para ajustarse al nuevo tiempo. Podemos establecer u obtener este tiempo de reproducción con:

```
long tiempo = player.getMediaTime();  
player.setMediaTime(tiempo);
```

Podemos también hacer que el medio se reproduzca de forma cíclica. Para ello deberemos indicar el número de ciclos que queremos que se reproduzca con:

```
player.setLoopCount(int numero);
```

Por defecto los medios se reproducirán una sola vez. Si queremos que se reproduzca en un ciclo infinito como número debemos especificar `-1`. Este método lo deberemos ejecutar siempre antes de que el medio haya empezado a reproducirse, nunca deberemos ejecutarlo en estado de reproducción ya que esto producirá una excepción.

Todas estas características son genéricas para todos los medios y por lo tanto podemos acceder a ellas a través del objeto `Player`. Si queremos acceder a otras características propias sólo de un determinado tipo de medios deberemos utilizar controles.

8.1.3. Controles de medios

Podemos encontrar controles para controlar determinadas características de los medios. Según el tipo de medio del que se trate tendremos disponibles unos controles u otros. Por ejemplo, para los clips de audio tendremos disponibles controles para cambiar el volumen o los tonos reproducidos, mientras que para secuencias de video tendremos controles que nos permitan mostrar el video en la pantalla.

Podremos obtener estos controles a través del reproductor proporcionando el nombre del control al que queremos acceder, o bien obtener la lista completa de controles disponibles:

```
Control [] controles = player.getControls();
Control control = player.getControl(nombre);
```

Para obtener estos controles deberemos estar por lo menos en estado *realized*. Deberemos hacer una conversión *cast* al tipo de control concreto que estemos obteniendo en cada momento, como puede ser `VolumeControl`, `ToneControl` o `VideoControl`, para poder acceder a las características que controla cada uno.

8.1.4. Listener de medios

Podemos utilizar un *listener* sobre el reproductor de medios para conocer cuándo suceden determinados eventos, como por ejemplo cuándo terminan de reproducirse los medios. Este *listener* lo crearemos en una clase que herede de `PlayerListener`:

```
public class MiListener implements PlayerListener {
    public void playerUpdate(Player player,
                             String evento, Object datos) {

        if(evento == PlayerListener.STARTED) {
            // Ha comenzado la reproduccion
        } else if(evento == PlayerListener.STOPPED) {
            // Se ha detenido la reproduccion
        } else
            ...
    }
}
```

Cuando suceda un evento en el reproductor, se invocará el método `playerUpdate` que hayamos definido proporcionándonos como parámetro el reproductor sobre el que se ha producido el evento, el tipo de evento y una serie de datos que podemos tener asociados al evento. Los tipos de eventos están definidos como constantes en la clase `PlayerListener`. Entre ellos podemos encontrar los siguientes:

<code>PlayerListener.STARTED</code>	Ha comenzado la reproducción
<code>PlayerListener.STOPPED</code>	Se ha detenido la reproducción
<code>PlayerListener.END_OF_MEDIA</code>	Se ha llegado al final del medio
<code>PlayerListener.VOLUME_CHANGED</code>	Se ha cambiado el volumen del

	dispositivo
<code>PlayerListener.CLOSED</code>	Se ha cerrado el reproductor
<code>PlayerListener.ERROR</code>	Se ha producido un error en el reproductor

Una vez creado el *listener*, para que empiece a funcionar deberemos registrarlo en el reproductor con:

```
player.addPlayerListener(new MiListener());
```

8.2. Reproducción de tonos

Vamos a comenzar viendo la reproducción básica de tonos en el altavoz del móvil. Una forma sencilla de reproducir un tono individual es invocando el método:

```
Manager.playTone(nota, duracion, volumen);
```

Aquí deberemos especificar la nota que queremos tocar, su duración en milisegundos y el volumen que será un valor entero de 0 a 100.

Para reproducir una secuencia de tonos ya deberemos crear un reproductor. Este reproductor se creará con una URL específica que representa el dispositivo de reproducción de tonos del móvil. Lo haremos de la siguiente forma:

```
Player player =  
Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
```

Para poder establecer la secuencia de tonos deberemos crear un control de tonos asociado al reproductor de la siguiente forma:

```
player.realize();  
ToneControl tc =  
(ToneControl)player.getControl("ToneControl");  
tc.setSequence(new byte[] {  
    ToneControl.VERSION, 1,  
    ToneControl.TEMPO, 30,  
    ToneControl.C4, 16,  
    ToneControl.C4+2, 16, //D4  
    ToneControl.C4+4, 16, //E4  
    ToneControl.C4+5, 16, //F4  
    ToneControl.C4+7, 16, //G4  
    ToneControl.C4+9, 16, //A4  
    ToneControl.C4+11, 16, //B4  
    ToneControl.C4+9, 8, //A4  
    ToneControl.C4+7, 8, //G4  
    ToneControl.C4+5, 8, //F4  
    ToneControl.C4+4, 8, //E4  
    ToneControl.C4+2, 8, //D4  
    ToneControl.C4, 8  
});
```

Como secuencia de tonos debemos crear un *array* de *bytes* en el que indicaremos la secuencia de notas y una serie de parámetros como el *tempo*. En esta secuencia se especifican las notas y su duración en posiciones consecutivas.

Hemos de destacar que es necesario estar en estado *realized* para poder obtener los controles asociados al reproductor. Por esta razón hemos realizado previamente la transición a este estado.

Una vez establecida esta secuencia podremos reproducirla con:

```
player.start();
```

8.3. Música y sonidos

Vamos a ver ahora cómo reproducir música y efectos de sonido utilizando el altavoz del móvil. Estos medios podemos leerlos por ejemplo de ficheros WAV o MIDI. Los ficheros MIDI serán adecuados para reproducir una música de fondo en nuestras aplicaciones, mientras que los ficheros WAV son más apropiados para efectos de sonido cortos.

De esta forma podemos poner música y efectos de sonido por ejemplo a los juegos que hagamos. Debemos tener en cuenta que algunos modelos de móviles no permiten realizar mezclado de medios, es decir, no podemos reproducir más de un medio al mismo tiempo. En este caso no podríamos hacer sonar efectos de sonido mientras oímos una música de fondo.

Para crear un reproductor para reproducir música o efectos de sonido deberemos crearlo a partir de un fichero de este tipo (WAV, AU, MID, MP3, etc). Podremos especificarlo o bien como una URL o como un flujo de entrada:

```
// A partir de una URL (fichero remoto en internet)
Player player =
    Manager.createPlayer("http://j2ee.ua.es/pdm/musica.mid");
player.start();

// A partir de un fichero local
InputStream in =
    getClass().getResourceAsStream("/musica.mid");
Player player = Manager.createPlayer(in, "audio/midi");
player.start();
```

Para todos los medios que reproduzcan audio podremos obtener un control de volumen que nos permitirá cambiar el volumen del altavoz, o bien silenciarlo por completo. Para obtener este control haremos lo siguiente:

```
VolumeControl vol =
    (VolumeControl)player.getControl("VolumeControl");
```

Una vez tengamos este control de volumen podremos cambiar el volumen o silenciarlo con los siguientes métodos:


```
vol.setLevel(volumen);  
vol.setMute(true);
```

El nivel de volumen será un número entero de 0 a 100. Tendremos también otros dos métodos que nos permitirán consultar el nivel del volumen y si está silenciado o no.

8.4. Reproducción de video

La API multimedia incorporada en MIDP 2.0 sólo soporta audio, como hemos visto anteriormente. Sin embargo los teléfonos que incorporen la API completa MMAPI podrán reproducir también video. Podremos crear un reproductor de video de la misma forma que los de audio, utilizando en este caso un fichero de video (como por ejemplo MPEG, o 3GPP). Podremos crearlo de la siguiente forma:

```
InputStream in = getClass().getResourceAsStream("/video.3gp");  
Player player = Manager.createPlayer(in, "video/3gpp");
```

Sin embargo, con esto todavía no podremos mostrar el video en pantalla. Para poder hacer esto necesitaremos obtener un control de video que nos permita mostrar el video en algún área de la pantalla del móvil. Para ello pasaremos a estado *realized* y obtendremos en control necesario:

```
player.realize();  
VideoControl vc =  
(VideoControl)player.getControl("VideoControl");
```

Ahora podremos añadir este video de distintas formas. Podemos o bien añadirlo a un *canvas*, o bien como elemento de un formulario. La primera forma nos permitirá tener un mayor control a bajo nivel sobre cómo se muestra el video en la pantalla.

Para añadirlo como primitiva de alto nivel a un formulario haremos lo siguiente:

```
Item item =  
    (Item)vc.initDisplayMode(VideoControl.USE_GUI_PRIMITIVE,  
    null);  
formulario.append(item);  
player.start();
```

Mientras que para visualizarlo en un *canvas* lo haremos de la siguiente forma:

```
vc.initDisplayMode(VideoControl.USE_DIRECT_VIDEO, canvas);  
vc.setVisible(true);  
player.start();
```



Figura 2. Reproductor de video

8.5. Captura

La API multimedia también nos permitirá capturar audio o video a través del micrófono o de la cámara que incorpore el teléfono móvil. Para acceder a estos dispositivos de captura utilizaremos una URL como la siguiente:

```
capture://dispositivo
```

Por ejemplo, con las siguientes URLs:

```
capture://audio
capture://video
capture://audio_video
```

Crearemos un reproductor para capturar audio, video o audio y video respectivamente. Con estas URLs se capturará con el formato que tengan estos medios por defecto. Además, en esta URL podremos añadir parámetros para establecer el formato y codificación de la captura. Por ejemplo, las siguientes URLs:

```
capture://audio?encoding=pcm
capture://video?width=160&height=120
```

Nos servirán para capturar audio con formato PCM y video con resolución de 160x120 *pixels* respectivamente.

Podremos crear un reproductor utilizando estas URLs igual que para los anteriores tipos de medios:

```
Player player = Manager.createPlayer("capture://video");
```

Ahora podremos mostrar el video capturado en pantalla igual que hemos visto anteriormente para la reproducción de ficheros de video:

```
player.realize();
VideoControl vc =
    (VideoControl)player.getControl("VideoControl");
```

```
vc.initDisplayMode(VideoControl.USE_DIRECT_VIDEO, canvas);  
vc.setVisible(true);  
player.start();
```

8.5.1. Grabación de medios

Normalmente nos interesará poder grabar el medio que estemos capturando. Para grabar un medio deberemos obtener un control de grabación `RecordControl`:

```
RecordControl rc =  
(RecordControl)player.getControl("RecordControl");
```

Ahora deberemos decidir donde grabar el medio. Deberemos proporcionar un flujo de salida para que almacene el video a través de él. Por ejemplo, podemos hacer que lo almacene en memoria asignando el siguiente flujo de salida:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();  
rc.setRecordStream(out);
```

Suponiendo que el medio ya se está reproduciendo, podemos comenzar a grabarlo en el flujo proporcionado con:

```
rc.startRecord();
```

Podemos detener la grabación con:

```
rc.stopRecord();
```

Después de detenerse puede reanudarse volviendo a llamar a `startRecord`. Si queremos finalizar la grabación, deberemos ejecutar:

```
rc.commit();
```

De esta forma se finalizará la escritura del video y en este caso ya no podremos reanudarlo.

8.5.2. Captura de imágenes

Mientras se reproduce el video capturado por la cámara o cualquier otro video podemos también capturar imágenes, de forma que se comporte como una cámara de fotos. Para capturar fotografías con la cámara utilizaremos el siguiente método del control de video:

```
byte [] img_png = vc.getSnapshot(null);
```

Con esto obtendremos la imagen en el formato por defecto, que es el formato PNG que soportan todos los móviles. Si queremos podemos especificar como parámetro de este método el formato en el que queremos que se capture la imagen.

Una vez tenemos los *bytes* de la imagen PNG, podremos crear un objeto `Image` a partir de ellos con:

```
Image img = Image.createImage(img_png, 0, img_png.length);
```

9. Almacenamiento persistente

Muchas veces las aplicaciones necesitan almacenar datos de forma persistente. Cuando realizamos aplicaciones para PCs de sobremesa o servidores podemos almacenar esta información en algún fichero en el disco o bien en una base de datos. Lo más sencillo será almacenarla en ficheros, pero en los dispositivos móviles no podemos contar ni tan solo con esta característica. Aunque los móviles normalmente tienen su propio sistema de ficheros, por cuestiones de seguridad MIDP no nos dejará acceder directamente a él. Es posible que en alguna implementación podamos acceder a ficheros en el dispositivo, pero esto no es requerido por la especificación, por lo que si queremos que nuestra aplicación sea portable no deberemos confiar en esta característica.

Para almacenar datos de forma persistente en el móvil utilizaremos RMS (*Record Management System*). Se trata de un sistema de almacenamiento que nos permitirá almacenar registros con información de forma persistente en los dispositivos móviles. No se especifica ninguna forma determinada en la que se deba almacenar esta información, cada implementación deberá guardar estos datos de la mejor forma posible para cada dispositivo concreto, utilizando memoria no volátil, de forma que no se pierda la información aunque reiniciemos el dispositivo o cambiemos las baterías. Por ejemplo, algunas implementaciones podrán utilizar el sistema de ficheros del dispositivo para almacenar la información de RMS, o bien cualquier otro dispositivo de memoria no volátil que contenga el móvil. La forma de almacenamiento real de la información en el dispositivo será transparente para los MIDlets, éstos sólo podrán acceder a la información utilizando la API de RMS. Esta API se encuentra en el paquete `javax.microedition.rms`.

9.1. Almacenes de registros

La información se almacena en almacenes de registros (*Record Stores*), que serán identificados con un nombre que deberemos asignar nosotros. Cada aplicación podrá crear y utilizar tantos almacenes de registros como quiera. Cada almacén de registros contendrá una serie de registros con la información que queramos almacenar en ellos.

Los almacenes de registros son propios de la suite. Es decir, los almacenes de registro creados por un MIDlet dentro de una suite, serán compartidos por todos los MIDlets de esa suite, pero no podrán acceder a ellos los MIDlets de suites distintas. Por seguridad, no se permite acceder a recursos ni a almacenes de registros de suites distintas a la nuestra.

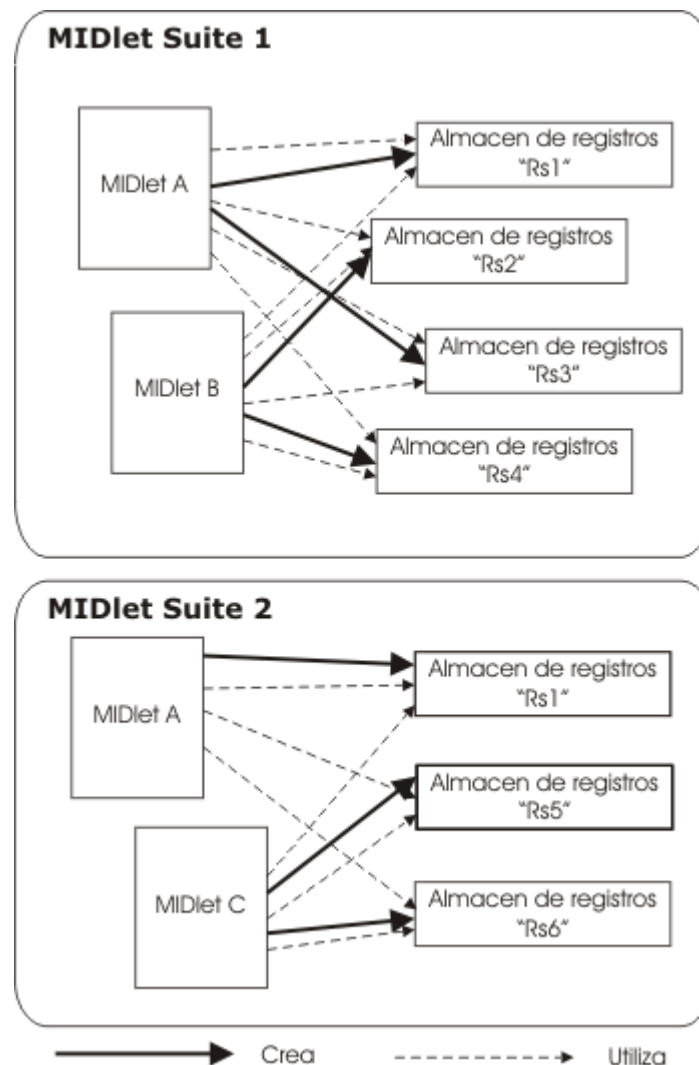


Figura 1. Acceso a los almacenes de registros

Cada *suite* define su propio espacio de nombres. Es decir, los nombres de los almacenes de registros deben ser únicos para cada *suite*, pero pueden estar repetidos en diferentes *suites*. Como hemos dicho antes, nunca podremos acceder a un almacén de registros perteneciente a otra *suite*.

9.1.1. Abrir el almacén de registros

Lo primero que deberemos hacer es abrir o crear el almacén de registros. Para ello utilizaremos el siguiente método:

```
RecordStore rs = RecordStore.open(nombre, true);
```

Con el segundo parámetro a `true` estamos diciendo que si el almacén de registros con nombre `nombre` no existiese en nuestra *suite* lo crearía. Si por el contrario estuviese a `false`, sólo intentaría abrir un almacén de registros existente, y si éste no existe se producirá una excepción `RecordStoreNotFoundException`.

El nombre que especificamos para el almacén de registros deberá ser un nombre de como mucho 32 caracteres codificado en Unicode.

Una vez hayamos terminado de trabajar con el almacén de registros, podremos cerrarlo con:

```
rs.close();
```

9.1.2. Listar los almacenes de registros

Si queremos ver la lista completa de almacenes de registros creados dentro de nuestra suite, podemos utilizar el siguiente método:

```
String [] nombres = RecordStore.listRecordStores();
```

Esto nos devolverá una lista con los nombres de los almacenes de registros que hayan sido creados. Teniendo estos nombres podremos abrirllos como hemos visto anteriormente para consultarlos, o bien eliminarlos.

9.1.3. Eliminar un almacén de registros

Podemos eliminar un almacén de registros existente proporcionando su nombre, con:

```
RecordStore.deleteRecordStore(nombre);
```

9.1.4. Propiedades de los almacenes de registros

Los almacenes de registros tienen una serie de propiedades que podemos obtener con información sobre ellos. Una vez hayamos abierto el almacén de registros para trabajar con él, podremos obtener los valores de las siguientes propiedades:

- **Nombre:** El nombre con el que hemos identificado el almacén de registros.

```
String nombre = rs.getName();
```

- **Estampa de tiempo:** El almacén de registros contiene una estampa de tiempo, que nos indicará el momento de la última modificación que se ha realizado en los datos que almacena. Este instante de tiempo se mide en milisegundos desde el 1 de enero de 1970 a las 0:00, y podemos obtenerlo con:

```
long timestamp = rs.getLastModified();
```

- **Versión:** También tenemos una versión del almacén de registros. La versión será un número que se incrementará cuando se produzca cualquier modificación en el almacén de registros. Esta propiedad, junto a la anterior, nos será útil para tareas de sincronización de datos.

```
int version = rs.getVersion();
```

- **Tamaño:** Nos dice el espacio en *bytes* que ocupa el almacén de registros actualmente.

```
int tam = rs.getSize();
```

- **Tamaño disponible:** Nos dice el espacio máximo que podrá crecer este almacén de registros. El dispositivo limitará el espacio asignado a cada almacén de registros, y con este método podremos saber el espacio restante que nos queda.

```
int libre = rs.getSizeAvailable();
```

9.2. Registros

El almacén de registros contendrá una serie de registros donde podemos almacenar la información. Podemos ver el almacén de registros como una tabla en la que cada fila corresponde a un registro. Los registros tienen un identificador y un *array* de datos.

Identificador	Datos
1	array de datos ...
2	array de datos ...
3	array de datos ...
...	...

Estos datos de cada registro se almacenan como un *array* de *bytes*. Podremos acceder a estos registros mediante su identificador o bien recorriendo todos los registros de la tabla.

Cuando añadamos un nuevo registro al almacén se le asignará un identificador una unidad superior al identificador del último registro que tengamos. Es decir, si añadimos dos registros y al primero se le asigna un identificador n , el segundo tendrá un identificador $n+1$.

Las operaciones para acceder a los datos de los registros son atómicas, por lo que no tendremos problemas cuando se acceda concurrentemente al almacén de registros.

9.2.1. Almacenar información

Tenemos dos formas de almacenar información en el almacén de registros. Lo primero que deberemos hacer en ambos casos es construir un *array* de *bytes* con la información que queramos añadir. Para hacer esto podemos utilizar un flujo `DataOutputStream`, como se muestra en el siguiente ejemplo:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();  
DataOutputStream dos = new DataOutputStream(baos);  
  
dos.writeUTF(nombre);
```



```
dos.writeInt(edad);  
  
byte [] datos = baos.toByteArray();
```

Una vez tenemos el *array* de datos que queremos almacenar, podremos utilizar uno de los siguientes métodos del objeto almacén de datos:

```
int id = rs.addRecord(datos, 0, datos.length);  
rs.setRecord(id, datos, 0, datos.length);
```

En el caso de `addRecord`, lo que se hace es añadir un nuevo registro al almacén con la información que hemos proporcionado, devolviéndonos el identificador `id` asignado al registro que acabamos de añadir.

Con `setRecord` lo que se hace es sobrescribir el registro correspondiente al identificador `id` indicado con los datos proporcionados. En este caso no se añade ningún registro nuevo, sólo se almacenan los datos en un registro ya existente.

9.2.2. Leer información

Si tenemos el identificador del registro que queremos leer, podemos obtener su contenido como *array* de *bytes* directamente utilizando el método:

```
byte [] datos = rs.getRecord(id);
```

Si hemos codificado la información dentro de este registro utilizando un flujo `DataOutputStream`, podemos decodificarlo realizando el proceso inverso con un flujo `DataInputStream`:

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);  
DataInputStream dis = DataInputStream(bais);  
  
String nombre = dis.readUTF();  
String edad = dis.readInt();  
  
dis.close();
```

9.2.3. Borrar registros

Podremos borrar un registro del almacén a partir de su identificador con el siguiente método:

```
rs.deleteRecord(id);
```

9.2.4. Almacenar y recuperar objetos

Si hemos definido una forma de serializar los objetos, podemos aprovechar esta serialización para almacenar los objetos de forma persistente en RMS y posteriormente poder recuperarlos.

Imaginemos que en nuestra clase `MisDatos` hemos definido los siguientes métodos para serializar y deserializar tal como vimos en el apartado de entrada/salida:

```
public void serialize(OutputStream out)
public static MisDatos deserialize(InputStream in)
```

Podemos serializar el objeto en un *array* de *bytes* utilizando estos métodos para almacenarlo en RMS de la siguiente forma:

```
MisDatos md = new MisDatos();
...
ByteArrayOutputStream baos = new ByteArrayOutputStream();
md.serialize(baos);

byte [] datos = baos.toByteArray();
```

Una vez tenemos este *array* de *bytes* podremos almacenarlo en RMS. Cuando queramos recuperar el objeto original, leeremos el *array* de *bytes* de RMS y deserializaremos el objeto de la siguiente forma:

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
MisDatos md = MisDatos.deserialize(bais);
```

9.3. Navegar en el almacén de registros

Si no conocemos el identificador del registro al que queremos acceder, podremos recorrer todos los registros del almacén utilizando un objeto `RecordEnumeration`. Para obtener la enumeración de registros del almacén podemos utilizar el siguiente método:

```
RecordEnumeration re = rs.enumerateRecords(null, null, false);
```

Con los dos primeros parámetros podremos establecer la ordenación y el filtrado de los registros que se enumeren como veremos más adelante. Por ahora vamos a dejarlo a `null` para obtener la enumeración con todos los registros y en un orden arbitrario. Esta es la forma más eficiente de acceder a los registros.

El tercer parámetro nos dice si la enumeración debe mantenerse actualizada con los registros que hay realmente almacenados, o si por el contrario los cambios que se realicen en el almacén después de haber obtenido la enumeración no afectarán a dicha enumeración. Será más eficiente establecer el valor a `false` para evitar que se tenga que mantener actualizado, pero esto tendrá el inconveniente de que puede que alguno de los registros de la enumeración se haya borrado o que se hayan añadido nuevos registros que no constan en la enumeración. En el caso de que especifiquemos `false` para que no actualice automáticamente la enumeración, podremos forzar manualmente a que se actualice invocando el método `rebuild` de la misma, que la reconstruirá utilizando los nuevos datos.

Recorreremos la enumeración de registros de forma similar a como recorremos los objetos `Enumeration`. Tendremos un cursor que en cada momento estará en uno de los elementos de la enumeración. En este caso podremos recorrer la enumeración de forma bidireccional.

Para pasar al siguiente registro de la enumeración y obtener sus datos utilizaremos el método `nextRecord`. Podremos saber si existe un siguiente registro llamando a `hasNextElement`. Nada más crear la enumeración el cursor no se encontrará en ninguno de los registros. Cuando llamemos a `nextRecord` por primera vez se situará en el primer registro y nos devolverá su `array` de datos. De esta forma podremos seguir recorriendo la enumeración mientras haya más registros. Un bucle típico para hacer este recorrido es el siguiente:

```
while(re.hasNextElement()) {  
    byte [] datos = re.nextRecord();  
    // Procesar datos obtenidos  
    ...  
}
```

Hemos dicho que el recorrido puede ser bidireccional. Por lo tanto, tenemos un método `previousRecord` que moverá el cursor al registro anterior devolviéndonos su contenido. De la misma forma, tenemos un método `hasPreviousElement` que nos dirá si existe un registro anterior. Si invocamos `previousRecord` nada más crear la enumeración, cuando el cursor todavía no se ha posicionado en ningún registro, moverá el cursor al último registro de la enumeración devolviéndonos su resultado. Podemos también volver al estado inicial de la enumeración en el que el cursor no apunta a ningún registro llamando a su método `reset`.

En lugar de obtener el contenido de los registros puede que nos interese obtener su identificador, de forma que podamos eliminarlos o hacer otras operaciones con ellos. Para ello tenemos los métodos `nextRecordId` y `previousRecordId`, que tendrán el mismo comportamiento que `nextRecord` y `previousRecord` respectivamente, salvo porque devuelven el identificador de los registros recorridos, y no su contenido.

9.3.1. Ordenación de registros

Puede que nos interese que la enumeración nos ofrezca los registros en un orden determinado. Podemos hacer que se ordenen proporcionando nosotros el criterio de ordenación. Para ello deberemos crear un comparador de registros que nos diga cuando un registro es mayor, menor o igual que otro registro. Para crear este comparador deberemos crear una clase que implemente la interfaz `RecordComparator`:

```
public class MiComparador implements RecordComparator {  
  
    public int compare(byte [] reg1, byte [] reg2) {  
  
        if( /* reg1 es anterior a reg2 */ ) {  
            return RecordComparator.PRECEDES;  
        } else if( /* reg1 es posterior a reg2 */ ) {
```

```
        return RecordComparator.FOLLOWS;
    } else if( /* reg1 es igual a reg2 */ ) {
        return RecordComparator.EQUIVALENT;
    }
}
```

De esta manera, dentro del código de esta clase deberemos decir cuando un registro va antes, después o es equivalente a otro registro, para que el enumerador sepa cómo ordenarlos. Ahora, cuando creemos el enumerador deberemos proporcionarle un objeto de la clase que hemos creado para que realice la ordenación tal como lo hayamos especificado en el método `compare`:

```
RecordEnumeration re =
    rs.enumerateRecords(new MiComparador(), null, false);
```

Una vez hecho esto, podremos recorrer los registros del enumerador como hemos visto anteriormente, con la diferencia de que ahora obtendremos los registros en el orden indicado.

9.3.2. Filtrado de registros

Es posible que no queramos que el enumerador nos devuelva todos los registros, sino sólo los que cumplan unas determinadas características. Es posible realizar un filtrado para que el enumerador sólo nos devuelva los registros que nos interesan. Para que esto sea posible deberemos definir qué características cumplen los registros que nos interesan. Esto lo haremos creando una clase que implemente la interfaz `RecordFilter`:

```
public class MiFiltro implements RecordFilter {

    public boolean matches(byte [] reg) {

        if( /* reg nos interesa */ ) {
            return true;
        } else {
            return false;
        }
    }
}
```

De esta forma dentro del método `matches` diremos si un determinado registro nos interesa, o si por lo contrario debe ser filtrado para que no aparezca en la enumeración. Ahora podremos proporcionar este filtro al crear la enumeración para que filtre los registros según el criterio que hayamos especificado en el método `matches`:

```
RecordEnumeration re =
    rs.enumerateRecords(null, new MiFiltro(), false);
```

Ahora cuando recorramos la enumeración, sólo veremos los registros que cumplan los criterios impuestos en el filtro.

9.4. Notificación de cambios

Es posible que queramos que en cuanto haya un cambio en el almacén de registros se nos notifique. Esto ocurrirá por ejemplo cuando estemos trabajando con la copia de los valores de un conjunto de registros en memoria, y queramos que esta información se mantenga actualizada con los últimos cambios que se hayan producido en el almacén.

Para estar al tanto de estos cambios deberemos utilizar un *listener*, que escuche los cambios en el almacén de registros. Este *listener* lo crearemos implementando la interfaz `RecordListener`, como se muestra a continuación:

```
public class MiListener implements RecordListener {

    public void recordAdded(RecordStore rs, int id) {
        // Se ha añadido un registro con identificador id a rs
    }

    public void recordChanged(RecordStore rs, int id) {
        // Se ha modificado el registro con identificador id en rs
    }

    public void recordDeleted(RecordStore rs, int id) {
        // Se ha eliminado el registro con identificador id de rs
    }
}
```

De esta forma dentro de estos métodos podremos indicar qué hacer cuando se produzca uno de estos cambios en el almacén de registros. Para que cuando se produzca un cambio en el almacén de registros se le notifique a este *listener*, deberemos añadir el *listener* en el correspondiente almacén de registros de la siguiente forma:

```
rs.addRecordListener(new MiListener());
```

De esta forma cada vez que se realice alguna operación en la que se añadan, eliminen o modifiquen registros del almacén se le notificará a nuestro *listener* para que éste pueda realizar la operación que sea necesaria.

Por ejemplo, cuando creamos una enumeración con registros poniendo a `true` el parámetro para que mantenga en todo momento actualizados los datos de la enumeración, lo que hará será utilizar un *listener* para ser notificada de los cambios que se produzcan en el almacén. Cada vez que se produzca un cambio, el *listener* hará que los datos de la enumeración se actualicen.

9.5. Optimización de consultas

Hemos visto que podemos realizar consultas en el almacén utilizando filtrado. Con el objeto `RecordFilter` podemos obtener un conjunto de registros que cumpla ciertas condiciones.

Por ejemplo, imaginemos una aplicación de agenda en la que tengamos almacenadas una serie de citas. Para cada cita tenemos fecha y hora de la cita, asunto, descripción, lugar de la reunión, nombre de la persona de contacto, y la posibilidad de activar una alarma para que el móvil nos avise cuando llegue la hora de la reunión.

```
public class Cita {  
    Date fecha;  
    String asunto;  
    String descripcion;  
    String lugar;  
    String contacto;  
    boolean alarma;  
}
```

En esta aplicación nos interesará obtener aquellas citas que tengan programada una alarma todavía pendiente, es decir, con una fecha posterior a la actual, para que la aplicación pueda activar estas alarmas cuando sea necesario. Esta será una consulta que se hará muy frecuentemente.

Como los datos están almacenados codificados en binario, para buscar aquellos registros que cumplan los criterios de búsqueda deseados tendremos que recorrer todo el conjunto de registros y decodificarlos para comprobar si cumplen estos criterios. Esto nos obligará a leer todos los datos almacenados cada vez que queramos obtener un subconjunto de ellos.

Podemos optimizar esta consulta creando un índice para el conjunto de registros. Para ello crearemos un nuevo almacén de registros donde almacenaremos los índices. De esta forma tendremos un almacén de datos y un almacén de índices. En el almacén de índices tendremos un registro por cada registro existente en el almacén de datos. Cada índice contendrá como datos el identificador del registro al que representa, y además aquellos campos de este registro que utilizamos frecuentemente para realizar las búsquedas.

De esta forma, podremos realizar búsquedas en el almacén de índices, en lugar de hacerlas directamente en el almacén de datos. Buscaremos aquellos índices que cumplan los criterios de búsqueda, y obtendremos los identificadores almacenados en estos índices. Con estos identificadores podremos acceder directamente a los registros buscados en el almacén de datos, habiendo evitado de esta forma tener que recorrer este almacén completo.

Por ejemplo, en el caso de nuestra aplicación de agenda, los índices contendrán para cada registro el fecha de la cita y el *flag* que nos indica si la alarma está activada. De esta forma, las búsquedas que realizamos más frecuentemente se podrán realizar de forma optimizada.

```
public class IndiceCita {  
    int id;  
    Date fecha;  
    boolean alarma;  
}
```

9.6. Patrón de diseño adaptador

Para implementar el acceso a RMS en nuestra aplicación es conveniente utilizar el patrón de diseño adaptador.

Un adaptador es una interfaz adaptada a las necesidades concretas de nuestra aplicación, que encapsula el acceso a una API genérica y nos aísla de ella.

En este caso, el adaptador será una clase que encapsulará todo el acceso a RMS, ofreciéndonos una serie de métodos para acceder a los tipos de datos concretos utilizados en nuestra aplicación. Por ejemplo, mientras en la API de RMS tenemos un método genérico `getRecord` para acceder a un registro, en el adaptador de nuestra aplicación tendremos un método `getCita` que leerá una cita de RMS.

En el caso de la agenda el adaptador podría ofrecernos los siguientes métodos:

```
Cita [] listaCitas();
int addCita(Cita cita);
void updateCita(Cita cita);
void removeCita(int id);
Cita getCita(id);
```

De esta forma aislaremos el resto del código de la forma en la que se encuentran almacenados los datos. Todo el código RMS estará dentro del adaptador. Por ejemplo, podemos tener un adaptador como el siguiente:

```
public class AdaptadorRMS {

    // Nombres de los almacenes
    public final static String RS_INDICE = "rs_indice";
    public final static String RS_DATOS = "rs_datos";

    // Almacenes de registros
    RecordStore rsIndice;
    RecordStore rsDatos;

    public AdaptadorRMS() throws RecordStoreException {
        rsIndice = RecordStore.openRecordStore(RS_INDICE, true);
        rsDatos = RecordStore.openRecordStore(RS_DATOS, true);
    }

    /*
     * Obtiene todas las citas
     */
    public Cita[] listaCitas()
        throws RecordStoreException, IOException {
        RecordEnumeration re = rsDatos.enumerateRecords(null,
                                                         null, false);

        Vector citas = new Vector();

        while (re.hasNextElement()) {
            int id = re.nextRecordId();
```

```
        byte[] datos = rsDatos.getRecord(id);

        ByteArrayInputStream bais = new
            ByteArrayInputStream(datos);
        DataInputStream dis = new DataInputStream(bais);

        Cita cita = Cita.deserialize(dis);
        citas.addElement(cita);
    }

    Cita[] result = new Cita[citas.size()];
    citas.copyInto(result);

    return result;
}

/*
 * Agrega una cita
 */
public int addCita(Cita cita)
    throws IOException, RecordStoreException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);

    cita.serialize(dos);
    byte[] datos = baos.toByteArray();

    int id = rsDatos.addRecord(datos, 0, datos.length);
    return id;
}

/*
 * Elimina una cita
 */
public void removeCita(int id) throws RecordStoreException {
    rsDatos.deleteRecord(id);
}

/*
 * Actualiza una cita
 */
public void updateCita(Cita cita)
    throws IOException, RecordStoreException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    cita.serialize(dos);

    byte[] datos = baos.toByteArray();
    rsDatos.setRecord(cita.getRmsID(), datos, 0,
        datos.length);
}

/*
 * Obtiene una cita
 */
public Cita getCita(int id)
    throws RecordStoreException, IOException {
    byte[] datos = rsDatos.getRecord(id);

    ByteArrayInputStream bais = new
        ByteArrayInputStream(datos);
```



```

        DataInputStream dis = new DataInputStream(bais);

        Cita cita = Cita.deserialize(dis);
        return cita;
    }

    ...

    /*
     * Cierra los almacenes de registros
     */
    public void cerrar() throws RecordStoreException {
        rsIndice.closeRecordStore();
        rsDatos.closeRecordStore();
    }
}

```

Clave primaria

Para poder referenciar un determinado registro necesitaremos que tenga una clave primaria que lo identifique. Como clave primaria podemos utilizar el ID que RMS asigna a cada registro. Para utilizar este valor como clave primaria, deberemos añadirlo como atributo al objeto que encapsule nuestros datos. Por ejemplo, en el caso de las citas añadiremos un nuevo atributo `rmsID` a la clase `Cita` en el que almacenaremos esta clave primaria:

```

public class Cita {
    int rmsID;

    Date fecha;
    String asunto;
    String descripcion;
    String lugar;
    String contacto;
    boolean alarma;
}

```

En el ejemplo del adaptador anterior hemos visto como en cada operación en la que se obtienen citas, se almacena en ellas su identificador, por si posteriormente quisiéramos modificar o eliminar dicha cita.

Gestión de índices

Además podemos añadir al adaptador el código necesario para gestionar los índices. Al igual que en el caso anterior, también deberemos definir una clave primaria para los índices de forma que puedan modificarse o eliminarse posteriormente. En este ejemplo utilizamos como clave primaria el ID asignado al registro por RMS.

```

/*
 * Busca citas con alarma posterior a la fecha indicada
 */
public IndiceCita[] buscaCitas(Date fecha, boolean alarma)
    throws RecordStoreException, IOException {
    RecordEnumeration re = rsIndice.enumerateRecords(

```

```

        new FiltroIndice(fecha, alarma),
                        new OrdenIndice(), false);

    Vector indices = new Vector();

    while (re.hasNextElement()) {
        int id = re.nextRecordId();
        byte[] datos = rsIndice.getRecord(id);

        ByteArrayInputStream bais = new
            ByteArrayInputStream(datos);
        DataInputStream dis = new DataInputStream(bais);

        IndiceCita indice = IndiceCita.deserialize(dis);
        indice.setRmsID(id);
        indices.addElement(indice);
    }

    IndiceCita[] result = new IndiceCita[indices.size()];
    indices.copyInto(result);
    return result;
}

/*
 * Agrega un indice
 */
public int addIndice(IndiceCita indice)
    throws IOException, RecordStoreException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    indice.serialize(dos);
    byte[] datos = baos.toByteArray();
    int id = rsIndice.addRecord(datos, 0, datos.length);
    return id;
}

```

Para realizar la búsqueda de índices de citas posteriores a una determinada fecha con alarma podemos utilizar un filtro de índices como el siguiente:

```

/*
 * Filtra fechas posteriores con alarma
 */
class FiltroIndice implements RecordFilter {
    Date fecha;
    boolean alarma;

    public FiltroIndice(Date fecha, boolean alarma) {
        this.fecha = fecha;
        this.alarma = alarma;
    }

    public boolean matches(byte[] datos) {

        try {
            ByteArrayInputStream bais = new
                ByteArrayInputStream(datos);
            DataInputStream dis = new DataInputStream(bais);
            IndiceCita indice = IndiceCita.deserialize(dis);

            return indice.isAlarma() == this.alarma

```

```
        && indice.getFecha().getTime() >=
            this.fecha.getTime();

    } catch (IOException e) {
        e.printStackTrace();
        return false;
    }
}
```

Este filtro podemos encapsularlo en el mismo adaptador.

10. Red y E/S

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

10.1. Marco de conexiones genéricas

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

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

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

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

```
protocolo:direccion;parámetros
```

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

"http://j2ee.ua.es/pdm"	Abre una conexión HTTP.
"datagram://192.168.0.4:6666"	Abre una conexión por datagramas.
"socket://192.168.0.4:4444"	Abre una conexión por <i>sockets</i> .
"comm:0;baudrate=9600"	Abre una conexión a través de un puerto de comunicaciones.
"file:/fichero.txt"	Abre un fichero.

Cuando especifiquemos uno de estos protocolos, la clase `Connector` buscará en tiempo de ejecución la clase que implemente dicho tipo de conexión, y si la

encuentra nos devolverá un objeto que implemente la interfaz `Connection` que nos permitirá comunicarnos a través de dicha conexión.

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

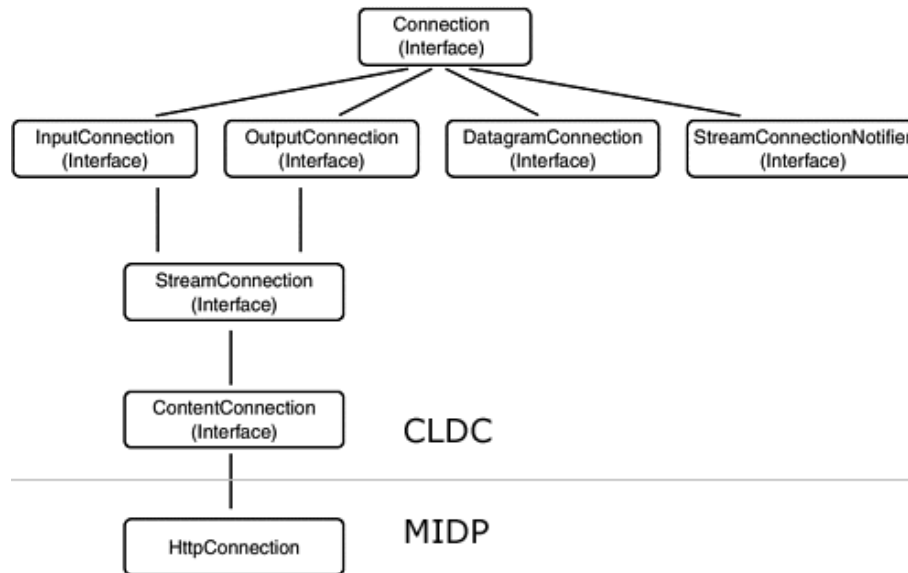


Figura 1. Componentes de GCF

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

10.2. Conexión HTTP

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

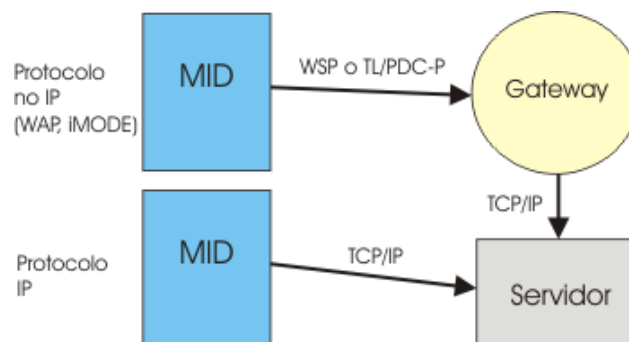


Figura 2. Gateway para protocolos no IP

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

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

```
HttpConnection con =  
(HttpConnection)Connector.open("http://j2ee.ua.es/datos.txt");
```

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

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

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

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

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

10.2.1. Lectura de la respuesta

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

Imaginemos que queremos leer el fichero en la URL `http://j2ee.ua.es/datos.txt`. Como primer paso deberemos crear una

conexión con dicha URL como hemos visto anteriormente. Una vez tengamos este objeto `HttpConnection` abriremos un flujo de entrada para leer su contenido de la siguiente forma:

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

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

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

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

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

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

10.2.2. Mensaje de petición

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

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

`HttpConnection.GET` Los parámetros que se envían al servidor se incluyen en la misma URL. Por ejemplo, podemos mandar un parámetro `login` en la petición de la siguiente forma:
`http://j2ee.ua.es/pdm?login=miguel`

`HttpConnection.POST` Los parámetros que se envían al servidor se incluyen como contenido del mensaje. Tiene la ventaja de que se puede enviar la cantidad de datos que queramos, a diferencia del método `GET` en el que esta cantidad puede estar limitada. Además los datos no serán visibles en la misma URL, ya que se incluyen como contenido del mensaje.

`HttpConnection.HEAD` No se solicita el contenido del recurso al servidor, sólo información sobre éste, es decir, las

cabeceras HTTP.

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

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

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

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

Por ejemplo, podemos mandar las siguiente cabeceras:

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

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

10.2.3. Envío de datos en la petición

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

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

```
HttpConnection con =  
(HttpConnection)Connector.open("http://www.j2ee.ua.es/aplic" +  
    "/registraUsuario?nombre=Pedro&apellidos=Lopez+Garcia&edad=25"  
    );
```

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

En el caso de que queramos simular un formulario con método POST, utilizamos una petición de tipo `HttpConnection.POST` y deberemos incluir los parámetros que enviemos al servidor como contenido del mensaje. Para ello deberemos indicar que el tipo de contenido de la petición es `application/x-`

www-form-urlencoded, y como contenido codificaremos los parámetros de la misma forma que se utiliza para codificarlos en la URL cuando se hace una petición GET:

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

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

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

application/x-www-form-urlencoded

Se envían los datos codificados de la misma forma en la que son codificados por un formulario HTML con método POST.

text/plain

Se envía como contenido texto ASCII.

application/octet-stream

Se envía como contenido datos binarios. Dentro de la secuencia de bytes podremos codificar la información como queramos. Por ejemplo, podemos codificar de forma binaria un objeto serializado, utilizando un `DataOutputStream`.

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

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

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

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

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

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

10.2.4. Tipo y cabeceras de la respuesta

En estado conectado, además del contenido del mensaje de la respuesta, podemos obtener el estado de la respuesta y sus cabeceras. Los estados de respuesta se componen de un código y un mensaje y nos permitirán saber si la petición ha podido atenderse correctamente o si por el contrario ha habido algún tipo de error. Por ejemplo, posibles estados son:

```
HttpConnection.HTTP_OK           200 OK
HttpConnection.HTTP_BAD_REQUEST  400 Bad Request
HttpConnection.HTTP_INTERNAL_ERROR 500 Internal Server Error
```

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

```
HTTP/1.1 200 OK
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.3. Acceso a la red a bajo nivel

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

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

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

10.3.1. Sockets

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

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

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

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

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

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

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

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

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

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

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

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

Debemos tener en cuenta que normalmente los móviles realizan conexiones puntuales cuando necesitan acceder a la red, y cada vez que se conecta se le asigna una nueva IP de forma dinámica. Esto hace difícil que un móvil pueda

comportarse como servidor, ya que no podremos conocer *a priori* la dirección en la que está atendiendo para poder conectarnos a ella desde un cliente.

10.3.2. Datagramas

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

Para trabajar con datagramas utilizaremos una URL como la siguiente:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.4. Envío y recepción de mensajes

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

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

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

10.4.1. Envío de mensajes

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

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

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

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

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

que con una codificación de 8 bits tendremos un juego de caracteres más amplio pero los mensajes estarán limitados a 140 caracteres.

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

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

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

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

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

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

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

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

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

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

10.4.2. Recepción de mensajes

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

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

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

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

Para recibir un mensaje utilizaremos el método:

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

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

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

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

Hemos visto que el método `receive` se queda bloqueado hasta que se reciba un mensaje. No debemos hacer que la aplicación se quede bloqueada esperando un mensaje, ya que éste puede tardar bastante, o incluso no llegar nunca. Podemos solucionar este problema realizando la lectura de los mensajes mediante un hilo en segundo plano. Otra solución es utilizar un *listener*.

10.4.3. Listener de mensajes

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

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

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

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

aplicación. Debemos hacer que la recepción del mensaje la realice un hilo independiente.

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

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

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

10.5. Servicios Web

Los servicios web son una tecnología interesante para las aplicaciones MIDP, ya que nos permiten acceder a información de nuestras aplicaciones corporativas de forma estándar sobre protocolo HTTP.

En una aplicación MIDP no nos servirá de nada obtener un documento HTML, ya que no es adecuado para presentarlo en la interfaz del móvil, y será muy complicado extraer de él información porque normalmente estará escrito en lenguaje natural y con un formato adecuado para su presentación, pero no para ser entendido por una máquina.

Los servicios web nos permiten obtener únicamente la información que necesitamos, pero no la presentación. Esta información vendrá codificada en XML de forma estándar. Podemos ver los servicios web como una web para aplicaciones, frente a los documentos HTML que serían una web para humanos.

El problema que encontramos con los servicios web es que el procesamiento de XML es bastante costoso en términos de procesamiento y memoria, lo cual lo hace poco adecuado para dispositivos muy limitados. Además la información codificada en XML ocupa mucho más espacio que utilizando la codificación binaria, por lo que tendremos que transferir una mayor cantidad de información a través de la red. Cuando la red es lenta y cara, esto es un gran inconveniente, que hace que los servicios web por el momento no se puedan utilizar en la práctica para este tipo de dispositivos.

Sin embargo, ya existen APIs que nos permiten utilizar esta tecnología desde móviles. Web Services API (WSA) nos permite crear clientes de servicios web SOAP desde dispositivos móviles. Podemos acceder a servicios existentes proporcionados por terceros, o crear nuestros propios servicios para acceder a nuestra aplicación.

Si queremos acceder a servicios proporcionados por terceros deberemos tener en cuenta que WSA sólo soporta servicios de tipo `document/literal`. Si el servicio al que queremos acceder no es de este tipo (podemos consultar esta información en su documento WSDL) como solución podremos crearnos un

servicio propio compatible con WSA que encapsule una llamada al servicio proporcionado por terceros.

10.5.1. Creación del servicio

Vamos a ver como crear un servicio compatible con WSA. Para esto deberemos especificar que debe ser de tipo `document/literal`. Consideraremos el caso de la creación del servicio con JWSDP 1.3.

Lo primero que deberemos hacer es implementar el servicio, creando una interfaz remota con los métodos que nos ofrece el servicio y una clase Java que implemente esta interfaz donde se definirá la funcionalidad del servicio.

Por ejemplo, podemos crear un servicio con la siguiente interfaz:

```
package es.ua.j2ee.sw.hola;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HolaMundoIF extends Remote {
    public String saluda(String nombre) throws RemoteException;
}
```

Y con la siguiente implementación de la anterior interfaz:

```
package es.ua.j2ee.sw.hola;

import java.rmi.RemoteException;

public class HolaMundoImpl implements HolaMundoIF {
    public String saluda(String nombre) throws RemoteException {
        return "Hola " + nombre;
    }
}
```

Para que el servicio creado sea del tipo `document/literal` antes de generar el servicio, deberemos generar un modelo donde se especifique el tipo de servicio. Podemos generar el modelo con la herramienta `wscompile`. Para utilizar esta herramienta necesitamos crear un fichero de configuración de nuestro servicio (`config.xml`) como el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service
    name="HolaMundoMovil"
    targetNamespace="http://j2ee.ua.es/sw"
    typeNamespace="http://j2ee.ua.es/sw"
    packageName="es.ua.j2ee.sw.hola">
    <interface name="es.ua.j2ee.sw.hola.HolaMundoIF"/>
  </service>
</configuration>
```

Podemos ejecutar la herramienta `wscompile` desde línea de comando o desde `ant`. Para utilizarla desde `ant` deberemos declarar en el fichero `build.xml` esta tarea y el `classpath` necesario para ejecutarla:

```
<!-- Propiedades -->

<property name="jwsdp.home" value="c:\\jwsdp-1.3"/>

<!-- Classpath -->

<path id="compile.classpath">
  <fileset dir="${jwsdp.home}/jwsdp-shared/lib">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${jwsdp.home}/jaxp/lib">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${jwsdp.home}/jaxp/lib/endorsed">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${jwsdp.home}/jaxrpc/lib">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${jwsdp.home}/saaj/lib">
    <include name="*.jar"/>
  </fileset>
  <fileset dir="${jwsdp.home}/apache-ant/lib">
    <include name="*.jar"/>
  </fileset>
</path>

<!-- Definicion de tareas -->

<taskdef name="wscompile"
  classname="com.sun.xml.rpc.tools.ant.Wscompile">
  <classpath refid="compile.classpath"/>
</taskdef>
<taskdef name="wsdeploy"
  classname="com.sun.xml.rpc.tools.ant.Wsdeploy">
  <classpath refid="compile.classpath"/>
</taskdef>
```

En la llamada a la tarea `wscompile` deberemos especificar como parámetro que el tipo de servicio es `documentliteral` y el fichero donde queremos que se genere el modelo:

```
<target name="generate">
  <wscompile
    keep="true"
    define="true"
    features="documentliteral"
    base="."
    xPrintStackTrace="true"
    verbose="true"
    model="model.gz"
    config="config.xml">
    <classpath>
      <path refid="compile.classpath"/>
    </classpath>
```

```
</wscompile>
</target>
```

Una vez tenemos el modelo generado, podemos crear un fichero descriptor de servicios web (`jaxrpc-ri.xml`) que utilice este modelo:

```
<?xml version="1.0" encoding="UTF-8"?>
<webServices
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
  version="1.0"
  targetNamespaceBase="http://j2ee.ua.es/wsdl"
  typeNamespaceBase="http://j2ee.ua.es/types"
  urlPatternBase="/ws">
  <endpoint
    name="HolaMundo"
    displayName="Servicio HolaMundo"
    description="Servicio Web Hola Mundo"
    interface="es.ua.j2ee.sw.hola.HolaMundoIF"
    model="/WEB-INF/model.gz"
    implementation="es.ua.j2ee.sw.hola.HolaMundoImpl"/>
  <endpointMapping
    endpointName="HolaMundo"
    urlPattern="/hola"/>
</webServices>
```

Una vez tenemos implementado el servicio, creado el modelo y el fichero descriptor de servicios web, organizaremos estos ficheros utilizando la estructura de directorios que deben seguir los servicios web de JWSDP:

```
/WEB-INF/web.xml
/WEB-INF/jaxrpc-ri.xml
/WEB-INF/model.gz
/WEB-INF/classes/es/ua/j2ee/sw/hola/HolaMundoIF.class
/WEB-INF/classes/es/ua/j2ee/sw/hola/HolaMundoImpl.class
```

Empaquetaremos toda esta estructura en un fichero WAR, y entonces podremos utilizar la tarea `wsdeploy` para generar el servicio a partir de dicho fichero, de la misma forma que se hace con cualquier otro servicio web en JWSDP. La única novedad en este caso ha sido que hemos incluido en el servicio un fichero `model.gz` donde se especifica el tipo de servicio que queremos generar.

Si observamos el fichero WSDL generado para el servicio, podremos comprobar que en el apartado `binding` se especifica que es de tipo `document` y `literal`:

```
<binding name="HolaMundoIFBinding" type="tns:HolaMundoIF">
  <operation name="saluda">
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
    <soap:operation soapAction=""/>
  </operation>
```

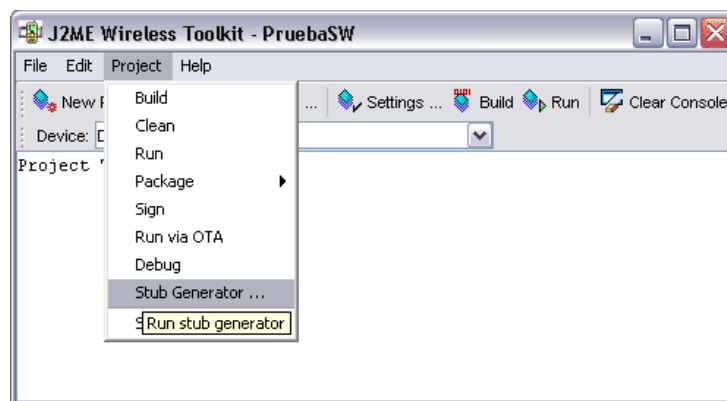
```
<soap:binding
  transport="http://schemas.xmlsoap.org/soap/http"
  style="document"/>
</binding>
```

Cualquier servicio que sea de este tipo se podrá ejecutar desde clientes J2ME. De esta forma, para comprobar si un servicio es compatible con la implementación de J2ME, podremos consultar su fichero WSDL y ver el tipo especificado en el apartado `binding`.

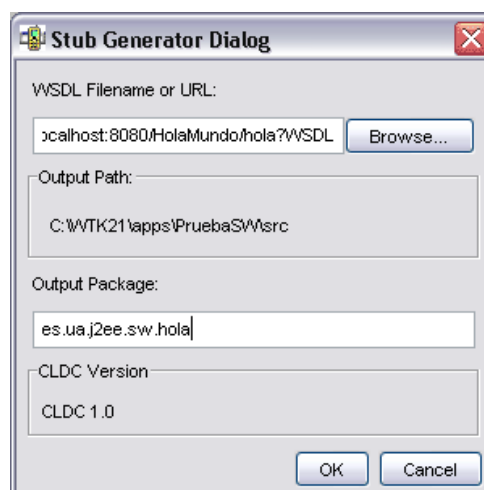
10.5.2. Creación del stub

Para acceder al servicio desde nuestra aplicación deberemos crear una capa *stub*. Esta capa será la encargada de acceder al servicio, de forma que el desarrollador no tenga que preocuparse de implementar este acceso. Simplemente accederemos al stub para invocar los métodos del servicio como si se tratase de un objeto local, sin tenernos que preocupar del mecanismo de invocación subyacente.

WTK 2.1 incluye herramientas para la generación automática de este *stub*. Podremos generar un *stub* en nuestro proyecto utilizando la opción **Project > Stub Generator ...** :



Cuando pulsemos sobre dicha opción, se abrirá una ventana donde deberemos introducir los datos del servicio para el cual queremos generar el *stub*:



En esta ventana indicaremos la dirección donde se encuentra el documento WSDL del servicio al que vamos a acceder, y el paquete donde generaremos las clases del *stub*. Una vez hayamos introducido los datos pulsaremos OK y el *stub* se generará automáticamente.

10.5.3. Invocación del servicio

Una vez tenemos generado el *stub* para acceder al servicio, podemos utilizar este *stub* desde nuestra aplicación MIDP. El *stub* generado será una clase con el nombre de la interfaz de nuestro servicio añadiendo el sufijo `_Stub`.

Esta clase implementará la misma interfaz que nuestro servicio, de forma que podremos instanciarla e invocar los métodos de este objeto para acceder a las operaciones del servicio como si se tratase de un acceso a un objeto local.

Por ejemplo, en el caso de nuestro servicio web *"Hola Mundo"*, podemos acceder a él desde nuestra aplicación cliente de la siguiente forma:

```
HolaMundoIF hola = new HolaMundoIF_Stub();
try {
    String saludo = hola.saluda("Miguel");
} catch (RemoteException re) {
    // Error
}
```

10.6. Conexiones bluetooth

Bluetooth es una tecnología que nos permite conectar dispositivos próximos por radio, sustituyendo de esta forma a las conexiones por cable o infrarrojos. Además *bluetooth* nos ofrece distintos servicios como voz, fax, modem, conexión por puerto serie para transmisión de datos, etc.

No se debe ver bluetooth como un competidor de las redes inalámbricas 802.11b, ya que cada tecnología tiene un fin distinto. Bluetooth se utilizará para conectar pequeños dispositivos en un radio pequeño (unos 10 metros), mientras que las redes 802.11b se utilizarán para conectar dispositivos más potentes como ordenadores de sobremesa y portátiles en un área más grande. Nos podemos referir a las redes 802.11b como redes de área local (LAN), y a las redes bluetooth como redes de área personal (PAN).

Aunque el alcance normal de bluetooth son 10 metros, aumentando la potencia del punto de acceso bluetooth podemos aumentar su radio de alcance hasta 100 metros.

La tecnología bluetooth se puede utilizar para conectar dispositivos, como por ejemplo kits de manos libres, para intercambiar datos con otros dispositivos, para acceder a funcionalidades u obtener información de dispositivos de nuestro entorno, etc. Podemos crear una red de dispositivos que se conecten entre si utilizando esta tecnología.

10.6.1. Topología de las redes bluetooth

Las redes que se crean utilizando bluetooth son redes "*ad hoc*", es decir, se crean dinámicamente. La comunicación entre distintos dispositivos hace que se cree un red de forma espontánea.

Los dispositivos bluetooth tienen la capacidad de "descubrir" otros dispositivos bluetooth de su entorno. De esta forma, los dispositivos pueden localizarse entre ellos y conectarse formando una red de forma dinámica, sin tener que haber creado previamente ninguna infraestructura para dicha red. Cada dispositivo tendrá un identificador bluetooth único con el que se identificará.

Las redes bluetooth se forman en grupos llamados *piconets*. Cada *piconet* es un grupo que puede contener hasta 8 dispositivos como máximo, en el que uno de ellos será el maestro, y los demás serán esclavos que estarán conectados a este maestro.

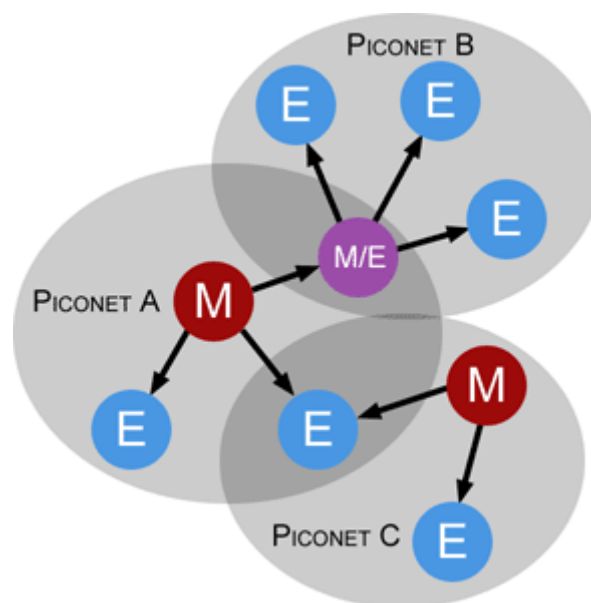


Figura 3. Topología de una red bluetooth

Un mismo dispositivo puede pertenecer a dos *piconets* distintas. Es más, puede que el dispositivo tenga una función distinta en cada *piconet*, por ejemplo puede actuar como maestro en uno de ellos, y como esclavo en el otro.

Cuando un dispositivo pertenece a varias *piconets*, estas piconets estarán conectadas entre ellas. Cuando tenemos varias *piconets* conectadas de esta forma, todas ellas formarán lo que se conoce como una *scatternet*. En el caso de las *scatternet*, dado que existen dispositivos que están conectados al mismo tiempo a dos *piconets*, el ancho de banda de la red será menor. Además, hay muchos dispositivos que no soportan este tipo de redes.

10.6.2. Capas de protocolos

Vamos a ver los protocolos que se utilizan en la comunicación mediante bluetooth.

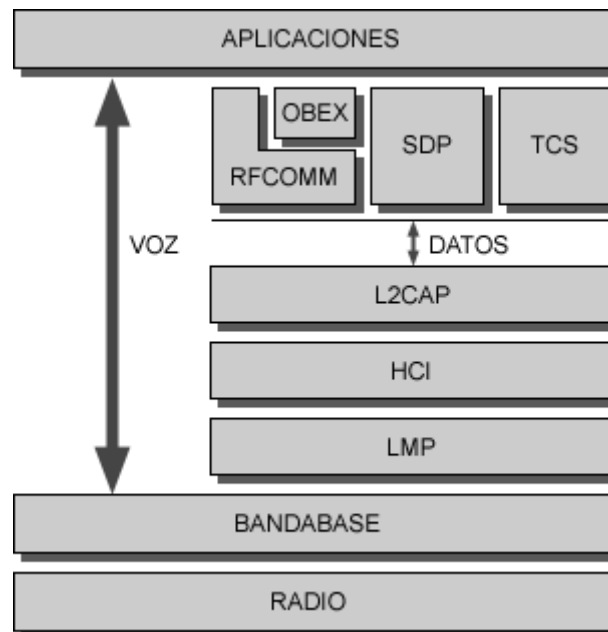


Figura 4. Capas de bluetooth

Encontramos las siguientes capas:

- **Radio:** Nivel físico, en el que se establecen las comunicaciones mediante ondas de radio.
- **Banda base:** Se encarga de enviar y recibir paquetes de datos utilizando la capa física subyacente. Proporciona canales para voz y datos
- **LMP (Link Manager Protocol):** Utiliza los enlaces de la banda base para establecer conexiones y gestionar las piconets.
- **HCI (Host Controller Interface):** Es la línea divisoria entre software y hardware. Las capas por encima de esta están implementadas en software, y las capas por debajo en hardware. Es la interfaz del bus físico que conecta estos componentes.
- **L2CAP (Logical Link Control and Adaptation Protocol):** Esta capa es el protocolo de comunicaciones más bajo nivel que pueden utilizar las aplicaciones para comunicarse mediante bluetooth. Esta capa ya es directamente accesible desde las aplicaciones, que podrán utilizarla para enviar y recibir paquetes de datos.

Para las comunicaciones bluetooth utiliza multiplexado en el tiempo, para poder utilizar una comunicación *full-duplex*. Los datos se envían en ranuras de tiempo de 625ms cada una, el maestro utilizará las ranuras impares y los esclavos utilizarán las pares. Un paquete podrá enviarse en un máximo de 5 ranuras de tiempo (2745 bits de longitud).

Las capas por encima de L2CAP serán los distintos protocolos de comunicaciones que podremos utilizar en las aplicaciones que establezcan conexiones bluetooth. Los protocolos que podremos utilizar para comunicarnos mediante bluetooth son:

- **L2CAP:** Es el protocolo a más bajo nivel sobre el que se construirán el resto de protocolos. Está basado en la transmisión de paquetes, que tendrán un tamaño limitado. Este protocolo no realiza control del flujo, por lo que es posible que se pierdan paquetes en las comunicaciones.
- **RFCOMM:** Emula un puerto serie sobre bluetooth. Es un tipo de conexión basada en flujo de datos, en la que el protocolo utilizado realiza el control necesario para evitar la pérdida de datos en las comunicaciones.
- **SDP (Service Discovery Protocol):** Se utiliza para poder "descubrir" de forma dinámica qué servicios están disponibles en un determinado dispositivo.
- **TCS (Telephony Control Protocol):** Protocolo para aplicaciones de telefonía.
- **OBEX (Object EXchange):** OBEX se podría describir como un protocolo HTTP binario, que se utiliza para transmitir todo tipo de objetos como ficheros, imágenes, etc. Está pensado para redes inalámbricas "ad hoc".

Cuando dos dispositivos se conectan por primera vez, por motivos de seguridad deben establecer un secreto compartido. Esto es lo que se conoce como *pairing*. Es decir, los usuarios de los distintos dispositivos que vayan a conectarse deben ponerse de acuerdo e introducir en ellos el mismo código, para de esta forma evitar que se realicen conexiones no deseadas a nuestro dispositivo. Una vez se ha realizado el *pairing*, este código se guarda en el dispositivo y ya no hará falta introducirlo para las sucesivas conexiones.

Una vez realizado el *pairing* de los dispositivos, estos pueden conectarse para formar una red bluetooth. Esta red se formará de la siguiente forma:

- Los dispositivos esclavos publican sus servicios. Estos servicios se identificarán mediante un UUID, que es una clave única en el espacio y en el tiempo.
- El dispositivo maestro busca los dispositivos bluetooth de su entorno.
- Para cada dispositivo encontrado, descubre los servicios que ofrece.
- Si tuviese el servicio que vamos a utilizar para comunicarnos, nos conectaremos a dicho dispositivo usando este servicio.
- Enviaremos y recibiremos información a través de las conexiones establecidas.

En las redes bluetooth el dispositivo maestro es el que establece los tiempos y el acceso en la piconet. Además hemos visto que tiene la responsabilidad de añadir a los esclavos a la piconet.

Podemos ver el maestro como el servidor de la red, que gestiona las conexiones con varios clientes que en este caso serían los esclavos. En las conexiones cliente/servidor normalmente el servidor permanece a la escucha

esperando peticiones de conexión de los clientes. Sin embargo, en el caso de bluetooth ocurre al contrario, es el servidor (maestro) el que se encarga de solicitar a los clientes (esclavos) que se añadan a la piconet. Para ello estos esclavos deben haber ofrecido (publicado) el servicio necesario, para que el maestro pueda descubrirlo y conectarse a él.

En el caso de conexiones punto-a-punto, es indiferente quien se conecte como esclavo y quien como maestro. Esta decisión tendrá mayor relevancia en el caso de conexiones punto-a-multipunto, en las que todos los esclavos estarán conectados a un mismo maestro.

Vamos a ver ahora cómo establecer estas conexiones bluetooth utilizando las APIs de Java para Bluetooth (JSR-82). Esta API es el primer estándar no propietario que ha aparecido para desarrollar aplicaciones bluetooth utilizando este lenguaje. Podemos distinguir dos APIs independientes:

```
javax.bluetooth  
javax.obex
```

Esto es así porque OBEX puede funcionar sobre distintos tipos de conexiones como cable o infrarrojos. De esta forma la API OBEX no estará ligada a la de bluetooth, sino que será independiente, pudiendo así ser utilizada para trabajar con este protocolo sobre los demás tipos de conexiones.

La API de bluetooth soporta los protocolos L2CAP, SDP y RFCOMM, pero no soporta comunicaciones de voz. Con la API OBEX también podremos utilizar este protocolo.

A partir de WTK 2.2 se incluye soporte para bluetooth en este kit de desarrollo. Para poder probar las aplicaciones bluetooth podemos usar esta versión que, además de incorporar la API JSR-82, incluye emuladores que simulan este tipo de conexiones. Podremos ejecutar varias instancias del emulador y simular conexiones bluetooth entre ellos, sin necesitar disponer de dispositivos bluetooth reales.

10.6.3. Registrar servicios

Lo primero que deberemos hacer para crear una red bluetooth es registrar los servicios de los esclavos, para que el maestro sea capaz de localizarlos y establecer una comunicación con ellos.

Deberemos asignar un UUID al servicio que vayamos a crear. Deberemos asegurarnos de que el UUID que generemos sea un identificador único que identifique el tipo de servicio que estamos implementando.

Un UUID es un número de 128 bits que tiene la siguiente forma (en hexadecimal):

```
UUID = 00000000-0000-1000-8000-0014e3a325f9  
      32      16   16   16   48  
      bits   bits bits bits bits
```

Podemos distinguir varios bloques dentro de este número con distinto número de bits cada uno de ellos.

Para generar este UUID podemos utilizar herramientas como `uuidgen`, que suelen generar este número basándose en el instante de tiempo actual o en generación de números aleatorios.

También podemos generar este número manualmente. Para garantizar que sea único, podemos poner como el último bloque de 48 bits el identificador de nuestro dispositivo bluetooth, y en el resto de bloques podremos poner lo que queramos, siempre que para dos servicios que hagamos no utilicemos el mismo UUID.

Una vez hayamos generado un UUID para nuestro servicio, podemos introducirlo como constante en el código de nuestra aplicación para tener acceso a él cuando sea necesario, tanto desde el cliente como desde el servidor.

```
public final static String UUID =  
"00000000000000010008000123456789ab";
```

Para poder registrar nuestro servicio, lo primero que debemos hacer es establecer nuestro dispositivo como descubrible. Para ello accederemos a nuestro dispositivo local a través de un objeto `LocalDevice` que obtenemos de la siguiente forma:

```
LocalDevice ld = LocalDevice.getLocalDevice();
```

Con este objeto `LocalDevice` podremos obtener datos de nuestro dispositivo local como su dirección bluetooth o su nombre.

```
String bt_addr = ld.getBluetoothAddress();  
String bt_name = ld.getFriendlyName();
```

Una vez tenemos acceso a este objeto, podremos cambiar el modo de ser descubierto del mismo utilizando el método `setDiscoverable`. Hay tres modos de ser descubiertos:

- **DiscoveryAgent.GIAC**: Es el modo general que se utiliza para que los dispositivos bluetooth sean descubiertos (GIAC).
- **DiscoveryAgent.LIAC**: Se trata de un modo limitado para acotar la búsqueda (LIAC, o también conocido como DIAC). Será de utilidad cuando estemos en un entorno con muchos dispositivos bluetooth y estemos buscando uno de ellos en concreto. Haciendo una analogía, sería como cuando estamos buscando a un conocido entre una multitud, y nuestro conocido levanta la mano para que le podamos encontrar rápidamente.
- **DiscoveryAgent.NOT_DISCOVERABLE**: No descubrible. Nuestro dispositivo no podrá ser localizado por ningún otro dispositivo de nuestro entorno.

Si no consiguiésemos hacer nuestro dispositivo localizable, lanzaremos una excepción para interrumpir el flujo del programa, ya que no podremos publicar servicios si nadie va a poder descubrirlos.

```
if (!ld.setDiscoverable(DiscoveryAgent.GIAC)) {  
    // Lanzar excepcion, no se puede descubrir  
}
```

Una vez establecido el dispositivo local como descubrible, pasaremos a crear y registrar el servicio. Debemos definir una URL para nuestro servicio a partir de su UUID. En la URL especificaremos el protocolo de comunicaciones que vamos a utilizar. Para utilizar RFCOMM utilizaremos el protocolo `btsp` (Bluetooth Serial Port Protocol), mientras que para L2CAP utilizaremos `btl2cap`:

```
String url = "btsp://localhost:" + UUID;
```

Utilizando esta URL crearemos una conexión que será la encargada de atender para prestar dicho servicio:

```
StreamConnectionNotifier scn =  
    (StreamConnectionNotifier) Connector.open(url);
```

Con esto tendremos ya nuestro servicio registrado en el dispositivo. Podemos obtener un registro del servicio que será accesible tanto desde el lado del cliente como desde el servidor. En este objeto podremos añadir atributos que el cliente del servicio podrá leer.

```
ServiceRecord sr = ld.getRecord(scn);
```

Ahora que tenemos el servicio creado y registrado, deberemos atender las conexiones que se produzcan a dicho servicio. Con `acceptAndOpen` nos quedaremos bloqueados esperando que se conecte algún cliente a nuestro servicio. Una vez conectado obtendremos una conexión de tipo `StreamConnection`, a partir de la cual podremos abrir flujos de entrada y salida para comunicarnos con el dispositivo remoto según el protocolo que hayamos establecido.

```
while(true) {  
    StreamConnection sc = (StreamConnection) scn.acceptAndOpen();  
  
    // Se ha conectado un maestro  
  
    InputStream is = sc.openInputStream();  
    OutputStream os = sc.openOutputStream();  
  
    // Enviar y recibir datos según el protocolo que  
    establezcamos  
  
    os.flush();  
    is.close();  
    os.close();  
    sc.close();  
}
```

10.6.4. Descubrimiento de dispositivos

Para establecer una conexión con otro dispositivo, lo primero que necesitaremos es localizar (descubrir) dicho dispositivo.

Para localizar los dispositivos necesitaremos utilizar un objeto `DiscoveryAgent` que obtendremos a partir del objeto `LocalDevice`:

```
LocalDevice ld = LocalDevice.getLocalDevice();  
DiscoveryAgent da = ld.getDiscoveryAgent();
```

La búsqueda se podrá hacer de tres formas diferentes:

- Buscar todos los dispositivos del entorno: Con el método `startInquiry` se realiza una búsqueda de todos los dispositivos que haya actualmente en nuestro entorno. Deberemos proporcionar un *listener*, que será llamado cada vez que se encuentre un nuevo dispositivo.
- Obtener dispositivos en caché: Obtiene la lista de dispositivos que se hayan almacenado en caché después de búsquedas anteriores. Obtendremos la lista de dispositivos remotos con:

```
RemoteDevice[] dispositivos =  
    da.retrieveDevices(DiscoveryAgent.CACHED);
```

- Obtener dispositivos preconocidos: Obtiene la lista de dispositivos preconocidos que tengamos configurada en nuestro dispositivo. Podemos añadir dispositivos a esta lista en el centro de control de bluetooth de nuestro móvil. Obtendremos esta lista de dispositivos con:

```
RemoteDevice[] dispositivos =  
    da.retrieveDevices(DiscoveryAgent.PREKNOWN);
```

Vamos a ver cómo descubrir los dispositivos que hay actualmente en nuestro entorno con `startInquiry`. Necesitaremos definir un listener que herede de `DiscoveryListener` e implemente los siguientes métodos:

```
public void deviceDiscovered(RemoteDevice rd, DeviceClass dc);  
public void inquiryCompleted(int tipo);  
public void servicesDiscovered(int transID, ServiceRecord[]  
    servicios);  
public void serviceSearchCompleted(int transID, int estado);
```

Los dos primeros métodos se utilizarán durante el descubrimiento de dispositivos, mientras que los dos últimos se utilizarán para el descubrimiento de servicios de un dispositivo, como veremos más adelante.

El método `deviceDiscovered` se invocará cada vez que un dispositivo remoto sea descubierto. El código que introduciremos en él normalmente será para añadir dicho dispositivo a una lista de dispositivos descubiertos:

```
public void deviceDiscovered(RemoteDevice rd, DeviceClass dc)  
{
```

```
remoteDevices.addElement(rd);  
}
```

A partir del objeto `RemoteDevice` obtenido podremos obtener información sobre el dispositivo remoto, como su nombre o su dirección bluetooth, al igual que con `LocalDevice` obteníamos información sobre el dispositivo local.

Una vez haya terminado la búsqueda de dispositivos se invocará el método `inquiryCompleted`. En este método podremos introducir código para que, en el caso de haberse completado la búsqueda con éxito, nuestra aplicación pase a realizar la siguiente tarea pendiente, que normalmente será la búsqueda de servicios que nos ofrecen los dispositivos descubiertos.

Cuando hayamos creado el listener, podremos comenzar la búsqueda con `startInquiry` especificando el modo de búsqueda (GIAC o LIAC) y el listener al que se notificarán los dispositivos encontrados:

```
da.startInquiry(DiscoveryAgent.GIAC, miListener);
```

Una vez terminada la búsqueda de dispositivos, necesitaremos buscar los servicios que nos ofrecen, para comprobar si está disponible el servicio que buscamos (el que tiene nuestro UUID).

Para cada dispositivo, podremos buscar los servicios identificados mediante una determinada UUID de la siguiente forma:

```
da.searchServices(null, new UUID[]{new UUID(UUID, false)},  
(RemoteDevice) remoteDevices.elementAt(i), miListener);
```

Con esto comenzará la búsqueda de servicios del dispositivo especificado. Cada vez que se encuentre uno o varios servicios nuevos, será invocado el método `servicesDiscovered` con una lista de servicios descubiertos. Para cada servicio remoto encontrado tendremos un objeto `ServiceRecord`. Podemos añadir los servicios encontrados en una lista, igual que en el caso de los dispositivos:

```
public void servicesDiscovered(int transID, ServiceRecord[]  
servicios) {  
    for(int i=0; i<servicios.length; i++) {  
        remoteServices.addElement(servicios[i]);  
    }  
}
```

Cuando haya finalizado la búsqueda de servicios, se invocará el método `serviceSearchComplete`.

10.6.5. Establecer conexiones

Como último paso, tendremos que establecer una conexión con alguno de los servicios que hayamos localizado. Para ello utilizaremos el objeto `ServiceRecord` obtenido correspondiente a dicho servicio.

```
ServiceRecord rs =  
(ServiceRecord) remoteServices.elementAt(0);
```

A partir de este objeto podremos obtener la URL necesaria para conectarnos al servicio:

```
String url = rs.getConnectionURL(  
    ServiceRecord.NOAUTHENTICATE_NOENCRYPT, true);
```

Con esta URL podremos crear una conexión de tipo flujo y abrir los correspondiente flujos de entrada salida para intercambiar datos con el servidor:

```
StreamConnection sc = (StreamConnection) Connector.open(url);  
  
InputStream is = sc.openInputStream();  
OutputStream os = sc.openOutputStream();  
  
// Enviar y recibir datos según el protocolo que establezcamos  
  
os.flush();  
os.close();  
is.close();  
sc.close();
```

10.6.6. Protocolo L2CAP

En el caso de utilizar directamente el protocolo de bajo nivel L2CAP, utilizaremos una URL como la siguiente para crear la conexión:

```
String url = "bt12cap://localhost:" + UUID;
```

Cuando abramos la conexión con `Connector.open` obtendremos un objeto de tipo `L2CAPConnectionNotifier`, y mediante el método `acceptAndOpen` de este objeto podremos aceptar conexiones L2CAP de clientes. Cuando aceptemos una conexión de un cliente obtendremos un objeto `L2CAPConnection` con el que podremos realizar la comunicación.

Este objeto `L2CAPConnection` tiene dos métodos `send` y `receive` con los que podremos enviar y recibir respectivamente paquetes de datos L2CAP.

Como este protocolo no proporciona control de flujo, este control lo deberemos hacer nosotros si queremos asegurarnos de que los paquetes enviados no se han perdido.

Utilizaremos este protocolo cuando necesitemos una comunicación rápida o cuando la pérdida de paquetes no sea crítica para nuestra aplicación.

11. Registro push

Hasta ahora hemos visto que cuando queremos obtener información primero debemos abrir la aplicación cliente que nos da acceso a esa información. Para recibir algo antes tenemos que solicitarlo desde nuestra aplicación, esto es lo que se conoce como una conexión *pull*, en la que el cliente debe "tirar" de los datos para obtenerlos.

Para determinadas aplicaciones puede ser interesante poder recibir datos sin tener que solicitarlos. Por ejemplo pensemos en una aplicación de foro, en el que varios usuarios pueden publicar mensajes. Nosotros queremos visualizar en nuestro cliente la lista de mensajes publicados en el servidor, pero no sabemos cuando llega un mensaje al servidor, ya que puede haberlo enviado cualquier otro cliente, sólo el servidor sabe cuando llegan nuevos mensajes.

Cuando el usuario abra la aplicación del foro se descargarán todos los mensajes publicados y se mostrarán. Esto sigue un modelo *pull*, en el que el usuario tiene que abrir la aplicación para obtener los datos deseados, debe "tirar" de los datos.

Pero imaginemos que queremos implementar una función de avisos en el foro, con la cual un usuario puede programar un aviso, de forma que cuando alguien publique una contestación a un tema que sea de su interés, el usuario reciba una notificación. De esta forma se evita tener que estar entrando en la aplicación periódicamente para ver si alguien ha contestado. Esto es un modelo *push*, en el que es la aplicación la que "empuja" los datos hacia nosotros en el momento en el que llegan.

Para implementar este comportamiento podríamos utilizar una técnica conocida como *polling*, que consiste en interrogar al servidor cada cierto periodo de tiempo para comprobar si han llegado mensajes nuevos, y en tal caso recibirlos y mostrarlos al usuario. Esto nos obligará a estar continuamente realizando peticiones al servidor, aunque no se haya recibido ningún mensaje, con lo que se estará produciendo un tráfico innecesario en la red. Además deberemos tener la aplicación continuamente en funcionamiento, aunque sea en segundo plano, para que haga las comprobaciones.

Con un modelo de tipo *push* podremos solucionar este problema, ya que en este caso el servidor podrá enviarnos información sin tener que pedirla nosotros previamente. De esta forma cuando el servidor haya recibido nuevos mensajes nos los enviará mediante *push*, sin tener que estar interrogándolo nosotros continuamente.

Es decir, *push* es un mecanismo que nos va a permitir recibir información de forma asíncrona, sin tenerla que solicitar nosotros previamente, y evitando el elevado consumo de recursos que producen las técnicas de *polling*.

En MIDP 2.0 aparece el registro *push*, que nos permitirá utilizar este mecanismo de recepción de información de forma asíncrona en nuestro dispositivo.

11.1. Aplicaciones activadas por push

El registro *push* nos permitirá que nuestras aplicaciones sean activadas automáticamente mediante *push*. Es decir, no hará falta que nosotros abramos la aplicación manualmente para que esta pueda realizar alguna función, sino que se podrá activar automáticamente cuando suceda algún evento externo.

Por ejemplo, una utilidad bastante clara de este mecanismo es la programación de alarmas. Imaginemos una agenda que deba hacer sonar una alarma cuando llegue la hora de una reunión. En MIDP 1.0 una aplicación sólo podrá programar una alarma si abrimos manualmente esa aplicación, y deberemos mantener esta aplicación abierta en segundo plano permanentemente, ya que si la cerrásemos también se cerraría la alarma programada. Esto hace que en MIDP 1.0 este tipo de aplicaciones sea poco útiles, ya que si se nos olvida abrir la aplicación cuando encendamos el móvil no sonará ninguna de las alarmas programadas.

Sin embargo, en MIDP 2.0 con el registro *push* podremos hacer que la aplicación se abra automáticamente cuando llegue la hora de la alarma, es decir, la aplicación se estará activando automáticamente mediante *push*. En este caso el evento externo que activa la aplicación será un temporizador.

11.1.1. Métodos de activación

La activación mediante push puede suceder por dos tipos de eventos:

- **Temporizador:** La aplicación se abrirá a una determinada hora que debemos programar.
- **Conexión de red entrante:** La aplicación se abrirá cuando se reciba una determinada conexión de red entrante. Esta conexión puede ser una conexión mediante sockets, datagramas o bien la recepción de un mensaje de texto por ejemplo.

11.1.2. Responsabilidad compartida

Deberemos tener en cuenta que el registro *push* sólo estará pendiente de estos eventos externos mientras nuestra aplicación esté cerrada. Cuando ejecutemos la aplicación, será responsabilidad suya escuchar las conexiones entrantes y registrar los temporizadores adecuados para que se disparen las alarmas.

Es decir, que la responsabilidad de la escucha y la gestión de los eventos *push* será compartida entre el MIDlet y el AMS:

- Mientras el MIDlet se esté ejecutando será responsabilidad suya atender a estos eventos.

- Mientras el MIDlet no se esté ejecutando será el AMS el que se encargará de escuchar estos eventos, y cuando uno de ellos se produzca ejecutará automáticamente el MIDlet para que éste realice la función oportuna.

La facilidad que nos proporciona el registro *push* es esta capacidad de ejecutar la aplicación automáticamente cuando se produzca un determinado evento externo, pero mientras nuestra aplicación se esté ejecutando este registro no realizará ninguna función.

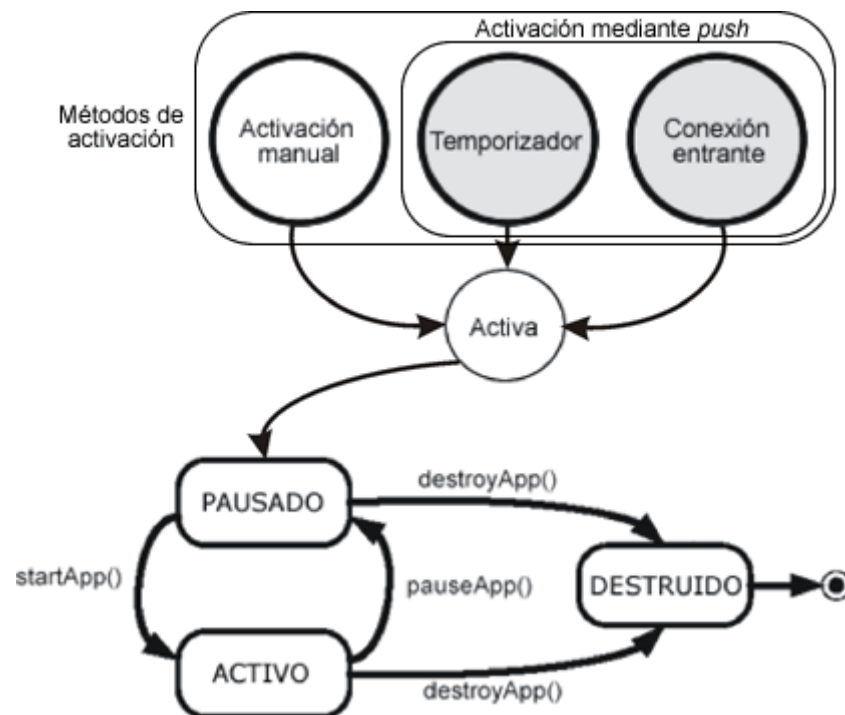


Figura 5. Métodos de activación de MIDlets

11.1.3. Emulación en WTK

A partir de WTK 2.0 se puede utilizar la activación *push* en los emuladores. Pero para disponer de esta funcionalidad deberemos cargar la aplicación utilizando provisionamiento OTA, instalando de esta forma la aplicación utilizando el AMS del emulador.

11.2. Temporizadores

Vamos a ver cómo podemos registrar un temporizador *push*, para que la aplicación se ejecute automáticamente a una determinada hora. Para hacer esto deberemos utilizar la API de `PushRegistry`, que nos permitirá registrar este tipo de eventos en el registro *push*.

Podremos registrar una alarma *push* con el siguiente método:

```
long t = PushRegistry.registerAlarm(String midletClassName,  
long hora);
```

Deberemos proporcionar como parámetros el nombre de la clase del MIDlet que queremos que se ejecute, y el tiempo del sistema en milisegundos del instante en el que queremos que se produzca la alarma.

Hemos de tener en cuenta que sólo podemos registrar una alarma por MIDlet, de forma que si hubiésemos registrado otra alarma previamente, al invocar este método de sobrescribirá.

Si hubiese una alarma registrada previamente, la llamada a esta función nos devolverá como valor de tiempo `t` el instante en el que estaba programada la alarma previa que ha sido sobrescrita. En caso contrario, nos devolverá `0`.

Si nuestra aplicación tuviese que programar varias alarmas, lo que deberíamos hacer es registrar mediante push la alarma que tenga la fecha más temprana, y una vez se haya producido esta alarma, se registrará la siguiente, y así consecutivamente.

11.2.1. Programar una alarma para una fecha absoluta

Si queremos programar una alarma para que se produzca en una determinada fecha, independientemente de la fecha actual, simplemente deberemos proporcionar al método `registerAlarm` el tiempo del sistema en milisegundos de la hora en la que queremos que se produzca.

Para hacer esto simplemente deberemos obtener un objeto `Date` que represente dicha hora. Podemos o bien solicitar que el usuario introduzca la fecha manualmente en un campo de fecha, y leer de ese campo el objeto `Date` con la fecha introducida, o bien generar desde nuestra aplicaciones una determinada fecha utilizando un objeto `Calendar`.

Una vez tengamos el objeto `Date` correspondiente a esta fecha, obtendremos de él el tiempo en milisegundos de dicha fecha para proporcionárselo a la función `registerAlarm`:

```
Date fecha = obtenerFecha();  
long t =  
PushRegistry.registerAlarm(midlet.getClass().getName(),  
fecha.getTime());
```

11.2.2. Programar una alarma para un intervalo de tiempo

Es posible que no queramos dar una fecha absoluta a la alarma, sino que queramos establecer una alarma que suene pasado cierto tiempo desde el instante actual. Por ejemplo, podemos querer poner una alarma para que suene dentro de 5 minutos.

Para hacer esto deberemos establecer la fecha de forma relativa a la fecha actual. Primero obtendremos la fecha actual mediante un objeto `Date`, y a esta

fecha podremos sumarle el número de milisegundos del intervalo de tiempo que queremos que tarde en sonar la alarma:

```
Date ahora = new Date(); // Obtiene la fecha actual
long t =
PushRegistry.registerAlarm(midlet.getClass().getName(),
                           ahora.getTime() +
                           intervalo);
```

11.2.3. Responsabilidad del MIDlet

Cuando el MIDlet esté en ejecución, los temporizadores registrados mediante *push* no tendrán efecto, estos temporizadores sólo servirán para activar la aplicación cuando esté cerrada.

Por lo tanto, normalmente durante la ejecución de la aplicación crearemos temporizadores utilizando la clase `Timer` de Java. Como estos temporizadores serán hilos de la aplicación, cuando ésta se cierre los temporizadores también se anularán. Entonces será este el momento en el que deberemos registrar el temporizador *push*.

El lugar adecuado para registrar los temporizadores *push* será el método `destroyApp` del MIDlet. De esta forma, si al destruirse la aplicación todavía tuviésemos algún temporizador pendiente, los registraremos mediante *push* para que siga siendo efectivo.

11.2.4. Activación del MIDlet

Cuando el MIDlet se active vía *push*, debido a un temporizador, simplemente se pondrá en marcha la aplicación como si la hubiésemos abierto manualmente, y no tendrá constancia en ningún momento de que se ha abierto debido a un temporizador.

Si queremos que cuando se abra automáticamente se ejecute la alarma, deberemos implementar este comportamiento manualmente. Para hacer esto deberemos registrar las alarmas pendientes utilizando RMS.

Cuando queramos programar una alarma, los datos de esta alarma se registrarán de forma persistente utilizando RMS, y se programará un `Timer` para ella.

Si salimos de la aplicación, en el método `destroyApp` se buscará la siguiente alarma pendiente, si hay alguna, y la registrará mediante *push*.

Cuando la aplicación se active la próxima vez, podrá leer las alarmas pendientes en RMS y programar un `Timer` para la siguiente. De esta forma, si la aplicación se hubiese abierto de forma manual y no correspondiese activar ninguna alarma, como en RMS no hay ninguna alarma programada para este momento no sucederá nada. Si por el contrario, se hubiese abierto debido al temporizador *push*, como tendremos en RMS una alarma programada justo para el momento actual, se activará la alarma. Así, aunque la aplicación no

sepa cual ha sido la forma de activarse, mediante la información almacenada en RMS podremos saber cuándo debemos disparar las alarmas y cuando no.

11.3. Conexiones *push*

Podemos hacer que las aplicaciones se activen mediante conexiones de red entrantes. Utilizando protocolo HTTP no tenemos este tipo de conexiones, ya que éste es un protocolo síncrono en el que debemos realizar una petición para obtener una respuesta.

Sin embargo, en MIDP 2.0 se definen otros tipos de conexiones a bajo nivel que si que soportan conexiones entrantes. Estas son las conexiones de sockets y datagramas. Podremos hacer que el dispositivo escuche en un determinado puerto conexiones entrantes mediante sockets o datagramas.

El problema de estos tipos de conexiones es que en la mayoría de los casos los operadores de telefonía móvil actuales utilizan IPs dinámicas, por lo que cada vez que se establezca una nueva conexión (por ejemplo mediante GPRS) se obtendrá una IP distinta. Esto complica la tarea de registrar nuestro móvil para recibir avisos, ya que si la IP cambia frecuentemente no se sabrá a qué dirección se debe enviar la información.

Otro tipo de conexión que tenemos disponible en un gran número de dispositivos y que nos resultará más útil es la conexión de mensajes que aporta la API WMA. Los mensajes de texto se envían a un número de teléfono que sabemos que no va a cambiar. De esta manera podremos tener identificado claramente el dispositivo en el cual queremos recibir notificaciones.

Tenemos dos formas de registrar conexiones *push* entrantes:

- **Registro estático:** Se registran en el momento en que se instala la suite. Para hacer esto podemos incluir la información sobre las conexiones entrantes en el fichero JAD.
- **Registro dinámico:** Utilizamos la API de `PushRegistry` para registrar las conexiones en tiempo de ejecución, de la misma forma en la que se registran los temporizadores.

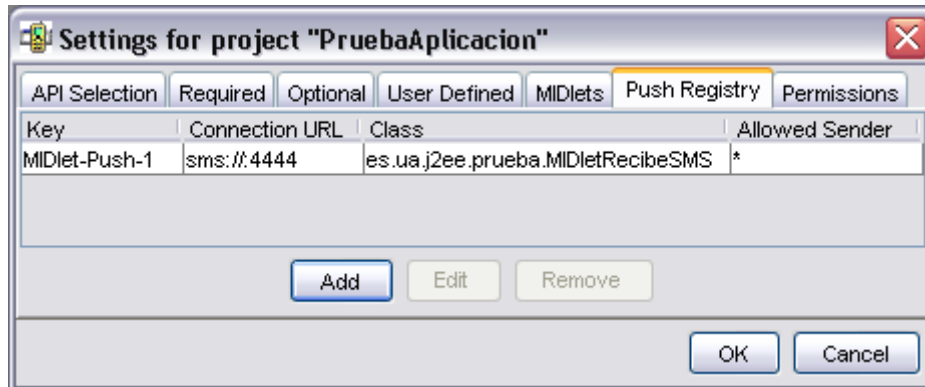
11.3.1. Registro estático

Podremos utilizar registro estático cuando las direcciones de nuestras conexiones entrantes sean estáticas. Este será el caso de las conexiones de mensajes (SMS) en cualquier dispositivo, o de las conexiones de sockets y datagramas en aquellos dispositivos que tengan asignada una IP estática y su puerto se configure también de forma estática.

Para registrar una conexión *push* entrante añadiremos un atributo como el siguiente en el fichero JAD:

```
MIDlet-Push-<n>: <URL>, <NombreClaseMIDlet>,  
<RemitentesPermitidos>
```

Debemos especificar para cada conexión *push* que queramos permitir la URL de la conexión entrante y el MIDlet que se ejecutará cuando recibamos datos. Además como tercer elemento podemos indicar los remitentes a los que les permitimos enviarnos datos. Con * indicamos que aceptamos datos de cualquier remitente.



Estas conexiones se registrarán en el momento en que se instale la aplicación, y se eliminarán cuando se desinstale.

Este tipo de registro no es posible cuando tengamos asignación dinámica de IPs, ya que en tiempo de despliegue no podremos conocer cual será la IP que tenga el dispositivo cuando se utilice la aplicación. En este caso deberemos utilizar registro dinámico, como veremos más adelante.

Para configurar conexiones entrantes de sockets, datagramas y mensajes podremos utilizar URLs de los siguientes tipos:

```
socket://:<puerto>
datagram://:<puerto>
sms://:<puerto>
```

Al utilizar registro estático siempre deberemos indicar explícitamente el puerto, ya que si dejamos que el puerto se asigne dinámicamente no sabremos a qué puerto del móvil hay que conectarse para activar la aplicación.

Por ejemplo, podemos registrar una conexión de mensajes entrantes de la siguiente forma:

```
MIDlet-Push-1: sms://:4444, es.ua.j2ee.sms.MIDletRecibirSMS, *
```

De esta forma se escuchará la llegada de SMSs en el puerto 4444, y en el momento que llegue uno se ejecutará el MIDlet `MIDletRecibirSMS`, permitiendo que lleguen desde cualquier remitente.

11.3.2. Registro dinámico

Cuando utilicemos registro dinámico podremos configurar, además de las conexiones anteriores, las conexiones de sockets y datagramas para las que se asigne el puerto de forma dinámica:

```
socket://  
datagram://
```

Utilizaremos este tipo de registro cuando estemos utilizando direcciones dinámicas, o bien cuando utilizando direcciones estáticas queramos registrar la conexión sólo si se cumplen ciertas condiciones.

En este caso el registro lo realizaremos en tiempo de ejecución, utilizando la API de `PushRegistry`. Podemos registrar una conexión entrante dinámicamente de la siguiente forma:

```
PushRegistry.registerConnection(url,  
    nombreClaseMIDlet, remitentesPermitidos);
```

Si estamos utilizando una conexión entrante asignada dinámicamente, encontramos el problema de que el sistema externo no sabrá a qué dirección debe enviar la información para contactar con nuestra aplicación. Para solucionar este problema deberemos comunicar a este sistema externo la dirección en la que estamos escuchando. Esto podemos hacerlo de la siguiente forma:

```
// Creamos socket servidor asignando el puerto dinámicamente  
ServerSocketConnection ssc =  
    (ServerSocketConnection)Connector.open("socket://");  
  
// Obtenemos el puerto que ha asignado el sistema  
String url = "socket://:" + ssc.getLocalPort();  
  
// Registramos en push una conexión entrante con este mismo  
puerto  
PushRegistry.registerConnection(url, midletClassName, filter);  
  
// Obtenemos la URL completa de nuestra aplicación  
url = "socket://" + ssc.getLocalAddress() + ":" +  
    ssc.getLocalPort();  
  
// Publicamos la URL en el sistema externo  
publicarURL(url);
```

De esta forma cada vez que el sistema asigne un nuevo puerto a nuestra aplicación deberemos publicar la nueva URL en el sistema externo.

Hemos visto que publicando la dirección en el sistema externo podemos utilizar direcciones en las que el puerto se asigna de forma dinámica por el sistema. Sin embargo, el caso en el que nuestro operador de telefonía asigne al dispositivo la IP de forma dinámica será más complicado, ya que cualquier pérdida de la conexión hará que la IP cambie y el sistema externo será incapaz de comunicarse con nuestra aplicación.

En este caso los cambios de dirección pueden ser muy frecuentes, y si la aplicación Java está cerrada no tendremos constancia del momento en el que esto ocurra, por lo que estos tipos de conexión push no serán de utilidad. En ese caso la solución será utilizar conexiones de mensajes, para las que si que tenemos una dirección (número de teléfono) asignada de forma estática.

11.3.3. Eliminar conexión entrante

Las conexiones entrantes *push* que registremos de forma dinámica podrán ser eliminadas. Para eliminar una conexión utilizaremos el siguiente método:

```
try {
    boolean estado = PushRegistry.unregisterConnection(url);
} catch (SecurityException e) {
    // Error de seguridad
}
```

El método `unregisterConnection` nos devolverá `true` si ha podido eliminar la conexión *push* correctamente, y `false` en caso contrario.

11.3.4. Activación de la aplicación

Cuando la aplicación haya sido activada mediante una conexión entrante *push*, a diferencia de la activación mediante temporizador, si que tendremos constancia de la forma en la que se ha activado.

Para esto utilizaremos el método `listConnections`, que nos devuelve la lista de conexiones *push* registradas para el MIDlet. Si a este método le pasamos `true` como parámetro, nos devolverá sólo aquellas conexiones en las que tengamos datos disponibles para ser leídos.

De esta forma, si al ejecutarse la aplicación existe alguna conexión *push* con datos disponibles sabremos que la aplicación se ha activado mediante *push* debido a la recepción de datos en dichas conexiones.

```
public boolean isPushActivated() {

    String [] conexiones = PushRegistry.listConnections(true);

    if (conexiones != null && conexiones.length > 0) {
        for (int i=0; i < conexiones.length; i++) {
            leerDatos(conexiones[i]);
        }
        return true ;
    } else {
        return false;
    }
}
```

Cuando hayamos recibido datos en una de estas conexiones será responsabilidad de nuestro MIDlet abrir esta conexión y leer los datos.

Hemos de tener en cuenta que en conexiones como los sockets el AMS no creará ningún buffer con los datos, sino que será el MIDlet activado el que los lea directamente de la red. Por lo tanto, deberemos leerlos lo más rápidamente posible, ya que si demorásemos la lectura de datos se podría producir un *timeout*.

12. Seguridad

Vamos a estudiar la seguridad de las aplicaciones MIDP. Nos referiremos a seguridad en cuanto a que las aplicaciones que se instale el usuario en el móvil no puedan realizar actividades dañinas para él. Otros tipos de seguridad dependientes de la aplicación, como son la autenticación y la confidencialidad de la información transmitida por la red los estudiaremos en el tema de aplicaciones corporativas.

El usuario navegando por la red puede encontrar aplicaciones MIDP que para utilizarlas deberán ser instaladas de forma local en su móvil. Si permitiésemos que estas aplicaciones, una vez instaladas, pudiesen realizar cualquier acción, sería bastante peligroso utilizar este tipo de aplicaciones. El usuario debería estar muy seguro que la aplicación que se instala es de su confianza, porque de no serlo la aplicación podría realizar tareas como por ejemplo:

- Eliminar datos de otras aplicaciones del móvil.
- Leer datos confidenciales del usuario almacenados en el móvil y enviarlos a través de la red.
- Establecer conexiones de red que le costarán dinero al usuario sin que éste lo sepa.

Por lo tanto, es importante garantizar que las aplicaciones MIDP son seguras y no pueden realizar ninguna acción dañina, para que de esta forma los usuarios puedan confiar en ellas y descargárselas sin ningún temor. A continuación veremos cómo se garantiza esta seguridad en las aplicaciones MIDP.

12.1. Sandbox

La seguridad de las aplicaciones MIDP se debe a que éstas aplicaciones se debe a que éstas se ejecutan en un entorno restringido y controlado. Al ejecutarse sobre una máquina virtual, y no directamente sobre el dispositivo, se puede limitar el número de acciones que estas aplicaciones pueden realizar, evitando de esta forma que se realicen acciones que puedan resultar dañinas.

Las aplicaciones se ejecutan dentro de lo que se conoce como un cajón de arena (*sandbox*), como los que existen en los parques para que los niños jueguen de forma segura. Este cajón de arena es un entorno limitado y cerrado en el que podrá trabajar la aplicación, y en el que no se tendrá acceso a nada que pudiera resultar dañino.

En el caso de las aplicaciones MIDP, este *sandbox* será la *suite* de MIDlets. Es decir, ninguna aplicación podrá acceder a nada externo a su *suite*:

- Sólo podremos instanciar clases contenidas en nuestra *suite*.
- Sólo podremos leer recursos incluidos en nuestra *suite*.
- Sólo podremos acceder a almacenes de registros RMS creados por MIDlets de nuestra *suite*.

Como las aplicaciones MIDP no permiten acceder al sistema de ficheros del dispositivo, ni tampoco permiten utilizar su API nativa, no tendremos problemas de seguridad en este aspecto.

La única funcionalidad que podría resultar peligrosa es la capacidad que tienen estas aplicaciones de establecer conexiones de red. Una aplicación podría estar intercambiando información por la red, lo cual le costará dinero al usuario, sin que éste se diese cuenta. Sin embargo, esta es una funcionalidad imprescindible de las aplicaciones MIDP, por lo que no podemos privar a estas aplicaciones de su API de red. Por esta razón, a partir de MIDP 2.0 surge un modelo de seguridad que limitará la utilización de estas funciones a las aplicaciones que obtengan permiso para hacerlo.

12.2. Solicitud de permisos

En la API de MIDP 2.0 existen diferentes permisos para cada tipo de conexión que pueda realizar el dispositivo. Estos permisos son los siguientes:

```
javax.microedition.io.Connector.http  
javax.microedition.io.Connector.socket  
javax.microedition.io.Connector.https  
javax.microedition.io.Connector.ssl  
javax.microedition.io.Connector.datagram  
javax.microedition.io.Connector.serversocket  
javax.microedition.io.Connector.datagramreceiver  
javax.microedition.io.Connector.comm  
javax.microedition.io.PushRegistry
```

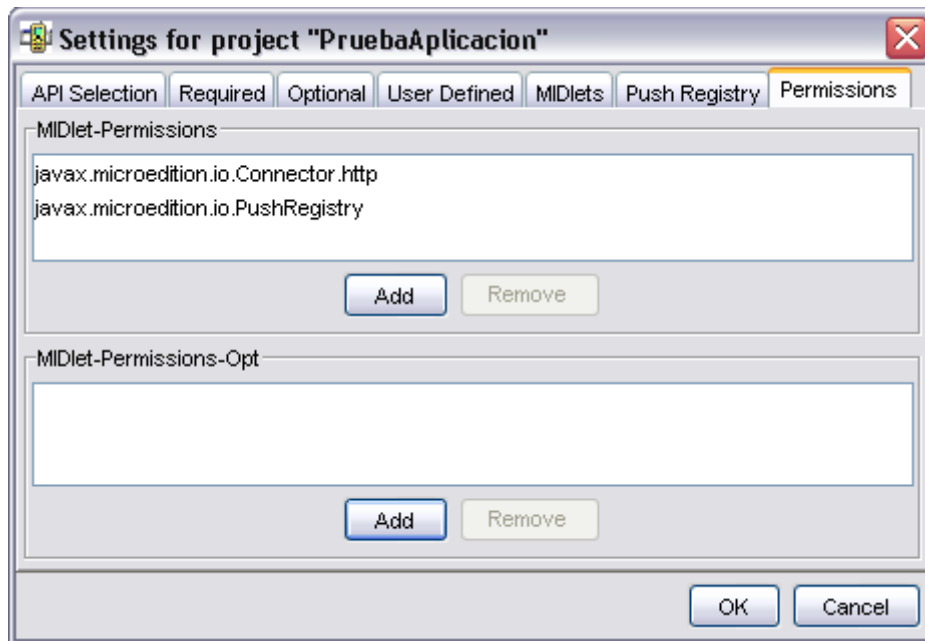
Cuando en nuestra aplicación necesitemos utilizar alguna conexión de cualquiera de estos tipos, deberemos solicitar el permiso para poder hacerlo. Solicitaremos los permisos en el fichero JAD, mediante las propiedades `MIDlet-Permissions` y `MIDlet-Permissions-Opt`, en las que especificaremos todos los permisos solicitados separados por comas:

```
MIDlet-Permissions:  
javax.microedition.io.Connector.http,javax.microedition.io.PushRegistry  
MIDlet-Permissions-Opt: javax.microedition.io.Connector.https
```

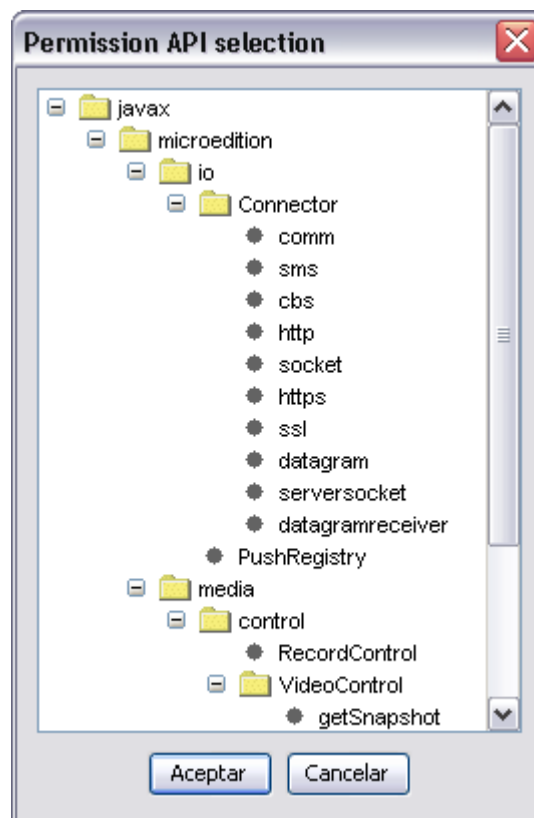
El atributo `MIDlet-Permissions` indicará aquellos permisos que son esenciales para que nuestra aplicación pueda funcionar. Si el dispositivo en el que se va a instalar la aplicación no pudiese conceder estos permisos a nuestra aplicación, debido a que no confía en ella, se producirá un error en la instalación ya que la aplicación no funcionará sin estos permisos.

En el caso de `MIDlet-Permissions-Opt`, especificamos permisos que solicitamos, pero que son opcionales. Si no se pudiesen obtener estos permisos la aplicación podría funcionar.

En WTK podremos introducir esta información en la pestaña **Permissions** de la ventana **Settings...** de nuestra aplicación:



Desde esta ventana, pulsando sobre **Add** podremos añadir permisos de forma visual:



Aquí podremos seleccionar los permisos deseados entre todos los permisos proporcionados por la API que estemos utilizando.

12.3. Dominios

Los permisos que se le otorguen a cada aplicación MIDP dependerán del dominio en el que se encuentre dicha aplicación. Un dominio comprende:

- Un conjunto de permisos que se otorgarán a los MIDlets que pertenezcan a este dominio.
- Un serie de criterios para decidir qué MIDlets pertenecen a dicho dominio.

Podemos distinguir distintos tipos de dominios según lo restrictivos que sean:

- Las aplicaciones que se ejecuten en un dominio de confianza tendrán permiso para establecer cualquiera de las conexiones vistas en el punto anterior.
- En el otro extremo tendremos un dominio de máxima seguridad, en el que se denieguen todas estas operaciones. En esta caso, cada vez que la aplicación quiera establecer una conexión se producirá una excepción de tipo `SecurityException`.
- También podemos encontrar un punto intermedio, en el que cada vez que se va a realizar una operación de este tipo, no se deniegue directamente, sino que se pregunte al usuario si autoriza la realización de dicha operación. Si la autoriza, la operación podrá realizarse, y si no lo hace, se producirá una excepción de tipo `SecurityException`.

Hemos visto tres casos de dominios extremos como ejemplo, pero podemos tener muchos más tipos de dominios distintos. Un dominio se definirá otorgando a cada operación sensible un determinado tipo de permiso. En un dominio se pueden conceder permisos de dos formas distintas:

- Permisos concedidos (`allow`): Se concede el permiso para que la aplicación pueda realizar la correspondiente operación sin tener que obtener la confirmación del usuario.
- Permisos de usuario: Cuando la aplicación vaya a realizar dicha operación, se pedirá la confirmación del usuario. Según cuantas veces se pida esta confirmación distinguimos los siguientes tipos:
 - `oneshot`: Se pedirá confirmación al usuario cada vez que se vaya a realizar la operación.
 - `session`: Se pedirá confirmación al usuario sólo la primera vez que se vaya a realizar la operación en cada sesión. Cuando se cierre la aplicación y vuelva a abrirse, se volverá a realizar la pregunta.
 - `blanket`: Se pedirá confirmación sólo la primera vez que se vaya a realizar la operación. La aplicación recordará la opción que tomó el usuario en las sucesivas sesiones, hasta que sea desinstalada.

Esta definición de los dominios será responsabilidad del fabricante del dispositivo, por lo que no deberemos ocuparnos de ello. Nuestras aplicaciones

simplemente serán asignadas a uno de los dominios disponibles en el dispositivo que las instalemos.

12.3.1. Dominios en dispositivos reales

Normalmente, los dispositivos reales incluyen a las aplicaciones por defecto en un dominio como el último. Es decir, cualquier aplicación que instalemos se considera que no es de confianza, y se le preguntará al usuario cada vez que la aplicación vaya a realizar una operación restringida.

Además, en el caso de los dispositivos reales, también se suele poder cambiar la configuración de este dominio por defecto en la pantalla de configuración del AMS de nuestro móvil. Por ejemplo, en el caso del Nokia 6600, podremos cambiar los permisos que se le otorgan a cada aplicación manualmente. Para cada operación restringida nos dará 4 posibilidades:

- No permitido
- Preguntar la primera vez
- Preguntar siempre
- Siempre permitido

Deberemos tener cuidado de no asignarle permisos a una aplicación que no sea de nuestra confianza.

La forma en la que el dispositivo decide en qué dominio se debe incluir la aplicación no está especificada en MIDP, sin embargo se recomienda la utilización de firmas criptográficas y certificados.

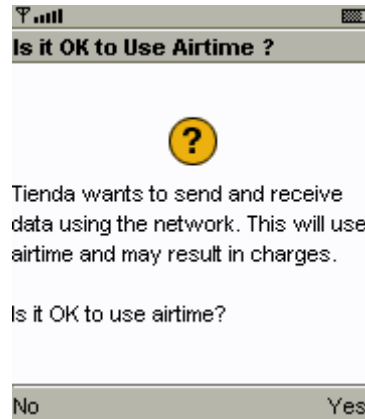
De esta forma, si la aplicación instalada no lleva una firma de confianza, la aplicación se incluirá en el dominio definido por defecto de aplicaciones que no son de confianza. Si por el contrario, contiene una firma reconocida por el dispositivo, la aplicación se añadirá a un dominio de confianza correspondiente a dicha firma, que nos otorgará un determinado conjunto de permisos para los cuales no será necesaria la confirmación del usuario.

Según la firma se puede añadir la aplicación a diferentes dominios, y en cada uno de ellos se pueden conceder o denegar distintas operaciones. De esta forma, puede ocurrir que una aplicación firmada por A tenga sólo permiso para utilizar conexiones HTTP, una aplicación firmada por B tenga sólo permiso para utilizar sockets, y una firmada por C tenga permiso para utilizar cualquier tipo de conexión. Los distintos tipos de dominios definidos y la asignación de permisos a cada uno de ellos dependerá del fabricante del dispositivo.

12.3.2. Dominios en los emuladores

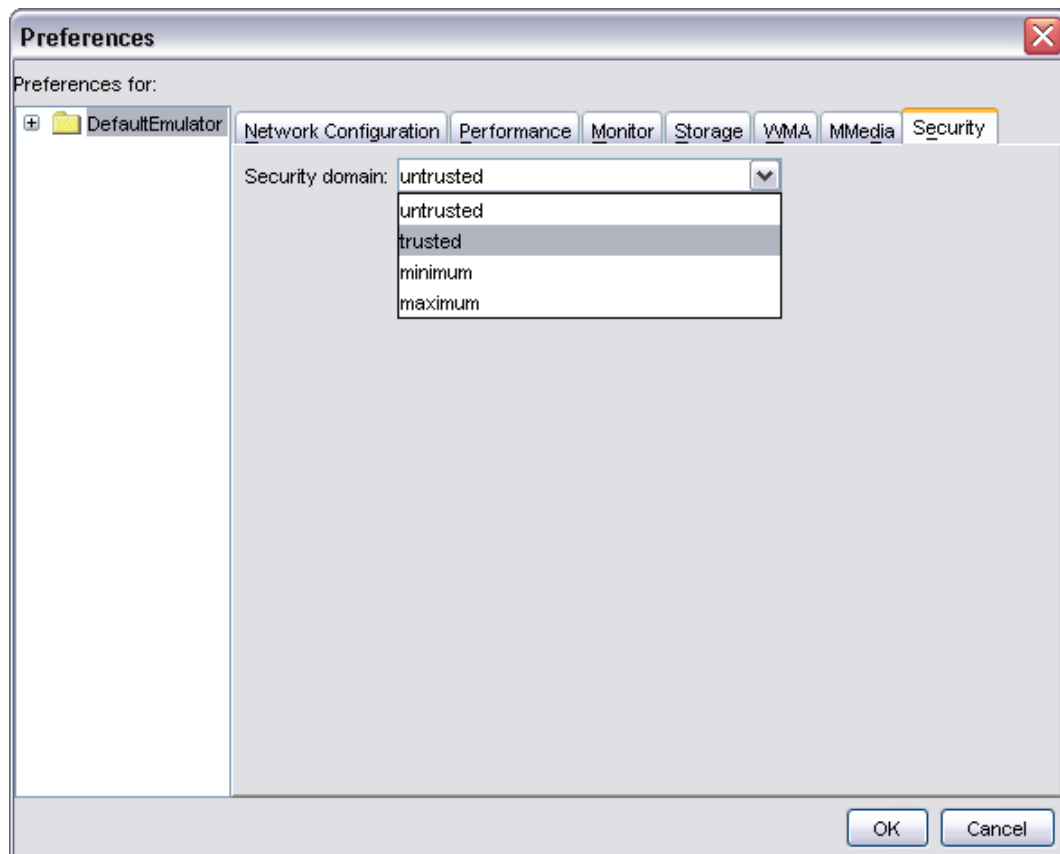
Los emuladores incluidos en WTK definen un conjunto de 4 dominios en los que se podrán incluir las aplicaciones. Estos dominios son:

- *minimum*: No se permite realizar ninguna operación restringida. Cuando se intente realizar alguna de ellas se producirá una excepción de tipo `SecurityException` directamente.
- *untrusted*: Cada vez que se intente realizar una operación no permitida se preguntará al usuario si desea permitirlo o no:



- *trusted/maximum*: La aplicación puede realizar cualquier operación directamente, sin necesidad de preguntar al usuario.

Podemos indicar en la configuración de WTK cual será el dominio por defecto en el que se ejecutarán las aplicaciones cuando las carguemos directamente en el emulador (sin utilizar OTA). Podemos cambiar este dominio por defecto en la ventana de preferencias (**Preferences...**) del WTK.



Cuando se utilice provisionamiento OTA, el emulador se comportará como un dispositivo real, asignando a la aplicación un dominio según su firma.

12.4. Firmar MIDlets

En la especificación de MIDP no se establece ningún método para decidir a qué dominio corresponde cada MIDlet, pero se recomienda que para tomar esta decisión se utilicen firmas y certificados.

Cada dispositivo contendrá una serie de certificados de confianza. Cada uno de estos certificados estará asociado a un determinado dominio. Es responsabilidad del fabricante decidir qué certificados se incluyen en el dispositivo y a qué dominio se asocia cada uno.

12.4.1. Certificados en los dispositivos

Para que a un MIDlet se le otorguen ciertos permisos, deberá estar firmado por un certificado que sea conocido por el dispositivo donde se instala. Al instalar el MIDlet en el dispositivo, si está firmado se comprobará si el dispositivo contiene el certificado que se ha utilizado para firmarlo. De ser así, se comprobará la autenticidad de la aplicación descargada mediante este certificado, y de ser correcta se instalará la aplicación en el dominio que el dispositivo tuviese asociado a dicho certificado.

Si el certificado con el que se ha firmado la aplicación no fuese conocido por el dispositivo, simplemente instalará la aplicación en el dominio de aplicaciones que no son de confianza. Normalmente en este dominio se preguntará al usuario cada vez que se va a realizar una operación restringida.

Por lo tanto, para conseguir que a nuestro MIDlet se le otorguen permisos deberemos firmarlo por un certificado que esté incluido en el dispositivo.

La especificación de MIDP 2.0 recomienda que se incluyan 3 tipos de certificados:

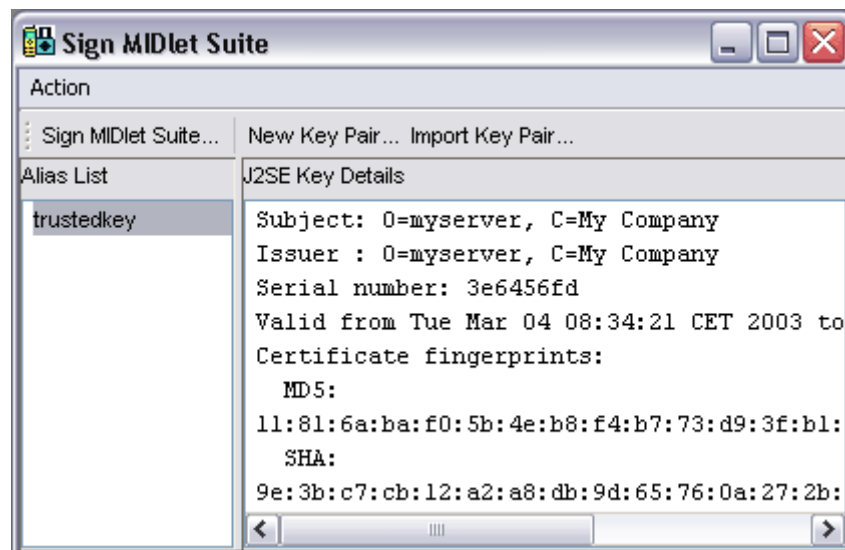
- **Del fabricante:** Certificados pertenecientes al fabricante del dispositivo. Por ejemplo, en el móviles de Nokia, se pueden incluir certificados con los que Nokia pueda firmar las aplicaciones que realice.
- **Del operador:** Certificados pertenecientes al operador de telefonía que utiliza el móvil. Estos certificados pueden almacenarse en la tarjeta SIM, ya que es propia del operador de telefonía utilizado. Por ejemplo, si en nuestro móvil utilizamos una tarjeta Movistar, en ella se pueden almacenar certificados de esta compañía, de forma que podamos instalar sus aplicaciones en un dominio de confianza.
- **De terceros:** Certificados pertenecientes a terceros adquiridos de Autoridades Certificadoras como Verisign o Thwate. Cada dispositivo contendrá un conjunto de estos certificados. Deberemos asegurarnos de que el certificado con el que vayamos a firmar nuestra aplicación esté incluido en los dispositivos en los que vamos a desplegarla.

Si el certificado con el que hemos firmado nuestra aplicación no está entre los anteriores, o bien no hemos firmado la aplicación, ésta se instalará en el dominio de aplicaciones no fiables que existirá en cualquier dispositivo.

12.4.2. Certificados en WTK

Para que nuestra aplicación obtenga permisos en un dispositivo móvil real deberíamos tenerla firmada por un certificado incorporado en el dispositivo en el que la instalamos. Sin embargo, cuando la probemos en emuladores podremos crear nuestros propios certificados y añadirlos al emulador, sin necesidad de obtenerlos a partir de una Autoridad Certificadora. De esta forma en el emulador las aplicaciones firmadas por nosotros podrán obtener los permisos que solicitemos.

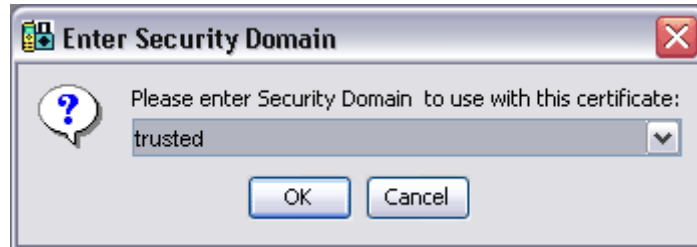
Vamos a ver como firmar aplicaciones MIDP utilizando WTK. Para firmar una aplicación, lo primero que deberemos hacer es crear el paquete con la aplicación, ya que es necesario contar con el fichero JAR para obtener su *digest*. Una vez creado el paquete utilizaremos la opción **Project > Sign** para firmar la aplicación. Aparecerá la siguiente ventana:



Aquí podremos utilizar alguna de las claves disponibles para firmar nuestra aplicación, o bien crear un nuevo par de claves. Pulsaremos sobre **New Key Pair...** para crear nuestras propias claves:



Aquí introducimos información sobre nuestra compañía, para crear un certificado correspondiente a esta compañía. Cuando pulsemos sobre **Create** se creará este certificado asignándole su correspondiente par de claves, y nos mostrará la siguiente ventana para indicar a qué dominio se asignarán las aplicaciones firmadas por nosotros:



De esta forma, nuestro certificado será instalado en el emulador. Cuando instalemos una aplicación firmada por nosotros en el emulador, esta aplicación se instalará en el dominio que hayamos indicado aquí. Si elegimos *trusted*, nuestras aplicaciones dispondrán de todos los permisos sin tener que pedir la confirmación del usuario.

El último paso que debemos realizar es firmar nuestra aplicación utilizando el certificado que acabamos de crear. Para esto simplemente seleccionaremos en la ventana **Sign MIDlet Suite** el certificado que queremos utilizar, y pulsaremos sobre el botón **Sign MIDlet Suite...**

Una vez firmada, podremos probarla en el emulador para comprobar que se le otorgan los permisos correspondientes. Para que esto sea así, deberemos probarla utilizando provisionamiento OTA, ya que si lo hacemos de otra forma la firma no se tendría en cuenta.

Recordemos que si ejecutamos la aplicación directamente se tiene en cuenta el dominio que se haya configurado por defecto para los emuladores. Cuando utilizamos OTA se comportará como un dispositivo real, utilizando los certificados de los que dispone para asignar un dominio a nuestra aplicación.

13. Aplicaciones corporativas

El poder establecer conexiones en red nos permitirá acceder a aplicaciones web corporativas desde el móvil. De esta forma, podremos hacer que estos dispositivos móviles se comporten como *front-end* de estas aplicaciones corporativas.

13.1. Front-ends de aplicaciones corporativas

Desde los PCs de sobremesa normalmente accedemos a las aplicaciones corporativas utilizando un navegador web. La aplicación web genera de forma dinámica la presentación en el servidor en forma de un documento HTML que será mostrado en los navegadores de los clientes. Podemos aplicar este mismo sistema al caso de los móviles, generando nuestra aplicación web la respuesta en forma de algún tipo de documento que pueda ser interpretado y mostrado en un navegador de teléfono móvil. Por ejemplo estos documentos pueden estar en formato WML, cHTML o XHTML. Esto puede ser suficiente para acceder a algunas aplicaciones desde los móviles.

Utilizar J2ME para realizar este *front-end* aporta una serie de ventajas sobre el paradigma anterior, como por ejemplo las siguientes:

- **Interfaz de usuario.** Las aplicaciones J2ME nos permite crear una interfaz de usuario flexible. La capa de presentación esta implementada totalmente en el cliente, por lo que se podrá ajustar mejor a las características de cada dispositivo.
- **Funcionamiento sin conexión.** La aplicación J2ME se ejecuta de forma local en el móvil, por lo que podremos trabajar sin conexión. Podemos utilizar RMS para guardar datos mientras trabajamos en forma local, y conectar al servidor únicamente cuando sea necesario.
- **Conexión HTTP.** Estas aplicaciones se conectan utilizando el protocolo estándar HTTP. No hará falta conocer la estructura de la red móvil subyacente. El poder establecer conexiones HTTP a Internet nos permitirá enlazar las aplicaciones J2EE y J2ME.

Normalmente será preferible utilizar HTTP a *sockets* o datagramas porque esto nos aportará una serie de ventajas. Por un lado, HTTP está soportado por todos los dispositivos MIDP. Al utilizar HTTP tampoco tendremos problema con *firewalls* intermedios, cosa que puede ocurrir si conectamos por *sockets* mediante un puerto que esté cerrado. Además las APIs de Java incluyen facilidades para trabajar con HTTP, por lo que será sencillo realizar la comunicación tanto en el cliente como en el servidor.

La conexión de red en los móviles normalmente tiene una alta latencia, un reducido ancho de banda y es posible que se produzcan interrupciones cuando la cobertura es baja. Debemos tener en cuenta todos estos factores cuando diseñemos nuestra aplicación. Por esta razón deberemos minimizar la cantidad de datos que se intercambian a través de la red, y permitir que la aplicación pueda continuar trabajando correctamente sin conexión.

13.1.1. Tráfico en la red

Para reducir el número de datos que se envían por la red, evitando que se hagan conexiones innecesarias, es conveniente realizar una validación de los datos introducidos por el usuario en el cliente.

Normalmente no podremos validarlos de la misma forma en que se validan en el servidor, ya que por ejemplo no tenemos acceso a las bases de datos de la aplicación, por lo que deberá volverse a validar por el servidor para realizar la validación completa. No obstante, es conveniente realizar esta validación en el cliente como una prevalidación, de forma que detecte siempre que sea posible los datos erróneos en el lado del cliente evitando así realizar una conexión innecesaria con el servidor.

Deberemos enviar y recibir sólo la información necesaria por la red. Podemos reducir este tráfico manteniendo en RMS una copia de los datos remotos que obtengamos (caché), para no tener que volver a solicitarlos si queremos volver a visualizarlos.

13.1.2. Operaciones de larga duración

Dado que la red es lenta, las operaciones que necesiten conectarse a la red serán costosas. Estas operaciones será conveniente que sean ejecutadas por hilos en segundo plano, y nunca deberemos establecer una conexión desde un *callback*. Otro tipo de operaciones que normalmente son de larga duración son las consultas de datos en RMS.

Cualquier operación que sea costosa temporalmente deberá ser ejecutada de esta forma, para evitar que bloquee la aplicación.

Además siempre que sea posible deberemos mostrar una barra de progreso mientras se realiza la operación, de forma que el usuario tenga constancia de que se está haciendo algo. También será conveniente permitir al usuario que interrumpa estas largas operaciones, siempre que la interrupción pueda hacerse y no cause inconsistencias en los datos.

13.1.3. Personalización

Un aspecto interesante en los clientes J2ME es la posibilidad de incorporar personalización. La personalización consiste en recordar los datos y las preferencias del usuario, de forma que la aplicación se adapte a estas preferencias y el usuario no tenga que introducir estos datos en cada sesión. Podremos pedir esta información de personalización la primera vez que ejecuta la aplicación y almacenar esta información utilizando RMS o bien registrarla en el servidor de forma remota. Si tuviésemos esta información por duplicado, en local y en remoto, deberemos proporcionar mecanismos para sincronizar ambos registros.

Normalmente los dispositivos móviles son personales, por lo que las aplicaciones instaladas en un móvil sólo va a utilizarlas una persona. Por este

motivo puede ser conveniente guardar la información del usuario para que no tenga que volver a introducirla cada vez que utilice la aplicación. Podemos hacer que la aplicación recuerde el login y el password del usuario.

Podemos incluir en la ficha de datos del usuario información sobre sus preferencias, de forma que la aplicación se adapte a sus gustos y resulte más cómoda de manejar.

13.2. Integración con aplicaciones corporativas

Vamos a considerar que en el servidor tenemos una aplicación J2EE. En este caso accederemos a la aplicación corporativa a través de un *Servlet*. Los *servlets* son componentes Java en el servidor que encapsulan el mecanismo petición/respuesta. Es decir, podemos enviar una petición HTTP a un *servlet*, y éste la analizará y nos devolverá una respuesta.

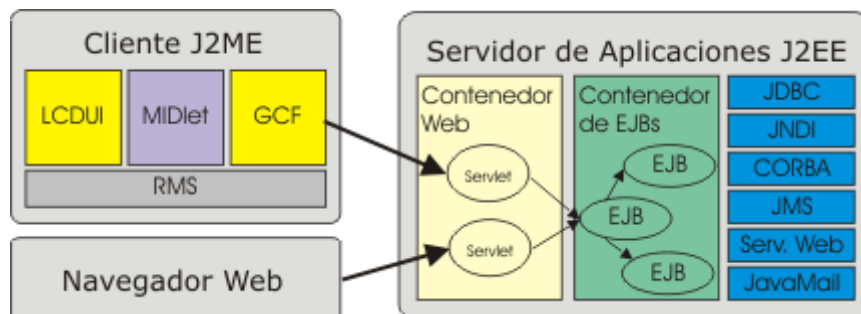


Figura 1. Integración de J2ME y J2EE

La aplicación J2ME se comunicará con un *servlet*. Dentro de la aplicación J2EE en el servidor este *servlet* utilizará EJBs para realizar las tareas necesarias. Los EJBs son componentes reutilizables que implementan la lógica de negocio de la aplicación. Estos EJBs podrán utilizar otras APIs para realizar sus funciones, como por ejemplo JMS para enviar o recibir mensajes, JDBC para acceder a bases de datos, CORBA para acceder a objetos distribuidos, o bien acceder a Servicios Web utilizando las APIs de XML.

Normalmente tendremos *servlets* que se encarguen de generar una respuesta para navegadores web HTML. Para las aplicaciones móviles esta respuesta no es adecuada, por lo que deberemos crear otra versión de estos *servlets* que devuelvan una respuesta adaptada a las necesidades de los móviles. La arquitectura de capas de J2EE nos permitirá crear *servlets* para distintos tipos de clientes minimizando la cantidad de código redundante, ya que la lógica de negocio está implementada en los EJBs y estos componentes pueden ser reutilizados desde los diferentes *servlets*.

El proceso de comunicación entre nuestra aplicación y un *servlet* será el siguiente:

- La aplicación J2ME envía la petición al *servlet*. Para ello establece como URL la dirección en la que está mapeado el *servlet* y añade al mensaje

el contenido que queramos enviar en la petición. Si incluimos contenido en un mensaje de tipo POST deberemos establecer la cabecera `Content-Type` al tipo correcto de datos que estemos enviando para que el mensaje sea procesado de forma correcta por los *gateways* intermedios por los que pase. Si enviamos el contenido del mensaje codificado como texto, deberemos establecer este tipo a `text/plain`, mientras que si los datos se envían en binario deberemos utilizar como tipo `application/octet-stream`.

- El *servlet* recibe la petición, la descodifica, la analiza y en caso necesario utiliza los EJBs que sean necesarios para realizar las acciones oportunas.
- Una vez se ha procesado la petición el *servlet* construye la respuesta con los resultados obtenidos y la codifica. Deberemos establecer las cabeceras `Content-Type` y `Content-Length` para asegurarnos de que el mensaje sea procesado correctamente por los *gateways* intermedios igual que en el caso del envío de la petición. En este caso como tipo podremos utilizar por ejemplo `text/plain` para texto `image/png` para devolver una imagen PNG al cliente, y `application/octet-stream` para devolver un mensaje codificado en binario.
- Para finalizar, el cliente recibirá y descodificará la respuesta que le ha enviado el *servlet*. Una vez descodificada podrá mostrarla en pantalla utilizando la API LCDUI.

13.2.1. Codificación de los datos

Hemos visto que la aplicación J2ME y el *servlet* de la aplicación J2EE intercambian datos con una determinada codificación. En J2ME no tenemos disponibles mecanismos de alto nivel para intercambiar información con componentes remotos, como por ejemplo RMI para la invocación de métodos de objetos Java remotos, o las API de análisis de XML para intercambiar información en este formato. Por lo tanto deberemos codificar la información con formatos propios. Seremos nosotros los que decidamos qué formato y codificación deben tener estos datos.

Podemos movernos entre dos extremos: la codificación de los datos en binario y la codificación en XML.

La codificación binaria de los datos será eficiente y compacta. Será sencillo codificar información en este formato utilizando los objetos `DataOutputStream` y `ByteArrayOutputStream`. Tiene el inconveniente de que tanto el cliente como el servidor deberán conocer cómo está codificada la información dentro del *array* de *bytes*, por lo que estos componentes estarán altamente acoplados.

Si hemos definido una serialización para los objetos, podemos aprovechar esta serialización para enviarlos a través de la red. En este caso la serialización la hemos definido manualmente nosotros en un método del objeto, y no se hace automáticamente como en el caso de J2SE, por lo que deberemos tener cuidado de que en el objeto del cliente y en el del servidor se serialice y deserialice de la misma forma. Además, al transferir un objeto entre J2ME y

J2EE deberemos asegurarnos de que este objeto utiliza solamente la parte común de la API de Java en ambas plataformas.

Por ejemplo, si definimos los métodos de serialización para la clase `Cita` de la siguiente forma:

```
public class Cita {  
  
    Date fecha;  
    String asunto;  
    String lugar;  
    String contacto;  
    boolean alarma;  
  
    public void serialize(DataOutputStream dos)  
                                throws IOException {  
        dos.writeLong(fecha.getTime());  
        dos.writeUTF(asunto);  
        dos.writeUTF(lugar);  
        dos.writeUTF(contacto);  
        dos.writeBoolean(alarma);  
    }  
  
    public static Cita deserialize(DataInputStream dis)  
                                throws IOException {  
        Cita cita = new Cita();  
  
        cita.setFecha(new Date(dis.readLong()));  
        cita.setAsunto(dis.readUTF());  
        cita.setLugar(dis.readUTF());  
        cita.setContacto(dis.readUTF());  
        cita.setAlarma(dis.readBoolean());  
  
        return cita;  
    }  
}
```

Podremos intercambiar objetos de este tipo mediante HTTP como se muestra a continuación:

```
HttpConnection con = (HttpConnection)  
Connector.open(URL_SERVLET);  
  
// Envía datos al servidor  
DataOutputStream dos = con.openDataOutputStream();  
Cita[] citasCliente = datosCliente.getCitas();  
if (citasCliente == null) {  
    dos.writeInt(0);  
} else {  
    dos.writeInt(citasCliente.length);  
    for (int i = 0; i < citasCliente.length; i++) {  
        citasCliente[i].serialize(dos);  
    }  
}  
  
// Recibe datos del servidor  
DataInputStream dis = con.openDataInputStream();  
int nCitasServidor = dis.readInt();  
Cita[] citasServidor = new Cita[nCitasServidor];
```

```
for (int i = 0; i < nCitasServidor; i++) {  
    citasServidor[i] = Cita.deserialize(dis);  
}
```

En el otro extremo, XML es un lenguaje complejo de analizar y la información ocupa más espacio. Como ventaja tenemos que XML es un lenguaje estándar y autodescriptivo, por lo que reduciremos el acoplamiento de cliente y servidor. Aunque en MIDP no se incluyen librerías para procesar XML, diversos fabricantes proporcionan sus propias implementaciones de las librerías de XML para J2ME. Podemos utilizar estas librerías para crear y analizar estos documentos en los clientes móviles.

Podemos encontrar incluso implementaciones de librerías de XML orientado a RPC para invocar Servicios Web SOAP directamente desde el móvil. Debemos tener cuidado al invocar Servicios Web directamente desde los móviles, ya que la construcción y el análisis de los mensajes SOAP es una tarea demasiado costosa para estos dispositivos. Para optimizar la aplicación, en lugar de invocarlos desde el mismo móvil, podemos hacer que sea la aplicación J2EE la que invoque el servicio, y que el móvil se comunique con esta aplicación a través de mensajes sencillos.

13.2.2. Mantenimiento de sesiones

El protocolo HTTP sigue un mecanismo de petición/respuesta, no mantiene información de sesión. Es decir, si realizamos varias peticiones HTTP a un servidor desde nuestro cliente, cada una de estas peticiones será tratada independientemente por el servidor, sin identificar que se trata de un mismo usuario. Para implementar sesiones sobre protocolo HTTP tendremos que recurrir a mecanismos como la reescritura de URLs o las *cookies*.

Normalmente cuando accedamos a una aplicación web necesitaremos mantener una sesión para que en todas las peticiones que hagamos al servidor, éste nos identifique como un mismo usuario. De esta forma por ejemplo podremos ir añadiendo con cada petición productos a un carrito de la compra en el lado del servidor, sin perder la información sobre los productos añadidos de una petición a otra.

Para mantener las sesiones lo que se hará es obtener en el cliente un identificador de la sesión, de forma que en cada petición que se haga al servidor se envíe este identificador para que el servidor sepa a qué sesión pertenece dicha petición. Este identificador puede ser obtenido mediante *cookies*, o bien incluirlo como parte de las URLs utilizando la técnica de reescritura de URLs.

Los navegadores web normalmente implementan las sesiones mediante *cookies*. Estas *cookies* son información que el servidor nos envía en la respuesta y que el navegador almacena de forma local en nuestra máquina. En la primera petición el servidor enviará una *cookie* al cliente con el identificador de la sesión, y el navegador almacenará esta *cookie* de forma local en el cliente. Cuando se vaya a hacer otra petición al servidor, el navegador envía esta *cookie* para identificarnos ante el servidor como el mismo cliente. De esta

forma el servidor podrá utilizar el valor de la *cookie* recibida para determinar la sesión correspondiente a dicho cliente, y de esta forma poder acceder a los datos que hubiese almacenado en peticiones anteriores dentro de la misma sesión.

Sin embargo, cuando conectamos desde un cliente J2ME estamos estableciendo una conexión con la URL del *servlet* desde nuestra propia aplicación, no desde un navegador que gestione automáticamente estas *cookies*. Por lo tanto será tarea nuestra implementar los mecanismos necesarios para mantener esta sesión desde el cliente.

Vamos a ver como implementar los mecanismos para mantenimiento de sesiones en las aplicaciones J2ME. Será más fiable utilizar reescritura de URLs, ya que algunos *gateways* podrían filtrar las *cookies* y por lo tanto este mecanismo fallaría.

Estos mecanismos de *cookies* y reescritura de URLs se utilizan para que los navegadores mantengan las sesiones de una forma estándar para todas las aplicaciones. Pero lo que pretendemos en última instancia es tener un identificador de la sesión en el cliente que pueda ser enviado al servidor en cada petición. Si nos conectamos desde nuestra propia aplicación podremos utilizar nuestro propio identificador y enviarlo al servidor de la forma que queramos (como cabecera, parámetro, en el post, etc). Por ejemplo, podríamos hacer que en cada petición que haga nuestra aplicación J2ME envíe nuestro *login* al servidor, de forma que esta información le sirva al servidor para identificarnos en cada momento.

Sin embargo, será conveniente que nuestra aplicación implemente alguno de los mecanismos estándar para el mantenimiento de sesiones, ya que así podremos aprovechar las facilidades que ofrecen los componentes del servidor para mantener las sesiones. Ahora veremos como implementar las técnicas de reescritura de URLs y *cookies* en nuestras aplicaciones J2ME.

Reescritura de URLs

Algunos navegadores no soportan *cookies*. Para mantener sesiones en este caso podemos utilizar la técnica de reescritura de URLs. Esta técnica consiste en modificar las URLs a las que accederá el cliente incluyendo en ellas el identificador de sesión como parámetro.

Para utilizar esta técnica deberemos codificar la URL en el servidor y devolverla de alguna forma al cliente. Por ejemplo, podemos devolver esta URL modificada como una cabecera HTTP propia. Supongamos que devolvemos esta URL reescrita como una cabecera URL-Reescrita. Podremos obtenerla en la aplicación cliente de la siguiente forma:

```
String url_con_ID = con.getHeaderField("URL-Reescrita");
```

En la próxima petición que hagamos al servidor deberemos utilizar la URL que hemos obtenido, en lugar de la URL básica a la que conectamos inicialmente:


```
HttpConnection con =  
(HttpConnection)Connector.open(url_con_ID);
```

De esta forma cuando establezcamos esta segunda conexión el *servlet* al que conectamos sabrá que se trata de la misma sesión y podremos acceder a la información de la sesión dentro del servidor.

En el código del *servlet* que atiende nuestra petición en el servidor deberemos rescribir la URL y devolvérsela al cliente como la cabecera que hemos visto anteriormente. Esto podemos hacerlo de la siguiente forma:

```
String url = request.getRequestURL().toString();  
String url_con_ID = response.encodeURL(url);  
response.setHeader("URL-Reescrita", url_con_ID);
```

Manejo de cookies

Los *servlets* utilizan las *cookies* para mantener la sesión siempre que detecten que el cliente soporta *cookies*. Para el caso de estas aplicaciones J2ME el *servlet* detectará que el cliente no soporta *cookies*, por lo que utilizará únicamente reescritura de URLs. Sin embargo, si que podremos crear *cookies* manualmente en el *servlet* para permitir mantener la información del usuario en el cliente durante el tiempo que dure la sesión o incluso durante más tiempo.

Desde las aplicaciones J2ME podremos implementar las sesiones utilizando *cookies*. Además con las *cookies* podremos mantener información del usuario de forma persistente, no únicamente durante una sola sesión. Por ejemplo, podemos utilizar RMS para almacenar las *cookies* con información sobre el usuario, de forma que cuando se vuelve a utilizar la aplicación en otro momento sigamos teniendo esta información. Con esto podemos por ejemplo evitar que el usuario tenga que autenticarse cada vez que entra a la aplicación.

Estas *cookies* consisten en una pareja *<nombre, valor>* y una fecha de caducidad. Con esta fecha de caducidad podemos indicar si la *cookie* debe mantenerse sólo durante la sesión actual, o por más tiempo.

Podemos recibir las *cookies* que nos envía el servidor leyendo la cabecera *set-cookie* de la respuesta desde nuestra aplicación J2ME:

```
String cookie = con.getHeaderField("set-cookie");
```

Una vez tengamos la *cookie* podemos guardárnosla en memoria, o bien en RMS si queremos almacenar persistentemente esta información. En las siguientes peticiones que hagamos al servidor deberemos enviarle esta *cookie* en la cabecera *cookie*:

```
con.setRequestProperty("cookie", cookie);
```

13.2.3. Seguridad

Vamos a ver como crear aplicaciones seguras en cuando a la autenticación de los usuarios y a la confidencialidad de los datos que circulan por la red.

Confidencialidad

Para proteger los datos que circulan por la red, y evitar que alguien pueda interceptarlos, podemos enviarlos codificados mediante SSL.

Podemos establecer una conexión segura con el servidor mediante SSL (*Secure Sockets Layer*) simplemente indicando como protocolo en la URL de la conexión `https` en lugar de `http`.

El problema es que la mayoría de móviles con MIDP 1.0 no soportan este tipo de protocolo. Si intentamos utilizar HTTPS desde un dispositivo que no lo soporta, obtendremos una excepción de tipo `ConnectionNotFoundException`. Además de tener que se soportada por el móvil, este tipo de conexión deberá ser soportada por el servidor.

Autenticación

En muchas aplicaciones necesitaremos que el usuario se autentique para que de esa forma pueda acceder a determinados datos privados alojados en el servidor, o pueda realizar determinadas acciones para las que no todos los usuarios tienen permiso.

Para autenticar a los usuarios que se conectan desde el móvil a las aplicaciones corporativas, normalmente utilizaremos seguridad a nivel de la aplicación. Es decir, enviaremos nuestros credenciales (*login* y *password*) a un *servlet* para que los verifique en la base de datos de usuarios que haya en el servidor.

Una vez autenticados, podremos mantener esta información de registro en la sesión del usuario, de forma que en sucesivas peticiones podamos obtener la información sin necesidad de volvernos a autenticar en cada una de ellas.

13.2.4. Transacciones

Cada petición HTTP que se haga el servidor será una transacción independiente. Todas las operaciones de la transacción se deben realizar dentro de esa conexión en el servidor, ya que las peticiones HTTP son independientes entre si y el dispositivo cliente móvil no realiza ninguna gestión de transacciones. Por lo tanto las transacciones nunca supondrán más de una petición HTTP.

En el caso en el que la inserción de unos datos en la memoria local dependa de que los datos sean aceptados por el servidor remoto, siempre realizaremos antes la petición al servidor remoto, y una vez hayamos enviados estos datos correctamente, haremos la inserción local. Esto se da por ejemplo en el caso

en que queremos mantener una copia local de los datos que enviamos al servidor. Sólo insertaremos los datos en la copia local una vez hayan sido insertados en el servidor.

13.2.5. Gestión de errores

Cuando estemos accediendo al lado del servidor de nuestra aplicación mediante HTTP, si la petición que hemos hecho produce un error no podremos gestionar este error mediante excepciones ya que lo único que vamos a recibir es una respuesta HTTP.

Para gestionar los errores del servidor deberemos almacenarlos de alguna forma en el mensaje de respuesta HTTP.

Podemos utilizar el código de estado de la respuesta para indicar la presencia de algún error. Si necesitamos una mayor flexibilidad podemos establecer nuestra propia codificación para los errores dentro del contenido de la respuesta, y hacer que nuestro cliente sea capaz de interpretar dicha codificación.

13.3. Arquitectura MVC

El patrón de diseño MVC (Modelo-Vista-Controlador) suele aplicarse para diseñar las aplicaciones web J2EE. Podemos aplicar este mismo patrón a las aplicaciones cliente J2ME. En una aplicación cliente tendremos los siguientes elementos:

- **Modelo:** Datos de la aplicación.
- **Vista:** Presentación de la aplicación. Serán las distintas pantallas de nuestro MIDlet.
- **Controlador:** El controlador será quien controle el flujo de la aplicación. Nos dirá qué pantalla se debe mostrar y qué operaciones se deben ejecutar en cada momento, según las acciones realizadas.

De esta forma con esta arquitectura estamos aislando datos, presentación y flujo de control. Es especialmente interesante el haber aislado los datos del resto de componentes. De esta forma la capa de datos podrá decidir si trabajar con datos de forma local con RMS o de forma remota a través de HTTP, sin afectar con ello al resto de la aplicación. Este diseño nos facilitará cambiar de modo remoto a modo local cuando no queramos tener que establecer una conexión de red.

13.3.1. Modelo

Normalmente en las aplicaciones para dispositivos móviles vamos a trabajar tanto con datos locales como con datos remotos. Por lo tanto, podemos ver el modelo dividido en dos subsistemas: modelo local y modelo remoto.

Modelo local

El modelo local normalmente utilizará un adaptador RMS para acceder a los datos almacenados de forma persistente en el móvil. Por ejemplo, en nuestra aplicación de agenda para la gestión de citas podemos crear la siguiente clase para acceder a los datos locales:

```
public class ModeloLocal {

    AdaptadorRMS rms;

    public ModeloLocal() throws RecordStoreException {
        rms = new AdaptadorRMS();
    }

    /*
     * Agrega una cita indicando si esta pendiente de ser
     * enviada al servidor
     */
    public void addCita(Cita cita, boolean pendiente)
        throws IOException, RecordStoreException {
        int id = rms.addCita(cita);
        IndiceCita indice = new IndiceCita();
        indice.setId(id);
        indice.setFecha(cita.getFecha());
        indice.setAlarma(cita.isAlarma());
        indice.setPendiente(pendiente);
        rms.addIndice(indice);
    }

    /*
     * Obtiene todas las citas
     */
    public Cita[] listaCitas()
        throws RecordStoreException, IOException {
        return rms.listaCitas();
    }

    /*
     * Obtiene las citas correspondientes a los indices
     * indicados
     */
    public Cita[] listaCitas(IndiceCita[] indices)
        throws RecordStoreException, IOException {
        Cita[] citas = new Cita[indices.length];
        for (int i = 0; i < indices.length; i++) {
            citas[i] = rms.getCita(indices[i].getId());
        }

        return citas;
    }

    /*
     * Busca las citas con alarmas pendientes
     */
    public IndiceCita[] listaAlarmasPendientes()
        throws RecordStoreException, IOException {
        IndiceCita[] indices = rms.buscaCitas(new Date(), true);
        return indices;
    }
}
```

```
}

/*
 * Busca las citas pendientes de ser enviadas al servidor
 */
public IndiceCita[] listaCitasPendientes()
    throws RecordStoreException, IOException {
    IndiceCita[] indices = rms.listaCitasPendientes();
    return indices;
}

/*
 * Marca las citas indicada como enviadas al servidor
 */
public void marcaEnviados(IndiceCita[] indices)
    throws IOException, RecordStoreException {
    for (int i = 0; i < indices.length; i++) {
        indices[i].setPendiente(false);
        rms.updateIndice(indices[i]);
    }
}

/*
 * Obtiene la configuracion local
 */
public InfoLocal getInfoLocal()
    throws RecordStoreException, IOException {

    try {
        InfoLocal info = rms.getInfoLocal();
        return info;
    } catch (Exception e) { }

    InfoLocal info = new InfoLocal();
    rms.setInfoLocal(info);

    return info;
}

/*
 * Modifica la configuracion local
 */
public void setInfoLocal(InfoLocal info)
    throws RecordStoreException, IOException {
    rms.setInfoLocal(info);
}

/*
 * Libera recursos del modelo
 */
public void destroy() throws RecordStoreException {
    rms.cerrar();
}
}
```

En esta clase definiremos métodos para acceder a todos los datos locales que nuestra aplicación necesite utilizar.

Modelo remoto

Además de la información local que almacenamos en el móvil, será importante tener acceso a los datos remotos de nuestra aplicación corporativa. Esta parte del modelo normalmente utilizará HTTP para acceder a estos datos.

Para implementar este subsistema del modelo podemos utilizar el patrón de diseño *proxy*. Este patrón se utiliza cuando accedemos a componentes remotos, encapsulando dentro de la clase *proxy* todo el mecanismo de comunicación necesario para acceder a las funcionalidades del servidor.

De esta forma, cuando utilizamos un *proxy* para acceder a componentes remotos, el que estos componentes se estén utilizando de forma remota es transparente para nosotros. El *proxy* implementa todo el mecanismo de comunicación necesario (por ejemplo HTTP), aislando al resto de la aplicación de él. Invocaremos los métodos del *proxy* directamente para utilizar funcionalidades de nuestro servidor como si se tratase de un acceso a un objeto local, sin preocuparnos de que realmente el *proxy* esté realizando una conexión remota.

Por ejemplo, en el caso de nuestra agenda, el servidor nos proporcionará la funcionalidad de sincronizar las citas almacenadas en el móvil con las notas almacenadas en el servidor, para hacer que en ambos lados se tenga el mismo conjunto de citas. Podemos encapsular esta operación en una clase *proxy* como la siguiente:

```
public class ProxyRemoto {

    public ProxyRemoto() {
    }

    public SyncItem sincroniza(SyncItem datosCliente)
        throws IOException {
        HttpURLConnection con = (HttpURLConnection)
            Connector.open(URL_SERVLET);

        // Envía datos al servidor
        DataOutputStream dos = con.openDataOutputStream();
        Cita[] citasCliente = datosCliente.getCitas();
        dos.writeLong(datosCliente.getTimestamp());
        if (citasCliente == null) {
            dos.writeInt(0);
        } else {
            dos.writeInt(citasCliente.length);
            for (int i = 0; i < citasCliente.length; i++) {
                citasCliente[i].serialize(dos);
            }
        }

        // Recibe datos del servidor
        DataInputStream dis = con.openDataInputStream();
        long tsServidor = dis.readLong();
        int nCitasServidor = dis.readInt();
        Cita[] citasServidor = new Cita[nCitasServidor];
        for (int i = 0; i < nCitasServidor; i++) {
            citasServidor[i] = Cita.deserialize(dis);
        }
    }
}
```

```
    }

    SyncItem datosServidor = new SyncItem();
    datosServidor.setTimeStamp(tsServidor);
    datosServidor.setCitas(citasServidor);

    return datosServidor;
}
}
```

De esta forma desde el resto de la aplicación simplemente tendremos que invocar el método `sincroniza` del *proxy* para utilizar esta funcionalidad del servidor.

Patrón de diseño fachada

Hasta ahora hemos visto que tenemos dos subsistemas en el modelo. Esta división podría causar que el acceso al modelo desde nuestra aplicación fuese demasiado complejo.

Para evitar esto podemos utilizar el patrón de diseño fachada (*facade*). Este patrón consiste en implementar una interfaz única que integre varios subsistemas, proporcionando de esta forma una interfaz sencilla para el acceso al modelo.

Desde el resto de la aplicación accederemos al modelo siempre a través de la fachada, y ésta será quien se encargue de coordinar los subsistemas local y remoto. Estos dos subsistemas se mantendrán independientes, pero accederemos a ellos mediante una interfaz única.

Por ejemplo, para nuestra agenda podemos definir una fachada como la siguiente para el modelo:

```
public class FachadaModelo {

    boolean online;
    ModeloLocal mLocal;
    ProxyRemoto mRemoto;

    public FachadaModelo()
        throws RecordStoreException, IOException {
        mLocal = new ModeloLocal();
        mRemoto = new ProxyRemoto();

        InfoLocal info = getConfig();
        online = info.isOnline();
    }

    /*
     * Crea una nueva cita
     */
    public void nuevaCita(Cita cita)
        throws IOException, RecordStoreException {
        mLocal.addCita(cita, true);
        if (online) {
            sincroniza();
        }
    }
}
```

```
    }  
}  
  
/*  
 * Obtiene la lista de todas las citas  
 */  
public Cita[] listaCitas()  
    throws RecordStoreException, IOException {  
    if (online) {  
        sincroniza();  
    }  
    return mLocal.listaCitas();  
}  
  
/*  
 * Obtiene la lista de citas con alarmas pendientes  
 */  
public Cita[] listaAlarmasPendientes()  
    throws RecordStoreException, IOException {  
    if (online) {  
        sincroniza();  
    }  
    IndiceCita[] indices = mLocal.listaAlarmasPendientes();  
    return mLocal.listaCitas(indices);  
}  
  
/*  
 * Obtiene la configuracion local  
 */  
public InfoLocal getConfig()  
    throws RecordStoreException, IOException {  
    return mLocal.getInfoLocal();  
}  
  
/*  
 * Actualiza la configuracion local  
 */  
public void updateConfig(InfoLocal config)  
    throws RecordStoreException, IOException {  
    InfoLocal info = mLocal.getInfoLocal();  
    info.setOnline(config.isOnline());  
    this.online = config.isOnline();  
    mLocal.setInfoLocal(info);  
}  
  
/*  
 * Sincroniza la lista de citas con el servidor  
 */  
public void sincroniza()  
    throws RecordStoreException, IOException {  
  
    // Obtiene datos del cliente  
  
    InfoLocal info = mLocal.getInfoLocal();  
  
    IndiceCita[] indices = mLocal.listaCitasPendientes();  
    Cita[] citas = mLocal.listaCitas(indices);  
    SyncItem datosCliente = new SyncItem();  
    datosCliente.setCitas(citas);  
}
```



```
datosCliente.setTimeStamp(info.getTimeStamp());

// Envía y recibe

SyncItem datosServidor = mRemoto.sincroniza(datosCliente);

mLocal.marcaEnviados(indices);

// Agrega los datos recibidos del servidor

Cita[] citasServidor = datosServidor.getCitas();
if (citasServidor != null) {
    for (int i = 0; i < citasServidor.length; i++) {
        mLocal.addCita(citasServidor[i], false);
    }
}

info.setTimeStamp(datosServidor.getTimeStamp());
mLocal.setInfoLocal(info);
}

public void destroy() throws RecordStoreException {
    mLocal.destroy();
}
}
```

Podemos ver en este ejemplo como este modelo nos permite trabajar de forma *online* u *offline*. Según el modo de conexión, se utilizarán las funciones de uno u otro subsistema. En modo *online* siempre que se haga una operación se accederá al servidor para leer o almacenar las novedades que haya. En modo *offline* sólo se leerán o almacenarán los datos de forma local, excepto cuando solicitemos de forma explícita sincronizar los datos con el servidor.

En el caso del método `sincroniza`, que será el método encargado de coordinar información local con información remota, podemos ver que la transacción de sincronización se realiza en una única petición HTTP. Además, los mensajes que se hayan enviado al servidor no se marcan como enviados hasta después de haber completado el envío (se llama a `marcaEnviados` después de llamar a `sincroniza`).

13.3.2. Vista

Para la vista crearemos una clase para cada pantalla de nuestra aplicación, como hemos visto en temas anteriores. La clase correspondiente a una pantalla heredarà del tipo de `displayable` correspondiente y encapsularà la creación de la interfaz y la respuesta a los comandos.

Por ejemplo, para la edición de datos de las citas en nuestra aplicación podemos utilizar la siguiente pantalla:

```
public class EditaCitaUI extends Form implements
CommandListener {

    ControladorUI controlador;

    DateField iFecha;
```

```
TextField iAsunto;
TextField iLugar;
TextField iContacto;
ChoiceGroup iAlarma;
int itemAlarmaOn;
int itemAlarmaOff;

Command cmdAceptar;
Command cmdCancelar;
int eventoAceptar = ControladorUI.EVENTO_AGREGA_CITA;
int eventoCancelar = ControladorUI.EVENTO_MUESTRA_MENU;

Cita cita;

public EditaCitaUI(ControladorUI controlador) {
    super(controlador.getString(Recursos.STR_DATOS_TITULO));
    this.controlador = controlador;

    iFecha = new DateField(
        controlador.getString(Recursos.STR_DATOS_ITEM_FECHA),
        DateField.DATE_TIME);
    iAsunto = new TextField(
        controlador.getString(Recursos.STR_DATOS_ITEM_ASUNTO),
        "", MAX LENGHT, TextField.ANY);
    iLugar = new TextField(
        controlador.getString(Recursos.STR_DATOS_ITEM_LUGAR),
        "", MAX LENGHT, TextField.ANY);
    iContacto = new TextField(
        controlador.getString(Recursos.STR_DATOS_ITEM_CONTACTO),
        "", MAX LENGHT, TextField.ANY);
    iAlarma = new ChoiceGroup(
        controlador.getString(Recursos.STR_DATOS_ITEM_ALARMA),
        ChoiceGroup.EXCLUSIVE);
    itemAlarmaOn = iAlarma.append(
        controlador.getString(
            Recursos.STR_DATOS_ITEM_ALARMA_ON), null);
    itemAlarmaOff = iAlarma.append(
        controlador.getString(
            Recursos.STR_DATOS_ITEM_ALARMA_OFF), null);

    this.append(iFecha);
    this.append(iAsunto);
    this.append(iLugar);
    this.append(iContacto);
    this.append(iAlarma);

    cmdAceptar = new Command(
        controlador.getString(Recursos.STR_CMD_ACEPTAR),
        Command.OK, 1);
    cmdCancelar = new Command(
        controlador.getString(Recursos.STR_CMD_CANCELAR),
        Command.CANCEL, 1);
    this.addCommand(cmdAceptar);
    this.addCommand(cmdCancelar);

    this.setCommandListener(this);
}

private void setCita(Cita cita) {
    if (cita == null) {
        this.cita = new Cita();
    }
}
```

```

        iFecha.setDate(new Date());
        iAsunto.setString(null);
        iLugar.setString(null);
        iContacto.setString(null);
        iAlarma.setSelectedIndex(itemAlarmaOff, true);
    } else {
        this.cita = cita;

        iFecha.setDate(cita.getFecha());
        iAsunto.setString(cita.getAsunto());
        iLugar.setString(cita.getLugar());
        iContacto.setString(cita.getContacto());
        if (cita.isAlarma()) {
            iAlarma.setSelectedIndex(itemAlarmaOn, true);
        } else {
            iAlarma.setSelectedIndex(itemAlarmaOff, true);
        }
    }
}

private Cita getCita() {
    if(cita==null) {
        this.cita = new Cita();
    }
    if(iFecha.getDate()==null) {
        cita.setFecha(new Date());
    } else {
        cita.setFecha(iFecha.getDate());
    }
    cita.setAsunto(iAsunto.getString());
    cita.setLugar(iLugar.getString());
    cita.setContacto(iContacto.getString());
    cita.setAlarma(iAlarma.getSelectedIndex() ==
                    itemAlarmaOn);

    return cita;
}

public void reset(Cita cita, int eventoAceptar,
                  int eventoCancelar) {
    this.setCita(cita);
    this.eventoAceptar = eventoAceptar;
    this.eventoCancelar = eventoCancelar;
}

public void commandAction(Command cmd, Displayable disp) {
    if (cmd == cmdAceptar) {
        controlador.procesaEvento(eventoAceptar,
                                   this.getCita());
    } else if (cmd == cmdCancelar) {
        controlador.procesaEvento(eventoCancelar, null);
    }
}
}

```

Intentaremos llevar la mayor parte de la gestión de eventos al controlador, para así aislar lo más posible la vista del controlador. En la vista implementaremos la interfaz `commandAction`, pero el procesamiento de los eventos lo hará el controlador mediante el método `procesaEvento` que veremos a continuación.

13.3.3. Controlador

El controlador será el encargado de controlar el flujo de la aplicación. Según las acciones realizadas, el controlador mostrará distintas pantalla y realizará diferentes operaciones en el modelo.

Por ejemplo, el controlador para nuestra agenda será el siguiente:

```
public class ControladorUI {  
  
    /*  
     * Tipos de eventos  
     */  
    public final static int EVENTO_MUESTRA_MENU = 0;  
    public final static int EVENTO_MUESTRA_NUEVA_CITA = 1;  
    public final static int EVENTO_MUESTRA_LISTA_CITAS = 2;  
    public final static int EVENTO_MUESTRA_DATOS_CITA = 3;  
    public final static int  
        EVENTO_MUESTRA_LISTA_ALARMAS_PENDIENTES = 4;  
    public final static int  
        EVENTO_MUESTRA_DATOS_ALARMA_PENDIENTE = 5;  
    public final static int EVENTO_AGREGA_CITA = 6;  
    public final static int EVENTO_SALIR = 8;  
    public final static int EVENTO_SINCRONIZAR = 9;  
    public final static int EVENTO_MUESTRA_CONFIG = 10;  
    public final static int EVENTO_APLICA_CONFIG = 11;  
  
    /*  
     * Componentes de la UI  
     */  
    DatosCitaUI uiDatosCita;  
    EditaCitaUI uiEditaCita;  
    ListaCitasUI uiListaCitas;  
    EditaConfigUI uiEditaConfig;  
    MenuPrincipalUI uiMenuPrincipal;  
    BarraProgresoUI uiBarraProgreso;  
  
    /*  
     * Gestor de recursos  
     */  
    Recursos recursos;  
  
    /*  
     * Modelo  
     */  
    FachadaModelo modelo;  
  
    MIDletAgenda midlet;  
    Display display;  
  
    public ControladorUI(MIDletAgenda midlet) {  
        this.midlet = midlet;  
        display = Display.getDisplay(midlet);  
  
        init();  
    }  
  
    /*  
     * Inicializacion de los componentes de la UI  
     */  
}
```

```
public void init() {

    // Crea gestor de recursos
    recursos = new Recursos();

    // Crea UI
    uiDatosCita = new DatosCitaUI(this);
    uiEditaCita = new EditaCitaUI(this);
    uiListaCitas = new ListaCitasUI(this);
    uiEditaConfig = new EditaConfigUI(this);
    uiMenuPrincipal = new MenuPrincipalUI(this);
    uiBarraProgreso = new BarraProgresoUI(this);

    try {
        // Crea modelo
        modelo = new FachadaModelo(alarmas);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void destroy() throws RecordStoreException {
    modelo.destroy();
}

public void showMenu() {
    display.setCurrent(uiMenuPrincipal);
}

public String getString(int cod) {
    return recursos.getString(cod);
}

/*
 * Lanza el procesamiento de un evento
 */
public void procesaEvento(int evento, Object param) {
    HiloEventos he = new HiloEventos(evento, param);
    he.start();
}

/*
 * Hilo para el procesamiento de eventos
 */
class HiloEventos extends Thread {
    int evento;
    Object param;

    public HiloEventos(int evento, Object param) {
        this.evento = evento;
        this.param = param;
    }

    public void run() {
        Cita cita;
        Cita [] citas;
        InfoLocal info;

        try {
            switch(evento) {
                case EVENTO_MUESTRA_MENU:
```

```
        showMenu();
        break;

    case EVENTO_MUESTRA_NUEVA_CITA:
        uiEditaCita.reset(null,
            ControladorUI.EVENTO_AGREGA_CITA,
            ControladorUI.EVENTO_MUESTRA_MENU);
        display.setCurrent(uiEditaCita);
        break;

    case EVENTO_AGREGA_CITA:
        cita = (Cita)param;
        modelo.nuevaCita(cita);
        showMenu();
        break;

    case EVENTO_MUESTRA_LISTA_CITAS:
        uiBarraProgreso.reset(
            getString(Recursos.STR_PROGRESO_CARGA_LISTA),
            10, 0, true);
        display.setCurrent(uiBarraProgreso);
        citas = modelo.listaCitas();
        uiListaCitas.reset(citas,
            ControladorUI.EVENTO_MUESTRA_DATOS_CITA,
            ControladorUI.EVENTO_MUESTRA_MENU);
        display.setCurrent(uiListaCitas);
        break;

    case EVENTO_MUESTRA_DATOS_CITA:
        cita = (Cita)param;
        uiDatosCita.reset(cita,
            ControladorUI.EVENTO_MUESTRA_LISTA_CITAS);
        display.setCurrent(uiDatosCita);
        break;

    case EVENTO_MUESTRA_LISTA_ALARMAS_PENDIENTES:
        uiBarraProgreso.reset(
            getString(Recursos.STR_PROGRESO_CARGA_LISTA),
            10, 0, true);
        display.setCurrent(uiBarraProgreso);
        citas = modelo.listaAlarmasPendientes();
        uiListaCitas.reset(citas,
            ControladorUI.
                EVENTO_MUESTRA_DATOS_ALARMA_PENDIENTE,
            ControladorUI.EVENTO_MUESTRA_MENU);
        display.setCurrent(uiListaCitas);
        break;

    case EVENTO_MUESTRA_DATOS_ALARMA_PENDIENTE:
        cita = (Cita)param;
        uiDatosCita.reset(cita,
            ControladorUI.
                EVENTO_MUESTRA_LISTA_ALARMAS_PENDIENTES);
        display.setCurrent(uiDatosCita);
        break;

    case EVENTO_SINCRONIZAR:
        uiBarraProgreso.reset(
            getString(Recursos.STR_PROGRESO_SINCRONIZA),
            10, 0, true);
        display.setCurrent(uiBarraProgreso);
```

```
        modelo.sincroniza();
        display.setCurrent(uiMenuPrincipal);
        break;

    case EVENTO_MUESTRA_CONFIG:
        info = modelo.getConfig();
        uiEditaConfig.reset(info,
            ControladorUI.EVENTO_APLICA_CONFIG,
            ControladorUI.EVENTO_MUESTRA_MENU);
        display.setCurrent(uiEditaConfig);
        break;

    case EVENTO_APLICA_CONFIG:
        info = (InfoLocal)param;
        modelo.updateConfig(info);
        display.setCurrent(uiMenuPrincipal);
        break;

    case EVENTO_SALIR:
        midlet.salir();
        break;
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
```

Vemos en este ejemplo que el controlador tiene acceso al modelo a través de la fachada, y también tiene acceso a las diferentes pantallas de la interfaz de usuario (UI). A través de estos elementos controlará la presentación de la aplicación y la lógica de negocio de la misma.

Este controlador se encargará de dar respuesta a los eventos que se produzcan en la aplicación. Vemos que para hacer esto se utiliza un método `procesaEvento` que a su vez utiliza un hilo para procesar el evento solicitado. Esto se hace así para no bloquear el hilo de la aplicación al realizar operaciones que normalmente serán de larga duración.

Tenemos varios tipos de eventos definidos como constantes, y para cada uno de ellos especificaremos las operaciones a realizar y los componentes de la UI que se deben mostrar. Se puede observar en el ejemplo que en las operaciones que son de más larga duración, el controlador mientras la realiza muestra en pantalla una barra de progreso.

También se puede observar que todas las cadenas de texto se encuentran centralizadas en la clase `Recursos`. Todos los componentes de la aplicación obtendrán el texto de esta clase. Esto nos facilitará tareas como la traducción de nuestra aplicación a otros idiomas.

Para la internacionalización de la aplicación, podríamos hacer que la clase `Recursos` cargase todas las cadenas de texto de un fichero externo. De esta forma, cambiando este fichero podremos cambiar el idioma de nuestra

aplicación. Podríamos permitir que el usuario se descargase ficheros con otros idiomas.

13.4. Modo sin conexión

Hemos visto que en las aplicaciones MIDP es conveniente permitir trabajar sin conexión, trabajando con datos locales. También hemos visto como con el patrón MVC podemos dividir el modelo en dos subsistemas para permitir los dos modos de funcionamiento: con y sin conexión.

Vamos ahora a estudiar más a fondo el funcionamiento de este tipo de aplicaciones.

13.4.1. Tipos de aplicaciones

Podemos distinguir varios tipos de aplicaciones:

- *Thin*: Este tipo de aplicaciones son por ejemplo los navegadores, en los que todo el procesamiento se realiza en el servidor. En el cliente tenemos una aplicación genérica (navegador) que se encarga de obtener documentos del servidor y mostrarlos en la pantalla.
- *Thick*: Aplicaciones dedicadas. Son aplicaciones para tareas concretas que llevan al cliente la lógica de presentación. Permiten realizar una presentación sofisticada de datos. Por ejemplo, podremos realizar una ordenación de los datos en el cliente.
- *Standalone*: Llevan todo el procesamiento al cliente. Estas aplicaciones son por ejemplo el bloc de notas, calculadora, juegos, etc. Pueden conectarse de forma ocasional al servidor para actualizar datos, normalmente a petición del usuario.

Nos centraremos en el estudio de las aplicaciones de tipo thick. Estas aplicaciones deberán trabajar de forma coordinada con el servidor, pero podremos permitir trabajar sin conexión. El permitir trabajar en este modo nos trae la ventaja de que minimiza el tráfico en la red, sin embargo aumentará el coste de procesamiento, memoria y almacenamiento en el dispositivo local. Deberemos por lo tanto buscar un compromiso entre estos dos factores.

13.4.2. Replicación de datos

Para poder trabajar sin conexión es imprescindible replicar en nuestro dispositivo datos del servidor. Por lo tanto tendremos dos tipos de datos en el cliente:

- Datos replicados: Necesitaremos sincronizarlos con el servidor.
- Datos propios: Son sólo del cliente.

Además, dentro de los datos replicados del servidor podemos distinguir:

- Datos de solo lectura: Caché de datos del servidor. Sólo tendremos que preocuparnos de renovar estos datos cuando hayan caducado.

- Datos de lectura/escritura: En este caso necesitaremos sincronizar los datos locales con el servidor. Si se modifica la copia local, deberemos aplicar esta modificación a la copia remota posteriormente.

El modelo de réplica de datos que utilicemos se caracterizará por los siguientes factores:

- ¿Se replican todos los datos o sólo una parte de ellos?
- ¿Las estructuras de datos se replican fielmente o no?
- ¿Los datos son de lectura/escritura o de sólo lectura?
- ¿Los mismos datos pueden ser compartidos y replicados por muchos usuarios?
- ¿Los datos tienen fecha de caducidad?

Según estos factores deberemos decidir el modelo de sincronización adecuado para nuestra aplicación.

Para facilitar la sincronización es conveniente reducir la granularidad de los datos en el almacenamiento, es decir, reducir la cantidad de datos que se almacenen juntos en el mismo registro.

13.4.3. Sincronización de los datos

Los datos de nuestra aplicación se pueden transferir de diferentes formas:

- El cliente descarga periódicamente datos del servidor. En este caso simplemente deberemos mantener una caché de datos. Podremos renovar esta caché cuando haya pasado la fecha de caducidad de los datos, cada cierto período fijo, o cuando el usuario lo solicite.
- El cliente envía datos propios (no compartidos) al servidor. Deberemos actualizar en el servidor los datos que hayamos modificado en el cliente.
- El cliente envía al servidor datos compartidos con varios usuarios. Este es el caso más complicado, ya que varios clientes podrían estar modificando los mismos datos al mismo tiempo en sus copias locales.

Descarga de datos del servidor

Este caso se puede resolver de forma sencilla. Las actualizaciones de los datos se pueden hacer de varias formas:

- Si conocemos cuando caducarán los datos, podemos ponerles una fecha de caducidad, y una vez pasada ésta forzaremos a que se descargue nuevos datos de la red. Por ejemplo, si tenemos una aplicación para consultar la cartelera de los cines, como sabemos que las películas se estrenan los viernes, podemos hacer que los datos caduquen este día. De esta forma cada viernes cuando utilicemos la aplicación volverá a descargar nuevas películas.
- Si desconocemos esta fecha de caducidad, podríamos hacer que el móvil los actualice cada cierto período fijo (*polling*), o bien dejar que el usuario decida cuando quiere actualizar estos datos.

En muchas aplicaciones necesitaremos identificar qué datos no descargados todavía hay en el servidor. Por ejemplo, imaginemos nuestra aplicación de agenda. Podemos crear nuevas citas utilizando un navegador web en nuestro PC o desde otro dispositivo. Cuando ejecutemos la aplicación en nuestro móvil, para actualizar la agenda deberá descargar las nuevas citas que se hayan creado en el servidor.

Pero, ¿cómo podemos saber qué citas son nuevas? Podemos utilizar estampas de tiempo (timestamps) para etiquetar cada cita, de forma que cada nueva cita que añadamos al servidor tenga una estampa de tiempo superior a la de la anterior cita.

```
public void sincroniza()
    throws RecordStoreException, IOException {

    // Obtiene datos del cliente

    InfoLocal info = mLocal.getInfoLocal();
    IndiceCita[] indices = mLocal.listaCitasPendientes();
    Cita[] citas = mLocal.listaCitas(indices);
    SyncItem datosCliente = new SyncItem();
    datosCliente.setCitas(citas);
    datosCliente.setTimeStamp(info.getTimeStamp());

    // Envía y recibe

    SyncItem datosServidor = mRemoto.sincroniza(datosCliente);

    mLocal.marcaEnviados(indices);

    // Agrega los datos recibidos del servidor

    Cita[] citasServidor = datosServidor.getCitas();
    if (citasServidor != null) {
        for (int i = 0; i < citasServidor.length; i++) {
            mLocal.addCita(citasServidor[i], false);
        }
    }

    info.setTimeStamp(datosServidor.getTimeStamp());
    mLocal.setInfoLocal(info);
}
```

En nuestro dispositivo nos guardaremos el número de la última estampa de tiempo obtenida. Cuando solicitemos los nuevos datos al servidor, le proporcionaremos esta estampa de tiempo para que nos devuelva sólo aquellas citas que sean posteriores. Junto a estas citas nos devolverá una nueva estampa de tiempo correspondiente al último dato que hayamos recibido. El cliente tendrá la responsabilidad de almacenar esta estampa de tiempo para poder utilizarla en la próxima petición.

Envío de datos no compartidos al servidor

En este caso deberemos actualizar cada cierto período en el servidor los datos que hayamos modificado en nuestro dispositivo. La actualización podemos

hacer que sea automática, cada cierto período de tiempo, o manual, a petición del usuario.

Para actualizar los datos deberemos conocer qué datos han cambiado desde la última actualización. Para ello podremos añadir a los datos almacenados en RMS un atributo que nos diga si hay cambios pendientes de actualizar en dicho dato.

Si hemos creado índices para nuestros registros de RMS, podremos añadir este atributo a los índices, para facilitar de esta forma la búsqueda de estos datos.

```
public class IndiceCita {  
    int rmsID;  
    int id;  
    Date fecha;  
    boolean alarma;  
    boolean pendiente;  
}
```

Podemos ver aquí la razón por la que preferimos una granularidad fina de los datos almacenados. Si tuviésemos almacenados muchos datos en un mismo registro, y modificamos una pequeña parte de estos datos, tendríamos que marcar todo el registro como modificado, por lo que habrá que subir al servidor un mayor volumen de datos. Si estos datos hubiesen estado repartidos en varios registros, sólo hubiese hecho falta actualizar la parte que hubiese cambiado.

Envío de datos compartidos al servidor

Este caso es el más complejo. Existe una copia maestra de los datos almacenada en el servidor, y varias copias locales, conocidas como copias legales, que se descargan los usuarios a sus dispositivos.

Si varios clientes han descargado los datos en sus móviles, y modifican su copia legal, cuando actualicen los datos en la copia maestra del servidor podrían sobrescribir el trabajo de otros usuarios.

Para poder corregir estos conflictos de la mejor forma posible, deberemos tener una granularidad muy fina de los datos, de forma que los usuarios sólo modifiquen las porciones de los datos que hayan modificado. De esta forma, si nosotros hemos modificado una parte de los datos que otro usuario no ha tocado, cuando el otro usuario actualice los datos no sobrescribirá dicha parte.

Para decidir qué versión de los datos mantener en la copia maestra podemos tomar diferentes criterios:

- Sobrescribir directamente el último que llegue al servidor
- Mantener el que tenga fecha de modificación posterior
- Mantener el que tenga fecha de modificación anterior
- etc

Deberemos intentar solucionar los conflictos sin necesidad de solicitar la intervención del usuario, aunque en ciertas ocasiones la mejor solución puede ser preguntar qué copia de los datos desea que se mantenga.

A. Ejercicios

A.1. Introducción y entorno de desarrollo

1.1. Abrir la herramienta `ktoolbar` de WTK 2.1. Abrir desde ella alguna de las aplicaciones de ejemplo que tenemos disponibles (`UIDemo`, `demos` o `games`).

a) Probar a compilar estas aplicaciones y ejecutarlas en distintos emuladores (creando el paquete en caso necesario). Observar las diferencias que encontramos de unos emuladores a otros.

b) Consultar los ficheros `JAD` y `MANIFEST.MF` de las aplicaciones probadas. Identificar los elementos encontrados en estos ficheros.

c) Cargar el monitor de red y el monitor de memoria para monitorizar la utilización de recursos que realizan las aplicaciones que estamos probando.

1.2. Cargar alguna de las aplicaciones de ejemplo, pero esta vez vía OTA. Seguir los pasos necesarios para descargar, instalar y ejecutar la aplicación en el emulador.

1.3. Crear un nuevo proyecto con WTK 2.1. El proyecto se llamará `PruebaAplicacion` y el MIDlet principal será `es.ua.j2ee.prueba.MIDletPrueba`.

A.2. Java para MIDs. MIDlets

2.1. Vamos a hacer nuestra primera aplicación *"Hola mundo"* en J2ME. Para ello debemos:

a) Abrir el proyecto `PruebaAplicacion` creado en la sesión anterior con WTK 2.1. Si no lo tuviésemos creado, crear el proyecto, cuyo MIDlet principal será `es.ua.j2ee.prueba.MIDletPrueba`.

b) Una vez creado abriremos el proyecto desde Eclipse, crearemos la clase del MIDlet principal, e introduciremos en ella el siguiente código:

```
package es.ua.j2ee.prueba;

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

public class MIDletPrueba extends MIDlet {

    protected void startApp() throws
MIDletStateException {
        Form f = new Form("Hola mundo");
        Display d = Display.getDisplay(this);
        d.setCurrent(f);
    }

    protected void pauseApp() {
```

```

    }

    protected void destroyApp(boolean unconditional)
        throws MIDletStateChangeException {
    }
}

```

c) Guardar el fichero y desde WTK compilar y ejecutar la aplicación en emuladores, comprobando que funciona correctamente.

d) Modificar el ejemplo para hacerlo parametrizable. Ahora en lugar de mostrar siempre el mensaje *"Hola mundo"*, tomaremos el mensaje a mostrar del parámetro `msg.bienvenida`. Crear este parámetro dentro del fichero JAD, y leerlo dentro del MIDlet para mostrarlo como título del formulario.

2.2. Vamos a añadir recursos a nuestra aplicación. Mostraremos una imagen en la pantalla, introduciendo el siguiente código en el método `startApp` de nuestro MIDlet:

```

protected void startApp() throws MIDletStateChangeException {
    Form f = new Form("Hola mundo");
    try {
        f.append(Image.createImage("/logo.png"));
    } catch(Exception e) {}

    Display d = Display.getDisplay(this);
    d.setCurrent(f);
}

```

Para poder mostrar esta imagen deberemos añadirla como recurso a la *suite*. Añadir una imagen con nombre `logo.png` al directorio de recursos. Puedes encontrar esta imagen en el directorio `PruebaAplicacion/res` de las plantillas de la sesión.

Compilar y ejecutar la aplicación para comprobar que la imagen se muestra correctamente. Utilizar para ello tanto `ktoolbar` como la herramienta *ant* desde Eclipse.

2.3. Ahora añadiremos sonido a la aplicación. Para ello deberemos utilizar la API multimedia que es una API adicional. Deberemos:

a) Incorporar la librería MMAPI a nuestro proyecto en Eclipse.

b) Una vez hecho esto podremos utilizar esta API multimedia en el editor de Eclipse sin que nos muestre errores en el código. Modificaremos el código del MIDlet de la siguiente forma:

```

package es.ua.j2ee.prueba;

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

public class MIDletPrueba extends MIDlet {

```

```
protected void startApp() throws
MIDletStateException {
    Form f = new Form("Hola mundo");
    try {
        f.append(Image.createImage("/logo.png"));
    } catch(Exception e) {}

    try {
        Manager.playTone(80, 1000, 100);
    } catch(MediaException e) {}

    Display d = Display.getDisplay(this);
    d.setCurrent(f);
}

protected void pauseApp() {
}

protected void destroyApp(boolean incondicional)
throws MIDletStateException {
}
}
```

c) Guardar y compilar desde WTK. Comprobar que la aplicación funciona correctamente.

2.4. Vamos a incorporar un temporizador a una aplicación. Lo único que haremos será mostrar un mensaje de texto en la consola cuando se dispare el temporizador, por lo que no será una aplicación útil para visualizar en el móvil.

a) En el directorio `Temporizador` de las plantillas de la sesión se encuentra implementado este temporizador. Compilarlo y ejecutarlo con WTK.

b) Modificar este temporizador para que en lugar de dispararse pasado cierto intervalo, se dispare a una hora fija.

2.5. Vamos a ver ahora como leer datos codificados de forma binaria. Tendremos entre los recursos de la aplicación un fichero de datos donde éstos se encuentran almacenados de forma binaria con un formato dado. Vamos a ver cómo leer este formato.

a) En el directorio `Serializacion` de las plantillas de la sesión tenemos una aplicación que lee un fichero binario y muestra los datos leídos en la consola. El fichero del que lee (`curso.dat`) almacena la información sobre un curso codificada de la siguiente forma:

```
<UTF>Nombre
<UTF>Departamento
<short>Número de horas
```

En la aplicación esta información se deserializa en el método `deserialize` de la clase `Curso`. Este método nos devolverá un objeto `Curso` con los datos leídos del fichero.

Compilar y probar la aplicación con WTK. Nos dará una salida como la siguiente:

```
Nombre: Programacion de Dispositivos Moviles
Departamento: DCCIA
Horas: 24
```

b) Vamos a hacer lo mismo, pero en este caso para leer datos de un alumno. En el fichero `alumno.dat` tenemos estos datos almacenados de la siguiente forma:

```
<UTF>DNI
<UTF>Nombre
<UTF>Apellidos
<char>Sexo
<short>Edad
<UTF>Teléfono
<boolean>Casado
```

Implementar un método `deserialize` para la clase `Alumno`, que deserialice los datos almacenados en este formato y obtenga un objeto `Alumno` que los contenga.

Deserializar este fichero desde nuestro `MIDlet` e imprimir el objeto `Alumno` resultante. En este caso deberemos obtener una salida como la siguiente:

```
DNI: 48123456-A
Nombre: Pedro
Apellidos: Lopez
Sexo: V
Edad: 24
Telefono: 965123456
Soltero
```

A.3. Interfaz gráfica de alto nivel

2.1. En el directorio `MenuBasico` tenemos implementada una aplicación básica en la que se muestra un menú típico de un juego mediante un `displayable` de tipo `List`.

- a) Consultar el código y probar la aplicación.
- b) Añadir una nueva opción al menú, de nombre "*Hi-score*".
- c) Probar cambiando a los distintos tipos de lista existentes.
- d) Añadir comandos a esta pantalla. Se pueden añadir los comandos "*OK*" y "*Salir*".

2.2. Vamos a implementar una alarma utilizando alertas y temporizadores. En las plantillas de los ejercicios se proporciona una base para realizar esta aplicación, contenida en el directorio `Alarma`. Tenemos un formulario donde

podemos establecer la fecha de la alarma y fijarla o anularla. Lo que deberemos hacer es:

- a) Crear una tarea (`TimerTask`) que al ser ejecutada muestre una alerta de tipo alarma y reproduzca un sonido de aviso (utilizando la clase `AlertType`). Después de mostrarse esta alerta deberá volver a la pantalla actual.
- b) Planificar la ejecución de esta alarma utilizando un temporizador (`Timer`). Esto lo haremos en el método `commandAction` como respuesta al comando de fijar alarma. También deberemos cancelar el temporizador en caso de que la alarma se anule.

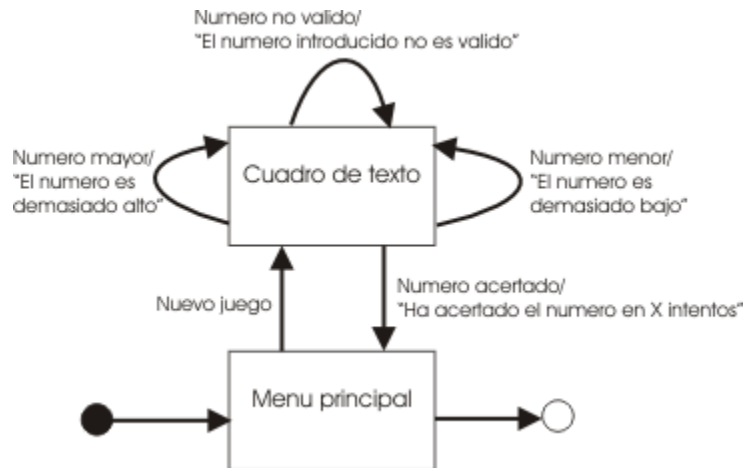
A.4. Proyecto: Aplicación básica

4.1. Vamos a implementar un juego consistente en adivinar un número del 1 a 100. Como para previo, vamos a crear el menú principal de nuestro juego, en el que deberemos tener las opciones *Nuevo juego* y *Salir*.

- a) ¿Qué tipo de *displayable* utilizaremos para realizar este menú?
- b) Implementar esta pantalla encapsulando todo su contenido en una misma clase.
- c) Añadir un comando que nos permita seleccionar la opción marcada del menú.
- d) Incorporar un *listener* de comandos para dar respuesta a este comando de selección de opciones. Por ahora lo que haremos será mostrar una alerta que diga *"Opcion no implementada todavia"*.

4.2. Implementar el juego de adivinar un número del 1 al 100. Para ello partiremos de la base realizada en el ejercicio anterior.

El juego pensará un número aleatorio de 1 a 100, y se mostrará al usuario un cuadro de texto donde deberá introducir el número del que piensa que se trata. Una vez introducido, pulsará OK y entonces la aplicación le dirá si el número de demasiado alto, si es demasiado bajo o si ha acertado. En caso de que el número sea muy alto o muy bajo, volveremos al mismo cuadro de texto para volver a probar. Si ha acertado el juego finalizará, mostrando el número de intentos que ha necesitado y volviendo al menú principal.



Para implementar esta aplicación crearemos una nueva pantalla encapsulada en una clase de nombre `EntradaTexto` que será de tipo `TextBox`, donde el usuario introducirá el número. Al construir esta pantalla se deberá determinar un número aleatorio de 1 a 100, cosa que podemos hacer de la siguiente forma:

```
Random rand = new Random();
this.numero = Math.abs(rand.nextInt()) % 100 + 1;
```

Deberemos añadir un comando para que el usuario notifique que ha introducido el número. Como respuesta a este comando deberemos obtener el número que ha introducido el usuario y compararlo con el número aleatorio. Según si el número es menor, mayor o igual mostraremos una alerta con el mensaje correspondiente, y volveremos a la pantalla actual o al menú principal según si el usuario ha fallado o acertado respectivamente.

4.3. Como parte optativa, se propone añadir un *ranking* con las mejores puntuaciones obtenidas. Para ello, deberemos añadir una opción adicional al menú principal para consultar dicho *ranking*. Además, cuando consigamos adivinar el número en un número record de intentos, la aplicación nos pedirá nuestro nombre en un cuadro de texto y nos añadirá al *ranking* de mejores puntuaciones, junto con el número de intentos que hemos necesitado.

A.5. Interfaz gráfica de bajo nivel

5.1. Vamos a probar una aplicación básica que dibuje contenido en la pantalla utilizando la API de bajo nivel. En el directorio `DibujoBasico` de las plantillas de la sesión tenemos implementada esta aplicación.

- Consultar el código y posteriormente compilar y probar la aplicación.
- Modificar el anchor de posicionamiento del texto para probar los distintos alineamientos existentes.
- Probar a cambiar las propiedades de los objetos dibujados (fuente, color, etc).

d) Añadir animación. Para ello crearemos un hilo que modifique las variables `x` e `y` a lo largo del tiempo y llame a `repaint` para dibujar en la nueva posición. Utilizar el evento `showNotify` para poner en marcha este hilo.

5.2. Vamos a implementar una aplicación similar al juego que se conocía como "TeleSketch". Esta aplicación nos deberá permitir dibujar en la pantalla utilizando las teclas de los cursores.

La idea es dibujar en *offscreen* (en una imagen mutable), de forma que no se pierda el contenido dibujado. En cada momento conoceremos la posición actual del cursor, donde dibujaremos un punto (puede ser un círculo o rectángulo de tamaño reducido). Al pulsar las teclas de los cursores del móvil moveremos este cursor por la pantalla haciendo que deje rastro, y de esta manera se irá generando el dibujo.

Tenemos una plantilla en el directorio `TeleSketch`. Sobre esta plantilla deberemos realizar lo siguiente:

a) En el constructor de la clase deberemos crear una imagen mutable donde dibujar, con el tamaño del `Canvas`.

b) En el método `actualiza` deberemos dibujar un punto en la posición actual del cursor y llamar a `repaint` para repintar el contenido de la pantalla.

c) En `paint` deberemos volcar el contenido de la imagen *offscreen* a la pantalla.

d) Deberemos definir los eventos `keyPressed` y `keyRepeated` para mover el cursor cada vez que se pulsen las teclas arriba, abajo, izquierda y derecha. Podemos utilizar las acciones de juegos (game actions) para conocer cuáles son estas teclas.

5.3. Ampliar la aplicación anterior permitiendo cambiar el color y el grosor del lápiz. Para hacer esto podemos añadir una serie de comandos con un conjunto de colores y grosores preestablecidos.

5.4. Si dibujamos usando los eventos de repetición el cursor muchas veces se moverá muy lentamente. En lugar de utilizar este evento para dibujar podemos crear un hilo que ejecute un bucle infinito. Dentro de este hilo mientras la tecla siga pulsada se irá moviendo el cursor, de esta manera se actualizará con la frecuencia que nosotros queramos sin tener que esperar al evento de repetición. Sabremos que una tecla se mantiene pulsada desde que se invoca un evento `keyPressed` hasta que se invoca un evento `keyReleased` para la misma tecla.

5.5. Utilizar la API de Nokia para dibujar en un *canvas* a pantalla completa.

5.6. En el directorio `Prueba3D` de las plantillas de la sesión podemos encontrar un ejemplo de aplicación M3G. En ella podemos ver un ejemplo de modo inmediato y otro de modo *retained*. Consultar el código y probar el ejemplo. Para poderlo probar necesitaremos al menos WTK 2.2.

A.6. Juegos

6.1. Vamos a ver un ejemplo de juego básico implementado en MIDP 2.0. Este juego se encuentra ya implementado en el directorio `Juego` de las plantillas de la sesión.

En el juego podemos encontrar los siguientes elementos:

- Tenemos un *sprite* de un coche que se mueve por la pantalla. Se utiliza como *sprite* la imagen `coche.png`.
- En el ciclo del juego se lee la entrada del usuario, y según ésta movemos el *sprite* por la pantalla (en las direcciones izquierda, derecha, arriba y abajo), y actualizamos los gráficos.
- Tenemos un fondo construido mediante un objeto `TiledLayer`. En la imagen `fondo.png` tenemos una serie de elementos con los que construir el fondo de la carretera. Utilizamos el fichero `datos` donde tenemos codificado un posible escenario. De este fichero leemos el índice del elemento que se debe mostrar en cada celda del fondo. Se encuentra codificado de la siguiente forma:

```
<ancho:int> <alto:int>
<celda_1_1:byte> <celda_1_2:byte> ... <celda_1_ancho>
<celda_2_1:byte> <celda_2_2:byte> ... <celda_2_ancho>
...
<celda_alto_1:byte> <celda_alto_2:byte> ... <celda_alto_ancho>
```

Leemos estos datos utilizando un `DataInputStream`. Dibujamos este fondo como capa en la pantalla, y en cada iteración lo vamos desplazando hacia abajo para causar el efecto de que el coche avanza por la carretera.

- Detección de colisiones con nuestro coche y el fondo. De esta forma podremos chocar con los bordes de la carretera, teniendo que evitar salirnos de estos márgenes. Cuando chocamos simplemente se muestra una alerta que nos avisa de que hemos chocado y se vuelva a iniciar el juego.

6.2. Ejemplo de juego completo. En las plantillas de la sesión se incluye un juego completo como ejemplo en el directorio `Panj`. Consultar el código fuente de este juego y probarlo.

A.7. Multimedia

7.1. Vamos a crear un reproductor de video que reproduzca un video con formato 3GPP. Lo reproduciremos dentro de un *canvas*. Para ello añadiremos un comando *Start* al *canvas* que nos permita comenzar la reproducción del video.

Podemos encontrar este reproductor implementado en el directorio `Video` de

las plantillas de la sesión. Para probarlo necesitaremos un emulador de Nokia Series 60, ya que otros no reconocen el formato 3GPP.

7.2. Añadir un *listener* para conocer cuando comienza a reproducirse y cuando se detiene el video. Este *listener* deberá modificar los comandos disponibles en el *canvas*. Cuando comience a reproducirse eliminará el comando *Start* y añadirá un comando *Stop* que nos servirá para detenerlo. Cuando el video se detenga, eliminaremos el comando *Stop* y volveremos a añadir el comando *Start*.

7.3. Permitir realizar captura de imágenes (*snapshots*) del video. Para ello cuando comience a reproducirse añadiremos un comando *Snap* con el que podremos capturar un fotograma del video. Una vez capturado, detendremos el video y mostraremos la imagen en el *canvas*.

A.8. Proyecto: Juego

8.1. Vamos a implementar un clon del clásico *Frogger*. En el directorio *Cochedrillo* de las plantillas de la sesión se encuentra una base sobre la que se puede implementar el juego. En esta plantilla tenemos:

- Recursos necesarios: imágenes y fichero con los datos de las fases.
- Clase con los datos del juego (*CommonData*). En ella tenemos toda la información acerca del tamaño de los *sprites*, las secuencias de *frames* de las animaciones, sus coordenadas iniciales, etc.
- Clase del MIDlet principal (*MIDletCDrillo*).
- Pantalla *splash* (*SplashScreen*).
- Clase para la gestión de los recursos (*Resources*).
- Clase que implementa un ciclo de juego genérico (*GameEngine*).
- Pantalla de título (*TitleScene*) e interfaz para las escenas del juego (*Scene*).
- Pantalla de juego (*GameScene*) vacía. Deberemos implementar la lógica y la presentación del juego en esta clase.

Debemos:

a) Crear las estructuras de datos necesarias para cargar la información sobre las fases. Los datos de cada fase son los siguientes:

- Título de la fase
- Carriles de la carretera. Para cada carril tendremos los siguientes datos:
 - Velocidad de los coches que circulan por él.
 - Separación que hay entre los coches del carril.
 - Tipo de coche que circula por el carril.

Estos datos se encuentran codificados en el fichero de datos de fases (*stages.dat*) de la siguiente forma:

```
<int> Numero de fases
Para cada fase
  <UTF> Titulo
```

```
<byte> Número de carriles  
Para cada carril  
  <byte> Velocidad  
  <short> Separación  
  <byte> Tipo de coche
```

Deberemos añadir a las clases que encapsulen estos datos métodos para deserializarlos de este fichero.

Introducir código en `Resources` para cargar esta información y añadir los datos de niveles como recurso global de nuestro juego.

b) Añadir el *sprite* de nuestro personaje. Hacer que el *sprite* se mueva por la pantalla como respuesta a las pulsaciones del cursor.

c) Añadir un fondo a la escena. El fondo se construirá utilizando un objeto `TiledLayer`. Utilizaremos la información sobre el número de carriles de la fase actual para generar este fondo.

d) Añadir los coches a la escena. Los coches deben aparecer a la izquierda y avanzar hacia la derecha. En cada carril los coches se generarán de distinta forma. Cada carril tiene los siguientes atributos:

- Velocidad: Los coches que circulen en dicho carril irán a la velocidad indicada. La velocidad se mide en píxeles que avanzan los coches en cada *tick*.
- Separación: El espacio en píxeles que queda entre un coche y el siguiente que aparece. Podemos implementar esto creando un contador para cada carril, que controle en qué momento debe hacer aparecer un nuevo coche en dicho carril. Por ejemplo, si tenemos un carril cuya velocidad es 3 y la separación 80, inicializaremos el contador a 80 y en cada tick lo decrementaremos en 3 unidades. Cuando el contador llegue a 0 haremos aparecer un coche y le sumaremos 80 para volver a empezar a contar lo que queda para aparecer el siguiente coche.
- Tipo de coche: En cada carril circulará un tipo de coche distinto. Tenemos definidos 3 tipos, cada uno de ellos con distinto aspecto y tamaño. Según el tipo (0, 1 ó 2) crearemos el *sprite* del coche utilizando una imagen distinta.

e) Implementar el resto de funcionalidades del juego:

- Comprobar colisiones de los coches con nuestro personaje. Cuando esto suceda perderemos una vida y volverá a comenzar el nivel. En el caso de que no queden vidas, terminará el juego.
- Comprobar si el personaje ha cruzado la carretera para pasar de fase. Cuando hayamos llegado al otro lado de la carretera pasaremos a la siguiente fase. Si fuese la última, podríamos hacer que vuelva al comienzo del juego (primera fase).

A.9. RMS

9.1. Vamos a implementar un almacén de notas. En el directorio `Notas` tenemos la base de esta aplicación. Cada nota será un objeto de la clase `Mensaje`, que tendrá un asunto y un texto. Además incorpora métodos de serialización y deserialización.

Vamos a almacenar estos mensajes de forma persistente utilizando RMS. Para ello vamos a utilizar el patrón de diseño adaptador, completando el código de la clase `AdaptadorRMS` de la siguiente forma:

a) En el constructor se debe abrir el almacén de registros de nombre `RS_DATOS` (creándolo si es necesario). En el método `cerrar` cerraremos el almacén de registros abierto.

b) En `listaMensajes` utilizaremos un objeto `RecordEnumeration` para obtener todos los mensajes almacenados. Podemos utilizar la deserialización definida en el objeto `Mensaje` para leerlos. Conforme leamos los mensajes los añadiremos a un vector.

El índice de cada registro en el almacén nos servirá para luego poder modificar o eliminar dicho mensaje. Por esta razón deberemos guardarnos este valor en alguna parte. Podemos utilizar para ello el campo `rmsID` de cada objeto `Mensaje` creado.

En `getMessage` deberemos introducir el código para obtener un mensaje dado su identificador de RMS.

c) En `addMensaje` deberemos introducir el código necesario para insertar un mensaje en el almacén. Con esto ya podremos probar la aplicación, añadiendo mensajes y comprobando que se han añadido correctamente. Probar a cerrar el emulador y volverlo a abrir para comprobar que los datos se han guardado de forma persistente.

d) En `removeMensaje` deberemos eliminar un mensaje del almacén dado su identificador.

e) En `updateMensaje` modificaremos un mensaje del almacén sobrescribiéndolo con el nuevo mensaje proporcionado. Para ello obtendremos el identificador correspondiente a al mensaje antiguo de su campo `rmsID`.

9.2. Obtener la información de la cantidad de memoria ocupada y disponible a partir del objeto `RecordStore`. Deberemos hacer que los métodos `getLibre` y `getOcupado` de la clase `AdaptadorRMS` devuelvan esta información.

A.10. Red

10.1. Vamos a ver como ejemplo una aplicación de chat para el móvil. En el directorio `ejemplos` de las plantillas de la sesión se encuentra una aplicación

web con todos los servlets que necesitaremos para probar los ejemplos. Podremos desplegar esta aplicación en Tomcat para hacer pruebas con nuestro propio servidor.

Podemos encontrar la aplicación de chat implementada en el directorio `Chat`, que realiza las siguientes tareas:

- Lo primero que se mostrará será una pantalla de *login*, donde el usuario deberá introducir el *login* con el que participar en el chat. Debemos enviar este *login* al servidor para iniciar la sesión. Para ello abriremos una conexión con la URL del *servlet* proporcionando los siguientes parámetros:

```
?accion=login&id=<nick_del_usuario>
```

Si el *login* es correcto, el servidor nos devolverá un código de respuesta `200 OK`. Además deberemos leer la cabecera `URL-Reescrita`, donde nos habrá enviado la URL rescrita que deberemos utilizar de ahora en adelante para mantener la sesión.

- Una vez hemos entrado en el chat, utilizaremos la técnica de *polling* para obtener los mensajes escritos en el chat y mostrarlos en la pantalla. Utilizando la URL rescrita, conectaremos al *servlet* del chat proporcionando el siguiente parámetro:

```
?accion=lista
```

Esto nos devolverá como respuesta una serie de mensajes, codificados mediante un objeto `DataOutputStream` de la siguiente forma:

```
<nick1> <mensaje1>
<nick2> <mensaje2>
...
<nickN> <mensajeN>
```

De esta forma podremos utilizar un objeto `DataInputStream` para ir leyendo con el método `readUTF` las cadenas del *nick* y del texto de cada mensaje del chat:

```
String nick = dis.readUTF();
String texto = dis.readUTF();
```

- Para enviar mensajes al chat utilizaremos el bloque de contenido, conectándonos a la URL rescrita proporcionando el siguiente parámetro:

```
?accion=enviar
```

El mensaje se deberá codificar en binario, escribiendo la cadena del mensaje con el método `writeUTF` de un objeto `DataOutputStream`. Si obtenemos una respuesta `200 OK` el mensaje habrá sido enviado correctamente.

10.2. Vamos a acceder desde el móvil a nuestra tienda virtual. Mostraremos en el móvil una pantalla en la que aparecerá la lista de productos que hay disponibles en nuestra tienda. Al pulsar sobre cada uno de ellos nos mostrará información detallada sobre el producto.

Tenemos la aplicación base implementada en el directorio `Tienda`. Debemos añadir en el método `leeProductos` de la clase `ListaProductos` el código necesario para leer la lista de productos de la red.

Para ello conectaremos a la URL donde tenemos el `servlet` de nuestra tienda y leeremos la información de los productos que nos envía en la respuesta. La información que se envía consiste en una serie de objetos `Producto` serializados. Para leerlos podremos deserializar objetos `Producto` del flujo de entrada hasta que se produzca una `EOFException`, indicándonos que se ha llegado al final del flujo.

10.3. En el directorio `PruebaBT` podemos encontrar una aplicación sencilla de ejemplo que establece una conexión bluetooth. Podemos cargar varios emuladores que se conecten como esclavos y un emulador que haga el papel de maestro. Realizaremos lo siguiente:

- Cargar un emulador que se comporte como esclavo, publicando su servicio y esperando una conexión entrante.
- Cargar un emulador que se comporte como maestro, descubriendo los dispositivos y servicios de su entorno.
- Establecer una conexión con el dispositivo esclavo que habrá localizado.

10.4. En el directorio `PruebaSW` se encuentra una aplicación que utiliza un servicio web sencillo. El servicio web que se utiliza se puede encontrar en el directorio `HolaMundoSW`, implementado con `JWS DP`. Desplegar el servicio en `JWS DP` y probar la aplicación `PruebaSW` en un emulador. Comprobar que se conecta correctamente al servicio.

10.5. En el directorio `AlarmaPush` se encuentra el ejemplo de la alarma, esta vez activable mediante *push*. Probar la aplicación desplegándola en un emulador vía OTA. Comprobar que el `MIDlet` se inicia a la hora que se haya programado la alarma.

A.11. Aplicaciones corporativas

11.1. En el directorio `Agenda` de las plantillas de la sesión tenemos implementada una aplicación de ejemplo. Se trata de una aplicación de agenda distribuida, que nos permite registrar citas que constarán de fecha y hora, asunto, lugar, contacto y posibilidad de programar una alarma.

Las citas que añadamos se sincronizarán con el servidor de la aplicación, de forma que tendremos las mismas citas en cliente y servidor. El servidor de la aplicación se incluye en el directorio `ejemplos` de las plantillas de la sesión de red. Podremos añadir citas utilizando directamente una interfaz web incluida en la aplicación web, o bien añadirlas en el móvil y sincronizarlas con el servidor.

Podemos encontrar la interfaz web en la dirección:

```
http://www.j2ee.ua.es/ejemplos/citas
```

Si añadimos la cita en el servidor, cuando sincronicemos desde el móvil veremos dicha cita en nuestro dispositivo móvil. Si añadimos la cita en el cliente móvil, cuando sincronicemos desde éste la cita se añadirá al servidor.

Además podemos utilizar la aplicación en modo online, que conectará al servidor cada vez que solicitemos consultar o añadir datos. De esta forma no hará falta sincronizar manualmente para actualizar la información, aunque tendrá el inconveniente de que realizará un mayor número de transferencias a través de la red.

Probar la aplicación para estudiar su funcionamiento. Una vez hecho esto, consultar el código fuente de la aplicación, en él podemos ver:

- Distinguir los diferentes componentes del modelo de diseño MVC: modelo, vista y controlador.
- Forma de procesar los eventos en el controlador.
- Adaptador RMS para el almacenamiento de citas de la agenda.
- *Proxy* remoto para acceder al servidor y sincronizar datos con éste.
- Ver cómo se ha implementado el patrón de diseño fachada, y el funcionamiento de los modos *online* y *offline*.
- Estudiar el mecanismo de sincronización: estampas de tiempo para recibir mensajes del servidor y marcar las citas como pendientes de enviar para enviar las nuevas citas al servidor.

A.12. Proyecto: Aplicación corporativa

12.1. Vamos a implementar una aplicación de mensajería interna para un grupo de trabajo. La aplicación nos debe permitir enviar mensajes a todos los usuarios y recibir los mensajes que hayan sido enviados por otros usuarios.

Estos mensajes constarán de un asunto y un texto. La aplicación nos debe permitir:

- Enviar un mensaje: Introducimos los datos de un mensaje y lo enviamos a todo el grupo.
- Ver mensajes: Ver los mensajes que se han enviado al grupo.
- Sincronizar: Sincronizar los mensajes del móvil con los del servidor. Enviará los mensajes que hayamos escrito y que todavía no hayan sido enviados, y recibirá aquellos mensajes que estén en el servidor y todavía no hayamos recibido.
- Modo *online* y *offline*: Deberá permitir trabajar en los dos modos. En modo *online* la sincronización con el servidor es automática cada vez que hagamos alguna operación de consulta o inserción de mensajes.

Esta aplicación deberá seguir el patrón MVC. Podemos encontrar una base para desarrollar la aplicación en el directorio `Mensajes` de las plantillas de la

sesión. Encontramos ya implementado el controlador y la vista, y tenemos una plantilla para la fachada del modelo. Debemos implementar el resto de clases para el modelo.

Para implementar la aplicación podemos basarnos en la aplicación *Agenda* que vimos como ejemplo en la sesión anterior.

Podemos seguir los siguientes pasos:

- a) Podemos reutilizar el adaptador RMS que implementamos en la sesión de almacenamiento persistente para almacenar los mensajes en RMS.
- b) Implementar el subsistema local del modelo. Esto nos debe permitir añadir y consultar los mensajes que hayamos creado.
- c) Implementar un *proxy* remoto que nos permita acceder a la función de sincronización del servidor. Debemos acceder a la siguiente URL:

```
http://www.j2ee.ua.es/ejemplos/servlet/ServletMensajes
```

En la petición deberemos enviar codificados mediante un `DataOutputStream` los siguientes datos:

```
<long> Timestamp cliente  
<int> Número de mensajes  
Para cada mensaje  
    Serializar mensaje
```

En la respuesta se recibirán los datos con un `DataInputStream`, llegando éstos con el siguiente formato:

```
<long> Timestamp servidor  
<int> Numero de mensajes  
Para cada mensaje  
    Deserializar mensajes
```

Podemos utilizar la interfaz web existente en la siguiente dirección para crear y consultar mensajes desde el servidor:

```
http://www.j2ee.ua.es/ejemplos/mensajes
```

De esta forma podremos hacer pruebas creando mensajes en el servidor y sincronizando desde el cliente para recibirlo, o creándolo en el cliente y comprobando si el mensaje se refleja en el servidor tras sincronizar.

- d) Incorporar la posibilidad de utilizar modo *online* de conexión, en el que las conexiones al servidor sean automáticas cada vez que consultamos o añadimos mensajes.

