Introducción a los MIDs. Java para MIDs. MIDlets.

Índice

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.

A continuación estudiaremos las características de este tipo de dispositivos y analizaremos de forma general las distintas APIs ofrecidas por la plataforma J2ME para la programación de dispositivos móviles.

1.1. 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.

1.1.1. 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 es 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.

1.1.2. Dispositivos móviles de información (MIDs)

Los distintos MIDs presentes actualmente en el mercado cuentan con un rango de memoria muy amplio y los tipos de interfaz con los que vamos a tener que trabajar son muy distintos. 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. (La web francesa "Java Club" cuenta con una base de datos con información sobre las características hardware y versiones de APIs de distintos modelos de teléfonos móviles, http://www.club-java.com/TastePhone/J2ME/MIDP_mobile.jsp).

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.

1.1.3. 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 communication*s). 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 esta 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 cuál 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.

1.1.4. 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.

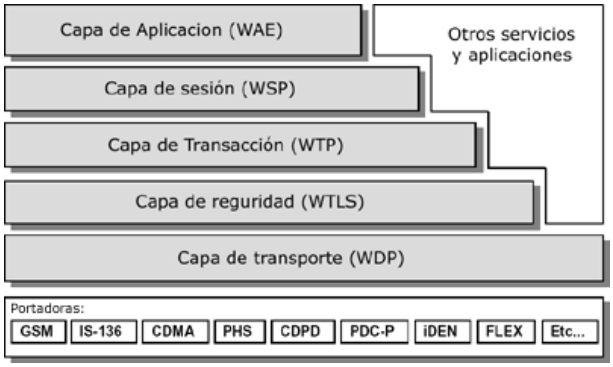
Debemos distinguir claramente las tecnologías portadoras de las tecnologías para desarrollar aplicaciones. Podemos desarrollar aplicaciones para dispositivos móviles utilizando diferentes tecnologías, y para acceder a una aplicación podremos utilizar diferentes portadores.

1.1.5. 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.

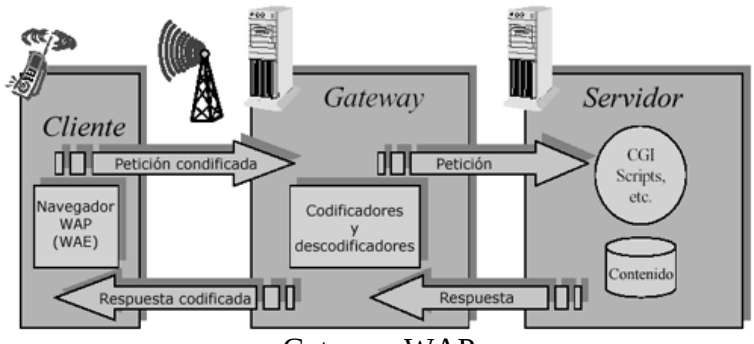
Han surgido diferentes tecnologías diseñadas para ofrecer contenidos aptos para este tipo de dispositivos. Entre ellas destacamos las siguientes:

- **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).



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.



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 tarificando 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 estar 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 nuestro 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.

1.1.6. 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.

1.2. Aplicaciones J2ME

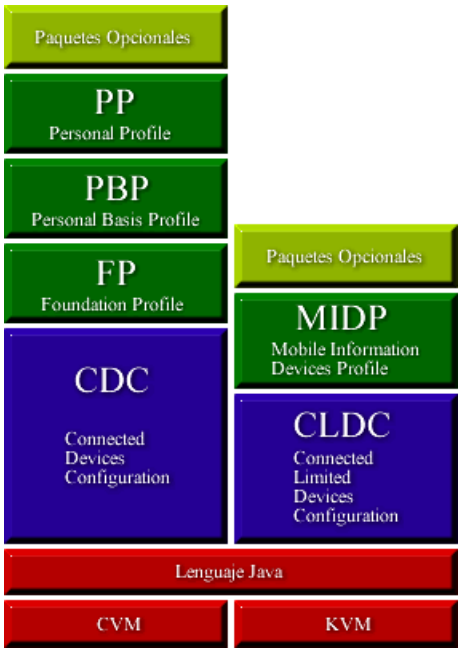
La plataforma J2ME nos ofrece una serie de APIs con las que desarrollar las aplicaciones en lenguaje Java. Una vez tengamos la aplicación podremos descargarla en cualquier dispositivo con soporte para J2ME y ejecutarla en él.

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.

1.2.1. 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 encuentras arquitecturadas en dos capas: configuraciones y perfiles.



Arquitectura de Java ME

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.

1.2.2. 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, descodificadores 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.

1.2.3. 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 pertenecienta 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 autentificación.
- **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.

1.2.4. 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.

1.2.5. MSA

MSA (*Mobile Service Architecture*) es el paso siguiente a JTWI. Incluye esta última especificación y añade nuevas tecnologías para dar soporte a las características de los nuevos dispositivos.

Esta especificación puede ser implementada en los dispositivos bajo dos diferentes modalidades: de forma parcial y de forma completa. Esto es debido al amplio abanico de dispositivos existentes, con características muy dispares, lo cual hace difícil que todos ellos puedan implementar todas las APIs definidas en MSA. Por ello se permite que algunos dispositivos implementen MSA de forma parcial. A continuación se indican las APIs incluidas en cada una de estas dos modalidades:

Implementación parcial	Implementación completa
CLDC 1.1 MIDP 2.1 PDA Optional Packages for J2ME Mobile Media API 1.2 Bluetooth API 1.1 Mobile 3D Graphics 1.1 Wireless Messaging API 2.0 Scalable 2D Vector Graphics 1.1	APIs de la implementación parcial J2ME Web Services 1.0 SIP API 1.0.1 Content Handler API 1.0 Payment API 1.1.0 Advanced Multimedia Supplements 1.0 Mobile Internationalization API 1.0 Security and Trust Services API 1.0 Location API 1.0.1

1.3. Aplicaciones J2ME

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.

Estudiaremos 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.

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.

1.3.1. 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.

1.3.2. 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 si, 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.

1.3.3. 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 *suites*:

- Obtendrá información de las *suites* 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.

1.3.4. 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 *bytes*.

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 *bytes* 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
```

1.3.5. 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:

`MIDlet-Name`	Nombre de la *suite*.
`MIDlet-Version`	Versión de la *suite*.
`MIDlet-Vendor`	Autor (Proveedor) de la *suite*.
`MicroEdition-Profile`	Perfil requerido para ejecutar la *suite*. Podrá tomar el valor `MIDP-1.0` ó `MIDP-2.0`, según las versión de MIDP que utilicen las aplicaciones incluidas.
`MicroEdition-Configuration`	Configuración requerida para ejecutar la *suite*. Tomará el valor `CLDC-1.0` 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.jtech.serpiente.SerpMIDlet
MIDlet-2: TeleSketch, /icons/ts.png,
es.ua.jtech.ts.TeleSketchMIDlet
MIDlet-3: Panj, /icons/panj.png, es.ua.jtech.panj.PanjMIDlet
```

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

`MIDlet-Icon`	Icono para la *suite*.

`MIDlet-Description`	Descripción de la *suite*.
`MIDlet-Info-URL`	Dirección URL con información
`MIDlet-Data-Size`	Número mínimo de *bytes* 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.

1.4. 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.

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.

1.4.1. 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:

1. Clases de núcleo de Java (*bootstrap*)
2. Extensiones instaladas
3. *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. Deberemos 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.

1.4.2. 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 ofuscadores, como ProGuard, RetroGuard o JODE. Deberemos obtener alguno de estos ofuscadores por separado, ya que no se incluyen en J2SE ni en los kits de desarrollo para MIDP que veremos más adelante.

1.4.3. 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*.

1.4.4. 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.

1.4.5. 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.

1.4.6. 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.

1.4.7. 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.

1.5. 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 despleguemos 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).
- **WTK 2.5**: Aparte de todo lo soportado por WTK 2.2, incorpora todas las APIs definidas en MSA. Nos centraremos en el estudio de esta versión por ser la que

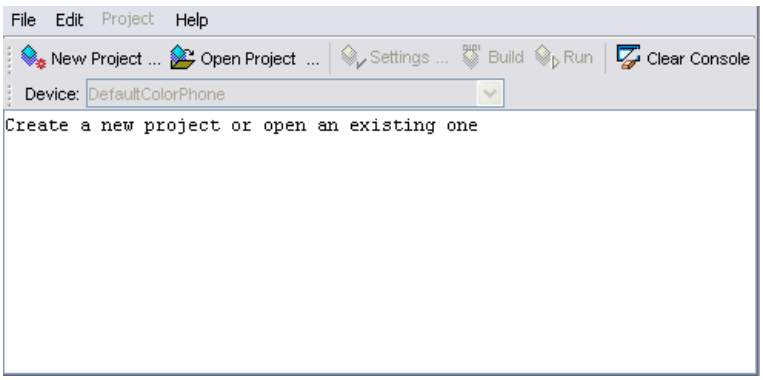
incorpora un mayor número de APIs en el momento de la escritura de este texto, además de ser genérica (se puede utilizar para cualquier versión de MIDP).

1.5.1. 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. Esta herramienta principal de WTK en versiones anteriores a la 2.5 recibía el nombre de `ktoolbar`:



Wireless Toolkit

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.

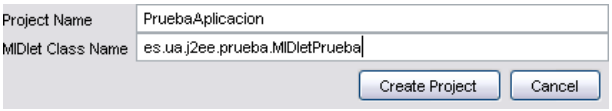
Por último, en `lib` deberemos poner las librerías adicionales que queramos incorporar a nuestra aplicación. Pondremos en este directorio el fichero JAR con la librería de clases que queramos añadir. Lo que se hará será añadir todas las clases contenidas en estas librerías, así como las contenidas en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

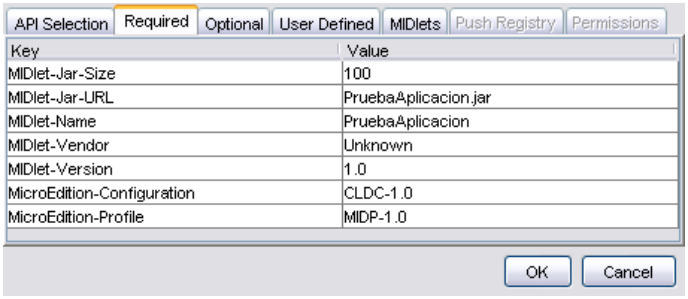


Crear una nueva aplicación

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 creemos. 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

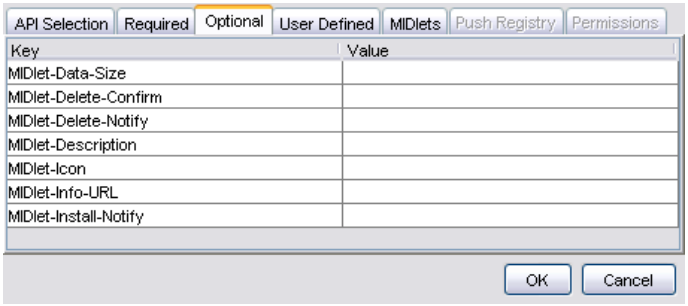
queremos personalizarlo más. La primera ficha nos muestra los datos obligatorios:



Configuración de la aplicación

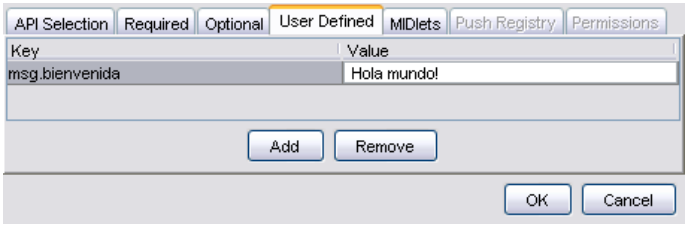
Como nombre de la *suite* y del JAR habrá tomado por defecto el nombre del proyecto que hayamos especificado. Será conveniente modificar los datos del fabricante y de la versión, para adaptarlos a nuestra aplicación. No debemos preocuparnos por especificar el tamaño del JAR, ya que este dato será actualizado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



Datos opcionales de configuración

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:

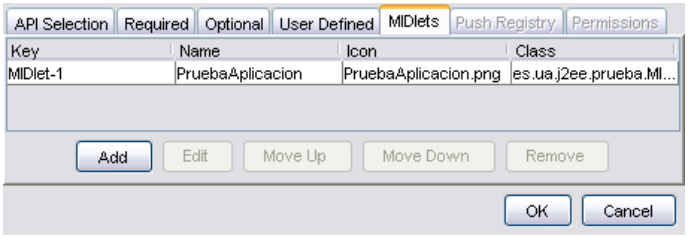


Propiedades personalizadas

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:

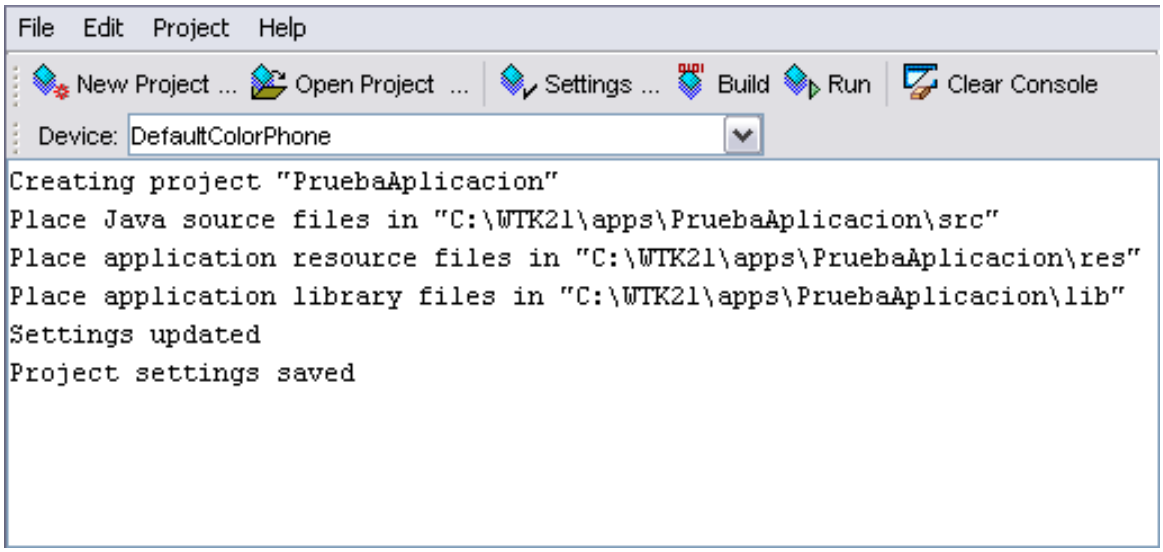


Datos de los MIDlets

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.

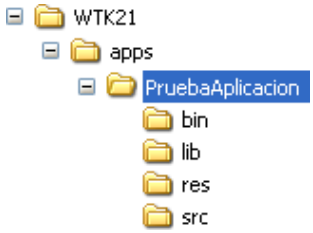
Dado que una *suite* puede contener más de un MIDlet, desde esta pestaña podremos añadir tantos MIDlets como queramos, especificando para cada uno de ellos su nombre, icono (de forma opcional) y clase.

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



Aplicación creada

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:

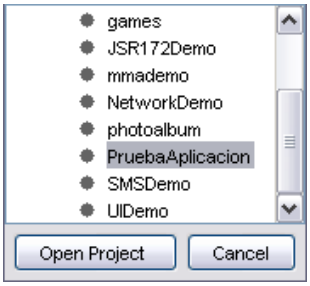


Estructura de directorios

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:



Abrir proyecto

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.

1.5.2. 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 ofuscadores 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.

1.5.3. 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.

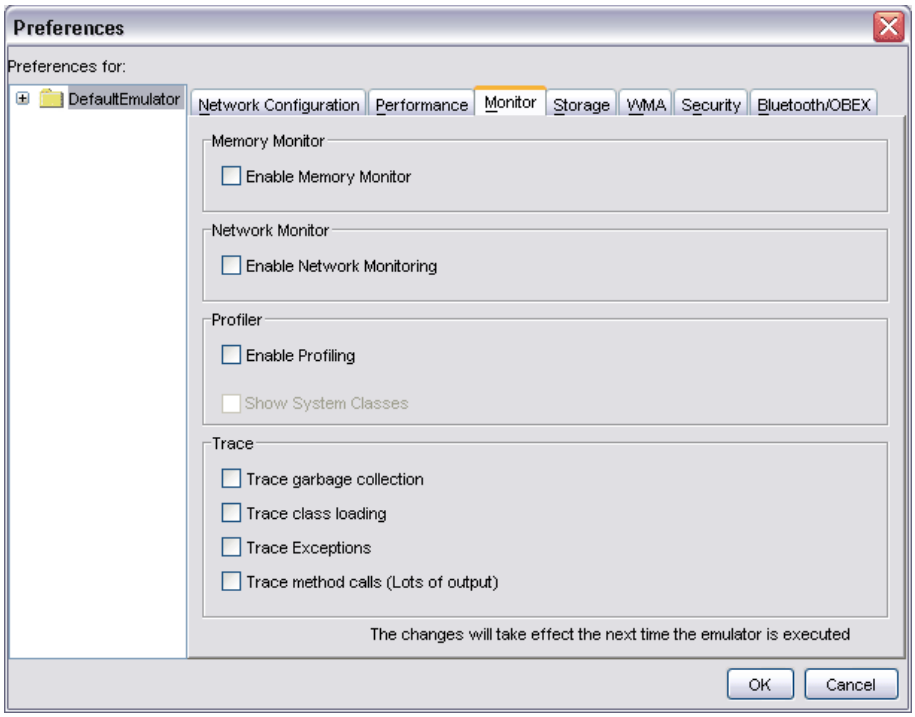
Para integrar los emuladores de teléfonos Nokia en WTK simplemente tendremos que instalar estos emuladores en el directorio `${WTK_HOME}/wtklib/devices`. Una vez instalados en este directorio, estos emuladores estarán disponibles dentro del kit de desarrollo, de forma que podremos seleccionarlos en el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

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:



Monitorización

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 cueles estemos desarrollando, sobretodo en cuanto a consumo de memoria, para asegurarnos de que la aplicación seguirá funcionando cuando la llevamos al dispositivo real.

1.5.4. 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.

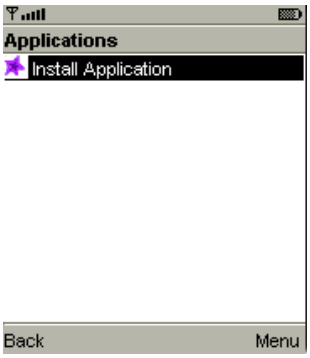
Para desplegar una aplicación de forma que sea accesible vía OTA, simplemente deberemos:

- 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.



Ejecución via OTA (I)

Ejecución via OTA (II)

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.

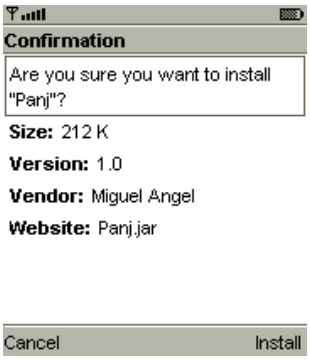
Este procedimiento puede ser demasiado costoso si queremos probar la aplicación repetidas veces utilizando este procedimiento, ya que nos obligaría, para cada nueva prueba que quisiésemos hacer, a volver a desplegar la aplicación en el servidor web.

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:

Ejecución via OTA (III)

Ejecución via OTA (IV)

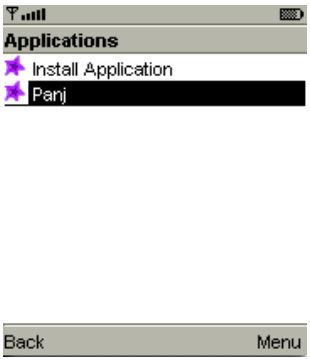
Una vez introducida la dirección del documento web donde tenemos publicada nuestra aplicación, nos mostrará la lista de enlaces a ficheros JAD que tengamos en esa página. Podremos seleccionar uno de estos enlaces para instalar la aplicación. En ese momento descargará el fichero JAD y nos mostrará la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



Ejecución via OTA (V)

Ejecución via OTA (VI)

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:



Ejecución via OTA (VII)

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.

1.6. 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.

1.6.1. 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.

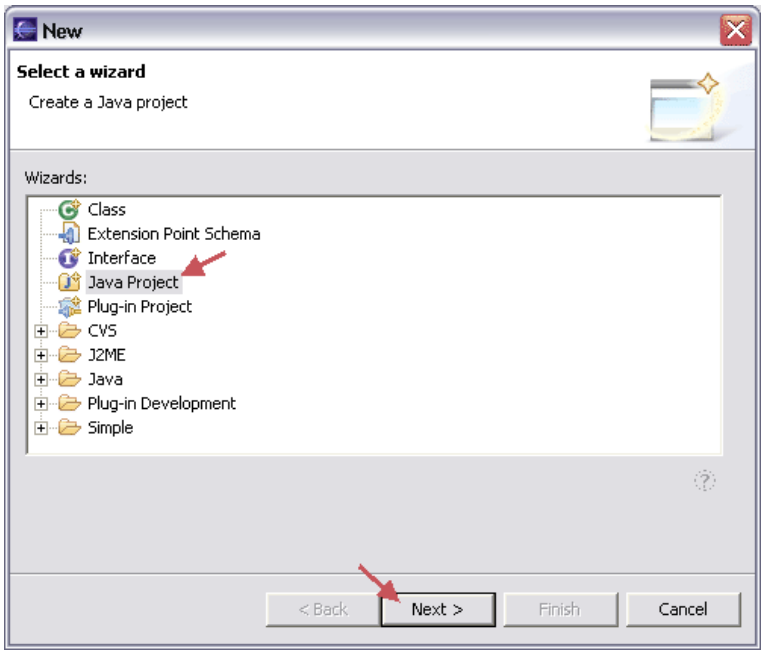
A continuación veremos como crear aplicaciones J2ME paso a paso siguiendo cada uno de estos tres métodos.

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.

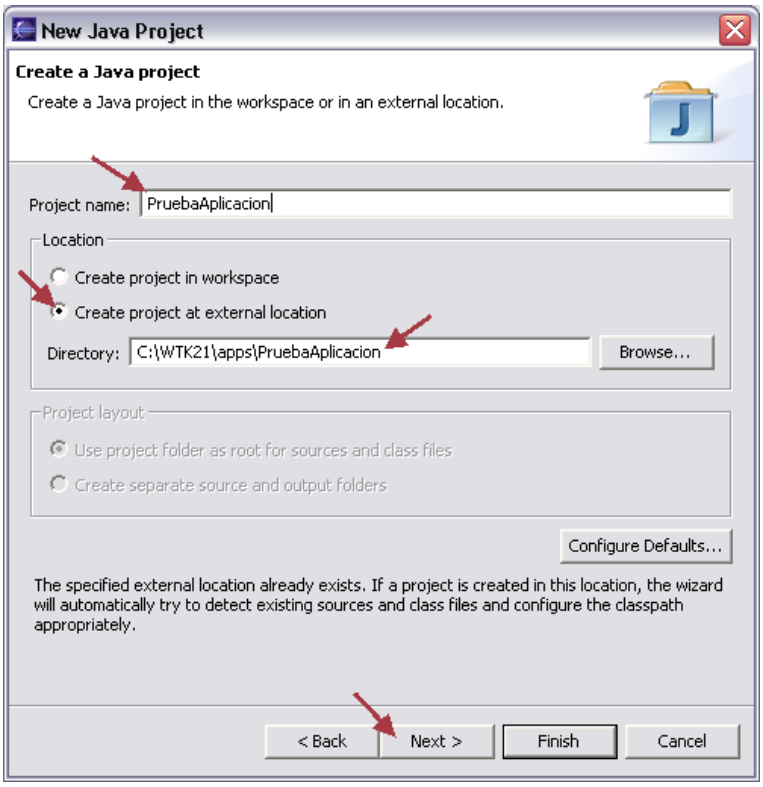
Lo primero que tenemos que hacer es crear una nueva aplicación utilizando WTK, como hemos visto en el punto anterior, de forma que nos cree la estructura de directorios necesaria en nuestro directorio de proyectos.

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



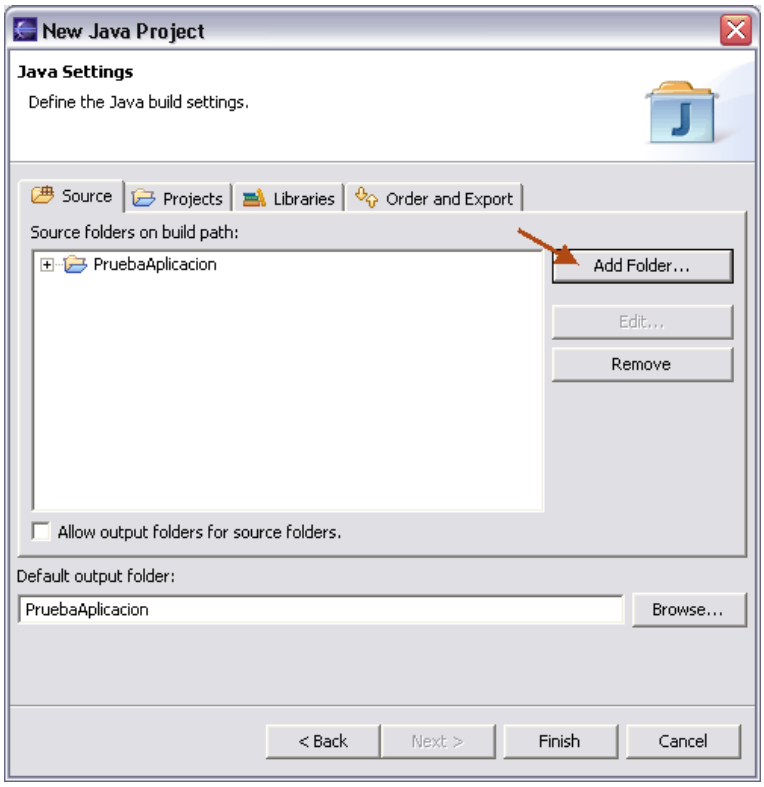
Eclipse (I)

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



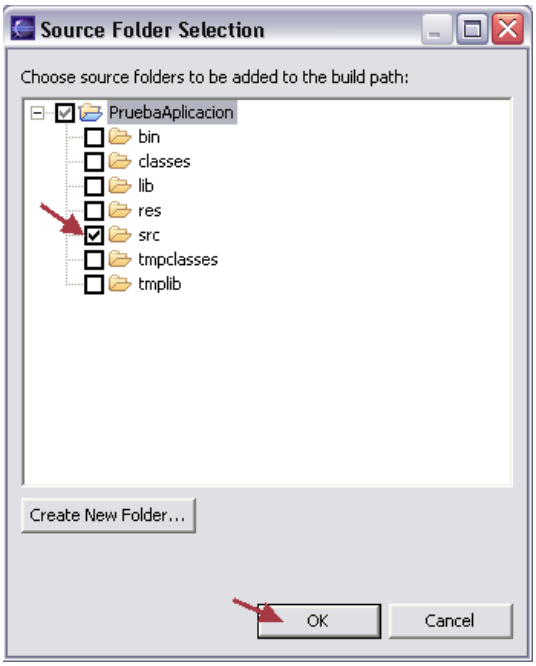
Eclipse (II)

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.



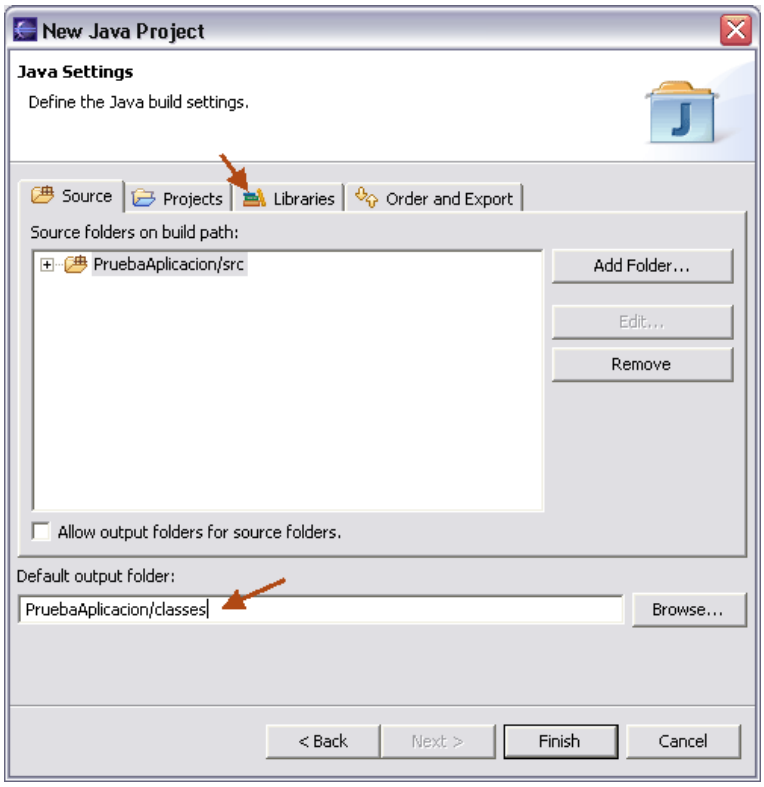
Eclipse (III)

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.



Eclipse (IV)

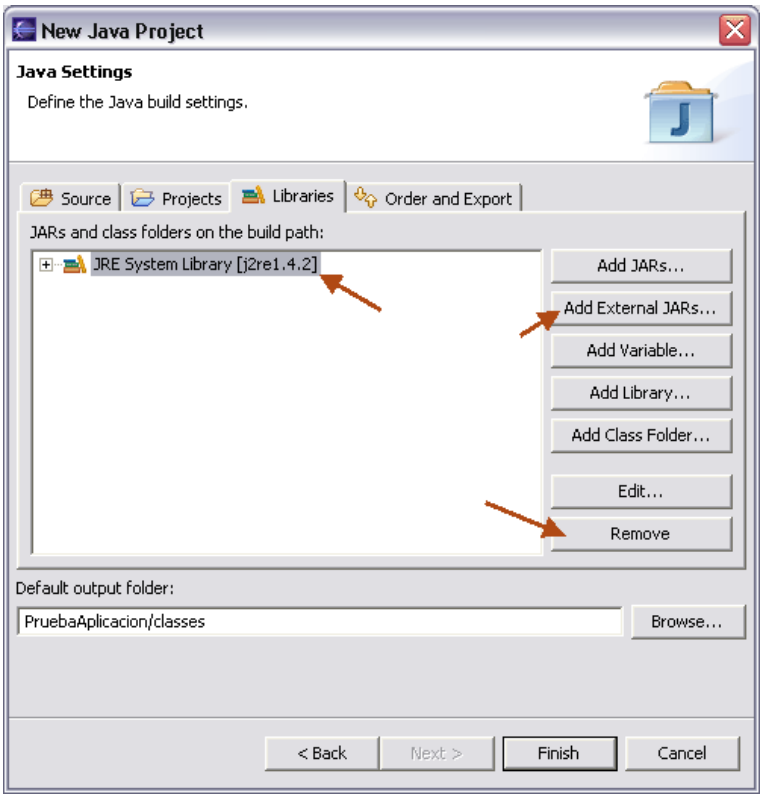
Como directorio de fuentes seleccionaremos el subdirectorio `src` del directorio de nuestro proyecto, si no estuviese seleccionado ya.



Eclipse (V)

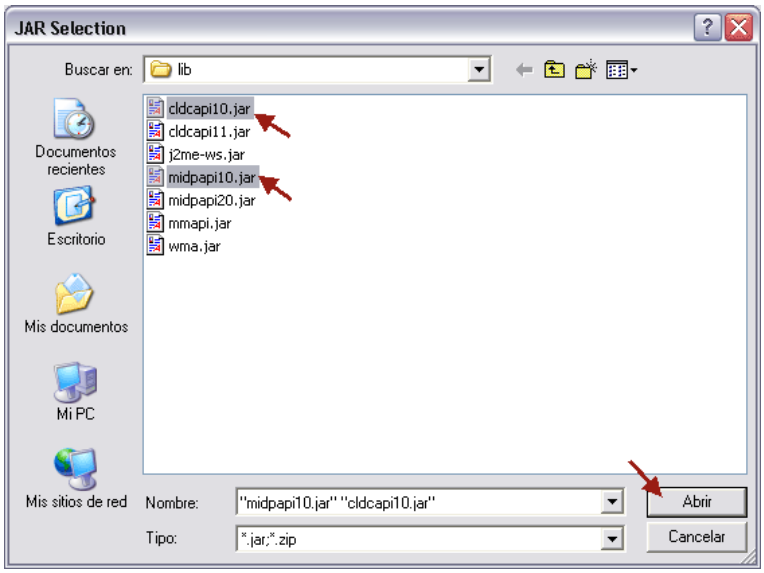
Como directorio de salida (**Default output folder**) podemos especificar el directorio `classes`.

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



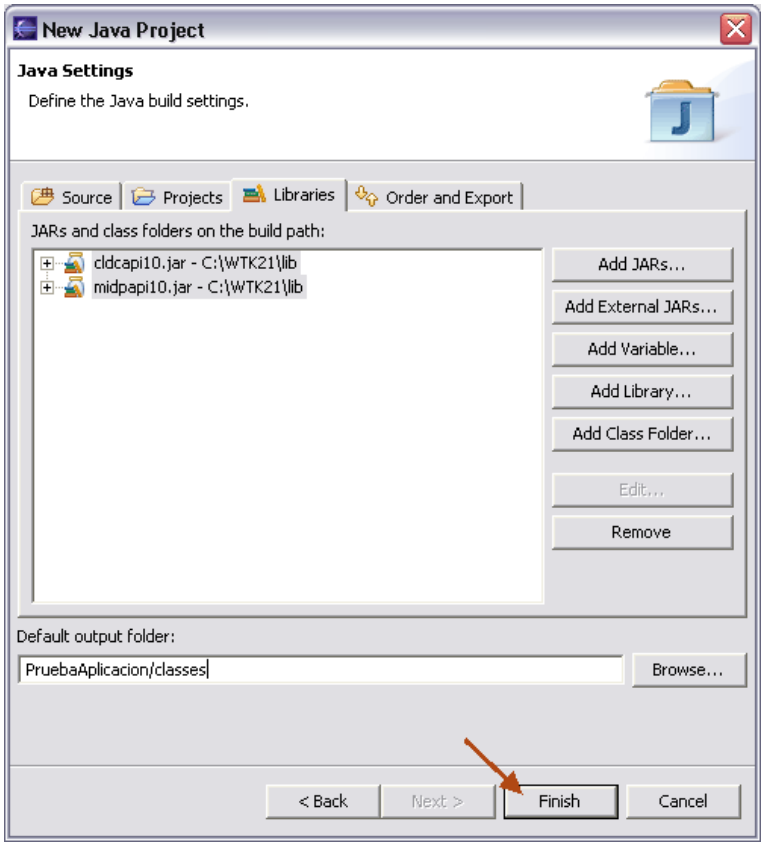
Eclipse (VI)

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 MMAPI o APIs propietarias, seguiremos el mismo proceso para añadir sus correspondientes ficheros JAR a la lista.



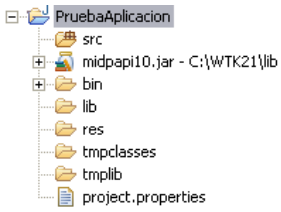
Eclipse (VII)

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.



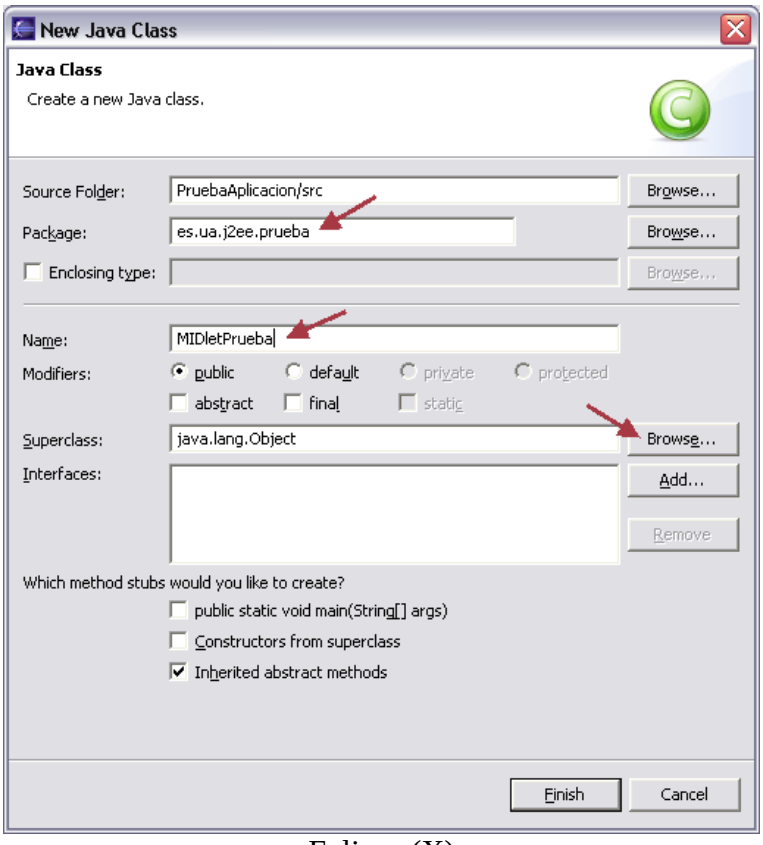
Eclipse (VIII)

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:



Eclipse (IX)

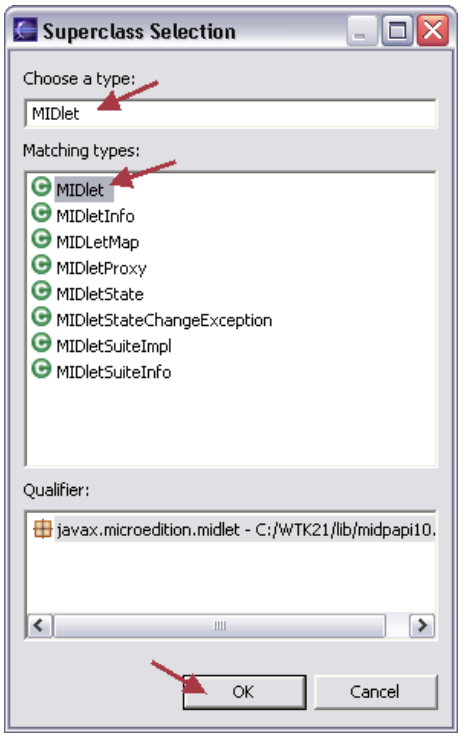
Ahora podemos empezar a crear las clases de nuestra aplicación. Para ello pulsaremos sobre **New** y elegiremos crear una nueva clase Java, con lo que se abrirá la siguiente ventana de creación de la clase:



Eclipse (X)

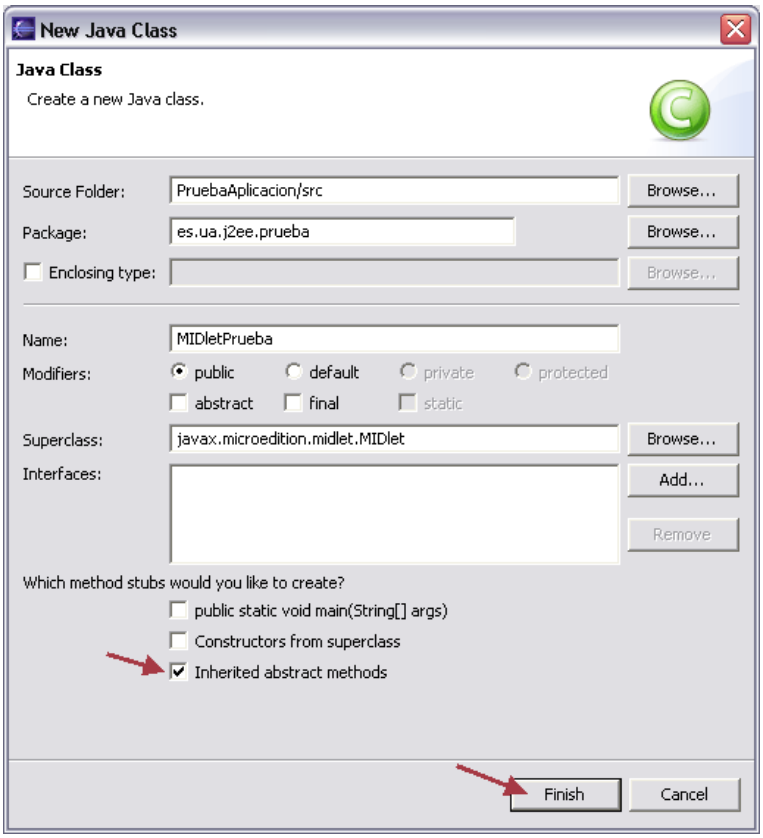
Aquí deberemos introducir el paquete al que pertenecerá la clase, y el nombre de la misma. También debemos indicar la superclase y las interfaces que implementa la clase

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:



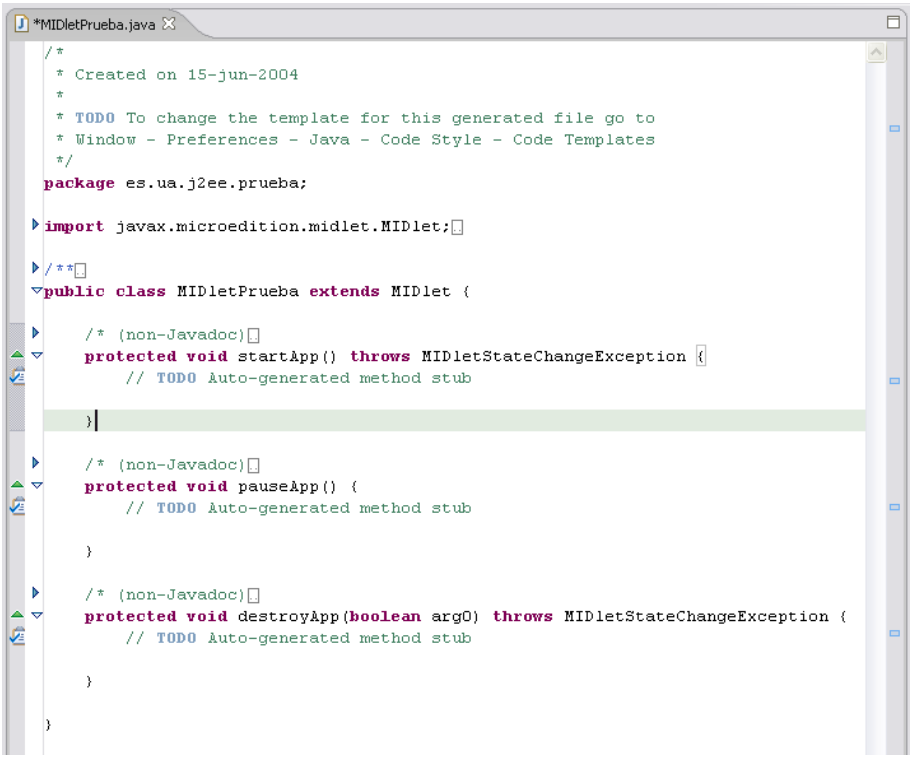
Eclipse (XI)

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**.



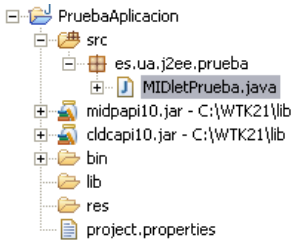
Eclipse (XII)

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:



Eclipse (XIII)

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:



Eclipse (XIV)

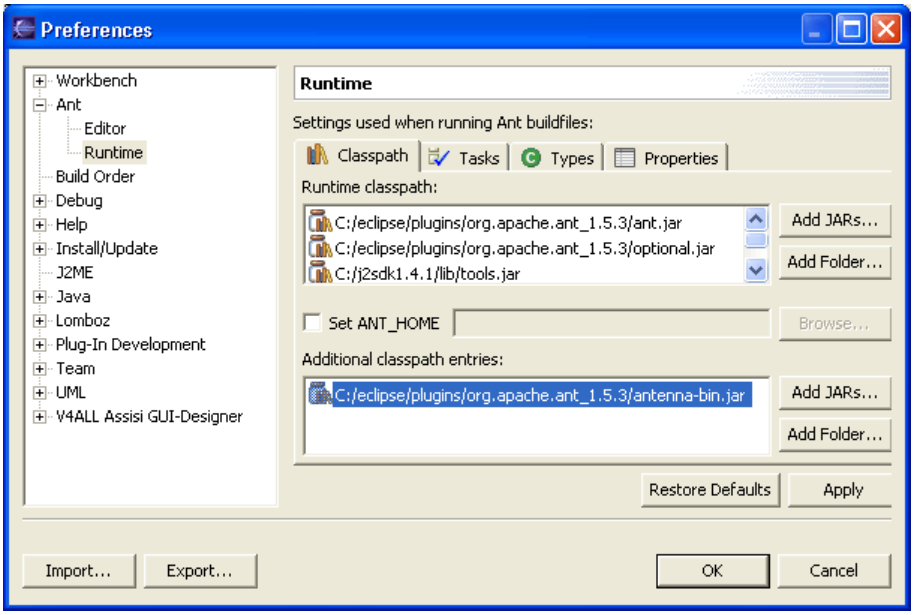
Aquí podemos ver la estructura de directorios de nuestro proyecto, los paquetes y las clases de nuestra aplicación, y las librerías utilizadas.

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.

Integracción 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:



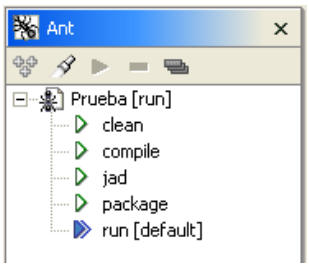
Antenna (I)

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.

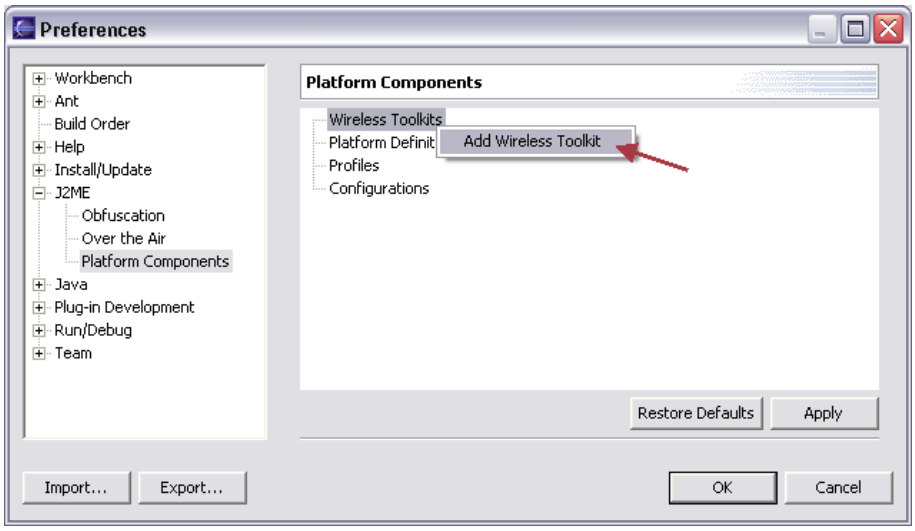


Antenna (II)

EclipseME

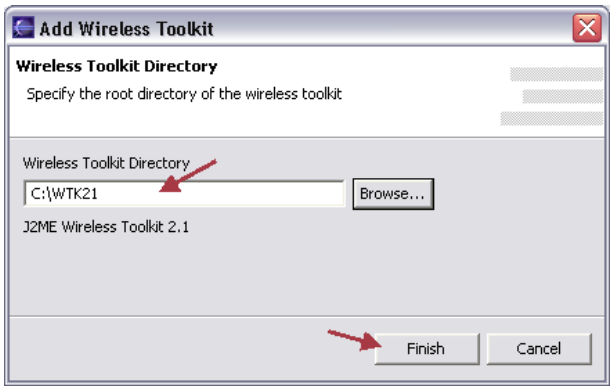
EclipseME es un *plugin* realizado por terceros que nos facilitará la creación de aplicaciones J2ME desde Eclipse.

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*:



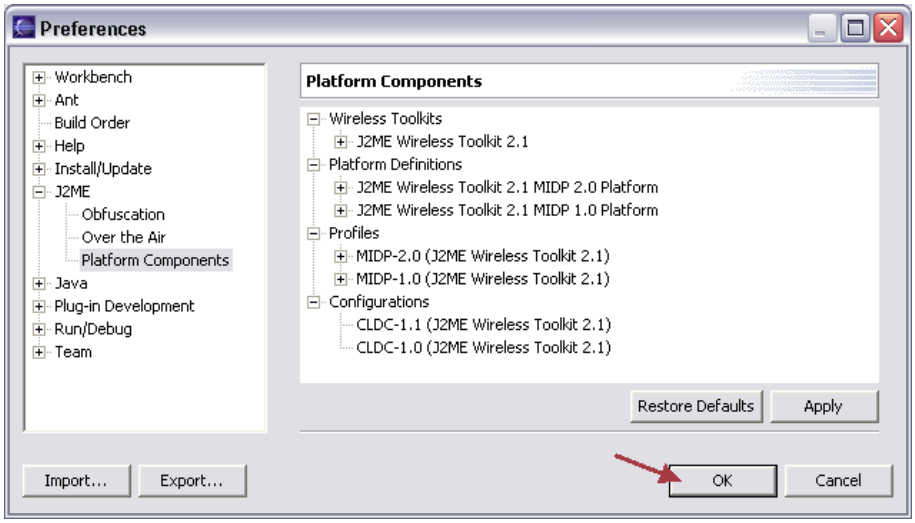
Eclipse ME (I)

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:



Eclipse ME (II)

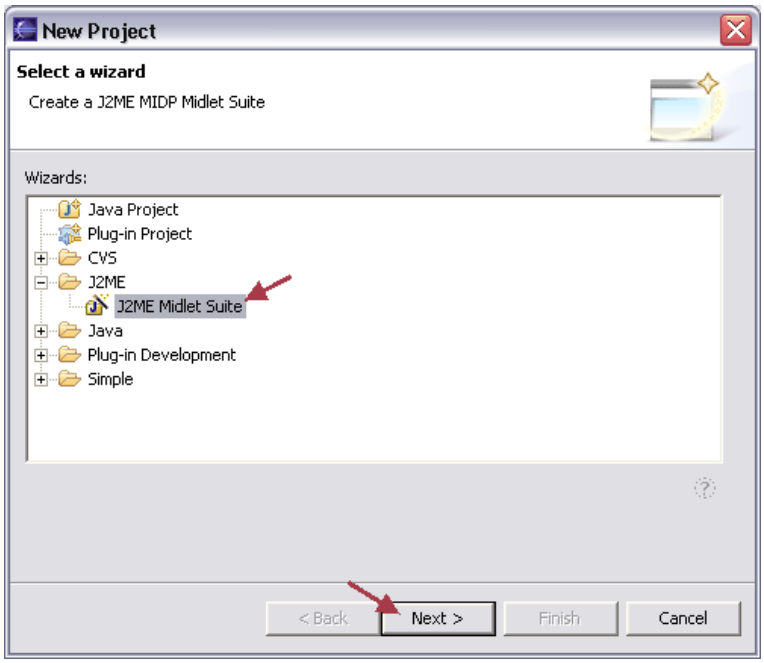
Una vez añadido un *toolkit*, se mostrarán los componentes añadidos en la ventana de configuración:



Eclipse ME (III)

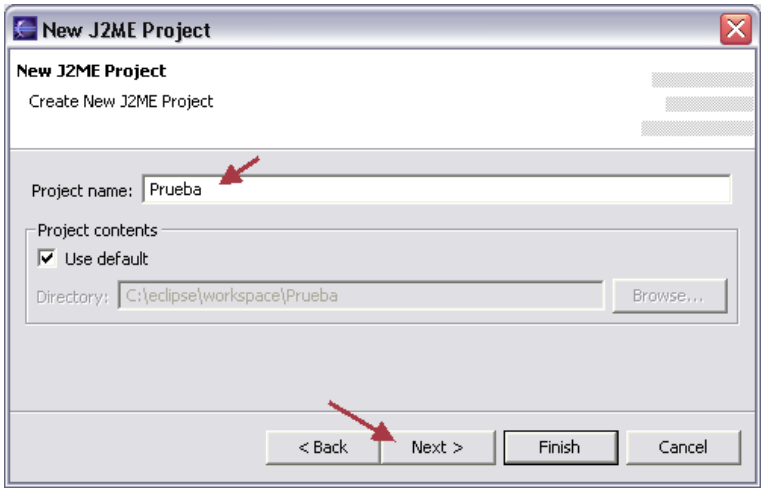
De esta forma vemos que al añadir WTK 2.1 hemos añadido soporte para los perfiles MIDP 1.0 y 2.0, y para las configuraciones CLDC 1.0 y 1.1. Podremos configurar varios *toolkits*. Por ejemplo, podemos tener configuradas las distintas versiones de WTK (1.0, 2.0, 2.1 y 2.2) para utilizar la que convenga 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:



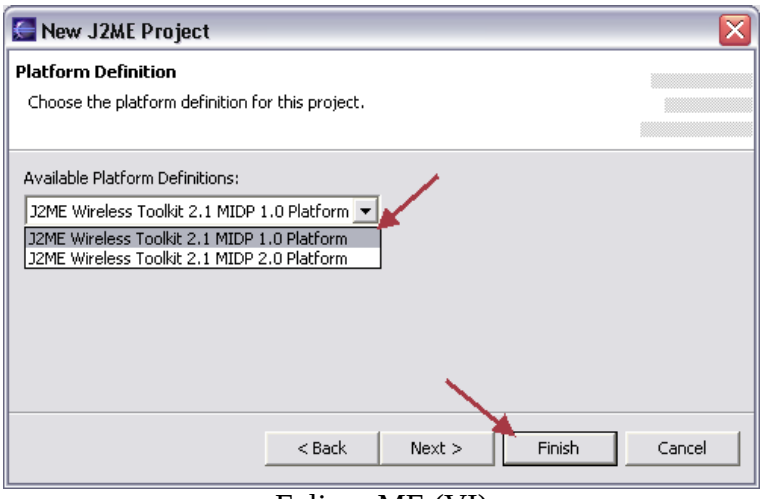
Eclipse ME (IV)

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:



Eclipse ME (V)

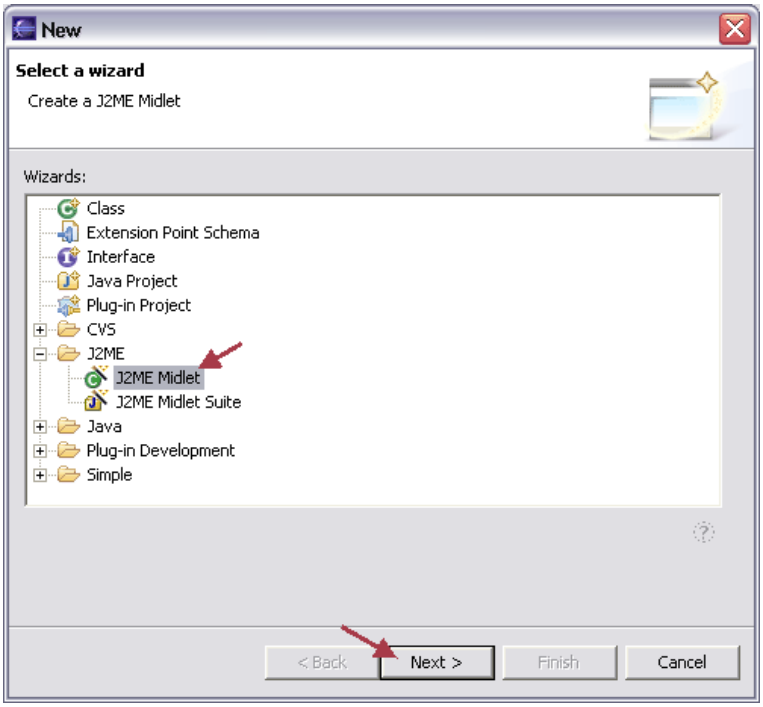
Deberemos darle un nombre al proyecto que estamos creando. En este caso podemos utilizar el directorio por defecto, ya que no vamos a utilizar WTK para construir la aplicación, la construiremos directamente desde Eclipse. Una vez asignado el nombre pulsamos sobre **Next** para ir a la siguiente ventana:



Eclipse ME (VI)

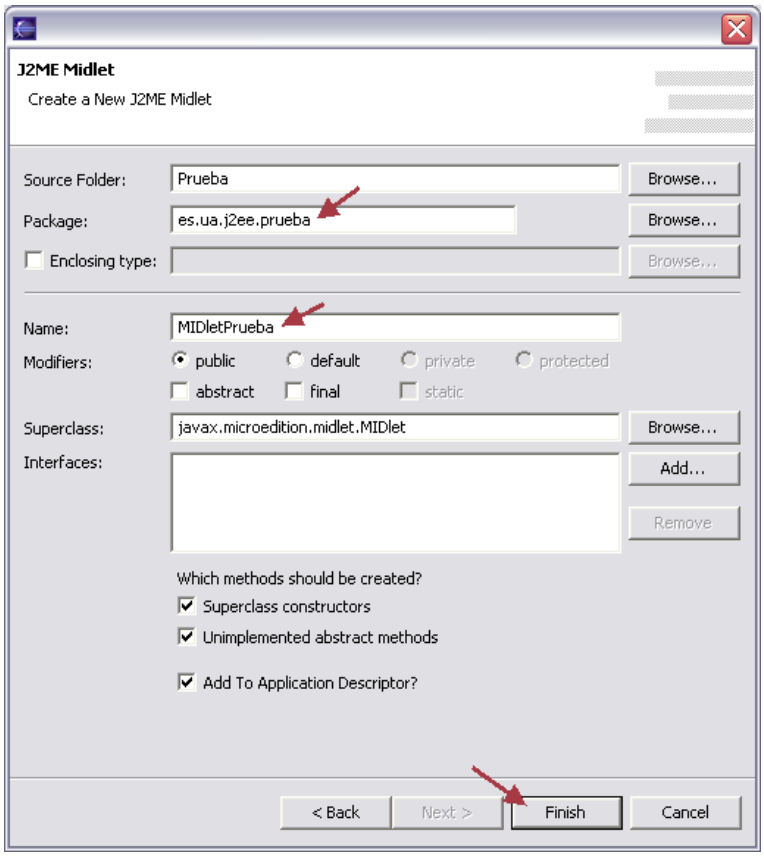
Aquí podemos elegir la versión de MIDP para la que queremos programar, siempre que tengamos instalado el WTK correspondiente para cada una de ellas. Una vez elegida la versión para la que queremos desarrollar pulsamos **Finish**, con lo que habremos terminado de configurar nuestro proyecto. En este caso 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*:



Eclipse ME (VII)

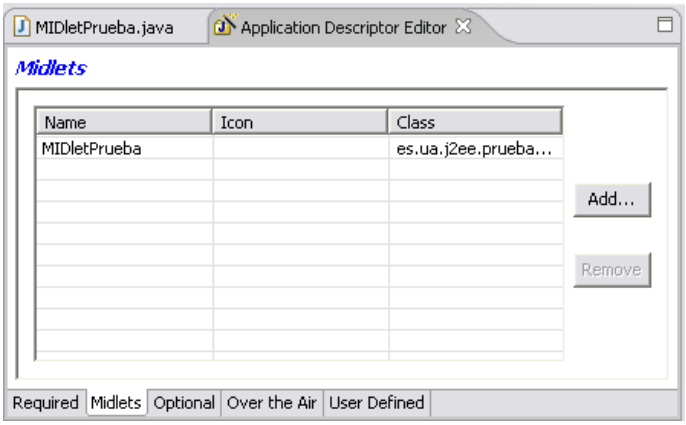
Si queremos crear un MIDlet, podremos utilizar la opción **J2ME Midlet** y pulsar **Next**, con lo que se mostrará la siguiente ventana para introducir los datos del MIDlet:



Eclipse ME (VIII)

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.

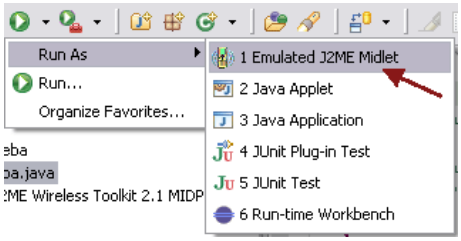
En el explorador de paquetes podemos ver las clases creadas, la librería utilizada y el fichero JAD del proyecto. Pinchando sobre el fichero JAD se mostrará en el editor la siguiente ficha con los datos de la *suite*:



Eclipse ME (IX)

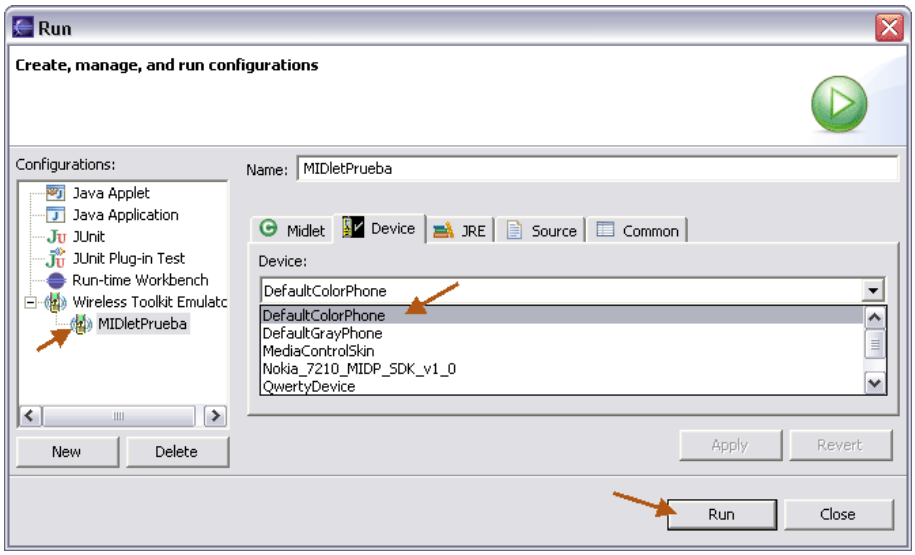
Aquí deberemos introducir la información necesaria, sobre los datos de la *suite* (**Required**) y los MIDlets que hayamos creado en ella (en la pestaña **Midlets**). Podemos ver que, cuando creemos un MIDlet mediante el asistente que acabamos de utilizar para la creación de MIDlets, los datos del MIDlet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



Eclipse ME (X)

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:



Eclipse ME (XI)

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.

1.6.2. 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.

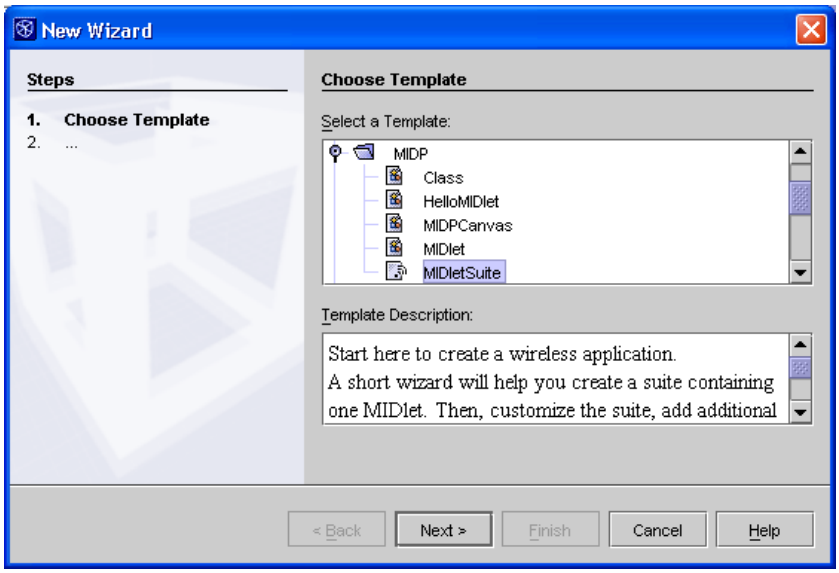
Al igual que Eclipse, el editor también nos permite autocompletar el código, colorea la sintaxis para una mayor claridad y detecta algunos fallos conforme vamos escribiendo.

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

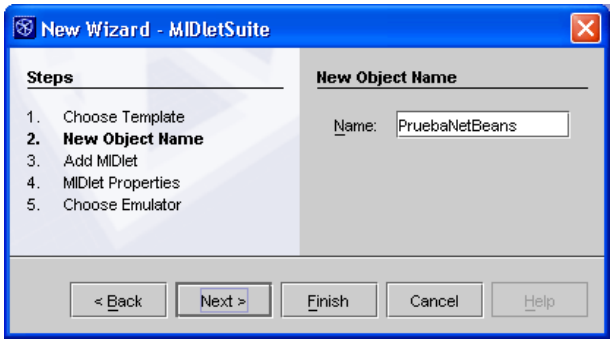
componentes MIDP y para instalar una versión de WTK integrada en el mismo entorno, por lo que no necesitaríamos instalar una versión externa. También disponemos de *plugins* con distintos ofuscadores, que podemos instalar de forma opcional, de forma que podamos ofuscar el código desde el mismo entorno.

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



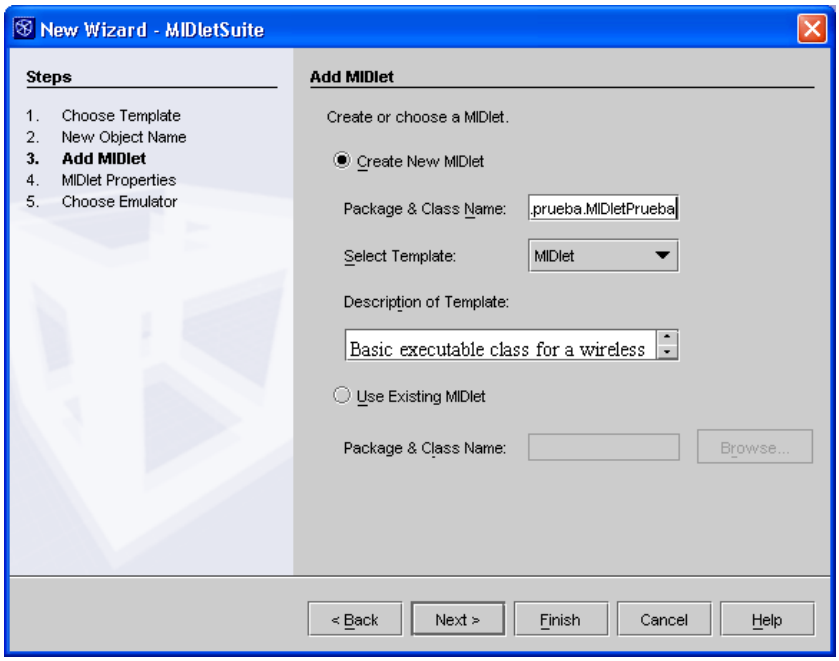
NetBeans (I)

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



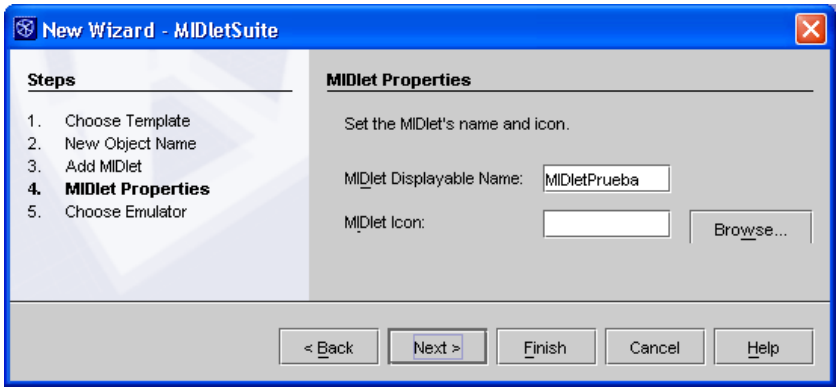
NetBeans (II)

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



NetBeans (III)

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 fuente nosotros. Deberemos 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:



NetBeans (IV)

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

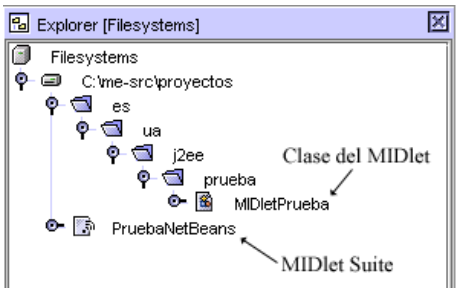
pulsar sobre **Finish** con lo que habremos terminado de crear la *suite*. Podremos ver en el explorador de NetBeans los elementos que se han creado.

Dentro del entorno tenemos tres pestañas como las siguientes:

NetBeans (V)

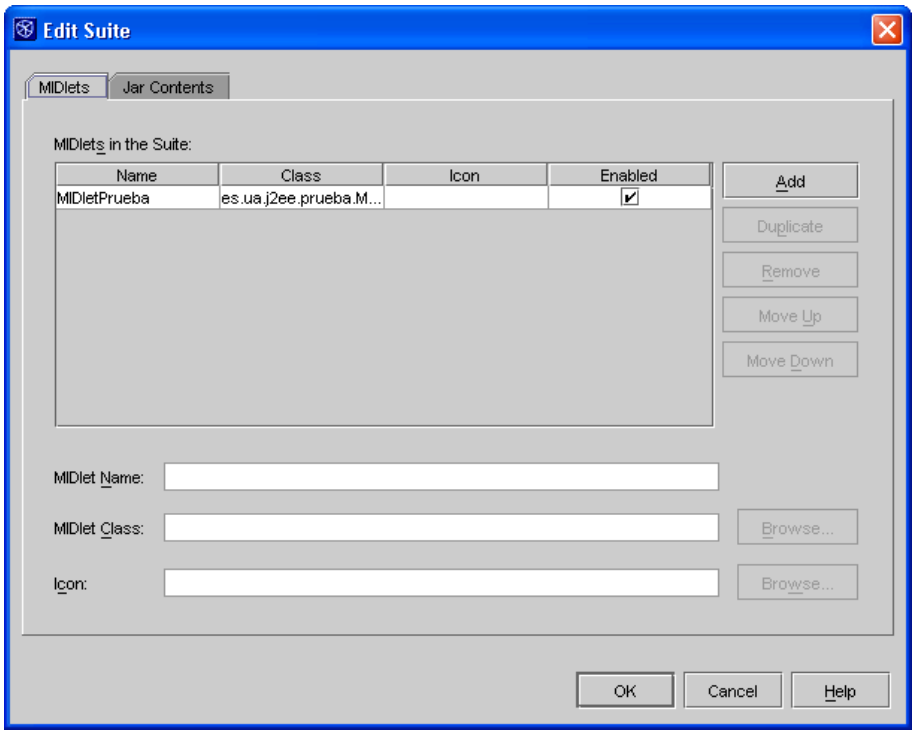
Para editar el código utilizaremos la vista de edición, teniendo seleccionada la primera pestaña (**Editing**). La segunda (**GUI Editing**) nos servirá para crear de forma visual la GUI de las aplicaciones AWT y Swing, por lo que no nos servirá para el desarrollo de aplicaciones J2ME. La tercera (**Debugging**)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



NetBeans (VI)

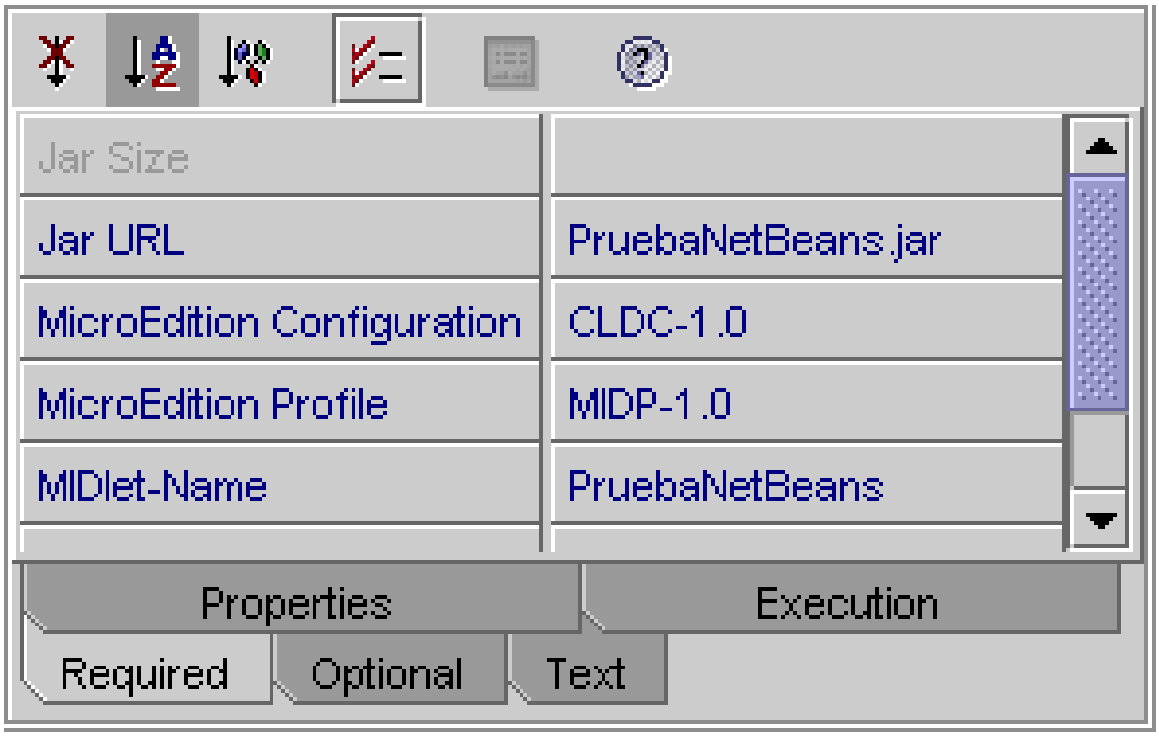
Haciendo doble *click* sobre el elemento correspondiente a la *suite* podremos modificar sus propiedades. Se abrirá la siguiente ventana:



NetBeans (VII)

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.

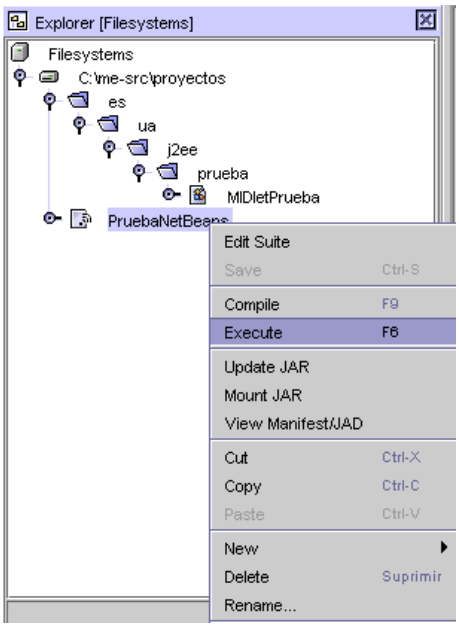
En la parte inferior el explorador tenemos el inspector de propiedades, donde podemos consultar o modificar las propiedades del elemento seleccionado actualmente en el explorador. Si tenemos seleccionado el elemento correspondientes a la *suite*, veremos las siguientes propiedades:



NetBeans (VIII)

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:



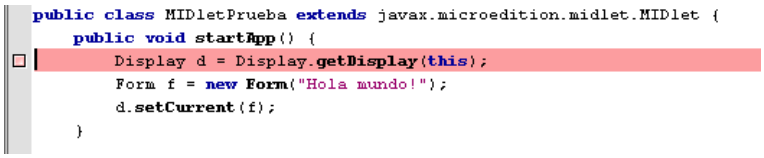
NetBeans (IX)

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

NetBeans (X)

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:

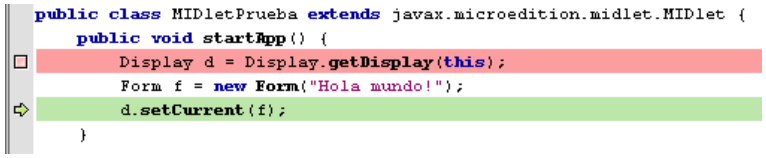


NetBeans (XI)

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

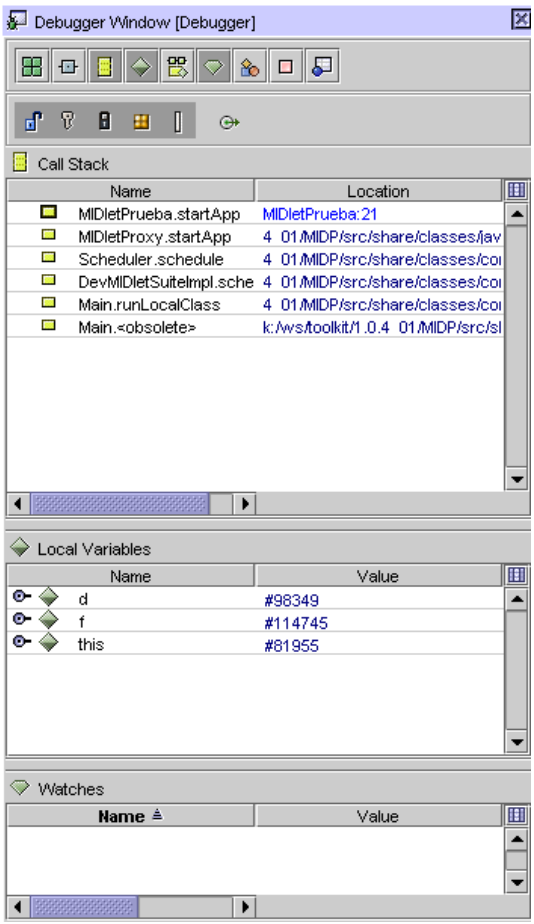
NetBeans (XII)

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:



NetBeans (XIII)

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:



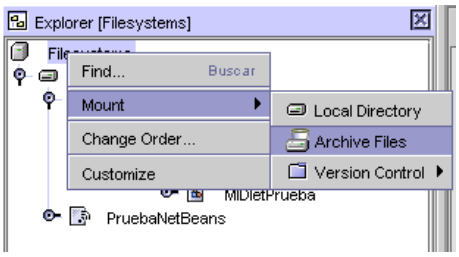
NetBeans (XIV)

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

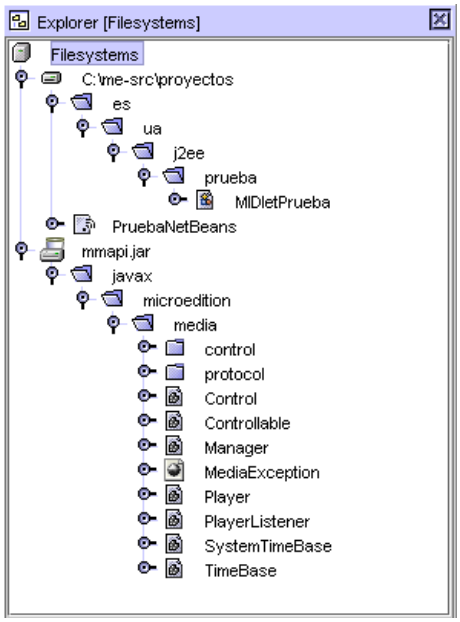
Las librerías que se utilizan al compilar y ejecutar las aplicaciones MIDP son las librerías que aporte el emulador seleccionado, al igual que ocurría con WTK. Sin embargo, conforme editamos el código sólo cuenta con que estemos utilizando la API de MIDP básica, por lo que todos los elementos que incluyamos 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.



NetBeans (XV)

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:



NetBeans (XVI)

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.

1.6.3. 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 esta 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.

Antenna

Antenna no se puede considerar un IDE sino más bien una librería de tareas para compilar y empaquetar aplicaciones para dispositivos móviles.

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.

2. Java para MIDs

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).

Hemos visto que 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**), lo cual producirá determinadas limitaciones en las aplicaciones desarrolladas para dicha máquina virtual.

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. En CLDC tendremos un subconjunto reducido y simplificado de las clases de J2SE, mientras que en MIDP tendremos clases que son exclusivas de J2ME ya que van orientadas a la programación de las características propias

de los dispositivos móviles.

En este punto estudiaremos las clases que CLDC toma de J2SE, y veremos las diferencias y limitaciones que tienen respecto a su versión en la plataforma estándar de Java.

2.1. Tipos de datos

En CLDC 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.

2.2. 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 euros.

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 euros, 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<10?"0":"") + 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 euros y 5.20 euros 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 euros.

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 euros queremos añadirle el IVA, tendremos que multiplicarlo por 1.16. Haremos la siguiente operación entera:

```
1915 * 116 = 222140
```

El resultado real será 22.2140 euros, 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.

2.3. 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.

2.3.1. Excepciones

El manejo de las excepciones se realiza de la misma forma que en J2SE.

2.3.2. Hilos

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*).

2.3.3. Colecciones

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.

2.3.4. Wrapper de tipos básicos

En CLDC tenemos *wrappers* para los tipos básicos soportados: `Boolean`, `Integer`, `Long`, `Byte`, `Short`, `Character`.

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`.

2.3.5. 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

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.

System

Podemos encontrar los objetos que encapsulan la salida y salida de error estándar. 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.

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 los propiedades del usuario que aparecen en el fichero JAD. En el próximo tema veremos cómo leer estas propiedades del usuario.

Runtime

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

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

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 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.

2.3.6. Flujos de entrada/salida

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.

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 descodificar 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

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.

2.3.7. 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.

2.4. 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.

2.4.1. 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).

2.4.2. 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.

- **Destruido**: 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.

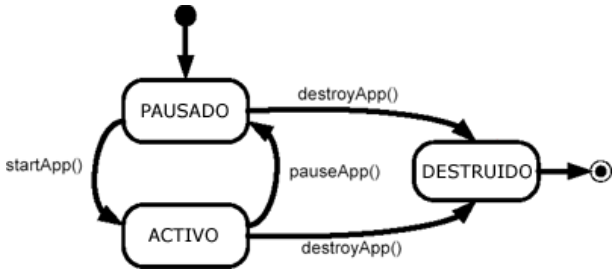
Pasajero: En el caso de que el error sea pasajero, lo notificaremos lanzando una excepción de tipo `MIDletStateChangeException`, de modo que el MIDlet pasará a estado pausado, y se volverá intentar activar más tarde.

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.



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.

2.4.3. 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) {
    }

}
```

2.4.4. 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*.

2.4.5. 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.jtech.ua.es/prueba/aplic.jad
  ```

Si como URL proporcionamos una cadena vacía (no `null`), se cancelarán todas las peticiones de servicios anteriores.