Java, JavaME y el entorno Eclipse

Índice

1 El lenguaje Java....2
1.1 Conceptos previos de POO....2
1.2 Componentes de un programa Java....4
2 Introducción a JavaME....12
2.1 Aplicaciones JavaME....13
2.2 Aplicaciones J2ME....21
2.3 Construcción de aplicaciones....25
2.4 Kits de desarrollo....30
2.5 Entornos de Desarrollo Integrados (IDEs)....44
3 Java para MIDs....59
3.1 Características ausentes en JavaME....60
3.2 MIDlets....60

1. El lenguaje Java

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

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

1.1. Conceptos previos de POO

Java es un lenguaje orientado a objetos (OO), por lo que, antes de empezara ver qué elementos componen los programas Java, conviene tener claros algunos conceptos de la programación orientada a objetos (POO).

1.1.1. Concepto de clase y objeto

El elemento fundamental a la hora de hablar de programación orientada a objetos es el concepto de objeto en sí, así como el concepto abstracto de clase. Un **objeto** es un conjunto de variables junto con los métodos relacionados con éstas. Contiene la información (las variables) y la forma de manipular la información (los métodos). Una **clase** es el prototipo que define las variables y métodos que va a emplear un determinado tipo de objeto, es la definición abstracta de lo que luego supone un objeto en memoria.

Poniendo un símil fuera del mundo de la informática, la clase podría ser el concepto de *coche*, donde nos vienen a la memoria los parámetros que definen un coche (dimensiones, cilindrada, maletero, etc), y las operaciones que podemos hacer con un coche (acelerar, frenar, adelantar, estacionar). La idea abstracta de coche que tenemos es lo que equivaldría a la clase, y la representación concreta de coches concretos (por ejemplo, Peugeot 307, Renault Megane, Volkswagen Polo...) serían los objetos de tipo coche.

1.1.2. Concepto de campo, método y constructor

Toda clase u objeto se compone internamente de constructores, campos y/o métodos. Veamos qué representa cada uno de estos conceptos: un **campo** es un elemento que contiene información relativa a la clase, y un **método** es un elemento que permite manipular la información de los campos. Por otra parte, un **constructor** es un elemento que permite reservar memoria para almacenar los campos y métodos de la clase, a la hora de crear un objeto de la misma.

1.1.3. Concepto de herencia y polimorfismo

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

Por ejemplo, podríamos tener una clase genérica *Animal*, y heredamos de ella para formar clases más específicas, como *Pato*, *Elefante*, o *León*. Estas clases tendrían todo lo de la clase padre *Animal*, y además cada una podría tener sus propios elementos adicionales.

Una característica derivada de la herencia es que, por ejemplo, si tenemos un método `dibuja(Animal a)`, que se encarga de hacer un dibujo del animal que se le pasa como parámetro, podremos pasarle a este método como parámetro tanto un *Animal* como un *Pato*, *Elefante*, o cualquier otro subtipo directo o indirecto de *Animal*. Esto se conoce como **polimorfismo**.

1.1.4. Modificadores de acceso

Tanto las clases como sus elementos (constructores, campos y métodos) pueden verse modificados por lo que se suelen llamar modificadores de acceso, que indican hasta dónde es accesible el elemento que modifican. Tenemos tres tipos de modificadores:

- **privado:** el elemento es accesible únicamente dentro de la clase en la que se encuentra.
- **protegido:** el elemento es accesible desde la clase en la que se encuentra, y además desde las subclases que hereden de dicha clase.
- **público:** el elemento es accesible desde cualquier clase.

1.1.5. Clases abstractas e interfaces

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

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

- En una **clase**, al definir *Animal* tendríamos que implementar el código de los métodos *dibuja()* e *imprime()*. Las subclases que hereden de *Animal* no tendrían por qué implementar los métodos, a no ser que quieran redefinirlos para adaptarlos a sus propias necesidades.
- En una **clase abstracta** podríamos implementar los métodos que nos interese, dejando sin implementar los demás (dejándolos como métodos abstractos). Dichos métodos tendrían que implementarse en las clases hijas.

- En un **interfaz** no podemos implementar ningún método en la clase padre, y cada clase hija tiene que hacer sus propias implementaciones de los métodos. Además, las clases hijas podrían implementar otros interfaces.

1.2. Componentes de un programa Java

En un programa Java podemos distinguir varios elementos:

1.2.1. Clases

Para definir una clase se utiliza la palabra reservada `class`, seguida del nombre de la clase:

```
class MiClase
        {
          ...
        }
```

Es recomendable que los nombres de las clases sean sustantivos (ya que suelen representar entidades), pudiendo estar formados por varias palabras. La primera letra de cada palabra estará en mayúscula y el resto de letras en minúscula. Por ejemplo, `DatosUsuario`, `Cliente`, `GestorMensajes`.

Cuando se trate de una clase encargada únicamente de agrupar un conjunto de recursos o de constantes, su nombre se escribirá en plural. Por ejemplo, `Recursos`, `MensajesError`.

1.2.2. Campos y variables

Dentro de una clase, o de un método, podemos definir campos o variables, respectivamente, que pueden ser de tipos simples, o clases complejas, bien de la API de Java, bien que hayamos definido nosotros mismos, o bien que hayamos copiado de otro lugar.

Al igual que los nombres de las clases, suele ser conveniente utilizar sustantivos que describan el significado del campo, pudiendo estar formados también por varias palabras. En este caso, la primera palabra comenzará por minúscula, y el resto por mayúscula. Por ejemplo, `apellidos`, `fechaNacimiento`, `numIteraciones`.

De forma excepcional, cuando se trate de variables auxiliares de corto alcance se puede poner como nombre las iniciales del tipo de datos correspondiente:

```
int i;
        Vector v;
        MiOtraClase moc;
```

Por otro lado, las constantes se declaran como `final static`, y sus nombres es escribirán totalmente en mayúsculas, separando las distintas palabras que los formen por caracteres de subrayado (`'_'`). Por ejemplo, `ANCHO_VENTANA`, `MSG_ERROR_FICHERO`.

1.2.3. Métodos

Los métodos o funciones se definen de forma similar a como se hacen en C: indicando el tipo de datos que devuelven, el nombre del método, y luego los argumentos entre paréntesis:

```
void imprimir(String mensaje)
        {
                ... // Código del método
        }

        double sumar(double... numeros){
                    //Número variable de argumentos
                    //Se accede a ellos como a un vector:
                    //numeros[0], numeros[1], ...
        }

        Vector insertarVector(Object elemento, int posicion)
        {
                ... // Código del método
        }
```

Al igual que los campos, se escriben con la primera palabra en minúsculas y el resto comenzando por mayúsculas. En este caso normalmente utilizaremos verbos.

Nota

Una vez hayamos creado cualquier clase, campo o método, podremos modificarlo pulsando con el botón derecho sobre él en el explorador de Eclipse y seleccionando la opción *Refactor* > *Rename...* del menú emergente. Al cambiar el nombre de cualquiera de estos elementos, Eclipse actualizará automáticamente todas las referencias que hubiese en otros lugares del código. Además de esta opción para renombrar, el menú *Refactor* contiene bastantes más opciones que nos permitirán reorganizar automáticamente el código de la aplicación de diferentes formas.

1.2.4. Constructores

Podemos interpretar los constructores como métodos que se llaman igual que la clase, y que se ejecutan con el operador `new` para reservar memoria para los objetos que se creen de dicha clase:

```
MiClase()
        {
                ... // Código del constructor
        }

        MiClase(int valorA, Vector valorV)
        {
                ... // Código de otro constructor
        }
```

No tenemos que preocuparnos de liberar la memoria del objeto al dejar de utilizarlo. Esto lo hace automáticamente el **garbage collector**. Aún así, podemos usar el método `finalize()` para liberar manualmente.

Si estamos utilizando una clase que hereda de otra, y dentro del constructor de la subclase queremos llamar a un determinado constructor de la superclase, utilizaremos `super`. Si no se hace la llamada a `super`, por defecto la superclase se construirá con su constructor vacío. Si esta superclase no tuviese definido ningún constructor vacío, o bien quisiesemos utilizar otro constructor, podremos llamar a `super` proporcionando los parámetros correspondientes al constructor al que queramos llamar. Por ejemplo, si heredamos de `MiClase` y desde la subclase queremos utilizar el segundo constructor de la superclase, al comienzo del constructor haremos la siguiente llamada a `super`:

```
SubMiClase()
        {
            super(0, new Vector());
            ... // Código de constructor subclase
        }
```

Nota

Podemos generar el constructor de una clase automáticamente con Eclipse, pulsando con el botón derecho sobre el código y seleccionando *Source > Generate Constructor Using Fields...* o *Source > Generate Constructors From Superclass...*

1.2.5. Paquetes

Las clases en Java se organizan (o pueden organizarse) en paquetes, de forma que cada paquete contenga un conjunto de clases. También puede haber subpaquetes especializados dentro de un paquete o subpaquete, formando así una jerarquía de paquetes, que después se plasma en el disco duro en una estructura de directorios y subdirectorios igual a la de paquetes y subpaquetes (cada clase irá en el directorio/subdirectorio correspondiente a su paquete/subpaquete).

Cuando queremos indicar que una clase pertenece a un determinado paquete o subpaquete, se coloca al principio del fichero la palabra reservada `package` seguida por los paquetes/subpaquetes, separados por '.' :

```
package paq1.subpaq1;
        ...
        class MiClase {
        ...
```

Si queremos desde otra clase utilizar una clase de un paquete o subpaquete determinado (diferente al de la clase en la que estamos), incluimos una sentencia `import` antes de la clase (y después de la línea `package` que pueda tener la clase, si la tiene), indicando qué paquete o subpaquete queremos importar:

```
import paq1.subpaq1.*;
```

```
import paq1.subpaq1.MiClase;
```

La primera opción (*) se utiliza para importar todas las clases del paquete (se utiliza cuando queremos utilizar muchas clases del paquete, para no ir importando una a una). La

segunda opción se utiliza para importar una clase en concreto.

Nota

Es recomendable indicar siempre las clases concretas que se están importando y no utilizar el *. De esta forma quedará más claro cuales son las clases que se utilizan realmente en nuestro código. Hay diferentes paquetes que contienen clases con el mismo nombre, y si se importasen usando * podríamos tener un problema de ambigüedad.

Al importar, ya podemos utilizar el nombre de la clase importada directamente en la clase que estamos construyendo. Si no colocásemos el `import` podríamos utilizar la clase igual, pero al referenciar su nombre tendríamos que ponerlo completo, con paquetes y subpaquetes:

```
MiClase mc;               // Si hemos hecho el 'import' antes

paq1.subpaq1.MiClase mc; // Si NO hemos hecho el 'import' antes
```

Existe un paquete en la API de Java, llamado `java.lang`, que no es necesario importar. Todas las clases que contiene dicho paquete son directamente utilizables. Para el resto de paquetes (bien sean de la API o nuestros propios), será necesario importarlos cuando estemos creando una clase fuera de dichos paquetes.

Los paquetes normalmente se escribirán totalmente en minúsculas. Es recomendable utilizar nombres de paquetes similares a la URL de nuestra organización pero a la inversa, es decir, de más general a más concreto. Por ejemplo, si nuestra URL es `jtech.ua.es` los paquetes de nuestra aplicación podrían recibir nombres como `es.ua.jtech.proyecto.interfaz`, `es.ua.jtech.proyecto.datos`, etc.

Importante

Nunca se debe crear una clase sin asignarle nombre de paquete. En este caso la clase se encontraría en el paquete `sin nombre`, y no podría ser referenciada por las clases del resto de paquetes de la aplicación.

Con Eclipse podemos importar de forma automática los paquetes necesarios. Para ello podemos pulsar sobre el código con el botón derecho y seleccionar *Source* > *Organize imports*. Esto añadirá y ordenará todos los `imports` necesarios. Sin embargo, esto no funcionará si el código tiene errores de sintaxis. En ese caso si que podríamos añadir un `import` individual, situando el cursor sobre el nombre que se quiera importar, pulsando con el botón derecho, y seleccionando *Source* > *Add import*.

1.2.6. Tipo enumerado

El tipo `enum` permite definir un conjunto de posibles valores o estados, que luego podremos utilizar donde queramos:

Ejemplo

```
// Define una lista de 3 valores y luego comprueba en un switch
        // cuál es el valor que tiene un objeto de ese tipo
        enum EstadoCivil {soltero, casado, divorciado};
        EstadoCivil ec = EstadoCivil.casado;
        ec = EstadoCivil.soltero;
        switch(ec)
        {
                case soltero: System.out.println("Es soltero");
                                break;
                case casado: System.out.println("Es casado");
                                break;
                case divorciado:System.out.println("Es divorciado");
                                break;
        }
```

Los elementos de una enumeración se comportan como objetos Java. Por lo tanto, la forma de nombrar las enumeraciones será similar a la de las clases (cada palabra empezando por mayúscula, y el resto de clases en minúscula).

Como objetos Java que son, estos elementos pueden tener definidos campos, métodos e incluso constructores. Imaginemos por ejemplo que de cada tipo de estado civil nos interesase conocer la retención que se les aplica en el sueldo. Podríamos introducir esta información de la siguiente forma:

```
enum EstadoCivil {soltero(0.14f), casado(0.18f), divorciado(0.14f);
                private float retencion;

                EstadoCivil(float retencion) {
                        this.retencion = retencion;
                }

                public float getRetencion() {
                        return retencion;
                }
        };
```

De esta forma podríamos calcular de forma sencilla la retención que se le aplica a una persona dado su salario y su estado civil de la siguiente forma:

```
public float calculaRetencion(EstadoCivil ec, float salario) {
            return salario * ec.getRetencion();
        }
```

Dado que los elementos de la enumeración son objetos, podríamos crear nuevos métodos o bien sobrescribir métodos de la clase `Object`. Por ejemplo, podríamos redefinir el método `toString` para especificar la forma en la que se imprime cada elemento de la enumeración (por defecto imprime una cadena con el nombre del elemento, por ejemplo `"soltero"`).

1.2.7. Modificadores de acceso

Tanto las clases como los campos y métodos admiten modificadores de acceso, para indicar si dichos elementos tienen ámbito *público, protegido* o *privado*. Dichos modificadores se marcan con las palabras reservadas `public`, `protected` y `private`, respectivamente, y se colocan al principio de la declaración:

```
public class MiClase {
        ...
        protected int b;
        ...
        private int miMetodo(int b) {
        ...
```

El modificador `protected` implica que los elementos que lo llevan son visibles desde la clase, sus subclases, y las demás clases del mismo paquete que la clase.

Si no se especifica ningún modificador, el elemento será considerado de tipo *paquete*. Este tipo de elementos podrán ser visibles desde la clase o desde clases del mismo paquete, pero no desde las subclases.

Cada fichero Java que creemos debe tener una y sólo una **clase pública** (que será la clase principal del fichero). Dicha clase debe llamarse igual que el fichero. Aparte, el fichero podrá tener otras clases internas, pero ya no podrán ser públicas.

Por ejemplo, si tenemos un fichero `MiClase.java`, podría tener esta apariencia:

```
public class MiClase
        {
                ...
        }

        class OtraClase
        {
                ...
        }

        class UnaClaseMas
        {
                ...
        }
```

Si queremos tener acceso a estas clases internas desde otras clases, deberemos declararlas como estáticas. Por ejemplo, si queremos definir una etiqueta para incluir en los puntos 2D definidos en el ejemplo anterior, podemos definir esta etiqueta como clase interna (para dejar claro de esta forma que dicha etiqueta es para utilizarse en `Punto2D`). Para poder manipular esta clase interna desde fuera deberemos declararla como estática de la siguiente forma:

```
public class Punto2D {
            ...
                static class Etiqueta {
                        String texto;
                        int tam;
                        Color color;
                }
        }
```

Podremos hacer referencia a ella desde fuera de `Punto2D` de la siguiente forma:

```
Punto2D.Etiqueta etiq = new Punto2D.Etiqueta();
```

1.2.8. Otros modificadores

Además de los modificadores de acceso vistos antes, en clases, métodos y/o campos se pueden utilizar también estos modificadores:

- `abstract`: elemento base para la herencia (los objetos subtipo deberán definir este elemento). Se utiliza para definir clases abstractas, y métodos abstractos dentro de dichas clases, para que los implementen las subclases que hereden de ella.
- `static`: elemento compartido por todos los objetos de la misma clase. Con este modificador, no se crea una copia del elemento en cada objeto que se cree de la clase, sino que todos comparten una sola copia en memoria del elemento, que se crea sin necesidad de crear un objeto de la clase que lo contiene. Como se ha visto anteriormente, también puede ser aplicado sobre clases, con un significado diferente en este caso.
- `final`: objeto final, no modificable (se utiliza para definir constantes) ni heredable (en caso de aplicarlo a clases).
- `synchronized`: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución.

Estos modificadores se colocan tras los modificadores de acceso:

```
// Clase abstracta para heredar de ella
        public abstract class Ejemplo
        {
          // Constante estática de valor 10
          public static final TAM = 10;

          // Método abstracto a implementar
          public abstract void metodo();

          public synchronized void otroMetodo()
          {
            ... // Aquí dentro sólo puede haber un hilo a la vez
          }
        }
```

Nota

Si tenemos un método estático (`static`), dentro de él sólo podremos utilizar elementos estáticos (campos o métodos estáticos), o bien campos y métodos de objetos que hayamos creado dentro del método.

Por ejemplo, si tenemos:

```
public class UnaClase
        {
          public int a;

          public static int metodo()
          {
            return a + 1;
          }
        }
```

Dará error, porque el campo a no es estático, y lo estamos utilizando dentro del método estático. Para solucionarlo tenemos dos posibilidades: definir a como estático (si el diseño

del programa lo permite), o bien crear un objeto de tipo `UnaClase` en el método, y utilizar su campo a (que ya no hará falta que sea estático, porque hemos creado un objeto y ya podemos acceder a su campo a):

```
public class UnaClase
        {
          public int a;

          public static int metodo()
          {
            UnaClase uc = new UnaClase();
            // ... Aquí haríamos que uc.a tuviese el valor adecuado
            return uc.a + 1;
          }
        }
```

Nota

Para hacer referencia a un elemento estático utilizaremos siempre el nombre de la clase a la que pertenece, y no la referencia a una instancia concreta de dicha clase.

Por ejemplo, si hacemos lo siguiente:

```
UnaClase uc = new UnaClase();
        uc.metodo();
```

Aparecerá un *warning*, debido a que el método `metodo` no es propio de la instancia concreta uc, sino da la clase `UnaClase` en general. Por lo tanto, deberemos llamarlo con:

```
UnaClase.metodo();
```

1.2.9. Imports estáticos

Los *imports* estáticos permiten importar los elementos estáticos de una clase, de forma que para referenciarlos no tengamos que poner siempre como prefijo el nombre de la clase. Por ejemplo, podemos utilizar las constantes de color de la clase `java.awt.Color`, o bien los métodos matemáticos de la case `Math`.

Ejemplo

```
import static java.awt.Color;
        import static java.lang.Math;

        public class...
        {
        ...
          JLabel lbl = new JLabel();
          lbl.setBackground(white);      // Antes sería Color.white
        ...
          double raiz = sqrt(1252.2);  // Antes sería Math.sqrt(...)
        }
```

1.2.10. Argumentos variables

Java permite pasar un número variable de argumentos a una función (como sucede con

funciones como `printf` en C). Esto se consigue mediante la expresión "..." a partir del momento en que queramos tener un número variable de argumentos.

Ejemplo

```
// Funcion que tiene un parámetro String obligatorio
        // y n parámetros int opcionales

        public void miFunc(String param, int... args)
        {
        ...
                // Una forma de procesar n parametros variables
                for (int argumento: args)
                {
                        ...
                }
        ...
        }

        ...
        miFunc("Hola", 1, 20, 30, 2);
        miFunc("Adios");
```

1.2.11. Metainformación o anotaciones

Se tiene la posibilidad de añadir ciertas **anotaciones** en campos, métodos, clases y otros elementos, que permitan a las herramientas de desarrollo o de despliegue leerlas y realizar ciertas tareas. Por ejemplo, generar ficheros fuentes, ficheros XML, o un *Stub* de métodos para utilizar remotamente con RMI.

Un ejemplo más claro lo tenemos en las anotaciones que ya se utilizan para la herramienta Javadoc. Las marcas `@deprecated` no afectan al comportamiento de los métodos que las llevan, pero previenen al compilador para que muestre una advertencia indicando que el método que se utiliza está desaconsejado. También se tienen otras marcas `@param`, `@return`, `@see`, etc, que utiliza Javadoc para generar las páginas de documentación y las relaciones entre ellas.

1.2.12. Ejecución de clases: método main

En las clases principales de una aplicación (las clases que queramos ejecutar) debe haber un método `main` con la siguiente estructura:

```
public static void main(String[] args)
        {
                ... // Código del método
        }
```

Dentro pondremos el código que queramos ejecutar desde esa clase. Hay que tener en cuenta que `main` es estático, con lo que dentro sólo podremos utilizar campos y métodos estáticos, o bien campos y métodos de objetos que creemos dentro del `main`.

2. Introducción a JavaME

2.1. Aplicaciones JavaME

La plataforma JavaME 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 JavaME y ejecutarla en él.

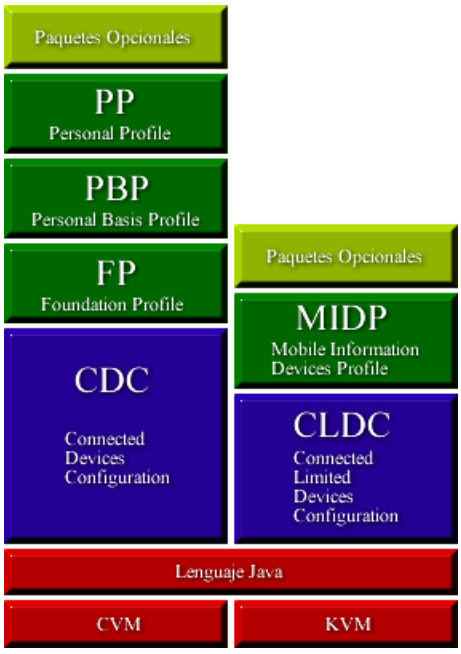
JavaME soporta una gran variedad de dispositivos, no únicamente MIDs. Actualmente define APIs para dar soporte a los dispositivos conectados en general, tanto aquellos con una gran capacidad como a tipos más limitados de estos dispositivos.

JavaME es el nombre corto de Java Micro Edition y también es ampliamente conocido con su anterior nombre comercial, J2ME que significa Java 2 Micro Edition. A partir de ahora nos referiremos a la plataforma por ambos nombres indistintamente

2.1.1. Arquitectura de JavaME

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.

2.1.2. 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.

2.1.3. 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.

2.1.4. JWTI

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

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

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

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

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

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

2.1.5. 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

2.2. 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.

2.2.1. Suite de MIDlets

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

El fichero JAR será el que contendrá las aplicaciones de la *suite*. En él tendremos tanto el código compilado como los recursos que necesite para ejecutarse (imágenes, sonidos,

etc). Estos ficheros JAR son un estándar de la plataforma Java, disponibles en todas las ediciones de esta plataforma, que nos permitirán empaquetar una aplicación Java en un solo fichero. Al ser un estándar de la plataforma Java será portable a cualquier sistema donde contemos con esta plataforma.

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

2.2.2. 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.

2.2.3. 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.

2.2.4. 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
```

2.2.5. 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.

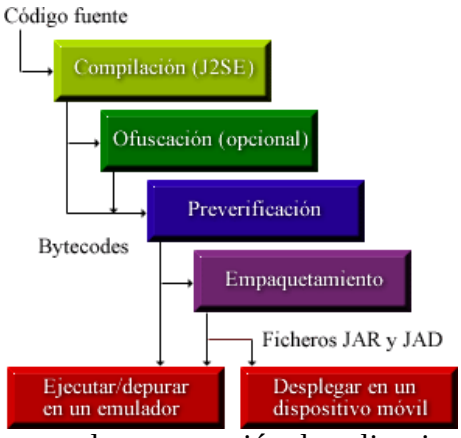
En este fichero, a diferencia del fichero JAD, no podremos introducir propiedades propias del usuario, ya que desde dentro de la aplicación no podremos acceder a los propiedades contenidas en este fichero.

2.3. 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.

2.3.1. 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.

2.3.2. 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.

2.3.3. Preverificación

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

La máquina virtual de Java hace una verificación de las clases que ejecuta en ella. Este proceso de verificación es bastante complejo para la KVM, por lo que deberemos reorganizar el código intermedio generado para facilitar esta tarea de verificación. En esto consiste la fase de preverificación que deberemos realizar antes de llevar la aplicación a un dispositivo real.

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

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

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

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

2.3.4. Creación de la suite

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

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

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

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

2.3.5. Prueba en emuladores

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

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

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

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

2.3.6. Prueba de la aplicación en dispositivos reales

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

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

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

2.3.7. Despliegue

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

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

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

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

2.4. 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:

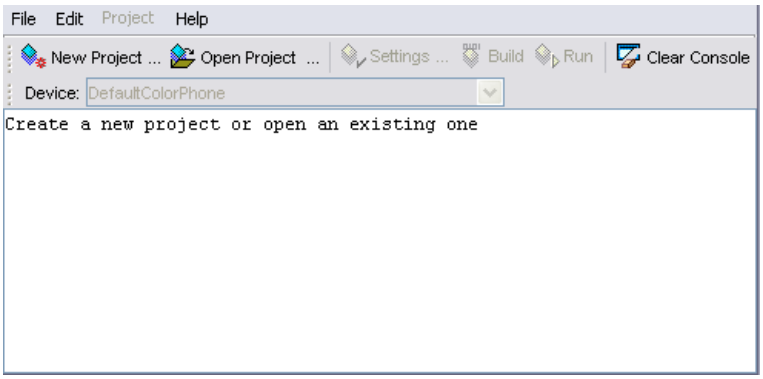
- **WTK 1.0.4**: Soporta MIDP 1.0 y CLDC 1.0. Será adecuado para desarrollar aplicaciones para móviles que soporten sólo esta versión de MIDP, aunque también funcionarán con modelos que soporten versiones posteriores de MIDP, aunque en estos casos no estaremos aprovechando al máximo las posibilidades 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).

2.4.1. 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.

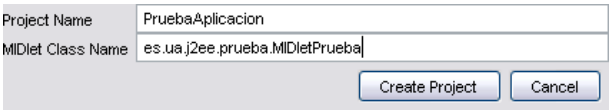
NOTA: Si lo que queremos es utilizar en nuestra aplicación una API opcional soportada por el móvil, no debemos introducirla en este directorio. En ese caso sólo deberemos añadirla al `classpath` a la hora de compilar, pero no introducirla en este directorio ya que el móvil ya cuenta con su propia implementación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

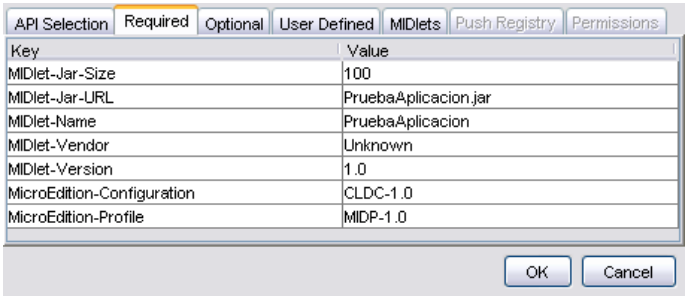
Cuando queramos crear una nueva aplicación, lo primero que haremos será pulsar el botón **"New Project ..."** para abrir el asistente de creación de aplicaciones. Lo primero que nos pedirá es el nombre que queremos darla a la aplicación, y el nombre de la clase principal (MIDlet) que vamos a crear:



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.

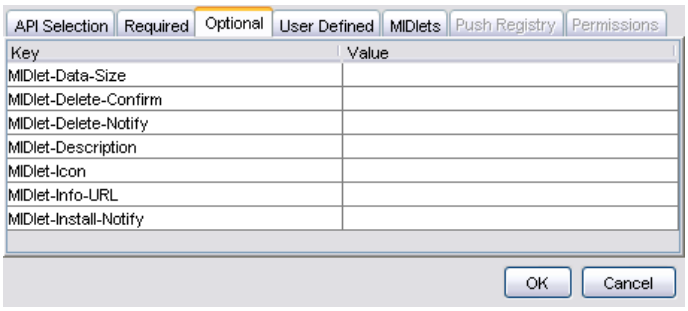
Una vez hayamos introducido estos datos, pulsamos **"Create Project"** y nos aparecerá una ficha para introducir todos los datos necesarios para crear el fichero JAD y el `MANIFEST.MF` del JAR. Con los datos introducidos en la ventana anterior habrá rellenado todos los datos necesarios, pero nosotros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



Configuración de la aplicación

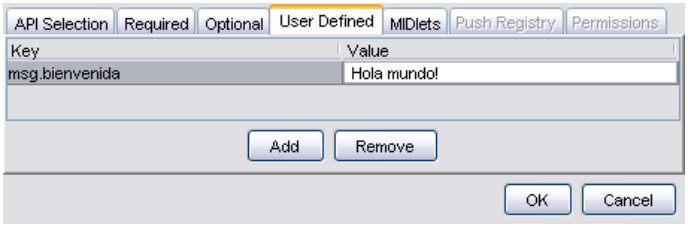
Como nombre de la *suite* y del JAR habrá tomado por defecto el nombre del proyecto que hayamos especificado. Será conveniente modificar los datos del fabricante y de la versión, para adaptarlos a nuestra aplicación. No debemos preocuparnos por especificar el tamaño del JAR, ya que este dato será actualizado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

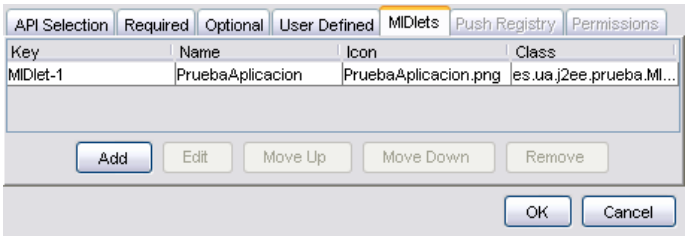
Estos datos están vacíos por defecto, ya que no son necesarios, pero podemos darles algún valor si lo deseamos. Estas son las propiedades opcionales que reconoce el AMS. Si queremos añadir propiedades propias de nuestra aplicación, podemos utilizar la tercera pestaña:



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:

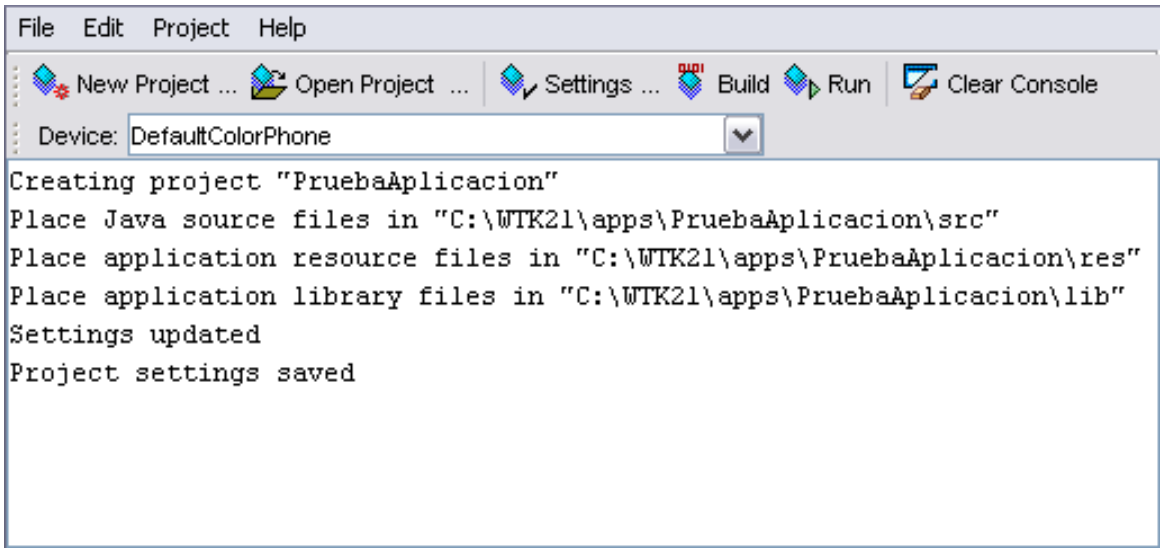


Datos de los MIDlets

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.

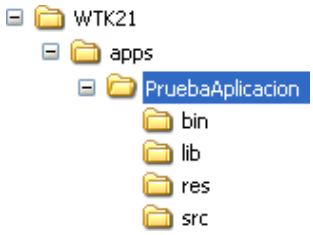
Dado que una *suite* puede contener más de un MIDlet, desde esta pestaña podremos añadir tantos MIDlets como queramos, especificando para cada uno de ellos su nombre, icono (de forma opcional) y clase.

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:

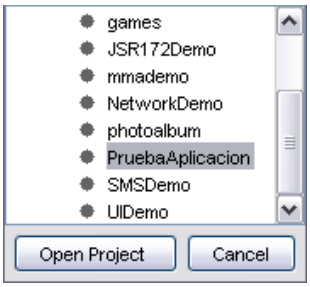


Estructura de directorios

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.

2.4.2. 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.

2.4.3. 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.

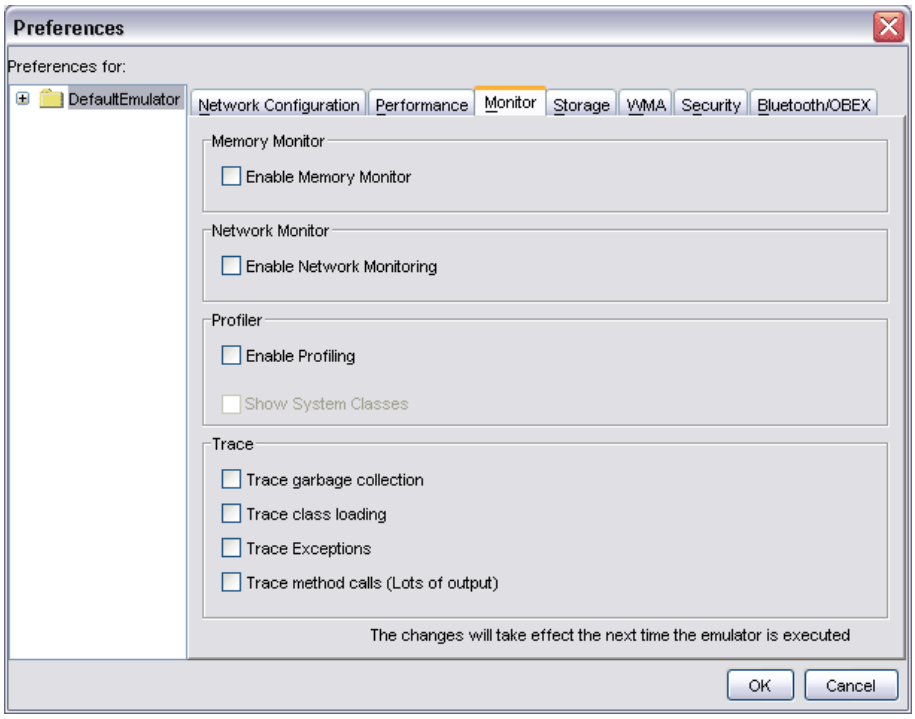
Además de estos, podemos incorporar otros emuladores al kit de desarrollo. Por ejemplo, los emuladores proporcionados por Nokia, imitando diversos modelos de teléfonos móviles de dicha compañía, pueden ser integrados fácilmente en WTK.

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.

2.4.4. 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.

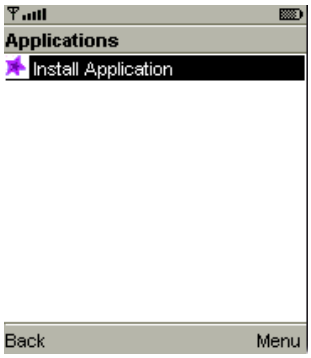
Para desplegar una aplicación de forma que sea accesible vía OTA, simplemente deberemos:

- 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.

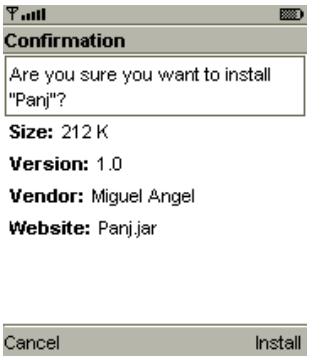
Este procedimiento puede ser demasiado costoso si queremos probar la aplicación repetidas veces utilizando este procedimiento, ya que nos obligaría, para cada nueva prueba que quisiésemos hacer, a volver a desplegar la aplicación en el servidor web.

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)

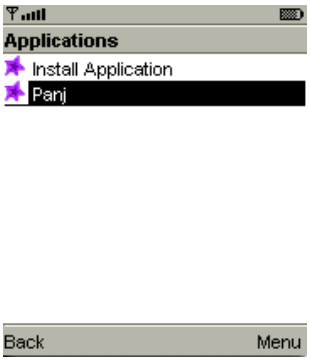
Una vez introducida la dirección del documento web donde tenemos publicada nuestra aplicación, nos mostrará la lista de enlaces a ficheros JAD que tengamos en esa página. Podremos seleccionar uno de estos enlaces para instalar la aplicación. En ese momento descargará el fichero JAD y nos mostrará la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

2.5. Entornos de Desarrollo Integrados (IDEs)

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

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

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

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

2.5.1. Eclipse

Eclipse es un entorno de desarrollo de libre distribución altamente modular. Una de sus ventajas es que no necesita demasiados recursos para ejecutarse correctamente, por lo que será adecuado para máquinas poco potentes.

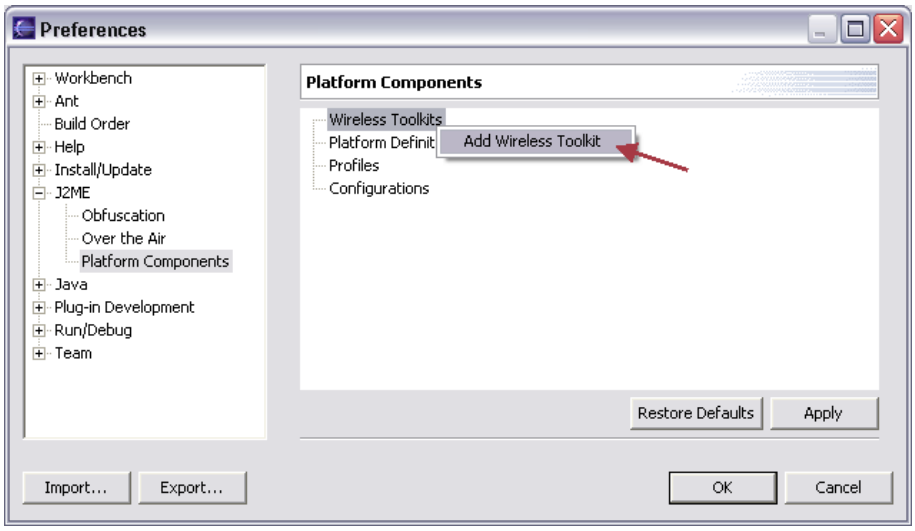
Este entorno nos permite crear proyectos en Java. Nos ofrece un editor, en el que podemos escribir el código, viendo la sintaxis coloreada para mayor claridad, y notificándonos de los errores que hayamos cometido al escribir el código, como por ejemplo haber escrito mal el nombre de un método, o usar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

Existe un *plugin* que nos permite desarrollar aplicaciones J2ME desde Eclipse. Su asistente nos permite crear el esqueleto de una aplicación con las librerías importadas. También permite lanzar el emulador a la hora de ejecutar ua aplicacion JavaME.

EclipseME

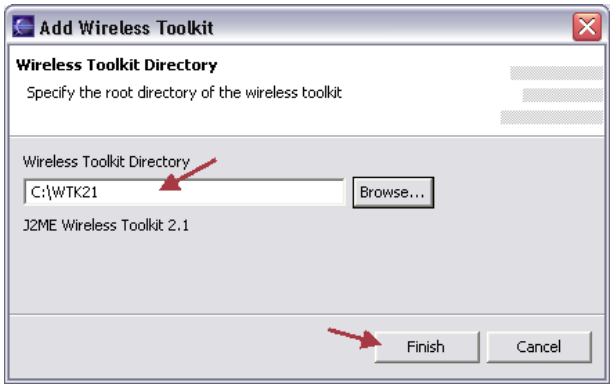
EclipseME es un *plugin* realizado por terceros que nos facilitará la creación de aplicaciones J2ME desde Eclipse.

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*:



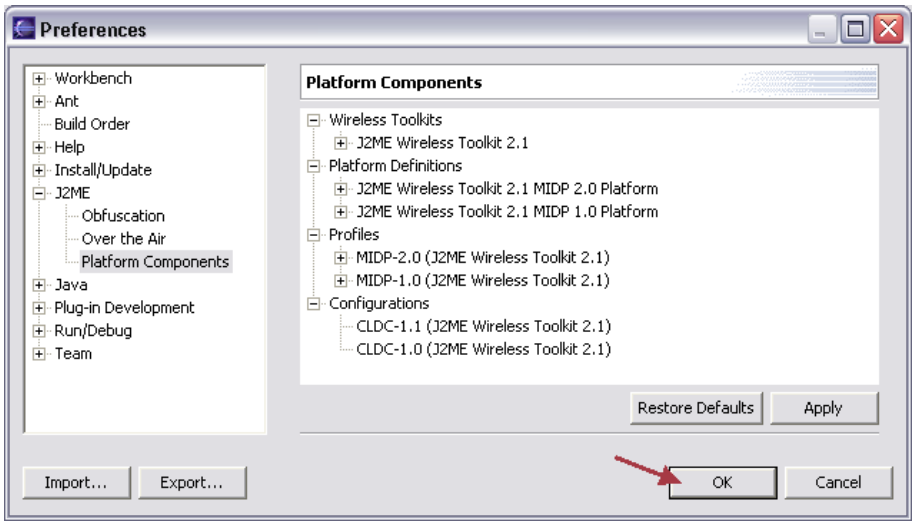
Eclipse ME (I)

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)

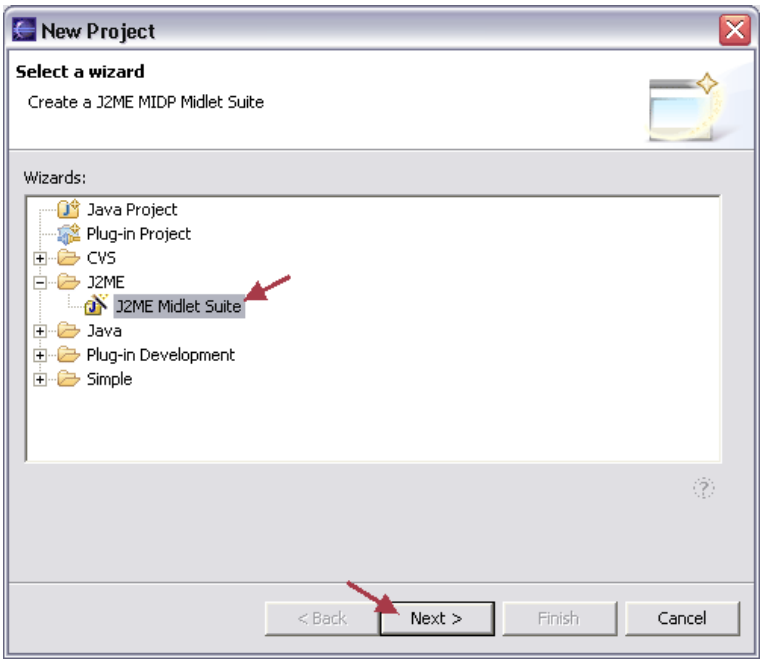
Una vez añadido un *toolkit*, se mostrarán los componentes añadidos en la ventana de configuración:



Eclipse ME (III)

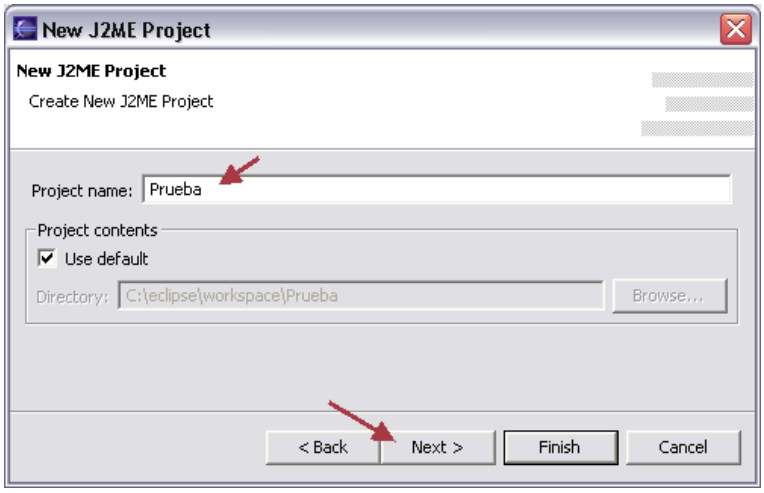
De esta forma vemos que al añadir WTK 2.1 hemos añadido soporte para los perfiles MIDP 1.0 y 2.0, y para las configuraciones CLDC 1.0 y 1.1. Podremos configurar varios *toolkits*. Por ejemplo, podemos tener configuradas las distintas versiones de WTK (1.0, 2.0, 2.1 y 2.2) para utilizar la que convenga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



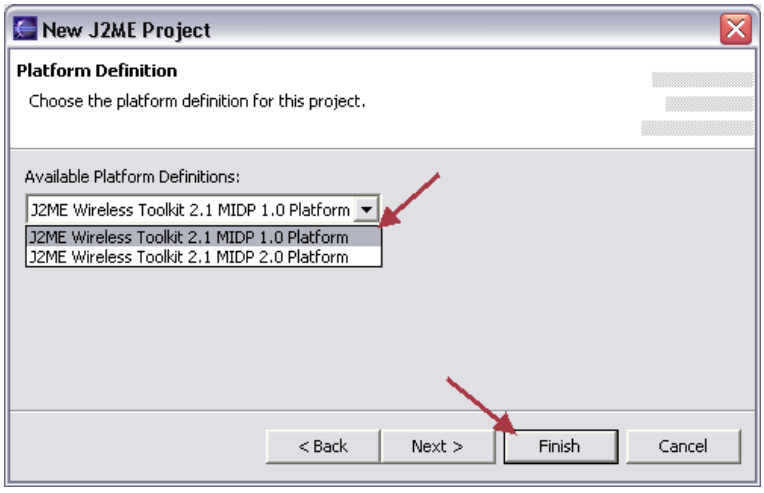
Eclipse ME (IV)

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:

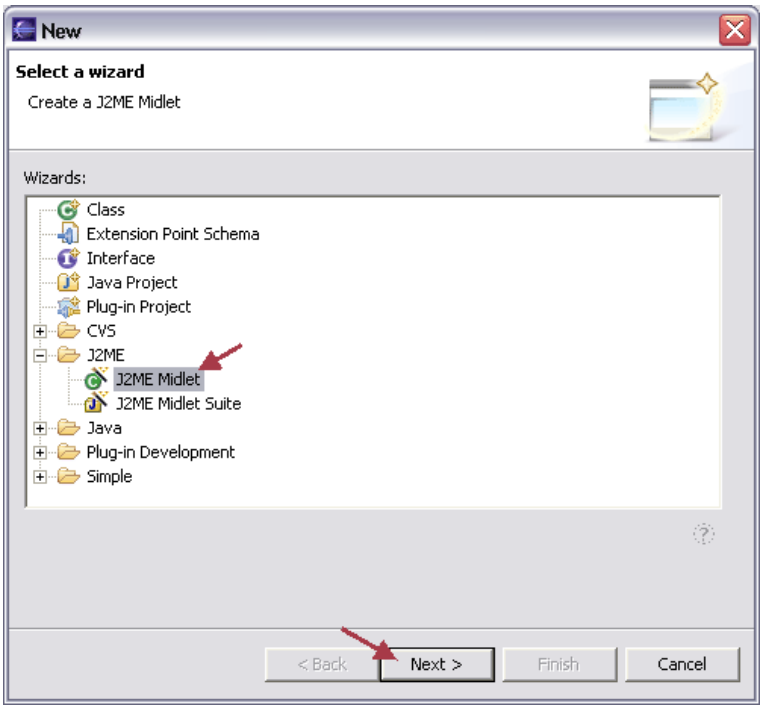


Eclipse ME (VI)

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

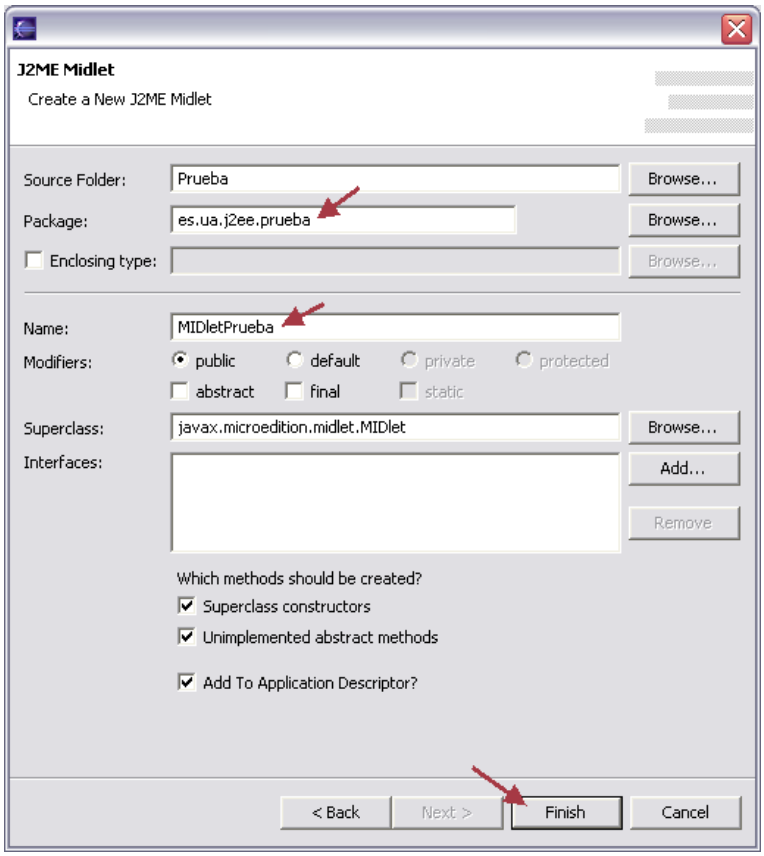
automática.

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)

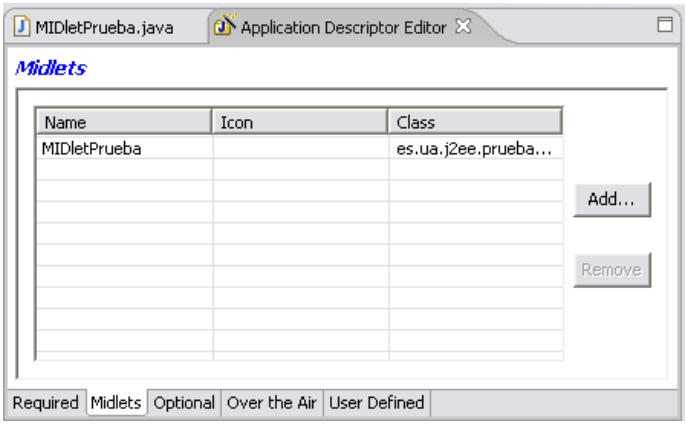
Si queremos crear un MIDlet, podremos utilizar la opción **J2ME Midlet** y pulsar **Next**, con lo que se mostrará la siguiente ventana para introducir los datos del MIDlet:



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.

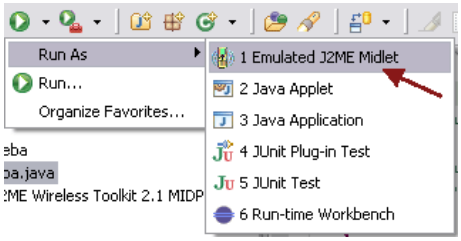
En el explorador de paquetes podemos ver las clases creadas, la librería utilizada y el fichero JAD del proyecto. Pinchando sobre el fichero JAD se mostrará en el editor la siguiente ficha con los datos de la *suite*:



Eclipse ME (IX)

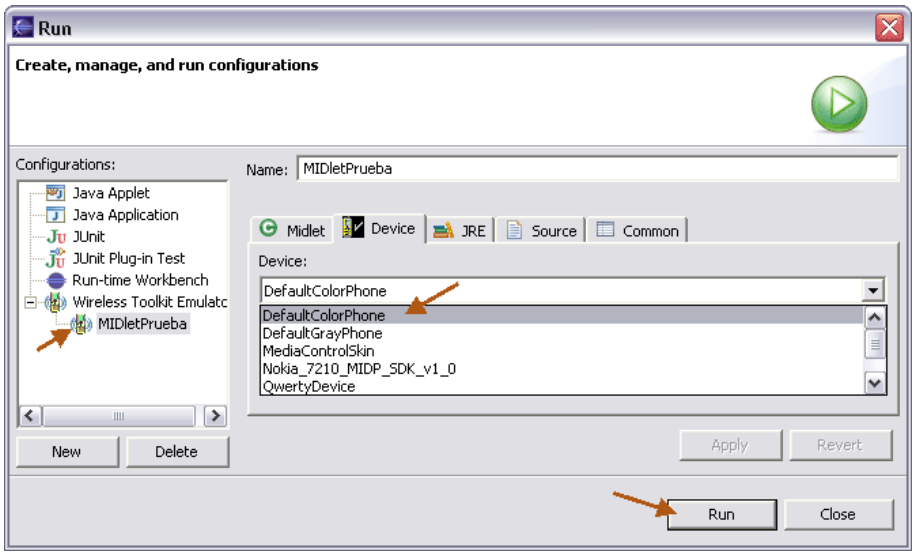
Aquí deberemos introducir la información necesaria, sobre los datos de la *suite* (**Required**) y los MIDlets que hayamos creado en ella (en la pestaña **Midlets**). Podemos ver que, cuando creemos un MIDlet mediante el asistente que acabamos de utilizar para la creación de MIDlets, los datos del MIDlet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

2.5.2. Depuración con Eclipse

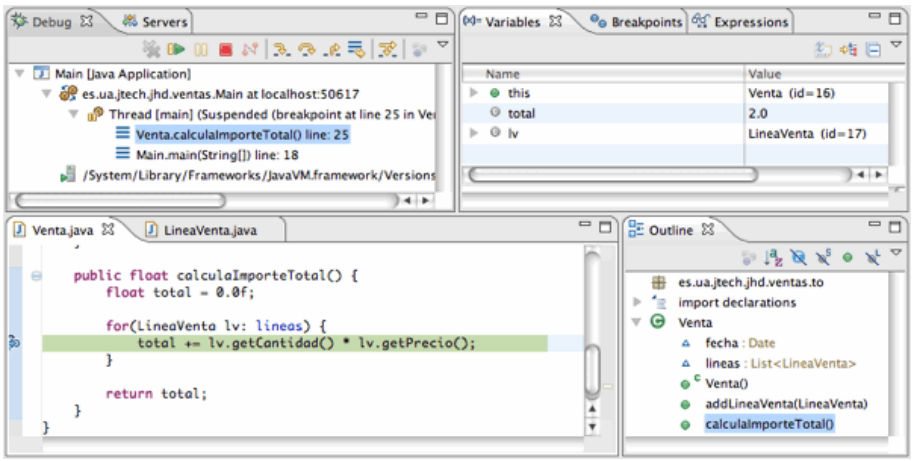
En este apartado veremos cómo podemos depurar el código de nuestras aplicaciones desde el depurador que incorpora Eclipse, encontrando el origen de los errores que provoque nuestro programa. Este depurador incorpora muchas funcionalidades, como la ejecución paso a paso, establecimiento de *breakpoints*, consulta y establecimiento de valores de variables, parar y reanudar hilos de ejecución, etc.

Eclipse proporciona una vista de depuración (*debug view*) que permite controlar el depurado y ejecución de programas. En él se muestra la pila de los procesos e hilos que tengamos ejecutando en cada momento.

2.5.2.1. Primeros pasos para depurar un proyecto

En primer lugar, debemos tener nuestro proyecto hecho y correctamente compilado. Una vez hecho eso, ejecutaremos la aplicación, pero en lugar de hacerlo desde el menú *Run As* lo haremos desde el menú *Debug As* (pinchando sobre el botón derecho sobre la clase que queramos ejecutar/depurar).

En algunas versiones de Eclipse, al depurar el código pasamos directamente a la perspectiva de depuración (*Debug perspective*), pero para cambiar manualmente, vamos a **Window - Open perspective - Debug**.



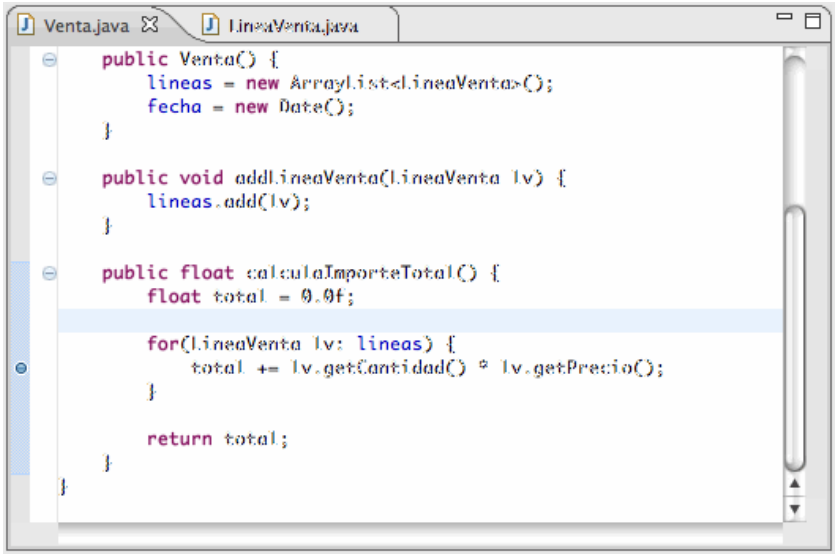
Perspectiva de depuración

En la parte superior izquierda vemos los hilos que se ejecutan, y su estado. Arriba a la derecha vemos los *breakpoints* que establezcamos, y los valores de las variables que entran en juego. Después tenemos el código fuente, para poder establecer marcas y *breakpoints* en él, y debajo la ventana con la salida del proyecto.

2.5.2.2. Establecer breakpoints

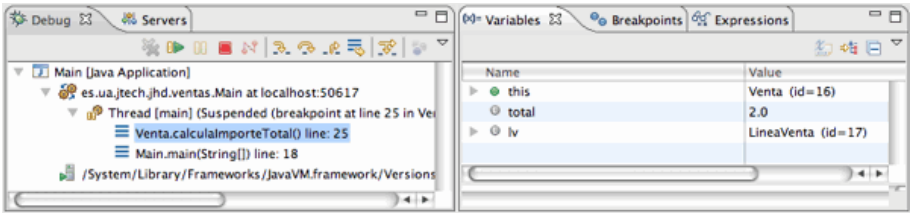
Una de las operaciones más habituales a la hora de depurar es establecer *breakpoints*, puntos en los que la ejecución del programa se detiene para permitir al programador examinar cuál es el estado del mismo en ese momento.

Para establecer *breakpoints*, vamos en la ventana de código hasta la línea que queramos marcar, y hacemos doble click en el margen izquierdo:



Establecer breakpoints

El *breakpoint* se añadirá a la lista de *breakpoints* de la pestaña superior derecha. Una vez hecho esto, re-arrancamos el programa desde **Run - Debug**, seleccionando la configuración deseada y pulsando en **Debug**.

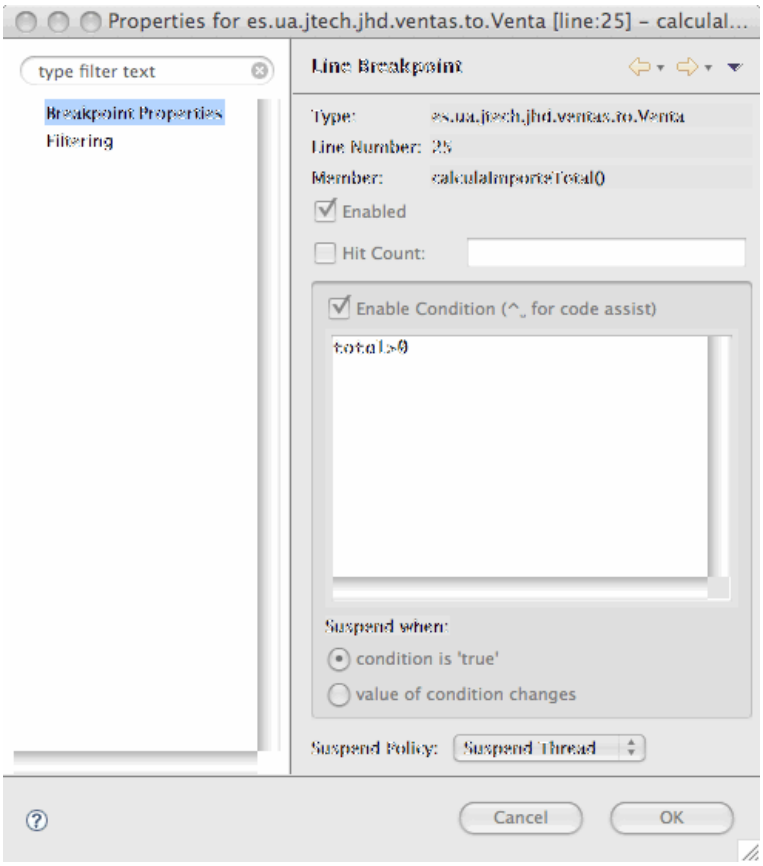


Control del estado del programa

En la parte superior podemos ver el estado de las variables (pestaña *Variables*), y de los hilos de ejecución. Tras cada *breakpoint*, podemos reanudar el programa pulsando el botón de *Resume* (la flecha verde), en la parte superior.

Breakpoints condicionales

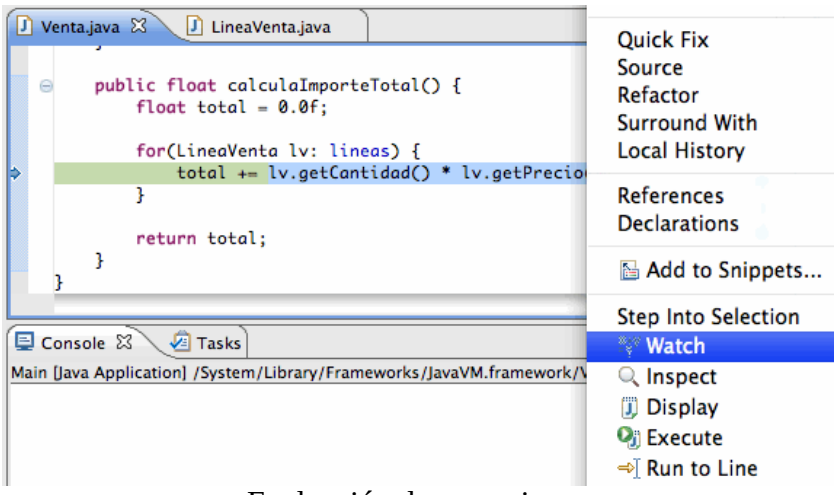
Podemos establecer también *breakpoints* condicionales, que se disparen únicamente cuando el valor de una determinada expresión o variable cambie. Para ello, pulsamos con el botón derecho sobre la marca del *breakpoint* en el código, y elegimos **Breakpoint Properties**. Allí veremos una casilla que indica **Enable Condition**. Basta con marcarla y poner la expresión que queremos verificar. Podremos hacer que se dispare el *breakpoint* cuando la condición sea cierta, o cuando el valor de esa condición cambie:



Breakpoints condicionales

2.5.2.3. Evaluar expresiones

Podemos evaluar una expresión del código si, durante la depuración, seleccionamos la expresión a evaluar, y con el botón derecho elegimos **Inspect**. Si queremos hacer un seguimiento del valor de la expresión durante la ejecución del programa, seleccionaremos **Watch** en lugar de **Inspect**. De esta forma el valor de la expresión se irá actualizando conforme ésta cambie.

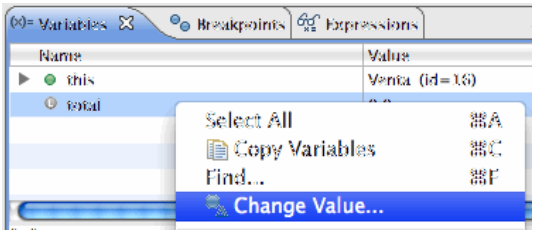


Evaluación de expresiones

2.5.2.4. Explorar variables

Como hemos dicho, en la parte superior derecha, en el cuadro **Variables** podemos ver el valor que tienen las variables en cada momento. Una vez que la depuración alcanza un *breakpoint*, podemos desde el menú **Run** ir a la opción **Step Over** (o pulsar **F6**), para ir ejecutando paso a paso a partir del *breakpoint*, y ver en cada paso qué variables cambian (se ponen en rojo), y qué valores toman.

También podemos, en el cuadro de variables, pulsar con el botón derecho sobre una, y elegir **Change Value**, para cambiar a mano su valor y ver cómo se comporta el programa.



Exploración de variables

2.5.2.5. Cambiar código "en caliente"

Si utilizamos Java 1.4 o superior, desde el depurador de Eclipse podemos, durante la depuración, cambiar el código de nuestra aplicación y seguir depurando. Basta con modificar el código durante una parada por un *breakpoint* o algo similar, y después pulsar en *Resume* para seguir ejecutando.

Esto se debe a que Java 1.4 es compatible con la JPDA (*Java Platform Debugger Architecture)*, que permite modificar código en una aplicación en ejecución. Esto es útil cuando es muy pesado re-arrancar la aplicación, o llegar al punto donde falla.

2.5.2.6. Gestión de Logs

Aunque el debugging con Eclipse parece sencillo y eficaz, no siempre es posible utilizarlo. Algunos casos de difícil depuración son los de aplicaciones distribuidas, aplicaciones multihilo, o aplicaciones en dispositivos embebidos. En cualquier caso utilizar logs para informar de excepciones, advertencias, o simplemente información útil para el programador, es una práctica habitual y recomendable.

Log4Java (log4j) es una librería open source de Jakarta que permite a los desarrolladores de software controlar la salida de los mensajes que generen sus aplicaciones, y hacia dónde direccionarlos, con una cierta granularidad. Es configurable en tiempo de ejecución, lo que permite establecer el tipo de mensajes que queremos mostrar y dónde mostrarlos, sin tener que detener ni recompilar nuestra aplicación.

Log4Java se puede combinar con el envoltorio que proporciona la librería commons-logging, también de Jakarta, para proporcionar todavía mayor flexibilidad.

2.5.3. Otros entornos

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

JavaME SKD 3.0

JavaME SKD 3.0 es un conjunto de herramientas que suceden al Wireless Toolkit 2.5.2. Están disponibles para Mac OS y Windows. Contiene la colección de APIs, un emulador de dispositivos y un entorno de desarrollo con facilidades.

NetBeans

La alternativa a Eclipse, ofreciendo todas las facilidades como completar código, depurar, plugins oficiales, etc.

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.

3. 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.

3.1. Características ausentes en JavaME

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.

3.2. 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.

3.2.1. 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).

3.2.2. 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.

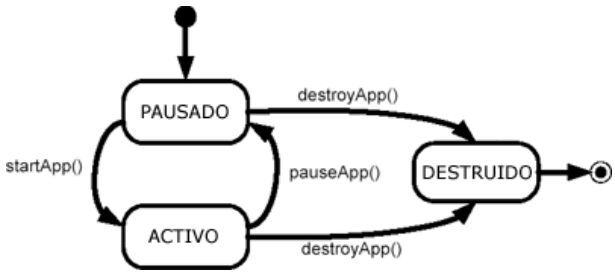
Permanente: Si por el contrario el error es permanente, entonces deberemos destruir el MIDlet llamando a `notifyDestroyed` porque sabemos que nunca podrá ejecutarse correctamente. Si se lanza una excepción de tipo `RuntimeException` dentro del método `startApp` tendremos el mismo efecto, se destruirá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

3.2.3. 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) {
    }

}
```

3.2.4. 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*.

3.2.5. 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.