UNIVERSIDAD DEL AZUAY

Facultad de Administración

Escuela de Ingeniería de Sistemas

"Desarrollo de aplicaciones con J2ME"

Monografía previa a la obtención del

Título de Ingeniería de Sistemas.

Director:

Ing. Pablo Esquivel

Autores:

Susan Doria Heredia Peña

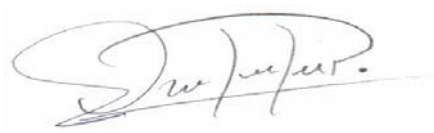
Braulio Rubén Ramírez Orellana

Cuenca – Ecuador

2006

Responsabilidad

El desarrollo de esta monografía representa un esfuerzo personal, todo el contenido, estudios, investigaciones e ideas desarrolladas en este trabajo son de exclusiva responsabilidad de los autores .



Braulio R. Ramírez O.	Susan D. Heredia P.

Agradecimiento

A Dios, porque sin su ayuda y protección nada de esto hubiera sido posible.

A mi Mami, por la confianza que deposito en mi, por enseñarme que todo en la vida se puede lograr con entrega, sacrificio, constancia y humildad, por estar conmigo siempre aunque hayamos estado lejos, por sus oraciones, y por todos sus consejos.

A mis hermanos, Lorena, Mariela y Winston, por todo el cariño y el apoyo que me entregan cada día.

A mis sobrinos Caridad, Junior y Nico, porque con cada una de sus locuras es muy fácil olvidarse de los problemas.

A Caridad, por ser esa lucecita que nunca deja de brillar, por ser el abrazo y el beso que nunca me falta, por su sensibilidad y por enseñarme que con una sonrisa todo es más fácil.

A ti, por ser tan especial por conmigo por haber estado pendiente de mi, por tus palabras de aliento, por tu apoyo, porque contigo todo siempre se veían mejor.

Agradecimiento

Al culminar mi carrera profesional quiero dar gracias a Dios por haberme dado la fuerza y voluntad para lograr cumplir una de mis metas propuestas.

En especial a mis padres y a mis hermanos, Alexandra, Victor, Wimber y Lidy que a pesar de la distancia siempre estuvieron conmigo aconsejándome , brindándome su amor incondicional y apoyo moral y económico que me han dado para realizar uno de mis mayores anhelos.

A mi hermana Alexandra por siempre apoyarme en todo momento en las buenas y en las malas y por ser conmigo como mi segunda madre con sus buenos consejos y gratitud.

A mi hermana Lidy y Anita que siempre estuvo a mi lado alentándome en todo momento y enseñándome que el que persevera siempre alcanza su objetivo gracias por la paciencia y dedicación en todo momento.

A mis amigas y amigos que de una u otra manera supieron brindarme su amistad verdadera.

Resumen

J2ME(Java 2 Micro Edition), es una tecnología derivada de Java, creada para desarrollar aplicaciones dirigidas a dispositivos pequeños con capacidades restringidas tanto en pantalla gráfica, como de procesamiento y memoria, como es el caso de teléfonos celulares, PDA`s, Handhelds, Pagers, etc.

Esta tecnología cuenta con un sinnúmero de elementos que forman parte de Java, pero a su vez cuenta con librerías propias, que son las que marcan su diferencia.

La tecnología J2ME divide a las interfaces de usuario en dos grupos de acuerdo a la aplicación que se desee realizar y para ofrecer una mayor flexibilidad a los desarrolladores y son: La interfaz de alto nivel que usa componentes como formularios y cajas de texto, con esta interfaz perdemos el control del aspecto de la aplicación ya que esta dependería del dispositivo donde se ejecute, pero nos permite tener un alto grado de portabilidad de la aplicación entre distintos dispositivos y son usados en aplicaciones de negocios. La interfaz de bajo nivel en cambio, no permite tener todo el control de la pantalla, esta interfaz es usada para el desarrollo de juegos, aplicaciones en las cuales se requiere control tanto de la pantalla como de las acciones que realiza el usuario.

Abstract

J2ME(Java 2 Micro Edition), it is a derivative technology of Java, created to develop directed applications to small devices with restricted capacities graph, like give processing and memory, for instance cellular telephones, PDA`s, Handhelds, Pagers, etc.

This technology has several elements that are part of Java, but in turn it has own bookstores that mark its difference.

The technology J2ME divides to the interfaces of user in two groups according to the application that is wanted to carry out and to offer a bigger flexibility to the developers and they are: The high-level interface that uses components as forms and text boxes, with this interface we lose the aspect control of the application since this it would depend of the device where it is executed, but it allows us to have a high degree of application portability among different devices and they are used in business applications. On the other hand, the low-level interface, doesn't allow to have the whole screen control, it is used for the games development, since these applications require to have screen control and the control of user action.

CAPITULO I

1. Introducción a java

El éxito del lenguaje de programación Java y de los diversos estándares que orbitan a su alrededor es un hecho indiscutible. Los programadores en Java son los profesionales que tienen más demandados en el área de Desarrollo de Software que se enmarca en la más general disciplina de las Tecnologías de la Información y las Comunicaciones. Tener conocimientos del lenguaje de Java se ha convertido en una necesidad en el ámbito laboral, y esperamos que con el presente documento los lectores vean facilitada su labor en el aprendizaje de tan extenso y completo lenguaje. No en vano el nombre del lenguaje, Java, alude a como se conoce popularmente al café de alta calidad en Estados Unidos.

Java supone un núcleo de sorprendente consistencia teórica en torno al cual giran una multitud de bibliotecas con las más diversas orientaciones: desde bibliotecas dedicadas a la gestión de interfaces de usuario hasta aquéllas que se dedican al tratamiento de imágenes bidimensionales, pasando por las que se centran en difundir información multimedia, envío y recepción de correo electrónico, distribución de componentes software en Internet, etc.

Para algunas de estas bibliotecas no existe una documentación clara y concisa sobre sus características y funcionamiento. En otros casos, la documentación existente es demasiado específica y no proporciona una visión de conjunto que permita apreciar en forma general la filosofía con que fueron creadas dificultando así su difusión. El objetivo de este documento es facilitar la comprensión en la medida de nuestras posibilidades. En este caso trataremos los mecanismos disponibles para desarrollar,

instalar y ejecutar software basado en Java en dispositivos de pequeña capacidad: principalmente teléfonos móviles.

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

1.1. Breve introducción al lenguaje Java.

En este capítulo se tratara de presentar de una manera general a J2ME y encuadrarla dentro de la tecnología Java. También vamos a hacer una breve introducción al lenguaje Java, al menos en sus aspectos básicos para poder adentrarse sin problemas en la programación con J2ME.

1.1.1. Variables y tipos de datos

Las variables nos permiten almacenar información, como su nombre lo indica, pueden variar a lo largo de la ejecución del programa. Antes de poder usar una variable, esta debe ser declararla, es decir, darle un nombre y un tipo.

El nombre de una variable puede ser cualquiera, aunque conviene utilizar nombres claros y relacionados con el uso de la variable. Sólo hemos de tener en cuenta algunas reglas en los nombres de variables:

- No pueden contener espacios en blanco.
- Dos variables no pueden tener el mismo nombre.
- No podemos utilizar palabras reservadas de Java.

Los programadores en Java suelen seguir una serie de convenciones a la hora de nombrar las variables. Esto facilita la lectura de código de terceros.

- Las variables comienzan con una letra minúscula.
- Si la variable está compuesta por dos o más palabras, la segunda (y las siguientes también) comienzan por letra mayúscula. Por ejemplo nombreDeVariable.
- Los nombres de las clases comienzan por letra mayúscula.
- Las variables tienen asociadas un tipo. El tipo de la variable define qué dato es capaz de almacenar. Los tipos de datos válidos en Java son los siguientes:
    - Byte: Ocho bits.
    - Short: Número entero de 16 bits.
    - Int: Número entero de 32 bits.
    - Long: Número entero de 64 bits.
    - Flota: Número en punto flotante de 32 bits.
    - Double: Número en punto flotante de 64 bits.
    - Char: Carácter ASCII.
    - Bolean: Valor verdadero o falso.

Hay que aclarar que los tipos float y double, aún formando parte del standard Java, no están disponibles en J2ME.

Una variable por sí misma no es muy útil, a no ser que podamos realizar operaciones con ellas. Estas operaciones se realizan por medio de operadores. Hay cinco tipos de operadores.

- De asignación
- Aritméticos
- Relacionales
- Lógicos
- A nivel de bit

Cuando declaramos una variable ésta no contiene ningún valor, o mas bien contiene el valor de null. Para asignar un valor a la variable usamos el operador = (signo de igualdad). En Java cada instrucción acaba con un punto y coma. A continuación un ejemplo de este caso.

ejemplo = 3;

En la siguiente tabla podemos apreciar otros operadores de asignación:

Operadores de Asignación	
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b
a &= b	a = a & b
a |= b	a = a | b

Los operadores que vamos a estudiar a continuación son los operadores aritméticos. Tenemos dos tipos, los unarios y los binarios. Los operadores aritméticos unarios son ++ y --. Pueden ir delante o detrás de una variable, y su misión es incrementar o decrementar en una unidad el valor de la variable. Si se sitúan tras la variable hablamos de postincremento o postdecremento), es decir, la variable es incrementada, o bien decrementada, después de haberse hecho uso de ella. Si por el contrario va delante hablamos de preincremento o predecremento, es decir, primero se modifica su valor y después se hace uso de la variable.

Para explicar mejor estos conceptos veamos un ejemplo:

nuevasVidas = ++vidas;

En este ejemplo, primero incrementamos el valor de la variable vidas, y después se lo asignamos a la variable nuevasVidas.

enemigoActual = enemigos--;

En este ejemplo en cambio, primero asignamos a la variable enemigoActual el valor de la variable enemigos, y después decrementamos el valor de esta última variable.

El otro tipo de operadores aritméticos son los binarios.

Operadores Aritméticos	
a + b	Sumas de a y b
a - b	Diferencia de a y b
a * b	Producto de a por b
a / b	Diferencia entre a y b
a % b	Resto de la división entre a y b

Los operadores relacionales nos permiten comparar dos variables o valores y nos devuelve un valor de tipo boolean, es decir, verdadero (true) o falso (false). Entre estos tenemos:

Operadores Relacionales	
a > b	true si a es mayor que b
a < b	true si a es menor que b
a >= b	true si a es mayor o igual que b
a <= b	true si a es menor o igual que b
a = = b	true si a es igual que b
a != b	true si a es distinto a b

Los operadores lógicos nos permiten realizar comprobaciones lógicas del tipo Y, O y NO, que al igual que los operadores relaciones devuelven true o false.

Operadores Lógicos	
a && b	true si a y b son verdaderos
a || b	true si a o b son verdaderos
!a	true si a es false, y false si a es true

Los operadores de bits trabajan a nivel de bits, permite manipularlos directamente.

Operadores de Bits	
a >> b	Desplaza los bits de a hacia la derecha b veces
a << b	Deslaza los bots de a hacia la izquierda b veces
a <<< b	Igual que el anterior pero sin signo
a & b	Suma lógica entre a y b
a | b	O lógico entre a y b
a ^ b	O exclusivo (xor) entre a y b
¬a	Negación lógica de a (not)

1.1.2. Clases y objetos

Explicaremos el concepto de objeto de la forma más intuitiva posible sin entrar en demasiados formalismos. Al pensar en un objeto, nos imaginamos en algo físico y material como un carro o cualquier otra cosa que caiga dentro de nuestro radio de visión, ésta es la idea intuitiva de objeto. Una de las diferencias básicas evidentes es que un objeto en Java puede hacer referencia a algo abstracto.

Como en el ejemplo del carro, un objeto puede estar compuesto por otra clase de objeto, como rueda, carrocería, etc... Un objeto siempre pertenece a una clase de objeto. Por ejemplo, todas las ruedas, con independencia de su tamaño, pertenecen a la clase rueda. Hay muchos objetos “rueda” diferentes que pertenecen a la clase rueda y cada uno de ellos forma una instancia de la clase rueda.

Tenemos, así, instancias de la clase rueda que son ruedas de avión, ruedas de autos o ruedas de motos.

Volvamos al ejemplo del coche. Vamos a definir otra clase de objeto, la clase carro. Esta clase define a “algo” que está compuesto por instancias u objetos de la clase rueda, carrocería, volante, etc...

Ahora vamos a crear un objeto de la clase carro, al que llamaremos carro_rojo. En este caso hemos instanciado un objeto de la clase carro y hemos definido uno de sus atributos, el color, al que hemos dado el valor de rojo. De esta forma vemos que un objeto puede poseer atributos. Sobre el objeto carro podemos definir también acciones u operaciones posibles.

Por ejemplo, el objeto carro, puede realizar las operaciones de acelerar, frenar, girar a la izquierda, etc... Estas operaciones que pueden ser ejecutadas sobre un objeto se llaman métodos del objeto.

Podríamos entonces hacer una primera definición de lo que es un objeto: Es la instancia de una clase de objeto concreta, que está compuesta por atributos y métodos. Esta definición nos muestra una de las tres principales características que definen a la POO. Me refiero al encapsulamiento, que no es otra cosa que la capacidad que tiene un objeto de contener datos (atributos) y código (métodos).

1.1.2.1. Clases y objetos en Java

Antes de poder crear un objeto hay que definirlo. Como decíamos anteriormente, un objeto pertenece a una clase, así que antes de crear nuestro objeto, hay que definir una clase (o utilizar una clase ya definida en las APIs de Java). La forma básica para declarar una clase en Java es.

```
class nombre_clase {
        // variables de la clase (atributos)
        ...
        // métodos de la clase
        }
```

En Java, se usan las dos barras inclinadas (//) para indicar el inicio de un comentario. Una vez definida la clase, podemos ya crear un objeto de la clase que hemos declarado. Lo hacemos así.

```
clase_objeto nombre_objeto;
```

Los métodos, son similares a las funciones de otros lenguajes. La declaración de un método tiene la siguiente forma.

```
tipo NombreMetodo(tipo arg1, tipo arg2, ...) {
// cuerpo del método (código)
}
```

El método tiene un tipo de retorno, y también tiene una lista de argumentos o parámetros.

Vamos a clarificar lo visto hasta ahora con un ejemplo.

```
class Carro {
// variables de clase
int velocidad;
// métodos de la clase
void acelerar(int nuevaVelocidad) {
velocidad = nuevaVelocidad;
}
void frenar() {
velocidad = 0;
}
}
```

Hemos declarado la clase carro, que tiene un sólo atributo, la velocidad, y dos métodos, uno para acelerar y otro para frenar. En el método acelerar, simplemente

recibimos como parámetro una nueva velocidad, y actualizamos este atributo con el nuevo valor. En el caso del método frenar, ponemos la velocidad a 0. Veamos ahora cómo declaramos un objeto de tipo carro y cómo utilizar sus métodos.

```
        // declaración del objeto
        Carro miCarro = new Carro();
    // acelerar hasta 100 km/h
    miCarro.acelerar(100);
    // frenar
    miCarro.frenar();
```

Se ha creado primero el objeto miCarro que pertenece a la clase Carro mediante el operador new. Luego, podemos acceder tanto a los métodos como a las variables miembro usando su nombre precedido de un punto y el nombre del objeto.

Si nuestro método tiene algún tipo de retorno, quiere decir que ha de devolver un valor de dicho tipo. Esto se hace mediante la palabra reservada return.

```
return vidas;
```

Esta línea al final del método devuelve el valor de la variable vidas.

Hay un método especial que se llama constructor, y se llama exactamente igual que la clase a la que pertenece. Cuando creamos un objeto con new, el método constructor es ejecutado de forma automática.

1.1.2.2. Herencia

La herencia biológica se transmite de padres a hijos, nunca al revés. En Java la herencia funciona igual, es decir, en un sólo sentido. Mediante la herencia, una clase hija llamada subclase hereda los atributos y los métodos de su clase padre.

Imaginemos que queremos crear una clase llamada CarroPolicia tomando como referencia el ejemplo anterior, que además de acelerar y frenar pueda activar y

desactivar una sirena. Podríamos crear una clase nueva llamada CarroPolicia con los atributos y clases necesarios tanto para frenar y acelerar como para activar y desactivar la sirena. En lugar de eso, vamos a aprovechar que ya tenemos una clase llamada Carro y que ya contiene algunas de las funcionalidades que queremos incluir en CarroPolicia, por ejemplo.

```
Class CarroPolicia extends Carro {
    // variables
    int sirena;
    // métodos
    void sirenaOn() {
        sirena=1;
    }
    void sirenaOff() {
        sirena=0;
    }
}
```

Lo que llama la atención de esta declaración es la primera línea. Tras el nombre de la clase hemos añadido la palabra extends seguido de la clase padre, es decir, de la cual heredamos los métodos y atributos. La clase CarroPolicia posee dos atributos, velocidad, que ha sido heredado y sirena, que ha sido declarada dentro de la clase CarroPolicia. Con los métodos sucede lo mismo, la clase hija ha heredado acelerar() y frenar(), y se ha añadido los métodos sirenaOn() y sirenaOff(). Un objeto instancia de CarroPolicia puede utilizar sin ningún problema los métodos acelerar() y frenar() tal y como hacíamos con los objetos instanciados de la clase Carro.

1.1.2.3. Polimorfismo

La palabra polimorfismo deriva de poli (múltiples) y del término griego morfos (forma). Es decir, múltiples formas.

Supongamos que queremos dotar al método frenar de más funcionalidad.

Queremos que nos permita reducir hasta la velocidad que queramos. Para ello le pasaremos como parámetro la velocidad, pero también sería útil que frenara completamente si no le pasamos ningún parámetro. El siguiente código cumple estos requisitos.

```
// Declaración de la clase carro
class Carro {
        // Atributos de la clase carro
        int velocidad;
        // Métodos de la clase carro
        void acelerar(int velocidad);
        void frenar() {
                // Ponemos a 0 el valor del atributo velocidad
                velocidad = 0;
        }
        void frenar(int velocidad) {
                // Reducimos la velocidad
                if (velocidad < this.velocidad)
                        this.velocidad = velocidad;
        }
}
```

Tenemos dos métodos frenar, cuando llamemos al método frenar(), Java sabrá cual tiene que ejecutar dependiendo de si lo llamamos con un parámetro de tipo entero o sin parámetros. Esto que hemos hecho se llama sobrecarga de métodos.

Podemos crear tantas versiones diferentes del método siempre y cuando sean diferentes.

El constructor de una clase también puede ser sobrecargado. En el ejemplo, encontramos la palabra reservada this. Ésta se utiliza para indicar que a la variable que nos referimos es la de la clase, y no la que se ha pasado como parámetro. Hay que hacer esta distinción, ya que tienen el mismo nombre.

1.1.3. Estructuras de control

Tenemos las estructuras de control condicionales y repetitivas clásicas de la programación estructurada.

La estructura de control más básica es if/else, que tiene la siguiente forma:

```
if (condición) {
    sentencias;
} else {
    sentencias;
}
```

Mediante esta estructura condicional, podemos ejecutar un código u otro dependiendo de si se cumple una condición concreta. La segunda parte de la estructura (else) es opcional.

La otra estructura condicional es switch, que permite un control condicional múltiple. Tiene el siguiente formato.

```
    switch (expresión) {
        case val1:
```

```
            sentencias;
            break;
        case val2:
            sentencias;
            break;
        case valN:
            sentencias;
            break;
        default:
            sentencias;
            break;
    }
```

Dependiendo del valor que tome la expresión, se ejecutará un código determinado por la palabra reservada case. Observe como al final de las sentencias se incluye la palabra reservada break, que hace que no se siga ejecutando el código perteneciente al siguiente bloque. Si el valor de la expresión no coincide con ninguno de los bloques, se ejecuta el bloque default.

Las estructuras vistas hasta ahora nos permiten tomar decisiones, pero las estructuras que veremos a continuación nos permitirán realizar acciones repetitivas, son los llamados bucles. El bucle más sencillo es el bucle for.

```
    for (inicialización_contador ; control ; incremento) {
        sentencias;
    }
```

Este bucle ejecuta el bloque de sentencias un número determinado de veces.

El siguiente bucle es el while y tiene la siguiente estructura.

```
while (condición) {
        sentencias;
}
```

El bloque de sentencias se ejecutará mientras se cumpla la condición del bucle.

El bucle do/while funciona de forma similar al anterior, pero hace la comprobación a la salida del bucle, y tiene la siguiente estructura.

```
do {
        sentencias;
} while (condición);
```

Veamos una última estructura propia de Java que nos permite ejecutar un código de forma controlada. Concretamente nos permite tomar acciones específicas en caso de error de ejecución en el código, y tiene la siguiente estructura.

```
try {
        sentencias;
} catch (excepción) {
        sentencias;
}
```

Si el código incluido en el primer bloque de código produce algún tipo de excepción, se ejecutará el código contenido en el segundo bloque de código. Una excepción es un tipo de error que Java es capaz de controlar, una excepción es un objeto de la clase Exception. Si por ejemplo, dentro del primer bloque de código intentamos leer un archivo, y no se encuentra en la carpeta especificada, el método encargado de abrir el archivo lanzará una excepción del tipo IOException.

1.1.4. Estructuras de datos

A continuación veremos las estructuras la cadena de caracteres y los arrays.

Una cadena de caracteres es una sucesión de caracteres continuos. Van encerrados siempre entre comillas. Por ejemplo:

"Esto es solo una frase para ejemplo..."

Para almacenar una cadena, Java dispone del tipo String.

```
String texto;
```

Una vez declarada la variable, para asignarle un valor, lo hacemos de la forma habitual.

```
texto = "Esto es solo una frase ejemplo";
```

Podemos concatenar dos cadenas utilizando el operador +. También podemos concatenar una cadena y un tipo de datos distinto. La conversión a cadena se hace de forma automática.

```
String texto;
    int vidas;
    texto = "Vidas:" + vidas;
```

Podemos conocer la longitud de una variable de tipo String haciendo uso de su método lenght.

```
longitud = texto.lenght();
```

Otro tipo de datos es el array, que nos permite almacenar varios elementos de un mismo tipo bajo el mismo nombre. Imaginemos un juego multijugador en el que pueden participar cinco jugadores a la vez. Cada uno llevará su propio contador de vidas. Mediante un array de 5 elementos de tipo entero (int) podemos almacenar estos datos. La declaración de un array se hace así.

```
public int[] vidas;
```

```
vidas = new int[5];
```

o directamente:

```
public int[] vidas = new int[5];
```

Hemos declarado un array de cinco elementos llamado vidas formado por cinco números enteros. Si quisiéramos acceder, por ejemplo, al tercer elemento del array, lo haríamos de la siguiente manera.

```
v = vidas[3];
```

La variable v tomará el valor del tercer elemento del array. La asignación de un valor es exactamente igual a la de cualquier variable.

```
vidas[3] = 1;
```

El siguiente ejemplo muestra el uso de los arrays.

```
tmp = 0;
for (i=1 ; i<= puntos.lenght ; i++) {
        if (tmp < puntos[i]) {
                tmp = puntos[i];
        }
}
record = tmp;
```

En este ejemplo, el bucle for recorre todos los elementos del array puntos que contiene los puntos de cada jugador (el método lenght nos devuelve el número de elementos el array).

Al finalizar el bucle, la variable tmp contendrá el valor de la puntuación más alta.

CAPITULO II

2. Introduccion a J2ME

El lenguaje de programación Java fue lanzado por la empresa Sun Microsystems a mediados de los años 90, en un principio fue diseñado para generar aplicaciones que controlaran electrodomésticos como lavadoras, frigoríficos, etc, pero gracias a su gran robustez e independencia de la plataforma donde se ejecutase el código, se utilizó para la creación de componentes interactivos integrados en páginas Web y programación de aplicaciones independientes. Estos componentes se denominaron applets y casi todo el trabajo de los programadores se dedicó al desarrollo de éstos.

Con el paso de los años, Java ha progresado enormemente en varios ámbitos como servicios HTTP, servidores de aplicaciones, acceso a bases de datos, etc.

Debido a las actuales explosiones tecnológicas, Java ha desarrollado soluciones personalizadas para cada ámbito tecnológico. Sun ha agrupado cada uno de esos ámbitos en una edición distinta de su lenguaje Java.

Estas ediciones son:

- Java 2 Stantard Edition(J2SE): orientada al desarrollo de aplicaciones independientes y de applets.
- Java 2 Enterprise Edition(J2EE): enfocada al entorno empresarial.
- Java 2 Micro Edition(J2ME): dirigida a la programación de aplicaciones para pequeños dispositivos.

En esta última edición de Java es en la que centraremos el presente documento.

Los teléfonos móviles se han convertido en una parte importante en nuestra vida cotidiana, cada vez son más los usuarios de estos dispositivos, y estos cada vez más

pequeños. Los teléfonos celulares nos han acompañado a todas partes y nos han permitido comunicarnos con cualquier otro terminal fijo o móvil. Aunque la comunicación telefónica por voz es el principal uso de estos terminales, pronto se han desarrollado nuevas formas de comunicación y otras capacidades en torno a estos pequeños teléfonos.

El primero, y quizás más lucrativo hasta la fecha, fue el uso de la mensajería SMS

(Short Message Service), luego aparecieron los terminales capaces de navegar por Internet, pero las limitaciones de la pantalla y de los teclados hacían inviable su uso con páginas web normales. Así nació la tecnología WAP, que nos permiten navegar por páginas especiales creadas en WML en lugar de en HTML. Sin embargo, las limitaciones de este medio, y quizás también por el elevado precio y la baja velocidad del servicio, han hecho que la tecnología WAP no se haya extendido tanto como su hermana mayor, la WEB.

Para combatir las bajas velocidades, sin contar con la baja fiabilidad de la tecnología GSM para la transmisión de datos, apareció la tecnología GPRS (General Packet Radio Service). GPRS nos ofrece una red de transferencia de datos sin hilos a una velocidad aceptable, permitiéndonos enviar grandes paquetes de información, como fotografías, música, e incluso video. De ahí, se hace patente la necesidad de una nueva generación de móviles capaces de reproducir músicas más complejas y mostrar gráficos y fotografías en color. A la vez que aparecen estos móviles en el mercado, aparece el nuevo servicio de mensajes cortos llamado MMS (Multimedia Message Service), que nos permite enviar, no solo texto, sino también fotografías, sonidos, gráficos, etc. Pero aún estaba por llegar la verdadera revolución.

En 1999 Sun Mycrosystems da un paso adelante dentro de su tecnología Java, y nos presenta J2ME (Java 2 Micro Edition): un entorno de producción para pequeños

dispositivos que permite la ejecución de programas creados en Java. Java Micro Edition es una versión orientada al desarrollo de aplicaciones para dispositivos pequeños con capacidades restringidas tanto en pantalla gráfica, como de procesamiento y memoria (teléfonos móviles, PDA`s, Handhelds, Pagers, etc), dando así un paso adelante dentro de su tecnología Java. Una de las principales capacidades que añade esta tecnología a nuestros terminales es la posibilidad de descargar y ejecutar juegos con una calidad razonable. Los teléfonos móviles actuales corren auténticos sistemas operativos. El más conocido quizás es Symbian, que es el corazón de gran cantidad de móviles, como los Nokia, Sony-Ericsson, Motorola y otros.

La aparición tardía de esta tecnología puede ser debido a que las necesidades de los usuarios de telefonía móvil han cambiado en estos últimos años y existe más demanda de servicios y prestaciones por parte tanto de los terminales como de las compañías. J2ME es la tecnología del futuro para la industria de los dispositivos móviles. Actualmente las compañías telefónicas y los fabricantes de móviles están implantando los protocolos y dispositivos necesarios para soportarla.

Este documento tiene como objetivos exponer los pilares sobre los que se asienta la plataforma J2ME y las clases que componen esta edición de Java para poder realizar rápidamente aplicaciones, especialmente juegos para teléfonos que dispongan de esta tecnología. Para la realización de dichas aplicaciones usaremos software de libre como es el caso de Netbeans 5.0. Usaremos también un emulador para ejecutar y depurar nuestros programas, evitando así la descarga insatisfactoria de nuestra aplicación en un verdadero terminal.

2.1. Análisis comparativo

Como indicamos anteriormente Sun, dispuesto a proporcionar las herramientas necesarias para cubrir las necesidades de todos los usuarios, creó distintas versiones de Java de acuerdo a sus necesidades, a continuación presentamos a cada una de ella describiendo el campo al cual están dirigidas.

- **Java 2 Platform, Standard Edition (J2SE)**: Contiene el conjunto básico de herramientas usadas para desarrollar Java Applets, así cómo las APIs orientadas a la programación de aplicaciones de usuario final: Interfaz gráfica de usuario, multimedia, redes de comunicación, etc.

- **Java 2 Platform, Enterprise Edition (J2EE)**: Orientada al entorno empresarial. El software empresarial está pensado no para ser ejecutado en un equipo, sino para ejecutarse sobre una red de ordenadores de manera distribuida y remota mediante EJBs (Enterprise Java Beans). Esta edición está orientada especialmente al desarrollo de servicios web, servicios de nombres, persistencia de objetos, XML, autenticación, APIs para la gestión de transacciones, etc.

- **Java 2 Platform, Micro Edition (J2ME)**: Enfocada a la aplicación de la tecnología Java en dispositivos electrónicos con capacidades computacionales y gráficas muy reducidas, tales como teléfonos móviles, PDAs o electrodomésticos inteligentes. Esta edición tiene unos componentes básicos que la diferencian de las otras versiones, como el uso de una máquina virtual denominada KVM (Kilo Virtual Machine, debido a que requiere sólo unos pocos Kilobytes de memoria para funcionar) en vez del uso de la

JVM clásica, inclusión de un pequeño y rápido recolector de basura y otras diferencias que ya iremos viendo más adelante.

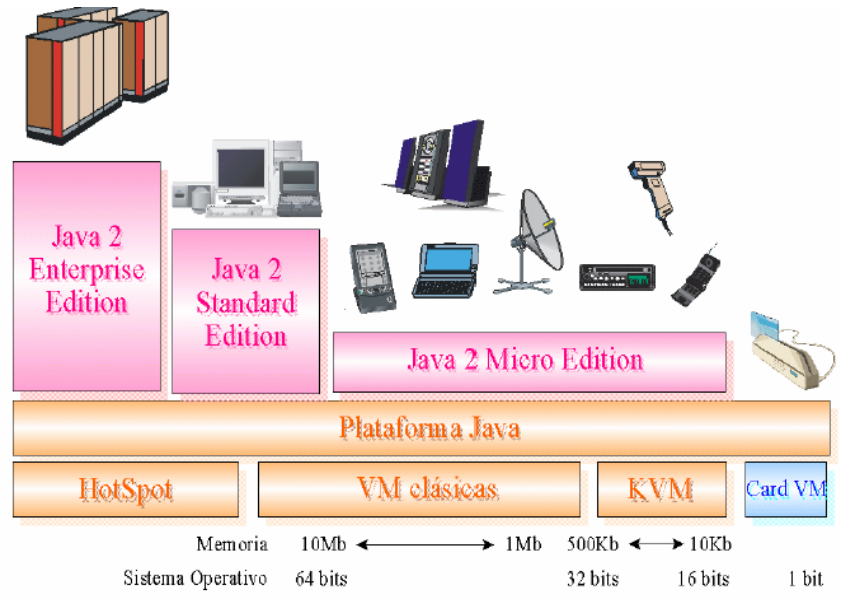
Java es un conjunto de tecnologías que abarca a todos los ámbitos de la computación con dos elementos en común:

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

J2ME contiene una mínima parte de las APIs de Java, debido a que la edición estándar de APIs de Java ocupa 20 Mb, y los dispositivos pequeños disponen de una cantidad de memoria mucho más reducida. En concreto, J2ME usa 37 clases de la plataforma J2SE provenientes de los paquetes java.lang, java.io, java.util. Esta parte de la API que se mantiene fija forma parte de lo que se denomina “configuración”. Otras diferencias con la plataforma J2SE vienen dadas por el uso de una máquina virtual distinta de la clásica JVM denominada KVM. Esta KVM tiene unas restricciones que hacen que no posea todas las capacidades incluidas en la JVM.

Como vemos, J2ME representa una versión simplificada de J2SE. Sun separó estas dos versiones ya que J2ME estaba pensada para dispositivos con limitaciones de proceso y capacidad gráfica. También separó J2SE de J2EE porque este último exigía unas características muy pesadas o especializadas de E/S, trabajo en red, etc.

Por tanto, separó ambos productos por razones de eficiencia. Hoy, J2EE es un superconjunto de J2SE pues contiene toda la funcionalidad de éste y más características, así como J2ME es un subconjunto de J2SE (excepto por el paquete javax.microedition) ya que, como se ha mencionado, contiene varias limitaciones con respecto a J2SE.



Arquitectura de la Plataforma Java 2 de Sun

2.2. Nociones básicas de J2ME

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

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

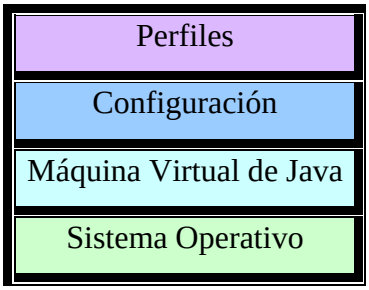
Un entorno de ejecución determinado de J2ME se compone entonces de una selección de:

a) Máquina virtual.

b) Configuración.

c) Perfil.

d) Paquetes Opcionales.



2.2.1. Máquinas Virtuales J2ME

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

De esta forma, la JVM proporciona al programa Java independencia de la plataforma con respecto al hardware y al sistema operativo subyacente. Las JVM son, muy pesadas en cuanto a memoria ocupada y requerimientos computacionales.

Ya hemos visto que existen 2 configuraciones CLDC y CDC, cada una requiere su propia máquina virtual. La VM (Virtual Machine) de la configuración CLDC se denomina KVM y la de la configuración CDC se denomina CVM.

KVM

Su nombre KVM proviene de Kilobyte (haciendo referencia a la baja ocupación de memoria. Se trata de una implementación de Máquina Virtual reducida y especialmente

orientada a dispositivos con bajas capacidades computacionales y de memoria. La KVM está escrita en lenguaje C, y fue diseñada para ser:

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

Sin embargo, posee algunas limitaciones con respecto a la clásica Java Virtual Machine (JVM):

1. No hay soporte para tipos en coma flotante. No existen por tanto los tipos double ni float. Esta limitación está presente porque los dispositivos carecen del hardware necesario para estas operaciones.
2. No existe soporte para JNI (Java Native Interface) debido a los recursos limitados de memoria.
3. No existen cargadores de clases (class loaders) definidos por el usuario. Sólo existen los predefinidos.
4. No se permiten los grupos de hilos o hilos daemon. Cuando queramos utilizar grupos de hilos utilizaremos los objetos Colección para almacenar cada hilo en el ámbito de la aplicación.
5. No existe la finalización de instancias de clases. No existe el método Object.finalize().
6. No hay referencias débiles.

7. Limitada capacidad para el manejo de excepciones debido a que el manejo de éstas depende en gran parte de las APIs de cada dispositivo por lo que son éstos los que controlan la mayoría de las excepciones.

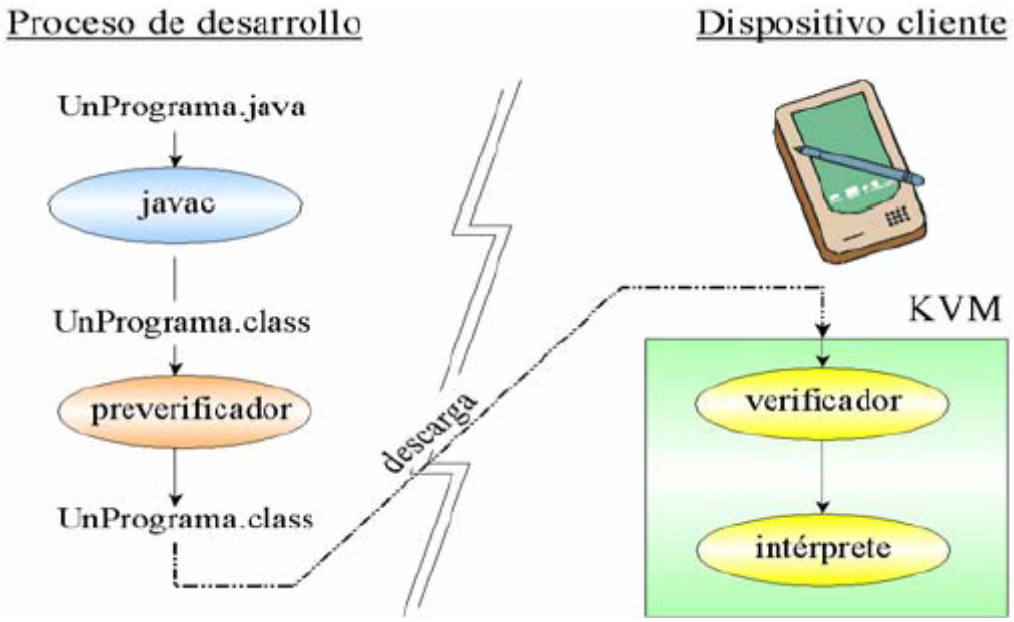
8. Reflexión.

El verificador de clases estándar de Java es demasiado grande para la KVM. De hecho es más grande que la propia KVM y el consumo de memoria es más de 100Kb para las aplicaciones típicas.

Este verificador de clases es el encargado de rechazar las clases no válidas en tiempo de ejecución. Este mecanismo verifica los *bytecodes* de las clases Java realizando las siguientes comprobaciones:

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

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



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

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

CVM

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

1. Sistema de memoria avanzado.
2. Tiempo de espera bajo para el recolector de basura.
3. Separación completa de la VM del sistema de memoria.
4. Recolector de basura Modularizado.
5. Portabilidad.
6. Rápida sincronización.

7. Ejecución de las clases Java fuera de la memoria de sólo lectura (ROM).
8. Soporte nativo de hilos.
9. Baja ocupación en memoria de las clases.
10. Proporciona soporte e interfaces para servicios en Sistemas Operativos de Tiempo Real.
11. Conversión de hilos Java a hilos nativos.
12. Soporte para todas las características de Java2 v1.3 y librerías de seguridad, referencias débiles, Interfaz Nativa de Java (JNI), invocación remota de métodos (RMI), Interfaz de depuración de la Máquina Virtual (JVMDI).

2.2.2. Configuraciones

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

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

Como ya hemos visto con anterioridad, existen dos configuraciones en J2ME:

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

La CDC está orientada a dispositivos con cierta capacidad computacional y de memoria. Por ejemplo, decodificadores de televisión digital, televisores con internet, algunos electrodomésticos y sistemas de navegación en automóviles. CDC usa una Máquina Virtual Java similar en sus características a una de J2SE, pero con limitaciones en el apartado gráfico y de memoria del dispositivo. Ésta Máquina Virtual es la que

hemos visto como CVM (Compact Virtual Machine). La CDC está enfocada a dispositivos con las siguientes capacidades:

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

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

La Tabla nos muestra las librerías incluidas en la CDC:

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

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

La CLDC está orientada a dispositivos dotados de conexión y con limitaciones en cuanto a capacidad gráfica, cómputo y memoria. Un ejemplo de estos dispositivos son: teléfonos móviles, buscapersonas (pagers), PDAs, organizadores personales, etc.

Ya hemos dicho que CLDC está orientado a dispositivos con ciertas restricciones que vienen dadas por el uso de la KVM, necesaria al trabajar con la CLDC debido a su pequeño tamaño. Los dispositivos que usan CLDC deben cumplir los siguientes requisitos:

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

La CLDC aporta las siguientes funcionalidades a los dispositivos:

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

Las librerías incluidas en la CLDC son:

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

2.2.3. Perfiles

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

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

Ya hemos visto los conceptos necesarios para entender cómo es un entorno de ejecución en Java Micro Edition. Anteriormente vimos que para una configuración

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

Para la configuración CDC tenemos los siguientes perfiles:

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

Para la configuración CLDC tenemos los siguientes:

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

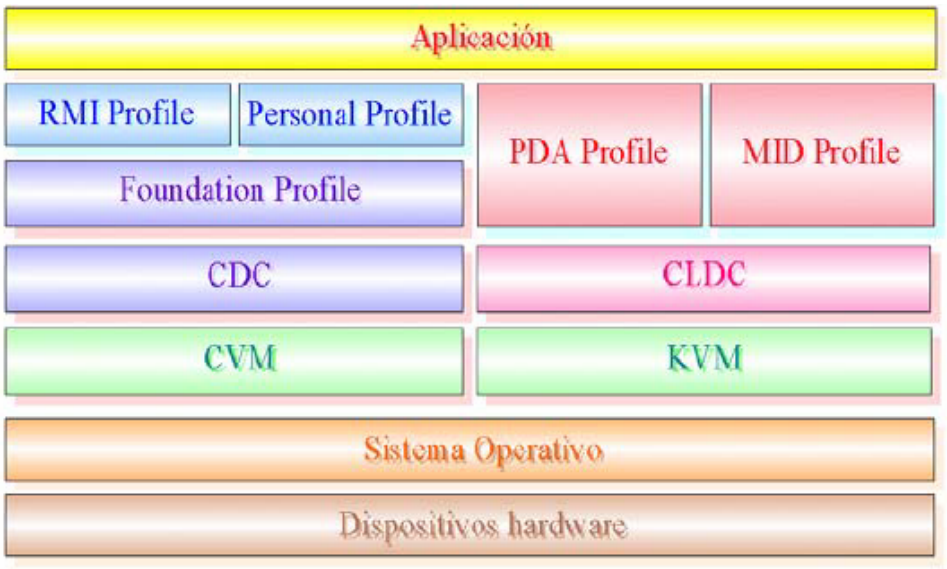


Figura 1.5 Arquitectura del entorno de ejecución de J2ME.

A continuación vamos a ver más detalladamente cada uno de estos perfiles:

Foundation Profile: Este perfil define una serie de APIs sobre la CDC orientadas a dispositivos que carecen de interfaz gráfica como, por ejemplo, decodificadores de televisión digital. Este perfil incluye gran parte de los paquetes de la J2SE, pero excluye totalmente los paquetes “java.awt” *Abstract Windows Toolkit* (AWT) y “java.swing”

que conforman la interfaz gráfica de usuario (GUI) de J2SE. Si una aplicación requiriera una GUI, entonces sería necesario un perfil adicional. Los paquetes que forman parte del *Foundation Profile* son:

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

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

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

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

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

- Java.rmi.server.disableHTTP.
- Java.rmi.activation.port.
- Java.rmi.loader.packagePrefix.
- Java.rmi.registry.packagePrefix.
- Java.rmi.server.packagePrefix.

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

Mobile Information Device Profile (MIDP): Este perfil está construido sobre la configuración CLDC. Al igual que CLDC fue la primera configuración definida para J2ME, MIDP fue el primer perfil definido para esta plataforma.

Este perfil está orientado para dispositivos con las siguientes características:

- Reducida capacidad computacional y de memoria.
- Conectividad limitada (en torno a 9600 bps).
- Capacidad gráfica muy reducida (mínimo un display de 96x54 pixels monocromo).
- Entrada de datos alfanumérica reducida.
- Mecanismos de entrada: “One handed keyboard” (teléfonos celulares), “Two handed keyboards” (pagers) o dispositivo puntero (PDAs).
- 128 Kb de memoria no volátil para componentes MIDP.
- 8 Kb de memoria no volátil para datos persistentes de aplicaciones.
- 32 Kb de memoria volátil en tiempo de ejecución para la pila Java.

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

El perfil MIDP establece las capacidades del dispositivo, por lo tanto, especifica las APIs relacionadas con:

- La aplicación (semántica y control de la aplicación MIDP).
- Interfaz de usuario.
- Almacenamiento persistente.
- Trabajo en red.
- Temporizadores. Es un mecanismo que proporcione una base de tiempo para timestamp (sello de tiempo) sobre los registros presentes en memoria.

Librerías del perfil MIDP.

MIDP, ha definido las siguientes librerías de clases para cumplir con las especificaciones mencionadas anteriormente:

- javax.microedition.midlet: Paquete para gestionar el ciclo de vida de la aplicación
- javax.microedition.lcdui: Paquete de interfaz de usuario.
- javax.microedition.rms: Paquete para gestionar almacenamiento persistente.
- javax.microedition.io: Paquete para networking.
- Javax.io: Clases e interfaces de E/S básica.
- java.lang y java.util: Un subconjunto de los paquetes J2SE para diferentes utilidades.

Las aplicaciones realizadas con MIDP reciben el nombre de *MIDlets* (por simpatía con *APPlets*).

2.3. J2ME y las comunicaciones

Ya hemos visto que una característica importante que tienen que tener los dispositivos que hagan uso de J2ME, más específicamente CLDC/MIDP es que necesitan poseer conexión a algún tipo de red, por lo que la comunicación de estos dispositivos cobra una gran importancia. Ahora vamos a ver cómo participan las distintas tecnologías en estos dispositivos y cómo influyen en el uso de la tecnología J2ME. Para ello vamos a centrarnos en un dispositivo en especial: los teléfonos móviles. Nuestro estudio se centra en aplicaciones para este dispositivo debido a la rápida evolución que han tenido los teléfonos móviles en el sector de las comunicaciones, lo que ha facilitado el desarrollo, por parte de algunas empresas, de herramientas que usaremos para crear las aplicaciones.

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

tratado de dotar a los dispositivos móviles de un pequeño y limitado navegador web. WAP exige la presencia de una puerta de enlace encargado de actuar cómo intermediario entre Internet y el terminal. Esta puerta de enlace o *gateway* es la que se encarga de convertir las peticiones WAP a peticiones web habituales y viceversa.

Las páginas que se transfieren en una petición usando WAP no están escritas en HTML, sino que están escritas en WML, un subconjunto de éste. WAP ha sido un gran avance, pero no ha resultado ser la herramienta que se prometía. La navegación es muy engorrosa (la introducción de URLs largas por teclado es muy pesada, además de que cualquier error en su introducción requiere que se vuelva a escribir la dirección completa por el teclado del móvil). Además su coste es bastante elevado ya que el pago de uso de esta tecnología se realiza en base al tiempo de conexión a una velocidad, que no es muy buena.

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

Los últimos avances de telefonía móvil nos llevan a las conocidas como generación 2 y 2´5 que hacen uso de las tecnologías GSM y GPRS respectivamente.

GSM es una conexión telefónica que soporta una circulación de datos, mientras que GPRS es estrictamente una red de datos que mantiene una conexión abierta en la que el usuario paga por la cantidad de información intercambiada y no por el tiempo que permanezca conectado. La aparición de la tecnología GPRS no hace más que favorecer el uso de J2ME y es, además, uno de los pilares sobre los que se asienta J2ME, ya que podemos decir que es el vehículo sobre el que circularán las futuras aplicaciones J2ME.

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

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

2.4. OTA

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

2.4.1. Requerimientos Funcionales

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

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

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

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

2.4.2. Localización de la Aplicación

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

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

Los atributos del JAD deben ser comprensibles, acorde con la sintaxis de la especificación MIDP, y todos los atributos requeridos por la especificación MIDP deben estar presentes en el JAD.

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

2.4.3. Instalación de MIDlets

La instalación de la aplicación es el proceso por el cual el *MIDlet* es descargado al dispositivo y puede ser utilizado por el usuario. Cuando existan múltiples *MIDlets* en la aplicación que deseamos descargar, el usuario debe ser avisado de que existen más de uno.

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

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

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

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

- Si no existe suficiente memoria para almacenar el *MIDlet*, el dispositivo debe retornar el Código de Estado (*Status Code*) 901.

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

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

2.4.4. Actualización de MIDlets

La actualización se realiza cuando instalamos un *MIDlet* sobre un dispositivo que ya contenía una versión anterior de éste. El dispositivo debe ser capaz de informar al usuario cual es la versión de la aplicación que tiene instalada. Cuando comienza la actualización, el dispositivo debe informar si la versión que va a instalar es más nueva, más vieja o la misma de la ya instalada y debe obtener verificación por parte del usuario antes de continuar con el proceso.

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

2.4.5. Ejecución de MIDlets

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

2.4.6. Eliminación de MIDlets

Los dispositivos deben permitir al usuario eliminar *MIDlets*. Antes de eliminar una aplicación el usuario debe dar su confirmación. El dispositivo debería avisar al usuario si ocurriese alguna circunstancia especial durante la eliminación del *MIDlet*.

Por ejemplo, el *MIDlet* a borrar podría contener a otros *MIDlets*, y el usuario debería de ser alertado ya que todos ellos quedarían eliminados.

CAPITULO III

3. Herramientas de desarrollo

3.1. Descripción del capítulo

- En este capítulo veremos una herramienta que nos permitirá construir nuestros *MIDlets*.

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

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

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

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

6. **Depuración:** Esta última fase nos permitirá depurar los fallos detectados en la fase anterior de nuestro *MIDlet*.

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

3.2. Instalación de componentes

La herramienta que utilizaremos para el desarrollo de *MIDlets* será las APIs de la configuración CLDC y del perfil MIDP que pueden ser descargadas desde http://java.sun.com/j2me/download.html.

Para la instalación de la herramienta necesitamos en primer lugar los instaladores de la aplicación que son los siguientes tres archivos:

- jdk-1_5_0_06-windows-i586-p.exe
- jdk-1_5_0_06-nb-5_0-win.exe
- netbeans_mobility-5_0-win.exe

Lo primero que haremos es instalar el archivo jdk-1_5_0_06-windows-i586-p haciendo doble click sobre él y a continuación seguiremos las instrucciones que aparecen en pantalla dando click en el botón *Next* y esperamos hasta que el proceso de instalación finalice.

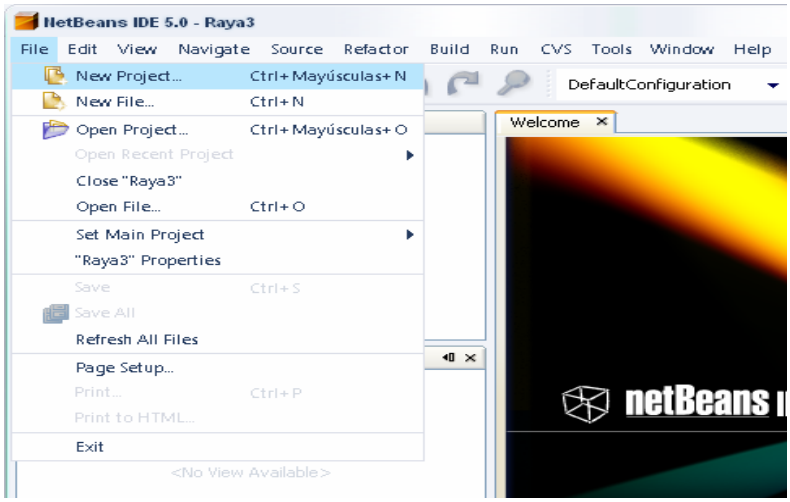
Como segundo paso instalaremos el archivo jdk-1_5_0_06-nb-5_0-win haciendo doble click en él y siguiendo las instrucciones que aparecerán en pantalla, presionaremos el botón *Next* y esperar hasta que el proceso de instalación finalice.

Hasta este punto hemos instalado ya en nuestro computador el NetBeans, pero aun falta de ejecutar el último archivo que es el paquete para desarrollar las aplicaciones para dispositivos móviles.

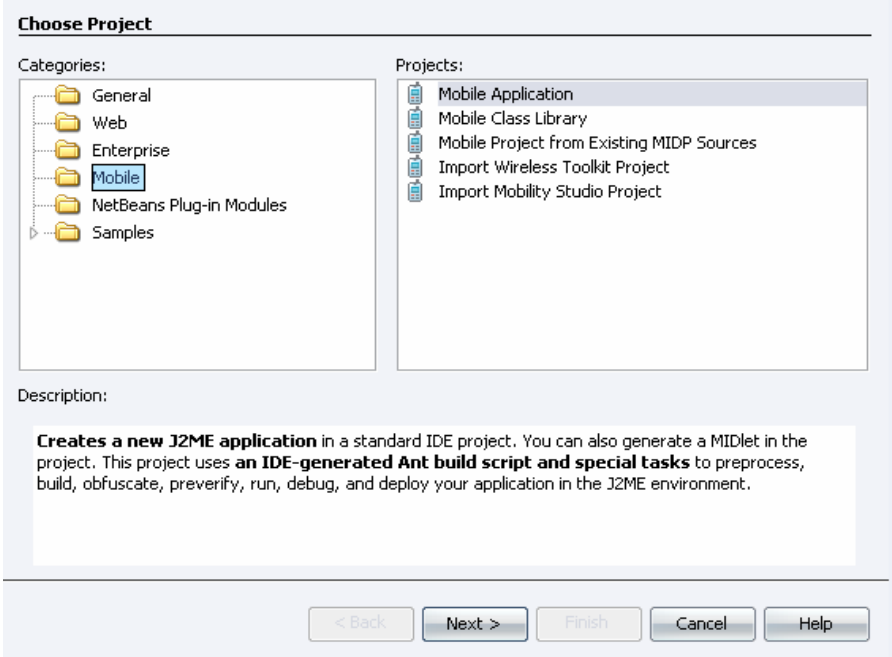
Ejecutamos entonces el archivo netbeans_mobility-5_0-win y seguimos las instrucciones que aparecerán en pantalla presionando el botón *Next* hasta que finalice el proceso de instalación.

Y ahora tenemos todos los componentes instalados para poder empezar a trabajar, ejecutando el programa NetBeans 5.0.

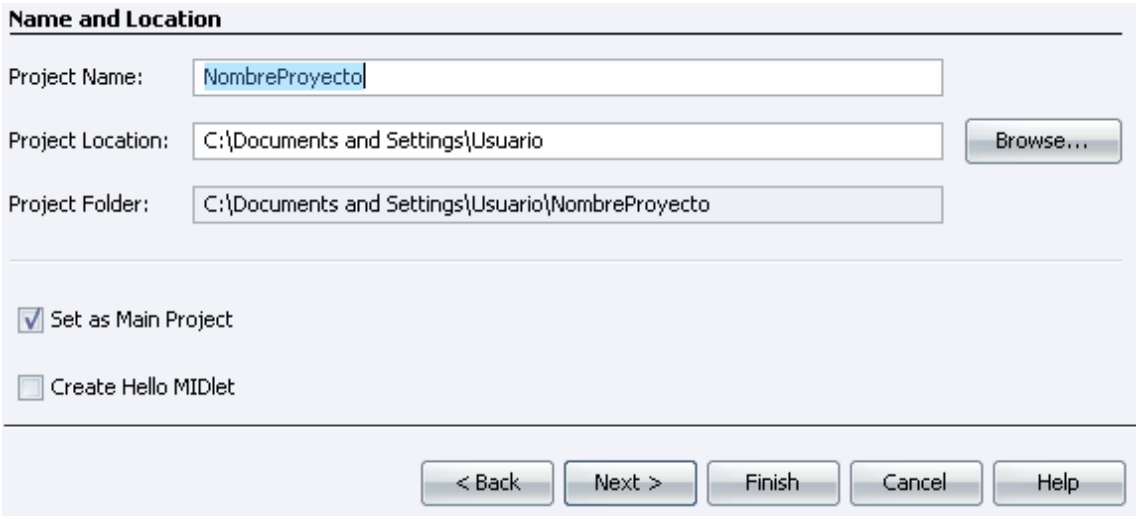
Para la creación de un nuevo proyecto debemos hacerlo a través del menú *File* y luego en *New Project.*



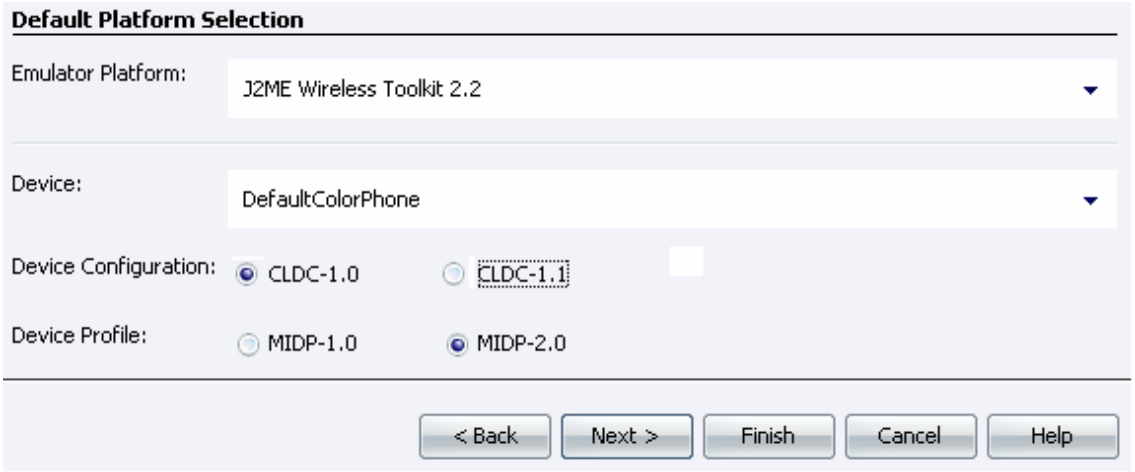
A continuación, en la pantalla que nos aparece escogemos las opciones de categoría *Mobile* y dentro de Project elegimos *Mobile Application* y damos un click en *Next*.



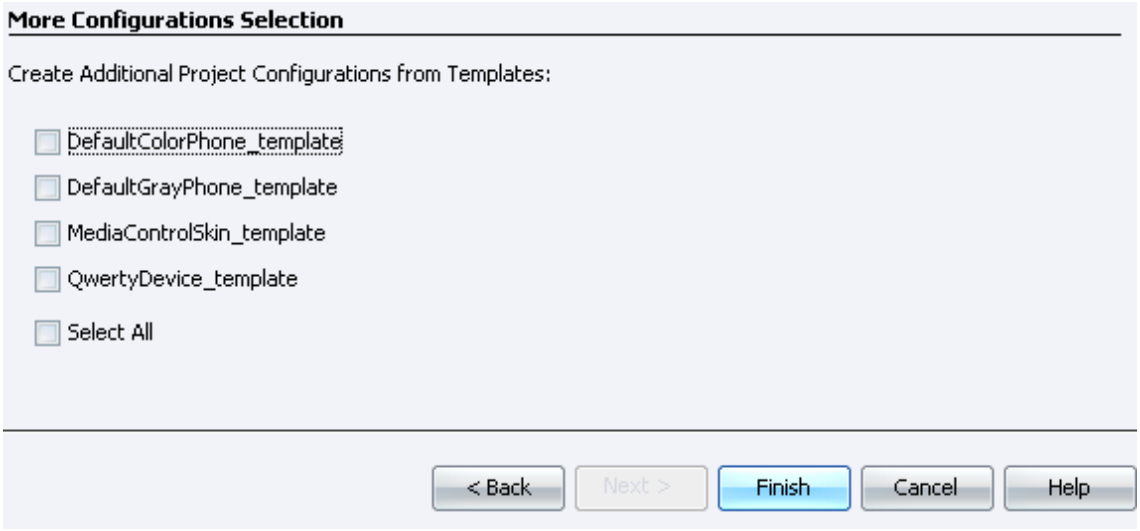
Luego en la siguiente pantalla procedemos a darle nombre a nuestra aplicación y damos click en *Next*.



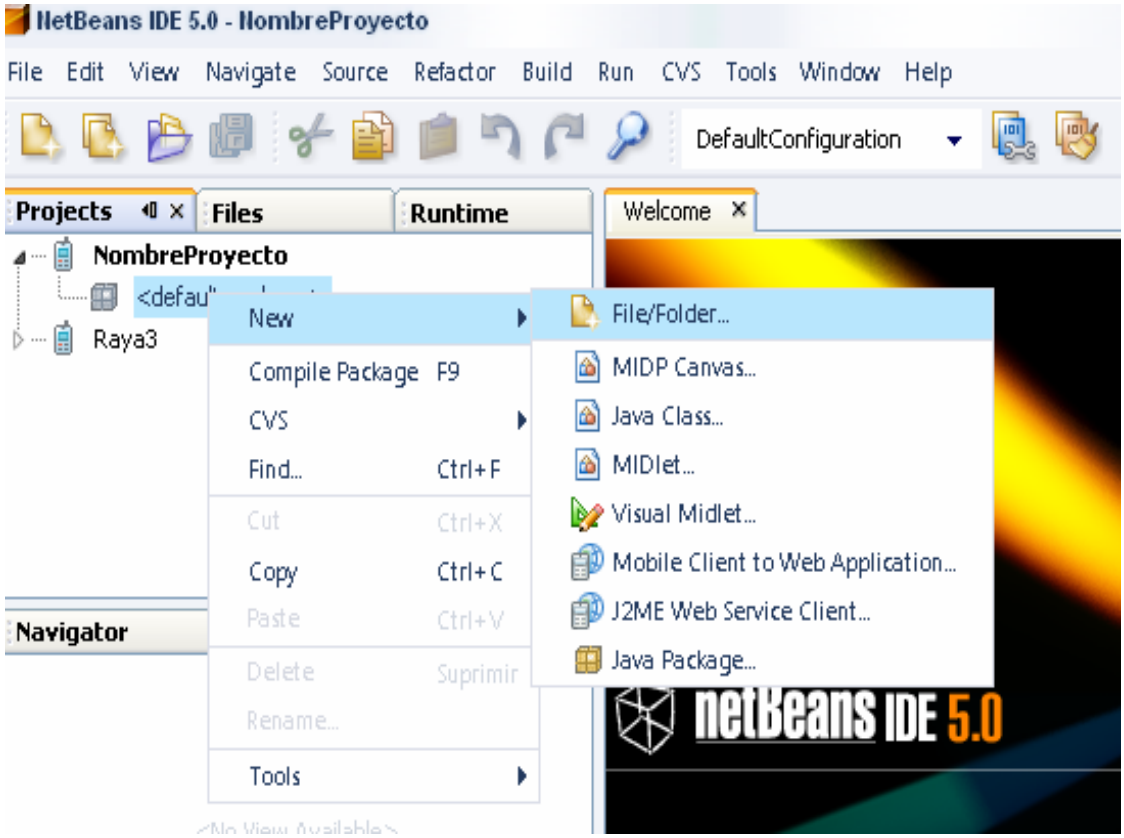
El siguiente paso es el de seleccionar el emulador, dispositivo, configuración y el profile, los cuales serán seleccionados tal como se muestra en este ejemplo.



Y damos click en el boton *Finish* en la siguiente pantalla, dejando sin seleccionar todas las opciones.



En este momento ya podemos proceder a la creación de archivos para el proyecto, para lo cual damos un click con el botón derecho en *Default package* del proyecto que hemos creado, seguido de esto seleccionamos *New* y luego el tipo de archivo que deseamos crear, ya sea un MIDlet, una clase o un Canvas.



El proyecto va a contener:

- Un archivo JAR con los ficheros que forman el *MIDlet*.
- Un archivo descriptor tambien llamado (archivo de manifiesto) de la aplicación que es opcional.

Un archivo JAR está formado por los siguientes elementos:

- Un archivo manifiesto que describe el contenido del archivo JAR y es opcional.
- Las clases Java que forman el *MIDlet*
- Los archivos de recursos usados por el *MIDlet*.

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

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

Atributos requeridos por el archivo JAD.

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

Descargar la aplicación al dispositivo

Una vez depuradas las fallas detectadas en la aplicación se compila, y se generan dos archivos el de manifiesto y el descriptor que son importantes a la hora de descargar los MIDlets a cualquier dispositivo.

Creación del archivo JAR

Antes de crear nuestro archivo JAR osea empaquetar nuestro *MIDlet* hay que realizar un proceso de preverificación de las clases Java. Revisar el código del *MIDlet* para ver que no viola ninguna restricción de seguridad de la plataforma J2ME.

Después de creado el archivo de manifiesto solo queda crear el archivo JAR que contiene los recursos que usa nuestra aplicación, y crearemos también un archivo descriptor JAD. Para ello lo podemos hacer desde la línea de comandos y escribiremos:

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

Manifiesto: Su finalidad es describir el contenido del fichero .JAR a través de información como el nombre, versión, fabricante, etc; también se incluye en este fichero una entrada por cada MIDlet que lo compone. Su extensión es .mf.

Nombrearchivo: Es el nombre del archivo en donde se guarda los midlets de la aplicación s como un ejecutable.

Clases Java: son las clases que se utilizan en la aplicación este archivo se genera al momento de la compilación.

Recursos: Recursos utilizados en la aplicación.

Pasar la Aplicación al dispositivo

Para descargar la aplicación al dipositivo primeramente hay que conocer si el dispositivo consta con las caracteristicas necesarias para soportar JAVA y sobre todo estar seguro de que el dispositivo cuenta con el módulo de software de gestión de aplicaciones, luego tener los drivers del dispositivo (teléfono) al que se desea bajar la

aplicación como por ejemplo el juego tres en raya, como los usuarios móviles no están conectados permanentemente a la red y por lo tanto, requieren descargar la información en su dispositivo para acceder y manipular una copia local de los datos.

Esta se la realiza mediante la conexión de un cable serie de datos para poder descargar la aplicación al dispositivo, para esto solo se descarga al dispositivo móvil el archivo JAR, el mismo que simpre que se desee será actualizado.

Sincronización

Posteriormente, los usuarios necesitan trasladar las modificaciones realizadas en su copia local a la base de datos central y obtener de ésta las modificaciones que haya tenido, resolviendo posibles conflictos. Este proceso se conoce como sincronización , se ha convertido en el talón de Aquiles de la computación móvil debido a la diversidad de protocolos existentes para llevarla a cabo, lo cual conlleva a serios problemas de interoperabilidad.

La adopción de SyncML como un estándar de sincronización parece favorable, puesto que los modelos 6800 y 7250 de Nokia y PA800 de Ericcson, pioneros en teléfonos celulares que soportan esta tecnología, muestran la confianza y la voluntad de reconocidas compañías para lograr un ambiente de interoperabilidad. Por otro lado, existe una implementación Java de fuente abierta del protocolo conocida como Sync4j, referida en el trabajo de grado “SyncML and his Java Implementation sync4j” .

CAPITULO IV

4. Los MIDlets

4.1. Descripción del capítulo

Dentro de este capítulo explicaremos más detalladamente el estudio de los MIDlets veremos cuáles son sus propiedades, conoceremos su ciclo de vida y estados por los que pasa. Vamos a tratar también del gestor de aplicaciones y la relación de éste con los *MIDlets*.

4.2. ¿Qué es un MIDlet?

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

- Todos los dispositivos de información móviles deben contar con un módulo software encargado de la gestión de los MIDlets (cargarlos, ejecutarlos…). Este software es denominado gestor de aplicaciones.
- Todos los MIDlets deben ofrecer la misma interfaz a todos los gestores de aplicaciones.

Así, independientemente de la funcionalidad interna que implementen (un juego, una agenda,…), a los dispositivos pueden identificar a los MIDlet y realizar acciones sobre ellos. Este comportamiento se consigue mediante el mecanismo de herencia: todos los

MIDlets heredan de la misma clase, javax.microedition.midlet.MIDlet.

4.3. El Gestor de Aplicaciones

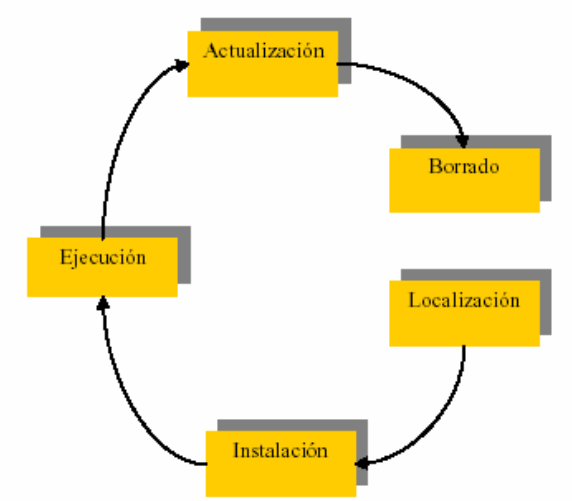
El gestor de aplicaciones o AMS (*Application Management System*) es el *software* encargado de gestionar los *MIDlets*. Este *software* reside en el dispositivo y es el que

nos permite ejecutar, pausar o destruir nuestras aplicaciones J2ME. A partir de ahora nos referiremos a él con las siglas de sus iniciales en inglés AMS. El AMS realiza dos grandes funciones:

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

4.3.1. Ciclo de vida de un MIDlet

El ciclo de vida de un MIDlet pasa por 5 fases: descubrimiento, instalación, ejecución, actualización y borrado.



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

1. Descubrimiento: Esta fase es la etapa previa a la instalación del MIDlet y es donde seleccionamos a través del gestor de aplicaciones la aplicación a descargar. Por tanto, el gestor de aplicaciones nos tiene que proporcionar los mecanismos necesarios para realizar la elección del MIDlet a descargar. El AMS puede ser capaz de realizar la descarga de aplicaciones de diferentes maneras, dependiendo de las capacidades del dispositivo. Por ejemplo, esta descarga la podemos realizar mediante un cable conectado a un ordenador o mediante una conexión inalámbrica.

2. Instalación: Una vez descargado el MIDlet en el dispositivo, comienza el proceso de instalación. En esta fase el gestor de aplicaciones controla todo el proceso informando al usuario tanto de la evolución de la instalación como de si existiese algún problema durante ésta. Cuándo un MIDlet está instalado en el dispositivo, todas sus clases, archivos y almacenamiento persistente están preparados y listos para su uso.

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

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

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

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

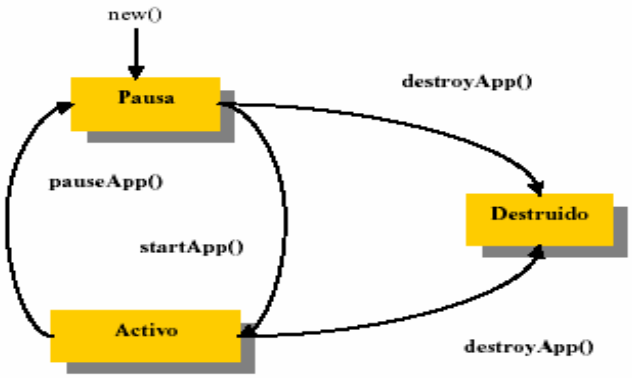
4.3.2. Estados de un MIDlet en fase de ejecución

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

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

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

Diagrama de estados de un *MIDlet* en ejecución:



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

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

Cuándo el dispositivo está preparado para ejecutar el MIDlet, el AMS invoca al método MIDlet.startApp() para entrar en el estado de "Activo". El *MIDlet* entonces, ocupa todos los recursos que necesita para su ejecución. Durante este estado, el MIDlet puede pasar al estado de "Pausa" por una acción del usuario, o bien, por el AMS que reduciría en todo lo posible el uso de los recursos del dispositivo por parte del MIDlet.

Tanto en el estado "Activo" como en el de "Pausa", el *MIDlet* puede pasar al estado "Destruido" realizando una llamada al método MIDlet.destroyApp(). Esto puede ocurrir porque el *MIDlet* haya finalizado su ejecución o porque una aplicación prioritaria necesite ser ejecutada en memoria en lugar del *MIDlet*. Una vez destruido el *MIDlet*, éste libera todos los recursos ocupados.

Ciclo de desarrollo de un MIDlet

- Editar
- Compilar
- Preverificar MIDlet
- Ejecución en el emulador

- Ejecución en el dispositivo

4.4. El paquete javax.microedition.midlet

Este paquete define las aplicaciones MIDP y su comportamiento con respecto al entorno de ejecución. Como ya sabemos, una aplicación creada usando MIDP es un *MIDlet*. A continuación se indican las clases que están incluidas en este paquete:

Clases	Descripción
MIDlet	Aplicación MIDP
MIDletstateChangeException	Indica que el cambio de estado ha fallado

4.4.1. Clase MIDlet

public abstract class MIDlet

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

Los métodos de los que dispone esta clase son los siguientes:

- protected MIDlet()

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

- public final int checkPermission(String permiso)

Consigue el estado del permiso especificado. Este permiso está descrito en el atributo MIDlet-Permission del archivo JAD. En caso de no existir el permiso por el que se pregunta, el método devolverá un 0. En caso de no conocer el estado del permiso en ese momento debido a que sea necesaria alguna acción por parte del usuario, el método devolverá un -1. Los valores devueltos por el método se corresponden con la siguiente descripción:

0 si el permiso es denegado

1 si el permiso es otorgado

-1 si el estado es desconocido

- protected abstract void destroyApp(boolean incondicional) throws MIDletstateChangeException

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

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

- public final String getAppProperty(String key)

Este método proporciona al *MIDlet* un mecanismo que le permite recuperar el valor de las propiedades desde el AMS. Las propiedades se consiguen por medio de los archivos *manifest* y *JAD*.

El nombre de la propiedad a recuperar debe ir indicado en el parámetro **key**. El método nos devuelve un String con el valor de la propiedad o *null* si no existe ningún valor asociado al parámetro **key**. Si **key** es **null** se lanzará la excepción NullPointerException.

- public final void notifyDestroyed()

Este método es utilizado por un MIDlet para indicar al AMS que ha entrado en el estado de “Destruido”. En este caso, todos los recursos ocupados por el MIDlet deben

ser liberados por éste de la misma forma que si se hubiera llamado al método MIDlet.destroyApp().

El AMS considerará que todos los recursos que ocupaba el MIDlet están libres para su uso.

- public final void notifyPaused()

Se notifica al AMS que el MIDlet no quiere estar “Activo” y que ha entrado en el estado de “Pausa”. Este método sólo debe ser invocado cuándo el MIDlet esté en el estado “Activo”.

Una vez invocado este método, el MIDlet puede volver al estado “Activo” llamando al método MIDlet.startApp(), o ser destruido llamando al método MIDlet.destroyApp(). Si la aplicación es pausada por sí misma, es necesario llamar al método MIDlet.resumeRequest() para volver al estado “Activo”.

- protected abstract void pauseApp()

Indica al MIDlet que entre en el estado de “Pausa”. Este método sólo debe ser llamado cuándo el MIDlet esté en estado “Activo”. Si ocurre una excepción RuntimeException durante la llamada a MIDlet.pauseApp(), el MIDlet será destruido inmediatamente. Se llamará a su método MIDlet.destroyApp() para liberar los recursos ocupados.

- public final boolean platformRequest(String url)

Establece una conexión entre el MIDlet y la dirección URL. Dependiendo del contenido de la URL, nuestro dispositivo ejecutará una determinada aplicación que sea capaz de leer el contenido y dejar al usuario que interactúe con él. Si, por ejemplo, la URL hace referencia a un archivo JAD o JAR, el dispositivo sabe que se desea instalar la aplicación asociada a este archivo y comenzará el proceso de descarga. Pero si la

URL tiene el formato tel:<número>, nuestro dispositivo entenderá que se desea realizar una llamada telefónica.

- public final void resumeRequest()

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

- protected abstract void startApp() throws MIDletstateChangeException

Este método indica al MIDlet que ha entrado en el estado "Activo". Este método sólo se invoca cuándo el MIDlet está en el estado de "Pausa". En el caso de que el MIDlet no pueda pasar al estado "Activo" en ese momento pero si puede hacerlo luego, se lanzaría la excepción MIDletstateChangeException.

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

Métodos de la clase MIDlet	
Costructores	
protected	MIDlet()
Métodos	
int	checkPermission(String permiso)
protected abstract void	destroyAp(Boolean unconditional)
String	getAppProperty(String key)
void	notifyDestroyed()

void	notifyPaused()
protected abstract void	pauseApp()
boolean	platformRequest()
void	resumeRequest()
protected abstract void	startApp()

4.4.2. Clase MIDletChangeStateException

public class MIDletstateChangeException extends Exception

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

4.5. Estructura de los MIDlets

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

Los *MIDlets* tienen la siguiente estructura:

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

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

Ejemplo de programa Hola Mundo

```
    import javax.microedition.midlet.*;
    import javax.microedition.lcdui.*;
    public class HolaMundo extends MIDlet{
        private Display pantalla;
        private Form formulario = null;
        public HolaMundo(){
            pantalla = Display.getDisplay(this);
            formulario = new Form("Hola Mundo");
        }
        public void startApp(){
            pantalla.setCurrent(formulario);
        }
        public void pauseApp(){
        }
```

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

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

CAPITULO V

5. LA CONFIGURACIÓN CLDC

5.1. Descripción del capítulo

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

La configuración CLDC se ocupa de las siguientes áreas:

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

Lo que se intenta con la configuración CLDC es mantener lo más pequeño posible el número de áreas abarcadas. Es mejor restringir el alcance de CLDC para no exceder las limitaciones de memoria o no excluir ningún dispositivo en particular. En el futuro, la configuración CLDC podría incluir otras áreas adicionales.

5.2. Objetivos y requerimientos

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

CLDC es la base para uno o más perfiles. Un perfil de J2ME define un conjunto adicional de APIs y características para un mercado concreto, dispositivo determinado o industria.

5.2.1. Objetivos

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

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

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

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

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

conectados a internet, comunicadores, etc. La transición está actualmente en marcha. Uno de los principales objetivos de CLDC es tomar parte en esta transición permitiendo el uso del lenguaje de programación Java como base para la distribución de contenido dinámico para esta próxima generación de dispositivos.

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

5.2.2. Requerimientos

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

• **Requerimientos hardware:** CLDC está diseñado para ejecutarse en una gran variedad de dispositivos, desde aparatos de comunicación inalámbricos como teléfonos móviles, buscapersonas, hasta organizadores personales, terminales de venta, etc. Las capacidades del hardware de estos dispositivos varían considerablemente y por esta razón, los únicos requerimientos que impone la configuración CLDC son los de memoria. La configuración CLDC asume que la máquina virtual, librerías de configuración, librerías de perfil y la aplicación debe ocupar una memoria entre 160 Kb y 512 Kb. Más concretamente:

- 128 Kb de memoria no volátil para la JVM y las librerías CLDC.
- Al menos 32 Kb de memoria volátil para el entorno de ejecución Java y objetos en memoria.

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

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

• **Requerimientos J2ME:** CLDC es definida como la configuración de J2ME. Esto tiene importantes implicaciones para la especificación CLDC:

o Una configuración J2ME solo deber definir un complemento mínimo de la tecnología Java. Todas las características incluidas en una configuración deben ser generalmente aplicables a una gran variedad de dispositivos. Características específicas para un dispositivo, mercado o industria deben ser definidas en un perfil en vez de en el CLDC. Esto significa que el alcance de CLDC está limitado y debe ser complementado generalmente por perfiles.

o El objetivo de la configuración es garantizar portabilidad e interoperabilidad entre varios tipos de dispositivos con recursos limitados. Una configuración no debe definir ninguna característica opcional. Esta limitación tiene un impacto significativo en lo que

se puede incluir en una configuración y lo que no. La funcionalidad más específica debe ser definida en perfiles.

5.3. Seguridad en CLDC

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

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

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

5.4. Librerías CLDC

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

5.4.1. Objetivos generales

Un objetivo general para el diseño de librerías Java para CLDC es proporcionar un conjunto mínimo de librerías útiles para el desarrollo de aplicaciones y definición de perfiles para una variedad de pequeños dispositivos. Dadas las estrictas restricciones de memoria y diferencias en las características de estos dispositivos, es completamente imposible ofrecer un conjunto de librerías que satisfagan a todo el mundo.

5.4.2. Compatibilidad

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

Las librerías CLDC pueden ser divididas en dos categorías:

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

5.4.3. Clases heredadas de J2SE

CLDC proporciona un conjunto de clases heredadas de la plataforma J2SE. En total, usa unas 37 clases provenientes de los paquetes java.lang, java.util y java.io.

Cada una de estas clases debe ser idéntica o ser un subconjunto de la correspondiente clase de J2SE. Tanto los métodos como la semántica de cada clase deben permanecer invariables. A continuación podremos ver cada una de estas clases agrupadas por paquetes.

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

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

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

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

5.4.4. Clases propias de CLDC

CLDC hereda algunas clases del paquete java.io, pero no hereda ninguna clase relacionada con la E/S de ficheros, por ejemplo. Esto es debido a la gran variedad de

dispositivos que abarca CLDC, ya que, para éstos puede resultar innecesario manejar ficheros. No se han incluido tampoco las clases del paquete java.net, basado en comunicaciones TCP/IP ya que los dispositivos CLDC no tienen por qué basarse en este protocolo de comunicación. Pero a su vez CLDC posee un conjunto de clases más genérico para la E/S y la conexión en red que recibe el nombre de "*Generic Connection Framework*". Estas clases están incluidas en el paquete javax.microedition.io y son:

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

CAPITULO VI

6. Interfaces gráficas de usuario

6.1. Descripción del capítulo

En este capítulo nos vamos a centrar en el paquete javax.microedition.lcdui (*Interfaz de usuario con pantalla LCD*), que es donde se encuentran los elementos gráficos que vamos a utilizar en nuestras aplicaciones.

En primer lugar realizaremos una introducción a las interfaces de usuario en los dispositivos MIDP. En este punto realizaremos una división entre interfaces de usuario de alto nivel e interfaces de usuario de bajo nivel. Se irá describiendo las clases que componen el paquete javax.microedition.lcdui y sus correspondientes métodos.

6.2. Introducción a las interfaces de usuario

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

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

Durante el proceso de especificación del MIDP, el lenguaje Java ya contaba (en su versión estándar, J2SE) con una completa API para generar interfaces de usuario, AWT (*Application Windows Toolkit*), además de las clases *Swing*. Sin embargo el MIDP *Expert Group* decidió definir una API especifica, ya que AWT fue ideada para su uso en ordenadores de sobremesa.

Para ofrecer máxima flexibilidad a los desarrolladores esta API se dividió en dos partes: una de alto nivel (*High Level* API) y otra de bajo nivel (*Low Level* API), teniendo en cuenta también la diversidad de aplicaciones que podemos realizar para los dispositivos MID y los elementos que nos proporcionan la configuración CLDC y el perfil MIDP, a continuación definiremos cada una de estas interfaces:

Interfaz de usuario de alto nivel. Esta interfaz usa componentes tales como botones, cajas de texto, formularios, etc. Estos elementos son implementados por cada dispositivo y la finalidad de usar las APIs de alto nivel es su portabilidad. Al usar estos elementos, perdemos el control del aspecto de nuestra aplicación ya que la estética de estos componentes depende exclusivamente del dispositivo donde se ejecute. En cambio, usando estas APIs de alto nivel ganaremos un alto grado de portabilidad de la misma aplicación entre distintos dispositivos. Fundamentalmente, se usan estas APIs cuando queremos construir aplicaciones de negocios.

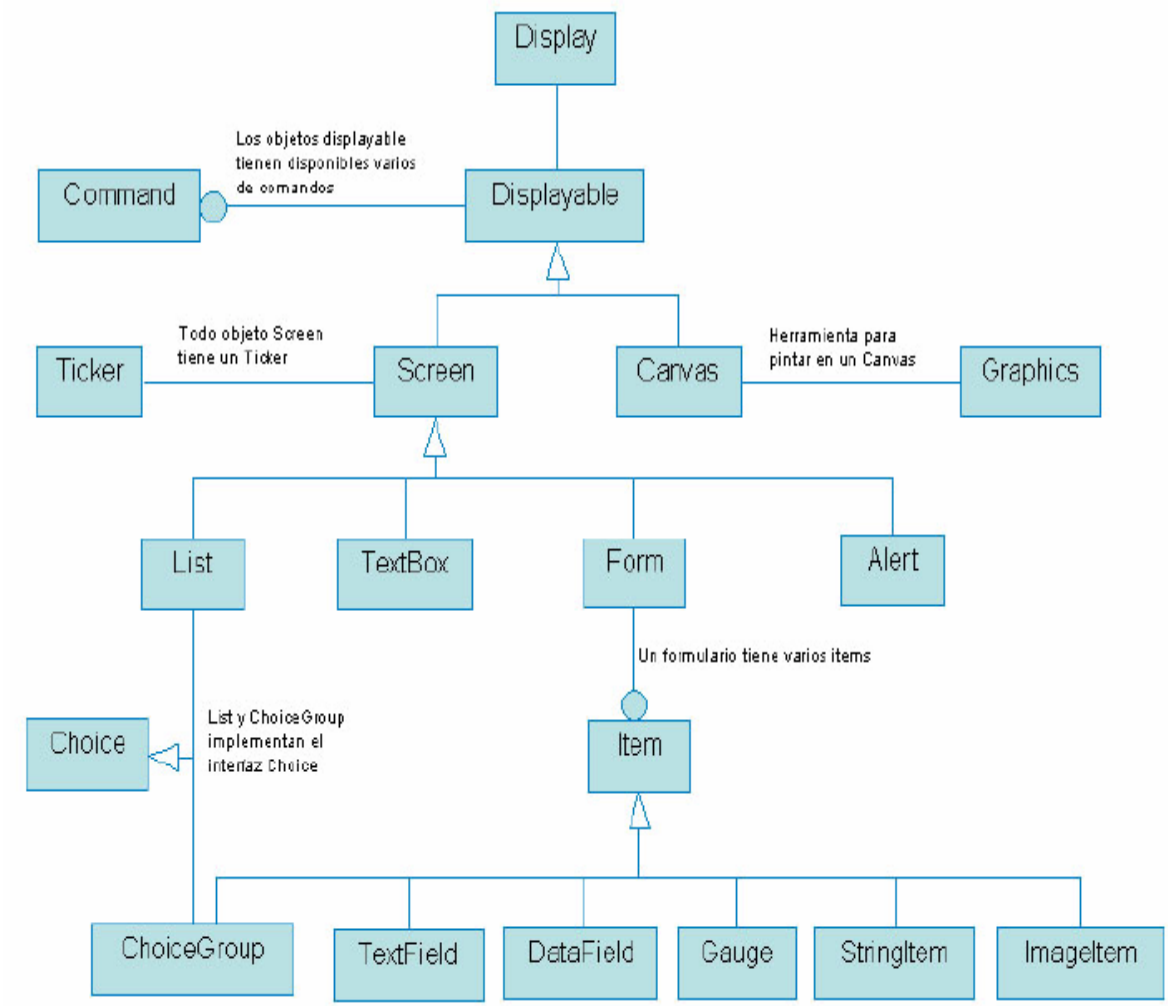
Interfaz de usuario de bajo nivel. Al crear una aplicación usando las APIs de bajo nivel, tendremos un control total de lo que aparecerá por pantalla. Estas APIs nos darán un control completo sobre los recursos del dispositivo y podremos controlar eventos de bajo nivel como, por ejemplo, el rastreo de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

6.3. Estructura general de la API de interfaz de usuario

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

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

6.4. Jerarquía de clases de interfaz de usuario en MIDP



El esquema anterior muestra la relación y jerarquía existente entre las clases de la tabla anterior.

6.5. Funcionamiento de la interfaz de usuario

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

• Pantallas de la API de alto nivel. Todas estas clases descienden de la clase Screen. No proporcionan ningún tipo de control sobre su apariencia, que es responsabilidad

concreta de cada dispositivo. Además, encapsulan toda la gestión de eventos de bajo nivel. Se pueden dividir en predefinidas (TextBox, List y Alert) a las que no se le pueden añadir componentes adicionales y genéricas (Form) que permiten la adición de componentes como imágenes, etiquetas de texto.

• Pantallas de la API de bajo nivel. Todas las pantallas de la API de bajo nivel descienden de la clase Canvas. Ofrecen al programador un control casi absoluto sobre lo que aparece en la ventana del MID. Además, dan acceso al manejo de eventos de bajo nivel.

6.6. La clase Display

public class Display

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

Métodos	Descripción
void callSerially(Runnable r)	Retrasa la ejecución del método run() del objeto r para no interferir con los eventos de usuario
boolean flashBacklight(int duracion)	Provoca un efecto de flash en la pantalla
int getBestImageHeight(int imagen)	Devuelve el mejor alto de imagen para un tipo dado
int getBestImageWidth(int imagen)	Devuelve el mejor ancho de imagen para un tipo dado
int getBorderStyle(bolean luminosidad)	Devuelve el estilo de borde actual
int getColor(int color)	Devuelve un color basado en el parámetro

	pasado
Displayable getCurrent()	Devuelve la pantalla actual
static Display getDisplay(MIDlet m)	Devuelve una referencia a la pantalla del MIDlet m
boolean isColor()	Devuelve trae o false si la pantalla es de color o b/n
int numAlphaLevels()	Devuelve el número de niveles alpha soportados
int numColors()	Devuelve el número de colores aceptados por el MID
void setCurrent(Alert a, Displayable d)	Establece la pantalla d después de la alerta a
void setCurrent(Displayable d)	Establece la pantalla actual
void setCurrent(Item item)	Establece la pantalla en la zona donde se encuentre el item
boolean vibrate(int duracion)	Realiza la operación de vibración del dispositivo

Todo *MIDlet* debe poseer al menos una instancia del objeto Display. Para obtenerla emplearemos el siguiente código:

Display pantalla = Display.getDisplay(this)

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

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

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

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

6.7. La clase Displayable

public abstract class Displayable

La clase Displayable representa a las pantallas de nuestra aplicación. Como hemos dicho, cada objeto Display puede tener tantos objetos Displayable como quiera. Mediante los métodos getCurrent y setCurrent controlamos qué pantalla queremos que sea visible y accesible en cada momento.

La clase abstracta Displayable incluye los métodos encargados de manejar los eventos de pantalla y añadir o eliminar comandos. Estos métodos son:

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

6.8. Las clases Command y CommandListener

public class Command

Un objeto de la clase Command mantiene información sobre un evento, se puede decir que es como un botón de Windows, por hacer una analogía. Generalmente, los

implementaremos en nuestros *MIDlets* cuando queramos detectar y ejecutar una acción simple.

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

- **Etiqueta:** La etiqueta es la cadena de texto que aparecerá en la pantalla de dispositivo que identificará a nuestro Command.
- **Tipo:** Indica el tipo de objeto Command que queremos crear. Los tipos que se pueden asignar son:

Tipo	Descripción
BACK	Petición para volver a la pantalla anterior
CANCEL	Petición para cancelar la acción en curso
EXIT	Petición para salir de la aplicación
HELP	Petición para mostrar información de ayuda
ITEM	Petición para introducir el comando en un “item” en la pantalla
OK	Aceptación de una acción por parte del usuario
SCREEN	Para Commands de propósito más general
STOP	Petición para parar una operación

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

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

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

```
new Command(“Atras”,Command.BACK,1)
```

Métodos de la clase Command

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

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

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

6.9. La interfaz de usuario de alto nivel

En este punto vamos a estudiar la clase Screen y todas las subclases derivadas de ella. Todas estas clases conforman la interfaz de usuario de alto nivel. Como decíamos, la clase Screen es la superclase de todas las clases que conforman la interfaz de usuario de alto nivel:

public abstract class Screen extends Displayable

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

6.9.1. La clase Alert

public class Alert extends Screen

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

Alert(String titulo)

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

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

- **Modal:** La pantalla de aviso permanece un tiempo indeterminado hasta que es cancelada por el usuario. Esto lo conseguimos invocando al método Alert.setTimeOut(Alert.FOREVER).

- **No Modal:** La pantalla de aviso permanece un tiempo definido por nosotros. Para ello indicaremos el tiempo en el método setTimeOut(tiempo). Una vez finalizado el tiempo, la pantalla de aviso se eliminará de pantalla y aparecerá el objeto Displayable que nosotros definamos.

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

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

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

AlertType.CONFIRMATION.playSound(display)

6.9.2. La clase List

public class List extends Screen implements Choice

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

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

List(String titulo, int listType)

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

Tipos de listas

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

6.9.3. La clase TextBox

public class TextBox extends Screen

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

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

Restricciones de entrada de caracteres

Valor	Descripción
ANY	Permite la inserción de cualquier caracter
CONSTRAINT_MASK	Se usa cuando necesitamos determinar el

	valor actual de las restricciones
EMAILADDR	Permite caracteres válidos para direcciones de correo electrónico
NUMERIC	Permite solo números
PASSWORD	Oculta los caracteres introducidos mediante una máscara para proporcionar privacidad
PHONENUMBER	Permite caracteres válidos sólo para números de teléfono
URL	Permite caracteres válidos sólo para direcciones URL

Ahora podemos crear cualquier tipo de TextBox. La llamada al constructor la realizaríamos de la siguiente manera:

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

6.9.4. La clase Form

public class Form extends Screen

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

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

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

6.10. INTERFAZ DE USUARIO DE BAJO NIVEL

6.10.1. La clase Canvas

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

La clase Canvas es la superclase de todas las pantallas que usan las APIs de bajo nivel. La clase Canvas permite manejar eventos de bajo nivel y dibujar cualquier cosa por pantalla. Por esta razón se usa como base para la realización de juegos. Esta clase posee un método abstracto paint() que debemos implementar obligatoriamente y es el que se encarga de dibujar en la pantalla del dispositivo MID.

En el siguiente ejemplo podemos ver como sería el programa HolaMundo creado en una pantalla derivada de Canvas.

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

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

6.10.2. Eventos de bajo nivel

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

• A través de Commands.

• A través de códigos de teclas. Este método es el que Canvas proporciona para detectar eventos de bajo nivel. Estos códigos son valores numéricos que están asociados a las diferentes teclas de un MID. Estos códigos se corresponden con las teclas de un teclado convencional de un teléfono móvil (0-9,*,#). La clase Canvas proporciona estos códigos a través de constantes que tienen asociados valores enteros.

Códigos de teclas

Nombre	Valor
KEY_NUM0	48
KEY_NUM1	49
KEY_NUM2	50
KEY_NUM3	51
KEY_NUM4	52
KEY_NUM5	53
KEY_NUM6	54
KEY_NUM7	55
KEY_NUM8	56
KEY_NUM9	57
KEY_STAR	42
KEY_POUND	35

Con estos códigos anteriores ya podemos conocer cual es la tecla que ha pulsado el usuario. Canvas, además proporciona unos métodos que permitirán manejar estos eventos con facilidad. La implementación de estos métodos es vacía, por lo que es

misión nuestra implementar los que necesitemos en nuestra aplicación de acuerdo con el propósito de ésta.

Métodos para manejar códigos de teclas

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

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

Métodos para detectar eventos de puntero

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

Al igual que hacíamos con los eventos de teclado, es nuestro deber implementar estos métodos si queremos que nuestro *MIDlet* detecte este tipo de eventos.

Para saber si el dispositivo MID está equipado con algún tipo de puntero podemos hacer uso de los siguientes métodos:

Métodos para ver si el MID permite eventos de puntero

Método	Descripción
boolean hasPointerEvents()	Devuelve trae si el dispositivo posee algún puntero
boolean hasPointerMotionEvents()	Devuelve trae si el dispositivo puede detectar acciones como pulsar, arrastrar y soltar el puntero

6.10.3. Manipulación de elementos en una pantalla Canvas

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

6.10.3.1. El método paint()

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

- El método paint() nunca debe ser llamado desde el programa. El gestor de aplicaciones es el que se encarga de realizar la llamada a éste método cuando sea necesario.
- Cuando deseemos que se redibuje la pantalla actual debido a alguna acción en concreto del usuario o como parte de alguna animación, deberemos realizar una llamada

al método repaint(). Al igual que ocurre con los eventos de teclado, la petición se encolará y será servida cuando retornen todas las peticiones anteriores a ella.

- La implementación del MID no se encarga de limpiar la pantalla antes de cada llamada al método paint(). Por esta razón, éste método debe pintar cada pixel de la pantalla para, de esta forma, evitar que se vean porciones no deseadas de pantallas anteriores.

La invocación al método paint() se realiza cuando se pone la pantalla Canvas como pantalla activa a través del método de la clase Display setCurrent(Canvas) o después de invocar al método showNotify(). Una pantalla Canvas no posee la capacidad por sí misma de restaurar su estado en caso de que el AMS interrumpa la ejecución normal de la aplicación, por ejemplo, avisar de una llamada entrante. Para resolver este inconveniente, Canvas nos proporciona dos métodos cuya implementación es vacía: hideNotify() y showNotify().

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

6.10.3.2. La clase Graphics

public class Graphics

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

• Dentro del método paint() de la clase Canvas. Aquí podemos usar el objeto Graphics para pintar en la pantalla del dispositivo.

• A través de una imagen usando el siguiente código:

Image imgtemp = Image.createImage(ancho,alto);

Graphics g = imgtemp.getGraphics();

La clase Graphics posee multitud de métodos que nos permitirán seleccionar colores, dibujar texto, figuras geométricas, etc.

6.10.3.3. Sistema de coordenadas

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

La clase Canvas nos proporciona los métodos necesarios para obtener el ancho y el alto de la pantalla a través de getWidth() y getHeight() respectivamente.

Una posición en la pantalla estará definida por dos coordenadas x e y que definirán el desplazamiento lateral y vertical, siendo la posición (0,0) el pixel situado en la esquina superior izquierda. Incrementando los valores de x e y, nos moveremos hacia la derecha y hacia abajo respectivamente.

Es posible cambiar el origen de coordenadas de la pantalla mediante el método de la clase Graphics translate(int x, int y). Éste método cambia el origen de coordenadas al punto definido por los parámetros x e y provocando un desplazamiento de todos los objetos en pantalla.

6.10.3.4. Manejo de colores

La clase Graphics nos proporciona métodos con los que podremos seleccionar colores, pintar la pantalla de un color determinado o zonas de ellas, dibujar líneas,

rectángulos, etc. Como programadores de *MIDlets* es misión nuestra adaptar nuestra aplicación a las capacidades gráficas del dispositivo donde se vaya a ejecutar.

Disponemos para ello de varios métodos que nos suministra la clase Display con los que podremos saber si el dispositivo en cuestión cuenta con pantalla a color o escala de grises (Display.isColor()), también podremos conocer el número de colores soportados (Display.numColors()). Para crear aplicaciones de calidad deberíamos usar constantemente estos métodos en ellas.

En particular, la clase Graphics nos proporciona un modelo de color de 24 bits, con 8 bits para cada componente de color: rojo, verde y azul. Podemos seleccionar un color invocando al método Graphics.setColor(int RGB) o Graphics.setColor(int rojo, int verde, int azul) donde podemos indicar los niveles de los componentes que conforman el color que deseamos usar. Veamos como se realizaría la selección de algunos colores básicos:

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

Si después de seleccionar un color se escribe lo siguiente: g.fillRect(0,0,getWidth(),getHeight()) se pintaría toda la pantalla del color seleccionado con anterioridad.

6.10.3.5. Manejo de texto

Es posible incluir texto en una pantalla Canvas, es posible seleccionar un estilo de letra con el que posteriormente podremos escribir texto por pantalla. Para ello nos

ayudaremos de la clase Font que nos permitirá seleccionar el tipo de letra y almacenarlo en un objeto de este tipo para posteriormente usarlo en nuestro Canvas.

Para seleccionar un tipo de letra tenemos que tener en cuenta los tres atributos que definen nuestra fuente.

Fuentes disponibles

	Atributos
Aspecto	FACE_SYSTEM
	FACE_MONOSPACE
	FACE_PROPORTIONAL
Estilo	STYLE_PLAIN
	STYLE_BOLD
	STYLE_ITALIC
	STYLE_UNDERLINED
Tamaño	SIZE_SMALL
	SIZE_MEDIUM
	SIZE_LARGE

Para crear una determinada fuente tendremos que invocar al método Font.getFont(int aspecto, int estilo, int tamaño) que nos devuelve el objeto Font deseado. Por ejemplo:

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

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

g.setFont(fuente); //Seleccionamos la fuente activa.

g.drawString(“Cadena de texto”, getWidth()/2, getHeight()/2,

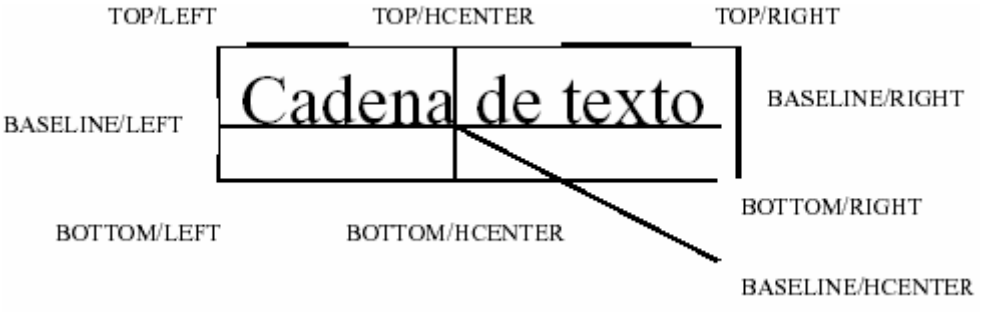
BASELINE|HCENTER);

Este código selecciona un tipo de fuente con la que posteriormente se escribe la cadena “Cadena de texto” posicionado en la pantalla en el punto medio (getWidth()/2,

getHeight()/2). El parámetro BASELINE|HCENTER indica el posicionamiento del texto con respecto al punto medio.

6.10.3.6. Posicionamiento del texto

Al mostrar una cadena de texto por pantalla hemos de indicar la posición en la que queremos situarla. Esta posición viene indicada por el punto de anclaje o *anchorpoint*.



En el ejemplo anterior hemos puesto el punto de anclaje BASELINE|HCENTER de la cadena “Cadena de texto” en la posición central definida por getWidth()/2, getHeight()/2.

6.10.3.7. Figuras geométricas

La clase Graphics nos proporcionaba métodos capaces de mostrar figuras geométricas por pantalla, las cuales pueden ser líneas, rectángulos y arcos.

- **Líneas**

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

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

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

- **Rectángulos**

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

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

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

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

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

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

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

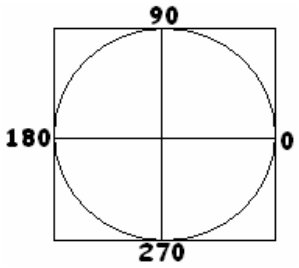
Un rectángulo de este tipo se realizaría de una de las siguientes maneras:

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

- **Arcos**

En este caso podemos dibujar dos tipos de arcos: Arco simple o arco con color de relleno. Para dibujar un arco tenemos que imaginarnos que ese arco va dentro de una “caja”.

Cuándo vayamos a crear un arco tenemos que tener en mente la “caja” que lo va a delimitar. Esto es así porque a la hora de crear un arco hemos de crear primero esa caja, especificando los parámetros como hacíamos cuando creábamos un rectángulo, además del ángulo inicial y ángulo total. El ángulo inicial es desde donde empezaremos a construir el arco, siendo 0 el punto situado a las 3 en punto, 90 el situado a las 12, 180 el situado a las 9 y 270 el situado a las 6. El ángulo total especifica la amplitud que tendrá el arco.



Si por ejemplo construimos un arco de la siguiente manera:

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

ó

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

- **Imágenes**

En una pantalla Canvas también es posible insertar imágenes. Además, aquí es posible usar dos tipos distintos de objetos Image:

Inmutables: Son imágenes que provienen de un archivo con formato gráfico. Se llaman inmutables ya que no es posible variar su aspecto.

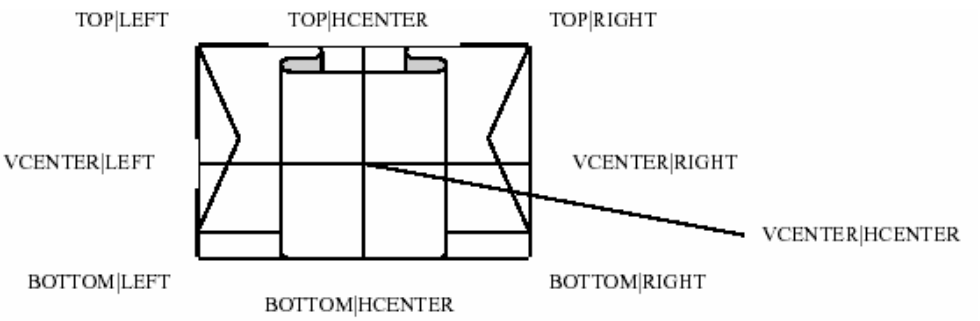
```
Image im = Image.createImage(“\imagen.png”);
```

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

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

im = crearImagen();

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



6.10.4. Conceptos básicos para la creación de juegos en MIDP

La tecnología MIDP es el de los videojuegos. Desde hace unos años, la mayoría de los teléfonos móviles traen incorporados uno que otro juego, la mayoría de ellos bastante simples y, por supuesto en blanco y negro. Debido al avance tecnológico de la telefonía móvil actualmente encontramos en el mercado teléfonos con pantalla a color de tamaño considerable, teléfonos que soportan GPRS, etc. Con este panorama no es de extrañar que los teléfonos móviles se conviertan en dispositivos de ocio, además de simples dispositivos de comunicación.

6.10.5. Eventos de teclado para juegos

Los eventos de bajo nivel son controlados en las pantallas Canvas a través de códigos. MIDP nos proporciona además de los códigos genéricos de teclado, un conjunto de constantes que se refieren a acciones específicas de un juego.

Códigos de teclas para juegos

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

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

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

Los métodos proporcionados por Canvas para controlar los eventos de bajo nivel keyPressed(int keyCode), keyRepeated(int keyCode) y keyReleased(int keyCode) trabajan con códigos genéricos de teclado (keyCodes). Para poder trabajar con códigos

de juegos tenemos que usar el método getGameAction(int keyCode) dentro de cada uno de los métodos anteriores:

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

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

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

CAPITULO VII

7. Sprite

7.1. La Clase Sprite

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

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

Podemos decir que un sprite es un elemento gráfico determinado (una nave, un coche, etc...) que tiene entidad propia y sobre la que podemos definir y modificar ciertos atributos, como la posición en la pantalla, si es o no visible, etc...

Un sprite, pues, tiene capacidad de movimiento. Distinguimos dos tipos de movimiento en los sprites: el movimiento externo, es decir, el movimiento del sprite por la pantalla, y el movimiento interno o animación.

Para posicionar un sprite en la pantalla hay que especificar sus coordenadas. Para identificar un cuadrante hay que indicar una letra para el eje vertical (lo llamaremos eje Y) y un número para el eje horizontal (al que llamaremos eje X). En un ordenador, un punto en la pantalla se representa de forma parecida. La esquina superior izquierda representa el centro de coordenadas.

Un punto se identifica dando la distancia en el eje X al lateral izquierdo de la pantalla y la distancia en el eje Y a la parte superior de la pantalla. Las distancias se miden en píxeles. Si queremos indicar que un sprite está a 100 píxeles de distancia del eje vertical y 150 del eje horizontal, decimos que está en la coordenada (100,150).

Piénsese en un juego dónde controlemos a un caballero que debe ir avanzando por pantallas dónde se encuentra con multitud de enemigos. Tanto el caballero como los enemigos estarían representados por un objeto Sprite.

Podremos observar cómo mueve las piernas y los brazos según avanza por la pantalla.

Éste es el movimiento interno o animación. La siguiente figura muestra la animación del sprite de un gato como ejemplo.

Animación de un sprite.

Otra característica muy interesante de los sprites es que estos objetos ya tendrían la información suficiente para realizar las animaciones de los movimientos de los personajes y dispondríamos de métodos que nos darían la posición de cada uno de ellos y evitaríamos de esta manera, que dos personajes ocuparan la misma posición en pantalla (detección de colisiones).

7.2. Control de Sprites

Vamos a realizar una pequeña librería para el manejo de los sprites. Luego podremos utilizar esta librería en cualquier juego que necesite de animación, así como ampliarla, ya que cubrirá sólo los aspectos básicos en lo referente a sprites.

Dotaremos a nuestra librería con capacidad para movimiento de sprites, animación (un soporte básico) y detección de colisiones. Para almacenar el estado de los Sprites utilizaremos las siguientes variables.

```
private int posx,posy;
private boolean active;
private int frame,nframes;
private Image[] sprites;
```

También necesitamos la coordenada en pantalla del sprite (que almacenamos en posx (que seria la posición x) y posy (que seria la posición y).

La variable *active* nos servirá para saber si el sprite está activo. La variable *frame* almacena el frame actual del sprite, y *nframes* el número total de frames de los que está compuesto. Por último, tenemos un array de objetos Image que contendrá cada uno de los frames del juego.

Como se puede observar no indicamos el tamaño del array, ya que aún no sabemos cuantos frames tendrá el sprite. Indicaremos este valor en el constructor del sprite.

```
// constructor. 'nframes' es el número de frames del Sprite.
public Sprite(int nframes) {
        // El Sprite no está activo por defecto.
        active=false;
        frame=1;
        this.nframes=nframes;
        sprites=new Image[nframes+1];
}
```

El constructor se encarga de crear tantos elementos de tipo Image como frames tenga el sprite. También asignamos el estado inicial del sprite. La operación más importante de un sprite es el movimiento por la pantalla. Veamos los métodos que nos permitirán moverlo.

```
public void setX(int x) {
        posx=x;
}
public void setY(int y) {
        posy=y;
```

```
    }
    int getX() {
        return posx;
    }
    int getY() {
        return posy;
    }
```

Como se puede observar, el código para posicionar el sprite en la pantalla no puede ser más simple. Los métodos setX() y setY() actualizan las variables de estado del sprite (posx,posy). Los métodos getX() y getY() realizan la operación contraria, es decir, nos devuelve la posición del sprite.

Además de la posición del sprite, nos va a interesar en determinadas condiciones conocer el tamaño del mismo.

```
    int getW() {
        return sprites[nframes].getWidth();
    }
    int getH() {
        return sprites[nframes].getHeight();
    }
```

Los métodos getW() y getH() nos devuelven el ancho y el alto del sprite en píxeles. Para ello recurrimos a los métodos getWidth() y getHeigth() de la clase Image.

Otro dato importante del sprite es si está activo en un momento determinado.

```
    public void on() {
        active=true;
    }
```

```
public void off() {
        active=false;
}
public boolean isActive() {
        return active;
}
```

También se necesitara un método que active el sprite, al que llamaremos *on()*, y otro para desactivarlo, que llamaremos *off()*. Nos resta un método para conocer el estado del sprite. Hemos llamado al método *isActive()* .

En lo referente al estado necesitamos algún método para el control de frames, o lo que es lo mismo, de la animación interna del sprite.

```
public void selFrame(int frameno) {
        frame=frameno;
}
public int frames() {
        return nframes;
}
public void addFrame(int frameno, String path) {
        try {
                sprites[frameno]=Image.createImage(path);
        } catch (IOException e) {
                System.err.println("Can`t load the image " + path + ": " +
                e.toString());
        } }
```

El método *selFrame()* fija el frame actual del sprite, mientras que el método *frame()* nos devolverá el número de frames del sprite.

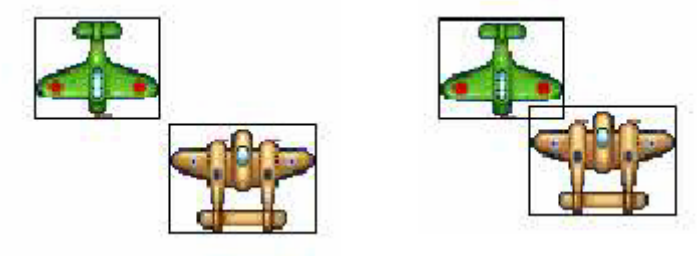
El método *addFrame()* nos permite añadir frames al sprite. Necesita dos parámetros. El parámetro *frameno,* indica el número de frame, mientras que el parámetro *path* indica el camino y el nombre del gráfico que conformará dicho frame.

Para dibujar el sprite, vamos a crear el método draw(). Lo único que hace este método es dibujar el frame actual del sprite en la pantalla.

```
public void draw(Graphics g) {
    g.drawImage (sprites[frame], posx, posy,
    Graphics.HCENTER|Graphics.VCENTER);
}
```

Nos resta dotar a nuestra librería con la capacidad de detectar colisiones entre sprites.

Imaginemos dos sprites, nuestro avión y un disparo enemigo. En cada vuelta del game loop tendremos que comprobar si el disparo ha colisionado con nuestro avión. Podríamos considerar que dos sprites colisionan cuando alguno de sus píxeles visibles (es decir, no transparentes) toca con un píxel cualquiera del otro sprite. Esto es cierto al 100%, sin embargo, la única forma de hacerlo es comprobando uno por uno los píxeles de ambos sprites. Evidentemente esto requiere un gran tiempo de computación, y es inviable en la práctica. En nuestra librería hemos asumido que la parte visible de nuestro sprite coincide más o menos con las dimensiones de la superficie que lo contiene. Si aceptamos esto, y teniendo en cuenta que una superficie tiene forma cuadrangular, la detección se simplifica al tener que detectar el caso en el que dos cuadrados se solapen.

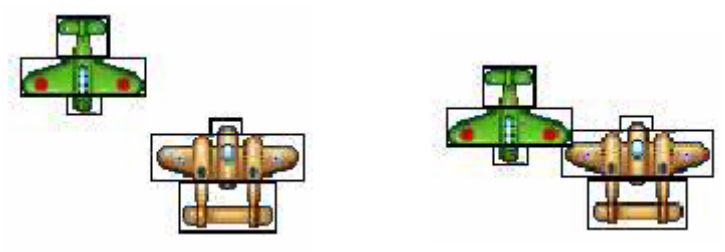


En la figura de la izquierda no existe colisión, ya que no se solapan las superficies. La segunda figura muestra el principal problema de este método, ya que nuestra librería considerará que ha habido colisión cuando realmente no ha sido así.A pesar de este pequeño inconveniente, este método de detección de colisiones es el más rápido. Es importante que la superficie tenga el tamaño justo para albergar el gráfico. Este es el aspecto que tiene nuestro método de detección de colisiones.

```
boolean collide(Sprite sp) {
        int w1,h1,w2,h2,x1,y1,x2,y2;
        w1=getW(); // ancho del sprite1
        h1=getH(); // altura del sprite1
        w2=sp.getW(); // ancho del sprite2
        h2=sp.getH(); // alto del sprite2
        x1=getX(); // pos. X del sprite1
        y1=getY(); // pos. Y del sprite1
        x2=sp.getX(); // pos. X del sprite2
        y2=sp.getY(); // pos. Y del sprite2
        if (((x1+w1)>x2)&&((y1+h1)>y2)&&((x2+w2)>x1)&&((y2+h2)>y1)) {
                return true;
        } else {
                return false;
        }
}
```

Se trata de comprobar si el cuadrado (superficie) que contiene el primer sprite, se solapa con el cuadrado que contiene al segundo.

Hay otro métodos más precisos que nos permiten detectar colisiones. Consiste en dividir el sprite en pequeñas superficies rectangulares tal y como se muestra a continuación.



Método más elaborado de detección de colisiones

Se puede observar la mayor precisión de este método. El proceso de detección consiste en comprobar si hay colisión de alguno de los cuadros del primer sprite con alguno de los cuadrados del segundo utilizando la misma comprobación que hemos utilizado en el primer método para detectar si se solapan dos rectangulos. La implementación de este método de detección de colisiones depende del programador. A continuación se muestra el listado completo de nuestra librería.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;
class Sprite {
        private int posx,posy;
        private boolean active;
        private int frame,nframes;
        private Image[] sprites;
        // constructor. 'nframes' es el número de frames del Sprite.
        public Sprite(int nframes) {
```

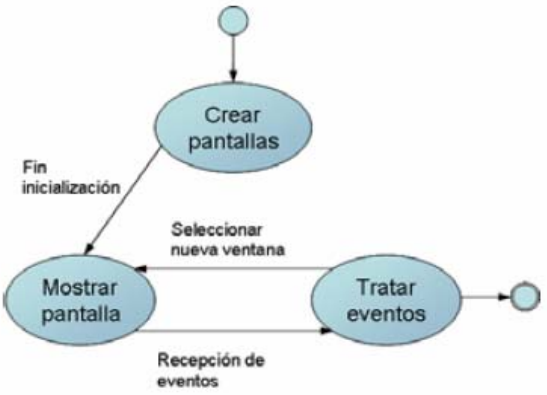
```
        // El Sprite no está activo por defecto.
        active=false;
        frame=1;
        this.nframes=nframes;
        sprites=new Image[nframes+1];
}
public void setX(int x) {
        posx=x;
}
public void setY(int y) {
        posy=y;
        }
int getX() {
        return posx;
        }
int getY() {
        return posy;
        }
int getW() {
        return sprites[nframes].getWidth();
        }
int getH() {
        return sprites[nframes].getHeight();
         }
public void on() {
```

```
        active=true;
        }
public void off() {
        active=false;
         }
public boolean isActive() {
        return active;
        }
public void selFrame(int frameno) {
        frame=frameno;
}
public int frames() {
        return nframes;
}
// Carga un archivo tipo .PNG y lo añade al sprite en
// el frame indicado por 'frameno'
public void addFrame(int frameno, String path) {
        try {
                sprites[frameno]=Image.createImage(path);
        } catch (IOException e) {
                System.err.println("Can`t load the image " + path + ": " +
                e.toString());
        }
        }
```

```
boolean collide(Sprite sp) {
        int w1,h1,w2,h2,x1,y1,x2,y2;
        w1=getW(); // ancho del sprite1
        h1=getH(); // altura del sprite1
        w2=sp.getW(); // ancho del sprite2
        h2=sp.getH(); // alto del sprite2
        x1=getX(); // pos. X del sprite1
        y1=getY(); // pos. Y del sprite1
        x2=sp.getX(); // pos. X del sprite2
        y2=sp.getY(); // pos. Y del sprite2
        if
        (((x1+w1)>x2)&&((y1+h1)>y2)&&((x2+w2)>x1)&&((y2+h2)>y
        1)) {return true;
        } else {
                return false;
        }}
// Dibujamos el Sprite
public void draw(Graphics g) {
        g.drawImage(sprites[frame],posx,posy,Graphics.HCENTER|Grap
        hics.VCENTER);
}
}
```

Navegación guiada por eventos

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



Funcionamiento de la interfaz de usuario de un MIDlet

El funcionamiento se divide en varias fases:

1. **Creación e iniciación:** en esta fase se crean y se inicializan los elementos de la interfaz de usuario. Entre las acciones realizadas destacan las siguientes:

- Creación de pantallas: estas pantallas pueden ser tanto de la interfaz de alto nivel como de la de bajo nivel.
- Creación de comandos y asignación a pantallas: se crean comandos que representan acciones de usuario, mediante llamadas al constructor de la clase Command. Estos comandos se asignan a las diferentes pantallas. Finalmente, se designa un manejador de eventos para cada pantalla, al que se enviarán notificaciones cuando el usuario ejecute un comando.

2. **Visualización de la primera pantalla:** en esta segunda fase, se selecciona la primera pantalla a mostrar al usuario, mediante una llamada al método setCurrent() del objeto Display. Esta selección se realiza dentro del método startApp() mostrándose la pantalla inmediatamente después de que startApp() retorne correctamente.

3. **Captura de eventos y toma de decisiones:** Cuando el gestor de aplicaciones llama al manejador de eventos asociadoa la pantalla, el manejador identifica la acción y actúa en consecuencia. En otros casos los eventos pueden dar paso a otra pantalla.

CAPITULO VIII

8. Desarrollo de la aplicación

8.1. Descripción del capítulo

Para poner en práctica parte de los conocimientos que adquiridos en las investigaciones realizadas para llevar a cabo el presente documento, desarrollaremos un juego para un teléfono celular, el mismo que será el conocido **"Tres en Raya",** se ha seleccionado este juego ya que el objetivo nuestro es demostrar en parte el uso de las clases y objetos usados en este tipo de aplicaciones, así mismo veremos el proceso que se realiza para llevar a cabo la descarga de la aplicación al dispositivo celular.

Para esta aplicación se ha creado en primer lugar un nuevo proyecto llamado Raya3, siguiendo los pasos estudiados anteriormente con la herramienta NetBeans.

Así mismo hemos creado dos archivos que son nuestro MIDlet y el archivo Canvas, al crear estos archivos automáticamente se importan las librerías que necesita cada una de ellas.

Dentro del proyecto hemos creado también un nuevo paquete llamado Image, en el cual se han colocado todos los gráficos se usaremos en la aplicación, cabe destacar que estos archivos deberán tener la extensión ".PNG".

Ahora procederemos a definir cada uno de los archivos que creamos anteriormente.

8.2. Canvas3raya.java

En este archivo definiremos en primer lugar la clase siguiente:

public class Canvas3raya extends Canvas implements CommandListener

Dentro de esta clase definiremos lo que se conoce como constructor, en el que se definirán todas la variables que utilizaremos en nuestra aplicación, como es el caso de las variables para los botones de comandos, y banderas.

Definiremos así mismo las variables que usaremos para manejar las imágenes, una variable por cada gráfico que utilizaremos

Y a continuación procederemos a crear la clase pública canvas que tiene como parámetros Raya3 y midlet. Dentro de esta sección definiremos dos variables cada una de las cuales contendrá el alto y ancho de la pantalla, esto lo utilizaremos para saber en que posición colocar las imágenes que contendrá en pantalla nuestra aplicación.

```
HEIGHT = this.getHeight();
WIDTH = this.getWidth();
```

Así mismo procederemos a la declaración de los botones que nos servirán para la manipulación de nuestra aplicación, es decir definiremos los botones que nos permitirán jugar, para ellos definimos de la siguiente forma.

```
this.uno=(new Command("Arriba IZQ ",Command.OK,1));
this.dos=(new Command("ARRIBA CENTRO ",Command.OK,1));
…
this.reanudar=(new Command("Nuevo Juego ",Command.OK,1));
```

Es importa indicar que para esto hemos tomado como base el tablero propio del juego, al que le hemos asignado un numero por cada posición, en el que por medio de las teclas del celular seleccionaremos en que lugar del tablero deseamos jugar.

Pero como previamente indicamos hasta el momento solo hemos definido los botones, lo que nos queda por hacer es proceder a agregar los botones en la pantalla, esto lo realizamos con las siguientes líneas de código, uno por cada comando que se creó.

```
this.addCommand(this.uno);
this.addCommand(this.dos);
this.addCommand(this.tres);
…
this.addCommand(this.reanudar);
```

Luego definimos el siguiente método public void paint(Graphics g), que es el que nos va a permitir dibujar el ambiente de juego en la pantalla.

```
g.fillRect(0, 0, WIDTH, HEIGHT);
g.drawImage(this.image, 120,150, g.VCENTER | g.HCENTER);
System.out.println("pain JUGAR ");
System.out.println("contador"+contador);
```

En este método definimos también todas la posibilidades de juego que pueden presentarse, es decir, al momento que el usuario juego, se dibuja en pantalla el lugar donde dio su primera partida.

El siguiente método a definir es en el que vamos a controlar las opciones que se le facilitan al usuario, recordemos que definimos los comandos en el contructor, pues bien, en esta parte del programa vamos a controlar cada una de ellas, a continuación se presente parte del código que permite realizar este control.

```
public void commandAction(Command command, Displayable displayable) {
    if(command==this.uno){
        if (j[1]==0){
            j[1]=1;
            contador++;
            bandera=1;
            this.repaint();
```

```
        }   }
            if(command==this.dos){
              if (j[2]==0){
                 j[2]=1;
                 contador++;
                 bandera=1;
                 this.repaint();
        }   }
…
              if(command==this.reanudar){
                inicializar();
              }
```

Como podemos apreciar la ultima parte de esta implementación permite reanudar la partida del juego, esto sirve para volver a jugar sin tener que salir del programa.

8.3. Raya3.java

Ahora explicaremos lo que contiene el otro archivo que creamos que es el MIDlet Raya3.java, en el cual controlaremos los estados por los que pasa el MIDlet, y que sera el que llame al archivo Canvas.

```
public class Raya3 extends MIDlet implements CommandListener{
    private  Canvas3raya canvas;
    private Display display;
    public Raya3(){
       this.display = Display.getDisplay(this);
       this.canvas = new Canvas3raya(this);
    }
```

Como dijimos anteriormente controlaremos los estados por los que pasa el MIDlet, en nuestro programa necesitamos simplemente controlar el momento en que inicia la aplicación.

```
public void startApp() {
    this.display.setCurrent(this.canvas);
}
```

8.4. Explicación del Juego Tres en Raya

Luego de ya terminada nuestra aplicación ósea el juego tres en raya, con sus respectivas verificaciones hacemos las compilaciones correspondientes y se crean los archivos de manifiesto y el archivo jar. En nuestro ejemplo son :

Archivo de manifiesto

Archivo descriptor

Archivo JAR ejecutable

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

jar cmf manifest.mf Raya3.jar canvas3raya.class Raya3.class .timestamp

Luego conectamos al computador el dispositivo en nuestro caso el teléfono motorola PBL U6 y conectamos el cable para pasar los datos.

Después se sincronizan el teléfono con el PC y se empieza a transferir el archivo JAR que es el ejecutable hacia el teléfono, en nuestro ejemplo el archivo Raya3, la transferencia es muy rápida.

Luego la aplicación esta lista para ejecutarse desde el dispositivo móvil.

Ahora nos dedicaremos a explicar el funcionamiento de la aplicación en si, para ello lo ejecutaremos en el emulador del NetBeans. La aplicación se ejecutaría de la misma forma en el teléfono celular.

Al momento de correr la aplicación tendríamos esta primera pantalla.

En la que nos muestra el nombre del MIDlet que creamos en la aplicación, en este caso Raya3, ahora lo que se debe hacer es seleccionar este archivo y esto lo hacemos seleccionando la opción *Launch* que nos presenta en pantalla.

Al momento se seleccionar el juego inmediatamente, tenemos la pantalla en la cual nos presenta el ambiente de juego, y tal como lo hicimos anteriormente haremos uso de la opción *Launch* para escoger una de las opciones que será nuestro primer lanzamiento de juego, recordemos que en la aplicaciones que desarrollamos habíamos definido 10

opciones, en la cuales las 9 primeras representan a cada una de las posiciones del tablero, y la ultima opción sirve para reiniciar la partida de juego.

Como podemos apreciar no nos presentan todas la opciones, por lo que es necesario, desplazar el menú hacia abajo con las flechas de desplazamiento del teclado ya sea del celular o de la computadora.

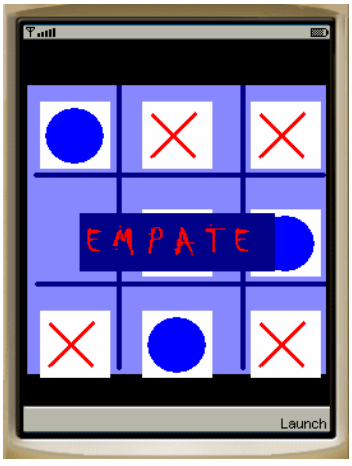
Para señalar una opción tenemos dos alternativas, una de ellas es posicionarnos en la opción y presionar *Launch*, y la otra alternativa es hacer uso del teclado numérico, ya que como podemos apreciar a cada opción se le ha asignado un número que representa su correspondiente en el teclado, cabe indicar que esto solo se lo debe realizar luego de haber presionado *Launch* y se este visualizando el menú de opciones.

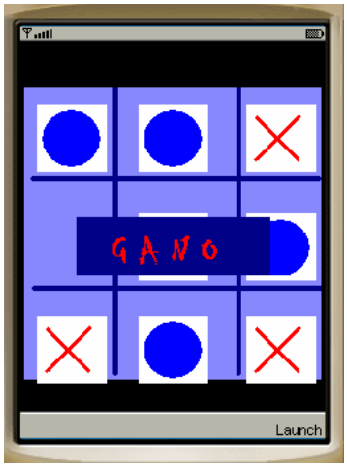
Inmediatamente después de haber seleccionado una de las opciones de juego, será el turno de juego de la aplicación quien internamente seleccionara una de las posiciones.

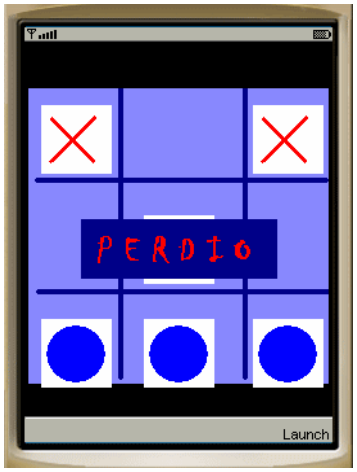
Cabe señalar también que esta predeterminado que el símbolo con el que juega el usuario sea “X” y el símbolo con el que juega la aplicación sea “O”.

Así continuaremos jugando hasta que exista un ganador en la partida de juego, repitiendo los procesos que se han señalado.

Al momento de finalizar el juego, el usuario podría ganar, perder o empatar con la aplicación para a cual, se le indicará con una de las siguientes pantallas, dependiendo de como finalice la partida de juego:







Si se desea jugar nuevamente se escoge la opción 0 y se reanuda una nueva partida.

Conclusiones

El desarrollo de este documento nos ha permitido conocer mucho más acerca de la tecnología J2ME, y la gran importancia que tiene en la actualidad.

De las investigaciones realizadas se ha abstraído lo más esencial que se requiere saber en el desarrollo de aplicaciones para dispositivos celulares básicamente, toda esta información ha sido tratada para obtener al final un documento que sirva como un manual de lo que es la tecnología J2ME, de tal manera que sirva como una guía práctica y sencilla en el desarrollo de aplicaciones.

La elaboración de un juego, ha permitido poner en práctica los conocimientos adquiridos a lo largo del desarrollo de este trabajo, empaparnos más en cómo y cuándo se deben usar cada uno de los elementos que conforman dicha tecnología fortaleciendo de esta manera mucho más sus conceptos.

Por medio de la implementación de la aplicación en un dispositivo móvil, hemos podido aprender que se requiere y los pasos que se deben seguir para poder utilizar nuestro juego en el celular reflejando de esta forma nuestra investigación y cumpliendo así con los objetivos planteados al iniciar este trabajo.

Recomendaciones

En el transcurso de las investigaciones que fueron realizadas para obtener la información suficiente y necesaria para la elaboración de este documento, nos encontramos con que existe muy poca información detallada de J2ME en español, la mayoría de información que se recopilo estaba en idioma extranjero, por lo que se recomienda buscar muy pacientemente información sobre este tema, ya que como dijimos es poca en nuestro idioma pero si existe en Internet.

Al momento de desarrollar la aplicación, las pruebas fueron realizadas en el computador haciendo uso de los simuladores, pero al momento de concluir con el programa he implementarlo en el celular, resulto que la resolución de la pantalla del dispositivo era diferente a la del simulador, lo que provocó que haya un descuadre en la pantalla dibujada de acuerdo al simulador, con la pantalla que se visualizo en el teléfono celular, por lo que se recomienda tener muy en cuenta que depende mucho del dispositivo móvil en el que se descargue la aplicación, en que conceda con la pantalla que se programó tomando como referencia el simulador. Este punto es importante destacar ya que solo existen cuatro tipos de simuladores en los que se pueden realizar las pruebas, mientras que existen infinidad de modelos de celulares con distintos tamaños de resolución en pantalla.

Bibliografía

Airmet, www.api.isiii.es/airmet/mobile2.htm, 03 / Julio / 2006

http://ma.ei.uvigo.es/desma2005/articulos/1043_Bustos.pdf , **http://ma.ei.uvigo.es/desma2005/articulos/1043_Bustos.pdf** , Fecha de ingreso: 03/ Julio/2006

Departamento de Ingeniería Telemática – UC3M - , http://it.uc3m.es, 04/ Julio / 2006

Ingeniería Electrónica U. de A. , http://electronica.udea.edu.co/academicos/proyectos/comunicaciones/juan_juan/pape_r.pdf., 05/ Julio/2006

Java ME - Micro App Development Made Easy, ,http://java.sun.com/j2me/ , 10 / agosto / 2006

Welcome to the the Foundation for Intelligent Physical Agents , http://www.fipa.org , 10 / agosto / 2006

Glosario

A

addCommand: Permite agregar un botón en la pantalla que se esta trabajando.

Alert: Objeto que representa una pantalla de aviso.

Animación: Movimientos que realiza un objetos.

Array: Es un arreglo que nos permite almacenar varios elementos de un mismo tipo bajo el mismo nombre.

C

Canvas: Es la superclase de todas las pantallas que usan las APIs de bajo nivel.

Clases: Define algo que está compuesto por objetos.

CLDC: Connected Limited Device Configuration (CLDC), es una configuración enfocada a dispositivos con restricciones de procesamiento y memoria.

Colisiones: Son el choque de dos objetos dentro de una pantalla de juego.

Command: Es una acción que el usuario realizada sobre la interfaz, se puede decir que es como un botón de Windows.

commandAction: Es un método en el que indicamos la acción que queremos que se realice cuando se produzca un evento.

createImage: Se usa para dibujar en una pantalla Canvas por medio de una imagen.

Ch

ChoiceGroup: Elemento de formulario para seleccionar opciones.

D

destroyApp: Es un método que nos permite cambiar a destruido el estado de un MIDlet.

Displayable: Es una clase que representa a las pantallas de nuestra aplicación.

do/while: Es tipo de estructura de control, el cual realiza la comprobación del bucle al final de la intruccion.

drawArc: Sentencia que se utiliza para dibujar arcos en pantalla.

drawImage: Método de la clase Graphic que nos permite mostrar una imagen

drawLine: Método que nos permite dibujar una línea.

drawRect: Método que nos permite dibujar un rectángulo.

drawRoundRect: Método que nos permite dibujar un rectángulo con puntas redondeadas.

F

fillArc: Método que nos permite dibujar un arco en pantalla

fillRect: Método que nos permite dibujar un rectángulo con color de relleno.

fillRoundRect: Método que nos permite dibujar un rectángulo de puntas redondeadas con color de relleno.

For: Es una estructura de control que nos permite realizar acciones repetitivas, este bucle ejecuta el bloque de sentencias un número determinado de veces.

Form: Clase que nos permite crear pantallas de formularios.

G

Gauge: Elemento de formulario con forma de diagrama de barras, para introducir valores enteros pequeños.

getFont: Método que se invoca para crear una determinada fuente con aspecto, estilo y tamaño.

H

Herencia: Es una característica de la POO en la cual una clase hija llamada subclase hereda los atributos y los métodos de su clase padre.

I

if/else: Estructura de control que nos permite ejecutar un bloque de instrucciones dependiendo si la condición es verdadera o falsa

Image: Es un objeto que nos permite inserter imagines en una pantalla Canvas.

ImageItem: Elemento de formulario que representa una imagen descriptiva

J

JAD: (*Java Archive Descriptor*) es un archivo que forma parte de una aplicación J2ME que contiene diversa información sobre la aplicación.

JAR: Archivo que forma parte de una aplicación J2ME que contiene a la aplicación en sí.

K

keyPressed: Método invocado cuando pulsamos una tecla.

keyReleased: Método invocado cuando soltamos una tecla.

keyRepeated: Método invocado cuando se deja presionada una tecla.

KVM: Su nombre proviene de Kilobyte y es una implementación de Máquina Virtual reducida especialmente orientada a dispositivos con bajas capacidades computacionales y de memoria.

L

List: Clase que nos permite construir pantallas que poseen una lista de opciones, muy útiles para crear menús de manera independiente.

M

MIDlet: Es una aplicación J2ME para dispositivos móviles cuyas limitaciones caen dentro de la especificación MIDP.

MIDP: *Mobile Information Device Profile,* es una configuración CLDC diseñada para dispositivos con limitaciones de memoria.

O

Objetos: Es la instancia de una clase que está compuesta por atributos y métodos.

P

pauseApp: Método que le indica al MIDlet que entre en el estado de pausa.

Polimorfismo: Es una característica de la POO que permite que un mismo método se comporte de diferentes formas.

S

Screen: La clase Screen es la superclase de todas las clases que conforman la interfaz de usuario de alto nivel.

setColor: Método de la clase Graphic que nos permite seleccionar un color.

setCommandListener: Es un método de la clase Displayable que nos permite establece un listener para la captura de eventos.

setFont: Nos permite seleccionar una fuente.

Sprite: Es una clase que representa a un layer animado, el cual suele estar formado por varios fotogramas gráficos que pueden representar desde una animación del movimiento del Sprite en pantalla a distintos puntos de vista de éste.

Sprites: Son elementos gráficos que tiene entidad propia y sobre la que podemos definir y modificar ciertos atributos, como la posición, visibilidad, etc. Los sprites tienen capacidad de movimiento.

startApp: Método que permite que el MIDlet entre en estado activo.

String: Estructura de datos que nos permite almacenar una cadena.

StringItem: Elemento de formulario que representa un texto descriptivo.

Switch: Estructura de control que permite un control condicional múltiple.

T

TextBox: Es una pantalla que nos permite editar texto en ella, se le debe especificar el número de caracteres que se desee que contenga como máximo.

TextField: Elemento de formulario que nos permite introducir texto.

Try: Estructura de control propia de Java que nos permite tomar acciones específicas en caso de error de ejecución en el código.

W

While: Estructura de control que ejecutará el bloque de sentencias mientras se cumpla la condición del bucle.

INDICE GENERAL

CAPITULO V

CAPITULO VI

CAPITULO VIII