PROGRAMACIÓN de JUEGOS PARA CELULARES

- INTRODUCCIÓN AL LENGUAJE JAVA - J2ME
- CLASES Y OBJETOS
- HERENCIA
- POLIMORFISMO
- ANATOMÍA DE UN MIDLET
- LA INTERFAZ GRÁFICA DE BAJO NIVEL
- CONSTRUYENDO EL MAPA DEL JUEGO

e-book

COLECCIÓN COMPUTEACHER

JAIME BE... S.

www.FreeLibros.com

PROGRAMACIÓN DE JUEGOS PARA CELULARES

- INTRODUCCIÓN AL LENGUAJE JAVA - J2ME
- CLASES Y OBJETOS
- HERENCIA
- POLIMORFISMO
- ANATOMÍA DE UN MIDLET
- LA INTERFAZ GRÁFICA DE BAJO NIVEL
- CONSTRUYENDO EL MAPA DEL JUEGO

Todos los nombres propios de programas, sistemas operativos, equipos hardware, captura de pantallas, etc. Que aparecen en este manual son marcas registradas de sus respectivas compañías u organizaciones. Denotamos estos tan solo con fines de divulgación.

PRÓLOGO

J2ME (Java 2 Micro Edition) es la plataforma basada en el lenguaje Java que Sun Microsystems ha creado para la programación de dispositivos inalámbricos pequeños como teléfonos celulares.

Este manual trata sobre como programar juegos para estos dispositivos utilizando J2ME. La primera versión de la especificación MIDP (Movile Information Device Profile), definía los requerimientos mínimos para poder ejecutar programas J2ME, sin embargo, ofrecían poca ayuda a la hora de crear juegos, por lo que había que recurrir a librerías propias de cada fabricante, haciendo necesario crear diferentes versiones de un juego para cada fabricante. La versión 2.0. subsana de alguna manera este problema, y nos ofrece una API mucho más adecuada para la programación de juegos.

1era. Edición, 2007

La presente y disposición en conjunto de:
PROGRAMACIÓN DE JUEGOS PARA CELULARES
Son propiedad del editor. Prohibida su reproducción total o parcial de esta obra, por cualquier medio o método, sin autorización por escrito del Editor.
RUC: : R.U.C. 20428063661
Depósito Legal : 2007 - 09057
EDITADO E IMPRESO : EDITORA Y DISTRIBUIDORA PALOMINO E.I.R.L.
DIRECCIÓN : Calle Tambo de Belén N° 194 (a ½ Cdra. de la Plaza Francia) Lima 01 – Perú
TELEFAX : 431-3896

Depósito en Cta. Cte. Banco Continental a nombre de:
INSTITUTO DE DESARROLLO HUMANO AMEX SAC
Cta. en soles (S/.) : 0100027488

COLECCIÓN COMPUTEACHER

CONTENIDO

PROGRAMACIÓN DE JUEGOS PARA CELULARES 5
INTRODUCCIÓN AL LENGUAJE JAVA - J2ME 5
EL LENGUAJE JAVA 5
VARIABLES Y TIPOS DE DATOS 5
CLASES Y OBJETOS 7
HERENCIA 7
POLIMORFISMO 7
ESTRUCTURAS DE CONTROL 8
ESTRUCTURAS DE DATOS 9
NUESTRO PRIMER MIDLET 10
COMPILANDO EL PRIMER MIDLET 10
ANATOMÍA DE UN MIDLET 12
LA INTERFAZ DE USUARIO DE ALTO NIVEL 12
ELEMENTOS DE LA INTERFAZ DE USUARIO 13
LA CLASE ALERT 14
LA CLASE LIST 14
LA CLASE TEXTBOX 14
LA CLASE FORM 15
LA CLASE STRINGITEM 15
LA CLASE IMAGEITEM 15
LA CLASE TEXTFIELD 16
LA CLASE DATEFIELD 16
LA CLASE CHOICEGROUP 17
LA CLASE GAUGE 17
LA INTERFAZ GRÁFICA DE BAJO NIVEL 17
PRIMITIVAS GRÁFICAS 18
COLORES 18
PRIMITIVAS 18
SPRITES 19
CONTROL DE SPRITES 20
ANIMANDO NUESTRO AVIÓN 24
LECTURA DEL TECLADO 24
THREADS 25
EL GAME LOOP 25
MOVIMIENTO DEL AVIÓN 26
CONSTRUYENDO EL MAPA DEL JUEGO 27
SCROLLING 28

Los Mejores e-book
www.FreeLibros.com

PROGRAMACIÓN DE JUEGOS PARA CELULARES

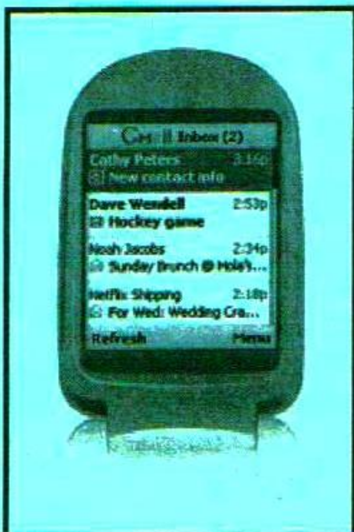
INTRODUCCIÓN AL LENGUAJE JAVA - J2ME

J2EE.- Cuando Sun decidió lanzar su nuevo standard Java, llamado Java2, creó tres diferentes entornos para desarrollo y ejecución de aplicaciones. Estos fueron J2SE, J2EE y J2ME.

J2SE (Java 2 Standard Edition) es, por decirlo de alguna manera, la base de la tecnología Java. Permite el desarrollo de applets (aplicaciones que se ejecutan en un navegador web) y aplicaciones independientes (standalone).

J2ME se basa en los conceptos de configuración y perfil. Una configuración describe las características mínimas en cuanto a la configuración hardware y software. La configuración que usa J2ME es la CLDC (Connected Limited Device Configuration).

J2ME está formado por la configuración CLDC y por el perfil MID (conocido por MIDP o MID Profile). CLDC es una especificación general para un amplio abanico de dispositivos, que van desde PDAs a teléfonos móviles y otros. Un perfil define las características del dispositivo de forma más especifica. MIDP (*Mobile Information Device Profile*) define las APIs y características hardware y software necesarias para el caso concreto de los teléfono móviles



▶ EL LENGUAJE JAVA

El lenguaje Java es un lenguaje completamente orientado a objetos. Todo en Java es un objeto.

▶ VARIABLES Y TIPOS DE DATOS

Las variables nos permiten almacenar información y tal como indica su propio nombre, pueden variar a lo largo de la ejecución del programa. Una variable se define a partir de un nombre y un tipo.

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.
- **float**. Número en punto flotante de 32 bits.
- **double.** Número en punto flotante de 64 bits.
- **char.** Carácter ASCII.
- **boolean.** Valor verdadero o falso.

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

Antes de poder utilizar una variable, hay que declararla, es decir, darle un nombre y un tipo. La siguiente línea declara una variable llamada **vidas** de tipo entero de 32 bits.

```
int vidas;
```

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
- Relaciónales
- Lógicos
- A nivel de bit

Cuando declaramos una variable ésta no contiene ningún valor (realmente si, tiene el valor **null**). Para darle un valor a la variable utilizamos el operador de asignación = (signo de igualdad). Así, para asignar el valor 3 a la variable vidas, procedemos de la siguiente forma.

```
vidas = 3;
```

Observa el punto y coma (;) al final de la línea. En Java cada instrucción acaba con un punto y coma.

Tenemos disponibles otros operadores de asignación:

Operador	Significado
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

El otro tipo de operadores aritméticos son los binarios.

Operador	Significado
a + b	Suma 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. Un operador relacional devuelve un valor de tipo boolean, es decir, verdadero (true) o falso (false).

Operador	Significado
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. Al igual que los operadores relaciones devuelven true o false.

Operador	Significado
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

Cuando veamos la estructura de control if() nos quedará más clara la utilidad de los operadores lógicos.

Los operadores de bits trabajan, como su propio nombre indica, a nivel de bits, es decir, permite manipularlos directamente.

Operador	Significado
a >> b	Desplaza los bits de a hacia la derecha b veces
a << b	Desplaza los bits 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)

Cuando una expresión está compuesta por más de un operador, estos se aplican en un orden concreto. Este orden se llama orden de precedencia de operadores. En la siguiente tabla se muestra el orden en el que son aplicados los operadores.

Orden	Operadores	
1	operadores sufijo	**[] . (params) expr++ expr—**
2	operadores unarios	**++expr —expr +expr -expr ~ !**
3	creación o tipo	**new (type)expr**
4	multiplicadores	*** / %**
5	suma/resta	**+ -**
6	desplazamiento	**<< >> >>>**
7	relacionales	**< > <= >= instanceof**
8>	igualdad	**== !=**
9	bitwise AND	**&**
10	bitwise exclusive OR	**^**
11	bitwise inclusive OR	**|**
12	AND lógico	**&&**
13	OR lógico	**||**
14	condicional	**? :**
15	asignación	**= += -= *= /= %= ^= &= |= <<= >>= >>>=**

Clases y Objetos

Antes de poder crear un objeto hay que definirlo. Un objeto, tal como decíamos antes, 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, utilizamos las dos barras inclinadas (//) para indicar que lo que sigue es 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;
```

Las variables de la clase o atributos son variables como las que vimos en la sección anterior.

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 (tipo que devuelve al ser llamado). También tiene una lista de argumentos o parámetros.

Vamos a clarificar lo visto hasta ahora con un ejemplo.

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

Hemos declarado la clase coche, 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 coche y cómo utilizar sus métodos.

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

Herencia

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 -volviendo al ejemplo del coche- que queremos crear una clase llamada CochePolicia, que además de acelerar y frenar pueda activar y desactivar una sirena. Podríamos crear una clase nueva llamada CochePolicia 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 Coche y que ya contiene algunas de las funcionalidades que queremos incluir en CochePolicia. Veámoslo sobre un ejemplo.

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

Polimorfismo

El polimorfismo es otra de las grandes característica de la POO. 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 coche
class Coche {
        // Atributos de la clase coche
```

```
int velocidad;

        // Métodos de la clase coche
        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;

        }
}
```

ESTRUCTURAS DE CONTROL

Las estructuras de control de Java son similares a las de C. 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. Las siguientes líneas muestran un ejemplo de uso de la estructura **if/else**.

```
if (vidas == 0) {
    terminar = true;
} else {
    vidas—;
}
```

En este ejemplo, si la variable vidas vale 0, la variable terminar tomará el valor true. En otro caso, se decrementa el valor de la variable vidas.
La otra estructura condicional es **switch**, que permite un control condicional múltiple. Tiene el formato siguiente.

```
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. Observa 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. Lo veremos mejor con un ejemplo.

```
switch (posicion) {
    case 1:
            medalla = «oro»;
            break;
    case 2:
            medalla = «plata»;
            break;
    case 3:
            medalla = «bronce»;
            break;
    default:
            medalla = «sin medalla»;
break;
}
```

Las estructuras que hemos visto hasta ahora nos permiten tomar decisiones. Las siguientes que vamos a ver nos van a permitir 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.

```
[
for (i=1 ; i<=10 ; i++) {
    suma+=i;
}
```

Este ejemplo de código suma los 10 primero números. La variable i lleva la cuenta, es decir, es el

contador del bucle. En la primera sección nos encargamos de inicializar la variable con el valor 1. La segunda sección es la condición que ha de darse para que se continúe la ejecución del bucle, en este caso, mientras i sea menor o igual a 10, se estará ejecutando el bucle. La tercera sección es la encargada de incrementar la variable en cada vuelta.

El siguiente bucle que te voy a presentar es el bucle while y tiene la siguiente estructura.

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

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

```
vueltas = 10;
while (vueltas > 0) {
    vueltas—;
}
```

A la entrada del bucle, la variable vueltas tiene el valor 10. Mientras el valor de esta variable sea mayor que 0, se va a repetir el bloque de código que contiene. En este caso, el bloque de código se encarga de decrementar la variable vuelta, por lo que cuando su valor llegue a 0, no volverá a ejecutarse. Lo que estamos haciendo es simular un bucle for que se ejecuta 10 veces.

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

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

El siguiente ejemplo, es igual que el anterior. La diferencia entre ambos es que con el bucle do/while, el código se ejecutará siempre al menos una vez, ya que la comprobación se hace al final, mientras que con el bucle while, es posible que nunca se ejecute el código interno si no se cumple la condición.

```
vueltas = 10;
do {
    vueltas—;
} while(vueltas > 0);
```

Veamos una última estructura propia de Java (no existe en C) y que nos permite ejecutar un código de forma controlada. Concretamente nos permite tomar acciones específicas en caso de error de ejecución en el código.

```
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 por decirlo de una forma sencilla, realmente, 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.**

▶ ESTRUCTURAS DE DATOS

Ya hemos visto los tipos de datos que soporta Java. Ahora vamos a ver un par de estructuras muy útiles. Concretamente la cadena de caracteres y los arrays.

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

```
"En un lugar de La Mancha..."
```

Es una cadena de caracteres. 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 un texto";
```

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 (realmente un objeto de tipo String) haciendo uso de su método lenght.

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

El otro tipo de datos que veremos a continuación es el array. Un array nos permite almacenar varios elementos de un mismo tipo bajo el mismo nombre. Imagina 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.

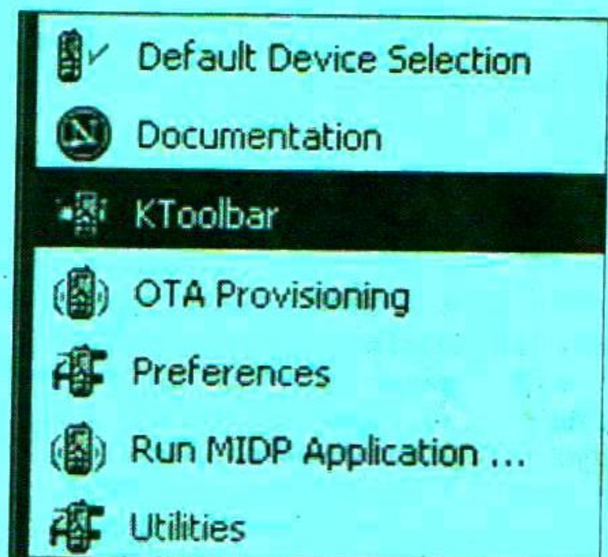
▶ NUESTRO PRIMER MIDLET

Existen diferentes herramientas válidas para construir programas bajo el standard J2ME, como el propio «Sun One Studio» de Sun Microsystems o «Jbuilder» de Borland. Nosotros vamos a valernos del «J2ME Wireless Toolkit 2.0» que proporciona Sun. Este entorno es el más sencillo de utilizar, y aunque no nos ofrece una gran potencia a la hora de desarrollar aplicaciones, no nos distraerá con grandes complejidades del principal objetivo que es aprender a hacer aplicaciones (juegos) en J2ME.

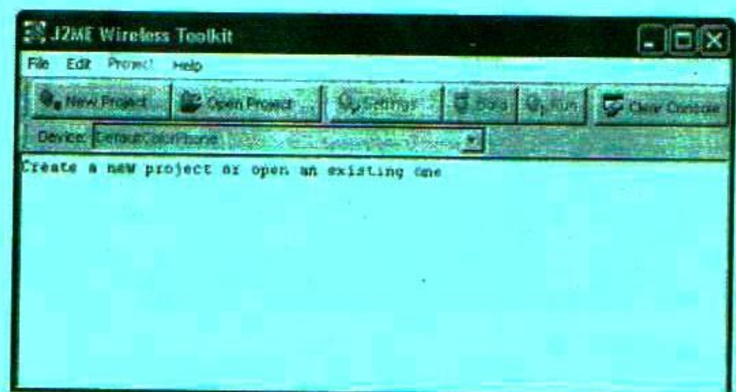
Para instalar J2ME Wireless Toolkit, primero hemos de instalar el entorno de programación de J2SE (JDK). Puedes descargar la última versión de JDK desde la URL http://java.sun.com/j2se/downloads.html. Una vez descargado e instalado, estaremos en condiciones de descargar e instalar J2ME desde la URL http://java.sun.com/j2me/download.html. El entorno de desarrollo que nos provee el Wireless Toolkit se llama KToolBar.

▶ COMPILANDO EL PRIMER MIDLET

Vamos a construir paso a paso nuestro primer MIDlet usando esta herramienta. Tras la instalación del wireless toolkit, tendremos un nuevo submenú en el menú inicio con un aspecto similar a éste:



Selecciona la aplicación **KToolBar** e inicializa el entorno. Verás aparecer la ventana del entorno.



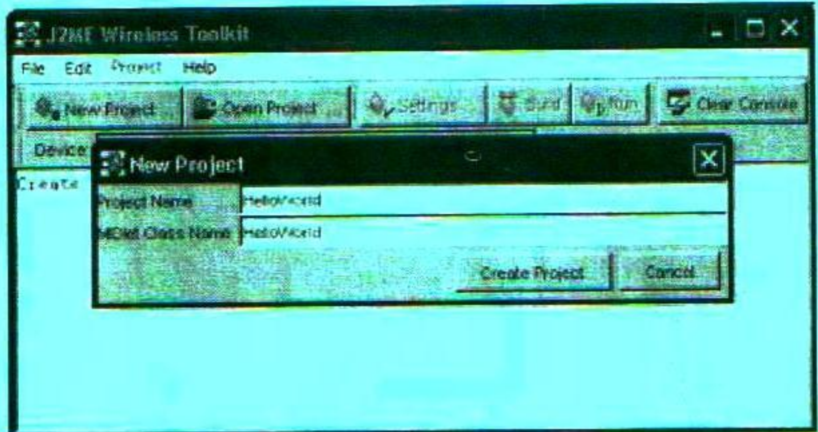
Vamos a crear un nuevo proyecto, así que pulsamos el botón **New Project**. Nos solicitará un nombre para el proyecto y otro para la clase principal de la aplicación.

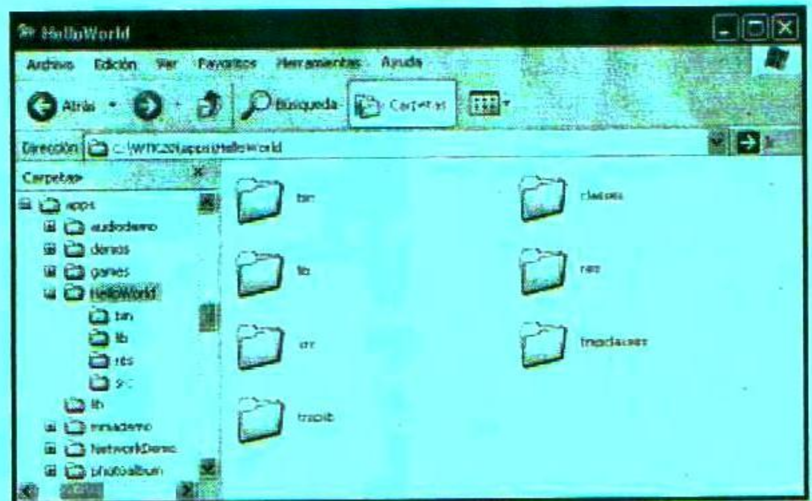
Tanto el proyecto como la clase principal se llamarán HelloWorld, así que introducimos este nombre en ambos cuadros de texto y pulsamos el botón **Create Project**. En este momento **KToolBar** crea la estructura de directorios necesaria para albergar el proyecto.

Cada una de las carpetas creadas tiene una misión concreta. Por ahora nos bastará saber que nuestros archivos fuente irán emplazados en el directorio src, y los recursos necesarios como gráficos, sonidos, etc... se alojarán en el directorio res.

A diferencia de otros entornos de programación, **KToolBar** no cuenta con un editor integrado para editar los programas, por lo tanto vamos a utilizar uno externo. Puedes utilizar el bloc de notas de Windows o tu editor favorito. Personalmente utilizo Crimson Editor (http://www.crimsoneditor.com/), que tiene soporte para Java.

Utilizando tu editor favorito introduce el programa siguiente:






```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class HelloWorld extends MIDlet
implements CommandListener {
	private Command exitCommand;
	private Display display;
	private Form screen;

	public HelloWorld() {
		// Obtenemos el objeto Display
del midlet.
display = Display.getDisplay(this);

		// Creamos el comando Salir.
exitCommand = new Command(«Salir»,
Command.EXIT,2);
		// Creamos la pantalla principal (un
formulario)
screen = new Form(«HelloWorld»);

		// Creamos y añadimos la cadena de
texto a la pantalla
StringItem saludo = new StringItem(«»,»Hola
Mundo...»);
screen.append(saludo);

		// Añadimos el comando Salir e
indicamos que clase lo manejará
screen.addCommand(exitCommand);
screen.setCommandListener(this);
}
	public void startApp() throws
MIDletStateChangeException {
		// Seleccionamos la pantalla a
mostrar
display.setCurrent(screen);
}

		public void pauseApp() {
}

		public void destroyApp(boolean
incondicional) {
}

		public void commandAction(Command c,
Displayable s) {
		// Salir
			if (c == exitCommand) {
				destroyApp(false);
				notifyDestroyed();
		}
}
}
```

No es necesario que trates de comprender el programa ahora. Entraremos en más detalles un poco más adelante. Por ahora simplemente lo vamos a almacenar en el directorio src que ha creado KToolBar con el nombre HelloWorld.java. Es importante que el nombre sea exactamente éste incluidas mayúsculas y minúsculas. Esto es así, ya que el nombre de la clase principal tiene que ser idéntico al nombre del archivo que lo contiene.

Una vez hecho esto, volvemos al entorno KTooBar y pulsamos el botón **Build.** Si todo va bien, aparecerá el texto Build Complete. Ya tenemos nuestro programa compilado y podemos ejecutarlo en el emulador. En el desplegable **Device** puedes seleccionar el emulador que quieres utilizar. El DefaultColorPhone tiene soporte de color, así que te resultará más atractivo. Pulsa el botón **Run.** Verás aparecer un emulador con forma de teléfono celular. En la pantalla aparece un menú con un sólo programa llamado **HelloWorld.** Pulsa select para ejecutarlo.Deberías ver como aparece la frase Hola Mundo... en la pantalla.

Ahora que hemos comprobado que el programa funciona en el emulador, estamos listos para empaquetar el programa y dejarlo listo para descargar a un dispositivo real. En **KToolBar** despliega el menú **project,** y selecciona **create package** del submenú **package**. KToolBar nos informa de que ha creado los archivos **HelloWorld.jar** y **HelloWorld.jad** dentro del directorio **bin.** Estos son los archivos que habremos de transferir al teléfono celular.

▶ Anatomía de un MIDlet

Si estas familiarizado con la programación de applets, conoces las diferencias que tiene con respecto a una aplicación Java normal. La primera es que un applet se ejecuta sobre un navegador web. Otra importante es que, a diferencia de un programa Java estándar, un applet no tiene un método main(), además, un applet tiene que ser una subclase de la clase Applet, e implementar unos métodos concretos (init, start, stop, destroy). En este sentido, un MIDlet es más parecido a un applet que a una aplicación Java estándar. Un MIDlet tiene que ejecutarse en un entorno muy concreto (un dispositivo con soporte J2ME), y tampoco cuenta con un método main(). Un MIDlet tiene que heredar de la clase MIDlet e implementar una serie de métodos de dicha clase.

Concretamente, la clase de la que ha de heredar cualquier MIDlet es javax.microedition.midlet.MIDlet Hay tres métodos heredados que son particularmente importantes:

- startApp()
- pauseApp()
- destroyApp()

Un MIDlet puede estar en tres estados diferentes: en ejecución, en pausa o finalizado. Dependiendo del estado en el que esté, la máquina virtual llamará al método correspondiente, es decir, startApp() cuando entre en ejecución, pauseApp() cuando el MIDlet entre en pausa y destroyApp() a la finalización del MIDlet. Fíjate en nuestro ejemplo cómo hemos implementado los tres métodos. Incluso si no vamos a hacer uso de ellos, es obligatorio declararlos.

En nuestro programa de ejemplo, no sólo importamos la clase MIDlet, también hemos importado las clases de javax.microedition.lcdui.*. Estas clases dan soporte para la interfaz de usuario. Nos va a permitir controlar la pantalla del dispositivo y también la entrada/salida desde el teclado. En el siguiente capítulo nos introduciremos en más profundidad en la interfaz de usuario

La interfaz de usuario de alto nivel

J2ME se sustenta en dos APIs, por un lado CLDC que hereda algunas de las clases de J2SE, y MIDP que añade nuevas clases que nos permitirán crear interfaces de usuario.

Las clases que nos ofrece CLDC, son las más importantes de los siguientes paquetes de J2SE:

java.lang

java.util

java.io

Además cuenta con el «Generic Conection Framework» que ofrece posibilidades de conexión y comunicación.

Por su parte la API MIDP también hereda de J2SE las clases:

Timer

TimerTask

Además MIDP añade los siguientes paquetes:

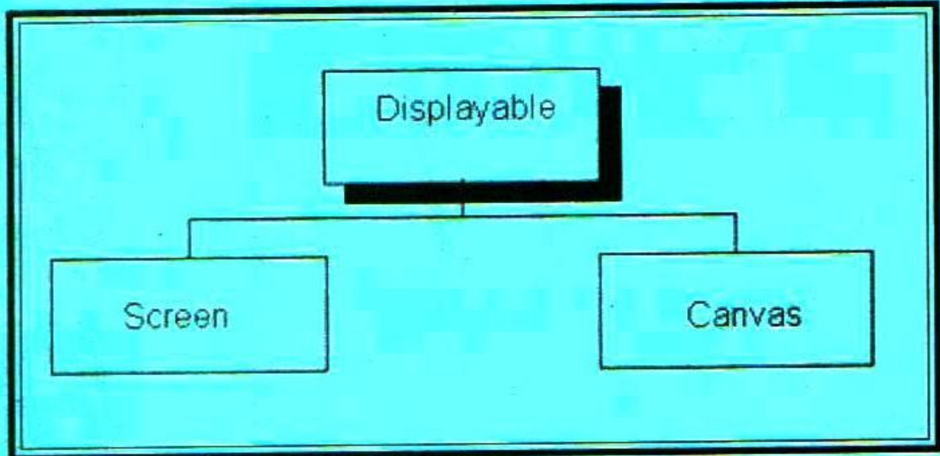
javax.microedition.midlet

javax.microedition.lcdui

javax.microedition.io

javax.microedition.rms

- El paquete **javax.microedition.midlet**, es el más importante de todos. Sólo contiene una clase: la clase MIDlet, que nos ofrece un marco de ejecución para nuestras aplicaciones sobre dispositivos móviles.
- El paquete **javax.microedition.lcdui** nos ofrece una serie de clases e interfaces de utilidad para crear interfaces de usuario. Es algo así como un pequeño entorno gráfico similar al AWT, pero, evidentemente, mucho más limitado. Básicamente, nos permite dos tipos de entorno, por un lado podremos trabajar con 'Screens' sobre las que podremos colocar elementos de la interfaz de usuario, como textos, menus, etc., por otro, podremos basar nuestras aplicaciones en 'Canvas' sobre las que podemos trabajar a nivel gráfico, es decir, a más bajo nivel. Tanto **Screen** como **Canvas** son objetos que heredan de la clase **'Displayable'**.



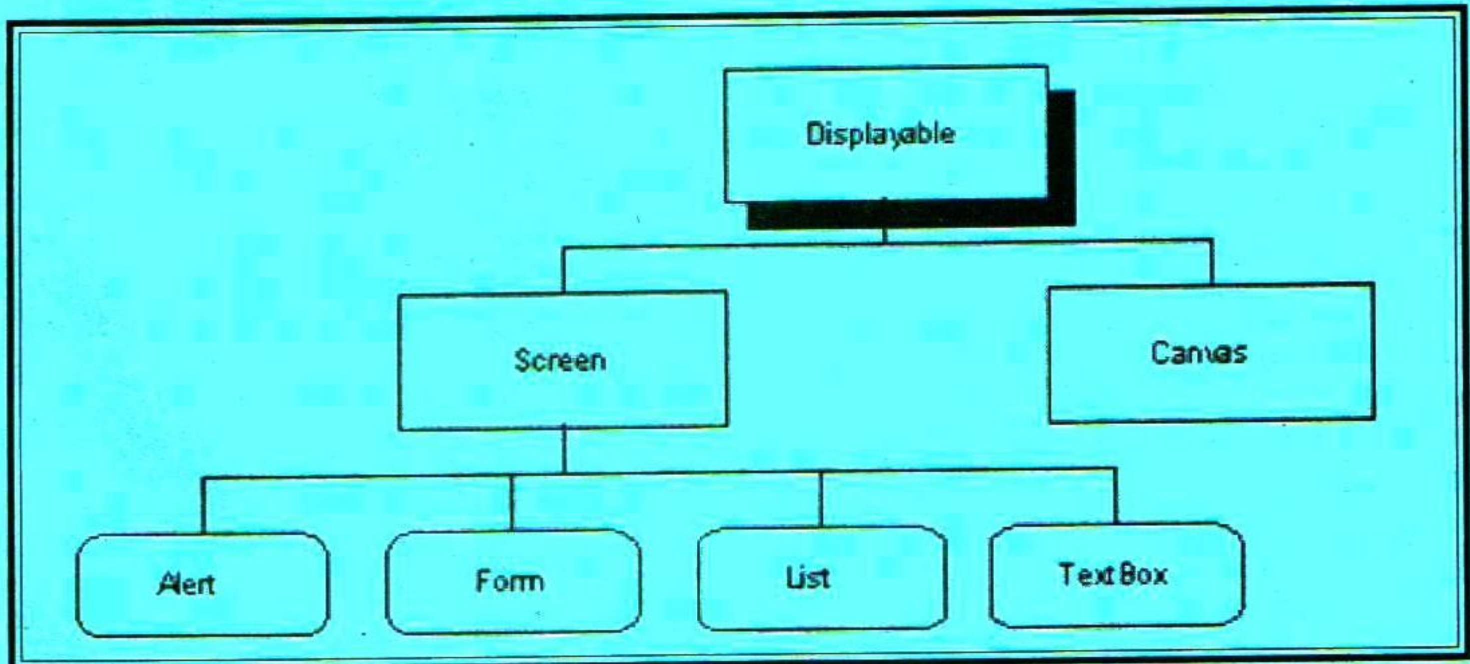
Todo aquello que puede ser mostrado por la pantalla del dispositivo hereda de forma directa o indirecta de la clase **Displayable**.

Para el desarrollo de juegos, el objeto **Canvas** es el que nos va a resultar más interesante.

▶ Elementos De La Interfaz De Usuario

Ahora que tenemos una idea básica sobre el funcionamiento de un MIDlet, pasaremos a describir los elementos gráficos de los que disponemos para crear interfaces de usuario.

Como ya vimos, la clase Screen hereda directamente de Displayable y permite crear las interfaces gráficas de alto nivel. Un objeto que herede de la clase Screen será capaz de ser mostrado en la pantalla. Disponemos de cuatro clases que heredan de Screen y que nos sirven de base para crear las interfaces de usuario. Son Alert, Form, List y TextBox.



Como ya vimos, la clase **Screen** hereda directamente de **Displayable** y permite crear las interfaces gráficas de alto nivel. Un objeto que herede de la clase Screen será capaz de ser mostrado en la pantalla. Disponemos de cuatro clases que heredan de Screen y que nos sirven de base para crear las interfaces de usuario. Son **Alert, Form, List y TextBox**.

▶ La clase Alert

Permiten mostrar una pantalla de texto durante un tiempo o hasta que se produzca un comando de tipo **OK**. Se utiliza para mostrar errores u otro tipo de mensajes al usuario.

Para crear una alerta utilizamos su constructor que tiene la siguiente forma:

Alert (String título, String texto_alerta, Image imagen_alerta, AlertType tipo_alerta)

El título aparecerá en la parte superior de la pantalla. El texto de alerta contiene el cuerpo del mensaje que queremos mostrar. El siguiente parámetro es una imagen que se mostrará junto al mensaje. Si no queremos imagen le pasamos null como parámetro. El tipo de alerta puede ser uno de los siguientes:

- **ALARM**
- **CONFIRMATION**
- **ERROR**
- **INFO**
- **WARNING**

La diferencia entre uno y otro tipo de alerta es básicamente el tipo de sonido o efecto que produce el dispositivo. Vemos un ejemplo:

Alert alerta = new Alert ("Error","El dato no es válido", null, AlertType.ERROR);

Y la siguiente línea mostrará la alerta:

display.setCurrent(alerta);

Lo hará durante 1 ó 2 segundos. Se puede establecer el tiempo del mensaje con el método

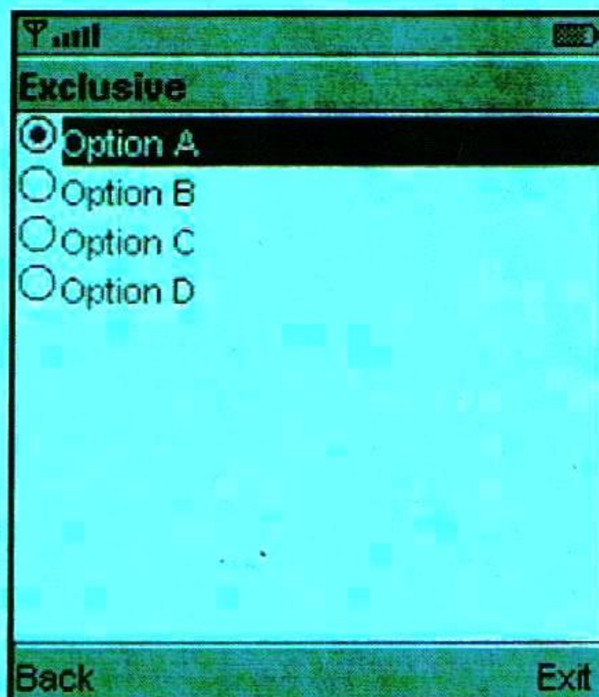
setTimeout(int tiempo)

donde podemos especificar el tiempo en milisegundos. También podemos hacer que el mensaje se mantenga hasta que se pulse un botón del dispositivo de la siguiente manera:

alerta.setTimeout(Alert.FOREVER);

▶ La clase List

Mediante la clase List podemos crear listas de elementos seleccionables



Veamos el constructor:

List (String título, int tipo_lista, String[] elementos, image[] imágenes)

Los posibles tipos de lista son:

·EXCLUSIVE - Sólo se puede seleccionar un elemento
·IMPLICIT - Se selecciona el elemento que tiene el foco
MULTIPLE - Permite la selección múltiple

▶ La clase TextBox

La clase TextBox permite introducir y editar texto a pantalla completa. Es como un pequeño editor de textos.

Veamos el constructor:

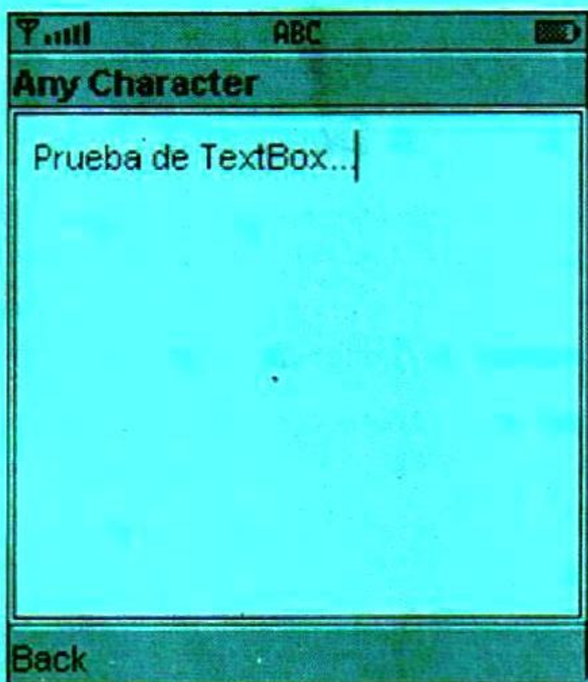
TextBox (String título, String texto, int tamaño_max, int limitación)

Las limitaciones pueden ser alguna de los siguientes:

ANY - Sin limitación
EMAILADDR - Sólo una dirección de email
NUMERIC - Sólo se permiten números
PASSWORD - Los caracteres no serán visibles
PHONENUMBER - Sólo números de telefono
URL - Sólo direcciones URL

El parámetro **tamaño_max** indica el máximo número de caracteres que se pueden introducir. El parámetro texto es el texto inicial que mostrará la caja.

TextBox texto = new TextBox ("Mensaje", "", 256, TextField.ANY);



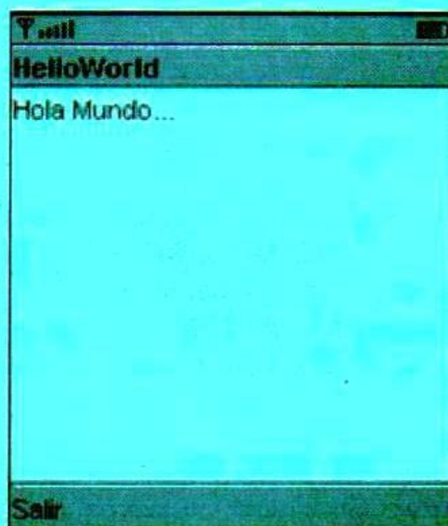
▶ La clase Form

Un Form es un elemento de tipo contenedor, es decir, es capaz de contener una serie de elementos visuales con los que podemos construir interfaces más elaboradas. Los elementos que podemos añadir a un formulario son:

- **StringItem**
- **ImageItem**
- **TextField**
- **DateField**
- **ChoiceGroup**
- **Gauge**

▶ La clase StringItem

Su función es añadir etiquetas de texto al formulario.



El constructor de la clase **StringItem** es el siguiente:

StringItem (String etiqueta, String texto)

Si sólo queremos mostrar un texto, sin etiqueta, paramos una cadena vacía como primer parámetro («»).

Como vimos antes, sólo hay que utilizar el método append() de la clase Form para añadir el texto.

La clase StringItem nos provee además de dos métodos:

String getText()

void setText(String texto)

El primer método devuelve el texto de un StringItem, el segundo, establece el texto que le pasamos como parámetro

▶ La clase ImageItem

Con esta clase podemos añadir elementos gráficos a un formulario. El constructor tiene la siguiente forma:

ImageItem (String etiqueta, Image img, int layout, String texto_alternativo)

El parámetro texto_alternativo es un texto que se mostrará en el caso en el que no sea posible mostrar el gráfico. El parámetro layout indica cómo se posicionará el gráfico en la pantalla. Sus posibles valores son:

LAYOUT_DEFAULT

LAYOUT_LEFT

LAYOUT_RIGHT

LAYOUT_CENTER

LAYOUT_NEWLINE_BEFORE

LAYOUT_NEWLINE_AFTER

Las cuatro primeras son auto explicativas. **LAYOUT_NEWLINE_BEFORE** añade un salto de línea antes de colocar la imagen. **LAYOUT_NEWLINE_AFTER** hace precisamente lo contrario, primero añade la imagen y después un salto de línea.

Para cargar una imagen, utilizamos el método createImage() de la clase Image. Veamos un ejemplo:

```
Image img;
try {
img = Image.createImage(«/logo.png»);
} catch (IOException e) {
System.err.println(«Error: « + e);
}
```

Añadir la imagen al formulario es similar a cómo lo hacemos con un **StringItem**:

```
ImageItem img = new ImageItem («», «/logo.png»
,ImageItem.LAYOUT_DEFAULT, «logotipo»);
formulario.append(img);
```

Hay que tener en cuenta que las imágenes han de almacenarse en el directorio '**res**' que crea **KToolBar**, por lo tanto la barra (**/**) hace referencia a la raíz de este directorio.

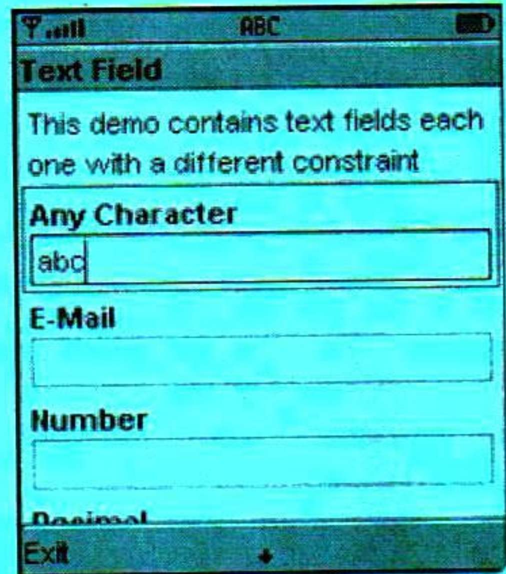
LA CLASE TEXTFIELD

La clase TextField es muy similar a la clase TextBox que ya vimos anteriormente. La principal diferencia es que TextField está diseñada para integrarse dentro de un formulario en vez de ocupar toda la pantalla.

El constructor de esta clase es similar al de TextBox:

TextField (String etiqueta, String texto, int tamaño_max, int limitación)

Los parámetros tienen el mismo significado que TextBox, exepto el primero, que permite especificar una etiqueta.



LA CLASE DATEFIELD

Con **DateField** tenemos una herramienta muy intuitiva que permite la entrada de datos de tipo fecha o tipo hora.

DateField (String etiqueta, int modo)

El parámetro modo puede tomar cualquiera de los siguientes valores:

DATE

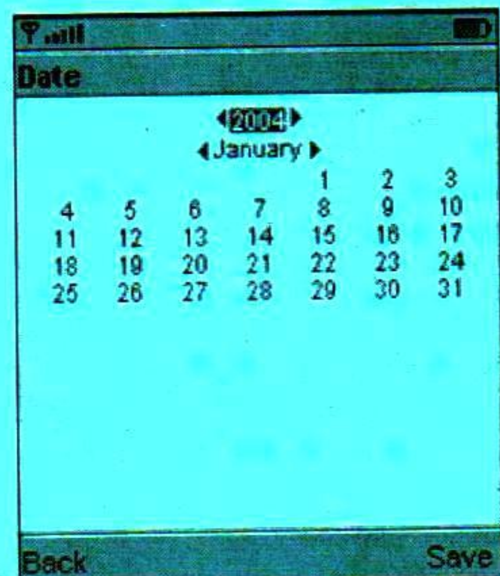
TIME

DATE_TIME

Para seleccionar la entrada de una fecha o una hora.

```
DateField fecha=new
DateField(«fecha»,DateField.DATE);
formulario.append (fecha);
```

La clase DateField nos provee estos dos métodos:



Date getDate()

void setDate (Date fecha)

El primer método recupera el valor del elemento **DateField**, y el segundo lo establece

▶ La clase ChoiceGroup

Este elemento es similar a la clase List, pero al igual que **DateField**, puede incluirse en un formulario, de hecho, su constructor es básicamente el mismo que el de List:

ChoiceGroup (String etiqueta, int tipo_lista, String[] elementos, image[] imágenes)

Excepto el primer parámetro (que ya conocemos), el resto es exactamente el mismo que el de la clase List.

```
String[] estados ={«Casado»,»Soltero»,
»Divorciado»,»Viudo»};
ChoiceGroup estado = new ChoiceGroup
(«Estado», List.EXCLUSIVE, estados, null);
screen.append(estado);
```

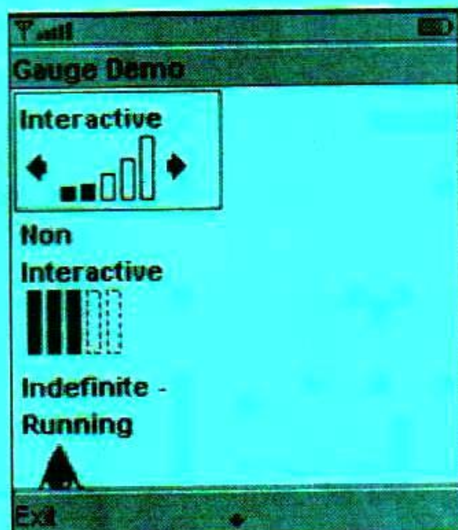
▶ La clase Gauge

La clase Gauge representa un elemento tipo barra de estados que permite indicar un valor gráficamente.

El constructor tiene la siguiente forma:

Gauge (String etiqueta, bolean interactivo, int val_max, int val_ini)

Los parámetros val_ini y val_max indican el valor inicial y el valor máximo de la barra gráfica. El parámetro interactivo si está a true, permitirá al usuario modificar el valor de la barra, si no, sólo podrá mostrar información.



La clase Gauge nos ofrece cuatro métodos muy útiles:

int getValue()

void setValue(int valor)

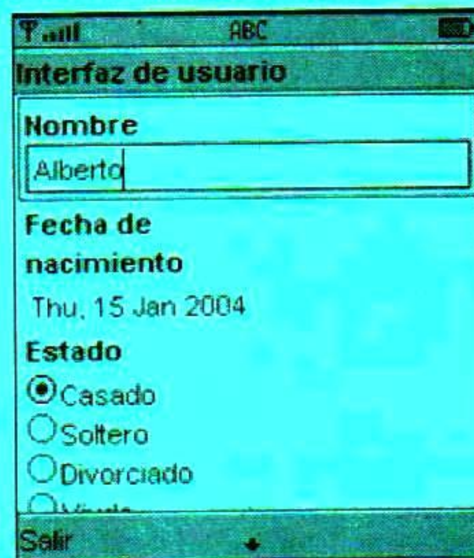
int getMaxValue()

void setMaxValue(int valor)

Las dos primeros para establecer y recoger el valor del Gauge, y las otras tienen el cometido de establecer y recoger el valor máximo del Gauge.

```
Gauge estado = new Gauge («estado», false, 1,
100);

fomulario.append(estado);
```

El siguiente grafico muestra el uso de varios elementos en un formulario a la vez.

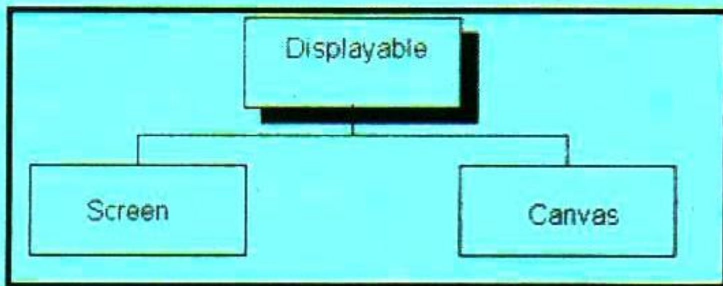


La Interfaz Gráfica de Bajo Nivel

Cuando se diseñó J2ME, los ingenieros de Sun ya sabían que una de las claves para que su tecnología tuviera éxito era que tenía que ser capaz de hacer funcionar juegos, y hacerlo de forma medianamente decente. Para ello debían dotar a los MIDlets de la capacidad de controlar la pantalla al más bajo nivel posible, es decir, a nivel gráfico.

En el capítulo anterior, hemos profundizado en las clases que nos permitían trabajar con interfaces de usuario. Todas ellas derivaban de la clase Screen, que a su vez derivaba de **Displayable.** Si nos fijamos en el diagrama de clases vemos como de Displayable también deriva la clase **Canvas.**

Esta clase es capaz de mostrar información gráfica a nivel de píxel. Es por ellos que la llamamos interfaz de bajo nivel.



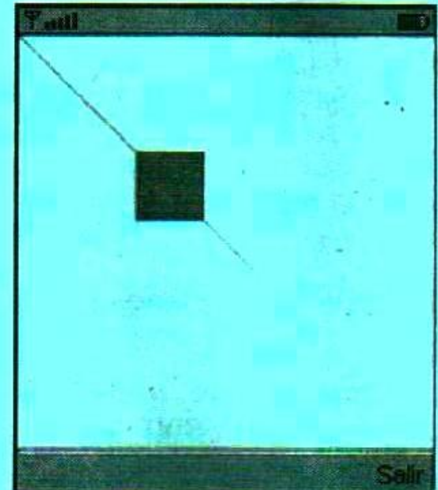
Básicamente podemos realizar tres operaciones sobre un Canvas:

- Dibujar primitivas gráficas
- Escribir texto
- Dibujar imágenes

Vamos a cubrir estas operaciones, y así sentar las bases necesarias para abordar las materias concretas relativas a la programación de videojuegos.

Utilizaremos un código de ejemplo para ir explicando sobre él los conceptos básicos.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class Canvas1 extends MIDlet
implements CommandListener {

private Command exitCommand;
private Display display;
private SSCanvas screen;

public Canvas1() {
      display=Display.getDisplay(this);
exitCommand = new
Command(«Salir»,Command.SCREEN,2);

      screen=new SSCanvas();
screen.addCommand(exitCommand);
      screen.setCommandListener(this);
}

public void startApp() throws
MIDletStateChangeException {
display.setCurrent(screen);
}
public void pauseApp() {}
public void destroyApp(boolean unconditional)
{}

public void commandAction(Command c,
Displayable s) {

      if (c == exitCommand) {
      destroyApp(false);
      notifyDestroyed();
}
}
}
class SSCanvas extends Canvas {
public void paint(Graphics g) {
      g.setColor(255,255,255);
      g.fillRect (0, 0, getWidth(), getHeight());

q.setColor(10,200,100);

      g.drawLine (0, 0, 100, 100);
      g.fillRect (50, 50, 30, 30);
}
}
```



Primitivas Gráficas

▶ Colores

Para especificar el color que queremos utilizar al dibujar en la pantalla utilizamos el método **setColor()** de la clase **Graphics**.

void setColor(int rojo, int verde, int azul)

Los parámetros de color tienen un rango de 0 a 255.

Pero, no todos los dispositivos poseen pantalla a color. El siguiente método establece un tono de gris dentro de la grama de grises de una pantalla monocromo.

void setGrayScale(int tono)

El parámetro tono puede tomar un valor entre 0 y 255

▶ Primitivas

Aunque no vamos a necesitarlas muy a menudo para desarrollar nuestros juegos, es interesante que conozcamos las primitivas básicas con las que contamos para dibujar.

void drawLine (int x1, int y1, int x2, int y2)

Este método dibuja una línea que une los puntos de la coordenada (x1, y1) de la pantalla y la coordenada (x2, y2).

void drawRect (int x, int y, int ancho, int alto)

Con **drawRect()** podemos dibujar rectángulos. Los parámetros x e y indican cual será la esquina superior izquierda del rectángulo, mientras que los otros dos parámetros nos indican el ancho y el alto que tendrá el rectángulo en píxeles.

▶ TEXTO

Aunque estemos trabajando a nivel gráfico, es muy probable que necesitemos mostrar información textual en pantalla. Un ejemplo claro es el marcador de puntuación de un juego.

El método que nos permite escribir texto en un **Canvas** es:

void drawString (String texto, int x, int y, int ancla)

El primer parámetro es el texto que queremos mostrar. Los parámetros x e y es la posición donde queremos situar el texto dentro de la pantalla. El cuarto parámetro indica cuál es el punto de referencia para situar el texto en las coordenadas deseadas. Los valores posibles son TOP, BASELINE y BUTTOM para la posición vertical del texto y LEFT, HCENTER y RIGHT para la posición horizontal del texto. Por ejemplo, si quisiéramos posicionar un texto en la posición 100,100 y con centro en la esquina superior izquierda utilizaremos la siguiente línea:

g.drawString («Hola.», 100, 100, Graphics.LEFT | Graphics.TOP);

▶ IMÁGENES

Las primitivas gráficas nos permiten ya cierta capacidad de dibujar gráficos, pero para crear un videojuego, necesitamos algo más elaborado. La clase Graphics nos ofrece dos métodos:

public static Image createImage(String name) throws IOException

El método createImage() carga un archivo gráfico en formato .PNG. Dependiendo del dispositivo podrá soportar más formatos gráficos, pero en principio, al menos, debe soportar el formato .PNG. Recuerda que los gráficos (y el resto de recursos, como sonidos, etc...) han de estar en el directorio 'res'.

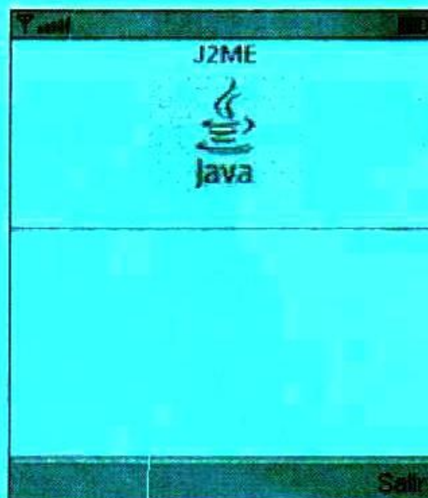
boolean drawImage(Image img, int x, int y, int ancla)

El último parámetro es similar al de drawString(). Sus posibles valores son **TOP**, **VCENTER** y **BUTTON** para la posición vertical y **LEFT**, **HCENTER**, **RIGHT** para la posición horizontal.

Veamos un ejemplo:

Image img = Image.createImage(«\logo.png»);

g.drawImage (img, 10, 10, Graphics.HCENTER, Graphics.VCENTER);



▶ SPRITES

Ya dispones de las herramientas necesarias para emprender la programación de videojuegos

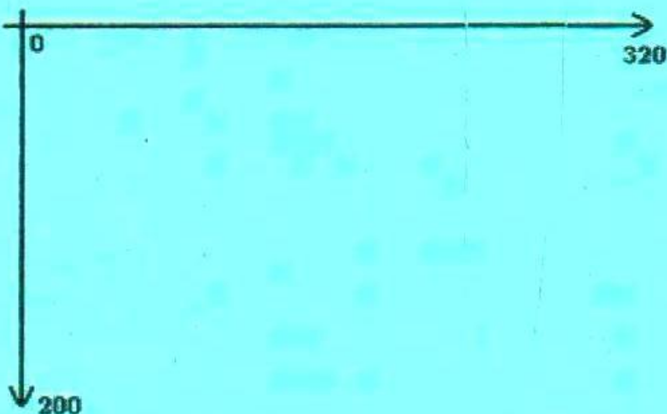
Desarrollaremos un pequeño videojuego. Va a ser un juego sin grandes pretensiones, pero que nos va a ayudar a entender los diferentes aspectos que encierra este fascinante mundo. Nuestro juego va a consistir en lo que se ha dado en llamar shooter en el argot de los videojuegos.

Quizás te resulte más familiar «matamarcianos». En este tipo de juegos manejamos una nave que tiene que ir destruyendo a todos los enemigos que se pongan en su camino.

En nuestro caso, va a estar ambientado en la segunda guerra mundial, y pilotaremos un avión que tendrá que destruir una orda de aviones enemigos.

Seguro que alguna vez has jugado a Space Invaders. En este juego, una pequeña nave situada en la parte inferior de la pantalla dispara a una gran cantidad de naves enemigas que van bajando por la pantalla hacia el jugador. Pues bien, nuestra nave es un sprite, al igual que los enemigos, las balas y los escudos. 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. Es como el juego de los barquitos, en el que 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. La figura siguiente muestra el eje de coordenadas en una pantalla con una resolución de 320 por 200 píxeles.



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).

Imagina ahora que jugamos a un videjuego en el que manejamos a un hombre. 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.

Otra característica muy interesante de los sprites es que nos permiten detectar colisiones entre ellos. Esta capacidad es realmente interesante si queremos conocer cuando nuestro avión ha chocado con un enemigo o con uno de sus misiles.

▶ CONTROL DE SPRITES

Vamos a realizar una pequeña librería (y cuando digo pequeña, quiero decir realmente pequeña) para el manejo de los sprites. Luego utilizaremos esta librería en nuestro juego, por supuesto, también puedes utilizarla en tus propios juegos, 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;
```

Necesitamos la coordenada en pantalla del sprite (que almacenamos en posx y posy. 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 puedes 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 puedes 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;
}
```

Necesitaremos un método que active el sprite, al que llamaremos on(), y otro para desactivarlo, que como podrás imaginar, 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());
}
}
```

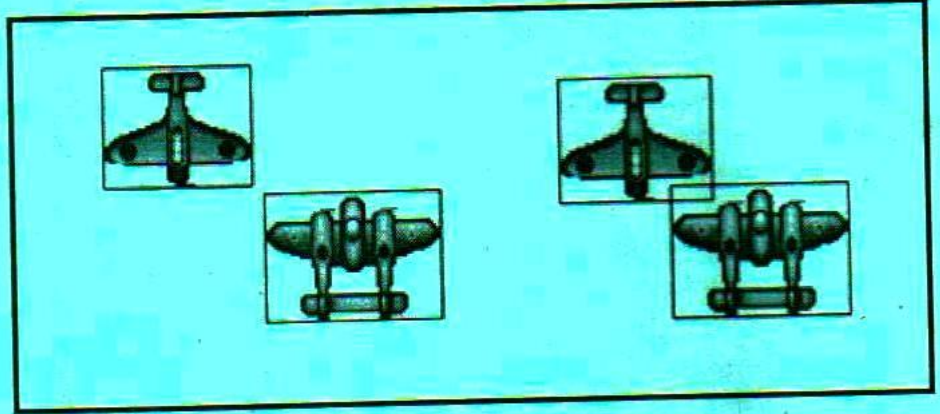
El método **selFrame()** fija el frame actual del sprite, mientras que el método **frame()** nos devolverá el número de frames del sprite.

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. La detección de colisiones entre sprites puede enfocarse desde varios puntos de vista. 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 de una colisión entre dos sprites se simplifica bastante. Sólo hemos de detectar el caso en el que dos cuadrados se solapen.

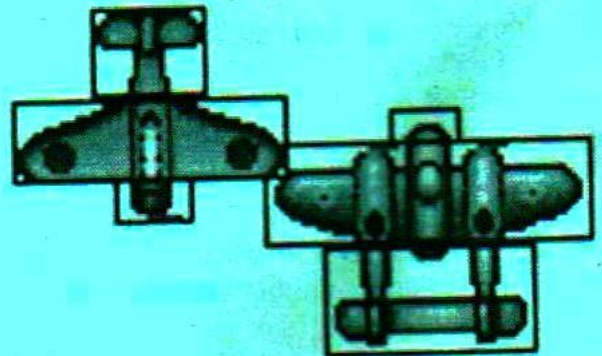


En la primera figura no existe colisión, ya que no se solapan las superficies (las superficies están representadas por el cuadrado que rodea al gráfico). La segunda figura muestra el principal problema de este método, ya que nuestra librería considerará que ha habido colisión cuando realmente no ha sido 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 otros métodos más precisos que nos permiten detectar colisiones. Consiste en dividir el sprite en pequeñas superficies rectangulares tal y como muestra la próxima figura.

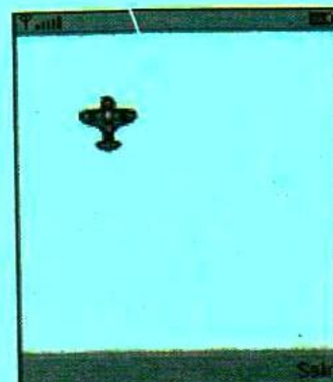


Se puede observar la mayor precisión de este método. El proceso de detección consiste en comprobar si hay colisión de alguno de los cuadros del primer sprite con alguno de los cuadrados del segundo utilizando la misma comprobación que hemos utilizado en el primer método para detectar si se solapan dos rectangulos. Se deja como ejercicio al lector la implementación de este método de detección de colisiones.

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)>y1)) {
return true;
} else {
return false;
}
}
// Dibujamos el Sprite
public void draw(Graphics g) {
g.drawImage(sprites[frame],posx,posy,Graphics.H
CENTER|Graphics.VCENTER);
}
```

Veamos un ejemplo práctico de uso de nuestra librería. Crea un nuevo proyecto en KToolBar, y añade el programa siguiente en el directorio 'src', junto con la librería Sprite.java. Por supuesto necesitarás incluir el gráfico hero.png en el directorio 'res´.



Vamos a basarnos en esta librería para el control de los Sprites del juego que vamos a crear.

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

public class SpriteTest extends MIDlet

implements CommandListener {

private Command exitCommand, playCommand,
endCommand;
private Display display;
private SSCanvas screen;

public SpriteTest() {
display=Display.getDisplay(this);
exitCommand = new
Command(«Salir»,Command.SCREEN,2);
screen=new SSCanvas();
screen.addCommand(exitCommand);
screen.setCommandListener(this);
}
public void startApp() throws
MIDletStateChangeException {
display.setCurrent(screen);
}
public void pauseApp() {}
public void destroyApp(boolean unconditional)
{}
public void commandAction(Command c,
Displayable s) {
if (c == exitCommand) {
destroyApp(false);
notifyDestroyed();
}
}
}

class SSCanvas extends Canvas {
private Sprite miSprite=new Sprite(1);
public SSCanvas() {
// Cargamos los sprites
miSprite.addFrame(1,»/hero.png»);
// Iniciamos los Sprites
miSprite.on();
}

public void paint(Graphics g) {
// Borrar pantalla
g.setColor(255,255,255);
g.fillRect(0,0,getWidth(),getHeight());
// situar y dibujar sprite
miSprite.setX(50);
miSprite.setY(50);
miSprite.draw(g);
}

}
```

ANIMANDO NUESTRO AVIÓN

▶ LECTURA DEL TECLADO

Toda aplicación interactiva necesita un medio para comunicarse con el usuario. Vamos a utilizar para ello tres métodos que nos ofrece la clase Canvas. Los métodos **keyPressed()**, **keyReleased()** y **keyRepeated()**. Estos métodos son llamados cuando se produce un evento relacionado con la pulsación de una tecla. **keyPressed()** es llamado cuando se produce la pulsación de una tecla, y cuando soltamos la tecla es invocado el método **keyReleased()**. El método **keyRepeated()** es invocado de forma repetitiva cuando dejamos una tecla pulsada.

Los tres métodos recogen como parámetro un número entero, que es el código unicode de la tecla pulsada. La clase **Canvas** también nos ofrece el método **getGameAction()**, que convertirá el código a una constante independiente del fabricante del dispositivo. La siguiente tabla, muestra una lista de constantes de códigos estándar.

Constantes	Teclas
KEY_NUM0, KEY_NUM1, KEY_NUM2, KEY_NUM3, KEY_NUM4, KEY_NUM5, KEY_NUM6, KEY_NUM7, KEY_NUM8, KEY_NUM9	Teclas numéricas
KEY_POUND	Tecla 'almohadilla'
KEY_STAR	Tecla asterisco
GAME_A, GAME_B, GAME_C, GAME_D	Teclas especiales de juego
UP	Arriba
DOWN	Abajo
LEFT	Izquierda
RIGHT	Derecha
FIRE	Disparo

Los fabricantes de dispositivos suelen reservar unas teclas con funciones más o menos precisas de forma que todos los juegos se controlen de forma similar.

Otros, como el caso del Nokia 7650 ofrecen un mini-joystick. Usando las constantes de la tabla anterior, podemos abstraernos de las peculiaridades de cada fabricante. Por ejemplo, en el Nokia 7650, cuando movamos el joystick hacia arriba se generara el código UP.

Vemos un ejemplo de uso:

```
public void keyPressed(int keyCode) {
int action=getGameAction(keyCode);
switch (action) {
    case FIRE:
        // Disparar
        break;
    case LEFT:
        // Mover a la izquierda
        break;
    case RIGHT:
        // Mover a la derecha
        break;
    case UP:
    // Mover hacia arriba
        break;
    case DOWN:
    // Mover hacia abajo
    break;
    }
}
```

Puede parecer lógico utilizar **keyRepeated()** para controlar un sprite en la pantalla, ya que nos interesa que mientras pulsemos una tecla, este se mantenga en movimiento. En principio esta sería la manera correcta de hacerlo, pero en la practica, no todos los dispositivos soportan la autorepetición de teclas (incluido el emulador de Sun). Vamos a solucionarlo con el uso de los otros dos métodos. Lo que queremos conseguir es que en el intervalo de tiempo que el jugador está pulsando una tecla, se mantenga la animación. Este intervalo de tiempo es precisamente el transcurrido entre que se produce la llamada al método **keyPressed()** y la llamada a **keyReleased().** Un poco más abajo veremos como se implementa esta técnica.

Threads

En Java, un thread puede estar en cuatro estados posibles.

- **Ejecutándose:** Está ejecutándose.
- **Preparado:** Está preparado para pasar al estado de ejecución.
- **Suspendido:** En espera de algún evento.
- **Terminado:** Se ha finalizado la ejecución.

La clase que da soporte para los threads en Java es java.lang.Thread. En todo momento podremos tener acceso al thread que está en ejecución usando el método Thread.currentThread(). Para que una clase pueda ser ejecutada como un thread ha de implementar la interfaz java.lang.Runnable, en concreto, el método run(). Éste es el método que se ejecutará cuando lancemos el thread:

```
public class Hilo implements Runnable {
   public void run(){
      // código del thread
   }
}
```

Para arrancar un thread usamos su método **start().**

```
// Creamos el objeto (que implementa Runable)
Hilo miHilo = new Hilo();

// Creamos un objeto de la clase Thread
// Al que pasamos como parámetro al objeto
miHilo
Thread miThread = new Thread( miHilo );

// Arrancamos el thread
miThread.start();
```

Si sólo vamos a utilizar el thread una vez y no lo vamos a reutilizar, siempre podemos simplificarlo.

```
Hilo miHilo = new Hilo();
new Thread(miHilo).start();
```

La clase Thread nos ofrece algunos métodos más, pero los más interesantes son stop(), que permite finalizar un thread, y sleep(int time), que lo detiene durante los milisegundos que le indiquemos como parámetro.

El Game Loop

El "game loop" o bucle de juego es el encargado de "dirigir" en cada momento que tarea se está realizando.

Lo primero que hacemos es leer los dispositivos de entrada para ver si el jugador ha realizado alguna acción. Si hubo alguna acción por parte del jugador, el siguiente paso es procesarla, esto es, actualizar su posición, disparar, etc..., dependiendo de qué acción sea.

En el siguiente paso realizamos la lógica de juego, es decir, todo aquello que forma parte de la acción y que no queda bajo control del jugador, por ejemplo, el movimiento de los enemigos, cálculo de trayectoria de sus disparos, comprobación de colisiones entre la nave enemiga y la del jugador, etc

Normalmente, el game loop tendrá un aspecto similar a lo siguiente:

```
int done = 0;
while (!done) {
   // Leer entrada
   // Procesar entrada
   // Lógica de juego
   // Otras tareas
   // Mostrar frame
}
```

Antes de que entremos en el game loop, tendremos que realizar múltiples tareas, como inicializar todas las estructuras de datos, etc...

El siguiente ejemplo es mucho más realista. Está implementado en un thread.

```
public void run() {
   iniciar();
   while (true) {

      // Actualizar fondo de pantalla
      doScroll();

      // Actualizar posición del jugador
      computePlayer();

      // Actualizar pantalla
      repaint();
      serviceRepaints();

      try {
         Thread.sleep(sleepTime);
      } catch (InterruptedException e) {
         System.out.println(e.toString());
      }
   }
}
```

Lo primero que hacemos es inicializar el estado del juego. Seguidamente entramos en el bucle principal del juego o game loop propiamente dicho. En este caso, es un bucle infinito, pero en un juego real, tendríamos que poder salir usando una variable booleana que se activara al producirse la destrucción de nuestro avión o cualquier otro evento que suponga la salida del juego.

Ya dentro del bucle, lo que hacemos es actualizar el fondo de pantalla -en la siguiente sección entraremos en los detalles de este proceso-, a continuación, calculamos la posición de nuestro avión para posteriormente forzar un repintado de la pantalla con una llamada a repaint() y serviceRepaints(). Por último, utilizamos el método sleep() perteneciente a la clase Thread para introducir un pequeño retardo.

Este retardo habrá de ajustarse a la velocidad del dispositivo en que ejecutemos el juego.

Movimiento del avión

Para mover nuestro avión utilizaremos, como comentamos en la sección dedicada a la lectura del teclado, los métodos keyPressed() y keyReleased(). Concretamente, lo que vamos a hacer es utilizar dos variables para almacenar el factor de incremento a aplicar en el movimiento de nuestro avión en cada vuelta del bucle del juego. Estas variables son deltaX y deltaY para el movimiento horizontal y vertical, respectivamente.

```
public void keyReleased(int keyCode) {
   int action=getGameAction(keyCode);

   switch (action) {

      case LEFT:
         deltaX=0;
         break;
      case RIGHT:
         deltaX=0;
         break;
      case UP:
         deltaY=0;
         break;
      case DOWN:
         deltaY=0;
         break;
   }
}

public void keyPressed(int keyCode) {
   int action=getGameAction(keyCode);

   switch (action) {

      case LEFT:
         deltaX=-5;
         break;
      case RIGHT:
         deltaX=5;
         break;
      case UP:
         deltaY=-5;
         break;
      case DOWN:
         deltaY=5;
         break;
   }
}
```

Cuando pulsamos una tecla, asignamos el valor correspondiente al desplazamiento deseado, por ejemplo, si queremos mover el avión a la derecha, el valor asignado a deltaX será 5. Esto significa que en cada vuelta del game loop, sumaremos 5 a la posición de avión, es decir, se desplazará 5 píxeles a la derecha. Cuando se suelta la tecla, inicializamos a 0 la variable, es decir, detenemos el movimiento.

La función encargada de calcular la nueva posición del avión es, pues, bastante sencilla.

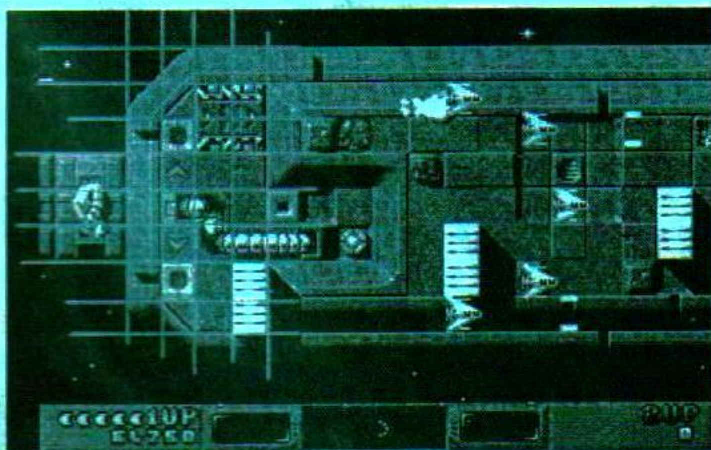
```
void computePlayer() {
  // actualizar posición del avión
          if   (hero.getX()+deltaX>0    &&
hero.getX()+deltaX<getWidth() &&
                hero.getY()+deltaY>0    &&
hero.getY()+deltaY<getHeight()) {

      hero.setX(hero.getX()+deltaX);
      hero.setY(hero.getY()+deltaY);
  }
}
```

Simplemente sumamos deltaX a la posición X del avión y deltaY a la posición Y. Antes comprobamos que el avión no sale de los límites de la pantalla.

Construyendo el mapa del juego

En nuestro juego vamos a utilizar una técnica basada en tiles para construir el mapa. La traducción de la palabra tile es baldosa o azulejo. Esto nos da una idea de en qué consiste la técnica: construir la imagen a mostrar en la pantalla mediante tiles de forma cuadrada, como si enlosáramos una pared. Mediante tiles distintos podemos formar cualquier imagen.

La siguiente figura pertenece al juego Uridium, un juego de naves o shooter de los años 80 parecido al que vamos a desarrollar para ordenadores de 8 bits.



Las líneas rojas dividen los tiles empleados para construir la imagen.

En nuestro juego manejaremos mapas sencillos que van a estar compuestos por mosaicos de tiles simples. Algunos juegos tienen varios niveles de tiles (llamados capas). Por ahora, vamos a almacenar la información sobre nuestro mapa en un array de enteros tal como éste:

```
int map[]= {1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,
        1,2,1,1,1,1,1,
        1,1,1,4,1,1,1,
        1,1,1,1,1,1,1,
        1,1,3,1,2,1,1,
        1,4,1,1,1,1,1};
```

Este array representa un mapa de 7x7 tiles. Vamos a utilizar los siguientes tiles, cada uno con un tamaño de 32x32 píxeles.

Para cargar y manejar los tiles nos apoyamos en la librería de manejo de sprites que desarrollamos en el capítulo anterior.

```
private Sprite[] tile=new Sprite[5];

// Inicializamos los tiles
for (i=1 ; i<=4 ; i++) {
   tile[i]=new Sprite(1);
   tile[i].on();
}

tile[1].addFrame(1,»/tile1.png»);
tile[2].addFrame(1,»/tile2.png»);
tile[3].addFrame(1,»/tile3.png»);
tile[4].addFrame(1,»/tile4.png»);
```

Hemos creado un array de sprites, uno por cada tile que vamos a cargar.

El proceso de representación del escenario consiste en ir leyendo el mapa y dibujar el sprite leído en la posición correspondiente. El siguiente código realiza este proceso:

```
// Dibujar fondo
for (i=0 ; i<7 ; i++) {
   for (j=0 ; j<7 ; j++) {
      t=map[i*xTiles+j];
      // calculo de la posición del tile
      x=j*32;
      y=(i-1)*32;

      // dibujamos el tile
```

```
        tile[t].setX(x);
        tile[t].setY(y);
        tile[t].draw(g);
    }
}
```

El mapa es de 7x7, así que los dos primero bucles se encargan de recorrer los tiles. La variable t, almacena el valor del tile en cada momento. El cálculo de la coordenada de la pantalla en la que debemos dibujar el tile tampoco es complicada. Al tener cada tile 32x32 píxeles, sólo hay que multiplicar por 32 el valor de los contadores i o j, correspondiente a los bucles, para obtener la coordenada de pantalla.

SCROLLING

Con lo que hemos visto hasta ahora, somos capaces de controlar nuestro avión por la pantalla, y mostrar el fondo del juego mediante la técnica de los tiles que acabamos de ver. Pero un fondo estático no es demasiado atractivo, además, un mapa de 7X7 no da demasiado juego, necesitamos un mapa más grande. Para el caso de nuestro juego será de 7X20.

```
// Mapa del juego
int map[]= {1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,
        1,2,1,1,1,1,1,
        1,1,1,4,1,1,1,
        1,1,1,1,1,1,1,
        1,1,3,1,2,1,1,
        1,1,1,1,1,1,1,
        1,4,1,1,1,1,1,
        1,1,1,1,3,1,1,
        1,1,1,1,1,1,1,
        1,4,1,1,1,1,1,
        1,1,1,3,1,1,1,
        1,1,1,1,1,1,1,
        1,1,1,1,1,1,1,
        1,2,1,1,1,1,1,
        1,1,1,4,1,1,1,
        1,1,1,1,1,1,1,
        1,1,3,1,2,1,1,
        1,1,1,1,1,1,1,
        1,4,1,1,1,1,1};
```

Se hace evidente que un mapa de 7x20 tiles no cabe en la pantalla. Lo lógico es que según se mueva nuestro avión por el escenario, el mapa avance en la misma dirección. Este desplazamiento del escenario se llama scrolling. Si nuestro avión avanza un tile en la pantalla hemos de dibujar el escenario pero desplazado (offset) un tile. Desafortunamente, haciendo esto exclusivamente veríamos como el escenario va dando saltitos, y lo que buscamos es un scroll suave. Necesitamos dos variables que nos indiquen a partir de que tile debemos dibujar el mapa en pantalla (la llamaremos indice) y otra variable que nos indique, dentro del tile actual, cuánto se ha desplazado (la llamaremos indice_in). Las variables xTile y yTile contendrán el número de tiles horizontales y verticales respectivamente que caben en pantalla. El siguiente fragmento de código cumple este cometido:

```
void doScroll() {

    // movimiento del escenario (scroll)
    indice_in+=2;
    if (indice_in>=32) {
        indice_in=0;
        indice-=xTiles;
    }

    if (indice <= 0) {
        // si llegamos al final, empezamos de nuevo.
        indice=map.length-(xTiles*yTiles);
        indice_in=0;
    }
}
```

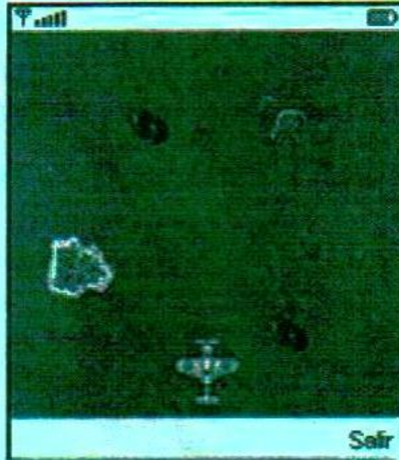
El código para dibujar el fondo quedaría de la siguiente manera:

```
// Dibujar fondo
for (i=0 ; i<yTiles ; i++) {
    for (j=0 ; j<xTiles ; j++) {
        t=map[indice+(i*xTiles+j)];
        // calculo de la posición del tile
        x=j*32;
        y=(i-1)*32+indice_in;
        // dibujamos el tile
        tile[t].setX(x);
        tile[t].setY(y);
        tile[t].draw(g);
    }
}
```

Como diferencia encontramos dos nuevas variables. La variable indice contiene el desplazamiento (en bytes) a partir del cual se comienza a dibujar el mapa. La variable indice_in, es la encargada de realizar el scroll fino. Al dibujar el tile, se le suma a la coordenada Y el valor de la variable indice_in, que va aumentando en cada iteración (2 píxeles).

Cuando esta variable alcanza el valor 32, es decir, la altura del tile, ponemos la variable a 0 y restamos el número de tiles horizonlates del mapa a la variable indice, o lo que es lo mismo. el offset a partir del que dibujamos el mapa. Se realiza una resta porque la lectura del mapa la hacemos de abajo a arriba (del último byte al primero). Recuerda que el mapa tiene 7 tiles de anchura, es por eso que restamos xTiles (que ha de valer 7) a la variable indice. Una vez que llegamos al principio del mapa, comenzamos de nuevo por el final, de forma que se va repitiendo el mismo mapa de forma indefinida.

Vamos a suponer que el dispositivo es capaz de mostrar 8 líneas de tiles verticalmente (yTiles vale 8). Si puede mostrar menos, no hay problema alguno. El problema será que la pantalla pueda mostrar más, es decir, sea mayor de lo que hemos supuesto. En ese caso aumentaremos la variable yTiles. Un valor de 8 es lo suficientemente grande para la mayoría de los dispositivos. Ten cuenta que las primeras 8 filas del mapa tienen que ser iguales que las 8 últimas si no quieres notar un molesto salto cuando recomienza el recorrido del mapa.




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

public class Scrolling extends MIDlet implements
CommandListener {

   private Command exitCommand;
   private Display display;
   private SSCanvas screen;

   public Scrolling() {
      display=Display.getDisplay(this);
      exitCommand = new
Command(«Salir»,Command.SCREEN,2);

      screen=new SSCanvas();

      screen.addCommand(exitCommand);
      screen.setCommandListener(this);
      new Thread(screen).start();
   }

   public void startApp() throws
MIDletStateChangeException {
      display.setCurrent(screen);
   }

   public void pauseApp() {}

   public void destroyApp(boolean unconditional) {}

   public void commandAction(Command c, Displayable
s) {

      if (c == exitCommand) {
         destroyApp(false);
         notifyDestroyed();
      }
   }
}
class SSCanvas extends Canvas implements Runnable {

   private int indice_in, indice, xTiles, yTiles, sleepTime;
   private int deltaX,deltaY;
   private Sprite hero=new Sprite(1);
   private Sprite[] tile=new Sprite[5];

   // Mapa del juego
   int map[] ={ 1,1,1,1,1,1,1,
               1,1,1,1,1,1,1,
               1,2,1,1,1,1,1,
               1,1,1,4,1,1,1,
               1,1,1,1,1,1,1,
               1,1,3,1,2,1,1,
               1,1,1,1,1,1,1,
               1,4,1,1,1,1,1,
               1,1,1,1,3,1,1,
               1,1,1,1,1,1,1,
               1,4,1,1,1,1,1,
               1,1,1,3,1,1,1,
               1,1,1,1,1,1,1,
               1,1,1,1,1,1,1,
               1,2,1,1,1,1,1,
               1,1,1,4,1,1,1,
               1,1,1,1,1,1,1,
               1,1,3,1,2,1,1,
               1,1,1,1,1,1,1,
               1,4,1,1,1,1,1};

   public SSCanvas() {
      // Cargamos los sprites
      hero.addFrame(1,»/hero.png»);

      // Iniciamos los Sprites
      hero.on();
   }

   void iniciar() {

      int i;
      sleepTime = 50;
      hero.setX(getWidth()/2);
      hero.setY(getHeight()-20);
      deltaX=0;
      deltaY=0;
      xTiles=7;
      yTiles=8;
      indice=map.length-(xTiles*yTiles);
      indice_in=0;

      // Inicializamos los tiles
      for (i=1 ; i<=4 ; i++) {
         tile[i]=new Sprite(1);
         tile[i].on();
      }

      tile[1].addFrame(1,»/tile1.png»);
```

```
    tile[2].addFrame(1,»/tile2.png»);
    tile[3].addFrame(1,»/tile3.png»);
    tile[4].addFrame(1,»/tile4.png»);
  }

  void doScroll() {

     // movimiento del scenario (scroll)
     indice_in+=2;
     if (indice_in>=32) {
        indice_in=0;
        indice-=xTiles;
     }

     if (indice <= 0) {
        // si llegamos al final, empezamos de nuevo.
        indice=map.length-(xTiles*yTiles);
        indice_in=0;
     }
  }
  void computePlayer() {
     // actualizar posición del avión
     if (hero.getX()+deltaX>0 &&
hero.getX()+deltaX<getWidth() &&
        hero.getY()+deltaY>0 &&
hero.getY()+deltaY<getHeight()) {
          hero.setX(hero.getX()+deltaX);
           hero.setY(hero.getY()+deltaY);
     }
  }

  // thread que contiene el game loop
  public void run() {
     iniciar();
     while (true) {

        // Actualizar fondo de pantalla
        doScroll();

        // Actualizar posición del jugador
        computePlayer();

        // Actualizar pantalla
        repaint();
        serviceRepaints();

         try {
           Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
           System.out.println(e.toString());
        }
     }
  }
  public void keyReleased(int keyCode) {
     int action=getGameAction(keyCode);
      switch (action) {
        case LEFT:
           deltaX=0;
           break;
        case RIGHT:
           deltaX=0;
           break;
        case UP:
           deltaY=0;
           break;
        case DOWN:
           deltaY=0;
           break;
     }
  }

  public void keyPressed(int keyCode) {

     int action=getGameAction(keyCode);

     switch (action) {

        case LEFT:
           deltaX=-5;
           break;
        case RIGHT:
           deltaX=5;
           break;
        case UP:
           deltaY=-5;
           break;
        case DOWN:
           deltaY=5;
           break;
     }
  }

  public void paint(Graphics g) {

     int x=0,y=0,t=0;
     int i,j;

     g.setColor(255,255,255);
     g.fillRect(0,0,getWidth(),getHeight());
     g.setColor(200,200,0);

     // Dibujar fondo
     for (i=0 ; i<yTiles ; i++) {
        for (j=0 ; j<xTiles ; j++) {
           t=map[indice+(i*xTiles+j)];
           // calculo de la posición del tile
           x=j*32;
           y=(i-1)*32+indice_in;

           // dibujamos el tile
           tile[t].setX(x);
           tile[t].setY(y);
           tile[t].draw(g);
        }
     }
     // Dibujar el jugador
     hero.setX(hero.getX());
     hero.setY(hero.getY());
     hero.draw(g);
  }
}
```