PROGRAMACIÓN DE JUEGOS PARA MÓVILES CON J2ME

Programación de juegos para móviles con J2ME

Email: j2me @ agserrano .com
ISBN: 84-609-1927-7

Índice.

Introducción

Los teléfonos móviles son ya parte esencial en nuestra forma de vida. Cada día son más los usuarios de estos terminales, y cada vez más pequeños. Hasta ahora, nos han acompañado a todas partes y nos han permitido comunicarnos con cualquier otro terminal, ya sea 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 nuestros pequeños teléfonos.

El primero, y quizás más lucrativo hasta la fecha, fue el uso de la mensajería SMS (Short Message Service). Nuestros pequeños terminales nos permiten enviar mensajes cortos de texto (hasta un tamaño de 160 caracteres) que son enviados desde el terminal al centro servidor de mensajes cortos o SMSC (Short Message Service Centre), que a su vez se encarga de hacer llegar el mensaje al móvil destinatario.

Más tarde, 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 paliar 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, tanto es así, que ahora se puede enviar grandes paquetes de información, como fotografías, música, e incluso video. A partir de aquí, 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). Gracias a MMS, además de texto, podemos enviar fotografías, sonidos, gráficos, etc. Pero aún estaba por llegar la verdadera revolución.

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. Una de las principales capacidades que añade esta tecnología a nuestros terminales es la posibilidad de descargar y ejecutar juegos con una calidad razonable. Hoy, nuestros teléfonos móviles 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.

Este libro 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. De cualquier forma, siempre que usemos las características nuevas de la segunda versión, se indicará convenientemente. No quisiera terminar sin agradecer a Ari Feldman (http://www.arifeldman.com/) sus gráficos con licencia GPL, que se han utilizado para

realizar el juego M1945. Espero que disfrutes con este libro. En cualquier caso puede contactar con el autor en la dirección *j2me @ agserrano .com*.

CAPÍTULO 1

Breve introducción al lenguaje Java.

En este capítulo quiero presentarte, de forma general, 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 adentrarte sin problemas en la programación con J2ME.

J2ME

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). J2SE es el heredero directo del Java inicial (antes de Java 2). J2EE (Java 2 Enterprise Edition) está basado en J2SE, pero añade una serie de características necesarias en entornos empresariales, relativos a redes, acceso a datos y entrada/salida que requieren mayor capacidad de proceso, almacenamiento y memoria. La decisión de separarlos es debida a que no todas estas características son necesarias para el desarrollo de aplicaciones standard.

Al igual que J2EE cubre unas necesidades más amplias que J2SE, se hace patente la necesidad de un subconjunto de J2SE para entornos más limitados. La respuesta de Sun es J2ME (Java 2 Micro Edition).

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). Concretamente CLDC define:

- Cuáles son las características del lenguaje Java incluidas.
- Qué funcionalidad será incluida en la máquina virtual Java.
- Las APIs necesarias para el desarrollo de aplicaciones en móviles.
- Los requerimientos Hardware de los dispositivos.

Debido a las limitaciones del hardware en el que correrá la máquina virtual, algunas de las características del lenguaje Java han sido recortadas. En concreto, se ha omitido el soporte de operaciones matemáticas en punto flotante, y por lo tanto, los tipos de datos que manejan esta de información. La otra gran diferencia es que la máquina virtual tampoco dará soporte al método `finalize()` encargado de eliminar los objetos de la memoria. También se limita el número de excepciones disponibles para el control de errores.

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. Las características concretas de la versión 1.0 y 2.0 de MIDP pueden ser consultadas en la página web de Sun (*http://java.sun.com/j2me/*).

El lenguaje Java

El lenguaje Java es un lenguaje completamente orientado a objetos. Todo en Java es un objeto. Durante el resto del capítulo vamos a ver las características generales del lenguaje Java (sólo las necesarias en J2ME), lo que nos permitirá entrar en el siguiente capítulo con la base necesaria para empezar a programar MIDlets. Un MIDlet es un programa capaz de correr en un dispositivo móvil. El nombre guarda cierta similitud (no casual) con los programas capaces de correr en un navegador (applets). Si quieres ampliar conocimientos sobre el lenguaje Java, puedes consultar la bibliografía en los apéndices de este libro.

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.

El nombre de una variable puede ser cualquiera, aunque conviene utilizar nombres claros y relacionados con el cometido 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 *numeroDeVidas*.
- 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.
- **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:

`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 siguientes operadores que vamos a ver son los operadores aritméticos. Hay dos tipos, los operadores 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 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. 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--;
```

Aquí, 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.

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).

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*.

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.

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.

operadores sufijo	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creación o tipo	new (type)expr
multiplicadores	* / %
suma/resta	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=

```
bitwise AND           &
bitwise exclusive OR  ^
bitwise inclusive OR  |
AND lógico            &&
OR lógico             ||
condicional           ? :
asignación            = += -= *= /= %= ^= &= |= <<= >>= >>>=
```

Clases y objetos

¿Qué es un objeto? Trataré de explicarlo de la forma más intuitiva posible sin entrar en demasiados formalismos. Si te pido que pienses en un objeto, seguramente pensarás en un lápiz, una mesa, unas gafas de sol, un coche o cualquier otra cosa que caiga dentro de tu radio de visión. Ésta es la idea intuitiva de objeto: algo físico y material. En POO, el concepto de objeto no es muy diferente. Una de las diferencias básicas evidentes es que un objeto en Java puede hacer referencia a algo abstracto.

Como en el ejemplo del coche, un objeto puede estar compuesto por otra *clase* de objeto, como rueda, carrocería, etc... Este concepto de *clase* de objeto es importante. 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 forman una *instancia* de la clase *rueda*. Tenemos, pues, instancias de la clase *rueda* que son ruedas de camión, ruedas de coches o ruedas de motocicleta.

Volvamos al ejemplo del coche. Vamos a definir otra clase de objeto, la clase *coche*. Esta clase define a "algo" que está compuesto por objetos (instancias) de la clase *rueda*, la clase *carrocería*, la clase *volante*, etc... Ahora vamos a crear un objeto de la clase *coche*, al que llamaremos coche_rojo. En este caso hemos *instanciado* un objeto de la clase *coche* y hemos definido uno de sus *atributos*, el color, al que hemos dado el valor de rojo. Vemos pues, que un objeto puede poseer atributos. Sobre el objeto coche podemos definir también acciones u operaciones posibles. Por ejemplo, el objeto coche, entre otras cosas, 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 hacer ya 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, ni más ni menos, que la capacidad que tiene un objeto de contener datos (atributos) y código (métodos).

Clases y objetos en Java

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();
```

En primer lugar, hemos creado el objeto *miCoche* que pertenece a la clase *Coche* mediante el operador *new*. Después, podemos acceder tanto a los métodos como a las variables miembro usando su nombre precedido de un punto y el nombre del objeto.

También podríamos haber accedido a la variable miembro: `miCoche.velocidad = 100;` Ésta no es una práctica aconsejable. Lo óptimo es que la clase ofrezca los métodos necesarios para acceder a las variables miembro para tener así control sobre el acceso a los atributos. No queremos que nadie haga algo como `miCoche.velocidad = 1200;` sin que podamos controlarlo.

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 tipo especial de método que se llama constructor. Un constructor es un método que 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.

Hay cuatro tipos de modificadores que permiten especificar qué tipo de clase estamos declarando. Los tipos de modificadores son los siguientes.

- **abstract**. Una clase abstract tiene al menos un método abstracto. Una clase abstracta sólo puede ser heredada para implementar los métodos abstractos que contiene. En ningún caso podemos instanciar un objeto de este tipo.
- **final**. Una clase final no puede ser heredada por ninguna otra.
- **public**. Una clase public puede ser accedida por otras clases pertenecientes al mismo paquete, o por cualquier otra siempre que sea importada o heredada.
- **synchronizable**. Significa que esta clase sólo puede ser accedida por un sólo thread a la vez. Se utiliza en aplicaciones multihebra para asegurar que no hay problemas de sincronización entre hilos.

Al igual que tenemos modificadores para las clases, también tenemos modificadores de acceso a las variables miembro y a los métodos.

- **public**. Se puede acceder desde fuera de la clase a la que pertenece.
- **protected**. Sólo las subclases pueden acceder a este miembro de la clase.
- **private**. Sólo se puede acceder a la variable o al método desde el interior de la clase.
- **friendly**. Es la opción por defecto si no se especifica nada. Permite sólo el acceso desde las clases pertenecientes al mismo paquete.

Un paquete nos permite agrupar clases bajo un nombre común, por ejemplo, si hiciéramos una librería capaz de manejar gráficos, tendríamos un montón de clases encargadas de manejar el color, píxeles, imágenes, etc... Tiene lógica agrupar todas estas clases dentro de un paquete. Cuando creamos un paquete, las clases que están incluidas se almacenan en un mismo directorio con el nombre del paquete. Indicamos que una clase pertenece a un paquete concreto mediante la palabra reservada *package* al principio del archivo fuente.

```
package nombre_paquete;
```

Si quisiéramos utilizar este paquete que acabamos de crear, hay que importarlo. Para ello utilizamos la palabra reservada *import*.

```
import nombre_paquete;
```

J2ME dispone de multitud de paquetes, por ejemplo, si queremos utilizar el interfaz de usuario propio de J2ME, debemos importar el paquete *lcdui*.

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

El punto se utiliza para indicar la jerarquía de paquetes, es decir, la jerarquía de directorios donde están almacenadas las clases. El asterisco indica que deben importarse todas las clases pertenecientes al paquete.

Herencia

No sé de color tienes los ojos, pero puedo asegurar que del mismo color que alguno de tus ascendientes. Este mecanismo biológico fue descrito por Mendel (armado con una buena dosis de paciencia y una gran cantidad de guisantes) y se llama herencia. La herencia 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 -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;
     }
}
```

Lo primero que nos llama la atención de la declaración de la clase es su 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 *CochePolicia* posee dos atributos, velocidad, que ha sido heredado y sirena, que ha sido declarada dentro de la clase *CochePolicia*. Con los métodos sucede exactamente igual. La clase hija ha heredado *acelerar()* y *frenar()*, además le hemos añadido los métodos *sirenaOn()* y *sirenaOff()*. Un objeto instancia de CochePolicia puede utilizar sin ningún problema los métodos *acelerar()* y *frenar()* tal y como hacíamos con los objetos instanciados de la clase *Coche*.

No es posible heredar de dos o más clases a la vez (al contrario que en C++). Esto se llama herencia múltiple, y suele conllevar más problemas que ventajas, así que los diseñadores de Java prefirieron no incluir esta característica.

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;

   }
}
```

Como ves 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.

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.

CAPÍTULO 2

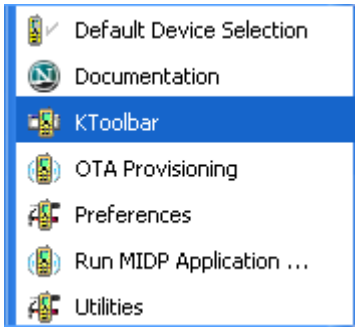
Nuestro primer MIDlet.

En este capítulo vamos a construir y ejecutar 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.

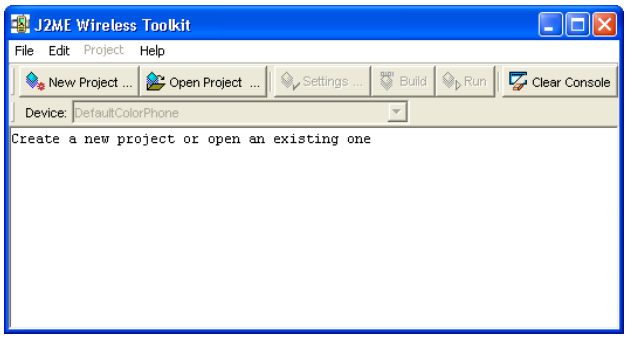
Para instalar J2ME Wireless Toolkit, primero hemos de instalar el entorno de programación de J2SE (JDK). Puedes descargar la última versión de JDK desde la URL http://java.sun.com/j2se/downloads.html. Una vez descargado e instalado, estaremos en condiciones de descargar e instalar J2ME desde la URL http://java.sun.com/j2me/download.html.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

Compilando el primer MIDlet

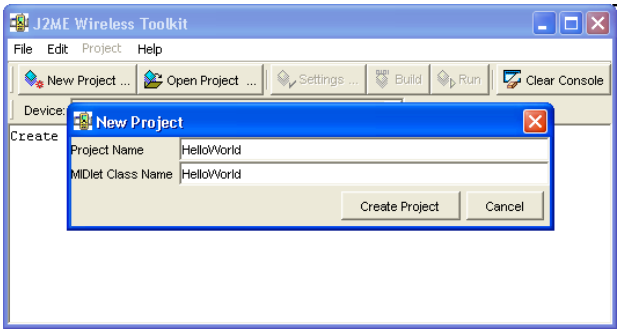
Vamos a construir paso a paso nuestro primer MIDlet usando esta herramienta. Tras la instalación del wireless toolkit, tendremos un nuevo submenú en el menú inicio con un aspecto similar a éste:



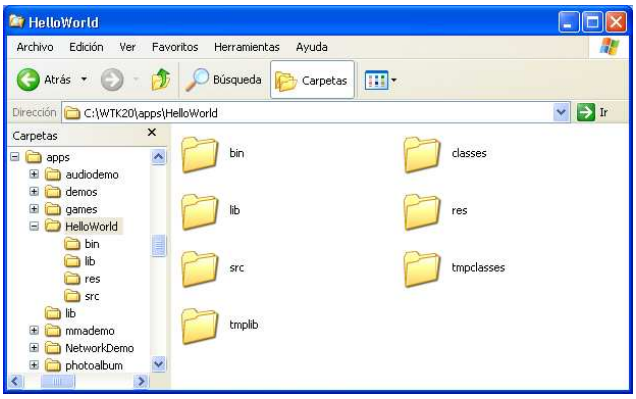
Selecciona la aplicación KToolBar e inicializa el entorno. Verás aparecer la ventana del entorno.



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 móvil. En la pantalla del móvil 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 móvil.

Desarrollo de MIDlets

Lo que acabamos de hacer es crear un MIDlet desde cero, aunque sin saber muy bien que es lo que estábamos haciendo. Vamos a profundizar en los pasos que hemos seguido hasta conseguir nuestro MIDlet. El ciclo de desarrollo de un MIDlet es el siguiente:

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

Como hemos visto en la primera parte del capítulo, utilizamos un editor de textos común para editar el programa. La compilación la llevamos a cabo con el comando *Build* de la herramienta KTooBar. Cuando compilamos un programa Java, y por lo tanto un MIDlet, se genera un archivo con la extensión .class por cada clase, que contiene el código intermedio

que es capaz de ejecutar la máquina virtual de Java. Estos archivos son almacenados en el directorio *classes*. El paso que sigue a la compilación es preverificar las clases que se han generado. Sin esta preverificación no será posible la ejecución del MIDlet. Pero, ¿para qué sirve este paso? La preverificación nos asegura que no existe ningún tipo de código malintencionado que pueda dañar o crear un comportamiento extraño en nuestro dispositivo o en la máquina virtual. Lo habitual es que la máquina Java sea lo suficientemente robusta como para que no haya ningún problema en este sentido, pero debido a que los dispositivos J2ME no tienen demasiada capacidad de proceso, es necesario que la máquina virtual sobre la que se ejecutan los MIDlets (llamada K Virtual Machine o KVM) sea lo más eficiente posible, por lo que se han eliminado muchas de las comprobaciones que realizan las máquinas virtuales habitualmente.

Ahora que hemos compilado y preverificado nuestro MIDlet, ya podemos ejecutarlo en el emulador. Esto lo hacemos con el comando *Run* de la herramienta KTooBar. Si ahora queremos ejecutarlo en el dispositivo real, hemos de realizar un paso más. Los MIDlets tienen que ser distribuidos en dos archivos especiales. Son los archivos JAR y los archivos JAD. Un archivo JAR es un archivo comprimido (en formato ZIP) que contiene las clases (.class) que ha generado la compilación de nuestro programa. Además puede contener los recursos necesarios para el MIDlet como sonidos, gráficos, etc... Finalmente, contiene un archivo con extensión .mf., es lo que se llama un archivo de manifiesto. Este archivo contiene información sobre las clases contenidas en el archivo JAR.

El segundo archivo necesario para la distribución de MIDlets son los archivos JAD. El archivo JAD contiene información necesaria para la instalación de los MIDlets contenidos en el archivo JAR. Un archivo puede contener más de un MIDlet. Cuando ocurre esto, hablamos de un *MIDlet suite*. Podemos editar los parámetros contenidos en el archivo JAD mediante el botón *Settings* de KTooBar. Aquí podemos editar información del MIDlet como el nombre, la versión o el autor del MIDlet (o de los MIDlets).

Sólo nos resta transferir los archivos JAR y JAD al dispositivo J2ME. Hay varias formas de hacerlo dependiendo de la marca y modelo del dispositivo. Si el teléfono tiene soporte de infrarrojos o bluetooth, y tu ordenador tiene puerto IrDA o bluetooth, podrás transferirlo fácilmente sin necesidad de cable alguno. Si no, tendrás que recurrir a un cable de datos (consulta el manual de tu teléfono). Otra posibilidad es poner los archivos JAR y JAD en un servidor wap o un espacio web y descargarlo desde tu móvil. Para ello es necesario que el dispositivo tenga un navegador wap o web y soporte GPRS para una descarga fiable.

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.

CAPÍTULO 3

La interfaz de usuario de alto nivel.

Como vimos en el anterior capítulo, 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:

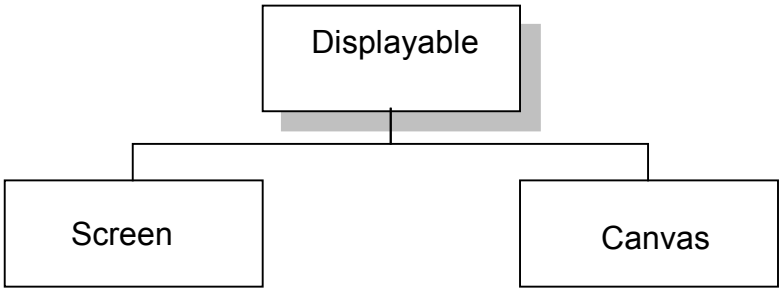
- Timer
- TimerTask

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, y es el que usaremos más intensivamente a partir del capítulo siguiente.

¿Cómo funciona un MIDlet?

Vamos a entrar directamente en materia analizando el programa de ejemplo del capítulo anterior.

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

Con estas dos líneas importamos las clase MIDlet, que es obligatoria para que nuestro programa se ejecute en el dispositivo móvil, y también importamos (con la segunda línea) los elementos que vamos a utilizar el la interfaz de usuario.

```
public class HelloWorld extends MIDlet implements CommandListener {
   private Command exitCommand;
   private Display display;
   private Form screen;
```

En la primera línea declaramos la clase principal del programa, que es pública. Hereda de la clase MIDLet e implementa la interfaz CommandListener (en concreto implementaremos el método `commandAction()`). También declaramos una serie de variables que utilizaremos a continuación.

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

Este es el constructor de la clase. Observa que tiene el mismo nombre que la clase (mayúsculas y minúsculas incluidas) y además no tiene tipo de retorno, ni siquiera void.

Al ejecutarse un MIDlet, éste crea un objeto display, que es el encargado de mostrar información en la pantalla. Para poder utilizarlo, tenemos que obtener una referencia a este objeto. Esto es lo que hace precisamente la siguiente línea mediante el método `getDisplay()` del objeto estático Display.

```
      //  Creamos el comando Salir.
      exitCommand = new Command("Salir", Command.EXIT,2);
```

Un comando es un elemento que nos permite interaccionar con el usuario y le permite introducir comandos. Para crear un comando creamos una instancia (con new) de la clase `Command()`. Como parámetro le pasamos el texto del comando, el tipo de comando y la prioridad.

Disponemos de los siguientes tipos de comandos:

OK	Confirma una selección
CANCEL	Cancela la acción actual
BACK	Traslada al usuario a la pantalla anterior
STOP	Detiene una operación
HELP	Muestra una ayuda
SCREEN	Tipo genérico para uso del programador referente a la pantalla actual
ITEM	Tipo genérico para uso del programador referente a un elemento de la pantalla actual

A veces, y dependiendo del modelo y marca del dispositivo, sólo se pueden mostrar un número limitados de comandos en la pantalla. Al resto se accederá mediante un menú.

El tercer parámetro nos permite dar más prioridad a unos comandos que a otros a la hora de mostrarlos en la pantalla.

```
    // Añadimos el comando Salir e indicamos que clase lo manejará
    screen.addCommand(exitCommand);
    screen.setCommandListener(this);
```

Nos resta añadir el comando (mediante el método `addCommand()` del objeto screen) a la lista de comandos y establecer que clase permanece a la "escucha" de esos comandos utilizando la clase `setCommandListener()`. En este caso, el método encargado de procesar los comandos están dentro de la propia clase *HelloWorld*, por lo que utilizamos el operador this. Si quisiéramos tener una clase separada encargada de procesar los comandos, la indicaríamos aquí. En concreto, el método que se encarga de procesar los comandos es `commandAction()`. Para eliminar un comando podemos utilizar `removeCommand(Command cmd)`.

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

Cuando el usuario genera un comando, se llama al método `commandAction()`. Este método recibirá dos parámetros. El comando que se generó, y un objeto de la clase Displayable, que contiene la pantalla del comando.

Cerraremos la aplicación con los métodos `destroyApp()` y `notifyDestroyed()`.

```
    // 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);
```

Dentro de la pantalla podemos situar diversos elementos gráficos. Vamos a crear un objeto de tipo Form (formuario) como elemento principal de la pantalla. Veremos dentro de este capítulo cuáles son estos elementos.

Seguidamente creamos una cadena de texto (StringItem) y la añadimos a la pantalla principal mediante el método `append()`.

```
   public void startApp() throws MIDletStateChangeException {
      // Seleccionamos la pantalla a mostrar
      display.setCurrent(screen);
   }
```

Mediante el método setCurrent() del objeto display (aquel del que obtuvimos la referencia al principio del constructor) seleccionamos la pantalla actual para ser mostrada. Lo hacemos en el método `startApp()` que es el que se ejecuta en primer lugar.

```
public void pauseApp() {}
public void destroyApp(boolean unconditional) {}
```

Estas dos líneas pueden parecer extrañas, ya que son métodos vacíos (sin código). Como ya vimos, hay que implementar todas las clases heredadas de MIDlet (pauseApp, destroyApp y startApp), por lo que, aunque no contengan código, hay que declararlas.

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.

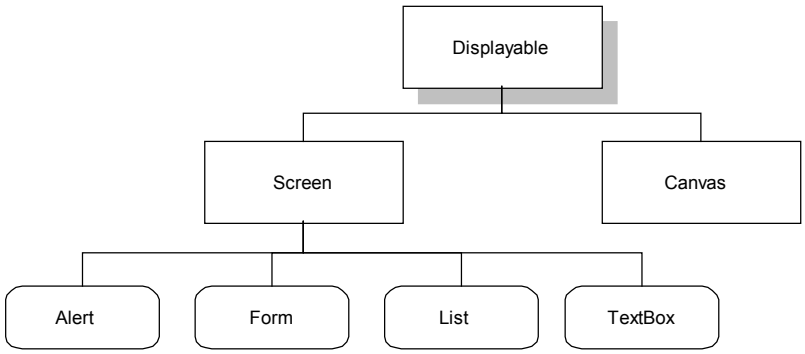


Figura 3.1.

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*.

Un MIDlet típico estará compuesto de varios de estos elementos. Por desgracia, y debido al pequeño tamaño de la pantalla, no pueden mostrarse más de un elemento a la vez, por lo que tendremos que ir mostrando el elemento que necesitemos que ocupará toda la pantalla.

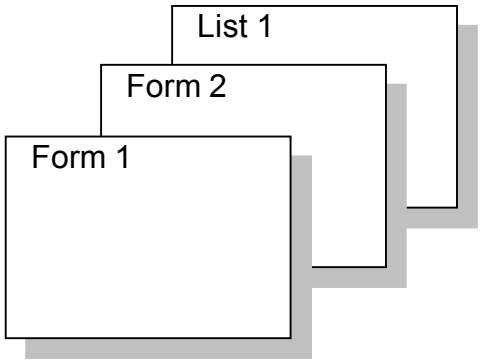


Figura 3.2.

Podemos imaginarlo como una serie de fichas de las cuales sólo podemos mostrar una cada vez.

Para cambiar de una pantalla a otra usamos el método *setCurrent* de la clase Display (como ya vimos en nuestro ejemplo):

```
display.setCurrent(list1);
```

Cada uno de las cuatro clases anteriores dispone de los métodos (realmente lo heredan de *Screen*):

```
String getTitle()              - Devuelve el título de la pantalla
void setTitle(String s)        - Establece el título de la pantalla
Ticker getTicker()             - Devuelve el ticker de la pantalla
void setTicker(Ticker ticker) - Establece el ticker de la pantalla
```

Estos métodos nos permiten establecer y recoger el Título y el ticker de la pantalla. Un ticker es una línea de texto que aparece en la parte superior de la pantalla con un scroll lateral.

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.

Figura 3.3. Una de las demos de WTK.

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.

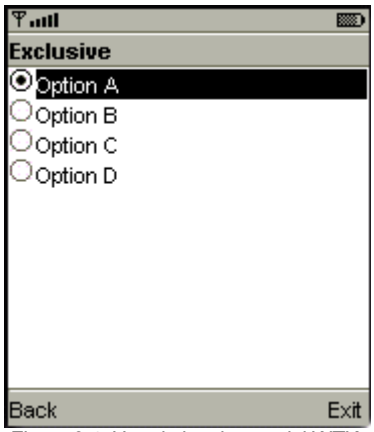


Figura 3.4. Una de las demos del WTK.

Veamos es 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

Un ejemplo real:

```
String[] ciudades = {"Málaga", "Madrid", "Melilla"};
List lista = new List ("Seleccione una ciudad", List.EXCLUSIVE, ciudades,
null);
```

En las listas de tipo EXCLUSIVE e IMPLICIT se puede utilizar el método getSelectedIndex() que retorna el índice del elemento seleccionado. Pasando como parámetro el índice al método `getString()` nos devuelve el texto del elemento seleccionado. En listas de tipo MÚLTIPLE podemos utilizar el método:

```
int getSelectedFlags(boolean[] array_seleccionados)
```

Esta función rellenará el array de tipo booleano que le pasamos como parámetro con valores *true* o *false* según el elemento correspondiente esté seleccionado. Evidentemente, el array debe tener una correspondencia uno a uno en número de elementos con la lista.

La clase TextBox

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

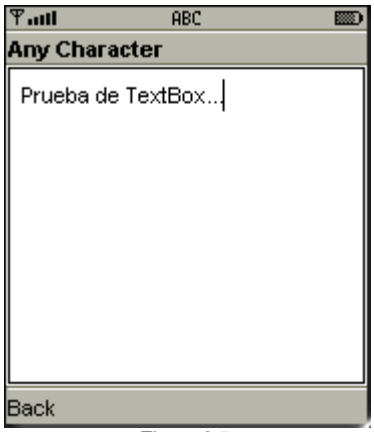


Figura 3.5.

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 número 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);
```

Para conocer el texto que contiene la caja puede usarse los métodos siguientes:

```
String getString()

int getChars (char[] texto)
```

En el caso de `getChars()`, el texto será almacenado en la variable texto en forma de array de caracteres.

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

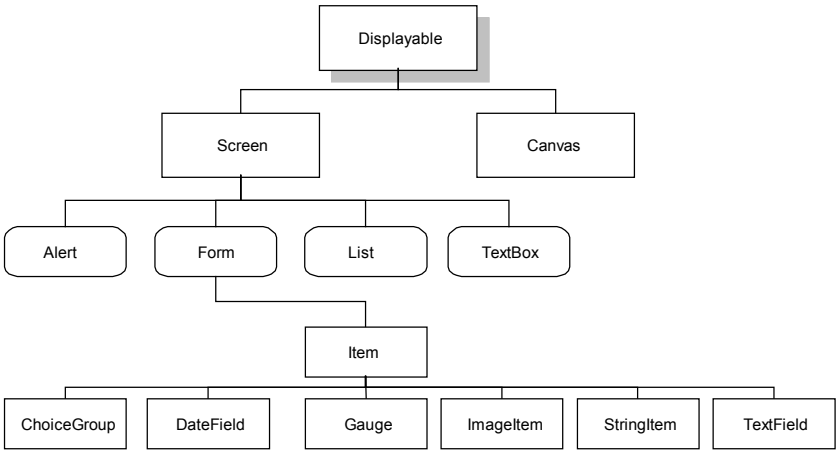


Figura 3.6.

Como vemos en el diagrama, la clase Form es capaz de manejar objetos derivados de la clase Item. La clase Item representa a un elemento visual que no ocupará toda la pantalla, sino que formará parte de la interfaz de usuario junto con otros elementos.

Ya hemos visto un ejemplo de la clase Form en el programa de ejemplo del anterior capítulo. En el ejemplo, creamos un elemento de tipo *StringItem* y lo añadimos al formulario con el método `append()`.

Los métodos de la clase Form que nos permiten añadir, eliminar y modificar elementos del formulario son las siguientes:

```
int append(Item elemento)
```

Como ya sabes, `append()` añade al formulario un elemento. Cada vez que añadimos algo al formulario, a éste se le asocia un número de índice para poder hacer referencia a él posteriormente. El primer elemento que añadamos tendrá el valor cero, y así sucesivamente. Si todo va bien, el método retornará el índice del elemento.

```
void delete(int índice)
```

El método delete() elimina un elemento del formulario.

```
void insert(int índice, Item elemento)
```

El método `insert()` inserta un elemento en la posición que indiquemos en el primer parámetro.

Si lo que queremos es sustituir un elemento por otro utilizaremos el método `set()`:

```
void set(int índice, Item elemento)
```

En algún momento es posible que necesitemos conocer el número de elementos del formulario. El método `size()` nos muestra esta información:

```
int size()
```

Por último, necesitaremos algún mecanismo que nos permitan responder a cambios en los elementos como, por ejemplo, un cambio de valor. El mecanismo es similar al de los

comandos que vimos algunas líneas atrás. Hemos de implementar la interface *ItemStateListener*. Concretamente el método siguiente:

```
void itemStateChanged(Item elemento)
```

Para indicar al formulario cuál será la clase que responderá a los eventos podemos utilizar:

```
formulario.setItemStateListener(this);
```

Si la clase que manejará los eventos es distinta a la que contiene el formulario sustituiremos el operando *this* por la clase deseada.

La clase StringItem

Esta clase ya la conocemos del ejemplo del capítulo anterior. Su función es añadir etiquetas de texto al formulario.

Figura 3.7.

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.

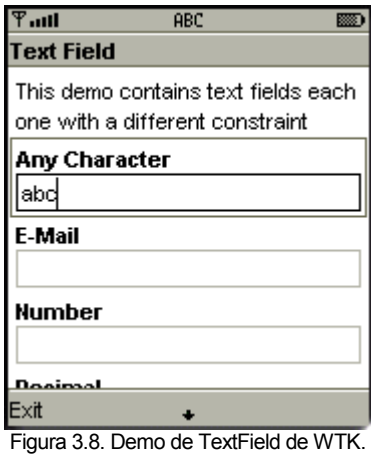


Figura 3.8. Demo de TextField de WTK.

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.

Figura 3.9. Demo de DateField.

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.

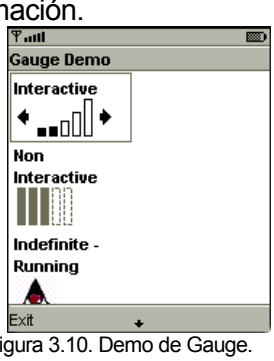


Figura 3.10. Demo de Gauge.

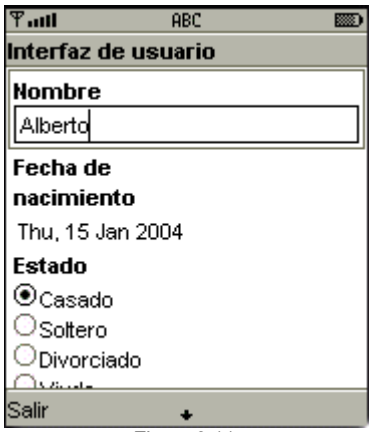
La clase *Gauge* nos ofrece cuatro métodos muy útiles:

```
int getValue()
void setValue(int valor)
int getMaxValue()
void setMaxValue(int valor)
```

Las dos primeras 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 programa muestra el uso de varios elementos en un formulario a la vez.


ABC
Interfaz de usuario
Nombre
Alberto
Fecha de
nacimiento
Thu, 15 Jan 2004
Estado
Casado
Soltero
Divorciado
Salir
Figura 3.11.

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

public class UI extends MIDlet implements CommandListener {
   private Command exitCommand;
   private Display display;
   private Form screen;

   public UI() {
      String[] estados = {"Casado","Soltero","Divorciado","Viudo"};

      // 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("Interfaz de usuario");

      // Creamos y añadimos los elemento que vamos a utilizar
      TextField nombre = new TextField("Nombre","",30,TextField.ANY);
      DateField fecha_nac = new DateField("Fecha de nacimiento", DateField.DATE);
      ChoiceGroup estado = new ChoiceGroup("Estado",List.EXCLUSIVE,estados,null);
      screen.append(nombre);
      screen.append(fecha nac);
      screen.append(estado);


      // 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();
      }
   }
}
```

CAPÍTULO 4

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.

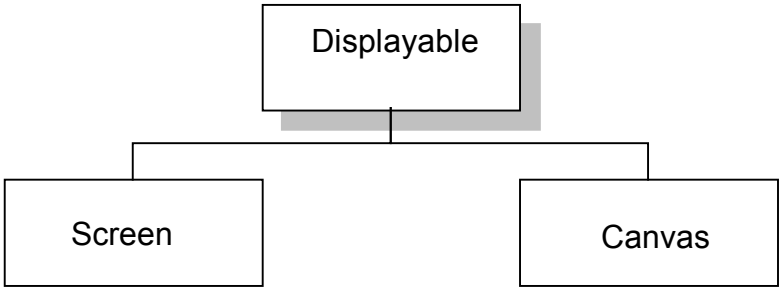


Figura 4.1.

Básicamente podemos realizar tres operaciones sobre un *Canvas*:

- Dibujar primitivas gráficas
- Escribir texto
- Dibujar imágenes

Es este capítulo vamos a cubrir estas operaciones, y así sentar las bases necesarias para abordar las materias concretas relativas a la programación de videojuegos.
Tal y como hicimos en el anterior capítulo utilizaremos un código de ejemplo para ir explicando sobre él los conceptos básicos.

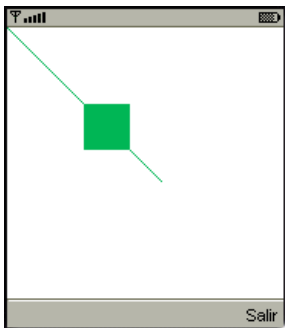


Figura 4.2. Resultado de ejecutar el código de ejemplo.

```
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);
 }
}
```

La Primera parte del código, es decir, la clase *Canvas1* debe ya sernos familiar. La única diferencia con el código del capítulo anterior, es que, en lugar de utilizar un objeto de la clase Form como pantalla principal de la aplicación, utilizamos uno derivado de la clase *SSCanvas* que implementamos justo debajo.

```
private SSCanvas screen;
```

Como puedes observar, la clase *SSCanvas* hereda de la clase *Canvas*. Es por ello que podemos utilizarla como pantalla principal (recuerda que Canvas es una clase derivada de *Displayable*).

```
class SSCanvas extends Canvas {
```

La clase *SSCanvas* implementa el método `pain()`, que hereda de la clase *Canvas*. Éste es el método que se invoca cada vez que la pantalla necesita ser redibujada. Por lo tanto, todo el código encargado de pintar en la pantalla ha de situarse aquí.

```
public void paint(Graphics g) {
```

El parámetro *g*, es el llamado contexto gráfico, que es de tipo *Graphics*. Esta clase posee los métodos necesarios para dibujar en pantalla, mostrar gráficos y mostrar textos.
Las siguientes líneas se ocupan de borrar la pantalla y dibujar una línea y un rectángulo utilizando los métodos de la clase *Graphics*.
Vamos a entrar a continuación en detalles sobre los métodos de esta clase.

Primitivas Gráficas

Colores

En la naturaleza hay millones de posibles tonalidades de color. necesitamos pues un método que nos permita expresar un color concreto de una gama muy amplia. Newton, experimentando con un prisma de vidrio, constató como la luz podía descomponerse en una serie de colores básicos. Lo cierto es que este experimento sólo trataba de reproducir lo que en la naturaleza conocemos como arco iris, que se produce por la difracción de la luz al atravesar las gotas de agua. Un color, por lo tanto, se forma a partir de las distintas aportaciones de los colores básicos. Según predomine más o menos cada color básico, el color resultante será distinto. Utilizando esta misma técnica, en informática utilizamos tres colores básicos para especificar colores complejos. Estos colores son el Rojo, el Verde y el Azul, y abreviamos con RGB (de Red, Green y Blue).

255, 0, 0	Rojo
0, 255, 0	Verde
0, 0, 255	Azul
128, 0, 0	Rojo oscuro
255, 255, 0	Amarillo
0, 0, 0	Negro
255, 255, 255	Blanco
128, 128, 128	Gris

Algunos ejemplos de colores (R,G,B)

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.

En nuestro programa de ejemplo, además de utilizarlo para dibujar un cuadrado, le hemos asignado la misión de borrar la pantalla. Esto se hace dibujando un rectángulo del tamaño de la pantalla del dispositivo y de color blanco.

```
void drawRoundRect (int x, int y, int ancho, int alto, int ancho_arco, int
alto_arco)
```

Este método es similar a `drawRect()`, excepto porque sus esquinas son redondeadas. Los otros dos parámetros son al ancho y el alto del arco de las esquinas.

```
void drawArc(int x, int y, int ancho, int alto, int ángulo_inicial, int
ángulo)
```

Con `drawArc()` seremos capaces de dibujar secciones de arco. Los cuatro primeros parámetros no deberían necesitar ya explicación. El perámetro *ángulo_inicial* indica cual será el ángulo a partir del que se comenzará a dibujar el segmento de arco. El parámetro *ángulo* indica la longitud de arco (en grados) que será dibujado.

```
void fillRect (int x, int y, int ancho, int alto)

void fillRoundRect (int x, int y, int ancho, int alto, int ancho_arco, int
alto_arco)

void fillArc(int x, int y, int ancho, int alto, int ángulo_inicial, int
ángulo)
```

Estos tres métodos son iguales que los anteriores con la diferencia de que la primitiva que dibujan estará rellena. Es decir, dibuja las primitivas como sólidas en lugar de huecas.

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 horizontal 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);
```

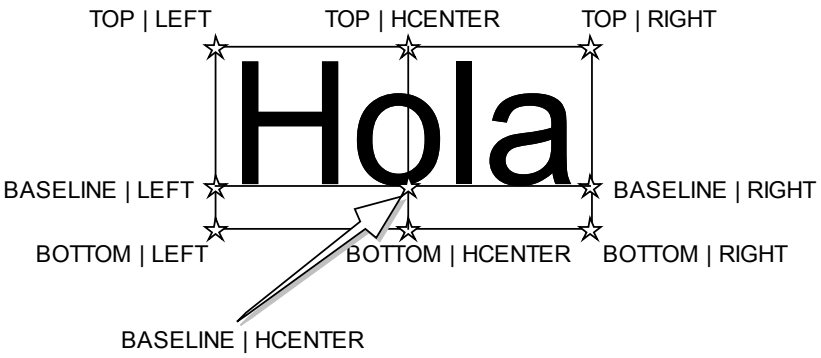


Figura 4.3.

Ahora que tenemos control sobre la posición del texto, nos falta por controlar el tipo de letra que queremos utilizar.

```
void setFont(Font fuente)
```

Esta función selecciona la fuente a utilizar. El método `getFont()` de la clase Font nos devuelve un objeto de tipo Font que podemos utilizar con `setFont()`.

```
static Font getFont (int espaciado, int estilo, int tamaño)
```

Los valores posibles para estos parámetros son:

Tamaño	`SIZE_SMALL`, `SIZE_MEDIUM`, `SIZE_LARGE`
Estilo	`STYLE_PLAIN`, `STYLE_ITALICS`, `STYLE_BOLD`, `STYLE_UNDERLINED`
Espaciado	`FACE_SYSTEM`, `FACE_MONOSPACE`, `FACE_PROPORTIONAL`

Veamos un ejemplo:

```
Font fuente = Font.getFont (Font.FACE_PROPORTIONAL, Font.STYLE_BOLD,
Font.SIZE_MEDIUM);

g.setFont(fuente);
```

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()cargar 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);
```

El siguiente código es un ejemplo real de lo que hemos visto en éste capítulo.

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

public class Canvas2 extends MIDlet implements CommandListener {

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

 public Canvas2() {
   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) {

    Image img=null;

   // Borrar la pantalla
   g.setColor(255,255,255);
   g.fillRect (0, 0, getWidth(), getHeight());

   // Dibujar línea
   g.setColor(10,200,100);
    g.drawLine (0, 80, getWidth(), 80);

    // Poner texto
   Font fuente = Font.getFont (Font.FACE_PROPORTIONAL, Font.STYLE_BOLD, Font.SIZE_MEDIUM);
   g.setFont(fuente);

   g.drawString("J2ME", getWidth()/2, 10,Graphics.BASELINE|Graphics.HCENTER);

   // Cargar y mostrar gráfico
   try {
      img = Image.createImage("/logo.png");
   } catch (Exception e) {
      System.err.println("error: " + e);
   }

   g.drawImage (img, getWidth()/2, 40, Graphics.HCENTER|Graphics.VCENTER);

 }
}
```

CAPÍTULO 5

Sprites

Durante los capítulos siguientes se profundiza en los diferentes aspectos concernientes a la programación de videojuegos. Ya dispones de las herramientas necesarias para emprender la aventura, así que siéntate cómodamente, flexiona tus dedos y prepárate para la diversión. Para ilustrar las técnicas que se describirán en los próximos capítulos 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. El juego es un homenaje al mítico 1942.

Figura 5.1. Juego 1942 en un Commodore 64

Este capítulo lo vamos a dedicar a los sprites. Seguro que alguna vez has jugado a S*pace 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.

Figura 5.2. Ejes 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).

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

Figura 5.3. Animación de un sprite.

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.

Figura 5.4. Método simple de detección de colisiones

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 otro métodos más precisos que nos permiten detectar colisiones. Consiste en dividir el sprite en pequeñas superficies rectangulares tal y como muestra la próxima figura.

Figura 5.5. Un 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. 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.HCENTER|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'.

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

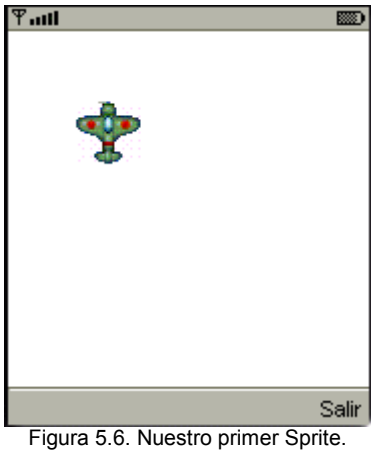


Figura 5.6. Nuestro primer Sprite.

```
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);
 }
}
```

CAPÍTULO 6

Un Universo en tu móvil.

Ahora que hemos desarrollado una herramienta para el control de sprites, vamos a aprender a sacarle partido. Con nuestra librería seremos capaces de mostrar en la pantalla del dispositivo todo lo que va ocurriendo en el juego, pero también hemos de ser capaces de leer la información desde el teclado del móvil para responder a las instrucciones que da el jugador. También es importante que el movimiento del juego sea suave y suficientemente rápido. En este capítulo examinaremos las capacidades de animación de los midlets, incluido el scrolling, así como la interfaz con el teclado.

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.

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 móviles 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

Comenzamos este libro con una introducción a Java. De forma intencionada, y debido a lo voluminoso que es el lenguaje Java, algunos temas no fueron cubiertos. Uno de estos temas fueron los threads. Vamos a verlos someramente, ahora que ya estamos algo más familiarizados con el lenguaje, y lo utilizaremos en nuestro juego.

Muy probablemente el sistema operativo que utilizas tiene capacidades de multiproceso o multitarea. En un sistema de este tipo, puedes ejecutar varias aplicaciones al mismo tiempo. A cada una de estas aplicaciones las denominamos procesos. Podemos decir que el sistema operativo es capaz de ejecutar múltiples procesos simultáneamente. Sin embargo, en ocasiones es interesante que dentro de proceso se lancen uno o más subprocesos de forma simultánea. Vamos a utilizar un ejemplo para aclarar el concepto. Piensa en tu navegador web favorito. Cuando lo lanzas, es un proceso más dentro de la lista de procesos que se estan ejecutando en el sistema operativo. Ahora, supongamos que cargamos en el navegador una web llena de imágenes, e incluso algunas de ellas animadas. Si observas el proceso de carga, verás que no se cargan de forma secuencial una tras otra, sino que comienzan a cargarse varias a la vez. Esto es debido a que el proceso del navegador lanza varios subprocesos, uno por cada imagen, que se encargan de cargarlas, y en su caso, de animarlas de forma independiente al resto de imágenes. Cada uno de estos subprocesos se denomina thread (hilo o hebra en castellano).

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

Cuando jugamos a un juego parece que todo pasa a la vez, en el mismo instante, sin embargo, sabemos que un procesador sólo puede realizar una acción a la vez. La clave es realizar cada una de las acciones tan rápidamente como sea posible y pasar a la siguiente, de forma que todas se completen antes de visualizar el siguiente frame del juego.

El “game loop” o bucle de juego es el encargado de “dirigir” en cada momento que tarea se está realizando. En la figura 6.1. podemos ver un ejemplo de game loop, y aunque más o menos todos son similares, no tienen por que tener exactamente la misma estructura. Analicemos el ejemplo.

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... Fuera de la lógica del juego quedan otras tareas que realizamos en la siguiente fase, como son actualizar el scroll de fondo (si lo hubiera), activar sonidos (si fuera

necesario), realizar trabajos de sincronización, etc.. Ya por último, nos resta volcar todo a la pantalla y mostrar el siguiente frame. Esta fase es llamada “fase de render”.

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.

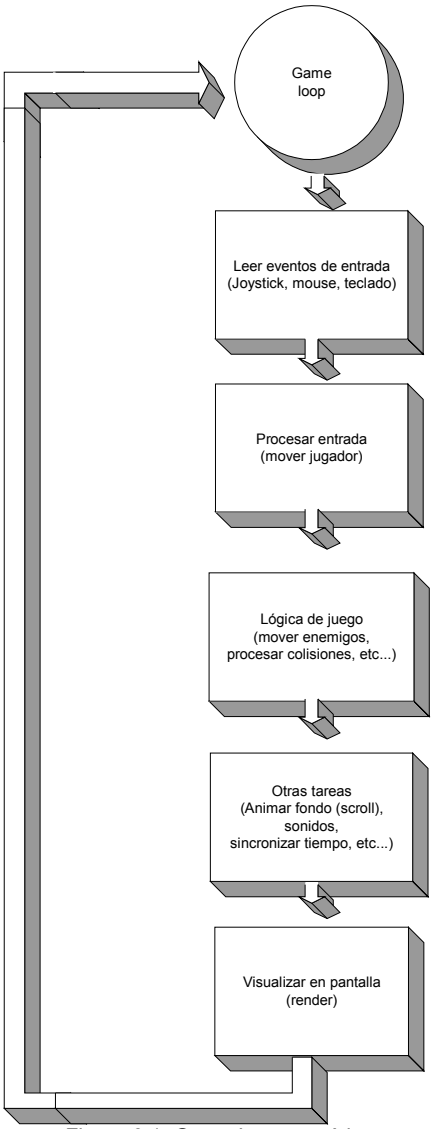


Figura 6.1. Game Loop genérico

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.

Figura 6.2. Uridium

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.

Figura 6.3. Tiles utilizados en nuestro juego

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.

Figura 6.4. Nuestro avión surcando el cielo.

```
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);
 }

}
```

CAPÍTULO 7

Enemigos, disparos y explosiones

Existen multiples técnicas relacionadas con la inteligencia artificial (IA) y que son ampliamente utilizadas en programación de juegos. La IA es un tópico lo suficientemente extenso como para rellenar varios libros del tamaño del que tienes ahora entre manos. Aún así, exploraremos algunas sencillas técnicas que nos permitiran dotar a los aviones enemigos de nuestro juego de una chispa vital. También haremos diaparar a los aviones enemigos y al nuestro, explosiones incluidas.

Tipos de inteligencia

Hay, al menos, tres tendencias dentro del campo de la inteligencia artificial.

- Redes neuronales
- Algoritmos de búsqueda
- Sistemas basados en conocimiento

Son tres enfoque diferentes que tratan de buscar un fin común. No hay un enfoque mejor que los demás, la elección de uno u otro depende de la aplicación.

Una red neuronal trata de simular el funcionamiento del cerebro humano. El elemento básico de una red neuronal es la neurona. En una red neuronal, un conjunto de neuronas trabajan al unísono para resolver un problema. Al igual que un niño tiene que aprender al nacer, una red de neuronas artificial tiene que ser entrenada para poder realizar su cometido. Este aprendizaje puede ser supervisado o no supervisado, dependiendo si hace falta intervención humana para entrenar a la red de neuronas. Este entrenamiento se realiza normalmente mediante ejemplos. La aplicación de las redes neuronales es efectiva en campos en los que no existen algoritmos concretos que resuelvan un problema o sean demasiado complejos de computar. Donde más se aplican es en problemas de reconocimiento de patrones y pronósticos.

El segundo enfoque es el de los algoritmos de búsqueda. Es necesario un conocimiento razonable sobre estructuras de datos como árboles y grafos. Una de las aplicaciones interesantes, sobre todo para videojuegos, es la búsqueda de caminos (pathfinding). Seguramente has jugado a juegos de estrategia como *Starcraft*, *Age of Empires* y otros del estilo. Puedes observar que cuando das la orden de movimiento a uno de los pequeños personajes del juego, éste se dirige al punto indicado esquivando los obstáculos que encuantra en su camino. Este algoritmo de búsqueda en grafos es llamado A*. La figura 7.1. es un ejemplo de árbol. Supongamos que es el mapa de un juego en el que nuestra misión es escapar de una casa.

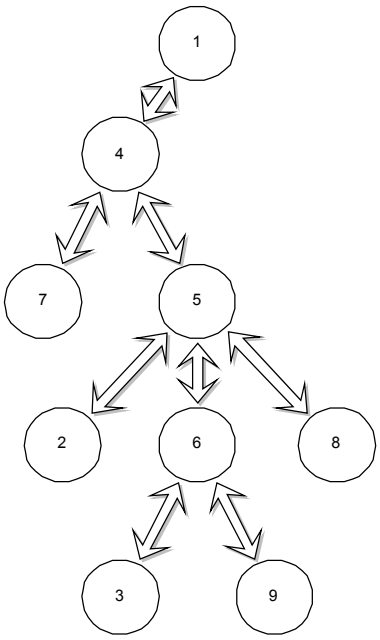


Figura 7.1. Árbol de los posibles caminos dentro del mapa de juego

Cada círculo representa un nodo del árbol. El número que encierra es el número de habitación. Si quisieramos encontrar la salida, usaríamos un algoritmo de búsqueda (por ejemplo A*) para recorrer todos los posibles caminos y quedarnos con el que nos interesa. El objetivo es buscar el nodo 8, que es el que tiene la puerta de salida. El camino desde la habitación 1 es: 1 4 5 8. El algoritmo A* además de encontrar el *nodo objetivo*, nos asegura que es el camino más corto. No vamos a entrar en más detalle, ya que cae fuera de las pretensiones de este libro profundizar en la implementación de algoritmos de búsqueda.

Por último, los sistemas basados en reglas se sirven, valga la redundancia, de conjuntos de reglas y hechos. Los hechos son informaciones relativas al problema y a su universo. Las reglas son precisamente eso, reglas aplicables a los elementos del universo y que permiten llegar a deducciones simples. Veamos un ejemplo:

Hechos: Las moscas tienen alas.
Las hormigas no tienen alas.

Reglas: Si (x) tiene alas, entonces vuela.

Un sistema basado en conocimiento, con estas reglas y estos hechos es capaz de deducir dos cosas. Que las moscas vuelan y que las hormigas no.

Si (la mosca) tiene alas, entonces vuela.

Uno de los problemas de los sistemas basados en conocimiento es que pueden ocurrir situaciones como estas.

Si (la gallina) tiene alas, entonces vuela.

Desgraciadamente para las gallinas, éstas no vuelan. Puedes observar que la construcción para comprobar reglas es muy similar a la construcción IF/THEN de los lenguajes de programación.

Comportamientos y máquinas de estado

Una máquina de estados está compuesta por una serie de estados y una serie de reglas que indican en que casos se pasa de un estado a otro. Estas máquinas de estados nos perniten modelar comportamientos en los personajes y elementos del juego. Vamos a ilustrarlo con un ejemplo. Imagina que en el hipotético juego que hemos planteado unas líneas más arriba hay un zombie. El pobre no tiene una inteligencia demasiado desarrollada y sólo es capaz de andar hasta que se pega contra la pared. Cuando sucede esto, lo único que sabe hacer es girar 45 grados a la derecha y continuar andando. Vamos a modelar el comportamiento del zombie con una máquina de estados. Para ello primero tenemos que definir los posibles estados.

- Andando (estado 1)
- Girando (estado 2)

Las reglas que hacen que el zombie cambie de un estado a otro son las siguientes.

- Si está en el estado 1 y choca con la pared pasa al estado 2.
- Si esta en el estado 2 y ha girado 45 grados pasa al estado 1.

Con estos estados y estas reglas podemos construir el grafo que representa a nuestra máquina de estados.

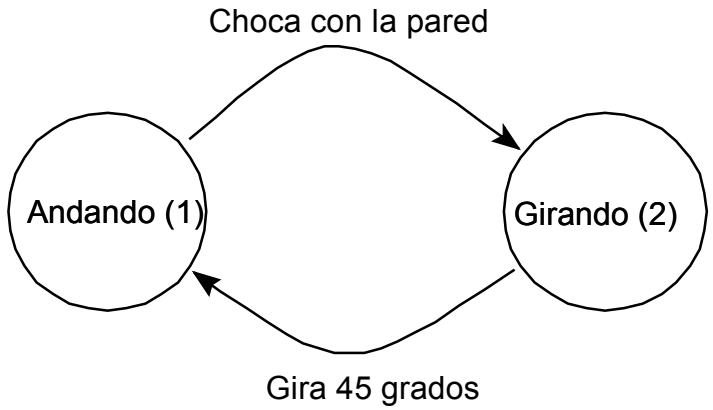


Figura 7.2.

La implementación de la máquina de estado es muy sencilla. La siguiente función simula el comportamiento del zombie.

```
int angulo;

void zombie() {
 int state, angulo_tmp;

  // estado 1
  if (state == 1) {
     andar();
     if (colision()) {
        state=2;
        angulo_tmp=45;
     }
  }
```

```
   // estado 2
   if (state == 2) {
      angulo_tmp=angulo_tmp-1;
      angulo=angulo+1;
      if (angulo_tmp <= 0) {
         state=1;
      }
   }
}
```

Éste es un ejemplo bastante sencillo, sin embargo utilizando este método podrás crear comportamientos inteligentes de cierta complejidad.

Enemigos

En nuestro juego vamos a tener dos tipos de aviones enemigos. El primero de ellos es un avión que cruzará la pantalla en diagonal, en dirección de nuestro avión. Al alcanzar una distancia suficiente a nosotros, disparará un proyectil. El segundo enemigo es algo menos violento, ya que no disparará. Sin embargo, al alcanzar cierta posición de la pantalla realizará un peligroso cambio de treyectoria que nos puede pillar desprevenidos. He aquí las máquinas de estado de ambos comprtamientos.

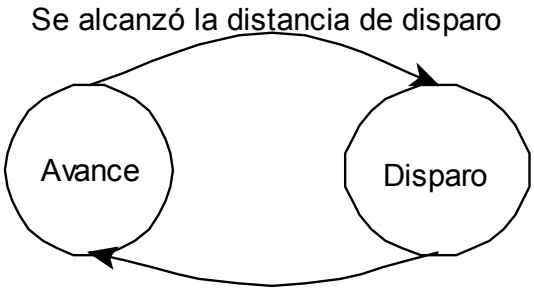


Figura 7.3. Máquina de estados para enemigo tipo 1

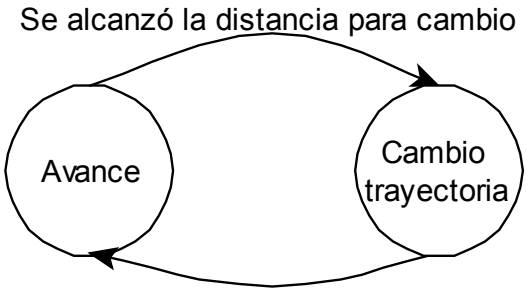


Figura 7.4. Máquina de estados para enemigo tipo 2.

Para representar las naves enemigas, crearemos una clase llamada *Enemy*. Como al fin y al cabo, las naves enemigas no dejan de ser sprites, vamos a crear la clase *Enemy* heredando los métodos y atributos de la clase *Sprite* y añadiendo aquello que necesitemos.

```
class Enemy extends Sprite {

   private int type,state,deltaX,deltaY;
```

```
   public void setState(int state) {
      this.state=state;
   }

   public int getState(int state) {
      return state;
   }

   public void setType(int type) {
      this.type=type;
   }

   public int getType() {
      return type;
   }

   public void doMovement() {

      Random random = new java.util.Random();

      // Los enemigos de tipo 2 cambiaran su trayectoria
      // al alcanzar una posición determinada (pos. 50)
      if (type == 2 && getY() > 50 && state != 2) {

         // paso al estado 2 (movimiento diagonal)
         state = 2;

         if ((Math.abs(random.nextInt()) % 2) + 1 == 1) {
            deltaX=2;
         } else {
            deltaX=-2;
         }
      }

      // movemos la nave
      setX(getX()+deltaX);
      setY(getY()+deltaY);

   }


   public void init(int xhero) {
      deltaY=3;
      deltaX=0;

      if (type == 1) {
         if (xhero > getX()) {
            deltaX=2;
         } else {
            deltaX=-2;
         }
      }
   }


   // Sobrecarga del método draw de la clase Sprite
   public void draw (javax.microedition.lcdui.Graphics g) {
      selFrame(type);
      // llamamos al método 'draw' de la clase padre (Sprite)
      super.draw(g);
   }

   public Enemy(int nFrames) {
      super(nFrames);
   }
}
```

A los atributos de la clase *Sprite* añadimos cuatro más. El atributo `type`, indicará cuál es el tipo de enemigo. En nuestro caso hay dos tipos. Para manejar este atributo dotamos a nuestra clase de los métodos `getType()` y `setType()` para consultar y establecer el tipo del enemigo.

El atributo `state` mantendrá el estado de la nave. La nave de tipo 2, es decir la que cambia su trayectoria, tiene dos estado. En el estado 1 simplemente avanza en horizontal. En el estado 2 su trayectoria es diagonal. Para manejar el estado de los enemigos añadimos las clases `getState()` y `setState()` para consultar y establecer el estado de los enemigos.

Los dos atributos que nos quedan son `deltaX` y `deltaY`, que contienen los desplazamientos en el eje horizontal y vertical respectivamente que se producirá en cada vuelta del game loop.

Al crear una instancia de nuestra clase, lo primero que hemos de hacer en el constructor es llamar a la clase padre, que es *Sprite*, para pasarle el parámetro que necesita para reservar el número de frames. También para inicializar el sprite.

```
super(nFrames);
```

Vamos tambien a sobrecargar el método `draw()` del método *Sprite*. En este método, primero seleccionaremos el tipo de avión que vamos a poner en la pantalla según su tipo, después, llamamos al método `draw()` de la clase padre.

Nuestro enemigo de tipo 1 debe tomar una trayectoria dependiendo de la posicón de nuestro avión. Para ello, necesitamos una forma de comunicarle a la nave enemiga dicha posición. Hemos creado un método llamado `init()` a la que le pasamos como parámetro la posición actual de nuestro avión. En este método ponemos los atributos `deltaX` y `deltaY` a sus valores iniciales.

Por último necesitaremos un método que se encargue de realizar el movimiento de la nave. Este método es `doMovement()`. Su función principal es actualizar la posición de la nave enemiga según los atributos `deltaX` y `deltaY`. También comprobamos la posición del enemigo de tipo 1 para cambiar su estado cuando sea necesario.

Ya disponemos de nuestra clase *Enemy* para manejar a los aviones enemigos. En nuestro juego permitiremos un máximo de 6 enemigos simultáneos en pantalla (realmente habrá menos), así que creamos un array de elementos de tipo *Enemy*. El siguiente paso es inicializar cada uno de estos seis elementos. Vamos a cargar dos frames, uno para la nave de tipo 1 y otro para la de tipo 2. Dependiendo del tipo de la nave, seleccionaremos un frame u otro antes de dibujar el avión.

```
private Enemy[] enemies=new Enemy[7];

// Inicializar enemigos
for (i=1 ; i<=6 ; i++) {
   enemies[i]=new Enemy(2);
   enemies[i].addFrame(1,"/enemy1.png");
   enemies[i].addFrame(2,"/enemy2.png");
   enemies[i].off();
}
```

Durante el trascurso de nuestro juego aparecerá un enemigo cada 20 ciclos del game loop. Cuando necesitemos crear un enemigo, hay que buscar una posición libre en el array de enemigos. Si hay alguno libre, ponemos su estado inicial (posición, tipo, etc...) de forma aleatoria y lo iniciamos (lo activamos).

```
// Creamos un enemigo cada 20 ciclos
if (cicle%20 == 0) {
    freeEnemy=0;

   // Buscar un enemigo libre
   for (i=1 ; i<=6 ; i++) {
      if (!enemies[i].isActive()) {
         freeEnemy=i;
```

```
      }
   }

   // Asignar enemigo si hay una posición libre
   // en el array de enemigos
   if (freeEnemy != 0) {
      enemies[freeEnemy].on();
   enemies[freeEnemy].setX((Math.abs(random.nextInt()) % getWidth()) + 1);
      enemies[freeEnemy].setY(0);
      enemies[freeEnemy].setState(1);
      enemies[freeEnemy].setType((Math.abs(random.nextInt()) % 2) + 1);
      enemies[freeEnemy].init(hero.getX());
   }
}
```

En cada ciclo del game loop, hemos de actualizar la posición de cada enemigo y comprobar si ha salido de la pantalla.

```
// Mover los enemigos
for (i=1 ; i<=6 ; i++) {
   if (enemies[i].isActive()) {
      enemies[i].doMovement();
   }

   // Mirar si la nave salió de la pantalla
   if ((enemies[i].getY() > getHeight()) || (enemies[i].getY() < 0)) {
      enemies[i].off();
   }
}
```

Disparos y explosiones

Ahora que conocemos una técnica para que nuestros aviones enemigos sean capaces de comportarse tal y como queremos que lo hagan, estamos muy cerca de poder completar nuestro juego. Lo siguiente que vamos a hacer es añadir la capacidad de disparar a nuestro avión. La gestión de los disparos y las explosiones va a ser muy similar a la de los aviones enemigos. Vamos a crear una clase a la que llamaremos *Bullet*, descendiente de la clase *Sprite* y que representará un disparo.

```
class Bullet extends Sprite {
   private int owner;

   public Bullet(int nFrames) {
      super(nFrames);
   }

   public void setOwner(int owner) {
      this.owner=owner;
   }

   public int getOwner() {
      return owner;
   }

   public void doMovement() {

      // si owner = 1 el disparo es nuestro
      // si no, es del enemigo
      if (owner == 1) {
         setY(getY()-6);
      } else {
         setY(getY()+6);
      }
   }

   // Sobrecarga del método draw de la clase Sprite
   public void draw (javax.microedition.lcdui.Graphics g) {
      selFrame(owner);
```

```
      // llamamos al método 'draw' de la clase padre (Sprite)
      super.draw(g);
   }

}
```

Añadimos un atributo llamado `owner` que indica a quien pertenece el disparo (a un enemigo o a nuestro avión). Para gestionar este atributo disponemos de los métodos `getOwner()` y `setOwner()`.

Este atributo lo vamos a utilizar para decidir si movemos el disparo hacia arriba o hacia abajo y para comprobar las colisiones.

En cuanto a las explosiones, vamos a crear una clase llamada *Explode*, también descendiente de *Sprite*.

```
class Explode extends Sprite {

   private int state;

   public Explode(int nFrames) {
      super(nFrames);
      state=1;
   }

   public void setState(int state) {
      this.state=state;
   }

   public int getState() {
      return state;
   }

   public void doMovement() {
      state++;

      if (state > super.frames())
         super.off();
   }

   // Sobrecarga del método draw de la clase Sprite
   public void draw (javax.microedition.lcdui.Graphics g) {
      selFrame(state);
      // llamamos al método 'draw' de la clase padre (Sprite)
      super.draw(g);
   }

}
```

El único atributo que añadimos es `state`, que nos indicará el estado de la explosión. En la práctica, el estado de la explosión va a ser el frame actual de su animación interna, de hecho, su clase `doMovement()` lo único que hace es aumentar el frame para realizar la animación.

La inicialización y gestión de explosiones y disparos es idéntica a la de los aviones enemigos, por lo que no vamos a entrar en mucho más detalle.

Nos resta comprobar las colisiones entre los sprites del juego. Vamos a realizar tres comprobaciones.

- Colisión héroe – enemigo
- Colisión héroe – disparo
- Colisión enemigo – disparo

Ten en cuenta que hay 6 posibles enemigos y 6 posibles disparos a la vez en pantalla, por lo que hay que realizar todas las combinaciones posibles.

```
// Colisión heroe-enemigo
for (i=1 ; i<=6 ; i++) {
   if (hero.collide(enemies[i])&&enemies[i].isActive()&&shield == 0) {
      createExplode(hero.getX(),hero.getY());
      createExplode(enemies[i].getX(),enemies[i].getY());
      enemies[i].off();
      collision=true;
      }
   }

   // Colisión heroe-disparo
   for (i=1 ; i<=6 ; i++) {
      if (aBullet[i].isActive() && hero.collide(aBullet[i]) && aBullet[i].getOwner() != 1  &&
shield == 0) {
         createExplode(hero.getX(),hero.getY());
         aBullet[i].off();
         collision=true;
      }
   }

// colisión enemigo-disparo
for (i=1 ; i<=6 ; i++) {
   if (aBullet[i].getOwner() == 1 && aBullet[i].isActive()) {
      for (j=1 ; j<=6 ; j++) {
         if (enemies[j].isActive()) {
            if (aBullet[i].collide(enemies[j])) {
               createExplode(enemies[j].getX(),enemies[j].getY());
               enemies[j].off();
               aBullet[i].off();
               score+=10;
            }
         }
      }
   }
}
```

El resultado nuestro juego puede verse en la siguiente figura. El listado fuente completo del juego puede consultarse en el apéndice A.

Figura 7.5. Nuestro juego M1945 en acción.

Sonido

Nuestro juego M1945 no tiene sonido. La razón para ello ha sido mantener la compatibilidad con los dispositivos con soporte MIDP 1.0, que es el más extendido por ahora. Desgraciadamente MIDP 1.0. no ofrece ninguna capacidad para reproducir sonidos, por lo que los fabricantes han creado APIs propias e incompatibles entre sí. A partir de la versión 2.0 de MIDP sí se ha añadido soporte multimedia, aunque aún no se han extendido demasiado estos dispositivos en el mercado. La API encargada del sonido se llama *MIDP 2.0 Media API*. Hasta este capítulo, todo lo expuesto es compatible con MIDP 1.0, incluido el juego M1945. Se ha optado, pues, por separar en un capítulo aparte lo concerniente al sonido para no romper esta compatibilidad.

La API multimedia esta compuesta por tres partes:

- Manager
- Player
- Control

La función de la clase *Manager* es crear objetos de tipo *Player*. Un *Player* es un objeto capaz de reproducir un tipo de información multimedia, ya sea audio o video. Por lo tanto, el *Manager* debe generar un tipo diferente de *Player* según la naturaleza de lo que queramos reproducir. Para utilizar estos objetos hemos de importar el paquete *javax.microedition.media*. Finalmente, la clase *Control* nos premite controlar y gestionar un objeto de tipo *Player*. Esta clase se encuentra en el paquete *javax.microedition.media.control*.

Sonidos

Hay eventos en los juegos que generan sonidos, como una explosión o un disparo. Este tipo de sonido suelen ser *samples* digitales. El formato más habitual para almacenar estos sonidos es el formato WAV. La siguiente línea crea un *Player* para un archivo WAV.

```
Player sonido =
Manager.createPlayer("http://www.dominio.com/music.wav");
```

Si queremos crear un *Player* para un objeto almacenado en nuestro archivo JAR, hemos de utilizar una corriente de entrada para leerlo. En este caso, hemos de indicarle al Manager que tipo de archivo es.

```
InputStream in =
   getClass().getResourceAsStream("/explosion.wav");
Player sonido = Manager.createPlayer(in, "audio/x-wav");
```

Debes capturar las excepciones `IOException` y `MediaException` para crear este *Player*. Para reproducir el sonido usamos el método `start()` del *Player*.

```
try {
 sonido.start();
```

```
} catch (MediaException me) { }
```

Música

Además de samples, la API multimedia nos permite reproducir notas musicales (tonos). La forma más simple de reproducir un tono es utilizar el método `playTone()` de la clase *Manager*.

```
try {
  Manager.playTone(ToneControl.C4, 100, 80);
} catch (Exception e){}
```

Este método tiene tres parámetros. El primero es la frecuencia del tono. En este caso hemos utilizado la constalnte `ToneControl.C4`, que es la frecuencia de la nota Do central. Otra constante interesante es `ToneControl.SILENCE`. El segundo parámetro es la duración de la nota en milisegundos, y el tercero el volumen de reproducción.

Reproducir una melodía con la ayuda del método playTone() puede ser un trabajo algo arduo. Es por ello que la API multimedia nos ofrece otra forma de reproducir secuencias de notas. Una *secuencia* es un array de bytes con un formato muy concreto. El array se va rellenando con pares de bytes cuyo significado analizaremos con un ejemplo.

```
byte[] secuencia = {
 ToneControl.VERSION, 1,
 ToneControl.TEMPO, tempo,

 // comienzo del bloque 0
 ToneControl.BLOCK_START, 0,

 // notas del bloque 0
 C4,d, F4,d, F4,d, C4,d, F4,d, F4,d, C4,d, F4,d,

 // fin del bloque 0
 ToneControl.BLOCK_END, 0,

 // inicio del bloque 1
 ToneControl.BLOCK_START, 1,

 // notas del bloque 1
 C4,d, E4,d, E4,d, C4,d, E4,d, E4,d, C4,d, E4,d,

 // fin del bloque 1
 ToneControl.BLOCK_END, 1,

 // reproducir bloque 0
 ToneControl.PLAY_BLOCK, 0,

 // reproducir bloque 1
 ToneControl.PLAY_BLOCK, 1,

 // reproducir bloque 0
 ToneControl.PLAY_BLOCK, 0,
};
```

Podemos observar que la secuencia está dividida en tres secciones bien diferenciadas. En la primera establecemos la versión (del formato de secuencia) y el tempo de la melodía.

Observa como la información se codifica en pares de bytes. El primero indica el atributo para el que queremos establecer un valor, y el segundo es el valor mismo.

En la segunda sección de la secuencia definimos bloques de notas. Las notas comprendidas entre `ToneControl.BLOCK_END` y `ToneControl.BLOCK_START` forman un bloque. Podemos definir tantos bloque como necesitemos. Dentro de un bloque, las notas van definidas en pares, cuyo primer byte es la nota y el segundo es la duración. Finalmente, la tercera sección indica el orden de reproducción de cada bloque de notas.

El código encargado de reproducir la secuencia es el siguiente.

```
Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
p.realize();
ToneControl c = (ToneControl)p.getControl("ToneControl");
c.setSequence(secuencia);
p.start();
```

Vamos a reunir en el siguiente ejemplo práctico todo lo expuesto en el presente capítulo.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import java.io.*;

public class Sonido extends MIDlet implements CommandListener {
  private Display display;
  private Form formulario;
  private Command exit;
  private Command wav, nota, secuencia;


  public Sonido() {

     display = Display.getDisplay(this);

     exit = new Command("Salir", Command.EXIT, 1);
     wav = new Command("WAV", Command.SCREEN, 2);
     nota = new Command("Nota", Command.SCREEN, 2);
     secuencia = new Command("Secuencia", Command.SCREEN, 2);

     formulario = new Form("Reproducir.");
     formulario.addCommand(exit);
     formulario.addCommand(wav);
     formulario.addCommand(nota);
     formulario.addCommand(secuencia);
     formulario.setCommandListener(this);
}


public void startApp() {
  display.setCurrent(formulario);
}

public void pauseApp() {}

public void destroyApp(boolean unconditional) {}

public void commandAction(Command c, Displayable s) {
  if (c == exit) {
     destroyApp(false);
     notifyDestroyed();
  } else {
     if (c == wav)
        playWav();

     if (c == nota)
        playNota();

     if (c == secuencia)
        playSecuencia();
  }
}


public void playWav() {
  try {
     // Abrir corriente de datos del archivo de sonido
     InputStream in = getClass().getResourceAsStream("/explosion.wav");
     Player p = Manager.createPlayer(in, "audio/x-wav");

     // comenzar reproducción
     p.start();

  } catch (Exception e) {
     Alert alr = new Alert("Error", "No se pudo reproducir el sonido.", null, AlertType.ERROR);
     alr.setTimeout(Alert.FOREVER);
     display.setCurrent(alr, formulario);
  }
}


public void playNota() {
  try {
   // reproducir nota
    Manager.playTone(ToneControl.C4, 100, 80);
```

```
   } catch (Exception e){}
 }


 public void playSecuencia() {
   byte tempo = 30;
   byte d = 8;

   // Creamos las notas a partir del Do central
   byte C4 = ToneControl.C4;;
   byte D4 = (byte)(C4 + 2);
   byte E4 = (byte)(C4 + 4);
   byte F4 = (byte)(C4 + 5);
   byte G4 = (byte)(C4 + 7);
   byte silencio = ToneControl.SILENCE;

   byte[] secuencia = {
    ToneControl.VERSION, 1,
    ToneControl.TEMPO, tempo,

    // comienzo del bloque 0
    ToneControl.BLOCK_START, 0,

    // notas del bloque 0
    C4,d, F4,d, F4,d, C4,d, F4,d, F4,d, C4,d, F4,d,

    // fin del bloque 0
    ToneControl.BLOCK_END, 0,

    // inicio del bloque 1
    ToneControl.BLOCK_START, 1,

    // notas del bloque 1
    C4,d, E4,d, E4,d, C4,d, E4,d, E4,d, C4,d, E4,d,

    // fin del bloque 1
    ToneControl.BLOCK END, 1,

    // reproducir bloque 0
    ToneControl.PLAY_BLOCK, 0,

    // reproducir bloque 1
    ToneControl.PLAY BLOCK, 1,

    // reproducir bloque 0
    ToneControl.PLAY_BLOCK, 0,
   };

   try{
    Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
    p.realize();
    ToneControl c = (ToneControl)p.getControl("ToneControl");
    c.setSequence(secuencia);
    p.start();
   } catch (IOException ioe) {
   } catch (MediaException me) {}
 }
}
```

CAPÍTULO 9

Almacenamiento. RMS

Hasta ahora hemos usado la memoria principal del móvil (la RAM) para almacenar datos temporales, pero al salir del midlet, estos datos son eliminados. Esto plantea algunos problemas. Por ejemplo, ¿cómo podemos almacenar las puntuaciones máximas de nuestro juego?

Un dispositivo móvil (al menos por ahora) no dispone de disco duro donde almacenar información permanentemente. J2ME resuelve el problema mediante el RMS (Record Management System). RMS es un pequeño sistema de bases de datos muy sencillo, pero que nos permite añadir información en una memoria no volátil del móvil. RMS no tiene nada que ver con JDBC debido a las limitaciones de los dispositivos J2ME, por lo tanto, el acceso y almacenamiento de la información se hace a mucho más bajo nivel. RMS no puede ser consultado con sentencias SQL ni nada parecido. En una base de datos RMS, el elemento básico es el registro (record). Un registro es la unidad de información más pequeña que puede ser almacenada. Los registros son almacenados en un *recordStore* que puede visualizarse como una colección de registros. Cuando almacenamos un registro en el *recordStore*, a éste se le asigna un identificador único que identifica unívocamente al registro.

Para poder utilizar RMS hemos de importar el paquete *javax.microedition.rms*. Este paquete nos provee de la clase R*ecordStore* y de cuatro interfaces, de las cuales solamente profundizaremos en la interfaz *RecordEnumeration*.

Trabajando con RMS

Cubrir todas las capacidades de RMS se escapa del ámbito de este libro. Sin entrar en profundidades, vamos a ver cómo realizar las operaciones básicas de almacenamiento, lectura y borrado de registros.

Abrir y cerrar un recordStore

Antes de poder almacenar un registro hemos de abrir un *recordStore* con el método `openRecordStore()`.

```
static RecordStore openRecordStore(String nombre, bolean crear)
```

El parámetro `nombre` es el nombre de la base de datos. El nombre puede tener un tamaño de 32 caracteres. El parámetro `crear`, si tiene su valor a *true*, creará la base de datos si no existe. Cuando creamos un *recordStore*, sólo puede ser accedido desde la suite de MIDlets que la creó.

Cuando terminamos de utilizar el *recordStore*, hemos de cerrarlo:

```
RecordStore.closeRecordStore();
```

Añadir registros

Una vez abierto nuestro recordStore podemos comenzar a añadir registros con el método `addRecord()`.

```
public int addRecord(byte[] dato,int offset, int numBytes)
```

El primer parámetro es el dato que queremos almacenar. Es un array de bytes. El `offset` es la posición a partir de la cual (dentro del array) se va a almacenar el dato. Finalmente, `numBytes` es el número de bytes que se van a almacenar. El método retorna el identificador que el RMS ha asignado al registro.

El método *addRecord* puede lanzar la excepción *RecordStoreException*, por lo tanto hemos de capturarla.

```
try {
   int id = recordStore.addRecord (datos, 0, datos.length);
} catch (RecordStoreException e) {}
```

Leer registros

El método getRecord() permite acceder al registro que deseemos, siempre que conozcamos su identificador.

```
public byte[] getRecord(int Id)
```

No es necesario que almacenemos y mantengamos una lista con todos los identificadores de los registros. Un poco más adelante veremos el método *recordEnumeration* que nos permitirá conocer el identificador de cada registro. Al igual que con el método `addRecord()`, hemos de capturar la excepción *RecordStoreException*.

```
byte[] dato = null;

try {
   dato = recordStore.getRecord(id);
} catch (RecordStoreException e) {}
```

Borrar registros

El borrado de registros se realiza con el método deleteRecord().

```
public void deleteRecord(int recordId)
```

Al igual que con la escritura y lectura de registros hemos de tener en cuenta que puede provocar la excepción *RecorStoreException*.

```
try {
   recordStore.deleteRecord(id);
} catch (RecordStoreException e) {}
```

Recorriendo registros

Vamos a valernos del objeto *RecordEnumeration* para recorrer todos los registros almacenados en la base de datos. Para crear una enumeración utilizamos el método `enumerateRecords()`.

```
public RecordEnumeration
enumerateRecords(RecordFilter filtro,RecordComparator,
comparador,boolean Actualizar)
```

Los dos primeros parámetros sirven para personalizar el recorrido de los registros. No entraremos en detalle, pero, gracias al primero podremos filtrar la búsqueda, y con el segundo podemos recorrer los registros de forma ordenada. El parámetro `Actualizar` indica si la enumeración debe actualizarse cuando realicemos alguna operación de inserción o borrado de registros. Si vas a hacer un recorrido rápido por los registros es mejor ponerlo a *false* para evitar la sobrecarga.

```
RecordEnumeration registro = null;

try {
   registro = recordStore.enumerateRecords(null, null, false);
   while (registro.hasNextElement())
      System.out.println (registro.nextRecordId());
} catch (Exception e) {}
```

Hay dos métodos interesantes del *RecordEnumeration*: `hasNextElement()` que devolverá el valor *true* si hay un siguiente elemento disponible para ser leído. Cada vez que leemos un elemento se adelanta el puntero al siguiente. El método `nextRecordId()` devuelve el identificador del siguiente registro.

El siguiente código muestra un ejemplo completo de uso de RMS.

```
import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.rms.*;

public class RMS extends MIDlet {

 // nombre de la BD
 static final String BD = "datos";

 String dato;
 int id, i;
 char b;

    public RMS() {

        RecordStore rs = null;

        // Borramos la BD si tenía algo
        try {
            RecordStore.deleteRecordStore(BD);
        } catch( Exception e ){}

        try {
           // Abrimos el recordStore
            rs = RecordStore.openRecordStore(BD, true);

         guardaRegistro(rs,"Datos del registro 1");
         guardaRegistro(rs,"Datos del registro 2");
         guardaRegistro(rs,"Datos del registro 3");

            // Leemos los registros
            RecordEnumeration registros = rs.enumerateRecords(null, null, false);

          // Recorremos todos los elementos
          while (registros.hasNextElement()) {

            // Obtenemos el ID del siguiente registro
            verRegistro(rs, registros.nextRecordId());
         }

           rs.closeRecordStore();

      } catch( RecordStoreException e ){
            System.out.println( e );
       }


       notifyDestroyed();
    }

    public void verRegistro(RecordStore rs, int id) {

       try {
         ByteArrayInputStream bais = new ByteArrayInputStream(rs.getRecord(id));
         DataInputStream is = new DataInputStream(bais);

         // leemos el registro
         try {
            dato = is.readUTF();
            System.out.println("-> "+dato);
            } catch (EOFException eofe) {
         } catch (IOException ioe) {}
      } catch (RecordStoreException e) {}
    }



    public void guardaRegistro(RecordStore rs, String dato) {

       ByteArrayOutputStream baos = new ByteArrayOutputStream();
      DataOutputStream os = new DataOutputStream(baos);

      try {
         // guadar el dato
         os.writeUTF(dato);
      } catch (IOException ioe) {}
```

```
      // extraer el array de butes
      byte[] b = baos.toByteArray();

      // lo añadimos al recordStore
      try {
          rs.addRecord(b, 0, b.length);
      } catch (RecordStoreException rse) {}
    }


    public void destroyApp( boolean unconditional ) {}

    public void startApp() {}

    public void pauseApp() {}
}
```

CAPÍTULO 10

Comunicaciones

Un teléfono móvil es capaz de realizar conexiones con un servidor. Esto abre un vasto campo de aplicaciones y también muchas posibilidades en el campo de los juegos. Una posible aplicación puede ser un servidor que almacene las puntuaciones máximas entre todos los jugadores de un juego, o incluso la posibilidad de juego online contra otros jugadores.

En principio un MIDlet puede establecer diversos tipos de conexiones: *Sockets*, *http*, *https*, *datagramas*, y otras, sin embargo, el standard sólo obliga a la implementación del protocolo *http*, así que dependiendo del dispositivo dispondremos de unos tipos de conexiones u otras. Para mantener la compatibilidad es aconsejable utilizar las basadas en el protocolo *http* y, por lo tanto, limitarnos a este tipo de conexión. La clase *Connector*, que se encuentra en el paquete *javax.microedition.io,* es la base de las comunicaciones en los MIDlets. Para abrir una conexión utilizamos el método `open()` de la clase *Connector*.

```
Connector.open(String conexion);
```

El parámetro conexión es la URL a la que queremos conectar. El formato de la URL es el standard. Los siguientes son formatos válidos.

```
http://www.dominio.com
http://www.dominio.com:8080
http://www.dominio.com/script.jsp?param1=1&param2=2
```

Una forma muy estandarizada de comunicación con un servidor es mediante llamadas a scripts escritos en JSP, PHP o cualquier otro lenguaje de script, y pasando los parámetros necesarios, ya sea por método *post* o método *get*. Este script procesará la información y hará uso de ella, por ejemplo, almacenándola en una base de datos. La escritura de scripts se escapa del ámbito de este libro, sin embargo, podrás encontrar numerosas publicaciones sobre el tema además de gran cantidad de información en Internet.

Además de conexiones http, podemos abrir otros tipos de conexiones siempre que nuestro teléfono móvil lo soporte.

`Socket`	`socket://www.dominio.com:8000`
`Datagramas`	`datagram://www.dominio.com:8000`
`Archivos`	`file:/datos.txt`
`Puertos`	`comm:0;baudrate=9600`

Una vez abierto el canal de comunicación podemos utilizar los métodos `openInputStream()` y `openOutputStream()` de la clase *Connector* para abrir una corriente de lectura o escritura mediante la cual, leer y escribir información.

El método `openInputStream()` devuelve un objeto del tipo *InputStream*, que representa una corriente de entrada de bytes. Contiene algunos métodos que permiten controlar la entrada de datos. Los más interesantes son:

`int available()`	`Número de bytes disponibles en la corriente de lectura`
`void close()`	`Cierra la corriente`
`abstract int read()`	`Lee el siguiente byte de la corriente`

De forma similar, el método `openOutputStream()` devuelve un objeto de tipo *OutputStream*, que representa una corriente de salida de bytes. Los métodos más interesantes son:

`void close()`	`Cierra la corriente`
`void flush()`	`Fuerza la salida de los bytes almacenados en el buffer`
`void write(byte[] b)` `abstract void write(int b)`	`Escribe un byte o un array de bytes en la corriente de salida`

Nos resta conocer la clase *HttpConnection*, que gestiona una conexión basada en el protocolo *http*. Podemos crear una conexión de este tipo mediante el método `open()` de la clase *Connector* tal y como ya vimos.

```
HttpConnection c = (HttpConnection)Connector.open(url);
```

Algunos métodos interesantes son los siguientes:

`String getHost()`	`Devuelve el nombre del host remoto`
`int getPort()`	`Devuelve el puerto de conexión`
`String getQuery()`	`Devuelve los parámetros pasados en la URL`
`String getRequestMethod()`	`Devuelve el método de petición`
`int getResponseCode()`	`Devuelve el código de respuesta`
`String getResponseMessage()`	`Devuelve el mensaje de respuesta`
`String getURL()`	`Devuelve la URL de la conexión`
`void setRequestMethod(String method)`	`Establece el método de petición (GET, POST o HEAD)`

Para ilustrar todo durante el capítulo, vamos a desarrollar un programa que es capaz de conectar con un servicio web y obtener información. Nos conectaremos a una web de noticias (weblog) y extraeremos las cabeceras de las noticias para mostrarlas en nuestro móvil. La web que vamos a utilizar es *www.barrapunto.com*.

Figura 10.1. Web de barrapunto.com

Esta web permite acceder a la información en formato XML en la URL *http://backends.barrapunto.com/barrapunto.xml*, que produce una salida como ésta (reproducimos aquí una sóla noticia):

```
<?xml version="1.0" encoding="iso-8859-1" ?>
  <backslash xmlns:backslash="http://barrapunto.com/backslash.dtd">
    <story>
      <title>Tutorial de Secure Shell</title>
      <url>http://barrapunto.com/article.pl?sid=04/01/21/1048222</url>
      <time>2004-01-21 10:46:17</time>
      <author>fernand0</author>
      <department>ssh-scp-ssh-keygen</department>
      <topic>85</topic>
      <comments>36</comments>
      <section>articles</section>
      <image>topicseguridad.png</image>
    </story>
  <story>
 </backslash>
```

Los títulos de las noticias están encerradas entre las etiquetas *<title>* y *</title>*, así que vamos a extraer el texto que se encuentre entre ambas.

Figura 10.2. Salida de nuestro lector de noticias.

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


public class Conexion extends MIDlet {

    private Display display;

    // Url que vamos a consultar
    String url = "http://backends.barrapunto.com/barrapunto.xml";

   public Conexion() {
       display = Display.getDisplay(this);
    }

    public void startApp() {
      try {
         verNoticias(url);
      } catch (IOException e) {}
   }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    void verNoticias(String url) throws IOException {
        HttpConnection c = null;
        InputStream is = null;
        StringBuffer b = new StringBuffer();
        StringBuffer salida = new StringBuffer();
        TextBox t = null;

        try {
          c = (HttpConnection)Connector.open(url);
          c.setRequestMethod(HttpConnection.GET);
          c.setRequestProperty("IF-Modified-Since", "10 Nov 2000 17:29:12 GMT");
          c.setRequestProperty("User-Agent","Profile/MIDP-1.0 Configuration/CLDC-1.0");
          c.setRequestProperty("Content-Language", "es-ES");
          is = c.openInputStream();
          int ch, i, j;

          // leer los datos desde la URL
          while ((ch = is.read()) != -1) {
            b.append((char) ch);
            if (ch == '\n') {
               if (b.toString().indexOf("<title>") > 0) {
                  i = b.toString().indexOf("<title>")+7;
                  j = b.toString().indexOf("</title>");
                salida.append(b.toString().substring(i,j));
                salida.append("\n-------------------\n");
                }
             b.delete(0,b.length());
             }
          }

          // mostrar noticias en la pantalla
          t = new TextBox("Noticias en barrapunto.com", salida.toString(), 1024, 0);
        } finally {
           if(is!= null) {
              is.close();
           }
           if(c != null) {
              c.close();
           }
        }
        display.setCurrent(t);
    }
}
```

Código Fuente de M1945

M1945.java

```
package mygame;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;
import java.io.*;

public class M1945 extends MIDlet implements CommandListener {

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

 public M1945() {
   display=Display.getDisplay(this);
   exitCommand = new Command("Salir",Command.SCREEN,2);
   playCommand = new Command("Jugar",Command.CANCEL,2);
   endCommand  = new Command("Salir",Command.SCREEN,2);

   screen=new SSCanvas();

   screen.addCommand(playCommand);
   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) {
      if (screen.isPlaying()) {
         screen.quitGame();
      } else {
         destroyApp(false);
         notifyDestroyed();
      }
   }

   if (c == playCommand && !screen.isPlaying()) {
      // Play!!!
      new Thread(screen).start();
   }

 }

}


class SSCanvas extends Canvas implements Runnable {

 private int score,sleepTime,cicle,lives,shield;
 private int indice_in, indice, xTiles, yTiles;
 private boolean playing,fireOn=false;
 private boolean done;
 private int deltaX,deltaY;
 private Hero hero=new Hero(1);
 private Enemy[] enemies=new Enemy[7];
 private Bullet[] aBullet=new Bullet[7];
```

```
private Sprite intro=new Sprite(1);
private Explode[] explode=new Explode[7];
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");
  intro.addFrame(1,"/intro.png");

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

void start() {
   int i;

  playing = true;
  sleepTime = 50;
  hero.setX(getWidth()/2);
  hero.setY(getHeight()-20);
  deltaX=0;
  deltaY=0;
  cicle=0;
  xTiles=7;
  yTiles=8;
  indice=map.length-(xTiles*yTiles);
  indice_in=0;
  score=0;
  lives=3;
  shield=0;


  // Inicializar enemigos
  for (i=1 ; i<=6 ; i++) {
     enemies[i]=new Enemy(2);
     enemies[i].addFrame(1,"/enemy1.png");
     enemies[i].addFrame(2,"/enemy2.png");
     enemies[i].off();
  }

  // Inicializar balas
  for (i=1 ; i<=6 ; i++) {
     aBullet[i]=new Bullet(2);
     aBullet[i].addFrame(1,"/mybullet.png");
     aBullet[i].addFrame(2,"/enemybullet.png");
     aBullet[i].off();
  }

  // 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");

   // Inicializamos explosiones
   for (i=1 ; i<=6 ; i++) {
      explode[i]=new Explode(6);

      explode[i].addFrame(1,"/explode1.png");
      explode[i].addFrame(2,"/explode2.png");
      explode[i].addFrame(3,"/explode3.png");
      explode[i].addFrame(4,"/explode4.png");
      explode[i].addFrame(5,"/explode5.png");
      explode[i].addFrame(6,"/explode6.png");
   }
}


 void computeEnemies() {
   int freeEnemy,i;
   Random random = new java.util.Random();

    // Creamos un enemigo cada 20 ciclos
    if (cicle%20 == 0) {

       freeEnemy=0;

      // Buscar un enemigo libre
      for (i=1 ; i<=6 ; i++) {
         if (!enemies[i].isActive()) {
            freeEnemy=i;
         }
      }

      // Asignar enemigo si hay una posición libre
      // en el array de enemigos
      if (freeEnemy != 0) {
         enemies[freeEnemy].on();
         enemies[freeEnemy].setX((Math.abs(random.nextInt()) % getWidth()) + 1);
         enemies[freeEnemy].setY(0);
         enemies[freeEnemy].setState(1);
         enemies[freeEnemy].setType((Math.abs(random.nextInt()) % 2) + 1);
         enemies[freeEnemy].init(hero.getX());
      }
    }

    // Mover los enemigos
    for (i=1 ; i<=6 ; i++) {
      if (enemies[i].isActive()) {
         enemies[i].doMovement();
      }

      // Mirar si la nave salió de la pantalla
      if ((enemies[i].getY() > getHeight()) || (enemies[i].getY() < 0)) {
         enemies[i].off();
      }
    }


 }


 void computeBullets() {
   int freeBullet,theEnemy,i,j;

   // Crear disparo del jugador
   freeBullet=0;
   if (fireOn) {
      // Buscar un disparo libre
      for (i=1 ; i<=6 ; i++) {
         if (!aBullet[i].isActive()) {
            freeBullet=i;
         }
      }
```

```
      if (freeBullet !=0) {
         aBullet[freeBullet].on();
         aBullet[freeBullet].setX(hero.getX());
         aBullet[freeBullet].setY(hero.getY()-10);
         aBullet[freeBullet].setOwner(1);
      }
   }

   // Crear disparo de enemigos
   freeBullet=0;
   theEnemy=0;

   for (i=1 ; i<=6 ; i++) {
      if (enemies[i].getType() == 1 && enemies[i].isActive() && enemies[i].getY() >
getHeight()/2 && enemies[i].getY() < (getHeight()/2)+5) {
         // Buscar un disparo libre
         for (j=1 ; j<=6 ; j++) {
            if (!aBullet[j].isActive()) {
               freeBullet=j;
               theEnemy=i;
            }
         }

         if (freeBullet !=0) {
            aBullet[freeBullet].on();
            aBullet[freeBullet].setX(enemies[theEnemy].getX());
            aBullet[freeBullet].setY(enemies[theEnemy].getY()+10);
            aBullet[freeBullet].setOwner(2);
         }
      }
   }


   // Mover los disparos
   for (i=1 ; i<=6 ; i++) {
      if (aBullet[i].isActive()) {
         aBullet[i].doMovement();
      }

      // Mirar si el disparo salió de la pantalla
      if ((aBullet[i].getY() > getHeight()) || (aBullet[i].getY() <= 0)) {
         aBullet[i].off();
      }
   }

 }


 void computeExplodes() {
    int i;

   for (i=1 ; i<=6 ; i++) {
      explode[i].doMovement();
   }
 }

 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 createExplode(int posx, int posy) {

    int freeExplode,i;
```

```
    freeExplode=0;

   // Buscar una explosión libre
   for (i=1 ; i<=6 ; i++) {
      if (!explode[i].isActive()) {
         freeExplode=i;
      }
   }

   if (freeExplode !=0) {
      explode[freeExplode].setState(1);
      explode[freeExplode].on();
      explode[freeExplode].setX(posx);
      explode[freeExplode].setY(posy);
   }

 }


 void checkCollide() {

   int i,j;
   boolean collision;

   collision=false;

   // Colisión heroe-enemigo
   for (i=1 ; i<=6 ; i++) {
      if (hero.collide(enemies[i]) && enemies[i].isActive() && shield == 0) {
         createExplode(hero.getX(),hero.getY());
         createExplode(enemies[i].getX(),enemies[i].getY());
         enemies[i].off();
         collision=true;
      }
   }

   // Colisión heroe-disparo
   for (i=1 ; i<=6 ; i++) {
      if (aBullet[i].isActive() && hero.collide(aBullet[i]) && aBullet[i].getOwner() != 1  &&
shield == 0) {
         createExplode(hero.getX(),hero.getY());
         aBullet[i].off();
         collision=true;
      }
   }

   // colisión enemigo-disparo
   for (i=1 ; i<=6 ; i++) {
      if (aBullet[i].getOwner() == 1 && aBullet[i].isActive()) {
         for (j=1 ; j<=6 ; j++) {
            if (enemies[j].isActive()) {
               if (aBullet[i].collide(enemies[j])) {
                  createExplode(enemies[j].getX(),enemies[j].getY());
                  enemies[j].off();
                  aBullet[i].off();
                  score+=10;
               }
            }
         }
      }
   }

   if (collision == true) {
      lives--;

      // poner heroe al estado inicial
      hero.setX(getWidth()/2);
      hero.setY(getHeight()-20);

      // Durante 20 ciclos nuestra nave será inmune
      shield=20;

      if (lives <= 0) {
         playing=false;
      }
   }
```

```
   if (shield > 0)
      shield--;

 }


 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);
   }
 }

 void quitGame() {
   playing = false;
 }


 boolean isPlaying() {
   return playing;
 }


 public void run() {
   start();

   while (playing) {

      // Actualizar fondo de pantalla
      doScroll();

      // Actualizar posición del jugador
      computePlayer();

      // Actualizar posición de los enemigos
      computeEnemies();

      // Actualizar balas
      computeBullets();

      // Actualizar explosiones
      computeExplodes();

      // Comprobar colisiones
      checkCollide();

      // Contador de ciclos
      cicle++;

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

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

   // Repintamos la pantalla
   // para mostrar pantalla de presentación
   repaint();
   serviceRepaints();
 }


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

   switch (action) {

      case FIRE:
         fireOn=false;
         break;
```

```
      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 FIRE:
         fireOn=true;
         break;
      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);

   g.setFont(Font.getFont(Font.FACE_PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_MEDIUM));

   if (playing == false) {
      intro.setX(getWidth()/2);
      intro.setY(getHeight()/2);
      intro.draw(g);
   } else {


      // 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 enemigos
      for (i=1 ; i<=6 ; i++) {
         if (enemies[i].isActive()) {
            enemies[i].setX(enemies[i].getX());
```

```
            enemies[i].setY(enemies[i].getY());
            enemies[i].draw(g);
         }
      }

      // Dibujar el jugador
      if (shield == 0 || shield%2 == 0) {
         hero.setX(hero.getX());
         hero.setY(hero.getY());
         hero.draw(g);
      }

      // Dibujar disparos
      for (i=1 ; i<=6 ; i++) {
         if (aBullet[i].isActive()) {
            aBullet[i].setX(aBullet[i].getX());
            aBullet[i].setY(aBullet[i].getY());
            aBullet[i].draw(g);
         }
      }

      // Dibujar explosiones
      for (i=1 ; i<=6 ; i++) {
         if (explode[i].isActive())
            explode[i].draw(g);
      }

      g.drawString(" "+score,getWidth()-20,20,Graphics.HCENTER|Graphics.BOTTOM);
      g.drawString(" "+lives,20,20,Graphics.HCENTER|Graphics.BOTTOM);
   }
 }
}
```

Sprite.java

```
package mygame;
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.HCENTER|Graphics.VCENTER);
   }
}
```

Enemy.java

```
package mygame;

import java.util.*;

class Enemy extends Sprite {

   private int type,state,deltaX,deltaY;

   public void setState(int state) {
      this.state=state;
   }

   public int getState(int state) {
      return state;
   }

   public void setType(int type) {
      this.type=type;
   }

   public int getType() {
      return type;
   }

   public void doMovement() {

      Random random = new java.util.Random();

      // Los enemigos de tipo 2 cambiaran su trayectoria
      // al alcanzar una posición determinada (pos. 50)
      if (type == 2 && getY() > 50 && state != 2) {

         // paso al estado 2 (movimiento diagonal)
         state = 2;

         if ((Math.abs(random.nextInt()) % 2) + 1 == 1) {
            deltaX=2;
         } else {
            deltaX=-2;
         }
      }

      // movemos la nave
      setX(getX()+deltaX);
      setY(getY()+deltaY);

   }

   public void init(int xhero) {
      deltaY=3;
      deltaX=0;

      if (type == 1) {
         if (xhero > getX()) {
            deltaX=2;
         } else {
            deltaX=-2;
         }
      }
   }


   // Sobrecarga del método draw de la clase Sprite
   public void draw (javax.microedition.lcdui.Graphics g) {
      selFrame(type);
      // llamamos al método 'draw' de la clase padre (Sprite)
      super.draw(g);
   }

   public Enemy(int nFrames) {
      super(nFrames);
   }
}
```

Bullet.java

```
package mygame;

class Bullet extends Sprite {
   private int owner;

   public Bullet(int nFrames) {
      super(nFrames);
   }

   public void setOwner(int owner) {
      this.owner=owner;
   }

   public int getOwner() {
      return owner;
   }

   public void doMovement() {

      // si owner = 1 el disparo es nuestro
      // si no, es del enemigo
      if (owner == 1) {
         setY(getY()-6);
      } else {
         setY(getY()+6);
      }
   }

   // Sobrecarga del método draw de la clase Sprite
   public void draw (javax.microedition.lcdui.Graphics g) {
      selFrame(owner);
      // llamamos al método 'draw' de la clase padre (Sprite)
      super.draw(g);
   }
}
```

Explode.java

```
package mygame;

class Explode extends Sprite {

   private int state;

   public Explode(int nFrames) {
      super(nFrames);
      state=1;
   }

   public void setState(int state) {
      this.state=state;
   }

   public int getState() {
      return state;
   }

   public void doMovement() {
      state++;

      if (state > super.frames())
         super.off();
   }

   // Sobrecarga del método draw de la clase Sprite
   public void draw (javax.microedition.lcdui.Graphics g) {
      selFrame(state);
      // llamamos al método 'draw' de la clase padre (Sprite)
      super.draw(g);
   }
}
```

Hero.java

```
package mygame;

class Hero extends Sprite {

   private int state;

   public void setState(int state) {
      this.state=state;
   }

   public int getState(int state) {
      return state;
   }

   public Hero(int nFrames) {
      super(nFrames);
   }
}
```

Recursos

Bibliografía

Java

Java 2. Manual de usuario y tutorial.
Agustín Froufe.
Ed. Ra-Ma
ISBN: 84-7897-389-3

Juegos en Java.
Joel Fan/Eric Ries/Calin Tenitchi
Ed. Anaya Multimedia
ISBN: 84-415-0410-5

J2ME

J2ME. Manual de usuario y tutorial.
Froufe, A/Jorge, P.
Ed. Ra-Ma
ISBN: 8478975977

Wireless Java with J2ME.
Michael Morrison.
Ed. Sams.
ISBN: 0-672-32142-4

Enlaces

J2ME

http://java.sun.com/j2me/ - *Página principal de Sun sobre J2ME*

http://developers.sun.com/techtopics/mobility/midp/reference/techart/index.html - *Artículos sobre J2ME*

http://developers.sun.com/techtopics/mobility/midp/samples/ - *Códigos J2ME de ejemplo*

http://www.developer.com/java/j2me/ - *Noticias relativas a J2ME*

http://www.midlet-review.com/ - *Página sobre juegos para móviles*

http://www.palowireless.com/java/tutorials.asp - *Tutoriales y artículos sobre J2ME*

Programación de videojuegos

http://www.flipcode.com/

http://www.gamedev.net/

http://www.gamasutra.com/

http://gpp.netfirms.com/

http://www.gametutorials.com/

http://www-cs-students.stanford.edu/~amitp/gameprog.html

Índice Alfabético

T

W