

# Introducción a Java con JavaME

## Índice

1 Java, JavaME y el entorno Eclipse.....	3
1.1 El lenguaje Java.....	3
1.2 Introducción a JavaME.....	14
1.3 Java para MIDs.....	60
2 Java, JavaME y el entorno Eclipse - Ejercicios.....	67
2.1 ¡Hola ME!.....	67
2.2 Clases.....	67
2.3 Métodos y campos de la clase.....	68
2.4 Métodos estáticos.....	69
2.5 Librerías opcionales (*).....	69
2.6 Temporizadores (*).....	70
3 Herencia e interfaces. MIDlets e interfaz de usuario.....	71
3.1 Herencia e interfaces.....	71
3.2 Colecciones de datos.....	74
3.3 MIDlets e interfaz de usuario.....	86
4 Herencia e interfaces. MIDlets e interfaz de usuario - Ejercicios.....	126
4.1 Menú básico.....	126
4.2 Adivina el número (I) (*).....	126
4.3 Adivina el número (II) (*).....	126
4.4 TeleSketch.....	127
5 Excepciones e hilos. Acceso a la red.....	129
5.1 Excepciones.....	129
5.2 Hilos.....	134
5.3 Conexiones de red.....	138
6 Excepciones e hilos. Acceso a la red - Ejercicios.....	153
6.1 Captura de excepciones (*).....	153

6.2 Lanzamiento de excepciones.....	153
6.3 Chat para el móvil.....	155
7 Flujos de E/S y serialización de objetos. RMS.....	157
7.1 Flujos de datos de entrada/salida.....	157
7.2 Entrada, salida y salida de error estándar.....	158
7.3 Acceso a ficheros.....	159
7.4 Acceso a los recursos.....	160
7.5 Codificación de datos.....	161
7.6 Serialización de objetos en Java.....	161
7.7 Serialización manual en JavaME.....	163
7.8 Almacenamiento de registros RMS en JavaME.....	163
8 Flujos de E/S y serialización de objetos. RMS - Ejercicios.....	174
8.1 Gestión de productos que implementan Serializable.....	174
8.2 Almacén de notas con serialización para JavaME.....	174

## 1. Java, JavaME y el entorno Eclipse

---

### 1.1. El lenguaje Java

---

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

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

#### 1.1.1. Conceptos previos de POO

---

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

##### 1.1.1.1. Concepto de clase y objeto

---

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

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

##### 1.1.1.2. Concepto de campo, método y constructor

---

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

que permite reservar memoria para almacenar los campos y métodos de la clase, a la hora de crear un objeto de la misma.

### 1.1.1.3. Concepto de herencia y polimorfismo

---

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

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

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

### 1.1.1.4. Modificadores de acceso

---

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

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

### 1.1.1.5. Clases abstractas e interfaces

---

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

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

- En una **clase**, al definir *Animal* tendríamos que implementar el código de los métodos `dibuja()` e `imprime()`. Las subclases que hereden de *Animal* no tendrían por qué implementar los métodos, a no ser que quieran redefinirlos para adaptarlos a sus propias necesidades.
- En una **clase abstracta** podríamos implementar los métodos que nos interese, dejando

sin implementar los demás (dejándolos como métodos abstractos). Dichos métodos tendrían que implementarse en las clases hijas.

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

### 1.1.2. Componentes de un programa Java

---

En un programa Java podemos distinguir varios elementos:

#### 1.1.2.1. Clases

---

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

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

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

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

#### 1.1.2.2. Campos y variables

---

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

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

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

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

Por otro lado, las constantes se declaran como `final static`, y sus nombres se escribirán totalmente en mayúsculas, separando las distintas palabras que los formen por

caracteres de subrayado ('\_'). Por ejemplo, ANCHO\_VENTANA, MSG\_ERROR\_FICHERO.

### 1.1.2.3. Métodos

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

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

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

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

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

#### Nota

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

### 1.1.2.4. Constructores

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

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

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

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

`finalize()` para liberar manualmente.

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

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

#### Nota

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

### 1.1.2.5. Paquetes

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

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

```
package paql.subpaql;
...
class MiClase {
    ...
}
```

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

```
import paql.subpaql.*;

import paql.subpaql.MiClase;
```

La primera opción (\*) se utiliza para importar todas las clases del paquete (se utiliza

cuando queremos utilizar muchas clases del paquete, para no ir importando una a una). La segunda opción se utiliza para importar una clase en concreto.

#### Nota

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

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

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

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

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

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

#### Importante

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

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

### 1.1.2.6. Tipo enumerado

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



## Ejemplo

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

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

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

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

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

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

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

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

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

### 1.1.2.7. Modificadores de acceso

Tanto las clases como los campos y métodos admiten modificadores de acceso, para indicar si dichos elementos tienen ámbito *público*, *protegido* o *privado*. Dichos

modificadores se marcan con las palabras reservadas `public`, `protected` y `private`, respectivamente, y se colocan al principio de la declaración:

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

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

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

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

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

```
public class MiClase
{
    ...
}

class OtraClase
{
    ...
}

class UnaClaseMas
{
    ...
}
```

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

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

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

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

### 1.1.2.8. Otros modificadores

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

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

Estos modificadores se colocan tras los modificadores de acceso:

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

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

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

#### Nota

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

Por ejemplo, si tenemos:

```
public class UnaClase
{
    public int a;

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

Dará error, porque el campo `a` no es estático, y lo estamos utilizando dentro del método estático. Para solucionarlo tenemos dos posibilidades: definir `a` como estático (si el diseño del programa lo permite), o bien crear un objeto de tipo `UnaClase` en el método, y utilizar su campo `a` (que ya no hará falta que sea estático, porque hemos creado un objeto y ya podemos acceder a su campo `a`):

```
public class UnaClase
{
    public int a;

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

#### Nota

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

Por ejemplo, si hacemos lo siguiente:

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

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

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

### 1.1.2.9. Imports estáticos

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

#### Ejemplo

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

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

### 1.1.2.10. Argumentos variables

Java permite pasar un número variable de argumentos a una función (como sucede con funciones como `printf` en C). Esto se consigue mediante la expresión `"..."` a partir del momento en que queramos tener un número variable de argumentos.

#### Ejemplo

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

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

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

### 1.1.2.11. Metainformación o anotaciones

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

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

### 1.1.2.12. Ejecución de clases: método main

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

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

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

estáticos, o bien campos y métodos de objetos que creemos dentro del `main`.

## 1.2. Introducción a JavaME

---

### 1.2.1. Aplicaciones JavaME

---

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

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

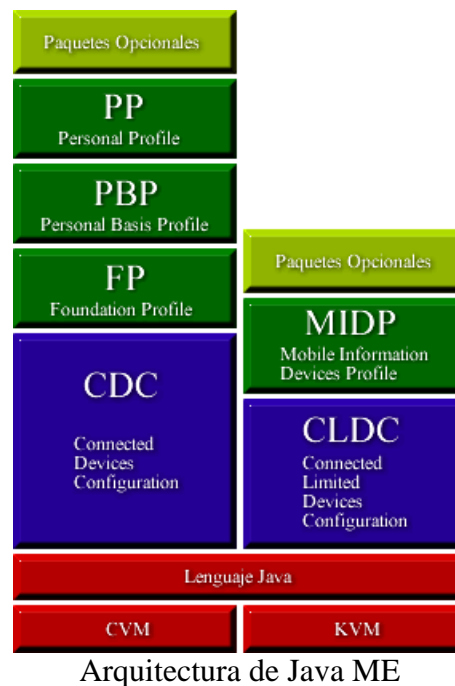
JavaME es el nombre corto de Java Micro Edition y también es ampliamente conocido con su anterior nombre comercial, J2ME que significa Java 2 Micro Edition. A partir de ahora nos referiremos a la plataforma por ambos nombres indistintamente

#### 1.2.1.1. Arquitectura de JavaME

---

Hemos visto que existen dispositivos de tipos muy distintos, cada uno de ellos con sus propias necesidades, y muchos con grandes limitaciones de capacidad. Si obtenemos el máximo común denominador de todos ellos nos quedamos prácticamente con nada, por lo que es imposible definir una única API en J2ME que nos sirva para todos.

Por ello en J2ME existirán diferentes APIs cada una de ellas diseñada para una familia de dispositivos distinta. Estas APIs se encuentran arquitecturadas en dos capas: configuraciones y perfiles.



## Configuraciones

Las configuraciones son las capas de la API de bajo nivel, que residen sobre la máquina virtual y que están altamente vinculadas con ella, ofrecen las características básicas de todo un gran conjunto de dispositivos. En esta API ofrecen lo que sería el máximo denominador común de todos ellos, la API de programación básica en lenguaje Java.

Encontramos distintas configuraciones para adaptarse a la capacidad de los dispositivos. La configuración CDC (*Connected Devices Configuration*) contiene la API común para todos los dispositivos conectados, soportada por la máquina virtual Java.

Sin embargo, para algunos dispositivos con grandes limitaciones de capacidad esta máquina virtual Java puede resultar demasiado compleja. Hemos de pensar en dispositivos que pueden tener 128 KB de memoria. Es evidente que la máquina virtual de Java (JVM) pensada para ordenadores con varias megas de RAM instaladas no podrá funcionar en estos dispositivos.

Por lo tanto aparece una segunda configuración llamada CLDL (*Connected Limited Devices Configuration*) pensada para estos dispositivos con grandes limitaciones. En ella se ofrece una API muy reducida, en la que tenemos un menor número de funcionalidades, adaptándose a las posibilidades de estos dispositivos. Esta configuración está soportada por una máquina virtual mucho más reducida, la KVM (*Kilobyte Virtual Machine*), que necesitará muchos menos recursos por lo que podrá instalarse en dispositivos muy limitados.

Vemos que tenemos distintas configuraciones para adaptarnos a dispositivos con distinta

capacidad. La configuración CDC soportada por la JVM (*Java Virtual Machine*) funcionará sólo con dispositivos con memoria superior a 1 MB, mientras que para los dispositivos con memoria del orden de los KB tenemos la configuración CLDC soportada por la KVM, de ahí viene el nombre de *Kilobyte Virtual Machine*.

## Perfiles

Como ya hemos dicho, las configuraciones nos ofrecen sólo la parte básica de la API para programar en los dispositivos, aquella parte que será común para todos ellos. El problema es que esta parte común será muy reducida, y no nos permitirá acceder a todas las características de cada tipo de dispositivo específico. Por lo tanto, deberemos extender la API de programación para cada familia concreta de dispositivos, de forma que podamos acceder a las características propias de cada familia.

Esta extensión de las configuraciones es lo que se denomina perfiles. Los perfiles son una capa por encima de las configuraciones que extienden la API definida en la configuración subyacente añadiendo las operaciones adecuadas para programar para una determinada familia de dispositivos.

Por ejemplo, tenemos un perfil MIDP (*Mobile Information Devices Profile*) para programar los dispositivos móviles de información. Este perfil MIDP reside sobre CLDC, ya que estos son dispositivos bastante limitados a la mayoría de las ocasiones.

## Paquetes opcionales

Además podemos incluir paquetes adicionales, como una tercera capa por encima de las anteriores, para dar soporte a funciones concretas de determinados modelos de dispositivos. Por ejemplo, los móviles que incorporen cámara podrán utilizar una API multimedia para acceder a ella.

### 1.2.1.2. Configuración CDC

La configuración CDC se utilizará para dispositivos conectados con una memoria de por lo menos 1 MB (se recomiendan al menos 2 MB para un funcionamiento correcto). Se utilizará en dispositivos como PDAs de gama alta, comunicadores, decodificadores de televisión, impresoras de red, *routers*, etc.

CDC se ha diseñado de forma que se mantenga la máxima compatibilidad posible con J2SE, permitiendo de este modo portar fácilmente las aplicaciones con las que ya contamos en nuestros ordenadores a CDC.

La máquina virtual utilizada, la CVM, cumple con la misma especificación que la JVM, por lo que podremos programar las aplicaciones de la misma forma que lo hacíamos en J2SE. Existe una nueva máquina virtual para soportar CDC, llamada CDC *Hotspot*, que incluye diversas optimizaciones utilizando la tecnología *Hotspot*.

La API que ofrece CDC es un subconjunto de la API que ofrecía J2SE, optimizada para



este tipo de dispositivos que tienen menos recursos que los PCs en los que utilizamos J2SE. Se mantienen las clases principales de la API, que ofrecerán la misma interfaz que en su versión de J2SE.

CDC viene a sustituir a la antigua API *PersonalJava*, que se aplicaba al mismo tipo de dispositivos. La API CDC, a diferencia de *PersonalJava*, está integrada dentro de la arquitectura de J2ME.

Tenemos diferentes perfiles disponibles según el tipo de dispositivo que estemos programando: *Foundation Profile* (FP), *Personal Basis Profile* (PBP) y *Personal Profile* (PP).

### Foundation Profile

Este es el perfil básico para la programación con CDC. No incluye ninguna API para la creación de una interfaz gráfica de usuario, por lo que se utilizará para dispositivos sin interfaz, como por ejemplo impresoras de red o *routers*.

Los paquetes que se incluyen en este perfil son:

```
java.io
java.lang
java.lang.ref
java.lang.reflect
java.net
java.security
java.security.acl
java.security.cert
java.security.interfaces
java.security.spec
java.text
java.util
java.util.jar
java.util.zip
```

Vemos que incluye todos los paquetes del núcleo de Java, para la programación básica en el lenguaje (`java.lang`), para manejar la entrada/salida (`java.io`), para establecer conexiones de red (`java.net`), para seguridad (`java.security`), manejo de texto (`java.text`) y clases útiles (`java.util`). Estos paquetes se incluyen en su versión íntegra, igual que en J2SE. Además incluye un paquete adicional que no pertenece a J2SE:

```
javax.microedition.io
```

Este paquete pertenece está presente para mantener la compatibilidad con CLDC, ya que pertenece a esta configuración, como veremos más adelante.

### Personal Basis Profile

Este perfil incluye una API para la programación de la interfaz gráfica de las aplicaciones. Lo utilizaremos para dispositivos en los que necesitemos aplicaciones con interfaz gráfica. Este es el caso de los descodificadores de televisión por ejemplo.

Además de los paquetes incluidos en FP, añade los siguientes:

```
java.awt
java.awt.color
java.awt.event
java.awt.image
java.beans
java.rmi
java.rmi.registry
```

Estos paquetes incluyen un subconjunto de las clases que contenían en J2SE. Tenemos el paquete AWT (`java.awt`) para la creación de la interfaz gráfica de las aplicaciones. Este paquete sólo incluye soporte para componentes ligeros (aquellos que definen mediante código Java la forma de dibujarse), y no incluye ningún componente de alto nivel (como botones, campos de texto, etc). Tendremos que crear nosotros nuestros propios componentes, definiendo la forma en la que se dibujará cada uno.

También incluye un soporte limitado para *beans* (`java.beans`) y objetos distribuidos RMI (`java.rmi`).

Además podemos encontrar un nuevo tipo de componente no existente en J2SE, que son los **Xlets**.

```
javax.microedition.xlet
javax.microedition.xlet.ixc
```

Estos *xlets* son similares a los *applets*, son componentes que se ejecutan dentro de un contenedor que controla su ciclo de vida. En el caso de los *applets* este contenedor era normalmente el navegador donde se cargaba el *applet*. Los *xlets* se ejecutan dentro del *xlet manager*. Los *xlets* pueden comunicarse entre ellos mediante RMI. De hecho, la parte de RMI incluida en PBP es únicamente la dedicada a la comunicación entre *xlets*.

Los *xlets* se diferencian también de los *applets* en que tienen un ciclo de vida definido más claramente, y que no están tan vinculados a la interfaz (AWT) como los *applets*. Por lo tanto podremos utilizar tanto *xlets* con interfaz gráfica, como sin ella.

Estos *xlets* se suelen utilizar en aplicaciones de televisión interactiva, instaladas en los descodificadores (*set top boxes*).

## Personal Profile

Este perfil incluye soporte para *applets* e incluye la API de AWT íntegra. De esta forma podremos utilizar los componentes pesados de alto nivel definidos en AWT (botones, menús, campos de texto, etc). Estos componentes pesado utilizan la interfaz gráfica nativa del dispositivo donde se ejecutan. De esta forma, utilizaremos este perfil cuando trabajemos con dispositivos que disponen de su propia interfaz gráfica de usuario (GUI) nativa.

Incluye los siguientes paquetes:

```
java.applet
java.awt
```

```
java.awt.datatransfer
```

En este caso ya se incluye íntegra la API de AWT (`java.awt`) y el soporte para applets (`java.applet`). Este paquete es el más parecido al desaparecido *PersonalJava*, por lo que será el más adecuado para migrar las aplicaciones *PersonalJava* a J2ME.

### Paquetes opcionales

En CDC se incluyen como paquetes opcionales subconjuntos de otras APIs presentes en J2SE: la API **JDBC** para conexión a bases de datos, y la API de **RMI** para utilizar esta tecnología de objetos distribuidos.

Además también podremos utilizar como paquete opcional la API **Java TV**, adecuada para aplicaciones de televisión interactiva (iTV), que pueden ser instaladas en descodificadores de televisión digital. Incluye la extensión JMF (*Java Media Framework*) para controlar los flujos de video.

Podremos utilizar estas APIs para programar todos aquellos dispositivos que las soporten.

#### 1.2.1.3. Configuración CLDC

La configuración CLDC se utilizará para dispositivos conectados con poca memoria, pudiendo funcionar correctamente con sólo 128 KB de memoria. Normalmente la utilizaremos para los dispositivos con menos de 1 ó 2 MB de memoria, en los que CDC no funcionará correctamente. Esta configuración se utilizará para teléfonos móviles (celulares) y PDAs de gama baja.

Esta configuración se ejecutará sobre la KVM, una máquina virtual con una serie de limitaciones para ser capaz de funcionar en estas configuraciones de baja memoria. Por ejemplo, no tiene soporte para tipos de datos `float` y `double`, ya que estos dispositivos normalmente no tienen unidad de punto flotante.

La API que ofrece esta configuración consiste en un subconjunto de los paquetes principales del núcleo de Java, adaptados para funcionar en este tipo de dispositivos. Los paquetes que ofrece son los siguientes:

```
java.lang  
java.io  
java.util
```

Tenemos las clases básicas del lenguaje (`java.lang`), algunas clases útiles (`java.util`), y soporte para flujos de entrada/salida (`java.io`). Sin embargo vemos que no se ha incluido la API de red (`java.net`). Esto es debido a que esta API es demasiado compleja para estos dispositivos, por lo que se sustituirá por una API de red propia más reducida, adaptada a sus necesidades de conectividad:

```
javax.microedition.io
```

En la actualidad encontramos únicamente un perfil que se ejecuta sobre CLDC. Este

perfil es MIDP (*Mobile Information Devices Profile*), y corresponde a la familia de dispositivos móviles de información (teléfonos móviles y PDAs).

Los dispositivos iMode utilizan una API de Java propietaria de NTT DoCoMo, llamada DoJa. Esta API se construye sobre CLDC, pero no es un perfil perteneciente a J2ME. Simplemente es una extensión de CLDC para teléfonos iMode.

### Mobile Information Devices Profile

Utilizaremos este perfil para programar aplicaciones para MIDs. En los siguientes capítulos nos centraremos en la programación de aplicaciones para móviles utilizando este perfil, que es el que más protagonismo ha cobrado tras la aparición de los últimos modelos de móviles que incluyen soporte para esta API.

La API que nos ofrece MIDP consiste, además de los paquetes ofrecidos en CLDC, en los siguientes paquetes:

```
javax.microedition.lcdui
javax.microedition.lcdui.game
javax.microedition.media
javax.microedition.media.control
javax.microedition.midlet
javax.microedition.pki
javax.microedition.rms
```

Las aplicaciones que desarrollaremos para estos dispositivos se llaman **MIDlets**. El móvil actuará como contenedor de este tipo de aplicaciones, controlando su ciclo de vida. Tenemos un paquete con las clases correspondientes a este tipo de componentes (`javax.microedition.midlet`). Además tendremos otro paquete con los elementos necesarios para crear la interfaz gráfica en la pantalla de los móviles (`javax.microedition.lcdui`), que además nos ofrece facilidades para la programación de juegos. Tenemos también un paquete con clases para reproducción de músicas y tonos (`javax.microedition.media`), para creación de certificados por clave pública para controlar la seguridad de las aplicaciones (`javax.microedition.pki`), y para almacenamiento persistente de información (`javax.microedition.rms`).

### Paquetes opcionales

Como paquetes opcionales tenemos:

- **Wireless Messaging API (WMA) (JSR-120)**: Nos permitirá trabajar con mensajes en el móvil. De esta forma podremos por ejemplo enviar o recibir mensajes de texto SMS.
- **Mobile Media API (MMAPI) (JSR-135)**: Proporciona controles para la reproducción y captura de audio y video. Permite reproducir ficheros de audio y video, generar y secuenciar tonos, trabajar con *streams* de estos medios, e incluso capturar audio y video si nuestro móvil está equipado con una cámara.
- **J2ME Web Services API (WSA) (JSR-172)**: Nos permitirá invocar Servicios Web desde nuestro cliente móvil. Muchos fabricantes de servidores de aplicaciones J2EE,

con soporte para desplegar Servicios Web, nos ofrecen sus propias APIs para invocar estos servicios desde los móviles J2ME, como es el caso de Weblogic por ejemplo.

- **Bluetooth API (JSR-82)**: Con esta API podremos establecer comunicaciones con otros dispositivos de forma inalámbrica y local vía *bluetooth*.
- **Security and Trust Services API for J2ME (JSR-177)**: Ofrece servicios de seguridad para proteger los datos privados que tenga almacenados el usuario, encriptar la información que circula por la red, y otros servicios como identificación y autenticación.
- **Location API for J2ME (JSR-179)**: Proporciona información acerca de la localización física del dispositivo (por ejemplo mediante GPS o E-OTD).
- **SIP API for J2ME (JSR-180)**: Permite utilizar SIP (*Session Initiation Protocol*) desde aplicaciones MIDP. Este protocolo se utiliza para establecer y gestionar conexiones IP multimedia. Este protocolo puede usarse en aplicaciones como juegos, videoconferencias y servicios de mensajería instantánea.
- **Mobile 3D Graphics (M3G) (JSR-184)**: Permite mostrar gráficos 3D en el móvil. Se podrá utilizar tanto para realizar juegos 3D como para representar datos.
- **PDA Optional Packages for J2ME (JSR-75)**: Contiene dos paquetes independientes para acceder a funcionalidades características de muchas PDAs y algunos teléfonos móviles. Estos paquetes son PIM (*Personal Information Management*) y FC (*FileConnection*). PIM nos permitirá acceder a información personal que tengamos almacenada de forma nativa en nuestro dispositivo, como por ejemplo nuestra libreta de direcciones. FC nos permitirá abrir ficheros del sistema de ficheros nativo de nuestro dispositivo.

#### 1.2.1.4. JWTI

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

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

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

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

dispositivos.

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

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

#### 1.2.1.5. MSA

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

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

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

#### 1.2.2. Aplicaciones J2ME

Vamos a ver cómo construir aplicaciones J2ME a partir del código fuente de forma que estén listas para ser instaladas directamente en cualquier dispositivo con soporte para esta tecnología.

Estudiaremos la creación de aplicaciones para MIDs, que serán normalmente teléfonos móviles o algunos PDAs. Por lo tanto nos centraremos en el perfil MIDP.

Para comenzar vamos a ver de qué se componen las aplicaciones MIDP que podemos instalar en los móviles (ficheros JAD y JAR), y cómo se realiza este proceso de instalación. A continuación veremos como crear paso a paso estos ficheros de los que se componen estas aplicaciones, y como probarlas en emuladores para no tener que

transferirlas a un dispositivo real cada vez que queramos hacer una prueba. En el siguiente punto se verá cómo podremos facilitar esta tarea utilizando los kits de desarrollo que hay disponibles, y algunos entornos integrados (IDEs) para hacer más cómodo todavía el desarrollo de aplicaciones para móviles.

Para distribuir e instalar las aplicaciones J2ME en los dispositivos utilizaremos ficheros de tipos JAR y JAD. Las aplicaciones estarán compuestas por un fichero JAR y un fichero JAD.

#### 1.2.2.1. Suite de MIDlets

---

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

El fichero JAR será el que contendrá las aplicaciones de la *suite*. En él tendremos tanto el código compilado como los recursos que necesite para ejecutarse (imágenes, sonidos, etc). Estos ficheros JAR son un estándar de la plataforma Java, disponibles en todas las ediciones de esta plataforma, que nos permitirán empaquetar una aplicación Java en un solo fichero. Al ser un estándar de la plataforma Java será portable a cualquier sistema donde contemos con esta plataforma.

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

#### 1.2.2.2. Instalación de aplicaciones

---

De esta forma cuando queramos instalar una aplicación deberemos localizar su fichero JAD. Una vez localizado el fichero JAD, deberemos indicar que deseamos instalar la aplicación, de forma que se descargue e instale en nuestro dispositivo el fichero JAR correspondiente. Además el fichero JAD localizado nos permitirá saber si una aplicación ya está instalada en nuestro dispositivo, y de ser así comprobar si hay disponible una versión superior y dar la opción al usuario de actualizarla. De esta forma no será necesario descargar el fichero JAR entero, cuyo tamaño será mayor debido a que contiene toda la aplicación, para conocer los datos de la aplicación y si la tenemos ya instalada en nuestro móvil.

Un posible escenario de uso es el siguiente. Podemos navegar con nuestro móvil mediante WAP por una página WML. En esa página puede haber publicadas una serie de aplicaciones Java para descargar. En la página tendremos los enlaces a los ficheros JAD de cada aplicación disponible. Seleccionando con nuestro móvil uno de estos enlaces, accederá al fichero JAD y nos dará una descripción de la aplicación que estamos

solicitando, preguntándonos si deseamos instalarla. Si decimos que si, descargará el fichero JAR asociado en nuestro móvil e instalará la aplicación de forma que podemos usarla. Si accedemos posteriormente a la página WML y pinchamos sobre el enlace al JAD de la aplicación, lo comparará con las aplicaciones que tenemos instaladas y nos dirá que la aplicación ya está instalada en nuestro móvil. Además, al incluir la información sobre la versión podrá saber si la versión que hay actualmente en la página es más nueva que la que tenemos instalada, y en ese caso nos dará la opción de actualizarla.

### 1.2.2.3. Software gestor de aplicaciones

Los dispositivos móviles contienen lo que se denomina AMS (*Application Management Software*), o software gestor de aplicaciones en castellano. Este software será el encargado de realizar el proceso de instalación de aplicaciones que hemos visto en el punto anterior. Será el que controle el ciclo de vida de las *suites*:

- Obtendrá información de las *suites* a partir de un fichero JAD mostrándosela al usuario y permitiendo que éste instale la aplicación.
- Comprobará si la aplicación está ya instalada en el móvil, y en ese caso comparará las versiones para ver si la versión disponible es más reciente que la instalada y por lo tanto puede ser actualizada.
- Instalará o actualizará las aplicaciones cuando se requiera, de forma que el usuario tenga la aplicación disponible en el móvil para ser utilizada.
- Ejecutará los MIDlets instalados, controlando el ciclo de vida de estos MIDlets como veremos en el capítulo 4.
- Permitirá desinstalar las aplicaciones, liberando así el espacio que ocupan en el móvil.

### 1.2.2.4. Fichero JAD

Los ficheros JAD son ficheros ASCII que contienen una descripción de la *suite*. En él se le dará valor a una serie de propiedades (parámetros de configuración) de la *suite*. Tenemos una serie de propiedades que deberemos especificar de forma obligatoria en el fichero:

MIDlet-Name	Nombre de la suite.
MIDlet-Version	Versión de la suite. La versión se compone de 3 números separados por puntos: <mayor>.<menor>.<micro>, como por ejemplo 1.0.0
MIDlet-Vendor	Autor (Proveedor) de la <i>suite</i> .
MIDlet-Jar-URL	Dirección (URL) de donde obtener el fichero JAR con la <i>suite</i> .
MIDlet-Jar-Size	Tamaño del fichero JAR en <i>bytes</i> .

Además podemos incluir una serie de propiedades adicionales de forma optativa:



MIDlet-Icon	Icono para la <i>suite</i> . Si especificamos un icono éste se mostrará junto al nombre de la <i>suite</i> , por lo que nos servirá para identificarla. Este icono será un fichero con formato PNG que deberá estar contenido en el fichero JAR.
MIDlet-Description	Descripción de la <i>suite</i> .
MIDlet-Info-URL	Dirección URL donde podemos encontrar información sobre la <i>suite</i> .
MIDlet-Data-Size	Número mínimo de <i>bytes</i> que necesita la <i>suite</i> para almacenar datos de forma persistente. Por defecto este número mínimo se considera 0.
MIDlet-Delete-Confirm	Mensaje de texto con el que se le preguntará al usuario si desea desinstalar la aplicación.
MIDlet-Delete-Notify	URL a la que se enviará una notificación de que el usuario ha desinstalado nuestra aplicación de su móvil.
MIDlet-Install-Notify	URL a la que se enviará una notificación de que el usuario ha instalado nuestra aplicación en su móvil.

Estas son propiedades que reconocerá el AMS y de las que obtendrá la información necesaria sobre la *suite*. Sin embargo, como desarrolladores puede interesarnos incluir una serie de parámetros de configuración propios de nuestra aplicación. Podremos hacer eso simplemente añadiendo nuevas propiedades con nombres distintos a los anteriores al fichero JAR. En el capítulo 4 veremos como acceder a estas propiedades desde nuestras aplicaciones.

Un ejemplo de fichero JAD para una *suite* de MIDlets es el siguiente:

```
MIDlet-Name: SuiteEjemplos
MIDlet-Version: 1.0.0
MIDlet-Vendor: Universidad de Alicante
MIDlet-Description: Aplicaciones de ejemplo para moviles.
MIDlet-Jar-Size: 16342
MIDlet-Jar-URL: ejemplos.jar
```

### 1.2.2.5. Fichero JAR

En el fichero JAR empaquetaremos los ficheros `.class` resultado de compilar las clases que componen nuestra aplicación, así como todos los recursos que necesite la aplicación, como pueden ser imágenes, sonidos, músicas, videos, ficheros de datos, etc.

Para empaquetar estos ficheros en un fichero JAR, podemos utilizar la herramienta `jar` incluida en J2SE. Más adelante veremos como hacer esto.

Además de estos contenidos, dentro del JAR tendremos un fichero `MANIFEST.MF` que

contendrá una serie de parámetros de configuración de la aplicación. Se repiten algunos de los parámetros especificados en el fichero JAD, y se introducen algunos nuevos. Los parámetros requeridos son:

MIDlet-Name	Nombre de la <i>suite</i> .
MIDlet-Version	Versión de la <i>suite</i> .
MIDlet-Vendor	Autor (Proveedor) de la <i>suite</i> .
MicroEdition-Profile	Perfil requerido para ejecutar la <i>suite</i> . Podrá tomar el valor MIDP-1.0 ó MIDP-2.0, según las versión de MIDP que utilicen las aplicaciones incluidas.
MicroEdition-Configuration	Configuración requerida para ejecutar la <i>suite</i> . Tomará el valor CLDC-1.0 para las aplicaciones que utilicen esta configuración.

Deberemos incluir también información referente a cada MIDlet contenido en la *suite*. Esto lo haremos con la siguiente propiedad:

MIDlet-<n>	Nombre, icono y clase principal del MIDlet número n
------------	---

Los MIDlets se empezarán a numerar a partir del número 1, y deberemos incluir una línea de este tipo para cada MIDlet disponible en la *suite*. Daremos a cada MIDlet un nombre para que lo identifique el usuario, un icono de forma optativa, y el nombre de la clase principal que contiene dicho MIDlet.

Si especificamos un icono, deberá ser un fichero con formato PNG contenido dentro del JAR de la *suite*. Por ejemplo, para una *suite* con 3 MIDlets podemos tener la siguiente información:

```
MIDlet-Name: SuiteEjemplos
MIDlet-Version: 1.0.0
MIDlet-Vendor: Universidad de Alicante
MIDlet-Description: Aplicaciones de ejemplo para moviles.
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
MIDlet-1: Snake, /icons/snake.png, es.ua.jtech.serpiente.SerpMIDlet
MIDlet-2: TeleSketch, /icons/ts.png, es.ua.jtech.ts.TeleSketchMIDlet
MIDlet-3: Panj, /icons/panj.png, es.ua.jtech.panj.PanjMIDlet
```

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

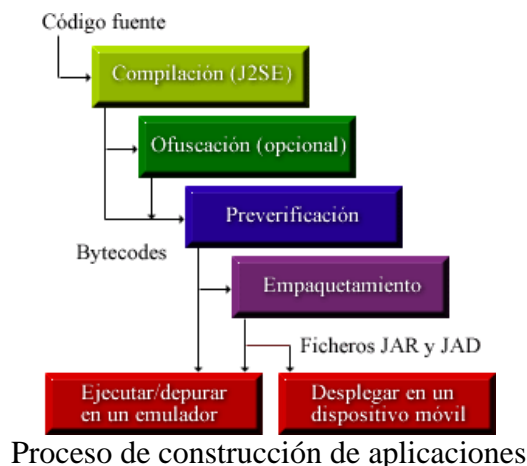
MIDlet-Icon	Icono para la <i>suite</i> .
MIDlet-Description	Descripción de la <i>suite</i> .
MIDlet-Info-URL	Dirección URL con información
MIDlet-Data-Size	Número mínimo de <i>bytes</i> para datos persistentes.

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

### 1.2.3. Construcción de aplicaciones

Vamos a ver los pasos necesarios para construir una aplicación con J2ME a partir del código fuente, obteniendo finalmente los ficheros JAD y JAR con los que podremos instalar la aplicación en dispositivos móviles.

El primer paso será compilar las clases, obteniendo así el código intermedio que podrá ser ejecutado en una máquina virtual de Java. El problema es que este código intermedio es demasiado complejo para la KVM, por lo que deberemos realizar una preverificación del código, que simplifique el código intermedio de las clases y compruebe que no utiliza ninguna característica no soportada por la KVM. Una vez preverificado, deberemos empaquetar todos los ficheros de nuestra aplicación en un fichero JAR, y crear el fichero JAD correspondiente. En este momento podremos probar la aplicación en un emulador o en un dispositivo real. Los emuladores nos permitirán probar las aplicaciones directamente en nuestro ordenador sin tener que transferirlas a un dispositivo móvil real.



Necesitaremos tener instalado J2SE, ya que utilizaremos las mismas herramientas para compilar y empaquetar las clases. Además necesitaremos herramientas adicionales, ya que la máquina virtual reducida de los dispositivos CLDC necesita un código intermedio simplificado.

#### 1.2.3.1. Compilación

Lo primero que deberemos hacer es compilar las clases de nuestra aplicación. Para ello utilizaremos el compilador incluido en J2SE, `javac`, por lo que deberemos tener instalada esta edición de Java.

Al compilar, el compilador buscará las clases que utilizamos dentro de nuestros programas para comprobar que estamos utilizándolas correctamente, y si utilizamos una clase que no existe, o bien llamamos a un método o accedemos a una propiedad que no pertenece a dicha clase nos dará un error de compilación. Java busca las clases en el siguiente orden:

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

Si estamos compilando con el compilador de J2SE, por defecto considerará que las clases del núcleo de Java son las clases de la API de J2SE. Debemos evitar que esto ocurra, ya que estas clases no van a estar disponibles en los dispositivos MIDP que cuentan con una API reducida. Debemos hacer que tome como clases del núcleo las clases de la API de MIDP, esto lo haremos mediante el parámetro `bootclasspath` del compilador:

```
javac -bootclasspath ${ruta_midp}/midpapi.zip <ficheros .java>
```

Con esto estaremos compilando nuestras clases utilizando como API del núcleo de Java la API de MIDP. De esta forma, si dentro de nuestro programa utilizásemos una clase que no pertenece a MIDP, aunque pertenezca a J2SE nos dará un error de compilación.

### 1.2.3.2. Ofuscación

Este es un paso opcional, pero recomendable. El código intermedio de Java incluye información sobre los nombres de los constructores, de los métodos y de los atributos de las clases e interfaces para poder acceder a esta información utilizando la API de *reflection* en tiempo de ejecución.

El contar con esta información nos permite descompilar fácilmente las aplicaciones, obteniendo a partir del código compilado unos fuentes muy parecidos a los originales. Lo único que se pierde son los comentarios y los nombres de las variables locales y de los parámetros de los métodos.

Esto será un problema si no queremos que se tenga acceso al código fuente de nuestra aplicación. Además incluir esta información en los ficheros compilados de nuestra aplicación harán que crezca el tamaño de estos ficheros ocupando más espacio, un espacio muy preciado en el caso de los dispositivos móviles con baja capacidad. Hemos de recordar que el tamaño de los ficheros JAR que soportan está limitado en muchos casos a 64kb o menos.

El proceso de ofuscación del código consiste en simplificar esta información, asignándoles nombres tan cortos como se pueda a las clases e interfaces y a sus constructores, métodos y atributos. De esta forma al descompilar obtendremos un código nada legible con nombres sin ninguna significancia.

Además conseguiremos que los ficheros ocupen menos espacio en disco, lo cuál será muy

conveniente para las aplicaciones para dispositivos móviles con baja capacidad y reducida velocidad de descarga.

La ofuscación de código deberemos hacerla antes de la preverificación, dejando la preverificación para el final, y asegurándonos así de que el código final de nuestra aplicación funcionará correctamente en la KVM. Podemos utilizar para ello diferentes ofuscadores, como ProGuard, RetroGuard o JODE. Deberemos obtener alguno de estos ofuscadores por separado, ya que no se incluyen en J2SE ni en los kits de desarrollo para MIDP que veremos más adelante.

### 1.2.3.3. Preverificación

---

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

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

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

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

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

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

### 1.2.3.4. Creación de la suite

---

Una vez tenemos el código compilado preverificado, deberemos empaquetarlo todo en un

fichero JAR para crear la *suite* con nuestra aplicación. En este fichero JAR deberemos empaquetar todos los ficheros `.class` generados, así como todos los recursos que nuestra aplicación necesite para funcionar, como pueden ser iconos, imágenes, sonidos, ficheros de datos, videos, etc.

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

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

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

#### 1.2.3.5. Prueba en emuladores

---

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

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

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

Podemos encontrar emuladores proporcionados por distintos fabricantes, como Nokia, Siemens o Sun entre otros. De esta forma tendremos emuladores que imitan distintos

modelos de teléfonos Nokia o Siemens existentes. Sun proporciona una serie de emuladores genéricos que podremos personalizar dentro de su kit de desarrollo que veremos en el próximo apartado.

#### 1.2.3.6. Prueba de la aplicación en dispositivos reales

---

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

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

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

#### 1.2.3.7. Despliegue

---

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

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

<b>.jad</b>	text/vnd.sun.j2me.app-descriptor
<b>.jar</b>	application/java-archive

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

#### 1.2.4. Kits de desarrollo

---

Para simplificar la tarea de desarrollar aplicaciones MIDP, tenemos disponibles distintos kits de desarrollo proporcionados por distintos fabricantes, como Sun o Nokia. Antes de instalar estos kits de desarrollo deberemos tener instalado J2SE. Estos kits de desarrollo



contienen todos los elementos necesarios para, junto a J2SE, crear aplicaciones MIDP:

- **API de MIDP.** Librería de clases que componen la API de MIDP necesaria para poder compilar las aplicaciones que utilicen esta API.
- **Preverificador.** Herramienta necesaria para realizar la fase de preverificación del código.
- **Emuladores.** Nos servirán para probar la aplicación en nuestro propio PC, sin necesidad de llevarla a un dispositivo real.
- **Entorno para la creación de aplicaciones.** Estos kits normalmente proporcionarán una herramienta que nos permita automatizar el proceso de construcción de aplicaciones MIDP que hemos visto en el punto anterior.
- **Herramientas adicionales.** Podemos encontrar herramientas adicionales, de configuración, personalización de los emuladores, despliegue de aplicaciones, conversores de formatos de ficheros al formato reconocido por MIDP, etc.

Vamos a centrarnos en estudiar cómo trabajar con el kit de desarrollo de Sun, ya que es el más utilizado por ser genérico y el que mejor se integra con otros entornos y herramientas. Este kit recibe el nombre de *Wireless Toolkit* (WTK). Existen diferentes versiones de WTK, cada una de ellas adecuada para un determinado tipo de aplicaciones:

- **WTK 1.0.4:** Soporta MIDP 1.0 y CLDC 1.0. Será adecuado para desarrollar aplicaciones para móviles que soporten sólo esta versión de MIDP, aunque también funcionarán con modelos que soporten versiones posteriores de MIDP, aunque en estos casos no estaremos aprovechando al máximo las posibilidades del dispositivo.
- **WTK 2.0:** Soporta MIDP 2.0, CLDC 1.0 y las APIs opcionales WMA y MMAPI. Será adecuado para realizar aplicaciones para móviles MIDP 2.0, pero no para aquellos que sólo soporten MIDP 1.0, ya que las aplicaciones que hagamos con este kit pueden utilizar elementos que no estén soportados por MIDP 1.0 y por lo tanto es posible que no funcionen cuando las despleguemos en este tipo de dispositivos. Además en esta versión se incluyen mejoras como la posibilidad de probar las aplicaciones vía OTA.
- **WTK 2.1:** Soporta MIDP 2.0, MIDP 1.0, CLDC 1.1 (con soporte para punto flotante), CLDC 1.0, y las APIs opcionales WMA, MMAPI y WSA. En este caso podemos configurar cuál es la plataforma para la que desarrollamos cada aplicación. Por lo tanto, esta versión será adecuada para desarrollar para cualquier tipo de móviles. Puede generar aplicaciones totalmente compatibles con JTWI.
- **WTK 2.2:** Aparte de todo lo soportado por WTK 2.1, incorpora las APIs para gráficos 3D (JSR-184) y bluetooth (JSR-82).
- **WTK 2.5:** Aparte de todo lo soportado por WTK 2.2, incorpora todas las APIs definidas en MSA. Nos centraremos en el estudio de esta versión por ser la que incorpora un mayor número de APIs en el momento de la escritura de este texto, además de ser genérica (se puede utilizar para cualquier versión de MIDP).

#### 1.2.4.1. Creación de aplicaciones con WTK

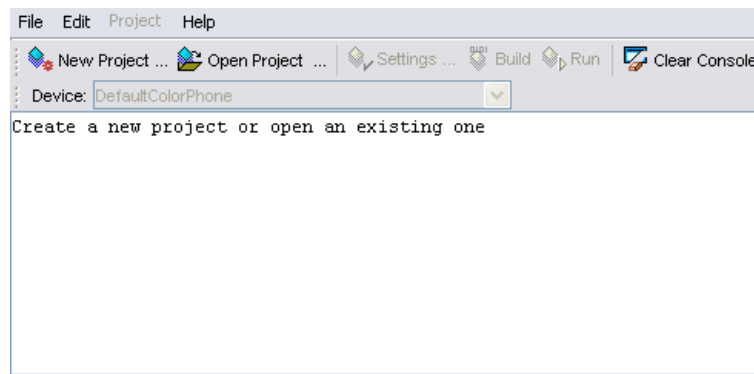
---



Hemos visto en el punto anterior los pasos que deberemos seguir para probar nuestras aplicaciones MIDP: compilar, preverificar, empaquetar, crear el archivo JAD y ejecutar en un emulador.

Normalmente, mientras escribimos el programa querremos probarlo numerosas veces para comprobar que lo que llevamos hecho funciona correctamente. Si cada vez que queremos probar el programa tuviésemos que realizar todos los pasos vistos anteriormente de forma manual programar aplicaciones MIDP sería una tarea tediosa. Además requeriría aprender a manejar todas las herramientas necesarias para realizar cada paso en la línea de comando.

Por ello los kits de desarrollo, y concretamente WTK, proporcionan entornos para crear aplicaciones de forma automatizada, sin tener que trabajar directamente con las herramientas en línea de comando. Esta herramienta principal de WTK en versiones anteriores a la 2.5 recibía el nombre de `ktoolbar`:



Wireless Toolkit

Este entorno nos permitirá construir la aplicación a partir del código fuente, pero no proporciona ningún editor de código fuente, por lo que tendremos que escribir el código fuente utilizando cualquier editor externo. Otra posibilidad es integrar WTK en algún entorno de desarrollo integrado (IDE) de forma que tengamos integrado el editor con todas las herramientas para construir las aplicaciones facilitando más aun la tarea del desarrollador. En el siguiente punto veremos como desarrollar aplicaciones utilizando un IDE.

### Directorio de aplicaciones

Este entorno de desarrollo guarda todas las aplicaciones dentro de un mismo directorio de aplicaciones. Cada aplicación estará dentro de un subdirectorio dentro de este directorio de aplicaciones, cuyo nombre corresponderá al nombre de la aplicación.

Por defecto, este directorio de aplicaciones es el directorio `${WTK_HOME}/apps`, pero podemos modificarlo añadiendo al fichero `ktools.properties` la siguiente línea:

```
kvem.apps.dir: <directorio de aplicaciones>
```

Además, dentro de este directorio hay un directorio `lib`, donde se pueden poner las librerías externas que queremos que utilicen todas las aplicaciones. Estas librerías serán ficheros JAR cuyo contenido será incorporado a las aplicaciones MIDP que creemos, de forma que podamos utilizar esta librería dentro de ellas.

Por ejemplo, después de instalar WTK podemos encontrar a parte del directorio de librerías una serie de aplicaciones de demostración instaladas. El directorio de aplicaciones puede contener por ejemplo los siguientes directorios (en el caso de WTK 2.1):

```
audiodemo
demos
FPDemo
games
JSR172Demo
lib
mmademo
NetworkDemo
photoalbum
SMSDemo
tmp1ib
UIDemo
```

Tendremos por lo tanto las aplicaciones `games`, `demos`, `photoalbum`, y `UIDemo`. El directorio `tmp1ib` lo utiliza el entorno para trabajar de forma temporal con las librerías del directorio `lib`.

NOTA: Dado que se manejan gran cantidad de herramientas y emuladores independientes en el desarrollo de las aplicaciones MIDP, es recomendable que el directorio donde está instalada la aplicación (ni ninguno de sus ascendientes) contenga espacios en blanco, ya que algunas aplicaciones puede fallar en estos casos.

### Estructura de las aplicaciones

Dentro del directorio de cada aplicación, se organizarán los distintos ficheros de los que se compone utilizando la siguiente estructura de directorios:

```
bin
lib
res
src
classes
tmpclasses
tmp1ib
```

Deberemos crear el código fuente de la aplicación dentro del directorio `src`, creando dentro de este directorio la estructura de directorios correspondiente a los paquetes a los que pertenezcan nuestras clases.

En `res` guardaremos todos los recursos que nuestra aplicación necesite, pudiendo crear dentro de este directorio la estructura de directorios que queramos para organizar estos recursos.

Por último, en `lib` deberemos poner las librerías adicionales que queramos incorporar a

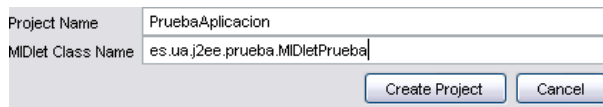
nuestra aplicación. Pondremos en este directorio el fichero JAR con la librería de clases que queramos añadir. Lo que se hará será añadir todas las clases contenidas en estas librerías, así como las contenidas en las librerías globales que hemos visto anteriormente, al fichero JAR que creemos para nuestra aplicación.

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

Esto es todo lo que tendremos que introducir nosotros. Todo lo demás será generado automáticamente por la herramienta `ktoolbar` como veremos a continuación. En el directorio `classes` se generarán las clases compiladas y preverificadas de nuestra aplicación, y en `bin` tendremos finalmente los ficheros JAR y JAD para desplegar nuestra aplicación.

### Creación de una nueva aplicación

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



Crear una nueva aplicación

Debemos indicar aquí un nombre para la aplicación (*Project Name*), que será el nombre del directorio donde se guardará la aplicación. Además deberemos indicar el nombre de la clase correspondiente al MIDlet principal de la *suite* (*MIDlet Class Name*). Es posible que nosotros todavía no hayamos creado esta clase, por lo que deberemos indicar el nombre que le asignaremos cuando la creemos. De todas formas este dato puede ser modificado más adelante.

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

Key	Value
MIDlet-Jar-Size	100
MIDlet-Jar-URL	PruebaAplicacion.jar
MIDlet-Name	PruebaAplicacion
MIDlet-Vendor	Unknown
MIDlet-Version	1.0
MicroEdition-Configuration	CLDC-1.0
MicroEdition-Profile	MIDP-1.0

OK Cancel

Configuración de la aplicación

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

En la segunda pestaña tenemos los datos opcionales que podemos introducir en estos ficheros:

Key	Value
MIDlet-Data-Size	
MIDlet-Delete-Confirm	
MIDlet-Delete-Notify	
MIDlet-Description	
MIDlet-Icon	
MIDlet-Info-URL	
MIDlet-Install-Notify	

OK Cancel

Datos opcionales de configuración

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

Key	Value
msg.bienvenida	Hola mundo!

Add Remove

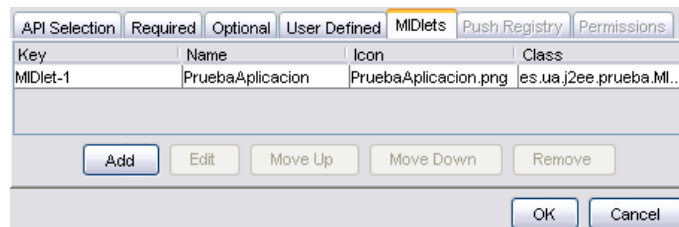
OK Cancel

Propiedades personalizadas

Aquí podemos añadir o eliminar cualquier otra propiedad que queramos definir para nuestra aplicación. De esta forma podemos parametrizarlas. En el ejemplo de la figura hemos creado una propiedad `msg.bienvenida` que contendrá el texto de bienvenida que

mostrará nuestra aplicación. De esta forma podremos modificar este texto simplemente modificando el valor de la propiedad en el JAD, sin tener que recompilar el código.

En la última pestaña tenemos los datos de los MIDlets que contiene la *suite*. Por defecto nos habrá creado un único MIDlet:

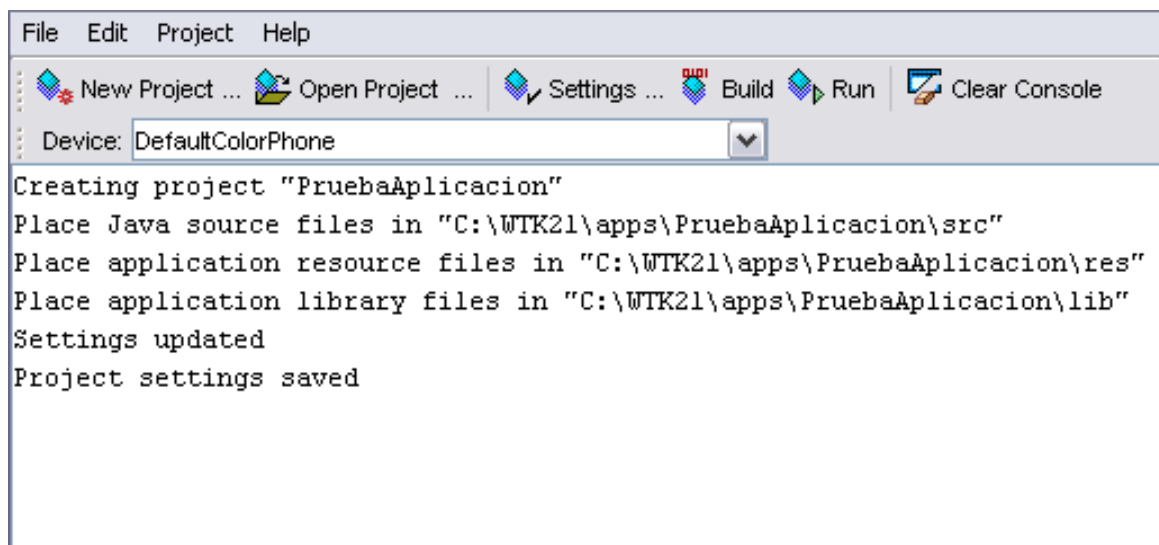


Datos de los MIDlets

Por defecto le habrá dado a este MIDlet el mismo nombre que a la aplicación, es decir, el nombre del proyecto que hemos especificado, al igual que ocurre con el nombre del icono. Como clase correspondiente al MIDlet habrá introducido el nombre de la clase que hemos especificado anteriormente.

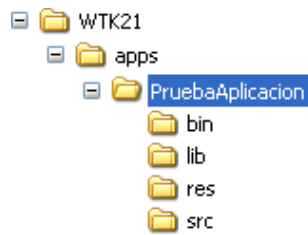
Dado que una *suite* puede contener más de un MIDlet, desde esta pestaña podremos añadir tantos MIDlets como queramos, especificando para cada uno de ellos su nombre, icono (de forma opcional) y clase.

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



Aplicación creada

Con este mensaje nos notifica el directorio donde se ha creado la aplicación, y los subdirectorios donde debemos introducir el código fuente, recursos y librerías externas de nuestra aplicación. Se habrá creado la siguiente estructura de directorios en el disco:

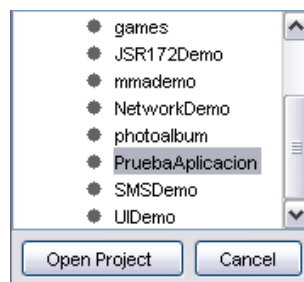


Estructura de directorios

En el directorio `bin` se habrán creado los ficheros `JAD` y `MANIFEST.MF` provisionales con los datos que hayamos introducido. Los demás directorios estarán vacíos, deberemos introducir en ellos todos los componentes de nuestra aplicación.

### Abrir una aplicación ya existente

Si tenemos una aplicación ya creada, podemos abrirla desde el entorno para continuar trabajando con ella. Para abrir una aplicación pulsamos **"Open Project ..."** y nos mostrará la siguiente ventana con las aplicaciones disponibles:



Abrir proyecto

Podemos seleccionar cualquiera de ellas y abrirla pulsando **"Open Project"**. Una vez abierta podremos modificar todos los datos que hemos visto anteriormente correspondientes a los ficheros `JAD` y `MANIFEST.MF` pulsando sobre el botón **"Settings ..."**.

Además podremos compilarla, empaquetarla y probarla en cualquier emulador instalado como veremos a continuación.

#### 1.2.4.2. Compilación y empaquetamiento

Una vez hemos escrito el código fuente de nuestra aplicación MIDP (en el directorio `src`) y hemos añadido los recursos y las librerías necesarias para ejecutarse dicha aplicación (en los directorios `res` y `lib` respectivamente) podremos utilizar la herramienta `ktoolbar` para realizar de forma automatizada todos los pasos para la construcción de la aplicación. Vamos a ver ahora como realizar este proceso.

### Compilación

Para compilar el código fuente de la aplicación simplemente deberemos pulsar el botón **"Build"** o ir a la opción del menú **Project > Build**. Con esto compilará y preverificará de forma automática todas las clases de nuestra aplicación, guardando el resultado en el directorio `classes` de nuestro proyecto.

Para compilar las clases utilizará como *classpath* la API proporcionada por el emulador seleccionado actualmente. Para los emuladores distribuidos con WTK estas clases serán las API básica de MIDP (1.0 ó 2.0 según la versión de WTK instalada). Sin embargo, podemos incorporar emuladores que soporten APIs adicionales, como por ejemplo MMAPI para dar soporte a elementos multimedia, o APIs propietarias de distintas compañías como Nokia. En caso de tener seleccionado un emulador con alguna de estas APIs adicionales, estas APIs también estarán incluidas en el *classpath*, por lo que podremos compilar correctamente programas que las utilicen. El emulador seleccionado aparece en el desplegable **Device**.

### Ofuscación

El entorno de desarrollo de WTK también nos permitirá ofuscar el código de forma automática. Este paso es opcional, y si queremos que WTK sea capaz de utilizar la ofuscación deberemos descargar alguno de los ofuscadores soportados por este entorno, como *ProGuard* (en WTK 2.X) o *RetroGuard* (en WTK 1.0). Estos ofuscadores son proporcionados por terceros.

Una vez tenemos uno de estos ofuscadores, tendremos un fichero JAR con las clases del ofuscador. Lo que deberemos hacer para instalarlo es copiar este fichero JAR al directorio `${WTK_HOME}/bin`. Una vez tengamos el fichero JAR del ofuscador en este directorio, WTK podrá utilizarlo de forma automática para ofuscar el código.

La ofuscación la realizará WTK en el mismo paso de la creación del paquete JAR, en caso de disponer de un ofuscador instalado, como veremos a continuación.

### Empaquetamiento

Para poder instalar una aplicación en el móvil y distribuirla, deberemos generar el fichero JAR con todo el contenido de la aplicación. Para hacer esto de forma automática deberemos ir al menú **Project > Package**. Dentro de este menú tenemos dos opciones:

- **Create Package**
- **Create Obfuscated Package**

Ambas realizan todo el proceso necesario para crear el paquete de forma automática: compilan los fuentes, ofuscan (sólo en el segundo caso), preverifican y empaquetan las clases resultantes en un fichero JAR. Por lo tanto no será necesario utilizar la opción **Build** previamente, ya que el mismo proceso de creación del paquete ya realiza la compilación y la preverificación.

Una vez construido el fichero JAR lo podremos encontrar en el directorio `bin` de la aplicación. Además este proceso actualizará de forma automática el fichero JAD, para

establecer el tamaño correcto del fichero JAR que acabamos de crear en la propiedad correspondiente.

#### 1.2.4.3. Ejecución en emuladores

---

Dentro del mismo entorno de desarrollo de WTK podemos ejecutar la aplicación en diferentes emuladores que haya instalados para probarla. Podemos seleccionar el emulador a utilizar en el cuadro desplegable **Device** de la ventana principal de `ktoolbar`.

Para ejecutar la aplicación en el emulador seleccionado solo debemos pulsar el botón **"Run"** o la opción del menú **Project > Run**. Normalmente, para probar la aplicación en un emulador no es necesario haber creado el fichero JAR, simplemente con las clases compiladas es suficiente. En caso de ejecutarse sin haber compilado las clases, el entorno las compilará de forma automática.

Sin embargo, hay algunos emuladores que sólo funcionan con el fichero JAR, por lo que en este caso deberemos crear el paquete antes de ejecutar el emulador. Esto ocurre por ejemplo con algún emulador proporcionado por Nokia.

Por ejemplo, los emuladores de teléfonos móviles proporcionados con WTK 2.2 son:

- **DefaultColorPhone**. Dispositivo con pantalla a color.
- **DefaultGrayPhone**. Dispositivo con pantalla monocroma.
- **MediaControlSkin**. Dispositivo con teclado orientado a la reproducción de elementos multimedia.
- **QwertyDevice**. Dispositivo con teclado de tipo QWERTY.

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

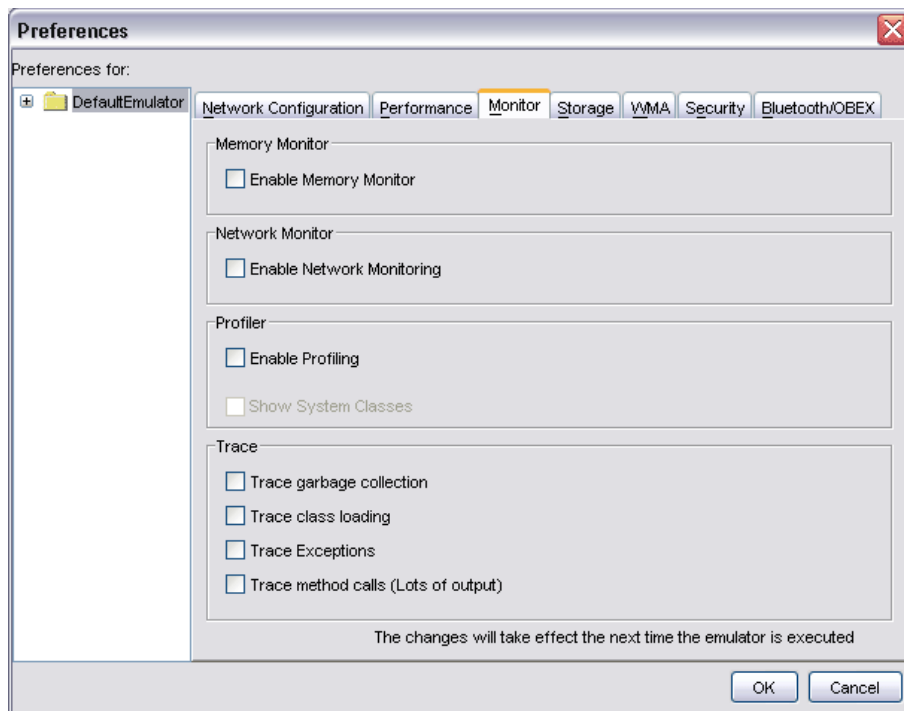
Para integrar los emuladores de teléfonos Nokia en WTK simplemente tendremos que instalar estos emuladores en el directorio `${WTK_HOME}/wtllib/devices`. Una vez instalados en este directorio, estos emuladores estarán disponibles dentro del kit de desarrollo, de forma que podremos seleccionarlos en el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

Podemos encontrar además emuladores proporcionados por otras compañías. WTK también nos permite personalizar los emuladores, cambiando su aspecto y características para adaptarlos a nuestras necesidades.

### Optimización

En WTK, además de los emuladores, contamos con herramientas adicionales que nos ayudarán a optimizar nuestras aplicaciones. Desde la ventana de preferencias podemos activar distintos monitores que nos permitirán monitorizar la ocupación de memoria y el tráfico en la red:





### Monitorización

Será conveniente utilizar estos monitores para medir el consumo de recursos de nuestra aplicación e intentar reducirlo al mínimo.

En cuanto a la memoria, deberemos intentar que el consumo sea lo menor posible y que nunca llegue a pasar de un determinado umbral. Si la memoria creciese sin parar en algún momento la aplicación fallaría por falta de memoria al llevarla a nuestro dispositivo real.

Es importante también intentar minimizar el tráfico en la red, ya que en los dispositivos reales este tipo de comunicaciones serán lentas y caras.

Desde esta ventana de preferencias podemos cambiar ciertas características de los emuladores, como el tamaño máximo de la memoria o la velocidad de su procesador. Es conveniente intentar utilizar los parámetros más parecidos a los dispositivos para los cuales estemos desarrollando, sobretodo en cuanto a consumo de memoria, para asegurarnos de que la aplicación seguirá funcionando cuando la llevamos al dispositivo real.

#### 1.2.4.4. Provisionamiento OTA

Hemos visto como probar la aplicación directamente utilizando emuladores. Una vez generados los ficheros JAR y JAD también podremos copiarlos a un dispositivo real y probarlos ahí.

Sin embargo, cuando un usuario quiera utilizar nuestra aplicación, normalmente lo hará

vía OTA (Over The Air), es decir, se conectará a la dirección donde hayamos publicado nuestra aplicación y la descargará utilizando la red de nuestro móvil.

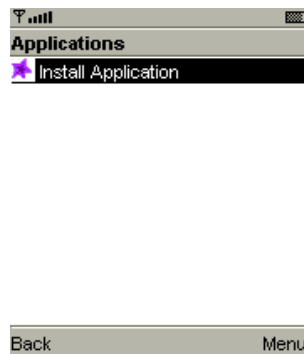
Para desplegar una aplicación de forma que sea accesible vía OTA, simplemente deberemos:

- Publicar los ficheros JAR y JAD de nuestra aplicación en un servidor web, que sea accesible a través de Internet.
- Crear un documento web que tenga un enlace al fichero JAD de nuestra aplicación. Este documento puede ser por ejemplo WML, XHTML o XHTML.
- Configurar el servidor web para que asocie los ficheros JAD y JAR al tipo MIME adecuado, tal como hemos visto anteriormente.
- Editar el fichero JAD. En la línea donde hace referencia a la URL del fichero JAR deberemos indicar la URL donde hemos desplegado realmente el fichero JAR.

Una vez está desplegada la aplicación vía OTA, el provisionamiento OTA consistirá en los siguientes pasos:

- El usuario accede con su móvil a nuestra dirección de Internet utilizando un navegador web.
- Selecciona el enlace que lleva al fichero JAD de nuestra aplicación
- El navegador descarga el fichero JAD
- El fichero JAD será abierto por el AMS del móvil, que nos mostrará sus datos y nos preguntará si queremos instalar la aplicación.
- Si respondemos afirmativamente, se descargará el fichero JAR utilizando la URL que se indica en el fichero JAD.
- Instalará la aplicación en el móvil.
- Una vez instalada, se añadirá la aplicación a la lista de aplicaciones instaladas en nuestro móvil. Desde esta lista el usuario del móvil podrá ejecutar la aplicación cada vez que quiera utilizarla. Cuando el usuario no necesite la aplicación, podrá desinstalarla para liberar espacio en el medio de almacenamiento del móvil.

A partir de WTK 2.0 podemos simular en los emuladores el provisionamiento OTA de aplicaciones. Ejecutando la aplicación *OTA Provisioning* se nos abrirá el emulador que tengamos configurado por defecto y nos dará la opción de instalar aplicaciones (*Install Application*) vía OTA. Si pulsamos sobre esta opción nos pedirá la URL donde hayamos publicado nuestra aplicación.



Ejecución via OTA (I)



Ejecución via OTA (II)

De esta forma podremos probar aplicaciones publicadas en algún servidor de Internet. Para probar nuestras aplicaciones utilizando este procedimiento deberemos desplegar previamente nuestra aplicación en un servidor web y utilizar la dirección donde la hayamos desplegados para instalar la aplicación desde ese lugar.

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

La aplicación `ktoolbar` nos ofrece una facilidad con el que simular el provisionamiento OTA utilizando un servidor web interno, de forma que no tendremos que publicar la aplicación manualmente para probarla. Para ello, abriremos nuestro proyecto en `ktoolbar` y seleccionaremos la opción **Project > Run via OTA**. Con esto, automáticamente nos rellenará la dirección de donde queremos instalar la aplicación con la dirección interna donde está desplegada:



## Ejecución via OTA (III)



## Ejecución via OTA (IV)

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



## Ejecución via OTA (V)



Ejecución via OTA (VI)

Si aceptamos la instalación de la aplicación, pulsando sobre *Install*, descargará el fichero JAR con la aplicación y lo instalará. Ahora veremos esta aplicación en la lista de aplicaciones instaladas:



Ejecución via OTA (VII)

Desde esta lista podremos ejecutar la aplicación e instalar nuevas aplicaciones que se vayan añadiendo a esta lista. Cuando no necesitemos esta aplicación desde aquí también podremos desinstalarla.

### 1.2.5. Entornos de Desarrollo Integrados (IDEs)

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

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

Para desarrollar aplicaciones J2ME podremos utilizar la mayoría de los IDEs existentes

para Java, añadiendo alguna extensión para permitirnos trabajar con este tipo de aplicaciones. También podemos encontrar entornos dedicados exclusivamente a la creación de aplicaciones J2ME.

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

#### 1.2.5.1. Eclipse

---

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

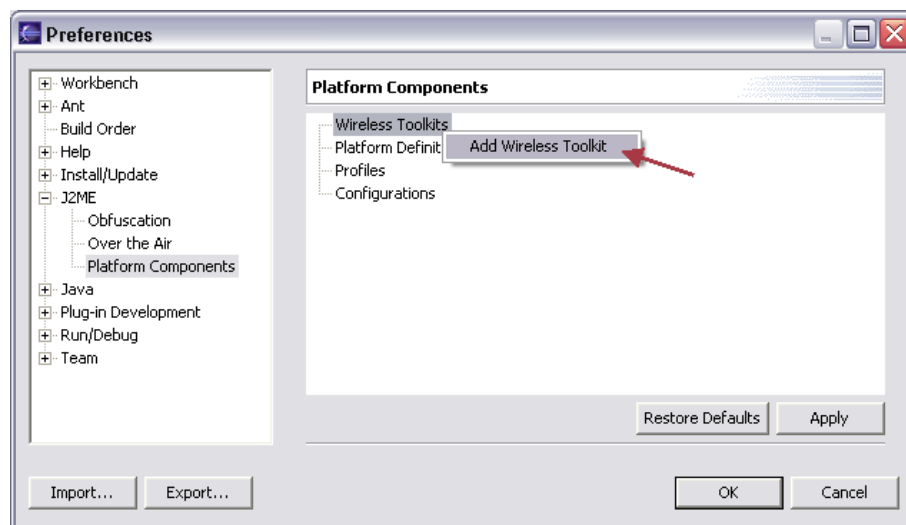
Este entorno nos permite crear proyectos en Java. Nos ofrece un editor, en el que podemos escribir el código, viendo la sintaxis coloreada para mayor claridad, y notificándonos de los errores que hayamos cometido al escribir el código, como por ejemplo haber escrito mal el nombre de un método, o usar un tipo o número incorrecto de parámetros. Además nos permitirá autocompletar los nombres de los métodos o las propiedades de las clases conforme los escribimos. Si el código ha quedado desordenado, nos permite darle formato automáticamente, poniendo la sangría adecuada para cada línea de código. Esto nos facilitará bastante la escritura del código fuente. Sin embargo, no nos permitirá crear visualmente la GUI de las aplicaciones.

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

#### EclipseME

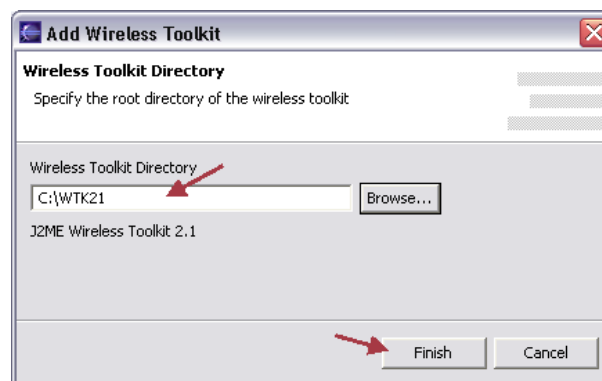
EclipseME es un *plugin* realizado por terceros que nos facilitará la creación de aplicaciones J2ME desde Eclipse.

Lo primero que debemos hacer es instalar el *plugin*, descomprimiéndolo en el directorio `${ECLIPSE_HOME}/plugins`. Una vez hecho esto, deberemos reiniciar el entorno, y entonces deberemos ir a **Window > Preferences** para configurar el *plugin*:



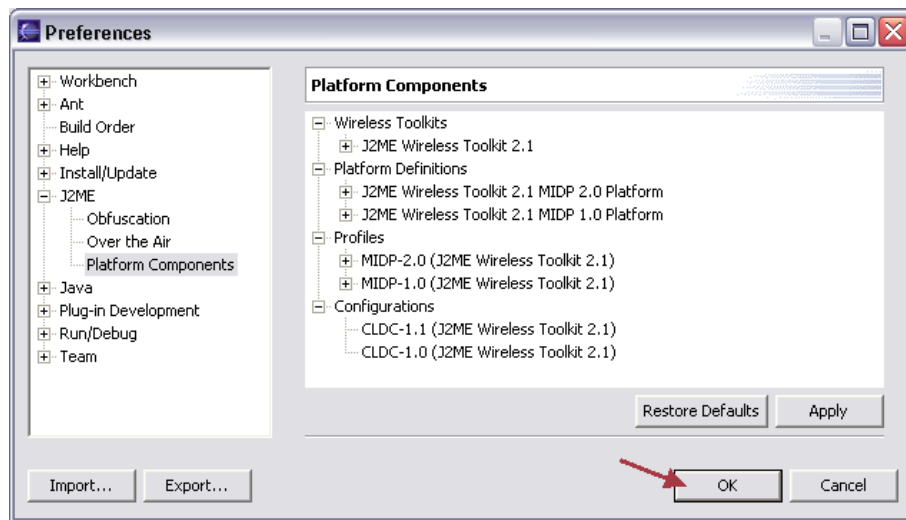
Eclipse ME (I)

En el apartado de configuración de J2ME, dentro del subapartado **Platform Components**, deberemos especificar el directorio donde tenemos instalado WTK. Para ello pulsamos con el botón derecho del ratón sobre **Wireless Toolkits** y seleccionamos la opción **Add Wireless Toolkit**. Nos mostrará la siguiente ventana, en la que deberemos seleccionar el directorio donde se encuentra WTK:



Eclipse ME (II)

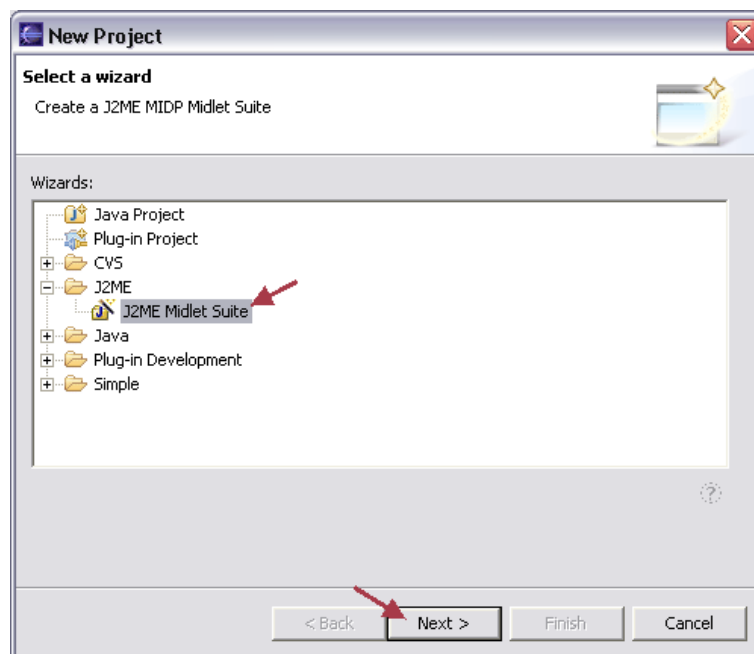
Una vez añadido un *toolkit*, se mostrarán los componentes añadidos en la ventana de configuración:



Eclipse ME (III)

De esta forma vemos que al añadir WTK 2.1 hemos añadido soporte para los perfiles MIDP 1.0 y 2.0, y para las configuraciones CLDC 1.0 y 1.1. Podremos configurar varios *toolkits*. Por ejemplo, podemos tener configuradas las distintas versiones de WTK (1.0, 2.0, 2.1 y 2.2) para utilizar la que convenga en cada momento. Una vez hayamos terminado de configurar los *toolkits*, pulsaremos **OK** para cerrar esta ventana.

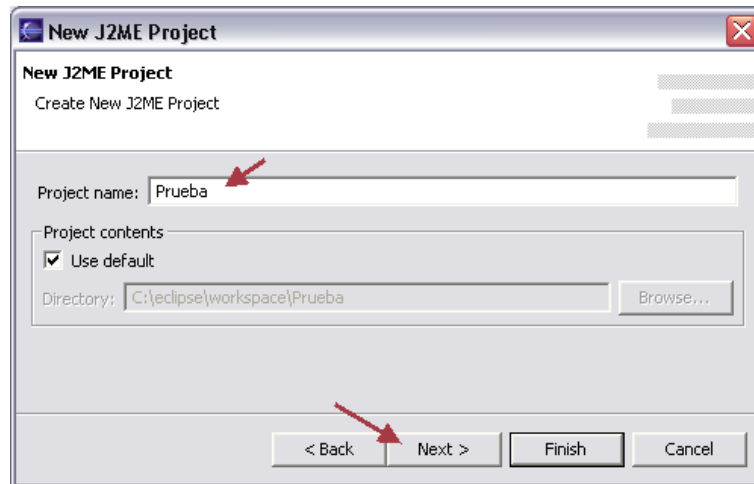
Una vez configurado, podremos pulsar sobre **New**, donde encontraremos disponibles asistentes para crear aplicaciones J2ME:



Eclipse ME (IV)

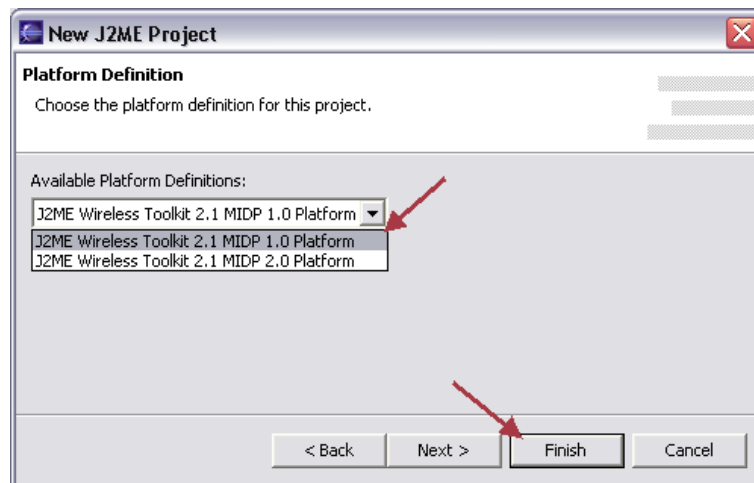


Lo primero que haremos será crear la *suite* (proyecto). Seleccionamos **J2ME Midlet Suite** y pulsamos **Next** para comenzar con el asistente de creación de la *suite* J2ME:



Eclipse ME (V)

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

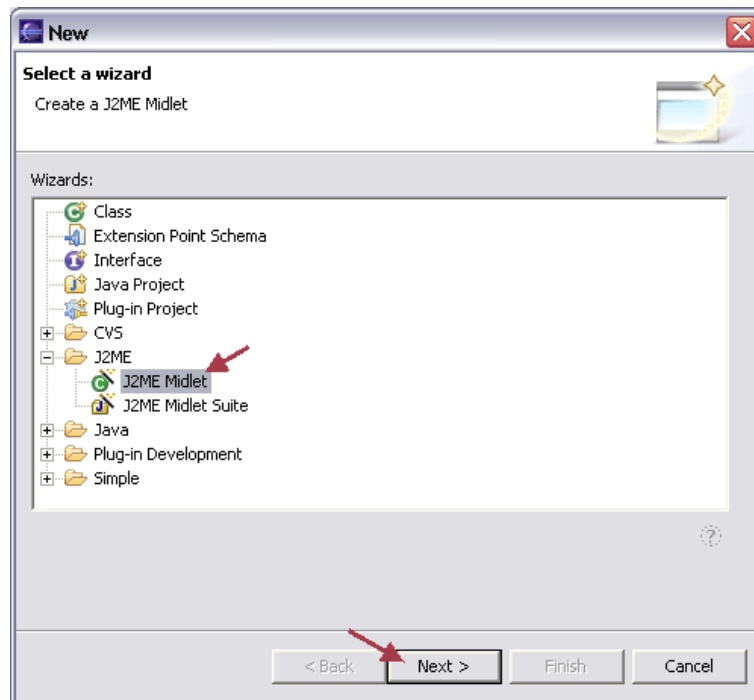


Eclipse ME (VI)

Aquí podemos elegir la versión de MIDP para la que queremos programar, siempre que tengamos instalado el WTK correspondiente para cada una de ellas. Una vez elegida la versión para la que queremos desarrollar pulsamos **Finish**, con lo que habremos terminado de configurar nuestro proyecto. En este caso no hace falta que especifiquemos las librerías de forma manual, ya que el asistente las habrá configurado de forma

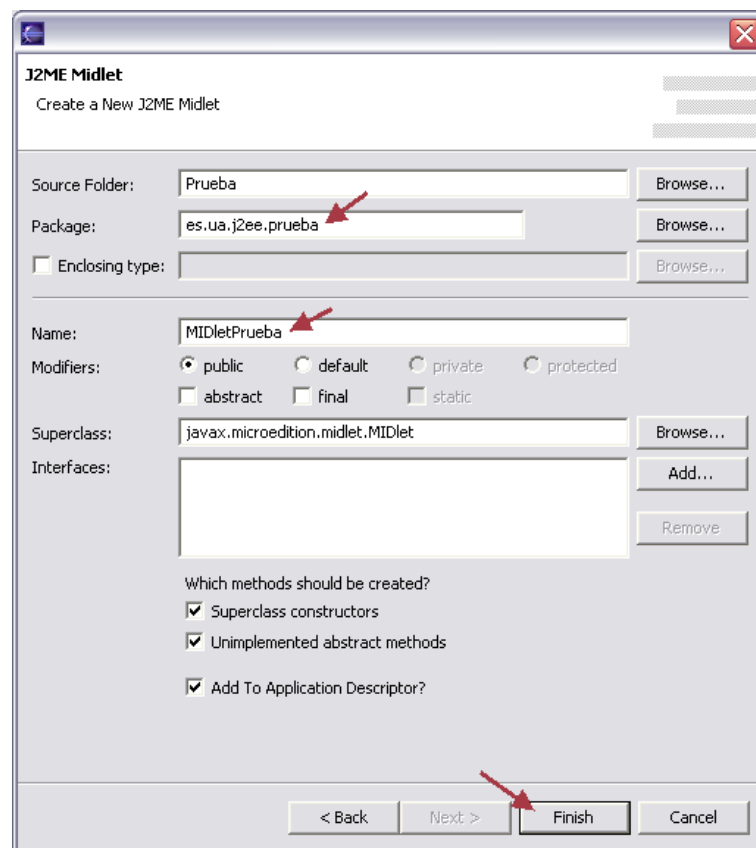
automática.

Una vez creado el proyecto, podremos añadir MIDlets u otras clases Java. Pulsando sobre **New** veremos los elementos que podemos añadir a la *suite*:



Eclipse ME (VII)

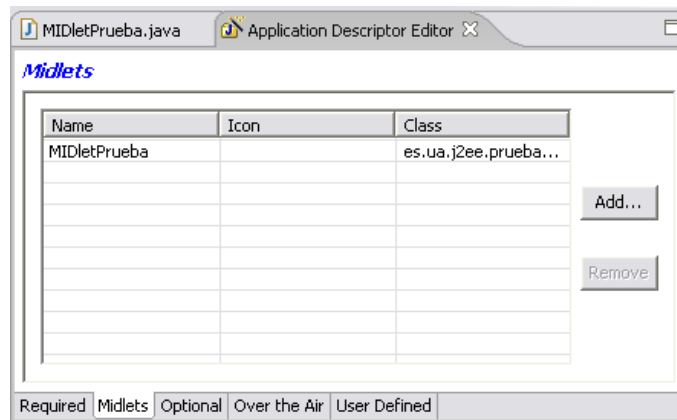
Si queremos crear un MIDlet, podremos utilizar la opción **J2ME Midlet** y pulsar **Next**, con lo que se mostrará la siguiente ventana para introducir los datos del MIDlet:



Eclipse ME (VIII)

Aquí deberemos dar el nombre del paquete y el nombre de la clase de nuestro MIDlet. Pulsando sobre **Finish** creará el esqueleto de la clase correspondiente al MIDlet, donde nosotros podremos insertar el código necesario.

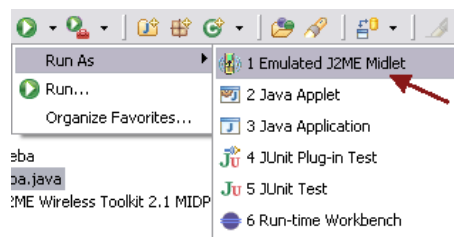
En el explorador de paquetes podemos ver las clases creadas, la librería utilizada y el fichero JAD del proyecto. Pinchando sobre el fichero JAD se mostrará en el editor la siguiente ficha con los datos de la *suite*:



Eclipse ME (IX)

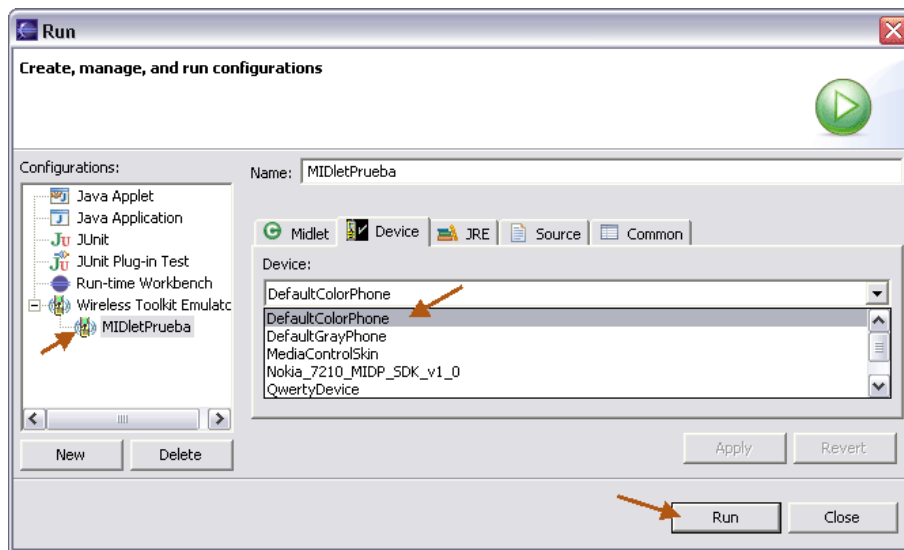
Aquí deberemos introducir la información necesaria, sobre los datos de la *suite* (**Required**) y los MIDlets que hayamos creado en ella (en la pestaña **Midlets**). Podemos ver que, cuando creemos un MIDlet mediante el asistente que acabamos de utilizar para la creación de MIDlets, los datos del MIDlet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

No es necesario compilar el proyecto manualmente, ya que Eclipse se ocupará de ello. Cuando queramos ejecutarlo, podemos seleccionar en el explorador de paquetes el MIDlet que queramos probar y pulsar sobre el botón **Run > Emulated J2ME Midlet**:



Eclipse ME (X)

Esto abrirá nuestro MIDlet en el emulador que se haya establecido como emulador por defecto del *toolkit* utilizado. Si queremos tener un mayor control sobre cómo se ejecuta nuestra aplicación, podemos utilizar la opción **Run...** que nos mostrará la siguiente ventana:



Eclipse ME (XI)

En esta ventana pulsaremos sobre **Wireless Toolkit Emulator** y sobre **New** para crear una nueva configuración de ejecución sobre los emuladores de J2ME. Dentro de esta configuración podremos seleccionar el emulador dentro de la pestaña **Device**, y una vez seleccionado ya podremos pulsar sobre **Run** para ejecutar la aplicación.

#### 1.2.5.2. Depuración con Eclipse

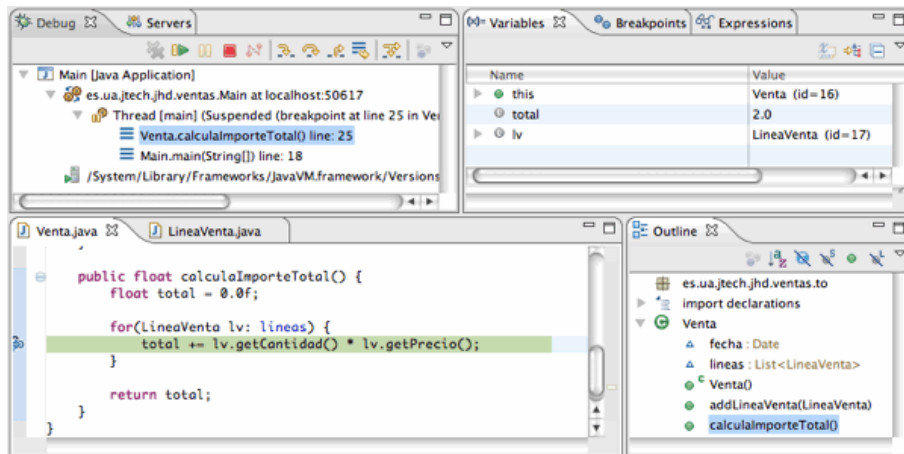
En este apartado veremos cómo podemos depurar el código de nuestras aplicaciones desde el depurador que incorpora Eclipse, encontrando el origen de los errores que provoque nuestro programa. Este depurador incorpora muchas funcionalidades, como la ejecución paso a paso, establecimiento de *breakpoints*,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

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

##### Primeros pasos para depurar un proyecto

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

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



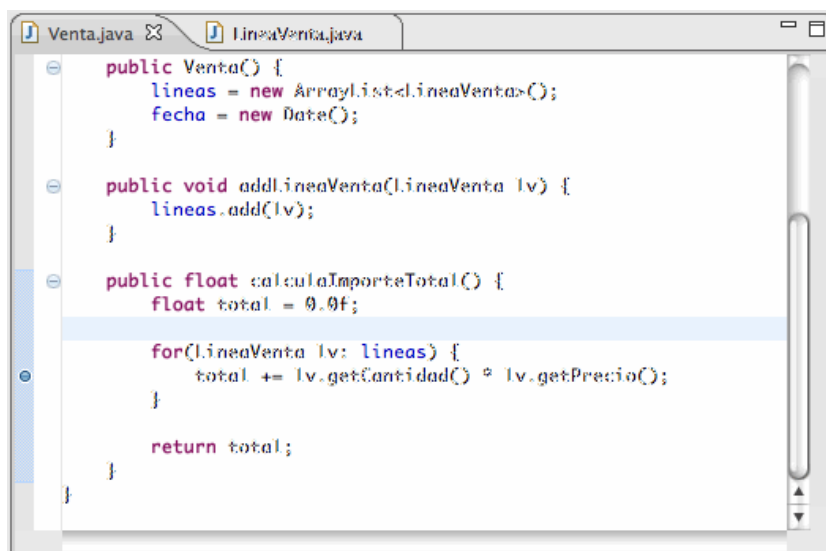
Perspectiva de depuración

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

### Establecer breakpoints

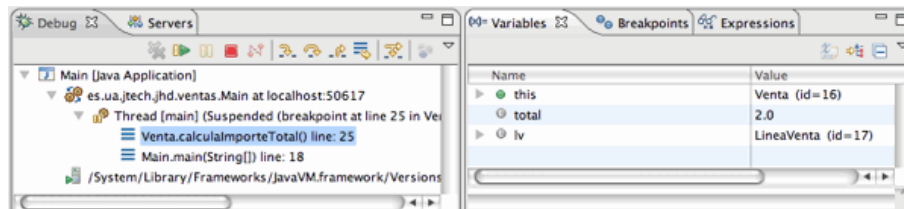
Una de las operaciones más habituales a la hora de depurar es establecer *breakpoints*, puntos en los que la ejecución del programa se detiene para permitir al programador examinar cuál es el estado del mismo en ese momento.

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



## Establecer breakpoints

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

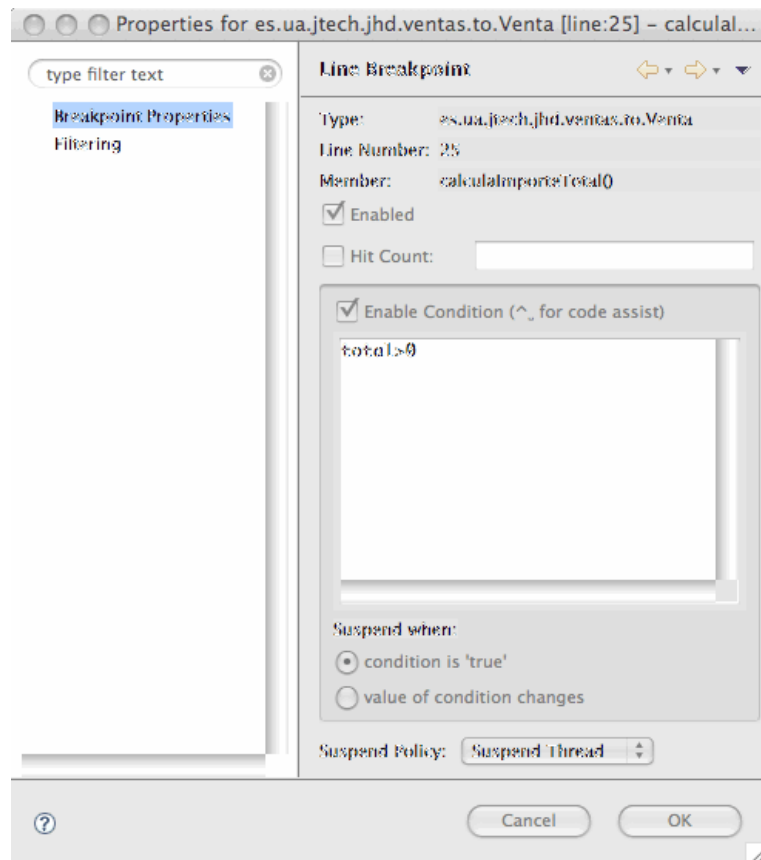


Control del estado del programa

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

## Breakpoints condicionales

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

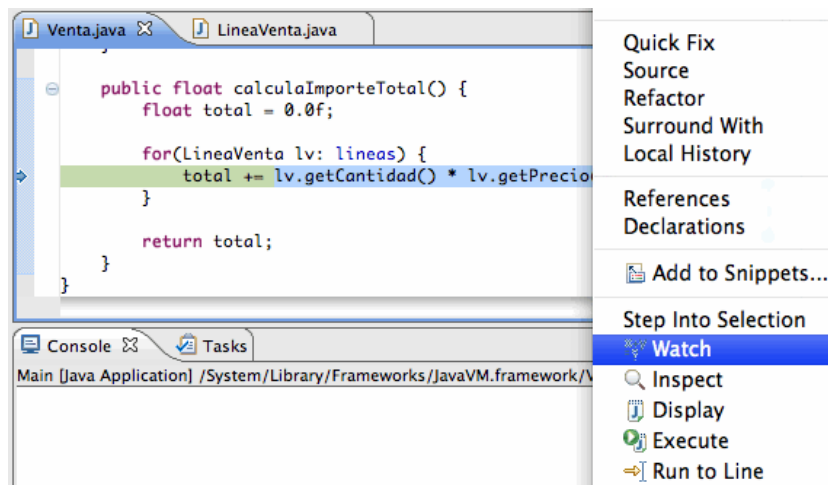


Breakpoints condicionales

### Evaluar expresiones

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



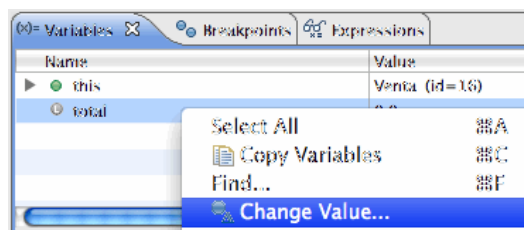


Evaluación de expresiones

### Explorar variables

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

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



Exploración de variables

### Cambiar código "en caliente"

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

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

## Gestión de Logs

---

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

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

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

### 1.2.5.3. Otros entornos

---

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

#### JavaME SKD 3.0

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

#### NetBeans

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

#### Sun One Studio ME

Se trata de la versión ME (*Micro Edition*) del entorno de desarrollo de Sun, Sun One Studio, anteriormente llamado Forte for Java. Esta versión ME está dirigida a crear aplicaciones J2ME, e incluye todo el software necesario para realizar esta tarea, no hace falta instalar por separado el WTK ni otras herramientas.

El entorno es muy parecido a NetBeans. Podemos descargar una versión de prueba sin ninguna limitación. Una ventaja de este entorno es que podemos integrarlo con otros kits de desarrollo como por ejemplo el kit de desarrollo de Nokia.

#### JBuilder y MobileSet

Podemos utilizar también el entorno de Borland, JBuilder, con la extensión MobileSet. A

partir de la versión 9 de JBuilder tenemos una edición Mobile para trabajar con aplicaciones J2ME directamente sin tener que instalar ninguna extensión. Podemos descargar de forma gratuita la versión personal del entorno JBuilder, pero tiene el inconveniente de estar bastante más limitada que las versiones de pago.

Este entorno puede también integrarse con el kit de desarrollo de Nokia. Además como característica adicional podremos crear de forma visual la GUI de las aplicaciones móviles. Esta característica no está muy extendida por este tipo de entornos debido a la simplicidad de las GUIs para móviles.

### **JDeveloper y J2ME Plugin**

El entorno de desarrollo de Oracle, JDeveloper, está dedicado principalmente a la creación de aplicaciones J2EE, permitiéndonos crear un gran número de componentes Java, como *servlets*, JSPs, EJBs, servicios web, etc. Para facilitar la tarea de creación de estos componentes, automatizando todo lo posible, utiliza APIs propietarias de Oracle.

Podemos trabajar directamente en vista de diseño, utilizar distintos patrones de diseño para desarrollar las aplicaciones web, etc. Tiene integrado un servidor de aplicaciones propio para probar las aplicaciones en modo local, y nos permite establecer conexiones a BDs y a servidores de aplicaciones para realizar el despliegue de estas aplicaciones.

Aunque está principalmente dedicado para aplicaciones web con J2EE, también podemos utilizarlo para aplicaciones J2SE. Además también podemos encontrar un *plugin* para realizar aplicaciones J2ME, permitiéndonos crear MIDlets y *suites* mediante asistentes, y ejecutar las aplicaciones directamente en emuladores.

Podemos descargar de forma gratuita una versión de prueba de este entorno de la web sin limitaciones.

### **Websphere Studio Device Developer**

Se trata de un entorno de IBM basado en Eclipse, por lo que tiene una interfaz similar. Este entorno esta dedicado a la programación de aplicaciones para dispositivos móviles. Integra los asistentes necesarios para la creación de los componentes de aplicaciones MIDP, así como las herramientas de desarrollo necesarias y nos permite probar la aplicación directamente en emuladores desde el mismo entorno.

Podemos encontrar en la web una versión de prueba sin limitaciones para descargar.

### **Codewarrior Wireless Studio**

Este es otro entorno bastante utilizado también para el desarrollo de aplicaciones para móviles. Está desarrollado por Metrowerks y se puede encontrar disponible para un gran número de plataformas distintas. Existe una versión de evaluación limitada a 30 días de uso que puede ser encargada desde la web.

### **Antenna**

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

La herramienta `ant` nos permite automatizar tareas como la compilación, empaquetamiento, despliegue o ejecución de aplicaciones. Es similar a la herramienta `make`, pero con la ventaja de que es totalmente independiente de la plataforma, ya que en lugar de utilizar comandos nativos utiliza clases Java para realizar las tareas.

Tiene una serie de tareas definidas, que servirán para compilar clases, empaquetar en ficheros JAR, ejecutar aplicaciones, etc. Todas estas tareas están implementadas mediante clases Java. Además, nos permitirá añadir nuevas tareas, incorporando una librería de clases Java que las implemente.

*Antenna* es una librería de tareas para *ant* que nos permitirán trabajar con aplicaciones MIDP. Entre estas tareas encontramos la compilación y el empaquetamiento (con preverificación y ofuscación), la creación de los ficheros JAD y `MANIFEST.MF`, y la ejecución de aplicaciones en emuladores.

Para realizar estas tareas utiliza WTK, por lo que necesitaremos tener este kit de desarrollo instalado. Los emuladores que podremos utilizar para ejecutar las aplicaciones serán todos aquellos emuladores instalados en WTK.

### 1.3. Java para MIDs

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

Hemos visto que en el caso de los MIDs, este código intermedio Java se ejecutará sobre una versión reducida de la máquina virtual, la **KVM (Kilobyte Virtual Machine)**, lo cual producirá determinadas limitaciones en las aplicaciones desarrolladas para dicha máquina virtual.

Cuando se programa con Java se dispone de antemano de un conjunto de clases ya implementadas. Estas clases (aparte de las que pueda hacer el usuario) forman parte del propio lenguaje (lo que se conoce como **API (Application Programming Interface)** de Java).

La API que se utilizará para programar las aplicaciones para MIDs será la API de MIDP, que contendrá un conjunto reducido de clases que nos permitan realizar las tareas fundamentales en estas aplicaciones. La implementación de esta API estará optimizada para ejecutarse en este tipo de dispositivos. En CLDC tendremos un subconjunto reducido y simplificado de las clases de J2SE, mientras que en MIDP tendremos clases que son exclusivas de J2ME ya que van orientadas a la programación de las características propias

de los dispositivos móviles.

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

### 1.3.1. Características ausentes en JavaME

---

Además de las diferencias que hemos visto en los puntos anteriores, tenemos APIs que han desaparecido en su totalidad, o prácticamente en su totalidad.

#### Reflection

En CLDC no está presente la API de *reflection*. Sólo está presente la clase `Class` con la que podremos cargar clases dinámicamente y comprobar la clase a la que pertenece un objeto en tiempo de ejecución. Tenemos además en esta clase el método `getResourceAsStream` que hemos visto anteriormente, que nos servirá para acceder a los recursos dentro del JAR de la aplicación.

#### Red

La API para el acceso a la red de J2SE es demasiado compleja para los MIDs. Por esta razón se ha sustituido por una nueva API totalmente distinta, adaptada a las necesidades de conectividad de estos dispositivos. Desaparece la API `java.net`, para acceder a la red ahora deberemos utilizar la API `javax.microedition.io` incluida en CLDC que veremos en detalle en el próximo tema.

#### AWT/Swing

Las librerías para la creación de interfaces gráficas, AWT y Swing, desaparecen totalmente ya que estas interfaces no son adecuadas para las pantallas de los MIDs. Para crear la interfaz gráfica de las aplicaciones para móviles tendremos la API `javax.microedition.lcdui` perteneciente a MIDP.

### 1.3.2. MIDlets

---

Hasta ahora hemos visto la parte básica del lenguaje Java que podemos utilizar en los dispositivos móviles. Esta parte de la API está basada en la API básica de J2SE, reducida y optimizada para su utilización en dispositivos de baja capacidad. Esta es la base que necesitaremos para programar cualquier tipo de dispositivo, sin embargo con ella por sí sola no podemos acceder a las características propias de los móviles, como su pantalla, su teclado, reproducir tonos, etc.

Vamos a ver ahora las APIs propias para el desarrollo de aplicaciones móviles. Estas APIs ya no están basadas en APIs existentes en J2SE, sino que se han desarrollado específicamente para la programación en estos dispositivos. Todas ellas pertenecen al paquete `javax.microedition`.

Los MIDlets son las aplicaciones para MIDs, realizadas con la API de MIDP. La clase principal de cualquier aplicación MIDP deberá ser un MIDlet. Ese MIDlet podrá utilizar cualquier otra clase Java y la API de MIDP para realizar sus funciones.

Para crear un MIDlet deberemos heredar de la clase `MIDlet`. Esta clase define una serie de métodos abstractos que deberemos definir en nuestros MIDlets, introduciendo en ellos el código propio de nuestra aplicación:

```
protected abstract void startApp();
protected abstract void pauseApp();
protected abstract void destroyApp(boolean incondicional);
```

A continuación veremos con más detalle qué deberemos introducir en cada uno de estos métodos.

### 1.3.2.1. Componentes y contenedores

Numerosas veces encontramos dentro de las tecnologías Java el concepto de componentes y contenedores. Los componentes son elementos que tienen una determinada interfaz, y los contenedores son la infraestructura que da soporte a estos componentes.

Por ejemplo, podemos ver los *applets* como un tipo de componente, que para poderse ejecutar necesita un navegador web que haga de contenedor y que lo soporte. De la misma forma, los *servlets* son componentes que encapsulan el mecanismo petición/respuesta de la web, y el servidor web tendrá un contenedor que de soporte a estos componentes, para ejecutarlos cuando se produzca una petición desde un cliente. De esta forma nosotros podemos deberemos definir sólo el componente, con su correspondiente interfaz, y será el contenedor quien se encargue de controlar su ciclo de vida (instanciarlo, ejecutarlo, destruirlo).

Cuando desarrollamos componentes, no deberemos crear el método `main`, ya que estos componentes no se ejecutan como una aplicación independiente (*stand-alone*), sino que son ejecutados dentro de una aplicación ya existente, que será el contenedor.

El contenedor que da soporte a los MIDlets recibe el nombre de *Application Management Software* (AMS). El AMS además de controlar el ciclo de vida de la ejecución MIDlets (inicio, pausa, destrucción), controlará el ciclo de vida de las aplicaciones que se instalen en el móvil (instalación, actualización, ejecución, desinstalación).

### 1.3.2.2. Ciclo de vida

Durante su ciclo de vida un MIDlet puede estar en los siguientes estados:

- **Activo:** El MIDlet se está ejecutando actualmente.
- **Pausado:** El MIDlet se encuentra a mitad de una ejecución pero está pausado. La ejecución podrá reanudarse, pasando de nuevo a estado activo.
- **Destruído:** El MIDlet ha terminado su ejecución y ha liberado todos los recursos, por lo que ya no se puede volver a estado activo. La aplicación está cerrada, por lo que

para volver a ponerla en marcha tendríamos que volver a ejecutarla.

Será el AMS quién se encargue de controlar este ciclo de vida, es decir, quién realice las transiciones de un estado a otro. Nosotros podremos saber cuando hemos entrado en cada uno de estos estados porque el AMS invocará al método correspondiente dentro de la clase del MIDlet. Estos métodos son los que se muestran en el siguiente esqueleto de un MIDlet:

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

public class MiMIDlet extends MIDlet {

    protected void startApp()
        throws MIDletStateChangeException {
        // Estado activo -> comenzar
    }

    protected void pauseApp() {
        // Estado pausa -> detener hilos
    }

    protected void destroyApp(boolean incondicional)
        throws MIDletStateChangeException {
        // Estado destruido -> liberar recursos
    }
}
```

Deberemos definir los siguientes métodos para controlar el ciclo de vida del MIDlet:

- **startApp()**: Este método se invocará cuando el MIDlet pase a estado activo. Es aquí donde insertaremos el código correspondiente a la tarea que debe realizar dicho MIDlet.

Si ocurre un error que impida que el MIDlet empiece a ejecutarse deberemos notificarlo. Podemos distinguir entre errores pasajeros o errores permanentes. Los errores pasajeros impiden que el MIDlet se empiece a ejecutar ahora, pero podría hacerlo más tarde. Los permanentes se dan cuando el MIDlet no podrá ejecutarse nunca.

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

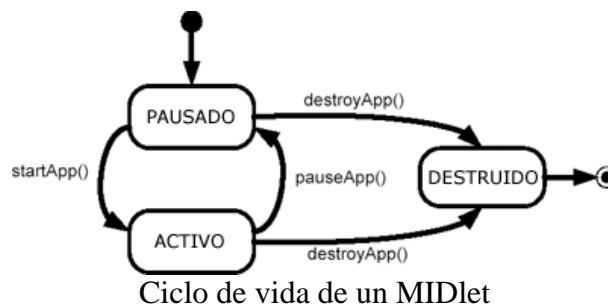
**Permanente:** Si por el contrario el error es permanente, entonces deberemos destruir el MIDlet llamando a `notifyDestroyed` porque sabemos que nunca podrá ejecutarse correctamente. Si se lanza una excepción de tipo `RuntimeException` dentro del método `startApp` tendremos el mismo efecto, se destruirá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

- **pauseApp()**: Se invocará cuando se pause el MIDlet. En él deberemos detener las actividades que esté realizando nuestra aplicación.

Igual que en el caso anterior, si se produce una excepción de tipo `RuntimeException` durante la ejecución de este método, el MIDlet se destruirá.

- **destroyApp(boolean incondicional)**: Se invocará cuando se vaya a destruir la

aplicación. En él deberemos incluir el código para liberar todos los recursos que estuviese usando el MIDlet. Con el *flag* que nos proporciona como parámetro indica si la destrucción es incondicional o no. Es decir, si `incondicional` es `true`, entonces se destruirá siempre. En caso de que sea `false`, podemos hacer que no se destruya lanzando la excepción `MIDletStateChangeException` desde dentro de este método.



Hemos visto que el AMS es quien realiza las transiciones entre distintos estados. Sin embargo, nosotros podremos forzar a que se produzcan transiciones a los estados pausado o destruido:

- **notifyDestroyed()**: Destruye el MIDlet. Utilizaremos este método cuando queramos finalizar la aplicación. Por ejemplo, podemos ejecutar este método como respuesta a la pulsación del botón "Salir" por parte del usuario.

NOTA: La llamada a este método notifica que el MIDlet ha sido destruido, pero no invoca el método `destroyApp` para liberar los recursos, por lo que tendremos que invocarlo nosotros manualmente antes de llamar a `notifyDestroyed`.

- **notifyPause()**: Notifica al AMS de que el MIDlet ha entrado en modo pausa. Después de esto, el AMS podrá realizar una llamada a `startApp` para volverlo a poner en estado activo.
- **resumeRequest()**: Solicita al AMS que el MIDlet vuelva a ponerse activo. De esta forma, si el AMS tiene varios MIDlets candidatos para activar, elegirá alguno de aquellos que lo hayan solicitado. Este método no fuerza a que se produzca la transición como en los anteriores, simplemente lo solicita al AMS y será éste quien decida.

### 1.3.2.3. Cerrar la aplicación

La aplicación puede ser cerrada por el AMS, por ejemplo si desde el sistema operativo del móvil hemos forzado a que se cierre. En ese caso, el AMS invocará el método `destroyApp` que nosotros habremos definido para liberar los recursos, y pasará a estado **destruido**.

Si queremos hacer que la aplicación termine de ejecutarse desde dentro del código, nunca utilizaremos el método `System.exit` (o `Runtime.exit`), ya que estos métodos se utilizan para salir de la máquina virtual. En este caso, como se trata de un componente, si



ejecutásemos este método cerraríamos toda la aplicación, es decir, el AMS. Por esta razón esto no se permite, si intentásemos hacerlo obtendríamos una excepción de seguridad.

La única forma de salir de una aplicación MIDP es haciendo pasar el componente a estado destruido, como hemos visto en el punto anterior, para que el contenedor pueda eliminarlo. Esto lo haremos invocando `notifyDestroyed` para cambiar el estado a destruido. Sin embargo, si hacemos esto no se invocará automáticamente el método `destroyApp` para liberar los recursos, por lo que deberemos ejecutarlo nosotros manualmente antes de marcar la aplicación como destruida:

```
public void salir() {
    try {
        destroyApp(true);
    } catch(MIDletStateChangeException e) {
    }
    notifyDestroyed();
}
```

Si queremos implementar una salida condicional, para que el método `destroyApp` pueda decidir si permitir que se cierre o no la aplicación, podemos hacerlo de la siguiente forma:

```
public void salir_cond() {
    try {
        destroyApp(false);
        notifyDestroyed();
    } catch(MIDletStateChangeException e) {
    }
}
```

#### 1.3.2.4. Parametrización de los MIDlets

Podemos añadir una serie de propiedades en el fichero descriptor de la aplicación (JAD), que podrán ser leídas desde el MIDlet. De esta forma, podremos cambiar el valor de estas propiedades sin tener que rehacer el fichero JAR.

Cada propiedad consistirá en una clave (*key*) y en un valor. La clave será el nombre de la propiedad. De esta forma tendremos un conjunto de parámetros de configuración (claves) con un valor asignado a cada una. Podremos cambiar fácilmente estos valores editando el fichero JAD con cualquier editor de texto.

Para leer estas propiedades desde el MIDlet utilizaremos el método:

```
String valor = getAppProperty(String key)
```

Que nos devolverá el valor asignado a la clave con nombre *key*.

#### 1.3.2.5. Peticiones al dispositivo

A partir de MIDP 2.0 se incorpora una nueva función que nos permite realizar peticiones

que se encargará de gestionar el dispositivo, de forma externa a nuestra aplicación. Por ejemplo, con esta función podremos realizar una llamada a un número telefónico o abrir el navegador web instalado para mostrar un determinado documento.

Para realizar este tipo de peticiones utilizaremos el siguiente método:

```
boolean debeSalir = platformRequest(url);
```

Esto proporcionará una URL al AMS, que determinará, según el tipo de la URL, qué servicio debe invocar. Además nos devolverá un valor *booleano* que indicará si para que este servicio sea ejecutado debemos cerrar el MIDlet antes. Algunos servicios de determinados dispositivos no pueden ejecutarse concurrentemente con nuestra aplicación, por lo que en estos casos hasta que no la cerremos no se ejecutará el servicio.

Los tipos servicios que se pueden solicitar dependen de las características del móvil en el que se ejecute. Cada fabricante puede ofrecer un serie de servicios accesibles mediante determinados tipos de URLs. Sin intentamos acceder a un servicio que no está disponible en el móvil, se producirá una excepción de tipo `ConnectionNotFoundException`.

En el estándar de MIDP 2.0 sólo se definen URLs para dos tipos de servicios:

- Iniciar una llamada de voz. Para ello se utilizará una URL como la siguiente:

```
tel:<numero>
```

Por ejemplo, podríamos poner:

```
tel:+34-965-123-456.
```

- Instalar una *suite* de MIDlets. Se proporciona la URL donde está el fichero JAD de la suite que queramos instalar. Por ejemplo:

```
http://www.jtech.ua.es/prueba/aplic.jad
```

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

## 2. Java, JavaME y el entorno Eclipse - Ejercicios

### 2.1. ¡Hola ME!

Vamos a hacer nuestra primera aplicación "*Hola mundo*" en J2ME. Para ello debemos:

a) Crear un nuevo proyecto `HolaME` con configuración CLDC1.1 y MIDP2.1. Crear en su carpeta de fuentes un MIDlet principal `es.ua.jtech.javame.holame.MIDletHolaME`.

b) Introduciremos en la clase del MIDlet principal el siguiente código:

```
package es.ua.jtech.javame.holame;

import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Form;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

public class MIDletHolaME extends MIDlet {
    protected void startApp() throws MIDletStateChangeException {
        Form form = new Form(";Hola ME!");
        Display display = Display.getDisplay(this);
        display.setCurrent(form);
    }

    protected void destroyApp(boolean unconditional)
        throws MIDletStateChangeException {
    }

    protected void pauseApp() {
    }
}
```

c) Guardar el fichero y tras comprobar que Eclipse no da ningún error de compilación, ejecutar la aplicación en dos emuladores distintos, comprobando que funciona correctamente.

Busca desde Eclipse la opción del menú contextual que permite crear un paquete. Se generarán dos archivos en la carpeta `deployed`. Uno de ellos es de texto plano, el `.jar`. El otro es un paquete que puedes descomprimir como zip. Explora su estructura. ¿Encuentras un archivo con información similar al `.jar`?

### 2.2. Clases

Vamos a crear en el anterior proyecto `HolaME` un paquete llamado `es.ua.jtech.javame.holame.calculos` y en él vamos a añadir las clases `Factorial` y `EcuacionCuadratica` que vienen en las plantillas de la sesión. Para utilizar sus métodos vamos a introducir al final del método `MIDletHolaME.startApp()` dos formas de imprimir el resultado: en el formulario de la pantalla y por la salida estándar que puede verse en la consola de Eclipse.

```

Factorial factorial = new Factorial();
int f1 = factorial.factorialRec(4);
System.out.println("Factorial = "+f1);
form.append("Factorial = "+f1);

```

Añadid a la clase `Factorial.java` el código necesario para que calcule el factorial de un número. Intentad hacer tanto la versión recursiva como la iterativa

- La versión recursiva (en el método `factorialRec()`) consiste en un método que se llama a sí mismo hasta completar el resultado:  
`factorialRec(n) = n · factorialRec(n - 1)`  
 Cuando `n` sea 0 se devuelve 1 y se termina la recursividad.
- La versión iterativa (en el método `factorialIter()`) consiste en realizar un bucle que vaya acumulando el resultado.

Ahora añadid a la clase `EcuacionCuadratica.java` el código necesario (dentro del método `solucion(...)`) para que resuelva una ecuación de segundo grado  $ax^2 + bx + c = 0$ :

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Raíces de la ecuación

Si no hay solución, debe devolver `null`.

Nota: el método `Math.pow()` no está en JavaME. Tendremos que utilizar el producto en su lugar.

Probar `Factorial.factorialIter()`, `Factorial.factorialRec()` y `EcuacionCuadratica.solucion()` visualizando cadenas con los resultados tanto en el Form como por la salida estándar.

## 2.3. Métodos y campos de la clase

En JavaME no se utiliza el método `main`, se trata de un método que se utiliza más en aplicaciones de escritorio. Una de sus utilidades podría ser introducir algún pequeño test de que la clase funciona, o un ejemplo de uso. En el caso del ejercicio anterior podríamos introducir ejemplos de uso del `Factorial` o de la `EcuacionCuadratica` en sus respectivos métodos `main()`. Para ejecutarlos es suficiente con utilizar la opción "Run as / Java application" de Eclipse. El resultado se verá por la consola (salida estándar).

El método `EcuacionCuadratica.java` incluye dos ejemplos de uso pero uno de ellos, el del método sin parámetros, está comentado. Fíjate que ambos métodos (con parámetros y sin parámetros) se llaman exactamente igual, lo que los diferencia es el número de parámetros. Para que el método sin parámetros funcione necesitamos crear getters y setters para las variables, para ello utiliza el menú "Source / Generate getters and setters"

de Eclipse. Descomenta el código y comprueba que funciona igual. Fíjate en que según el código comentado de `main()`, el getter `getRaices(i)` toma la posición del array como parámetro. Modifica el getter o implementa otro, con esta funcionalidad.

Incluye una tercera forma de uso que, en lugar de usar los setters, utilice directamente el constructor para dar valor a los campos de la clase (las variables). Implementa dicho constructor usando la opción "Generate constructor using fields" del menú Source de Eclipse.

## 2.4. Métodos estáticos

¿Tiene sentido mantener las variables de la ecuación y las soluciones como campos de la clase? Mantener un estado del objeto `EcuacionCuadratica` del ejercicio anterior no tiene mucho sentido, ya que lo normal será que cada vez la utilicemos para resolver una ecuación diferente, y cada siguiente ecuación no depende en nada de la anterior. Así, en ambas clases del ejercicio anterior, sería suficiente con llamar a sus métodos pasándoles los parámetros necesarios y obteniendo como valor de retorno la solución.

Vamos a crear una segunda versión de las clases, duplicándolas desde Eclipse con nombres nuevos: `Factorial2` y `EcuacionCuadratica2`. Convierte en estáticos los métodos que podamos llamar sin depender de los campos de las clases. Elimina los campos de la clase `EcuacionCuadratica2` y el método que depende de ellos.

Cambia los métodos `main()` para que hagan un uso estático de los métodos. ¿Tiene ahora sentido crear objetos de las clases? Implementa un mecanismo que lo prohíba: el constructor por defecto privado. Comprueba que es imposible instanciarlas desde el MIDlet.

## 2.5. Librerías opcionales (\*)

Vamos a añadir sonido a una aplicación JavaME. Para ello deberemos utilizar la API multimedia que es una API adicional. Crearemos un proyecto nuevo llamado `PruebaSonido` y le añadiremos un midlet `MIDletPruebaSonido`. A continuación deberemos:

a) Comprobar si está la librería `MMAPI` en nuestro proyecto en Eclipse. Se puede ver en la pestaña `Libraries` del `Java Build Path` de las preferencias del proyecto.

b) Una vez hecho esto podremos utilizar esta API multimedia en el editor de Eclipse sin que nos muestre errores en el código. Modificaremos el código del MIDlet de la siguiente forma:

```
package es.ua.jtech.javame.pruebasonido;

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

```

public class MIDletPruebaSonido extends MIDlet {

    protected void startApp() throws MIDletStateChangeException {
        Form f = new Form("Prueba de sonido");

        try {
            Manager.playTone(80, 1000, 100);
        } catch(MediaException e) {}

        Display d = Display.getDisplay(this);
        d.setCurrent(f);
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean incondicional)
        throws MIDletStateChangeException {
    }
}

```

c) Guardar y comprobar que la aplicación funciona correctamente.

## 2.6. Temporizadores (\*)

Vamos a incorporar un temporizador a una aplicación. Lo único que haremos será mostrar un mensaje de texto en la consola cuando se dispare el temporizador, por lo que no será una aplicación útil para visualizar en el móvil.

a) En el directorio `Temporizador` de las plantillas de la sesión se encuentra implementado este temporizador. Compilarlo y ejecutarlo.

b) Modificar este temporizador para que en lugar de dispararse pasado cierto intervalo, se dispare a una hora fija. Para ello puedes usar la clase `Date` y ayudarte de la clase `Calendar`.

### 3. Herencia e interfaces. MIDlets e interfaz de usuario

Características básicas del lenguaje Java como la herencia, interfaces y polimorfismo funcionan exactamente igual en JavaME. Sin embargo no todas las colecciones de Java están presentes en dicha plataforma. La ausencia más notable es la de `ArrayList`, en lugar de la que se usa `Vector`.

En cuanto a la interfaz de usuario, la de JavaME es particular y diseñada para las plataformas móviles. Veremos los componentes de alto nivel y también el control de la interfaz a bajo nivel.

#### 3.1. Herencia e interfaces

##### 3.1.1. Herencia

Cuando queremos que una clase herede de otra, se utiliza al declararla la palabra `extends` tras el nombre de la clase, para decir de qué clase se hereda. Para hacer que `Pato` herede de `Animal`:

```
class Pato extends Animal
```

Con esto automáticamente `Pato` tomaría todo lo que tuviese `Animal` (aparte, `Pato` puede añadir sus características propias). Si `Animal` fuese una clase abstracta, `Pato` debería implementar los métodos abstractos que tuviese.

##### 3.1.2. Punteros `this` y `super`

El puntero `this` apunta al objeto en el que nos encontramos. Se utiliza normalmente cuando hay variables locales con el mismo nombre que variables de instancia de nuestro objeto:

```
public class MiClase
{
    int i;
    public MiClase(int i)
    {
        this.i = i;           // i de la clase = parametro i
    }
}
```

También se suele utilizar para remarcar que se está accediendo a variables de instancia.

El puntero `super` se usa para acceder a un elemento en la clase padre. Si la clase `Usuario` tiene un método `getPermisos`, y una subclase `UsuarioAdministrador` sobrescribe dicho método, podríamos llamar al método de la super-clase con:

```
public class UsuarioAdministrador extends Usuario {
    public List<String> getPermisos() {
```

```

        List<String> permisos = super.getPermisos();
        permisos.add(PERMISO_ADMINISTRADOR);
        return permisos;
    }
}

```

También podemos utilizar `this` y `super` como primera instrucción dentro de un constructor para invocar a otros constructores. Dado que toda clase en Java hereda de otra clase, siempre será necesario llamar a alguno de los constructores de la super-clase para que se construya la parte relativa a ella. Por lo tanto, si al comienzo del constructor no se especifica ninguna llamada a `this` o `super`, se considera que hay una llamada implícita al constructor sin parámetros de la super-clase (`super()`). Es decir, los dos constructores siguientes son equivalentes:

```

public Punto2D(int x, int y, String etiq) {
    // Existe una llamada implícita a super()

    this.x = x;
    this.y = y;
    this.etiq = etiq;
}

public Punto2D(int x, int y, String etiq) {
    super();

    this.x = x;
    this.y = y;
    this.etiq = etiq;
}

```

Pero es posible que la super-clase no disponga de un constructor sin parámetros. En ese caso, si no hacemos una llamada explícita a `super` nos dará un error de compilación, ya que estará intentando llamar a un constructor inexistente de forma implícita. Es posible también, que aunque el constructor sin parámetros exista, nos interese llamar a otro constructor a la hora de construir la parte relativa a la super-clase. Imaginemos por ejemplo que la clase `Punto2D` anterior deriva de una clase `PrimitivaGeometrica` que almacena, como información común de todas las primitivas, una etiqueta de texto, y ofrece un constructor que toma como parámetro dicha etiqueta. Podríamos utilizar dicho constructor desde la subclase de la siguiente forma:

```

public Punto2D(int x, int y, String etiq) {
    super(etiq);

    this.x = x;
    this.y = y;
}

```

También puede ocurrir que en lugar de querer llamar directamente al constructor de la super-clase nos interese basar nuestro constructor en otro de los constructores de nuestra misma clase. En tal caso llamaremos a `this` al comienzo de nuestro constructor, pasándole los parámetros correspondientes al constructor en el que queremos basarnos. Por ejemplo, podríamos definir un constructor sin parámetros de nuestra clase `punto`, que se base en el constructor anterior (más específico) para crear un punto con una serie de datos por defecto:



```
public Punto2D() {
    this(DEFAULT_X, DEFAULT_Y, DEFAULT_ETIQ);
}
```

Es importante recalcar que las llamadas a `this` o `super` deben ser siempre la primera instrucción del constructor.

### 3.1.3. Interfaces y clases abstractas

Ya hemos visto cómo definir clases normales, y clases abstractas. Si queremos definir un interfaz, se utiliza la palabra reservada `interface`, en lugar de `class`, y dentro declaramos (no implementamos), los métodos que queremos que tenga la interfaz:

```
public interface MiInterfaz
{
    public void metodoInterfaz();
    public float otroMetodoInterfaz();
}
```

Después, para que una clase implemente los métodos de esta interfaz, se utiliza la palabra reservada `implements` tras el nombre de la clase:

```
public class UnaClase implements MiInterfaz
{
    public void metodoInterfaz()
    {
        ... // Código del método
    }

    public float otroMetodoInterfaz()
    {
        ... // Código del método
    }
}
```

Notar que si en lugar de poner `implements` ponemos `extends`, en ese caso `UnaClase` debería ser un interfaz, que heredaría del interfaz `MiInterfaz` para definir más métodos, pero no para implementar los que tiene la interfaz. Esto se utilizaría para definir interfaces partiendo de un interfaz base, para añadir más métodos a implementar.

Una clase puede heredar sólo de otra única clase, pero puede implementar cuantos interfaces necesite:

```
public class UnaClase extends MiClase
    implements MiInterfaz, MiInterfaz2, MiInterfaz3
{
    ...
}
```

Cuando una clase implementa una interfaz se está asegurando que dicha clase va a ofrecer los métodos definidos en la interfaz, es decir, que la clase al menos nos ofrece esa interfaz para acceder a ella. Cuando heredamos de una clase abstracta, heredamos todos los campos y el comportamiento de la superclase, y además deberemos definir algunos métodos que no habían sido implementados en la superclase.

Desde el punto de vista del diseño, podemos ver la herencia como una relación *ES*, mientras que la implementación de una interfaz sería una relación *ACTÚA COMO*.

## 3.2. Colecciones de datos

---

La plataforma Java nos proporciona un amplio conjunto de clases dentro del que podemos encontrar tipos de datos que nos resultarán muy útiles para realizar la programación de aplicaciones en Java. Estos tipos de datos nos ayudarán a generar código más limpio de una forma sencilla.

Se proporcionan una serie de operadores para acceder a los elementos de estos tipos de datos. Decimos que dichos operadores son *polimórficos*, ya que un mismo operador se puede emplear para acceder a distintos tipos de datos. Por ejemplo, un operador *add* utilizado para añadir un elemento, podrá ser empleado tanto si estamos trabajando con una lista enlazada, con un array, o con un conjunto por ejemplo.

Este *polimorfismo* se debe a la definición de interfaces que deben implementar los distintos tipos de datos. Siempre que el tipo de datos contenga una colección de elementos, implementará la interfaz `Collection`. Esta interfaz proporciona métodos para acceder a la colección de elementos, que podremos utilizar para cualquier tipo de datos que sea una colección de elementos, independientemente de su implementación concreta.

Podemos encontrar los siguientes elementos dentro del marco de colecciones de Java:

- Interfaces para distintos tipos de datos: Definirán las operaciones que se pueden realizar con dichos tipos de datos. Podemos encontrar aquí la interfaz para cualquier colección de datos, y de manera más concreta para listas (secuencias) de datos, conjuntos, etc.
- Implementaciones de tipos de datos reutilizables: Son clases que implementan tipos de datos concretos que podremos utilizar para nuestras aplicaciones, implementando algunas de las interfaces anteriores para acceder a los elementos de dicho tipo de datos. Por ejemplo, dentro de las listas de elementos, podremos encontrar distintas implementaciones de la lista como puede ser listas enlazadas, o bien arrays de capacidad variable, pero al implementar la misma interfaz podremos acceder a sus elementos mediante las mismas operaciones (polimorfismo).
- Algoritmos para trabajar con dichos tipos de datos, que nos permitan realizar una ordenación de los elementos de una lista, o diversos tipos de búsqueda de un determinado elemento por ejemplo.

### 3.2.1. Colecciones

---

Las colecciones representan grupos de objetos, denominados elementos. Podemos encontrar diversos tipos de colecciones, según si sus elementos están ordenados, o si permitimos repetición de elementos o no.

Es el tipo más genérico en cuanto a que se refiere a cualquier tipo que contenga un grupo

de elementos. Viene definido por la interfaz `Collection`, de la cual heredarán cada subtipo específico. En esta interfaz encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea. Estos métodos generales son:

```
boolean add(Object o)
```

Añade un elemento (objeto) a la colección. Nos devuelve *true* si tras añadir el elemento la colección ha cambiado, es decir, el elemento se ha añadido correctamente, o *false* en caso contrario.

```
void clear()
```

Elimina todos los elementos de la colección.

```
boolean contains(Object o)
```

Indica si la colección contiene el elemento (objeto) indicado.

```
boolean isEmpty()
```

Indica si la colección está vacía (no tiene ningún elemento).

```
Iterator iterator()
```

Proporciona un iterador para acceder a los elementos de la colección.

```
boolean remove(Object o)
```

Elimina un determinado elemento (objeto) de la colección, devolviendo *true* si dicho elemento estaba contenido en la colección, y *false* en caso contrario.

```
int size()
```

Nos devuelve el número de elementos que contiene la colección.

```
Object [] toArray()
```

Nos devuelve la colección de elementos como un array de objetos. Si sabemos de antemano que los objetos de la colección son todos de un determinado tipo (como por ejemplo de tipo `String`) podremos obtenerlos en un array del tipo adecuado, en lugar de usar un array de objetos genéricos. En este caso NO podremos hacer una conversión cast descendente de array de objetos a array de un tipo más concreto, ya que el array se habrá instanciado simplemente como array de objetos:

```
// Esto no se puede hacer!!!  
String [] cadenas = (String []) coleccion.toArray();
```

Lo que si podemos hacer es instanciar nosotros un array del tipo adecuado y hacer una conversión cast ascendente (de tipo concreto a array de objetos), y utilizar el siguiente método:

```
String [] cadenas = new String[coleccion.size()];  
coleccion.toArray(cadenas); // Esto si que funcionará
```

Esta interfaz es muy genérica, y por lo tanto no hay ningún tipo de datos que la implemente directamente, sino que implementarán subtipos de ellas. A continuación veremos los subtipos más comunes.

### 3.2.1.1. Listas de elementos

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, existe una secuencia de elementos. En ellas cada elemento estará en una determinada posición (índice) de la lista.

Las listas vienen definidas en la interfaz `List`, que además de los métodos generales de las colecciones, nos ofrece los siguientes para trabajar con los índices:

```
void add(int indice, Object obj)
```

Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.

```
Object get(int indice)
```

Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.

```
int indexOf(Object obj)
```

Nos dice cual es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.

```
Object remove(int indice)
```

Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.

```
Object set(int indice, Object obj)
```

Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

Podemos encontrar diferentes implementaciones de listas de elementos en Java:

#### **ArrayList**

Implementa una lista de elementos mediante un array de tamaño variable. Conforme se añaden elementos el tamaño del array irá creciendo si es necesario. El array tendrá una capacidad inicial, y en el momento en el que se rebase dicha capacidad, se aumentará el tamaño del array.

Las operaciones de añadir un elemento al final del array (*add*), y de establecer u obtener el elemento en una determinada posición (*get/set*) tienen un coste temporal constante. Las inserciones y borrados tienen un coste lineal  $O(n)$ , donde  $n$  es el número de elementos del array.

Hemos de destacar que la implementación de `ArrayList` no está sincronizada, es decir, si múltiples hilos acceden a un mismo `ArrayList` concurrentemente podríamos tener problemas en la consistencia de los datos. Por lo tanto, deberemos tener en cuenta cuando usemos este tipo de datos que debemos controlar la concurrencia de acceso. También podemos hacer que sea sincronizado como veremos más adelante.

## Vector

El `Vector` es una implementación similar al `ArrayList`, con la diferencia de que el `Vector` si que **está sincronizado**. Este es un caso especial, ya que la implementación básica del resto de tipos de datos no está sincronizada.

Esta clase existe desde las primeras versiones de Java, en las que no existía el marco de las colecciones descrito anteriormente. En las últimas versiones el `Vector` se ha acomodado a este marco implementando la interfaz `List`.

Sin embargo, si trabajamos con versiones previas de JDK, hemos de tener en cuenta que dicha interfaz no existía, y por lo tanto esta versión previa del vector no contará con los métodos definidos en ella. Los métodos propios del vector para acceder a su contenido, que han existido desde las primeras versiones, son los siguientes:

```
void addElement(Object obj)
```

Añade un elemento al final del vector.

```
Object elementAt(int indice)
```

Devuelve el elemento de la posición del vector indicada por el índice.

```
void insertElementAt(Object obj, int indice)
```

Inserta un elemento en la posición indicada.

```
boolean removeElement(Object obj)
```

Elimina el elemento indicado del vector, devolviendo *true* si dicho elemento estaba contenido en el vector, y *false* en caso contrario.

```
void removeElementAt(int indice)
```

Elimina el elemento de la posición indicada en el índice.

```
void setElementAt(Object obj, int indice)
```

Sobrescribe el elemento de la posición indicada con el objeto especificado.

```
int size()
```

Devuelve el número de elementos del vector.

Por lo tanto, si programamos para versiones antiguas de la máquina virtual Java, será recomendable utilizar estos métodos para asegurarnos de que nuestro programa funcione. Esto será importante en la programación de Applets, ya que la máquina virtual incluida en

muchos navegadores corresponde a versiones antiguas.

Sobre el vector se construye el tipo pila (*Stack*), que apoyándose en el tipo vector ofrece métodos para trabajar con dicho vector como si se tratase de una pila, apilando y desapilando elementos (operaciones *push* y *pop* respectivamente). La clase *Stack* hereda de *Vector*, por lo que en realidad será un vector que ofrece métodos adicionales para trabajar con él como si fuese una pila.

## LinkedList

En este caso se implementa la lista mediante una lista doblemente enlazada. Por lo tanto, el coste temporal de las operaciones será el de este tipo de listas. Cuando realicemos inserciones, borrados o lecturas en los extremos inicial o final de la lista el tiempo será constante, mientras que para cualquier operación en la que necesitemos localizar un determinado índice dentro de la lista deberemos recorrer la lista de inicio a fin, por lo que el coste será lineal con el tamaño de la lista  $O(n)$ , siendo  $n$  el tamaño de la lista.

Para aprovechar las ventajas que tenemos en el coste temporal al trabajar con los extremos de la lista, se proporcionan métodos propios para acceder a ellos en tiempo constante:

```
void addFirst(Object obj) / void addLast(Object obj)
```

Añade el objeto indicado al principio / final de la lista respectivamente.

```
Object getFirst() / Object getLast()
```

Obtiene el primer / último objeto de la lista respectivamente.

```
Object removeFirst() / Object removeLast()
```

Extrae el primer / último elemento de la lista respectivamente, devolviéndonos dicho objeto y eliminándolo de la lista.

Hemos de destacar que estos métodos nos permitirán trabajar con la lista como si se tratase de una pila o de una cola. En el caso de la pila realizaremos la inserción y la extracción de elementos por el mismo extremo, mientras que para la cola insertaremos por un extremo y extraeremos por el otro.

### 3.2.1.2. Conjuntos

Los conjuntos son grupos de elementos en los que no encontramos ningún elemento repetido. Consideramos que un elemento está repetido si tenemos dos objetos *o1* y *o2* iguales, comparándolos mediante el operador *o1.equals(o2)*. De esta forma, si el objeto a insertar en el conjunto estuviese repetido, no nos dejará insertarlo. Recordemos que el método *add* devolvía un valor *booleano*, que servirá para este caso, devolviéndonos *true* si el elemento a añadir no estaba en el conjunto y ha sido añadido, o *false* si el elemento ya se encontraba dentro del conjunto. Un conjunto podrá contener a lo sumo un elemento *null*.

Los conjuntos se definen en la interfaz `Set`, a partir de la cuál se construyen diferentes implementaciones:

### HashSet

Los objetos se almacenan en una tabla de dispersión (*hash*). El coste de las operaciones básicas (inserción, borrado, búsqueda) se realizan en tiempo constante siempre que los elementos se hayan dispersado de forma adecuada. La iteración a través de sus elementos es más costosa, ya que necesitará recorrer todas las entradas de la tabla de dispersión, lo que hará que el coste esté en función tanto del número de elementos insertados en el conjunto como del número de entradas de la tabla. El orden de iteración puede diferir del orden en el que se insertaron los elementos.

### LinkedHashSet

Es similar a la anterior pero la tabla de dispersión es doblemente enlazada. Los elementos que se inserten tendrán enlaces entre ellos. Por lo tanto, las operaciones básicas seguirán teniendo coste constante, con la carga adicional que supone tener que gestionar los enlaces. Sin embargo habrá una mejora en la iteración, ya que al establecerse enlaces entre los elementos no tendremos que recorrer todas las entradas de la tabla, el coste sólo estará en función del número de elementos insertados. En este caso, al haber enlaces entre los elementos, estos enlaces definirán el orden en el que se insertaron en el conjunto, por lo que el orden de iteración será el mismo orden en el que se insertaron.

### TreeSet

Utiliza un árbol para el almacenamiento de los elementos. Por lo tanto, el coste para realizar las operaciones básicas será logarítmico con el número de elementos que tenga el conjunto  $O(\log n)$ .

#### 3.2.1.3. Mapas

---

Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que no heredan de la interfaz `Collection`.

Los mapas se definen en la interfaz `Map`. Un mapa es un objeto que relaciona una clave (*key*) con un valor. Contendrá un conjunto de claves, y a cada clave se le asociará un determinado valor. En versiones anteriores este mapeado entre claves y valores lo hacía la clase `Dictionary`, que ha quedado obsoleta. Tanto la clave como el valor puede ser cualquier objeto.

Los métodos básicos para trabajar con estos elementos son los siguientes:

```
Object get(Object clave)
```

Nos devuelve el valor asociado a la clave indicada

```
Object put(Object clave, Object valor)
```

Inserta una nueva clave con el valor especificado. Nos devuelve el valor que tenía antes dicha clave, o *null* si la clave no estaba en la tabla todavía.

```
Object remove(Object clave)
```

Elimina una clave, devolviendonos el valor que tenía dicha clave.

```
Set keySet()
```

Nos devuelve el conjunto de claves registradas

```
int size()
```

Nos devuelve el número de parejas (clave,valor) registradas.

Encontramos distintas implementaciones de los mapas:

### HashMap

Utiliza una tabla de dispersión para almacenar la información del mapa. Las operaciones básicas (*get* y *put*) se harán en tiempo constante siempre que se dispersen adecuadamente los elementos. Es coste de la iteración dependerá del número de entradas de la tabla y del número de elementos del mapa. No se garantiza que se respete el orden de las claves.

### TreeMap

Utiliza un árbol rojo-negro para implementar el mapa. El coste de las operaciones básicas será logarítmico con el número de elementos del mapa  $O(\log n)$ . En este caso los elementos se encontrarán ordenados por orden ascendente de clave.

### Hashtable

Es una implementación similar a `HashMap`, pero con alguna diferencia. Mientras las anteriores implementaciones no están sincronizadas, esta sí que lo está. Además en esta implementación, al contrario que las anteriores, no se permitirán claves nulas (*null*). Este objeto extiende la obsoleta clase `Dictionary`, ya que viene de versiones más antiguas de JDK. Ofrece otros métodos además de los anteriores, como por ejemplo el siguiente:

```
Enumeration keys()
```

Este método nos devolverá una enumeración de todas las claves registradas en la tabla.

#### 3.2.1.4. Wrappers

La clase `Collections` aporta una serie métodos para cambiar ciertas propiedades de las listas. Estos métodos nos proporcionan los denominados *wrappers* de los distintos tipos de colecciones. Estos *wrappers* son objetos que 'envuelven' al objeto de nuestra colección, pudiendo de esta forma hacer que la colección esté sincronizada, o que la colección pase a ser de solo lectura.

Como dijimos anteriormente, todos los tipos de colecciones no están sincronizados,



excepto el `Vector` que es un caso especial. Al no estar sincronizados, si múltiples hilos utilizan la colección concurrentemente, podrán estar ejecutándose simultáneamente varios métodos de una misma colección que realicen diferentes operaciones sobre ella. Esto puede provocar inconsistencias en los datos. A continuación veremos un posible ejemplo de inconsistencia que se podría producir:

1. Tenemos un `ArrayList` de nombre *letras* formada por los siguiente elementos: [ "A", "B", "C", "D" ]
2. Imaginemos que un hilo de baja prioridad desea eliminar el objeto "C". Para ello hará una llamada al método *letras.remove("C")*.
3. Dentro de este método primero deberá determinar cuál es el índice de dicho objeto dentro del array, para después pasar a eliminarlo.
4. Se encuentra el objeto "C" en el índice 2 del array (recordemos que se empieza a numerar desde 0).
5. El problema viene en este momento. Imaginemos que justo en este momento se le asigna el procesador a un hilo de mayor prioridad, que se encarga de eliminar el elemento "A" del array, quedándose el array de la siguiente forma: [ "B", "C", "D" ]
6. Ahora el hilo de mayor prioridad es sacado del procesador y nuestro hilo sigue ejecutándose desde el punto en el que se quedó.
7. Ahora nuestro hilo lo único que tiene que hacer es eliminar el elemento del índice que había determinado, que resulta ser ¡el índice 2!. Ahora el índice 2 está ocupado por el objeto "D", y por lo tanto será dicho objeto el que se elimine.

Podemos ver que haciendo una llamada a *letras.remove("C")*, al final se ha eliminado el objeto "D", lo cual produce una inconsistencia de los datos con las operaciones realizadas, debido al acceso concurrente.

Este problema lo evitaremos sincronizando la colección. Cuando una colección está sincronizada, hasta que no termine de realizarse una operación (inserciones, borrados, etc), no se podrá ejecutar otra, lo cual evitará estos problemas.

Podemos conseguir que las operaciones se ejecuten de forma sincronizada envolviendo nuestro objeto de la colección con un *wrapper*, que será un objeto que utilice internamente nuestra colección encargándose de realizar la sincronización cuando llamemos a sus métodos. Para obtener estos *wrappers* utilizaremos los siguientes métodos estáticos de `Collections`:

```
Collection synchronizedCollection(Collection c)
List synchronizedList(List l)
Set synchronizedSet(Set s)
Map synchronizedMap(Map m)
SortedSet synchronizedSortedSet(SortedSet ss)
SortedMap synchronizedSortedMap(SortedMap sm)
```

Como vemos tenemos un método para envolver cada tipo de datos. Nos devolverá un objeto con la misma interfaz, por lo que podremos trabajar con él de la misma forma, sin embargo la implementación interna estará sincronizada.

Podemos encontrar también una serie de *wrappers* para obtener versiones de sólo lectura

de nuestras colecciones. Se obtienen con los siguientes métodos:

```
Collection unmodifiableCollection(Collection c)
List unmodifiableList(List l)
Set unmodifiableSet(Set s)
Map unmodifiableMap(Map m)
SortedSet unmodifiableSortedSet(SortedSet ss)
SortedMap unmodifiableSortedMap(SortedMap sm)
```

### 3.2.1.5. Genéricos

Podemos tener colecciones de tipos concretos de datos, lo que permite asegurar que los datos que se van a almacenar van a ser compatibles con un determinado tipo o tipos. Por ejemplo, podemos crear un `ArrayList` que sólo almacene `Strings`, o una `HashMap` que tome como claves `Integers` y como valores `ArrayLists`. Además, con esto nos ahorramos las conversiones *cast* al tipo que deseemos, puesto que la colección ya se asume que será de dicho tipo.

#### Ejemplo

```
// Vector de cadenas
ArrayList<String> a = new ArrayList<String>();
a.add("Hola");
String s = a.get(0);
a.add(new Integer(20)); // Daría error!!

// HashMap con claves enteras y valores de vectores
HashMap<Integer, ArrayList> hm = new HashMap<Integer,
ArrayList>();
hm.put(1, a);
ArrayList a2 = hm.get(1);
```

A partir de JDK 1.5 deberemos utilizar genéricos siempre que sea posible. Si creamos una colección sin especificar el tipo de datos que contendrá normalmente obtendremos un *warning*.

Los genéricos no son una característica exclusiva de las colecciones, sino que se pueden utilizar en muchas otras clases, incluso podemos parametrizar de esta forma nuestras propias clases.

### 3.2.1.6. Recorrer las colecciones

Vamos a ver ahora como podemos iterar por los elementos de una colección de forma eficiente y segura, evitando salirnos del rango de datos. Dos elementos utilizados comunmente para ello son las enumeraciones y los iteradores.

Las enumeraciones, definidas mediante la interfaz `Enumeration`, nos permiten consultar los elementos que contiene una colección de datos. Muchos métodos de clases Java que deben devolver múltiples valores, lo que hacen es devolvernos una enumeración que podremos consultar mediante los métodos que ofrece dicha interfaz.

La enumeración irá recorriendo secuencialmente los elementos de la colección. Para leer

cada elemento de la enumeración deberemos llamar al método:

```
Object item = enum.nextElement();
```

Que nos proporcionará en cada momento el siguiente elemento de la enumeración a leer. Además necesitaremos saber si quedan elementos por leer, para ello tenemos el método:

```
enum.hasMoreElements()
```

Normalmente, el bucle para la lectura de una enumeración será el siguiente:

```
while (enum.hasMoreElements()) {  
    Object item = enum.nextElement();  
    // Hacer algo con el item leído  
}
```

Vemos como en este bucle se van leyendo y procesando elementos de la enumeración uno a uno mientras queden elementos por leer en ella.

Otro elemento para acceder a los datos de una colección son los iteradores. La diferencia está en que los iteradores además de leer los datos nos permitirán eliminarlos de la colección. Los iteradores se definen mediante la interfaz `Iterator`, que proporciona de forma análoga a la enumeración el método:

```
Object item = iter.next();
```

Que nos devuelve el siguiente elemento a leer por el iterador, y para saber si quedan más elementos que leer tenemos el método:

```
iter.hasNext()
```

Además, podemos borrar el último elemento que hayamos leído. Para ello tendremos el método:

```
iter.remove();
```

Por ejemplo, podemos recorrer todos los elementos de una colección utilizando un iterador y eliminar aquellos que cumplan ciertas condiciones:

```
while (iter.hasNext())  
{  
    Object item = iter.next();  
    if(condicion_borrado(item))  
        iter.remove();  
}
```

Las enumeraciones y los iteradores no son tipos de datos, sino elementos que nos servirán para acceder a los elementos dentro de los diferentes tipos de colecciones.

A partir de JDK 1.5 podemos recorrer colecciones y arrays sin necesidad de acceder a sus iteradores, previniendo índices fuera de rango.

## Ejemplo

```
// Recorre e imprime todos los elementos de un array
```

```
int[] arrayInt = {1, 20, 30, 2, 3, 5};
for(int elemento: arrayInt)
    System.out.println (elemento);

// Recorre e imprime todos los elementos de un ArrayList
ArrayList<String> a = new ArrayList<String>();
for(String cadena: a)
    System.out.println (cadena);
```

### 3.2.2. Polimorfismo e interfaces

En Java podemos conseguir tener objetos polimórficos mediante la implementación de interfaces. Un claro ejemplo está en las colecciones vistas anteriormente. Por ejemplo, todos los tipos de listas implementan la interfaz `List`. De esta forma, en un método que acepte como entrada un objeto de tipo `List` podremos utilizar cualquier tipo que implemente esta interfaz, independientemente del tipo concreto del que se trate.

Es por lo tanto recomendable hacer referencia siempre a estos objetos mediante la interfaz que implementa, y no por su tipo concreto. De esta forma posteriormente podríamos cambiar la implementación del tipo de datos sin que afecte al resto del programa. Lo único que tendremos que cambiar es el momento en el que se instancia.

Por ejemplo, si tenemos una clase `Cliente` que contiene una serie de cuentas, tendremos algo como:

```
public class Cliente {
    String nombre;
    List<Cuenta> cuentas;

    public Cliente(String nombre) {
        this.nombre = nombre;
        this.cuentas = new ArrayList<Cuenta>();
    }

    public List<Cuenta> getCuentas() {
        return cuentas;
    }

    public void setCuentas(List<Cuenta> cuentas) {
        this.cuentas = cuentas;
    }

    public void addCuenta(Cuenta cuenta) {
        this.cuentas.add(cuenta);
    }
}
```

Si posteriormente queremos cambiar la implementación de la lista a `LinkedList` por ejemplo, sólo tendremos que cambiar la línea del constructor en la que se hace la instanciación.

Como ejemplo de la utilidad que tiene el polimorfismo podemos ver los algoritmos predefinidos con los que contamos en el marco de colecciones.

#### 3.2.2.1. Ejemplo: Algoritmos

---

Como hemos comentado anteriormente, además de las interfaces y las implementaciones de los tipos de datos descritos en los apartados previos, el marco de colecciones nos ofrece una serie de algoritmos útiles cuando trabajamos con estos tipos de datos, especialmente para las listas.

Estos algoritmos los podemos encontrar implementados como métodos estáticos en la clase `Collections`. En ella encontramos métodos para la ordenación de listas (*sort*), para la búsqueda binaria de elementos dentro de una lista (*binarySearch*) y otras operaciones que nos serán de gran utilidad cuando trabajemos con colecciones de elementos.

Estos métodos tienen como parámetro de entrada un objeto de tipo `List`. De esta forma, podremos utilizar estos algoritmos para cualquier tipo de lista.

### 3.2.3. Tipos de datos básicos en las colecciones

---

#### 3.2.3.1. Wrappers de tipos básicos

---

Hemos visto que en Java cualquier tipo de datos es un objeto, excepto los tipos de datos básicos: *boolean*, *int*, *long*, *float*, *double*, *byte*, *short*, *char*.

Cuando trabajamos con colecciones de datos los elementos que contienen éstas son siempre objetos, por lo que en un principio no podríamos insertar elementos de estos tipos básicos. Para hacer esto posible tenemos una serie de objetos que se encargarán de envolver a estos tipos básicos, permitiéndonos tratarlos como objetos y por lo tanto insertarlos como elementos de colecciones. Estos objetos son los llamados wrappers, y las clases en las que se definen tienen nombre similares al del tipo básico que encapsulan, con la diferencia de que comienzan con mayúscula: *Boolean*, *Integer*, *Long*, *Float*, *Double*, *Byte*, *Short*, *Character*.

Estas clases, además de servirnos para encapsular estos datos básicos en forma de objetos, nos proporcionan una serie de métodos e información útiles para trabajar con estos datos. Nos proporcionarán métodos por ejemplo para convertir cadenas a datos numéricos de distintos tipos y viceversa, así como información acerca del valor mínimo y máximo que se puede representar con cada tipo numérico.

#### 3.2.3.2. Autoboxing

---

Esta característica aparecida en JDK 1.5 evita al programador tener que establecer correspondencias manuales entre los tipos simples (*int*, *double*, etc) y sus correspondientes *wrappers* o tipos complejos (*Integer*, *Double*, etc). Podremos utilizar un *int* donde se espere un objeto complejo (*Integer*), y viceversa.

#### Ejemplo

```
ArrayList<Integer> a = new ArrayList<Integer>();  
a.add(30);
```

```
Integer n = v.get(0);
n = n+1;
int num = n;
```

### 3.3. MIDlets e interfaz de usuario

Vamos a ver ahora como crear la interfaz de las aplicaciones MIDP. En la reducida pantalla de los móviles no tendremos una consola en la que imprimir utilizando la salida estándar, por lo que toda la salida la tendremos que mostrar utilizando una API propia que nos permita crear componentes adecuados para ser mostrados en este tipo de pantallas.

Esta API propia para crear la interfaz gráfica de usuario de los MIDlets se denomina LCDUI (*Limited Connected Devices User Interface*), y se encuentra en el paquete `javax.microedition.lcdui`.

#### 3.3.1. Acceso al visor

El visor del dispositivo está representado por un objeto `Display`. Este objeto nos permitirá acceder a este visor y a los dispositivos de entrada (normalmente el teclado) del móvil.

Tendremos asociado un único *display* a cada aplicación (MIDlet). Para obtener el *display* asociado a nuestro MIDlet deberemos utilizar el siguiente método estático:

```
Display mi_display = Display.getDisplay(mi_midlet);
```

Donde *mi\_midlet* será una referencia al MIDlet del cual queremos obtener el `Display`. Podremos acceder a este *display* desde el momento en que `startApp` es invocado por primera vez (no podremos hacerlo en el constructor del MIDlet), y una vez se haya terminado de ejecutar `destroyApp` ya no podremos volver a acceder al *display* del MIDlet.

Cada MIDlet tiene un *display* y sólo uno. Si el MIDlet ha pasado a segundo plano (pausado), seguirá asociado al mismo *display*, pero en ese momento no se mostrará su contenido en la pantalla del dispositivo ni será capaz de leer las teclas que pulse el usuario.

Podemos utilizar este objeto para obtener propiedades del visor como el número de colores que soporta:

```
boolean color = mi_display.isColor();
int num_color = mi_display.numColors();
```

#### 3.3.2. Componentes disponibles

Una vez hemos accedido al *display*, deberemos mostrar algo en él. Tenemos una serie de elementos que podemos mostrar en el *display*, estos son conocidos como elementos *displayables*.

En el *display* podremos mostrar a lo sumo un elemento *displayable*. Para obtener el elemento que se está mostrando actualmente en el visor utilizaremos el siguiente método:

```
Displayable elemento = mi_display.getCurrent();
```

Nos devolverá el objeto *Displayable* correspondiente al objeto que se está mostrando en la pantalla, o *null* en el caso de que no se esté mostrando ningún elemento. Esto ocurrirá al comienzo de la ejecución de la aplicación cuando todavía no se ha asignado ningún elemento al *Display*. Podemos establecer el elemento que queremos mostrar en pantalla con:

```
mi_display.setCurrent(nuevo_elemento);
```

Como sólo podemos mostrar simultáneamente un elemento *displayable* en el *display*, este elemento ocupará todo el visor. Además será este elemento el que recibirá la entrada del usuario.

Entre estos elementos *displayables* podemos distinguir una API de bajo nivel, y una API de alto nivel.

#### 3.3.2.1. API de alto nivel

---

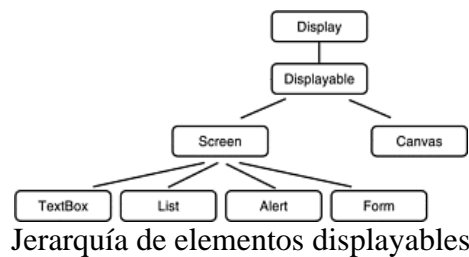
Consiste en una serie de elementos predefinidos: *Form*, *List*, *Alert* y *TextBox* que son extensiones de la clase abstracta *Screen*. Estos son elementos comunes que podemos encontrar en la interfaz de todos los dispositivos, por lo que el tenerlos predefinidos nos permitirá utilizarlos de forma sencilla sin tenerlos que crear nosotros a mano en nuestras aplicaciones. Se implementan de forma nativa por cada dispositivo concreto, por lo que pueden variar de unos dispositivos a otros. Estos componentes hacen que las aplicaciones sean más sencillas y portables, pero nos limita a una serie de controles predefinidos.

Este tipo de componentes serán adecuados para realizar *front-ends* de aplicaciones corporativas. De esta forma obtendremos aplicaciones totalmente portables, en las que la implementación nativa será la que se deberá encargar de dibujar estos componentes. Por lo tanto, en cada dispositivo podrán mostrarse de una forma distinta. Además no se permitirá acceder directamente a los eventos de entrada del teclado.

#### 3.3.2.2. API de bajo nivel

---

Consiste en la clase *Canvas*, que nos permitirá dibujar lo que queramos en la pantalla. Tendremos que dibujarlo todo nosotros a mano. Esto nos permitirá tener un mayor control sobre lo que dibujamos, y podremos recibir eventos del teclado a bajo nivel. Esto provocará que las aplicaciones sean menos portables. Esta API será conveniente para las aplicaciones que necesitan tener control total sobre lo que se dibuja y sobre la entrada, como por ejemplo los juegos.



### 3.3.3. Componentes de alto nivel

Todos los componentes de alto nivel derivan de la clase `Screen`. Se llama así debido a que cada uno de estos componentes será una pantalla de nuestra aplicación, ya que no puede haber más de un componente en la pantalla al mismo tiempo. Esta clase contiene las propiedades comunes a todos los elementos de alto nivel:

**Título:** Es el título que se mostrará en la pantalla correspondiente al componente. Podemos leer o asignar el título con los métodos:

```
String titulo = componente.getTitle();
componente.setTitle(titulo);
```

**Ticker:** Podemos mostrar un *ticker* en la pantalla. El *ticker* consiste en un texto que irá desplazándose de derecha a izquierda. Podemos asignar o obtener el *ticker* con:

```
Ticker ticker = componente.getTicker();
componente.setTicker(ticker);
```

A continuación podemos ver cómo se muestra el título y el *ticker* en distintos modelos de móviles:



Los componentes de alto nivel disponibles son cuadros de texto (`TextBox`), listas (`List`), formularios (`Form`) y alertas (`Alert`).

#### 3.3.3.1. Cuadros de texto

Este componente muestra un cuadro donde el usuario puede introducir texto. La forma en la que se introduce el texto es dependiente del dispositivo. Por ejemplo, los teléfonos que soporten texto predictivo podrán introducir texto de esta forma. Esto se hace de forma



totalmente nativa, por lo que desde Java no podremos modificar este método de introducción del texto.

Para crear un campo de texto deberemos crear un objeto de la clase `TextBox`, utilizando el siguiente constructor:

```
TextBox tb = new TextBox(titulo, texto, capacidad, restricciones);
```

Donde `titulo` será el título que se mostrará en la pantalla, `texto` será el texto que se muestre inicialmente dentro del cuadro, y `capacidad` será el número de caracteres máximo que puede tener el texto. Además podemos añadir una serie de restricciones, definidas como constantes de la clase `TextField`, que limitarán el tipo de texto que se permita escribir en el cuadro. Puede tomar los siguientes valores:

<code>TextField.ANY</code>	Cualquier texto
<code>TextField.NUMERIC</code>	Números enteros
<code>TextField.PHONENUMBER</code>	Números de teléfono
<code>TextField.EMAILADDR</code>	Direcciones de e-mail
<code>TextField.URL</code>	URLs
<code>TextField.PASSWORD</code>	Se ocultan los caracteres escritos utilizando, por ejemplo utilizando asteriscos (*). Puede combinarse con los valores anteriores utilizando el operador OR ( ).

Una vez creado, para que se muestre en la pantalla debemos establecerlo como el componente actual del *display*:

```
mi_display.setCurrent(tb);
```

Una vez hecho esto este será el componente que se muestre en el *display*, y el que recibirá los eventos y comandos de entrada, de forma que cuando el usuario escriba utilizando el teclado del móvil estará escribiendo en este cuadro de texto.

Podemos obtener el texto que haya escrito el usuario en este cuadro de texto utilizando el método:

```
String texto = tb.getString();
```

Esto lo haremos cuando ocurra un determinado evento, que nos indique que el usuario ya ha introducido el texto, como por ejemplo cuando pulse sobre la opción OK.

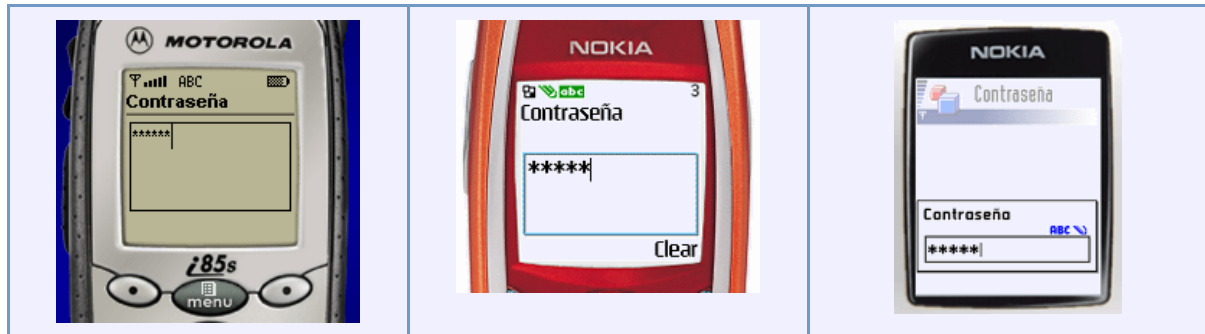
Además tiene métodos con los que podremos modificar el contenido del cuadro de texto, insertando, modificando o borrando caracteres o bien cambiando todo el texto, así como para obtener información sobre el mismo, como el número de caracteres que se han escrito, la capacidad máxima o las restricciones impuestas.

Por ejemplo, podemos crear y mostrar un campo de texto para introducir una contraseña

de 8 caracteres de la siguiente forma:

```
TextBox tb = new TextBox("Contraseña", "", 8,
                          TextField.ANY | TextField.PASSWORD);
Display d = Display.getDisplay(this);
d.setCurrent(tb);
```

El aspecto que mostrará esta pantalla en distintos modelos de móviles será el siguiente:



### 3.3.3.2. Listas

Este componente muestra una lista de elementos en la pantalla. Las listas pueden ser de distintos tipos:

- **Implícita:** Este tipo de listas nos servirán por ejemplo para hacer menús. Cuando pulsemos sobre un elemento de la lista se le notificará inmediatamente a la aplicación el elemento sobre el que hemos pulsado, para que ésta pueda realizar la acción correspondiente.
- **Exclusiva:** A diferencia de la anterior, en esta lista cuando se pulsa sobre un elemento no se notifica a la aplicación, sino que simplemente lo que hace es marcar el elemento como seleccionado. En esta lista podremos tener sólo un elemento marcado, si previamente ya tuviésemos uno marcado, cuando pulsemos sobre uno nuevo se desmarcará el anterior.
- **Múltiple:** Es similar a la exclusiva, pero podemos marcar varios elementos simultáneamente. Pulsando sobre un elemento lo marcaremos o lo desmarcaremos, pudiendo de esta forma marcar tantos como queramos.

Las listas se definen mediante la clase `List`, y para crear una lista podemos utilizar el siguiente constructor:

```
List l = new List(titulo, tipo);
```

Donde `titulo` será el título de la pantalla correspondiente a nuestra lista, y `tipo` será uno de los tipos vistos anteriormente, definidos como constantes de la clase `Choice`:

<code>Choice.IMPLICIT</code>	Lista implícita
<code>Choice.EXCLUSIVE</code>	Lista exclusiva
<code>Choice.MULTIPLE</code>	Lista múltiple

También tenemos otro constructor en el que podemos especificar un *array* de elementos a mostrar en la lista, para añadir toda esa lista de elementos en el momento de su construcción. Si no lo hacemos en este momento, podremos añadir elementos posteriormente utilizando el método:

```
l.append(texto, imagen);
```

Donde *texto* será la cadena de texto que se muestre, e *imagen* será una imagen que podremos poner a dicho elemento de la lista de forma opcional. Si no queremos poner ninguna imagen podemos especificar *null*.

Podremos conocer desde el código los elementos que están marcados en la lista en un momento dado. También tendremos métodos para insertar, modificar o borrar elementos de la lista, así como para marcarlos o desmarcarlos.

Por ejemplo, podemos crear un menú para nuestra aplicación de la siguiente forma:

```
List l = new List("Menu", Choice.IMPLICIT);
l.append("Nuevo juego", null);
l.append("Continuar", null);
l.append("Instrucciones", null);
l.append("Hi-score", null);
l.append("Salir", null);
Display d = Display.getDisplay(this);
d.setCurrent(l);
```

A continuación se muestra el aspecto de los distintos tipos de listas existentes:



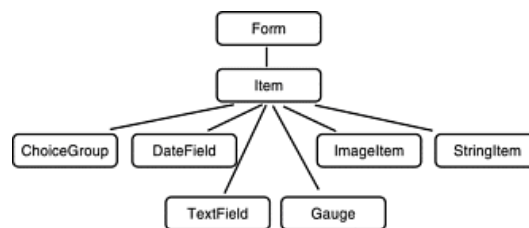
### 3.3.3.3. Formularios

Este componente es más complejo, permitiéndonos mostrar varios elementos en una misma pantalla. Los formularios se encapsulan en la clase *Form*, y los elementos que podemos incluir en ellos son todos derivados de la clase *Item*. Tenemos disponibles los siguientes elementos:

- **Etiquetas** (*StringItem*): Muestra una etiqueta de texto estático, es decir, que no podrá ser modificado por el usuario. Se compone de un título del campo y de un texto como contenido.
- **Imágenes** (*ImageItem*): Muestra una imagen en el formulario. Esta imagen también

es estática. Se compone de un título, la imagen, y un texto alternativo en el caso de que el dispositivo no pueda mostrar imágenes.

- **Campo de texto** (TextField): Muestra un cuadro donde el usuario podrá introducir texto. Se trabaja con él de forma similar al componente `TextBox` visto anteriormente.
- **Campo de fecha** (DateField): Permite al usuario introducir una fecha. La forma de introducir la fecha variará de un modelo de móvil a otro. Por ejemplo, puede introducirse directamente introduciendo numéricamente la fecha, o mostrar un calendario donde el usuario pueda seleccionar el día.
- **Cuadro de opciones** (ChoiceGroup): Muestra un grupo de opciones para que el usuario marque una o varias de ellas. Se trabaja con él de forma similar al componente `List` visto anteriormente, pudiendo en este caso ser de tipo exclusivo o múltiple.
- **Barra de nivel** (Gauge): Muestra una barra para seleccionar un nivel, como por ejemplo podría ser el nivel de volumen. Cada posición de esta barra corresponderá a un valor entero. Este valor irá de cero a un valor máximo que podremos especificar nosotros. La barra podrá ser interactiva o fija.



Jerarquía de los elementos de los formularios

Para crear el formulario podemos utilizar el siguiente constructor, en el que especificamos el título de la pantalla:

```
Form f = new Form(titulo);
```

También podemos crear el formulario proporcionando el *array* de elementos (items) que tiene en el constructor. Si no lo hemos hecho en el constructor, podemos añadir items al formulario con:

```
f.append(item);
```

Podremos añadir como item o bien cualquiera de los items vistos anteriormente, derivados de la clase `Item`, o una cadena de texto o una imagen. También podremos insertar, modificar o borrar los items del formulario.

A continuación mostramos un ejemplo de formulario:

```
Form f = new Form("Formulario");

Item itemEtiqueta = new StringItem("Etiqueta:",
    "Texto de la etiqueta");
Item itemTexto = new TextField("Telefono:", "", 8,
    TextField.PHONENUMBER);
Item itemFecha = new DateField("Fecha", DateField.DATE_TIME);
Item itemBarra = new Gauge("Volumen", true, 10, 8);
```

```

ChoiceGroup itemOpcion = new ChoiceGroup("Opcion",
                                           Choice.EXCLUSIVE);
itemOpcion.append("Si", null);
itemOpcion.append("No", null);

f.append(itemEtiqueta);
f.append(itemTexto);
f.append(itemFecha);
f.append(itemBarra);
f.append(itemOpcion);

Display d = Display.getDisplay(this);
d.setCurrent(f);

```

El aspecto de este formulario es el siguiente:



En MIDP 2.0 aparecen dos nuevos tipos de items que podremos añadir a los formularios. Estos items son:

- **Spacer:** Se trata de un item vacío, al que se le asigna un tamaño mínimo, que nos servirá para introducir un espacio en blanco en el formulario. El item tendrá una altura y anchura mínima, y al insertarlo en el formulario se creará un espacio en blanco con este tamaño. El siguiente item que añadamos se posicionará después de este espacio.
- **CustomItem:** Este es un item personalizable, en el que podremos definir totalmente su aspecto y su forma de interactuar con el usuario. La forma en la que se definen estos items es similar a la forma en la que se define el Canvas, que estudiaremos en temas posteriores. Al igual que el Canvas, este componente pertenece a la API de bajo nivel, ya que permite al usuario dibujar los gráficos y leer la entrada del usuario a bajo nivel.

### 3.3.3.4. Alertas

Las alertas son un tipo especial de pantallas, que servirán normalmente de transición entre dos pantallas. En ellas normalmente se muestra un mensaje de información, error o advertencia y se pasa automáticamente a la siguiente pantalla.

Las alertas se encapsulan en la clase `Alert`, y se crearán normalmente con el siguiente constructor:

```

Alert a = new Alert(titulo, texto, imagen, tipo);

```

Donde `título` es el título de la pantalla y `texto` será el texto que se muestre en la alerta. Podemos mostrar una imagen de forma opcional. Si no queremos usar ninguna imagen pondremos `null` en el campo correspondiente. Además debemos dar un tipo de alerta. Estos tipos se definen como constantes de la clase `AlertType`:

<code>AlertType.ERROR</code>	Muestran un mensaje de error de la aplicación.
<code>AlertType.WARNING</code>	Muestran un mensaje de advertencia.
<code>AlertType.INFO</code>	Muestran un mensaje de información.
<code>AlertType.CONFIRMATION</code>	Muestran un mensaje de confirmación de alguna acción realizada.
<code>AlertType.ALARM</code>	Notifican de un evento en el que está interesado el usuario.

A estas alertas se les puede asignar un tiempo límite (*timeout*), de forma que transcurrido este tiempo desde que se mostró la alerta se pase automáticamente a la siguiente pantalla.

Para mostrar una alerta lo haremos de forma distinta a los componentes que hemos visto anteriormente. En este caso utilizaremos el siguiente método:

```
mi_display.setCurrent(alerta, siguiente_pantalla);
```

Debemos especificar además de la alerta, la siguiente pantalla a la que iremos tras mostrar la alerta, ya que como hemos dicho anteriormente la alerta es sólo una pantalla de transición.

Por ejemplo, podemos crear una alerta que muestre un mensaje de error al usuario y que vuelva a la misma pantalla en la que estamos:

```
Alert a = new Alert("Error", "No hay ninguna nota seleccionada",
                    null, AlertType.ERROR);
Display d = Display.getDisplay(midlet);
d.setCurrent(a, d.getCurrent());
```

A continuación podemos ver dos alertas distintas, mostrando mensajes de error, de información y de alarma respectivamente:



Puede ser interesante combinar las alertas con temporizadores para implementar agendas

en las que el móvil nos recuerde diferentes eventos mostrando una alerta a una hora determinada, o hacer sonar una alarma, ya que estas alertas nos permiten incorporar sonido.

Por ejemplo podemos implementar una tarea que dispare una alarma, mostrando una alerta y reproduciendo sonido. La tarea puede contener el siguiente código:

```
class Alarma extends TimerTask {
    public void run() {
        Alert a = new Alert("Alarma",
            "Se ha disparado la alarma", null, AlertType.ALARM);
        a.setTimeout(Alert.FOREVER);

        Display d = Display.getDisplay(midlet);
        AlertType.ALARM.playSound(d);

        d.setCurrent(a, d.getCurrent());
    }
}
```

Una vez definida la tarea que implementa la alarma, podemos utilizar un temporizador para planificar el comienzo de la alarma a una hora determinada:

```
Timer temp = new Timer();
Alarma a = new Alarma();
temp.schedule(a, tiempo);
```

Como tiempo de comienzo podremos especificar un retardo en milisegundos o una hora absoluta a la que queremos que se dispare la alarma.

### 3.3.4. Imágenes

En muchos de los componentes anteriores hemos visto que podemos incorporar imágenes. Estas imágenes se encapsularán en la clase `Image`, que contendrá el *raster* (matriz de *pixels*) correspondiente a dicha imagen en memoria. Según si este *raster* puede ser modificado o no, podemos clasificar las imágenes en mutables e inmutables.

#### 3.3.4.1. Imágenes mutable

Nos permitirán modificar su contenido dentro del código de nuestra aplicación. En la API de interfaz gráfica de bajo nivel veremos cómo modificar estas imágenes. Las imágenes mutables se crean como una imagen en blanco con unas determinadas dimensiones, utilizando el siguiente método:

```
Image img = Image.createImage(ancho, alto);
```

Nada más crearla estará vacía. A partir de este momento podremos dibujar en ella cualquier contenido, utilizando la API de bajo nivel.

#### 3.3.4.2. Imágenes inmutables

Las imágenes inmutables una vez creadas no pueden ser modificadas. Las imágenes que nos permiten añadir los componentes de alto nivel vistos previamente deben ser inmutables, ya que estos componentes no están preparados para que la imagen pueda cambiar en cualquier momento.

Para crear una imagen inmutable deberemos proporcionar el contenido de la imagen en el momento de su creación, ya que no se podrá modificar más adelante. Lo normal será utilizar ficheros de imágenes. Las aplicaciones MIDP soportan el formato PNG, por lo que deberemos utilizar este formato.

- Carga de imágenes desde recursos:  
Podemos cargar una imagen de un fichero PNG incluido dentro del JAR de nuestra aplicación utilizando el siguiente método:

```
Image img = Image.createImage(nombre_fichero);
```

De esta forma buscará dentro del JAR un recurso con el nombre que hayamos proporcionado, utilizando internamente el método `Class.getResourceAsStream` que vimos en el capítulo anterior.

NOTA: Las imágenes son el único tipo de recurso que proporcionan su propio método para cargarlas desde un fichero dentro del JAR. Para cualquier otro tipo de recurso, como por ejemplo ficheros de texto, deberemos utilizar `Class.getResourceAsStream` para abrir un flujo de entrada que lea de él y leerlo manualmente.

- Carga desde otra ubicación:  
Si la imagen no está dentro del fichero JAR, como por ejemplo en el caso de que queramos leerla de la web, no podremos utilizar el método anterior. Encontramos un método más genérico para la creación de una imagen inmutable que crea la imagen a partir de la secuencia de *bytes* del fichero PNG de la misma:

```
Image img = Image.createImage(datos, offset, longitud);
```

Donde *datos* es un *array* de *bytes*, *offset* la posición del *array* donde comienza la imagen, y *longitud* el número de *bytes* que ocupa la imagen.

Por ejemplo, si queremos cargar una imagen desde la red podemos hacer lo siguiente:

```
// Abre una conexión en red con la URL de la imagen
String url = "http://jtech.ua.es/imagenes/logo.png";
URLConnection con = (URLConnection) Connector.open(url);
InputStream in = con.getInputStream();

// Lee bytes de la imagen
int c;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
while( (c=in.read()) != -1 ) {
    baos.write(c);
}

// Crea imagen a partir de array de bytes
byte [] datos = baos.toByteArray();
```



```
Image img = Image.createImage(datos,0,datos.length);
```

- **Conversión de mutable a inmutable:**

Es posible que queramos mostrar una imagen que hemos modificado desde dentro de nuestro programa en alguno de los componentes de alto nivel anteriores. Sin embargo ya hemos visto que sólo se pueden mostrar en estos componentes imágenes inmutables.

Lo que podemos hacer es convertir la imagen de mutable a inmutable, de forma que crearemos una versión no modificable de nuestra imagen mutable, que pueda ser utilizada en estos componentes. Si hemos creado una imagen mutable `img_mutable`, podemos crear una versión inmutable de esta imagen de la siguiente forma:

```
Image img_inmutable = Image.createImage(img_mutable);
```

Una vez tenemos creada la imagen inmutable, podremos mostrarla en distintos componentes de alto nivel, como alertas, listas y algunos items dentro de los formularios (cuadro de opciones e item de tipo imagen).

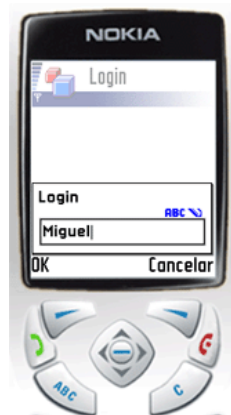
En las alertas, listas y cuadros de opciones de los formularios simplemente especificaremos la imagen que queremos mostrar, y éste se mostrará en la pantalla de alerta o junto a uno de los elementos de la lista. En los items de tipo imagen (`ImageItem`) de los formularios, podremos controlar la disposición (*layout*) de la imagen, permitiéndonos por ejemplo mostrarla centrada, a la izquierda o a la derecha.

### 3.3.5. Comandos de entrada

Hemos visto como crear una serie de componentes de alto nivel para mostrar en nuestra aplicación. Sin embargo no hemos visto como interactuar con las acciones que realice el usuario, para poderles dar una respuesta desde nuestra aplicación.

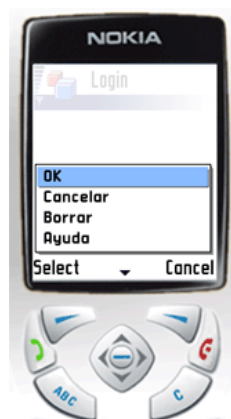
En estos componentes de alto nivel el usuario podrá interactuar mediante una serie de comandos que podrá ejecutar. Para cada pantalla podremos definir una lista de comandos, de forma que el usuario pueda seleccionar y ejecutar uno de ellos. Esta es una forma de interacción de alto nivel, que se implementará a nivel nativo y que será totalmente portable.

En el móvil estos comandos se encontrarán normalmente en una o en las dos esquinas inferiores, y se podrán activar pulsando sobre el botón situado justo bajo dicha esquina:



Comandos de las pantallas

Según el dispositivo tendremos uno o dos botones de este tipo. Si tenemos varios comandos, al pulsar sobre el botón de la esquina correspondiente se abrirá un menú con todos los comandos disponibles para seleccionar uno de ellos.



Despliegue del menú de comandos

### 3.3.5.1. Creación de comandos

Estos comandos se definen mediante la clase `Command`, y pueden ser creados utilizando el siguiente constructor:

```
Command c = new Command(etiqueta, tipo, prioridad);
```

En `etiqueta` especificaremos el texto que se mostrará en el comando. Los otros dos parámetros se utilizarán para mejorar la portabilidad entre dispositivos. En `tipo` podremos definir el tipo del comando, pudiendo ser:

<code>Command.OK</code>	Dar una respuesta positiva
<code>Command.BACK</code>	Volver a la pantalla anterior
<code>Command.CANCEL</code>	Dar una respuesta negativa

<code>Command.EXIT</code>	Salir de la aplicación
<code>Command.HELP</code>	Mostrar una pantalla de ayuda
<code>Command.STOP</code>	Detener algún proceso que se esté realizando
<code>Command.SCREEN</code>	Comando propio de nuestra aplicación para la pantalla actual.
<code>Command.ITEM</code>	Comando específico para ser aplicado al ítem seleccionado actualmente. De esta forma se comportará como un menú contextual.

El asignar uno de estos tipos no servirá para que el comando realice una de estas acciones. Las acciones que se realicen al ejecutar el comando las deberemos implementar siempre nosotros. El asignar estos tipos simplemente sirve para que la implementación nativa del dispositivo conozca qué función desempeña cada comando, de forma que los sitúe en el lugar adecuado para dicho dispositivo. Cada dispositivo podrá distribuir los distintos tipos de comandos utilizando diferentes criterios.

Por ejemplo, si en nuestro dispositivo la acción de volver atrás suele asignarse al botón de la esquina derecha, si añadimos un comando de este tipo intentará situarlo en este lugar.

Además les daremos una prioridad con la que establecemos la importancia de los comandos. Esta prioridad es un valor entero, que cuanto menor sea más importancia tendrá el comando. Un comando con prioridad 1 tiene importancia máxima. Primero situará los comandos utilizando el tipo como criterio, y para los comandos con el mismo tipo utilizará la prioridad para poner más accesibles aquellos con mayor prioridad.

Una vez hemos creado los comandos, podemos añadirlos a la pantalla actual utilizando el método:

```
pantalla.addCommand(c);
```

Esta pantalla podrá ser cualquier elemento *displayable* de los que hemos visto anteriormente excepto `Alarm`. ya que no está permitido añadir comandos a las alarmas. De esta forma añadiremos todos los comandos necesarios.

Por ejemplo, podemos añadir una serie de comandos a la pantalla de *login* de nuestra aplicación de la siguiente forma:

```
TextBox tb = new TextBox("Login", "", 8, TextField.ANY);

Command cmdOK = new Command("OK", Command.OK, 1);
Command cmdAyuda = new Command("Ayuda", Command.HELP, 1);
Command cmdSalir = new Command("Salir", Command.EXIT, 1);
Command cmdBorrar = new Command("Borrar", Command.SCREEN, 1);
Command cmdCancelar = new Command("Cancelar", Command.CANCEL, 1);

tb.addCommand(cmdOK);
tb.addCommand(cmdAyuda);
tb.addCommand(cmdSalir);
tb.addCommand(cmdBorrar);
tb.addCommand(cmdCancelar);
```

```
Display d = Display.getDisplay(this);
d.setCurrent(tb);
```

### 3.3.5.2. Listener de comandos

Una vez añadidos los comandos a la pantalla, deberemos definir el código para dar respuesta a cada uno de ellos. Para ello deberemos crear un *listener*, que es un objeto que escucha las acciones del usuario para darles una respuesta.

El *listener* será una clase en la que introduciremos el código que queremos que se ejecute cuando el usuario selecciona uno de los comandos. Cuando se pulse sobre uno de estos comandos, se invocará dicho código.

Para crear el *listener* debemos crear una clase que implemente la interfaz `commandListener`. El implementar esta interfaz nos obligará a definir el método `commandAction`, que será donde deberemos introducir el código que dé respuesta al evento de selección de un comando.

```
class MiListener implements CommandListener {
    public void commandAction(Command c, Displayable d) {
        // Código de respuesta al comando
    }
}
```

Cuando se produzca un evento de este tipo, conoceremos qué comando se ha seleccionado y en qué *displayable* estaba, ya que esta información se proporciona como parámetros. Según el comando que se haya ejecutado, dentro de este método deberemos decidir qué acción realizar.

Por ejemplo, podemos crear un listener para los comandos añadidos a la pantalla de *login* del ejemplo anterior:

```
class ListenerLogin implements CommandListener {
    public void commandAction(Command c, Displayable d) {
        if(c == cmdOK) {
            // Aceptar
        } else if(c == cmdCancelar) {
            // Cancelar
        } else if(c == cmdSalir) {
            // Salir
        } else if(c == cmdAyuda) {
            // Ayuda
        } else if(c == cmdBorrar) {
            // Borrar
        }
    }
}
```

Una vez creado el *listener* tendremos registrarlo en el *displayable* que contiene los comandos para ser notificado de los comandos que ejecute el usuario. Para establecerlo

como *listener* utilizaremos el método `setCommandListener` del *displayable*.

Por ejemplo, en el caso del campo de texto de la pantalla de *login* lo registraremos de la siguiente forma:

```
tb.setCommandListener(new ListenerLogin());
```

Una vez hecho esto, cada vez que el usuario ejecute un comando se invocará el método `commandAction` del *listener* que hemos definido, indicándonos el comando que se ha invocado.

### 3.3.5.3. Listas implícitas

En las listas implícitas dijimos que cuando se pulsa sobre un elemento de la lista se notifica inmediatamente a la aplicación para que se realice la acción correspondiente, de forma que se comporta como un menú.

La forma que tiene de notificarse la selección de un elemento de este tipo de listas es invocando un comando. En este caso se invocará un tipo especial de comando definido como constante en la clase `List`, se trata de `List.SELECT_COMMAND`.

Dentro de `commandAction` podemos comprobar si se ha ejecutado un comando de este tipo para saber si se ha seleccionado un elemento de la lista. En este caso, podremos saber el elemento del que se trata viendo el índice que se ha seleccionado:

```
class ListenerLogin implements CommandListener {
    public void commandAction(Command c, Displayable d) {
        if(c == List.SELECT_COMMAND) {
            int indice = l.getSelectedIndex();
            if(indice == 0) {
                // Nuevo juego
            } else if(indice == 1) {
                // Continuar
            } else if(indice == 2) {
                // Instrucciones
            } else if(indice == 3) {
                // Hi-score
            } else if(indice == 4) {
                // Salir
            }
        }
    }
}
```

### 3.3.5.4. Listener de items

En el caso de los formularios, podremos tener constancia de cualquier cambio que el usuario haya introducido en alguno de sus campos antes de que se ejecute algún comando para realizar alguna acción.

Por ejemplo, esto nos puede servir para validar los datos introducidos. En el momento

que el usuario cambie algún campo, se nos notificará dicho cambio pudiendo comprobar de esta forma si el valor introducido es correcto o no. Además, de esta forma sabremos si ha habido cambios, por lo que podremos volver a grabar los datos del formulario de forma persistente sólo en caso necesario.

Para recibir la notificación de cambio de algún ítem del formulario, utilizaremos un *listener* de tipo `ItemStateListener`, en el que deberemos definir el método `itemStateChanged` donde introduciremos el código a ejecutar en caso de que el usuario modifique alguno de los campos modificables (cuadros de opciones, campo de texto, campo de fecha o barra de nivel). El esqueleto de un *listener* de este tipo será el siguiente:

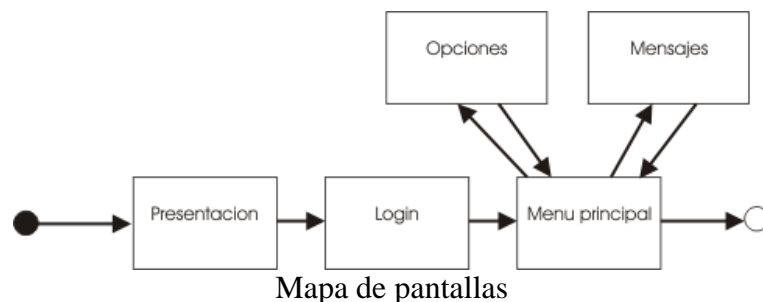
```
class MiListener implements ItemStateListener {
    public void itemStateChanged(Item i) {
        // Se ha modificado el ítem i
    }
}
```

### 3.3.6. Transiciones entre pantallas

Hemos visto que cada uno de los componentes *displayables* que tenemos disponibles representa una pantalla, y podemos cambiar esta pantalla utilizando el método `setCurrent` del *display*.

De esta forma podremos pasar de una pantalla a otra de la aplicación cuando ocurra un determinado evento, como puede ser por ejemplo que el usuario ejecute un determinado comando o que se ejecute alguna tarea planificada por un temporizador.

Cuando tengamos una aplicación con un número elevado de pantallas, será recomendable hacer previamente un diseño de esta aplicación. Definiremos un diagrama de navegación, en el que cada bloque representará una pantalla, y las flechas que unen dichos bloques serán las transiciones entre pantallas.



Debemos asegurarnos en este mapa de pantallas que el usuario en todo momento puede volver atrás y que hemos definido todos los enlaces necesarios para acceder a todas las pantallas de la aplicación.

#### 3.3.6.1. Vuelta atrás

Normalmente las aplicaciones tendrán una opción que nos permitirá volver a la pantalla visitada anteriormente. Para implementar esto podemos utilizar una pila (*Stack*), en la que iremos apilando todas las pantallas conforme las visitamos. Cuando pulsemos el botón para ir atrás desapilaremos la ultima pantalla y la mostraremos en el *display* utilizando *setCurrent*.

### 3.3.6.2. Diseño de pantallas

Es conveniente tomar algún determinado patrón de diseño para implementar las pantallas de nuestra aplicación. Podemos crear una clase por cada pantalla, donde encapsularemos todo el contenido que se debe mostrar en la pantalla, los comandos disponibles, y los *listeners* que den respuesta a estos comandos.

Las clases implementadas según este patrón de diseño cumplirán lo siguiente:

- Heredan del tipo de *displayable* al que pertenecen. De esta forma nuestra pantalla será un tipo especializado de este *displayable*.
- Implementan la interfaz *CommandListener* para encapsular la respuesta a los comandos. Esto nos forzará a definir dentro de esta clase el método *commandAction* para dar respuesta a los comandos. Podemos implementar también la interfaz *ItemStateListener* en caso necesario.
- Al constructor se le proporciona como parámetro el *MIDlet* de la aplicación, además de cualquier otro parámetro que necesitemos añadir. Esto será necesario para poder obtener una referencia al *display*, y de esa forma poder provocar la transición a otra pantalla. Así podremos hacer que sea dentro de la clase de cada pantalla donde se definan las posibles transiciones a otras pantallas.

Por ejemplo, podemos implementar el menú principal de nuestra aplicación de la siguiente forma:

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

public class MenuPrincipal extends List
    implements CommandListener {

    MiMIDlet owner;
    Command selec;
    int itemNuevo;
    int itemSalir;

    public MenuPrincipal(MiMIDlet owner) {
        super("Menu", List.IMPLICIT);
        this.owner = owner;

        // Añade opciones al menu
        itemNuevo = this.append("Nuevo juego", null);
        itemSalir = this.append("Salir", null);

        // Crea comandos
        selec = new Command("Seleccionar", Command.SCREEN, 1);
        this.addCommand(selec);
        this.setCommandListener(this);
    }
}
```

```

    }

    public void commandAction(Command c, Displayable d) {
        if(c == selec || c == List.SELECT_COMMAND) {
            if(getSelectedIndex() == itemNuevo) {
                // Nuevo juego
                Display display = Display.getDisplay(owner);
                PantallaJuego pj = new PantallaJuego(owner, this);
                display.setCurrent(pj);
            } else if(getSelectedIndex() == itemSalir) {
                // Salir de la aplicación
                owner.salir();
            }
        }
    }
}

```

Si esta es la pantalla principal de nuestra aplicación, la podremos mostrar desde nuestro MIDlet de la siguiente forma:

```

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

public class MiMIDlet extends MIDlet {
    protected void startApp() throws MIDletStateChangeException {
        Display d = Display.getDisplay(this);
        MenuPrincipal mp = new MenuPrincipal(this);
        d.setCurrent(mp);
    }

    protected void pauseApp() {
    }

    protected void destroyApp(boolean incondicional)
        throws MIDletStateChangeException {
    }

    public void salir() {
        try {
            destroyApp(false);
            notifyDestroyed();
        } catch(MIDletStateChangeException e) {
            // Evitamos salir de la aplicacion
        }
    }
}

```

Este patrón de diseño encapsula el comportamiento de cada pantalla en clases independientes, lo cual hará más legible y reutilizable el código.

Con este diseño, si queremos permitir volver a una pantalla anterior podemos pasar como parámetro del constructor, además del MIDlet, el elemento *displayable* correspondiente a esta pantalla anterior. De esta forma cuando pulsemos *Atrás* sólo tendremos que mostrar este elemento en el *display*.

### 3.3.7. Interfaz gráfica de bajo nivel

Hasta ahora hemos visto la creación de aplicaciones con una interfaz gráfica creada a partir de una serie de componentes de alto nivel definidos en la API LCDUI (alertas,



campos de texto, listas, formularios).

En este punto veremos como dibujar nuestros propios gráficos directamente en pantalla. Para ello Java nos proporciona acceso a bajo nivel al contexto gráfico del área donde vayamos a dibujar, permitiéndonos a través de éste modificar los *pixels* de este área, dibujar una serie de figuras geométricas, así como volcar imágenes en ella.

También podremos acceder a la entrada del usuario a bajo nivel, conociendo en todo momento cuándo el usuario pulsa o suelta cualquier tecla del móvil.

Este acceso a bajo nivel será necesario en aplicaciones como juegos, donde debemos tener un control absoluto sobre la entrada y sobre lo que dibujamos en pantalla en cada momento. El tener este mayor control tiene el inconveniente de que las aplicaciones serán menos portables, ya que dibujaremos los gráficos pensando en una determinada resolución de pantalla y un determinado tipo de teclado, pero cuando la queramos llevar a otro dispositivo en el que estos componentes sean distintos deberemos hacer cambios en el código.

Por esta razón para las aplicaciones que utilizan esta API a bajo nivel, como los juegos Java para móviles, encontramos distintas versiones para cada modelo de dispositivo. Dada la heterogeneidad de estos dispositivos, resulta más sencillo rehacer la aplicación para cada modelo distinto que realizar una aplicación adaptable a las características de cada modelo.

Al programar las aplicaciones deberemos facilitar en la medida de lo posible futuros cambios para adaptarla a otros modelos, permitiendo reutilizar la máxima cantidad de código posible.

### 3.3.7.1. Gráficos en LCDUI

---

La API de gráficos a bajo nivel de LCDUI es muy parecida a la existente en AWT, por lo que el aprendizaje de esta API para programadores que conozcan la de AWT va a ser casi inmediato.

Las clases que implementan la API de bajo nivel en LCDUI son `Canvas` y `Graphics`. Estas clases reciben el mismo nombre que las de AWT, y se utilizan de una forma muy parecida. Tienen alguna diferencia en cuanto a su interfaz para adaptarse a las necesidades de los dispositivos móviles.

El `Canvas` es un tipo de elemento *displayable* correspondiente a una pantalla vacía en la que nosotros podremos dibujar a bajo nivel el contenido que queramos. Además este componente nos permitirá leer los eventos de entrada del usuario a bajo nivel.

Esta pantalla del móvil tiene un contexto gráfico asociado que nosotros podremos utilizar para dibujar en ella. Este objeto encapsula el *raster* de pantalla (la matriz de *pixels* de los que se compone la pantalla) y además tiene una serie de atributos con los que podremos modificar la forma en la que se dibuja en este *raster*. Este contexto gráfico se definirá en

un objeto de la clase `Graphics`. Este objeto nos ofrece una serie de métodos que nos permiten dibujar distintos elementos en pantalla. Más adelante veremos con detalle los métodos más importantes.

Este objeto `Graphics` para dibujar en la pantalla del dispositivo nos lo deberá proporcionar el sistema en el momento en que se vaya a dibujar, no podremos obtenerlo nosotros por nuestra cuenta de ninguna otra forma. Esto es lo que se conoce como *render* pasivo, definimos la forma en la que se dibuja pero es el sistema el que decidirá cuándo hacerlo.

### Creación de un Canvas

Para definir la forma en la que se va a dibujar nuestro componente deberemos extender la clase `Canvas` redefiniendo su método `paint`. Dentro de este método es donde definiremos cómo se realiza el dibujo de la pantalla. Esto lo haremos de la siguiente forma:

```
public class MiCanvas extends Canvas {
    public void paint(Graphics g) {
        // Dibujamos en la pantalla
        // usando el objeto g proporcionado
    }
}
```

Con esto en la clase `MiCanvas` hemos creado una pantalla en la que nosotros controlamos lo que se dibuja. Este método `paint` nunca debemos invocarlo nosotros, será el sistema el que se encargue de invocarlo cuando necesite dibujar el contenido de la pantalla. En ese momento se proporcionará como parámetro el objeto correspondiente al contexto gráfico de la pantalla del dispositivo, que podremos utilizar para dibujar en ella. Dentro de este método es donde definiremos cómo dibujar en la pantalla, utilizando para ello el objeto de contexto gráfico `Graphics`.

Siempre deberemos dibujar utilizando el objeto `Graphics` dentro del método `paint`. Guardarnos este objeto y utilizarlo después de haberse terminado de ejecutar `paint` puede producir un comportamiento indeterminado, y por lo tanto no debe hacerse nunca.

### Propiedades del Canvas

Las pantallas de los dispositivos pueden tener distintas resoluciones. Además normalmente el área donde podemos dibujar no ocupa toda la pantalla, ya que el móvil utiliza una franja superior para mostrar información como la cobertura o el título de la aplicación, y en la franja inferior para mostrar los comandos disponibles.

Sin embargo, a partir de MIDP 2.0 aparece la posibilidad de utilizar modo a pantalla completa, de forma que controlaremos el contenido de toda la pantalla. Para activar el modo a pantalla completa utilizaremos el siguiente método:

```
setFullScreenMode(true); // Solo disponible a partir de MIDP 2.0
```

Es probable que nos interese conocer desde dentro de nuestra aplicación el tamaño real

del área del Canvas en la que podemos dibujar. Para ello tenemos los métodos `getWidth` y `getHeight` de la clase `Canvas`, que nos devolverán el ancho y el alto del área de dibujo respectivamente.

Para obtener información sobre el número de colores soportados deberemos utilizar la clase `Display` tal como vimos anteriormente, ya que el número de colores es propio de todo el visor y no sólo del área de dibujo.

### Mostrar el Canvas

Podemos mostrar este componente en la pantalla del dispositivo igual que mostramos cualquier otro *displayable*:

```
MiCanvas mc = new MiCanvas();
mi_display.setCurrent(mc);
```

Es posible que queramos hacer que cuando se muestre este *canvas* se realice alguna acción, como por ejemplo poner en marcha alguna animación que se muestre en la pantalla. De la misma forma, cuando el *canvas* se deje de ver deberemos detener la animación. Para hacer esto deberemos tener constancia del momento en el que el *canvas* se muestra y se oculta.

Podremos saber esto debido a que los métodos `showNotify` y `hideNotify` de la clase `Canvas` serán invocados son invocados por el sistema cuando dicho componente se muestra o se oculta respectivamente. Nosotros podremos en nuestra subclase de `Canvas` redefinir estos métodos, que por defecto están vacíos, para definir en ellos el código que se debe ejecutar al mostrarse u ocultarse nuestro componente. Por ejemplo, si queremos poner en marcha o detener una animación, podemos redefinir los métodos como se muestra a continuación:

```
public class MiCanvas extends Canvas {
    public void paint(Graphics g) {
        // Dibujamos en la pantalla
        // usando el objeto g proporcionado
    }

    public void showNotify() {
        // El Canvas se muestra
        comenzarAnimacion();
    }

    public void hideNotify() {
        // El Canvas se oculta
        detenerAnimacion();
    }
}
```

De esta forma podemos utilizar estos dos métodos como respuesta a los eventos de aparición y ocultación del *canvas*.

### 3.3.7.2. Contexto gráfico

El objeto `Graphics` nos permitirá acceder al contexto gráfico de un determinado componente, en nuestro caso el *canvas*, y a través de él dibujar en el *raster* de este componente. En el caso del contexto gráfico del *canvas* de LCDUI este *raster* corresponderá a la pantalla del dispositivo móvil. Vamos a ver ahora como dibujar utilizando dicho objeto `Graphics`. Este objeto nos permitirá dibujar distintas primitivas geométricas, texto e imágenes.

### Atributos

El contexto gráfico tendrá asociados una serie de atributos que indicarán cómo se va a dibujar en cada momento, como por ejemplo el color o el tipo del lápiz que usamos para dibujar. El objeto `Graphics` proporciona una serie de métodos para consultar o modificar estos atributos. Podemos encontrar los siguientes atributos en el contexto gráfico de LCDUI:

- **Color del lápiz:** Indica el color que se utilizará para dibujar la primitivas geométricas y el texto. MIDP trabaja con color de 24 bits (*truecolor*), que codificaremos en modelo RGB. Dentro de estos 24 bits tendremos 8 bits para cada uno de los tres componentes: rojo (R), verde (G) y azul (B). No tenemos canal *alpha*, por lo que no soportará transparencia. No podemos contar con que todos los dispositivos soporten color de 24 bits. Lo que hará cada implementación concreta de MIDP será convertir los colores solicitados en las aplicaciones al color más cercano soportado por el dispositivo.

Podemos trabajar con los colores de dos formas distintas: tratando los componentes R, G y B por separado, o de forma conjunta. En MIDP desaparece la clase `Color` que teníamos en AWT, por lo que deberemos asignar los colores proporcionando directamente los valores numéricos del color.

Si preferimos tratar los componentes de forma separada, tenemos los siguientes métodos para obtener o establecer el color actual del lápiz:

```
g.setColor(rojo, verde, azul);
int rojo = g.getRedComponent();
int green = g.getGreenComponent();
int blue = g.getBlueComponent();
```

Donde `g` es el objeto `Graphics` del contexto donde vamos a dibujar. Estos componentes rojo, verde y azul tomarán valores entre 0 y 255.

Podemos tratar estos componentes de forma conjunta empaquetándolos en un único entero. En hexadecimal se codifica de la siguiente forma:

0x00RRGGBB

Podremos leer o establecer el color utilizando este formato empaquetado con los siguientes métodos:

```
g.setColor(rgb);
int rgb = g.getColor();
```

Tenemos también métodos para trabajar con valores en escala de grises. Estos métodos nos pueden resultar útiles cuando trabajemos con dispositivos monocromos.

```
int gris = g.getGrayScale();
g.setGrayScale(gris);
```

Con estos métodos podemos establecer como color actual distintos tonos en la escala de grises. El valor de gris se moverá en el intervalo de 0 a 255. Si utilizamos `getGrayScale` teniendo establecido un color fuera de la escala de grises, convertirá este color a escala de grises obteniendo su brillo.

- **Tipo del lápiz:** Además del color del lápiz, también podemos establecer su tipo. El tipo del lápiz indicará cómo se dibujan las líneas de las primitivas geométricas. Podemos encontrar dos estilos:

<code>Graphics.SOLID</code>	Línea sólida (se dibujan todos los <i>pixels</i> )
<code>Graphics.DOTTED</code>	Línea punteada (se salta algunos <i>pixels</i> sin dibujarlos)

Podemos establecer el tipo del lápiz o consultarlo con los siguientes métodos:

```
int tipo = g.getStrokeStyle();
g.setStrokeStyle(tipo);
```

- **Fuente:** Indica la fuente que se utilizará para dibujar texto. Utilizaremos la clase `Font` para especificar la fuente de texto que vamos a utilizar, al igual que en AWT. Podemos obtener o establecer la fuente con los siguientes métodos:

```
Font fuente = g.getFont();
g.setFont(fuente);
```

- **Área de recorte:** Podemos definir un rectángulo de recorte. Cuando definimos un área de recorte en el contexto gráfico, sólo se dibujarán en pantalla los *pixels* que caigan dentro de este área. Nunca se dibujarán los *pixels* que escribamos fuera de este espacio. Para establecer el área de recorte podemos usar el siguiente método:

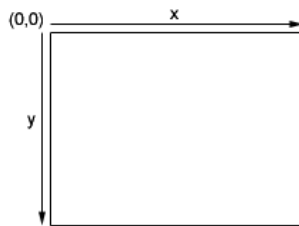
```
g.setClip(x, y, ancho, alto);
```

También tenemos disponible el siguiente método:

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

Este método establece un recorte en el área de recorte anterior. Si ya existía un rectángulo de recorte, el nuevo rectángulo de recorte será la intersección de ambos. Si queremos eliminar el área de recorte anterior deberemos usar el método `setClip`.

- **Origen de coordenadas:** Indica el punto que se tomará como origen en el sistema de coordenadas del área de dibujo. Por defecto este sistema de coordenadas tendrá la coordenada (0,0) en su esquina superior izquierda, y las coordenadas serán positivas hacia la derecha (coordenada x) y hacia abajo (coordenada y), tal como se muestra a continuación:

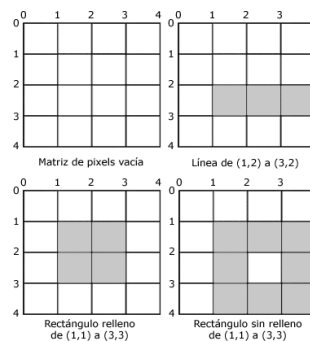


Sistema de coordenadas del área de dibujo

Podemos trasladar el origen de coordenadas utilizando el método `translate`. También tenemos métodos para obtener la traslación del origen de coordenadas.

```
int x = g.getTranslateX();
int y = g.getTranslateY();
g.translate(x, y);
```

Estas coordenadas no corresponden a *pixels*, sino a los límites de los *pixels*. De esta forma, el *pixel* de la esquina superior izquierda de la imagen se encontrará entre las coordenadas (0,0), (0,1), (1,0) y (1,1).



Coordenadas de los límites de los pixels

### Dibujar primitivas geométricas

Una vez establecidos estos atributos en el contexto gráfico, podremos dibujar en él una serie de elementos utilizando una serie de métodos de `Graphics`. Vamos a ver en primer lugar cómo dibujar una serie de primitivas geométricas. Para ello tenemos una serie de métodos que comienzan por `draw_` para dibujar el contorno de una determinada figura, o `fill_` para dibujar dicha figura con relleno.

- **Líneas:** Dibuja una línea desde un punto ( $x1, y1$ ) hasta ( $x2, y2$ ). Dibujaremos la línea con:

```
g.drawLine(x1, y1, x2, y2);
```

En este caso no encontramos ningún método `fill` ya que las líneas no pueden tener relleno. Al dibujar una línea se dibujarán los *pixels* situados inmediatamente abajo y a la derecha de las coordenadas indicadas. Por ejemplo, si dibujamos con `drawLine(0, 0, 0, 0)` se dibujará el *pixel* de la esquina superior izquierda.

- **Rectángulos:** Podemos dibujar rectángulos especificando sus coordenadas y su altura y anchura. Podremos dibujar el rectángulo relleno o sólo el contorno:

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

En el caso de `fillRect`, lo que hará será rellenar con el color actual los *pixels* situados entre las coordenadas limítrofes. En el caso de `drawRect`, la línea inferior y derecha se dibujarán justo debajo y a la derecha respectivamente de las coordenadas de dichos límites, es decir, se dibuja con un *pixel* más de ancho y de alto que en el caso relleno. Esto es poco intuitivo, pero se hace así para mantener la coherencia con el comportamiento de `drawLine`.

Al menos, lo que siempre se nos asegura es que cuando utilizamos las mismas dimensiones no quede ningún hueco entre el dibujo del relleno y el del contorno.

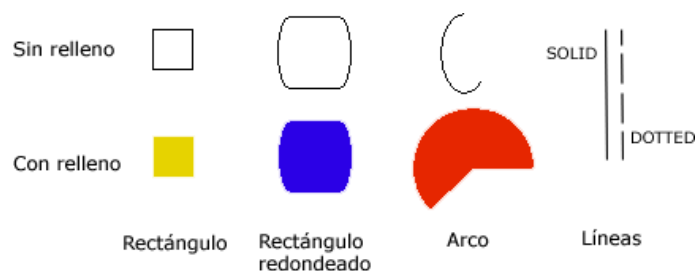
Podemos también dibujar rectángulos con las esquinas redondeadas, utilizando los métodos:

```
g.drawRoundRect(x, y, ancho, alto, ancho_arco, alto_arco);
g.fillRoundRect(x, y, ancho, alto, ancho_arco, alto_arco);
```

- **Arcos:** A diferencia de AWT, no tenemos un método para dibujar directamente elipses, sino que tenemos uno más genérico que nos permite dibujar arcos de cualquier tipo. Nos servirá tanto para dibujar elipses y círculos como para cualquier otro tipo de arco.

```
g.drawArc(x, y, ancho, alto, angulo_inicio, angulo_arco);
g.fillArc(x, y, ancho, alto, angulo_inicio, angulo_arco);
```

Los ángulos especificados deben estar en grados. Por ejemplo, si queremos dibujar un círculo o una elipse en `angulo_arco` pondremos un valor de 360 grados para que se cierre el arco. En el caso del círculo los valores de `ancho` y `alto` serán iguales, y en el caso de la elipse serán diferentes.



Ejemplos de diferentes primitivas

Por ejemplo, el siguiente *canvas* aparecerá con un dibujo de un círculo rojo y un cuadrado verde:

```
public class MiCanvas extends Canvas {
    public void paint(Graphics g) {
        g.setColor(0x00FF0000);
        g.fillArc(10,10,50,50,0,360);
    }
}
```

```

        g.setColor(0x0000FF00);
        g.fillRect(60,60,50,50);
    }
}

```

### Puntos anchor

En MIDP se introduce una característica no existente en AWT que son los puntos *anchor*. Estos puntos nos facilitarán el posicionamiento del texto y de las imágenes en la pantalla. Con los puntos *anchor*, además de dar una coordenada para posicionar estos elementos, diremos qué punto del elemento vamos a posicionar en dicha posición.

Para el posicionamiento horizontal tenemos las siguientes posibilidades:

Graphics.LEFT	En las coordenadas especificadas se posiciona la parte izquierda del texto o de la imagen.
Graphics.HCENTER	En las coordenadas especificadas se posiciona el centro del texto o de la imagen.
Graphics.RIGHT	En las coordenadas especificadas se posiciona la parte derecha del texto o de la imagen.

Para el posicionamiento vertical tenemos:

Graphics.TOP	En las coordenadas especificadas se posiciona la parte superior del texto o de la imagen.
Graphics.VCENTER	En las coordenadas especificadas se posiciona el centro de la imagen. No se aplica a texto.
Graphics.BASELINE	En las coordenadas especificadas se posiciona la línea de base del texto. No se aplica a imágenes.
Graphics.BOTTOM	En las coordenadas especificadas se posiciona la parte inferior del texto o de la imagen.

### Cadenas de texto

Podemos dibujar una cadena de texto utilizando el método `drawString`. Deberemos proporcionar la cadena de texto de dibujar y el punto *anchor* donde dibujarla.

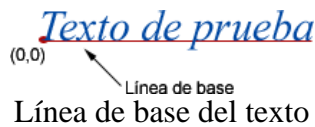
```
g.drawString(cadena, x, y, anchor);
```

Por ejemplo, si dibujamos la cadena con:

```
g.drawString("Texto de prueba", 0, 0, Graphics.LEFT|Graphics.BASELINE);
```

Este punto corresponderá al inicio de la cadena (lado izquierdo), en la línea de base del texto como se muestra a continuación:

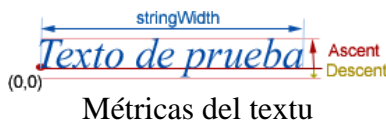




Con esto dibujaremos un texto en pantalla, pero es posible que nos interese conocer las coordenadas que limitan el texto, para saber exactamente el espacio que ocupa en el área de dibujo. En AWT podíamos usar para esto un objeto `FontMetrics`, pero este objeto no existe en MIDP. En MIDP la información sobre las métricas de la fuente está encapsulada en la misma clase `Font` por lo que será más sencillo acceder a esta información. Podemos obtener esta información utilizando los siguientes métodos de la clase `Font`:

- **`stringWidth(cadena)`**: Nos devuelve el ancho que tendrá la cadena *cadena* en *pixels*.
- **`getHeight()`**: Nos devuelve la altura de la fuente, es decir, la distancia entre las líneas de base de dos líneas consecutivas de texto. Llamamos ascenso (*ascent*) a la altura típica que suelen subir los caracteres desde la línea de base, y descenso (*descent*) a lo que suelen bajar desde esta línea. La altura será la suma del ascenso y el descenso de la fuente, más un margen para evitar que se junten los caracteres de las dos líneas. Es la distancia existente entre el punto superior (`TOP`) y el punto inferior (`BOTTOM`) de la cadena de texto.
- **`getBaselinePosition()`**: Nos devuelve el ascenso de la fuente, es decir, la altura típica desde la línea de base hasta la parte superior de la fuente.

Con estas medidas podremos conocer exactamente los límites de una cadena de texto, tal como se muestra a continuación:



## Imágenes

Hemos visto como crear imágenes y como utilizarlas en componentes de alto nivel. Estas mismas imágenes encapsuladas en la clase `Image`, podrán ser mostradas también en cualquier posición de nuestro área de dibujo.

Para ello utilizaremos el método:

```
g.drawImage(img, x, y, anchor);
```

En este caso podremos dibujar tanto imágenes mutables como inmutables.

Vimos que las imágenes mutables son aquellas cuyo contenido puede ser modificado. Vamos a ver ahora como hacer esta modificación. Las imágenes mutables, al igual que el *canvas*, tienen un contexto gráfico asociado. En el caso de las imágenes, este contexto gráfico representa el contenido de la imagen que es un *raster* en memoria, pero podremos dibujar en él igual que lo hacíamos en el *canvas*. Esto es así debido a que dibujaremos

también mediante un objeto de la clase `Graphics`. Podemos obtener este objeto de contexto gráfico en cualquier momento invocando el método `getGraphics` de la imagen:

```
Graphics offg = img.getGraphics();
```

Si queremos modificar una imagen que hemos cargado de un fichero o de la red, y que por lo tanto es inmutable, podemos crear una copia mutable de la imagen para poder modificarla. Para hacer esto lo primero que deberemos hacer es crear la imagen mutable con el mismo tamaño que la inmutable que queremos copiar. Una vez creada podremos obtener su contexto gráfico, y dibujar en él la imagen inmutable, con lo que habremos hecho la copia de la imagen inmutable a una imagen mutable, que podrá ser modificada más adelante.

```
Image img_mut = Image.createImage(img.getWidth(), img.getHeight());
Graphics offg = img_mut.getGraphics();
offg.drawImage(img, 0, 0, Graphics.TOP|Graphics.LEFT);
```

### 3.3.7.3. Animación

Hasta ahora hemos visto como dibujar gráficos en pantalla, pero lo único que hacemos es definir un método que se encargue de dibujar el contenido del componente, y ese método será invocado cuando el sistema necesite dibujar la ventana.

Sin embargo puede interesarnos cambiar dinámicamente los gráficos de la pantalla para realizar una animación. Para ello deberemos indicar el momento en el que queremos que se redibujen los gráficos.

#### Redibujado del área

Para forzar que se redibuje el área de la pantalla deberemos llamar al método `repaint` del *canvas*. Con eso estamos solicitando al sistema que se repinte el contenido, pero no lo repinta en el mismo momento en el que se llama. El sistema introducirá esta solicitud en la cola de eventos pendientes y cuando tenga tiempo repintará su contenido.

```
MiCanvas mc = new MiCanvas();
...
mc.repaint();
```

En MIDP podemos forzar a que se realicen todos los repintados pendientes llamando al método `serviceRepaints`. La llamada a este método nos bloqueará hasta que se hayan realizado todos los repintados pendientes. Por esta razón deberemos tener cuidado de no causar un interbloqueo invocando a este método.

```
mc.serviceRepaints();
```

Para repintar el contenido de la pantalla el sistema llamará al método `paint`, en MIDP no existe el método `update` de AWT. Por lo tanto, deberemos definir dentro de `paint` qué se va a dibujar en la pantalla en cada instante, de forma que el contenido de la pantalla varíe con el tiempo y eso produzca el efecto de la animación.

Podemos optimizar el redibujado repintando únicamente el área de la pantalla que haya cambiado. Para ello en MIDP tenemos una variante del método `repaint` que nos permitirá hacer esto.

```
repaint(x, y, ancho, alto);
```

Utilizando este método, la próxima vez que se redibuje se invocará `paint` pero se proporcionará un objeto de contexto gráfico con un área de recorte establecida, correspondiente a la zona de la pantalla que hemos solicitado que se redibuje.

Al dibujar cada *frame* de la animación deberemos borrar el contenido del *frame* anterior para evitar que quede el rastro, o al menos borrar la zona de la pantalla donde haya cambios.

Imaginemos que estamos moviendo un rectángulo por pantalla. El rectángulo irá cambiando de posición, y en cada momento lo dibujaremos en la posición en la que se encuentre. Pero si no borramos el contenido de la pantalla en el instante anterior, el rectángulo aparecerá en todos los lugares donde ha estado en instantes anteriores produciendo este efecto indeseable de dejar rastro. Por ello será necesario borrar el contenido anterior de la pantalla.

Sin embargo, el borrar la pantalla y volver a dibujar en cada *frame* muchas veces puede producir un efecto de parpadeo de los gráficos. Si además en el proceso de dibujado se deben dibujar varios componentes, y vamos dibujando uno detrás de otro directamente en la pantalla, en cada *frame* veremos como se va construyendo poco a poco la escena, cosa que también es un efecto poco deseable.

Para evitar que esto ocurra y conseguir unas animaciones limpias utilizaremos la técnica del *doble buffer*.

### Técnica del doble buffer

---

La técnica del *doble buffer* consiste en dibujar todos los elementos que queremos mostrar en una imagen en memoria, que denominaremos *backbuffer*, y una vez se ha dibujado todo volcarlo a pantalla como una unidad. De esta forma, mientras se va dibujando la imagen, como no se hace directamente en pantalla no veremos efectos de parpadeo al borrar el contenido anterior, ni veremos como se va creando la imagen, en pantalla se volcará la imagen como una unidad cuando esté completa.

Para utilizar esta técnica lo primero que deberemos hacer es crearnos el *backbuffer*. Para implementarlo en Java utilizaremos una imagen (objeto `Image`) con lo que tendremos un *raster* en memoria sobre el que dibujar el contenido que queramos mostrar. Deberemos crear una imagen del mismo tamaño de la pantalla en la que vamos a dibujar.

Crearemos para ello una imagen mutable en blanco, como hemos visto anteriormente, con las dimensiones del *canvas* donde vayamos a volcarla:

```
Image backbuffer = Image.createImage(getWidth(), getHeight());
```

Obtenemos su contexto gráfico para poder dibujar en su *raster* en memoria:

```
Graphics offScreen = backbuffer.getGraphics();
```

Una vez obtenido este contexto gráfico, dibujaremos todo lo que queremos mostrar en él, en lugar de hacerlo en pantalla. Una vez hemos dibujado todo el contenido en este contexto gráfico, deberemos volcar la imagen a pantalla (al contexto gráfico del *canvas*) para que ésta se haga visible:

```
g.drawImage(backbuffer, 0, 0, Graphics.TOP|Graphics.LEFT);
```

La imagen conviene crearla una única vez, ya que la animación puede redibujar frecuentemente, y si cada vez que lo hacemos creamos un nuevo objeto imagen estaremos malgastando memoria inútilmente. Es buena práctica de programación en Java instanciar nuevos objetos las mínimas veces posibles, intentando reutilizar los que ya tenemos.

Podemos ver como quedaría nuestra clase ahora:

```
public MiCanvas extends Canvas {
    // Backbuffer
    Image backbuffer = null;

    // Ancho y alto del backbuffer
    int width, height;

    // Coordenadas del rectangulo dibujado
    int x, y;

    public void paint(Graphics g) {
        // Solo creamos la imagen la primera vez
        // o si el componente ha cambiado de tamaño
        if( backbuffer == null ||
            width != getWidth() ||
            height != getHeight() ) {
            width = getWidth();
            height = getHeight();
            backbuffer = Image.createImage(width, height);
        }

        Graphics offScreen = backbuffer.getGraphics();

        // Vaciamos el área de dibujo
        offScreen.clearRect(0,0,getWidth(), getHeight());

        // Dibujamos el contenido en offScreen
        offScreen.setColor(0x00FF0000);
        offScreen.fillRect(x,y,50,50);

        // Volcamos el back buffer a pantalla
        g.drawImage(backbuffer,0,0,Graphics.TOP|Graphics.LEFT);
    }
}
```

En ese ejemplo se dibuja un rectángulo rojo en la posición (x,y) de la pantalla que podrá ser variable, tal como veremos a continuación añadiendo a este ejemplo métodos para realizar la animación.

Algunas implementaciones de MIDP ya realizan internamente el doble *buffer*, por lo que en esos casos no será necesario que lo hagamos nosotros. Es más, convendrá que no lo hagamos para no malgastar innecesariamente el tiempo. Podemos saber si implementa el doble *buffer* o no llamando al método `isDoubleBuffered` del Canvas.

Podemos modificar el ejemplo anterior para en caso de realizar el doble *buffer* la implementación de MIDP, no hacerla nosotros:

```
public MiCanvas extends Canvas {
    ...
    public void paint(Graphics gScreen) {
        boolean doblebuffer = isDoubleBuffered();

        // Solo creamos el backbuffer si no hay doble buffer
        if( !doblebuffer ) {
            if ( backbuffer == null ||
                width != getWidth() ||
                height != getHeight() )
            {
                width = getWidth();
                height = getHeight();
                backbuffer = Image.createImage(width, height);
            }
        }

        // g sera la pantalla o nuestro backbuffer segun si
        // el doble buffer está ya implementado o no

        Graphics g = null;

        if(doblebuffer) {
            g = gScreen;
        } else {
            g = backbuffer.getGraphics();
        }

        // Vaciamos el área de dibujo
        g.clearRect(0,0,getWidth(), getHeight());

        // Dibujamos el contenido en g
        g.setColor(0x00FF0000);
        g.fillRect(x,y,50,50);

        // Volcamos si no hay doble buffer implementado
        if(!doblebuffer) {
            gScreen.drawImage(backbuffer,0,0,
                             Graphics.TOP|Graphics.LEFT);
        }
    }
}
```

### Código para la animación

Si queremos hacer una animación tendremos que ir cambiando ciertas propiedades de los objetos de la imagen (por ejemplo su posición) y solicitar que se redibuje tras cada cambio. Esta tarea deberá realizarla un hilo que se ejecute en segundo plano. El bucle para la animación podría ser el siguiente:

```

public class MiCanvas extends Canvas {
    ...
    public void run() {
        // El rectangulo comienza en (10,10)
        x = 10;
        y = 10;

        while(x < 100) {
            x++;
            repaint();

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {}
        }
    }
}

```

Con este código de ejemplo veremos una animación en la que el rectángulo que dibujamos partirá de la posición (10,10) y cada 100ms se moverá un *pixel* hacia la derecha, hasta llegar a la coordenada (100,10).

Si queremos que la animación se ponga en marcha nada más mostrarse la pantalla del canvas, podremos hacer que este hilo comience a ejecutarse en el método `showNotify` como hemos visto anteriormente.

```

public class MiCanvas extends Canvas implements Runnable {
    ...
    public void showNotify() {
        Thread t = new Thread(this);
        t.start();
    }
}

```

Para implementar estas animaciones podemos utilizar un hilo que duerma un determinado período tras cada iteración, como en el ejemplo anterior, o bien utilizar temporizadores que realicen tareas cada cierto periodo de tiempo. Los temporizadores nos pueden facilitar bastante la tarea de realizar animaciones, ya que simplemente deberemos crear una tarea que actualice los objetos de la escena en cada iteración, y será el temporizador el que se encargue de ejecutar cíclicamente dicha tarea.

### Hilo de eventos

Hemos visto que existen una serie de métodos que se invocan cuando se produce algún determinado evento, y nosotros podemos redefinir estos métodos para indicar cómo dar respuesta a estos eventos. Estos métodos que definimos para que sean invocados cuando se produce un evento son denominados *callbacks*. Tenemos los siguientes *callbacks*:

- **showNotify** y **hideNotify**, para los eventos de aparición y ocultación del canvas.
- **paint** para el evento de dibujado.
- **commandAction** para el evento de ejecución de un comando.
- **keyPressed**, **keyRepeated**, **keyReleased**, **pointerPressed**, **pointerDragged** y **pointerReleased** para los eventos de teclado y de puntero, que veremos más adelante.

Estos eventos son ejecutados por el sistema de forma secuencial, desde un mismo hilo de eventos. Por lo tanto, estos *callbacks* deberán devolver el control cuanto antes, de forma que bloqueen al hilo de eventos el mínimo tiempo posible.

Si dentro de uno de estos *callbacks* tenemos que realizar una tarea que requiera tiempo, deberemos crear un hilo que realice la tarea en segundo plano, para que el hilo de eventos siga ejecutándose mientras tanto.

En algunas ocasiones puede interesarnos ejecutar alguna tarea de forma secuencial dentro de este hilo de eventos. Por ejemplo esto será útil si queremos ejecutar el código de nuestra animación sin que interfiera con el método `paint`. Podemos hacer esto con el método `callSerially` del objeto `Display`. Deberemos proporcionar un objeto `Runnable` para ejecutar su método `run` en serie dentro del hilo de eventos. La tarea que definamos dentro de este `run` deberá terminar pronto, al igual que ocurre con el código definido en los *callbacks*, para no bloquear el hilo de eventos.

Podemos utilizar `callSerially` para ejecutar el código de la animación de la siguiente forma:

```
public class MiCanvas extends Canvas implements Runnable {
    ...
    public void anima() {
        // Inicia la animación
        repaint();
        mi_display.callSerially(this);
    }

    public void run() {
        // Actualiza la animación
        ...
        repaint();
        mi_display.callSerially(this);
    }
}
```

La llamada a `callSerially` nos devuelve el control inmediatamente, no espera a que el método `run` sea ejecutado.

### Optimización de imágenes

Si tenemos varias imágenes correspondientes a varios *frames* de una animación, podemos optimizar nuestra aplicación guardando todas estas imágenes como una única imagen. Las guardaremos en forma de mosaico dentro de un mismo fichero de tipo imagen, y en cada momento deberemos mostrar por pantalla sólo una de las imágenes dentro de este mosaico.



Imagen con los frames de una animación

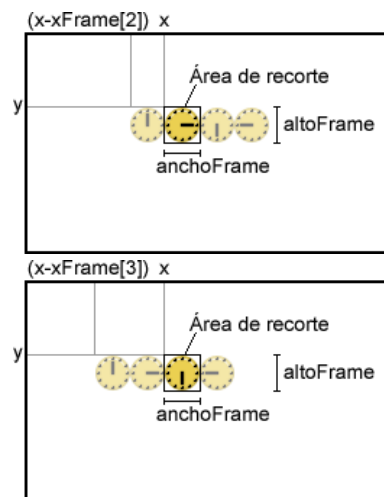
De esta forma estamos reduciendo el número de ficheros que incluimos en el JAR de la aplicación, por lo que por una lado reduciremos el espacio de este fichero, y por otro lado

tendremos que abrir sólo un fichero, y no varios.

Para mostrar sólo una de las imágenes del mosaico, lo que podemos hacer es establecer un área de recorte del tamaño de un elemento del mosaico (*frame*) en la posición donde queramos dibujar esta imagen. Una vez hecho esto, ajustaremos las coordenadas donde dibujar la imagen de forma que dentro del área de recorte caiga el elemento del mosaico que queremos mostrar en este momento. De esta forma, sólo será dibujado este elemento, ignorándose el resto.

Podemos ver esto ilustrado en la Figura 18. En ella podemos ver a la izquierda cómo mostrar el segundo frame del reloj, y a la derecha cómo mostrar el tercer frame. Queremos dibujar el reloj en la posición  $(x, y)$  de la imagen. Cada frame de este reloj tiene un tamaño `anchoFrame` x `altoFrame`. Por lo tanto, el área de recorte será un rectángulo cuya esquina superior izquierda estará en las coordenadas  $(x, y)$  y tendrá una altura y una anchura de `altoFrame` y `anchoFrame` respectivamente, para de esta manera poder dibujar en esa región de la pantalla cada *frame* de nuestra imagen.

Para dibujar cada uno de los *frames* deberemos desplazar la imagen de forma que el *frame* que queramos mostrar caiga justo bajo el área de recorte establecida. De esta forma al dibujar la imagen, se volcará a la pantalla sólo el *frame* deseado, y el resto, al estar fuera del área de recorte, no se mostrará. En la figura podemos ver los *frames* que quedan fuera del área de recorte representados con un color más claro, al volcar la imagen estos *frames* no se dibujarán.



A continuación se muestra el código fuente del ejemplo anterior, con el que podremos dibujar cada *frame* de la imagen.

```
// Guardamos el área de recorte anterior
int clipX = g.getClipX();
int clipY = g.getClipY();
int clipW = g.getClipWidth();
int clipH = g.getClipHeight();
```



```
// Establecemos nuevo área de recorte
g.clipRect(x, y, anchoFrame, altoFrame);

// Dibujamos la imagen con el desplazamiento adecuado
g.drawImage(imagen,
            x - xFrame[frameActual],
            y - yFrame[frameActual],
            Graphics.TOP | Graphics.LEFT);

// Reestablecemos el área de recorte anterior
g.setClip(clipX, clipY, clipW, clipH);
```

### 3.3.7.4. Eventos de entrada

La clase Canvas nos permite acceder a los eventos de entrada del usuario a bajo nivel. De esta forma podremos saber cuando el usuario pulsa o suelta cualquier tecla del dispositivo. Cuando ocurra un evento en el teclado se invocará uno de los siguientes métodos de la clase Canvas:

keyPressed(int cod)	Se ha presionado la tecla con código cod
keyRepeated(int cod)	Se mantiene presionada la tecla con código cod
keyReleased(int cod)	Se ha soltado la tecla con código cod

Estos dispositivos, además de generar eventos cuando presionamos o soltamos una tecla, son capaces de generar eventos de repetición. Estos eventos se producirán cada cierto período de tiempo mientras mantengamos pulsada una tecla.

Al realizar aplicaciones para móviles debemos tener en cuenta que en la mayoría de estos dispositivos no se puede presionar más de una tecla al mismo tiempo. Hasta que no hayamos soltado la tecla que estemos pulsando, no se podrán recibir eventos de pulsación de ninguna otra tecla.

Para dar respuesta a estos eventos del teclado deberemos redefinir estos métodos en nuestra subclase de Canvas:

```
public class MiCanvas extends Canvas {
    ...
    public void keyPressed(int cod) {
        // Se ha presionado la tecla con código cod
    }

    public void keyRepeated(int cod) {
        // Se mantiene pulsada la tecla con código cod
    }

    public void keyReleased(int cod) {
        // Se ha soltado la tecla con código cod
    }
}
```

### Códigos del teclado

Cada tecla del teclado del dispositivo tiene asociado un código identificativo que será el

parámetro que se le proporcione a estos métodos al presionarse o soltarse. Tenemos una serie de constantes en la clase `Canvas` que representan los códigos de las teclas estándar:

<code>Canvas.KEY_NUM0</code>	0
<code>Canvas.KEY_NUM1</code>	1
<code>Canvas.KEY_NUM2</code>	2
<code>Canvas.KEY_NUM3</code>	3
<code>Canvas.KEY_NUM4</code>	4
<code>Canvas.KEY_NUM5</code>	5
<code>Canvas.KEY_NUM6</code>	6
<code>Canvas.KEY_NUM7</code>	7
<code>Canvas.KEY_NUM8</code>	8
<code>Canvas.KEY_NUM9</code>	9
<code>Canvas.KEY_POUND</code>	#
<code>Canvas.KEY_STAR</code>	*

Los teclados, además de estas teclas estándar, normalmente tendrán otras teclas, cada una con su propio código numérico. Es recomendable utilizar únicamente estas teclas definidas como constantes para asegurar la portabilidad de la aplicación, ya que si utilizamos cualquier otro código de tecla no podremos asegurar que esté disponible en todos los modelos de teléfonos.

Los códigos de tecla corresponden al código Unicode del carácter correspondiente a dicha tecla. Si la tecla no corresponde a ningún carácter Unicode entonces su código será negativo. De esta forma podremos obtener fácilmente el carácter correspondiente a cada tecla. Sin embargo, esto no será suficiente para realizar entrada de texto, ya que hay caracteres que corresponden a múltiples pulsaciones de una misma tecla, y a bajo nivel sólo tenemos constancia de que una misma tecla se ha pulsado varias veces, pero no sabemos a qué carácter corresponde ese número de pulsaciones. Si necesitamos que el usuario escriba texto, lo más sencillo será utilizar uno de los componentes de alto nivel.

Podemos obtener el nombre de la tecla correspondiente a un código dado con el método `getKeyName` de la clase `Canvas`.

### Acciones de juegos

Tenemos también definidas lo que se conoce como acciones de juegos (*game actions*) con las que representaremos las teclas que se utilizan normalmente para controlar los juegos, a modo de *joystick*. Las acciones de juegos principales son:

Canvas.LEFT	Movimiento a la izquierda
Canvas.RIGHT	Movimiento a la derecha
Canvas.UP	Movimiento hacia arriba
Canvas.DOWN	Movimiento hacia abajo
Canvas.FIRE	Fuego

Una misma acción puede estar asociada a varias teclas del teléfono, de forma que el usuario pueda elegir la que le resulte más cómoda. Las teclas asociadas a cada acción de juego serán dependientes de la implementación, cada modelo de teléfono puede asociar a las teclas las acciones de juego que considere más apropiadas según la distribución del teclado, para que el manejo sea cómodo. Por lo tanto, el utilizar estas acciones hará la aplicación más portable, ya que no tendremos que adaptar los controles del juego para cada modelo de móvil.

Para conocer la acción de juego asociada a un código de tecla dado utilizaremos el siguiente método:

```
int accion = getGameAction(keyCode);
```

De esta forma podremos realizar de una forma sencilla y portable aplicaciones que deban controlarse utilizando este tipo de acciones.

Podemos hacer la transformación inversa con:

```
int codigo = getKeyCode(accion);
```

Hemos de resaltar que una acción de código puede estar asociada a más de una tecla, pero con este método sólo podremos obtener la tecla principal que realiza dicha acción.

### Punteros

Algunos dispositivos tienen punteros como dispositivos de entrada. Esto es común en los PDAs, pero no en los teléfonos móviles. Los *callbacks* que deberemos redefinir para dar respuesta a los eventos del puntero son los siguientes:

pointerPressed(int x, int y)	Se ha pinchado con el puntero en (x,y)
pointerDragged(int x, int y)	Se ha arrastrado el puntero a (x,y)
pointerReleased(int x, int y)	Se ha soltado el puntero en (x,y)

En todos estos métodos se proporcionarán las coordenadas (x,y) donde se ha producido el evento del puntero.

### 3.3.7.5. APIs propietarias

Existen APIs propietarias de diferentes vendedores, que añaden funcionalidades no soportadas por la especificación de MIDP. Los desarrolladores de estas APIs propietarias no deben incluir en ellas nada que pueda hacerse con MIDP. Estas APIs deben ser únicamente para permitir acceder a funcionalidades que MIDP no ofrece.

Es recomendable no utilizar estas APIs propietarias siempre que sea posible, para hacer aplicaciones que se ajusten al estándar de MIDP. Lo que podemos hacer es desarrollar aplicaciones que cumplan con el estándar MIDP, y en el caso que detecten que hay disponible una determinada API propietaria la utilicen para obtener alguna mejora. A continuación veremos como detectar en tiempo de ejecución si tenemos disponible una determinada API.

Vamos a ver la API Nokia UI, disponible en gran parte de los modelos de teléfonos Nokia, que incorpora nuevas funcionalidades para la programación de la interfaz de usuario no disponibles en MIDP 1.0. Esta API está contenida en el paquete `com.nokia.mid`.

## Gráficos

---

En cuanto a los gráficos, tenemos disponibles una serie de mejoras respecto a MIDP.

Añade soporte para crear un *canvas* a pantalla completa. Para crear este *canvas* utilizaremos la clase `FullCanvas` de la misma forma que utilizábamos `Canvas`.

Define una extensión de la clase `Graphics`, en la clase `DirectGraphics`. Para obtener este contexto gráfico extendido utilizaremos el siguiente método:

```
DirectGraphics dg = DirectUtils.getDirectGraphics(g);
```

Siendo `g` el objeto de contexto gráfico `Graphics` en el que queremos dibujar. Este nuevo contexto gráfico añade:

- Soporte para nuevos tipos de primitivas geométricas (triángulos y polígonos).
- Soporte para transparencia, incorporando un canal *alpha* al color. Ahora tenemos colores de 32 bits, cuya forma empaquetada se codifica como `0xAARRGGBB`.
- Acceso directo a los *pixels* del *raster* de pantalla. Podremos dibujar *pixels* en pantalla proporcionando directamente el *array* de *pixels* a dibujar, o bien obtener los *pixels* de la pantalla en forma de un *array* de *pixels*. Cada *pixel* de este *array* será un valor `int`, `short` o `byte` que codificará el color de dicho *pixel*.
- Permite transformar las imágenes a dibujar. Podremos hacer rotaciones de 90, 180 o 270 grados y transformaciones de espejo con las imágenes al mostrarlas.

## Sonido

---

Una limitación de MIDP 1.0 es que no soporta sonido. Por ello para incluir sonido en las aplicaciones de dispositivos que sólo soporten MIDP 1.0 como API estándar deberemos recurrir a APIs propietarias para tener estas funcionalidades. La API Nokia UI nos

permitirá solucionar esta carencia.

Nos permitirá reproducir sonidos como tonos o ficheros de onda (WAV). Los tipos de formatos soportados serán dependientes de cada dispositivo.

### Control del dispositivo

---

Además de las características anteriores, esta API nos permitirá utilizar funciones propias de los dispositivos. En la clase `DeviceControl` tendremos métodos para controlar la vibración del móvil y el parpadeo de las luces de la pantalla.

### Detección de la API propietaria

---

Si utilizamos una API propietaria reduciremos la portabilidad de la aplicación. Por ejemplo, si usamos la API Nokia UI la aplicación sólo funcionará en algunos dispositivos de Nokia. Hay una forma de utilizar estas APIs propietarias sin afectar a la portabilidad de la aplicación. Podemos detectar en tiempo de ejecución si la API propietaria está disponible de la siguiente forma:

```
boolean hayNokiaUI = false;
try {
    Class.forName("com.nokia.mid.sound.Sound");
    hayNokiaUI = true;
} catch(ClassNotFoundException e) {}
```

De esta forma, si la API propietaria está disponible podremos utilizarla para incorporar más funcionalidades a la aplicación. En caso contrario, no deberemos ejecutar nunca ninguna instrucción que acceda a esta API propietaria.

## 4. Herencia e interfaces. MIDlets e interfaz de usuario - Ejercicios

### 4.1. Menú básico

En el directorio `MenuBasico` tenemos implementada una aplicación básica en la que se muestra un menú típico de un juego mediante un `displayable` de tipo `List`.

- a) Consultar el código y probar la aplicación.
- b) Añadir una nueva opción al menú, de nombre "*Hi-score*".
- c) Probar cambiando a los distintos tipos de lista existentes.
- d) Añadir comandos a esta pantalla. Se pueden añadir los comandos "*OK*" y "*Salir*".

### 4.2. Adivina el número (I) (\*)

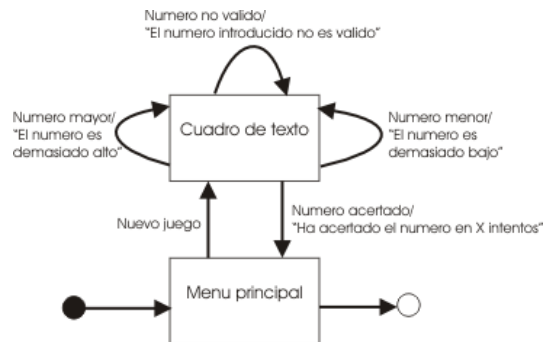
Vamos a implementar un juego consistente en adivinar un número del 1 a 100. Como para previo, vamos a crear el menú principal de nuestro juego, en el que deberemos tener las opciones *Nuevo juego* y *Salir*.

- a) ¿Qué tipo de *displayable* utilizaremos para realizar este menú?
- b) Implementar esta pantalla encapsulando todo su contenido en una misma clase.
- c) Añadir un comando que nos permita seleccionar la opción marcada del menú.
- d) Incorporar un *listener* de comandos para dar respuesta a este comando de selección de opciones. Por ahora lo que haremos será mostrar una alerta que diga "*Opcion no implementada todavia*".

### 4.3. Adivina el número (II) (\*)

Implementar el juego de adivinar un número del 1 al 100. Para ello partiremos de la base realizada en el ejercicio anterior.

El juego pensará un número aleatorio de 1 a 100, y se mostrará al usuario un cuadro de texto donde deberá introducir el número del que piensa que se trata. Una vez introducido, pulsará OK y entonces la aplicación le dirá si el número es demasiado alto, si es demasiado bajo o si ha acertado. En caso de que el número sea muy alto o muy bajo, volveremos al mismo cuadro de texto para volver a probar. Si ha acertado el juego finalizará, mostrando el número de intentos que ha necesitado y volviendo al menú principal.



Para implementar esta aplicación crearemos una nueva pantalla encapsulada en una clase de nombre `EntradaTexto` que será de tipo `TextBox`, donde el usuario introducirá el número. Al construir esta pantalla se deberá determinar un número aleatorio de 1 a 100, cosa que podemos hacer de la siguiente forma:

```
Random rand = new Random();
this.numero = Math.abs(rand.nextInt()) % 100 + 1;
```

Deberemos añadir un comando para que el usuario notifique que ha introducido el número. Como respuesta a este comando deberemos obtener el número que ha introducido el usuario y compararlo con el número aleatorio. Según si el número es menor, mayor o igual mostraremos una alerta con el mensaje correspondiente, y volveremos a la pantalla actual o al menú principal según si el usuario ha fallado o acertado respectivamente.

#### 4.4. TeleSketch

Vamos a implementar una aplicación similar al juego que se conocía como *"TeleSketch"*. Esta aplicación nos deberá permitir dibujar en la pantalla utilizando las teclas de los cursores.

La idea es dibujar en *offscreen* (en una imagen mutable), de forma que no se pierda el contenido dibujado. En cada momento conoceremos la posición actual del cursor, donde dibujaremos un punto (puede ser un círculo o rectángulo de tamaño reducido). Al pulsar las teclas de los cursores del móvil moveremos este cursor por la pantalla haciendo que deje rastro, y de esta manera se irá generando el dibujo.

Tenemos una plantilla en el directorio `TeleSketch`. Sobre esta plantilla deberemos realizar lo siguiente:

- En el constructor de la clase deberemos crear una imagen mutable donde dibujar, con el tamaño del `Canvas`.
- En el método `actualiza` deberemos dibujar un punto en la posición actual del cursor y llamar a `repaint` para repintar el contenido de la pantalla.
- En `paint` deberemos volcar el contenido de la imagen *offscreen* a la pantalla.

*d)* Deberemos definir los eventos `keyPressed` y `keyRepeated` para mover el cursor cada vez que se pulsen las teclas arriba, abajo, izquierda y derecha. Podemos utilizar las acciones de juegos (game actions) para conocer cuáles son estas teclas.



## 5. Excepciones e hilos. Acceso a la red

Las excepciones e hilos son imprescindibles para las aplicaciones que se comunican por red y que al mismo tiempo deben responder a la interfaz gráfica de usuario. Ambos mecanismos están presentes tanto en Java como en JavaME. Sin embargo la librería de comunicación por red es un poco más reducida en JavaME, y es la que se estudia en esta sesión.

### 5.1. Excepciones

#### 5.1.1. Introducción

Las excepciones son eventos que ocurren durante la ejecución de un programa y hacen que éste salga de su flujo normal de instrucciones. Este mecanismo permite tratar los errores de una forma elegante, ya que separa el código para el tratamiento de errores del código normal del programa. Se dice que una excepción es *lanzada* cuando se produce un error, y esta excepción puede ser *capturada* para tratar dicho error.

#### 5.1.2. Tipos de excepciones

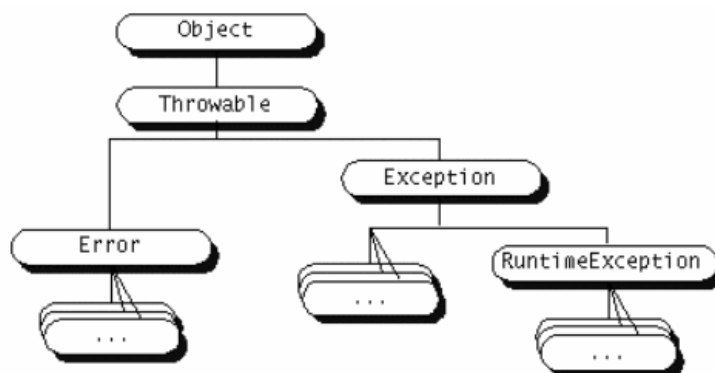
Tenemos diferentes tipos de excepciones dependiendo del tipo de error que representen. Todas ellas descienden de la clase `Throwable`, la cual tiene dos descendientes directos:

- **Error:** Se refiere a errores graves en la máquina virtual de Java, como por ejemplo fallos al enlazar con alguna librería. Normalmente en los programas Java no se tratarán este tipo de errores.
- **Exception:** Representa errores que no son críticos y por lo tanto pueden ser tratados y continuar la ejecución de la aplicación. La mayoría de los programas Java utilizan estas excepciones para el tratamiento de los errores que puedan ocurrir durante la ejecución del código.

Dentro de `Exception`, cabe destacar una subclase especial de excepciones denominada `RuntimeException`, de la cual derivarán todas aquellas excepciones referidas a los errores que comúnmente se pueden producir dentro de cualquier fragmento de código, como por ejemplo hacer una referencia a un puntero `null`, o acceder fuera de los límites de un `array`.

Estas `RuntimeException` se diferencian del resto de excepciones en que no son de tipo *checked*. Una excepción de tipo *checked* debe ser capturada o bien especificar que puede ser lanzada de forma obligatoria, y si no lo hacemos obtendremos un error de compilación. Dado que las `RuntimeException` pueden producirse en cualquier fragmento de código, sería impensable tener que añadir manejadores de excepciones y declarar que éstas pueden ser lanzadas en todo nuestro código. Deberemos:

- Utilizar excepciones *unchecked* (no predecibles) para indicar errores graves en la lógica del programa, que normalmente no deberían ocurrir. Se utilizarán para comprobar la consistencia interna del programa.
- Utilizar excepciones *checked* para mostrar errores que pueden ocurrir durante la ejecución de la aplicación, normalmente debidos a factores externos como por ejemplo la lectura de un fichero con formato incorrecto, un fallo en la conexión, o la entrada de datos por parte del usuario.



Tipos de excepciones

Dentro de estos grupos principales de excepciones podremos encontrar tipos concretos de excepciones o bien otros grupos que a su vez pueden contener más subgrupos de excepciones, hasta llegar a tipos concretos de ellas. Cada tipo de excepción guardará información relativa al tipo de error al que se refiera, además de la información común a todas las excepciones. Por ejemplo, una `ParseException` se suele utilizar al procesar un fichero. Además de almacenar un mensaje de error, guardará la línea en la que el *parser* encontró el error.

### 5.1.3. Captura de excepciones

Cuando un fragmento de código sea susceptible de lanzar una excepción y queramos tratar el error producido o bien por ser una excepción de tipo *checked* debamos capturarla, podremos hacerlo mediante la estructura `try-catch-finally`, que consta de tres bloques de código:

- Bloque `try`: Contiene el código regular de nuestro programa que puede producir una excepción en caso de error.
- Bloque `catch`: Contiene el código con el que trataremos el error en caso de producirse.
- Bloque `finally`: Este bloque contiene el código que se ejecutará al final tanto si se ha producido una excepción como si no lo ha hecho. Este bloque se utiliza para, por ejemplo, cerrar algún fichero que haya podido ser abierto dentro del código regular del programa, de manera que nos aseguremos que tanto si se ha producido un error como si no este fichero se cierre. El bloque `finally` no es obligatorio ponerlo.

Para el bloque `catch` además deberemos especificar el tipo o grupo de excepciones que tratamos en dicho bloque, pudiendo incluir varios bloques `catch`, cada uno de ellos para un tipo/grupo de excepciones distinto. La forma de hacer esto será la siguiente:

```
try {
    // Código regular del programa
    // Puede producir excepciones
} catch(TipoDeExcepcion1 e1) {
    // Código que trata las excepciones de tipo
    // TipoDeExcepcion1 o subclases de ella.
    // Los datos sobre la excepción los encontraremos
    // en el objeto e1.
} catch(TipoDeExcepcion2 e2) {
    // Código que trata las excepciones de tipo
    // TipoDeExcepcion2 o subclases de ella.
    // Los datos sobre la excepción los encontraremos
    // en el objeto e2.
...
} catch(TipoDeExcepcionN eN) {
    // Código que trata las excepciones de tipo
    // TipoDeExcepcionN o subclases de ella.
    // Los datos sobre la excepción los encontraremos
    // en el objeto eN.
} finally {
    // Código de finalización (opcional)
}
```

Si como tipo de excepción especificamos un grupo de excepciones este bloque se encargará de la captura de todos los subtipos de excepciones de este grupo. Por lo tanto, si especificamos `Exception` capturaremos cualquier excepción, ya que está es la superclase común de todas las excepciones.

En el bloque `catch` pueden ser útiles algunos métodos de la excepción (que podemos ver en la API de la clase padre `Exception`):

```
String getMessage()
void printStackTrace()
```

con `getMessage` obtenemos una cadena descriptiva del error (si la hay). Con `printStackTrace` se muestra por la salida estándar la traza de errores que se han producido (en ocasiones la traza es muy larga y no puede seguirse toda en pantalla con algunos sistemas operativos).

Un ejemplo de uso:

```
try {
    ... // Aqui va el codigo que puede lanzar una excepcion
} catch (Exception e) {
    System.out.println ("El error es: " + e.getMessage());
    e.printStackTrace();
}
```

Nunca deberemos dejar vacío el cuerpo del `catch`, porque si se produce el error, nadie se va a dar cuenta de que se ha producido. En especial, cuando estemos con excepciones *no-checked*.

### 5.1.4. Lanzamiento de excepciones

Hemos visto cómo capturar excepciones que se produzcan en el código, pero en lugar de capturarlas también podemos hacer que se propaguen al método de nivel superior (desde el cual se ha llamado al método actual). Para esto, en el método donde se vaya a lanzar la excepción, se siguen 2 pasos:

- Indicar en el método que determinados tipos de excepciones o grupos de ellas pueden ser lanzados, cosa que haremos de la siguiente forma, por ejemplo:

```
public void lee_fichero()
    throws IOException, FileNotFoundException
{
    // Cuerpo de la función
}
```

Podremos indicar tantos tipos de excepciones como queramos en la cláusula `throws`. Si alguna de estas clases de excepciones tiene subclases, también se considerará que puede lanzar todas estas subclases.

- Para lanzar la excepción utilizamos la instrucción `throw`, proporcionándole un objeto correspondiente al tipo de excepción que deseamos lanzar. Por ejemplo:

```
throw new IOException(mensaje_error);
```

- Juntando estos dos pasos:

```
public void lee_fichero()
    throws IOException, FileNotFoundException
{
    ...
    throw new IOException(mensaje_error);
    ...
}
```

Podremos lanzar así excepciones en nuestras funciones para indicar que algo no es como debiera ser a las funciones llamadas. Por ejemplo, si estamos procesando un fichero que debe tener un determinado formato, sería buena idea lanzar excepciones de tipo `ParseException` en caso de que la sintaxis del fichero de entrada no sea correcta.

NOTA: para las excepciones que no son de tipo *checked* no hará falta la cláusula *throws* en la declaración del método, pero seguirán el mismo comportamiento que el resto, si no son capturadas pasarán al método de nivel superior, y seguirán así hasta llegar a la función principal, momento en el que si no se captura provocará la salida de nuestro programa mostrando el error correspondiente.

### 5.1.5. Creación de nuevas excepciones

Además de utilizar los tipos de excepciones contenidos en la distribución de Java, podremos crear nuevos tipos que se adapten a nuestros problemas.

Para crear un nuevo tipo de excepciones simplemente deberemos crear una clase que herede de `Exception` o cualquier otro subgrupo de excepciones existente. En esta clase podremos añadir métodos y propiedades para almacenar información relativa a nuestro tipo de error. Por ejemplo:

```
public class MiExcepcion extends Exception
{
    public MiExcepcion (String mensaje)
    {
        super(mensaje);
    }
}
```

Además podremos crear subclases de nuestro nuevo tipo de excepción, creando de esta forma grupos de excepciones. Para utilizar estas excepciones (capturarlas y/o lanzarlas) hacemos lo mismo que lo explicado antes para las excepciones que se tienen definidas en Java.

### 5.1.6. Nested exceptions

Cuando dentro de un método de una librería se produce una excepción, normalmente se propagará dicha excepción al llamador en lugar de gestionar el error dentro de la librería, para que de esta forma el llamador tenga constancia de que se ha producido un determinado error y pueda tomar las medidas que crea oportunas en cada momento. Para pasar esta excepción al nivel superior puede optar por propagar la misma excepción que le ha llegado, o bien crear y lanzar una nueva excepción. En este segundo caso la nueva excepción deberá contener la excepción anterior, ya que de no ser así perderíamos la información sobre la causa que ha producido el error dentro de la librería, que podría sernos de utilidad para depurar la aplicación. Para hacer esto deberemos proporcionar la excepción que ha causado el error como parámetro del constructor de nuestra nueva excepción:

```
public class MiExcepcion extends Exception
{
    public MiExcepcion (String mensaje, Throwable causa)
    {
        super(mensaje, causa);
    }
}
```

En el método de nuestra librería en el que se produzca el error deberemos capturar la excepción que ha causado el error y lanzar nuestra propia excepción al llamador:

```
try {
    ...
} catch(IOException e) {
    throw new MiExcepcion("Mensaje de error", e);
}
```

Cuando capturemos una excepción, podemos consultar la excepción previa que la ha causado (si existe) con el método:

```
Exception causa = (Exception)e.getCause();
```

Las *nested exceptions* son útiles para:

- Encadenar errores producidos en la secuencia de métodos a los que se ha llamado.
- Facilitan la depuración de la aplicación, ya que nos permite conocer de dónde viene el error y por qué métodos ha pasado.
- El lanzar una excepción propia de cada método permite ofrecer información más detallada que si utilizásemos una única excepción genérica. Por ejemplo, aunque en varios casos el origen del error puede ser una `IOException`, nos será de utilidad saber si ésta se ha producido al guardar un fichero de datos, al guardar datos de la configuración de la aplicación, al intentar obtener datos de la red, etc.
- Aislar al llamador de la implementación concreta de una librería. Por ejemplo, cuando utilicemos los objetos de acceso a datos de nuestra aplicación, en caso de error recibiremos una excepción propia de nuestra capa de acceso a datos, en lugar de una excepción propia de la implementación concreta de esta capa, como pudiera ser `SQLException` si estamos utilizando una BD SQL o `IOException` si estamos accediendo a ficheros.

## 5.2. Hilos

Un hilo es un flujo de control dentro de un programa. Creando varios hilos podremos realizar varias tareas simultáneamente. Cada hilo tendrá sólo un contexto de ejecución (contador de programa, pila de ejecución). Es decir, a diferencia de los procesos UNIX, no tienen su propio espacio de memoria sino que acceden todos al mismo espacio de memoria común, por lo que será importante su sincronización cuando tengamos varios hilos accediendo a los mismos objetos.

### 5.2.1. Creación de hilos

En Java los hilos están encapsulados en la clase **Thread**. Para crear un hilo tenemos dos posibilidades:

- Heredar de **Thread** redefiniendo el método `run()`.
- Crear una clase que implemente la interfaz **Runnable** que nos obliga a definir el método `run()`.

En ambos casos debemos definir un método `run()` que será el que contenga el código del hilo. Desde dentro de este método podremos llamar a cualquier otro método de cualquier objeto, pero este método `run()` será el método que se invoque cuando iniciemos la ejecución de un hilo. El hilo terminará su ejecución cuando termine de ejecutarse este método `run()`.

Para crear nuestro hilo mediante herencia haremos lo siguiente:

```
public class EjemploHilo extends Thread
{
```

```

        public void run()
        {
            // Código del hilo
        }
    }

```

Una vez definida la clase de nuestro hilo deberemos instanciarlo y ejecutarlo de la siguiente forma:

```

Thread t = new EjemploHilo();
t.start();

```

Al llamar al método *start* del hilo, comenzará ejecutarse su método *run*. Crear un hilo heredando de **Thread** tiene el problema de que al no haber herencia múltiple en Java, si heredamos de **Thread** no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

Este problema desaparece si utilizamos la interfaz **Runnable** para crear el hilo, ya que una clase puede implementar varios interfaces. Definiremos la clase que contenga el hilo como se muestra a continuación:

```

public class EjemploHilo implements Runnable
{
    public void run()
    {
        // Código del hilo
    }
}

```

Para instanciar y ejecutar un hilo de este tipo deberemos hacer lo siguiente:

```

Thread t = new Thread(new EjemploHilo());
t.start();

```

Esto es así debido a que en este caso **EjemploHilo** no deriva de una clase **Thread**, por lo que no se puede considerar un hilo, lo único que estamos haciendo implementando la interfaz es asegurar que vamos a tener definido el método *run()*. Con esto lo que haremos será proporcionar esta clase al constructor de la clase **Thread**, para que el objeto **Thread** que creamos llame al método *run()* de la clase que hemos definido al iniciarse la ejecución del hilo, ya que implementando la interfaz le aseguramos que esta función existe.

### 5.2.2. Estado y propiedades de los hilos

Un hilo pasará por varios estados durante su ciclo de vida.

```

Thread t = new Thread(this);

```

Una vez se ha instanciado el objeto del hilo, diremos que está en estado de *Nuevo hilo*.

```

t.start();

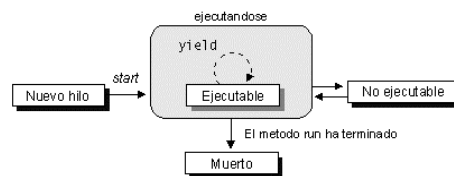
```

Cuando invoquemos su método *start()* el hilo pasará a ser un hilo *vivo*, comenzándose a ejecutar su método *run()*. Una vez haya salido de este método pasará a ser un hilo *muerto*.

La única forma de parar un hilo es hacer que salga del método *run()* de forma natural. Podremos conseguir esto haciendo que se cumpla una condición de salida de *run()* (lógicamente, la condición que se nos ocurra dependerá del tipo de programa que estemos haciendo). Las funciones para parar, pausar y reanudar hilos están desaprobadas en las versiones actuales de Java.

Mientras el hilo esté *vivo*, podrá encontrarse en dos estados: *Ejecutable* y *No ejecutable*. El hilo pasará de *Ejecutable* a *No ejecutable* en los siguientes casos:

- Cuando se encuentre dormido por haberse llamado al método *sleep()*, permanecerá *No ejecutable* hasta haber transcurrido el número de milisegundos especificados.
- Cuando se encuentre bloqueado en una llamada al método *wait()* esperando que otro hilo lo desbloquee llamando a *notify()* o *notifyAll()*. Veremos cómo utilizar estos métodos más adelante.
- Cuando se encuentre bloqueado en una petición de E/S, hasta que se complete la operación de E/S.



Ciclo de vida de los hilos

Lo único que podremos saber es si un hilo se encuentra vivo o no, llamando a su método *isAlive()*.

### Prioridades de los hilos

Además, una propiedad importante de los hilos será su prioridad. Mientras el hilo se encuentre vivo, el *scheduler* de la máquina virtual Java le asignará o lo sacará de la CPU, coordinando así el uso de la CPU por parte de todos los hilos activos basándose en su prioridad. Se puede forzar la salida de un hilo de la CPU llamando a su método *yield()*. También se sacará un hilo de la CPU cuando un hilo de mayor prioridad se haga *Ejecutable*, o cuando el tiempo que se le haya asignado expire.

Para cambiar la prioridad de un hilo se utiliza el método *setPriority()*, al que deberemos proporcionar un valor de prioridad entre *MIN\_PRIORITY* y *MAX\_PRIORITY* (tenéis constantes de prioridad disponibles dentro de la clase *Thread*, consultad el API de Java para ver qué valores de constantes hay).

### Hilo actual

En cualquier parte de nuestro código Java podemos llamar al método *currentThread* de la clase *Thread*, que nos devuelve un objeto hilo con el hilo que se encuentra actualmente ejecutando el código donde está introducido ese método. Por ejemplo, si tenemos un código como:



```
public class EjemploHilo implements Runnable
{
    public EjemploHilo()
    {
        ...
        int i = 0;
        Thread t = Thread.currentThread();
        t.sleep(1000);
    }
}
```

La llamada a *currentThread* dentro del constructor de la clase nos devolverá el hilo que corresponde con el programa principal (puesto que no hemos creado ningún otro hilo, y si lo creáramos, no ejecutaría nada que no estuviese dentro de un método *run*).

Sin embargo, en este otro caso:

```
public class EjemploHilo implements Runnable
{
    public EjemploHilo()
    {
        Thread t1 = new Thread(this);
        Thread t2 = new Thread(this);
        t1.start();
        t2.start();
    }

    public void run()
    {
        int i = 0;
        Thread t = Thread.currentThread();
        t.sleep(1000);
    }
}
```

Lo que hacemos es crear dos hilos auxiliares, y la llamada a *currentThread* se produce dentro del *run*, con lo que se aplica a los hilos auxiliares, que son los que ejecutan el *run*: primero devolverá un hilo auxiliar (el que primero entre, t1 o t2), y luego el otro (t2 o t1).

### Dormir hilos

Como hemos visto en los ejemplos anteriores, una vez obtenemos el hilo que queremos, el método *sleep* nos sirve para dormirlo, durante los milisegundos que le pasemos como parámetro (en los casos anteriores, dormían durante 1 segundo). El tiempo que duerme el hilo, deja libre el procesador para que lo ocupen otros hilos. Es una forma de no sobrecargar mucho de trabajo a la CPU con muchos hilos intentando entrar sin descanso.

### 5.2.3. Sincronización de hilos

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

Un primer mecanismo de comunicación es la variable cerrojo incluida en todo objeto **Object**, que permitirá evitar que más de un hilo entre en la sección crítica para un objeto determinado. Los métodos declarados como *synchronized* utilizan el cerrojo del objeto al

que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

```
public synchronized void seccion_critica()
{ // Código sección crítica }
```

Todos los métodos *synchronized* de un mismo objeto (no clase, sino objeto de esa clase), comparten el mismo cerrojo, y es distinto al cerrojo de otros objetos (de la misma clase, o de otras).

También podemos utilizar cualquier otro objeto para la sincronización dentro de nuestro método de la siguiente forma:

```
synchronized (objeto_con_cerrojo)
{ // Código sección crítica }
```

de esta forma sincronizaríamos el código que escribiésemos dentro, con el código *synchronized* del objeto *objeto\_con\_cerrojo*.

Además podemos hacer que un hilo quede bloqueado a la espera de que otro hilo lo desbloquee cuando suceda un determinado evento. Para bloquear un hilo usaremos la función *wait()*, para lo cual el hilo que llama a esta función debe estar en posesión del monitor, cosa que ocurre dentro de un método *synchronized*, por lo que sólo podremos bloquear a un proceso dentro de estos métodos.

Para desbloquear a los hilos que haya bloqueados se utilizará *notifyAll()*, o bien *notify()* para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor.

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la sección crítica del objeto y desbloquearlo.

Por último, puede ser necesario esperar a que un determinado hilo haya finalizado su tarea para continuar. Esto lo podremos hacer llamando al método *join()* de dicho hilo, que nos bloqueará hasta que el hilo haya finalizado.

### 5.3. Conexiones de red

En J2SE tenemos una gran cantidad de clases en el paquete `java.net` para permitir establecer distintos tipos de conexiones en red. Sin embargo, el soportar esta gran API no es viable en la configuración CLDC dedicada a dispositivos muy limitados. Por lo tanto en CLDC se sustituye esta API por el marco de conexiones genéricas (GCF, Generic Connection Framework), con el que se pretenden cubrir todas las necesidades de conectividad de estos dispositivos a través de una API sencilla.

#### 5.3.1. Marco de conexiones genéricas

Los distintos dispositivos móviles pueden utilizar distintos tipos de redes para conectarse. Algunos utilizan redes de conmutación de circuitos, orientadas a conexión, que

necesitarán protocolos como TCP. Otros utilizan redes de transmisión de paquetes en las que no se establece una conexión permanente, y con las que deberemos trabajar con protocolos como por ejemplo UDP. Incluso otros dispositivos podrían utilizar otras redes distintas en las que debamos utilizar otro tipo de protocolos.

El marco de conexiones genéricas (GFC) hará que esta red móvil subyacente sea transparente para el usuario, proporcionando a éste protocolos estándar de comunicaciones. La API de GFC se encuentra en el paquete `javax.microedition.io`. Esta API utilizará un único método que nos servirá para establecer cualquier tipo de conexión que queramos, por esta razón recibe el nombre de marco de conexiones genéricas, lo cuál además lo hace extensible para incorporar nuevos tipos de conexiones. Para crear la conexión utilizaremos el siguiente método:

```
Connection con = Connector.open(url);
```

En el que deberemos especificar una URL como parámetro con el siguiente formato:

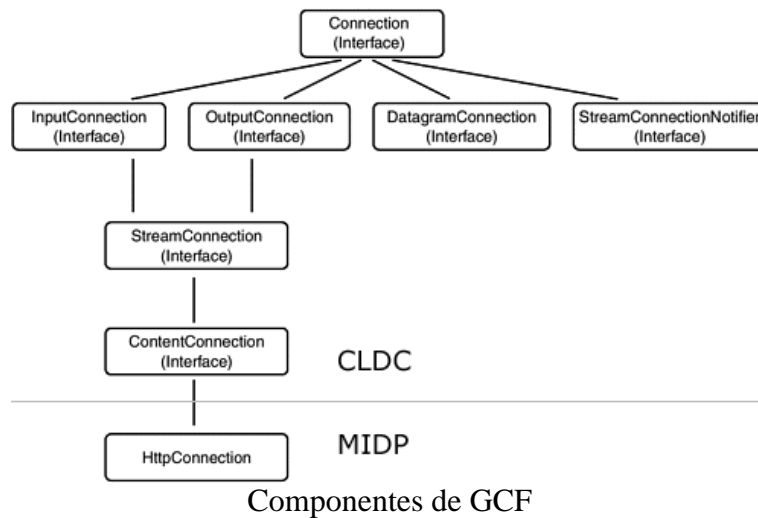
`protocolo:direccion;parámetros`

Cambiando el protocolo podremos especificar distintos tipos de conexiones. Por ejemplo, podríamos utilizar las siguientes URLs:

"http://jtech.ua.es/pdm"	Abre una conexión HTTP.
"datagram://192.168.0.4:6666"	Abre una conexión por datagramas.
"socket://192.168.0.4:4444"	Abre una conexión por <i>sockets</i> .
"comm:0;baudrate=9600"	Abre una conexión a través de un puerto de comunicaciones.
"file:/fichero.txt"	Abre un fichero.

Cuando especifiquemos uno de estos protocolos, la clase `Connector` buscará en tiempo de ejecución la clase que implemente dicho tipo de conexión, y si la encuentra nos devolverá un objeto que implemente la interfaz `Connection` que nos permitirá comunicarnos a través de dicha conexión.

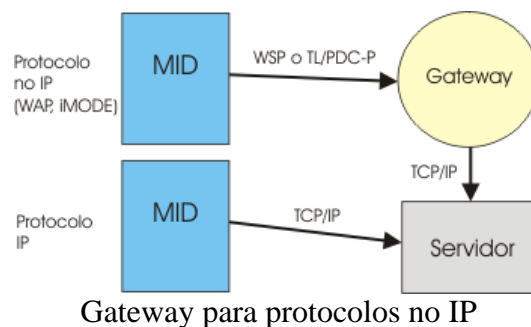
CLDC nos proporciona interfaces para cada tipo genérico de conexión, pero las implementaciones reales de los protocolos pertenecen a los perfiles.



El único protocolo que la especificación de MIDP exige que se implemente es el protocolo HTTP. Este protocolo pertenece a MIDP, y no a CLDC como era el caso de las clases genéricas anteriores. Distintos modelos de dispositivos pueden soportar otro tipo de conexiones, pero si queremos hacer aplicaciones portables deberemos utilizar HTTP.

### 5.3.2. Conexión HTTP

La conexión mediante el protocolo HTTP es el único tipo de conexión que sabemos que va a estar soportado por todos los dispositivos MIDP. Este protocolo podrá ser implementado en cada modelo de móvil bien utilizando protocolos IP como TCP/IP o bien protocolos no IP como WAP o i-Mode.



De esta forma nosotros podremos utilizar directamente HTTP de una forma estándar sin importarnos el tipo de red que el móvil tenga por debajo.

Cuando establezcamos una conexión mediante protocolo HTTP, podemos hacer una conversión *cast* del objeto `Connection` devuelto a un subtipo `HttpConnection` especializado en conexiones HTTP:

```

HttpConnection httpConn = (HttpConnection) conn;

```

```
HttpConnection con =  
(HttpConnection)Connector.open("http://jtech.ua.es/datos.txt");
```

Este objeto `HttpConnection` contiene gran cantidad de métodos dedicados a trabajar con el protocolo HTTP, lo cuál facilitará en gran medida el trabajo de los desarrolladores.

HTTP es un protocolo de petición/respuesta. El cliente crea un mensaje de petición y lo envía a una determinada URL. El servidor analizará esta petición y le devolverá una respuesta al cliente. Estos mensajes de petición y respuesta se compondrán de una serie de cabeceras y del bloque de contenido. Cada cabecera tendrá un nombre y un valor. El contenido podrá contener cualquier tipo de información (texto, HTML, imágenes, mensajes codificados en binario, etc). Tendremos una serie de cabeceras estándar con las que podremos intercambiar datos sobre el cliente o el servidor, o bien sobre la información que estamos transmitiendo. También podremos añadir nuestras propias cabeceras para intercambiar datos propios.

Una vez creada la conexión, ésta pasará por tres estados:

- **Configuración:** No se ha establecido la conexión, todavía no se ha enviado el mensaje de petición. Este será el momento en el que deberemos añadir la información necesaria a las cabeceras del mensaje de petición.
- **Conectada:** El mensaje de petición ya se ha enviado, y se espera recibir una respuesta. En este momento podremos leer las cabeceras o el contenido de la respuesta.
- **Cerrada:** La conexión se ha cerrado y ya no podemos hacer nada con ella.

La conexión nada más crearse se encuentra en estado de configuración. Pasará automáticamente a estado conectada cuando solicitemos cualquier información sobre la respuesta.

#### 5.3.2.1. Lectura de la respuesta

Vamos a comenzar viendo cómo leer el contenido de una URL. En este caso no vamos a añadir ninguna información al mensaje de petición, ya que no es necesario. Sólo queremos obtener el contenido del recurso solicitado en la URL.

Imaginemos que queremos leer el fichero en la URL `http://jtech.ua.es/datos.txt`. Como primer paso deberemos crear una conexión con dicha URL como hemos visto anteriormente. Una vez tengamos este objeto `HttpConnection` abriremos un flujo de entrada para leer su contenido de la siguiente forma:

```
InputStream in = con.openInputStream();
```

Una vez hecho esto, la conexión pasará a estado conectada, ya que estamos solicitando leer su contenido. Por lo tanto en este momento será cuando envíe el mensaje de petición al servidor, y se quede esperando a recibir la respuesta. Con el flujo de datos obtenido podremos leer el contenido de la misma, al igual que leemos cualquier otro flujo de datos

en Java.

Dado que en este momento ya se ha enviado el mensaje de petición, ya no tendrá sentido realizar modificaciones en la petición. Es por esta razón por lo que la creación del mensaje de petición debe hacerse en el estado de configuración.

Una vez hayamos terminado de leer la respuesta, deberemos cerrar el flujo y la conexión:

```
in.close();
con.close();
```

Con esto la conexión pasará a estado cerrada, liberando todos los recursos.

### 5.3.2.2. Mensaje de petición

En muchos casos podemos necesitar enviar información al servidor, como por ejemplo el login y el password del usuario para autenticarse en la aplicación web. Esta información deberemos incluirla en el mensaje de petición. Existen distintas formas de enviar información en la petición.

Encontramos los diferentes tipos de mensajes de petición soportados por MIDP:

HttpConnection.GET	Los parámetros que se envían al servidor se incluyen en la misma URL. Por ejemplo, podemos mandar un parámetro login en la petición de la siguiente forma: <code>http://www.jtech.ua.es/dadm?login=miguel</code>
HttpConnection.POST	Los parámetros que se envían al servidor se incluyen como contenido del mensaje. Tiene la ventaja de que se puede enviar la cantidad de datos que queramos, a diferencia del método GET en el que esta cantidad puede estar limitada. Además los datos no serán visibles en la misma URL, ya que se incluyen como contenido del mensaje.
HttpConnection.HEAD	No se solicita el contenido del recurso al servidor, sólo información sobre éste, es decir, las cabeceras HTTP.

Podemos establecer uno de estos tipos utilizando el método `setRequestMethod`, por ejemplo para utilizar una petición POST haremos lo siguiente:

```
con.setRequestMethod(HttpConnection.POST);
```

Además podremos añadir cabeceras a la petición con el siguiente método:

```
con.setRequestProperty(nombre, valor);
```

Por ejemplo, podemos mandar las siguiente cabeceras:

```
c.setRequestProperty("IF-Modified-Since",  
    "22 Sep 2002 08:00:00 GMT");  
c.setRequestProperty("User-Agent",  
    "Profile/MIDP-1.0 Configuration/CLDC-1.0");  
c.setRequestProperty("Content-Language", "es-ES");
```

Con esto estaremos diciendo al servidor que queremos que nos devuelva una respuesta sólo si ha sido modificada desde la fecha indicada, y además le estamos comunicando datos sobre el cliente. Indicamos mediante estas cabeceras estándar que el cliente es una aplicación MIDP, y que el lenguaje es español de España.

### 5.3.2.3. Envío de datos en la petición

Cuando necesitemos enviar datos al servidor mediante HTTP mediante nuestra aplicación Java, podemos simular el envío de datos que realiza un formulario HTML. Podremos simular tanto el comportamiento de un formulario que utilice método GET como uno que utilice método POST.

En el caso del método GET, simplemente utilizaremos una petición de tipo `HttpConnection.GET` e incluiremos estos datos codificados en la URL. Por ejemplo, si estamos registrando los datos de un usuario (nombre, apellidos y edad) podemos incluir estos parámetros en la URL de la siguiente forma:

```
HttpConnection con =  
    (HttpConnection)Connector.open("http://www.jtech.ua.es/aplic"  
+  
    "/registraUsuario?nombre=Pedro&apellidos=Lopez+Garcia&edad=25");
```

Cada parámetro tiene la forma `nombre=valor`, pudiendo incluir varios parámetros separados por el carácter '&'. Como en la URL no puede haber espacios, estos caracteres se sustituyen por el carácter '+' como podemos ver en el ejemplo.

En el caso de que queramos simular un formulario con método POST, utilizamos una petición de tipo `HttpConnection.POST` y deberemos incluir los parámetros que enviemos al servidor como contenido del mensaje. Para ello deberemos indicar que el tipo de contenido de la petición es `application/x-www-form-urlencoded`, y como contenido codificaremos los parámetros de la misma forma que se utiliza para codificarlos en la URL cuando se hace una petición GET:

```
nombre=Pedro&apellidos=Lopez+Garcia&edad=25
```

De esta forma podemos enviar al servidor datos en forma de una serie de parámetros que toman como valor cadenas de texto. Sin embargo, puede que necesitemos intercambiar datos más complejos con el servidor. Por ejemplo, podemos querer serializar objetos Java y enviarlos al servidor, o enviar documentos XML.

Para enviar estos tipos de información podemos utilizar también el bloque de contenido, debiendo especificar en cada caso el tipo MIME del contenido que vamos a añadir. Ejemplos de tipos MIME que podemos utilizar para el bloque de contenido son:

<code>application/x-www-form-urlencoded</code>	Se envían los datos codificados de la misma forma en la que son codificados por un formulario HTML con método POST.
<code>text/plain</code>	Se envía como contenido texto ASCII.
<code>application/octet-stream</code>	Se envía como contenido datos binarios. Dentro de la secuencia de bytes podremos codificar la información como queramos. Por ejemplo, podemos codificar de forma binaria un objeto serializado, utilizando un <code>DataOutputStream</code> .

Para establecer el tipo de contenido la cabecera estándar de HTTP `Content-Type`. Por ejemplo, si añadimos texto ASCII, podemos establecer esta cabecera de la siguiente forma:

```
con.setRequestProperty("Content-Type", "text/plain");
```

Para escribir en el contenido del mensaje de petición deberemos abrir un flujo de salida como se muestra a continuación:

```
OutputStream out = con.openOutputStream();
```

Podremos escribir en este flujo de salida igual que lo hacemos en cualquier otro flujo de salida, con lo que de esta forma podremos escribir cualquier contenido en el mensaje de petición.

Al abrir el flujo para escribir en la petición provocaremos que se pase a estado conectado. Por lo tanto deberemos haber establecido el tipo de petición y todas las cabeceras previamente a la apertura de este flujo, cuando todavía estábamos en estado de configuración.

#### 5.3.2.4. Tipo y cabeceras de la respuesta

En estado conectado, además del contenido del mensaje de la respuesta, podemos obtener el estado de la respuesta y sus cabeceras. Los estados de respuesta se componen de un código y un mensaje y nos permitirán saber si la petición ha podido atenderse correctamente o si por el contrario ha habido algún tipo de error. Por ejemplo, posibles



estados son:

<code>HttpConnection.HTTP_OK</code>	200	OK
<code>HttpConnection.HTTP_BAD_</code>	400	Bad Request
<code>HttpConnection.HTTP_INTE</code>	500	Internal Server Error

Este mensaje de estado encabeza el mensaje de respuesta. Si el servidor nos devuelve un mensaje con código 200 como el siguiente:

```
HTTP/1.1 200 OK
```

Es que se ha procesado correctamente la petición y nos devuelve su respuesta. Si ha ocurrido un error, nos mandará el código y mensaje de error correspondiente. Por ejemplo, el error 400 indica que el servidor no ha entendido la petición que hemos hecho, posiblemente porque la hemos escrito incorrectamente. El error 500 nos dice que se trata de un error interno del servidor, no de la petición realizada.

Podemos obtener tanto el código como el mensaje de estado con los siguientes métodos:

```
int cod = con.getResponseCode();
String msg = con.getResponseMessage();
```

Los códigos de estado podemos encontrarlos como constantes de la clase `HttpConnection` como hemos visto para los tres códigos anteriores.

También podemos utilizar este objeto para leer las cabeceras que nos ha devuelto la respuesta. Nos ofrece métodos para leer una serie de cabeceras estándar de HTTP como los siguientes:

<code>getLength</code>	<code>content-length</code>	Longitud del contenido, o -1 si la longitud es desconocida
<code>getType</code>	<code>content-type</code>	Tipo MIME del contenido devuelto
<code>getEncoding</code>	<code>content-encoding</code>	Codificación del contenido
<code>getExpiration</code>	<code>expires</code>	Fecha de expiración del recurso
<code>getDate</code>	<code>date</code>	Fecha de envío del recurso
<code>getLastModified</code>	<code>last-modified</code>	Fecha de última modificación del recurso

Puede ser que queramos obtener otras cabeceras, como por ejemplo cabeceras propias no estándar. Para ello tendremos una serie de métodos que obtendrán las cabeceras directamente por su nombre:

```
String valor = con.getHeaderField(nombre);
int valor = con.getHeaderFieldInt(nombre);
long valor = con.getHeaderFieldDate(nombre);
```

De esta forma podemos obtener el valor de la cabecera o bien como una cadena, o en los datos que sean de tipo fecha (valor `long`) o enteros también podremos obtener su valor directamente en estos tipos de datos.

Podremos acceder a las cabeceras también a partir de su índice:

```
String valor = con.getHeaderField(int indice);
String nombre = con.getHeaderFieldKey(int indice);
```

Podemos obtener de esta forma tanto el nombre como el valor de la cabecera que ocupa un determinado índice.

Esta respuesta HTTP, además de un estado y una serie de cabeceras, tendrá un bloque que contenido que podremos leer abriendo un flujo de entrada en la conexión como hemos visto anteriormente. Normalmente cuando hacemos una petición a una URL de una aplicación web nos devuelve como contenido un documento HTML. Sin embargo, en el caso de nuestra aplicación MIDP este tipo de contenido no es apropiado. En su lugar podremos utilizar como contenido de la respuesta cualquier otro tipo MIME, que vendrá indicado en la cabecera `content-type` de la respuesta. Por ejemplo, podremos devolver una respuesta codificada de forma binaria que sea leída y decodificada por nuestra aplicación MIDP.

Tanto los métodos que obtienen un flujo para leer o escribir en la conexión, como estos métodos que acabamos de ver para obtener información sobre la respuesta producirán una transición al estado conectado.

### 5.3.3. Acceso a la red a bajo nivel

Como hemos comentado, el único tipo de conexión especificada en MIDP 1.0 es HTTP, la cual es suficiente y adecuada para acceder a aplicaciones corporativas. Sin embargo, con las redes 2.5G y 3G tendremos una mayor capacidad en las conexiones, nos permitirán realizar cualquier tipo de conexión TCP y UDP (no sólo HTTP) y además la comunicación podrá ser más fluida.

Para poder acceder a estas mejoras desde nuestra aplicación Java, surge la necesidad de que en MIDP 2.0 se incorpore soporte para tipos de conexiones a bajo nivel: sockets (TCP) y datagramas (UDP). Estos tipos de conexiones de MIDP 2.0 son optativos, de forma que aunque se encuentran definidos en la especificación de MIDP 2.0, no se obliga a que los fabricantes ni los operadores de telefonía lo soporten. Es decir, que la posibilidad de utilizar estas conexiones dependerá de que la red de telefonía de nuestro operador y el modelo de nuestro móvil las soporte.

Si intentamos utilizar un tipo de conexión no soportada por nuestro sistema, se producirá una excepción de tipo `ConnectionNotFoundException`.

### 5.3.3.1. Sockets

Los sockets nos permiten crear conexiones TCP. En este tipo de conexiones se establece un circuito virtual de forma permanente entre los dispositivos que se comunican. Se nos asegura que los datos enviados han llegado al servidor y que han llegado en el mismo orden en el que los enviamos. El inconveniente que tienen es que el tener un canal de comunicación abierto permanentemente consume una mayor cantidad de recursos.

Para abrir una conexión mediante sockets utilizaremos una URL como la siguiente:

```
SocketConnection sc =  
    (SocketConnection) Connector.open("socket://host:puerto");
```

Una vez abierta la conexión, podremos abrir sus correspondientes flujos de entrada y salida para enviar y recibir datos a través de ella:

```
InputStream in = sc.openInputStream();  
OutputStream out = sc.openOutputStream();
```

Es posible también hacer que nuestro dispositivos actúe como servidor. En este caso utilizaremos una URL como las siguientes para crear el socket servidor:

```
ServerSocketConnection ssc =  
    (ServerSocketConnection) Connector.open("socket://:puerto");  
ServerSocketConnection ssc =  
    (ServerSocketConnection) Connector.open("socket://");
```

En el primer caso indicamos el puerto en el que queremos que escuche nuestro servidor. En el segundo caso este puerto será asignado automáticamente por el sistema. Para conocer la dirección y el puerto donde escucha nuestro servidor podremos utilizar los siguientes métodos:

```
int puerto = ssc.getLocalPort();  
String host = ssc.getLocalAddress();
```

Para hacer que el servidor comience a escuchar y aceptar conexiones utilizaremos el siguiente método:

```
SocketConnection sc = (SocketConnection) ssc.acceptAndOpen();
```

Obtendremos un objeto `SocketConnection` con el que podremos comunicarnos con el

cliente que acaba de conectarse a nuestro servidor.

Debemos tener en cuenta que normalmente los móviles realizan conexiones puntuales cuando necesitan acceder a la red, y cada vez que se conecta se le asigna una nueva IP de forma dinámica. Esto hace difícil que un móvil pueda comportarse como servidor, ya que no podremos conocer a priori la dirección en la que está atendiendo para poder conectarnos a ella desde un cliente.

### 5.3.3.2. Datagramas

Cuando trabajemos con datagramas estaremos utilizando una conexión UDP. En ella no se establece un circuito virtual permanente, sino que cada paquete (datagrama) es enrutado de forma independiente. Esto produce que los paquetes puedan perderse o llegar desordenados al destino. Cuando la pérdida de paquetes o su ordenación no sea críticos, convendrá utilizar este tipo de conexiones, ya que consume menos recursos que los circuitos virtuales.

Para trabajar con datagramas utilizaremos una URL como la siguiente:

```
DatagramConnection dc =  
    (DatagramConnection) Connector.open("datagram://host:puerto");
```

En este caso no hemos abierto una conexión, ya que sólo se establecerá una conexión cuando se envíe un datagrama, simplemente hemos creado el objeto que nos permitirá intercambiar estos paquetes. Podemos crear un datagrama que contenga datos codificados en binario de la siguiente forma:

```
byte[] datos = obtenerDatos();  
Datagram dg = dc.newDatagram(datos, datos.length);
```

Una vez hemos creado el datagrama, podemos enviarlo al destinatario utilizando la conexión:

```
dc.send(dg);
```

En el caso del servidor, crearemos la conexión de datagramas de forma similar, pero sin especificar la dirección a la que conectar, ya que dependiendo del cliente deberemos enviar los datagramas a diferentes direcciones.

```
DatagramConnection dc =  
    (DatagramConnection) Connector.open("datagram://:puerto");
```

El servidor no conocerá las direcciones de sus clientes hasta que haya recibido algún datagrama de ellos. Para recibir un datagrama crearemos un datagrama vacío indicando su

capacidad (en bytes) y lo utilizaremos para recibir en él la información que se nos envía desde el cliente de la siguiente forma:

```
Datagram dg = dc.newDatagram(longitud);  
dc.receive(dg);
```

Una vez obtenido el datagrama, podremos obtener la dirección desde la cual se nos envía:

```
String direccion = dg.getAddress();
```

Ahora podremos crear un nuevo datagrama con la respuesta indicando la dirección a la que vamos a enviarlo. En este caso en cada datagrama se deberá especificar la dirección a la que se envía:

```
Datagram dg = dc.newDatagram(datos, datos.length, direccion);
```

El datagrama será enviado de la misma forma en la que se hacía en el cliente. Posteriormente el cliente podrá recibir este datagrama de la misma forma en que hemos visto que el servidor recibía su primer datagrama. De esta forma podremos establecer una conversación entre cliente y servidor, intercambiando estos datagramas.

#### 5.3.4. Envío y recepción de mensajes

---

Podemos utilizar la API adicional WMA para enviar o recibir mensajes cortos (SMS, Short Message Service) a través del teléfono móvil. Esta API extiende GFC, permitiendo establecer conexiones para recibir o enviar mensajes. Cuando queramos enviar mensajes nos comportaremos como clientes en la conexión, mientras que para recibirlos actuaremos como servidor. La URL para establecer una conexión con el sistema de mensajes para ser enviados o recibidos a través de una portadora SMS sobre GSM tendrá el siguiente formato:

```
sms://telefono:puerto
```

Las clases de esta API se encuentran en el paquete `javax.wireless.messaging`. Aquí se definen una serie de interfaces para trabajar con los mensajes y con la conexión.

##### 5.3.4.1. Envío de mensajes

---

Si queremos enviar mensajes, deberemos crear una conexión cliente proporcionando en la URL el número del teléfono al que vamos a enviar el mensaje y el puerto al que lo enviaremos de forma opcional:

```
sms://+34555000000
```

```
sms://+34555000000:4444
```

Si no especificamos el puerto se utilizará el puerto que se use por defecto para los mensajes del usuario en el teléfono móvil. Debemos abrir una conexión con una de estas URLs utilizando GFC, con lo que nos devolverá una conexión de tipo `MessageConnection`

```
MessageConnection mc =
    (MessageConnection)Connector.open("sms://+34555000000");
```

Una vez creada la conexión podremos utilizarla para enviar mensajes cortos. Podremos mandar tanto mensajes de texto como binarios. Estos mensajes tienen un tamaño limitado a un máximo de 140 bytes. Si el mensaje es de texto el número de caracteres dependerá de la codificación de éstos. Por ejemplo si los codificamos con 7 bits tendremos una longitud de 160 caracteres, mientras que con una codificación de 8 bits tendremos un juego de caracteres más amplio pero los mensajes estarán limitados a 140 caracteres.

WMA permite encadenar mensajes, de forma que esta longitud podrá ser por lo menos 3 veces mayor. El encadenamiento consiste en que si el mensaje supera la longitud máxima de 140 bytes que puede transportar SMS, entonces se fracciona en varios fragmentos que serán enviados independientemente a través de SMS y serán unidos al llegar a su destino para formar el mensaje completo. Esto tiene el inconveniente de que realmente por la red están circulando varios mensajes, por lo que se nos cobrará por el número de fragmentos que haya enviado.

Podremos crear el mensaje a enviar a partir de la conexión. Los mensajes de texto los crearemos de la siguiente forma:

```
String texto = "Este es un mensaje corto de texto";
TextMessage msg = mc.newMessage(mc.TEXT_MESSAGE);
msg.setPayloadText(texto);
```

Para el caso de un mensaje binario, lo crearemos de la siguiente forma:

```
byte [] datos = codificarDatos();
BinaryMessage msg = mc.newMessage(mc.BINARY_MESSAGE);
msg.setPayloadData(datos);
```

Antes de enviar el mensaje, podemos ver en cuántos fragmentos deberá ser dividido para poder ser enviado utilizando la red subyacente con el siguiente método:

```
int num_segmentos = mc.numberOfSegments(msg);
```

Esto nos devolverá el número de segmentos en los que se fraccionará el mensaje, ó 0 si el mensaje no puede ser enviado utilizando la red subyacente.

Independientemente de si se trata de un mensaje de texto o de un mensaje binario, podremos enviarlo utilizando el siguiente método:

```
mc.send(msg);
```

#### 5.3.4.2. Recepción de mensajes

Para recibir mensajes deberemos crear una conexión de tipo servidor. Para ello en la URL sólo especificaremos el puerto en el que queremos recibir los mensajes:

```
sms://:4444
```

Crearemos una conexión utilizando una URL como esta, en la que no se especifique el número de teléfono destino.

```
MessageConnection mc =  
    (MessageConnection)Connector.open("sms://:4444");
```

Para recibir un mensaje utilizaremos el método:

```
Message msg = mc.receive();
```

Si hemos recibido un mensaje que todavía no hay sido leído este método obtendrá dicho mensaje. Si todavía no se ha recibido ningún mensaje, este método se quedará bloqueado hasta que se reciba un mensaje, momento en el que lo leerá y nos lo devolverá.

Podemos determinar en tiempo de ejecución si se trata de un mensaje de texto o de un mensaje binario. Para ello deberemos comprobar de qué tipo es realmente el objeto devuelto, y según este tipo leer sus datos como texto o como array de bytes:

```
if(msg instanceof TextMessage) {  
    String texto = ((TextMessage)msg).getPayloadText();  
    // Procesar texto  
} else if(msg instanceof BinaryMessage) {  
    byte [] datos = ((BinaryMessage)msg).getPayloadData();  
    // Procesar datos  
}
```

Hemos visto que el método `receive` se queda bloqueado hasta que se reciba un mensaje. No debemos hacer que la aplicación se quede bloqueada esperando un mensaje, ya que éste puede tardar bastante, o incluso no llegar nunca. Podemos solucionar este problema realizando la lectura de los mensajes mediante un hilo en segundo plano. Otra solución es utilizar un listener.

### 5.3.4.3. Listener de mensajes

Estos *listeners* nos servirán para que se nos notifique el momento en el que se recibe un mensaje corto. De esta forma no tendremos que quedarnos bloqueados esperando recibir el mensaje, sino que podemos invocar `receive` directamente cuando sepamos que se ha recibido el mensaje.

Para crear un *listener* de este tipo deberemos crear una clase que implemente la interfaz `MessageListener`:

```
public MiListener implements MessageListener {
    public void notifyIncomingMessage(MessageConnection mc) {
        // Se ha recibido un mensaje a través de la conexión mc
    }
}
```

Dentro del método `notifyIncomingMessage` deberemos introducir el código a ejecutar cuando se reciba un mensaje. No debemos ejecutar la operación `receive` directamente dentro de este método, ya que es una operación costosa que no debe ser ejecutada dentro de los *callbacks* que deben devolver el control lo antes posible para no entorpecer el procesamiento de eventos de la aplicación. Deberemos hacer que la recepción del mensaje la realice un hilo independiente.

Para que la recepción de mensajes le sea notificada a nuestro listener deberemos registrarlo como listener de la conexión con:

```
mc.setMessageListener(new MiListener());
```

En WTK 2.0 tenemos disponible una consola WMA con la que podremos simular el envío y la recepción de mensajes cortos que se intercambien entre los emuladores, de forma que podremos probar estas aplicaciones sin tener que enviar realmente los mensajes y pagar por ellos.



## 6. Excepciones e hilos. Acceso a la red - Ejercicios

### 6.1. Captura de excepciones (\*)

En el proyecto `java-excepciones` de las plantillas de la sesión tenemos una aplicación de Java en `Ej1.java` que toma un número como parámetro, y como salida muestra el logaritmo de dicho número. Sin embargo, en ningún momento comprueba si se ha proporcionado algún parámetro, ni si ese parámetro es un número. Se pide:

a) Compilar el programa y ejecutarlo de tres formas distintas:

- Sin parámetros

```
java Ej1
```

- Poniendo un parámetro no numérico

```
java Ej1 pepe
```

- Poniendo un parámetro numérico

```
java Ej1 30
```

Anotad las excepciones que se lanzan en cada caso (si se lanzan)

b) Modificar el código de `main` para que capture las excepciones producidas y muestre los errores correspondientes en cada caso:

- Para comprobar si no hay parámetros se capturará una excepción de tipo `ArrayIndexOutOfBoundsException` (para ver si el *array* de `String` que se pasa en el `main` tiene algún elemento).
- Para comprobar si el parámetro es numérico, se capturará una excepción de tipo `NumberFormatException`.

Así, tendremos en el `main` algo como:

```
try
{
    // Tomar parámetro y asignarlo a un double
} catch (ArrayIndexOutOfBoundsException e1) {
    // Código a realizar si no hay parametros
} catch (NumberFormatException e2) {
    // Código a realizar con parametro no numerico
}
```

Probad de nuevo el programa igual que en el caso anterior comprobando que las excepciones son capturadas y tratadas.

### 6.2. Lanzamiento de excepciones

El fichero Ej2.java es similar al anterior, aunque ahora no vamos a tratar las excepciones del `main`, sino las del método `logaritmo`: en la función que calcula el logaritmo se comprueba si el valor introducido es menor o igual que 0, ya que para estos valores la función `logaritmo` no está definida. Se pide:

a) Buscar entre las excepciones de Java la más adecuada para lanzar en este caso, que indique que a un método se le ha pasado un argumento ilegal. (Pista: Buscar entre las clases derivadas de `Exception`. En este caso la más adecuada se encuentra entre las derivadas de `RuntimeException`).

b) Una vez elegida la excepción adecuada, añadir código (en el método `logaritmo`) para que en el caso de haber introducido un parámetro incorrecto se lance dicha excepción.

```
throw new ... // excepcion elegida
```

Probar el programa para comprobar el efecto que tiene el lanzamiento de la excepción.

c) Al no ser una excepción del tipo *checked* no hará falta que la capturemos ni que declaremos que puede ser lanzada. Vamos a crear nuestro propio tipo de excepción derivada de `Exception` (de tipo *checked*) para ser lanzada en caso de introducir un valor no válido como parámetro. La excepción se llamará `WrongParameterException` y tendrá la siguiente forma:

```
public class WrongParameterException extends Exception
{
    public WrongParameterException(String msg) {
        super(msg);
    }
}
```

Deberemos lanzarla en lugar de la escogida en el punto anterior.

```
throw new WrongParameterException(...);
```

Intentar compilar el programa y observar los errores que aparecen. ¿Por qué ocurre esto? Añadir los elementos necesarios al código para que compile y probarlo.

d) Por el momento controlamos que no se pase un número negativo como entrada. ¿Pero qué ocurre si la entrada no es un número válido? En ese caso se producirá una excepción al convertir el valor de entrada y esa excepción se propagará automáticamente al nivel superior. Ya que tenemos una excepción que indica cuando el parámetro de entrada de nuestra función es incorrecto, sería conveniente que siempre que esto ocurra se lance dicha excepción, independientemente de si ha sido causada por un número negativo o por algo que no es un número, pero siempre conservando la información sobre la causa que produjo el error. Utilizar *nested exceptions* para realizar esto.

#### Ayuda

Deberemos añadir un nuevo constructor a `WrongParameterException` en el que se proporcione la excepción que causó el error. En la función `logaritmo` capturaremos cualquier excepción que se produzca al convertir la cadena a número, y lanzaremos una excepción `WrongParameterException` que incluya la excepción causante.

### 6.3. Chat para el móvil

Vamos a ver como ejemplo una aplicación de chat para el móvil. En el directorio ejemplos de las plantillas de la sesión se encuentra una aplicación web con todos los servlets que necesitaremos para probar los ejemplos. Podremos desplegar esta aplicación en Tomcat para hacer pruebas con nuestro propio servidor.

Podemos encontrar la aplicación de chat implementada en el directorio Chat, que realiza las siguientes tareas:

- Lo primero que se mostrará será una pantalla de *login*, donde el usuario deberá introducir el *login* con el que participar en el chat. Debemos enviar este *login* al servidor para iniciar la sesión. Para ello abriremos una conexión con la URL del *servlet* proporcionando los siguientes parámetros:

```
?accion=login&id=<nick_del_usuario>
```

Si el *login* es correcto, el servidor nos devolverá un código de respuesta 200 OK. Además deberemos leer la cabecera URL-Rescrita, donde nos habrá enviado la URL rescrita que deberemos utilizar de ahora en adelante para mantener la sesión.

- Una vez hemos entrado en el chat, utilizaremos la técnica de *polling* para obtener los mensajes escritos en el chat y mostrarlos en la pantalla. Utilizando la URL rescrita, conectaremos al *servlet* del chat proporcionando el siguiente parámetro:

```
?accion=lista
```

Esto nos devolverá como respuesta una serie de mensajes, codificados mediante un objeto `DataOutputStream` de la siguiente forma:

```
<nick1> <mensaje1>
<nick2> <mensaje2>
...
<nickN> <mensajeN>
```

De esta forma podremos utilizar un objeto `DataInputStream` para ir leyendo con el método `readUTF` las cadenas del *nick* y del texto de cada mensaje del chat:

```
String nick = dis.readUTF();
String texto = dis.readUTF();
```

- Para enviar mensajes al chat utilizaremos el bloque de contenido, conectándonos a la URL rescrita proporcionando el siguiente parámetro:

```
?accion=enviar
```

El mensaje se deberá codificar en binario, escribiendo la cadena del mensaje con el método `writeUTF` de un objeto `DataOutputStream`. Si obtenemos una respuesta 200 OK el mensaje habrá sido enviado correctamente.

## 7. Flujos de E/S y serialización de objetos. RMS

Una de las posibilidades que ofrecen los flujos de entrada y salida es la de transportar objetos Java codificados en binario (serializados). En esta sesión se verá cómo hacer esto en JavaME, ya que no soporta la serialización automática. Otro mecanismo de persistencia muy usado en JavaME son los registros RMS.

### 7.1. Flujos de datos de entrada/salida

Existen varios objetos que hacen de flujos de datos, y que se distinguen por la finalidad del flujo de datos y por el tipo de datos que viajen a través de ellos. Según el tipo de datos que transporten podemos distinguir:

- Flujos de caracteres
- Flujos de *bytes*

Dentro de cada uno de estos grupos tenemos varios pares de objetos, de los cuales uno nos servirá para leer del flujo y el otro para escribir en él. Cada par de objetos será utilizado para comunicarse con distintos elementos (memoria, ficheros, red u otros programas). Estas clases, según sean de entrada o salida y según sean de caracteres o de *bytes* llevarán distintos sufijos, según se muestra en la siguiente tabla:

	Flujo de entrada / lector	Flujo de salida / escritor
<b>Caractéres</b>	<code>_Reader</code>	<code>_Writer</code>
<b>Bytes</b>	<code>_InputStream</code>	<code>_OutputStream</code>

Donde el prefijo se referirá a la fuente o sumidero de los datos que puede tomar valores como los que se muestran a continuación:

<code>File_</code>	Acceso a ficheros
<code>Piped_</code>	Comunicación entre programas mediante tuberías ( <i>pipes</i> )
<code>String_</code>	Acceso a una cadena en memoria (solo caracteres)
<code>CharArray_</code>	Acceso a un <i>array</i> de caracteres en memoria (solo caracteres)
<code>ByteArray_</code>	Acceso a un <i>array</i> de <i>bytes</i> en memoria (solo <i>bytes</i> )

Además podemos distinguir los flujos de datos según su propósito, pudiendo ser:

- Canales de datos, simplemente para leer o escribir datos directamente en una fuente o sumidero externo.
- Flujos de procesamiento, que además de enviar o recibir datos realizan algún procesamiento con ellos. Tenemos por ejemplo flujos que realizan un filtrado de los datos que viajan a través de ellos (con prefijo `Filter`), conversores datos (con prefijo

Data), *bufferes* de datos (con prefijo `Buffered`), preparados para la impresión de elementos (con prefijo `Print`), etc.

Un tipo de filtros de procesamiento a destacar son aquellos que nos permiten convertir un flujo de *bytes* a flujo de caracteres. Estos objetos son `InputStreamReader` y `OutputStreamWriter`. Como podemos ver en su sufijo, son flujos de caracteres, pero se construyen a partir de flujos de *bytes*, permitiendo de esta manera acceder a nuestro flujo de *bytes* como si fuese un flujo de caracteres.

Para cada uno de los tipos básicos de flujo que hemos visto existe una superclase, de la que heredaran todos sus subtipos, y que contienen una serie de métodos que serán comunes a todos ellos. Entre estos métodos encontramos los métodos básicos para leer o escribir caracteres o *bytes* en el flujo a bajo nivel. En la siguiente tabla se muestran los métodos más importantes de cada objeto:

<b>InputStream</b>	<code>read()</code> , <code>reset()</code> , <code>available()</code> , <code>close()</code>
<b>OutputStream</b>	<code>write(int b)</code> , <code>flush()</code> , <code>close()</code>
<b>Reader</b>	<code>read()</code> , <code>reset()</code> , <code>close()</code>
<b>Writer</b>	<code>write(int c)</code> , <code>flush()</code> , <code>close()</code>

A parte de estos métodos podemos encontrar variantes de los métodos de lectura y escritura, otros métodos, y además cada tipo específico de flujo contendrá sus propios métodos. Todas estas clases se encuentran en el paquete `java.io`. Para más detalles sobre ellas se puede consultar la especificación de la API de Java.

## 7.2. Entrada, salida y salida de error estándar

Al igual que en C, en Java también existen los conceptos de entrada, salida, y salida de error estándar. La entrada estándar normalmente se refiere a lo que el usuario escribe en la consola, aunque el sistema operativo puede hacer que se tome de otra fuente. De la misma forma la salida y la salida de error estándar lo que hacen normalmente es mostrar los mensajes y los errores del programa respectivamente en la consola, aunque el sistema operativo también podrá redirigirlas a otro destino.

En Java esta entrada, salida y salida de error estándar se tratan de la misma forma que cualquier otro flujo de datos, estando estos tres elementos encapsulados en tres objetos de flujo de datos que se encuentran como propiedades estáticas de la clase `System`:

	<b>Tipo</b>	<b>Objeto</b>
<b>Entrada estándar</b>	<code>InputStream</code>	<code>System.in</code>
<b>Salida estándar</b>	<code>PrintStream</code>	<code>System.out</code>
<b>Salida de error estándar</b>	<code>PrintStream</code>	<code>System.err</code>

Para la entrada estándar vemos que se utiliza un objeto `InputStream` básico, sin embargo para la salida se utilizan objetos `PrintWriter` que facilitan la impresión de texto ofreciendo a parte del método común de bajo nivel `write` para escribir *bytes*, dos métodos más: `print` y `println`. Estas funciones nos permitirán escribir cualquier cadena, tipo básico, o bien cualquier objeto que defina el método `toString` que devuelva una representación del objeto en forma de cadena. La única diferencia entre los dos métodos es que el segundo añade automáticamente un salto de línea al final del texto impreso, mientras que en el primero deberemos especificar explícitamente este salto.

Para escribir texto en la consola normalmente utilizaremos:

```
System.out.println("Hola mundo");
```

En el caso de la impresión de errores por la salida de error de estándar, deberemos utilizar:

```
System.err.println("Error: Se ha producido un error");
```

Además la clase `System` nos permite sustituir estos flujos por defecto por otros flujos, cambiando de esta forma la entrada, salida y salida de error estándar.

#### Truco

Podemos ahorrar tiempo si en Eclipse en lugar de escribir `System.out.println` escribimos simplemente `sysout` y tras esto pulsamos *Ctrl + Espacio*.

## 7.3. Acceso a ficheros

Podremos acceder a ficheros bien por caracteres, o bien de forma binaria (por *bytes*). Las clases que utilizaremos en cada caso son:

	Lectura	Escritura
Caracteres	<code>FileReader</code>	<code>FileWriter</code>
Binarios	<code>FileInputStream</code>	<code>FileOutputStream</code>

Para crear un lector o escritor de ficheros deberemos proporcionar al constructor el fichero del que queremos leer o en el que queramos escribir. Podremos proporcionar esta información bien como una cadena de texto con el nombre del fichero, o bien construyendo un objeto `File` representando al fichero al que queremos acceder. Este objeto nos permitirá obtener información adicional sobre el fichero, a parte de permitirnos realizar operaciones sobre el sistema de ficheros.

A continuación vemos un ejemplo simple de la copia de un fichero carácter a carácter:

```
public void copia_fichero() {
    int c;
    try {
        FileReader in = new FileReader("fuente.txt");
```

```

FileWriter out = new FileWriter("destino.txt");

while( (c = in.read()) != -1) {
    out.write(c);
}

in.close();
out.close();

} catch(FileNotFoundException e1) {
    System.err.println("Error: No se encuentra el fichero");
} catch(IOException e2) {
    System.err.println("Error leyendo/escribiendo fichero");
}
}

```

En el ejemplo podemos ver que para el acceso a un fichero es necesario capturar dos excepciones, para el caso de que no exista el fichero al que queramos acceder y por si se produce un error en la E/S.

Para la escritura podemos utilizar el método anterior, aunque muchas veces nos resultará mucho más cómodo utilizar un objeto `PrintWriter` con el que podamos escribir directamente líneas de texto:

```

public void escribe_fichero() {
    FileWriter out = null;
    PrintWriter p_out = null;

    try {
        out = new FileWriter("result.txt");
        p_out = new PrintWriter(out);
        p_out.println(
            "Este texto será escrito en el fichero de salida");
    } catch(IOException e) {
        System.err.println("Error al escribir en el fichero");
    } finally {
        p_out.close();
    }
}

```

## 7.4. Acceso a los recursos

Hemos visto como leer y escribir ficheros, pero cuando ejecutamos una aplicación contenida en un fichero JAR, puede que necesitemos leer recursos contenidos dentro de este JAR.

Para acceder a estos recursos deberemos abrir un flujo de entrada que se encargue de leer su contenido. Para ello utilizaremos el método `getResourceAsStream` de la clase `Class`:

```

InputStream in = getClass().getResourceAsStream("/datos.txt");

```

De esta forma podremos utilizar el flujo de entrada obtenido para leer el contenido del fichero que hayamos indicado. Este fichero deberá estar contenido en el JAR de la aplicación.



Especificamos el carácter '/' delante del nombre del recurso para referenciarlo de forma relativa al directorio raíz del JAR. Si no lo especificásemos de esta forma se buscaría de forma relativa al directorio correspondiente al paquete de la clase actual.

## 7.5. Codificación de datos

Si queremos guardar datos en un fichero binario, enviarlos a través de la red, o en general transferirlos mediante cualquier flujo de E/S, deberemos codificar estos datos en forma de *array* de *bytes*. Los flujos de procesamiento `DataInputStream` y `DataOutputStream` nos permitirán codificar y decodificar respectivamente los tipos de datos simples en forma de *array* de *bytes* para ser enviados a través de un flujo de datos.

Por ejemplo, podemos codificar datos en un *array* en memoria (`ByteArrayOutputStream`) de la siguiente forma:

```
String nombre = "Jose";
String edad = 25;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);

dos.writeUTF(nombre);
dos.writeInt(edad);
dos.close();
baos.close();

byte [] datos = baos.toByteArray();
```

Podremos decodificar este *array* de *bytes* realizando el procedimiento inverso, con un flujo que lea un *array* de *bytes* de memoria (`ByteArrayInputStream`):

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = new DataInputStream(bais);
String nombre = dis.readUTF();
int edad = dis.readInt();
```

Si en lugar de almacenar estos datos codificados en una *array* en memoria queremos guardarlos codificados en un fichero, haremos lo mismo simplemente sustituyendo el flujo canal de datos `ByteArrayOutputStream` por un `FileOutputStream`. De esta forma podremos utilizar cualquier canal de datos para enviar estos datos codificados a través de él.

## 7.6. Serialización de objetos en Java

Si queremos enviar un objeto a través de un flujo de datos, deberemos convertirlo en una serie de *bytes*. Esto es lo que se conoce como serialización de objetos, que nos permitirá leer y escribir objetos directamente.

Para leer o escribir objetos podemos utilizar los objetos `ObjectInputStream` y `ObjectOutputStream` que incorporan los métodos `readObject` y `writeObject` respectivamente. Los objetos que escribamos en dicho flujo deben tener la capacidad de

ser *serializables*.

Serán *serializables* aquellos objetos que implementan la interfaz `Serializable`. Cuando queramos hacer que una clase definida por nosotros sea *serializable* deberemos implementar dicho interfaz, que no define ninguna función, sólo se utiliza para identificar las clases que son *serializables*. Para que nuestra clase pueda ser *serializable*, todas sus propiedades deberán ser de tipos de datos básicos o bien objetos que también sean *serializables*.

Un uso común de la serialización se realiza en los *Transfer Objects*. Este tipo de objetos deben ser serializables para así poderse intercambiar entre todas las capas de la aplicación, aunque se encuentren en máquinas diferentes.

Por ejemplo, si tenemos un objeto como el siguiente:

```
public class Punto2D implements Serializable {
    private int x;
    private int y;

    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
}
```

Podríamos enviarlo a través de un flujo, independientemente de su destino, de la siguiente forma:

```
Punto2D p = crearPunto();
FileOutputStream fos = new FileOutputStream(FICHERO_DATOS);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(p);
oos.close();
```

En este caso hemos utilizado como canal de datos un flujo con destino a un fichero, pero se podría haber utilizado cualquier otro tipo de canal (por ejemplo para enviar un objeto Java desde un servidor web hasta una máquina cliente). En aplicaciones distribuidas los objetos *serializables* nos permitirán mover estructuras de datos entre diferentes máquinas sin que el desarrollador tenga que preocuparse de la codificación y transmisión de los datos.

Muchas clases de la API de Java son *serializables*, como por ejemplo las colecciones. Si tenemos una serie de elementos en una lista, podríamos serializar la lista completa, y de esa forma guardar todos nuestros objetos, con una única llamada a `writeObject`.

Cuando una clase implemente la interfaz `Serializable` veremos que Eclipse nos da un *warning* si no añadimos un campo `serialVersionUID`. Este es un código numérico que

se utiliza para asegurarnos de que al recuperar un objeto serializado éste se asocie a la misma clase con la que se creó. Así evitamos el problema que puede surgir al tener dos clases que puedan tener el mismo nombre, pero que no sean iguales (podría darse el caso que una de ellas esté en una máquina cliente, y la otra en el servidor). Si no tuviésemos ningún código para identificarlas, se podría intentar recuperar un objeto en una clase incorrecta.

Eclipse nos ofrece dos formas de generar este código pulsando sobre el icono del *warning*: con un valor por defecto, o con un valor generado automáticamente. Será recomendable utilizar esta segunda forma, que nos asegura que dos clases distintas tendrán códigos distintos.

## 7.7. Serialización manual en JavaME

CLDC no soporta la serialización de objetos, por tanto tendremos que serializar los objetos manualmente, definiendo los métodos `serialize()` y `deserialize()`.

En el ejemplo de un objeto que representa un punto en el plano, serializaríamos el objeto y lo deserializaríamos con:

```
public class Punto2D {
    int x;
    int y;
    String etiqueta;

    public void serialize(OutputStream out) throws IOException {
        DataOutputStream dos = new DataOutputStream( out );
        dos.writeInt(x);
        dos.writeInt(y);
        dos.writeUTF(etiqueta);
        dos.flush();
    }

    public static Punto2D deserialize(InputStream in)
        throws IOException {
        DataInputStream dis = new DataInputStream( in );
        Punto2D p = new Punto2D();
        p.x = dis.readInt();
        p.y = dis.readInt();
        p.etiqueta = dis.readUTF();
        return p;
    }
}
```

## 7.8. Almacenamiento de registros RMS en JavaME

Muchas veces las aplicaciones necesitan almacenar datos de forma persistente. Cuando realizamos aplicaciones para PCs de sobremesa o servidores podemos almacenar esta información en algún fichero en el disco o bien en una base de datos. Lo más sencillo será almacenarla en ficheros, pero en los dispositivos móviles no podemos contar ni tan solo con esta característica. Aunque los móviles normalmente tienen su propio sistema de

ficheros, por cuestiones de seguridad MIDP no nos dejará acceder directamente a él. Es posible que en alguna implementación podamos acceder a ficheros en el dispositivo, pero esto no es requerido por la especificación, por lo que si queremos que nuestra aplicación sea portable no deberemos confiar en esta característica.

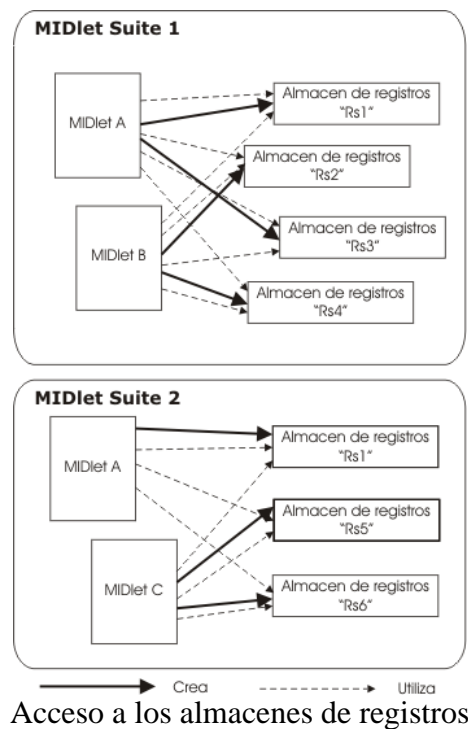
Para almacenar datos de forma persistente en el móvil utilizaremos RMS (Record Management System). Se trata de un sistema de almacenamiento que nos permitirá almacenar registros con información de forma persistente en los dispositivos móviles. No se especifica ninguna forma determinada en la que se deba almacenar esta información, cada implementación deberá guardar estos datos de la mejor forma posible para cada dispositivo concreto, utilizando memoria no volátil, de forma que no se pierda la información aunque reiniciemos el dispositivo o cambiemos las baterías. Por ejemplo, algunas implementaciones podrán utilizar el sistema de ficheros del dispositivo para almacenar la información de RMS, o bien cualquier otro dispositivo de memoria no volátil que contenga el móvil. La forma de almacenamiento real de la información en el dispositivo será transparente para los MIDlets, éstos sólo podrán acceder a la información utilizando la API de RMS. Esta API se encuentra en el paquete `javax.microedition.rms`.

### 7.8.1. Almacenes de registros

---

La información se almacena en almacenes de registros (Record Stores), que serán identificados con un nombre que deberemos asignar nosotros. Cada aplicación podrá crear y utilizar tantos almacenes de registros como quiera. Cada almacén de registros contendrá una serie de registros con la información que queramos almacenar en ellos.

Los almacenes de registros son propios de la suite. Es decir, los almacenes de registro creados por un MIDlet dentro de una suite, serán compartidos por todos los MIDlets de esa suite, pero no podrán acceder a ellos los MIDlets de suites distintas. Por seguridad, no se permite acceder a recursos ni a almacenes de registros de suites distintas a la nuestra.



Cada suite define su propio espacio de nombres. Es decir, los nombres de los almacenes de registros deben ser únicos para cada suite, pero pueden estar repetidos en diferentes suites. Como hemos dicho antes, nunca podremos acceder a un almacén de registros perteneciente a otra suite.

#### 7.8.1.1. Abrir el almacén de registros

Lo primero que deberemos hacer es abrir o crear el almacén de registros. Para ello utilizaremos el siguiente método:

```
RecordStore rs = RecordStore.open(nombre, true);
```

Con el segundo parámetro a `true` estamos diciendo que si el almacén de registros con nombre `nombre` no existiese en nuestra suite lo crearía. Si por el contrario estuviese a `false`, sólo intentaría abrir un almacén de registros existente, y si éste no existe se producirá una excepción `RecordStoreNotFoundException`.

El nombre que especificamos para el almacén de registros deberá ser un nombre de como mucho 32 caracteres codificado en Unicode.

Una vez hayamos terminado de trabajar con el almacén de registros, podremos cerrarlo con:

```
rs.close();
```

### 7.8.1.2. Listar los almacenes de registros

Si queremos ver la lista completa de almacenes de registros creados dentro de nuestra suite, podemos utilizar el siguiente método:

```
String [] nombres = RecordStore.listRecordStores();
```

Esto nos devolverá una lista con los nombres de los almacenes de registros que hayan sido creados. Teniendo estos nombres podremos abrirlos como hemos visto anteriormente para consultarlos, o bien eliminarlos.

### 7.8.1.3. Eliminar un almacén de registros

Podemos eliminar un almacén de registros existente proporcionando su nombre, con:

```
RecordStore.deleteRecordStore(nombre);
```

### 7.8.1.4. Propiedades de los almacenes de registros

Los almacenes de registros tienen una serie de propiedades que podemos obtener con información sobre ellos. Una vez hayamos abierto el almacén de registros para trabajar con él, podremos obtener los valores de las siguientes propiedades:

- Nombre: El nombre con el que hemos identificado el almacén de registros.

```
String nombre = rs.getName();
```

- Estampa de tiempo: El almacén de registros contiene una estampa de tiempo, que nos indicará el momento de la última modificación que se ha realizado en los datos que almacena. Este instante de tiempo se mide en milisegundos desde el 1 de enero de 1970 a las 0:00, y podemos obtenerlo con:

```
long timestamp = rs.getLastModified();
```

- Versión: También tenemos una versión del almacén de registros. La versión será un número que se incrementará cuando se produzca cualquier modificación en el almacén de registros. Esta propiedad, junto a la anterior, nos será útil para tareas de sincronización de datos.

```
int version = rs.getVersion();
```

- Tamaño: Nos dice el espacio en bytes que ocupa el almacén de registros actualmente.

```
int tam = rs.getSize();
```

- **Tamaño disponible:** Nos dice el espacio máximo que podrá crecer este almacén de registros. El dispositivo limitará el espacio asignado a cada almacén de registros, y con este método podremos saber el espacio restante que nos queda.

```
int libre = rs.getSizeAvailable();
```

## 7.8.2. Registros

El almacén de registros contendrá una serie de registros donde podemos almacenar la información. Podemos ver el almacén de registros como una tabla en la que cada fila corresponde a un registro. Los registros tienen un identificador y un array de datos.

Identificador	Datos
1	array de datos ...
2	array de datos ...
3	array de datos ...
...	...

Estos datos de cada registro se almacenan como un array de bytes. Podremos acceder a estos registros mediante su identificador o bien recorriendo todos los registros de la tabla.

Cuando añadamos un nuevo registro al almacén se le asignará un identificador una unidad superior al identificador del último registro que tengamos. Es decir, si añadimos dos registros y al primero se le asigna un identificador  $n$ , el segundo tendrá un identificador  $n+1$ .

Las operaciones para acceder a los datos de los registros son atómicas, por lo que no tendremos problemas cuando se acceda concurrentemente al almacén de registros.

### 7.8.2.1. Almacenar información

Tenemos dos formas de almacenar información en el almacén de registros. Lo primero que deberemos hacer en ambos casos es construir un array de bytes con la información que queramos añadir. Para hacer esto podemos utilizar un flujo `DataOutputStream`, como se muestra en el siguiente ejemplo:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);

dos.writeUTF(nombre);
dos.writeInt(edad);
```

```
byte [] datos = baos.toByteArray();
```

Una vez tenemos el array de datos que queremos almacenar, podremos utilizar uno de los siguientes métodos del objeto almacén de datos:

```
int id = rs.addRecord(datos, 0, datos.length);  
rs.setRecord(id, datos, 0, datos.length);
```

En el caso de `addRecord`, lo que se hace es añadir un nuevo registro al almacén con la información que hemos proporcionado, devolviéndonos el identificador `id` asignado al registro que acabamos de añadir.

Con `setRecord` lo que se hace es sobrescribir el registro correspondiente al identificador `id` indicado con los datos proporcionados. En este caso no se añade ningún registro nuevo, sólo se almacenan los datos en un registro ya existente.

#### 7.8.2.2. Leer información

---

Si tenemos el identificador del registro que queremos leer, podemos obtener su contenido como array de bytes directamente utilizando el método:

```
byte [] datos = rs.getRecord(id);
```

Si hemos codificado la información dentro de este registro utilizando un flujo `DataOutputStream`, podemos decodificarlo realizando el proceso inverso con un flujo `DataInputStream`:

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);  
DataInputStream dis = DataInputStream(bais);  
  
String nombre = dis.readUTF();  
String edad = dis.readInt();  
  
dis.close();
```

#### 7.8.2.3. Borrar Registros

---

Podremos borrar un registro del almacén a partir de su identificador con el siguiente método:

```
rs.deleteRecord(id);
```

#### 7.8.2.4. Almacenar y recuperar objetos

---



Si hemos definido una forma de serializar los objetos, podemos aprovechar esta serialización para almacenar los objetos de forma persistente en RMS y posteriormente poder recuperarlos.

Imaginemos que en nuestra clase `MisDatos` hemos definido los siguientes métodos para serializar y deserializar tal como vimos en el apartado de entrada/salida:

```
public void serialize(OutputStream out)
public static MisDatos deserialize(InputStream in)
```

Podemos serializar el objeto en un array de bytes utilizando estos métodos para almacenarlo en RMS de la siguiente forma:

```
MisDatos md = new MisDatos();
...
ByteArrayOutputStream baos = new ByteArrayOutputStream();
md.serialize(baos);

byte [] datos = baos.toByteArray();
```

Una vez tenemos este array de bytes podremos almacenarlo en RMS. Cuando queramos recuperar el objeto original, leeremos el array de bytes de RMS y deserializaremos el objeto de la siguiente forma:

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
MisDatos md = MisDatos.deserialize(bais);
```

### 7.8.3. Navegar en el almacén de registros

Si no conocemos el identificador del registro al que queremos acceder, podremos recorrer todos los registros del almacén utilizando un objeto `RecordEnumeration`. Para obtener la enumeración de registros del almacén podemos utilizar el siguiente método:

```
RecordEnumeration re = rs.enumerateRecords(null, null, false);
```

Con los dos primeros parámetros podremos establecer la ordenación y el filtrado de los registros que se enumeren como veremos más adelante. Por ahora vamos a dejarlo a `null` para obtener la enumeración con todos los registros y en un orden arbitrario. Esta es la forma más eficiente de acceder a los registros.

El tercer parámetro nos dice si la enumeración debe mantenerse actualizada con los registros que hay realmente almacenados, o si por el contrario los cambios que se realicen en el almacén después de haber obtenido la enumeración no afectarán a dicha enumeración. Será más eficiente establecer el valor a `false` para evitar que se tenga que mantener actualizado, pero esto tendrá el inconveniente de que puede que alguno de los

registros de la enumeración se haya borrado o que se hayan añadido nuevos registros que no constan en la enumeración. En el caso de que especifiquemos `false` para que no actualice automáticamente la enumeración, podremos forzar manualmente a que se actualice invocando el método `rebuild` de la misma, que la reconstruirá utilizando los nuevos datos.

Recorreremos la enumeración de registros de forma similar a como recorremos los objetos `Enumeration`. Tendremos un cursor que en cada momento estará en uno de los elementos de la enumeración. En este caso podremos recorrer la enumeración de forma bidireccional.

Para pasar al siguiente registro de la enumeración y obtener sus datos utilizaremos el método `nextRecord`. Podremos saber si existe un siguiente registro llamando a `hasNextElement`. Nada más crear la enumeración el cursor no se encontrará en ninguno de los registros. Cuando llamemos a `nextRecord` por primera vez se situará en el primer registro y nos devolverá su array de datos. De esta forma podremos seguir recorriendo la enumeración mientras haya más registros. Un bucle típico para hacer este recorrido es el siguiente:

```
while(re.hasNextElement()) {
    byte [] datos = re.nextRecord();
    // Procesar datos obtenidos
    ...
}
```

Hemos dicho que el recorrido puede ser bidireccional. Por lo tanto, tenemos un método `previousRecord` que moverá el cursor al registro anterior devolviéndonos su contenido. De la misma forma, tenemos un método `hasPreviousElement` que nos dirá si existe un registro anterior. Si invocamos `previousRecord` nada más crear la enumeración, cuando el cursor todavía no se ha posicionado en ningún registro, moverá el cursor al último registro de la enumeración devolviéndonos su resultado. Podemos también volver al estado inicial de la enumeración en el que el cursor no apunta a ningún registro llamando a su método `reset`.

En lugar de obtener el contenido de los registros puede que nos interese obtener su identificador, de forma que podamos eliminarlos o hacer otras operaciones con ellos. Para ello tenemos los métodos `nextRecordId` y `previousRecordId`, que tendrán el mismo comportamiento que `nextRecord` y `previousRecord` respectivamente, salvo porque devuelven el identificador de los registros recorridos, y no su contenido.

### 7.8.3.1. Ordenación de registros

Puede que nos interese que la enumeración nos ofrezca los registros en un orden determinado. Podemos hacer que se ordenen proporcionando nosotros el criterio de ordenación. Para ello deberemos crear un comparador de registros que nos diga cuando un registro es mayor, menor o igual que otro registro. Para crear este comparador

deberemos crear una clase que implemente la interfaz `RecordComparator`:

```
public class MiComparador implements RecordComparator {
    public int compare(byte [] reg1, byte [] reg2) {
        if( /* reg1 es anterior a reg2 */ ) {
            return RecordComparator.PRECEDES;
        } else if( /* reg1 es posterior a reg2 */ ) {
            return RecordComparator.FOLLOWS;
        } else if( /* reg1 es igual a reg2 */ ) {
            return RecordComparator.EQUIVALENT;
        }
    }
}
```

De esta manera, dentro del código de esta clase deberemos decir cuando un registro va antes, después o es equivalente a otro registro, para que el enumerador sepa cómo ordenarlos. Ahora, cuando creamos el enumerador deberemos proporcionarle un objeto de la clase que hemos creado para que realice la ordenación tal como lo hayamos especificado en el método `compare`:

```
RecordEnumeration re =
    rs.enumerateRecords(new MiComparador(), null, false);
```

Una vez hecho esto, podremos recorrer los registros del enumerador como hemos visto anteriormente, con la diferencia de que ahora obtendremos los registros en el orden indicado.

### 7.8.3.2. Filtrado de registros

Es posible que no queramos que el enumerador nos devuelva todos los registros, sino sólo los que cumplan unas determinadas características. Es posible realizar un filtrado para que el enumerador sólo nos devuelva los registros que nos interesan. Para que esto sea posible deberemos definir qué características cumplen los registros que nos interesan. Esto lo haremos creando una clase que implemente la interfaz `RecordFilter`:

```
public class MiFiltro implements RecordFilter {
    public boolean matches(byte [] reg) {
        if( /* reg nos interesa */ ) {
            return true;
        } else {
            return false;
        }
    }
}
```

De esta forma dentro del método `matches` diremos si un determinado registro nos

interesa, o si por lo contrario debe ser filtrado para que no aparezca en la enumeración. Ahora podremos proporcionar este filtro al crear la enumeración para que filtre los registros según el criterio que hayamos especificado en el método `matches`:

```
RecordEnumeration re =
    rs.enumerateRecords(null, new MiFiltro(), false);
```

Ahora cuando recorramos la enumeración, sólo veremos los registros que cumplan los criterios impuestos en el filtro.

#### 7.8.4. Notificación de cambios

Es posible que queramos que en cuanto haya un cambio en el almacén de registros se nos notifique. Esto ocurrirá por ejemplo cuando estemos trabajando con la copia de los valores de un conjunto de registros en memoria, y queramos que esta información se mantenga actualizada con los últimos cambios que se hayan producido en el almacén.

Para estar al tanto de estos cambios deberemos utilizar un listener, que escuche los cambios en el almacén de registros. Este listener lo crearemos implementando la interfaz `RecordListener`, como se muestra a continuación:

```
public class MiListener implements RecordListener {
    public void recordAdded(RecordStore rs, int id) {
        // Se ha añadido un registro con identificador id a rs
    }
    public void recordChanged(RecordStore rs, int id) {
        // Se ha modificado el registro con identificador id en rs
    }
    public void recordDeleted(RecordStore rs, int id) {
        // Se ha eliminado el registro con identificador id de rs
    }
}
```

De esta forma dentro de estos métodos podremos indicar qué hacer cuando se produzca uno de estos cambios en el almacén de registros. Para que cuando se produzca un cambio en el almacén de registros se le notifique a este listener, deberemos añadir el listener en el correspondiente almacén de registros de la siguiente forma:

```
rs.addRecordListener(new MiListener());
```

De esta forma cada vez que se realice alguna operación en la que se añadan, eliminen o modifiquen registros del almacén se le notificará a nuestro listener para que éste pueda realizar la operación que sea necesaria.

Por ejemplo, cuando creamos una enumeración con registros poniendo a `true` el

parámetro para que mantenga en todo momento actualizados los datos de la enumeración, lo que hará será utilizar un listener para ser notificada de los cambios que se produzcan en el almacén. Cada vez que se produzca un cambio, el listener hará que los datos de la enumeración se actualicen.

## 8. Flujos de E/S y serialización de objetos. RMS - Ejercicios

### 8.1. Gestión de productos que implementan Serializable

Vamos a hacer una aplicación Java para gestionar una lista de productos que vende nuestra empresa. Escribiremos la información de estos productos en un fichero, para almacenarlos de forma persistente. A partir del proyecto `java-serializacion` de las plantillas de la sesión se pide:

- a) Introducir el código necesario en el método `almacenar` de la clase `GestorProductos` para guardar la información de los productos en el fichero definido en la constante `FICHERO_DATOS`. Guardaremos esta información codificada en un fichero binario. Debemos codificar los datos de cada producto (título, autor, precio y disponibilidad) utilizando un objeto `DataOutputStream`.
- b) Introducir en el método `recuperar` el código para cargar la información de este fichero. Para hacer esto deberemos realizar el procedimiento inverso, utilizando un objeto `DataInputStream` para leer los datos de los productos almacenados. Leeremos productos hasta llegar al final del fichero, cuando esto ocurra se producirá una excepción del tipo `EOFException` que podremos utilizar como criterio de parada.
- c) Modificar el código anterior para, en lugar de codificar manualmente los datos en el fichero, utilizar la serialización de objetos de Java para almacenar y recuperar objetos `ProductoTO` del fichero.

### 8.2. Almacén de notas con serialización para JavaME

Vamos a implementar un almacén de notas. En el directorio `Notas` tenemos la base de esta aplicación. Cada nota será un objeto de la clase `Mensaje`, que tendrá un asunto y un texto. Además incorpora métodos de serialización y deserialización.

Vamos a almacenar estos mensajes de forma persistente utilizando RMS. Para ello vamos a utilizar el patrón de diseño adaptador, completando el código de la clase `AdaptadorRMS` de la siguiente forma:

- En el constructor se debe abrir el almacén de registros de nombre `RS_DATOS` (creándolo si es necesario). En el método `cerrar` cerraremos el almacén de registros abierto.
- En `listaMensajes` utilizaremos un objeto `RecordEnumeration` para obtener todos los mensajes almacenados. Podemos utilizar la deserialización definida en el objeto `Mensaje` para leerlos. Conforme leamos los mensajes los añadiremos a un vector.

El índice de cada registro en el almacén nos servirá para luego poder modificar o eliminar dicho mensaje. Por esta razón deberemos guardarnos este valor en alguna

parte. Podemos utilizar para ello el campo `rmsID` de cada objeto `Mensaje` creado.

En `getMensaje` deberemos introducir el código para obtener un mensaje dado su identificador de RMS.

- En `addMensaje` deberemos introducir el código necesario para insertar un mensaje en el almacén. Con esto ya podremos probar la aplicación, añadiendo mensajes y comprobando que se han añadido correctamente. Probar a cerrar el emulador y volverlo a abrir para comprobar que los datos se han guardado de forma persistente.
- En `removeMensaje` deberemos eliminar un mensaje del almacén dado su identificador.
- (\*) En `updateMensaje` modificaremos un mensaje del almacén sobrescribiéndolo con el nuevo mensaje proporcionado. Para ello obtendremos el identificador correspondiente a al mensaje antiguo de su campo `rmsID`.
- (\*) Obtener la información de la cantidad de memoria ocupada y disponible a partir del objeto `RecordStore`. Deberemos hacer que los métodos `getLibre` y `getOcupado` de la clase `AdaptadorRMS` devuelvan esta información.

