Interfaz de usuario

Índice

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`.

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();
```

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.

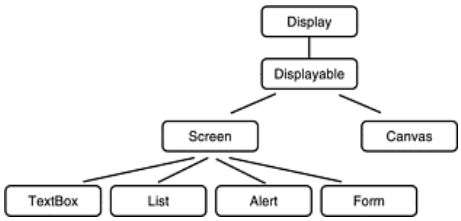
2.1. API de alto nivel

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.

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.



Jerarquía de elementos displayables

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`).

3.1. 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:

`TextField.ANY`	Cualquier texto
`TextField.NUMERIC`	Números enteros
`TextField.PHONENUMBER`	Números de teléfono
`TextField.EMAILADDR`	Direcciones de e-mail
`TextField.URL`	URLs
`TextField.PASSWORD`	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.

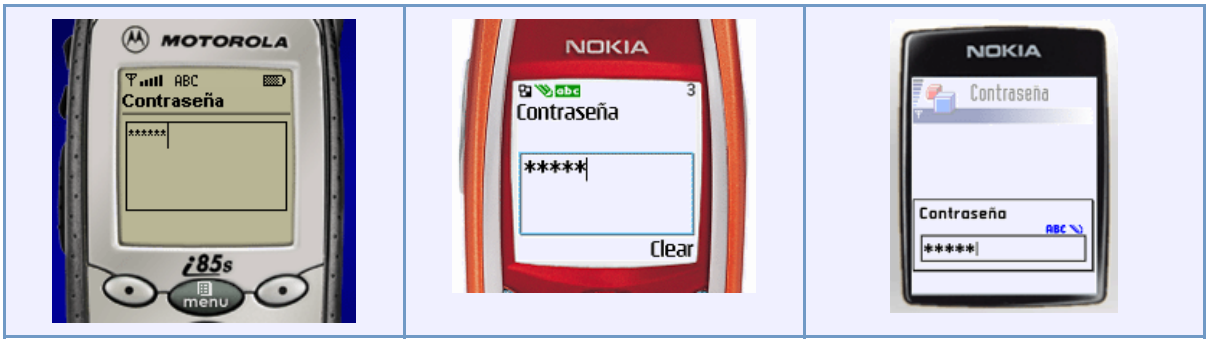
Además tiene métodos con los que podremos modificar el contenido del cuadro de texto, insertando, modificando o borrando caracteres o bien cambiando todo el texto, así como para obtener información sobre el mismo, como el número de caracteres que se han

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:



3.2. 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`:

`Choice.IMPLICIT`	Lista implícita

`Choice.EXCLUSIVE`	Lista exclusiva
`Choice.MULTIPLE`	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);
```

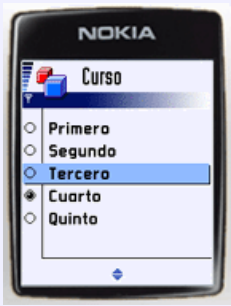
Donde `texto` será la cadena de texto que se muestre, e `imagen` será una imagen que podremos poner a dicho elemento de la lista de forma opcional. Si no queremos poner ninguna imagen podemos especificar `null`.

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);
```

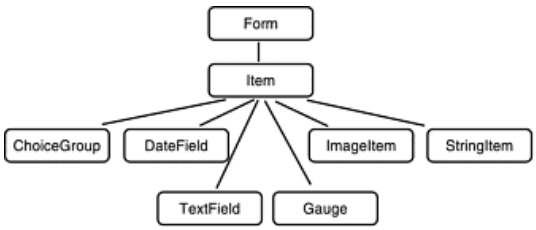
A continuación se muestra el aspecto de los distintos tipos de listas existentes:

IMPLÍCITA	EXCLUSIVA	MÚLTIPLE
		

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.

3.4. 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 `titulo` 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`:

`AlertType.ERROR`	Muestran un mensaje de error de la aplicación.
`AlertType.WARNING`	Muestran un mensaje de advertencia.
`AlertType.INFO`	Muestran un mensaje de información.
`AlertType.CONFIRMATION`	Muestran un mensaje de confirmación de alguna acción realizada.
`AlertType.ALARM`	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.

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.

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.

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 conexion en red con la URL de la imagen
String url = "http://jtech.ua.es/imagenes/logo.png";
ContentConnection con = (ContentConnection) Connector.open(url);
InputStream in = con.openInputStream();

// 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.

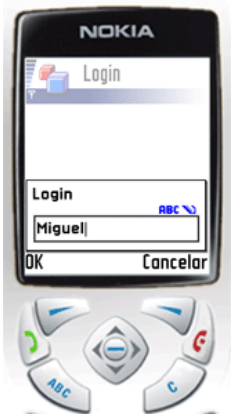
5. Comandos de entrada

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,

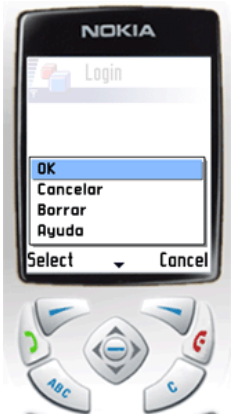
de forma que el usuario pueda seleccionar y ejecutar uno de ellos. Esta es una forma de interacción de alto nivel, que se implementará a nivel nativo y que será totalmente portable.

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

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:

`Command.OK`	Dar una respuesta positiva
`Command.BACK`	Volver a la pantalla anterior
`Command.CANCEL`	Dar una respuesta negativa
`Command.EXIT`	Salir de la aplicación
`Command.HELP`	Mostrar una pantalla de ayuda
`Command.STOP`	Detener algún proceso que se esté realizando
`Command.SCREEN`	Comando propio de nuestra aplicación para la pantalla actual.
`Command.ITEM`	Comando específico para ser aplicado al item 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 esta 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);
```

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 que *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.

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
            }
        }
    }
}
```

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 item 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:

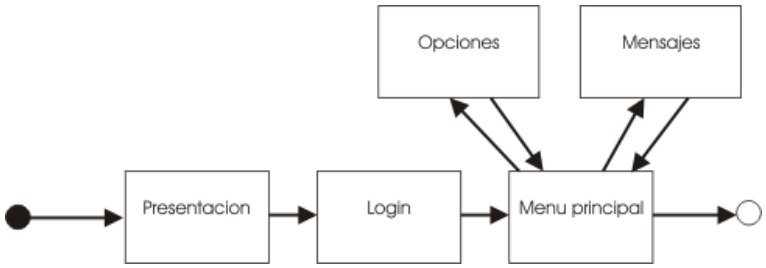
```
class MiListener implements ItemStateListener {
    public void itemStateChanged(Item i) {
        // Se ha modificado el item i
    }
}
```

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.



Mapa de 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.

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`.

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*.