Herencia e interfaces. MIDlets e interfaz de usuario

Índice

1 Herencia e interfaces 2
1.1 Herencia 2
1.2 Punteros this y super 2
1.3 Interfaces y clases abstractas 4
2 Colecciones de datos 5
2.1 Colecciones 5
2.2 Polimorfismo e interfaces 15
2.3 Tipos de datos básicos en las colecciones 16
3 MIDlets e interfaz de usuario 17
3.1 Acceso al visor 17
3.2 Componentes disponibles 17
3.3 Componentes de alto nivel 19
3.4 Imágenes 26
3.5 Comandos de entrada 28
3.6 Transiciones entre pantallas 33
3.7 Interfaz gráfica de bajo nivel 36

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.

1. Herencia e interfaces

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.

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 implicita 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.

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*.

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.

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 heredará 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.

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 podriamos 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.

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, comparandolos 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, devolviendonos *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)*.

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 si 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.

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)
```

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.

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 leido
        }
```

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 leido. 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);
```

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.

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 utiles 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.

2.3. Tipos de datos básicos en las colecciones

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.

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. 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`.

3.1. 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();
```

3.2. 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.

3.2.1. API de alto nivel

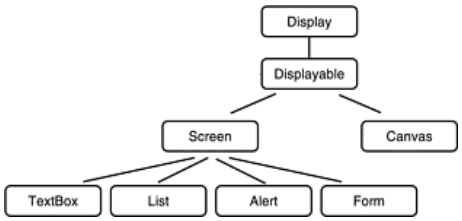
Consiste en una serie de elementos predefinidos: `Form`, `List`, `Alert` y `TextBox` que son extensiones de la clase abstracta `Screen`. Estos son elementos comunes que podemos encontrar en la interfaz de todos los dispositivos, por lo que el tenerlos predefinidos nos permitirá utilizarlos de forma sencilla 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.

3.2.2. 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

3.3. 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`).

3.3.1. 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.

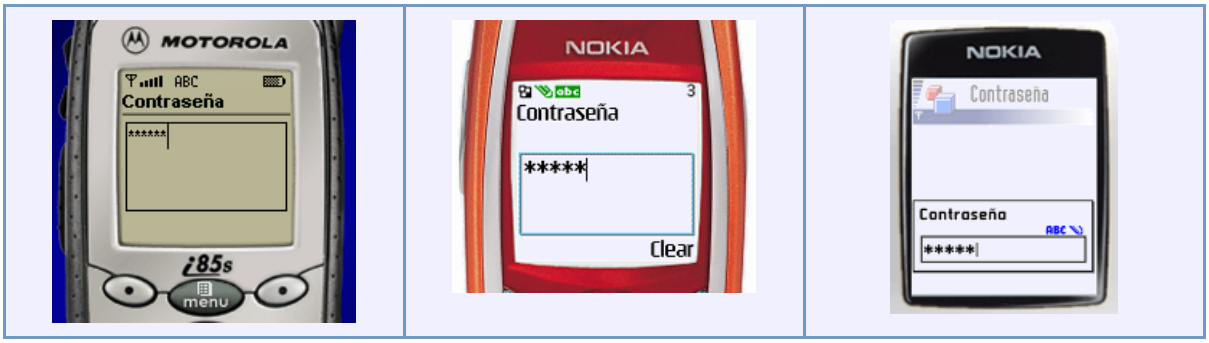
Además tiene métodos con los que podremos modificar el contenido del cuadro de texto, insertando, modificando o borrando caracteres o bien cambiando todo el texto, así como para obtener información sobre el mismo, como el número de caracteres que se han

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:



3.3.2. 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);
```

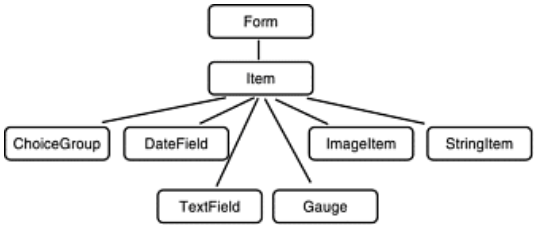
A continuación se muestra el aspecto de los distintos tipos de listas existentes:

IMPLÍCITA	EXCLUSIVA	MÚLTIPLE
		

3.3.3. 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.

3.3.4. 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.

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.

3.4.1. 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.

3.4.2. 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.

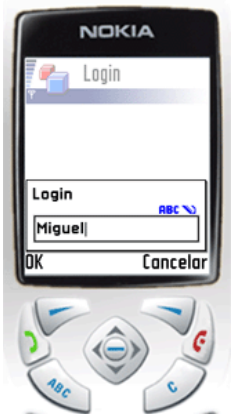
3.5. Comandos de entrada

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

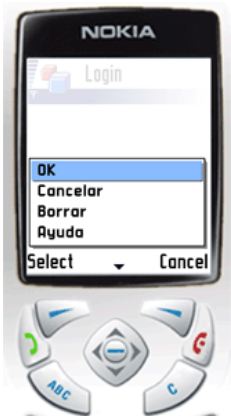
portable.

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

3.5.1. 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);
```

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 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.

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
            }
        }
    }
}
```

3.5.4. 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:

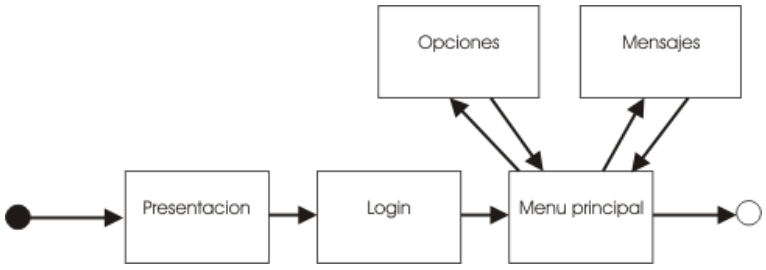
```
class MiListener implements ItemStateListener {
    public void itemStateChanged(Item i) {
        // Se ha modificado el item i
    }
}
```

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.



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.

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`.

3.6.2. 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*.

3.7. 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.

3.7.1. 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.

3.7.1.1. 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 dibujado 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.

3.7.1.2. 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.

3.7.1.3. 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*.

3.7.2. 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.

3.7.2.1. 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:

`Graphics.SOLID`	Línea sólida (se dibujan todos los *pixels*)
`Graphics.DOTTED`	Línea punteada (se salta algunos *pixels* 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);
```

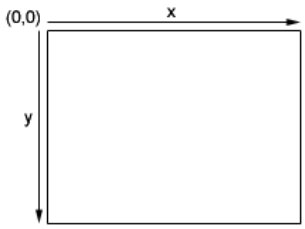
También tenemos disponible el siguiente método:

```
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:

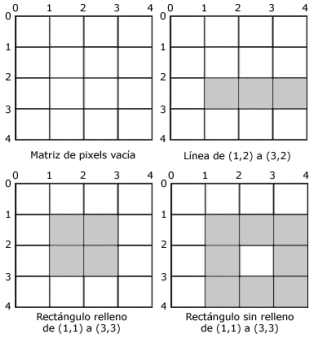


Sistema de coordenadas del área de dibujo

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 *píxel* 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

3.7.2.2. 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 *píxels* 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.

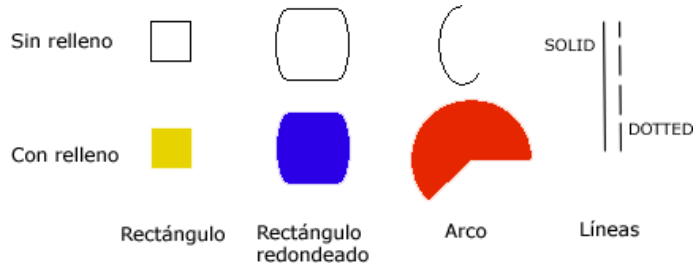
  Podemos también dibujar rectángulos con las esquinas redondeadas, utilizando los métodos:

```
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);
        }
}
```

3.7.2.3. Puntos anchor

En MIDP se introduce una características 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.

3.7.2.4. Cadenas de texto

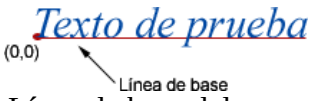
Podemos dibujar una cadena de texto utilizando el método `drawString`. Deberemos proporcionar la cadena de texto de dibujar y el punto *anchor* donde dibujarla.

```
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:



Línea de base del texto

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:

Métricas del textu

3.7.2.5. 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);
```

3.7.3. 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.

3.7.3.1. 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*.

3.7.3.2. 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);
        }
    }
}
```

3.7.3.3. 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.

3.7.3.4. 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.

3.7.3.5. 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.

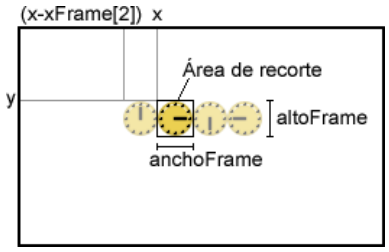
Imagen con los frames de una animación

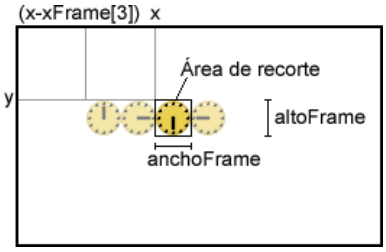
De esta forma estamos reduciendo el número de ficheros que incluimos en el JAR de la aplicación, por lo que por una lado reduciremos el espacio de este fichero, y por otro lado tendremos que abrir sólo un fichero, y no varios.

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 tendra 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);
```

3.7.4. 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
    }
}
```

3.7.4.1. 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:

`Canvas.KEY_NUM0`	0
`Canvas.KEY_NUM1`	1
`Canvas.KEY_NUM2`	2
`Canvas.KEY_NUM3`	3
`Canvas.KEY_NUM4`	4
`Canvas.KEY_NUM5`	5
`Canvas.KEY_NUM6`	6
`Canvas.KEY_NUM7`	7
`Canvas.KEY_NUM8`	8
`Canvas.KEY_NUM9`	9
`Canvas.KEY_POUND`	#
`Canvas.KEY_STAR`	*

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`.

3.7.4.2. 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.

3.7.4.3. 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.

3.7.5. 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`.

3.7.5.1. 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.

3.7.5.2. 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.

3.7.5.3. 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.

3.7.5.4. 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.