Mobile Information Device Profile (MIDP II)

Florina Almenárez Mendoza
Celeste Campo
Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
florina@it.uc3m.es, celeste@it.uc3m.es

Parte de este material se basa en transparencias de Natividad Martínez Madrid (nati@it.uc3m.es)

Índice

- Generalidades y conceptos básicos
  - MIDlets y MIDlet Suite
  - desarrollo y despliegue
- Librerías de MIDP
- **Interfaz de usuario**
  - API de alto nivel
  - **API de bajo nivel**
  - API de juegos
- Almacenamiento persistente
- Conectividad

API UI de bajo nivel

- Control gráfico de la pantalla a nivel de píxel.
- Definición de eventos propios de entrada:
  - Detección de eventos de teclado.
  - Detección de eventos de puntero.
- Más flexibilidad, menos portabilidad:
  - El resultado final puede depender del teléfono concreto en el que se ejecute.
- Clases `Canvas` y `Graphics`
- Clase fundamental para la realización de juegos:
  - Mejorada en MIDP 2.0 con la extensión **`GameCanvas`**.

API UI de bajo nivel
Clase `Canvas`

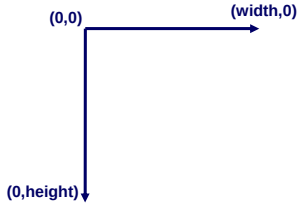
- Subclase abstracta de `Displayable` que permite realizar interfaces gráficos de bajo nivel en MIDP.
- Necesario obtener el tamaño del `display` y programar teniendo en cuenta estas dimensiones:
  - **`int getWidth()`**, **`int getHeight()`**
- Método **`void paint(Graphics g);`**
  - Debe pintar todos los píxeles de la pantalla.
- Métodos para la gestión de eventos a bajo nivel
  - Entradas teclado.
  - Puntero pantalla táctil.
  - Cuando se visualiza el `Canvas` en el `display`:
    - **`showNotify:`** antes de visualizarlo.
    - **`hideNotify:`** después de visualizarlo.

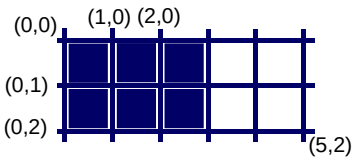
API UI de bajo nivel
Clase `Canvas` (II)

- Una clase que extienda `Canvas`:
  - Debe implementar obligatoriamente el método (abstracto) **`paint`**.
  - No es necesario que implemente todos los métodos relacionados con eventos a bajo nivel:
    - No son métodos abstractos y su implementación por defecto es vacía (no hacen nada).
    - El desarrollador sólo debe implementar los métodos correspondientes a los eventos que quiere gestionar.

API UI de bajo nivel
Clase `Graphics`

- Similar a `java.awt.Graphics`: geometría bidimensional
- Se pasa como parámetro al método **`paint`** de **`Canvas`**
- Sistema de coordenadas empieza en el extremo superior izquierdo.





- Métodos para dibujar:
  - `void drawLine(int x1, int y1, int x2, int y2)`
  - `void drawRect(int x, int y, int width, int height)`

API UI de bajo nivel
Clase `Graphics` (II)

- Métodos para dibujar:
  - `void` **`drawArc`**`(int x, int y, int width, int height, int startAngle, int arcAngle)`
  - `void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`
  - `void` **`fillArc`**`(int x, int y, int width, int height, int startAngle, int arcAngle)`
  - `void fillRect(int x, int y, int width, int height)`
  - `void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`
  - **`void drawImage(Image img, int x, int y, int anchor)`**
- Métodos de soporte a colores:
  - `int getBlueComponent() | int getRedComponent()`
  - `int getColor() | void setColor(int RGB)`
  - `void setColor(int red, int green, int blue)`
  - `int getGrayScale() | void setGrayScale(int value)`

API UI de bajo nivel
Ejemplo `Canvas` y `Graphics`

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

public class LineCanvasExample extends MIDlet {
  private Display display;
  public LineCanvasExample() {
    display=Display.getDisplay(this);
  }
  public void startApp() throws MIDletStateChangeException {
    display.setCurrent(new LineCanvas());
  }
  public void pauseApp() {
  }
  public void destroyApp(boolean unconditional) {
  }
```

API UI de bajo nivel
Ejemplo `Canvas` y `Graphics` (II)

```
class LineCanvas extends Canvas {
  public void paint(Graphics g) {
    // Tamaño del área de dibujo
    int width=this.getWidth();
    int height=this.getHeight();
    // Fondo de la pantalla blanco
    g.setColor(255,255,255); //0xFFFFFF
    g.fillRect(0,0,width,height);
    // Líneas en negro
    g.setColor(0,0,0);
    // Dibujamos dos líneas (ejes)
    g.drawLine(0,height,width,0);
    g.drawLine(0,0,width,height);
  }
}
}
```

API UI de bajo nivel
Ejemplo `Canvas` y `Graphics` (III)

Ver código: **`LineCanvasExample.zip`** (proyecto WTK) LineTest**`Canvas.zip`** (proyecto WTK)

API UI de bajo nivel
Eventos

- Eventos de teclado:
  - Métodos que gestionan los eventos:
    - `void keyPressed(int keyCode)`
    - `void keyReleased(int keyCode)`
    - `void keyRepeated(int keyCode)`
  - Cada tecla está definida por un código `KEY_NUM1`,..., `KEY_NUM9`, `KEY_STAR` (*) y `KEY_POUND` (#).
- Acciones de juego:
  - Se definen una serie de códigos: `UP`, `DOWN`, `RIGHT`, `LEFT`, `FIRE`, `GAME_A`, `GAME_B`, `GAME_C` y `GAME_D`.
  - Mapeo a teclas con los métodos: `getKeyCode(int gameAction)` y `getGameAction(int keyCode)`
- Eventos de puntero:
  - Métodos que gestionan los eventos:
    - `void pointerPressed(int x, int y)`
    - `void pointerReleased(int x, int y)`
    - `void pointerDragged(int x, int y)`

Índice

- Generalidades y conceptos básicos
  - MIDlets y MIDlet Suite
  - desarrollo y despliegue
- Librerías de MIDP
- **Interfaz de usuario**
  - API de alto nivel
  - API de bajo nivel
  - **API de juegos**
- Almacenamiento persistente
- Conectividad

API para juegos

- Introducida en MIDP 2.0 para mejorar el soporte de desarrollo de juegos 2D.
- Principal ventaja: el `display` puede dividirse en capas de forma que se puede tratar cada una de ellas de forma independiente.
- Paquete:
  - **`javax.microedition.lcdui.game`**
- Proporciona cinco nuevas clases:
  - **`GameCanvas`**
  - **`Layer`**
  - **`LayerManager`**
  - **`Sprite`**
  - **`TiledLayer`**

API para juegos
Clase Canvas

- Para programar juegos en MIDP 1.0 se empleaba la clase **`Canvas`**, y los programas tenían una estructura similar a la siguiente:

```
public class JuegoCanvas extends Canvas implements Runnable {
  public void run() {
    while (true) {
      // Modifica el estado del juego
      repaint();
      // Espera un tiempo
    }
  }
  public void paint(Graphics g) {
    // Pintar la pantalla
  }
  protected void keyPressed(int keyCode) {
    // Responder a eventos de pulsación de teclas
  }
}
```

API para juegos
Clase `GameCanvas`

- Problemas en MIDP 1.0:
  - Tres hilos distintos:
    - Controla el estado del juego.
    - Pinta la pantalla.
    - Gestiona eventos.
  - Animación e interacción defectuosa.
- **`GameCanvas`** permite:
  - Tener un único hilo para controlar el juego, pintar en pantalla y gestionar eventos.
  - Construir sobre un buffer lo que se quiere representar en pantalla antes de visualizarlo: "*off-screen buffer*".
  - Emplear un mecanismo de "*polling*" para saber el estado de las teclas.

API para juegos
Clase `GameCanvas` (II)

- Con **`GameCanvas`** los programas tienen una estructura similar a la siguiente:

```
public class JuegoCanvas extends GameCanvas
                                 implements Runnable {
  public void run() {
    Graphics g = getGraphics();
    while (true) {
      // Se actualiza el estado del juego
      int keyState = getKeyStates();
      // Responde a eventos del teclado
      // Pintar en pantalla
      flushGraphics();
      // Esperar un tiempo
    }
  }
}
```

Devuelve el objeto **`Graphics`** correspondiente al "off-screen buffer".

Devuelve un entero (1 pulsada y 0 no pulsada).

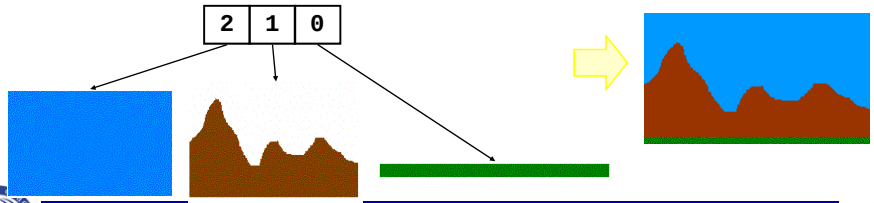
Permite volcar el contenido del "off-screen buffer" en la pantalla del dispositivo

API para juegos
Clase Layer

- Clase abstracta que representa un elemento visual
- Tiene tres propiedades:
  - Posición.
  - Tamaño.
  - Si es visible o no.
- Métodos de esta clase permiten obtener los valores de estas propiedades.
- El programador puede extender esta clase para ofrecer una funcionalidad específica:
  - Siempre debe realizar una implementación del método **`paint`**.
- API MIDP 2.0 define dos subclases de **`Layer`**:
  - **`TiledLayer`**
  - **`Sprite`**

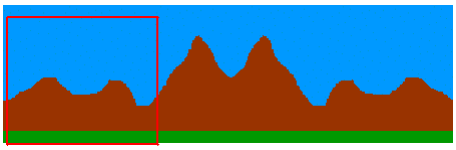
API para juegos
Clase LayerManager

- Clase que permite trabajar con varios objetos **`Layer`** en una aplicación.
- Mantiene una lista ordenada de **`Layers`**:
  - Para cada uno de ellos se puede obtener sus propiedades.
  - Los diferentes **`Layer`** se insertan en la lista según su profundidad:
    - Posición 0 indica que es el **`Layer`** más próximo al usuario.
    - Última posición es el **`Layer`** sobre el que se superponen todos los demás.



API para juegos
Clase LayerManager (II)

- Ventana visible:
  - Se puede indicar qué parte del **LayerManager** se presenta en pantalla.
  - Muy útil cuando se quieren generar barridos o panorámicas de una escena.
  - La ventana visible se establece a través del método:
    - **setViewWindow(int x, int y, int width, int height)**



Ventana visible

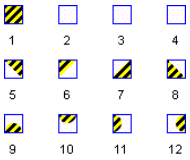
API para juegos
Clase LayerManager (III)

Método	Descripción
append(Layer l)	Añade un objeto **Layer**
insert(Layer l, int index)	Inserta un objeto **Layer** en una posición determinada en la lista de capas
getLayerAt(int)	Obtiene el objeto **Layer** que está en una determinada posición en la lista de capas
remove(Layer l)	Elimina el **Layer** especificado
setViewWindow(int x, int y, int width, int height)	Indica cuál es la ventana visible
paint(Graphics g, int x, int y)	Dibuja la ventana visible actual a partir de la posición x e y

API para juegos
Clase `TiledLayer`

- Clase que extiende de **`Layer`**.
- Representa un elemento visual compuesto por un conjunto de “baldosas”.
- Nos permite obtener como un rompecabezas donde podremos colocar las diferentes piezas en distintas posiciones.

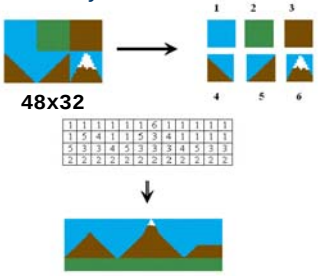
64x48



16x16

API para juegos
Clase `TiledLayer` (II)

- Cuando se crean las baldosas se crea también una matriz de un determinado número de posiciones:
  - Cada posición hace referencia a una de las baldosas.
  - Las posiciones de la matriz empiezan en 0 y las baldosas se numeran a partir del 1.



48x32

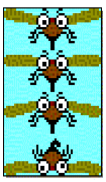
```
Image image = Image.createImage("/board.png");
TiledLayer tiledLayer = new TiledLayer(12, 4, image, 16, 16);
```

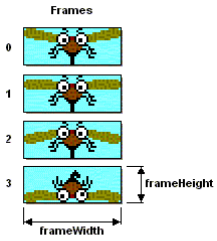
API para juegos
Clase `TiledLayer` (III)

Método	Descripción
`getColumns()`	Obtiene el número de columnas
`getRows()`	Obtiene el número de filas
`getCellHeight()`	Obtiene la altura de las celdas. Es igual para todas las celdas
`getCellWidth()`	Obtiene la anchura de las celdas. Es igual para todas las celdas
`getCell(int col, int row)`	Obtiene el índice de la baldosa que se encuentra en esa posición
`setCell(int col, int row, int tileIndex)`	Asigna una baldosa a una posición de la matriz
`paint(Graphics g)`	Dibuja el objeto en pantalla. Si pertenece a un **`LayerManager`**, éste se invoca cuando se llama al **`paint`** del **`LayerManager`**

API para juegos
Clase `Sprite`

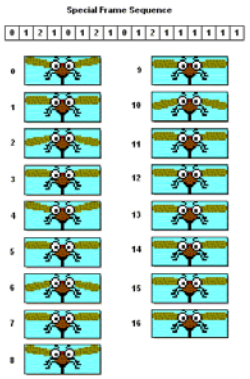
- Clase que extiende de **`Layer`**.
- Divide una imagen en una serie de *frames* que se pueden reproducir en una determinada secuencia y de esta manera realizar animaciones.






```
Image image = Image.createImage("/board.png");
Sprite sprite = new Sprite(image, frameWidth, frameHeight);
```

API para juegos
Clase `Sprite` (II)



```
int[] sequence = {0,1,2,1,0,1,2,1,0,1,2,1,1,1,1,1,1}
sprite.setFrameSequence(sequence);
```


API para juegos
Clase `Sprite` (III)

Método	Descripción
`setFrameSequence(int[] sequence)`	Secuencia cíclica de *frames* a reproducir
`getFrame()`	Devuelve el índice del *frame* que está siendo mostrado
`nextFrame()`, `prevFrame()`	Selecciona el siguiente y el anterior *frame*, respectivamente
`paint(Graphics g)`	Permite dibujar el objeto en pantalla
`setTransform(int transform)`	Permite aplicar transformaciones sobre el objeto **`Sprite`**: rotarlo 90º , 180º y 270º
`setRefPixelPosition(int x, int y)`	Permite fijar un píxel de referencia para el **`Sprite`**, por defecto es el (0,0)

API para juegos
Ejemplo `LayerManager`



`Games-WormGame` (proyecto WTK) `Games-PushPuzzle` (proyecto WTK) `LayerManagerExample.zip`

Índice

- Generalidades y conceptos básicos
  - MIDlets y MIDlet Suite
  - desarrollo y despliegue
- Librerías de MIDP
- Interfaz de Usuario
  - API de alto nivel
  - API de bajo nivel
  - API para juegos
- **Almacenamiento persistente**
- Conectividad

Almacenamiento persistente

- API independiente del dispositivo.
- Base de datos sencilla orientada a registros (RMS)
  - *Record Management System*
  - registros (**`record`**) son array de bytes de tamaño variable
  - los registros se guardan en almacenes de registro (**`record stores`**).
  - los almacenes de registros se comparten entre MIDlets de un mismo MIDlet suite.
- Soporta enumeración, ordenamiento y filtrado.
- Actualización atómica de registros.
- Definido en el paquete **`javax.microedition.rms.`**

RMS
`Records`

- Los `records` se identifican unívocamente mediante el identificador (**`recordID`** de tipo **`int`**).
- Manipular `records`:
  - `int addRecord(byte[] data, int offset, int numBytes)`
  - `void deleteRecord(int recordId)`
  - `byte[] getRecord(int recordId)` y `int getRecord(int recordId, byte[] buffer, int offset)`
  - `void setRecord(int recordId, byte[] newData, int offset, int numBytes)`

Registro	Datos
1	Datos 1
2	Datos 2
...	...

RMS
Clase `RecordStore`

- Un *record store* es una colección de *records*
- Reglas:
  - Nombre único en un mismo MIDlet Suite.
  - El nombre puede ser una combinación de 32 caracteres (“*case sensitive*”).
  - Se almacenan en el mismo espacio de nombres de un MIDlet Suite.
  - Se comparten únicamente entre los MIDlets pertenecientes al MIDlet Suite.
- Operaciones:
  - **Crear/abrir:** `static RecordStore openRecordStore (String recordStoreName, boolean createIfNecessary)`
  - **Cerrar** (después de utilizarse): `void closeRecordStore()`
  - **Borrar** (debe estar cerrado): `static void deleteRecordStore(String recordStoreName)`

RMS
Cabecera `RecordStore`

- La cabecera de un `RecordStore` proporciona la siguiente información:
  - Número de `records` en el `RecordStore`:
    - **`int getNumRecords()`**
  - Número de versión:
    - **`int getVersion()`**
  - Fecha de la última modificación:
    - **`long getLastModified()`**
  - Identificador del siguiente recordID:
    - **`getNextRecordID()`**

RMS
Interfaz `RecordEnumeration`

- Después de borrar `records` sus identificadores ya no son consecutivos.
- Para recorrerlos se proporciona la clase **`RecordEnumeration`**:
  - Lista doblemente enlazada ⇨ cada nodo representa un `record`.
  - Se obtiene a través del método:
    - `RecordEnumeration enumerateRecords(RecordFilter f, RecordComparator comparator, boolean keepUpdated)`
  - Métodos:
    - **`void reset()`**: puntero al primer elemento de la lista.
    - **`int nextRecordId()`**: ID del siguiente elemento de la lista.
    - **`int previousRecordId()`**: ID del anterior elemento de la lista.
  - Se define como interfaz pero los fabricantes deben realizar una implementación de ella: para los desarrolladores es una clase.

RMS
Interfaces `RecordFilter` y `RecordComparator`

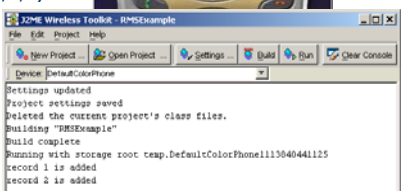
- Permiten ordenar o clasificar los `records` en un `RecordStore` según algún criterio.
- Interfaz **`RecordFilter`**:
  - **`boolean matches(byte[] candidate)`**
  - La aplicación determina si el `record` (**`candidate`**) verifica el criterio de selección.
- Interfaz **`RecordComparator`**:
  - **`int compare(byte[] rec1, byte[] rec2)`**
  - Clasificación de records por algún criterio (ej. fecha creación)
  - Devuelve el orden de `records` (**`rec1`** y **`rec2`**):
    - **`PRECEDES`**, **`FOLLOWS`**, **`EQUIVALENT`**.

RMS
Interfaz `RecordListener`

- Monitorizar cambios en `RecordStores`.
- Gestión de eventos mediante los métodos:
  - `void recordAdded(RecordStore recordStore, int recordId)`
  - `void recordChanged(RecordStore recordStore, int recordId)`
  - `void recordDeleted(RecordStore recordStore, int recordId)`
- Para añadir y borrar `listeners`:
  - `void addRecordListener(RecordListener listener)`
  - `void removeRecordListener(RecordListener l)`

RMS
Ejemplo

```
public void startApp() throws MIDletStateChangeException {
   RecordStore rs = null;
   try {
     rs = RecordStore.openRecordStore("file1", true);
     byte data[] = new byte[4];
     for ( int j=0; j<2; j++) {
       int i = rs.getNextRecordID();
       data[0] = (byte)((i >> 24) & 0xff);
       data[1] = (byte)((i >> 16) & 0xff);
       data[2] = (byte)((i >> 8) & 0xff);
       data[3] = (byte)(i & 0xff);
       System.out.println("record " + rs.addRecord(data,0,4) +
                          " is added");
     }
   } catch (Exception e) {}
   finally{
     try {
         rs.closeRecordStore();
     } catch (Exception e) {}
   }
   destroyApp(true);
   notifyDestroyed();
}
```



Ver código: **`RecordStoreTest.java`** – **`RMSExample.zip`** (proyecto WTK)

Índice

- Generalidades y conceptos básicos
  - MIDlets y MIDlet Suite
  - desarrollo y despliegue
- Librerías de MIDP
- Interfaz de Usuario
  - API de alto nivel
  - API de bajo nivel
  - API para juegos
- Almacenamiento persistente
- **Conectividad**

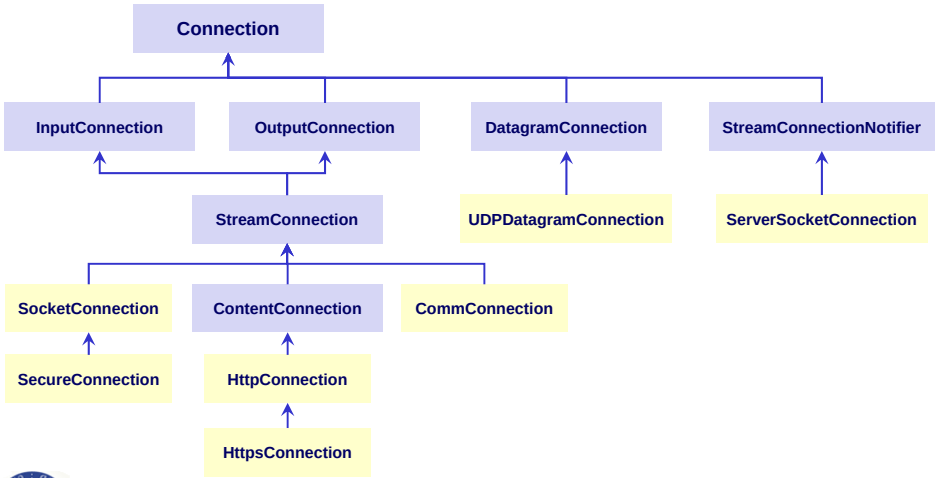
Conectividad

- Implementa el *Generic Connection Framework* (GCF) definido en el paquete **`javax.microedition.io`**:
  - Requiere soporte de conexiones HTTP (RFC 2616) como cliente.
- Añade e implementa el interfaz **`HttpConnection`**, hereda directamente del interfaz **`ContentConnection`**.
- La implementación del interfaz **`DatagramConnection`**, definido en CLDC es opcional, pero recomendable.

GCF
Interfaces de conexión en MIDP

- **`CommConnection`**: extiende de **`StreamConnection`**
  - Conexión a puerto serie (**`comm:`**).
- **`HttpConnection`**: extiende **`ContentConnection`**
  - Conexión HTTP (**`http:`**).
- **`HttpsConnection`**: extiende **`HttpConnection`**
  - Conexión HTTP segura (**`https`**:).
- **`SocketConnection`**: extiende de **`StreamConnection`**
  - Socket TCP/IP (**`socket:`**)
- **`ServerSocketConnection`**: extiende de **`StreamConnectionNotifier`**
  - Socket pasivo TCP/IP (**`socket:`**)
- **`SecureConnection`**: extiende de **`SocketConnection`**
  - Socket seguro (**`ssl:`**)
- **`UDPDatagramConnection`**: extiende de **`DatagramConnection`**
  - Conexión de datagramas TCP/IP (**`datagram:`**)

GCF
Interfaces de conexión en MIDP



GCF
Clase `Connector`

- Método open:
  - Conexión HTTP:
    - **`Connector.open("http://www.it.uc3m.es/")`**
  - Conexión datagrama:
    - **`Connector.open("datagram://www.webyu.com:9000")`**
  - Conexión puerto serie:
    - **`Connector.open("comm:0;baudrate=9600")`**
  - Conexión segura:
    - **`Connector.open("ssl://host.com:79")`**

http://developers.sun.com/mobility/midp/articles/genericframework/

GCF
Clase `HTTPConnection`

- Tres estados de la conexión: *Setup*, *connected* y *closed*
- Transición de *Setup* a *Connected* motivada por cualquier método que requiera enviar o recibir datos.

Métodos	Estado
`setRequestMethod` `setRequestProperty`	*Setup*

`openInputStream`
`openDataInputStream`
`getLength, getType, getEncoding`
`getHeaderField`
`getResponseCode`
`getResponseMessage`
`getHeaderFieldInt`
`getHeaderFieldDate`
`getExpiration`
`getDate`
`getLastModified`
`getHeaderFieldKey`
	Connected

`close`
`getRequestMethod`
`getRequestProperty`
`getURL, getProtocol, getHost, getFile, getRef, getPort, getQuery`
	Setup o connected

GCF
Ejemplo HTTPConnection

```
private void download (String url) throws IOException {
   StringBuffer sb = new StringBuffer();
   InputStream is = null;
   HttpConnection c = null;
   TextBox t = null;
   try {
     long len = 0 ;
     int ch = 0;
     c = (HttpConnection)Connector.open(url);
     is = c.openInputStream();
     while ((ch = is.read()) != -1) {
       sb.append((char)ch); }
     t = new TextBox("Hola...", sb.toString(), 1024, 0);
   } finally {
       if (is != null) is.close();
       if (c != null) c.close();
   }
```

GCF
Ejemplo HTTPConnection

Ver código:
HTTPTest.java
(proyecto WTK)
HTTPExample.zip