Développement mobile MIDP 2.0

Frédéric BERTIN
fbertin@neotilus.com

API Haut Niveau

API haut niveau : intro

- L'API haut niveau est un framework :
  - Comprend tous les composants d'interface utilisateur simple.
  - Basé sur un concept de développement rapide d'applications (on utilise des éléments existants).
  - Beaucoup plus riche en classe que l'API bas niveau.
  - Prend complètement en charge l'affichage des composants.

Intercepter les événements utilisateur

- Dans l’API haut niveau, les événements utilisateur sont de deux types :
  - Evénement Command
    Déclenché par l’appui de l’utilisateur d’une touche du clavier
  - Evénement Item
    Déclenché par le changement graphique d’un objet `Item`

Objet Command

- L’objet `Command` contient toutes des informations sur l’événement intercepté :
  - Le type de commande exécuté
  - Le label de la commande
  - La priorité de l’action

- Marche à suivre pour intercepter un événement :
  - Créer un objet `Command`
  - Ajouter l’objet `Command` à un `Form`, `TextBox`, `List` ou `Canvas`
  - Créer un `listener`

- Dès la détection d’un événement, le listener va appeler la méthode `commandAction()`

Objet Command (exemple)

```
private Form form;      // Form
private Command cmFin; // Command Fin d'exécution
...
form = new Form("Test Command"); // Création du formulaire, avec un titre

// Création objet Command, avec label, type et priorité
cmFin = new Command("Fin", Command.EXIT, 1);
...
form.addCommand(cmFin);         // Ajout Command à Form
form.setCommandListener(this); // Listener d'événements sur form
...
public void commandAction(Command c, Displayable s)
{
   if (c == cmFin)
   {
        destroyApp(true);
        notifyDestroyed();
   }
}
```

Evénements Item

- A l'exception des items `Spacer`, `ImageItem` et `StringItem`, tous les `Items` on la possibilité d'intercepter des événements.

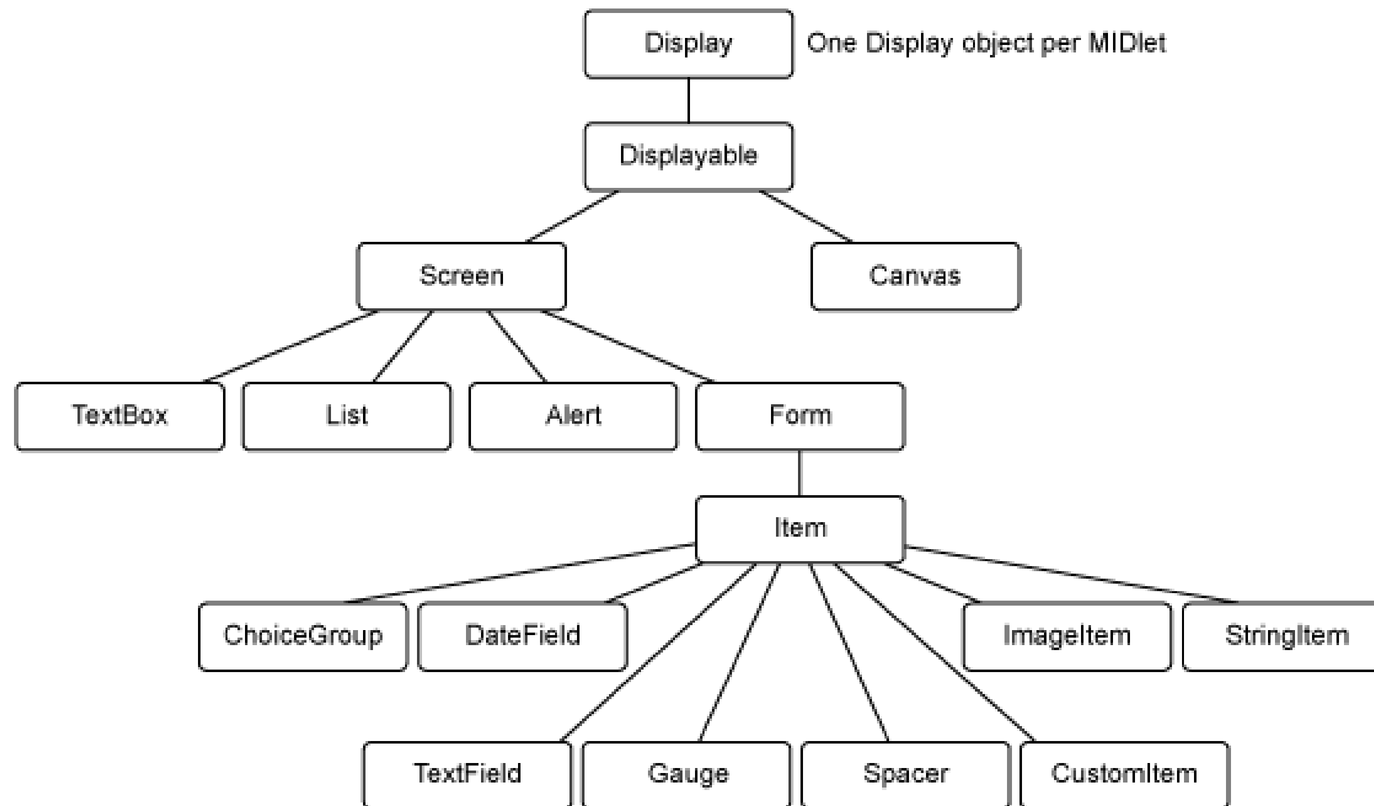
- Marche à suivre pour intercepter un événement :
  - Enregistrer un `ItemStateListener` sur le formulaire sur lequel on veut intercepter les événements d'`Item`.

- Dès la détection d'un événement sur un Item, le listener va appeler la méthode `itemStateChanged()`.
  - C'est dans cette méthode qu'on va analyser les paramètres de la commande (qui a envoyé la commande ?)

Evénements Item (exemple)

```
private Form form; // Form
private DateField dfToday; // DateField item
...

form = new Form(“Test ItemStateChanged"); // Création formulaire
dfToday = new DateField("Today:", DateField.DATE); // Création DateField
...

form.append(dfToday); // Ajout DateField au formulaire
form.setItemStateListener(this); // Listener événements formulaire
...

public void itemStateChanged(Item item)
{
   // Si c’est dfToday qui est à l’origine de l’événement
   if (item == dfToday)
   ...
}
```

Hiérarchie des objets haut niveau



Objet Display

- Une instance de `Display` par midlet (singleton).
  - On retrouve un display par `Display.getDisplay(MIDlet m)`

- Utilisé pour obtenir des infos sur l'affichage.
  - `getCurrent()` : retourne l'objet `Displayable` affiché par la MIDlet.

- Utilisé par les objets devant être affichés.
  - `Form`
  - `TextBox`
  - …

- Les objets devant être affichés appellent la méthode `setCurrent(Displayable d)`

```
Form f = new Form(" Form " );           //Creation d'un Formulaire
...
Display d = Display.getDisplay(this); //recupère le Display
d.setCurrent(f); // Demande au Display d'afficher le Form.
```

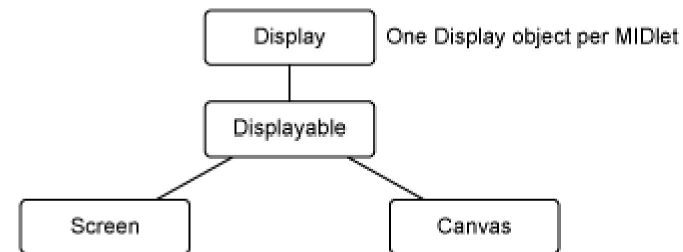
- L'objet `Display` de la MIDlet est le« manager » de l'affichage sur le téléphone.

Objet Displayable

- Classe abstraite. C'est la superclasse de tous les objets affichés par une MIDlet : ils sont tous« `Displayable`»

- Un objet `Displayable` a la capacité d'être affiché par l'objet `Display`.

- Dans une MIDlet, tous les objets avec une apparence graphique héritent de `Displayable`
  - `Form`
  - `TextBox`
  - `ChoiceGroup`
  - …



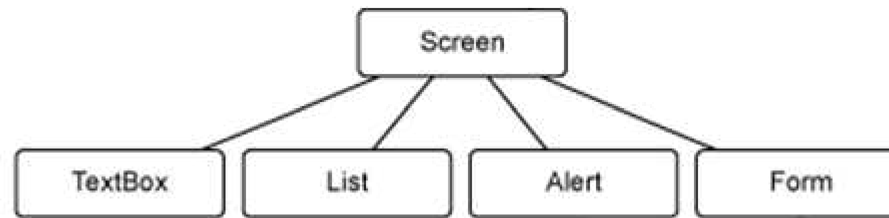
- MIDP contient deux sous classes de `Displayable`

```
public abstract class Displayable
public abstract class Screen extends Displayable   //API Haut niveau
public abstract class Canvas extends Displayable   //API bas niveau
```

Objet Screen

- Classe abstraite. C'est la superclasse de tous les objets « haut niveau », avec lesquels les utilisateurs interagissent.

- Les contenus affiché à l'écran et les interactions utilisateur sont définis par les sous classes de `Screen` :
  - `Alert`
  - `Form`
  - `List`
  - `TextBox`



- Il est recommandé que l'application change le contenu d'un objet `Screen` uniquement quand il n'est pas visible à l'écran (« off screen »), i e quand un autre objet `Displayable` est affiché.
  - Problème de performance
  - Source de confusion pour l'utilisateur

TextBox, List et Alert

`TextBox`

- Sous classe de `Screen` (haut niveau) qui permet à l'utilisateur de rentrer et d'éditer du texte.

`List`

- Sous classe de `Screen` (haut niveau) qui contient une liste de choix.
- L'utilisateur peut interagir et naviguer parmi les éléments listés.

`Alert`

- Sous classe de `Screen` (haut niveau) qui permet d'afficher des messages à l'utilisateur. Le but de cette classe est d'avertir l'utilisateur en cas d'erreur.
  - Une `Alert` peut contenir :
    - une chaîne texte,
    - une image (optionnel)
    - une gauge (optionnel) : barre de progression.
  - Une `Alert` peut attendre indéfiniment : elle devient modale. L'implémentation de la KVM propose alors à l'utilisateur de fermer l'alert.

Form

- Sous classe de `Screen` (haut niveau). C'est principalement un conteneur et manager d'objets Item.
- L'objet `Form` prend en charge l'affichage des items qui la composent par le `Display` : c'est lui qui hérite de `Displayable`.
- Les items d'un `Form` se mettent les uns en dessous des autres. Le form devient « scrollable » si les items à afficher dépassent l'affichage de l'écran.
- Les items sont managés de la manière suivante

  f.append(Item i)

    Ajoute l'item i à la form f

  f.insert (int itemIndex, Item item)

    insert dans le form l'Item item avant un autre item ayant pour index itemIndex.

  f.delete (int itemIndex)

    Supprime l'item ayant pour index itemIndex

  f.deleteAll()

    Supprime tous les items du form f

Item

- C'est la superclasse de tous les objets pouvant être affichés à un `Form`
- Les items d'un `Form` se mettent les uns en dessous des autres. Le `form` devient « scrollable » si les items à afficher dépassent l'affichage de l'écran.
- Les items sont managés de la manière suivante
  - `f.append(Item i)`
    Ajoute l'item i à la form f
  - `f.insert (int itemIndex, Item item)`
    insert dans le `form` l'Item item avant un autre item ayant pour index `itemIndex`.
  - `f.delete (int itemIndex)`
    Supprime l'item ayant pour index `itemIndex`
  - `f.deleteAll()`
    Supprime tous les items de la `form` f

Les Items de base

`DateField`

- Composant permettant d'éditer graphiquement la date et l'heure.

`Gauge`

- Un composant « progress bar », interactif (ajustement de volume) ou non (affichage valeur).

`StringItem`

- Elément non interactif, affichant une chaîne texte.

`TextField`

- Composant permettant d'éditer du texte

Les Items de base (suite)

`ChoiceGroup`

- Groupe d’éléments pouvant être sélectionnés (choix simples ou multiples).

`Spacer`

- Elément non interactif, utilisé pour séparer de multiples items sur une même ligne de formulaires.

`ImageItem`

- Item contenant une image (image donnée au constructeur).

CustomItem

- Sous classe abstraite de Item permettant de développer des éléments spécifiques (nouveauté MIDP 2.0).

- Offre un cadre de développement par ses méthodes abstraites :
  - `getMinContentHeight() et getMinContentWidth()`
    Doit retourner les valeurs minimum de hauteur et largeur du composant (en pixel)
  - `getPrefContentHeight() et getPrefContentWidth()`
    Doit retourner les valeurs préférées de largeur et hauteur du composant (en pixel)
  - `traverse()`
    Le comportement de l'item quand il est sélectionné
  - `paint()`
    Doit implémenter le rendu du composant (design graphique).

CustomItem : un squelette

```
public class MyItem extends CustomItem implements ItemCommandListener {

   public MyItem (String title, Display d, String caption1, String caption2) { ... }

   protected int getMinContentHeight() { ... }

   protected int getMinContentWidth() { ... }

   protected int getPrefContentHeight(int width) { ... }

   protected int getPrefContentWidth(int height) { ... }

   protected void paint(Graphics g, int w, int h) { ... }

   protected boolean traverse(int dir, int viewportWidth, int viewportHeight, int[]
   visRect_inout) { ...
      return true;
   }
   public void commandAction(Command c, Item i) { ... }

   protected void sizeChanged(int w, int h){ }
}
```

API Bas Niveau

API bas niveau : intro

- L'API haut niveau est flexible, mais trop limitative :
  - On ne peut pas dessiner directement sur le `Display`
  - On ne peut pas dessiner de lignes, de formes, …
- L'API bas niveau
  - Permet un contrôle total de l'affichage
  - Mais nécessite un développement plus lourd !

Canvas

- `Canvas` hérite de `Displayable` (au même titre que `Screen` en « haut niveau »)

- C'est la classe de base pour écrire des applications « bas niveau »

- La méthode `paint()` de `Canvas` est abstraite. C'est au développeur de fournir son implémentations. C'est dans cette méthode qu'on :
  - Dessinera des formes
  - Affichera des textes
  - Affichera des images
  - …

- Les attributs `height` et `width` de `Canvas` représentent la surface complète sur laquelle on peut dessiner : on ne spécifie pas une taille de `Canvas` !

Le système de coordonnées :

L'origine est en haut à gauche de l'affichage

```
int getWidth()   //Canvas width (x)
int getHeight () //Canvas heigh (y)
```

Canvas : Exemple

```
class TestCanvas extends Canvas implements CommandListener
{
   private Command cmdExit;
   ...
   Display display = Display.getDisplay(this);
   Command cmdExit = new Command("Exit", Command.EXIT, 1);
   addCommand(cmdExit);
   setCommandListener(this);
   ...
   protected void paint(Graphics g)
   {
        g.setColor(255, 255, 255);                  // Couleur de fond blanche
        g.fillRect(0, 0, getWidth(), getHeight()); // On remplit le Canvas
   }
}
...
TestCanvas canvas = new TestCanvas(this);
```

Graphics

- C'est la classe qu'on utilise pour dessiner des formes géométriques 2D ou afficher des textes.

- On peut dessiner des primitives graphiques :
  - `drawRect()`
  - `drawArc()`
  - `drawLine()`
  - …

- On affiche des images :
  - `drawImage()`

- On affiche des chaînes de caractères :
  - `drawString()`

API bas niveau : Game API

GameCanvas

- Un Canvas dédié aux IHM pour les jeux !
  - Sous classe abstraite de `Canvas`

- Contient un buffer off screen, créé pour chaque instanciation de GameCanvas
  - `getGraphics()` : obtient le buffer off screen
  - `flushGraphics()`

- Contient un moyen spécifique d'obtenir l'état des boutons sur lequel l'utilisateur a appuyé.
  - `getKeyStates()`

- L'intérèt du `GameCanvas` est de concentrer la logique du game design.

- **Attention :** créer un seul `GameCanvas` par MIDlet est conseillé afin de ménager la performance (double buffer gourmand en mémoire).

GameCanvas : exemple

```
public MyGameCanvas extends GameCanvas implements Runnable {
  public MyGameCanvas() { // instantiation code }
  public void start() {
    // Démarre le thread de jeu
    Thread runner = new Thread(this);
    runner.start();
  }
  private void run() {
    while(true) { //boucle infinie
            // Verifie l'état du jeu
            verifyGameState();

            // Récupère les événements utilisateurs
            checkUserInput();

            // Peint les éléments à l'écran réflétant l'état du jeu
            updateGameScreen(getGraphics());

            // Controle la fréquence de rafraichissement
            Thread.sleep(milliseconds);
    }
  }
}
```

Layer

- Classe abstraite représentant un élément du jeu.
- Chaque layer a une position et une taille, et peut être visible ou invisible.
- Méthodes principales :
  - `getHeigth()` et `getWidth()` :
    la hauteur, la longueur du layer.
  - `getX()` et `getY()` :
    la position du layer en X et Y
  - `move(int x,int y)` :
    Déplacement selon x et y.
  - `paint()` :
    Redéfinition du `paint()` du layer.
  - `isVisible() et setVisible(boolean)` :
    rendre le Layer visible ou non.

Sprite

- Sous classe de `Layer`, c'est l'élément visuel de base. Le `Sprite` est utilisé pour l'animation

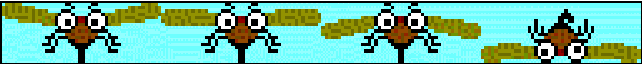
- Il est composé d'une image qui elle-même peut contenir
  - Une seule image
  - Plusieurs images de même taille : les « frames »
  - L'animation d'un `sprite` est une séquence d'affichage de frames

- Un sprite gère également des transformations sur l'image qu'il affiche :
  - Rotation multime de 90°
  - Réflexion (mirroir)

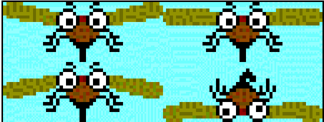
- Un `sprite` gère également les collisions, avec quatre méthodes :

```
collidesWith(Image image, int x, int y, boolean pixelLevel)

collidesWith(Sprite s, boolean pixelLevel)

collidesWith(TiledLayer t, boolean pixelLevel)

defineCollisionRectangle(int x, int y, int width, int height)
```

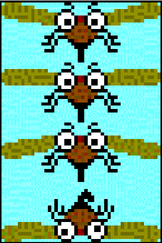
Sprite : les frames

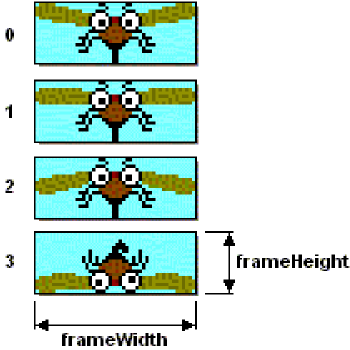


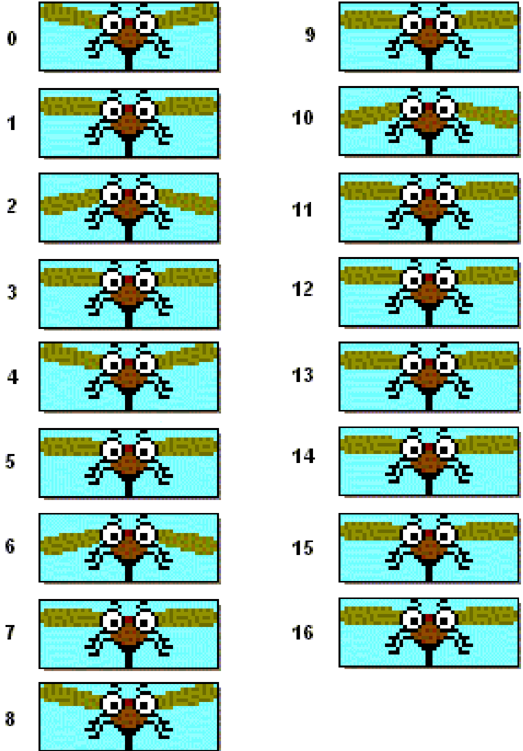
OR



OR







TiledLayer

- Sous classe de `Layer`, c'est l'élément qui permet de définir le « monde » du jeu, le plateau.

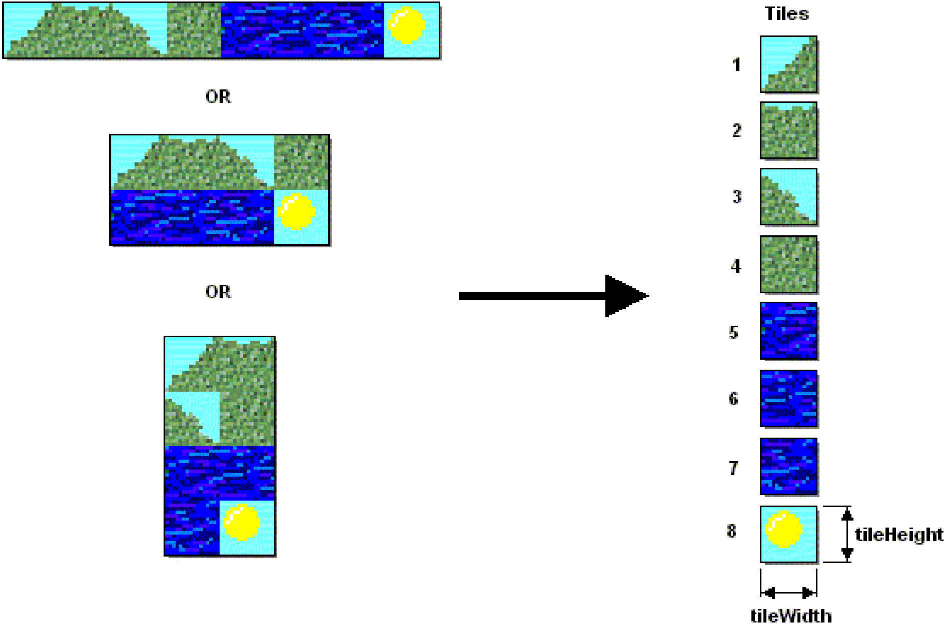
- L'intérèt du TiledLayer est de pouvoir définir un « monde » très large, avec une image très petite.

- Un TilesLayer est composé principalement de :
  - Une image. L'image est une grille de cellules, chaque cellule représente une « dalle ».
  - Une grille, représentant des cellules remplies par des dalles.
    - Les dalles sont définies par `setCell()` et `fillCells()`

- Le constructeur attend
  - Le nombre de cellules horizontales
  - Le nombre de cellules verticales
  - L'image représentant les dalles
  - La largeur des dalles dans l'image
  - La hauteur des dalles dans l'image
  - `tiles = new TiledLayer(40, 16, tilesImage, 7, 7);`

- Une cellule peut être animée

TiledLayer

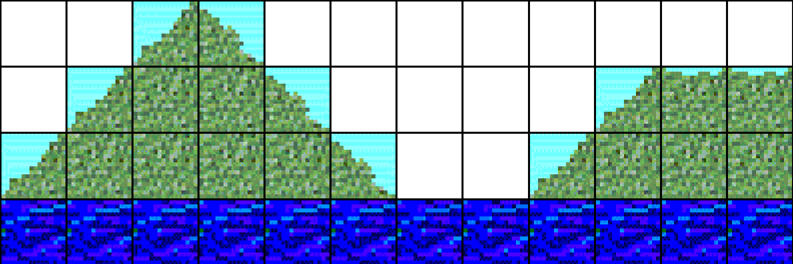


Cells

0	0	1	3	0	0	0	0	0	0	0	0
0	1	4	4	3	0	0	0	0	1	2	2
1	4	4	4	4	3	0	0	1	4	4	4
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Animated Tiles

-1	=	5



LayerManager

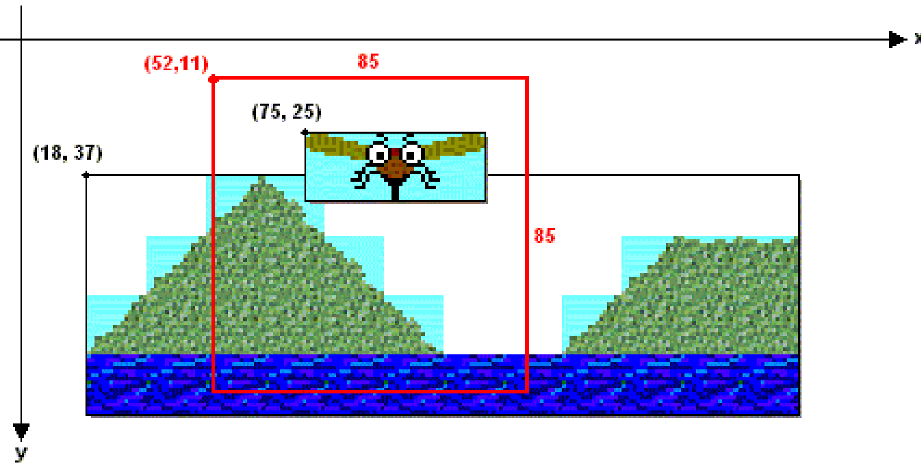
- Comme son nom l’indique, il manage des Layers (Sprites, TiledLayer).

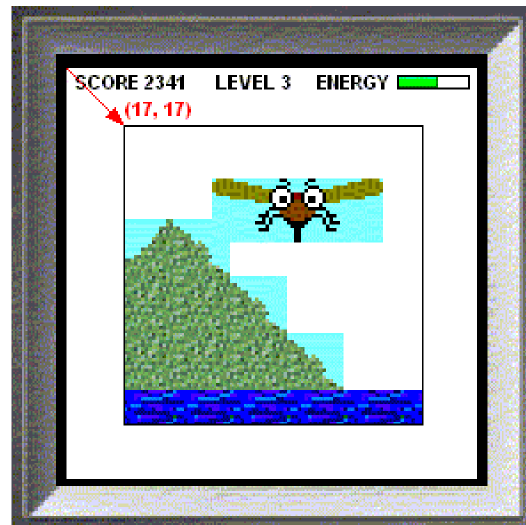
- Il maintient une liste ordonnée à laquelle les layers peuvent être ajoutés ou supprimés.
  - `append(Layer l)`
    Ajoute un layer au LayerManager
  - `insert(Layer l, int index)`
    Insert un Layer à un certain index
  - `remove(Layer l)`
    Supprime un layer

- Il gère la profondeur (quel layer doit être affiché au dessus de quel autre ?).
  - C’est l’index de chaque Layer qui donne sa profondeur
  - Layer d’index 0 : le premier affiché

- Il gère une « View Window » : partie visible à afficher dans son système de coordonnées.
  - `setViewWindow(int x, int y, int heigth, int width)`
    Défini taille et emplacement de la ViewWindow dans le `LayerManager`.
  - `paint(Graphics g, int x, int y)`
    Défini à quelles coordonnées de Graphics la ViewWindow du LayerManager doit être dessinée.

LayerManager





TP1

- Coder une MIDlet :
  - référencant le GameCanvas
  - Démarrant le GameCanvas
  - Ajoutant au GameCanvas un bouton « exit »

- Coder dans le run() de GameCanvas
  - La boucle (à l'aide du booléen mTrucking)
  - L'appel des méthodes gérant le jeu.

TP2

- Dans la méthode tick() deMicroTankCanvas
  - Détecter les collisions entre le tank et les murs du TiledLayer
  - Appeler la méthode mTank.undo() du sprite tank en cas de collision

- Dans la MIDlet MicroTankCanvas
  - Se servir du TiledLayer pour faire un mur entourant tout l'écran, avec une seule sortie.

- Si le tank réussi à sortir de l'écran par la sortie :
  - afficher au milieu de l'écran :« ESCAPED !! »
  - Afficher Le temps qu'il a fallu au tank pour s'échapper