Java et systèmes embarqués

Jean-Francois Lalande - September 2012

Le but de ce cours est de découvrir les technologies Java permettant de développer des applications embarquées, notamment sur des téléphones portables. Ce cours permet de découvrir deux technologies supportées par Oracle, Java ME et Java FX. Une troisième partie aborde le développement d'applications clients serveurs en environnement embarqué.

1 Plan du cours

Plan du cours

Licence: la plupart des exemples de codes sont des extraits de code sources fournis par Oracle, sous licence L1 (cf fin du document).

2 Java ME

2.1 Les grandes familles de technologies Java

[EO] Java est à la fois un langage de programmation et une plate-forme de développement fournissant un environnement d'exécution rendant les applications indépendantes de la plate-forme. Les plate-formes étant multiples, il existe plusieurs editions des technologies Java:

- Java Platform, Standard Edition (Java SE): un environnement pour les applications de bureaux.
- Java Platform, Enterprise Edition (Java EE): un sur-ensemble de Java SE, orienté transaction et centré sur les bases de données (besoin orienté entreprise).
- Java Platform, Micro Edition (Java ME): environnement d'exécution et API pour les systèmes embarqués (téléphone, smartphone, assistants, TV).

Java ME n'est donc pas un nouveau langage de programmation. Il reste compatible avec Java SE autant que faire ce peut. Il peut définir de nouvelles interfaces ou APIs mais il tronque surtout une grosse partie de Java SE permettant d'alléger la machine virtuelle et les applications qui s'exécuteront sur un système embarqué très contraint.

Java ME se subdivise ensuite en 3 familles pour les systèmes embarqués:

- Java Micro Edition: téléphones, pages, pdas, ...
- Java Card: carte à puce
- Java SE Embedded: une version optimisée de Java ME

2.2 Les normes de Java ME

Java ME recouvre un ensemble de normes ou spécifications qui évoluent au cours du temps [MEDOC]. Sun (Oracle) a défini une numérotation JSR XXX qui correspond à chaque type de technologie qui constitue Java ME. Par exemple, la brique "RMI" est notée JSR 66, et les éléments du package java.security dédié à l'embarqué est numéroté JSR 219.

La page de référence qui décrit les différentes API [MEDOC] est située à:

http://www.oracle.com/technetwork/java/javame/documentation/apis-jsp-137855.html

On distingue dans Java ME trois grandes familles de technologies:

- CLDC: Connected Limited Device Configuration

- CDC: Connected Device Configuration
- Optional packages: bluetooth, sécurité, web service, graphiques, etc...

Les technologies de type CLDC définissent des spécifications permettant de développer des applications pour des périphériques nomades et à ressources limitées. Les spécifications définies dans CDC s'adressent à des applications distribuées, connectées au réseau et embarquées.

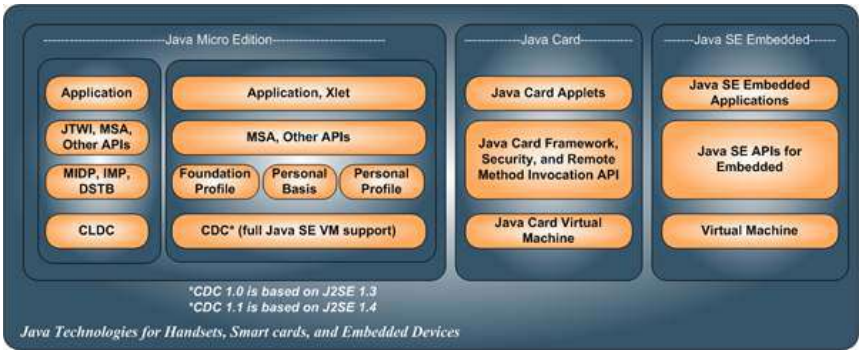
2.3 Profils et configurations

Java ME est découpé en 4 grandes parties:

- Les configurations (VM + Core Java APIs)
- Les profils (environnement de l'application)
- La partie applicative
- Les packages/APIs optionnels

Une configuration définit les possibilités de la machine virtuelle ainsi que l'API disponible pour le développeur. Une configuration est relativement indépendante du système embarqué cible. Elle définit une base commune à une famille de systèmes embarqués.

Un profil définit une API pour l'environnement de l'application c'est à dire le principe de fonctionnement et d'interaction entre le système embarqué et l'application.



Java Technologies for Handsets, Smart Cards, and Embedded Devices [EO]

2.4 Configuration CLDC

A l'heure actuelle, les configurations suivantes sont publiées par Oracle dans la catégorie CLDC:

- Connected Limited Device Configuration 1.0 JSR 30
- Connected Limited Device Configuration 1.1 JSR 139

La configuration CLDC 1.1 définit l'ensemble des classes fournies par un machine virtuelle compatible. Le nombre de package étant très limité, il est même possible de les lister exhaustivement:

```
java.io: I/O
java.lang: classes fondamentales (types, exceptions, erreurs)
java.lang.ref: weak references
java.util: collections et dates
javax.microedition.io: définition des interfaces de connection
```

Ces packages regroupent un sous ensemble restreint de classes de la spécification J2SE 1.4.

java.io

Interfaces

```
DataInput
DataOutput
```

Classes

```
ByteArrayInputStream ByteArrayOutputStream
DataInputStream DataOutputStream
InputStream InputStreamReader
OutputStream OutputStreamWriter
PrintStream
Reader Writer
```

Exceptions

```
EOFException
InterruptedIOException
IOException
UnsupportedEncodingException
UTFDataFormatException
```

java.lang

Interfaces

```
Runnable
```

Classes

```
Boolean Byte Character Class Double Float
Integer Long Math Object Runtime Short
String StringBuffer System Thread Throwable
```

Exceptions

```
Exception ArithmeticException ArrayIndexOutOfBoundsException
ArrayStoreException ClassCastException ClassNotFoundException
IllegalAccessException IllegalArgumentException
IllegalMonitorStateException IllegalThreadStateException
IndexOutOfBoundsException InstantiationException
...
```

Errors

```
Error NoClassDefFoundError OutOfMemoryError VirtualMachineError
```

java.lang.ref

Classes

```
Reference
WeakReference
```

Apparté sur les weak références

Il s'agit de permettre au programeur de créer un pointeur sur un objet qui n'empêchera pas le *garbage collector* de collecter l'objet pointé par une *weak reference*. On construit un objet de la classe **WeakReference** en passant au constructeur l'objet pointé. Il y a donc une sorte d'indirection puisque le programmeur possède un pointeur sur la *weak reference* qui elle même pointe vers l'objet. Cependant, l'implémentation des *weak references* signale à la machine virtuelle que l'objet est pointé par une référence dite "faible" et qui peut donc laisser le *garbage collector* agir.

```
public WeakReference(Object ref)
public Object get(); // Returns this reference object's referent
public void clear(); // Clears this reference object.
```

java.util

Interfaces

```
Enumeration
```

Classes

```
Calendar
Date
Hashtable
Random
Stack
TimeZone
Vector
```

Exceptions

```
EmptyStackException
NoSuchElementException
```

javax.microedition.io

Interfaces

```
Connection
ContentConnection
Datagram
DatagramConnection
InputConnection
```

```
OutputConnection
StreamConnection
StreamConnectionNotifier
```

Classes

```
Connector
```

Exceptions

```
ConnectionNotFoundException
```

Explications: cf javax.microedition.io

2.5 Profils CLDC

Le profil Mobile Information Device Profile (2.0, JSR 118) permet de définir comment l'application intéragit avec le système embarqué. Il s'agit par exemple de définir le cycle de vie de l'application, son interaction avec l'interface graphique, les échanges des données à faire persister avec le système embarqué. On trouve alors des classes spécifiques en plus des classes fournies par la configuration.

Dans CLDC, on trouve 3 profils différents:

- Mobile Information Device Profile (MIDP), versions 1.0, 2.0, and the upcoming 3.0 (le plus utilisé)
- Information Module Profile (IMP), versions 1.0 and 2.0 (périphériques sans écran)
- Digital Set Top Box Profile (défini pour le cable)

Pour supporter le profil MID, un système embarqué doit disposer d'un écran de 96x54 pixels, un clavier, un *keypad*, ou un écran tactile. La version sans écran et sans entrées de MID est IMP (Informatio Module Profile), ce qui se traduit par la disparition du package **javax.microedition.lcdui** [IMP].

Profil MID 2.0

Les packages définis dans MID, en plus de ceux définis par la configuration CLDC sont:

```
java.lang.util: +Random +Timer +TimerTask
javax.microedition.io: +réseau: sockets, udp, série, push
javax.microedition.lcdui: interface graphique
javax.microedition.lcdui.game: jeux (sprites, layers, canvas)
javax.microedition.media: audio (control, player)
javax.microedition.media.control: contrôle du son
javax.microedition.midlet: interface avec l'application, cycle de vie
javax.microedition.pki: certificats, authentification
javax.microedition.rms: gestion des données persistantes
```

2.6 Réseau (javax.microedition.io)

Toutes les connections réseaux se basent sur l'interface **Connection** qui possède un certain nombre de sous interfaces:

- HttpConnection, HttpsConnection

- SocketConnection
- UDPDatagramConnection, CommConnection (port série), ...

On utilise ces interfaces au travers de la factory **Connector**. Le rôle de Connector est d'interpréter l'url l'ors de l'appel à **Connector.open(url)**. Dans le cas d'une url commencant par "http://", l'appel retourne une classe implémentant **HttpConnection**. Le format BNF de l'url suit les règles suivantes:

```
<socket_connection_string>    ::= "ssl://"<hostport>
<hostport>                    ::= host ":" port
<host>                        ::= host name or IP address
<port>                        ::= numeric port number
```

Les classes gérant l'accès au réseau masquent aussi les particularitées dues au type du réseau (filaire, sans fil). L'utilisation d'un point d'accès ou une gateway ne doit pas transparaitre au travers de l'utilisation des classes. L'élégance de Java au niveau des interfaces de manipulation du réseau est d'être capable d'implémenter plusieurs super-interfaces différentes. Autrement dit, une classe réseau fait de l'héritage multiple de plusieurs comportements, par exemple en tant que flux (**StreamConnection**) ou contenu (**ContentConnection**). Ainsi, une connection **HttpConnection** peut-être lue de 3 manières différentes.

HttpConnection comme StreamConnection

Dans cet exemple, la réponse est lue comme un flux. L'exemple ignore le code de réponse HTTP et considère le flux entrant comme valide.

```
void getViaStreamConnection(String url) throws IOException {
    StreamConnection c = null;
    InputStream s = null;
    try {
        c = (StreamConnection)Connector.open(url);
        s = c.openInputStream();
        int ch;
        while ((ch = s.read()) != -1) {
            ...
        }
    } finally {
        if (s != null)
            s.close();
        if (c != null)
            c.close();
    }
}
```

HttpConnection comme ContentConnection

En tant que contenu, les éléments reçus ont une longueur. Une fois la longueur récupérée, les données sont lus en une seule fois.

```
void getViaContentConnection(String url) throws IOException {
      ContentConnection c = null;
      DataInputStream dis = null;
      try {
```

```
            c = (ContentConnection)Connector.open(url);
            int len = (int)c.getLength();
            dis = c.openDataInputStream();
            if (len > 0) {
                byte[] data = new byte[len];
                dis.readFully(data);
            } else {
                int ch;
                while ((ch = dis.read()) != -1) {
                    ...
                }
            }
        } finally {
            if (dis != null) dis.close();
            if (c != null) c.close();
}}
```

HttpConnection comme HttpConnection

```
void getViaHttpConnection(String url) throws IOException {
     HttpConnection c = null; InputStream is = null;
     int rc;
     try { c = (HttpConnection)Connector.open(url);
           rc = c.getResponseCode();
           if (rc != HttpConnection.HTTP_OK) {
               throw new IOException("HTTP response code: " + rc);
           }
           is = c.openInputStream();
           // Get the ContentType
           String type = c.getType();
           // Get the length and process the data
           int len = (int)c.getLength();
           if (len > 0) {
               int actual = 0; int bytesread = 0 ;
               byte[] data = new byte[len];
               while ((bytesread != len) && (actual != -1)) {
                  actual = is.read(data, bytesread, len - bytesread);
                  bytesread += actual;
               } else {
               int ch;
               while ((ch = is.read()) != -1) {
                   ...
```

SocketConnection

Une socket est une version très simplifiée de la classe **Socket** de Java SE.

```
SocketConnection sc = (SocketConnection)
                      Connector.open("socket://host.com:79");
sc.setSocketOption(SocketConnection.LINGER, 5);
```

```
InputStream is  = sc.openInputStream();
OutputStream os = sc.openOutputStream();

os.write("\r\n".getBytes());
int ch = 0;
while(ch != -1) {
    ch = is.read();
}

is.close();
os.close();
sc.close();
```

Connexion SSL

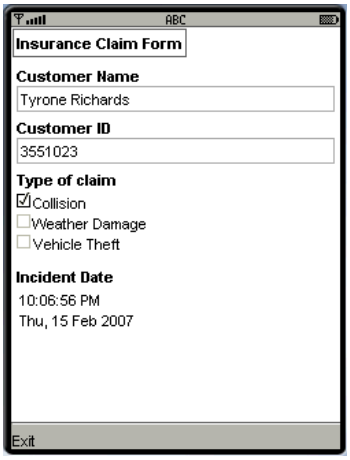
Enfin, un exemple d'utlisation de **SecureConnection** pour une connexion SSL.

```
SecureConnection sc = (SecureConnection)
                      Connector.open("ssl://host.com:79");
SecurityInfo info = sc.getSecurityInfo();
boolean isTLS = (info.getProtocolName().equals("TLS"));

sc.setSocketOption(SocketConnection.LINGER, 5);

InputStream is  = sc.openInputStream();
OutputStream os = sc.openOutputStream();

os.write("\r\n".getBytes());
int ch = 0;
while(ch != -1) {
    ch = is.read();
}

is.close();
os.close();
sc.close();
```

2.7 Interface utilisateur dans MIDP

[GUI] Le package **javax.microedition.lcdui** permet de réaliser des interfaces graphiques spécifiques aux systèmes embarqués. Les classes de ce package permettent d'utiliser une API de bas ou de haut niveau. L'API de haut niveau permet de manipuler simplement les éléments de l'interface graphique à moindre effort, alors que l'API de bas niveau permet de dessiner et manipuler finement les objets graphiques.

Dans l'API de haut niveau les composants de l'interface se manipulent au travers des sous classes de **Screen** et **Item**.



A simple application using the MIDP high-level APIs shown in the Java ME Wireless Toolkit's Emulator [GUI]

API graphique de haut niveau

Les composants graphiques de haut niveau disponibles sont les suivants:

- **Alert** (used for showing dialog boxes and warnings)
- **ChoiceGroup** (used for showing group of checkboxes or radio buttons)
- **DateField** (used for selecting dates and times)
- **Form** (used for displaying the subclasses of Item)
- **Gauge** (used to showing the progress of an operation or process, but it can also be used to represent the percent of a whole)
- **ImageItem** (used for showing images in high-level UI applications)
- **List** (used for showing lists and combo boxes)
- **StringItem** (used for showing simple text strings)
- **TextBox** (used for entering text)
- **TextField** (an alternate method used for entering text)

Par exemple:

```
Display display = Display.getDisplay(this);
Form f = new Form("StringItem");
f.append(new StringItem("Label1", "Text1"));
f.append(new StringItem("Label2", "Text."));
display.setCurrent(f);
```

API graphique de haut niveau: Display

Dans l'API graphique de haut niveau, la classe **Display** représente l'écran du téléphone mobile. On peut récupérer le **Display** à l'aide de **getCurrent()** et inversement, remplacer ce qui est affiché à l'aide de **setCurrent(Displayable nextDisplayable)**. La classe abstraite **Displayable** possède en fait deux sous classes l'implémentant: **Canvas** et **Screen**.

Pour construire un écran, on utilise donc un objet dérivant de l'une de ses deux classes. Par exemple, pour un menu, on peut utiliser un objet **L** de type **List** (**javax.microedition.lcdui.List** héritant de **Screen**, lui même héritant de **Displayable**). On appelle alors:

```
Display display = Display.getDisplay(this);
List L = new List(...);
display.setCurrent(L);
```

Ainsi, l'objet **Display** passé en paramètre se dessine dans l'écran.

API graphique de bas niveau

L'API de bas niveau s'appuie sur **Canvas** et **Graphics**. Elles sont largement utilisées pour les jeux et doivent être évitées au profit de l'API de haut niveau si possible. La classe **Canvas** permet de controler l'espace de rendu ainsi que les interactions utilisateurs. La classe **Graphics** permet de réaliser les dessins proprement dits.

Quelques exemples de méthodes et appels:

```
// Dans la classe Canvas:
public final void repaint();
protected  void keyPressed(int keyCode);
if (keyCode > 0)
  char ch = (char)keyCode;

// Dans la classe Graphics
public void drawString(String text, int x, int y, int anchor);
void drawArc(int x, int y, int width, int height,
             int startAngle, int arcAngle)
g.drawString(str, x, y, TOP|LEFT);
g.fillRect(x, y, w, h); // 1
g.drawRect(x, y, w, h); // 2
```

Les menus

[EVENT] La génération des menus se fait en utilisant la classe **Command**. On utilise un label, un type (BACK, CANCEL, EXIT, HELP, ITEM, OK, SCREEN, STOP) et une priorité représentant l'importance de la commande par rapport aux autres. On peut par exemple créer les commandes "Info" et "Buy" ainsi:

```
Command infoCommand = new Command("Info", Command.SCREEN, 2);
Command buyCommand = new Command("Buy", Command.SCREEN, 2);
```

La priorité 1 est très utile pour placer la commande en évidence sur l'interface (en bas à gauche ou à droite).

Pour réaliser le menu, on ajoute les commandes aux objets graphiques qui peuvent les supporter, par exemple **Form**

```
Form form = new Form("Your Details");
...
form.addCommand(new Command("EXIT", Command.EXIT, 2));
form.addCommand(new Command("HELP", Command.HELP, 2));
```

```
form.addCommand(new Command("OK", Command.OK, 1));
```

2.8 Persistance des données

[RMS] La persistence des données (javax.microedition.rms) utilise la classe **RecordStore** qui est une collection d'enregistrements qui resteront persistants entre plusieurs invocations de l'application. L'implémentation des structures de données sous jacentes est dépendant de la plate-forme et évite les conflits de *namespace* entre les applications. Le nom du record store utilise le nom de l'application concaténé avec le nom du **RecordStore**.

Un **RecordStore** assure que les opérations d'enregistrements sont atomiques, ce qui est particulièrement utile si l'application est multi-threadée. Les écritures dans le même **RecordStore** est possible et est aussi garanti atomique. Une numérotation automatique et un *timestamp* de chaque enregistrement permet de suivre l'évolution de la donnée, ce qui est particulièrement utile pour synchroniser des données.

Un record est un tableau de *byte* indexé par un entier unique.

```
static RecordStore openRecordStore(String recordStoreName,
                                   boolean createIfNecessary);
int addRecord(byte[] data, int offset, int numBytes);
void deleteRecord(int recordId);
byte[] getRecord(int recordId);
```

Exemple d'utilisation du RecordStore

Exemple issu de [RMS] pour le stockage des scores d'un jeu:

```
try {
    recordStore = RecordStore.openRecordStore("scores", true);
} catch ...
int recId;  // returned by addRecord but not used
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream outputStream = new DataOutputStream(baos);
try {
    // Push the score into a byte array.
    outputStream.writeInt(score);
    // Then push the player name.
    outputStream.writeUTF(playerName);
 } catch ...
 // Extract the byte array
 byte[] b = baos.toByteArray();
 try {
 recId = recordStore.addRecord(b, 0, b.length);
 } catch ...
```

Contrôle d'accès

Deux modes d'accès sont prévus pour le **RecordStore**. Un mode privé ou seul l'application accède au **RecordStore** et un mode publique ou toute application peu y accéder. Le contrôle s'effectue à l'ouverture:

```
public static RecordStore openRecordStore(String recordStoreName,
                                          boolean createIfNecessary,
                                          int authmode,
                                          boolean writable)
```

AUTHMODE_PRIVATE

L'ouverture dans ce mode réussit s'il s'agit de l'application qui a créé le RecordStore (revient à *openRecordStore(recordStoreName, createIfNecessary)*)

AUTHMODE_ANY

Toutes les application peuvent accéder à ce **RecordStore**

Le mode d'accès peut éventuellement changer ultérieurement:

```
public void setMode(int authmode, boolean writable)
```

2.9 Modèle de développement J2ME

[EG] Le modèle de développement classique (J2SE) utilise un *main()* qui définit le point d'entrée du programme et sa sortie. Plus rarement, une application J2SE qui veut se comporter comme un démon ne prévoit pas de méthode de sortie, comme dans l'exemple suivant:

```
public class NeverEnding extends Thread {
  public void run(){
      while( true ){
          try {
              sleep( 1000 );
          }
          catch( InterruptedException e ){
          }
  } }
  public static void main( String[] args ){
      NeverEnding ne = new NeverEnding();
      ne.start();
} }
```

Une telle application ne peut être que tuée par l'OS. Elle n'aura pas l'occasion d'effectuer des opérations de nettoyage ou de sauvegarde lorsqu'elle se termine, comme le fait une application J2SE classique.

Le modèle de développement des applets

Une applet s'exécute dans un navigateur web, incrusté dans une page. Le programme qui s'exécute dans une JVM standard mais à ressource limitées subit les contraintes de la navigation: interruption (destruction) lors d'un changement de page, relancement si l'utilisation revient sur la page. On distingue 4 états dans le cycle de vie de l'applet:

- chargée: l'instance de l'applet est construite, mais pas initialisée;
- stoppée: l'applet a été initialisée mais est inactive;
- démarrée: l'applet est active;
- détruite: l'applet est terminée et peut être collectée.

Une fois l'applet chargée, le constructeur a été exécuté (il ne doit pas contenir trop de code). Le navigateur attend alors que le contexte d'exécution (resources du navigateur, image, status bar) soit prêt avant de basculer l'applet en "stoppée" et d'appeler la méthode *init()*. Puis, le navigateur peut basculer l'applet de "stoppée" à "démarrée". Son activation provoque l'appel à *start()* et sa désactivation l'appel à *stop()*. Un appel à *stop()* peut permettre à l'applet de relacher les ressources acquises. Le navigateur peut même décider de détruire complètement l'applet et invoque dans ce cas *destroy()*.

Le grand intérêt de ce mécanisme est que le navigateur peut maintenir une applet "stoppée" même si l'utilisateur change de page pour y revenir plus tard. Inversement, le navigateur peut stopper une applet d'une page visible s'il le souhaite.

Le modèle de développement des applets (2)

Pour mémoire, le squelette d'une applet est de la forme:

```
public class BasicApplet extends java.applet.Applet {
  public BasicApplet(){
      // constructor - don't do much here
  }
  public void init(){
      // applet context is ready, perform all one-time
      // initialization here
  }
  public void start(){
      // the applet is being displayed
  }
  public void stop(){
      // the applet is being hidden
  }
  public void destroy(){
      // free up all resources
  }
}
```

```
<applet code="BasicApplet.class"
      width="300" height="200">
</applet>
```

Le modèle de développement des MIDlets

Le cycle de vie d'une MIDlet s'inspire de celle d'une applet mais est un peu différente:

- pause: la MIDlet est construite mais inactive;
- active: la MIDlet est active;
- détruite: la MIDlet est terminée et peut être collectée.

Dans ce modèle, il n'y a plus d'état "chargée" car il n'y a pas de méthode d'initialisation *init()*. L'initialisation se fait lorsque *startApp()* est invoquée.

```
import javax.microedition.midlet.*;
public class BasicMIDlet extends MIDlet {
  public BasicMIDlet(){
```

```
        // constructor - don't do much here
    }
    protected void destroyApp( boolean unconditional )
              throws MIDletStateChangeException {
        // called when the system destroys the MIDlet
    }
    protected void pauseApp(){
        // called when the system pauses the MIDlet
    }
    protected void startApp() throws MIDletStateChangeException {
        // called when the system activates the MIDlet
}}
```

Le modèle de développement des MIDlets (2)

Comme dans les applet, on évite de réaliser les initialisations dans le constructeur, car le contexte d'exécution de la MIDlet n'est pas forcément encore disponible. La MIDlet est dans l'état initial "pause", puis est basculé par l'AMS (Application Management Software, un autre processus du système embarqué) dans l'état "actif" avec l'appel à *startApp()*.

Si la MIDlet ne peut s'initialiser, elle lève une exception de type **MIDletStateChangeException** ce qui la fait basculer dans l'état "détruite".

Enfin, la MIDlet peut basculer de l'état "actif" à "pause" par l'AMS. Comme l'applet, la MIDlet doit essayer de relacher le maximum de ressources lorsque l'AMS invoque la méthode *pauseApp()*.

Si la MIDlet se met en pause elle même (resp. se détruit), la méthode *pauseApp()* (resp. *destroyApp()*) n'est pas appellée. Pour cela la MIDlet utilise la méthode *notifyPause()* (resp. *notifyDestroyed()*) pour informer l'AMS.

Concernant le contexte d'exécution de la MIDlet, la méthode *getAppProperty(String key)* permet de récupérer une propriétée nommée au travers de l'AMS située dans le *manifest* de l'application.

A l'inverse, la méthode *platformRequest(String URL)* permet de demander au système embarqué de gérer une URL dans une application externe. L'appel est non non bloquant et laissse la MIDlet continuer à s'exécuter en tâche de fond.

2.10 MIDLets event Handling [EVENT]

Les MIDlet sont notifiés d'événements utilisateurs ou systèmes au travers de callbacks. Les événements utilisateurs sont sérialisés par le système, ce qui fait que les appels aux callbacks ne peuvent pas être parallèles. Le prochain input est donc appelé lorsque la callback termine son traitement.

On distingue deux types d'événements qui fonctionnent au travers de callbacks:

- les événements de haut niveau (reliés à **Screen** et **Canvas**, par exemple la sélection d'un menu)
- les événements de bas niveau (reliés à **Canvas**, par exemple un appui de touche)

Evenements de haut niveau

Classiquement en java, les événements utilisent des **Listener**.

Pour les menus de commandes, on utilise l'interface **CommandListener**. On appelle la méthode **addCommand()** pour enregistrer la commande dans l'objet graphique de référence puis on ajoute un listener qui va permettre d'appeler la callback de réponse à l'événement.

```
Canvas canvas = new Canvas() {

exit = new Command("Exit", Command.STOP, 1);
canvas.addCommand(exit);
   canvas.setCommandListener(new CommandListener() {
     public void commandAction(Command c, Displayable d) {
       if(c == exit) {
         notifyDestroyed();
       } else {
         System.out.println("Saw the command: "+c);
       }
     }
   });
```

Evenements de bas niveau

En surchargeant la classe **Canvas** il est possible de modifier le comportement par défaut des méthodes gérant les événements bas niveau (par exemple **keyPressed**).

```
public void startApp () {
    display = Display.getDisplay(this);

    // anonymous class
    Canvas canvas = new Canvas() {

      public void paint(Graphics g) { }

      protected void keyPressed(int keyCode) {
        if (keyCode > 0) {
          System.out.println("keyPressed "
             + ((char)keyCode));
        } else {
          System.out.println("keyPressed action "
             + getGameAction(keyCode));
        }
      }
    }
}
```

Exemple: gestion des menus

Ces mécanismes sont particulièrement utiles pour la gestion des menus:

```
public class Hello extends MIDlet implements CommandListener {
private Display display;
private List mainMenu;

protected void startApp() throws MIDletStateChangeException {

  display = Display.getDisplay(this);
```

```
  String[] items = {"Gauge", "Appel téléphonique !", "Sortir"};
  mainMenu = new List("Main", Choice.IMPLICIT, items, null);

  // Je suis mon propre écouteur
  mainMenu.setCommandListener(this);

  // J'affiche
  display.setCurrent(mainMenu);
}

public void commandAction(Command c, Displayable d) {
  System.out.println("" + mainMenu.getSelectedIndex());
  if (mainMenu.getSelectedIndex() == 0)
    display.setCurrent(form);
  if (mainMenu.getSelectedIndex() == 1)
    this.notifyPaused();
  if (mainMenu.getSelectedIndex() == 2)
    this.notifyDestroyed();
}}
```

Démonstration: menus, gauge, formulaires

3 Java FX

3.1 Langage de programmation de script

[FXT] Java FX 1 définit un nouveau langage de script, sensé être plus simple et plus efficace pour le développement graphique. Il sera abandonné dans Java FX 2.

De manière classique, les scripts peuvent définir des variables et des fonctions. Les variables sont définies par les mots clef *var* ou *def* et les fonctions par *function*. Les typages sont omis car le compilateur fait de l'inférence de type.

```
def numOne = 100;
def numTwo = 2;
var diff;

add(); // invocation
diff = substract(50,5); // invocation avec arguments

function add() {
result = numOne + numTwo;
println("{numOne} + {numTwo} = {result}");
}

function subtract(argOne: Integer, argTwo: Integer) {
  result = argOne - argTwo;
  println("{argOne} - {argTwo} = {result}");
  return result;
}
```

Main

Le point d'entrée principal est la méthode *run*:

```
var result;

function run(args : String[]) {

  // Convert Strings to Integers
  def numOne = java.lang.Integer.parseInt(args[0]);
  def numTwo = java.lang.Integer.parseInt(args[1]);

  // Invoke Functions
  add(numOne,numTwo);
```

```
}


function add(argOne: Integer, argTwo: Integer) {
  result = argOne + argTwo;
  println("{argOne} + {argTwo} = {result}");
}
```

qui, si le fichier s'appelle calculator.fx, s'exécute par un appel à:

```
javafx calculator 100 50
```

Les objets

Le langage de script est orienté objet et tente de "masquer" la complexité de la notion d'instance d'objets. Par exemple, le listing suivant instancie un objet de la classe **Adress** et initialise ses attributs:

```
 Address {
   street: "1 Main Street";
   city: "Santa Clara";
   state: "CA";
   zip: "95050";
}
```

La classe correspondante se situe dans le fichier *Address.fx*:

```
// LICENCE: cf [L1]
public class Address {
  public var street: String;
  public var city: String;
  public var state: String;
  public var zip: String;
}
```

Les objets: variables et fonctions

De manière originale, il est possible de créer des variables et d'appeler les méthodes d'objets. Les objets agrégés, peuvent s'instancier en encapsulant la définition de l'instance dans l'instance encapsulante.

```
var customer = Customer {
   firstName: "John";
   lastName: "Doe";
   address: Address {
        street: "1 Main Street";
        city: "Santa Clara";
        state: "CA";
        zip: "95050";
}}
customer.printName();
```

```
public class Customer {
   public var firstName: String;
   public var lastName: String;
   public var address: Address;
  public function printName() {
      println("Name: {firstName} {lastName}");
  }
```

Les types: string

Les chaines de caractères se définissent de facon classique. Attention à l'opérateur {} qui permet d'insérer inline du code scripté dans une chaine.

```
def name = 'Joe';
var s = "Hello {name}"; // s = 'Hello Joe'
def answer = true;
var s = "The answer is {if (answer) "Yes" else "No"}";
def one = "This example ";
def two = "joins two strings.";
def three = "{one}{two}";    // join string one and string two
println(three);              // 'This example joins two strings.'
```

Les types: nombres et booléens

Le type d'un nombre est déduit de la déclaration, si aucun type n'est spécifié. Cependant, le type peut être explicitement précisé. On retrouve les types non primitifs de Java ainsi que le type **boolean**.

```
def numOne = 1.0; // compiler will infer Number
def numTwo = 1;   // compiler will infer Integer
def numOne2 : Number = 1.0;
def numTwo2 : Integer = 1;
var isAsleep = true;
if (isAsleep) {
   wakeUp();
}
```

Typage des fonctions

Pour les fonctions sans retour, le type vide est **Void**. Le pointeur **null** est toujours d'actualité:

```
function printMe(arg1: Address) : Void {
   if (arg1 != null)
          println("I don't return anything!");
}
```

Une fonction est aussi une variable du programme. Elle peut être définie à la déclaration de la variable ou ultérieurement.

```
var myFunc : function(:Object,:Integer):String;
myFunc = function(obj : Object, k : Integer)
```

```
        { "Here is the Object: {obj}, and the Integer: {k}" }
println(myFunc(null, 1234));
// Prints: Here is the Object: null, and the Integer: 1234
```

Séquences

Une séquence est en fait une liste d'item, déclarée entre crochets. Les séquences de séquences sont autorisées. Pour les séquences arithmétiques, des intervalles peuvent définir une énumération. Des énumérations peuvent être construite à partir de prédicats. L'opérateur **..** permet de *slicer* une séquence.

```
def weekDays = ["Lundi", "Mardi", "Samedi", "Dimanche"];
def days = [weekDays, "Sat","Sun"];
def samedidimanche = days[2..3];
def nums = [1..100];
def numsGreaterThanTwo = nums[n | n > 2];
var nums = [1..10 step 2];

var weekDays = ["Lundi","Mardi","Mercredi","Jeudi", "Vendredi"];
var days = [weekDays, ["Samedi","Dimanche"]];
def tmp = days[3..4];
println(tmp); //retourne : [ Jeudi, Vendredi ]
```

On accède à un élément d'une séquence à l'aide de l'opérateur [], comme un tableau. L'opérateur **sizeof** permet de récupérer la taille d'une séquence. Le mot clef **insert** permet de pousser un élément dans la séquence. **reverse** inverse une séquence.

```
insert "Tue" into days;
insert "Thu" before days[2];
insert "wed" after days[1];
nums = reverse nums;
```

Une séquence est donc un mix entre un tableau, un vecteur et une liste...

if, for, while, exceptions

Rien de transcendant pour ces structures de contrôle. Le **for** peut employer une séquence (c'est logique), comme par exemple:

```
var days = ["Mon","Tue","Wed","Thu","Fri","Sat","Sun"];

for (day in days) {
   println(day);
}
```

Les exceptions sont très similaires à Java:

```
try {
   foo();
} catch (e: Exception) {
   println("{e.getMessage()} (but we caught it)");
```

```
} finally {
   println("We are now in the finally expression...");
}
```

3.2 Data binding

[FXDOC] Le terme "bind" désigne le fait de relier une variable à une expression qui, lorsque elle change, met à jour la variable. L'expression peut-être une autre variable ou une expression complexe (calcul, ensemble, fonction, ...). On déclare un *bind* ainsi:

```
def x = bind someExpression;
// Par exemple:
var a = 0;
def x = bind a;
a = 1; // met à jour x
```

Par exemple, la variable sum du script ci-dessous montre un bind entre sum et une expression constituée d'une variable sommée à une fonction:

```
var y = 3;
function ten() : Integer {
  println("Called");
  10
}
def sum = bind ten() + y;
println(sum);
y = 7;
println(sum); // Output: Called 13 17
```

Bind conditionnels et de séquences

Une expression conditionnelle peut-être bindée en utilisant une construction **if (...) expr1 else expr2**. A chaque fois que la condition change, l'expression qui est sélectionnée est recalculée et bindée à la variable. La valeur de l'expression non selectionnée n'est pas stockée pour une potentielle utilisation extérieure.

```
def x = bind if (condExpr) expr1 else expr2;
```

On peut définir une séquence à l'aide l'opérateur *for* dont le résultat est bindé dans une variable (une séquence). La syntaxe est la suivante:

```
def newSeq = bind for (elem in seq) expr;
```

Comme seq définit la séquence que parcourt elem, une mise à jour de seq provoque le recalcul des éléments ajoutés ou enlevés à seq et la mise à jour de newSeq. Il faut noter que newSeq n'est pas entièrement recalculé mais seulement mis à jour.

```
var max = 3;
function square(x : Integer) : Integer { x*x }
```

```
def values = bind for (x in [0..max]) square(x);  1
println(values); // Output: [ 0, 1, 4, 9 ]
max = 5; // Output: [ 0, 1, 4, 9, 16, 25 ]
```

Binding de fonctions

Il faut être vigilent avec le binding de fonctions qui ne lie que les paramètres d'appels au bind. Autrement dit, la fonction est une boite noire pour le bind, sauf si la fonction est déclarée *bound*.

```
class Point {
  var x : Number;
  var y : Number;}

var scale = 1.0;
function makePoint(x0 : Number, y0 : Number) : Point {
  Point {
    x: x0 * scale
    y: y0 * scale
}}

var myX = 3.0; var myY = 3.0;
def pt = bind makePoint(myX, myY);
println(pt.x); // Output: 3.0
myX = 10.0; println(pt.x); // Output: 10.0
scale = 2.0; println(pt.x); // Output: 10.0 (20.0 si bound)

// Mise à jour de pt.x lorsque scale = 2.0 si makePoint est:
bound function makePoint(x0 : Number, y0 : Number) : Point
{ ...
```

Binding d'objets

Lorsqu'un objet est bindée sur une variable, le changement d'un attribut de l'objet provoque la mise à jour de la variable. Comme pour un bind d'expression qui est recalculé, l'objet est réinstancié.

```
def pt = bind Point {
 x: myX
 y: myY
}
```

Pour éviter une réinstanciation et provoquer une mise à jour de la variable pt, il faut binder les attributs de l'instance de classe:

```
def pt = bind Point {
 x: bind myX
 y: bind myY
}
```

Exemple d'utilisation du Data binding

Dans cet exemple, on affiche une image élément d'une séquence d'image. La séquence est constituée d'images 7 segments. Chaque click de souris doit permettre de passer au chiffre suivant.

```
var count = 0;
def images = for(i in [0..9]){Image {url: "{__DIR__}{i}.png"};}
var currImg = images[count];

Stage {
    scene: Scene {
        content: ImageView {
            image: bind currImg
            onMouseClicked:
                function(e: MouseEvent) {
                    println("Click number {++count} ...");
                    currImg = images[count];
} } } }
```

Binding bidirectionnel

Il est même possible de binder une variable sur une autre et de garantir la mise à jourde l'une par rapport à l'autre et inversement ! On déclare cela de la façon suivante:

```
var z = bind x with inverse; // Et non pas:
var maVariable = bind autreVariable;
var autreVariable = bind maVariable; // qui retourne :
/home/jf/workspaceJavaME/JavaFX Project/src/gstutorial/Main.fx:22: This
is the cyclic reference to maVariable that prevents type inference.
```

Cela a peut d'intérêt pour les variables locales car cela revient à faire une variable et un alias. Cela devient plus utile pour maintenir des attributs de deux objets cohérents:

```
class Point {
  var x : Number;
  var y : Number;
  override function toString() : String { "Point({x},{y})" }
}
def pt = Point {
   x: 3.0
   y: 5.0
}
def projectionOnX = Point {
   x: bind pt.x with inverse
   y: 0.0
}
```

3.3 Héritage et héritage mixin

Dans Java FX, l'héritage fonctionne de manière similaire à Java. Une classe peut hériter d'une autre classe, éventuellement abstraite. En fait, une classe Java FX peut même hériter d'une classe Java. La redéfinition d'une fonction se fait en utilisant le mot clef *override*:

```
abstract class MaClasse { ... }
class SousClasse extends Account {
...
   override function truc() ...
}
```

Une sorte d'héritage multiple appelé héritage *mixin* permet aux classes notées *mixin* de servir de "pères" à une classe unique.

```
mixin class Classe1  { ... }
mixin class Classe2  { ... }
class SousClasse12 extends Classe1, Classe2 { }
```

Il faut noter qu'une classe (mixin ou non) peut étendre plusieurs interfaces Java (on n'utilise pas le mot clef *implements*).

3.4 Access modifiers

Les *access modifiers* permettent de définir la portée/visibilité en lecture/écriture des variables ou attributs de classe. Par défaut, une variable est *script-only* lorsqu'on ne précise aucun modifier, ce qui signifie que la variable est visible dans le code du fichier courant.

Cependant, comme dans un modèle objet, il peut être nécessaire d'augmenter la visibilité d'une variable pour d'autres scripts Java FX. On distingue alors plusieurs modificateurs d'accès (*access modifiers*):

- package
- protected
- public-read
- public-init

Access modifier package

La visibilité des variable est restreinte/étendue au package.

```
// Inside file tutorial/one.fx
package tutorial;
public def someMessage = "This is a public script variable, in one.fx";
public class one {
     public var message = "Hello from class one!";
     public function printMessage() {
          println("{message} (in function printMessage)");
     }
}

// Inside file two.fx
import tutorial.one;
println(one.someMessage);
```

```
var o = one{};
println(o.message);
o.printMessage();
```

Access modifier protected

L'accès *protected* étend les droits *package* aux sous-classes hors package.

```
// Inside file tutorial/one.fx
package tutorial;
public class one {
     protected var message = "Hello!";
}

// Inside file two.fx
import tutorial.one;
class two extends one {
     function printMessage() {
          println("Class two says {message}");
     }
};

var t = two{};
t.printMessage();
```

Access modifier public-read

La visibilité *public-read* d'une variable est totale mais seulement modifiable à l'intérieur du script ou elle est définie.

```
// Inside file tutorial/one.fx
package tutorial;
public-read var x = 1;

// Inside tutorial/two.fx
package tutorial;
println(one.x);
one.x = 2; // interdit
```

Une modification hors du script provoque une erreur à la compilation:

```
tutorial/two.fx:3: x has script only (default) write access in tutorial.one
one.x = 2;
 ^
1 error
```

Une solution pour résoudre ce problème est de combiner public-read avec package:

```
package public-read var x = 1;
```

Access modifier public-init

Cet *access modifier* permet de laisser le droit publique d'initialiser la variable. Les modifications ultérieures ne sont alors possibles que pour le script ayant déclaré la variable, comme avec **public-read**.

```
// Inside file tutorial/one.fx
 package tutorial;
 public class one {
      public-init var message;
 }

 // Inside file two.fx
 import tutorial.one;
 var o = one {
      message: "Initialized this variable from a different package!"
 }
 println(o.message);
```

3.5 GUI avec Java FX 1

[FXGUI] Pour créer une interface graphique, on déclare un objet **Scene** dans un objet **Stage**. L'objet **Stage** est le conteneur de plus haut niveau c'est à dire en général la fenêtre. L'attribut de titre change le titre de la fenêtre.

```
Stage {
    title: "Welcome to JavaFX!"
    scene: Scene {
        content: Text {
            x: 25
            y: 25
            content: "Hello World!"
            font: Font{ size: 32 }
} } }
```

L'attribut *content* de la scène peut contenir un ou plusieurs objets. Il faut dans ce cas utiliser une séquence d'objet qui se disposeront empilés.

```
content: [
       Rectangle { ... },
       Circle { ... }      ]
```

Code pour Java FX 2

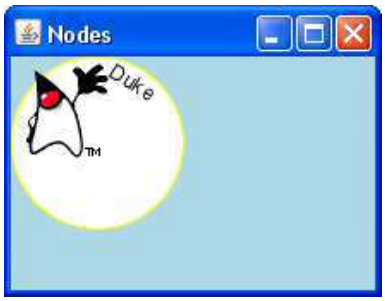
```
 public class JavaFXApplication1 extends Application {

  @Override
  public void start(Stage primaryStage) {

      StackPane root = new StackPane();
      Label l = new Label("Hello World!");
```

```
        root.getChildren().add(l);
        l.translateXProperty().set(25.0);
        l.translateYProperty().set(25.0);


        Scene scene = new Scene(root, 300, 250);

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Gérer la scene (JFX 1)

Afin de gérer la disposition des objets dans la scene, on peut jouer sur des transformations à l'aide de l'attribut *transforms*, comme dans l'exemple ci-dessous:

```
content: [
        Circle {
            centerX: 50   centerY: 50   radius: 50
            stroke: Color.YELLOW
            fill: Color.WHITE
        },
        Text {
            transforms: Transform.rotate(33, 10, 100)
                content: "Duke"
        },
            ImageView {
            image: Image {url: "{__DIR__}dukewave.png"}
        } ]
```



Code pour Java FX 2

```
* @see http://www.java2s.com/Code/Java/JavaFX/UsingRadialGradienttocreateaBall.htm
    StackPane root = new StackPane();
```

```
    Circle c = new Circle(50,50,50);
    c.strokeProperty().setValue(Color.YELLOW);
    c.fillProperty().setValue(Color.WHITE);
    Text t = new Text("Duke");
    t.rotateProperty().setValue(33);
    t.translateXProperty().setValue(20); t.translateYProperty().setValue(-30);
    Image i = new Image("file:/home/jf/ensib/cours/java_embarque/images/ensi.png");
    ImageView iv1 = new ImageView();
    iv1.setFitWidth(90); iv1.setPreserveRatio(true);
    iv1.setImage(i);

    root.getChildren().add(c); root.getChildren().add(t); root.getChildren().add(iv1);

    Scene scene = new Scene(root, 300, 250);
    primaryStage.setTitle("Circle");
    primaryStage.setScene(scene);
    primaryStage.show();
```

Combiner un effet graphique et un input (JFX 1)

Tout l'intérêt d'un binding prend son sens pour pouvoir lier un effet graphique à un input utilisateur, comme par exemple un curseur.

```
 def slider = Slider {
     min: 0 max: 60 value : 0
     translateX: 10 translateY: 110 }
content: [ slider, Circle {
       centerX: bind slider.value + 50 centerY : 60 radius: 50
       stroke: Color.YELLOW
       fill: RadialGradient {
           centerX:  50 centerY: 60 radius: 50 focusX: 50 focusY: 30
           proportional: false
           stops: [ Stop {offset: 0 color: Color.RED},
                    Stop {offset: 1 color: Color.WHITE}   ]
}} ] // End of RadialGradient and Circle
```

Code pour Java FX 2

```
// @see thanks to http://www.asgteach.com/blog/?p=272
public void start(Stage primaryStage) {
    StackPane root = new StackPane();
    final Slider s = new Slider(0,60,0);
    s.translateXProperty().setValue(10);
    s.translateYProperty().setValue(60);
    Circle c = new Circle(50,60,50);
```

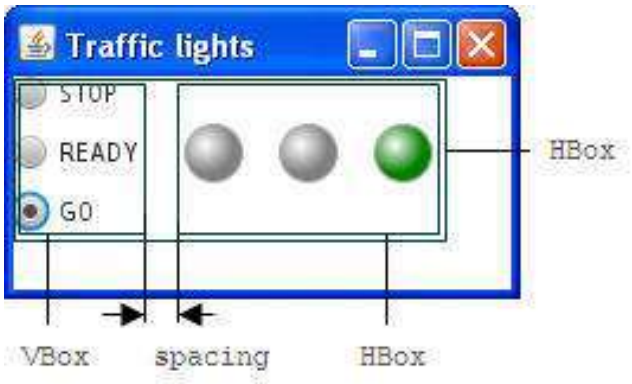
```
    c.strokeProperty().setValue(Color.YELLOW);
    RadialGradient radial = new RadialGradient(270, 50, 50, 60, 50, false, CycleMethod.NO_CYCLE,
            new Stop(0, Color.RED),
            new Stop(1, Color.WHITE));
    c.setFill(radial);

    DoubleBinding doubleBinding = new DoubleBinding() {
        { // Constructor here
          super.bind(s.valueProperty());
        }
        protected double computeValue() {
             return s.valueProperty().doubleValue();
    }};

    c.centerXProperty().bind(doubleBinding);
    root.getChildren().add(c); root.getChildren().add(s);
    Scene scene = new Scene(root, 300, 250);
    primaryStage.setTitle("Circle"); primaryStage.setScene(scene);
    primaryStage.show();
}
```

Résultat pour Java FX 2

Gérer la disposition des objets graphiques

Pour découper la scène en zone, il est possible d'utiliser des **VBox** et **HBox** qui s'incluent mutuellement les unes les autres.

```
content:
     HBox{
        spacing: 20
        content:[
           VBox{ spacing: 10
                 content: choices
           }
           HBox{ spacing: 15 content: lights nodeVPos: VPos.CENTER}
        ]
     }//HBox
```



Les principes d'une animation

A l'aide de l'objet **Timeline**, on peut définir une règle de modification sur une variable dépendante du temps. Il faut ensuite binder la variable sur une composante de l'objet graphique à animer.

```
var x: Number;

Timeline {
  keyFrames: [
         at (0s) {x => 0.0},
         at (4s) {x => 158.0 tween Interpolator.LINEAR}
  ]
}.play();
```

```
Path{
...
translateX: bind x
...
}
```

Les principes de l'interaction utilisateur

```
def playNormal = Image { url: "{__DIR__}play_onMouseExited.png"};
def playHover = Image { url: "{__DIR__}play_onMouseEntered.png"}; ...
var mode = true; //true for the Play mode, false for the Stop mode
var X: Number; var Y: Number;
Stage {
 title: "Play Button"
 scene: Scene {
     fill: Color.WHITE
     width: 300
     height: 240
     content: Group {
         content: [ button ]
         onMouseEntered: function(event) {
             image = if (mode){ playHover; } else { stopHover }
         }
         onMousePressed: function(event) {
             X = event.sceneX - event.node.translateX;
             Y = event.sceneY - event.node.translateY;
             image = if (mode){ playPressed; } else { stopPressed; }
         }
         onMouseDragged: function(event) {
             if (event.sceneX - X <0) {
                 event.node.translateX = 0;
             } else { if (event.sceneX - X > 300 - image.width){
                 event.node.translateX = 300 - image.width;
             } else {
                 event.node.translateX = event.sceneX - X;
             }
   ...  }}}
```

Démonstration du résultat

Sans redonner tout le code issu de [FXGUI], on comprend au travers du code précédent les grands principes de la programmation événementielle qui permet de gérer l'interaction utilisateur. Le rendu interactif obtenu est alors le suivant:

4 Développement client-serveur

4.1 Utilisation d'une socket

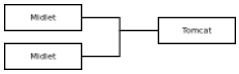
Une manière très primitive de faire communiquer une midlet avec une autre, est d'utiliser une socket. L'objet **Connector** fournit une classe implémentant **ServerSocketConnection** qui permet d'écouter sur un port donné.

```
ServerSocketConnection ssc = (ServerSocketConnection)
Connector.open("socket://:9002");
StreamConnection sc = null;
InputStream is = null;
try{
  sc = ssc.acceptAndOpen();
  is = sc.openInputStream();
  int ch = 0;
  StringBuffer sb = new StringBuffer();
  while ((ch = is.read()) != -1){
    sb.append((char)ch);
  }
  System.out.println(sb.toString());
```

Pour la partie cliente, se référer à SocketConnection

4.2 Midlets et serveur d'application Tomcat

[WDT] Une manière simple de réaliser une application client-serveur en Java utilise le serveur d'application Tomcat et véhicule les données sous forme d'un flot de *byte* sur le protocole HTTP. On dispose alors d'une solution complète de bout en bout basée sur le langage Java. Bien entendu, le serveur n'est pas destiné à s'exécuter sur le système embarqué. Cette solution est donc bien adaptée à une application client-serveur ou un seul serveur sert plusieurs clients nomades.



Exemple de code du côté serveur

A partir de la classe HttpServlet, il suffit d'implémenter la méthode **doGet** qui renvoie, via l'objet **response** la réponse au client.

```
public class HitServlet extends HttpServlet {
private int mCount;

public void doGet(HttpServletRequest request,
```

```
    HttpServletResponse response)
    throws ServletException, IOException {
  String message = "Hits: " + ++mCount;

  response.setContentType("text/plain");
  response.setContentLength(message.length());
  PrintWriter out = response.getWriter();
  out.println(message);
}
}
```

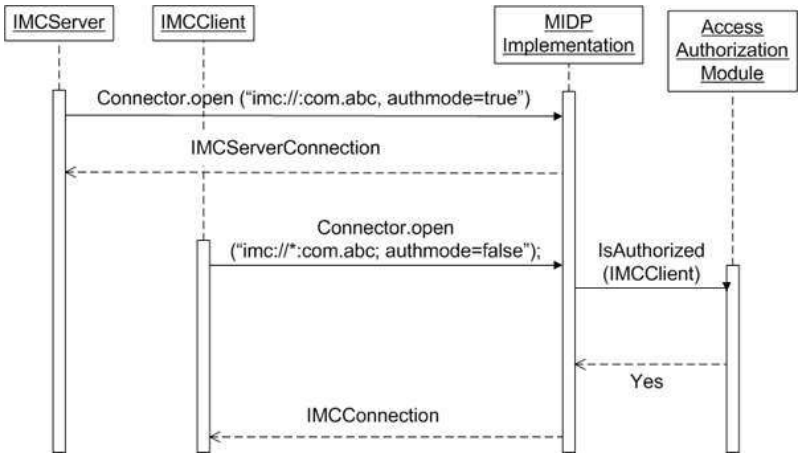
Exemple de code du côté client

Pour la midlet, rien d'extraordinaire à l'horizon:

```
HttpConnection hc = null;
 InputStream in = null;
 String url = getAppProperty("HitMIDlet.URL");

 try {
   hc = (HttpConnection)Connector.open(url);
   in = hc.openInputStream();

   int contentLength = (int)hc.getLength();
   byte[] raw = new byte[contentLength];
   int length = in.read(raw);

   in.close();
   hc.close();

   // Show the response to the user.
   String s = new String(raw, 0, length);
   ...
```

4.3 Communication inter-MIDlet

[IMC] La notion d'*Inter-MIDlet Communication* (IMC) recouvre la capacité des MIDlets à partager des informations entre elles. Dans la spécification MIDP 2.0 (2002), les midlets pouvaient déjà partager des informations au travers des *Record Store*. MIDP 3.0 (2009, [MIDP3.0]) apporte une vraie réponse pour la communication inter MIDlet. Cependant, la spécification étant récente, les implémentations ne sont pas encore très répandues sur le marché. C'est en fait une façon polie de dire que personne n'utilise cela.



The MIDP 3.0 IMC Connection Establishment Process [IMC]

Fonctionnement du client et du serveur

Classiquement, le serveur ouvre une connexion et attendant la connexion d'un client. Un client est autorisé à faire une requête même si le serveur n'est pas lancé, ce qui est très utile dans un système embarqué ou les contraintes peuvent nécessiter d'éteindre le serveur. Dans ce cas, le serveur IMC doit pouvoir s'enregistrer au travers de l'API **PushRegistry.registerConnection()**. Ainsi, lors d'une connexion d'un client, la MV sait quel instance du serveur lancer s'il n'est pas déjà présent.

La communication inter-MIDlet fonctionne ensuite de manière très similaire à une socket. Par exemple, la classe implémentant **IMCConnection** peut appeler les méthodes **openDataOutputStream** ou **openDataInputStream**, puis faire des *write* ou des *read*.

Exemple d'application: un portefeuille

Par exemple, [IMC] montre comment implémenter une sorte de portefeuille qui stocke le *login status* d'un utilisateur, pour éviter d'avoir à redemander l'information pour chaque application.

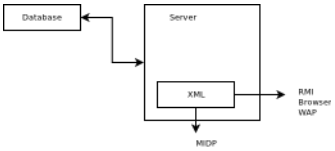
```
IMCServerConnection imcServerConnection = (IMCServerConnection)
  Connector.open("imc://:com.abc.sessionManager:1.0");
IMCConnection imcConnection = (IMCConnection)
  imcServerConnection.acceptAndOpen();
DataOutputStream dataOutputStream =
  imcConnection.openDataOutputStream();

// Now that the connection is open,
// send the login status.
dataOutputStream.writeUTF(loginStatus);
dataOutputStream.flush();
```

```
IMCConnection imcConnection = (IMCConnection)
  Connector.open("imc://:com.abc.sessionManager:1.0");
DataInputStream dataInputStream = imcConnection.openDataInputStream();

// Now that the connection is open,
// retrieve the log-in status.
```

```
loginStatus = dataInputStream.readUTF();
```

4.4 XML

[PXML] Dans une architecture multi-tiers, l'intérêt d'introduire une communication basée sur le XML est d'éviter la multiplication des interfaces de communication du serveur avec les différents clients. Le serveur doit juste produire les données XML adaptées à la nature du client.



Du point de vue d'un client MIDP, il y a des intérêts et inconvénients:

- La validation du XML peut être désactivée en production
- Le XML peut être tronqué pour alléger les traitements clients
- Le XML peut tout de même être verbeux et lourd pour un client MIDP
- Le client MIDP peut être déconnecté fréquemment

Parsers

La table suivante [PXML] montre les différents parseurs compatibles avec les plate-formes MIDP.

```
Name          License          Size    MIDP    Type
ASXMLP 020308 Modified BSD     6 kB    yes     push, model
kXML 2.0      EPL              9 kB    yes     pull
kXML 1.2      EPL              16 kB   yes     pull
MinML 1.7     BSD              14 kB   no      push
NanoXML 1.6.4 zlib/libpng      10 kB   patch   model
TinyXML 0.7   GPL              12 kB   no      model
Xparse-J 1.1  GPL              6 kB    yes     model
```

Chaque parseur possède ses spécificités (licence, interface), sa propre empreinte mémoire (taille du jar) et efficacité d'exécution. Ils peuvent être classés dans trois grandes familles:

- Push parser: lit le document au fur et à mesure et appelle les call-backs / écouteurs.
- Pull parser: lit le document au fur et à mesure et donne chaque élément à chaque appel de l'application
- Model parser: représente l'intégralité du document en mémoire.

Performances

[PXML] présente un ensemble de conseils pour améliorer la performance des applications qui doivent parser du XML.

1) Runtime performance: l'établissement d'une connexion réseau prend un temps non négligeable. Il est donc important d'agréger les données au mieux dans un document XML unique. Le serveur peut par exemple prendre les différents morceaux et construire un document les regroupants. Il pourra en même temps filtrer les parties inutiles du document afin de minimiser la taille et l'utilisation de la bande passante.

2) Perception utilisateur: bien que cela n'optimise pas la longueur des traitements, il est intéressant de soigner la perception utilisateur. Par exemple, la partie réseau et *parsing* peut être déportée dans un thread à part, afin d'éviter le gèle de l'interface utilisateur. De plus, l'interface utilisateur peut-être remise à jour fréquemment, au fur et à mesure de l'arrivée des données, *pendant* le traitement de celles-ci.

3) Taille du code: le code, et donc la taille du JAR final, peut-être réduit en coupant les classes, méthodes, variables inutiles. Des outils d'obfuscation peuvent réaliser ces tâches, et même *refactorer* les noms de méthodes/classes afin de gagner en taille (JAX (IBM), RetroGuard (Retrologic System)).

kXML et l'interface XmlPullParser

kXML est un bon exemple de *pull parser* pour MIDP. Ainsi, une application MIDP peut controler le parsing et agir sur l'interface au fur et à mesure que le document XML est téléchargé. kXML 2 implémente désormais l'interface **org.xmlpull.v1.XmlPullParser**.

Le *pull parsing* se base principalement sur l'appel à **next()** et **nextToken()**. Un appel à **nextToken()** fait avancer le parseur jusqu'au prochain événement et l'entier retourné détermine l'état du parseur:

- START_TAG / END_TAG: signalent le début et la fin d'un tag
- TEXT: des données ont été lues et sont disponibles via **getText()**
- END_DOCUMENT: plus de données à lire
- COMMENT: un commentaire a été lu
- CDSECT: une donnée CDATA a été lue

nextToken() permet d'avancer d'événement en évenement. La méthode **next()** permet de se focaliser sur les tags, en avancement directement jusqu'au prochain START_TAG / END_TAG / TEXT / END_DOCUMENT.

require(int type, java.lang.String namespace, java.lang.String name) permet de tester si l'événement courant est du bon type et si le nom attendu correspond, sinon une exception est levée.

Exemple: kXML manipulé via XmlPullParser

Cet exemple, issu de la javadoc de **XmlPullParser**, peut fonctionner avec un parseur kXML 2.

```
XmlPullParser xpp = new KXmlParser();

xpp.setInput( new StringReader ( "<foo>Hello World!</foo>" ) );
int eventType = xpp.getEventType();
while (eventType != XmlPullParser.END_DOCUMENT) {
 if(eventType == XmlPullParser.START_DOCUMENT) {
     System.out.println("Start document");
 } else if(eventType == XmlPullParser.END_DOCUMENT) {
     System.out.println("End document");
 } else if(eventType == XmlPullParser.START_TAG) {
     System.out.println("Start tag "+xpp.getName());
 } else if(eventType == XmlPullParser.END_TAG) {
     System.out.println("End tag "+xpp.getName());
 } else if(eventType == XmlPullParser.TEXT) {
     System.out.println("Text "+xpp.getText());
 }
 eventType = xpp.next();
}
```

Exemple 2 de parsing kXML 2

Ce deuxième exemple, issu de la documentation de kXML, montre comment utiliser la méthode **require()** pour tester le bon formattage du document (sans passer par une DTD ou autre).

```
<elements>
  <text>text1</text>
  <text>text2</text>
</elements>
```

```
parser.nextTag();
parser.require(XmlPullParser.START_TAG, null, "elements");
while(parser.nextTag() == XmlPullParser.START_TAG) {
  parser.require(XmlPullParser.START_TAG, null, "text");
  // handle element content
  System.out.println("text content: "+ parser.nextText());
  parser.require(XmlPullParser.END_TAG, null, "text");
}

parser.require(XmlPullParser.END_TAG, null, "elements");
```

Bien que cela semble très pratique, il s'agit de bricolage. Toute évolution du document XML (insertion de nouveaux tags par exemple) va lever une exception au travers des appels à **require()** (ce qui est le but recherché). Si l'on souhaite contraindre fortement un document XML, une bonne vieille DTD ou une validation RelaxNG sont plus élégantes. Cependant, des contraintes de performances peuvent pousser à utiliser la méthode **require()** pour réaliser des vérifications minimalistes.

XMLSerializer

La spécification **org.xmlpull.v1** définit aussi une interface pour sérialiser un document XML.

- **setOutput(java.io.Writer writer)**: spécifier la sortie du *writer*.
- **startDocument(java.lang.String encoding, java.lang.Boolean standalone)**: débute le document.
- **startTag(java.lang.String namespace, java.lang.String name)** permet d'écrire un tag.
- **endTag(java.lang.String namespace, java.lang.String name)** permet de clore un tag.
- **text(char[] buf, int start, int len)**: écrit du texte.
- **flush()**: écrit les données en attente.

4.5 JavaScript Object Notation

[JSON] JSON est un format d'échange de données léger, issu du langage de script JavaScript. De ce fait, il est orienté navigateur web. Ceci étant, ses caractéristiques en font un bon candidat pour une utilisation en environnement embarqué.

JSON repose sur du texte, ce qui le rend très intéropérable. Il supporte tous les types primitifs de Java. La syntaxe est simple:

- les objets sont délimités par des accolades
- les paires nom=valeur sont séparées par des deux points
- les tableaux sont délimités par des crochets

Par exemple, la requête au web service JSON-Time donne:

```
Requête : http://json-time.appspot.com/time.json?tz=UCT
Réponse:
{
"tz": "UCT",
"hour": 21,
"datetime": "Sun, 17 Oct 2010 21:25:40 +0000",
"second": 40,
"error": false,
"minute": 25
}
```

JSON ME Java API

L'API JSON est découpée en deux *packages*:

org.json.me:

- JSONArray is an ordered sequence of values
- JSONException is error detected
- JSONObject is an ordered collection of name/value pairs
- JSONString is an Interface for JSON serialization
- JSONStringer is a subclass of JSONWriter for generating JSON syntax
- JSONTokener is a parsing utility class used by JSONObject and JSONArray
- JSONWriter is a convenience class to generate JSON syntax
- StringWriter is StringBuffer-based implementation of StringWriter. This a Sun Microsystems helper class, not part of the standard JSON Java API

org.json.me.util:

- XML is a helper class to generation JSON from XML
- XMLTokener extends JSONTokener in support of XML parsing

Exemple: les données à serializer

```
class DataTypes {
  public String   aString; // a string
  public String[] anArray; // an array
  public int      anInteger; // an integer
//public double   aDouble; // double not supported on CLDC 1.0
  public long     aLong; // a long
  public boolean  aBoolean; // a boolean

  public DataTypes(
          String   aString,
          String[] anArray,
          int      anInteger,
        //double   aDouble, // Not supported on CLDC 1.0
```

```
            long     aLong,
            boolean  aBoolean) {
        this.aString = aString;
        this.anArray = anArray;
        this.anInteger = anInteger;
      //this.aDouble = aDouble; // Not supported on CLDC 1.0
        this.aLong = aLong;
        this.aBoolean = aBoolean;
 }}
```

Exemple: génération du texte JSON

```
public String toJSON() {
  // Define an external an a nexted JSONObjects
  JSONObject outer = new JSONObject();
  JSONObject inner = new JSONObject();
  // Now generate the JSON output
  try {
      outer.put("datatypes", inner); // the outer object name
      inner.put("aString", aString); // a name/value pair
      JSONArray ja = new JSONArray();
      for (int i=0; i<anArray.length; i++) {
          ja.put(anArray[i]);
      }
      inner.put("anArray", ja); a name/value pair
      inner.put("anInteger", anInteger); a name/value pair
    //inner.put("aDouble", aDouble); // Not supported on CLDC 1.0
      inner.put("aLong", aLong); a name/value pair
      inner.put("aBoolean", aBoolean); a name/value pair
  } catch (JSONException ex) {
      // ...process exception
  }
  return outer.toString(); // return the JSON text
}
```

Exemple: relecture d'objets depuis un texte JSON

```
aString     = null; anArray     = null;
anInteger   = 0;
aLong       = 0; aBoolean    = false;

try {
    JSONObject outer = new JSONObject(ji); // the outer objet
    if (outer != null) {
        // Get the inner object and parse out the data
        JSONObject inner = outer.getJSONObject("datatypes");
        if (inner != null) {
            // Parse the name/value pairs
            aString = inner.getString("aString");
            JSONArray ja = inner.getJSONArray("anArray");
```

```
            if (ja != null) {
                anArray = new String[ja.length()];
                for (int i=0; i<ja.length(); i++) {
                    anArray[i] = (String) ja.get(i);
            }}
            anInteger = inner.getInt("anInteger");
            aLong     = inner.getLong("aLong");
            aBoolean  = inner.getBoolean("aBoolean");
```

Exemple: serialisation et deserialisation

```
 // Create an initialize instance of DataTypes
DataTypes dt = new DataTypes(
  "C. Enrique Ortiz", // a String
  new String[] {"tech","mobile","web", "apps"},  // an Array
  15569,      // an int
//0.0,        // a double, not supported on CLDC 1.0
  1234567890, // a long
  true);      // a boolean

// Covert object to JSON
String j = dt.toJSON();
System.out.println("*** toJSON: " + j);

// Initialize object from JSON
System.out.println("*** fromJSON:");
dt.fromJSON(j);

// Dump to console to see if it worked
dt.dump();
```

5 Licences

5.1 L1

```
/*
* DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER
*
* Copyright &copy; 2010, Oracle and/or its affiliates. All rights reserved.
* Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
* Oracle and Java are registered trademarks of Oracle and/or its affiliates.
* Other names may be trademarks of their respective owners.
*
* This file is available and licensed under the following license:
*
* Redistribution and use in source and binary forms, with or without modification,
* are permitted provided that the following conditions are met:

*   *  Redistributions of source code must retain the above copyright notice,
       trademark notice, this list of conditions, and the following disclaimer.

*   *  Redistributions in binary form must reproduce the above copyright notice,
       trademark notice, this list of conditions, and the following disclaimer in
       the documentation and/or other materials provided with the distribution.

*   *  Neither the name of Oracle nor the names of its contributors may be used
       to endorse or promote products derived from this software without specific
       prior written permission.

* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
* IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
* INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
* BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES LOSS OF USE,
* DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
* LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
* OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
*/
```

6 Bibliographie

EG	Understanding J2ME Application Models, Eric Giguere, SUN Developper Network, 2002.
MEDOC	Java ME Technology APIs & Docs.
EO	A Survey of Java ME Today, C. Enrique Ortiz, November 2007.
IMP	The Information Module Profile, Eric Giguere, SUN Developper Network, August 2004.
GUI	The Java ME GUI APIs at a Glance, Bruce Hopkins, SUN Developper Network, June 2007.
MIDP	MID Profile Java doc reference.
EVENT	MIDP Event Handling, Qusay H. Mahmoud, SUN Developer Network.
RMS	Package javax.microedition.rms.
FXT	Learning the JavaFX Script Programming Language, Scott Hommel, Oracle.
FXDOC	JavaFX Language Reference, Robert Field, Sun Microsystems.
FXGUI	Building GUI Applications With JavaFX, Dmitry Bondarenko, Irina Fedortsova, Alla Redko, Oracle.
WDT	Wireless Development Tutorial Part II, Jonathan Knudsen, Oracle.
IMC	MIDP 3.0 Features: Inter-MIDlet Communication and Events, Bruce Hopkins, octobre 2009.
JSON	Using JavaScript Object Notation (JSON) in Java ME for Data Interchange, C. Enrique Ortiz, aout 2008, Oracle.
PXML	Parsing XML in J2ME, Jonathan Knudsen, mars 2002, Oracle.
MIDP3.0	JSR 271 Specification: Mobile Information Device Profile for JavaTM Micro Edition, Motorola.