1 Introduction.

1.1 JGraphX : c'est quoi

JGraphx est une libraire JAVA qui permet de dessiner des graphes en 2D dans une application. Vous allez dire, c'est quoi un graphe. Pour cela, il suffit de lire le topic sur wikipedia qui décrit la théorie des graphes à cette adresse (http://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes). Ce n'est pas pour vous faire peur, mais maintenant vous allez comprendre la raison du développement de cette librairie.

Dans ce tutoriel, JGraphX n'est pas la pour résoudre des parcours de graphes, mais va plutôt servir à les dessiner, les représenter graphiquement et fournir un modèle de donnée qui va permettre de parcourir le graphe qui a été dessiné.

1.2 Deux versions mxGraph et JGraphx.

A l'origine du projet il existait JGraph, une libraire JAVA gratuite permettant de dessiner et parcourir des graphes. Deux versions distinctes ont été crées.

- JGraphx est la version gratuite du produit. Elle permet de dessiner des graphes dans une application java. C'est cette version qui sera décrite dans le tutoriel.
- mxGraph, quant à elle, permet de dessiner des graphes dans des applications WEB, tel que c'est décrit dans le site WEB de JGraph. Vous avez ici, un exemple d'application. Cette version n'est pas gratuite, il faut bien évidemment que son auteur puisse vivre.

1.3 Téléchargement de JGraphX.

Le téléchargement de JGraphX se fait ici

Le fichier à télécharger est un fichier ZIP que l'on peut décompresser sur votre disque et est constituer ainsi :

- Le répertoire doc contient la documentation de JGraphX et est constituée d'un manuel utilisateur (assez rudimentaire) et de la javadoc.
- Le répertoire examples contient quelques applications pour comprendre comment utiliser cette librairie, et je peux vous assurer que c'est bien utile.
- Le répertoire lib contient la libraire java que vous devez inclure dans le classpath pour pouvoir faire une application basée sur JGraphX
- Le répertoire src contient les sources de la libraire. Cela peut aider aussi pour mieux comprendre comment la libraire fonctionne.

2 Projet java pour JGraphX.

Afin de pouvoir utiliser la librairie JGraphX, il suffit simplement de mettre le fichier jgraphx.jar dans le class-path de l'application. Ce jar se trouve dans le répertoire lib du zip que l'on a téléchargé.

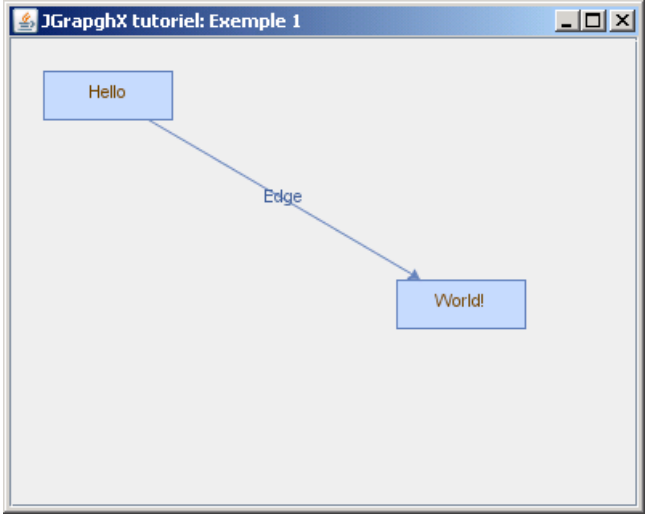
3 Un première application.

3.1 Comprendre le code de l'application.

Pour débuter, on va commencer par créer une petite application simple qui affiche un hello World dans une JFrame. Je ne l'ai pas écrite, elle provient directement du manuel.

```
package com.bpy.jgraph.tutorial;

import javax.swing.JFrame;

import com.mxgraph.model.mxCell;
import com.mxgraph.model.mxGeometry;
import com.mxgraph.swing.mxGraphComponent;
import com.mxgraph.view.mxGraph;

public class JGraphExemple1 extends JFrame {

   /** Pour eviter un warning venant du JFrame */
   private static final long serialVersionUID = -8123406571694511514L;

   public JGraphExemple1() {
      super("JGrapghX tutoriel: Exemple 1");

      mxGraph graph = new mxGraph();
      Object parent = graph.getDefaultParent();

      graph.getModel().beginUpdate();
      try
      {
         Object v1 = graph.insertVertex(parent, null, "Hello", 20, 20, 80,30);
         Object v2 = graph.insertVertex(parent, null, "World!", 240, 150,80, 30);
         graph.insertEdge(parent, null, "Edge", v1, v2);
      }
      finally
      {
         graph.getModel().endUpdate();
      }

      mxGraphComponent graphComponent = new mxGraphComponent(graph);
      getContentPane().add(graphComponent);   }

   /**
    * @param args
    */
   public static void main(String[] args) {
      JGraphExemple1 frame = new JGraphExemple1();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(400, 320);
      frame.setVisible(true);
   }
}
```

L'exécution de ce code permet d'obtenir la fenêtre suivante.



Que voit-on dans cette fenêtre ?

- Deux rectangles « Hello » et « World » de couleur bleu ; ces deux élément sont nommé vertex dans le langage JGraphX.
- Et une flèche qui relie ces deux éléments. Cette flèche est un Edge et on peut remarqué aussi que cette flèche est aussi nommé. La flèche indique, dans ce cas, la destination.

Et bien ce n'est pas tout, car il existe aussi un élément non visible qui est le graphe par lui-même. C'est le parent des trois éléments que l'on voit.

En regardant de plus près le code.

```
mxGraph graph = new mxGraph();
```

Tous graphe doit commencer par ceci. C'est le constructeur du graphe.

```
Object parent = graph.getDefaultParent();
```

On commence par récupérer le point d'entrer du graphe. Pour comprendre la raison de raison, il faut imaginer que votre graphe se présente sous la forme d'une structure de donnée de type arbre. Dans un arbre, il existe toujours une racine et ce parent est la racine de votre graphe.

```
Object v1 = graph.insertVertex(parent, null, "Hello", 20, 20, 80,30);
Object v2 = graph.insertVertex(parent, null, "World!", 240, 150,80, 30);
```

Ici, on créer les deux vertex (rectangle) en définissant leurs propriétés. Explication des paramètres.

- parent : les vertex sont lié directement à la racine du graphe
- null : Ici on attend une chaîne de caractère qui sert d'identifiant pour le vertex, cette chaîne est optionnel, on peut ne peut la renseigner.
- Objet contenu par le vertex. Ici c'est une string mais, cela pourrait être n'importe quel objet, tout dépend de ce que l'on veut faire avec ce graphe après.
- Les quatre derniers paramètres correspondent au x, y, largeur et hauteur du rectangle.

```
Object e1 = graph.insertEdge(parent, null, "Edge", v1, v2);
```

et ici, on ajoute le lien entre les deux vertex, ce que l'on nomme un edge

- parent : les vertex sont lié directement à la racine du graphe
- null : Ici on attend une chaîne de caractère qui sert d'identifiant pour ce edge, cette chaîne est optionnel, on peut ne peut la renseigner.
- V1 est le vertex source du edge

- V2 est vertex destination du edge

Pour finir, on voit que la création du graphe est encapsuler dans le code suivant :

```
graph.getModel().beginUpdate();
try
{
   …
}
finally
{
   graph.getModel().endUpdate();
}
```

Cette encapsulation bloque l'affiche du graphe pendant la modification du modèle. Cette protection est basée sur le principe d'un compteur, beginUpdate incrémente le compteur, endUpdate le décrémente et l'affichage est possible que si le compteur vaut 0.

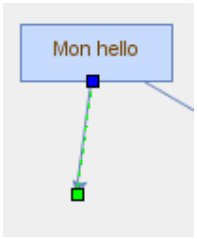
3.2 Comportement des objets dessinés

Jouons un peu avec notre graphe :

Si on place le curseur de la souris sur un vertex, on peur voir que le curseur réagit différemment en fonction de la position du curseur sur le vertex.

- Vers le centre du vertex on a le curseur en forme de main 

. ce curseur indique que vous pouvez faire de l'édition sur ce curseur.
    - Double clic permet de modifier le texte dans la boite. Pour valider le nouveau, il suffit de cliquer ailleurs dans le graphe.
    - Simple clic permet de rajouter un edge à partir de ce vertex. Pour cela, clique gauche sans relâcher le bouton de la souris et ensuite déplacer la souris dans le graphe. Au relâchement de la souris, un nouveau edge est créé.

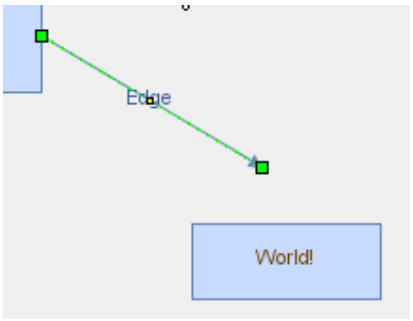


- Proche du bord, on a un curseur en forme de quatre flèches 

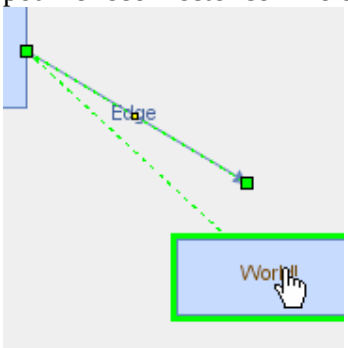
. ce curseur indique que l'on peut modifier la position et la taille du vertex.
    - Un simple clic permet de sélectionner le vertex, son affichage est modifié pour indiquer les points d'action possible (petits rectangles aux coins et centre du contours). En utilisant ces points d'actions, vous pouvez modifier la taille du vertex. Si on ne clique pas dans les points d'action, on peut déplacer le vertex dans le graphe. On peut remarque aussi que l'edge qui lie les deux vertex. suit le mouvement et reste connecté aux vertex.

L'egde est aussi modifiable, comme pour le vertex. on peut le sélectionner en cliquant dessus.

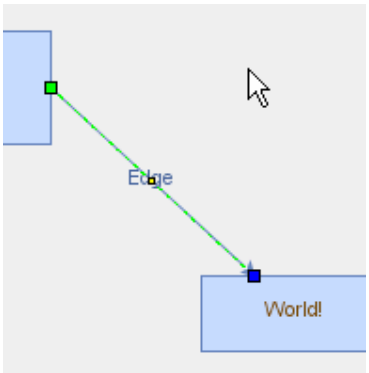
Dans ce cas, on remarque qu'un edge qui est sélectionné affiche deux petits rectangles à ces extrémités. La couleur de ces rectangles est importante, car elle indique que l'edge est bien relié aux vertex. En cliquant su l'edge, on peut le déplacer et obtenir ceci.



on voit que les petits rectangles ont changé de couleurs, ils sont maintenant vert. Cela signifie qu'ils ne sont plus liés aux vertex. Pour le vérifier, il suffit de bouger un vertex pour s'apercevoir que l'edge ne le suit pas. Pour reconnecter votre edge au vertex, il suffit de ramener un des petits rectangle vers le centre du vertex pour le reconnecter comme ceci :



et on obtient



En conclusion, on s'aperçoit que ce simple bout code apporte inclut déjà un grand nombre de mécanismes complexes pour le dessin des graphes.

3.3 Compréhension du model de donnée.

JGraphX est basé sur le principe Modèle/Vue, c'est-à-dire que vous voyer la représentation graphique d'un modèle. Ce modèle est basé sur une structure de donnée de type arbre. Afin de visualiser l'arbre de donnée qui a servi on va ajouter le code suivant à notre application.

```
   public JGraphExemple1() {
      . . .
      displayModel((mxCell) parent,"");
   }

   private void displayModel(mxCell cell, String indent) {
      System.out.println(indent+cell.getValue()+"("+cell.getClass().getName()+")");
      int nbChilds = cell.getChildCount();
      indent = indent + "  ";
      for (int i=0; i<nbChilds ; i++) {
         displayModel((mxCell) cell.getChildAt(i), indent);
      }
   }
```

La méthode displayModel va permettre d'afficher le contenu de notre modèle de graph.

Avec notre modèle, on va obtenir l'affichage suivant :

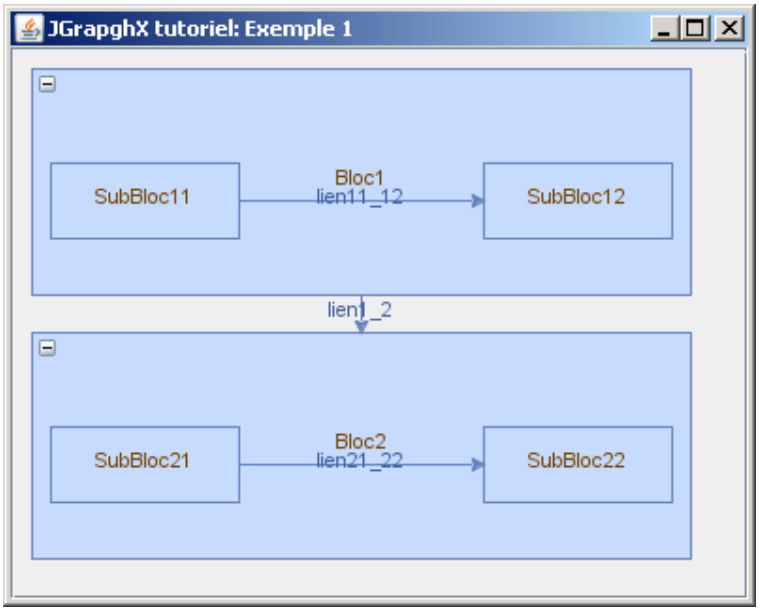
```
null(com.mxgraph.model.mxCell)
  Hello(com.mxgraph.model.mxCell)
  World!(com.mxgraph.model.mxCell)
  Edge(com.mxgraph.model.mxCell)
```

Ceci montre une architecture avec un élément null (c'est l'élément parent) qui contient trois enfants (Hello, World ! et Egde) ;

Maintenant, on va compliquer un notre graphe pour voir comment notre modèle va être impacter

```
  graph.getModel().beginUpdate();
  try {
    Object level1 = graph.insertVertex(parent, null, "Bloc1", 10, 10, 350, 120);
    Object level2 = graph.insertVertex(parent, null, "Bloc2", 10, 150, 350, 120);
    Object level1_1 = graph.insertVertex(level1, null, "SubBloc11", 10, 50, 100, 40);
    Object level1_2 = graph.insertVertex(level1, null, "SubBloc12", 240, 50, 100, 40);
    Object level2_1 = graph.insertVertex(level2, null, "SubBloc21", 10, 50, 100, 40);
    Object level2_2 = graph.insertVertex(level2, null, "SubBloc22", 240, 50, 100, 40);

    graph.insertEdge(level1, null, "lien11_12", level1_1, level1_2);
    graph.insertEdge(level2, null, "lien21_22", level2_1, level2_2);
    graph.insertEdge(parent, null, "lien1_2", level1, level2);
  }
  finally {
     graph.getModel().endUpdate();
  }
```

On va alors obtenir le graphe suivant. :



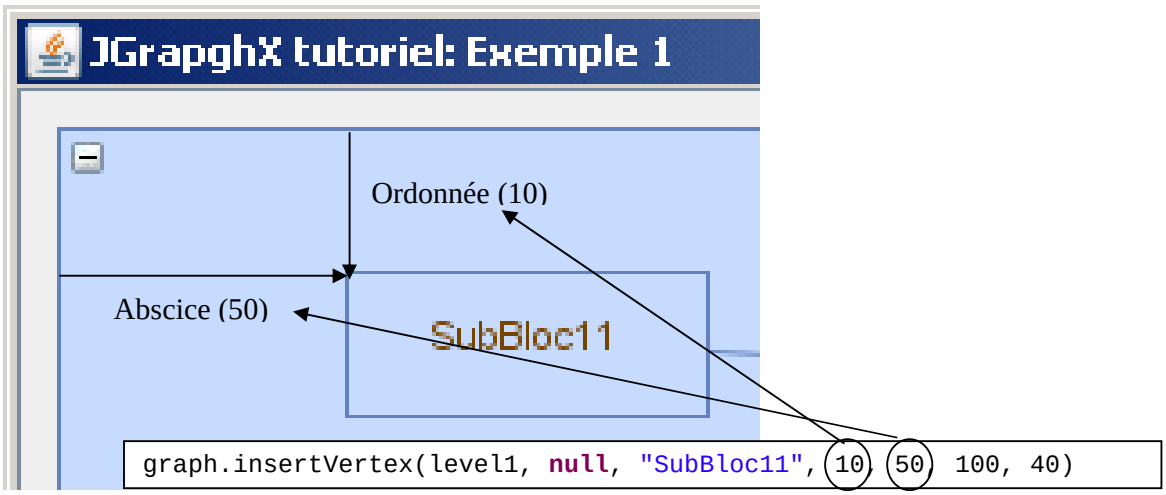
et l'affichage du modèle donnera :

```
null(com.mxgraph.model.mxCell)
  Bloc1(com.mxgraph.model.mxCell)
    SubBloc11(com.mxgraph.model.mxCell)
    SubBloc12(com.mxgraph.model.mxCell)
    lien11_12(com.mxgraph.model.mxCell)
  Bloc2(com.mxgraph.model.mxCell)
    SubBloc21(com.mxgraph.model.mxCell)
    SubBloc22(com.mxgraph.model.mxCell)
    lien21_22(com.mxgraph.model.mxCell)
  lien1_2(com.mxgraph.model.mxCell)
```

on voit que nous avons ajouter un nouveau niveau à notre modèle (Root, enfants, petits enfants)

Du point de vue comportement graphique, on peut noter les points suivants :

- Les coordonnées d'un élément du graphe se fait toujours par rapport à son parent.



- Lorsque l'on déplace un bloc parent, les blocs enfants suivent le mouvement.

4 Comment agir sur le comportement du graphe.

Ce chapitre va expliquer les possibilités offertes par JGraphX pour customiser les actions que l'on peut faire sur le graphe. Par exemple, on peut souhaiter ne pas pouvoir éditer le texte dans un Vertex ou un Edge, interdire les déplacements, etc.…

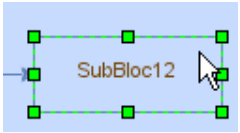
4.1 Comportement générale du graphe

Si on prend, par exemple, le cas du déplacement, JGraphX offre la possibilité de pouvoir autoriser ou interdire le déplacement des objets d'une manière globale ou individuelle.

Pour interdire globalement le déplacement de tous les objets dans u graph, c'est au niveau graphe que l'on doit indiquer notre choix. Pour cela, il faut utiliser la méthode setCellsMovable comme ceci :

```
public JGraphExemple1() {
  super("JGrapghX tutoriel: Exemple 1");

  mxGraph graph = new mxGraph();
  Object parent = graph.getDefaultParent();

  graph.getModel().beginUpdate();
  try
  {
    Object level1 = graph.insertVertex(parent, null, "Bloc1", 10, 10, 350, 120);
    Object level2 = graph.insertVertex(parent, null, "Bloc2", 10, 150, 350, 120);
    Object level1_1 = graph.insertVertex(level1, null, "SubBloc11", 10, 50, 100, 40);
    Object level1_2 = graph.insertVertex(level1, null, "SubBloc12", 240, 50, 100, 40);
    Object level2_1 = graph.insertVertex(level2, null, "SubBloc21", 10, 50, 100, 40);
    Object level2_2 = graph.insertVertex(level2, null, "SubBloc22", 240, 50, 100, 40);

    graph.insertEdge(level1, null, "lien11_12", level1_1, level1_2);
    graph.insertEdge(level2, null, "lien21_22", level2_1, level2_2);
    graph.insertEdge(parent, null, "lien1_2", level1, level2);
  }
  finally
  {
    graph.getModel().endUpdate();
  }

  graph.setCellsMovable(false);

  mxGraphComponent graphComponent = new mxGraphComponent(graph);
  getContentPane().add(graphComponent);
}
```

Si on relance l'application, on peut vérifier que les objets dans le graphe ne peuvent plus se déplacer. On peut aussi remarquer que le curseur de la souris n'est plus modifié



4.2 Comportement individuel des objets.

On sait maintenant modifier le comportement en général des objets d'un graphe. Mais on peut aussi vouloir modifier le comportement de quelques objets seulement. JGraphX nous offre aussi cette possibilité en jouant sur les styles des objets.

On souhaite, par exemple, ne pouvoir bouger les vextex level1 et level2 tout en autorisant le déplacement des vertex qui sont contenu par ces deux vertex parents.

Pour cela, lorsque va créer ces deux vertex, on ajoute un style comme ceci.

```
String style = mxConstants.STYLE_MOVABLE +"=0";
Object level1 = graph.insertVertex(parent, null, "Bloc1", 10, 10, 350, 120,style);
Object level2 = graph.insertVertex(parent, null, "Bloc2", 10, 150, 350, 120,style);
```

Quand on relance l'application, on peut vérifier que ces deux objets ne peuvent plus se déplacer, par contre, les objets contenus par ces deux vertex peuvent bouger.

Que peut on configurer :

Le tableau ci-dessous va décrire quelques comportements «customisables» fournis par la libraire.

Comportement	Méthode globale	Propriété
Déplacement	`graph.setCellsMovable(false);` `graph.setCellsMovable(true);`	`mxConstants.STYLE_MOVABLE +"=0"` `mxConstants.STYLE_MOVABLE +"=1"`
Edition	`graph.setCellsEditable(false);` `graph.setCellsEditable(true);`	`mxConstants.STYLE_EDITABLE +"=0"` `mxConstants.STYLE_EDITABLE +"=1"`
Redimension	`graph.setCellsResizable(true);` `graph.setCellsResizable(false);`	`mxConstants.STYLE_RESIZABLE +"=0";` `mxConstants.STYLE_RESIZABLE +"=1";`

Cette liste n'étant pas exhaustive, il faut se référer à la JAVADOC de la classe mxConstants pour avoir la lite complète des options possibles.

Pour en finir avec ce chapitre, on peut cumuler plusieurs option comme ceci, chaque propriété étant séparé pars des « ; »

```
String style = mxConstants.STYLE_RESIZABLE +"=0;" + mxConstants.STYLE_MOVABLE+"=0";
Object level1 = graph.insertVertex(parent, null, "Bloc1", 10, 10, 350, 120,style);
Object level2 = graph.insertVertex(parent, null, "Bloc2", 10, 150, 350, 120,style);
```

5 Comment agir sur l'aspect visuel d'un graphe.

JGraphX offre la possibilité de modifier l'aspect visuel d'un graphe en jouant sur les styles des objets dessinés. Comme vu dans le chapitre précédent, il suffit de passer un String lors de la création de l'objet pour en modifier l'aspect.

5.1 Aspect visuel des edges.

Pour ces éléments on peut modifier les débuts et fin de flèches

5.1.1 Modification des fins de flèches.

La fin de flèche se modifie avec la propriété `mxConstants.STYLE_ENDARROW`. La JAVADOC spécifie que l'on peut utiliser les constantes débutant par ARROW, c'est vrai, mais il faut appliquer une limitation supplémentaire : Du type String.

Voici quelques exemples :

Propriété :	Aperçu
`mxConstants.STYLE_ENDARROW + "=" + mxConstants.ARROW_BLOCK`	
`mxConstants.STYLE_ENDARROW + "=" + mxConstants.ARROW_CLASSIC`	
`mxConstants.STYLE_ENDARROW + "=" + mxConstants.ARROW_DIAMOND`	
`mxConstants.STYLE_ENDARROW + "=" + mxConstants.ARROW_OPEN`	
`mxConstants.STYLE_ENDARROW + "=" + mxConstants.ARROW_OVAL`	

5.1.2 Modification des débuts de flèches.

Comme pour les fin de flèche, on peut utiliser les mêmes styles pour la constante `mxConstants.STYLE_STARTARROW`

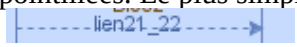
La question qui reste posée pourrait être : Mais comment fait on pour supprimer un bout de flèche. Toutes les constantes débutant par ARROW du type string ont été utilisées, mais aucune ne supprime le bout de flèche.

La solution à cette question est :

```
String edgeStyle = mxConstants.STYLE_STARTARROW + "=" + mxConstants.NONE;
```

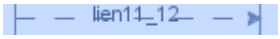
C'est logique, puisque NONE commence bien par ARROW.

5.1.3 Le tracé des lignes

Par défaut les lignes des edge sont des lignes pleines. Mais il est possible d'utiliser d'autres types de lignes comme par exemple les pointillées. Le plus simple est d'utiliser le style `mxConstants.STYLE_DASHED` qui donne le résultat suivant 

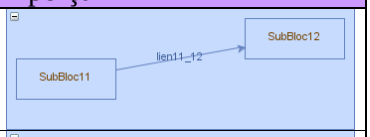
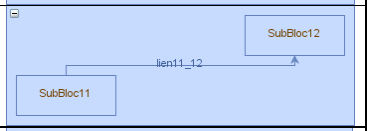
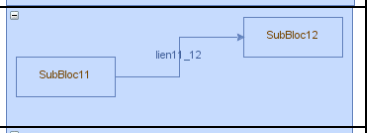
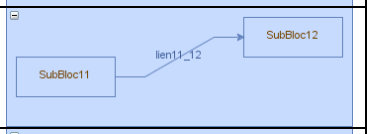
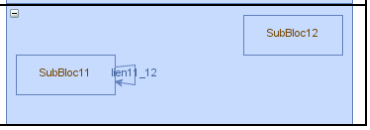
On peut vouloir aussi modifier l'apparence du pointillé. On le fait en combinant les constantes `mxConstants.STYLE_DASHED` et `mxConstants.STYLE_DASH_PATTERN`.

```
String edgeStyle = mxConstants.STYLE_DASHED + "=1;" +
                   mxConstants.STYLE_DASH_PATTERN + "=10";

Object edge11 = graph.insertEdge(level1,
                                 null,
                                 "lien11_12",
                                 level1_1,
                                 level1_2,
                                 edgeStyle);
```

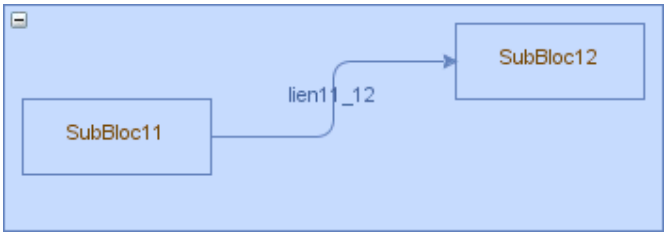
on obtient ce tracé. 

5.1.4 La forme des lignes.

De base les edges sont représentés par des lignes droites, mais il est possible de modifier cet aspect aussi. Le tableau suivant montre les différentes options possibles.

Propriété :	Aperçu
`Valeur par défaut`	
`mxConstants.STYLE_EDGE + "=" + mxConstants.EDGESTYLE_TOPTOBOTTOM;`	
`mxConstants.STYLE_EDGE + "=" + mxConstants.EDGESTYLE_SIDETOSIDE ;`	
`mxConstants.STYLE_EDGE + "=" + mxConstants.EDGESTYLE_ENTITY_RELATION;`	
`mxConstants.STYLE_EDGE + "=" + mxConstants.EDGESTYLE_LOOP;`	

En associant ces styles avec `mxConstants.STYLE_ROUNDED+"=1"` on arrondi alors les angles et cela donne



5.2 Aspect visuel des vertex.

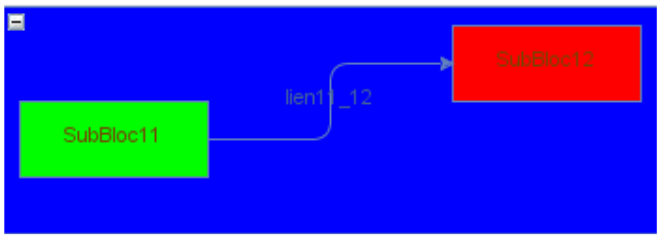
Comme pour les edges, il est possible aussi de modifier l'aspect visuel des vertex en jouant avec leurs styles.

5.2.1 Les couleurs des vertex.

On peut facilement jouer sur les couleurs de fond et du tour des edges. Prenons l'exemple suivant.

```
 String styleParent = mxConstants.STYLE_FILLCOLOR + "=#0000ff";
 Object level1 = graph.insertVertex(parent, null, "", 10, 10, 350, 120,styleParent);

 String styleEnfant1 = mxConstants.STYLE_FILLCOLOR + "=#00ff00";
 String styleEnfant2 = mxConstants.STYLE_FILLCOLOR + "=#ff0000";
 Object level1_1 = graph.insertVertex(level1, null, "SubBloc11", 10, 50, 100,
40,styleEnfant1);
 Object level1_2 = graph.insertVertex(level1, null, "SubBloc12", 240, 10, 100,
40,styleEnfant2);
```

Donnera le résultat suivant :



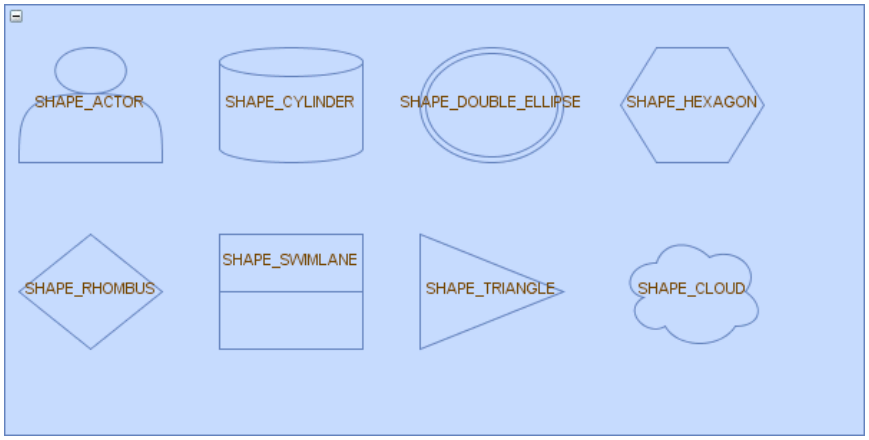
Les couleurs de fond des vertex a donc été modifiées. Le codage de la couleur respecte le principe #RRVVBB ou RR est la valeur hexadécimal du rouge, VV la valeur hexadécimal du vert et BB celle du bleu.

5.2.2 Forme des vertex.

Il est possible aussi de modifier la forme des vertex en jouant avec le style `mxConstants.STYLE_SHAPE` et de lui associé une des constantes commançaant par SHAPE
Voici quelques exemples :

```
        Object level1 = graph.insertVertex(parent, null, "", 10, 10, 600, 300);

        graph.insertVertex(level1, null, "SHAPE_ACTOR", 10, 30, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_ACTOR);
        graph.insertVertex(level1, null, "SHAPE_CYLINDER", 150, 30, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_CYLINDER);
        graph.insertVertex(level1, null, "SHAPE_DOUBLE_ELLIPSE", 290, 30, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_DOUBLE_ELLIPSE);
        graph.insertVertex(level1, null, "SHAPE_HEXAGON", 430, 30, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_HEXAGON);

        graph.insertVertex(level1, null, "SHAPE_RHOMBUS", 10, 160, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_RHOMBUS);
        graph.insertVertex(level1, null, "SHAPE_SWIMLANE", 150, 160, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_SWIMLANE);
        graph.insertVertex(level1, null, "SHAPE_TRIANGLE", 290, 160, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_TRIANGLE);
        graph.insertVertex(level1, null, "SHAPE_CLOUD", 430, 160, 100, 80,
              mxConstants.STYLE_SHAPE + "="+mxConstants.SHAPE_CLOUD);
```

qui donneront le résultat suivant



Je ne listerais pas ici toutes les styles que l'on peut utiliser pour modifier l'aspect visuel d'un vertex, je vous laisse le soin les rechercher dans la JAVADOC de la classe mxConstants.

6 Conclusion.

Ce petit tutoriel vous a expliqué les mécanismes de base pour afficher graphiquement un graphe dans une application JAVA. A vous de jouer maintenant.