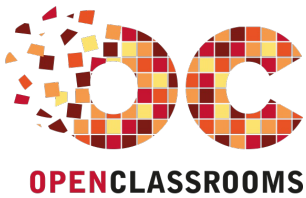
Du Qt en Java avec Qt Jambi

Par Natim



www.openclassrooms.com

Dernière mise à jour le 4/12/2012

Sommaire

Du Qt en Java avec Qt Jambi

Par  Natim

Mise à jour : 04/12/2012
Difficulté : Facile

Vous avez envie d'avoir des interfaces qui s'adaptent à votre environnement de travail ?
Et en plus, pour tout un tas de raisons, vous souhaitez le faire en Java plutôt qu'en C++ ?

Allons-y, je vais vous expliquer pas à pas ma démarche.

Ce tuto n'a pas pour vocation d'être la bible du Qt Jambi mais plutôt de vous aider à vous jeter dans la gueule du loup relativement simplement (ce qui est écrit juste après est le fruit de plusieurs heures de recherches).

Depuis la rédaction de ce tutoriel il y a bientôt 2 ans et demi, Qt a été racheté par Nokia qui a décidé de stopper le développement de Qt pour Java et donc de Qt Jambi. Cependant une communauté de développeur a repris le projet ici : http://qtjambi.sourceforge.net/ donc mon tutoriel a encore une utilité

Mon conseil est de vous orienter vers PyQt4 qui est le binding de Qt4 en Python. **Plaisir et Succès garantis !**

Partie 1 : Introduction à Qt Jambi

Avant toute chose, je vais vous expliquer comment je procède pour l'élaboration de ce tutoriel.

Avant d'écrire ce tutoriel, je ne connaissais rien à Qt et pas grand chose à Java  , j'ai donc lu la doc.

La documentation officielle de Jambi est très importante car elle décrit toutes les classes de Qt Jambi avec leurs méthodes publiques.
Donc si vous avez besoin de plus d'informations sur un objet ou ses méthodes, la réponse est dans la doc (ou dans le code source mais la doc est plus accessible.)

Pour la suite des parties de ce cours, je vais suivre le cours de Qt de M@teo21 et vous parler des différences et des spécificités de Java.

Sachez que ce qui est faisable en C++ avec Qt l'est en Java avec Qt Jambi.

Le point fort, c'est qu'il n'y a pas de pointeur à gérer et qu'il n'y a pas besoin de recompiler pour que ça fonctionne partout où il y a Qt Jambi et Java d'installés.

Le point faible, c'est que ce sera "moins rapide" qu'avec C++.

Je mets le moins rapide entre guillemet car ça ne nous importe pas dans la majorité des cas. On est pas à la seconde hein ...
Si par contre, notre programme est à destination scientifique et qu'il y a des algorithmes de calculs poussés, il vaut mieux tout faire en C++ bien que vous puissiez utiliser vos codes C++ en Java grâce à JNI.

Vous pouvez commencer à lire la présentation de Qt écrite par M@teo..

Pour la phase d'installation, ce sera un peu différent.
Voyez plutôt ...

Qt Jambi : Utiliser Qt avec Java

Pour programmer en Java, chaque programmeur a ses propres habitudes.
Pour ma part, je suis partisan du non IDE, tout en ligne de commandes ...

De la même manière que j'ai choisi Linux pour comprendre ce que je faisais, je préfère la ligne de commande quand je programme.

Seulement, si vous êtes sous Windows ou que vous n'aimez pas la ligne de commande, vous pouvez utiliser un IDE.
Je partirais volontier du principe que vous savez utiliser votre IDE  .

Téléchargement de la bibliothèque

Pré-requis

Pour commencer, vous devez avoir les outils pour utiliser Java en temps que programmeur sur votre machine.
Ensuite, il vous faut récupérer Qt Jambi pour votre système.

Ce que vous avez à faire :

- Télécharger et installer la dernière version du JDK pour votre système d'exploitation
  - Windows et Linux
  - Mac OS X possède déjà Java
- Télécharger et installer la dernière version de Qt Jambi

Installation de Qt Jambi

Installation

Vous installez le JDK.

Vous décompressez l'archive QtJambi de votre plateforme à un endroit accessible. (c:\qtjambi\, /opt/qtjambi/ ou ~/java/qtjambi/ par exemple)

Il vous faut ensuite configurer les variables d'environnement.

Configuration des variables systèmes

Plateforme	Configuration
Windows	La variable système *PATH* doit inclure le c:\qtjambi\bin
Linux	La variable système *LD_LIBRARY_PATH* doit inclure le /opt/qtjambi/lib
Mac OS	La variable système *DYLD_LIBRARY_PATH* doit inclure le ~/java/qtjambi/lib

Il vous faut aussi configurer le classpath pour ajouter qtjambi/qtjambi.jar à la liste des bibliothèques nécessaires à la compilation et à l'exécution.

Dans votre IDE favori, vous pouvez configurer qtjambi comme une nouvelle bibliothèque et ajouter la javadoc.
Il vous faudra ensuite ajouter QtJambi comme bibliothèque de votre projet.

Notes selon les plateformes

Plateforme	Configuration
Mac OS	Sous peine d'erreur de segmentation, vous devez lancer vos programmes avec l'option -XstartOnFirstThread

Si tout est installé, la fête peut commencer ...

Codons notre première fenêtre

Dans ce tutoriel, nous allons voir les bases de la création d'un projet en utilisant Qt Jambi.
Nous verrons comment ouvrir une fenêtre.

En avant

Code minimal d'un projet Qt Jambi

Un projet en Java est composé d'au moins une classe contenant une fonction main.
Selon votre IDE, vous pouvez créer un projet Java et y ajouter les librairies (qtjambi.jar en l'occurence)

Voici a quoi doit ressembler la methode main de votre objet principal :

Code : Java

```
import com.trolltech.qt.gui.QApplication;
...
public static void main(String[] args) {
    QApplication.initialize(args);

    QApplication.exec();
}
```

C'est le **code minimal** d'une application Qt en java.
Comme vous pouvez le constater, c'est vraiment très court et c'est un des points forts de Qt.

Analysons Pas à Pas ...

Import ...

Code : Java

```
import com.trolltech.qt.gui.QApplication;
```

On utilise la classe QApplication de Jambi qui est la classe de base de tout programme Qt.
Il faut donc la situer dans l'arborescence pour que le compilateur java la retrouve.

J'ai utilisé le chemin complet car je n'ai utilisé que cet objet.
On peut aussi décider d'inclure tous les objets du GUI comme ceci :

Code : Java

```
import com.trolltech.qt.gui.*;
```

D'un point de vue performance ça ne change rien.
D'un point de vue lisibilité du code, ça peut permettre à un programmeur de voir tout de suite de quoi il va s'agir et ce que la classe va utiliser.

Je vais dans ce cours utiliser des *import* explicites dans la mesure du possible (tant que ça ne nuit pas à la lisibilité).

QApplication, la classe de base

La première ligne du *main* crée un nouvel objet de type QApplication.
Cet objet est un Singleton (une classe dont il n'existe qu'une seule instance), cela veut dire qu'on n'en créé aucun objet, on l'utilise seulement.

À la différence du C++, il n'y a pas de constructeur puisqu'on ne crée pas d'objet QApplication.
On utilise donc la méthode statique *initialize* de la classe QApplication pour initialiser l'application et c'est cette méthode qui s'occupe de créer l'instance unique.

De manière à pouvoir passer des ordres (comme des options d'affichages par exemple) au framework Qt, on donne les paramètres de la ligne de commande.

Lancement de l'application

La méthode statique de la classe QApplication permet de lancer l'application.
Entre l'initialisation et l'exécution, on va créer nos objets (boites de dialogues, fenêtres, ...).
Ce n'est que lors de l'exécution de cette méthode que tout va s'enclencher.

Lorsque la méthode *exec* s'arrête, notre programme est terminé.

Affichage d'un widget

Dans la plupart des bibliothèques GUI, dont Qt fait partie, tous les éléments d'une fenêtre sont appelés des widgets. Les boutons, les cases à cocher, les images... tout ça ce sont des widgets. La fenêtre elle-même est considérée comme un widget.

Pour provoquer l'affichage d'une fenêtre, il suffit de demander à afficher n'importe quel widget. Ici par exemple, nous allons afficher un bouton.

Voici le code complet que j'aimerais que vous utilisiez. Il utilise le code de base de tout à l'heure mais y ajoute quelques lignes :

Code : Java

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QPushButton;
...
public static void main(String[] args) {
    QApplication.initialize(args);

    QPushButton bouton = new QPushButton("Salut les Zéros, la forme ?
");
    bouton.show();

    QApplication.exec();
}
```

On a ajouté un *import*, celui de notre nouveau widget le *QPushButton* (un bouton poussoir pour les anglophobes).

On crée notre objet *bouton* avec comme texte *"Salut les Zéros, la forme ?"*
Et on l'affiche.

Comme un bouton ne peut pas "flotter" comme ça sur votre écran, Qt l'insère automatiquement dans une fenêtre. On a en quelque sorte créé une "fenêtre-bouton"  .

Bien entendu, dans un vrai programme plus complexe, on crée d'abord une fenêtre et on y insère ensuite plusieurs widgets, mais là on commence simplement  .

Notre code est prêt, il ne reste plus qu'à compiler et exécuter le programme !

Compilation de notre premier programme en Qt Jambi

La classe Test.java

Pour les exemples, je préfère vous donner le fichier complet :

Test.java

Code : Java

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QPushButton;
```

```
public class Test
{
    public static void main(String[] args) {
        QApplication.initialize(args);

        QPushButton bouton = new QPushButton("Salut les Zéros, la
forme ?");
        bouton.show();

        QApplication.exec();
    }
}
```

Compilation

Si vous avez tout installé convenablement, un simple javac devrait fonctionner.

Code : Console

```
javac Test.java
```

Si votre classpath n'est pas bien défini, ça ne marchera pas ...
Je vous conseille de le définir (ça dépend de votre système d'exploitation)

Sinon, vous pouvez le définir juste pour cette commande comme cela :

Code : Console

```
javac -cp /chemin/de/qtjambi.jar:. Test.java
```

Sous Windows ça ressemblera surement plus à :

Code : Console

```
javac -cp c:\chemin\de\qtjambi.jar:. Test.java
```

Lancement

Pour les utilisateurs de Windows et Linux :

Code : Console

```
java Test
```

Pour les utilisateurs de Mac OS X :

Code : Console

```
java -XstartOnFirstThread Test
```

Améliorons notre application ...

Tout à l'heure, on a un peu innové en transformant le *Hello World* habituel par un *Salut les Zéros, la forme ?*

Maintenant, on peut s'amuser un petit peu avec notre fenêtre-bouton.
Ce qui serait bien serait que l'on puisse cliquer sur le bouton pour fermer la fenêtre.
On peut aussi fixer une taille par défaut ou changer la couleur et la police.
Mettre un titre à la fenêtre aussi.

Code : Java

```
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;

public class HelloWorld
{
    public static void main(String args[])
    {
      // Initialise l'application Qt
      QApplication.initialize(args);

      // Crée un bouton poussoir, bouton, dont le texte est Salut
les Zéros, la forme ?
      QPushButton bouton = new QPushButton("Salut les Zéros, la
forme ?");

      // Fixe la taille du bouton à 120x40
      bouton.resize(120, 40);

      // Fixe la police de caractères en Times 18 Gras
      bouton.setFont(new QFont("Times", 18,
QFont.Weight.Bold.value()));

      // Définit l'action du clic sur le bouton. (on quitte
l'application)
      bouton.clicked.connect(QApplication.instance(), "quit()");

      // Fixe le titre de la fenêtre
      bouton.setWindowTitle("Hello World");

      // Affiche le bouton
      bouton.show();

      // Lance l'application //
      QApplication.exec();
    }
}
```

C'est bien commenté donc je doute que vous ayez des problèmes pour comprendre.

Il y a un point un peu plus compliqué :

Code : Java

```
bouton.clicked.connect(QApplication.instance(), "quit()");
```

La méthode connect va nous permettre de mettre une action sur un widget.
En l'occurrence, on décide de connecter l'action *cliked()* de notre bouton à la méthode *quit()* de notre objet QApplication.

Comme QApplication est une classe statique et que la fonction connect attend une instance d'objet, la classe QApplication possède une méthode instance() qui permet de retourner l'objet de notre application principale.

Ce que je vous disais tout à l'heure à propos de QApplication, c'est qu'on l'utilisait en temps que classe statique.
Et cela est une protection. Pour être sûr que les programmeurs ne créent pas deux objets QApplication dans le même programme, les développeurs du framework ont décidé que ce serait la classe elle même qui allait gérer son propre et unique objet.

Cet objet est créé par la méthode statique *initialize()*. Et lorsqu'on a besoin de l'objet, on utilise la méthode statique *instance()* pour le récupérer.

Comme cela, on est sur de parler toujours du même objet et en plus on a pas besoin de se demander si on a accès à notre objet. Il suffit d'importer QApplication pour y avoir accès.

On importe les dépendances et ensuite on crée une simple fenêtre avec un bouton qui quitte l'application lorsque l'on clique dessus.

Si vous souhaitez compiler en ligne de commande :

Code : Console

```
# Compiler
javac HelloWorld.java
# Lancer
java HelloWorld.java
```

Si vous n'avez aucune erreur, c'est que vous êtes prêts à continuer et que vous êtes relativement habiles avec Java.
Si vous avez des erreurs, c'est que tout n'est pas encore très bien configurer et qu'il vous faut relire le chapitre expliquant comment installer Qt Jambi.
Nous y sommes enfin arrivés, champagne ! 

Vous l'avez vu, le code nécessaire pour ouvrir une fenêtre toute simple constituée d'un bouton est ridicule. Quelques lignes à peine, et rien de bien compliqué à comprendre au final.
C'est ce qui fait la force de Qt : "un code simple est un beau code" dit-on. Qt s'efforce de respecter ce dicton à la lettre, vous vous en rendrez compte dans les prochains chapitres.

Dans les prochains chapitres, nous allons voir comment changer l'apparence du bouton, comment faire une fenêtre un peu plus complexe. Nous découvrirons aussi le mécanisme des signaux et des slots, un des principes les plus importants de Qt qui permet de gérer les évènements : un clic sur un bouton pourra par exemple provoquer l'ouverture d'une nouvelle fenêtre ou sa fermeture !

Tout est installé, accrochez vos ceintures.

Partie 2 : Créer une fenêtre

Avec Qt, on a envie d'avoir des fenêtres.
Ces fenêtres peuvent être de deux types principaux.

- QDialog
- QMainWindow

L'objet QDialog a été largement étendu pour simplifier la tâche et vous aurez surement à l'étendre pour vos propres DialogBox.

L'objet QMainWindow vous permettra d'ajouter de nombreuses fonctionnalités à vos fenêtres (menus, barres d'outils, barre de status)

Disposer les choses

Pour pouvoir organiser les objets Qt comme vous le souhaitez dans vos fenêtres, il va falloir apprendre à les disposer.
Et connaître les objets nécessaires.

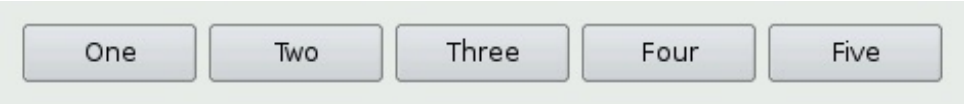
Trois types de disposition

Pour disposer les objets dans une fenêtre, on a trois possibilités :

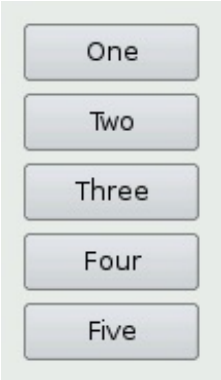
- QHBoxLayout : Pour placer les éléments horizontalement
- QVBoxLayout : Pour placer les éléments verticalement
- QGridLayout : Pour disposer les éléments comme on veut

En Anglais, disposition se traduit par Layout.

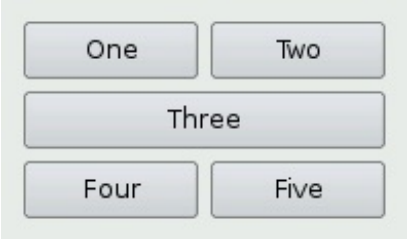
Voici à quoi ça ressemble :



QHBoxLayout



QVBoxLayout



QGridLayout

QHBoxLayout

Le QHBoxLayout permet d'aligner des objets horizontalement.

Code : Java

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QWidget;
import com.trolltech.qt.gui.QPushButton;
import com.trolltech.qt.gui.QHBoxLayout;

public class LoginBox extends QWidget
{
        public LoginBox()
        {
            QPushButton button1 = new QPushButton("One");
            QPushButton button2 = new QPushButton("Two");
            QPushButton button3 = new QPushButton("Three");
            QPushButton button4 = new QPushButton("Four");
            QPushButton button5 = new QPushButton("Five");

            QHBoxLayout layout = new QHBoxLayout();
            layout.addWidget(button1);
            layout.addWidget(button2);
            layout.addWidget(button3);
            layout.addWidget(button4);
            layout.addWidget(button5);

            setLayout(layout);
        }

        public static void main(String args[])
        {
            QApplication.initialize(args);

            LoginBox widget = new LoginBox();
            widget.show();

            QApplication.exec();
        }
}
```

Comme on peut le voir, ce n'est pas bien compliqué.

Code : Java

```
QHBoxLayout layout = new QHBoxLayout();
```

On crée un Layout

Code : Java

```
layout.addWidget(button1);
layout.addWidget(button2);
```

```
layout.addWidget(button3);
layout.addWidget(button4);
layout.addWidget(button5);
```

Puis on ajoute des widgets dedans (dans l'ordre)

Code : Java

```
setLayout(layout);
```

On fixe le layout comme élément principal du Widget.

À noter qu'il y a deux méthodes pour fixer l'élément central :

- setLayout()
- setWidget()

De même, si on voulait ajouter un layout dans un autre layout on utiliserait addLayout() et non addWidget()

.

QVBoxLayout

Cela fonctionne exactement de la même manière que pour QHBoxLayout sauf que c'est vertical.

Code : Java

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QWidget;
import com.trolltech.qt.gui.QPushButton;
import com.trolltech.qt.gui.QVBoxLayout;

public class LoginBox extends QWidget
{
        public LoginBox()
        {
            QPushButton button1 = new QPushButton("One");
            QPushButton button2 = new QPushButton("Two");
            QPushButton button3 = new QPushButton("Three");
            QPushButton button4 = new QPushButton("Four");
            QPushButton button5 = new QPushButton("Five");

            QVBoxLayout layout = new QVBoxLayout();
            layout.addWidget(button1);
            layout.addWidget(button2);
            layout.addWidget(button3);
            layout.addWidget(button4);
            layout.addWidget(button5);

            setLayout(layout);
        }

        public static void main(String args[])
        {
            QApplication.initialize(args);

            LoginBox widget = new LoginBox();
            widget.show();

            QApplication.exec();
        }
}
```

La seule chose qui change, c'est la classe du layout utilisé de QHBoxLayout à QVBoxLayout.

QGridLayout

Avec QGridLayout, les choses se modifient un petit peu.

Code : Java

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QWidget;
import com.trolltech.qt.gui.QPushButton;
import com.trolltech.qt.gui.QGridLayout;

public class GridBox extends QWidget
{
        public GridBox()
        {
            QPushButton button1 = new QPushButton("One");
            QPushButton button2 = new QPushButton("Two");
            QPushButton button3 = new QPushButton("Three");
            QPushButton button4 = new QPushButton("Four");
            QPushButton button5 = new QPushButton("Five");

            QGridLayout layout = new QGridLayout();
            layout.addWidget(button1, 0, 0);
            layout.addWidget(button2, 0, 1);
            layout.addWidget(button3, 1, 0, 1, 2);
            layout.addWidget(button4, 2, 0);
            layout.addWidget(button5, 2, 1);

            setLayout(layout);
        }

    public static void main(String args[])
    {
        QApplication.initialize(args);

        GridBox widget = new GridBox();
        widget.show();

        QApplication.exec();
    }
}
```

Pour chaque Widget ou Layout qu'on ajoute à notre Grid, on définie sa case de départ (ligne, colonne) et si elle fait plus de une case on donne le nombre de cases qu'elle prend sur la ligne et sur la colonne. (rowspan, colspan)

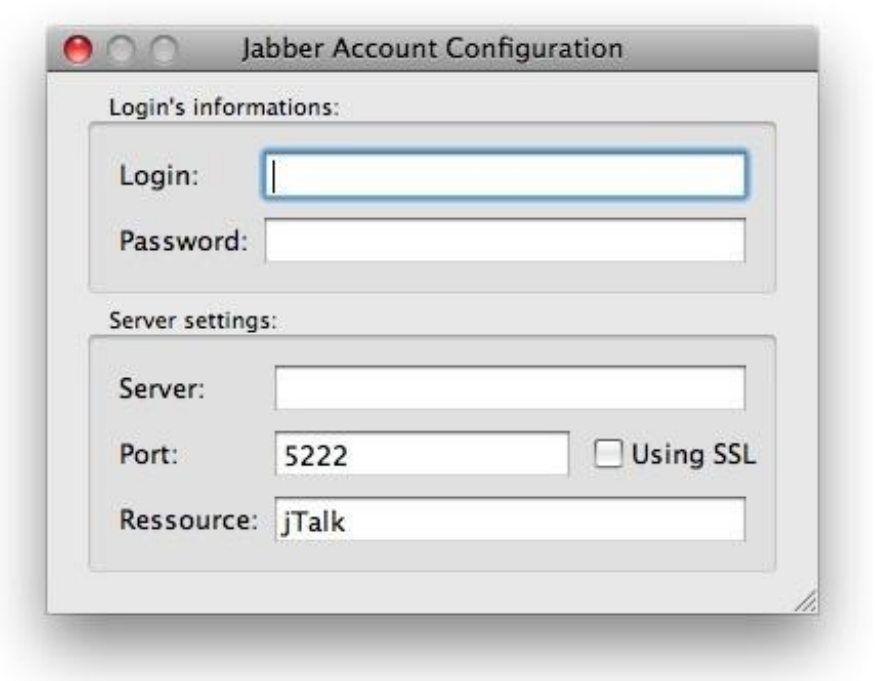
La première ligne et la première colonne commencent à 0 donc, le premier bouton se trouve en (0,0), le deuxième sur la même ligne mais dans la cellule 1 (0,1).

Le bouton de la ligne deux va prendre 2 cases. Il commence à la cellule 1 de la 2ème ligne (0,1) et s'étend sur une ligne et deux colonnes (0, 1, 1, 2).

Exercice

Maintenant, il faut pratiquer un peu.

Voici à quoi doit ressembler votre fenêtre :



Cette fenêtre vous permet de mettre en pratique tous les types de dispositions vus précédemment (Bien qu'on puisse tous les faire avec des Grilles) :

- QHBoxLayout
- QVBoxLayout
- QGridLayout

Pour les widgets, on utilise :

- QGroupBox
- QLabel
- QLineEdit
- QCheckBox
- QDialogButtonBox

Aide

Comment utiliser QGroupBox ?

Code : Java

```
QGridLayout loginLayout = new QGridLayout();
// On fabrique le layout ou le widget principal
loginGroup.setLayout(loginLayout);
// ou
loginGroup.setWidget(loginWidget);
```

Comment utiliser QDialogButtonBox ?

Code : Java

```
        QDialogButtonBox boutons = new QDialogButtonBox();
        boutons.addButton(QDialogButtonBox.StandardButton.Ok);
        boutons.addButton(QDialogButtonBox.StandardButton.Cancel);
```

Et pour les autres ?

Pour les autres widget, on lit la documentation et on fait marcher ses méninges
Vous pouvez maintenant commencer à créer vos propres fenêtres.
Mais il va nous falloir apprendre à les utiliser.

Une boite de dialogue

Maintenant qu'on sait disposer nos objets dans nos widgets, il faut finalement mettre ce widget dans une fenêtre pour pouvoir l'afficher.

Mais jusque là tout s'affichait très bien ?
Oui car votre gestionnaire de fenêtres ne pouvant pas afficher un widget seul nous créait une fenêtre pour que nous puissions afficher notre widget.
Mais si on veut changer le titre par exemple, il nous faut une vraie fenêtre.

L'objet QDialog

Voici ce que permettent les boites de dialogue :

- Notre boite de dialogue peut retourner une valeur (oui ou non par exemple)
- Les boites de dialogues contiennent le QSizeGrip, le widget en bas à droite qui permet de redimentionner la fenêtre
- L'affichage d'une boite de dialogue ayant un parent (un objet l'ayant lancé) se retrouve afficher au centre de cet objet.
- Une boite de dialog peut émettre trois signaux (nous verrons ce que sont les signaux dans une partie à cet effet)
  - accepted
  - rejected
  - finished

Une boite de dialogue est là pour une interaction brève avec l'utilisateur.

Une première boite de dialogue

Cette première boite de dialogue me permettra de corriger l'exercice précédent.

Code : Java

```
import com.trolltech.qt.gui.*;

public class LoginBox extends QWidget
{
        /* Données dont un accès peut-être utile depuis le reste de
la classe */
    private QLineEdit loginLineEdit;
    private QLineEdit passwordLineEdit;
    private QLineEdit serverLineEdit;
    private QLineEdit portLineEdit;
    private QCheckBox sslCheckBox;
    private QLineEdit ressourceLineEdit;

        public LoginBox()
        {
        /** Login's GroupBox' **/
        QGroupBox loginGroup = new QGroupBox(tr("Login's
informations:"));

        /* Login Line Edit with Label */
        QLabel loginLabel = new QLabel(tr("Jabber ID:"));
        loginLineEdit = new QLineEdit();
        loginLineEdit.setFocus();

        /* Password LineEdit with Label */
        QLabel passwordLabel = new QLabel(tr("Password:"));
        passwordLineEdit = new QLineEdit();
        passwordLineEdit.setEchoMode(QLineEdit.EchoMode.Password);

        /* Login's Layout */
        QGridLayout loginLayout = new QGridLayout();
        loginLayout.addWidget(loginLabel, 0, 0);
        loginLayout.addWidget(loginLineEdit, 0, 1);
        loginLayout.addWidget(passwordLabel, 1, 0);
        loginLayout.addWidget(passwordLineEdit, 1, 1);
        loginGroup.setLayout(loginLayout);

        /** Account GroupBox **/
```

```
            QGroupBox accountGroup = new QGroupBox(tr("Server
    settings:"));

            /* Server Line Edit with Label */
            QLabel serverLabel = new QLabel(tr("Server:"));
            serverLineEdit = new QLineEdit();

            /* Port LineEdit with Label and SSL selection */
            QLabel portLabel = new QLabel(tr("Port:"));
            portLineEdit = new QLineEdit("5222");
            sslCheckBox = new QCheckBox(tr("Using SSL"));

            /* Ressource Line Edit with Label */
            QLabel ressourceLabel = new QLabel(tr("Ressource:"));
            ressourceLineEdit = new QLineEdit("jTalk");

            /* Login's Layout */
            QGridLayout accountLayout = new QGridLayout();
            accountLayout.addWidget(serverLabel, 0, 0);
            accountLayout.addWidget(serverLineEdit, 0, 1, 1, 2);
            accountLayout.addWidget(portLabel, 1, 0);
            accountLayout.addWidget(portLineEdit, 1, 1);
            accountLayout.addWidget(sslCheckBox, 1, 2);
            accountLayout.addWidget(ressourceLabel, 2, 0);
            accountLayout.addWidget(ressourceLineEdit, 2, 1, 1, 2);
            accountGroup.setLayout(accountLayout);

            /* Dialog Button Box */
            QDialogButtonBox boutons = new QDialogButtonBox();

    boutons.addButton(QDialogButtonBox.StandardButton.Ok);

    boutons.addButton(QDialogButtonBox.StandardButton.Cancel);

            // Dialog Layout
            QGridLayout layout = new QGridLayout();
            layout.addWidget(loginGroup, 0, 0);
            layout.addWidget(accountGroup, 1, 0);
            layout.addWidget(boutons, 2, 0);
            setWindowTitle(tr("Jabber Account Configuration"));
            setLayout(layout);
            }

            public static void main(String args[])
            {
                QApplication.initialize(args);

                LoginBox widget = new LoginBox();
                widget.show();

                QApplication.exec();
            }
    }
```

On peut noter le setTitle() permettant de donner un titre à la fenêtre.
Notre fenêtre est redimensionnable.

Les boites de dialogues par défaut

Il existe tout un tas de boites de dialogue qui retournent un QMessageBox.StandardButton convertible implicitement en int.

On les trouve comme attributs de QMessageBox.

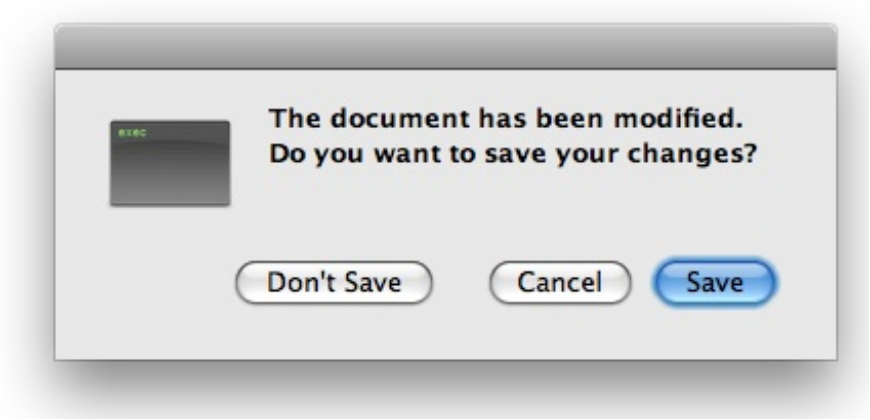
Code : Java

```
    QMessageBox.warning(null, tr("My Application"),
                            tr("The document has been modified.\n"+
                            "Do you want to save your changes?"), new
```

```
                                Do you want to save your changes? ), new
        QMessageBox.StandardButtons(QMessageBox.StandardButton.Save,
        QMessageBox.StandardButton.Discard,
        QMessageBox.StandardButton.Cancel));
```

Le fait d'ajouter le bouton Cancel vous permet d'utiliser Esc pour quitter la fenêtre.
Si vous souhaitez en mettre un autre, il faut utiliser setEscapeButton().
Vous pouvez aussi utiliser setDefaultButton() pour utiliser la touche Entrée.

En utilisant les boutons standards on a l'avantage de ne pas s'en préoccuper.



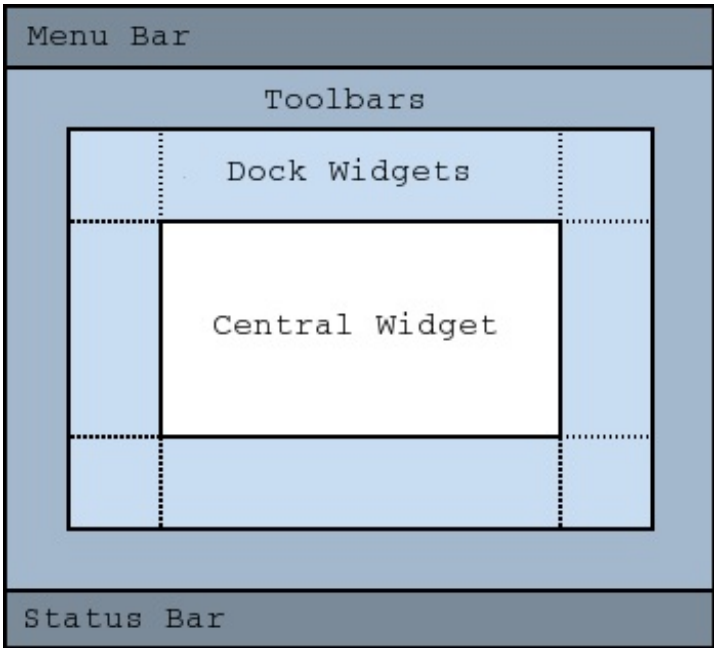
De manière à coller avec les règles graphiques de chaque système, on se rend compte que je n'ai pas de titre dans ma message box et que l'icône qui devait être un warning est en fait celui du programme. (Ici une console car je lance java en console)
Il est temps de passer à un peu de pratique ...

Une application avec des menus et des toolbars

Le plus souvent, pour organiser tout cela, on utilise des menus, des barres d'outils et une barre de status.
Qt nous propose un objet QMainWindow pour nous permettre de gérer tout cela.

L'objet QMainWindow

La structure d'une fenêtre QMainWindow se compose comme ceci :



La barre de menu

La barre de menu en haut de la fenêtre sous Windows et Linux et détachée sous Mac OS.
Les barres de menus sont de type QMenuBar

Les barres d'outils

Les barres d'outils qui peuvent être en haut, à gauche, à droite ou en bas et déplaçable par l'utilisateur.
Les barres d'outils sont de type QToolbar

Le dock

Les objets docks sont les objets comme les palettes de couleurs ou les outils dans des logiciels tels que Photoshop ou Flash.

Ces objets sont de type QDockWidget

Le Widget central

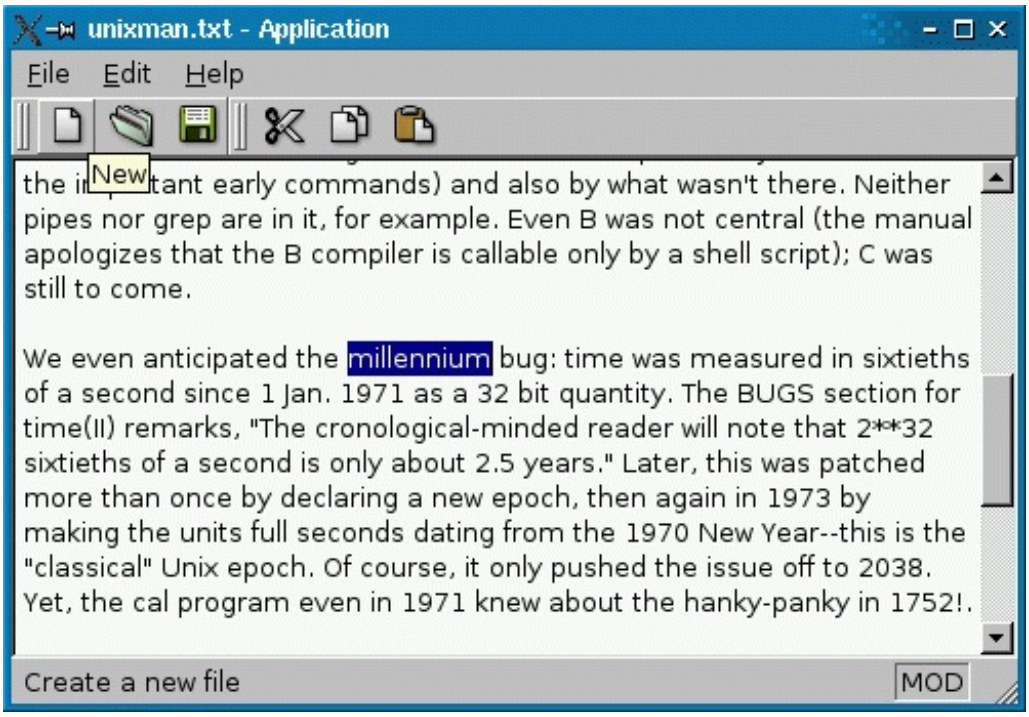
Le Widget central est le contenu proprement dit de notre application.
Ca peut être un widget ou un layout.

La barre de Status

Cette barre permet d'afficher un message décrivant le status du programme, la progression d'une action ...
La barre d'outil est un objet de type QStatusBar

Une première fenêtre d'application

Pour commencer, on va créer notre fenêtre :



Pour ce code vous aurez besoin du pack d'image.

Code : Java

```
/*
* JEditorDialog.java
* Version de JEditor sans les signaux
*
* Created on 2 novembre 2007
* Modified on 17 novembre 2007
*/

/* Import des classes espace de nomage nécessaires */
import com.trolltech.qt.QVariant;

/* Importation des éléments du GUI */
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QMainWindow;
import com.trolltech.qt.gui.QTextEdit;
import com.trolltech.qt.gui.QMenu;
import com.trolltech.qt.gui.QToolBar;
import com.trolltech.qt.gui.QAction;
import com.trolltech.qt.gui.QMenuBar;
import com.trolltech.qt.gui.QFileDialog;
import com.trolltech.qt.gui.QCloseEvent;
import com.trolltech.qt.gui.QMessageBox;
import com.trolltech.qt.gui.QIcon;
import com.trolltech.qt.gui.QKeySequence;
import com.trolltech.qt.gui.QCursor;

/* Définition de l'application et de ses attributs */
public class JEditorDialog extends QMainWindow {
    private String curFile;          // Fichier actuellement ouvert
    private QTextEdit textEdit;      // Widget permettant l'affichage
et la modification du texte
    private QMenu fileMenu;          // Menu Fichier
    private QMenu editMenu;          // Menu Edition
    private QMenu helpMenu;          // Menu Aide
    private QToolBar fileToolBar;    // Barre d'outil Fichier
    private QToolBar editToolBar;    // Barre d'outil Edition
    private QAction newAct;          // Action Nouveau
```

```
    private QAction openAct;         // Action Ouvrir
    private QAction saveAct;         // Action Enregistrer
    private QAction saveAsAct;       // Action Enregistrer Sous
    private QAction exitAct;         // Action Quitter
    private QAction cutAct;          // Action Couper
    private QAction copyAct;         // Action Copier
    private QAction pasteAct;        // Action Coller
    private QAction aboutAct;        // Action A Propos de JEditor
    private QAction aboutQtAct;      // Action A propos de Qt Jambi
    private String rsrcPath = "classpath:images"; // Répertoire des
images

    /* Définition du constructeur */
    public JEditorDialog()
    {
        QMenuBar menuBar = new QMenuBar();                // On crée
la barre de menu
        textEdit = new QTextEdit(this);
        setMenuBar(menuBar);                              // On
ajoute la barre de menu à notre Application
        setCentralWidget(textEdit);                       // On
ajoute la zone de Texte
        /* On lance les méthodes de création des différents
attributs de notre fenêtre */
        createActions();
        createMenus();
        createToolBars();
        createStatusBar();
        setUnifiedTitleAndToolBarOnMac(true);
    }

    /* Création des actions des menus et des toolbars */
    private void createActions()
    {
        /* Actions du menu Fichier */
        newAct = new QAction(new QIcon(rsrcPath + "/new.png"),
tr("&Nouveau"), this);
        newAct.setShortcut(new QKeySequence(tr("Ctrl+N")));
        newAct.setStatusTip(tr("Nouveau fichier"));

        openAct = new QAction(new QIcon(rsrcPath + "/open.png"),
tr("&Ouvrir..."), this);
        openAct.setShortcut(tr("Ctrl+O"));
        openAct.setStatusTip(tr("Ouvrir un fichier"));

        saveAct = new QAction(new QIcon(rsrcPath + "/save.png"),
tr("&Enregistrer..."), this);
        saveAct.setShortcut(tr("Ctrl+S"));
        saveAct.setStatusTip(tr("Enregistrer le fichier"));

        saveAsAct = new QAction(new QIcon(rsrcPath +
"/save_as.png"), tr("Enregistrer Sous..."), this);
        saveAsAct.setStatusTip(tr("Enregistrer le fichier sous
..."));

        exitAct = new QAction(tr("Quitter"), this);
        exitAct.setStatusTip(tr("Quitter l'application"));

        /* Actions du Menu Edition */
        cutAct = new QAction(new QIcon(rsrcPath + "/cut.png"),
tr("Cou&per"), this);
        cutAct.setShortcut(new QKeySequence(tr("Ctrl+X")));
        cutAct.setStatusTip(tr("Couper la séléction"));

        copyAct = new QAction(new QIcon(rsrcPath + "/copy.png"),
tr("&Copier..."), this);
        copyAct.setShortcut(tr("Ctrl+C"));
        copyAct.setStatusTip(tr("Copier la séléction"));

        pasteAct = new QAction(new QIcon(rsrcPath + "/paste.png"),
```

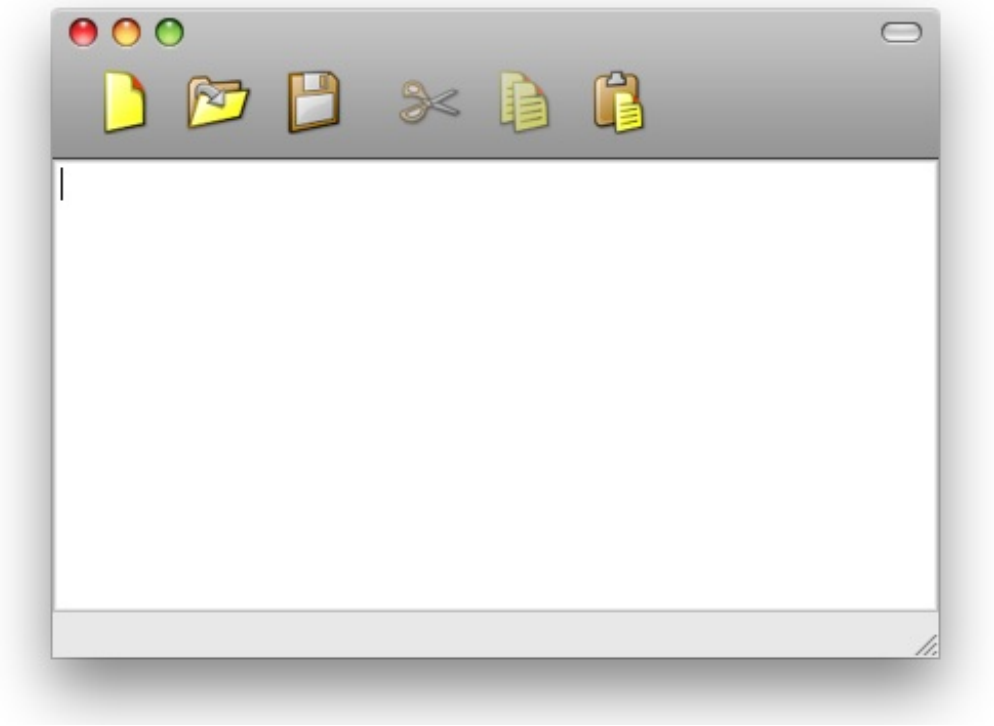
```
tr("Co&ller..."), this);
            pasteAct.setShortcut(tr("Ctrl+V"));
            pasteAct.setStatusTip(tr("Coller le texte précédement
couper ou copier"));

            /* Action du menu Aide */
            aboutAct = new QAction(new QIcon(rsrcPath + "/about.png"),
tr("A Propos de &JEditor"), this);
            aboutAct.setStatusTip(tr("A Propos de JEditor"));

            aboutQtAct = new QAction(new QIcon(rsrcPath + "/qt.png"),
tr("A Propos de &Qt"), this);
            aboutQtAct.setStatusTip(tr("Show the Qt library's About
box"));

            cutAct.setEnabled(false);
            copyAct.setEnabled(false);
        }

        /* Création des Menus */
        private void createMenus()
        {
            /* Menu fichier */
            fileMenu = menuBar().addMenu(tr("&Fichier"));
            fileMenu.addAction(newAct);
            fileMenu.addAction(openAct);
            fileMenu.addAction(saveAct);
            fileMenu.addAction(saveAsAct);
            fileMenu.addSeparator();
            fileMenu.addAction(exitAct);

            /* Menu Edition */
            editMenu = menuBar().addMenu(tr("&Edition"));
            editMenu.addAction(cutAct);
            editMenu.addAction(copyAct);
            editMenu.addAction(pasteAct);

            menuBar().addSeparator();

            /* Menu Aide */
            helpMenu = menuBar().addMenu(tr("&Aide"));
            helpMenu.addAction(aboutAct);
            helpMenu.addAction(aboutQtAct);
        }

        /* Création de la barre de menu */
        private void createToolBars()
        {
            fileToolBar = addToolBar(tr("Fichier"));
            fileToolBar.addAction(newAct);
            fileToolBar.addAction(openAct);
            fileToolBar.addAction(saveAct);

            editToolBar = addToolBar(tr("Edition"));
            editToolBar.addAction(cutAct);
            editToolBar.addAction(copyAct);
            editToolBar.addAction(pasteAct);
        }

        /* Création de la Barre de Status */
        private void createStatusBar()
        {
            statusBar().showMessage(tr("Pret"));
        }

        /* Lancement de l'application */
        public static void main(String[] args) {
              QApplication.initialize(args);

              JEditorDialog application = new JEditorDialog();
```

```
                application.show();

                QApplication.exec();
            }
    }
```

Il n'y a rien de vraiment compliqué.
En lançant l'application, vous pourrez essayer de déplacer les boites de dialogue.
Il existe une fonction spéciale permettant de transformer la barre d'outils sous Mac OS X.

Code : Java

```
    setUnifiedTitleAndToolBarOnMac(true);
```



Il nous reste à voir comment intéragir avec notre fenêtre.

Partie 3 : Les signaux et les slots

Maintenant que vous avez pu vous essayer à l'art d'une application simple, il s'agit de voir de plus prêt comment faire communiquer nos objets entre eux.

En effet, dans une GUI (prononcer gouï), on doit sans arrêt attendre l'action d'un utilisateur. Et il n'y a pas souvent une seule action faisable.

Lorsque l'on utilisait la console, on savait lorsqu'on demandait une information, ça ne pouvait être rien d'autre.
Maintenant les choses sont différentes, nous avons nos menus, nos barres d'outils, notre bloc principal, ...

Le but de cette partie est de vous faire découvrir comment fonctionne les signaux et les slots.
Comment on connecte un signal envoyé par un objet à une action à réaliser (par une méthode).

Accrochez votre ceinture, nous allons décoller.

Connecter les objets entre eux

Un objet Qt (dérivant de QObject) peut utiliser les mécanismes de signaux/slots.
Ce mécanisme est une implémentation du pattern observateur qui consiste à émettre un signal sans connaitre le(s) destinataire(s), et recevoir un signal sans connaitre l'émetteur (bien que l'on puisse le savoir si nécessaire, ce n'est en général pas nécessaire).

La manière dont elle a été portée pour Qt Jambi est très intéressante.
Lisez plutôt ...

Le principe du MVC (Model, View, Controller)

Lorsqu'on programme une interface graphique, il est recommandé d'utiliser le principe du MVC.

Je vais vous expliquer ce principe tel que je l'ai compris.
Si vous le connaissez mieux que moi, merci de corriger mes erreurs ou imprécisions pour en faire profiter tout le monde.

Le MVC est une méthode de conception pour le développement d'applications qui permet de séparer le développement en trois parties.

- Le modèle de données (Le Modèle)
- L'interface utilisateur (La Vue)
- La logique de contrôle (Le contrôleur)

Vous pouvez lire le très bon article de wikipédia en français à ce sujet.
Pour ceux qui n'ont pas de problèmes avec l'anglais vous pouvez lire l'explication du MVC dans la doc de Qt ainsi que le très bon article de Steve Burbeck sur la manière d'utiliser cette méthode.

- Le Modèle représente le comportement de l'application : traitement des données, interactions avec la base de données, etc. Il décrit les données manipulées par l'application et définit les méthodes d'accès.
- la Vue correspond à l'interface avec laquelle l'utilisateur interagit. Les résultats renvoyés par le modèle sont dénués de toute présentation mais sont présentés par les vues. Plusieurs vues peuvent afficher les informations d'un même modèle. Elle peut être conçue en html, ou tout autre « langage » de présentation. La vue n'effectue aucun traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle, et de permettre à l'utilisateur d'interagir avec elles.
- le Contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle. Il n'effectue aucun traitement, ne modifie aucune donnée, il analyse la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande.

On utilise très souvent ce type de méthode pour la réalisation d'applications web.
On peut citer par exemple des framework tels que :

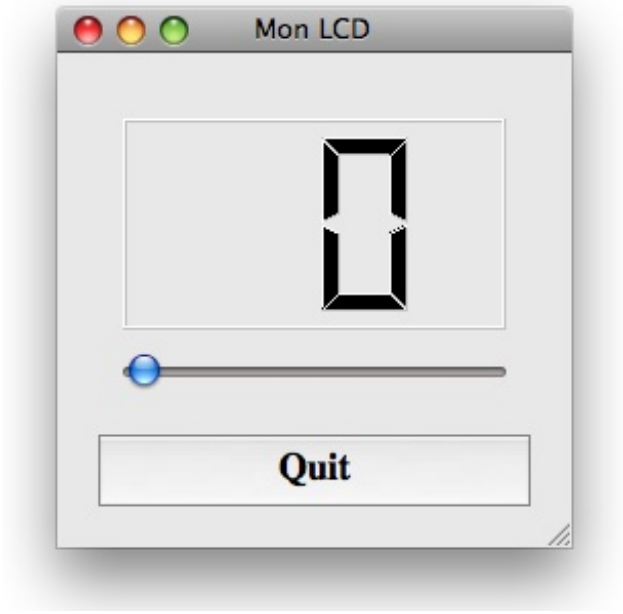
- Ruby on Rails : Framework en Ruby

- Django : Framework en Python
- Code Igniter : Framework en PHP

Le fait d'utiliser le MVC dans le milieu du web est un choix de développement très important.
Avec Qt, il est aussi largement souhaitable d'utiliser cette méthode.

Le principe avec Qt

On voit que les objets ont besoin de communiquer entre eux.
Prenons pour exemple ce petit programme :



Comme nous pouvons le voir, il est composé d'un QLCDNumber, d'un QSlider et d'un QPushButton le tout agencé dans un QVBoxLayout

Vous pouvez le réaliser comme cela :

Code : Java

```
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QWidget;
import com.trolltech.qt.gui.QLCDNumber;
import com.trolltech.qt.gui.QSlider;
import com.trolltech.qt.gui.QPushButton;
import com.trolltech.qt.gui.QVBoxLayout;

import com.trolltech.qt.core.Qt;

public class MonLCD1 extends QWidget
{
        public MonLCD1()
        {
            QLCDNumber lcd = new QLCDNumber(2);
            QSlider slider = new QSlider(Qt.Orientation.Horizontal);
            QPushButton quit = new QPushButton(tr("Quit"));

            QVBoxLayout layout = new QVBoxLayout();
            layout.addWidget(lcd);
            layout.addWidget(slider);
            layout.addWidget(quit);
            setLayout(layout);

            setWindowTitle(tr("Mon LCD"));
        }

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

```
                QApplication.initialize(args);

                MonLCD1 widget = new MonLCD1();
                widget.show();

                QApplication.exec();
            }
        }
```

Si vous lancez le code, vous verrez que ce n'est qu'une simple boite de dialogue avec quelques widget posés dedans.
Le but maintenant ce serait de connecter un peu tout cela.
Pour que lorsqu'on bouge le slider ça change la valeur inscrite sur le LCD et que lorsque l'on clique sur le bouton Quit, ça quitte l'application.

Le principe des Signaux

Si on regarde la doc pour la classe QSlider, on voit que celle ci possède les signaux suivant :

Signal	Description
valueChanged	Ce signal est émis lorsque la valeur du slider a changé
sliderPressed	Ce signal est émis lorsque l'utilisateur commence à déplacer le slider.
sliderMoved	Ce signal est émis lorsque l'utilisateur bouge le slider
sliderReleased	Ce signal est émis lorsque l'utilisateur relâche le bouton.

Pour savoir qu'un signal a été émis, on doit le connecter à une méthode.

Dans le cas de notre objet, ce qui pourrait être bien c'est d'afficher la valeur de notre slider sur l'écran LCD.

Connecter notre signal

Ce qu'il nous faut, c'est changer la valeur du LCD en fonction de celle du slider.

La méthode display de QLCDNumber, nous permet de faire cela.
Le signal envoyé par valueChanged nous envoi la valeur modifiée.

Il nous suffit donc de connecter le signal valueChanged(int) du slider avec la méthode display(int) de lcd.

Voici comment on fait cela :

Code : Java

```
slider.valueChanged.connect(lcd, "display(int)");
```

De même, nous souhaitons que notre bouton quitte l'application.

Nous allons donc connecter le signal clicked() du bouton à la méthode quit de QApplication.
Cependant, nous avons besoin de passer en paramètre la référence de l'objet QApplication.
Pour la récupérer, il nous faut utiliser QApplication.instance().

Code : Java

```
quit.clicked.connect(QApplication.instance(), "quit()");
```

Ajoutez ces deux lignes à la fin du constructeur de MonLCD et observez le résultat ...

Une application fonctionnelle

Pour bien faire les choses, nous aurions pu créer un nouveau widget LCDRange qui soit un LCD modifiable par un slider.

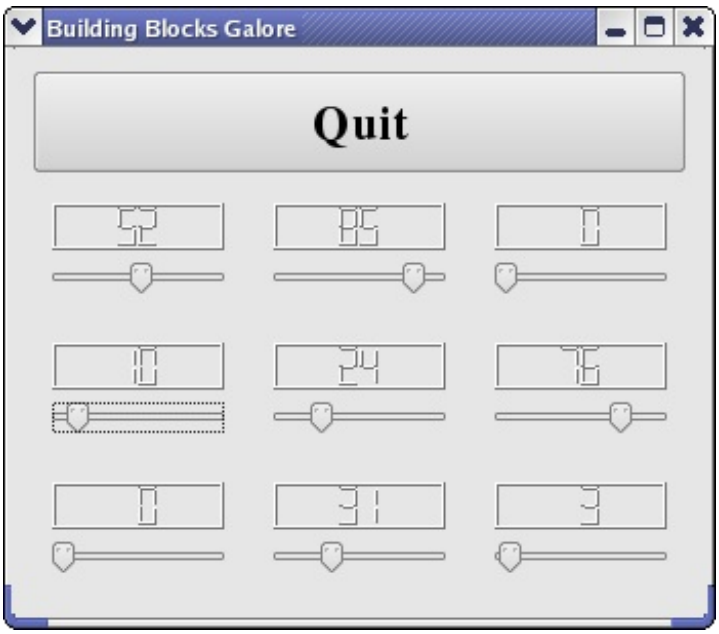
Voici ce que ça donne avec quelques modifications de style en plus :

Code : Java

```
import com.trolltech.qt.gui.*;
import com.trolltech.qt.core.Qt;

public class MonLCD extends QWidget
{
        public MonLCD()
        {
            LCDRange lcdRange = new LCDRange();

            QPushButton quit = new QPushButton(tr("Quit"));
            quit.setFont(new QFont("Times", 18,
QFont.Weight.Bold.value()));
            quit.clicked.connect(QApplication.instance(), "quit()");

            QVBoxLayout layout = new QVBoxLayout();
            layout.addWidget(lcdRange);
            layout.addWidget(quit);
            setLayout(layout);

            setWindowTitle(tr("Mon LCD"));
        }

        class LCDRange extends QWidget
        {
            public LCDRange()
            {
                QLCDNumber lcd = new QLCDNumber(2);
                lcd.setSegmentStyle(QLCDNumber.SegmentStyle.Filled);

                QSlider slider = new
QSlider(Qt.Orientation.Horizontal);
                slider.setRange(0, 99);
                slider.setValue(0);

                slider.valueChanged.connect(lcd, "display(int)");

                QVBoxLayout layout = new QVBoxLayout();
                layout.addWidget(lcd);
                layout.addWidget(slider);
                setLayout(layout);
            }
        }

        public static void main(String args[])
        {
            QApplication.initialize(args);

            MonLCD widget = new MonLCD();
            widget.show();

            QApplication.exec();
        }
}
```

Un petit exercice

Pour mettre en œuvre tout ce que nous avons vu au cours des précédents chapitres, vous pouvez essayer de faire quelque chose dans le genre :



Secret **(cliquez pour afficher)**

La solution se trouve ici : http://doc.trolltech.com/qtjambi-4.5.2 [...] utorial6.html

Nous avons vu comment utiliser les signaux, mais si on veut en envoyer ?

Emettre un signal

On peut bien sur utiliser les signaux existants, mais ça peut-être intéressant d'émettre des signaux entre ses propres objets.

Heureusement pour nous, avec Qt Jambi c'est très facile.
Voyez plutôt ...

Qu'est-ce que c'est un signal ?

Un signal c'est un Objet qui est défini comme attribut public d'une classe. (c'est assez rare d'avoir un attribut public pour le préciser)

Cet objet contient deux méthodes publiques principales :

- connect : Elle nous permet de connecter une méthode (slot) à ce signal
- emit : Elle nous permet d'émettre un signal.

Un signal peut émettre avec lui des variables en Java entre 0 et 9 variables.
Le nom du signal dépend de ce nombre de variables que l'on veut envoyer.

Voici comment on ajoute un signal à notre classe :

Code : Java

```
public Signal1<Integer> valueChanged = new Signal1<Integer>();
```

Comme vous le comprendrez surement, ce signal émet avec lui un entier (Integer).
C'est le signal de LCDNumber.

Et voilà, un signal ce n'est rien de plus que ça.

Emettre un signal

Pour qu'un signal soit utile, il faut un jour ou l'autre l'émettre.
Cela veut dire qu'on active la connection avec le slot.

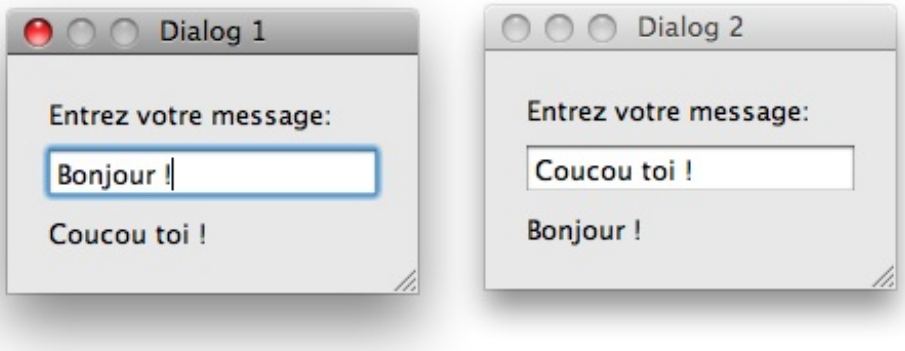
Autant se donner un but 

Cahier des charges

On va commencer par créer un nouveau widget qui contiendra un champ de texte QLineEdit ainsi qu'un QLabel.

Notre objet, lorsque le texte du QLineEdit va être changé va émettre un signal.
Ainsi, il pourra modifier le texte d'une autre boite de dialog. (Un minichat entre deux dialog).

ça va ressembler à ça :



Étant donné que QLineEdit s'occupe déjà de nous envoyer un signal, on ne va faire que le retransmettre.

Code : Java

```
import com.trolltech.qt.gui.*;

public class DialogButton extends QDialog {
        private QLabel text;

        public final Signal1<String> configured = new
Signal1<String>();

        public DialogButton(String titre){
                QLabel label = new QLabel(tr("Entrez votre
message:"));
                QLineEdit lineEdit = new QLineEdit();
                text = new QLabel();
                QVBoxLayout layout = new QVBoxLayout( this );
                layout.addWidget(label);
                layout.addWidget(lineEdit);
                layout.addWidget(text);
                setLayout(layout);
                setWindowTitle(titre);
                lineEdit.textChanged.connect(this, "emit(String)");
        }

        public void emit(String s){
                configured.emit(s);
        }

        public void write(String lineEditText){
                if(text.text() != lineEditText){
                        text.setText(lineEditText);
                }
        }

        public static void main(String[] args){
                QApplication.initialize(args);
                DialogButton app = new DialogButton("Dialog 1");
                DialogButton ap2 = new DialogButton("Dialog 2");
                app.configured.connect(ap2, "write(String)");
                ap2.configured.connect(app, "write(String)");
                app.show();
                ap2.show();
                QApplication.exec();
        }
}
```

Regardons de plus prêt :

Code : Java

```
public final Signal1<String> configured = new Signal1<String>();
```

On ajoute un signal *configured* à notre widget qui va émettre avec lui une chaine de caractères.
Cette chaine sera le contenu du LineEdit.

Code : Java

```
lineEdit.textChanged.connect(this, "emit(String)");
```

Ici on connecte le signal reçu du lineEdit quand le texte a changé avec notre méthode emit qui va émettre notre signal.

Code : Java

```
public void emit(String s){
      configured.emit(s);
}
```

Notre méthode emit() va seulement émettre notre signal configured.

Code : Java

```
public void write(String lineEditText){
       if(text.text() != lineEditText){
                text.setText(lineEditText);
       }
}
```

Dans notre méthode write, on vérifie que le texte a bien changé.
C'est une simple précaution pour éviter des boucles qui pourraient provoquer des scintillements de l'interface ou une augmentation indésirable de l'activité du processeur.

On ne sait en effet pas quels sont les mécanismes mis en place par la méthode setText. Donc par précaution, (surement inutile dans ce cas) on évite des calculs inutiles.

Dans notre méthode main, on crée des boites de dialogue que l'on nomme pour les différencier.
On connecte le signal de l'une avec la méthode write de l'autre et vice-versa.

Maintenant, il est temps de tester.

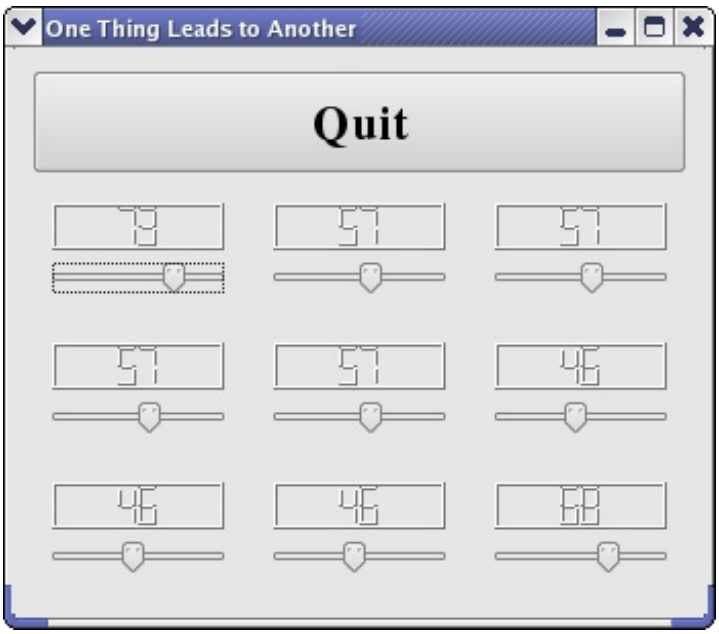
Petit exercice

Énoncé de l'exercice

Citation

En reprenant le résultat de l'exercice précédent, on souhaite, lorsqu'on modifie un slider, modifier les états de tous les sliders précédents.

Résultat Attendu



Question subsidiaire :

Citation

Définir l'ordre dans lequel ont été modifiés les sliders de la boite de dialogue du screenshot.
Pour cela on numérotera les sliders ainsi (0,0) pour le premier et (0,1) pour le deuxième, ... (2,0) pour le premier de la troisième ligne

Secret (cliquez pour afficher)

Le premier à avoir été modifié est le slider(2,2). Il est à 68 et a donc modifié tous les autres à 68.
Le deuxième est le slider(2,1) il est à 46 et à modifié tous les précédents à cette même valeur.
Le troisième est le slider(1,1) de même il a mis tous les précédents à 57.
Le quatrième et dernier et les slider(0,0) qui est à 73.

Solution

Secret (cliquez pour afficher)

La solution (si jamais vous en aviez besoin) se trouve dans les sources de qtjambi.
(com/trolltech/examples/tutorial/ConnectedSliders.java)

Maintenant, vous êtes prêts à réaliser le TP suivant.

Partie 4 : Les objets d'une fenêtre

Maintenant, il s'agit d'aller un peu plus loin.
On a la base, mais comment ça fonctionne tout cela ?

Comme j'ai décidé de coller au plus prêt du cours de M@teo21 sur Qt en C++, je vais attendre que ces cours sortent pour vous les proposer.

En attendant, comme je sens que vous avez envie de commencer à voir comment ça se passe sans attendre, je vous propose deux petits TP.

TP : JEditor, un simple éditeur de fichier texte

Pour commencer à nous familiariser avec Qt, je vous propose de réaliser un simple éditeur de texte.

Cahier des charges

Ce lecteur possédera 3 menus :

- Fichier
  - Nouveau
  - Ouvrir
  - Enregistrer
  - Enregistrer Sous
  - Quitter
- Édition
  - Couper
  - Copier
  - Coller
- Aide
  - A propos de JEditor
  - A propos de Qt Jambi

On pourra donc créer, ouvrir, enregistrer et enregistrer sous un fichier texte.
En plus d'avoir ces menus dans la barre de menu, on ajoutera une toolbar contenant les plus utiles.

La solution

Pour avoir une barre de menu, il nous faudra hériter de la classe QMainWindow de Qt.
En effet, avec Qt, lorsque l'on veut créer un widget, on hérite de la classe en Qt et ensuite on ajoute des méthodes d'actions spécifiques.
Par exemple les méthodes new(), open(), save(), saveas(), quit().

N'oubliez pas la documentation de Qt Jambi.
C'est votre source de renseignement par excellence. Dedans, il y a absolument tout.

Vous pouvez aussi la trouver dans votre installation de Qt Jambi en local.

Vous le savez peut-être, Qt Jambi, nous propose un outil très intéressant que l'on appelle QtDesigner.
Malgrès les apparences, cet outil est difficile à prendre en main.
En le lançant, on s'aperçoit très vite que l'on peut faire des choses très jolies.
Mais c'est un faux semblant car ensuite, vous n'arriverez pas à l'exploiter correctement dans vos programmes.

Pour bien comprendre la manière dont les choses se présentent, on ne va pas utiliser QtDesigner dans un premier temps.
Voici le code commenté de l'application :
Vous pouvez télécharger le pack d'image.

Code : Java

```
/*
* JEditor.java
* Version : 2
*
* Created on 2 novembre 2007
* Modified on 13 novembre 2007
*/
```

```
/* Import des classes espace de nommage nécessaires */
import com.trolltech.qt.QVariant;

/* Importation des éléments du GUI */
import com.trolltech.qt.gui.QApplication;
import com.trolltech.qt.gui.QMainWindow;
import com.trolltech.qt.gui.QTextEdit;
import com.trolltech.qt.gui.QMenu;
import com.trolltech.qt.gui.QToolBar;
import com.trolltech.qt.gui.QAction;
import com.trolltech.qt.gui.QMenuBar;
import com.trolltech.qt.gui.QFileDialog;
import com.trolltech.qt.gui.QCloseEvent;
import com.trolltech.qt.gui.QMessageBox;
import com.trolltech.qt.gui.QIcon;
import com.trolltech.qt.gui.QKeySequence;
import com.trolltech.qt.gui.QCursor;

/* Importation des éléments de gestions */
import com.trolltech.qt.core.QFile;
import com.trolltech.qt.core.QTextStream;
import com.trolltech.qt.core.Qt;


/* Définition de l'application et de ses attributs */
public class JEditor extends QMainWindow {
    private String curFile;          // Fichier actuellement ouvert
    private QTextEdit textEdit;      // Widget permettant l'affichage
et la modification du texte
    private QMenu fileMenu;          // Menu Fichier
    private QMenu editMenu;          // Menu Edition
    private QMenu helpMenu;          // Menu Aide
    private QToolBar fileToolBar;    // Barre d'outil Fichier
    private QToolBar editToolBar;    // Barre d'outil Edition
    private QAction newAct;          // Action Nouveau
    private QAction openAct;         // Action Ouvrir
    private QAction saveAct;         // Action Enregistrer
    private QAction saveAsAct;       // Action Enregistrer Sous
    private QAction exitAct;         // Action Quitter
    private QAction cutAct;          // Action Couper
    private QAction copyAct;         // Action Copier
    private QAction pasteAct;        // Action Coller
    private QAction aboutAct;        // Action A Propos de JEditor
    private QAction aboutQtAct;      // Action A propos de Qt Jambi
    private String rsrcPath = "classpath:images"; // Répertoire des
images

    /* Définition du constructeur */
    public JEditor()
    {
        QMenuBar menuBar = new QMenuBar();                // On crée
la barre de menu
        textEdit = new QTextEdit(this);
        setMenuBar(menuBar);                              // On
ajoute la barre de menu à notre Application
        setCentralWidget(textEdit);                       // On
ajoute la zone de Texte
        /* On lance les méthodes de création des différents
attributs de notre fenêtre */
        try {
              createActions();
          } catch (Exception e) {
              e.printStackTrace();
          }
          createMenus();
          createToolBars();
          createStatusBar();
          /* On lance la méthode documentWasModified lorsque le
contenu du texte est modifié */
```

```
            textEdit.document().contentsChanged.connect(this,
"documentWasModified()");
            /* Le fichier de départ est un fichier vide */
            setCurrentFile("");
        }

        /* On redéfinit la méthode lorsque l'on clique sur le bouton
fermer de la fenêtre pour demander à l'utilisateur
s'il veut enregistrer avant de quitter */
        public void closeEvent(QCloseEvent event)
        {
            if (maybeSave()) {
                event.accept();
            } else {
                event.ignore();
            }
        }

        /* Lors de l'ouverture d'un nouveau fichier, on doit fermer
le précédent, on avertit donc l'utilisateur.
On efface le contenu du widget textEdit et on met à 0 la valeur du
nom de fichier */
        public void newFile()
        {
            if (maybeSave()) {
                textEdit.clear();
                setCurrentFile("");
            }
        }

        /* Méthode d'ouverture du fichier */
        public void open()
        {
            if (maybeSave()) {
                String fileName = QFileDialog.getOpenFileName(this);
                if (fileName.length() != 0)
                    loadFile(fileName);
            }
        }

        /* On enregistre les modifications effectuées dans le fichier
*/
        public boolean save()
        {
            if (curFile.length() == 0) {
                return saveAs();
            } else {
                return saveFile(curFile);
            }
        }

        /* On enregistre les modifications dans un nouveau fichier */
        public boolean saveAs()
        {
            String fileName = QFileDialog.getSaveFileName(this);
            if (fileName.length() == 0)
                return false;

            return saveFile(fileName);
        }

        /* Fonction A propos de JEditor */
        public void about()
        {
            QMessageBox.about(this,
                              tr("A Propos de JEditor"),
                              tr("<b>JEditor<b> est une application
d'exemple"+
                                 "pour faire une démonstration de
l'utilisation de Qt Jambi"));
```

```
        }

        /* Méthode qui définit que le document a été modifié pour
savoir si on a besoin de le sauvegarder ou pas */
        public void documentWasModified()
        {
            setWindowModified(textEdit.document().isModified());
        }

        /* Création des actions des menus et des toolbars */
        private void createActions()
        {
            /* Actions du menu Fichier */
            newAct = new QAction(new QIcon(rsrcPath + "/new.png"),
tr("&Nouveau"), this);
            newAct.setShortcut(new QKeySequence(tr("Ctrl+N")));
            newAct.setStatusTip(tr("Nouveau fichier"));
            newAct.triggered.connect(this, "newFile()");

            openAct = new QAction(new QIcon(rsrcPath + "/open.png"),
tr("&Ouvrir..."), this);
            openAct.setShortcut(tr("Ctrl+O"));
            openAct.setStatusTip(tr("Ouvrir un fichier"));
            openAct.triggered.connect(this, "open()");

            saveAct = new QAction(new QIcon(rsrcPath + "/save.png"),
tr("&Enregistrer..."), this);
            saveAct.setShortcut(tr("Ctrl+S"));
            saveAct.setStatusTip(tr("Enregistrer le fichier"));
            saveAct.triggered.connect(this, "save()");

            saveAsAct = new QAction(new QIcon(rsrcPath +
"/save_as.png"), tr("Enregistrer Sous..."), this);
            saveAsAct.setStatusTip(tr("Enregistrer le fichier sous
..."));
            saveAsAct.triggered.connect(this, "saveAs()");

            exitAct = new QAction(tr("Quitter"), this);
            exitAct.setStatusTip(tr("Quitter l'application"));
            exitAct.triggered.connect(QApplication.instance(),
"quit()");

            /* Actions du Menu Edition */
            cutAct = new QAction(new QIcon(rsrcPath + "/cut.png"),
tr("Cou&per"), this);
            cutAct.setShortcut(new QKeySequence(tr("Ctrl+X")));
            cutAct.setStatusTip(tr("Couper la séléction"));
            cutAct.triggered.connect(textEdit, "cut()");

            copyAct = new QAction(new QIcon(rsrcPath + "/copy.png"),
tr("&Copier..."), this);
            copyAct.setShortcut(tr("Ctrl+C"));
            copyAct.setStatusTip(tr("Copier la séléction"));
            copyAct.triggered.connect(textEdit, "copy()");

            pasteAct = new QAction(new QIcon(rsrcPath + "/paste.png"),
tr("Co&ller..."), this);
            pasteAct.setShortcut(tr("Ctrl+V"));
            pasteAct.setStatusTip(tr("Coller le texte précédement
couper ou copier"));
            pasteAct.triggered.connect(textEdit, "paste()");

            /* Action du menu Aide */
            aboutAct = new QAction(new QIcon(rsrcPath + "/about.png"),
tr("A Propos de &JEditor"), this);
            aboutAct.setStatusTip(tr("A Propos de JEditor"));
            aboutAct.triggered.connect(this, "about()");

            aboutQtAct = new QAction(new QIcon(rsrcPath + "/qt.png"),
tr("A Propos de &Qt"), this);
```

```
            aboutQtAct.setStatusTip(tr("Show the Qt library's About
box"));
            aboutQtAct.triggered.connect(QApplication.instance(),
"aboutQt()");

            cutAct.setEnabled(false);
            copyAct.setEnabled(false);
            textEdit.copyAvailable.connect(cutAct,
"setEnabled(boolean)");
            textEdit.copyAvailable.connect(copyAct,
"setEnabled(boolean)");
        }

        /* Création des Menus */
        private void createMenus()
        {
            /* Menu fichier */
            fileMenu = menuBar().addMenu(tr("&Fichier"));
            fileMenu.addAction(newAct);
            fileMenu.addAction(openAct);
            fileMenu.addAction(saveAct);
            fileMenu.addAction(saveAsAct);
            fileMenu.addSeparator();
            fileMenu.addAction(exitAct);

            /* Menu Edition */
            editMenu = menuBar().addMenu(tr("&Edition"));
            editMenu.addAction(cutAct);
            editMenu.addAction(copyAct);
            editMenu.addAction(pasteAct);

            menuBar().addSeparator();

            /* Menu Aide */
            helpMenu = menuBar().addMenu(tr("&Aide"));
            helpMenu.addAction(aboutAct);
            helpMenu.addAction(aboutQtAct);
        }

        /* Création de la barre de menu */
        private void createToolBars()
        {
            fileToolBar = addToolBar(tr("Fichier"));
            fileToolBar.addAction(newAct);
            fileToolBar.addAction(openAct);
            fileToolBar.addAction(saveAct);

            editToolBar = addToolBar(tr("Edition"));
            editToolBar.addAction(cutAct);
            editToolBar.addAction(copyAct);
            editToolBar.addAction(pasteAct);
        }

        /* Création de la Barre de Status */
        private void createStatusBar()
        {
            statusBar().showMessage(tr("Prêt"));
        }

        /* Test si le fichier doit être sauvegarder ou non */
        private boolean maybeSave()
        {
            if (textEdit.document().isModified()) {
                QMessageBox.StandardButton ret =
QMessageBox.warning(this, tr("JEditor"),

tr("Le document a été modifié.\n" +

"Voulez-vous sauvegarder les modifications ?"),
```

```
        new QMessageBox.StandardButtons(QMessageBox.StandardButton.Ok,

        QMessageBox.StandardButton.Discard,

        QMessageBox.StandardButton.Cancel));
                    if (ret == QMessageBox.StandardButton.Ok) {
                        return save();
                    } else if (ret == QMessageBox.StandardButton.Cancel) {
                        return false;
                    }
                }
                return true;
            }

            /* Charger un fichier */
            public void loadFile(String fileName)
            {
                QFile file = new QFile(fileName);
                if (!file.open(new
    QFile.OpenMode(QFile.OpenModeFlag.ReadOnly,
    QFile.OpenModeFlag.Text))) {
                    QMessageBox.warning(this, tr("JEditor"),
    String.format(tr("Impossible de lire le fichier %1$s:\n%2$s."),
    fileName, file.errorString()));
                    return;
                }

                QTextStream in = new QTextStream(file);
                QApplication.setOverrideCursor(new
    QCursor(Qt.CursorShape.WaitCursor));
                textEdit.setPlainText(in.readAll());
                QApplication.restoreOverrideCursor();

                setCurrentFile(fileName);
                statusBar().showMessage(tr("Fichier Chargé"), 2000);
            }

            /* Sauvegarde du fichier */
            public boolean saveFile(String fileName)
            {
                QFile file = new QFile(fileName);
                if (!file.open(new
    QFile.OpenMode(QFile.OpenModeFlag.WriteOnly,
    QFile.OpenModeFlag.Text))) {
                    QMessageBox.warning(this, tr("JEditor"),
    String.format(tr("Impossible d'écrire dans le fichier
    %1$s:\n%2$s."), fileName, file.errorString()));
                    return false;
                }

                QTextStream out = new QTextStream(file);
                QApplication.setOverrideCursor(new
    QCursor(Qt.CursorShape.WaitCursor));
                out.writeString(textEdit.toPlainText());
                QApplication.restoreOverrideCursor();

                setCurrentFile(fileName);
                statusBar().showMessage(tr("Fichier sauvegardé"), 2000);
                file.close();
                return true;
            }

            /* On enregistre le nom du fichier ouvert comme nom du
    fichier courant */
            public void setCurrentFile(String fileName)
            {
                curFile = fileName;
                textEdit.document().setModified(false);
                setWindowModified(false);
```

```
                String shownName;
                if (curFile.length() == 0)
                    shownName = tr("sans_titre.txt");
                else
                    shownName = curFile;

                setWindowTitle(String.format(tr("%1$s[*] - %2$s"),
      shownName, tr("JEditor")));
            }

            /* Lancement de l'application */
            public static void main(String[] args) {
                  QApplication.initialize(args);

                  JEditor application = new JEditor();
                  application.show();

                  QApplication.exec();
              }
      }
```

Voilà le programme commenté.
Je pense que le programmeur chevronné que vous êtes comprendra assez facilement cet exemple.
On peut très facilement ajouter une fonction CTRL+Z.
C'est un petit exemple mais il vous montre déjà pas mal de petites choses.
L'étape suivante serait de pouvoir ouvrir plusieurs fichiers en même temps. (Petite piste : Les threads et l'implémentation de Clonable)
Vous pouvez noter aussi le déplacement géré par Qt des barres de menus, l'agrandissement de la fenêtre aussi.
Voilà, vous avez créé votre première application multi plates-formes en Java avec Qt Jambi.
Vous pouvez avoir différents rendus en utilisant cette commande :

Code : Console

```
java JEditor -style=motif
java JEditor -style=plastique
java JEditor -style=windows
java JEditor -style=cde
```

Pour voir l'application finie, allez ici.

TP : JViewer, un lecteur d'image

Dans ce chapitre, nous allons réaliser une application de lecture d'images.

Cahier des charges

Une image vaut plus qu'un long discours ...

Image utilisateur

Ici on ne voit pas la barre de menu, mais c'est le même principe que le TP précédent.
La difficulté réside dans la gestion des images.

Vous aurez besoin d'utiliser QLabel pour votre widget central.
Pour charger les images QPixmap.

Bon courage.

La solution

Voici ma proposition :

Code : Java

```
/*
 * JViewer.java
 * Version : 2
 *
 * Created on 2 novembre 2007
 * Modified on 13 novembre 2007
 */


/* Importation des espaces de nom nécessaires */
import com.trolltech.qt.core.*;

import com.trolltech.qt.gui.*;
import com.trolltech.qt.core.Qt.*;
import com.trolltech.qt.gui.QSizePolicy.Policy;

/* Définition de l'application et de ses attributs */
public class JViewer extends QMainWindow {
    private String curFile;         // Fichier actuellement ouvert
    private QPixmap pixImage;
    private QLabel widgetImage;     // Conteneur de l'image
    private QMenu fileMenu;         // Menu Fichier
    private QMenu helpMenu;         // Menu Aide
    private QToolBar fileToolBar;   // Barre d'outil Fichier
    private QAction openAct;        // Action Ouvrir
    private QAction sizeToFitAct;       // Action Ouvrir
    private QAction exitAct;        // Action Quitter
    private QAction aboutAct;       // Action A Propos de JEditor
    private QAction aboutQtAct;     // Action A propos de Qt Jambi
    private String rsrcPath = "classpath:images"; // Répertoire des images

    /* Définition du constructeur */
    public JViewer()
    {
        setSizePolicy(Policy.Expanding, Policy.Expanding);
        setMinimumSize(240, 160);

        QMenuBar menuBar = new QMenuBar();                  // On crée la barre d
        pixImage = new QPixmap(rsrcPath+"/JViewer.png");
        widgetImage = new QLabel(this);
        widgetImage.setSizePolicy(Policy.Expanding, Policy.Expanding);
        widgetImage.setAlignment(AlignmentFlag.AlignCenter);
        widgetImage.setPixmap(pixImage);
        resize();
```

```
        setMenuBar(menuBar);                                    // On ajoute la barre
        setCentralWidget(widgetImage);                          // On ajoute la zone
        /* On lance les méthodes de création des différents attributs de notre
        try {
              createActions();

          } catch (Exception e) {
              e.printStackTrace();
          }
          createMenus();
          createToolBars();
          createStatusBar();

          /* Le fichier de départ est un fichier vide */
          setCurrentFile("");
    }

    /* Méthode d'ouverture du fichier */
    public void open()
    {
      String fileName = QFileDialog.getOpenFileName(this);
      if (fileName.length() != 0)
        loadFile(fileName);
    }

    public void sizeToFit(){
      widgetImage.setPixmap(pixImage.scaled(widgetImage.size(),
                                AspectRatioMode.KeepAspectRatio,
                                TransformationMode.SmoothTransformation));
    }

    /* Fonction A propos de JEditor */
    public void about()
    {
        QMessageBox.about(this,
                          tr("A Propos de JViewer"),
                          tr("<b>JViewer<b> est une petite application "+
                            "réalisé par <a href='http://www.trunat.fr/'>Nati
    }

    /* Création des actions des menus et des toolbars */
    private void createActions()
    {
        /* Actions du menu Fichier */

        openAct = new QAction(new QIcon(rsrcPath + "/open.png"), tr("&Ouvrir.
        openAct.setShortcut(tr("Ctrl+O"));
        openAct.setStatusTip(tr("Ouvrir un fichier"));
        openAct.triggered.connect(this, "open()");

        sizeToFitAct = new QAction(new QIcon(rsrcPath + "/sizeToFit.tif"), tr
        sizeToFitAct.setShortcut(tr("Ctrl+A"));
        sizeToFitAct.setStatusTip(tr("Taille d'origine"));
        sizeToFitAct.triggered.connect(this, "resize()");

        exitAct = new QAction(new QIcon(rsrcPath + "/exit.png"), tr("Quitter'
        exitAct.setStatusTip(tr("Quitter l'application"));
        exitAct.triggered.connect(QApplication.instance(), "quit()");

        /* Action du menu Aide */
        aboutAct = new QAction(new QIcon(rsrcPath + "/about.png"), tr("A Prop
        aboutAct.setStatusTip(tr("A Propos de JEditor"));
        aboutAct.triggered.connect(this, "about()");
        aboutQtAct = new QAction(new QIcon(rsrcPath + "/qt-logo.png"), tr("A
        aboutQtAct.setStatusTip(tr("Show the Qt library's About box"));
        aboutQtAct.triggered.connect(QApplication.instance(), "aboutQt()");
    }

    /* Création des Menus */
    private void createMenus()
```

```
        {
            /* Menu fichier */
            fileMenu = menuBar().addMenu(tr("&Fichier"));
            fileMenu.addAction(openAct);
            fileMenu.addAction(sizeToFitAct);
            fileMenu.addSeparator();
            fileMenu.addAction(exitAct);

            /* Menu Aide */
            helpMenu = menuBar().addMenu(tr("&Aide"));
            helpMenu.addAction(aboutAct);
            helpMenu.addAction(aboutQtAct);
        }

        /* Création de la barre de menu */
        private void createToolBars()
        {
            fileToolBar = addToolBar(tr("Fichier"));
            fileToolBar.addAction(openAct);
            fileToolBar.addAction(sizeToFitAct);
            fileToolBar.addSeparator();
            fileToolBar.addAction(exitAct);
        }

        /* Création de la Barre de Status */
        private void createStatusBar()
        {
            statusBar().showMessage(tr("Pret"));
        }

        /* Charger un fichier */
        public void loadFile(String fileName)
        {
            pixImage = new QPixmap(fileName);
            if(pixImage == null){
              QMessageBox.warning(this, tr("JViewer"), String.format(tr("Erreur à
              return;
            }
            widgetImage.clear();
            widgetImage.setPixmap(pixImage);
            resize();
            setCurrentFile(fileName);

            statusBar().showMessage(tr("Image Chargée"), 2000);
        }

        /* On enregistre le nom du fichier ouvert comme nom du fichier courant */
        public void setCurrentFile(String fileName)
        {
            curFile = fileName;
            setWindowModified(false);

            String shownName;
            if (curFile.length() == 0)
                shownName = tr("Aucun fichier");
            else
                shownName = curFile;

            setWindowTitle(String.format(tr("%1$s[*] - %2$s"), shownName, tr("JVi
        }

        protected void paintEvent(QPaintEvent e) {
          sizeToFit();
        }

        protected void resize(){
          if(pixImage.size().subtract(new QSize(240,160)).isValid()){
            super.resize(pixImage.size().add(size().subtract(widgetImage.size()))
          }else{
            super.resize(new QSize(240,160));
```

```
                widgetImage.setPixmap(pixImage);
            }
        }

        /* Lancement de l'application */
        public static void main(String[] args) {
            QApplication.initialize(args);

            JViewer application = new JViewer();
            application.show();

            QApplication.exec();
        }
    }
```

Si vous en avez une différente, vous pouvez la poster en commentaire de ce tutoriel, on pourra comparer les fonctionnalités et si elle est mieux que la mienne, elle pourra la remplacer comme solution.
En conclusion, je dirai simplement que vous avez toutes les clefs en mains pour réussir, alors à vous de jouer.
Pour voir l'application finie, allez ici.

Si vous vous en êtes sortis avec les TP, c'est que vous êtes prêts pour faire vos propres applications.
N'oubliez pas la doc .
Il ne vous reste plus qu'à continuer ...