Exercice 25 - Mon premier projet Qt

Créer un **nouveau projet** de la manière suivante :

- Lancer QtCreator.
- Dans le menu, choisir `Fichier>Nouveau Fichier ou Projet`.
- Dans la fenêtre, choisir `Autre Projet` puis `Projet Qt vide` puis cliquer sur le bouton `Choisir...`.
- Appeler le projet `Exercice1`, choisir un emplacement pour le sauvegarder, puis cliquer sur le bouton `Suivant>` 2 fois, et enfin sur `Terminer`.
- Pour une version de `Qt` à partir de 5.0, ajoutez l'instruction `QT += widgets` sur la première ligne du fichier `.pro` de votre projet (il s'agit d'un fichier de configuration).
- Si votre projet nécessite l'utilisation de la norme C++11 (gcc), ajouter les instructions suivantes dans le fichier `.pro` :

```
QMAKE_CXXFLAGS = -std=c++11
QMAKE_LFLAGS = -std=c++11
```

Ajouter un **nouveau fichier** `main.cpp` de la manière suivante :

- Dans le menu, choisir `Fichier>Nouveau Fichier ou Projet`.
- Dans la fenêtre, choisir `C++` puis `Fichier source C++` puis cliquer sur le bouton `Choisir...`.
- Appeler le fichier `main`, puis cliquer sur le bouton `Suivant>` et enfin sur `Terminer`.

Recopier le code suivant dans le fichier `main.cpp` :

```
#include <QApplication>
int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    return app.exec();
}
```

main.cpp

Ce code représente le code minimal utilisant les possibilités de Qt pour une application GUI. Un objet `app` de type `QApplication` est créé (en lui retransmettant les éventuels arguments reçus en ligne de commande) et la méthode `exec()` est appliquée pour démarrer la boucle d'évènements permettant d'intéragir avec l'application. La méthode se chargera de renvoyer le résultat du programme.

Assurez-vous que le code compile correctement. Avant la compilation, il peut être nécessaire d'exécuter la commande `Compiler/ Exécuter qmake` pour que le fichier `.pro` de votre projet soit traité.

Question 1

Après avoir inclu le fichier d'entête `<QPushButton>`, créer un objet `QPushButton`, juste avant l'exécution de la méthode `app.exec()`, en utilisant le constructeur qui permet de l'initialiser avec le texte `Quitter` (voir la documentation de `QpushButton` sur http://qt-project.org/doc/qt-4.8/qpushbutton.html). Envoyer le message `show()` à cet objet. Compiler et exécuter le programme.

Question 2

A quoi sert la méthode `show()` ? Quelle est la nature de la méthode `show()` ? Dans quelle classe cette méthode est-elle définie ?

Question 3

Pour l'instant, cliquer sur le bouton n'a aucun effet visible. Faire en sorte que l'application s'arrête lorsque l'on clique sur ce bouton.
Vous pouvez consulter la ressource suivante pour quelques rappels sur les signaux et les slots :
https://openclassrooms.com/courses/programmez-avec-le-langage-c/les-signaux-et-les-slots-2

Question 4

Que se passe t-il si on ajoute un deuxième bouton initialisé avec le texte "coucou" et qu'on lui applique la méthode `show()` ?

Exercice 26 - Ma première fenêtre avec des trucs dessus

Créer un nouveau projet Qt et ajouter le code suivant :

```
#include <QApplication>
#include <QWidget>
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QWidget fenetre;
    fenetre.setFixedSize(200, 200);
    //...
    fenetre.show();
    return app.exec();
}
```

main.cpp

La fenêtre principale va maintenant être un objet `Qwidget` dont on aura fixé une taille de 200×200.

Question 1

Ajouter des instructions qui permettent de disposer sur cette fenêtre :

- trois objets `idl`, `titrel` et `textl` de type `QLabel` initialisés respectivement avec les chaînes de caractères `Identificateur`, `Titre`, et `Texte` et placés respectivement aux positions $(10, 10)$, $(10, 45)$, et $(10, 80)$;
- deux objets `id` et `titre` de type `QLineEdit` et de largeur 180 placés respectivement aux positions $(100, 10)$ et $(100, 45)$;
- un objet `text` de type `QTextEdit`, de taille 180×110 et placé à la position $(100, 80)$;
- et un objet `save` de type `QPushButton` initialisé avec le texte `"Sauver"` et de largeur 80 à la position $(200, 170)$.

Question 2

Recommencer la question 1, mais en utilisant des objets de types `QVBoxLayout` et `QHBoxLayout` pour positionner relativement les objets. Commentez les instructions qui permettent de fixer la taille des widgets.
Vous pouvez consulter la ressource suivante pour quelques explications sur les layouts :
https://openclassrooms.com/courses/programmez-avec-le-langage-c/positionner-ses-widgets-avec-les-layouts.

Question 3

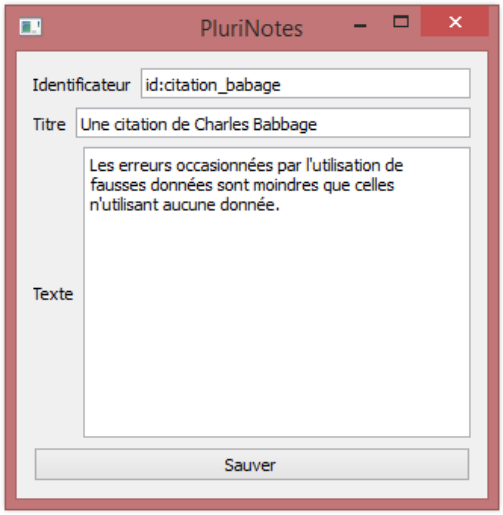
Ajouter l'instruction `QT += xml` dans le fichier `.pro` du projet et importer les classes de gestion d'articles fournies dans "Notes.h" et "Notes.cpp".
Utiliser l'instruction `QString filename = QFileDialog::getOpenFileName();` pour aller chercher le chemin du fichier de notes `notes.xml` avec une fenêtre de dialogue fichier.
Après avoir chargé le fichier de tâches `notes.xml` (avec la méthode `load` de `NotesManager`), obtenir de l'instance `NotesManager` un objet de type `Article` pour ensuite changer l'affichage des éléments `id`, `titre` et `text` avec les attributs de cet article.
Faire en sorte que l'utilisateur ne puisse pas modifier le texte de l'identificateur.

Attention: Les fichiers "Notes.h" et "Notes.cpp" contiennent des classes similaires à celles qui ont été développées dans les TD précédents mais qui ont été modifiées en utilisant des librairies de Qt (`QString`, `QTextStream`, `QFile`, `QtXml`) de façon à faciliter l'interface avec Qt et assurer une bonne gestion des accents dans votre future application.

Exercice 27 - Mon premier éditeur de notes

Question 1

Créer une classe `ArticleEditeur` qui hérite de la classe `QWidget` de manière à pouvoir voir les informations d'un objet `Article` avec une fenêtre similaire à l'Exercice 26 :



Un objet d'une telle classe pourra, par exemple, être utilisé avec le code suivant :

```
#include <QApplication>
#include <QString>
#include "ArticleEditeur.h"
int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
    QString fichier = QFileDialog::getOpenFileName();
    NotesManager& m=NotesManager::getManager();
    m.setFilename(fichier);
    m.load();
    Article& a=m.getArticle("id:citation_babage");
    ArticleEditeur fenetre(a);
    fenetre.show();
    return app.exec();
}
```

main.cpp

Consignes :

- Utiliser des attributs de type `QVBoxLayout*`, `QHBoxLayout*`, `QLineEditor*`, `QTextEditor*`, `QLabel*` et `QPushButton*`. Ces attributs contiendront les adresses des widgets utilisés sur la fenêtre et qui seront alloués dynamiquement dans le constructeur de la classe.
- Ajouter également un attribut `article` de type `Article*` qui contiendra l'adresse de l'article en cours d'édition.
- Définir un unique constructeur de la classe qui aura comme paramètre `article` de type `Article&` et un paramètre `parent` de type `QWidget*`.
- Ajouter la macro `Q_Object` qui permet de gérer les signaux et slots dans une classe.
- Il ne doit pas être possible de modifier un identificateur d'article.

Question 2

Faire en sorte que lorsque l'on clique sur le bouton `Sauver`, les modifications effectuées dans l'éditeur se répercutent sur l'objet `Article` correspondant. Afin d'informer l'utilisateur que son article a bien été enregistré, utiliser la méthode statique `QMessageBox::information` pour l'avertir. Après avoir quitter votre application, vérifier sur le fichier ressource correspondant que les modifications ont bien été prises en compte.

Question 3

Faire en sorte que le bouton `Sauver` soit initialement désactivé. Il ne devra s'activer que lorsque l'on édite le titre ou le texte de l'article.