Exercice 27 - 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 ne compile pas directement à cause du fichier d'entête `type_traits` (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.

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 :
http://www.siteduzero.com/informatique/tutoriels/programmez-avec-le-langage-c/le-principe-des-signaux-et-slots

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 28 - 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 :

- un objet `identificateur` de type `QLineEdit` et de largeur 180 à la position $(10, 10)$,
- un objet `titre` de type `QTextEdit`, de taille 180×110 à la position $(10, 45)$,
- et un objet `save` de type `QPushButton` initialisé avec le texte `"Sauver"` et de largeur 80 à la position $(10, 170)$.

Question 2

Recommencer la question 1, mais en utilisant un objet `couche` de type `QVBoxLayout` pour positionner relativement les objets.
Vous pouvez consulter la ressource suivante pour quelques explications sur les layouts :
http://www.siteduzero.com/informatique/tutoriels/programmez-avec-le-langage-c/l-architecture-des-classes-de-layout

Question 3

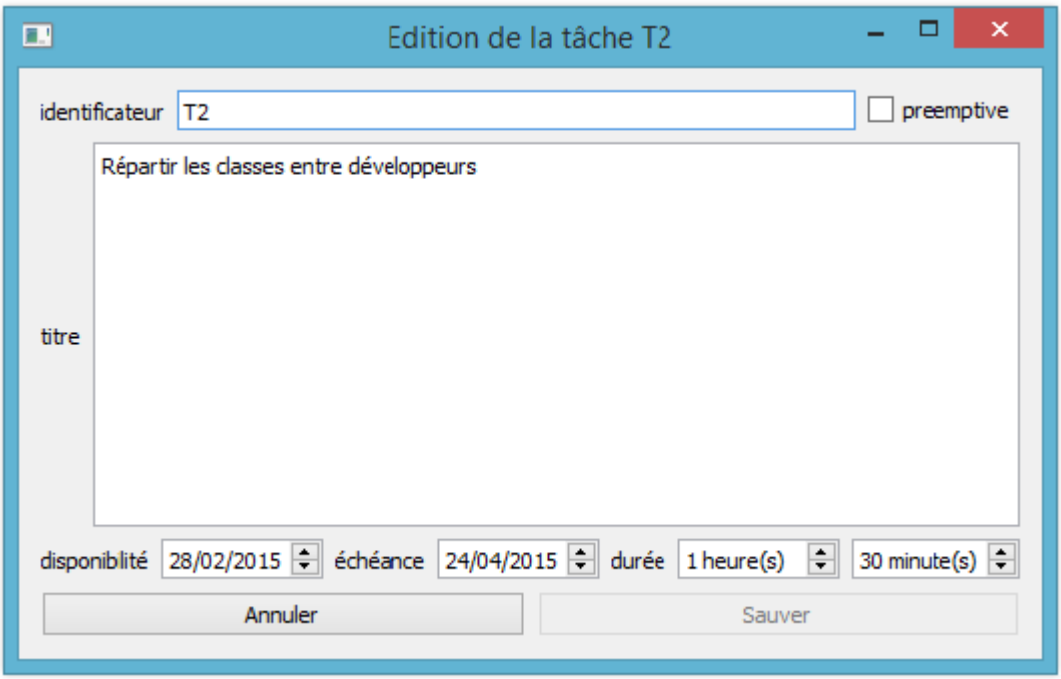
Ajouter l'instruction `QT += xml` dans le fichier `.pro` du projet et importer les classes de gestion de tâches fournies dans "Calendar.h" et "Calendar.cpp".
Utiliser l'instruction `QString chemin = QFileDialog::getOpenFileName();` pour aller chercher le chemin du fichier de tâches `taches.xml` avec une fenêtre de dialogue fichier.
Après avoir chargé le fichier de tâches `taches.xml` (avec la méthode `load` de `TacheManager`), obtenir de l'instance `TacheManager` un objet de type `Tache` pour ensuite changer l'affichage des éléments `identificateur` et `titre` avec respectivement l'identificateur et le titre de cette tâche.

Attention: Les fichiers "Calendar.h" et "Calendar.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 29 - Mon premier éditeur d'UV

Question 1

Créer une classe `TacheEditeur` qui hérite de la classe `QWidget` de manière à pouvoir voir les informations d'un objet `Tache` avec une fenêtre similaire à l'exemple suivant :



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

```
#include <QApplication>
#include <QFileDialog>
#include <QComboBox>
#include "Calendar.h"
#include "TacheEditeur.h"
int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    TacheManager& m=TacheManager::getInstance();
    QString chemin = QFileDialog::getOpenFileName();
    m.load(chemin);
    Tache& t=m.getTache("T2");
    TacheEditeur fenetre(t);
    fenetre.show();
    return app.exec();
}
```

main.cpp

Consignes :

- Utiliser des attributs de type `QVBoxLayout*`, `QHBoxLayout*`, `QLineEdit*`, `QTextEdit*`, `QSpinBox*`, `QDateEdit*`, `QLabel*`, `QCheckBox*` 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 `tache` de type `Tache&` qui référencera la tâche en cours d'édition.
- Définir un unique constructeur de la classe qui aura comme paramètre `tacheToEdit` de type `Tache&` et un paramètre `parent` de type `QWidget*`.
- Ajouter la macro `Q_Object` qui permet de gérer les signaux et slots dans une classe.

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 `tache` correspondant. La sauvegarde ne pourra pas être effectuée si l'utilisateur essaye de changer l'identificateur de la tâche par un identificteur qui existe déjà pour une autre tâche. On pourra informer l'utilisateur de cette erreur en utilisant la méthode statique `QMessageBox::warning`. Afin d'informer l'utilisateur que la tâche a bien été sauvegardée, utiliser la méthode statique `QMessageBox::information` pour l'avertir. Après avoir quitté l'application, vérifier sur le fichier ressource correspondant que les modifications ont bien été prises en compte. Faire aussi en sorte que la fenêtre d'édition se ferme lorsque l'utilisateur clique sur le bouton `Annuler`. Dans ce cas, les modifications ne sont pas reportées sur la tâche correspondante.

Question 3

Faire en sorte que le bouton `Sauver` soit initialement désactivé. Il ne devra s'activer que lorsque l'on édite un des éléments de la tâche. Faire en sorte que la date de disponibilité soit toujours cohérente avec la date d'échéance.