Cours Qt

par Thierry Vaira © v.1.00

Sommaire

A Présentation de Qt

Qt est une **bibliothèque logicielle orientée objet** (API) développée en **C++** par Qt Development Frameworks, filiale de Digia.

Une API (*Application Programming Interface*) est une interface de programmation pour les application et donc les développeurs. C'est un ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade pour accéder aux services fournis.

Qt est une plateforme de **développement d'interfaces graphiques GUI** (*Graphical User Interface*) fournie à l'origine par la société norvégienne Troll Tech, rachetée par Nokia en février 2008 puis cédée intégralement en 2012 à Digia (www.qt.io).

Qt fournit également un ensemble de classes décrivant des éléments non graphiques : accès aux données (fichier, base de données), connexions réseaux (*socket*), gestion du multitâche (*thread*), XML, etc.

Qt permet la **portabilité des applications** (qui n'utilisent que ses composants) par simple **recompilation du code source**.

Les environnements supportés sont les **Unix** (dont **Linux**), **Windows** et **Mac OS X**.

De plus en plus de développeurs utilisent Qt, y compris parmi de grandes entreprises. On peut notamment citer : Google, Adobe Systems, Asus, Samsung, Philips, ou encore la NASA et bien évidemment Nokia. Qt est notamment connu pour être la bibliothèque sur laquelle repose l'environnement graphique KDE, l'un des environnements de bureau les plus utilisés dans le monde Linux.

Lire aussi les licences Qt : www.qt.io/licensing/.

Question 1 (1 point)
Peut-on écrire en Qt une application graphique qui exploite les donnnées en provenance d'une base de données MySQL et qui exporte celles-ci au format XML ?
☐ oui ☐ non

Question 2 (1 point)
Un code source écrit en Qt pourra-t-il produire une application pour Linux et Windows ?
☐ oui ☐ non

B Notions de base

B.1 Programmation évènementielle

La programmation événementielle est une **programmation basée sur les événements**.

Elle s'oppose à la programmation **séquentielle** (une suite d'instructions, d'actions, ...).

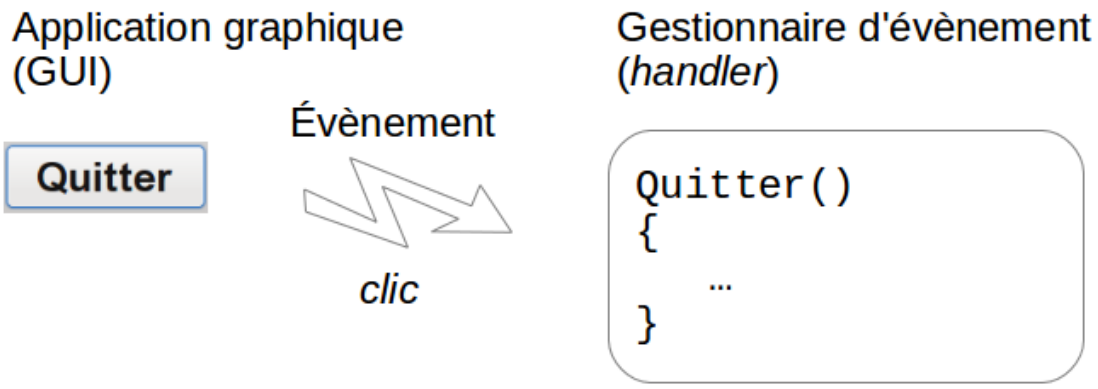
Le programme sera principalement défini par ses réactions aux différents événements qui peuvent se produire, c'est-à-dire des changements d'état, par exemple l'incrémentation d'une liste, un mouvement de souris ou de clavier etc.

Elle est particulièrement mise en œuvre dans le domaine des interfaces graphiques.

La programmation événementielle est architecturée autour d'une **boucle principale** fournie et divisée en deux sections : la première section détecte les événements, la seconde les gère.

Pour chaque évènement à gérer, il faut lui associer une action à réaliser (le code d'une fonction ou méthode) : c'est le **gestionnaire d'évènement (*handler*)**.

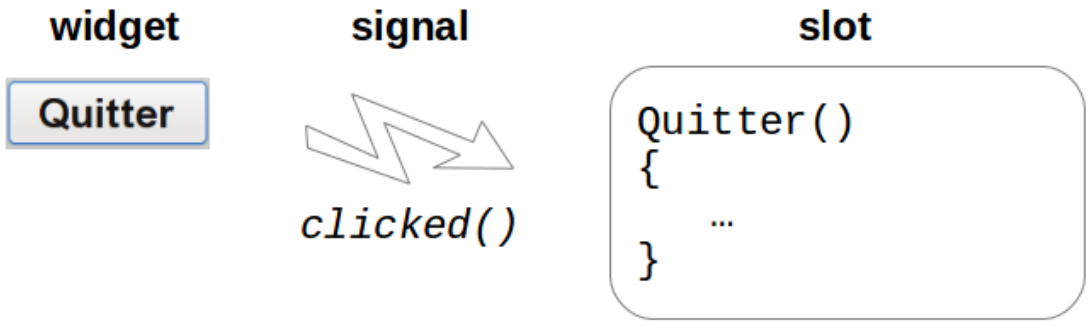
Ensuite, à chaque fois que l'évènement sera détecté par la boucle d'évènement, le gestionnaire d'évènement sera alors exécuté.



La programmation évènementielle des applications Qt est basée sur un **mécanisme appelé *signal/slot*** :

- un ***signal*** est émis lorsqu'un événement particulier se produit. Les classes de Qt possèdent de nombreux signaux prédéfinis mais vous pouvez aussi hériter de ces classes et leur ajouter vos propres signaux.
- un ***slot*** est une fonction qui va être appelée en réponse à un signal particulier. De même, les classes de Qt possèdent de nombreux slots prédéfinis, mais il est très courant d'hériter de ces classes et de créer ses propres slots afin de gérer les signaux qui vous intéressent.

L'association d'un signal à un slot est réalisée par une connexion (`connect()`).



Question 3 (1 point)
Parmi ces propositions, identifier celles qui sont des évènements ?
☐ un déplacement de la souris ☐ un appui sur une touche ☐ afficher une image
☐ se connecter à une base de données

Question 4 (1 point)
Parmi ces propositions, identifier celles qui sont des gestionnaire d'évènements (*handler*) ?
☐ un déplacement de la souris ☐ un appui sur une touche ☐ afficher une image
☐ se connecter à une base de données

B.2 Structure générale de Qt

L'API Qt est constituée de classes aux noms préfixés par un **Q** et dont chaque mot commence par une majuscule (`QLineEdit`, `QLabel`, ...).

B.2.1 La classe `QObject`

L'ensemble des classes est basé sur l'**héritage**. La classe `QObject` est la **classe mère de toutes les classes Qt**.

En dérivant de `QObject`, un certain nombre de spécificités Qt sont hérités, notamment :
- le mécanisme signal/slot de communication entre objets
- une gestion simplifiée de la mémoire

Les objets Qt (ceux héritant de `QObject`) peuvent s'organiser d'eux-mêmes sous forme d'**arbre d'objets**. Ainsi, lorsqu'une classe est instanciée, on peut lui définir un **objet parent**. Ceci permet entre autres :
- lorsque le parent est détruit (`delete`), tous ces enfant le seront aussi
- lorsque le parent est affiché (`show()`), tous ces enfant le seront aussi
- lorsque le parent est déplacé (`show()`), tous ces enfant le seront aussi
- ...

Pour bénéficier des spécificités de Qt, il faudra **hériter** de `QObject` ou d'une classe fille de `QObject` :

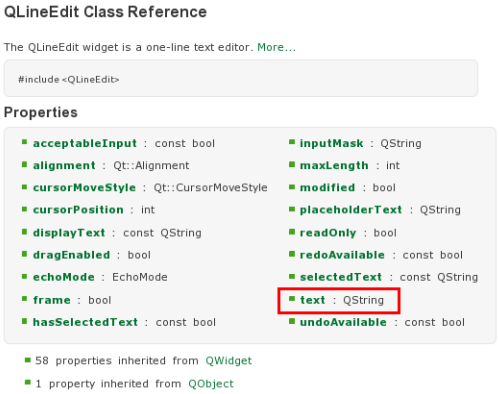
```
class MaClasse : public QObject
{
   Q_OBJECT

   public:
      MaClasse( QObject *parent=0 ) : QObject(parent) {}

   signals:
      void send( int ); // un signal

   public slots:
      void receive( int ); // un slot
};
```

Les spécificités Qt nécessitent l'utilisation du `moc`, un outil fourni par Qt, pour obtenir du code 100% C++.

Un objet Qt peut avoir des **propriétés**. Toutes les propriétés sont des attributs de la classe que l'on peut lire et éventuellement modifier.

Qt suit cette convention pour le nom des accesseurs :
- `propriete()` : c'est la méthode qui permet de lire la propriété
- `setPropriete()` : c'est la méthode qui permet de modifier la propriété

À partir de la documentation :



On pourra écrire :

```
#include <QLineEdit>

...

QLineEdit myLineEdit; // une zone de saisie

// Pour lire la chaîne de caractères saisie dans un QLineEdit
QString text = myLineEdit.text();
```

```
// Pour modifier la chaîne de caractères d'un QLineEdit
myLineEdit.setText("mon texte");
```

Pour connaître l'ensemble des classes, méthodes et propriétés de Qt, il faut consulter la documentation en ligne : doc.qt.io.

B.2.2 Les modules

Depuis, **Qt4** sépare sa bibliothèque en **modules** :

- QtCore : pour les fonctionnalités non graphiques utilisées par les autres modules ;
- QtGui : pour les composants graphiques (qt4), maintenant QtWidgets (qt5) ;
- QtNetwork : pour la programmation réseau ;
- QtSql : pour l'utilisation de base de données SQL ;
- QtXml : pour la manipulation et la génération de fichiers XML ;
- et de nombreux autres modules, etc.

Il faut activer un module dans un projet Qt pour pouvoir accéder aux classes qu'il regroupe (cf. la variable `QT` d'un fichier `.pro`).

B.2.3 La classe `QApplication`

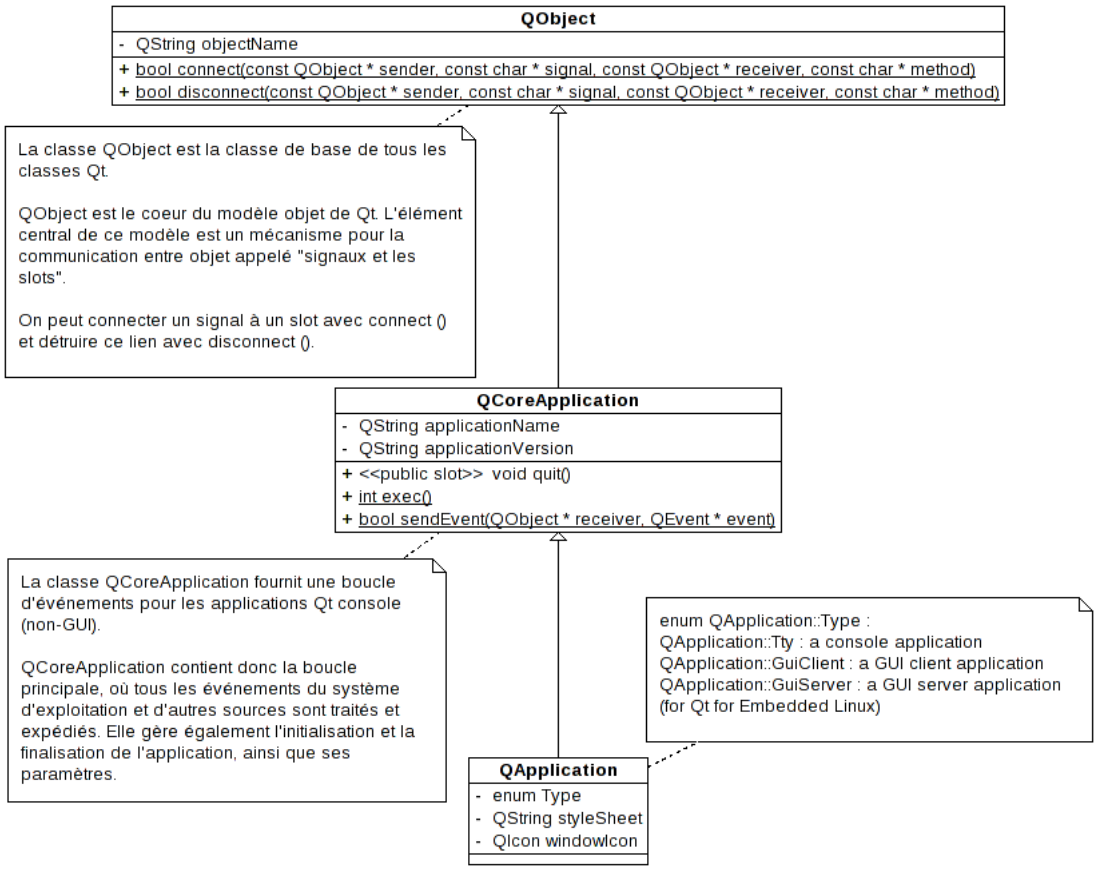
La classe `QApplication` (qui hérite de `QCoreApplication`) fournit une **boucle principale d'événements** pour les applications Qt : tous les événements du système sont traités et expédiés. Elle gère également l'initialisation et la finalisation de l'application, ainsi que ses paramètres.

L'instance de `QApplication` doit être créée avant tout objet graphique.

```
#include <QApplication>

int main(int argc, char **argv)
{
   QApplication app(argc, argv);

   int ret;

   ret = app.exec(); // exécute la boucle principale d'évènement

   return ret;
}
```

La méthode `exec()` exécute une boucle d'attente des événements jusqu'à la fermeture du dernier objet de l'application. Pour l'instant, cela donne un boucle infinie !

Diagramme de classes partiel :



Les méthodes soulignées sont des méthodes statiques.

Question 5 (1 point)
Quelle est la classe de base de toutes les autres classes de Qt ?
☐ `QClass` ☐ `QApplication` ☐ `QObject` ☐ `Qt` ☐ `QBase`

Question 6 (1 point)
La classe `Window` est-elle une classe de Qt ?
☐ oui ☐ non

Question 7 (1 point)
Quelle est la classe qui fournit la boucle d'évènement ?
☐ `QApplication` ☐ `QObject`

Question 8 (1 point)
Quelle est la classe qui fournit le mécanisme signal/slot ?
☐ `QApplication` ☐ `QObject`

Question 9 (1 point)
Quelle est le nom de la classe que toute application Qt doit instancier ?
☐ `QApplication` ☐ `QObject`

Question 10 (1 point)
Quelle(s) notion(s) n'existe(nt) pas normalement en C++ ?
☐ méthode statique ☐ *slot* ☐ *signal* ☐ héritage

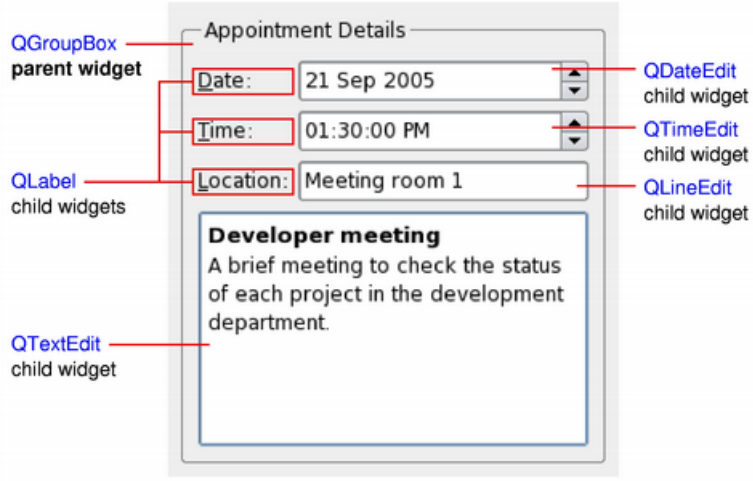
Question 11 (1 point)
Si une propriété d'une classe Qt s'appelle `value`, comment puis-je la modifier ?
☐ En appelant la méthode `setValue()` ☐ En appelant la méthode `setvalue()`
☐ En appelant la méthode `value()` ☐ On ne peut pas, il faut faire un héritage de cette classe

B.3 Élément graphique (*widget*)

Avec Qt, les éléments (ou composants) graphiques prédéfinis sont appelés des **widgets** (pour *windows gadgets*).

Les *widgets* sont les éléments principaux de la création d'interfaces utilisateur avec Qt.

Les *widgets* peuvent afficher des données et des informations sur un état, recevoir des actions de l'utilisateur et agir comme un conteneur pour d'autres *widgets* qui doivent être regroupés.



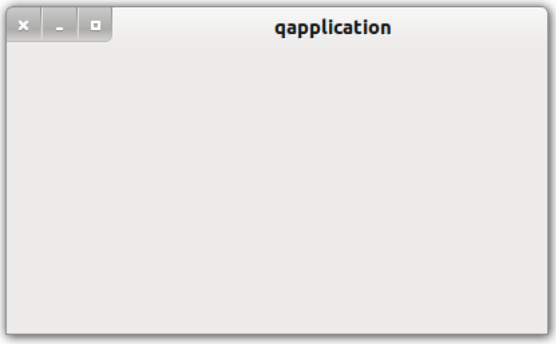
La classe `QWidget` fournit la capacité de base d'**affichage à l'écran** et de **gestion des événements**. elle est la classe mère de toutes les classes servant à réaliser des interfaces graphiques.

Tous les éléments graphiques que Qt fournit sont hérités de `QWidget` ou sont utilisés avec une classe fille de `QWidget`.

Les widgets :
- sont crées "cachés"
- sont capable de se "peindre"
- sont capable de recevoir les évènements souris, clavier
- sont tous rectangulaires
- sont initialisés par défaut en coordonnées 0,0
- sont ordonnés suivant l'axe z (gestion de la profondeur)
- peuvent avoir un widget parent et des widgets enfants

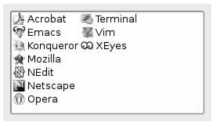
```
#include <QApplication>
#include <QWidget>

int main(int argc, char **argv)
{
   QApplication app(argc, argv);
   QWidget w; // un objet widget qui n'a pas de parent

   // un widget est crée caché, il faut donc l'afficher
   w.show(); // on obtient une fenêtre "vide"

   // on exécute la boucle principale d'évènements
   int ret = app.exec();

   // lorsqu'on ferme la fenêtre, on sort de la boucle
   // et on quitte l'application
   return ret;
}
```

On obtient :

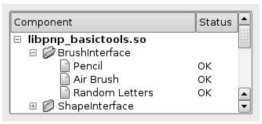


Un *widget* qui n'est pas intégré dans un *widget* parent est appelé une **fenêtre**.

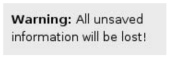
Qt fournit des widgets prédéfinis permettant de composer ses propres applications graphiques :



QListView(liste)

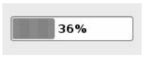


QTreeView



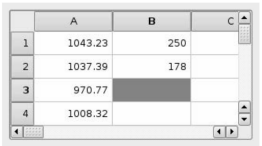
QLabel(texte)

QLCDNumber



QProgressBar

QListView(icônes)



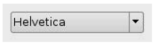
QTableView


QLabel (image)

QTextBrowser

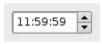
QSpinBox

QDoubleSpinBox



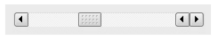
QComboBox

QDateEdit

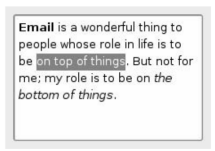


QTimeEdit

QDateTimeEdit


QScrollBar

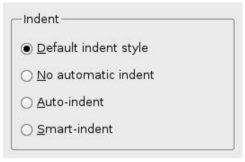
QSlider



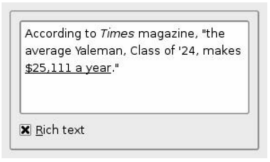
QTextEdit

QLineEdit

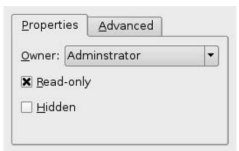

QDial



QGroupBox



QFrame

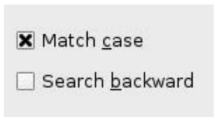


QTabWidget

QToolBox

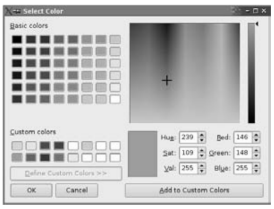
QPushButton


QToolButton



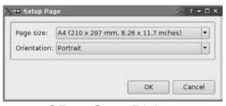
QCheckBox

QRadioButton

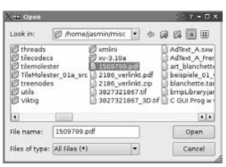


QColorDialog

QFontDialog



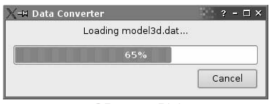
QPageSetupDialog



QFileDialog

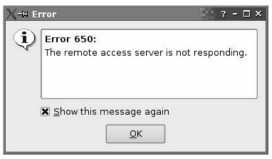
QPrintDialog

QInputDialog



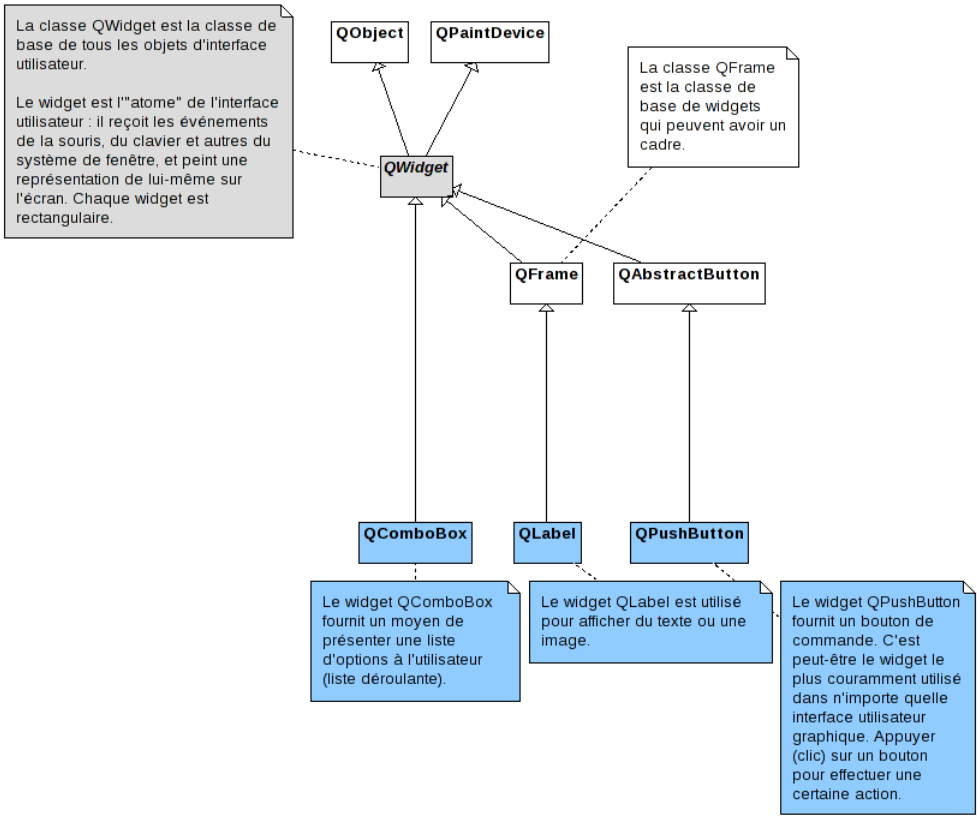
QProgressDialog

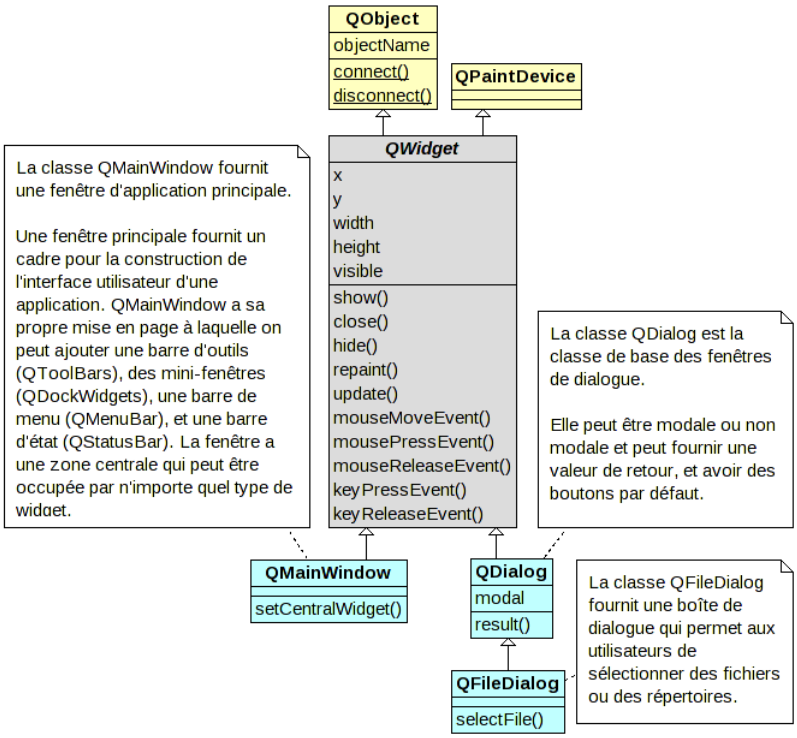
QMessageBox



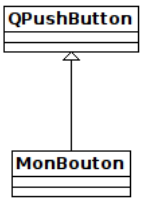
QErrorMessage

Diagrammes de classes partiels :

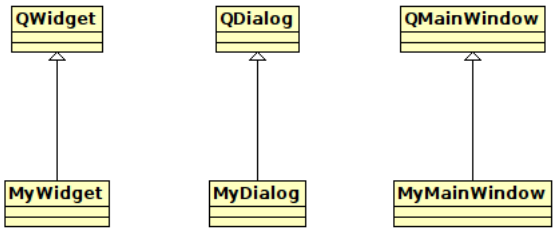




La création de ***widgets* personnalisés** est faite **en héritant** de `QWidget` ou une classe fille :

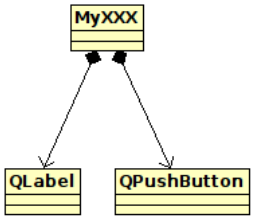


La création de **fenêtres personnalisées** est faite **en héritant** de `QWidget`, `QDialog` ou `QMainWindow` :



La classe Qt dédiée au fenêtre principale est bien évidemment `QMainWindow`.

Ensuite, on **compose** sa fenêtre personnalisée en y intégrant des *widgets* :

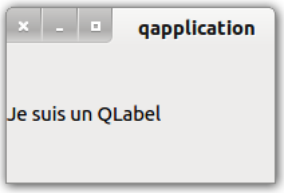


Généralement, cela est réalisé dans le **constructeur** de sa classe "Fenêtre" (ici `MyMainWindow`) :

```
// mymainwindow.h
#include <QMainWindow>
#include <QLabel>

// MA classe fenêtre principale
class MyMainWindow : public QMainWindow
{
   Q_OBJECT

   public:
      MyMainWindow( QWidget *parent=0 ) : QMainWindow(parent)
      {
          // on instancie un QLabel en lui indiquant son parent (this => moi)
          label = new QLabel("Je suis un QLabel", this);
```

```
        // etc ...

        // on fixe le widget QLabel au centre de la fenêtre
        setCentralWidget(label);
    }
    ~MyMainWindow() {}

  private:
    QLabel *label;

  signals:

  public slots:
};


// main.cpp
#include <QApplication>
#include "mymainwindow.h"

int main(int argc, char **argv)
{
  QApplication app(argc, argv); // mon objet application
  MyMainWindow w; // mon objet fenêtre

  w.show(); // affichage

  return app.exec(); // boucle
}
```

On obtient :



Question 12 (1 point)
Qu'est-ce qu'un *widget* qui n'a pas de parent ?
☐ Une belle erreur de compilation ☐ Une fenêtre
☐ Un plantage potentiel de l'ordinateur à l'exécution

Question 13 (1 point)
Quelle méthode de la classe `QMainWindow` permet de provoquer l'affichage de la fenêtre ?
☐ `open()` ☐ `show()` ☐ `run()` ☐ `display()`

Question 14 (1 point)
Laquelle de ces classes ne permet pas de s'afficher comme une fenêtre ?
☐ `QObject` ☐ `QWidget` ☐ `QDialog` ☐ `QMainWindow`

Question 15 (1 point)
Est-ce qu'un objet `QComboBox` est un *widget* ?
☐ `oui` ☐ `non`

Question 16 (1 point)
De quelle classe hérite `QDialog` ?
☐ `QWidget` ☐ `QApplication` ☐ `QMainWindow`

Question 17 (1 point)
Qu'est-ce qu'une fenêtre modale ?
☐ La dernière fenêtre qui s'affiche avant la fin du programme
☐ Une fenêtre qui empêche l'utilisation de toutes les autres tant qu'elle est ouverte
☐ Une fenêtre qui n'empêche pas l'utilisation de toutes les autres tant qu'elle est ouverte

Question 18 (1 point)
Est-ce qu'une `QMainWindow` doit forcément comporter une barre de menu ?
☐ Oui ☐ Non

Question 19 (1 point)
Quelle méthode de la classe `QMainWindow` permet d'indiquer le *widget* qui sera affiché au centre de la fenêtre ?
☐ `addCentralWidget()` ☐ `setMainWidget()` ☐ `setCentralWidget()`

B.4 Mécanisme *signal/slot*

Ensemble, les **signaux** et les **slots** forment un **mécanisme de communication entre objets**.

Un **signal** est émis lorsqu'un événement particulier se produit. Les *widgets* de Qt possèdent de nombreux signaux prédéfinis mais vous pouvez aussi hériter de ces classes et leur ajouter vos propres signaux.

Un **slot** est une méthode qui va être appelée en réponse à un signal particulier. Les *widgets* de Qt possèdent de nombreux slots prédéfinis, mais il est très courant d'hériter de ces *widgets* et de créer ses propres slots afin de gérer les signaux qui vous intéressent.


Toutes les classes qui héritent de `QObject` ou d'une de ses sous-classes (par exemple, `QWidget`) peuvent contenir des signaux et des slots. Il faut aussi ajouter la macro `Q_OBJECT` au sein de cette classe.

Les **signaux et les slots sont faiblement couplés** : une classe qui émet un signal ne sait pas (et ne se soucie pas de) quels slots vont recevoir ce signal. De la même façon, un objet interceptant un signal ne sait pas quel objet a émis le signal.

Une **connexion signal / slot** doit être réalisée par la méthode `connect()` :

```
bool QObject::connect( const QObject * sender, const char * signal,
                       const QObject * receiver, const char * method,
                       Qt::ConnectionType type = Qt::AutoConnection ) const
```

Une connexion signal/slot peut être supprimée par la méthode `disconnect()`.

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char **argv)
{
   QApplication app(argc, argv);
   QPushButton bouton("Quitter");

   // on connecte le signal clicked() de l'objet bouton
   // au slot quit() de l'objet app
   QObject::connect(&bouton, SIGNAL(clicked()), qApp, SLOT(quit()));

   bouton.show();

   int ret = app.exec();

   return ret;
}
```

qApp est un pointeur global qui contient toujours l'adresse de l'objet `QApplication`. Les applications doivent se terminer proprement en appelant `QApplication::quit()`. Cette méthode est appelée automatiquement lors de la fermeture du dernier widget.

Lorsqu'on clique sur le bouton "Quitter", on sort de l'application :

Il est possible de **créer ses propres signaux et slots**.

Pour déclarer un **signal personnalisé**, on utilise le mot clé `signals` dans la déclaration de la classe et il faut savoir qu'un signal n'a :
- pas de valeur de retour (donc `void`) et
- pas de définition de la méthode (donc pas de corps `{}`)

Pour émettre un signal, on utilise la méthode `emit` :

```
emit nomDuSignal( parametreDuSignal );
```

Les **slots personnalisés** se déclarent et définissent comme des méthodes `private`, `protected` ou `public`. On utilise le mot clé `slots` dans la déclaration de la classe. Les slots étant des méthodes normales, ils peuvent être appelés directement comme toute méthode.

```
#include <QObject>
#include <QDebug>

class MaClasse : public QObject
{
   Q_OBJECT

   private:
    int numero;

   public:
      MaClasse( int numero=0, QObject *parent=0 ) : QObject(parent), numero(
          numero) {}

      void emettre()
      {
          emit send(numero); // envoie le signal send avec la valeur de la
              variable numero
      }

   signals:
      void send( int ); // un signal personnalisé

   public slots:
      void receive( int valeur ) // un slot personnalisé
      {
          qDebug() << "signal recu " << valeur;
      }
};
```

La classe QDebug fournit un flux de sortie pour déboguer les informations. Elles peuvent être éliminées de l'exécutable en définissant `QT_NO_DEBUG_OUTPUT`. Qt fournit quatres fonctions globales pour l'écriture de texte d'avertissement et de débogage : `qDebug()` pour les sorties de débogage personnalisées, `qWarning()` pour des avertissements et des erreurs récupérables dans l'application, `qCritical()` des erreurs critiques et les erreurs systèmes et `qFatal()` pour tracer des erreurs fatales juste avant l'arrêt de l'application.

On pourra alors faire :

```
MaClasse monObjet1(1), monObjet2(2);

// le signal et le slot doivent êtres compatibles (même signature)
QObject::connect(&monObjet1, SIGNAL(send(int)), &monObjet2, SLOT(receive(int))
    );

monObjet1.emettre();

// la méthode emettre() de l'objet1 enverra le signal send avec la valeur 1,
// ce qui déclenchera l'exécution du slot receive de l'objet2 et
// cela affichera "signal recu 1"
```

Le mécanisme des signaux et slots fournit un contrôle des types : la signature d'un signal doit correspondre à la signature du slot récepteur (en réalité, un slot peut avoir une signature plus courte que celle du signal qu'il reçoit car il peut ignorer les arguments en trop).

Question 20 (1 point)
Quelle(s) condition(s) doi(ven)t être remplie(s) pour que l'on puisse définir un signal ou un *slot* personnalisé dans une classe ?
☐ La classe doit contenir la macro `Q_OBJECT`
☐ La classe doit dériver de `QObject`
☐ Les deux

Question 21 (1 point)
Comment émet-on un signal ?
☐ `monSignal;` ☐ `monSignal();` ☐ `signal(monSignal());` ☐ `emit monSignal();`

Question 22 (1 point)
Le signal `envoi()` est-il compatible avec le *slot* `reception(int)` ?
☐ Oui ☐ Non

Question 23 (1 point)
Peut-on connecter un signal à plusieurs *slots* ?
☐ Oui ☐ Non

Question 24 (1 point)
Peut-on connecter plusieurs signaux à un même *slot* ?
☐ Oui ☐ Non

B.5 Projet Qt

Un **projet Qt** est défini par un fichier d'extension `.pro` décrivant la liste des fichiers sources, les dépendances, les paramètres passés au compilateur, etc... Le fichier de projet est fait pour être très facilement éditable par un développeur. Il consiste en une série d'affectations de variables.

Pour contrôler ses propres utilitaires (`moc`, `uic`, ...), Qt fournit un moteur de production spécifique : le programme `qmake`.

`qmake` prend en entrée un fichier de projet `.pro` et génère un fichier de fabrication spécifique à la plateforme. Ainsi, sous les systèmes UNIX/Linux, `qmake` produira un `Makefile`.

Exemple d'une génération basique d'une application :

```
// on crée un répertoire et on se déplace à l'intérieur
$ mkdir exemple
$ cd exemple

// on édite un simple fichier C++ main.cpp (avec vim par exemple)
$ cat main.cpp
#include <QApplication>
#include <QPushButton>

int main(int argc, char **argv)
{
   QApplication app(argc, argv);
   QPushButton bouton("Quitter");

   QObject::connect(&bouton, SIGNAL(clicked()), qApp, SLOT(quit()));

   bouton.show();

   int ret = app.exec();

   return ret;
}

// on génère le fichier de projet Qt .pro
$ qmake -project
$ ls
exemple.pro main.cpp

// le fichier .pro décrit le contenu du projet en une série d'affectations de
    variables
$ cat exemple.pro
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
```

```
# Input
HEADERS +=
SOURCES += main.cpp

// on génère le fichier Makefile à partir du fichier .pro
$ qmake
$ ls
exemple.pro main.cpp Makefile

// on fabrique l'application
$ make
g++ -c -m64 -pipe -O2 -Wall -W -D_REENTRANT -DQT_WEBKIT -DQT_NO_DEBUG -
    DQT_GUI_LIB -DQT_CORE_LIB -DQT_SHARED -I/usr/share/qt4/mkspecs/linux-g++-64
     -I. -I/usr/include/qt4/QtCore -I/usr/include/qt4/QtGui -I/usr/include/qt4
    -I. -I. -o main.o main.cpp
g++ -m64 -Wl,-O1 -o exemple main.o -L/usr/lib/x86_64-linux-gnu -lQtGui -
    lQtCore -lpthread
$ ls
exemple exemple.pro main.cpp main.o Makefile

// on exécute l'application
$ ./exemple
$
```

À chaque fois que l'on modifie le fichier `.pro`, il faudra exécuter à nouveau la commande `qmake` pour que celle-ci mette à jour le fichier `Makefile`. D'autre part, le fichier `Makefile` est toujours spécifique à la plateforme. Si vous changez de plateforme (Linux → Windows), il vous suffira d'exécuter à nouveau la commande `qmake` pour générer un fichier `Makefile` adapté à votre système.

Ici, le nom initial du répertoire détermine le nom du projet et donc de l'exécutable qui sera produit. On peut choisir un nom d'exécutable différent en l'affectant à la variable `TARGET`.

Le principe de fonctionnement du fichier `.pro` est donc très simple. Par exemple, la variable permettant d'indiquer les **modules** Qt à intégrer pour l'application se nomme `QT`.

Exemples :

```
QT += sql xml pour activer les modules SQL et XML
QT -= gui pour désactiver le module gui
```

Sous Qt4, les modules `core` et `gui` (dont les *widgets*) sont activés par défaut.

D'autres variables utiles :

```
# Répertoires (cf. shadow build)
DESTDIR    = bin            # pour l'exécutable
OBJECTS_DIR = build         # pour les fichiers objets (.o, .obj)
MOC_DIR    = build          # pour les fichiers générés par moc
```

```
UI_DIR     = build           # pour les fichiers générés par uic

FORMS      = WiimoteIHM.ui   # fiche(s) de Qt Designer

# Bibliothèques (multiplateforme)
linux*:LIBS += -lm -lwiiuse
macx:LIBS  += -framework IOKit -framework CoreFoundation
win32:LIBS += -lsetupapi -ladvapi32 -luser32

INCLUDEPATH += .             # -> -I.
DEFINES    += DEBUG          # -> -DDEBUG
```

Lire le manuel de `qmake` : doc.qt.io/qt-5/qmake-manual.html

Question 25 (1 point)
Dans quel ordre les commandes doivent-elles être tapées pour fabriquer une application Qt ? (Notez que `make` et `mingw32-make` sont équivalents)
☐ `qmake -project` ; `make` ; `qmake`
☐ `make` ; `qmake -project` ; `make`
☐ `qmake -project` ; `qmake` ; `make`

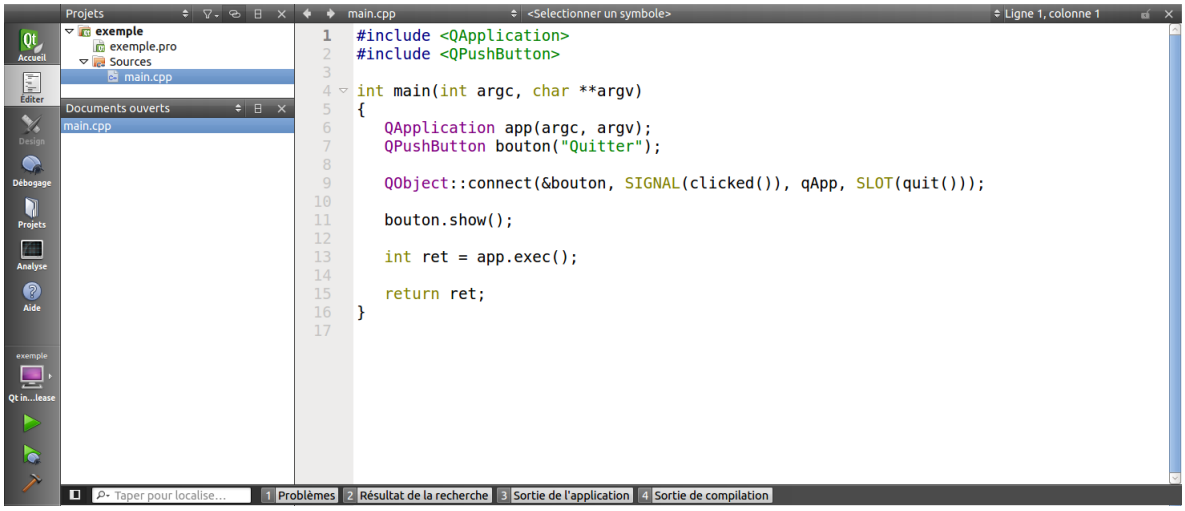
Question 26 (1 point)
Quel fichier est généré par la commande `qmake` ?
☐ Un fichier `.pro` ☐ Un fichier `Makefile` ☐ Un fichier `.cpp` ☐ Un fichier `.conf`

B.6 Environnement de Développement Intégré (EDI)

Qt Creator est l'environnement de développement intégré dédié à Qt et facilite la gestion d'un projet Qt. Son éditeur de texte offre les principales fonctions que sont la coloration syntaxique, le complètement, l'indentation, etc...

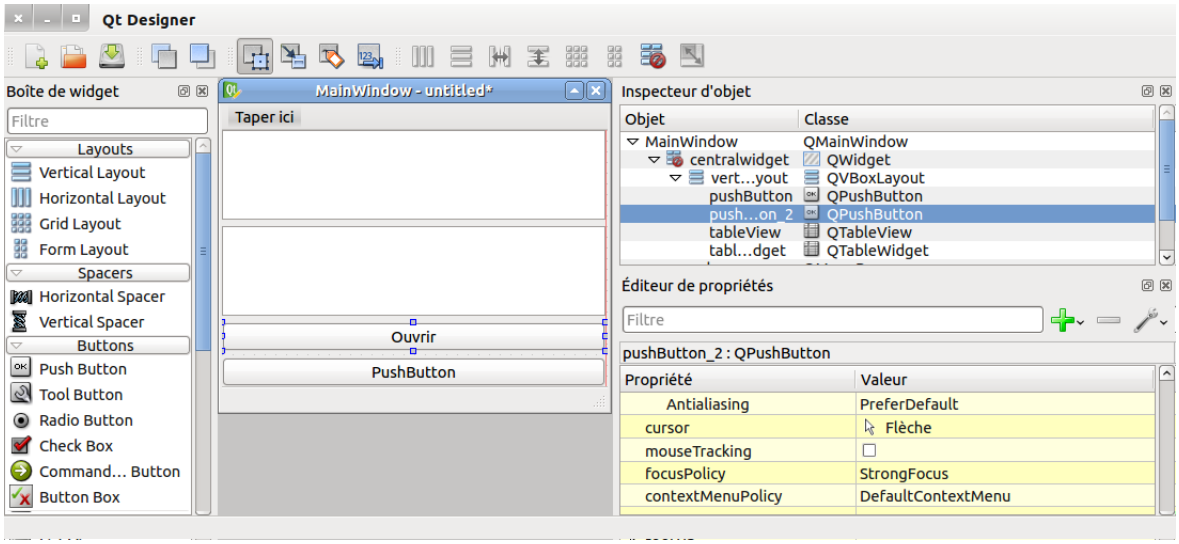
Qt Creator intègre en son sein les outils Qt Designer et Qt Assistant. Il intègre aussi un **mode débuggage** et la documentation.

Qt Creator lit en entrée un fichier de projet `.pro`. Il fournit aussi des assistants (*wizard*) pour créer des projets types.



Même si Qt Creator est présenté comme l'environnement de développement de référence pour Qt, il existe des modules Qt pour les environnements de développement Eclipse et Visual Studio. Il existe d'autres EDI dédiés à Qt et développés indépendamment de Nokia, comme QDevelop et Monkey Studio.

Qt Designer est un logiciel qui permet de créer des interfaces graphiques Qt dans un environnement convivial. L'utilisateur, par glisser-déposer, place les composants d'interface graphique et y règle leurs propriétés facilement. Les fichiers d'interface graphique sont formatés en XML et portent l'extension `.ui`. Lors de la compilation, un fichier d'interface graphique est converti en classe C++ par l'utilitaire `uic`, fourni par Qt.



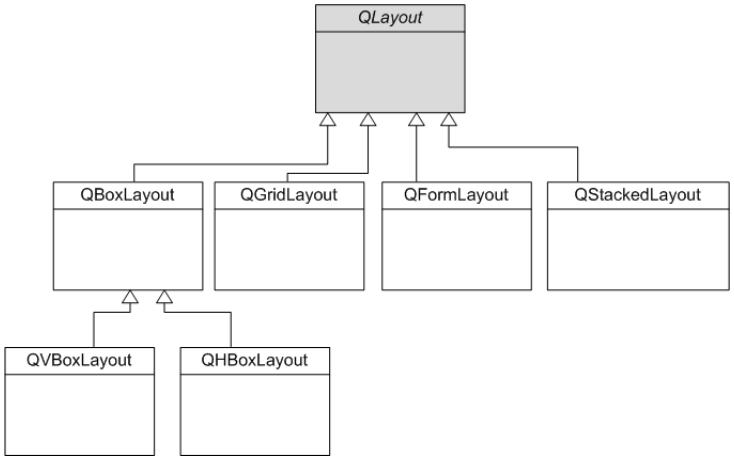
B.7 Positionnement (*layout*)

Vous pouvez gérer vos *widgets* de deux manières :
- avec un positionnement absolu
- avec un positionnement relatif

Le positionnement absolu est généralement déconseillé car il pose des problèmes de résoltion d'écran, de redimensionnement, ...

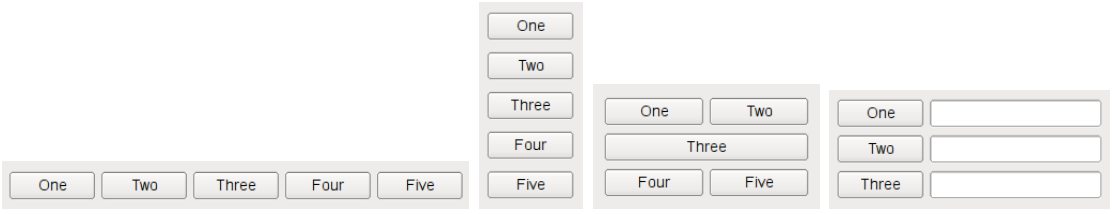
Qt fournit un **système de disposition (*layout*)** pour l'organisation et le positionnement automatique des *widgets* enfants dans un *widget*. Ce gestionnaire de placement permet l'agencement facile et le bon usage de l'espace disponible.

Qt inclut un ensemble de classes `QxxxLayout` qui sont utilisés pour décrire la façon dont les *widgets* sont disposés dans l'interface utilisateur d'une application.



Toutes les classes héritent de la classe abstraite `QLayout`.

Les plus utilisés sont :
- `QHBoxLayout` : boîte horizontale
- `QVBoxLayout` : boîte verticale
- `QGridLayout` : grille
- `QFormLayout` : formulaire



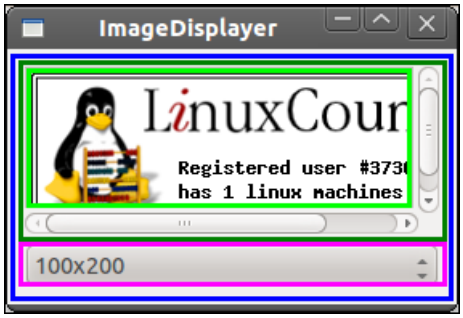
Toutes les sous-classes de `QWidget` peuvent utiliser les *layouts* pour gérer leurs enfants. `QWidget::setLayout()` applique une mise en page à un *widget*.

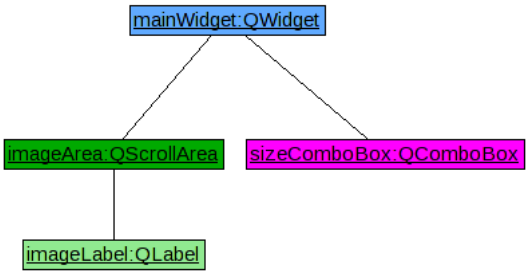
Lorsqu'un *layout* est défini sur un *widget* de cette manière, il prend en charge les tâches suivantes :
- le positionnement des *widgets* enfants
- la gestion des tailles (minimale, préférée, ...)
- le redimensionnement
- la mise à jour automatique lorsque le contenu change

D'une manière générale, les *widgets* sont hiérarchiquement inclus les uns dans les autres. Le principal avantage est que si le parent est déplacé, les enfants le sont aussi.

Les gestionnaires de disposition (les classes `QxxxLayout`) simplifient ce travail :
- on peut ajouter des *widgets* dans un *layout* :
  `void QLayout::addWidget(QWidget *widget)`
- on peut associer un *layout* à un *widget* qui devient alors le propriétaire du *layout* et **parent** des *widgets* inclus dans le *layout* :
  `void QWidget::setLayout (QLayout *layout)`
- on peut ajouter des *layouts* dans un *layout* :
  `void QLayout::addLayout(QLayout *layout)`

Exemple : on utilise le gestionnaire de disposition `QVBoxLayout` que l'on associe au *widget* parent (`mainWidget`). La classe `QScrollArea` fournit une zone de défilement utilisée pour afficher le contenu d'un *widget* enfant dans un cadre (`imageLabel`).






```
#include <QApplication>
#include <QLabel>
#include <QScrollArea>
#include <QComboBox>
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
   QApplication a(argc, argv);

   // une image
   QLabel * imageLabel = new QLabel;
   imageLabel->setPixmap(QPixmap("373058.png"));

   // une zone de défilement
   QScrollArea * imageArea = new QScrollArea;
   imageArea->setWidget(imageLabel); // imageArea parent de imageLabel

   // une liste déroulante
   QComboBox * sizeComboBox = new QComboBox;
   sizeComboBox->addItem("100x200");
   sizeComboBox->addItem("200x400");

   // un positionnement vertical
   QVBoxLayout * layout = new QVBoxLayout; // parent de imageArea et de
       sizeComboBox
   layout->addWidget(imageArea);
```

```
    layout->addWidget(sizeComboBox);

    // le widget central
    QWidget mainWidget;
    mainWidget.setLayout(layout); // parent de layout

    mainWidget.show(); // affichage de mainWidget et de ses enfants

    return a.exec(); // boucle d'attente d'évènements

    // destruction de mainWidget et de ses enfants :
    //  -> destruction de layout
    //  -> destruction de sizeComboBox
    //  -> destruction de imageArea
    //  -> destruction de imageLabel
}
```

Question 27 (1 point)
Les *layouts* permettent-ils d'obtenir un positionnement absolu ou relatif des *widgets* ?
☐ Positionnement relatif ☐ Positionnement absolu

Question 28 (1 point)
Laquelle de ces classes permet de disposer plusieurs *widgets* sur une même ligne ?
☐ `QHBoxLayout` ☐ `QVBoxLayout`

Question 29 (1 point)
Quelle méthode de la fenêtre doit-on appeler pour lui indiquer le *layout* principal qu'elle doit utiliser ?
☐ `setLayout` ☐ `addLayout` ☐ `setWidget` ☐ `addWidget`

Question 30 (1 point)
Quelle méthode du *layout* doit-on appeler pour lui ajouter un *widget* ?
☐ `setLayout` ☐ `addLayout` ☐ `setWidget` ☐ `addWidget`

Question 31 (1 point)
Un *layout* peut-il en contenir un autre ?
☐ Oui ☐ Non

B.8 Transition Qt4 → Qt5

Qt 5.0 est sorti le 19 décembre 2012. Bien que marquant des changements majeurs sur bien des points, le passage à Qt5 casse au minimum la compatibilité au niveau des sources. De cette façon, le passage est bien plus facile que celui de Qt3 vers Qt4.

Les principales conséquences sont :

→ Intégrer **QtWidgets** qui est un module séparé donc il faudra ajouter dans le fichier `.pro` :

```
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

→ Corriger les erreurs de compilation comme : `error: QPushButton: No such file or directory`

```
#if QT_VERSION >= 0x050000
#include <QtWidgets/QPushButton> /* pour Qt5 */
#else
#include <QPushButton> /* pour Qt4 */
#endif
```

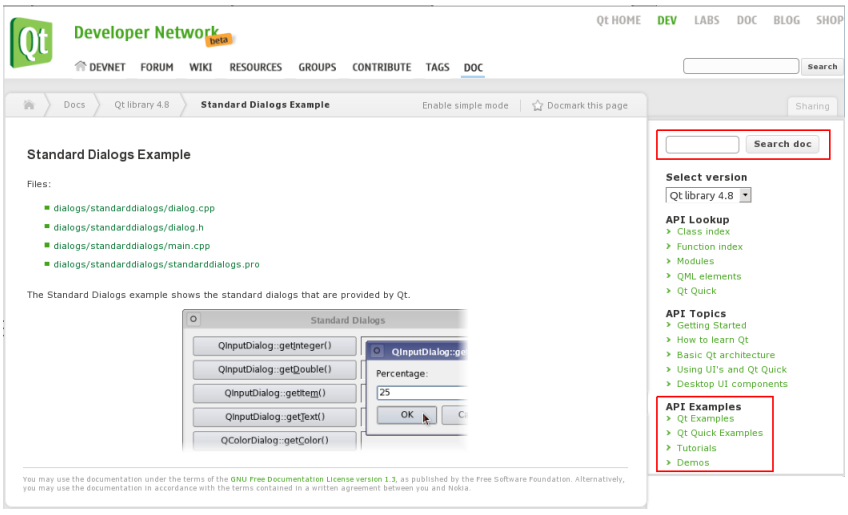
De manière plus générale :

```
#if QT_VERSION >= 0x050000
#include <QtWidgets> /* tous les widgets de Qt5 */
#else
#include <QtGui> /* tous les widgets de Qt4 */
#endif
```

Les deux corrections ont été réalisées afin de laisser la compatibilité Qt4/Qt5. Pour en savoir plus, lire : wiki.qt.io/Transition_from_Qt_4.x_to_Qt5

B.9 Documentation

La documentation générale de Qt se trouve à cette adresse : doc.qt.io



La documentation de référence Qt4 : doc.qt.io/qt-4.8/index.html, et plus particulièrement les concepts de base (http ://doc.qt.io/qt-4.8/qt-basic-concepts.html) : *Signals and Slots*, *Main Classes*, *Main Window Architecture*, *Internationalization*, ...

Une partie de cette documentation est traduite en français : qt.developpez.com/doc/4.7/index/

La documentation de Qt4 fournit de nombreux exemples (plus de 400), notamment :

- doc.qt.io/qt-4.8/examples-widgets.html
- doc.qt.io/qt-4.8/dialogs-standarddialogs.html
- doc.qt.io/qt-4.8/examples-mainwindow.html
- doc.qt.io/qt-4.8/widgets-windowflags.html
- doc.qt.io/qt-4.8/examples-layouts.html

Les exemples pour Qt5 sont ici : doc.qt.io/qt-5/

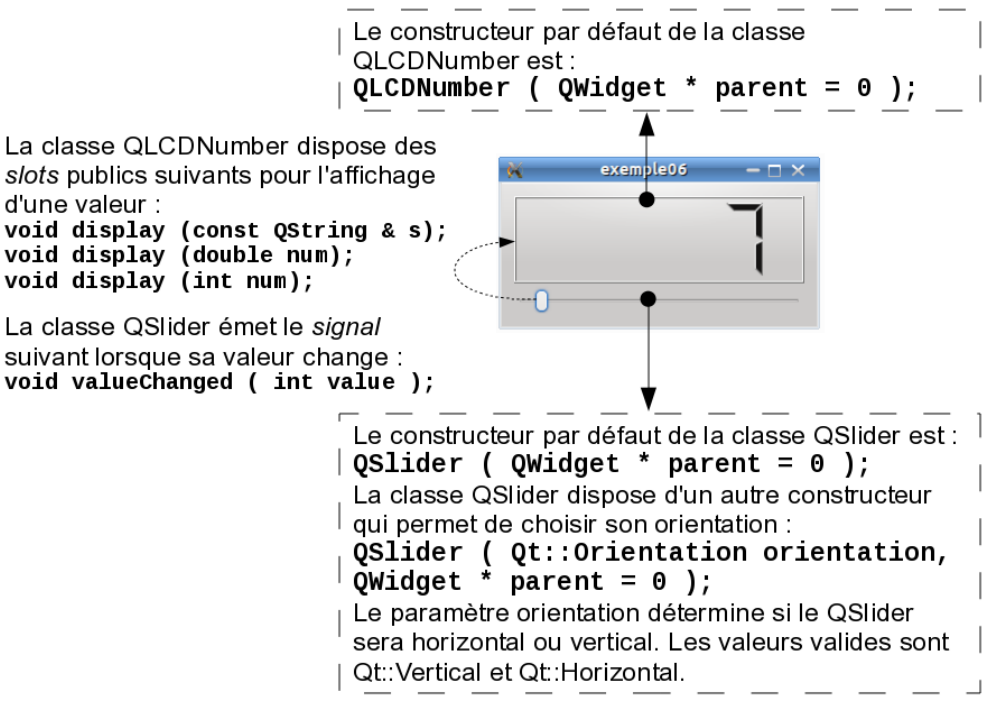
Les tutoriels : doc.qt.io/qt-4.8/tutorials.html

Les documentations des outils Qt :
- `qmake` : doc.qt.io/qt-4.8/qmake-tutorial.html
- Qt Designer : doc.qt.io/qt-4.8/designer-manual.html
- Qt Linguist : doc.qt.io/qt-4.8/linguist-manual.html
- Qt Creator : doc.qt.io/qtcreator/index.html

Quelques autres liens :
- Qt Wiki : wiki.qt.io
- Qt Centre : www.qtcentre.org
- Qt Forum (en) : www.qtforum.org
- QtFr (fr) : www.qtfr.org
- Qt Tutorial (fr) : www.digitalfanatics.org
- Qt-Apps.org : www.qt-apps.org
- Qt-Prop.org : www.qt-prop.org
- Qt developpez.com (fr) : qt.developpez.com

B.10 Exemple

On désire réaliser une application affichant ceci :



Le principe de cette application est simple : lorsque l'utilisateur déplace le curseur d'un `QSlider`, la valeur correspondante s'affiche dans un `QLCDNumber`. Pour cela on va créer une classe `MyWidget` contenant ces deux *widgets* (enfants). On vous fournit la déclaration de la classe `MyWidget` :

```
#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QtGui>

class MyWidget : public QWidget
{
   Q_OBJECT
   public:
        MyWidget( QWidget *parent = 0 );

   private:
        // mes widgets
        QLCDNumber *lcd;
        QSlider *slider;
};

#endif
```

Question 32 (1 point)
Donner la définition du constructeur de la classe `MyWidget` :
☐ `MyWidget::MyWidget( QWidget *parent ) : QWidget( 0 ) { ... }`
☐ `MyWidget::MyWidget( QWidget *parent ) : QWidget( this ) { ... }`
☐ `MyWidget::MyWidget( QWidget *parent ) : QWidget( parent ) { ... }`

Question 33 (1 point)
Instancier l'objet `lcd` dans le constructeur de la classe `MyWidget` :
☐ `lcd = new QLCDNumber( this );`
☐ `lcd = new QLCDNumber();`
☐ `lcd = new QLCDNumber( Qt::Horizontal );`

Question 34 (1 point)
Instancier l'objet `slider` dans le constructeur de la classe `MyWidget` :
☐ `slider = new QSlider( Qt::Vertical, this );`
☐ `slider = new QSlider( Qt::Horizontal, this );`
☐ `slider = new QSlider( this );`
☐ `slider = new QSlider();`

Question 35 (1 point)
Pour assurer l'affichage désiré, on a besoin d'instancier un *layout* dans le constructeur de la classe `MyWidget` :
☐ `QLayout *mainLayout = new QLayout;`
☐ `QHBoxLayout *mainLayout = new QHBoxLayout;`
☐ `QVBoxLayout *mainLayout = new QVBoxLayout;`
☐ `QFormLayout *mainLayout = new QFormLayout;`

Question 36 (1 point)
Placer le widget `lcd` dans le *layout* dans le constructeur de la classe `MyWidget` :
☐ `lcd->addLayout( mainLayout );` ☐ `mainLayout->addWidget( this );`
☐ `mainLayout->addWidget( lcd );` ☐ `mainLayout->addLayout( lcd );`

Question 37 (1 point)
Fixer le *layout* `mainLayout` comme *layout* principal du widget dans le constructeur de la classe `MyWidget` :
☐ `addLayout( mainLayout );` ☐ `setLayout( this );`
☐ `setLayout( mainLayout );` ☐ `mainLayout->addWidget( this );`

Question 38 (1 point)
Assurer la connexion signal/slot entre le `slider` et le `lcd` dans le constructeur de la classe `MyWidget` :
☐ `connect( lcd, SIGNAL(display(int)), slider, SLOT(valueChanged(int)) );`
☐ `connect( this, SIGNAL(valueChanged(int)), this, SLOT(display(int)) );`
☐ `connect( slider, SIGNAL(valueChanged()), lcd, SLOT(display() );`
☐ `connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );`

Question 39 (1 point)
Instancier un `myWidget` et lancer son affichage comme fenêtre principale dans la fonction `main()` de l'application :
☐ `MyWidget myWidget( this ); myWidget->show();`
☐ `MyWidget myWidget; myWidget->show();`
☐ `MyWidget myWidget(); myWidget.exec();`
☐ `MyWidget myWidget; myWidget.show();`

Question 40 (1 point)
Instancier un objet `QApplication` et lancer l'exécution de sa boucle d'évènements dans la fonction `main()` :
☐ `QApplication a(argc, argv); ...; return a.exec();`
☐ `QApplication a; ...; return a.exec();`
☐ `QApplication a(argc, argv); ...; return a->exec();`
☐ `QApplication a(myWidget); ...; return a.exec();`