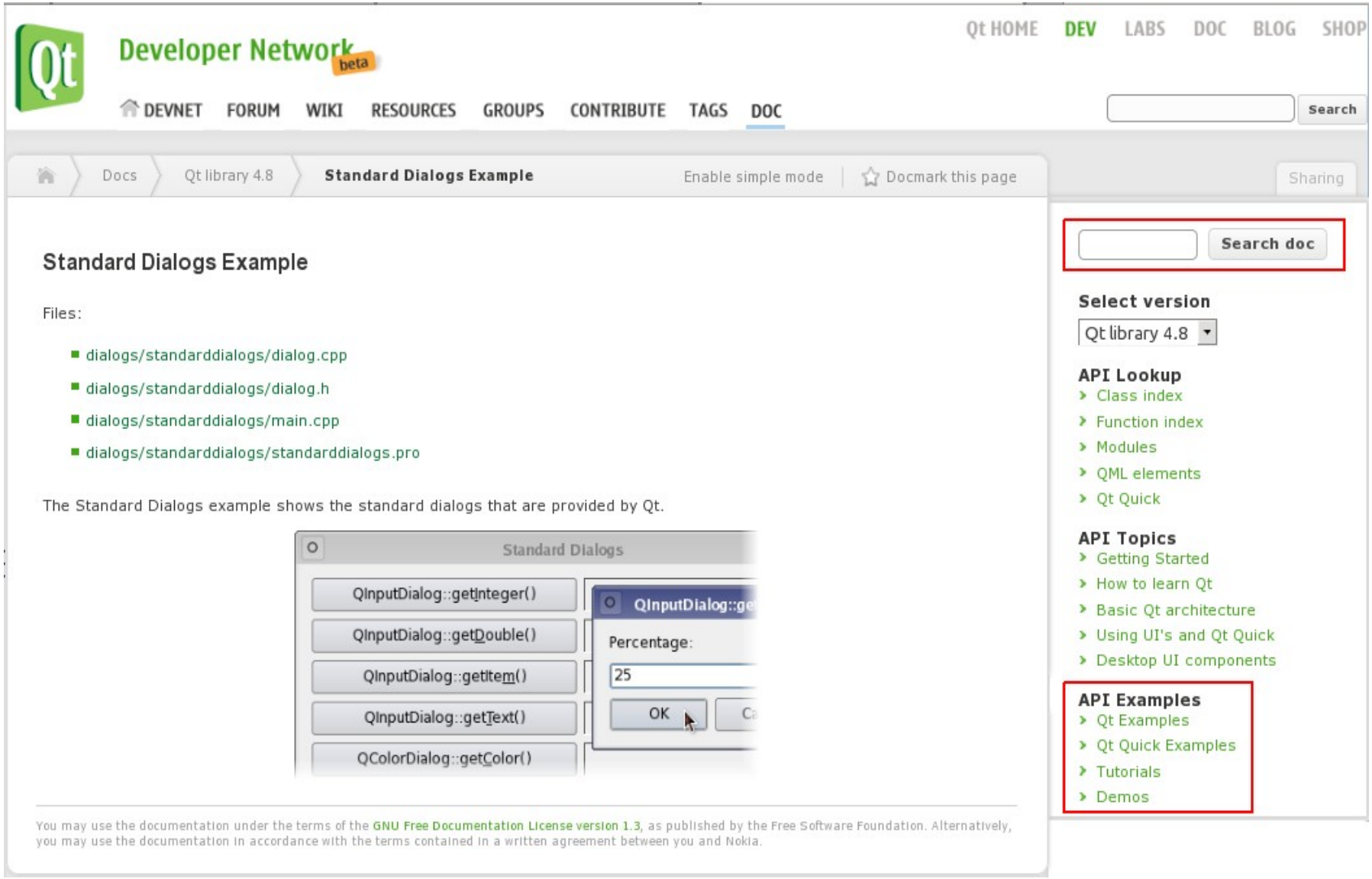
Interface Utilisateur Graphique (GUI)

- Notion de fenêtre
- Utilisation des classes : QWidget, QDialog et QMainWindow
- Les layouts
- Boîte de dialogue modale et non modale
- Les application SDI et MDI
- Utilisation des classes : QAction, QMenu, QToolBar, QDockWidget et QStatusBar
- Graphique et dessin 2D : le framework Graphics View et QPainter
- QPicture et QImage
- La gestion d'évènement souris et le « glisser-déposer » (drag & drop)

Interface Utilisateur Graphique



Les exemples de Qt

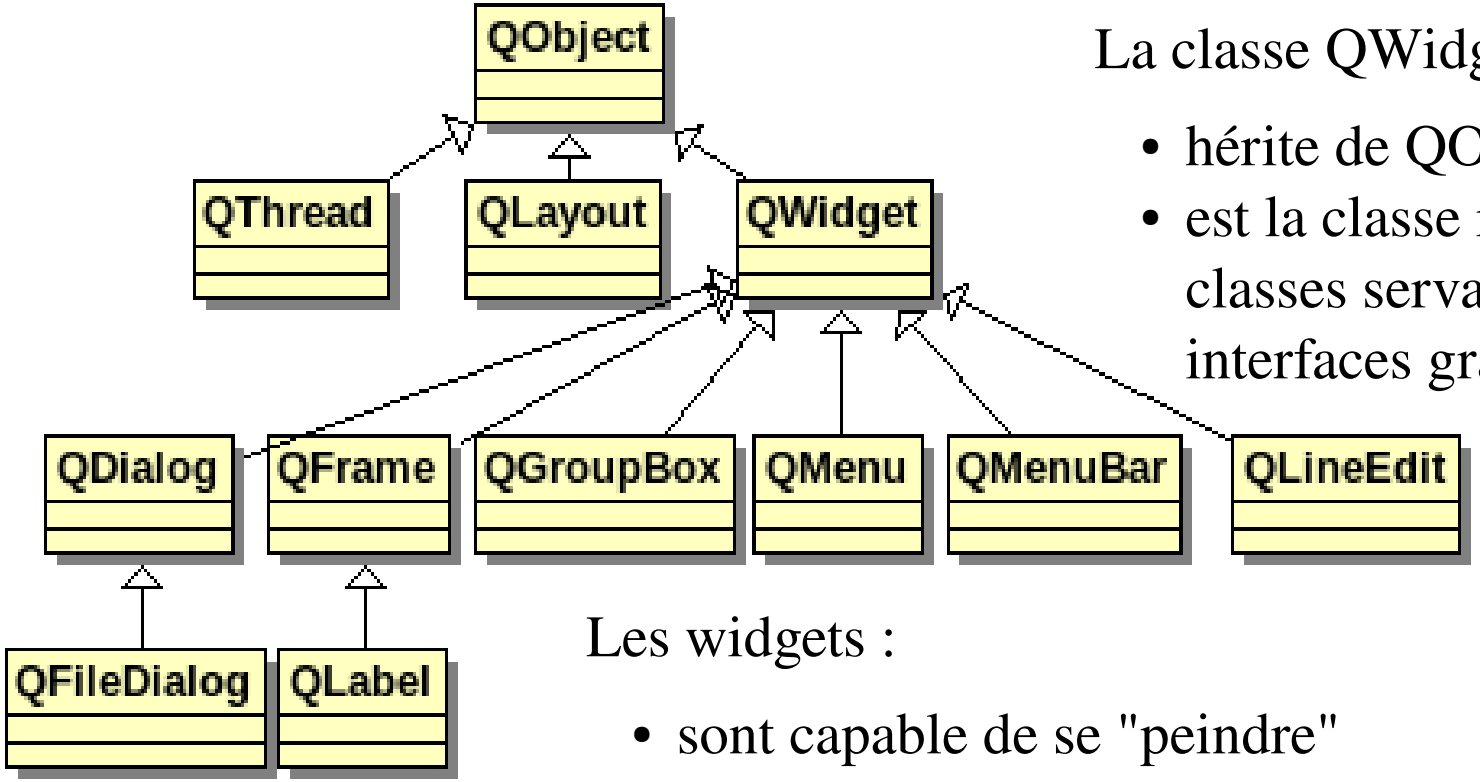
- Tous les exemples de ce cours sont disponibles à l'adresse suivante :
  - http://tvaira.free.fr/dev/qt/exemples-qt.zip

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

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

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

La classe QWidget



La classe QWidget :

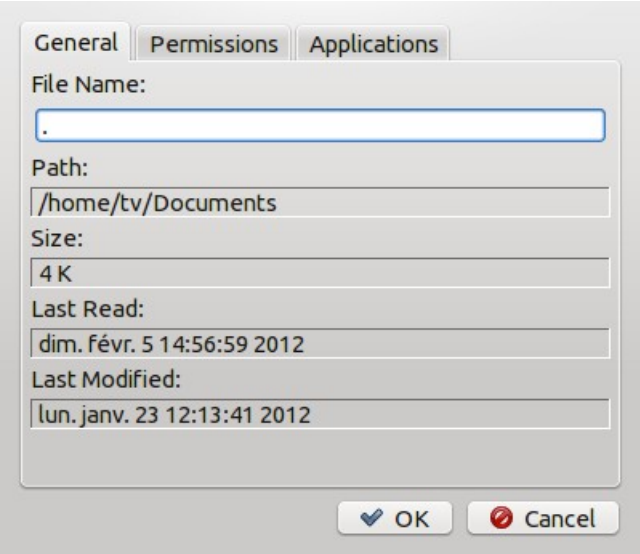
- hérite de QObject
- est la classe mère de toutes les classes servant à réaliser des interfaces graphiques

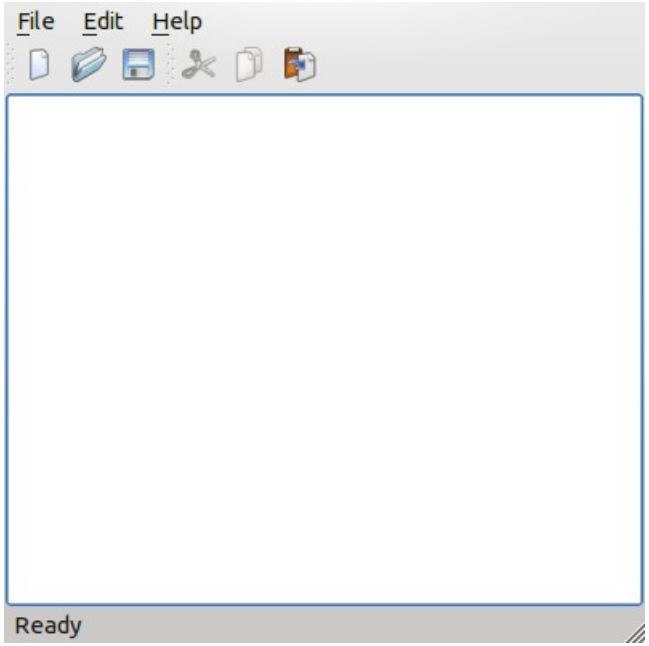
Les widgets :

- sont capable de se "peindre"
- sont capable de recevoir les évènements souris, clavier
- sont les éléments de base des interfaces graphiques
- sont tous rectangulaires
- ils sont ordonnés suivant l'axe z (gestion de la profondeur)
- ils peuvent avoir un widget parent

Notion de fenêtre

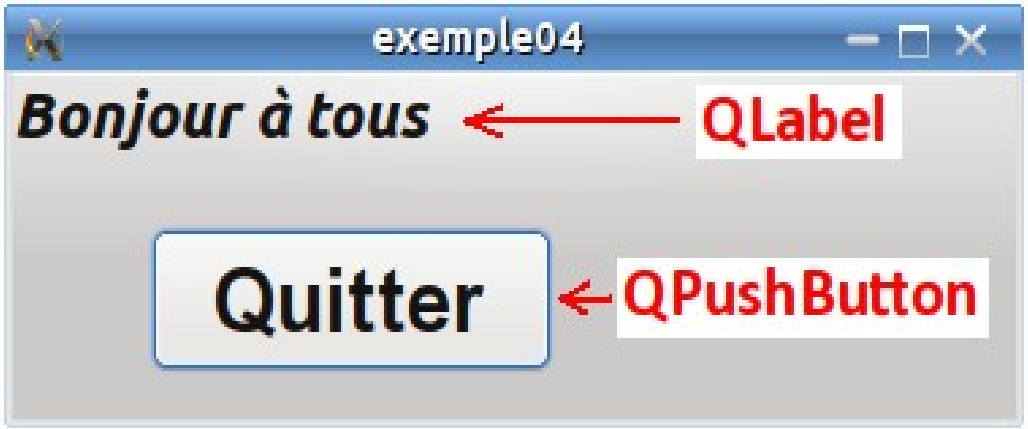
- Un widget qui n'est pas incorporé dans un widget parent est appelé une fenêtre.
- Habituellement, les fenêtres ont un cadre et une barre de titre, mais il est également possible de créer des fenêtres sans décoration en utilisant des propriétés spécifiques (*window flags*).
- Dans Qt, **QMainWindow** et les différentes sous-classes de **QDialog** sont les types de fenêtres les plus courantes.





Les widgets

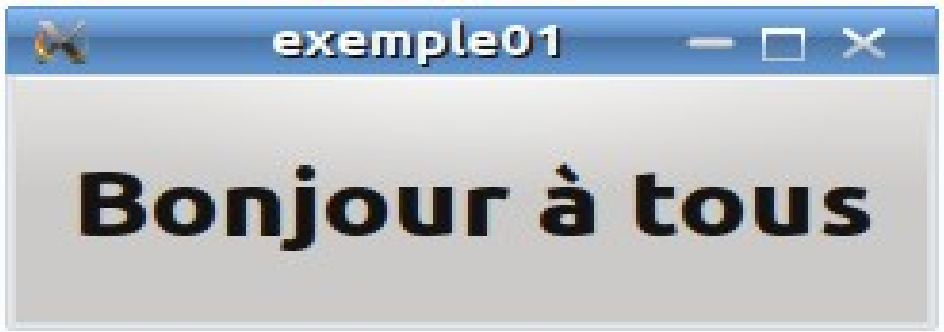
- Il existe beaucoup de sous-classes de QWidget qui fournissent une réelle fonctionnalité, telle que **QLabel**, **QPushButton**, QListWidget et QTabWidget.
- Les widgets qui ne sont pas des fenêtres sont des widgets enfants, affichés dans leur widget parent. La plupart des widgets dans Qt sont principalement utiles en tant que widgets enfants.
- Par exemple, il est possible d'afficher un bouton en tant que fenêtre de haut niveau, mais on préfère généralement mettre les boutons à l'intérieur d'autres widgets, tels que QDialog.



Widget : exemple n°1

- Un widget est toujours créé caché, il est donc nécessaire d'appeler la méthode **show()** pour l'afficher

```
#include <QApplication>
#include <QLabel>

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

    QLabel *pMonTexte = new QLabel(
        QString::fromUtf8("<H1><center>Bonjour à
        tous</center></H1>"), NULL);

    pMonTexte->show();

    return MonAppli.exec();
}
```



Widget : exemple n°2

- 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.
- On ajoute un bouton et les deux éléments seront inclus dans un même widget :

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

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

   QWidget *pMaFenetre = new QWidget();

   QLabel* pMonTexte = new QLabel("...", pMaFenetre);
   QPushButton *pMonBouton = new
           QPushButton("Quitter", pMaFenetre);

   pMaFenetre->show();

   return MonAppli.exec();
}
```

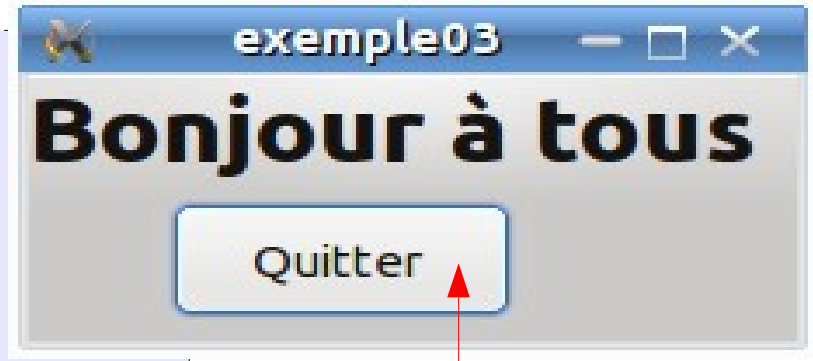
Widget parent

Widgets enfants

Widget : exemple n°3

- Pour rendre actif le bouton, on **connecte le signal clicked() émis par l'objet pMonBouton au slot quit() de l'objet MonAppli** :

```
#include <QApplication>
#include <QLabel>
#include <QPushButton>
int main( int argc, char* argv[] ) {
 QApplication MonAppli( argc, argv );
 QWidget *pMaFenetre = new QWidget();
 QLabel* pMonTexte = new QLabel("...",pMaFenetre);
 QPushButton *pMonBouton = new
                QPushButton("Quitter", pMaFenetre);
 QObject::connect(pMonBouton, SIGNAL(clicked()),
                  &MonAppli, SLOT(quit()));
 pMaFenetre->show();
 return MonAppli.exec();
}
```



Si on clique sur le bouton, on quitte l'application.

Les applications doivent se terminer proprement en appelant QApplication::quit(). Cette méthode est appelée automatiquement lors de la fermeture du dernier widget.

Widget : exemple n°4

- Les sous-classes de QWidget possèdent de **nombreuses méthodes** qui permettent d'agir sur l'aspect visuel :

```
int main( int argc, char* argv[] ) {
 QApplication MonAppli( argc, argv );
 QWidget *pMaFenetre = new QWidget();

 QLabel* pMonTexte = new QLabel("<h2><em>Bonj
tous</em></h2>",pMaFenetre);
 QPushButton *pMonBouton = new
QPushButton("Quitter",pMaFenetre);

 pMonBouton->setGeometry(0,0,pMonTexte->width(),40);
 pMonBouton->move(50, 50);
 pMonBouton->resize(150, 50);
 pMonBouton->setFont(QFont("Arial", 18, QFont::Bold));

 QObject::connect(pMonBouton, SIGNAL(clicked()),
&MonAppli, SLOT(quit()));
 pMaFenetre->show();

 return MonAppli.exec();
}
```

Widget : exemple n°5

- Les **feuilles de style Qt (QSS)** sont un mécanisme puissant qui permet de personnaliser l'apparence des widgets.
- Les concepts, la terminologie et la syntaxe des feuilles de style Qt sont fortement inspirés par les feuilles de style en cascade (CSS) utilisées en HTML mais adaptées au monde de widgets.

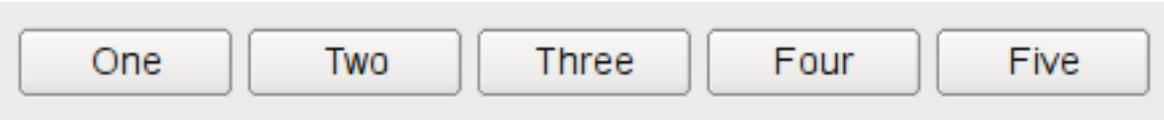

```
int main( int argc, char* argv[] ) {
   QApplication MonAppli( argc, argv );
   QWidget *pMaFenetre = new QWidget();

   QFile file("qss/default.qss");
   if(file.open(QFile::ReadOnly)) {
    QString styleSheet = QLatin1String(file.readAll());
    MonAppli.setStyleSheet(styleSheet);
   }

   ...
   pMonBouton->setStyleSheet("background-color:
green");
   ...
   return MonAppli.exec();
}
```

Les *layouts* (1/2)

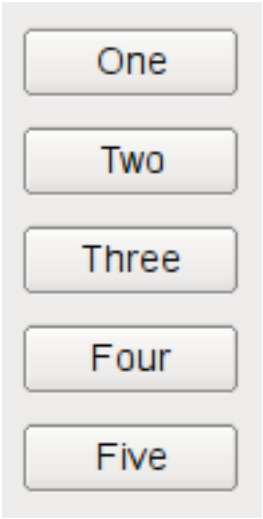
- Qt fournit un système de disposition (*layout*) pour l'organisation et le positionnement automatique des widgets enfants dans un widget. Ce gestionnaire de placement permet l'agencement facile et le bon usage de l'espace disponible.
- Qt inclut un ensemble de classes **QxxxLayout** qui sont utilisés pour décrire la façon dont les widgets sont disposés dans l'interface utilisateur d'une application.
- Toutes les 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 :
  - Positionnement des widgets enfants
  - Gestion des tailles (minimale, préférée)
  - Redimensionnement
  - Mise à jour automatique lorsque le contenu change

Les *layouts* (2/2)

- Quelques layouts : horizontal, vertical, grille, formulaire ...



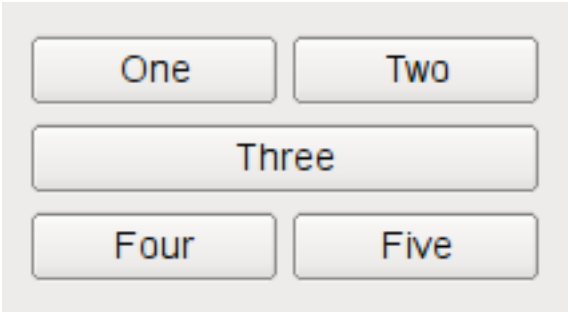
QHBoxLayout



QVBoxLayout

QFormLayout

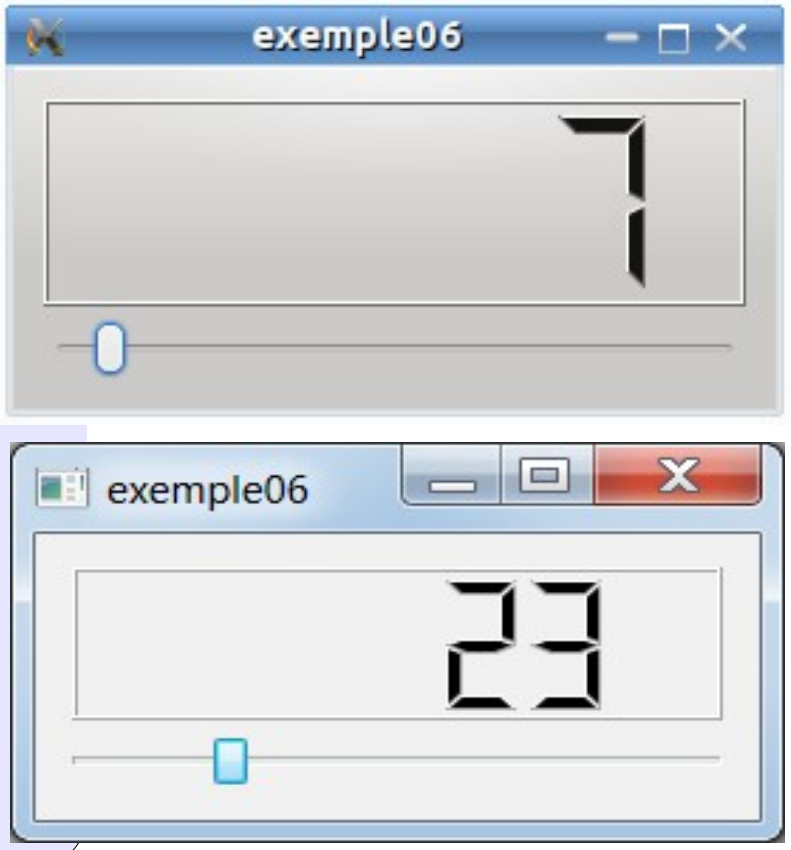
QGridLayout



Widget : exemple n°6 (1/3)

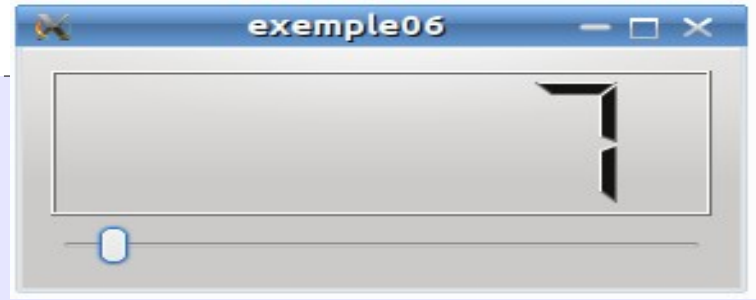
- On peut **réutiliser** les widgets de Qt :
  - Par **héritage** : extension d'un type de widget
  - Par **composition** : assemblage de widgets

```
#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QWidget>
#include <QLCDNumber>
#include <QSlider>

class MyWidget : public QWidget
{
    Q_OBJECT

    public:
        MyWidget( QWidget *parent = 0 );

    private :
        QLCDNumber *lcd;
        QSlider *slider;
};

#endif
```



Widget : exemple n°6 (2/3)

- On instancie deux objets widgets : la barre QSlider et l'affichage LCD QLCDNumber.
- On connecte : slider->valueChanged(int) (méthode déclencheuse) à lcd->display(int) (méthode déclenchée)



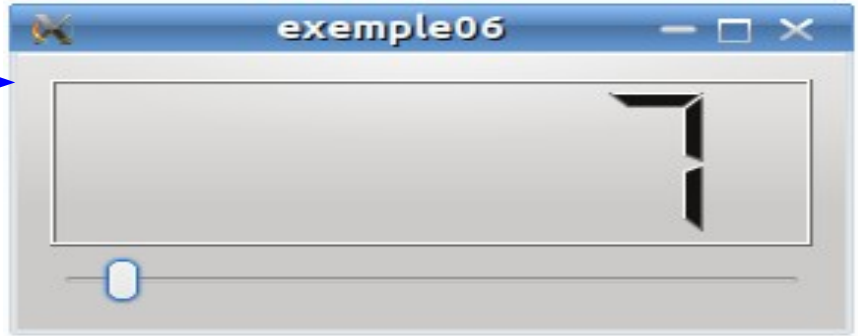

```
#include <QVBoxLayout>

#include "mywidget.h"

MyWidget::MyWidget( QWidget *parent ) : QWidget( parent )
{
    lcd = new QLCDNumber( this );
    slider = new QSlider( Qt::Horizontal, this );
    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addWidget(lcd);
    mainLayout->addWidget(slider);
    setLayout(mainLayout);

    connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}
```

*Il n'y a pas de **delete** car le widget parent se chargera de la destruction de ses widgets enfants.*

Widget : exemple n°6 (3/3)

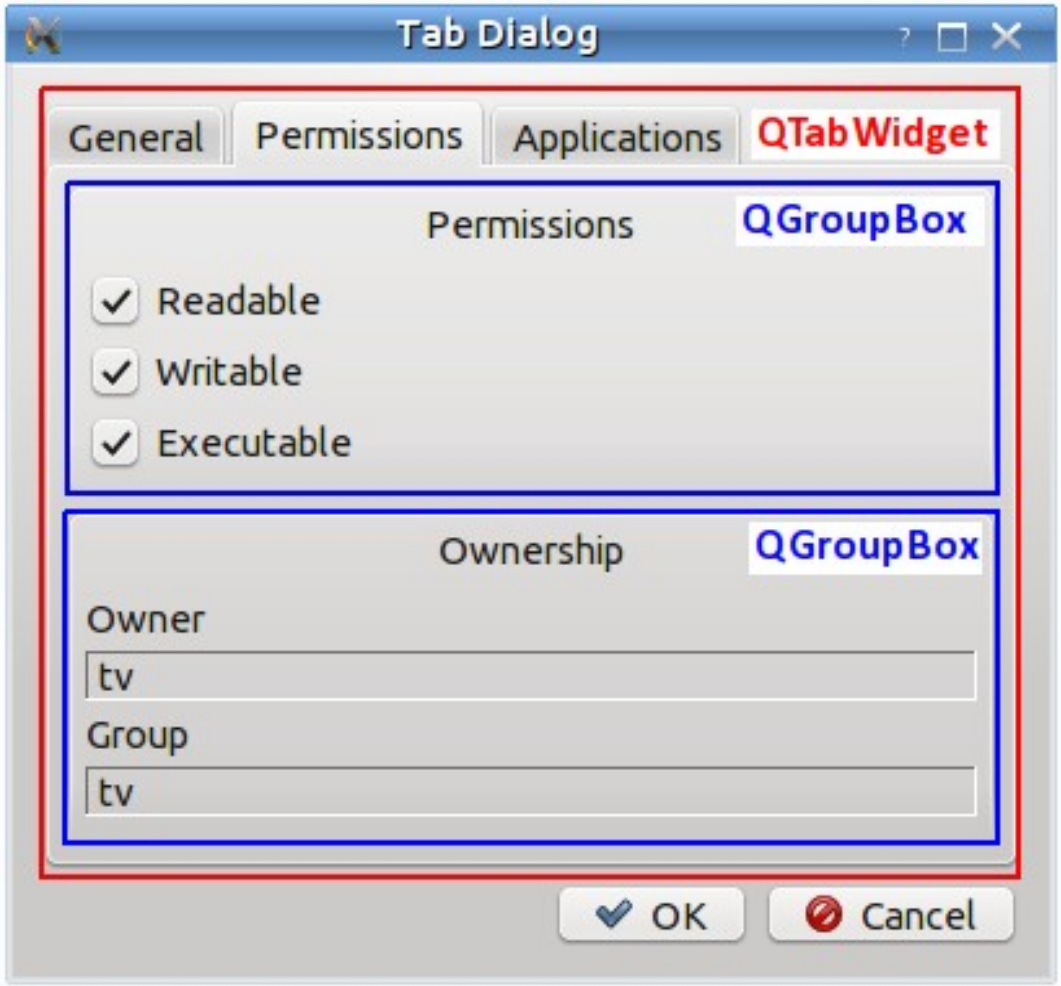
- Pour finir, on instancie notre nouveau widget et on l'affiche :

```
#include <QApplication>

#include "mywidget.h"

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

    MyWidget w;

    w.show();

    return a.exec();
}
```



Encapsulation de Widgets

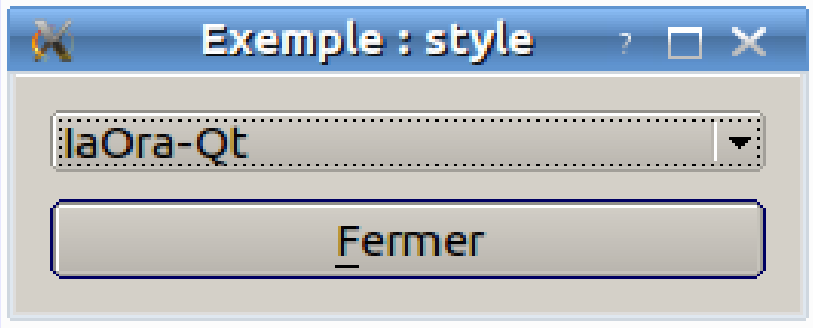
- Exemple de widgets Qt encapsulant d'autres widgets : **QGroupBox**, **QTabWidget**



Widget : exemple n°7 (1/3)

- En utilisant **QStyleFactory::keys()**, on obtient la liste des styles disponibles :

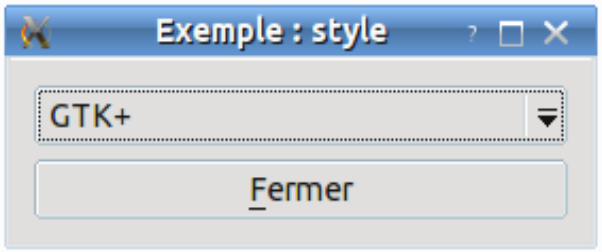
```
#include <QApplication>
#include <QtGui>
#include <QDebug>

class MyDialog : public QDialog
{
 Q_OBJECT
 public:
   MyDialog(QWidget *parent) : QDialog(parent)
   {
     QComboBox *styleComboBox = new QComboBox;
     styleComboBox->addItems(QStyleFactory::keys());
     qDebug() << QStyleFactory::keys();
     QLabel *styleLabel = new QLabel(tr("&Style :"));
     styleLabel->setBuddy(styleComboBox);

     connect(styleComboBox, SIGNAL(activated(QString)),
this, SLOT(changeStyle(QString)));
     …
   }
```

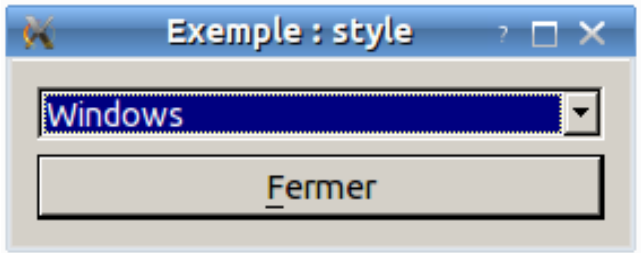


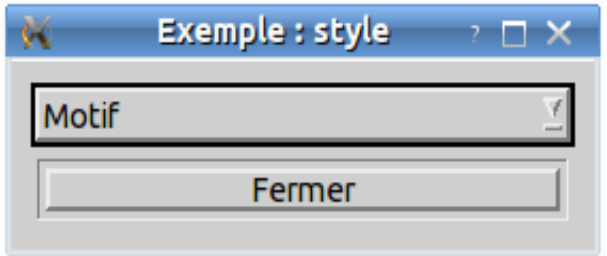
Widget : exemple n°7 (2/3)

- La classe **QStyle** est une classe de base abstraite qui encapsule le *look and feel* de l'interface graphique. Qt contient un ensemble de sous-classes QStyle qui émulent les styles des différentes plates-formes prises en charge par Qt (QWindowsStyle, QMacStyle, QMotifStyle, etc.) Par défaut, ces modèles sont construits dans la bibliothèque **QtGui**.
- On peut changer pendant l'exécution le style de l'application avec **setStyle()** :

```
...
private slots:
  void changeStyle(const QString &styleName)
  {
   QApplication::setStyle(QStyleFactory::create(styleName));
   QApplication::setPalette(QApplication::style()->standardPalette());
  }
};
```

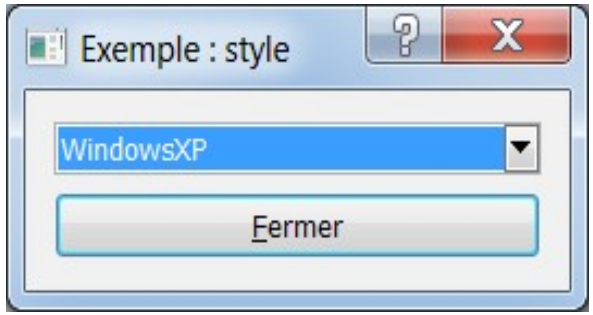


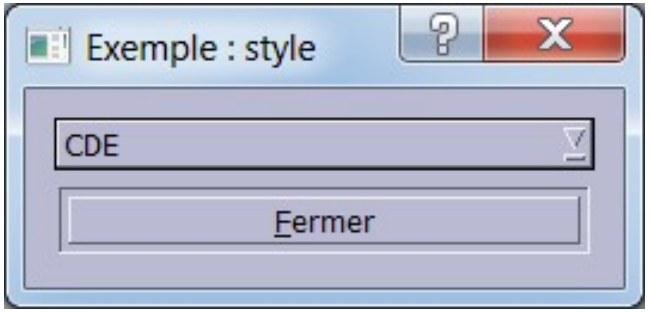


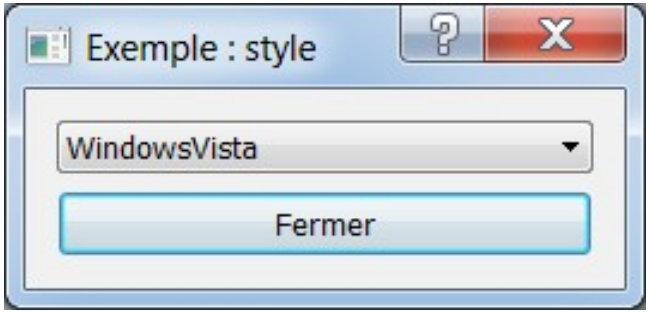


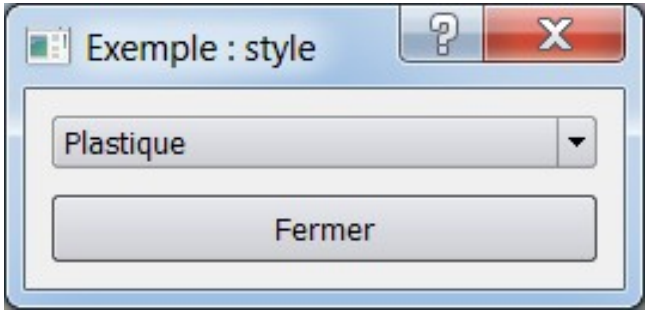
Widget : exemple n°7 (2/3)

- La classe **QStyle** est une classe de base abstraite qui encapsule le *look and feel* de l'interface graphique. Qt contient un ensemble de sous-classes QStyle qui émulent les styles des différentes plates-formes prises en charge par Qt (QWindowsStyle, QMacStyle, QMotifStyle, etc.) Par défaut, ces modèles sont construits dans la bibliothèque **QtGui**.



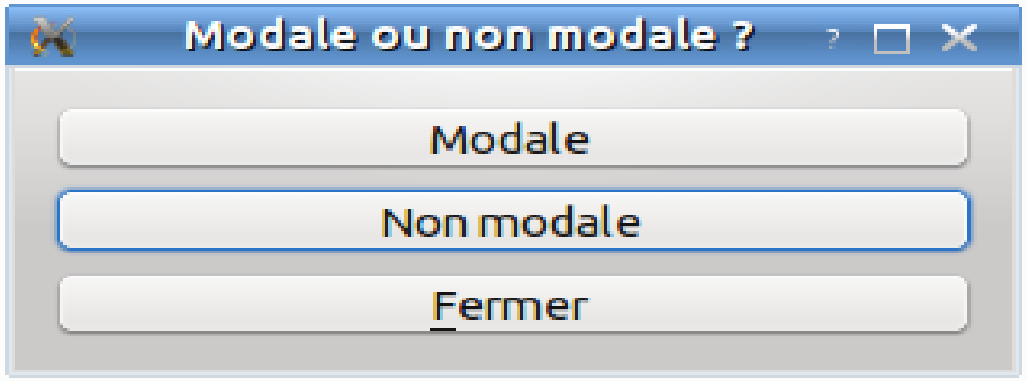






La classe QDialog

- La classe **QDialog** est la classe de base des fenêtres de dialogue. Elle hérite de **QWidget**.
- Une fenêtre de dialogue (ou boîte de dialogue) est principalement utilisée pour des tâches de courte durée et de brèves communications avec l'utilisateur.
- Une fenêtre de dialogue (ou boîte de dialogue) :
  - peut être modale ou non modale.
  - peut fournir une valeur de retour
  - peut avoir des boutons par défaut
  - peut aussi avoir un QSizeGrip (une poignée de redimensionnement) dans le coin inférieur droit

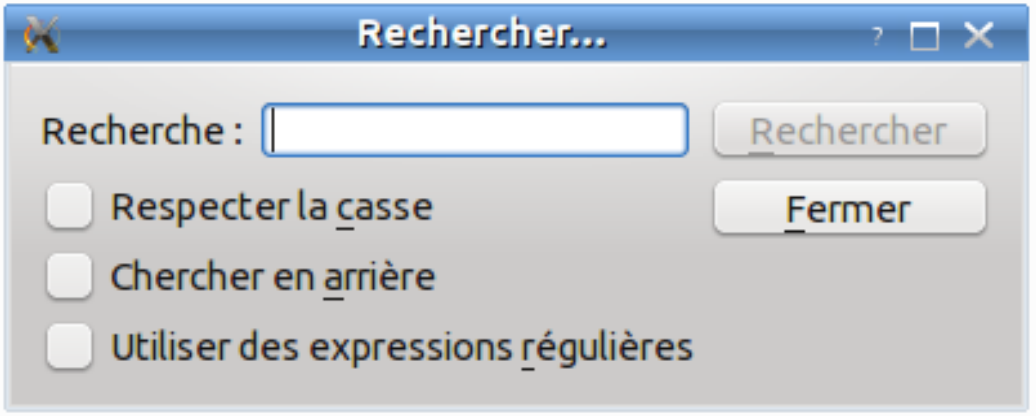


Exemple n°3

Boîte de dialogue non modale

- Une boîte de dialogue non modale (***modeless dialog***) est un dialogue qui fonctionne indépendamment des autres fenêtres de la même application.
- Exemple : rechercher du texte dans les traitements de texte
- Une boîte de dialogue non modale est affichée en utilisant **show()** qui retourne le contrôle à l'appelant immédiatement.

  Remarque : si la boîte de dialogue est visuellement cachée, il suffira d'appeler successivement show(), raise() et activateWindow() pour la replacer sur le dessus de la pile.

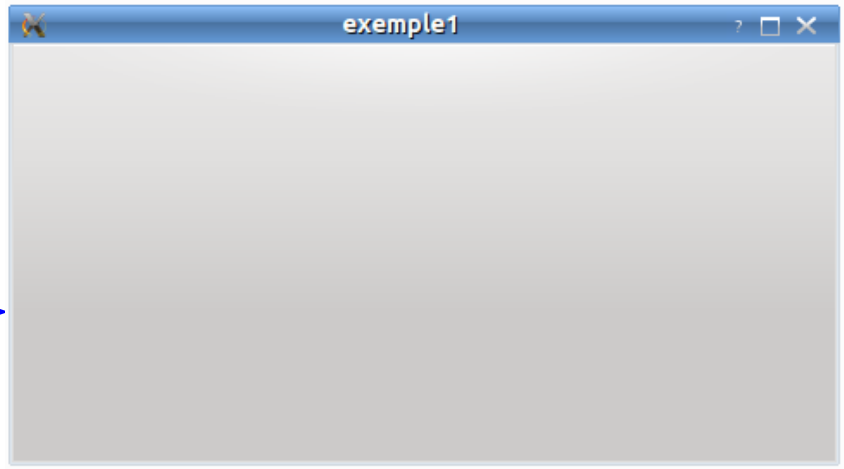


QDialog : exemple n°1

- Pour créer sa propre boîte de dialogue, il suffit de créer une classe qui **hérite de QDialog**.

```
#include <QApplication>
#include <QtGui>

class MyDialog : public QDialog
{
   public:
      MyDialog() {}
};

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

  MyDialog myDialog;

  myDialog.show();

  return app.exec();
}
```

Boîte de dialogue non modale



Boîte de dialogue modale

- Une boîte de dialogue modale (***modal dialog***) est un dialogue qui bloque l'entrée à d'autres fenêtres visibles de la même application.
- Exemple : les dialogues qui sont utilisés pour demander un nom de fichier ou qui sont utilisés pour définir les préférences de l'application sont généralement modaux.

Remarque : les dialogues peuvent être application modale (par défaut) ou fenêtre modale.

- La façon la plus commune pour afficher une boîte de dialogue modale est de faire appel à sa fonction **exec()**. Lorsque l'utilisateur ferme la boîte de dialogue, exec() fournira une valeur de retour utile.

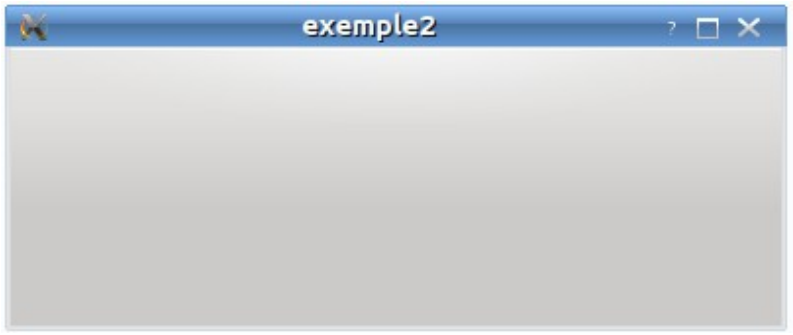
Remarque : une alternative est d'appeler setModal(true) ou setWindowModality(), puis show(). Contrairement à exec(), show() retourne le contrôle à l'appelant immédiatement (voir QProgressDialog).

QDialog : exemple n°2

- Pour fabriquer une boîte de dialogue, il suffit de créer une classe qui **hérite de QDialog**.

```
#include <QApplication>
#include <QtGui>

class MyDialog : public QDialog
{
   public:
      MyDialog() {}
};

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

  MyDialog myDialog;

  return myDialog.exec();
}
```

Boîte de dialogue modale



Les boîtes de dialogue Qt (1/2)

- Qt fournit un certain nombre de boîte de dialogue prêtes à l'emploi :

- La classe **QInputDialog** fournit un dialogue simple pour obtenir une valeur unique de l'utilisateur. La valeur d'entrée peut être une chaîne, un numéro ou un élément d'une liste (getText, getInt, getDouble, getItem). Une étiquette (Label) doit être placée afin de préciser à l'utilisateur ce qu'il doit entrer.

- La classe **QColorDialog** fournit un dialogue pour la spécification des couleurs. Cela permet aux utilisateurs de choisir les couleurs (getColor). Par exemple, vous pourriez l'utiliser dans un programme de dessin pour permettre à l'utilisateur de définir la couleur du pinceau.

- La classe **QFontDialog** fournit un widget de dialogue de sélection d'une police (getFont).

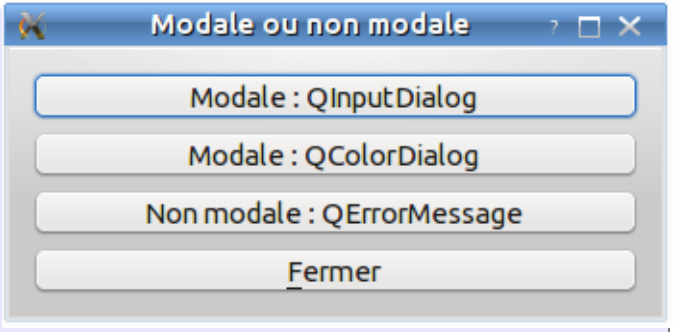
Les boîtes de dialogue Qt (2/2)

- La classe **QFileDialog** fournit une boîte de dialogue qui permet aux utilisateurs de sélectionner des fichiers ou des répertoires. Elle permet de parcourir le système de fichiers afin de sélectionner un ou plusieurs fichiers ou un répertoire (getExistingDirectory, getOpenFileName, getOpenFileNames, getSaveFileName).
- La classe **QMessageBox** fournit un dialogue modal pour informer l'utilisateur ou pour demander à l'utilisateur une question et recevoir une réponse. Elle fournit aussi quatre types prédéfinis : **QMessageBox::critical(), QMessageBox::information(), QMessageBox::question(), QMessageBox::warning()**
- La classe **QErrorMessage** fournit une boîte de dialogue qui affiche un message d'erreur (showMessage()).

- Exemple : http://developer.qt.nokia.com/doc/qt-4.8/dialogs-standarddialogs.html

QDialog : exemple n°4 (1/6)

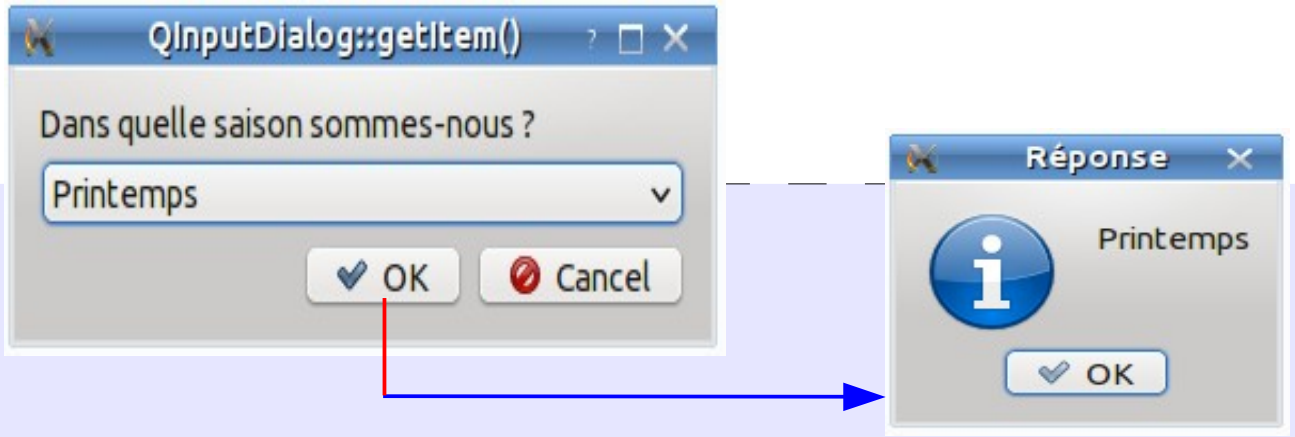
- Cet exemple montre l'utilisation de quelques boîtes de dialogue Qt :




```
#include <QApplication>
#include <QtGui>
class MyDialog : public Qdialog {
 Q_OBJECT
 public:
  MyDialog() {
     QPushButton *modalButton1 = new QPushButton("Modale : QInputDialog");
     QPushButton *modalButton2 = new QPushButton("Modale : QColorDialog");
     QPushButton *nomodalButton = new QPushButton("Non modale :
                                                   QErrorMessage");
     QPushButton *closeButton = new QPushButton("&Fermer");
     errorMessageDialog = new QErrorMessage(this);
     connect(modalButton1, SIGNAL(clicked()), this, SLOT(setItem()));
     connect(modalButton2, SIGNAL(clicked()), this, SLOT(setColor()));
     connect(nomodalButton, SIGNAL(clicked()), this, SLOT(showMessage()));
     connect(closeButton, SIGNAL(clicked()), this, SLOT(close()));
     ...
     setWindowTitle(tr("Modale ou non modale"));
  }
 private :
  QErrorMessage *errorMessageDialog;
```

QDialog : exemple n°4 (2/6)

- Utilisation de la classe **QInputDialog** qui fournit un dialogue simple pour obtenir une valeur unique de l'utilisateur. Ici la valeur d'entrée est un élément d'une liste (getItem). La valeur de retour (l'élément choisi) est affiché en utilisant la classe **QMessageBox**.

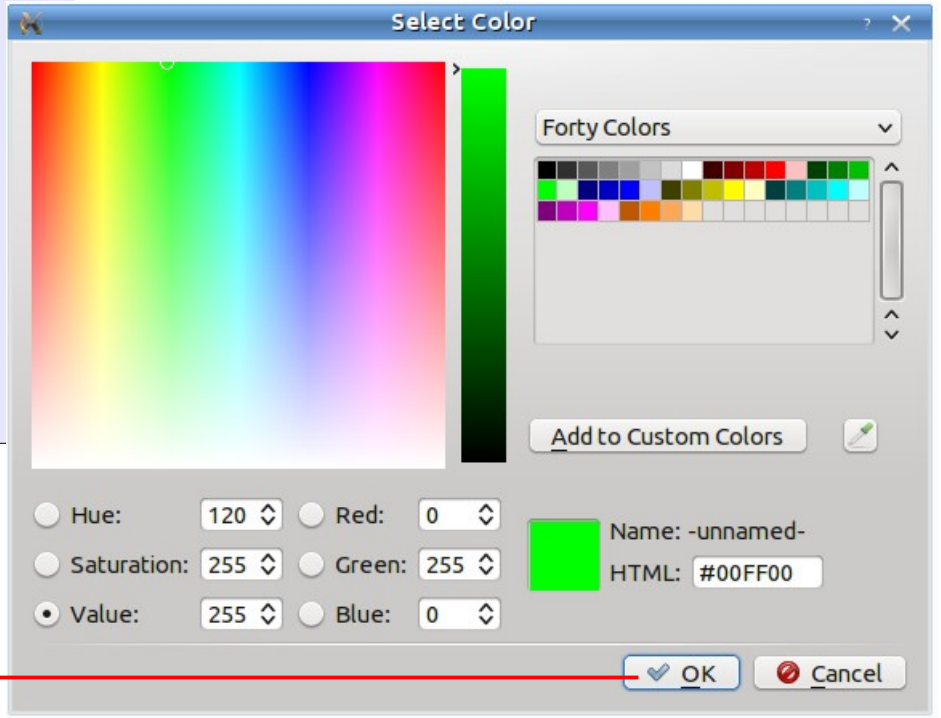



```
...
private slots:
 void setItem()
 {
   QStringList items;
   items << QString::fromUtf8("Printemps") << QString::fromUtf8("Été") <<
QString::fromUtf8("Automne") << QString::fromUtf8("Hiver");
   bool ok;
   QString item = QInputDialog::getItem(this, "QInputDialog::getItem()",
"Dans quelle saison sommes-nous ?", items, 0, false, &ok);
   if (ok && !item.isEmpty())
       QMessageBox::information(this, QString::fromUtf8("Réponse"), item);
 }
...
```

QDialog : exemple n°4 (3/6)

- Utilisation de la classe **QColorDialog** qui fournit un dialogue pour la spécification des couleurs. Cela permet ici à l'utilisateur de choisir la couleur de fond (getColor)pour l'application.

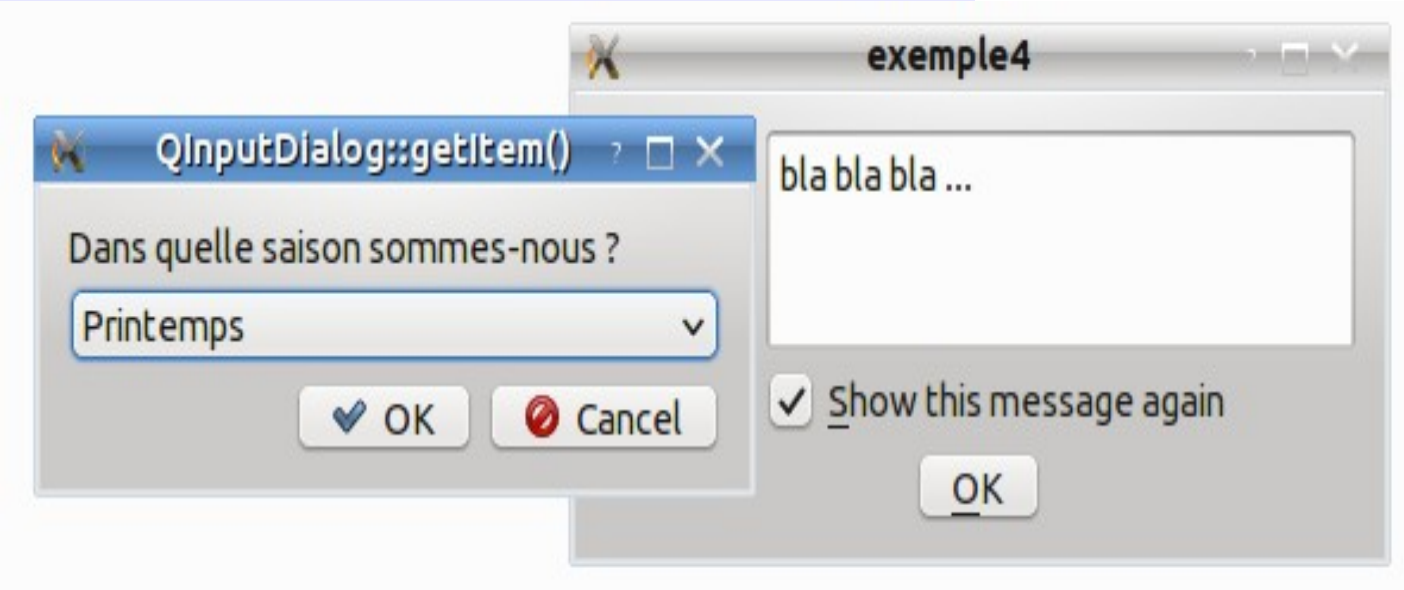
```
...
void setColor()
{
   QColor color =
QColorDialog::getColor(Qt::green, this);
   if (color.isValid())
   {
      this->setPalette(QPalette(color));
      this->setAutoFillBackground(true);
   }
}
...
```



QDialog : exemple n°4 (4/6)

- Utilisation de la classe **QErrorMessage** qui fournit une boîte de dialogue affichant un message d'erreur (showMessage()). C'est un exemple aussi de boîte de dialogue non modale.

```
    void showMessage()
    {
      errorMessageDialog->showMessage("bla bla bla ...");
    }
};
```



QDialog : exemple n°4 (5/6)

- Dans cet exemple, la macro Q_OBJECT est nécessaire dès qu'un dispositif propre à Qt est utilisé (ici **private slot**). L'outil **moc** permet l'implémentation de ces mécanismes.

- Si vous fournissez vos classes sous la forme de fichiers séparés : déclaration (.h) et définition (.cpp) alors le moc sera appelé automatiquement. Il génèrera un fichier **`moc_nomclasse.cpp`** à partir de votre fichier **`nomclasse.h`**. Le fichier sera ensuite automatiquement compilé et lié grâce au **Makefile** généré par **qmake**.

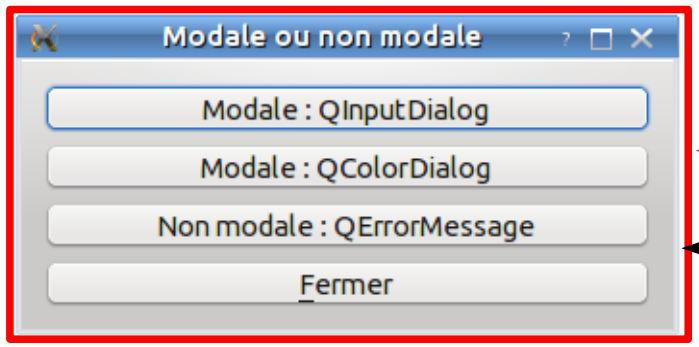
- Exemple de règle présente dans un **Makefile** pour Qt :

```
moc_mywidget.cpp: mywidget.h
  moc $(DEFINES) $(INCPATH) mywidget.h -o moc_mywidget.cpp
moc_mywidget.o: moc_mywidget.cpp
  $(CXX) -c $(CXXFLAGS) $(INCPATH) -o moc_mywidget.o moc_mywidget.cpp
```

QDialog : exemple n°4 (6/6)

- Pour certains exemples du cours, l'ensemble du programme est fourni dans un seul fichier par souci de simplicité de lecture. Il faut alors appeler l'outil **moc** manuellement pour générer un fichier **moc_MyDialog.h** qu'il faut ensuite inclure :

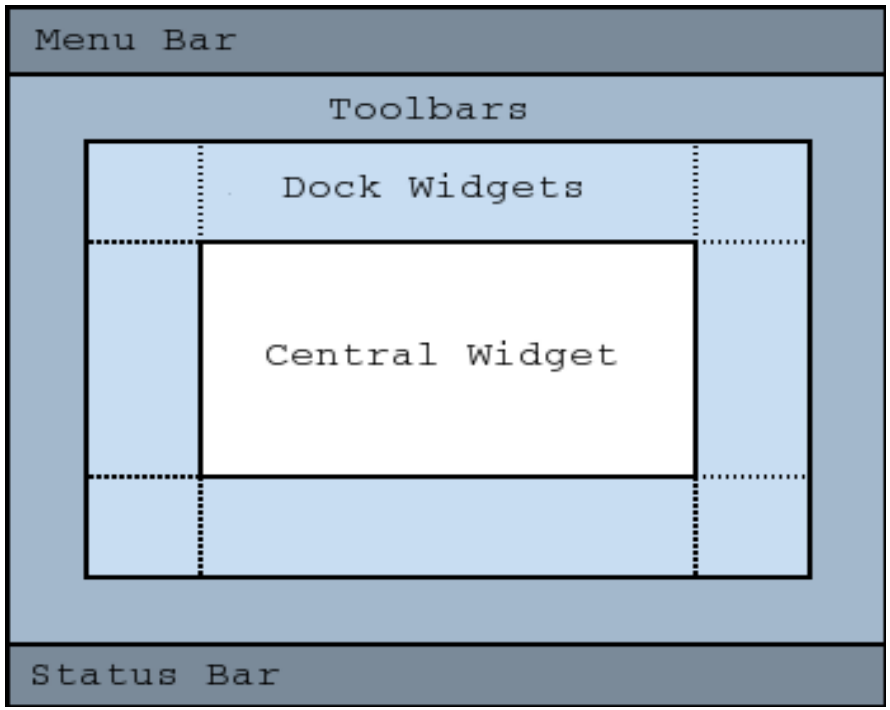
```
moc MyDialog.cpp -o moc_MyDialog.h
```




```
...

#include "moc_MyDialog.h"

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

  MyDialog myDialog;

  myDialog.show();

  return app.exec();
}
```

La classe QMainWindow

- La classe **QMainWindow** offre une fenêtre d'application principale.
- Une fenêtre principale fournit un cadre pour la construction de l'interface utilisateur d'une application.
- QMainWindow a sa propre mise en page à laquelle vous pouvez ajouter **QToolBars**, **QDockWidgets**, un **QMenuBar**, et un **QStatusBar**. Le tracé a une zone centrale qui peut être occupée par n'importe quel type de widget.



- Le widget central sera généralement un widget standard de Qt comme un **QTextEdit** ou un **QGraphicsView**. Les widgets personnalisés peuvent également être utilisés pour des applications avancées. On définit le widget central avec **setCentralWidget**().

QMainWindow : exemple n°1

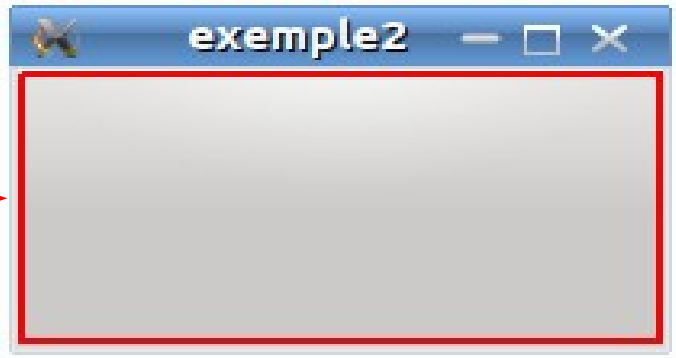
- Pour créer sa propre application principale, il suffit de créer une classe qui **hérite de QMainWindow** et de l'**afficher avec show()** :

```
#include <QApplication>
#include <QtGui>

class MyMainWindow : public QMainWindow
{
   public:
      MyMainWindow() {}
};

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

  MyMainWindow myMainWindow;

  myMainWindow.show();

  return app.exec();
}
```

QMainWindow : exemple n°2

- Le **widget central** sera généralement un widget standard de Qt comme un QTextEdit ou un QGraphicsView. Les widgets personnalisés peuvent également être utilisés pour des applications avancées. On définit le widget central avec **setCentralWidget()** :

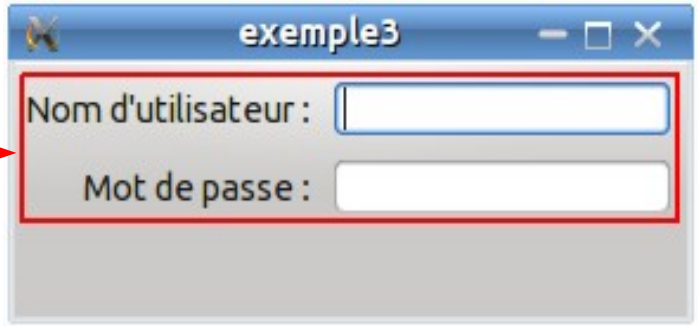
```
#include <QApplication>
#include <QtGui>

class MyMainWindow : public QmainWindow {
  public:
   MyMainWindow() {
    QWidget *centralWidget = new QWidget;
    setCentralWidget(centralWidget);
   }
};

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  MyMainWindow myMainWindow;
  myMainWindow.show();
  return app.exec();
}
```



QMainWindow : exemple n°3

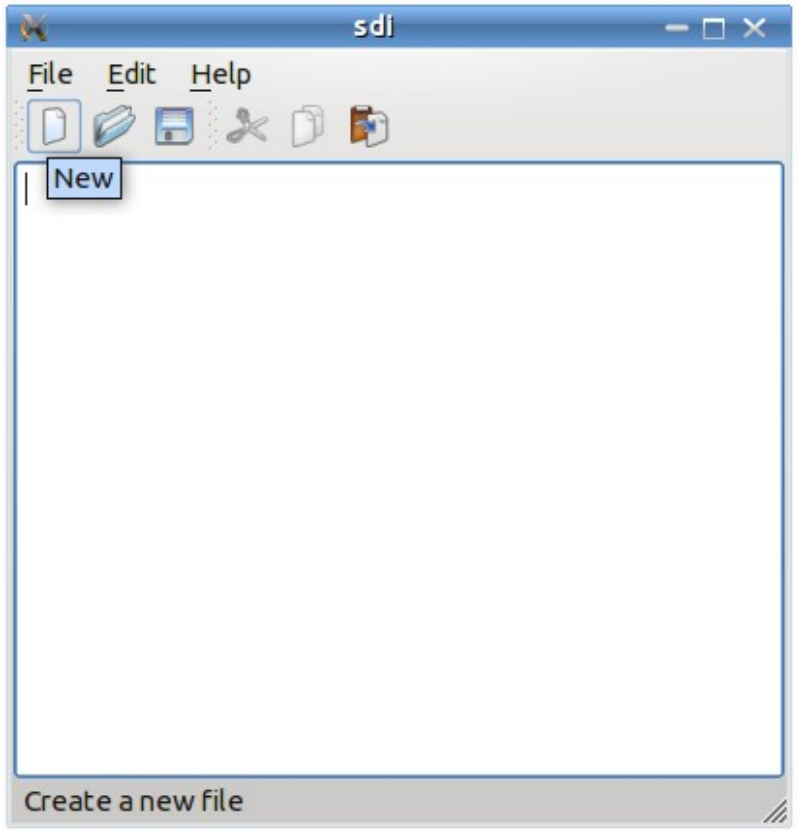
- Comme on l'a vu précédemment, on peut maintenant ajouter d'autres widgets dans ce widget principal :

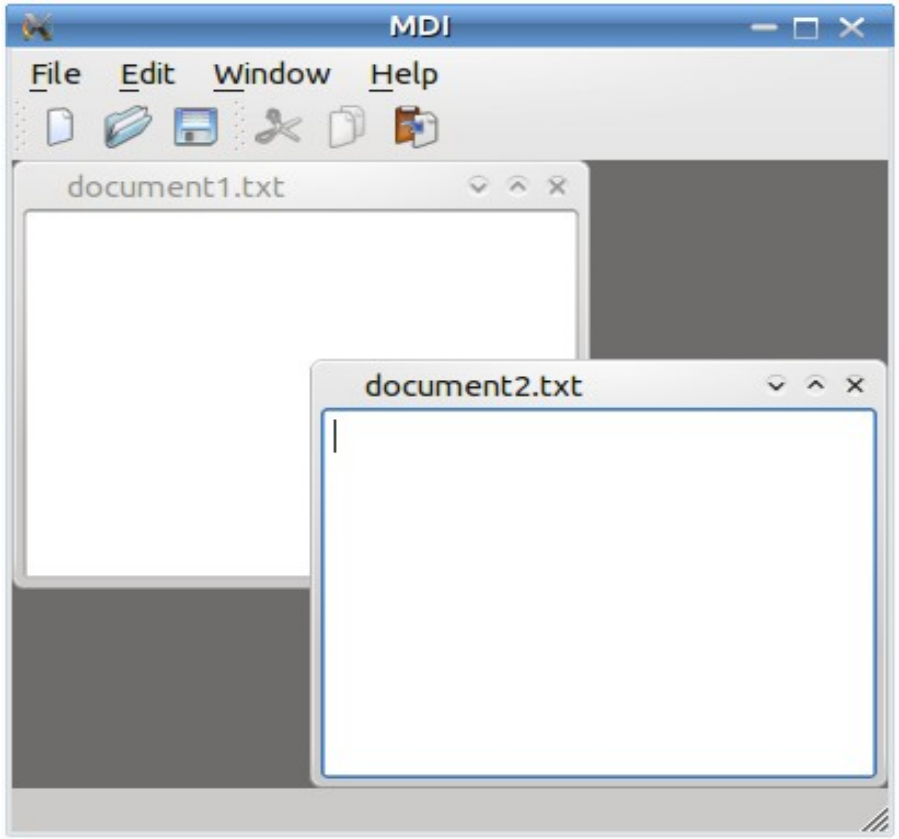
```
#include <QApplication>
#include <QtGui>
class MyMainWindow : public QmainWindow {
  public:
   MyMainWindow() {
    QWidget *centralWidget = new QWidget;

    QLineEdit *login = new QLineEdit;
    QLineEdit *password = new QLineEdit;
    QFormLayout *formLayout = new QFormLayout;
    formLayout->addRow("Nom d'utilisateur : ",
                       login);
    formLayout->addRow("Mot de passe : ",
                       password);
    centralWidget->setLayout(formLayout);
    setCentralWidget(centralWidget);
   }
};
```



SDI ou MDI

- La fenêtre principale a soit une interface unique (**SDI** pour *Single Document Interface*) ou multiples (**MDI** pour *Multiple Document Interface*). Pour créer des applications MDI dans Qt, on utilisera un **QMdiArea** comme widget central.

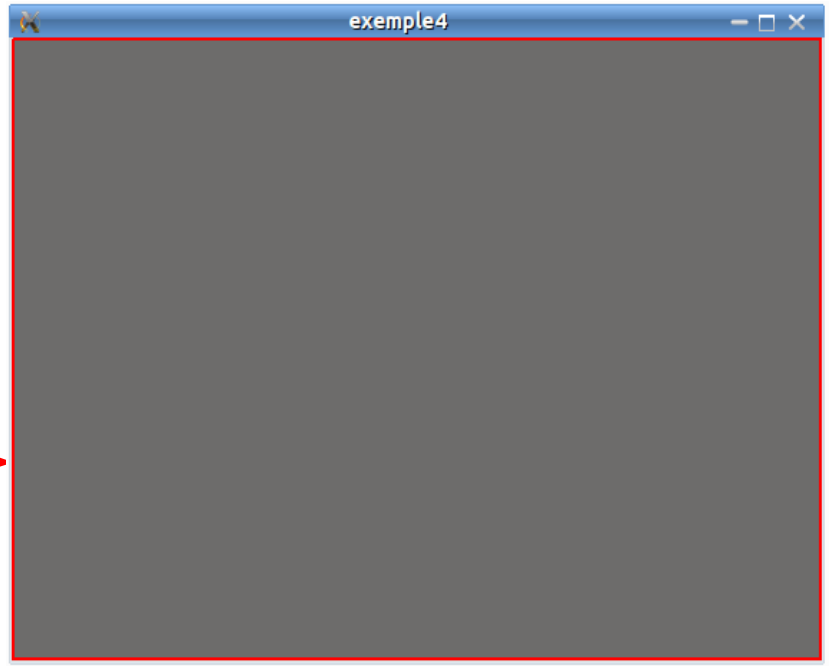




QMainWindow : exemple n°4

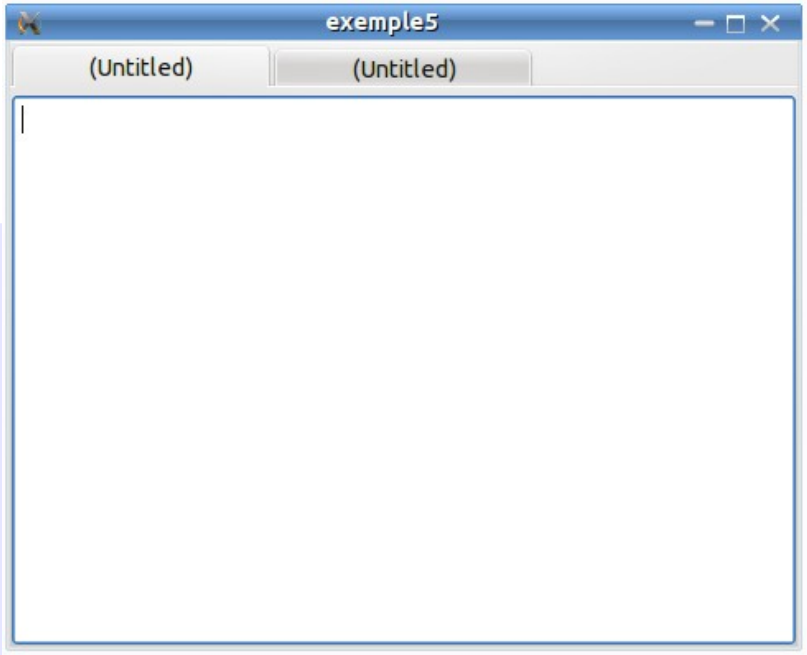
- Le widget **QMdiArea** est utilisé comme le widget central de QMainWindow pour créer des applications MDI :

```
#include <QApplication>
#include <QtGui>
class MyMainWindow : public QMainWindow
{
  public:
   MyMainWindow()
   {
     QMdiArea *mdiArea = new QMdiArea;
     setCentralWidget(mdiArea);
   }
};
```



QMainWindow : exemple n°5

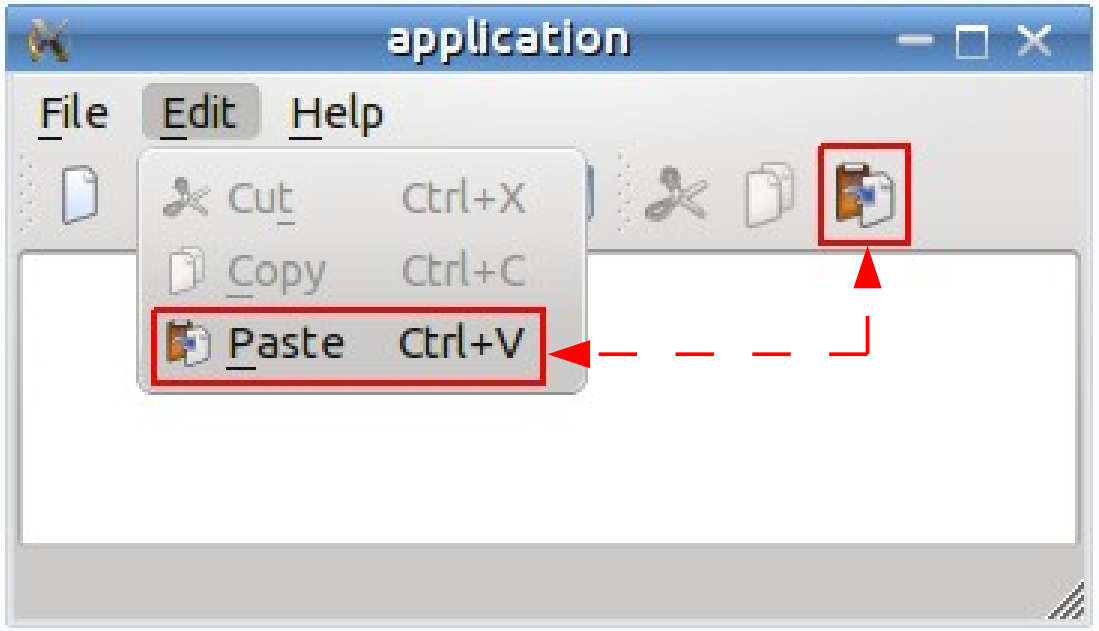
- Les sous-fenêtres de QMdiArea sont des instances de **QMdiSubWindow**. Elles sont ajoutées à une zone MDI avec **addSubWindow()**.




```
#include <QApplication>
#include <QtGui>

class MyMainWindow : public QMainWindow {
  public:
   MyMainWindow() {
     QMdiArea *mdiArea = new QMdiArea;
     QTextEdit *textEdit1 = new QTextEdit;
     QTextEdit *textEdit2 = new QTextEdit;
     QMdiSubWindow *mdiSubWindow1 = mdiArea->addSubWindow(textEdit1);
     QMdiSubWindow *mdiSubWindow2 = mdiArea->addSubWindow(textEdit2);

     // ou : QMdiArea::SubWindowView
     mdiArea->setViewMode(QMdiArea::TabbedView);
     setCentralWidget(mdiArea);
   }
};
```

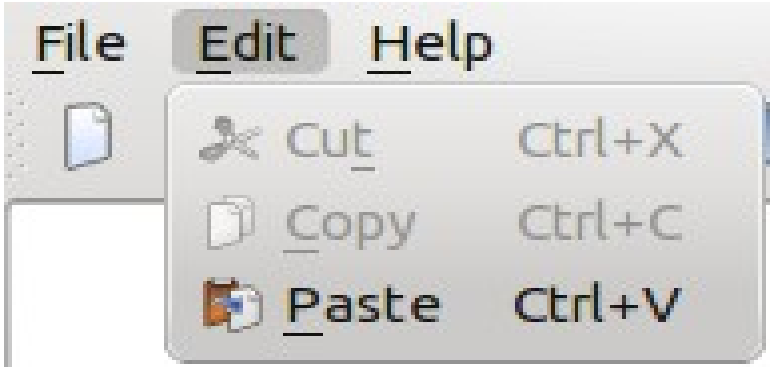
La classe QAction

- La classe **QAction** fournit une interface abstraite pour décrire une action (= commande) qui peut être insérée dans les widgets.
- Dans de nombreuses applications, des commandes communes peuvent être invoquées via des menus, boutons, et des raccourcis clavier. Puisque l'utilisateur s'attend à ce que chaque commande soit exécutée de la même manière, indépendamment de l'interface utilisateur utilisée, il est utile de représenter chaque commande comme une action.
- Les actions peuvent être ajoutés aux menus et barres d'outils, et seront automatiquement synchronisées.



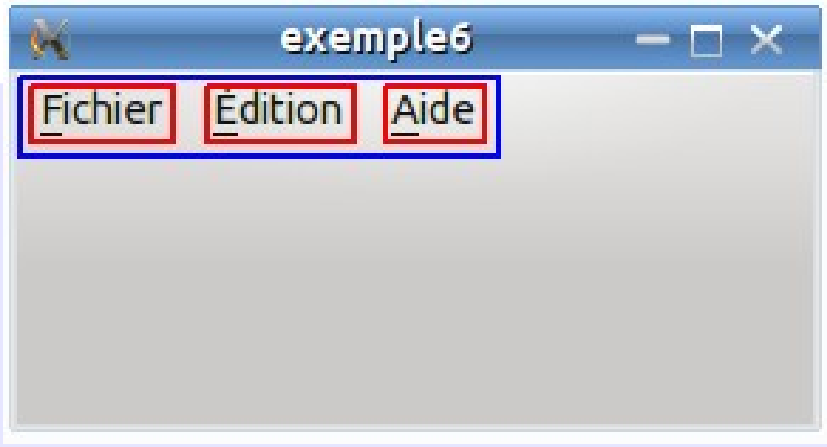
La classe QMenu

- La classe **QMenu** fournit un widget pour une utilisation dans les barres de menus et les menus contextuels. Un menu contextuel est un menu qui s'affiche lorsqu'on fait un clic droit sur un widget.
- Un widget menu est un menu de sélection. Il peut être soit un menu déroulant dans une barre de menu ou un menu contextuel autonome. Les menus déroulants sont indiquées par la barre de menu lorsque l'utilisateur clique sur l'élément concerné ou appuie sur la touche de raccourci spécifié.
- Qt implémente donc les menus avec QMenu et QMainWindow les garde dans un **QMenuBar**. On utilise QMenuBar::**addMenu()** pour insérer un menu dans une barre de menu.
- La classe QMenuBar fournit une barre de menu horizontale. Une barre de menu se compose d'une liste d'éléments de menu déroulant.



QMainWindow : exemple n°6

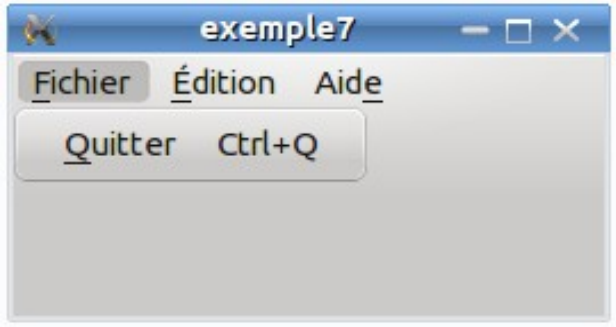
- On peut ajouter de nouveaux menus à la barre de menus de la fenêtre principale en appelant **menuBar()** qui retourne la QMenuBar de la fenêtre, puis ajoutez un menu avec **QMenuBar::addMenu()** :




```
#include <QApplication>
#include <QtGui>
class MyMainWindow : public QMainWindow
{
    public:
        MyMainWindow()
        {
          QMenu *fileMenu = new QMenu(tr("&Fichier"), this);
          menuBar()->addMenu(fileMenu);

          QMenu *editMenu = new QMenu(QString::fromUtf8("&Édition"), this);
          menuBar()->addMenu(editMenu);

          QMenu *helpMenu = new QMenu(tr("&Aide"), this);
          menuBar()->addMenu(helpMenu);
        }
};
```

QMainWindow : exemple n°7

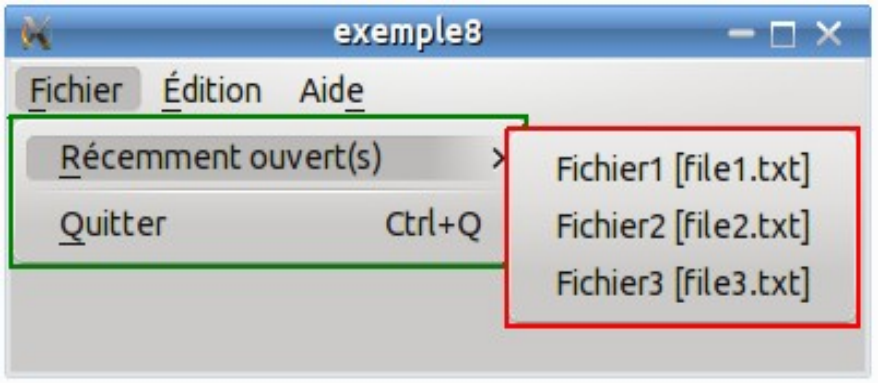
- On peut soit **créer une instance de QAction** puis l'ajouter avec **addAction()** soit **créer la QAction directement** en utilisant **addAction()** :




```
class MyMainWindow : public QmainWindow {
  public:
   MyMainWindow() {
    QMenu *fileMenu = new QMenu(tr("&Fichier"),this);
    menuBar()->addMenu(fileMenu);
    fileMenu->addAction(tr("&Quitter"), qApp, SLOT(quit()),
                        QKeySequence::Quit);
    ...
    QMenu *helpMenu = new QMenu(tr("Aid&e"), this);
    menuBar()->addMenu(helpMenu);
    QAction *actionHelp = new QAction(QString::fromUtf8("À propos de
                                      Qt"), this);
    helpMenu->addAction(actionHelp);
    actionHelp->setShortcut(QKeySequence(Qt::Key_F1)); //ou :
    //actionHelp->setShortcut(QKeySequence(QKeySequence::HelpContents));
    connect(actionHelp, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
   }
};
```

QMainWindow : exemple n°8 (1/4)

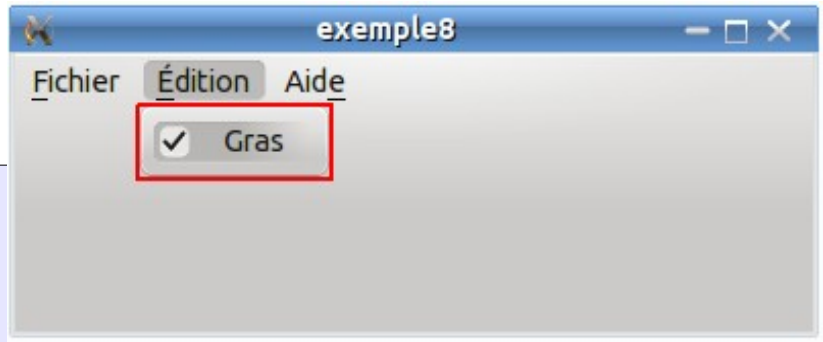
- On peut ajouter un **QMenu** à un **Qmenu** avec **addMenu()** pour créer un **sous-menu** :




```
class MyMainWindow : public QmainWindow {
  public:
   MyMainWindow() {
    QMenu *fileMenu = new QMenu(tr("&Fichier"), this);
    menuBar()->addMenu(fileMenu);

    QMenu *fileSubMenu = fileMenu->addMenu(QString::fromUtf8("&Récemment
                                                    ouvert(s)"));

    fileSubMenu->addAction("Fichier1 [file1.txt]");
    fileSubMenu->addAction("Fichier2 [file2.txt]");
    fileSubMenu->addAction("Fichier3 [file3.txt]");

    //ajout d'une barre de séparation
    fileMenu->addSeparator();
    ...
```

QMainWindow : exemple n°8 (2/4)

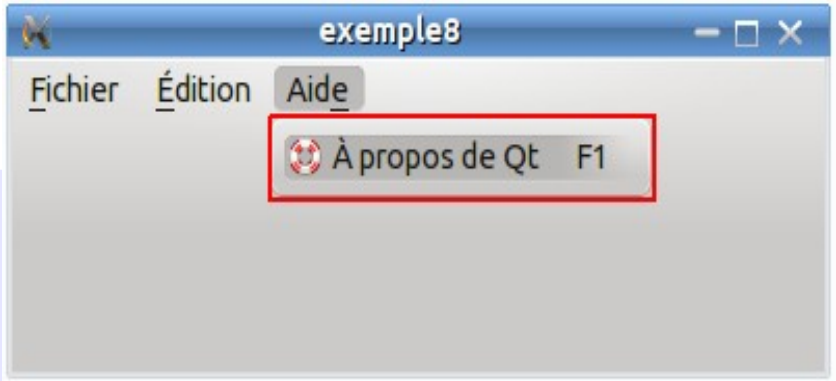
- Une action peut avoir 2 états (activée, désactivée) en utilisant **setCheckable()** :




```
class MyMainWindow : public QMainWindow {
  public:
   MyMainWindow() {
    ...
    QMenu *editMenu = new QMenu(QString::fromUtf8("&Édition"), this);
    menuBar()->addMenu(editMenu);
    QAction *actionEdit = new QAction(QString::fromUtf8("Gras"), this);
    actionEdit->setCheckable(true);
    actionEdit->setChecked(true);
    editMenu->addAction(actionEdit);
    ...
```

QMainWindow : exemple n°8 (3/4)

- Chaque action d'un menu ou d'une barre d'outils peut avoir une icône **QIcon**.




```
class MyMainWindow : public QMainWindow {
  public:
   MyMainWindow() {
    ...
    QMenu *helpMenu = new QMenu(tr("Aid&e"), this);
    menuBar()->addMenu(helpMenu);

    QAction *actionHelp = new QAction(QIcon("help.png"),
                                      QString::fromUtf8("À propos de Qt"), this);
    helpMenu->addAction(actionHelp);
    actionHelp->setShortcut(QKeySequence(QKeySequence::HelpContents));
    //actionHelp->setIcon(QIcon(":/help.png"));
    connect(actionHelp, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
    ...
```

```
<!DOCTYPE RCC><RCC version="1.0">
 <qresource><file>help.png</file></qresource>
</RCC>
```

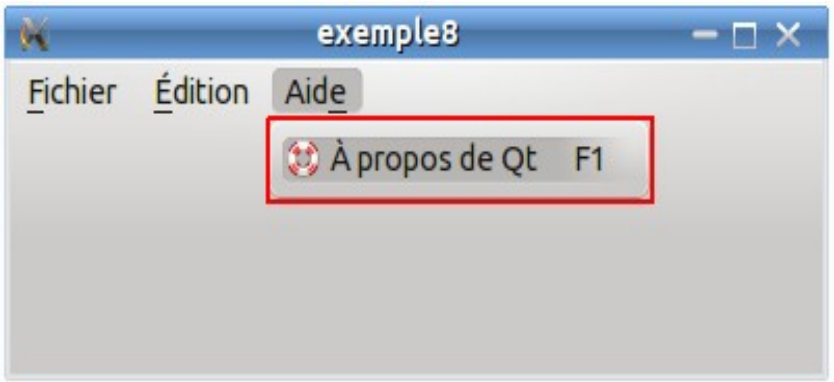
QMainWindow : exemple n°8 (4/4)

- On peut aussi utiliser un fichier ressource de Qt (**.qrc**) pour référencer l'image de l'icône. L'outil **rcc** est alors utilisé pour incorporer les ressources dans l'application Qt au cours du processus de construction. rcc génére un fichier C++ à partir des données spécifiées dans le fichier .qrc.

```
...
actionHelp->setIcon(QIcon(":/help.png"));
...
```

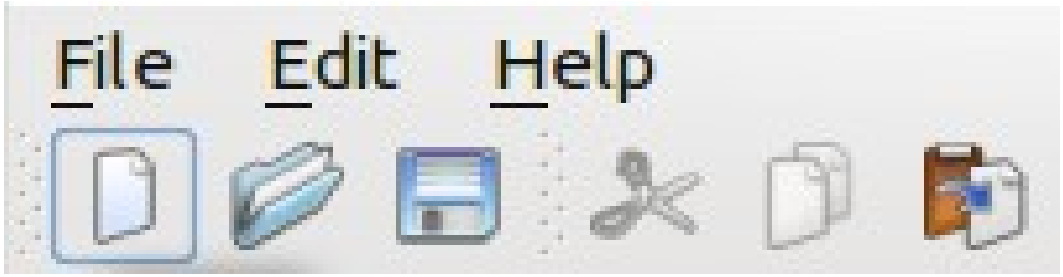
exemple8.qrc

```
<!DOCTYPE RCC>
<RCC version="1.0">
  <qresource>
    <file>help.png</file>
  </qresource>
</RCC>
```



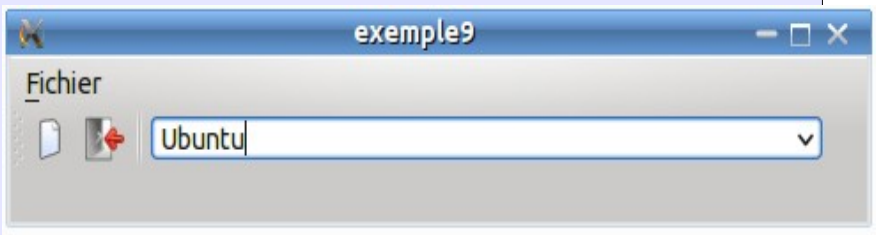
La classe QToolBar

- La classe **QToolBar** fournit une **barre d'outils** qui contient un ensemble de contrôles (généralement des icônes) et située sous les menus.
- Pour ajouter une barre d'outils, on doit tout d'abord appeler la méthode **addToolBar()** de QMainWindow.
- Avec Qt, la barre d'outils utilise des actions pour construire chacun des éléments de celle-ci. Les boutons de la barre d'outils sont donc insérés en ajoutant des actions et en utilisant **addAction()** ou insertAction().
- Les boutons peuvent être séparés en utilisant addSeparator() ou insertSeparator().
- Mais on peut aussi insérer un widget (comme QSpinBox, QDoubleSpinBox ou QComboBox) à l'aide de **addWidget()** ou insertWidget().
- Quand un bouton de la barre est enfoncée, il émet le signal actionTriggered().



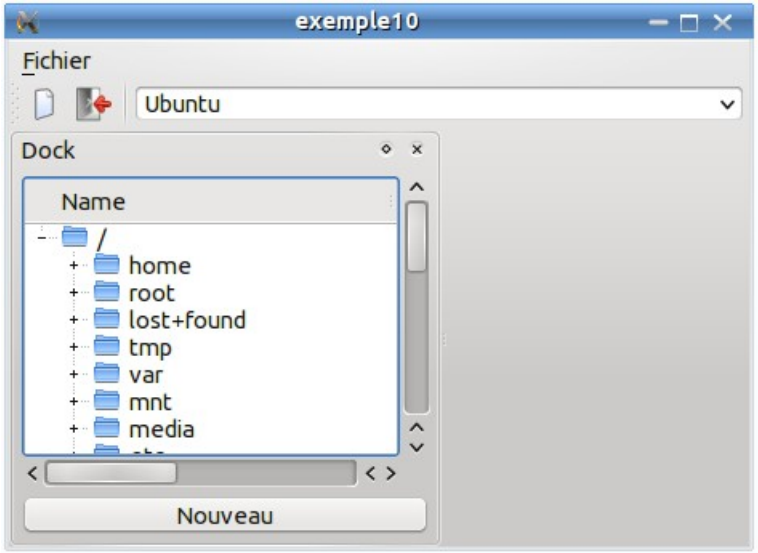
QMainWindow : exemple n°9

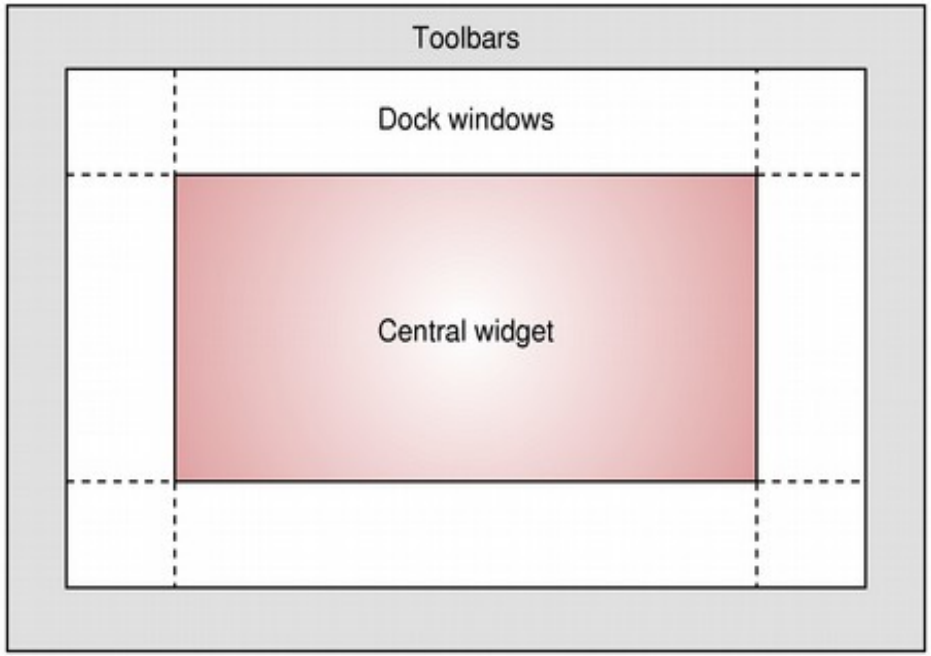
```
class MyMainWindow : public QmainWindow {
  public: MyMainWindow() {
    QMenu *fileMenu = new QMenu(tr("&Fichier"), this);
    menuBar()->addMenu(fileMenu);
    QAction *actionNouveau = new QAction(QIcon(":/images/new.png"),
                                         tr("&Nouveau"), this);
    actionNouveau->setShortcuts(QKeySequence::New);
    fileMenu->addAction(actionNouveau);
    fileMenu->addSeparator();
    QAction *actionQuit = fileMenu->addAction(tr("&Quitter"), qApp,
                                      SLOT(quit()), QKeySequence::Quit);
    actionQuit->setIcon(QIcon(":/images/quit.png"));

    QToolBar *fileToolBar = addToolBar("Fichier");
    //fileToolBar->setMovable(false);
    //fileToolBar->setFloatable(false);
    fileToolBar->addAction(actionNouveau);
    fileToolBar->addAction(actionQuit);
    fileToolBar->addSeparator();
    QFontComboBox *fontComboBox = new QFontComboBox;
    fileToolBar->addWidget(fontComboBox);
   }};
```



La classe QDockWidget

- La classe **QDockWidget** fournit un widget qui peut être ancré dans une QMainWindow ou "flotter" comme une fenêtre de haut niveau sur le bureau.
- QDockWidget fournit le concept de *dock windows* (palettes d'outils ou de fenêtres d'utilité). Ces *dock windows* sont des fenêtres secondaires (ou mini-fenêtres) placés dans la zone autour du widget central d'une QMainWindow.
- Beaucoup d'applications connues les utilisent : Qt Designer, OpenOffice, Photoshop, Code::Blocks , ...

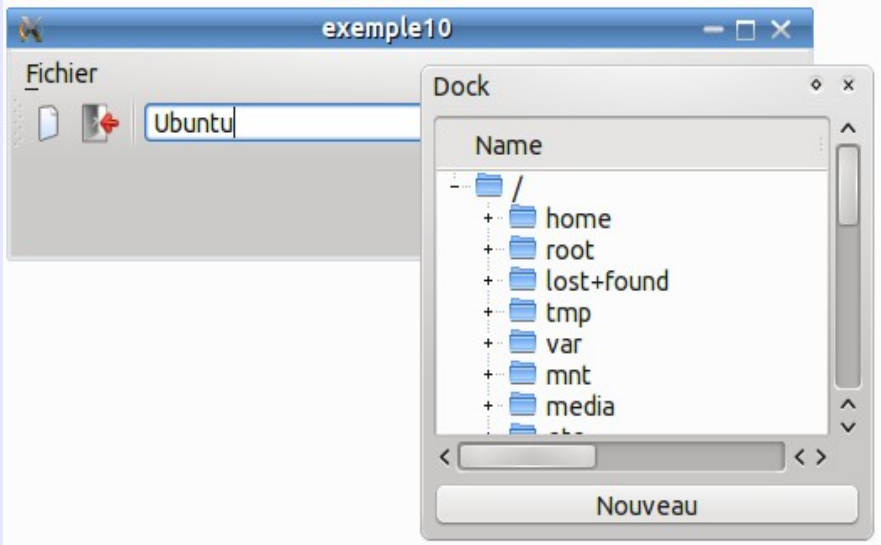




QMainWindow : exemple n°10

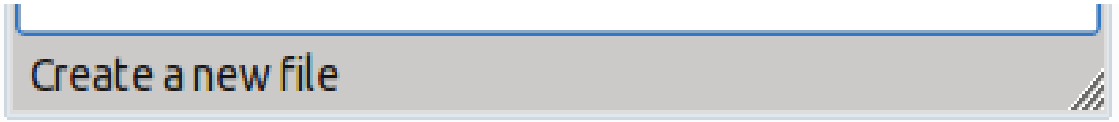
- On peut placer ses propres widgets dans une fenêtre “dockable” :

```
class MyMainWindow : public QMainWindow {
 public:
  MyMainWindow() { ...
   QDockWidget *dockWidget = new QDockWidget("Dock", this);
   addDockWidget(Qt::LeftDockWidgetArea, dockWidget);

   QWidget *dockContenu = new QWidget;
   dockWidget->setWidget(dockContenu);

 QVBoxLayout *dockLayout = new QVBoxLayout;
   QFileSystemModel *model = new
                        QFileSystemModel;
   model->setRootPath(QDir::currentPath());
   QTreeView *vueArbre = new QTreeView;
   vueArbre->setModel(model);
   dockLayout->addWidget(vueArbre);
   QPushButton *pushButton = new
                        QPushButton("Nouveau");
   dockLayout->addWidget(pushButton);
   dockContenu->setLayout(dockLayout);
   ...
```



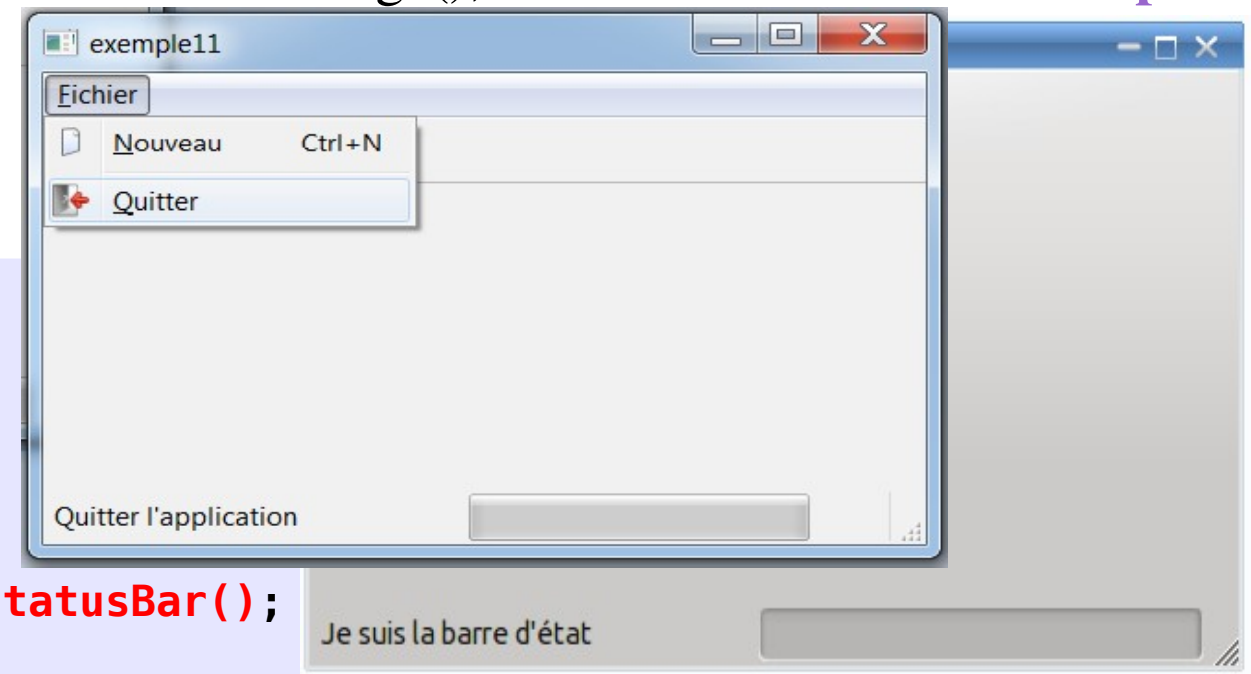
La classe QStatusBar

- La classe **QStatusBar** fournit une barre horizontale appropriée pour la présentation des informations d'état. QStatusBar permet d'afficher différents types d'indicateurs.
- Une barre d'état peut afficher trois types de messages différents :
  - temporaire : affiché brièvement. Exemple : utilisé pour afficher les textes explicatifs de la barre d'outils ou des entrées de menu.
  - normal : affiché tout le temps, sauf quand un message temporaire est affiché. Exemple : utilisé pour afficher la page et le numéro de ligne dans un traitement de texte.
  - permanent : jamais caché. Exemple : utilisé pour des indications de mode important comme le verrouillage des majuscules.
- La barre d'état peut être récupéré à l'aide de QMainWindow::statusBar() et remplacé à l'aide de QMainWindow::setStatusBar().



QMainWindow : exemple n°11

- On utilisez **showMessage()** pour afficher un message temporaire. Pour supprimer un message temporaire, il faut appeler clearMessage(), ou **fixer une limite de temps** lors de l'appel showMessage().



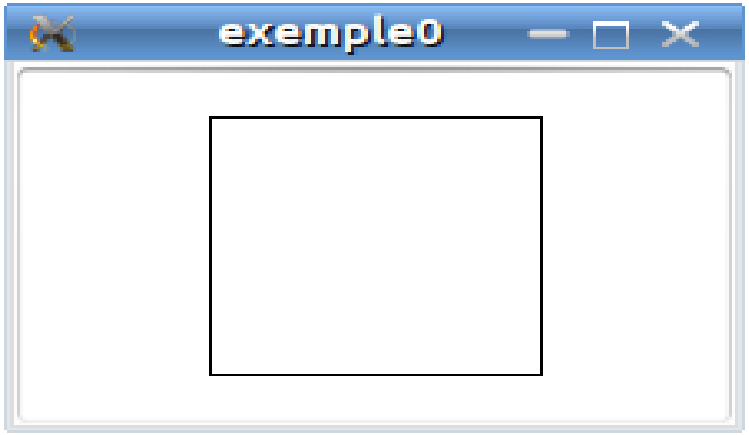

```
class MyMainWindow
        : public QMainWindow
{
 public:
  MyMainWindow() {
    ...
    QStatusBar *barreEtat = statusBar();

    QProgressBar *progression = new QProgressBar;
    barreEtat->addPermanentWidget(progression);

    barreEtat->showMessage(QString::fromUtf8("Je suis la barre d'état"),
                           2000);
    ...
```

QMainWindow : exemple n°12

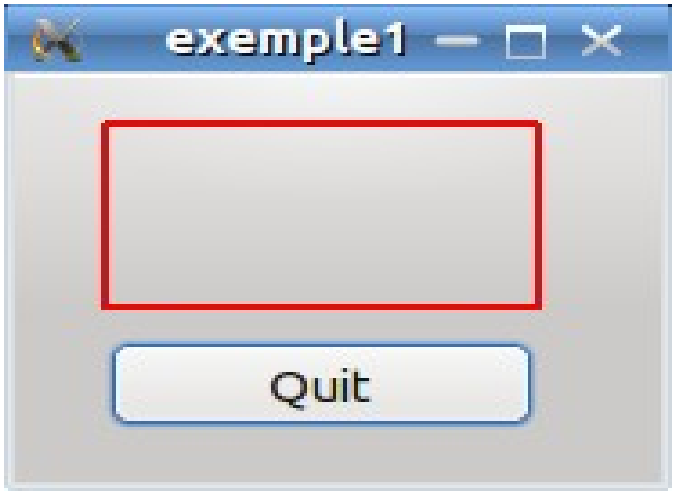
```
#include <QApplication>
#include <QtGui>
#include <QtWebKit>

class MyMainWindow : public QmainWindow {
 public:
  MyMainWindow() {
   QWidget *centralWidget = new QWidget;
   QToolBar *urlToolBar = addToolBar("URL");
   QLineEdit *urlEdit = new QLineEdit;
   QLabel *label = new QLabel(tr("Adresse :"));
   urlToolBar->addWidget(label);
   urlToolBar->addWidget(urlEdit);
   webView = new QWebView;
   QVBoxLayout *layout = new QVBoxLayout;
   layout->addWidget(webView);
   centralWidget->setLayout(layout);
   webView->load(QUrl("http://www.google.fr/"));
   setCentralWidget(webView);
  }
 private:
  QWebView *webView;
};
```

Graphique 2D

- Deux approches pour dessiner en 2D dans Qt :
  - un modèle fonctionnel basé sur **QPainter**
  - un modèle objet basé sur le framework **Graphics View**



Exemple utilisant l'architecture Graphics View



Exemple utilisant QPainter

Le framework Graphics View (1/4)

- Le framework **Graphics View** se décompose en 3 parties essentielles :
  - La scène
  - La vue
  - Les éléments graphiques
- L'architecture Graphics View offre une approche basée sur le pattern **modèle-vue**. Plusieurs vues peuvent observer une scène unique constituée d'éléments de différentes formes géométriques.

Le framework Graphics View (2/4)

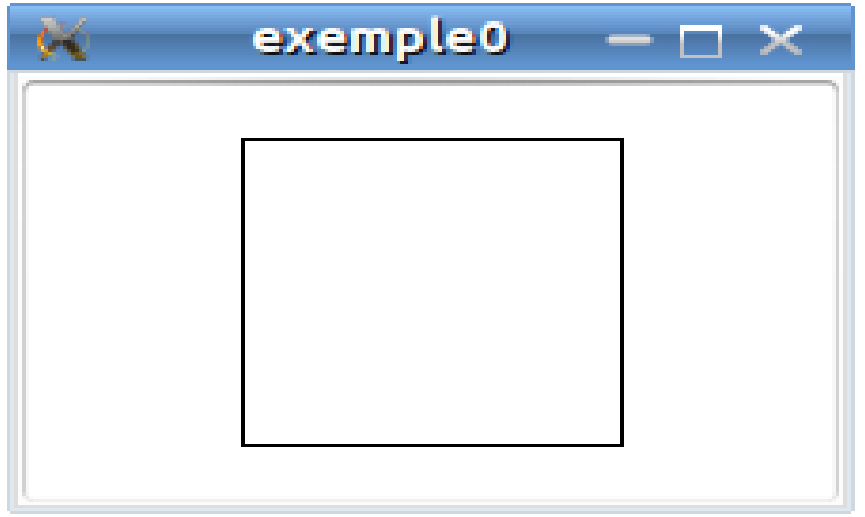
- La classe **QGraphicsScene** fournit la scène pour l'architecture Graphics View. La scène a les responsabilités suivantes :
  - fournir une interface rapide pour gérer un grand nombre d'éléments,
  - propager les événements à chaque élément,
  - gérer les états des éléments (telles que la sélection et la gestion du focus)
  - et fournir des fonctionnalités de rendu non transformée (principalement pour l'impression).
- La classe **QGraphicsView** fournit la vue "widget" qui permet de visualiser le contenu d'une scène.

Le framework Graphics View (3/4)

- La classe **QGraphicsItem** est la classe de base pour les éléments graphiques dans une scène.
- Elle fournit plusieurs éléments standard pour les formes typiques, telles que :
  - des rectangles (QGraphicsRectItem),
  - des ellipses (QGraphicsEllipseItem) et
  - des éléments de texte (QGraphicsTextItem).
- Mais les fonctionnalités les plus puissantes seront disponibles lorsque on écrira un élément personnalisé.
- Entre autres choses, QGraphicsItem supporte les fonctionnalités suivantes : les événements souris et clavier, le glissez et déposez (drag and drop), le groupement d'éléments, la détection des collisions.

Le framework Graphics View (4/4)

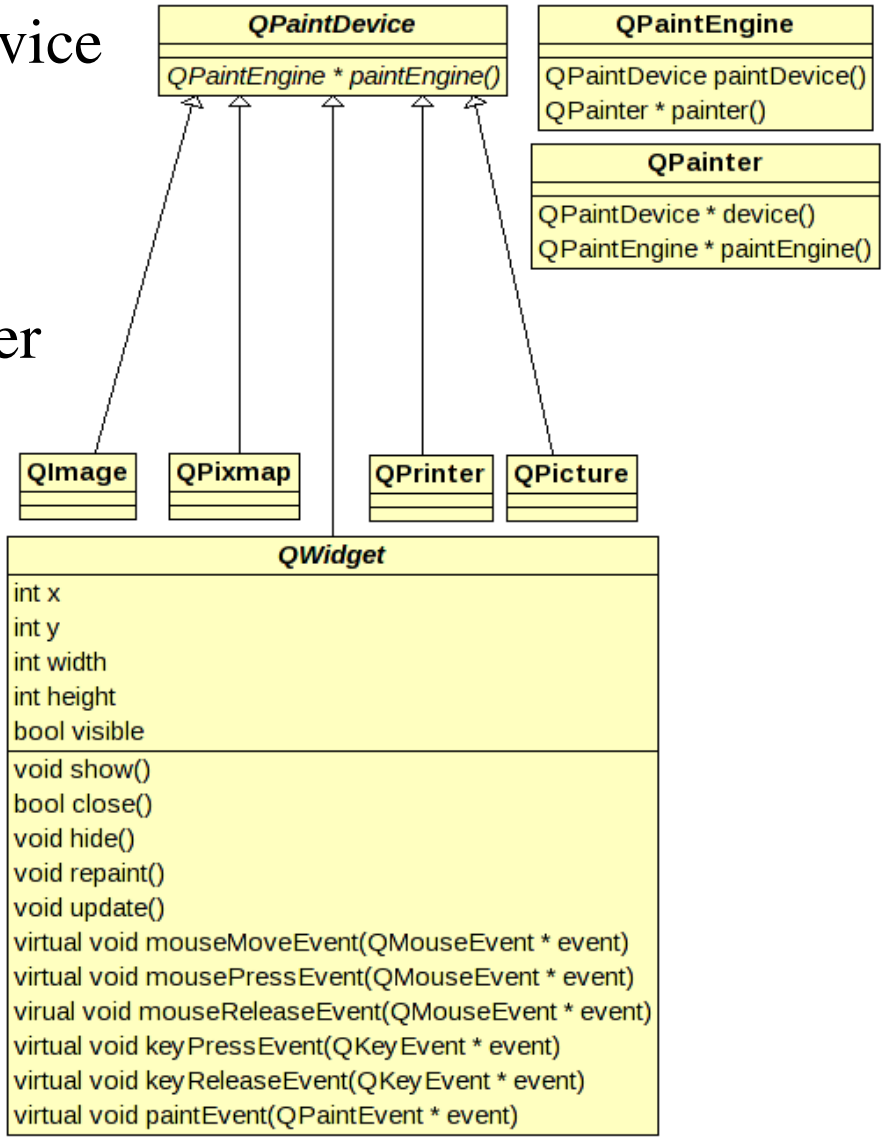
- La scène sert de conteneur pour les objets QGraphicsItem. La classe **QGraphicsView** fournit la vue "widget" qui permet de visualiser le contenu d'une scène.




```
#include <QApplication>
#include <QGraphicsScene>
#include <QGraphicsRectItem>
#include <QgraphicsView>

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

   QGraphicsScene scene;

   QGraphicsRectItem *rect = scene.addRect(QRectF(0, 0, 100, 100));

   QGraphicsView view(&scene);

   view.show();

   return app.exec();
}
```

Dessin 2D

- En collaboration avec les classes QPaintDevice et QPaintEngine, QPainter est la base du système de dessin de Qt.
- **QPainter** est la classe utilisée pour effectuer les opérations de dessin.
- **QPaintDevice** représente un dispositif qui peut être peint en utilisant un QPainter.
- **QPaintEngine** fournit le moteur de rendu et l'interface (abstraite) que QPainter utilise pour dessiner sur différents types de dispositifs suivant la plate-forme utilisée.



QPainter (1/12)

- La classe QPainter est la classe de base de dessin bas niveau sur les widgets et les autres dispositifs de dessins.
- QPainter fournit des fonctions hautement optimisées : il peut tout dessiner des lignes simples à des formes complexes.
- QPainter peut fonctionner sur n'importe quel objet qui hérite de la classe QPaintDevice.
- L'utilisation courante de QPainter est à l'intérieur de la méthode paintEvent() d'un widget : construire, personnaliser (par exemple le pinceau), dessiner et détruire l'objet QPainter après le dessin.

QPainter (2/12)

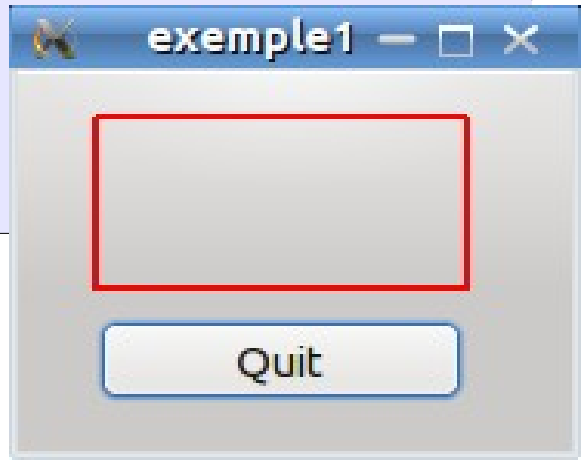
- Un **widget** est "repeint" :
  - Lorsque une fenêtre passe au dessus
  - Lorsque l'on déplace le composant
  - ...
  - Lorsque l'on le lui demande explicitement :
    - **`repaint()`** entraîne un rafraichissement immédiat
    - **`update()`** met une demande de rafraîchissement en file d'attente
- Dans tous les cas, c'est la méthode paintEvent qui est appelée :

```
void paintEvent(QPaintEvent* e);
```

- Pour dessiner dans un widget, il faut donc redéfinir QWidget::paintEvent().

QPainter : exemple 1 (3/12)

```
class MyWidget : public QWidget
{
  public:
   MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}

   void paintEvent(QPaintEvent* e)
   {
     QWidget::paintEvent(e); // effectue le comportement standard

     QPainter painter(this); // construire

     painter.setPen( QPen(Qt::red, 2) ); // personnaliser

     painter.drawRect( 25, 15, 120, 60 ); // dessiner

   } // détruire
};
```

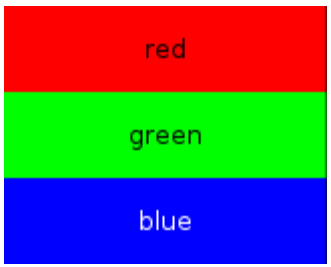


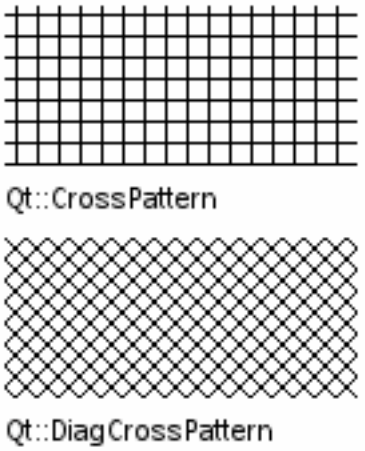
QPainter (4/12)

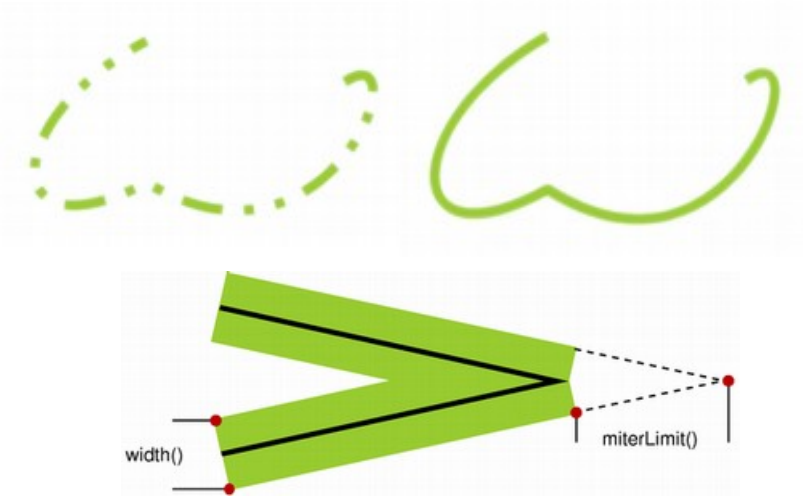
- La classe QPainter fournit de nombreuses méthodes :
  - setPen( ) : lignes et contours
  - setBrush( ) : remplissage
  - setFont( ) : texte
  - setTransform( ), etc. : transformations affines
  - setClipRect/Path/Region( ) : clipping (découpage)
  - setCompositionMode( ) : composition
- Lignes et contours : drawPoint(), drawPoints(), drawLine(), drawLines(), drawRect(), drawRects(), drawArc(), drawEllipse(), drawPolygon(), drawPolyline(), etc … et drawPath() pour des chemins complexes
- Remplissage : fillRect(), fillPath()
- Divers : drawText(), drawPixmap(), drawImage(), drawPicture()

QPainter (5/12)

- Conjointement à la classe Qpainter, on utilise de nombreuses autres classes utiles :
  - Entiers : QPoint, QLine, QRect, QPolygon
  - Flottants : QPointF, QLineF, ...
  - Chemin complexe : QPainterPath
  - Zone d'affichage : QRegion
  - Stylo (trait) : QPen
  - Pinceau (remplissage) :
  - Couleur : QColor

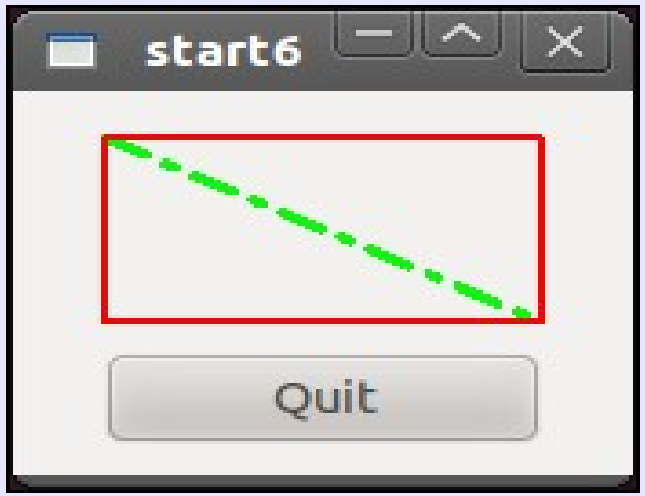






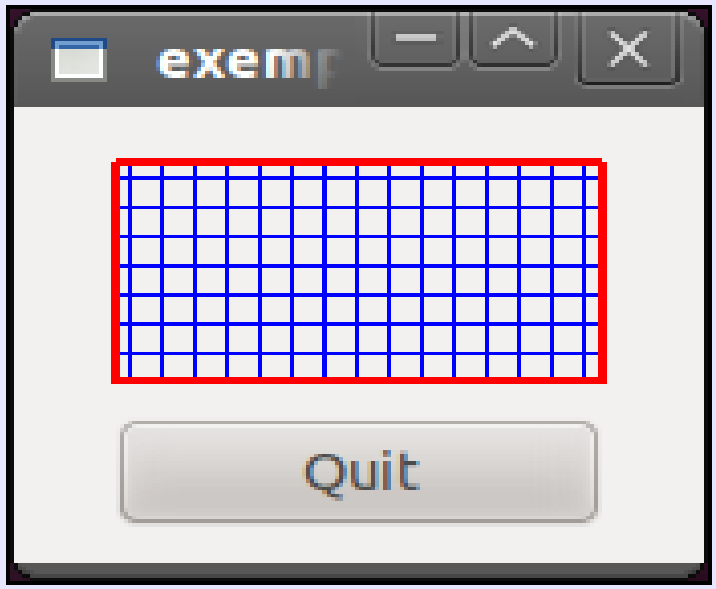
QPainter : exemple 2 (6/12)

```
class MyWidget : public QWidget
{
  public:
   MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
   void paintEvent(QPaintEvent* e)
   {
      QPainter painter(this);
      QPen pen;

      pen.setStyle(Qt::DashDotLine);
      pen.setWidth(3);
      pen.setBrush(Qt::green);
      pen.setCapStyle(Qt::RoundCap);
      pen.setJoinStyle(Qt::RoundJoin);

      painter.setPen(pen);
      painter.drawLine( 25, 15, 145, 75 );
      painter.setPen( QPen(Qt::red, 2) );
      painter.drawRect( 25, 15, 120, 60 );
   }
};
```



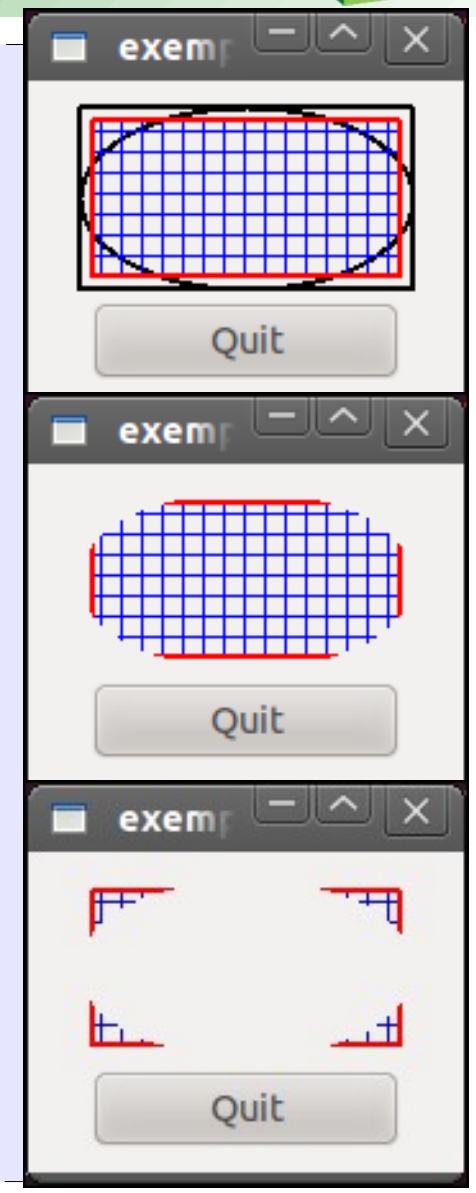
QPainter : exemple 3 (7/12)

```
class MyWidget : public QWidget
{
  public:
   MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
   void paintEvent(QPaintEvent* e)
   {
      QPainter painter(this);
      QBrush brush;

      brush.setStyle(Qt::CrossPattern);
      brush.setColor(Qt::blue);

      painter.setBrush( brush );
      painter.setPen( QPen(Qt::red, 2) );
      painter.drawRect( 25, 15, 120, 60 );
   }
};
```



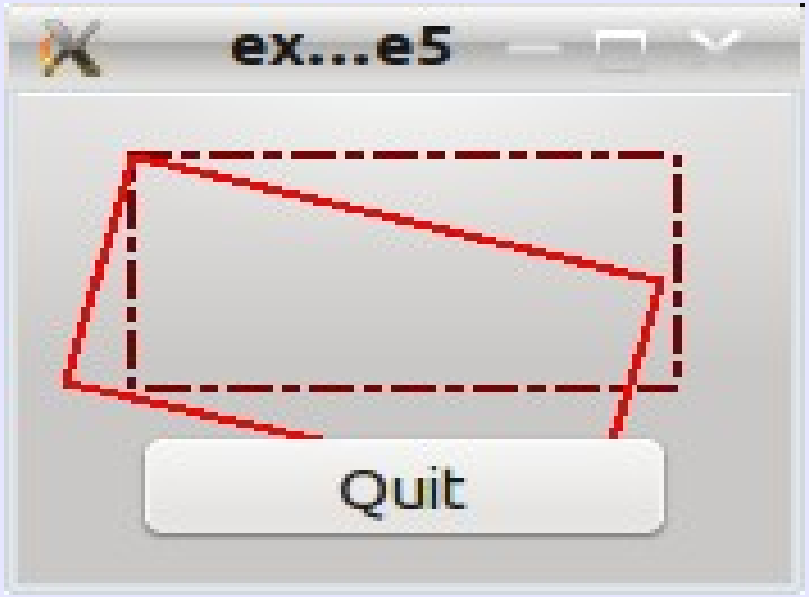
QPainter : exemple 4 (8/12)

```
class MyWidget : public QWidget
{
 public:
  MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
  void paintEvent(QPaintEvent* e)
  {
   QPainter painter(this);
   ...
 /*painter.setPen( QPen(Qt::black, 2) );
   painter.drawEllipse(QRect(20, 10, 130, 70));
   painter.drawRect(QRect(20, 10, 130, 70));*/

   QRegion r1(QRect(20, 10, 130, 70), QRegion::Ellipse);
   QRegion r2(QRect(20, 10, 130, 70));
   QRegion r3 = r1.intersected(r2);
   QRegion r4 = r1.xored(r2);

 //painter.setClipRegion(r3);
   painter.setClipRegion(r4);
      ...
   }
};
```



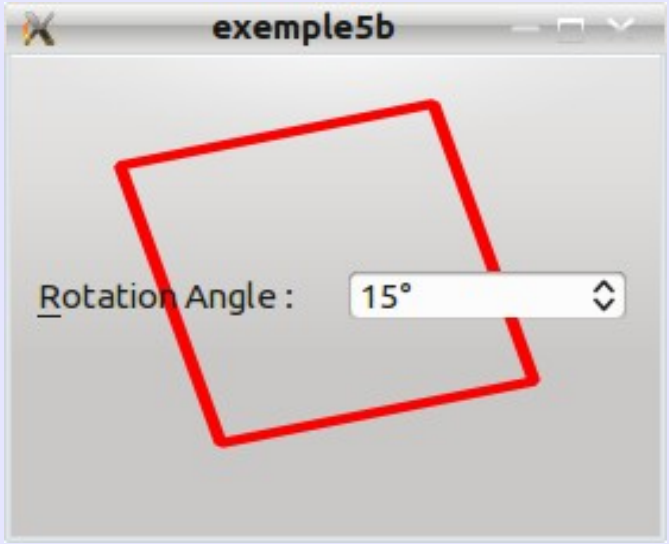
QPainter : exemple 5 (9/12)

```
class MyWidget : public QWidget
{
 public:
  MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
  void paintEvent(QPaintEvent* e)
  {
   QPainter painter(this);

   painter.setPen( QPen(Qt::darkRed, 2, Qt::DashDotLine) );
   painter.drawRect( 25, 15, 120, 60 );

   painter.translate(25, 15);
   painter.rotate( 15.0 );
   painter.translate(-25, -15);

   painter.setPen( QPen(Qt::red, 2) );
   painter.drawRect( 25, 15, 120, 60 );
  }
};
```



QPainter : exemple 5b (10/12)

```
class MyWidget : public Qwidget {
   Q_OBJECT

   qreal rotationAngle;

   public:
      MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
      void paintEvent(QPaintEvent* e) {
         QPainter painter(this);
         painter.setRenderHint(QPainter::Antialiasing);
         painter.scale(width() / 100.0, height() / 100.0);
         painter.translate(50.0, 50.0);
         painter.rotate(-rotationAngle);
         painter.translate(-50.0, -50.0);
         painter.setPen( QPen(Qt::red, 2) );
         painter.drawRect( 25, 15, 50, 60 );
      }
   public slots:
      void setRotationAngle(int degrees) {
         rotationAngle = (qreal)degrees;
         update();
      }
};
```



QPainter : les images (11/12)

- Qt fournit quatre classes de traitement des données d'image : QImage, QPixmap, les QBitmap et QPicture.
  - **QImage** fournit une représentation d'image indépendante du matériel qui permet un accès direct aux pixels. QImage est conçu et optimisé pour les E/S.
  - **QPixmap** est conçu et optimisé pour afficher les images sur l'écran.
  - **QBitmap** n'est qu'une classe de commodité qui hérite QPixmap.
  - **QPicture** est un dispositif permettant d'enregistrer des commandes d'un QPainter et de les rejouer.
- Ces 4 classes héritent de QPaintDevice et on peut donc dessiner dedans avec un QPainter.
- Elles possèdent aussi les méthodes **load()** et **save()** d'accès aux fichiers (dont les principaux formats sont supportés).

QPainter : exemple 6 (12/12)

```
class MyWidget : public QWidget
{
  public:
   MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
   void paintEvent(QPaintEvent* e)
   {
     QWidget::paintEvent(e);
     QPainter painter(this);

     QRect rect(0, 0, 170, 45);
     QPixmap pixmap;
     pixmap.load("qt-logo.png");

     painter.setPen( QPen(Qt::green, 2);
     painter.drawText(rect, Qt::AlignCenter, tr("Qt by\nNokia"));
     painter.drawPixmap(45, 50, pixmap);
   }
};
```

Gestionnaire d'évènements souris

- Gestion des événements sur un QWidget :
  - Lorsqu'on presse un bouton
  - Lorsqu'on relâche un bouton
  - Lorsqu'on déplace la souris
  - Lorsqu'on double-clique

```
void mousePressEvent(QMouseEvent* e);
void mouseReleaseEvent(QMouseEvent* e);
void mouseMoveEvent(QMouseEvent* e);
void mouseDoubleClickEvent(QMouseEvent* e);
```

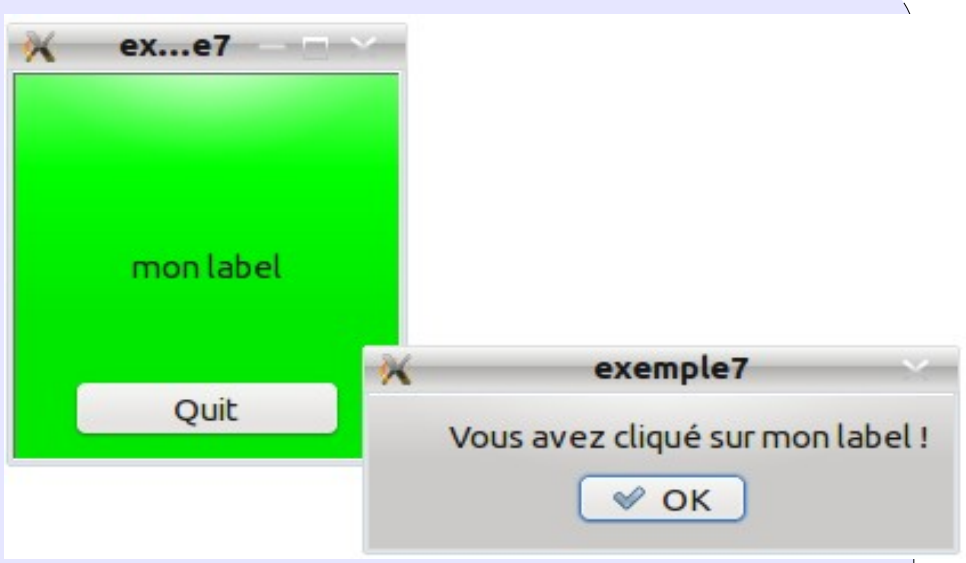
- Pour recevoir des événements de la souris dans ses propres widgets, il suffit donc de réimplémenter ces gestionnaires d'évènements (*event handler*).

- La classe **QMouseEvent** contient les paramètres qui décrivent un événement de la souris : le bouton qui a déclenché l'événement, l'état des autres boutons et la position de la souris.

Exemple 7 (1/2)

- Le widget **QLabel** ne possède pas de signal **clicked()** (comme les QPushButton par exemple). On va donc le créer à partir du gestionnaire **mousePressEvent()**.

```
class MyLabel : public QLabel
{
 Q_OBJECT
 ... // voir diapo suivante
  void mousePressEvent(QMouseEvent *e)
  {
    if(e->button() == Qt::LeftButton)
       emit clicked();
  }
 private slots:
  void selection() {
   QMessageBox msgBox;
   msgBox.setText(QString::fromUtf8("Vous avez cliqué sur mon label !"));
   msgBox.exec();
  }
 signals:
  void clicked();
};
```



Exemple 7 (2/2)


```
class MyLabel : public Qlabel {
 Q_OBJECT
 public:
  MyLabel( QLabel *parent = 0 ) : QLabel( parent ) {
   QPalette palette;
   palette.setColor(QPalette::Window,
                        QColor(QColor(0,255,0)));
   setAutoFillBackground(true);
   setPalette(palette);
   setFrameStyle(QFrame::Panel | QFrame::Sunken);
   setText("mon label");
   setAlignment(Qt::AlignHCenter | Qt::AlignVCenter);
   connect(this,SIGNAL(clicked()),this,SLOT(selection())); // clic bouton
                                                           gauche
   QAction *quitAction = new QAction(tr("E&xit"), this);
   quitAction->setShortcut(tr("Ctrl+Q"));
   connect(quitAction,SIGNAL(triggered()),qApp,SLOT(quit()));
   addAction(quitAction);
   setContextMenuPolicy(Qt::ActionsContextMenu); // clic bouton droit
  }
};
```

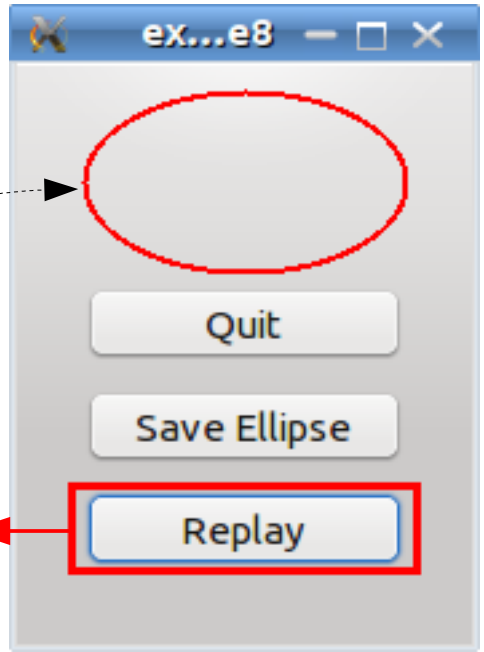
QPicture : exemple 8

- **QPicture** est un dispositif permettant d'enregistrer des commandes d'un QPainter et de les rejouer.


```
QPicture picture;
QPainter painter;
painter.begin(&picture);
painter.setPen( QPen(Qt::red, 2) );
painter.drawEllipse(25, 10, 120, 70);
painter.end();
picture.save("drawing.pic");
```

drawing.pic

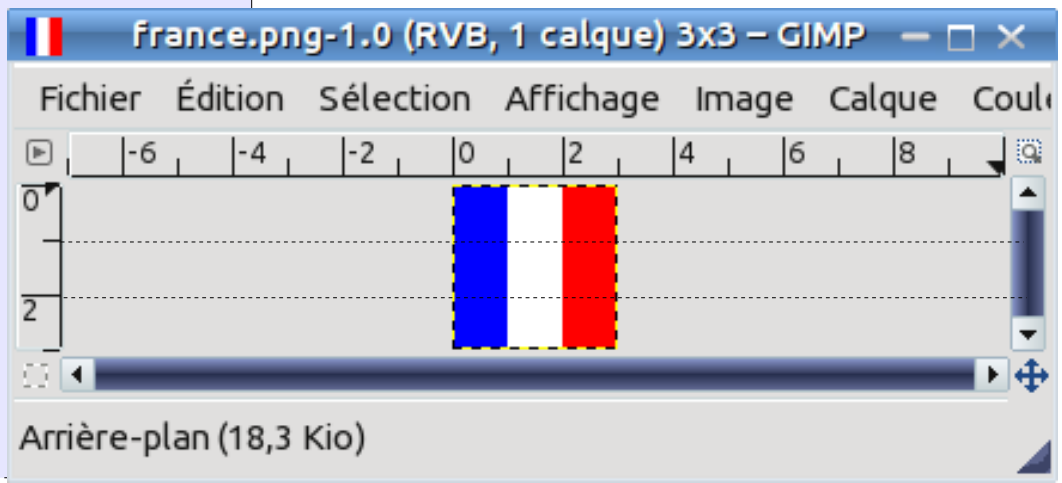
```
QPicture picture;
picture.load("drawing.pic");
QPainter painter;
painter.begin(this);
painter.drawPicture(0, 0, picture);
painter.end();
```



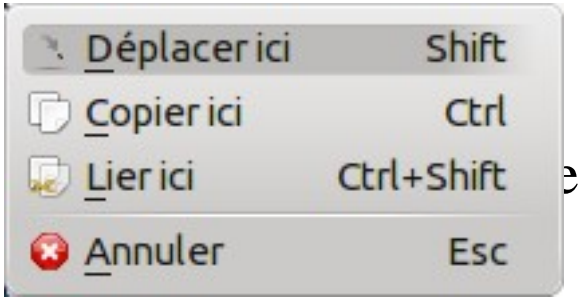
QImage : exemple 9

- **QImage** fournit une représentation d'image indépendante du matériel qui permet un accès direct aux pixels.

```
#include <QImage>
int main(int argc, char **argv) {
   QImage image(3, 3, QImage::Format_RGB32);
   QRgb value;
   value = qRgb(0, 0, 255); // 0x0000FF
   for(int i = 0; i < 3; i++)
      image.setPixel(0, i, value);
   value = qRgb(255, 255, 255); // blanc
   for(int i = 0; i < 3; i++)
      image.setPixel(1, i, value);
   value = qRgb(255, 0, 0);
   for(int i = 0; i < 3; i++)
      image.setPixel(2, i, value);
   image.save("france.png", "PNG");
   return 0;
}
```



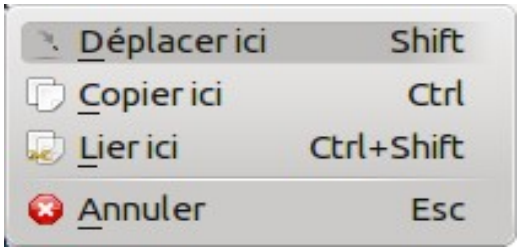
Le « glisser-déposer » (1/11)



- Le "glisser-déposer" (drag & drop) est un mécanisme visuel simple qui permet aux utilisateurs de transférer des informations entre et au sein des applications.
  glisser-déposer est une forme de copier/couper/coller.
- Il y a 4 classes de base pour gérer les évènements associés à l'action de de glisser-déposer :
  - **QDragEnterEvent** : Événement qui est envoyé à un widget lorsque l'action de glisser débute
  - **QDragLeaveEvent** : Événement qui est envoyé à un widget lorsque l'action de glisser se termine
  - **QDragMoveEvent** : Événement qui est envoyé quand une action de glisser-déposer est en cours
  - **QDropEvent** : Événement qui est envoyé quand une action de glisser-déposer est terminée

Le « glisser-déposer » (2/11)

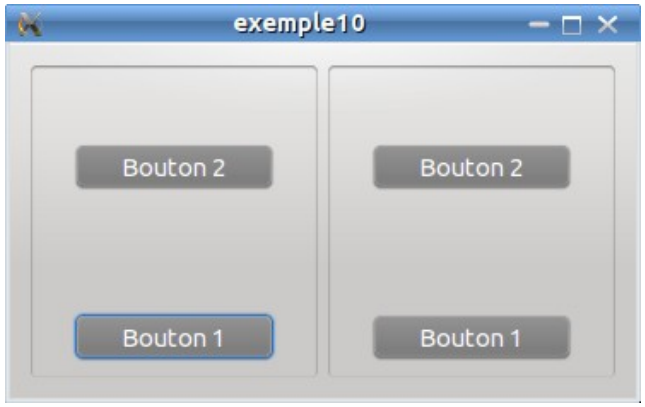
- Il y a par conséquence 4 gestionnaires d'évènements associés :
  - **`void dragEnterEvent(QDragEnterEvent *event);`**
  - **`void dragLeaveEvent(QDragLeaveEvent *event);`**
  - **`void dragMoveEvent(QDragMoveEvent *event);`**
  - **`void dropEvent(QDropEvent *event);`**
- Pour démarrer un glisser-déposer, il faut créer un objet **QDrag** et appeler sa fonction **exec**().



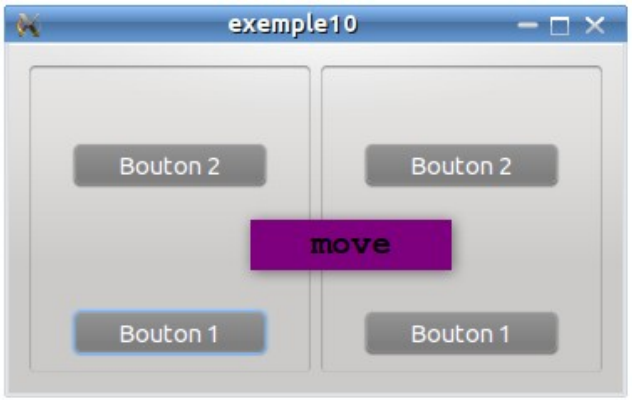
- Les actions possibles sont :
  - **`Qt::CopyAction`** : Copie les données vers la cible.
  - **`Qt::MoveAction`** : Déplacer les données de la source vers la cible.
  - **`Qt::LinkAction`** : Créer un lien de la source vers la cible.
- N'importe quel type d'information peut être transféré dans une opération de glisser-déposer. Les applications concernées doivent être en mesure d'indiquer à l'autre quels sont les formats de données gérés en utilisant des types MIME.

Le « glisser-déposer » (3/11)

- Pour illustrer le mécanisme de base du glisser-déposer, on va prendre un exemple simple.
- L'application (exemple n°10) va créer deux zones (QFrame) dans lesquelles seront placées deux boutons. Les boutons peuvent être glissés-déposés dans l'autre zone mais pas à l'intérieur de sa zone d'origine. Si un bouton est glissé-déposé, un nouveau bouton de même libellé est créé et le bouton source est détruit.



Situation initiale



Le bouton 1 de gauche en cours de glissage

Le bouton 1 a été déposé dans la zone de droite

Le « glisser-déposer » : exemple 10 (4/11)

- On crée son propre widget bouton :

```
class MyPushButton : public QPushButton {
 Q_OBJECT
 public:
  MyPushButton(QString libelle, QWidget *p=0):QPushButton(libelle, p)
  {
    resize(120, 30);
    QPalette palette;
    palette.setColor(QPalette::Button, QColor(127,127,127));
    palette.setColor(QPalette::ButtonText, QColor(Qt::white));
    this->setPalette(palette);
  }
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  QPoint dragStartPosition() {
    return _dragStartPosition;
  }
 private:
  QPoint _dragStartPosition;
};
```

Le « glisser-déposer » : exemple 10 (5/11)

- Dans la plupart des cas, l'opération de glisser-déposer démarre lorsqu'un bouton de la souris a été pressé et que le curseur a été déplacé sur une certaine distance. En effet, certains widgets ont besoin de faire la distinction entre les clics de la souris et le glisser.
- Il faudra donc réimplémenter le gestionnaire d'évènement **mousePressEvent()** du widget pour pour enregistrer la position de départ du glisser. Puis, il faudra réimplémenter **mouseMoveEvent()** pour déterminer si un glisser devrait commencer.

```
class MyPushButton : public QPushButton {
 Q_OBJECT
  void mousePressEvent(QMouseEvent *event)
  {
    if(event->button() == Qt::LeftButton)
      _dragStartPosition = event->pos();
  }

};
```

Le « glisser-déposer » : exemple 10 (6/11)

- Il faudra réimplémenter **mouseMoveEvent()** pour déterminer si un glisser devrait commencer. On utilisera la fonction manhattanLength() pour obtenir une estimation approximative de la distance entre l'endroit où le clic de souris a eu lieu et la position actuelle du curseur.
- Les objets QMimeData et QDrag créés par le widget source ne doivent pas être supprimés car ils seront détruits par Qt.

```
void mouseMoveEvent(QMouseEvent *event)
{
 if (!(event->buttons() & Qt::LeftButton)) return;
 if ((event->pos() - _dragStartPosition).manhattanLength()
      < QApplication::startDragDistance()) return;

 QDrag *drag = new QDrag(this);
 QMimeData *mimeData = new QMimeData;
 mimeData->setText(this->text());
 drag->setMimeData(mimeData);
 Qt::DropAction dropAction = drag->exec();
 if(dropAction == Qt::MoveAction)
   close();
}
```

Le « glisser-déposer » : exemple 10 (7/11)

- On crée sa propre frame qui contient deux boutons :

```
class MyFrame : public QFrame {
   Q_OBJECT
   public:
      MyFrame( QWidget *parent = 0 ) : QFrame( parent ) {
         setMinimumSize(170, 185);
         setFrameStyle(QFrame::Sunken | QFrame::StyledPanel);
         setAcceptDrops(true);
         btn1 = new MyPushButton("Bouton 1", this);
         btn1->move(25, 145);
         btn2 = new MyPushButton("Bouton 2", this);
         btn2->move(25, 45);
         btn1->show(); btn2->show();
      }
      void dragEnterEvent(QDragEnterEvent *event) ;
      void dragMoveEvent(QDragMoveEvent *event) ;
      void dropEvent(QDropEvent *event) ;
   private:
      MyPushButton *btn1;
      MyPushButton *btn2;
};
```

Le « glisser-déposer » : exemple 10 (8/11)

- Le widget receveur devra accepter le déposer en appelant la méthode **setAcceptDrops(true)** et réimplémenter les gestionnaires d'évènements associés au glisser-déposer.
- Tout d'abord lorsque l'action de glisser débute :

```
class MyFrame : public QFrame {
   Q_OBJECT
   void dragEnterEvent(QDragEnterEvent *event)
   {
    if(event->mimeData()->hasFormat("text/plain") &&
       event->proposedAction() == Qt::MoveAction)
    {
      event->acceptProposedAction();
    }
    else
    {
      event->ignore();
    }
   }
};
```

Le « glisser-déposer » : exemple 10 (9/11)

- Ensuite, on traite le déplacement du "glisser" en vérifiant si on est dans la bonne zone de "déposer".

```
class MyFrame : public QFrame {
   Q_OBJECT
   void dragMoveEvent(QDragMoveEvent *event)
   {
     if(event->mimeData()->hasFormat("text/plain") &&
         event->proposedAction() == Qt::MoveAction)
     {
       MyPushButton *childBtn = static_cast<MyPushButton*>(event-
>source());
       if(this == childBtn->parentWidget())
            event->ignore();
       else event->acceptProposedAction();
     }
     else   event->ignore();
   }
};
```

Le « glisser-déposer » : exemple 10 (10/11)

- On termine par gérer le "déposer" en créant un nouveau bouton à l'endroit où il a été posé en en récupérant son libellé dans les données transférées.

```
class MyFrame : public QFrame {
    Q_OBJECT
    void dropEvent(QDropEvent *event)
    {
      if (event->mimeData()->hasFormat("text/plain") &&
          event->proposedAction() == Qt::MoveAction) {
        MyPushButton *childBtn = static_cast<MyPushButton*>(event-
>source());
        if(this != childBtn->parentWidget()) {
          MyPushButton *newBtn = new MyPushButton(event->mimeData()-
>text(), this);
          newBtn->move(event->pos() - childBtn->dragStartPosition());
          newBtn->show();
        }
        event->acceptProposedAction();
      }
      else event->ignore();
    }
};
```

Le « glisser-déposer » : exemple 10 (11/11)

- On peut aussi "glisser-déposer" un bouton de notre application vers une autre application. On utilise ici un exemple fourni par Qt : dropsite.

- Le bouton 1 a été déplacé dans l'application dropsite qui a récupéré les données associées : c'est-à-dire son libellé soit « Bouton 1 ».

- Documentation sur le "glisse
  http://qt-project.org/doc/q