Cours Qt (2° partie)
Boîtes de dialogue et Application principale

Thierry Vaira

IUT Arles

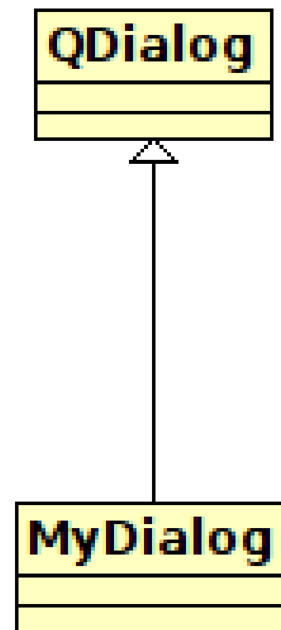
tvaira@free.fr ☺ v1

Sommaire

1. Boîtes de dialogue
2. Fenêtre principale
3. Aspect visuel et ergonomique

Création d'une "boîte de dialogue"

La création d'une "boîte de dialogue" est réalisée à partir d'une nouvelle classe qui **hérite** de `QDialog` :



Squelette d'une "boîte de dialogue"

mydialog.h

```
class MyDialog : public QDialog
{
    Q_OBJECT
private:

public:
    MyDialog(QWidget *parent=0);

public slots:

signals:

};
```

mydialog.cpp

```
#include "mydialog.h"

MyDialog::MyDialog(QWidget *
     parent)
    : QDialog(parent)
{
  // TODO
}
```

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 :
  - **modale** ou **non modale**
  - fournir une valeur de retour
  - avoir des boutons par défaut
  - avoir un `QSizeGrip` (une poignée de redimensionnement) dans le coin inférieur droit

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.

Boîte de dialogue modale

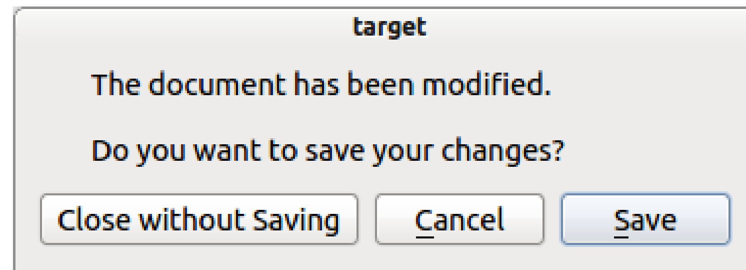
- Une **boîte de dialogue modale** (*modal dialog*) est un dialogue qui **bloque l'entrée** à d'autres fenêtres visibles de la même application.
- Exemple : les dialogues qui sont utilisés pour demander un nom de fichier ou qui sont utilisés pour définir les préférences de l'application (couleur, police,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

Les boîtes de dialogue en “français”

```
#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QString locale = QLocale::system().name().section('_', 0, 0);
    QTranslator translator;
    translator.load(QString("qt_") + locale, QLibraryInfo::location(
        QLibraryInfo::TranslationsPath));
    a.installTranslator(&translator);

    // ...

    return a.exec();
}
```

La classe QMessageBox

- La classe `QMessageBox` fournit un dialogue modale pour **informer l'utilisateur** ou pour demander à l'utilisateur une question et recevoir une réponse.




```
QMessageBox msgBox;
msgBox.setText("The document has been modified.");
msgBox.setInformativeText("Do you want to save your changes?");
msgBox.setStandardButtons(QMessageBox::Save | QMessageBox::Discard |
    QMessageBox::Cancel);
int ret = msgBox.exec();
```

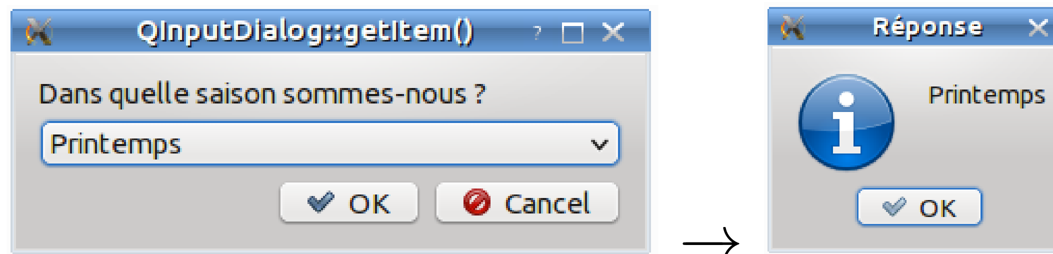
La classe QMessageBox

- Elle fournit aussi quatre types prédéfinis : QMessageBox::critical(), QMessageBox::information(), QMessageBox::question(), QMessageBox::warning().

```
QMessageBox::critical(0, "Exemple", "QMessageBox::critical()");
```

La classe QInputDialog

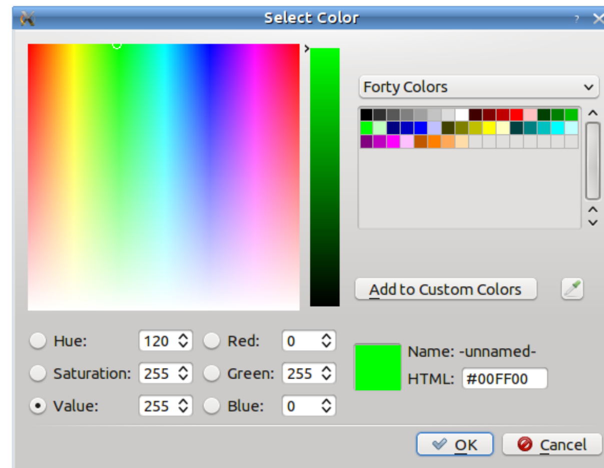
- La classe `QInputDialog` fournit un dialogue simple pour **obtenir une valeur unique de l'utilisateur**. La valeur d'entrée peut être une chaîne, un numéro ou un élément d'une liste (`getText()`, `getInt()`, `getDouble()`, `getItem()`). Une étiquette (`Label`) doit être placée afin de préciser à l'utilisateur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




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

La classe QColorDialog

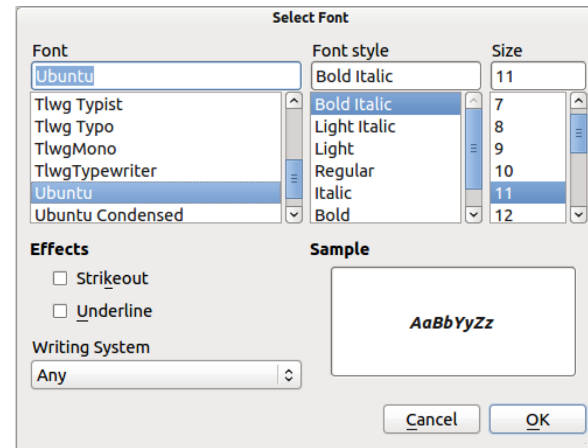
- La classe QColorDialog fournit un dialogue pour la **spécification des couleurs**. Cela permet aux utilisateurs de choisir les couleurs (getColor()). Par exemple, vous pourriez l'utiliser dans un programme de dessin pour permettre à l'utilisateur de définir la couleur du pinceau.




```
QColor color = QColorDialog::getColor(Qt::green, 0);
if (color.isValid()) { // ... }
```

La classe QFontDialog

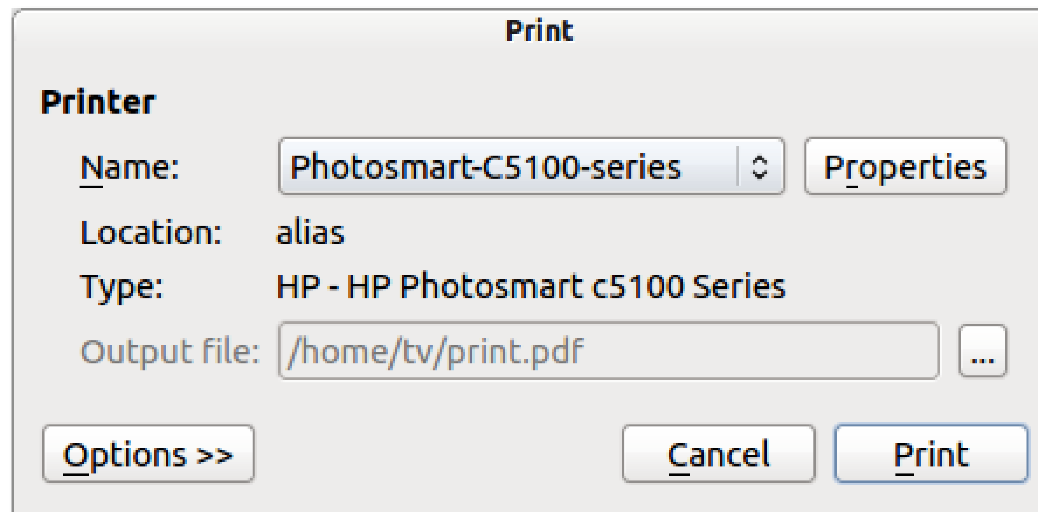
- La classe `QFontDialog` fournit un *widget* de dialogue de **sélection d'une police** (`getFont()`).




```
bool ok;
QLabel pMonLabel("Hello world!");
QFont font = QFontDialog::getFont(&ok, pMonLabel.font());
if (ok) {
   pMonLabel.setFont(font);
   pMonLabel.show();
}
```

La classe QPrintDialog

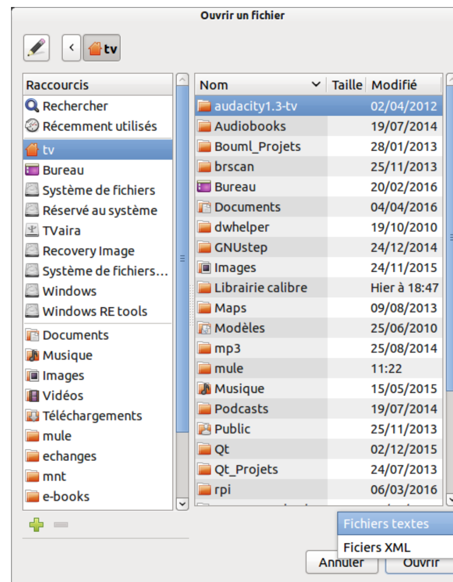
- La classe `QPrintDialog` fournit une boîte de dialogue pour **spécifier la configuration de l'imprimante et imprimer**.




```
QPrinter printer;
QPrintDialog printDialog(&printer, 0);
if (printDialog.exec() == QDialog::Accepted) { // print ... }
```

La classe QFileDialog

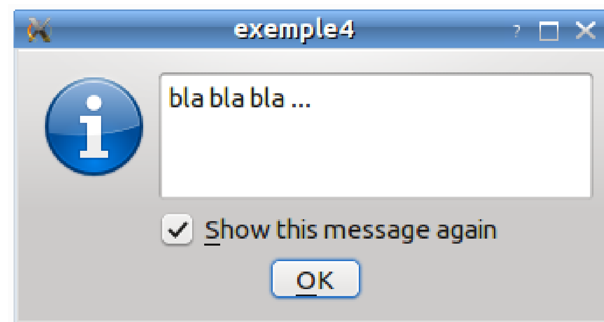
- La classe `QFileDialog` fournit une boîte de dialogue modale qui permet aux utilisateurs de **sélectionner des fichiers ou des répertoires**. Elle permet de parcourir le système de fichiers afin de sélectionner un ou plusieurs fichiers ou un répertoire (`getExistingDirectory()`, `getOpenFileName()`, `getOpenFileNames()`, `getSaveFileName()`).




```
QString fileName = QFileDialog::getOpenFileName(0, "Ouvrir un fichier",
    "/home/tv", "Fichiers textes (*.txt);;Ficiers XML (*.xml)");
```

La classe QErrorMessage

- La classe QErrorMessage fournit une boîte de dialogue non modale qui **affiche un message d'erreur** (showMessage()).



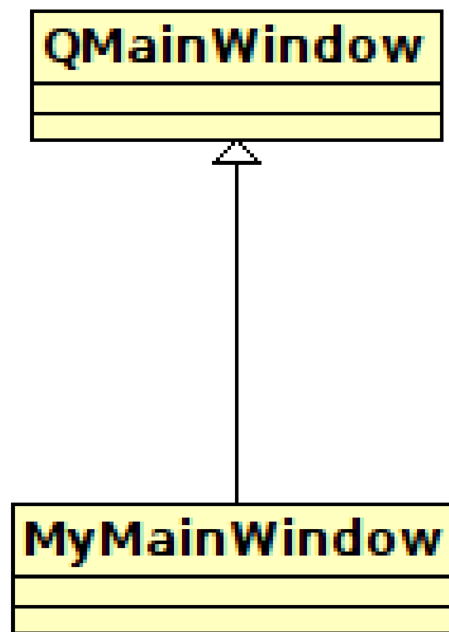

```
QErrorMessage *errorMessageDialog;
errorMessageDialog = new QErrorMessage;
errorMessageDialog->showMessage("bla bla bla ...");
```

Sommaire

1. Boîtes de dialogue
2. Fenêtre principale
3. Aspect visuel et ergonomique

Création d'une application "fenêtre principale"

La création d'une application "fenêtre principale" est réalisée à partir d'une nouvelle classe qui **hérite** de `QMainWindow` :



Squelette de l'application "fenêtre principale"

mymainwindow.h

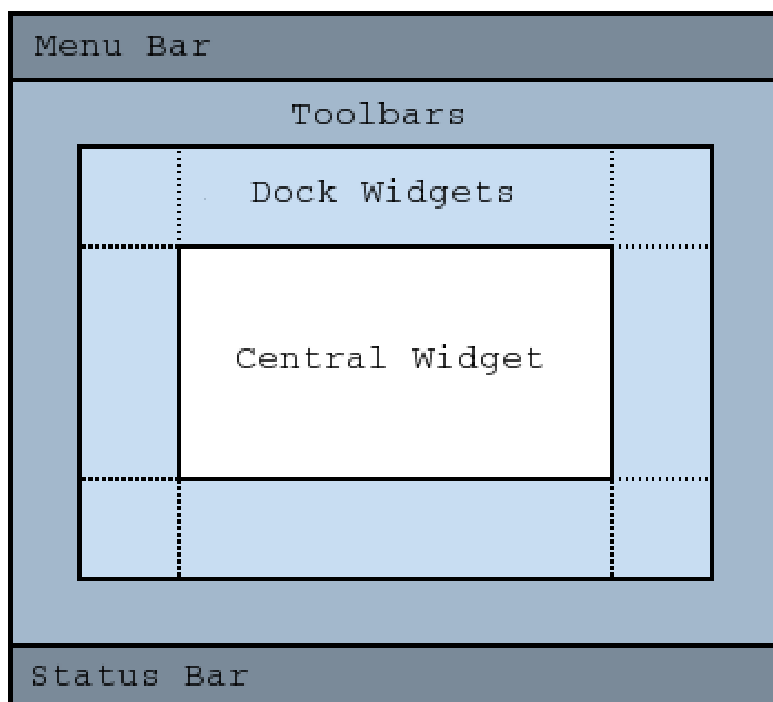
```
class MyMainWindow : public
     QMainWindow
{
    Q_OBJECT
private:

public:
    MyMainWindow(QWidget *parent
        = 0);

public slots:

signals:

};
```

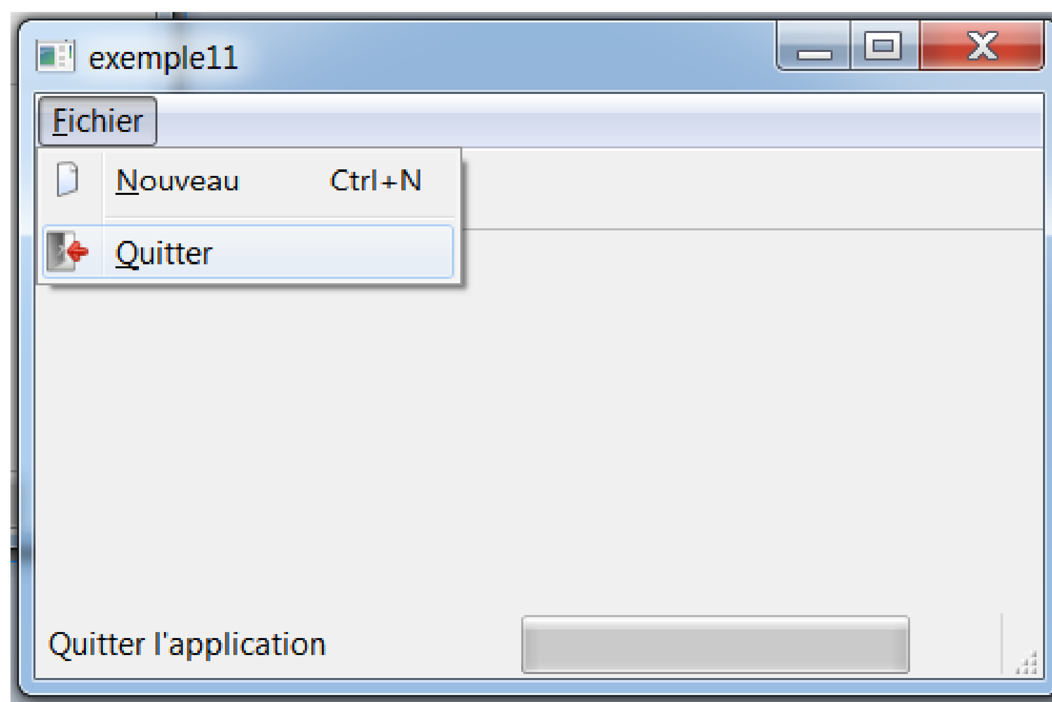
mymainwindow.cpp

```
#include "mymainwindow.h"

MyMainWindow::MyMainWindow(
     QWidget *parent)
    : QMainWindow(parent)
{
  // TODO
}
```

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.





La structure d'un `QMainWindow`

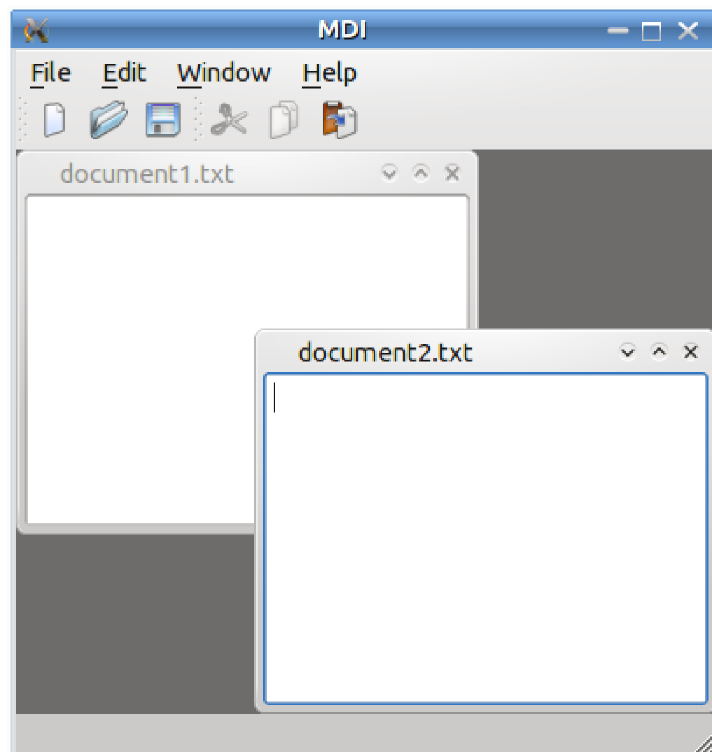
- `QMainWindow` a sa propre mise en page à laquelle vous pouvez ajouter `QToolBars`, `QDockWidgets`, un `QMenuBar`, et un `QStatusBar`.
- La fenêtre possède une zone centrale (*central widget*) qui peut être occupée par n'importe quel type de *widget*.
- Le *widget* central sera généralement un *widget* standard de Qt comme un `QTextEdit` (logiciel de type "texte") ou un `QGraphicsView` (logiciel de "dessin"). Les *widgets* personnalisés peuvent également être utilisés pour des applications avancées.
- On définit le *widget* central avec `setCentralWidget()`.

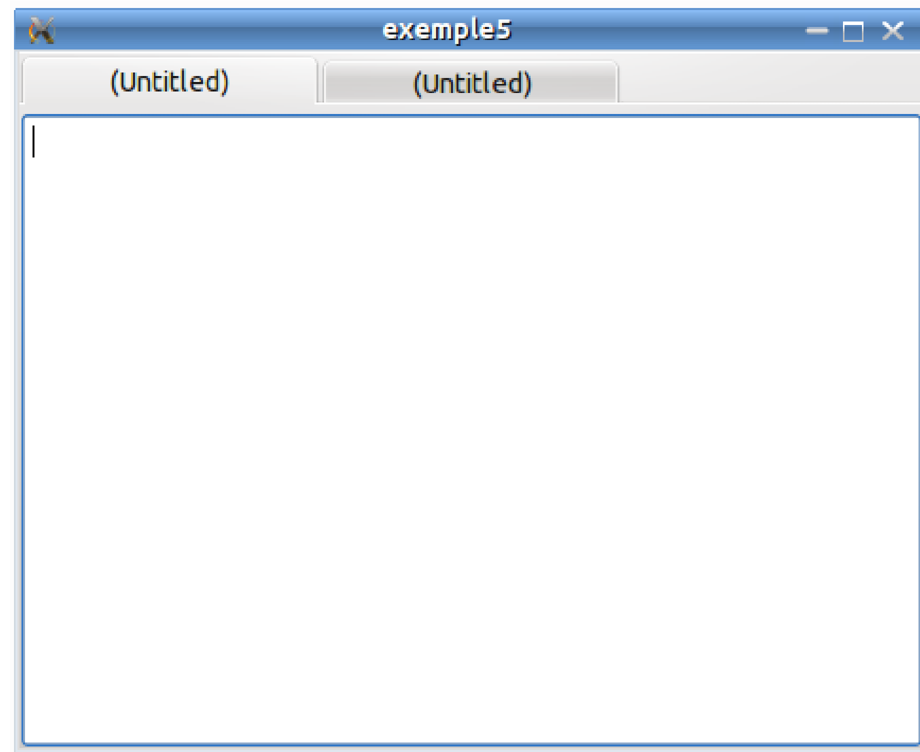
SDI ou MDI

- Une application qui gère des documents 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.
- Les sous-fenêtres de `QMdiArea` sont des instances de `QMdiSubWindow`. Elles sont ajoutées à une zone MDI avec `addSubWindow()`.

L'interface MDI

Avec `setViewMode()`, on peut choisir un affichage des sous-fenêtres soit indépendantes (`QMdiArea::SubWindowView`) soit regroupées en onglet (`QMdiArea::TabbedView`).



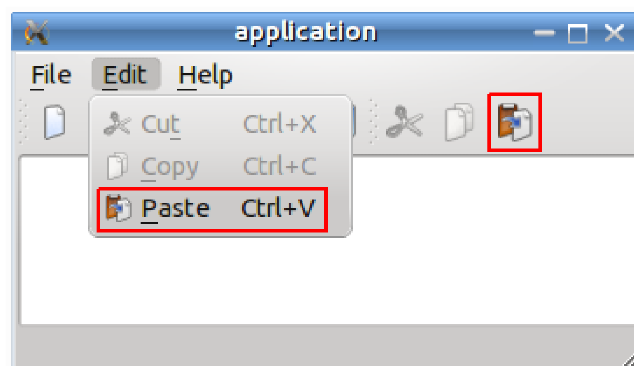


La classe QMdiArea

```
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);
```

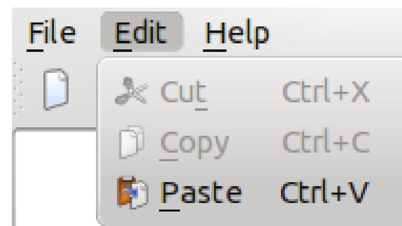
La classe `QAction`

- La classe `QAction` fournit une **interface abstraite pour décrire une action** (= commande) qui peut être insérée dans les *widgets*.
- Cela permet de créer des commandes communes pouvant être invoquées via des menus, boutons, et des raccourcis clavier.
- Les actions peuvent être ajoutés aux menus et barres d'outils, et seront automatiquement synchronisées.



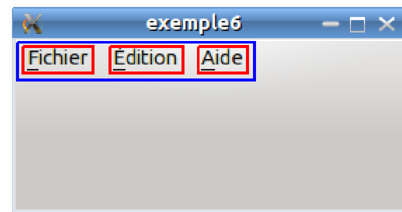
La classe QMenu

- 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.
- 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

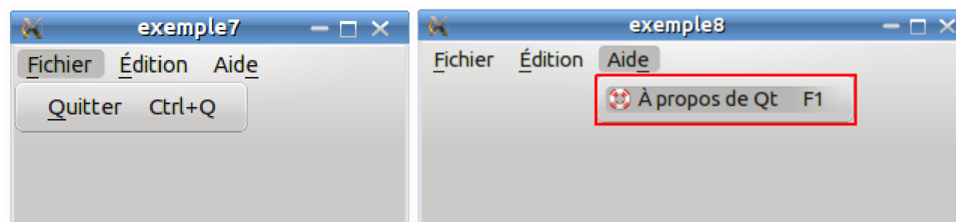
- La classe QMenuBar fournit une **barre de menu horizontale**. Une barre de menu se compose d'une liste d'éléments de menu déroulant.
- On peut ajouter de nouveaux menus à la barre de menus de la fenêtre principale en appelant menuBar() qui retourne la QMenuBar de la fenêtre, puis ajoutez un menu avec QMenuBar::addMenu()




```
QMenu *fileMenu = new QMenu(QString::fromUtf8("&Fichier"), this);
menuBar()->addMenu(fileMenu);

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

// ...
```

La classe QMenu

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

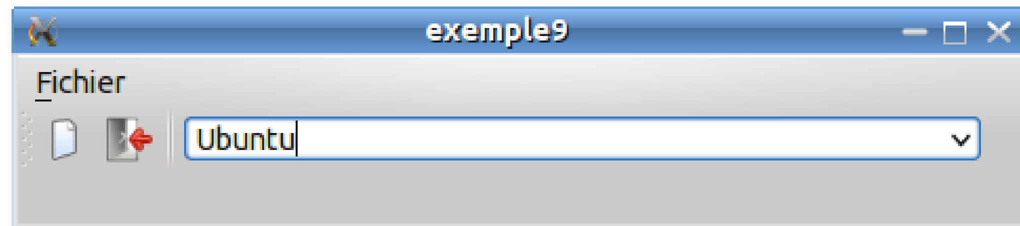



```
// Solution 1 :
fileMenu->addAction(QString::fromUtf8("&Quitter"), qApp, SLOT(quit()),
     QKeySequence::Quit);

// Solution 2 :
QAction *actionHelp = new QAction(QString::fromUtf8("À propos de Qt"),
     this);
actionHelp->setShortcut(QKeySequence(Qt::Key_F1));
actionHelp->setIcon(QIcon(":/help.png"));
connect(actionHelp, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
helpMenu->addAction(actionHelp);
```

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().
- Mais on peut aussi insérer un *widget* (comme QSpinBox, QDoubleSpinBox ou QComboBox) à l'aide de addWidget() ou insertWidget().

La classe QToolBar

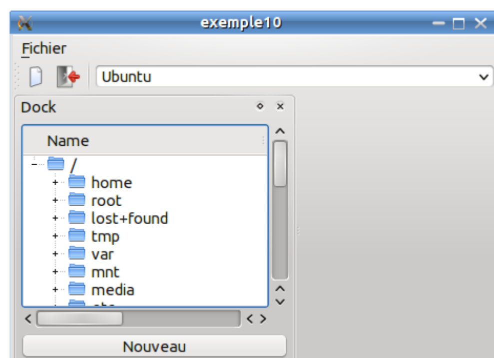


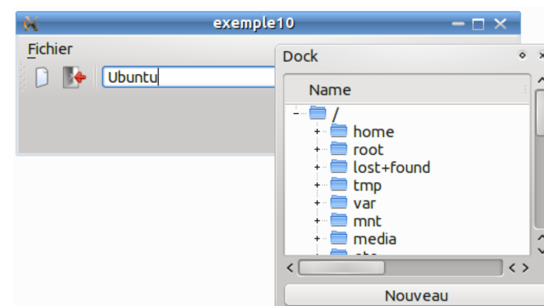

```
QToolBar *fileToolBar = addToolBar("Fichier");

//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)** placées dans la zone autour du *widget* central d'une `QMainWindow`.
- Beaucoup d'applications connues les utilisent : Qt Designer, OpenOffice, Gimp, Photoshop, Code::Blocks , ...

La classe QDockWidget



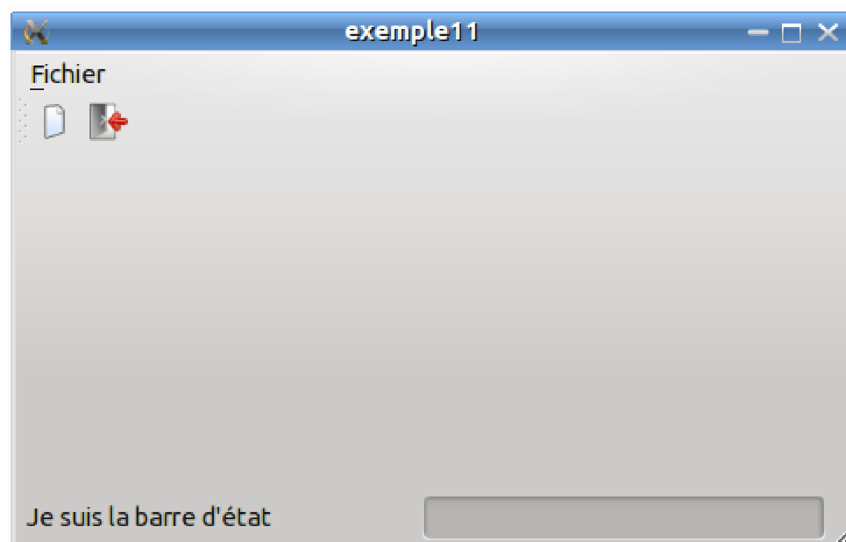



```
QDockWidget *dockWidget = new QDockWidget("Dock", this);
addDockWidget(Qt::LeftDockWidgetArea, dockWidget);

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);
//...
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()`.

La classe QStatusBar




```
QStatusBar *barreEtat = statusBar();

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

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

Les fichiers ressources .qrc

- On peut utiliser un **fichier ressource** de Qt (.qrc) pour référencer l'image d'une icône par exemple.
- 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.
- Exemple : `actionHelp->setIcon(QIcon(":/help.png"));`

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

Sommaire

1 Boîtes de dialogue

2 Fenêtre principale

3 Aspect visuel et ergonomique

Aspect visuel des *widgets*

Les *widgets* possèdent de nombreuses méthodes qui permettent d'agir sur l'**aspect visuel** :

- **taille** : la propriété `sizeHint` détient les dimensions recommandées du *widget*. Cela est important pour le système de gestion de *layout*. La propriété `sizePolicy` définit le comportement par défaut du *layout* du *widget*. Il est possible de redimensionner manuellement le *widget* avec `setGeometry()`, `resize()`, ...
- **position** : généralement les *widgets* sont positionnés dans un *layout*. Il est possible de les déplacer manuellement avec `move()`. Il est aussi possible de définir l'alignement du contenu d'un widget avec `setAlignment()`.
- **visibilité** : `setVisible(true)` ou `show()` définit le *widget* à un statut visible si la totalité de ses *widgets* parents jusqu'à la fenêtre sont visibles. `setVisible(false)` ou `hide()` cache explicitement un *widget*.

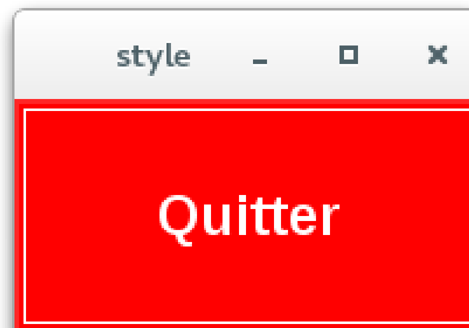
Aspect ergonomique des *widgets*

Les *widgets* possèdent de nombreuses méthodes qui permettent d'agir sur l'**aspect ergonomique** :

- **style** : la méthode `setStyle()` définit le style de GUI du *widget*. On peut par exemple appliquer un style "GTK+", "Windows", ...
- **feuille de style** : la propriété `styleSheet` définit la feuille de style du *widget*. La feuille de style (`qss`) contient une description textuelle des personnalisations pour le style du *widget*.
- **police** : la propriété `font` contrôle la police actuellement définie pour le *widget*.
- **palette** : la propriété `palette` décrit la palette du *widget*.

Exemple : modifier l'aspect visuel et ergonomique d'un widget

```
QPushButton *pMonBouton = new QPushButton("Quitter");

//pMonBouton->setGeometry(0,0,200,100);
pMonBouton->resize(200, 100);
pMonBouton->move(50, 50);
pMonBouton->setFont(QFont("Arial", 18, QFont::Bold));
pMonBouton->setStyleSheet("background-color: red; color: white;");
pMonBouton->setStyle(QStyleFactory::create("Motif"));
pMonBouton->show();
```



Exemple : modifier l'aspect visuel et ergonomique d'un widget

```
QLabel *pMonLabel = new QLabel("Hello world!");

pMonLabel->setFrameStyle(QFrame::Panel | QFrame::Sunken);
pMonLabel->setMargin(10);
pMonLabel->setFont(QFont("Verdana", 14, QFont::DemiBold, QFont::
    StyleItalic));
pMonLabel->setAlignment(Qt::AlignHCenter);
QPalette palette;
palette.setColor(QPalette::WindowText, Qt::blue);
pMonLabel->setAutoFillBackground(true);
pMonLabel->setPalette(palette);
pMonLabel->show();
```