

Chapitre 1

Programmation événementielle avec C++ et Qt

Ressource R2.02 : Développement d'application avec IHM

Bref historique

- ◆ 1988 : Idée de création en C++ d'une Qt **bibliothèque logicielle orientée objet** (*API-Application Programming Interface*) par Haavard Nord et Eirik Chambe-Eng.
- ◆ 1991 : début de développement de la bibliothèque.
- ◆ 1993 : le noyau est prêt et permet de créer des composants graphiques sous Windows et Unix à partir de la même API.
Les 2 auteurs créent la société (Quasar Technologies, puis TrollTech) pour commercialiser « le meilleur framework GUI en langage C++ ».
- ◆ Au fil des années, la bibliothèque se complète : conception d'un environnement KDE, exécution sur des périphériques embarqués, plateforme pour applications mobiles, ...
- ◆ Aujourd'hui, Qt est un **framework - plateforme de développement d'interfaces graphiques GUI** (*Graphical User Interface*), appartenant depuis 2012 à Digia, après avoir été rachetée en 2008 par Nokia (www.qt.io).



Qu'est-ce Qt ?

- ◆ Qt est un *framework* - **plateforme de développement d'interfaces graphiques GUI** (*Graphical User Interface*).
- ◆ Qt fournit un **ensemble de classes** décrivant
 - des éléments **graphiques (widgets)**, pour **windows gadgets**)
 - et des éléments **non graphiques** : accès aux données (fichier, base de données), connexions réseaux (socket), gestion du multitâche (*thread*), XML, etc.
- ◆ Qt facilite la **portabilité des applications** sur Unix (dont Linux), Windows et Mac OS X, Android, iOS :
 - Son API est **la même** pour toutes ces plateformes : pas de nécessité d'apprendre les APIs spécifiques à chaque OS cible
 - Le portage est fait par simple **recompilation du code source** (s'il n'utilise que ses composants)
 - Gain de temps pour les programmeurs
- ◆ Qt dispose d'un **moteur de rendu graphique 2D**.
- ◆ Autres bibliothèques multi-plateformes équivalentes connues :
 - **GTK+**, utilisée par l'environnement graphique GNOME
 - **WxWidgets**, utilisé pour développer FileZilla, CodeBlocks, Audacity, iMule... et jusqu'en 2021, ici à l'IUT, dans le module « programmation C++ avec interface graphique » ☺

Utilisation de Qt

Clients

- ◆ Google, Adobe Systems, Asus, Samsung, Philips, ou encore la NASA et bien évidemment Nokia
- ◆ Qt est notamment connu pour être la bibliothèque sur laquelle repose l'environnement graphique KDE, l'un des environnements de bureau les plus utilisés dans le monde Linux

Licences (www.qt.io/licensing/)

- ◆ GNU GPL : gratuite avec obligation de gratuité du code produit
- ◆ GNU LGPL : possibilité d'utiliser gratuitement certains outils de Qt (ex., la bibliothèque graphique) dans du code propriétaire sans contraindre à rendre le code produit libre
- ◆ Licence commerciale : nécessaire si le développeur souhaite produire du code propriétaire
- ◆ A l'IUT : Utilisation d'une licence commerciale mention *Education* : *chaque utilisateur s'engage à ne pas utiliser la plateforme à des fins commerciales*



EDI et principaux outils

Qt Creator

- ◆ C'est l'environnement de développement intégré dédié à Qt et facilite la gestion d'un projet Qt.
- ◆ Son éditeur de texte offre les principales fonctions que sont la **coloration syntaxique, l'autocomplétion** (ou **complètement automatique**), **l'indentation**, etc...
- ◆ Qt Creator intègre les outils :
 - **Qt Designer** : outil de dessin d'interfaces graphiques.
 - **Qt Linguist** : il permet la mise en œuvre rapide de l'internationalisation, c'est-à-dire le développement d'applications multilingues (sans multiplier le développement)
 - **Qt Assistant** : assistant à la création de projets Qt
 - Un mode débogage et beaucoup d'autres *plugins*.



Qt Creator

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

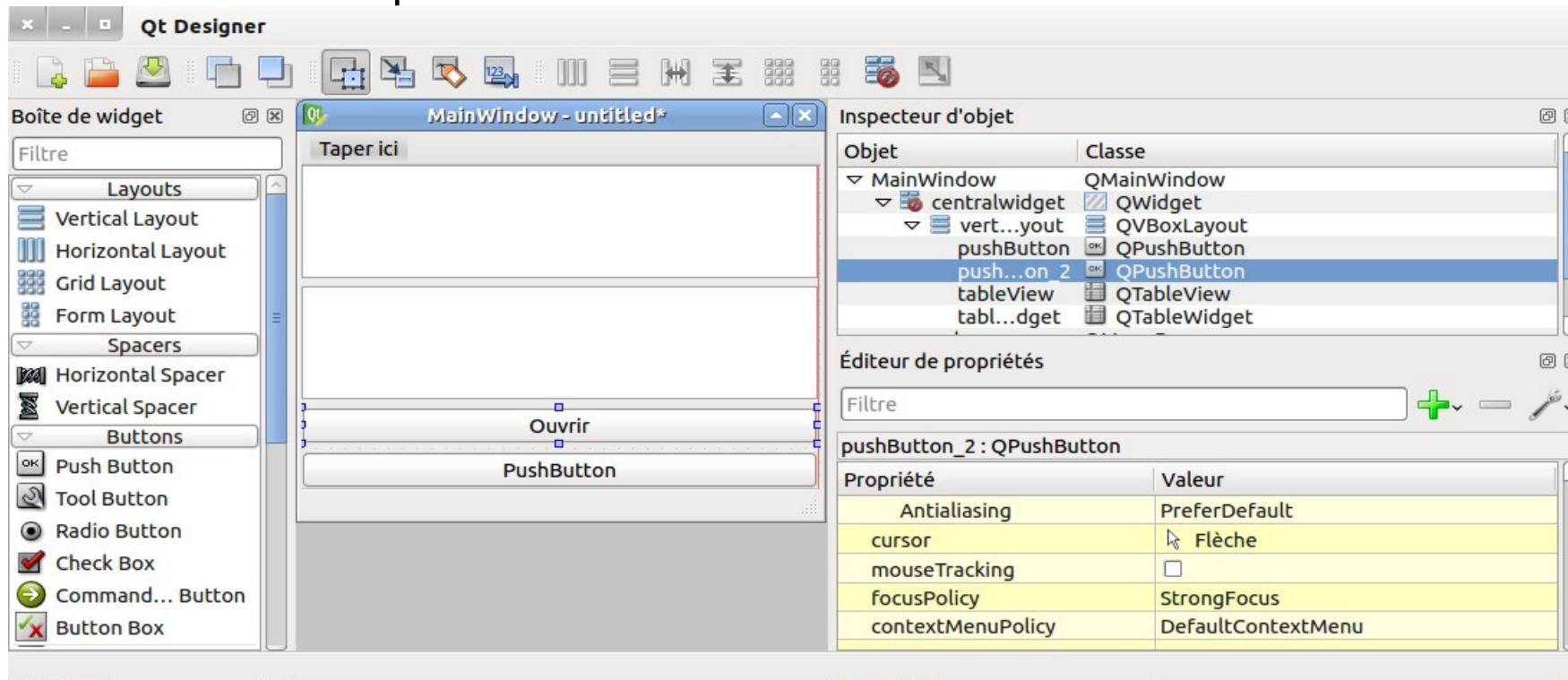
The screenshot shows the Qt Creator interface with the following details:

- Title Bar:** QC main.cpp @ zero - Qt Creator
- Menu Bar:** Fichier, Édition, View, Compiler, Déboguer, Analyse, Outils, Fenêtre, Aide
- Toolbars:** Projects, Search, Open, New, Line: 12, Col: 1
- Project Explorer (Left):** Accueil, Éditer, Design, Debug, Projets, Aide, zero, Debug, Run, Stop.
- File Tree (Top Left):** Projets, zero (selected), zero.pro, Headers, Sources (selected), main.cpp (selected), principale0.cpp, Forms.
- Code Editor (Main Area):** File: main.cpp, Content:

```
#include "principale0.h"
#include <QApplication>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Principale0 w;
    w.show();
    return a.exec();
}
```
- Bottom Bar:** Type to locate (Ctrl...), Navigation icons (1 Pr..., 2 Se..., 3 So..., 4 So..., 5 Q..., 6 Me..., 7 Ve..., 8 Te...).

Qt Designer

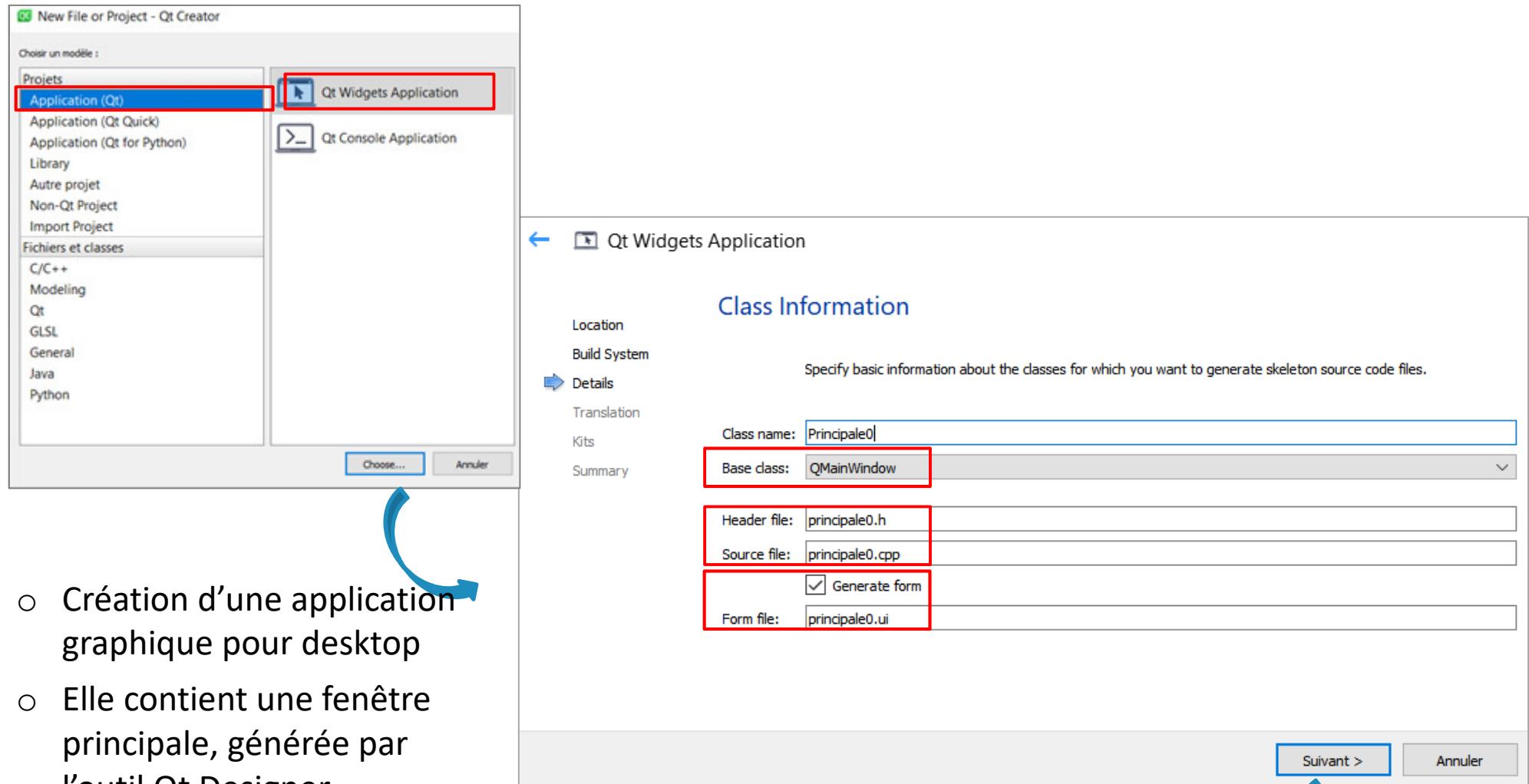
- ◆ Qt Designer est l'outil de dessin d'interfaces graphiques.
- ◆ Les éléments graphiques sont placés par glisser-déposer, et leurs propriétés précisées via des menus.
- ◆ L'interface graphique est sauvegardée sous la forme d'un fichier XML d'extension **.ui**.
- ◆ Lors de la compilation, l'utilitaire **uic**, fourni par Qt, génère les fichiers C++ avec les classes correspondant au contenu du fichier XML.



Qt Assistant – assistant nouveau projet (1/2)

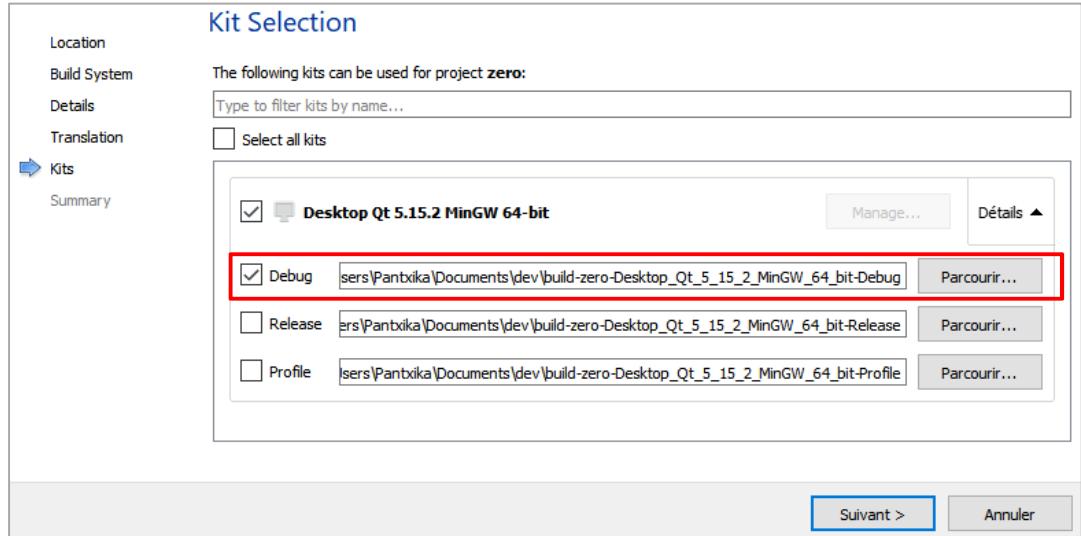
- ◆ Aide à la création des ‘bons’ fichiers correspondant au type d’application souhaitée

Fichier → Nouveau fichier ou projet ...



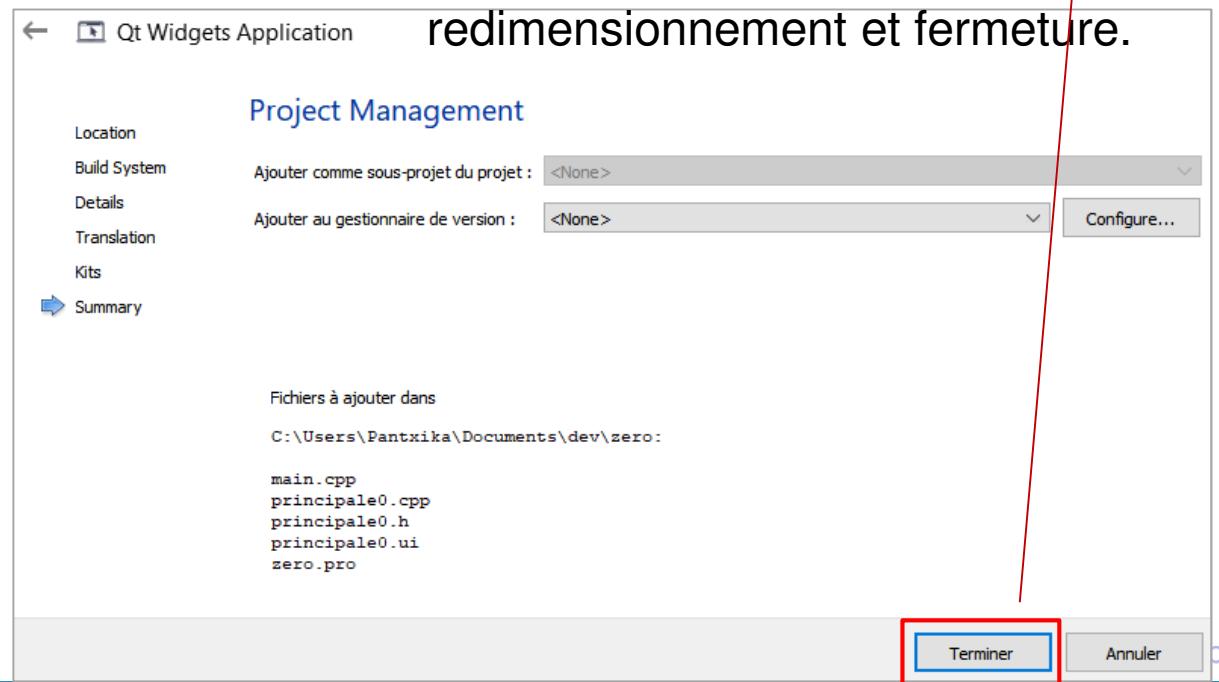
- Création d’une application graphique pour desktop
- Elle contient une fenêtre principale, générée par l’outil Qt Designer

Qt Assistant – assistant nouveau projet (2/2)



- L’application finale : une fenêtre principale vide, comportement classique : redimensionnable, déplaçable, avec barre de titre, menu système et boutons de redimensionnement et fermeture.

- Compilation en mode Debug uniquement, pour faciliter la mise au point du programme
 - Un répertoire est créé pour y ranger les ressources générées par l’Assistant : fichiers .h .cpp et .ui
 - La description des ressources du projet est rangée dans le fichier .pro
- A blue curved arrow points from the 'Debug' configuration in the first screenshot to the 'Terminar' button in the third screenshot.

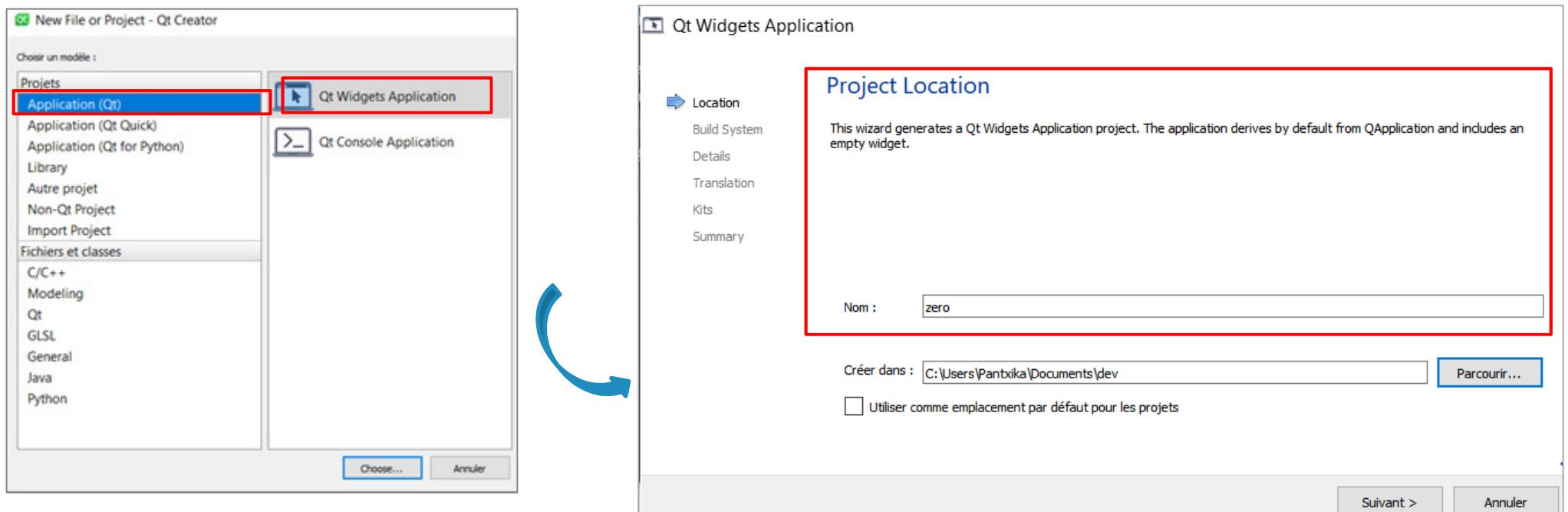


Notion de projet (1/2)

Projet Qt

- Qt Creator enregistre dans un fichier **projet** (`.pro`) toutes les informations / chemins d'accès vers les ressources participant à la création d'une application : modules de Qt utilisés, *chemins d'accès aux fichiers sources* (*), dépendances entre fichiers, paramètres passés au compilateur, répertoires de déploiement...

(*) Ainsi, un même fichier source (par exemple, un module composé de `pile.h` et `pile.cpp`) peut être utilisé dans le développement de plusieurs applications sans nécessité de dupliquer son code

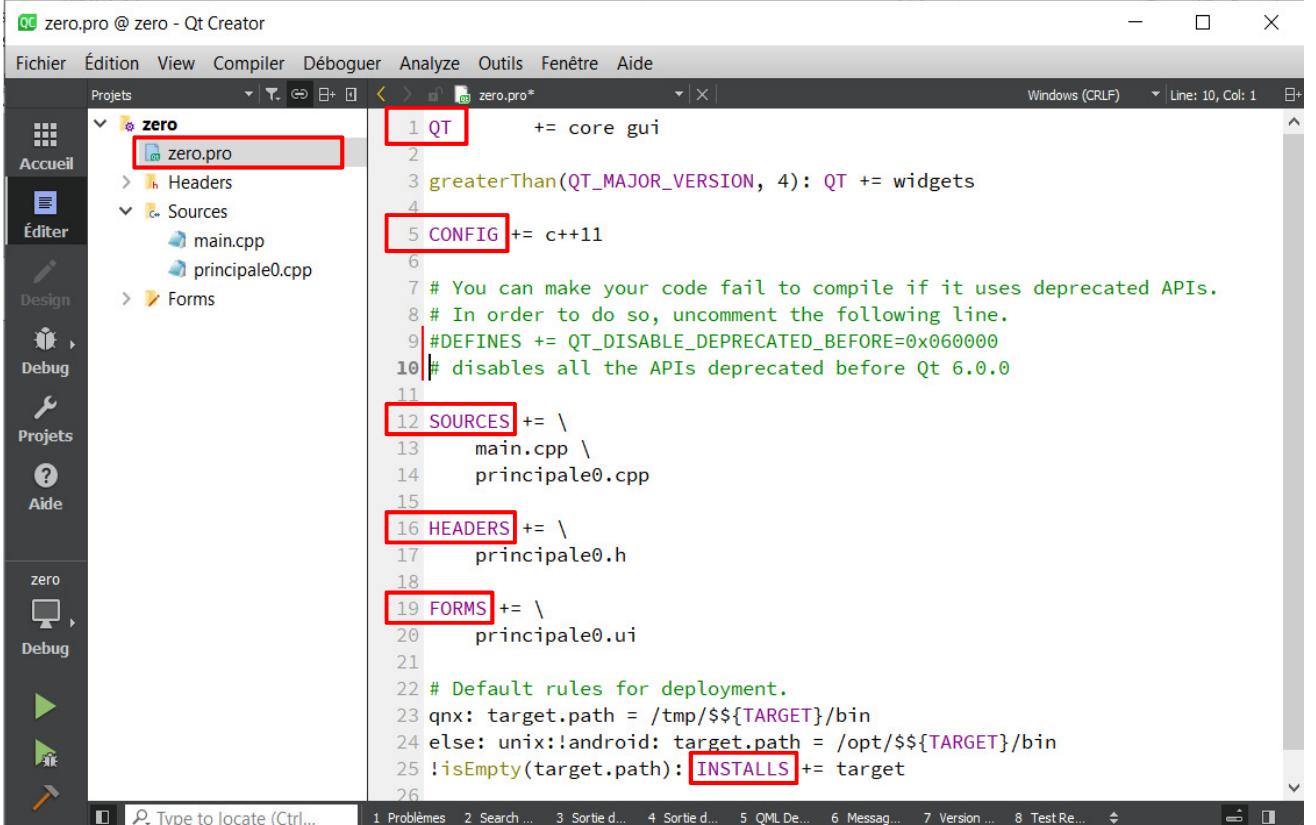


Notion de projet (2/2)

- Le fichier `.pro` est éditable : il peut être créé ‘à la main’, ou bien généré par Qt Assistant. Le paramétrage est fait par affectation de variables.

qmake

- Qt fournit le moteur de production (*Build system*) **qmake** permettant de créer la commande de compilation spécifique à la plateforme à partir du contenu du fichier `.pro`. Ainsi, sous les systèmes UNIX/Linux, qmake produira un **Makefile**



The screenshot shows the Qt Creator interface with the following details:

- Project View:** Shows a project named "zero" with files: zero.pro, Headers, Sources (main.cpp, principale0.cpp), and Forms.
- Code Editor:** Displays the content of zero.pro. Key parts highlighted with red boxes are:
 - Line 1: QT += core gui
 - Line 5: CONFIG += c++11
 - Line 12: SOURCES += \ main.cpp \
 principale0.cpp
 - Line 16: HEADERS += \
 principale0.h
 - Line 19: FORMS += \
 principale0.ui
 - Line 25: !isEmpty(target.path): INSTALLS += target
- Status Bar:** Shows tabs for Problemes, Search, Sortie d..., etc.

- Qt Creator fournit aussi des assistants (*wizard*) pour créer des modèles de projets servant à créer rapidement de nouveaux projets par duplication des modèles.

Vocation première de Qt

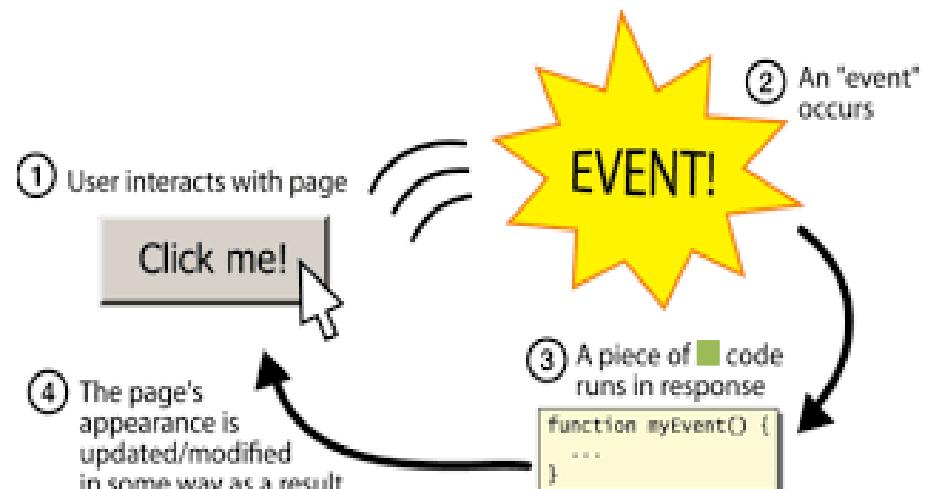
- ◆ Qt est principalement dédiée au développement d'interfaces graphiques en fournissant des éléments prédéfinis appelés **widgets**.
- ◆ Les **widgets** peuvent être utilisés pour créer ses propres **fenêtres** et **boîtes de dialogue** complètement prédéfinies.
(ouverture/enregistrement de fichiers, progression d'opération, etc)
- ◆ Les interactions avec l'utilisateur sont gérées par un mécanisme appelé **signal/slot**. Ce mécanisme est la base de la **programmation événementielle** des applications basées sur Qt.



Programmation événementielle (1/4)

Principe

- ◆ La programmation événementielle est une **programmation basée sur les événements**.
- ◆ Le programme est principalement défini par ses **réactions** aux différents événements qui peuvent se produire, c'est-à-dire des **changements d'état**, comme par exemple :
 - L'incrémantation d'une liste
 - Un mouvement de souris
 - L'appui sur une touche du clavier
- ◆ Les événements peuvent être provoqués **par l'action d'un utilisateur** ou bien **par le système**.
- ◆ **L'application est toujours prête à réagir à tout type d'événement !**



Programmation événementielle (2/4)

Architecture d'une application événementielle

- ◆ Une application événementielle est organisée autour de deux sections :
 - La première section est une **boucle principale**. Elle **déetecte** les événements
 - La seconde section **traite** les événements qui ont été détectés
- ◆ A chaque **événement** que l'application doit prendre en compte, il faut lui **associer** une **action à réaliser**.
- ◆ Cette action (sous-programme, méthode qui traite un événement), s'appelle un **gestionnaire d'évènement** (*event handler*).



Programmation événementielle (3/4)

Fonctionnement d'une application événementielle non graphique

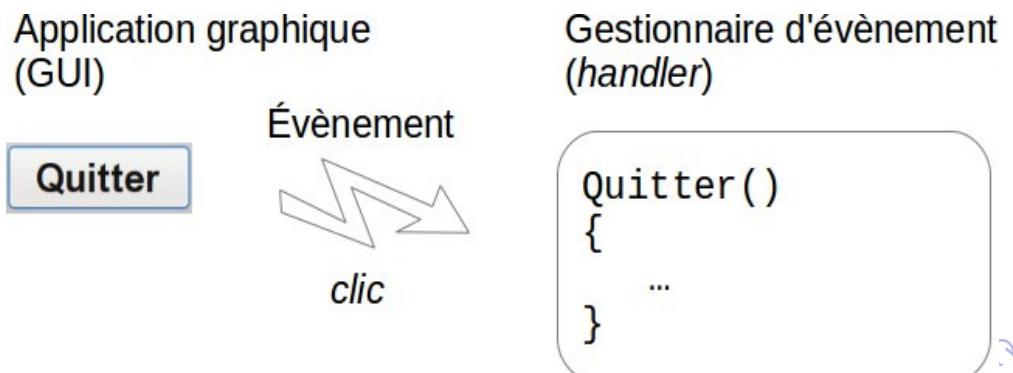
- ◆ Une fois lancée, l'application **se met en attente** des événements qui la concernent.
 - Cette attente constitue la **boucle principale**.
 - Les événements proviennent du système d'exploitation (par ex. horloge), d'objets de l'application, ou d'autres sources extérieures à l'ordinateur (ex. capteurs)
- ◆ Lorsqu'un **événement est détecté** par la boucle principale, le **gestionnaire d'événement** correspondant **est exécuté**.



Programmation événementielle (4/4)

Fonctionnement d'une application à interface graphique

- ◆ Une application graphique **est** une application événementielle
- ◆ L'application **crée** la **fenêtre principale** à l'intérieur de laquelle auront lieu les interactions avec l'utilisateur.
- ◆ Puis l'application **se met en attente** des événements qui la concernent.
 - Cette attente constitue la **boucle principale**.
 - Les événements proviennent du système d'exploitation, des objets graphiques de l'application, des mouvements de la souris, ou d'autres sources extérieures à l'ordinateur
- ◆ Lorsqu'un **événement** **est détecté** par la boucle principale, le **gestionnaire d'événement** correspondant **est exécuté**.



Mise en œuvre avec Qt

Signal - Slot

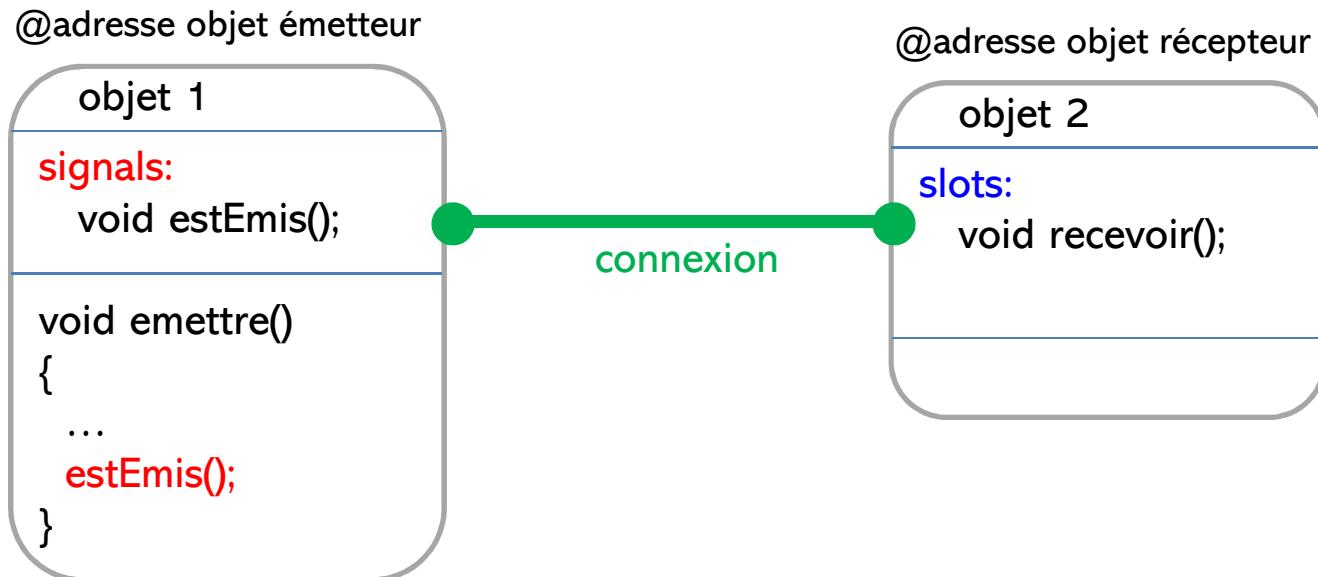
- ◆ La programmation événementielle des applications Qt est basée sur un **mécanisme appelé *signal / slot*** :
 - un ***signal*** est émis lorsqu'un **événement** particulier se produit.
 - un ***slot*** est un **gestionnaire d'événement**, c'est-à-dire un **sous-programme** qui sera appelé / exécuté en réponse à un ***signal*** particulier.
- ◆ L'association d'un ***signal*** à un ***slot*** est réalisée par une **connexion** grâce à la méthode (statique) **connect()**.
- ◆ Cette connexion relie 2 objets :
 - Un objet **émetteur**, chargé de produire le ***signal***
 - Un objet **récepteur**, chargé d'exécuter le ***slot***
- ◆ **Les classes graphiques de Qt possèdent de nombreux signaux et slots prédéfinis.** Il est possible d'en créer des supplémentaires en utilisant le mécanisme d'héritage.



Exemples (1/2)

Application non graphique

- ◆ Connexion entre l'événement `estEmis()` d'un objet et la méthode `recevoir()` d'un autre objet



- ◆ Effet à l'exécution
Lors de l'appel `objet1.emettre()`, un signal est émis vers l'`objet2`, qui, du coup, exécute la méthode `recevoir()`.

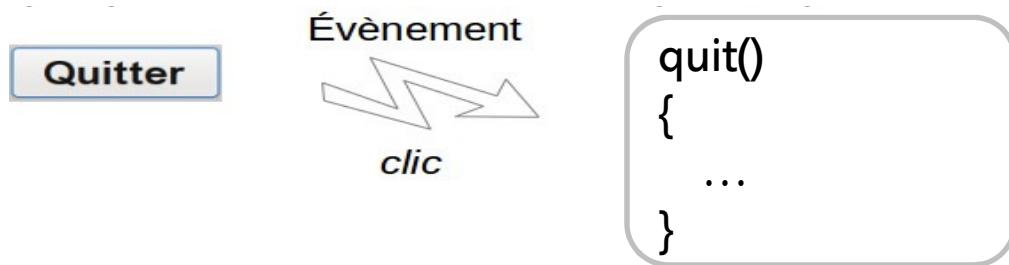
Exemples (2/2)

Application graphique

- Connexion entre l'événement clic d'un bouton situé à l'écran, et la méthode faisant arrêter l'application



- Effet à l'exécution
Le clic sur le bouton a pour effet de fermer l'application



Classe QObject – Tout est QObject – Tout est Q...

Classe QObject

- ◆ Toutes les classes de prédéfinies de l'API Qt ont un nom :
Commençant par la lettre **Q**
 - Puis suivi par un nom dont chaque mot commence par une majuscule.
Exemples : **QLabel**, **QPushButton**, ...
- ◆ L'API Qt est basée sur l'**héritage** : la classe **QObject** est la **classe mère** (*Base class*) **de toutes les classes Qt**.
- ◆ La classe **QObject** fournit à ses objets un certain nombre de capacités, comme :
 - La capacité de **communiquer entre eux** via le mécanisme *signal/slot*
 - Une gestion simplifiée de la mémoire
- ◆ Ainsi, par **héritage**, toutes les classes de Qt possèdent ces capacités.
- ◆ Les objets Qt (ceux héritant de **QObject**) peuvent s'organiser sous forme d'**arbres d'objets**. Ainsi, lorsqu'une classe est instanciée, on peut lui définir un **objet parent**.



Exemple de classe issue de QObject (1/2)

- ◆ Pour bénéficier de l'API Qt, il faut hériter de **QObject** ou d'une classe fille de **QObject**
- ◆ ☹ Le fichier contient des **macros**. La compilation nécessitera un outil spécifique fourni par Qt

maclasse.h

```

1 #ifndef...
2 #include <QObject>
3
4 class MaClasse : public QObject
5 {
6     Q_OBJECT
7     public:
8         MaClasse(QObject *parent = nullptr);
9     public slots:
10        void recevoir(int);           // slot : corps à définir dans .cpp
11    private:
12    signals:
13        void estEmis(int);          // signal : pas de corps à définir dans .cpp
14    //...
15 };
16 #endif // MACLASSE_H

```



Exemple de classe issue de QObject (2/2)

- ◆ L'outil **moc** fourni par Qt, transforme les fichiers sources en code 100% C++
- ◆ Cette particularité constitue un point faible de Qt...

maclasse.h

```

1 #ifndef...
2 #include <QObject>
3
4 class MaClasse : public QObject
{
5     Q_OBJECT
6     public:
7         MaClasse(QObject *parent = nullptr);
8     public slots:
9         void recevoir(int);      // slot : corps à définir dans .cpp
10    private:
11        signals:
12        void estEmis(int) ;    // signal : pas de corps à définir dans .cpp
13    //...
14 };
15 #endif // MACLASSE_H

```



maclasse.h

100% C++

```

1 #ifndef...
2 #include <QObject>
3
4 class MaClasse : public QObject
{
5
6
7
8
9
10
11
12
13
14 //...
15 };
16 #endif // MACLASSE_H

```

Classes QCoreApplication et QApplication (1/2)

Classe **QCoreApplication**

- ◆ Cette classe hérite de la classe **QObject**
- ◆ Dans une application Qt non graphique, il **doit y avoir une et une instance de cette classe**; **elle représente l'application**.
- ◆ Cette classe fournit la **boucle principale d'événements**, où tous les événements provenant du système d'exploitation (horloge, événement réseau,...) ou d'autres sources sont expédiés pour être traités.
- ◆ Sa méthode **exec()** se charge d'exécuter la boucle principale d'événements jusqu'à la fermeture du dernier objet de l'application.
- ◆ La classe **QCoreApplication** gère aussi l'initialisation et la finalisation de l'application, ainsi que ses paramètres.



Classes QCoreApplication et QApplication (2/2)

Classe QApplication

- ◆ Cette classe hérite de la classe **QCoreApplication**
- ◆ Tout comme la classe **QCoreApplication**
 - Dans une application Qt graphique, il doit y avoir **une et une instance de cette classe**, quel que soit le nombre de fenêtre graphiques contenues dans l'application; **elle représente l'application**.
 - Elle fournit la **boucle principale d'événements**, où tous les événements provenant du système d'exploitation (horloge, événement réseau,...) ou d'autres sources sont expédiés pour être traités.
 - Sa méthode **exec()** se charge d'exécuter la boucle principale d'événements jusqu'à la fermeture du dernier objet de l'application.
 - Elle gère aussi l'initialisation et la finalisation de l'application, ainsi que ses paramètres.
- ◆ **Ses particularités**
 - **L'instance de QApplication doit être créée avant tout objet graphique.**
 - Cette classe prend en charge des fonctionnalités spécifiques aux applications graphiques, notamment liées à l'initialisation et destruction des objets graphiques.



Exemple d'application Qt non graphique

main.cpp

```
1 #include <QCoreApplication>
2
3 int main(int argc, char **argv)
4 {
5     QCoreApplication app(argc, argv); // l'application
6
7     int retour = app.exec(); // exécute la boucle principale
8                     // d'événements
9
10    return retour;
11 }
```



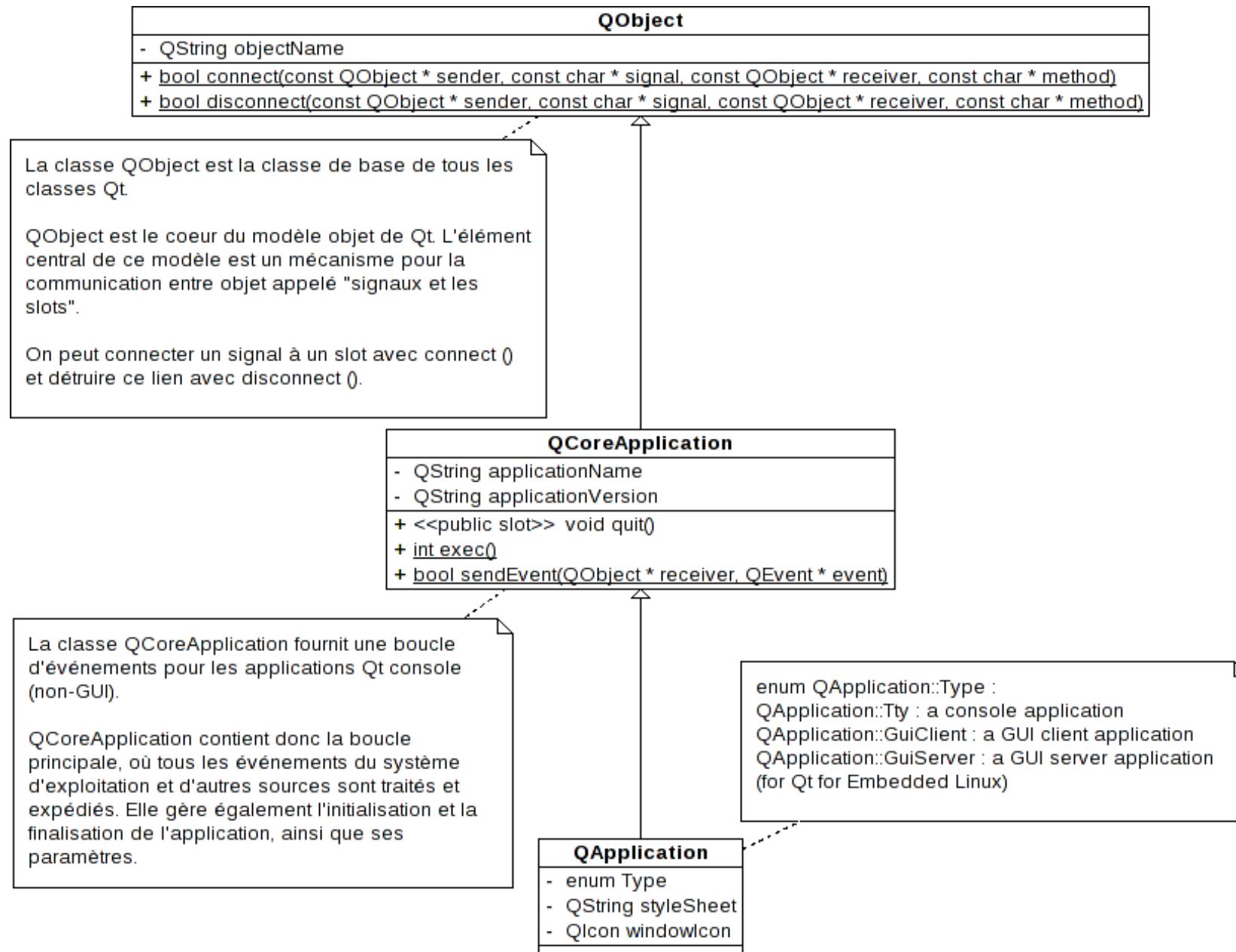
Exemple d'application Qt graphique

main.cpp

```
1 #include <QApplication>
2 #include "principale.h"
3
4 int main(int argc, char **argv)
5 {
6     QApplication app(argc, argv); // l'application
7
8     Principale ihm;           // ma fenêtre principale
9     ihm.show();               // affichage
10
11    int retour = app.exec(); // exécute la boucle principale
12                    // d'événements
13
14    return retour;
15 }
```

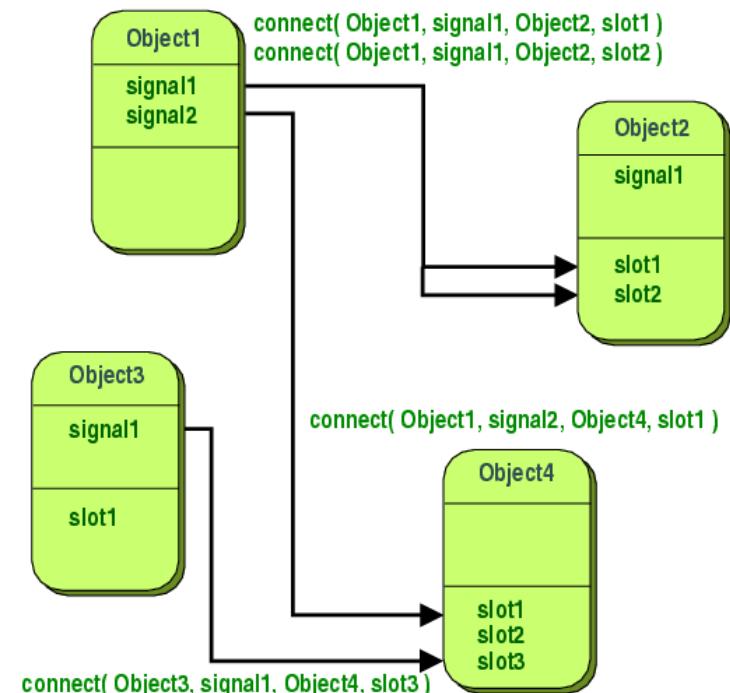


Hiérarchie des classes Qt – schéma partiel 1



Signal et Slot

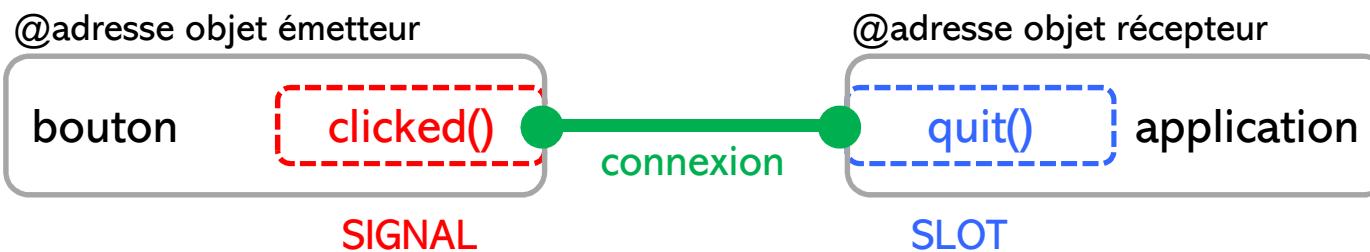
- ◆ Les signaux et slots constituent le mécanisme de communication entre objets sous Qt
 - ◆ Toute classe qui hérite de la classe **QObject** peut utiliser des signaux et des slots
 - ◆ La déclaration de la classe doit contenir la macro **Q_OBJECT**
 - ◆ Le mécanisme signal/slot est flexible et modulaire.
- Possibilité de :
- Connecter plusieurs signaux à un même slot
 - Connecter un signal à plusieurs slots.
- Attention** : l'ordre d'activation des slots est **arbitraire**.
- ◆ Faible couplage du mécanisme signal/slot :
 - Un émetteur ne connaît pas et n'a pas besoin de connaître le/les récepteurs
 - L'émetteur ne sait pas si le signal a été reçu
 - Le récepteur ne connaît pas l'émetteur
 - ◆ Contrôle de type
 - Les **types** des paramètres d'un duo signal/slot doivent être les mêmes
 - Un slot peut avoir moins de paramètres qu'un signal



Connexion - Déconnexion

- ◆ La connexion entre un signal et un slot est réalisée par la méthode **connect()**
- ◆ C'est une **méthode statique** de la classe **QObject**

Exemple dans appli graphique : connexion entre signal / slot prédéfinis



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

- ◆ Une connexion entre un signal et un slot peut être supprimée par la méthode **disconnect()**



Exemple dans application Qt graphique

main.cpp

```
1 #include <QApplication>
2 #include <QPushButton>
3
4 int main(int argc, char **argv)
5 {
6     QApplication app(argc, argv);
7     QPushButton bouton("Quitter");
8
9     // connexion du signal prédéfini clicked() de l'objet bouton
10    // au slot prédéfini quit() de l'objet app
11
12    QObject::connect(&bouton, SIGNAL(clicked()), &app, SLOT(quit()));
13    bouton.show();
14
15    return app.exec();
16 }
```



Signaux personnalisés

- ◆ Lorsque l'on crée une classe personnalisée, il est aussi possible de créer des signaux et slots propres à cette classe
- ◆ La classe doit hériter de la classe **QObject**
- ◆ **Créer** un signal personnalisé
 - Utiliser le mot-clé **signal** dans la déclaration de la classe
 - Il s'agit obligatoirement d'une méthode **void**
 - La méthode n'aura pas définition → pas de corps dans le fichier .cpp)
- ◆ **Emettre** un signal (dans le code de la classe où est déclaré le signal)
 - Utiliser la méthode **emit()** :

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

Propriétés

- ◆ Un signal peut être connecté à un autre signal. Ainsi, lorsque le premier signal est émis, il entraîne automatiquement l'émission du second signal
 - ◆ **L'émission d'un signal peut être automatique**
C'est le cas des classes prédéfinies, où les signaux sont déjà définis.
- Exemple** : l'appui sur un bouton (classe **QPushButton**) entraîne automatiquement l'envoi du signal prédéfini **clicked()**



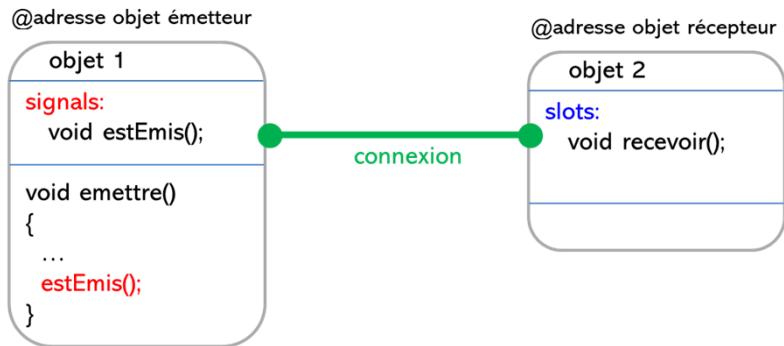
SLOTS PERSONNALISÉS

- ◆ Lorsque l'on crée une classe personnalisée, il est aussi possible de créer des signaux et slots propres à cette classe
- ◆ La classe doit hériter de la classe **QObject**
- ◆ **Créer** un signal personnalisé
 - Utiliser le mot-clé **slot** dans la déclaration de la classe
- ◆ A part cette particularité, les slots sont des méthodes ‘normales’
- ◆ Les slots peuvent donc être appelés par une autre méthode



Exemple dans application Qt non graphique (1/2)

- ◆ Mise en œuvre de l'exemple déjà présenté



maclasse.h

```

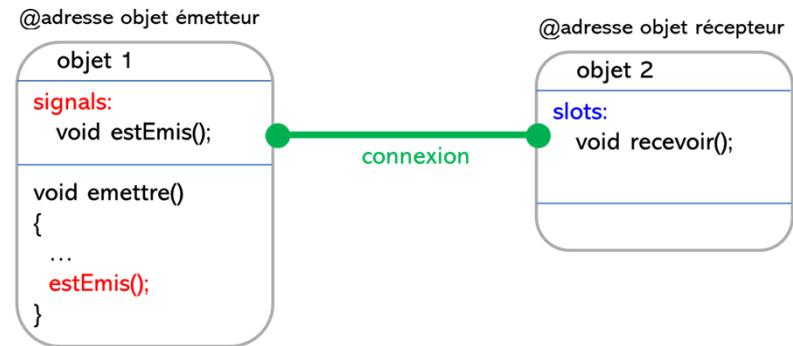
1 #ifndef...
2 #include <QObject>
3
4 class MaClasse : public QObject
5 {
6     Q_OBJECT
7     public:
8         MaClasse(int num = 0, QObject *parent = nullptr);
9         void emettre(); // émettra le signal
10    public slots:
11        void recevoir(int); // slot : corps à définir dans .cpp
12    private:
13        int _numero; // rang de création
14    signals:
15        void estEmis(int); // signal P. : pas de corps à définir dans .cpp
16    };
17 #endif // MACLASSE_H
    
```



Exemple dans application Qt non graphique (2/2)

Utilisation de la classe

- Dans le main.cpp, connexion des signaux/slots de 2 objets de la classe MaClasse
- Signatures de signal / slot compatibles
- L'appel objet1.emettre() enverra un signal, qui déclenchera l'exécution du slot recevoir().



main .cpp

```

1 #include <QCoreApplication>
2 #include "maclasse.h"
3
4 int main(int argc, char *argv[])
5 {
6     QCoreApplication a(argc, argv);
7     MaClasse objet1(1);
8     MaClasse objet2(2);
9
10    QObject::connect (&objet1, SIGNAL(estEmis(int)),
11                      &objet2, SLOT(recevoir(int)));
12    objet1.emettre();
13
14    return a.exec();
15 }

```



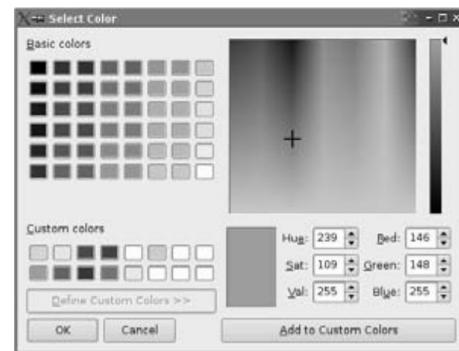
Notion de *widget*

- ◆ Une interface graphique utilisateur (GUI) Qt est composée de **widgets**.
- ◆ Dans Qt, le terme **widget** (**windows gadget**) est un terme générique pour désigner **un composant graphique prédéfini**.
- ◆ Un **widget** peut :
 - Afficher des informations
 - Recevoir des actions de l'utilisateur
 - Agir comme un **conteneur** pour d'autres widgets qui ont besoin d'être regroupés

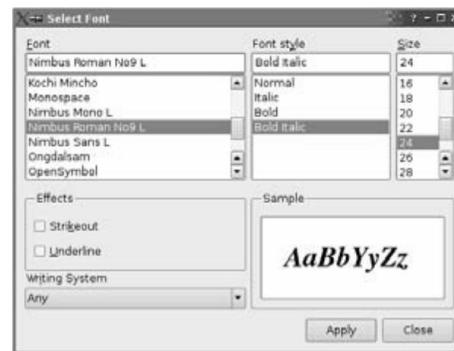


Les *widgets* pré-définis

- Qt fournit des widgets pré-définis, qui, une fois assemblés, composent les interfaces d'applications graphiques



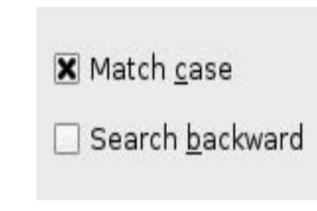
QColorDialog



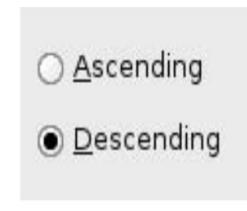
QFontDialog



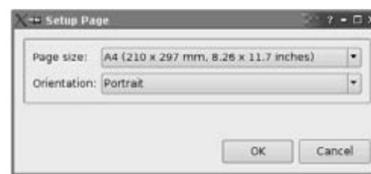
QPushButton QToolButton



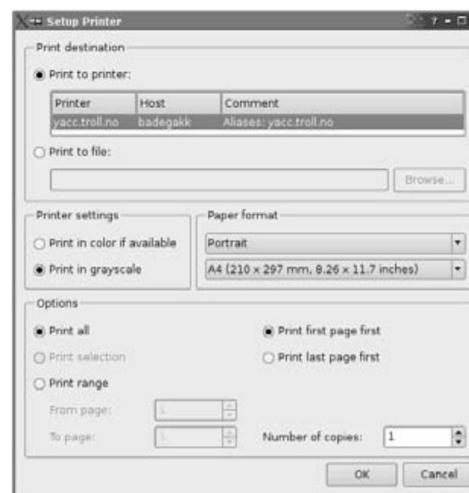
QCheckBox



QRadioButton



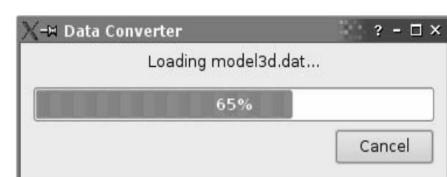
QPageSetupDialog



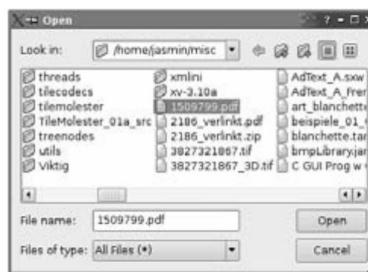
QPrintDialog



QInputDialog



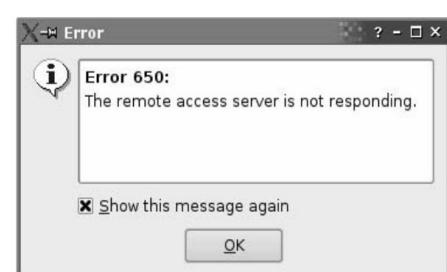
QProgressDialog



QFileDialog



QMessageBox



QErrorMessage

Les widgets pré-définis



QListView(liste)



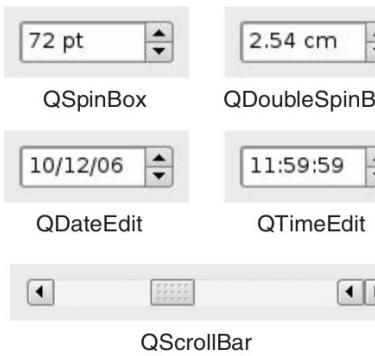
QTreeView



QListView(icônes)

	A	B	C
1	1043.23	250	
2	1037.39	178	
3	970.77		
4	1008.32		

QTableView



QSpinBox

QDoubleSpinBox

QComboBox

QDateEdit

QTimeEdit

QDateTimeEdit

QScrollBar

QSlider

Email is a wonderful thing to people whose role in life is to be on top of things. But not for me; my role is to be on the bottom of things.

QTextEdit



QDial

Warning: All unsaved information will be lost!

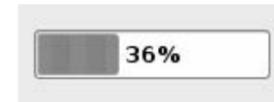
QLabel(texte)



QLabel (image)



QLCDNumber



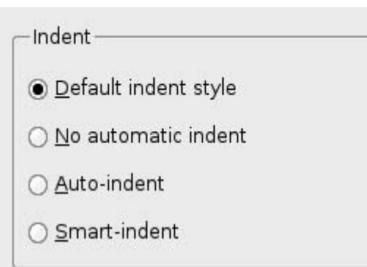
QProgressBar



Static Public Mem

- QUrl fromEncoded (co)
- QUrl fromLocalFile (co)
- QString fromPercentEn (co)

QTextBrowser

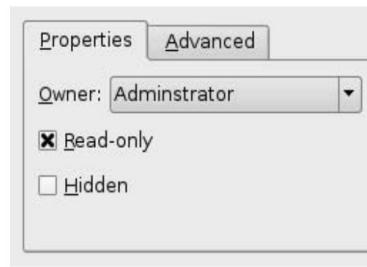


QGroupBox

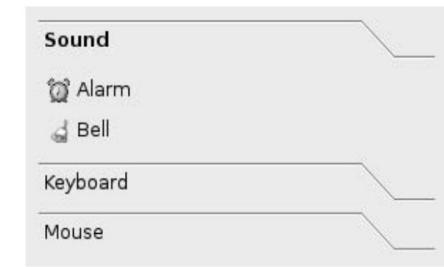
According to *Times* magazine, "the average Yaleman, Class of '24, makes \$25,111 a year."

Rich text

QFrame



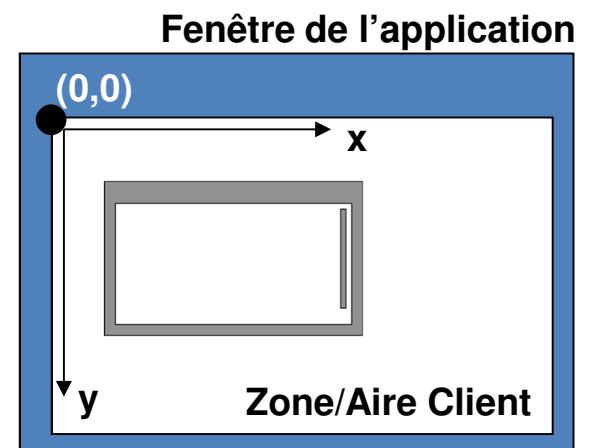
QTabWidget



QToolBox

La classe QWidget

- ◆ La classe **QWidget** fournit aux objets graphiques la **capacité d'affichage et la gestion des événements**.
- ◆ Elle est la **classe mère** (*Base class*) de toutes les classes servant à créer des classes graphiques.
- ◆ Les **widgets** :
 - sont créées "cachés" (*hide/show*)
 - sont capables de se "peindre" (*paint.repaint/update*)
 - sont capable de recevoir les événements souris, clavier
 - sont tous rectangulaires (*x, y, width, height*)
 - sont initialisés par défaut en coordonnées **0,0** de la zone Client de la fenêtre principale de l'application
 - sont ordonnés suivant l'axe **z** (la profondeur)
 - peuvent avoir un **widget parent** et des **widgets enfants**



Exemple d'application Qt graphique

Exemple 1

- ◆ → voir la déclaration de la variable `monWidget`

main.cpp

```
1 #include <QApplication>
2 #include <QWidget>           // nécessaire pour utiliser un widget
3
4 int main(int argc, char**argv)
5 {
6     QApplication app(argc, argv);
7     QWidget monWidget;          // objet widget qui n'a pas de parent
8                             // par défaut, caché, il faut l'afficher
9
10    monWidget.show();           // affichage de la fenêtre, zone client 'vide'
11
12    // exécution de la boucle d'événements
13    int ret = app.exec();
14
15    // lorsque l'utilisateur ferme la fenêtre, on sort de l'itération
16    // et l'application est arrêtée
17    return ret;
18 }
```



Exemple d'application Qt graphique

Exemple 2

- ◆ → voir la déclaration de la variable monWidget

main.cpp

```
1 #include <QApplication>
2 #include <QWidget>           // nécessaire pour utiliser un widget
3
4 int main(int argc, char**argv)
5 {
6     QApplication app(argc, argv);
7     QWidget* monWidget = new QWidget(nullptr);
8             // objet pointé par monWidget n'a pas de parent
9             // par défaut, caché, il faut l'afficher
10
11    monWidget->show();          // affichage de la fenêtre, zone client 'vide'
12
13    // exécution de la boucle d'événements
14    int ret = app.exec();
15
16    // lorsque l'utilisateur ferme la fenêtre, on sort de l'itération
17    // et l'application est arrêtée
18    return ret;
19 }
```



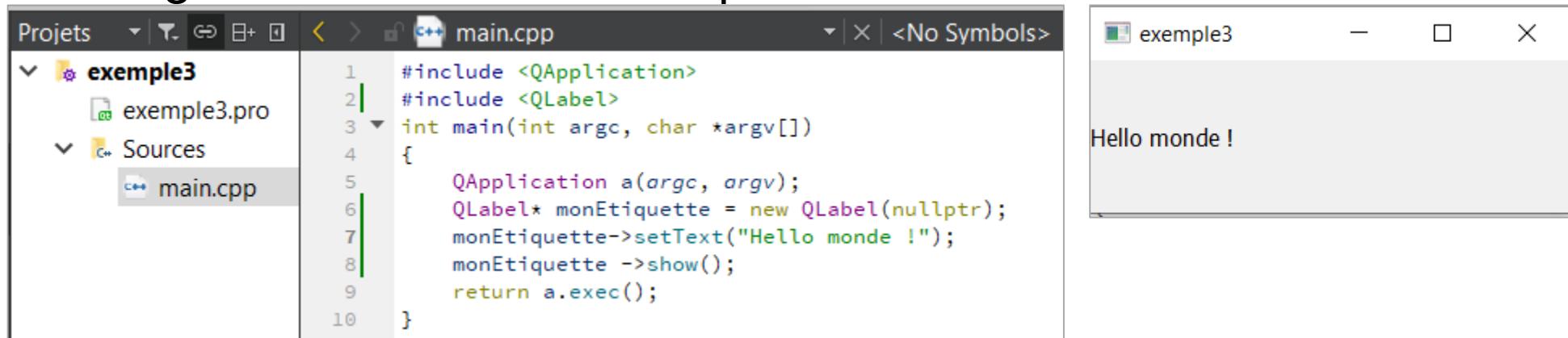
Fenêtre

Dans Qt : Notion de *fenêtre*

- ◆ Un **widget qui n'a pas de widget parent** est appelé une **fenêtre**.
- ◆ Un widget qui n'est pas une fenêtre est un **widget enfant**, affiché dans son **widget parent**.

Exemple

Affichage d'un QLabel en tant que fenêtre



```

Projets   |   main.cpp   |   <No Symbols>
-----|-----|-----
exemple3 |   |   |
  |- exemple3.pro |   |
  |- Sources |   |
    |- main.cpp |   |
-----|-----|-----
1 #include <QApplication>
2 #include <QLabel>
3 int main(int argc, char *argv[])
4 {
5     QApplication a(argc, argv);
6     QLabel* monEtiquette = new QLabel(nullptr);
7     monEtiquette->setText("Hello monde !");
8     monEtiquette ->show();
9     return a.exec();
10 }

```

- ◆ La plupart des widgets Qt sont principalement utilisés comme widgets enfants de fenêtres de plus haut niveau.



Fenêtre WIMP — rappel source : IHM-Chapitre4-ElémentsGraphiques.pdf

En IHM – Interfaces graphiques WIMP : Notion de *fenêtre*

♦ WIMP

Acronyme désignant les interfaces graphiques qui intègrent des composants graphiques en basées sur les concepts suivants :

- **Window** : Fenêtre (zone d'interaction indépendante).
- **Icon** : Éléments graphiques visuels (images, icônes de boutons, de champs de texte, de bulles d'aide, ...) représentant un document, un programme,...
- **Menu** : Choix d'actions regroupés dans des contrôles typés (barre de menus, menus déroulants, contextuels, circulaires,...)
- **Pointer** : Manipulé par la souris, il indique le point de l'écran où l'utilisateur peut interagir avec les autres composants (par pointage, sélection, tracé, glisser-déposer,...)

♦ Ressource à consulter

IHM-Chapitre4-ElémentsGraphiques.pdf



Fenêtre WIMP — rappel source : IHM-Chapitre4-ElémentsGraphiques.pdf

Extrait de IHM-Chapitre4-ElémentsGraphiques

Window - Fenêtre

- Représente l'interface standard de présentation de l'information à l'utilisateur
- 4 types de fenêtre
 - Fenêtre **principale / mère** (une seule) :
 - Toute application possède une fenêtre principale qui s'ouvre dès son lancement
 - Elle peut être rendue invisible si non intéressante pour l'utilisateur
 - Fenêtre **secondaire / fille**
 - Une application peut avoir 0 ou plusieurs fenêtres secondaires.
 - Une fenêtre secondaire peut être indépendante ou fonctionner en étroite liaison avec sa fenêtre principale
 - Fenêtre **utilitaire**
 - Fenêtre de **dialogue** (dite aussi **boîte de dialogue**)
...parmi lesquelles les **boîtes de message**

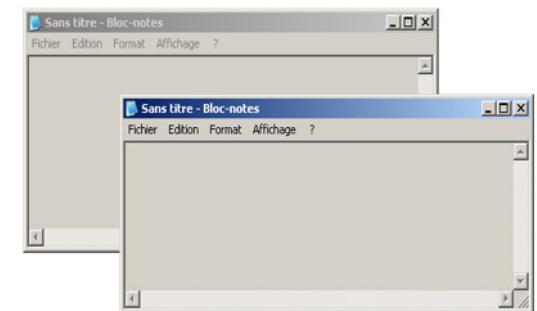
□ Organisation des interfaces avec fenêtres

Les fenêtres d'une interface WIMP peuvent être organisées selon 3 modalités différentes :

- Interfaces à Document Simple (**SDI-Single Document Interface**)
- Interfaces à Documents Multiples (**MDI-Multiple Document Interface**)
- Interfaces à Documents Tabulés (**TDI-Tabbed Document Interface**)

□ Interface à Document Simple (**SDI-Single Document Interface**)

- **1 unique fenêtre**
= LE document en cours
- la métaphore du document remplace celle de l'application



Fenêtre WIMP – rappel source : IHM-Chapitre4-ElémentsGraphiques.pdf

Extrait de IHM-Chapitre4-ElémentsGraphiques

Window - Fenêtre

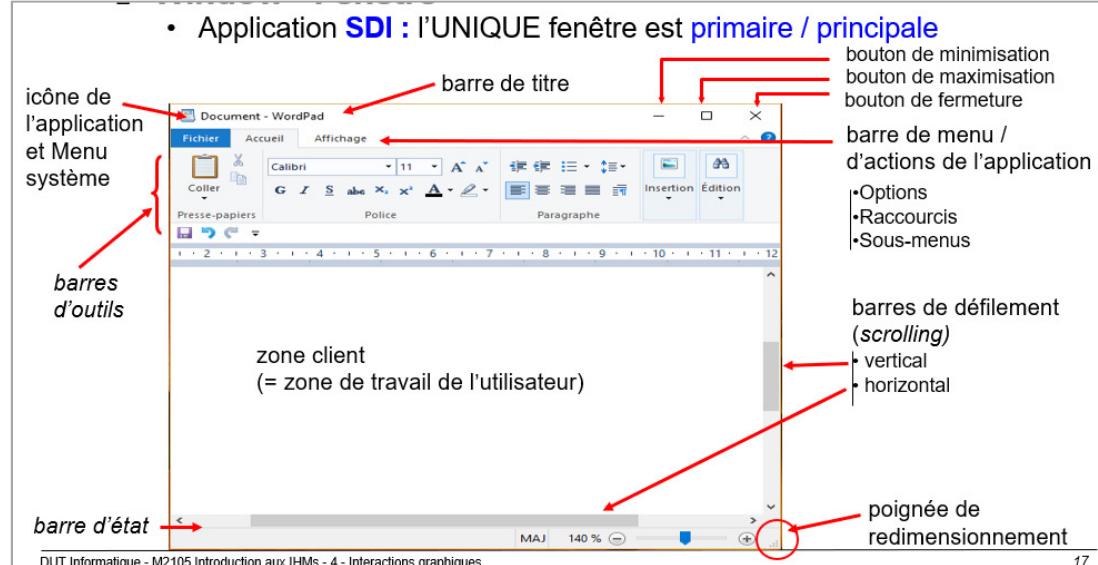
- Fenêtre primaire / principale**

Dite **standard** si elle contient les 10 composants **principaux** suivants :

- 1 poignée de redimensionnement
- 1 barre de titre
(généralement, le titre d'une fenêtre primaire est le titre de l'application elle-même)
- 1 menu système
- 3 boutons de minimisation / maximisation / fermeture
- 1 barre de menu / d'actions
- 2 barres de défilement (horizontal / vertical)
- 1 zone client

Elle peut aussi contenir les composants **complémentaires** suivants :

- des *barres d'outils*
- 1 *barre d'état*



- Fenêtre utilitaire**

- Palette d'options



- Fenêtres jaillissantes (*pop-up*)
 - infobulle, bulle d'aide, aide contextuelle



Fenêtre WIMP — rappel source : IHM-Chapitre4-ElémentsGraphiques.pdf

Extrait de IHM-Chapitre4-ElémentsGraphiques

Window - Fenêtre

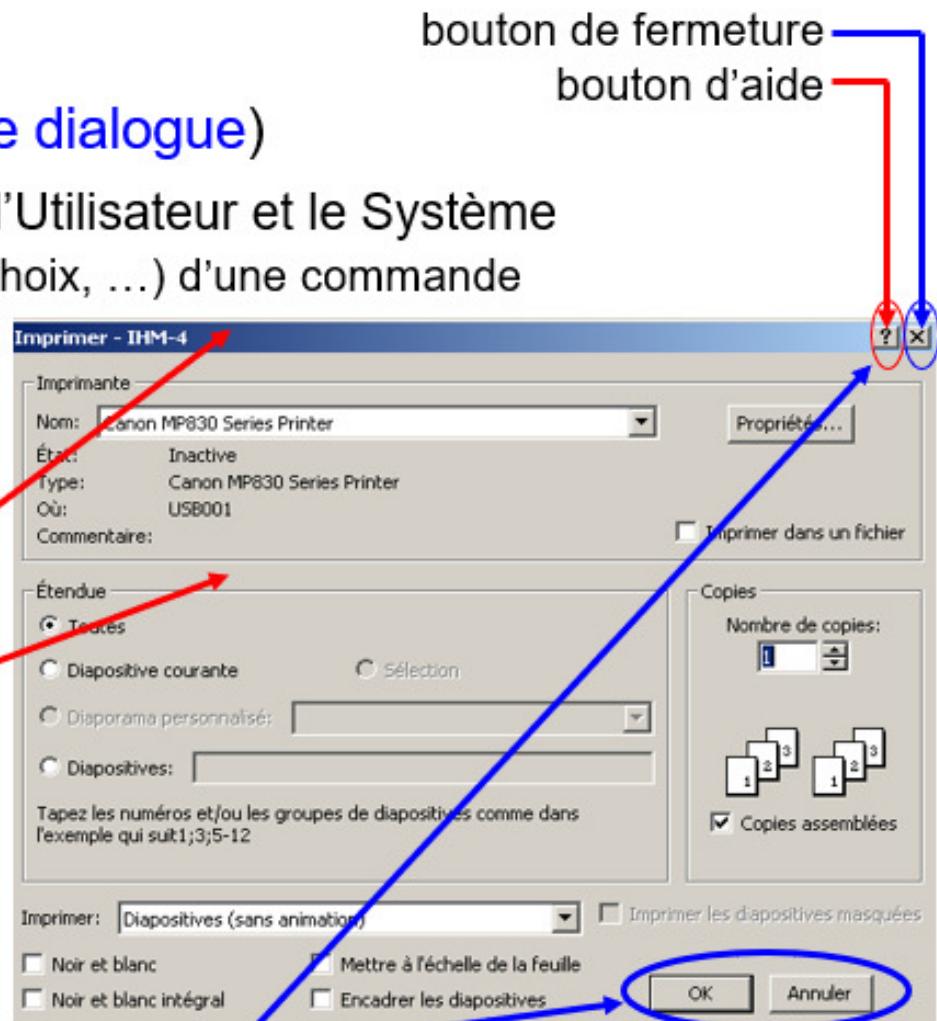
- **Fenêtre de dialogue (= boîte de dialogue)**

– Rôle : assure le dialogue entre l'Utilisateur et le Système

- Entrée d'informations (saisie, choix, ...) d'une commande en cours de spécification
- Visualisation d'informations

– Caractéristiques physiques

- Non redimensionnable
- Déplaçable
- Composants
 - Barre de titre
 - Région client
 - PAS de barre de défilement
 - barre de menus
 - barre d'outils
 - barre d'état



- Un/des composants gérant la **fin du dialogue** : l'utilisateur a toujours le choix : **validation de l'action initiée** ou bien **annulation**

Fenêtre WIMP — rappel source : IHM-Chapitre4-ElémentsGraphiques.pdf

Extrait de IHM-Chapitre4-ElémentsGraphiques

□ Window - Fenêtre

- **Fenêtre de dialogue (= boîte de dialogue)**

– 2 types de fenêtres de dialogue

- **Fenêtre modale** :

– **Une fois initié, le dialogue est obligatoire jusqu'à sa finalisation.**

cad : tant que la fenêtre est ouverte, la Boîte de Dialogue bloque toute interaction de l'utilisateur avec d'autres fenêtres de l'application

– **La fenêtre est déplaçable** pour laisser l'utilisateur voir la tâche amont

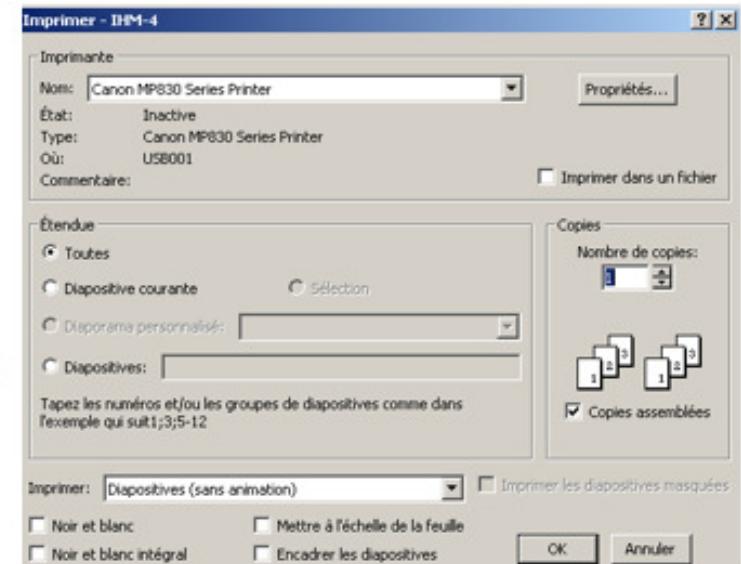
- **Fenêtre amodale (non modale)** :

– **Après ouverture du dialogue, l'utilisateur reste autorisé à interagir en parallèle avec d'autres fenêtres.**

cad : l'utilisateur peut laisser la fenêtre momentanément ouverte pour interagir dans une autre fenêtre de l'application

– **La fenêtre est déplaçable** pour laisser l'utilisateur voir la tâche amont

modale



amodale



Fenêtre WIMP

 — rappel source : IHM-Chapitre4-ElémentsGraphiques.pdf

Extrait de IHM-Chapitre4-ElémentsGraphiques

Window - Fenêtre

- Boîte de message
 - = Fenêtre de dialogue simplifiée
 - Rôle
 - Fournit un message à l'utilisateur
 - Conseil, Information, Avertissement
 - ...avec demande de confirmation
 - ...avec demande d'action immédiate
 - Pas d'autre entrée d'information de l'utilisateur !
 - Type :
 - Fenêtre Modale
 - Caractéristiques physiques simplifiées
 - Barre de titre
 - 1 icône représentant la nature du message
 - Région client = Texte du message
 - 1/2/3 boutons



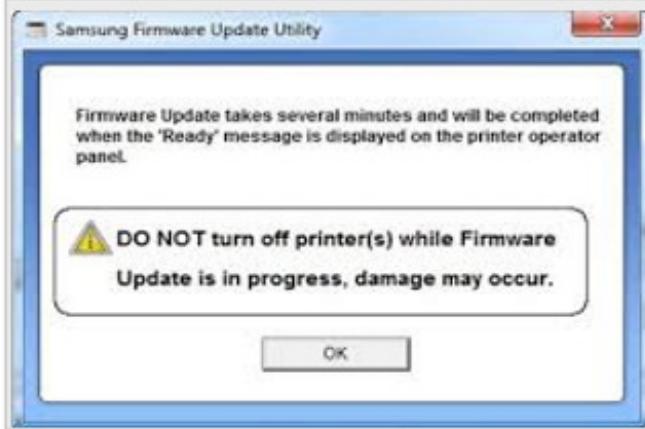
Fenêtre WIMP — rappel source : IHM-Chapitre4-ElémentsGraphiques.pdf

Extrait de IHM-Chapitre4-ElémentsGraphiques

Exemples

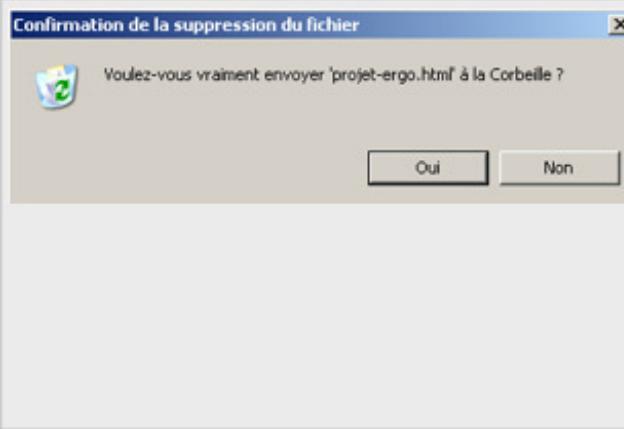
Message d'**information**

- 1 bouton (OK)



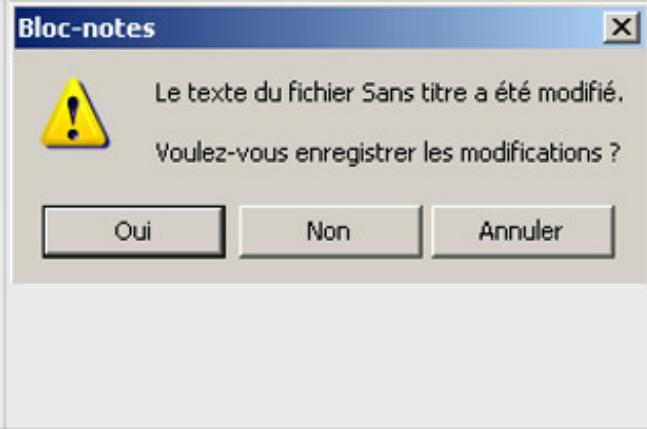
Message d'information avec **demande de confirmation**

- 2 boutons (Oui/Non)



Message d'information avec **demande d'action immédiate**

- 2/3 boutons :
(Réessayer/Annuler) ou bien
(Oui/Non/Annuler)



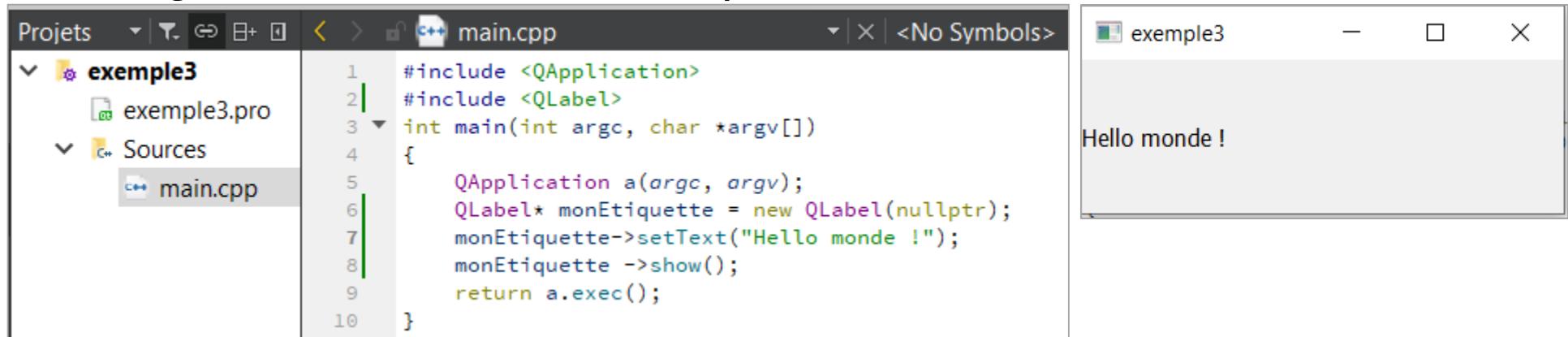
Fenêtre

Dans Qt : Notion de *fenêtre*

- ◆ Un **widget qui n'a pas de widget parent** est appelé une **fenêtre**.
- ◆ Un widget qui n'est pas une fenêtre est un **widget enfant**, affiché dans son **widget parent**.

Exemple

Affichage d'un QLabel en tant que fenêtre



```

Projets  |  Projets  |  main.cpp  |  <No Symbols>
└── exemple3
    └── Sources
        └── main.cpp
1 #include <QApplication>
2
3 #include <QLabel>
4
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8     QLabel* monEtiquette = new QLabel(nullptr);
9     monEtiquette->setText("Hello monde !");
10    monEtiquette ->show();
11    return a.exec();
12 }

```

- ◆ Mais la plupart de ces widgets Qt sont principalement utilisés comme widgets enfants de fenêtres de plus haut niveau.

Fenêtre

Fenêtres de haut niveau

La plupart des widgets sont affichés dans des fenêtres de plus haut niveau

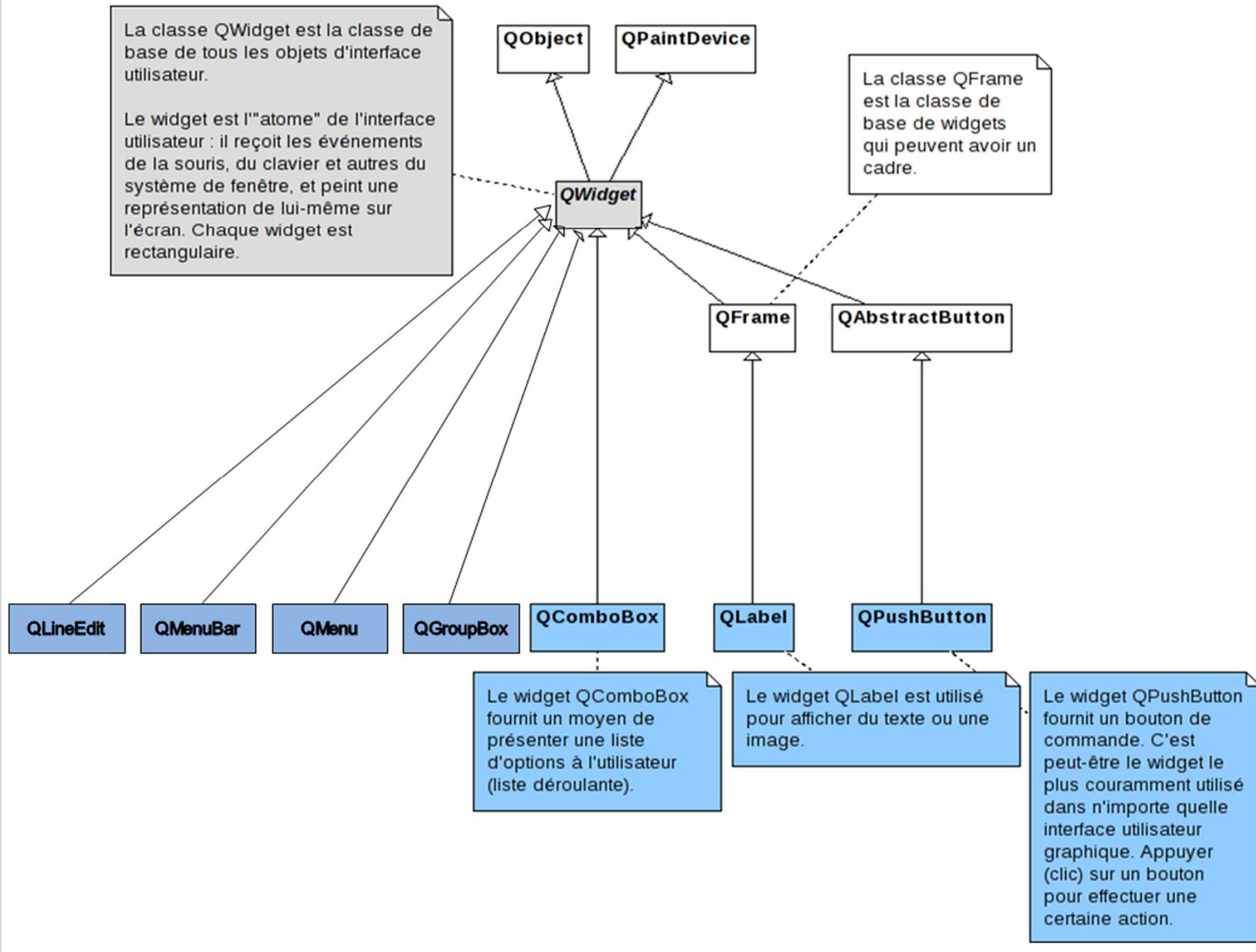
- ◆ Fenêtre principale : [QMainWindow](#)
- ◆ Fenêtre de dialogue : [QDialog](#)
- ◆ [QMainWindow](#) et [Qdialog](#) sont des classes dérivées de [QWidget](#)

Fenêtre principale

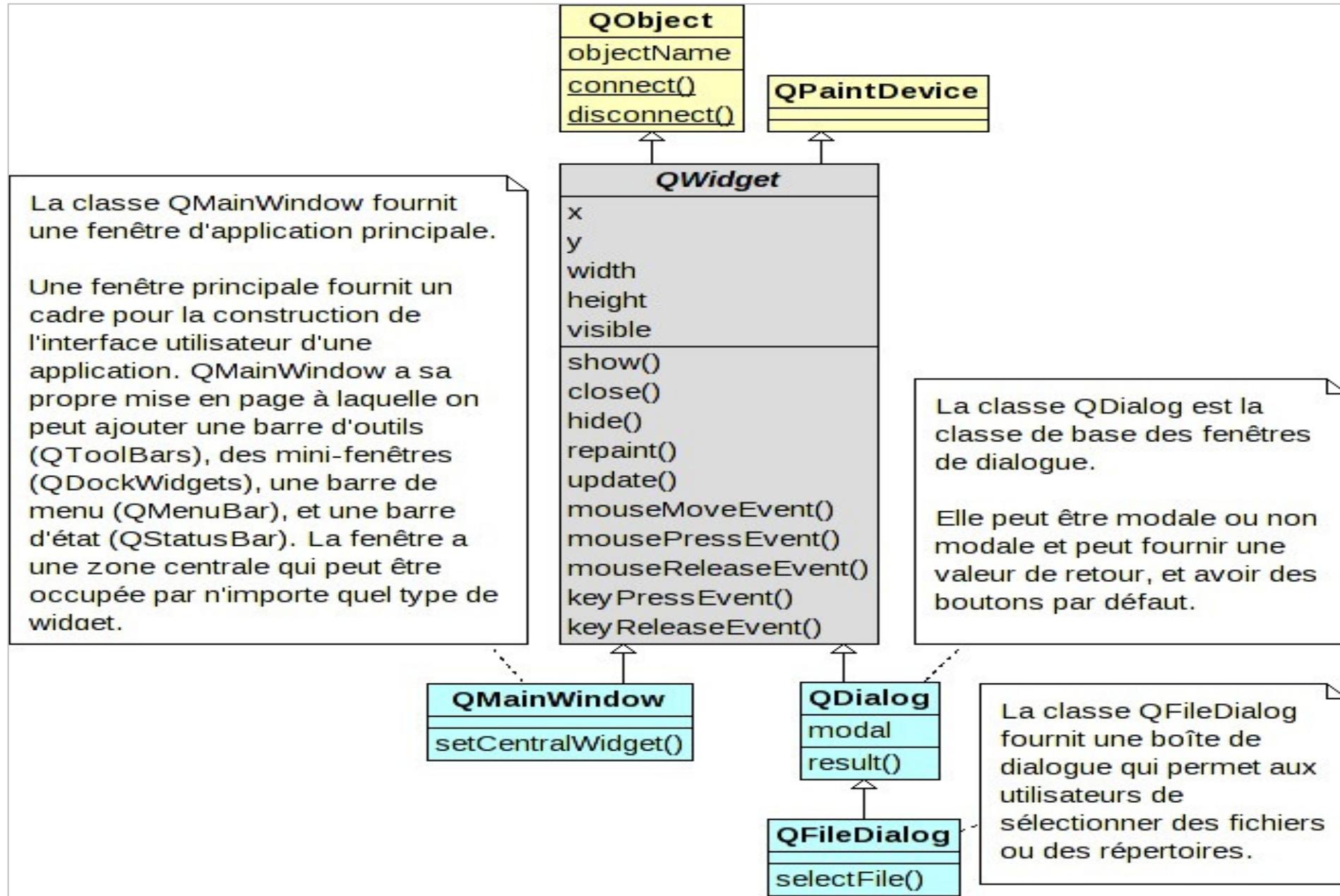
La classe à privilégier pour créer une fenêtre principale est [QMainWindow](#)



Hiérarchie des classes Qt – schéma partiel 2 (1/2)



Hiérarchie des classes Qt – schéma partiel 2 (2/2)



Accès aux propriétés d'une classe Qt (1/2)

Propriétés d'une classe Qt

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

Convention pour nommer les accesseurs d'une propriété

- ◆ `propriete()` pour lire la propriété
- ◆ `setPropriete()` pour la modifier.

Documentation en ligne complète

<https://doc.qt.io/>



Accès aux propriétés d'une classe Qt (2/2)

Exemple

♦ Documentation

QLineEdit Class Reference

The QLineEdit widget is a one-line text editor. More...

```
#include <QLineEdit>
```

Properties

- | | |
|--|---------------------------------------|
| ▪ acceptableInput : const bool | ▪ inputMask : QString |
| ▪ alignment : Qt::Alignment | ▪ maxLength : int |
| ▪ cursorMoveStyle : Qt::CursorMoveStyle | ▪ modified : bool |
| ▪ cursorPosition : int | ▪ placeholderText : QString |
| ▪ displayText : const QString | ▪ readOnly : bool |
| ▪ dragEnabled : bool | ▪ redoAvailable : const bool |
| ▪ echoMode : EchoMode | ▪ selectedText : const QString |
| ▪ frame : bool | ▪ text : QString |
| ▪ hasSelectedText : const bool | ▪ undoAvailable : const bool |
- 58 properties inherited from **QWidget**
 ▪ 1 property inherited from **QObject**

QLineEdit



Getter et Setter d'une propriété

```

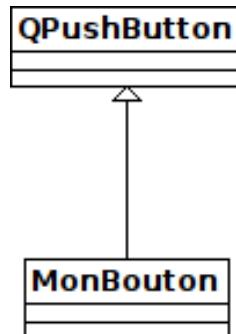
1 // Pour récupérer la chaîne de caractères contenue dans un QLineEdit
2 QString uneChaine1 = monLineEdit1.text();
3 QString uneChaine2 = monLineEdit2->text();
4
5 // Pour modifier le contenu d'un champ de texte de type QLineEdit
6 monLineEdit1.setText("Bonjour");
7 monLineEdit2->setText("Bonjour");

```

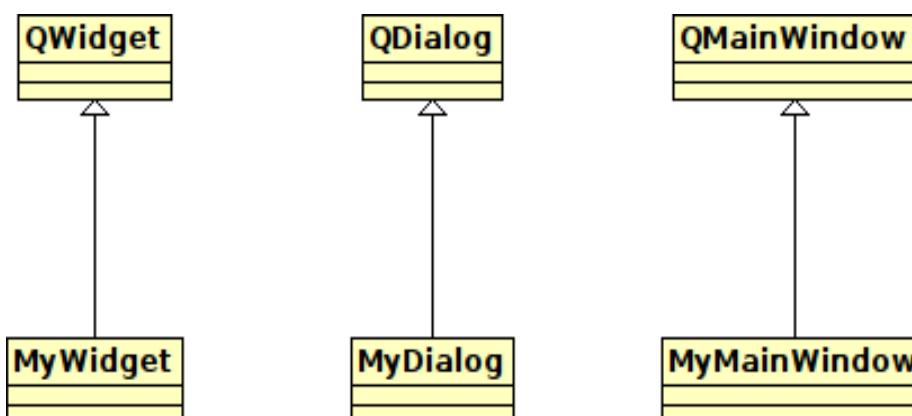


Création de *widget* versus *fenêtre* personnalisés

- La création de **widgets personnalisés** est réalisée par **héritage** de la classe **QWidget** ou d'une **classe fille** de **QWidget**.

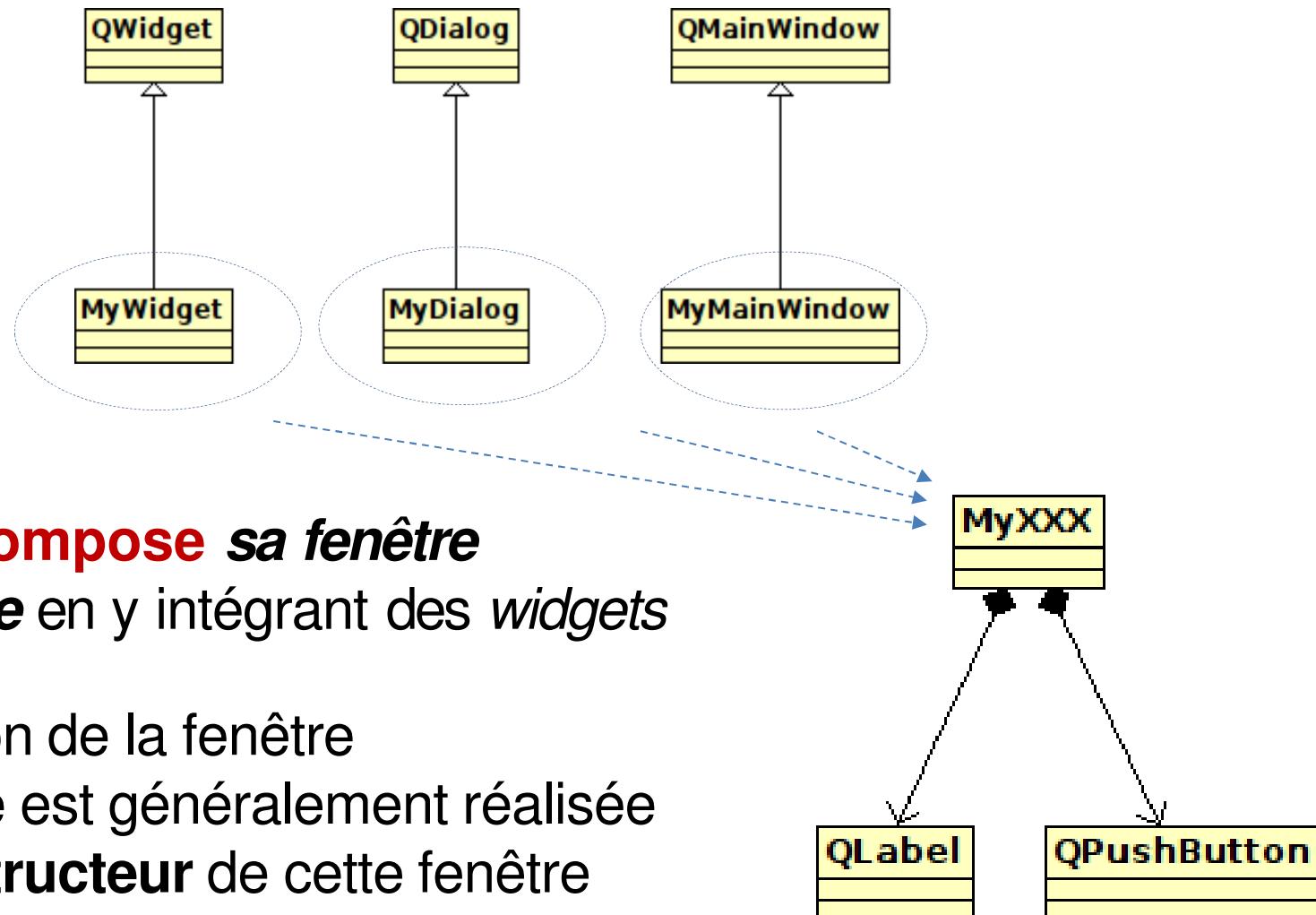


- La création de **fenêtres personnalisées** est réalisée par **héritage** des classes **QWidget**, **QDialog** ou **QMainWindow**



Création de *fenêtres personnalisées*

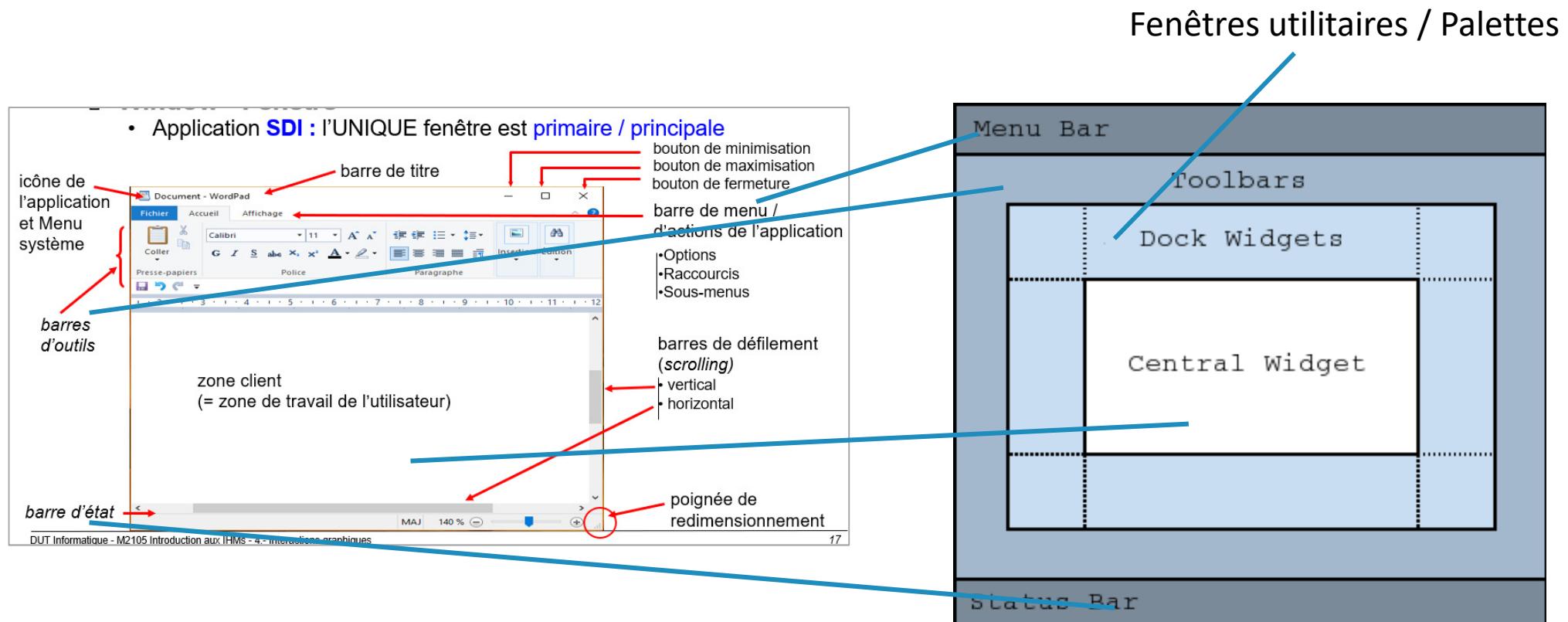
- ◆ La **création** de *fenêtres personnalisées* est réalisée par **héritage** des classes `QWidget`, `QDialog` ou `QMainWindow`



- ◆ *Ensuite, on compose sa fenêtre personnalisée en y intégrant des widgets*
- ◆ La composition de la fenêtre personnalisée est généralement réalisée dans le **constructeur** de cette fenêtre

Création de fenêtre personnalisée

La classe **QMainWindow** propose un squelette (de fenêtre) adapté à la création de fenêtres personnalisées



Exemple d'application avec fenêtre personnalisée

mymainwindow.h

```
1 #include <QMainWindow>
2 #include <QLabel>
3
4 // MA classe fenêtre principale
5 class MyMainWindow : public QMainWindow
6 {
7     Q_OBJECT
8     public:
9     MyMainWindow( QWidget *parent=0 );
10    ~MyMainWindow();
11
12    private:
13    QLabel *label; // pointeur sur une instance de QLabel
14}
```



Exemple d'application avec fenêtre personnalisée

mymainwindow.cpp

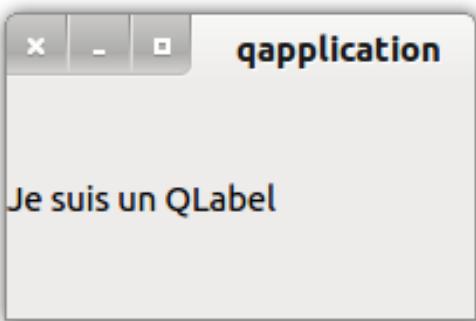
```
1 #include "MyMainWindow.h"
2
3 MyMainWindow::MyMainWindow(QWidget *parent) : QMainWindow(parent)
4 {
5     // instantiation d'un QLabel en indiquant son parent (this => moi)
6     label = new QLabel("Je suis un QLabel", this);
7
8     // on fixe le widget QLabel au centre de la fenêtre
9     setCentralWidget(label);
10 }
11
12 MyMainWindow::~MyMainWindow()
13 {
14 }
```



Exemple d'application avec fenêtre personnalisée

main.cpp

```
1 #include <QApplication>
2 #include "MyMainWindow.h"
3
4 int main(int argc, char **argv)
5 {
6     QApplication app(argc, argv); // mon objet application
7     MyMainWindow maFen;          // mon objet fenêtre
8
9     maFen.show();                // affichage de la fenêtre
10    return app.exec();           // boucle d'événement de l'application
11 }
```



Introduction

- ◆ La **gestion des dispositions** des objets graphiques comporte 2 volets :
 - Le **positionnement** des objets graphiques dans le conteneur (widget ou fenêtre) qui les contient.
 - Le **redimensionnement** des objets graphiques lors du redimensionnement du conteneur.
- ◆ La taille et la position de chaque objet graphique d'une interface doivent être appropriées, c'est-à-dire telle de telle sorte que l'interface graphique puisse s'adapter automatiquement aux diverses polices, langues, plateformes, ainsi qu'aux actions de redimensionnement réalisées par les utilisateurs.
- ◆ La gestion des dispositions est donc un volet important de la conception d'interfaces graphiques.



Gestion des dispositions (1/4)

- ◆ Il existe plusieurs moyens de gérer la disposition des objets graphiques dans un conteneur (widget ou fenêtre) qui les contient :
 - **Disposition absolue** : le positionnement et dimensionnement des objets graphiques est fixé par des constantes dans le constructeur du conteneur.
 - **Disposition manuelle** : le positionnement des objets graphiques est absolu, mais leur taille est proportionnelle à la taille du conteneur
 - **Disposition automatique** : par utilisation des *Gestionnaires de disposition de Qt*



Gestion des dispositions (2/4)

Disposition absolue

Le positionnement et dimensionnement des objets graphiques est fixé par des constantes dans le constructeur du conteneur.

- ◆ Exemple – dans constructeur de la classe ConvertisseurTemp

```
bConvertir->setGeometry(116, 160, 93, 29);
```

- ◆ Avantages : simplicité
- ◆ Inconvénients
 - Redimensionnement souvent incorrect de la fenêtre graphique qui les contient,
 - Taille inadaptée de certains éléments par rapport au style de fenêtre choisi par l'utilisateur,
 - Contenus tronqués / cachés,
 - Maintenance du code difficile dans le cas où les tailles et repositionnement sont calculés par le programme.



Gestion des dispositions (3/4)

Disposition manuelle

Le positionnement des objets graphiques est fixé par des constantes, mais leurs tailles sont définies proportionnellement à la taille du conteneur qui les contient.

- ◆ Exemple – dans constructeur de la classe ConvertisseurTemp

```
bConvertir->setGeometry(116 + extraWidth, 160 + extraHeight,  
                         93, 29);  
  
labelIntitule->setGeometry(16, 20,  
                            430 + extraWidth, 30 + extraHeight);
```

- ◆ Inconvénients
 - Il faut préalablement calculer les proportions d'expansion de chaque objet.
- ◆ Remarque

Nous n'utiliserons pas cette méthode



Gestion des dispositions (4/4)

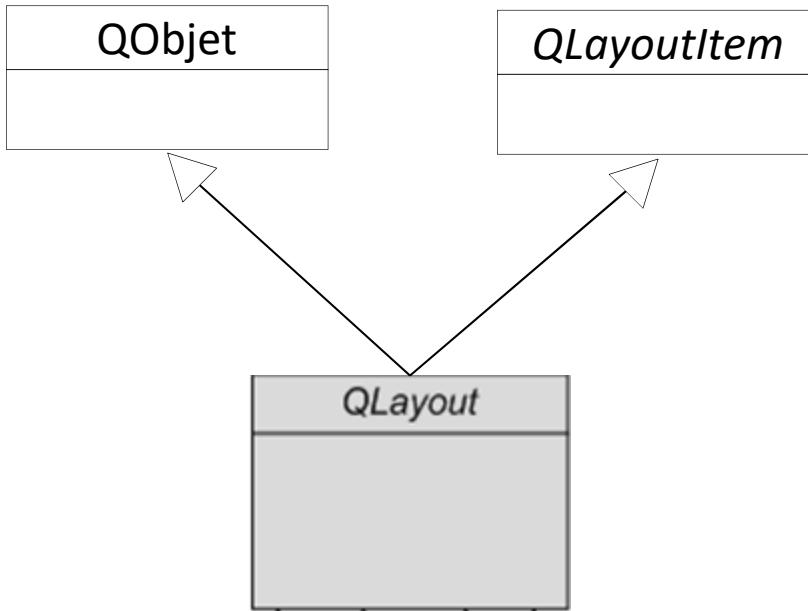
Disposition automatique : Gestionnaire des dispositions de Qt, *Layout*

- ◆ Le positionnement et redimensionnement des widgets contenus dans un conteneur (widget ou fenêtre) est **calculé automatiquement** par Qt, à partir de règles fournies lors de la création des widgets.
- ◆ Les règles de positionnement s'intéressent à l'alignement horizontal et vertical des widgets.
- ◆ Les règles de dimensionnement précisent les tailles mini-maxi d'un widgets, s'il peut / ou pas s'agrandir, dans quelle proportion par rapport à ses voisins,....
- ◆ La disposition automatique permet un agencement facile d'une interface et un bon usage de l'espace disponible
- ◆ La gestion des dispositions est gérée par la classe **QLayout**.



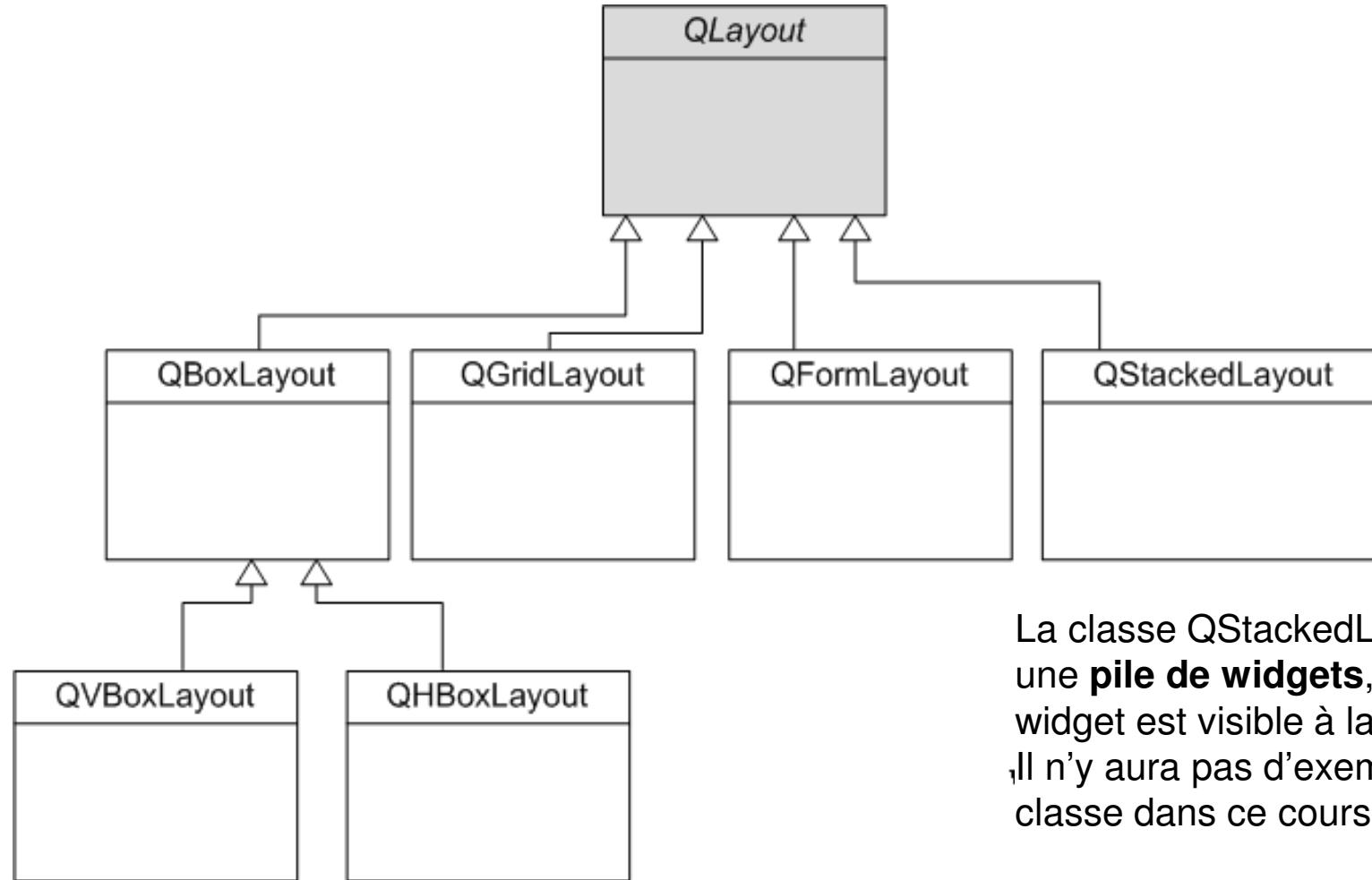
La classe QLayout (1/2)

- ◆ La classe **QLayout** fournit aux objets graphiques la **capacité à se positionner et redimensionner automatiquement**.
- ◆ Elle hérite des classes **QObject** et **QLayoutItem**.



La classe QLayout (2/2)

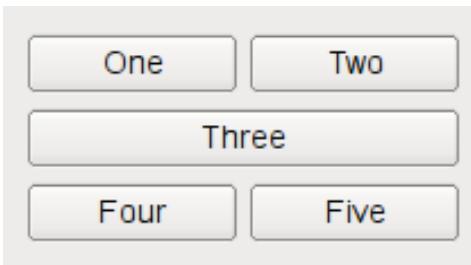
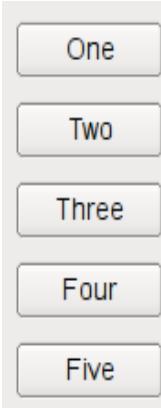
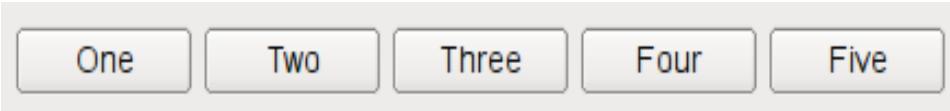
- ◆ Elle est la **classe mère** (*Base class*) de toutes les classes servant cet objectif.



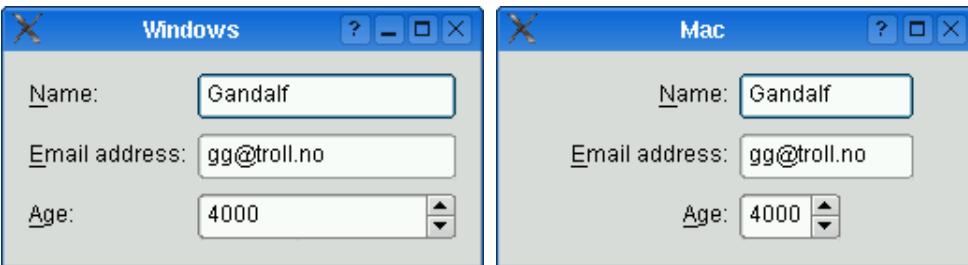
La classe **QStackedLayout** fournit une **pile de widgets**, où un seul widget est visible à la fois.
Il n'y aura pas d'exemple sur cette classe dans ce cours.

Les sous-classes de QLayout

- ◆ La classe **QHBoxLayout** permet la disposition **horizontale** de widgets
- ◆ La classe **QVBoxLayout** permet la disposition **verticale** de widgets
- ◆ La classe **QGridLayout** permet la disposition de widgets sous la forme d'une **grille**
- ◆ La classe **QFormLayout** permet la disposition de widgets sous la forme d'un **formulaire** (de saisie)



Une case peut s'étendre sur plusieurs lignes et/ou colonnes



Exemple de création de layouts (1/2)

mafenetre.cpp

```
1 #include "mafenetre.h"
2 #include <QLayout>
3 #include <QFormLayout>
4
5 MaFenetre::MaFenetre(QWidget *parent)
6     : QWidget(parent)
7 {
8     QHBoxLayout *hLayout;           // déclaration de variables
9     QVBoxLayout *vLayout;          // (ici, des pointeurs)
10    QGridLayout *gLayout;
11    QFormLayout *fLayout;
12
13    hLayout = new QHBoxLayout;      // instanciations classes
14    vLayout = new QVBoxLayout;       // = créations des objets layouts
15    gLayout = new QGridLayout;
16    fLayout = new QFormLayout;
17
18    setWindowTitle(tr("Exemple création de layouts"));
19 }
```

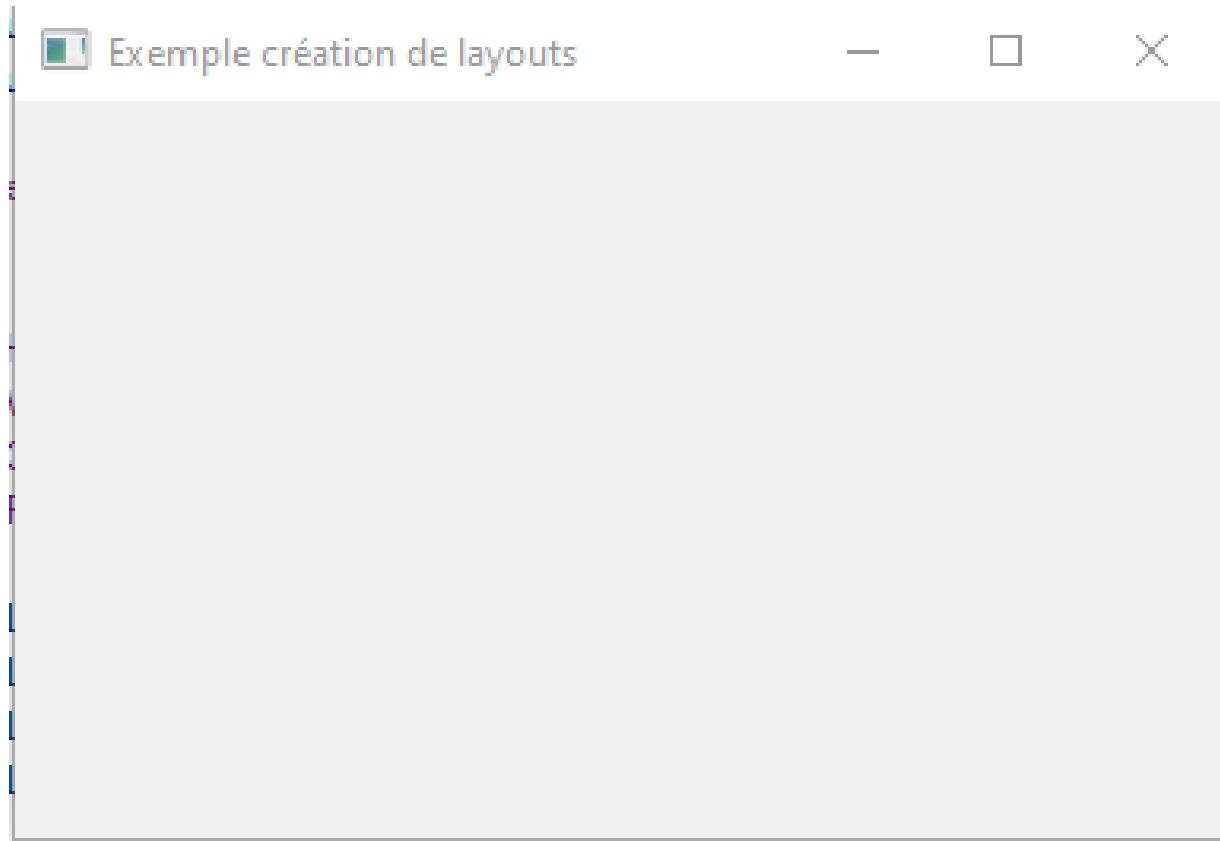


Exemple de création de layouts (2/2)

Résultat

Les layouts **ne sont pas visibles**.

Ils ne servent qu'à gérer la disposition des objets graphiques visibles.



Ajouter des objets à un layout

♦ Ajouter un *widget* à un layout

- Syntaxe

```
void QLayout::addWidget(QWidget *widget)
```

- Effet

Le widget ajouté change alors de parent et **devient enfant** du layout.

♦ Ajouter un *layout* à un layout

- Syntaxe

```
void QLayout::addLayout(QLayout *layout)
```

- Remarque

Si le parent est **QHBoxLayout** ou **QVBoxLayout**, le layout ajouté peut indiquer le **facteur d'expansion** qu'il aura à l'intérieur du layout ‘englobant’, dans le sens principal (horizontal/vertical) du layout parent. C'est la proportion d'expansion par rapport autres autres layouts ajoutés au layout ‘englobant’



Associer un layout à un widget

♦ Associer un *layout* à un *widget*

- Syntaxe

```
void QWidget::setLayout (QLayout *layout)
```

- Effet

Il y a changement de parent : Le widget devient alors parent de tous les widgets inclus dans le layout.

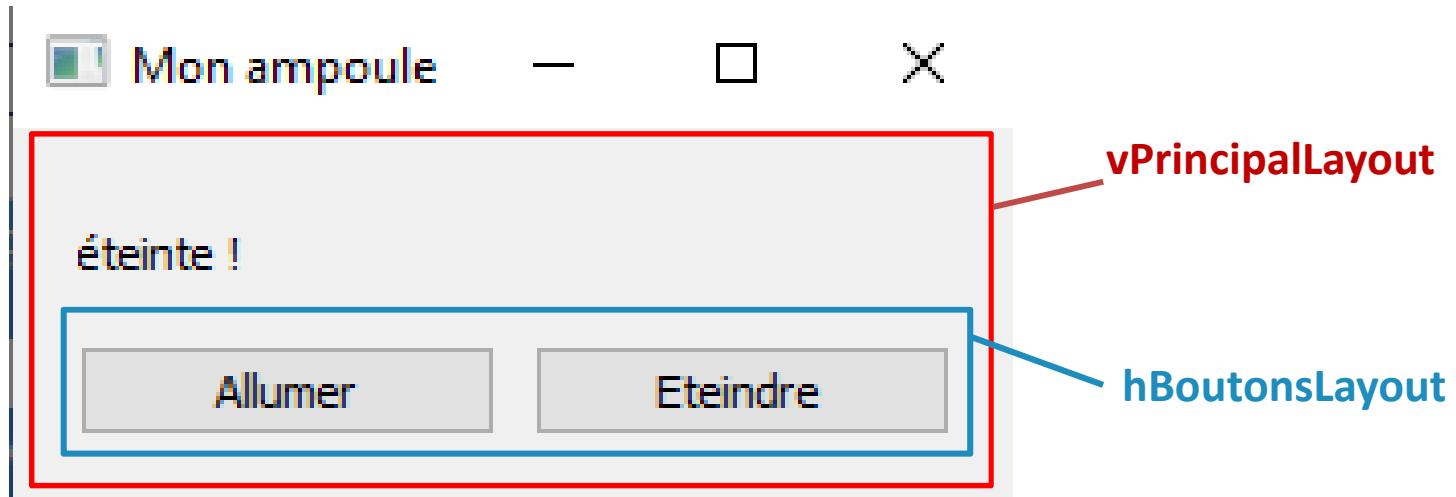
- Remarque

C'est la raison pour laquelle on trouve du code où les widgets ne sont associés à aucun parent lors de leur création, car la suite du programme les ajoutera à un layout qui sera lui-même associé à la fenêtre principale de l'application.



Exemples (1/11)

Exemple 1 – Mon ampoule



Lorsque la fenêtre est agrandie, les widgets sont repositionnés/redimensionnés par le layout principal.

Celui-ci s'appuie sur la **stratégie de taille et d'agencement** de chaque widget, définie grâce à la classe **QSizePolicy**.

Par exemple :

- Par défaut, le texte d'un label est aligné à gauche dans le label

Exemples (2/11)

Exemple 1 – Mon ampoule

monAmpoule.h

```
1 //... les include etc...
2 class MonAmpoule : public QWidget
3 {
4     Q_OBJECT
5
6     public:
7         MonAmpoule(QWidget *parent = nullptr);
8         ~MonAmpoule();
9
10    public slots:
11        void echoAllumer();
12        void echoEteindre();
13
14    private:
15        QLabel *labelEtat; // éteinte / allumée
16        QPushButton *bAllumer;
17        QPushButton *bEteindre;
18
19    };
20 #endif // MONAMPOULE_H
```



Exemples (3/11)

Exemple 1 – Mon ampoule

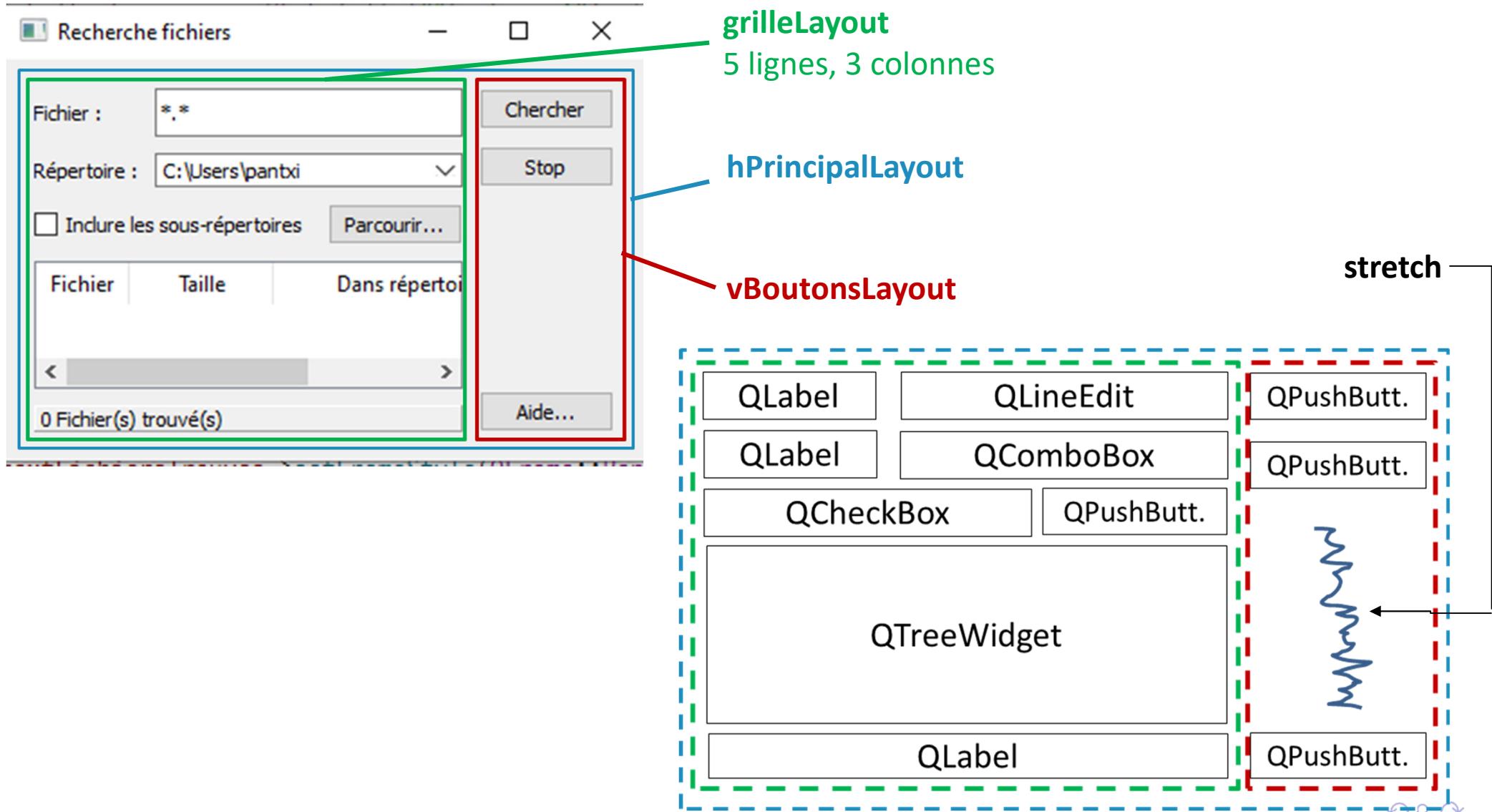
monampoule.cpp (extrait)

```
1  MonAmpoule::MonAmpoule(QWidget *parent)
2      : QWidget(parent)
3  {
4      // création et paramétrage des objets
5      labelEtat = new QLabel(tr("éteinte !"));
6      bAllumer = new QPushButton(tr("Allumer"));
7      bEteindre = new QPushButton(tr("Eteindre"));
8      setWindowTitle(tr("Mon ampoule"));
9
10     // Création des layouts
11     QVBoxLayout *vPrincipalLayout = new QVBoxLayout;
12     QHBoxLayout *hBoutonsLayout = new QHBoxLayout;
13
14     // Peuplement des layouts
15     hBoutonsLayout->addWidget(bAllumer);
16     hBoutonsLayout->addWidget(bEteindre);
17     vPrincipalLayout->addWidget(labelEtat);
18     vPrincipalLayout->addLayout(hBoutonsLayout);
19
20     // Associer le layout principal à la fenêtre principale
21     this->setLayout(vPrincipalLayout);
22     //...
23 }
```



Exemples (4/11)

Exemple 2 – findFiles revisité



Exemples (5/11)

Exemple 2 – findFiles revisité

mafenetre.h

```
1 //... les include etc...
2 class MaFenetre : public QWidget
3 {
4     // Q_OBJECT - Constructeur et destructeur...
5
6 private:
7     QLabel *labelFichier;
8     QLabel *labelRepertoire;
9     QCheckBox *caseSousRepertoires;
10    QLineEdit *textFichier;
11    QComboBox *lDeroulRepertoires;
12    QLabel *textFichiersTrouves;
13    QPushButton *bChercher;
14    QPushButton *bParcourir;
15    QPushButton *bStopperChercher;
16    QPushButton *bAide;
17    QTableWidget *tableFichiers;
18    QDir currentDir;
19 };
20 #endif // MAFENETRE_H
```



Exemples (6/11)

Exemple 2 – finfFiles revisité

mafenetre.cpp (1/6)

```
1 MaFenetre::MaFenetre(QWidget *parent)
2     : QWidget(parent)
3 {
4     // création des widgets
5     setWindowTitle(tr("Recherche fichiers"));
6     labelFichier = new QLabel (tr("&Fichier :"), this);
7     textFichier = new QLineEdit(tr("*.*"), this);
8     labelRepertoire = new QLabel(tr("&Répertoire :"), this);
9     lDeroulRepertoires =
10         createComboBox(QDir::toNativeSeparators(QDir::homePath()));
11     caseSousRepertoires =
12         new QCheckBox(tr("Inclure les sous-répertoires"), this);
13     bParcourir = new QPushButton(tr("&Parcourir..."), this);
14     textFichiersTrouves = new QLabel(tr("0 Fichier(s) trouvé(s)"), this);
15     createFilesTable();
16     bChercher = new QPushButton(tr("&Chercher"), this);
17     bStopperChercher = new QPushButton(tr("&Stop"), this);
18     bAide = new QPushButton(tr("&Aide..."), this);
19     .../...
```



Exemples (7/11)

Exemple 2 – finfFiles revisité

mafenetre.cpp (2/6)

```

20 .../...
21     // Paramétrage des widgets
22     // widgets copains- buddies, pour les Raccourcis clavier
23     labelFichier->setBuddy(textFichier);    // lier label et zone de texte
24     labelReertoire->setBuddy(lDeroulReertoires);
25     // effet sur le dernier label
26     textFichiersTrouves-
27         setFrameStyle(QFrame::Panel | QFrame::Raised);

```

Raccourcis clavier et widget compagnon

- ◆ Ligne 6 : Le caractère & indique que LabelFichier a un raccourci clavier accessible via Alt+F:
`labelFichier = new QLabel (tr("&Fichier :"), this);`
- ◆ Ligne 23 : la zone de texte est définie comme **compagnon-buddy** de l'étiquette : c'est la zone de texte qui recevra le **focus** lorsque l'utilisateur appuiera sur le raccourci-clavier Alt+F



Exemples (8/11)

Exemple 2 – finfFiles revisité

mafenetre.cpp (3/6)

```

.../...
28 // Politique de positionnement, taille et d'expansion
29     labelFichier->setSizePolicy(QSizePolicy::Maximum,
30                                 QSizePolicy::Minimum);
31     textFichier->setSizePolicy(QSizePolicy::Minimum,
32                                 QSizePolicy::MinimumExpanding);
33     bParcourir->setSizePolicy(QSizePolicy::Maximum,
34                                 QSizePolicy::Maximum);
35     bStopperChercher->setStyleSheet("text-align:center");

```

Les cases d'une grille n'évoluent pas de manière indépendante...

- ◆ LabelFichier et LabelRépertoire s'expandent HORIZONTALEMENT ensemble car
 - ils sont sur la même colonne
 - si on bloque l'expansion horizontale de l'un, on la bloque pour l'autre
- ◆ LabelRépertoire et IDeroulRepertoires s'expandent VERTICIALEMENT ensemble car
 - ils sont sur la même ligne
 - si on bloque l'expansion verticale de l'un, on la bloque pour l'autre



Exemples (9/11)

Exemple 2 – finfFiles revisité

mafenetre.cpp (4 / 6)

```
.../...
36 // Gestion des dispositions
37 QBoxLayout* layoutHPrincipal =new QHBoxLayout(this);
38 QGridLayout *layoutGrilleGauche = new QGridLayout;
39 QVBoxLayout *layoutVDroite = new QVBoxLayout;
40 // Peuplement des layouts
41 layoutGrilleGauche->addWidget(labelFichier, 0, 0);
42 layoutGrilleGauche->addWidget(textFichier, 0, 1, 1, 2);
43 layoutGrilleGauche->addWidget(labelRepertoire, 1, 0);
44 layoutGrilleGauche->addWidget(lDeroulRepertoires, 1, 1, 1, 2);
45 layoutGrilleGauche->addWidget(caseSousRepertoires, 2, 0, 1, 2 );
46 layoutGrilleGauche->addWidget(bParcourir, 2, 2);
47 layoutGrilleGauche->addWidget(tableFichiers, 3, 0, 1, 3);
48 layoutGrilleGauche->addWidget(textFichiersTrouves, 4, 0, 1, 3);
49 layoutVDroite->addWidget(bChercher);
50 layoutVDroite->addWidget(bStopperChercher);
51 layoutVDroite->addStretch();
52 layoutVDroite->addWidget(bAide);

53
54 layoutHPrincipal->addLayout(layoutGrilleGauche, 1);
55 layoutHPrincipal->addLayout(layoutVDroite, 0);
56 setLayout(layoutHPrincipal);
```

Exemples (10/11)

Exemple 2 – finFiles revisité

mafenetre.cpp (5/6)

```
.../...
57 // Organisation de la navigation
58 // A faire APRÈS le peuplement des layouts
59 // on commence par le premier
60 QWidget::setTabOrder(textFichier, lDeroulRepertoires);
61 QWidget::setTabOrder(lDeroulRepertoires, caseSousRepertoires);
62 QWidget::setTabOrder(caseSousRepertoires, bParcourir);
63 QWidget::setTabOrder(bParcourir, bChercher);
64
65 // focus sur ce widget
66 textFichier->setFocus();
67
68 // bouton par défaut quand l'utilisateur presse 'Entrée'
69 bChercher->setDefault(true);
70
71 // Connexions : cf. diapo suivante
} // fin constructeur
```

Exemples (11/11)

Exemple 2 – finfFiles revisité

mafenetre.cpp (6 / 6)

```
.../...
1 // connexions
2 connect(textFichier, SIGNAL(returnPressed()), 
3         this, SLOT/animateFindClick()));
4 connect(lDeroulRepertoires->lineEdit(), SIGNAL(returnPressed()), 
5         this, SLOT/animateFindClick());
6 connect(bChercher, SIGNAL(clicked()), this, SLOT/find()));
7 connect(bParcourir, SIGNAL(clicked()), this, SLOT/browse()));
8 connect(bStopperChercher, SIGNAL(clicked()), 
9         this, SLOT(stop()));
10 connect(bAide, SIGNAL(clicked()), this, SLOT(help()));
```



Synthèse des propriétés des layouts

- ◆ Les gestionnaires de disposition proposent des valeurs par défaut raisonnables pour chaque type de widget
- ◆ Ils tiennent compte de la taille requise de chacun d'eux, qui dépend de la police, du style et du contenu du widget
- ◆ Ils respectent les dimensions minimales et maximales des widgets, et ajustent automatiquement la disposition en réponse à des changements de police ou de contenu et à un redimensionnement de la fenêtre
- ◆ Si l'on ajoute ou supprime un widget dans une disposition, celle-ci s'adapte automatiquement à la nouvelle situation
- ◆ Il en est de même si l'on cache ou montre un widget (= invocation des méthodes `hide()`/`show()`)
- ◆ Pour ce faire, les layout s'appuient sur la **stratégie de la taille** de chaque widget, gérée par la classe **QSizePolicy**.



La classe QSizePolicy

- ◆ La **stratégie de taille** d'un objet définit sa capacité / propension à se redimensionner en largeur / hauteur et affecte la manière dont l'objet est traité par le gestionnaire de dispositions.
- ◆ Pour chaque widget, la méthode `QSizePolicy QWidget::sizePolicy()` retourne un objet **QSizePolicy** qui décrit, à l'aide de 2 valeurs indépendantes, sa stratégie de redimensionnement horizontal et vertical
- ◆ Les valeurs ont été regroupées dans le type énuméré **Policy** décrit dans la diapositive suivante.
- ◆ Cet objet contient 2 valeurs indépendantes, Il est possible de modifier la propriété `QWidget::sizePolicy` de chaque objet.



Type enum Policy (1/2)

- ◆ La **stratégie de taille** d'un objet définit sa capacité / propension à se redimensionner en largeur / hauteur et affecte la manière dont l'objet est traité par le gestionnaire de dispositions.
- ◆ Pour chaque widget, la méthode `QSizePolicy QWidget::sizePolicy()` retourne un objet **QSizePolicy** qui décrit, à l'aide de 2 composantes indépendantes, sa stratégie de redimensionnement horizontal et vertical
- ◆ Les valeurs de ces composantes ont été regroupées dans le type énuméré **Policy** décrit dans la diapositive suivante.
- ◆ Il est possible de modifier la propriété **QWidget::sizePolicy** de chaque objet.



Type enum Policy (2/2)

Valeurs d'énumération du type <code>enum QSizePolicy::Policy</code>	Description
<code>QSizePolicy::Fixed</code>	Le widget ne peut être rétréci ou étiré, il conserve toujours sa taille requise.
<code>QSizePolicy::Minimum</code>	La taille requise du widget correspond à sa taille minimale. Le widget ne peut pas être rétréci en dessous de sa taille requise, mais il peut s'étirer si nécessaire pour combler un espace disponible
<code>QSizePolicy::Maximum</code>	La taille requise correspond à la taille maximale. Le widget peut être rétréci jusqu'à sa taille mainimum.
<code>QSizePolicy::Preferred</code>	La taille requise du widget est sa taille favorite. Cependant, le widget peut toujours s'étirer ou se rétrécir si nécessaire.
<code>QSizePolicy::Expanding</code>	Le widget peut être rétréci, mais il préfère être élargi.
<code>QSizePolicy::MinimumExpanding</code>	La taille requise est minimale, et suffisante. Le widget peut utiliser de l'espace supplémentaire, il devrait donc obtenir autant d'espace que possible.
<code>QSizePolicy::Ignored</code>	La taille requise est ignorée. Le widget prend toute la place possible.



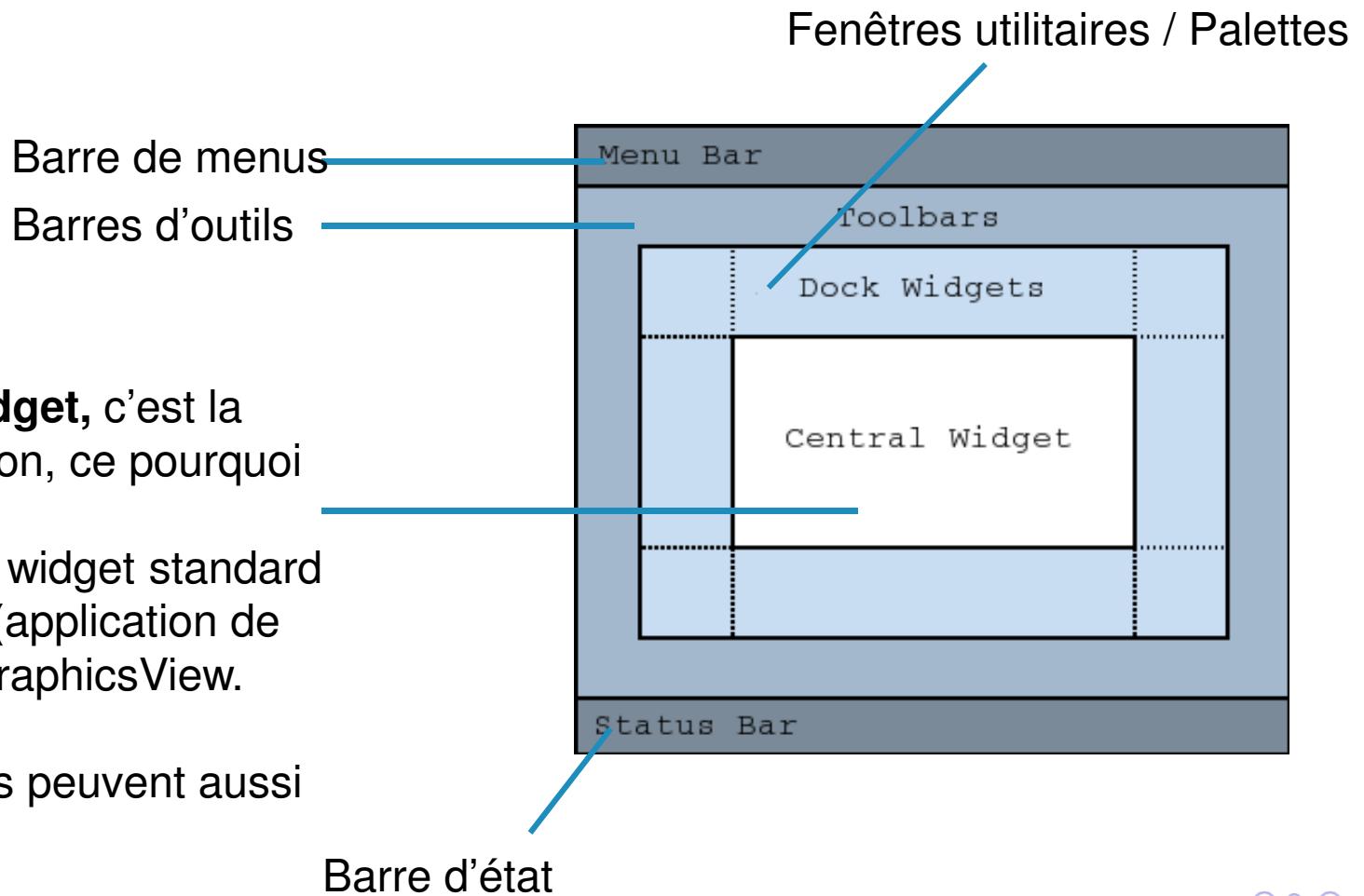
Classe QMainWindow

La classe **QMainWindow** propose un squelette (de fenêtre) adapté à la création de fenêtres personnalisées

Zone client : Central Widget, c'est la raison d'être de l'application, ce pourquoi elle a été créé.

Ce sera généralement un widget standard de Qt, comme QTextEdit (application de type NotePad), ou un QGraphicsView.

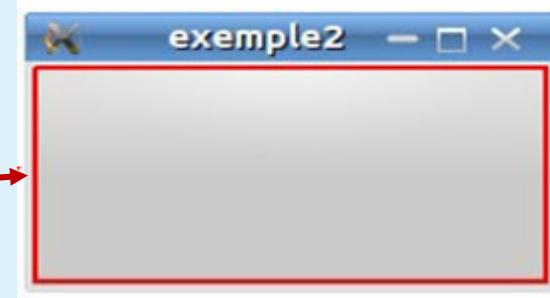
Les widgets personnalisés peuvent aussi être utilisés.



Classe QMainWindow – exemple 2

On définit le widget central avec **setCentralWidget()**

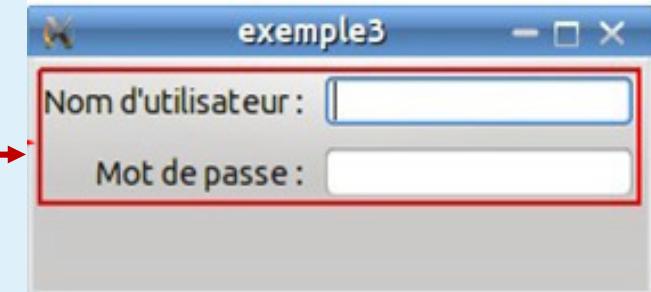
```
1 #include <QApplication>
2 #include <QWidget>
3 class MyMainWindow : public QMainWindow {
4     public:
5         MyMainWindow() {
6             QWidget *centralWidget = new QWidget;
7             setCentralWidget(centralWidget); —————→
8         }
9     };
10
11    int main(int argc, char *argv[]) {
12        QApplication app(argc, argv); MyMainWindow myMainWindow;
13        myMainWindow.show();
14        return app.exec();
15    }
```



Classe QMainWindow – exemple 3

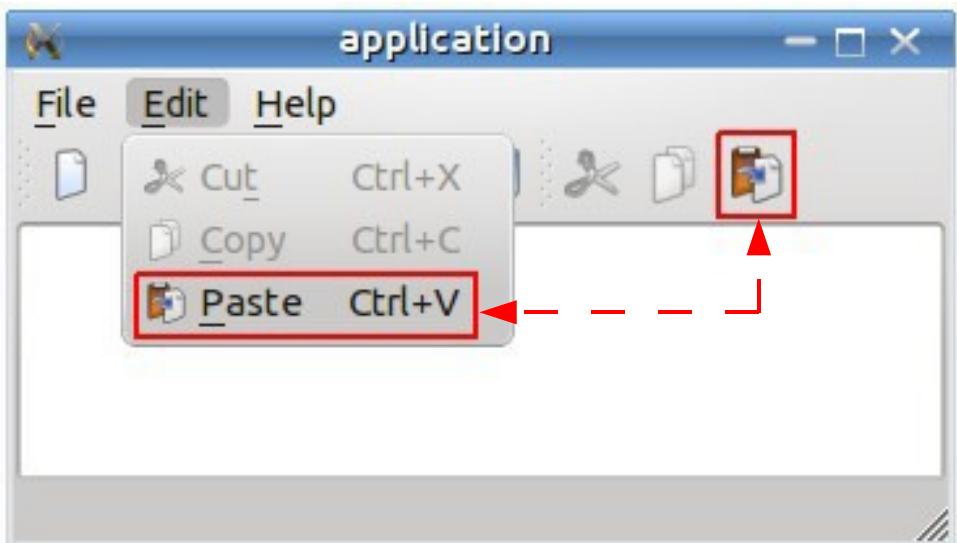
On peut alors ajouter d'autres widgets dans le widget principal :

```
1 #include <QApplication>
2 #include <QWidget>
3 class MyMainWindow : public QMainWindow {
4     public:
5         MyMainWindow() {
6             QWidget *centralWidget = new QWidget;
7
7             QLineEdit *login = new QLineEdit;
8             QLineEdit *password = new QLineEdit;
9
10            QFormLayout *formLayout = new QFormLayout;
11            formLayout->addRow("Nom d'utilisateur : ", login);
12            formLayout->addRow("Mot de passe : ", password);
13            centralWidget->setLayout(formLayout);
14            setCentralWidget(centralWidget);
15        }
16    };
17 };
```



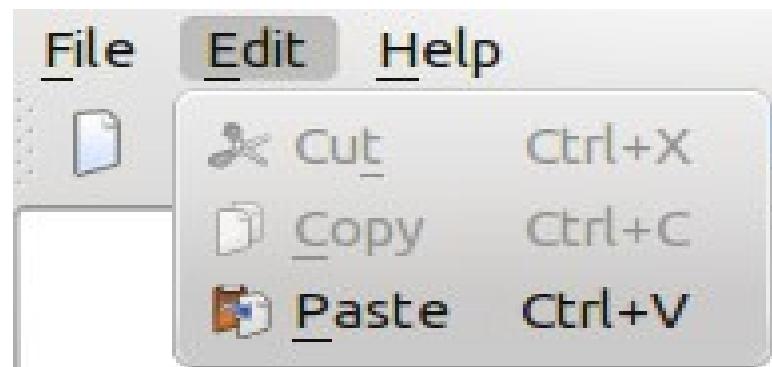
Classe QAction

- ◆ La classe **QAction** fournit une interface abstraite pour décrire une action (= une commande) à insérer dans une widget
- ◆ 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 ajoutées aux menus et barres d'outils, et seront automatiquement synchronisées.



Classe QMenu

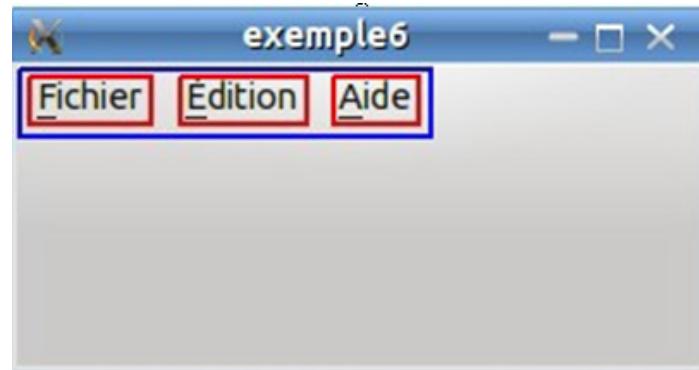
- ◆ La classe **QMenu** fournit un widget pour une utilisation dans les barres de menus et les menus contextuels. Un menu contextuel est un menu qui s'affiche lorsqu'on fait un clic droit sur un widget.
- ◆ Un widget menu est un menu de sélection. Il peut être soit un menu déroulant dans une barre de 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.



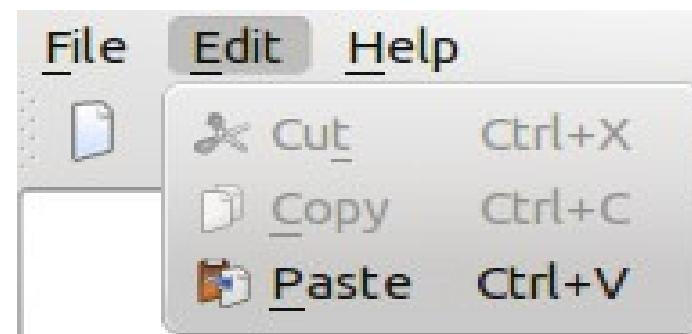
Peuplement d'une barre de menus

A voir directement sur QtDesigner

- ◆ Ajout de nouveaux éléments de menus à la barre de menus de la fenêtre principale :
 - Appeler **menuBar()**, qui retourne la QMenuBar de la fenêtre
 - Puis ajouter un menu avec **QMenuBar::addMenu()**



- ◆ Ajout d'une instance de **QAction** à un élément de menu
 - Crédation d'une instance de QAction, puis ajout de l'instance à un élément de menu, avec méthode **addAction()**
 - Ajout à un élément de menu d'une instance créée à la volée



Merci pour
votre attention

