

IHM avec Qt

Fenêtres, labels, interactions, couleurs,
disposition

Dakkar Borhen-eddine
BTS SN



Table des matières

1 Table des matières

2 La classe QMainWindow

3 La classe QLabel

- QLabel
- Exemple
- Méthodes de la classe QLabel
- Exemple

4 Interactions

- Programmation événementielle
- Signaux et slots (Signals and slots)

5 Les boutons

- QPushButton
- Exemple signal-slot

6 Les boîtes de messages

7 Gestion de la mise en page

- QHBoxLayout et QVBoxLayout
- QVBoxLayout
- QGridLayout



La classe QMainWindow

- **QMainWindow** fournit un cadre pour la construction de l'interface utilisateur d'une application.
- **QMainWindow** a sa propre disposition à laquelle vous pouvez ajouter QToolBars, QDockWidgets, QMenuBar et QStatusBar.

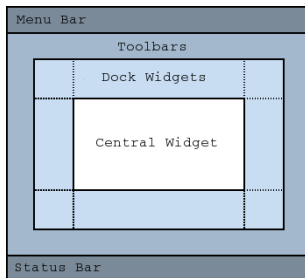


Figure: Cadre de la fenêtre principale QMainWindow



QLabel

La classe `QLabel` permet d'afficher un texte ou une image.
Afin d'utiliser `QLabel`, on inclut cette dernière dans le fichier entête:

```
#include <QLabel>
```

Utilisation

- `QLabel *label = new QLabel(this);`
- `label->setFrameStyle(QFrame::Panel | QFrame::Sunken);`
- `label->setText("first line");`
- `label->setAlignment(Qt::AlignBottom | Qt::AlignRight);`



QLabel

- **setFrameStyle:** définit le style du cadre.
Les formes du cadre sont données dans `QFrame :: Shape` et les styles d'ombre dans `QFrame :: Shadow`.
- **setText:** définit le texte à afficher
- **setAlignment:** Cette propriété aligne le contenu de l'étiquette (Label). il existe différentes possibilités d'alignement :
 - `Qt.AlignLeft`: Alignement à gauche
 - `Qt.AlignRight`: Alignement à droite
 - `Qt.AlignJustify`: Justifie le texte sur tout l'espace disponible
 - `Qt.AlignTop`: Alignement tout en haut
 - `Qt.AlignBottom`: Alignement tout en bas

Pour plus d'informations consulter: <https://doc.qt.io/qt-5/qlabel.html>



Exemple

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QLabel>
namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
    /** La classe QLabel
    /** Elle est utilisée pour afficher du texte
    /** ou une image/
    QLabel *label = new QLabel(this);

private:
    Ui::MainWindow *ui;
```

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    /** Titre de la fenêtre **/
    this->setWindowTitle("Ma première fenêtre");
    /** On appelle notre objet QLabel **/
    /** On spécifie le style de la fenêtre **/
    label->setFrameStyle(QFrame::Panel | QFrame::Sunken);
    /** Le text à afficher
    this->label->setText("Voici ma première fenêtre Qt");
    /** Centrer le texte par rapport à la widget contenante **/
    this->label->setAlignment(Qt::AlignCenter);
    /** Mettre le label dans le Widget central **/
    this->setCentralWidget(this->label);
}
MainWindow::~MainWindow()
{
    delete ui;
}
```

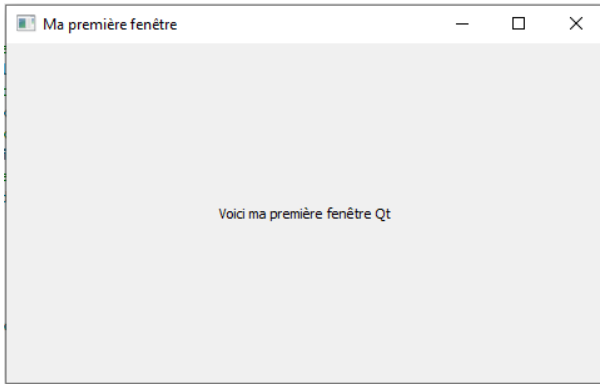
main.cpp

```
#include "mainwindow.h"
#include <QApplication>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```



Résultat

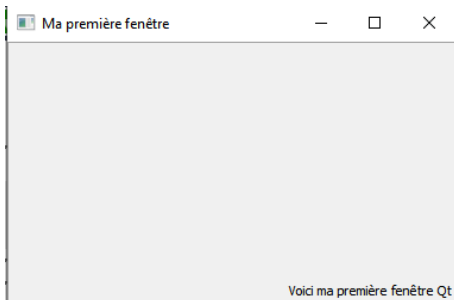


Méthodes de QLabel

Alignement

On peut cumuler des paramètres d'alignement non contradictoires par exemple alignement du texte en bas à droite en changeant la ligne suivante:

```
this->label->setAlignment (Qt::AlignBottom | Qt::AlignRight);
```



Méthodes de QLabel

Images

- La classe `QPixmap` est conçue pour afficher des images.
- Un `QPixmap` peut être utilisé avec un `QLabel` pour afficher une image.

Utilisation

Nous créons un objet `QPixmap`:

```
QPixmap *pixmap_img = new QPixmap("Nom_fichier_image");
```

Ensuite nous appellerons l'objet avec un `QLabel`:

```
this->label->setPixmap(*pixmap_img);
```

La méthode `setScaledContents` lorsqu'elle est activée redimensionnera le pixmap pour remplir l'espace disponible.

```
this->label->setScaledContents(true);
```



Exemple QPixmap

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QLabel>
namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();
    //-- La classe QLabel
    //-- Elle est utilisée pour afficher du texte
    //-- ou une image/
    QLabel *label = new QLabel(this);
    //-- Création d'un objet QPixmap
    QPixmap *pixmap_img = new QPixmap("QT_image.png");
private:
    Ui::MainWindow *ui;
};
#endif // MAINWINDOW_H
```



mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    //-- Titre de la fenetre --//
    this->setWindowTitle("Ma première fenêtre");
    //-- On appelle notre objet QLabel --//
    //-- On spécifie le style de la fenêtre --//
    label->setFrameStyle(QFrame::Panel | QFrame::Sunken);

    //-- Appel de la méthode set QPixmap avec comme paramètre
    //-- l'objet QPixmap
    this->label->setPixmap(*pixmap_img);
    this->label->setScaledContents(true);
    this->label->setAlignment(Qt::AlignCenter);
    //-- Mettre le label dans le Widget central --//
    this->setCentralWidget(this->label);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```



Programmation événementielle

- Pour rendre la programmation GUI intuitive, nous voulons que des objets de toute nature puissent communiquer entre eux. Par exemple, si un utilisateur clique sur un bouton Fermer, nous souhaitons que la fonction `close()` de la fenêtre soit appelée.
- Dans la programmation séquentielle ce type d'action n'est pas pris en compte.
- Nous faisons appel à la programmation événementielle. Elle est l'inverse de la programmation séquentielle dans laquelle les instructions sont exécutées les unes après les autres.



suite

- La programmation événementielle est définie par des réactions aux différents événements qui peuvent se produire.
- Le mouvement de la souris ou le changement des données dans un champ peut être considéré comme un événement.
- Il existe deux étapes dans la programmation événementielle:
 - La première consiste la détection des événements.
 - La seconde consiste leur gestion.



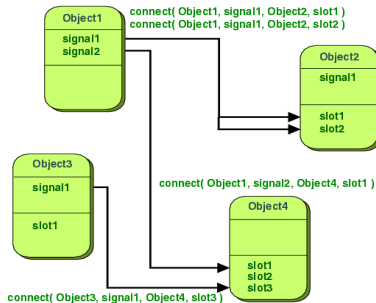
Signaux et slots (Signals and slots)

Un signal: est émis lorsqu'un événement particulier se produit. Les widgets de Qt ont de nombreux signaux prédéfinis, mais nous pouvons toujours créer nos propres signaux.

Un slot: est une fonction qui est appelée en réponse à un signal particulier. Les widgets de Qt ont de nombreux slots prédéfinis, mais il est courant d'ajouter de nouveaux slots.



Connexion Signaux et slots



- Pour connecter un signal à un slot, nous utilisons `QObject :: connect()`.

```
connect (objet1, SIGNAL(signal1()), objet2, SLOT(slot1()))  
connect (objet1, SIGNAL(signal1()), objet2, SLOT(slot2()))
```



La classe QPushButton

Le bouton poussoir, ou bouton de commande, est le widget le plus utilisé dans toute interface graphique.

Un bouton a généralement une forme rectangulaire et affiche une étiquette de texte décrivant son action. Une touche de raccourci peut être spécifiée pour chaque bouton.

```
QPushButton *button = new QPushButton("&Ouvrir", this);
```

Un bouton émet le signal `clicked()` lorsqu'il est activé par la souris.



Exemple

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QPushButton>
QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    QPushButton *button;
public slots:
    void clickedSlot();
private:
    Ui::MainWindow *ui;

};
#endif // MAINWINDOW_H
```



mainwindow.cpp

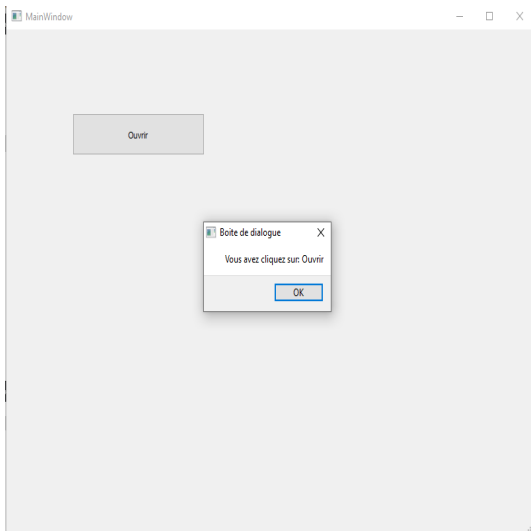
```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QMessageBox>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    this->button = new QPushButton("&Ouvrir", this);
    this->button->setGeometry(100, 100, 200, 50);
    //Connexion du signal "clicked()" et du slot clickedSlot()
    connect(button, SIGNAL(clicked()),this, SLOT(clickedSlot()));
}

void MainWindow:: clickedSlot()
{
    QMessageBox msgBox;
    msgBox.setWindowTitle("Boite de dialogue");
    msgBox.setText("You Clicked " + ((QPushButton*)sender())->text());
    msgBox.exec();
}

MainWindow::~MainWindow()
{
    delete ui;
}
```









QMessageBox

Nous avons vu l'utilisation d'une boîte de messages dans l'exemple précédent. Elle est utilisée pour alerter l'utilisateur d'une situation.

```
QMessageBox msgBox;  
msgBox.setText("The document has been modified.");  
msgBox.exec();
```

QMessageBox prend en charge quatre niveaux d'alerte de message prédéfinis, ou types de message, qui ne diffèrent vraiment que par l'icône qu'ils affichent chacun.

	Question	For asking a question during normal operations.
	Information	For reporting information about normal operations.
	Warning	For reporting non-critical errors.
	Critical	For reporting critical errors.



Gestion de la mise en page (Layout)

Qt comprend un ensemble de classes de gestion de mise en page qui sont utilisées pour décrire la façon dont les widgets sont disposés dans l'interface graphique.

Ces classes positionnent, redimensionnent et garantissant que les widgets sont disposés de manière cohérente pour faciliter l'utilisation de l'application. Toutes les sous-classes QWidget peuvent utiliser des

dispositions pour gérer leurs enfants. La fonction `QWidget :: setLayout()` applique une disposition à un widget.

La meilleure façon de donner à vos widgets une bonne disposition consiste à utiliser les gestionnaires de mise en page intégrés: `QHBoxLayout`, `QVBoxLayout`, `QGridLayout` et `QFormLayout`. Ces classes héritent de `QLayout`, qui dérive à son tour de `QObject`.



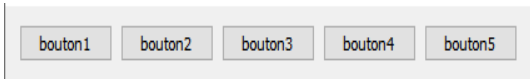
QHBoxLayout et QVBoxLayout

Un **QHBoxLayout** dispose les widgets sur une ligne horizontale, de gauche à droite.

```
QPushButton *button1 = new QPushButton("bouton1");  
QPushButton *button2 = new QPushButton("bouton2");  
QPushButton *button3 = new QPushButton("bouton3");  
QPushButton *button4 = new QPushButton("bouton4");  
QPushButton *button5 = new QPushButton("bouton5");
```

```
QHBoxLayout *layout = new QHBoxLayout;  
layout->addWidget(button1);  
layout->addWidget(button2);  
layout->addWidget(button3);  
layout->addWidget(button4);  
layout->addWidget(button5);
```

```
window->setLayout(layout);
```



QVBoxLayout

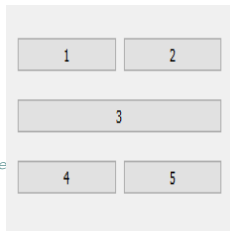
Un `QVBoxLayout` dispose les widgets dans une colonne verticale, de haut en bas.



QGridLayout

Un `QGridLayout` dispose des widgets dans une grille à deux dimensions.
Les widgets peuvent occuper plusieurs cellules.

```
QWidget *window = new QWidget;  
QPushButton *button1 = new QPushButton("1");  
QPushButton *button2 = new QPushButton("2");  
QPushButton *button3 = new QPushButton("3");  
QPushButton *button4 = new QPushButton("4");  
QPushButton *button5 = new QPushButton("5");  
  
QGridLayout *layout = new QGridLayout;  
layout->addWidget(button1, 0, 0);  
layout->addWidget(button2, 0, 1);  
layout->addWidget(button3, 1, 0, 1, 2); // prend 2 places  
layout->addWidget(button4, 2, 0);  
layout->addWidget(button5, 2, 1);  
  
window->setLayout(layout);  
window->show();
```



QFormLayout

Un **QFormLayout** présente les widgets dans un style de champ d'étiquette descriptif à 2 colonnes.

```
QWidget *window = new QWidget;  
QPushButton *button1 = new QPushButton("1");  
QLabel *Label1 = new QLabel(this);  
Label1->setText("Bouton1");  
QPushButton *button2 = new QPushButton("2");  
QLabel *Label2 = new QLabel(this);  
Label2->setText("Bouton2");  
QPushButton *button3 = new QPushButton("3");  
QLabel *Label3 = new QLabel(this);  
Label3->setText("Bouton3");  
  
QFormLayout *layout = new QFormLayout;  
layout->addRow(button1, Label1);  
layout->addRow(button2, Label2);  
layout->addRow(button3, Label3);  
  
window->setLayout(layout);  
window->show();
```

