# 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.

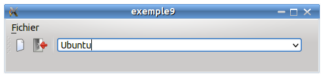
Pour ajouter une barre d'outils, on doit tout d'abord appeler la méthode `addToolBar()` de `QMainWindow`.

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`






```
1  QToolBar *fileToolBar = addToolBar("Fichier");
2  //fileToolBar->setMovable(false);
3  //fileToolBar->setFloatable(false);
4
5  fileToolBar->addAction(actionNouveau);
6  fileToolBar->addAction(actionQuit);
7  fileToolBar->addSeparator();
8
9  QFontComboBox *fontComboBox = new QFontComboBox
       ;
10 fileToolBar->addWidget(fontComboBox);
```

# La classe `QDockWidget`



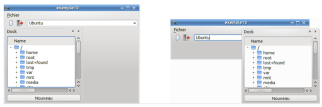
La classe `QDockWidget` fournit un widget qui peut être « ancré » dans une `QMainWindow` ou « flotter » comme une fenêtre de haut niveau sur le bureau.

`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`.

De nombreuses applications connues les utilisent : Qt Designer, OpenOffice, Gimp, Photoshop, Code : :Blocks...

# La classe `QDockWidget`





```
1  QDockWidget *dockWidget = new QDockWidget("Dock", this);
2  addDockWidget(Qt : :LeftDockWidgetArea, dockWidget);
3
4  QWidget *dockContenu = new QWidget;
5  dockWidget->setWidget(dockContenu);
6
7  QVBoxLayout *dockLayout = new QVBoxLayout;
8  QFileSystemModel *model = new QFileSystemModel;
9
10 model->setRootPath(QDir : :currentPath());
11 QTreeView *vueArbre = new QTreeView;
12 vueArbre->setModel(model);
13 dockLayout->addWidget(vueArbre);
14 //...
15 dockContenu->setLayout(dockLayout);
```

## La classe `QStatusBar`



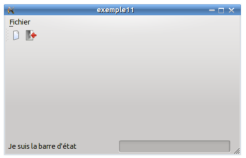
La classe `QStatusBar` fournit une barre horizontale appropriée pour la présentation des informations d'état. `QStatusBar` permet d'afficher différents types d'indicateurs. Une barre d'état peut afficher trois types de messages différents :

- 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 récupérée à l'aide de `QMainWindow::statusBar()` et remplacée à l'aide de `QMainWindow::setStatusBar()`.

## La classe `QStatusBar`






```
1 QStatusBar *barreEtat = statusBar();
2 QProgressBar *progression = new QProgressBar;
3
4 barreEtat->addPermanentWidget(progression);
5 barreEtat->showMessage(QString::fromUtf8("Je suis la
      barre d'état"),
6 2000);
```

## Plan intermédiaire





## Les fichiers ressources `.qrc`



On peut utiliser un fichier ressource de Qt (d'extension `.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` génère un fichier C++ à partir des données spécifiées dans le fichier `.qrc`.
Exemple :

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

```
1 <!DOCTYPE RCC>
2 <RCC version="1.0">
3 <qresource>
4 <file>help.png</file>
5 </qresource>
6 </RCC>
```

# Plan intermédiaire





# 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+, Ms 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 :


```
1  QPushButton *pMonBouton = new QPushButton("Quitter");
2  //pMonBouton->setGeometry(0,0,200,100);
3  pMonBouton->resize(200, 100);
4  pMonBouton->move(50, 50);
5
6  pMonBouton->setFont(QFont("Arial", 18, QFont::Bold));
7
8  pMonBouton->setStyleSheet("background-color : red; color : white;");
9
10 pMonBouton->setStyle(QStyleFactory::create("Motif"));
11
12 pMonBouton->show();
```

# Exemple

Modifier l'aspect visuel et ergonomique d'un widget :


```
1  QLabel *pMonLabel = new QLabel("Hello world !");
2
3  pMonLabel->setFrameStyle(QFrame::Panel | QFrame::Sunken);
4  pMonLabel->setMargin(10);
5
6  pMonLabel->setFont(QFont("Verdana", 14, QFont::DemiBold, QFont::StyleItalic));
7  pMonLabel->setAlignment(Qt::AlignHCenter);
8
9  QPalette palette;
10 palette.setColor(QPalette::WindowText, Qt::blue);
11
12 pMonLabel->setAutoFillBackground(true);
13 pMonLabel->setPalette(palette);
14 pMonLabel->show();
```



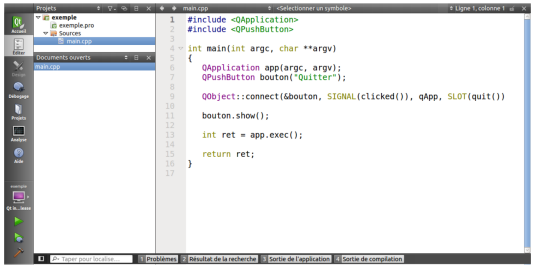
# Plan intermédiaire





# QtCreator

`QtCreator` est installé avec votre environnement Qt. Il permet : de créer un projet, de l'associer à un gestionnaire de version (comme `git`), de gérer les ressources et langues de votre application, de définir les cible de compilation, d'avoir une aide contextuelle (sur un mot-clé, par l'auto-indentation et le remplissage de votre source, une première analyse de syntaxe).



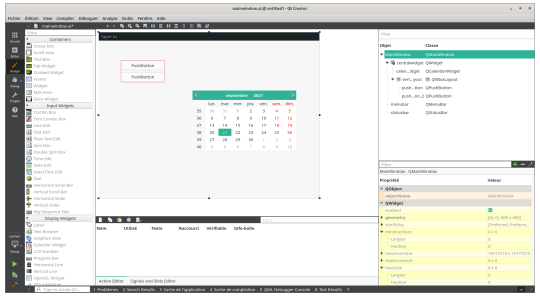



# Plan intermédiaire

## QtDesigner



`QtDesigner` permet de construire vos interfaces mais également de gérer leurs propriétés, les signaux/slots, les actions. Il crée un objte `ui`, utilisable depuis votre source en C++. Ce n'est pas un RAD[9] (pas de connexion à une base de données avec filtrage des données, pas de bibliothèque d'algorithmes).



[9] `https://fr.wikipedia.org/wiki/Outils_RAD`

## 4e partie





## Plan intermédiaire



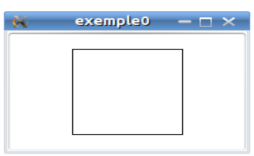


## Graphiques 2D

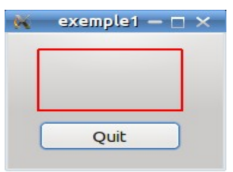


Il existe 2 approches pour dessiner en 2D avec Qt :

- un modèle fonctionnel basé sur `QPainter`
- un modèle objet basé sur le framework `Graphics View`




*Exemple utilisant l'architecture Graphics View*




*Exemple utilisant QPainter*

# Plan intermédiaire



# Introduction

Le framework `Graphics View` se décompose en 3 parties essentielles :

- la scène,
- la vue,
- les éléments graphiques

L'architecture `Graphics View` offre une approche basée sur le pattern modèle-vue. Plusieurs vues peuvent observer une scène unique constituée d'éléments de différentes formes géométriques.

# QGraphicsScene

La classe `QGraphicsScene` fournit la **scène** pour l'architecture `Graphics View`. La scène a les responsabilités suivantes :

- fournir une interface rapide pour gérer un grand nombre d'éléments,
- propager les événements à chaque élément,
- gérer les états des éléments (telles que la sélection et la gestion du focus)
- et fournir des fonctionnalités de rendu non transformée (principalement pour l'impression).

La classe `QGraphicsView` fournit la **vue** « widget » qui permet de visualiser le contenu d'une scène.

# QGraphicsItem

La classe `QGraphicsItem` est la classe de base pour les éléments graphiques dans une scène.

Elle fournit plusieurs éléments standard pour les formes typiques, telles que :

- des rectangles (`QGraphicsRectItem`),
- des ellipses (`QGraphicsEllipseItem`),
- des éléments de texte (`QGraphicsTextItem`).

Mais les fonctionnalités les plus puissantes seront disponibles lorsque on écrira un élément personnalisé.

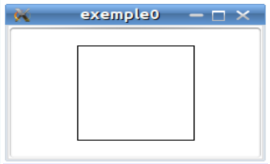
Entre autres choses, `QGraphicsItem` supporte les fonctionnalités suivantes : les événements souris et clavier, le glisser/déposer (*drag and drop*), le groupement d'éléments, la détection des collisions.

## Exemple



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

```
1  #include <QApplication>
2  #include <QGraphicsScene>
3  #include <QGraphicsRectItem>
4  #include <QgraphicsView>
5
6  int main(int argc, char **argv) {
7    QApplication app(argc, argv);
8    QGraphicsScene scene;
9
10   QGraphicsRectItem *rect = scene.addRect(QRectF(0, 0, 100, 100));
11
12   QGraphicsView view(&scene);
13
14   view.show();
15
16 return app.exec();
17 }
```




## Plan intermédiaire



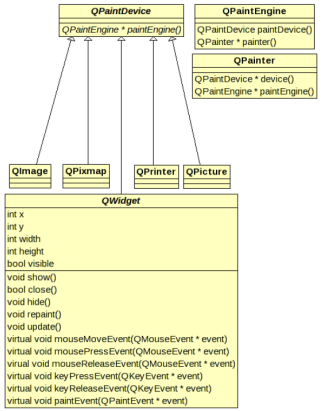


## Organisation



En collaboration avec les classes `QPaintDevice` et `QPaintEngine`, `QPainter` est la base du système de dessin de Qt.

- `QPainter` est la classe utilisée pour effectuer les opérations de dessin,
- `QPaintDevice` représente un dispositif qui peut être peint en utilisant un `QPainter`.
- `QPaintEngine` fournit le moteur de rendu et l'interface (abstraite) que `QPainter` utilise pour dessiner sur différents types de dispositifs suivant la plateforme utilisée.




## QPainter



La classe `QPainter` est la classe de base de dessin bas niveau sur les widgets et les autres dispositifs de dessins.

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

# repaint()

Un widget est « repeint » :

- lorsque une fenêtre passe au dessus,
- lorsque l'on déplace le composant
- ...
- lorsque l'on le lui demande explicitement :
  - `repaint()` entraîne un rafraichissement immédiat,
  - `update()` met une demande de rafraîchissement en file d'attente

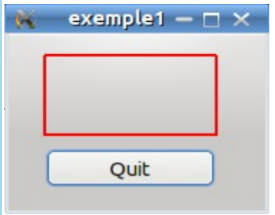
Dans tous les cas, c'est la méthode `paintEvent` qui est appelée :

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

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

# Exemple

```
1  class MyWidget : public QWidget
2  {
3  public:
4  MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
5  void paintEvent(QPaintEvent* e)
6  {
7    QWidget::paintEvent(e); // effectue le comportement
         standard
8    QPainter painter(this); // construire
9
10   painter.setPen( QPen(Qt::red, 2) ); // personnaliser
11   painter.drawRect( 25, 15, 120, 60 ); // dessiner
12 };
13 } // détruire
```




# Méthodes

La classe `QPainter` fournit de nombreuses méthodes :

- `setPen()` : lignes et contours
- `setBrush()` : remplissage
- `setFont()` : texte
- `setTransform()`, etc. : transformations affines
- `setClipRect/Path/Region()` : clipping (découpage)
- `setCompositionMode()` : composition

Lignes et contours : `drawPoint()`, `drawPoints()`, `drawLine()`, `drawLines()`, `drawRect()`, `drawRects()`, `drawArc()`, `drawEllipse()`, `drawPolygon()`, `drawPolyline()`... et `drawPath()` pour des chemins complexes

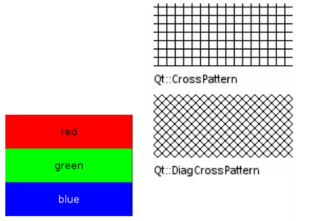
Remplissage : `fillRect()`, `fillPath()`

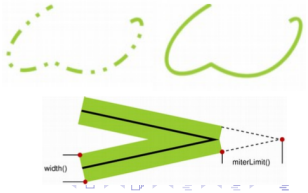
Divers : `drawText()`, `drawPixmap()`, `drawImage()`, `drawPicture()`.

# Compléments

Conjointement à la classe `Qpainter`, on utilise de nombreuses autres classes utiles :

- entiers : `QPoint`, `QLine`, `QRect`, `QPolygon`
- flottants : `QPointF`, `QLineF`...
- chemin complexe : `QPainterPath`
- zone d'affichage : `QRegion`
- stylo (trait) : `QPen`
- pinceau (remplissage) : `QBrush`
- couleur : `QColor`



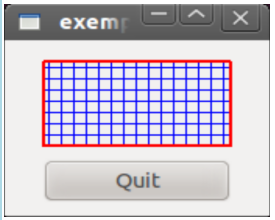



# Exemple2

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

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

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

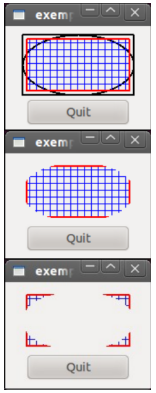

# Exemple3

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

void paintEvent(QPaintEvent* e)
{
 QPainter painter(this);
 QBrush brush;
 brush.setStyle(Qt::CrossPattern);
 brush.setColor(Qt::blue);

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



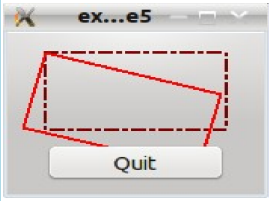

# Exemple4

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

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

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



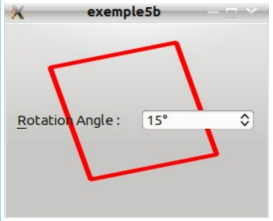

# Exemple 5

```
class MyWidget : public QWidget
{
public:
MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
void paintEvent(QPaintEvent* e)
{
QPainter painter(this);
painter.setPen( QPen(Qt::darkRed, 2, Qt::DashDotLine) )
    ;

 painter.drawRect( 25, 15, 120, 60 );
 painter.translate(25, 15);
 painter.rotate( 15.0 );
 painter.translate(-25, -15);
 painter.setPen( QPen(Qt::red, 2) );
 painter.drawRect( 25, 15, 120, 60 );
}
};
```



## Exemple5b

```
1  class MyWidget  : public Qwidget {
2  Q_OBJECT
3  qreal rotationAngle;
4
5  public :
6  MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
7  void paintEvent(QPaintEvent* e) {
8     QPainter painter(this);
9     painter.setRenderHint(QPainter::Antialiasing);
10    painter.scale(width() / 100.0, height() / 100.0);
11    painter.translate(50.0, 50.0);
12    painter.rotate(-rotationAngle);
13    painter.translate(-50.0, -50.0);
14
15    painter.setPen( QPen(Qt::red, 2) );
16    painter.drawRect( 25, 15, 50, 60 );
17 }
18 public slots :
19 void setRotationAngle(int degrees) {
20    rotationAngle = (qreal)degrees;
21    update();
22 }
23 };
```




## Les images

Qt fournit quatre classes de traitement des données d'image : `QImage`, `QPixmap`, les `QBitmap` et `QPicture`.

- `QImage` fournit une représentation d'image indépendante du matériel qui permet un accès direct aux pixels. `QImage` est conçu et optimisé pour les E/S.
- `QPixmap` est conçu et optimisé pour afficher les images sur l'écran.
- `QBitmap` n'est qu'une classe de commodité qui hérite `QPixmap`.
- `QPicture` est un dispositif permettant d'enregistrer des commandes d'un `QPainter` et de les rejouer.

Ces 4 classes héritent de `QPaintDevice` et on peut donc dessiner dedans avec un `QPainter`.

Elles possèdent aussi les méthodes `load()` et `save()` d'accès aux fichiers (dont les principaux formats sont supportés).

## Exemple

```
1  class MyWidget : public QWidget
2  {
3  public:
4  MyWidget( QWidget *parent = 0 ) : QWidget( parent ) {}
5  void paintEvent(QPaintEvent* e)
6  {
7  QWidget::paintEvent(e);
8  QPainter painter(this);
9
10 QRect rect(0, 0, 170, 45);
11 QPixmap pixmap;
12 pixmap.load("qt-logo.png");
13
14 painter.setPen( QPen(Qt::green, 2);
15 painter.drawText(rect, Qt::AlignCenter, tr("Qt by\
       nNokia"));
16 painter.drawPixmap(45, 50, pixmap);
17 };
18 }
```


## Plan intermédiaire

# Événements

Gestion des événements sur un `QWidget` :

- lorsqu'on presse un bouton :
  `void mousePressEvent(QMouseEvent* e);`
- lorsqu'on relâche un bouton :
  `void mouseReleaseEvent(QMouseEvent* e);`
- lorsqu'on déplace la souris :
  `void mouseMoveEvent(QMouseEvent* e);`
- lorsqu'on double-clique :
  `void mouseDoubleClickEvent(QMouseEvent* e);`

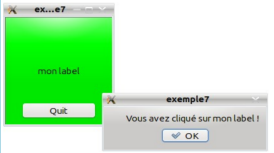
Pour recevoir des événements de la souris dans ses propres widgets, il suffit donc de réimplémenter ces gestionnaires d'évènements (*event handler*).

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

# Exemple 7

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

```
1  class MyLabel  : public QLabel
2  {
3  Q_OBJECT
4  ... // voir diapo suivante
5
6  void mousePressEvent(QMouseEvent *e)
7  {
8     if(e->button() == Qt : :LeftButton)
9     emit clicked();
10 }
11
12 private slots :
13 void selection() {
14    QMessageBox msgBox;
15    msgBox.setText(QString : :fromUtf8("Vous avez cliqué sur mon label !"));
16    msgBox.exec();
17 }
18
19 signals :
20    void clicked();
21 };
```




(Exemple du TP3)

# Exemple - suite

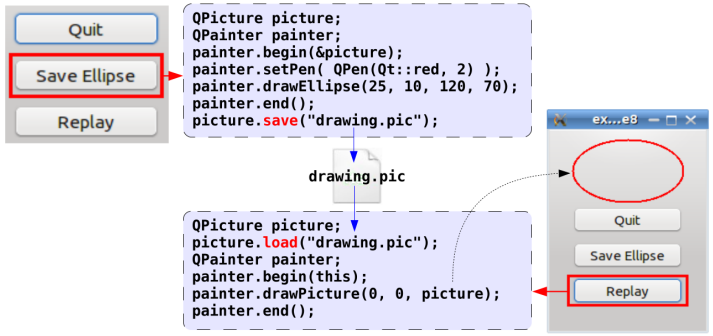
```
1  class MyLabel  : public Qlabel {
2  Q_OBJECT
3  public :
4  MyLabel( QLabel *parent = 0 ) : QLabel( parent ) {
5     QPalette palette;
6     palette.setColor(QPalette : :Window,
7
8     QColor(QColor(0,255,0)));
9        setAutoFillBackground(true);
10       setPalette(palette);
11       setFrameStyle(QFrame : :Panel | QFrame : :Sunken);
12       setText("mon label");
13       setAlignment(Qt : :AlignHCenter | Qt : :AlignVCenter);
14
15    connect(this,SIGNAL(clicked()),this,SLOT(selection()));
16
17 //clic bouton gauche
18    QAction *quitAction = new QAction(tr("E&xit"), this);
19    quitAction->setShortcut(tr("Ctrl+Q"));
20
21    connect(quitAction,SIGNAL(triggered()),qApp,SLOT(quit()));
22
23    addAction(quitAction);
24    setContextMenuPolicy(Qt : :ActionsContextMenu); // clic bouton droit
25 }
26 };
```


# Plan intermédiaire

# QPicture



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

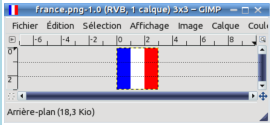



# QImage



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

```
1  #include <QImage>
2
3  int main(int argc, char **argv) {
4     QImage image(3, 3, QImage : :Format_RGB32);
5     QRgb value;
6
7  value = qRgb(0, 0, 255); // 0x0000FF
8
9  for (int i = 0; i < 3; i++)
10    image.setPixel(0, i, value);
11
12 value = qRgb(255, 255, 255); // blanc
13
14 for (int i = 0; i < 3; i++)
15    image.setPixel(1, i, value);
16
17 value = qRgb(255, 0, 0);
18
19 for (int i = 0; i < 3; i++)
20    image.setPixel(2, i, value);
21
22 image.save("france.png", "PNG");
23 return 0;
24 }
```



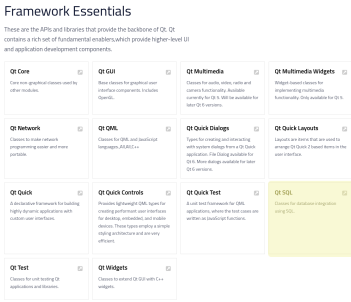

# Plan intermédiaire





# Introduction






Qt propose un mécanisme simple de connexion à une base de données relationnelle `SQL` via des « *drivers* » et des interfaces.
Le module principal de cette liaison est `QtSQL`[10]

[10] `https://doc.qt.io/qt-5/qtsql-index.html`