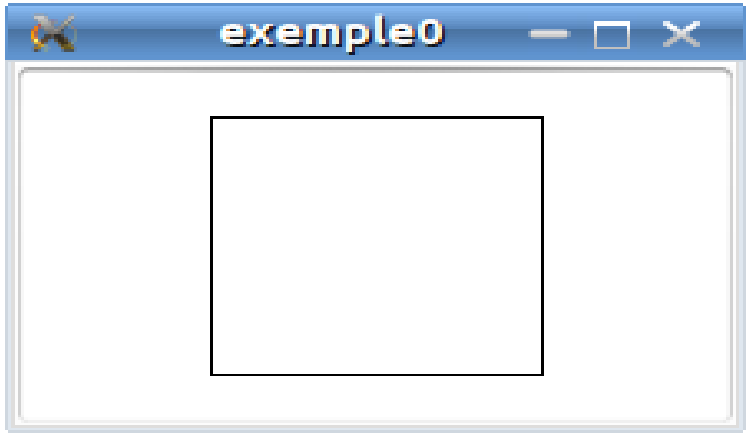
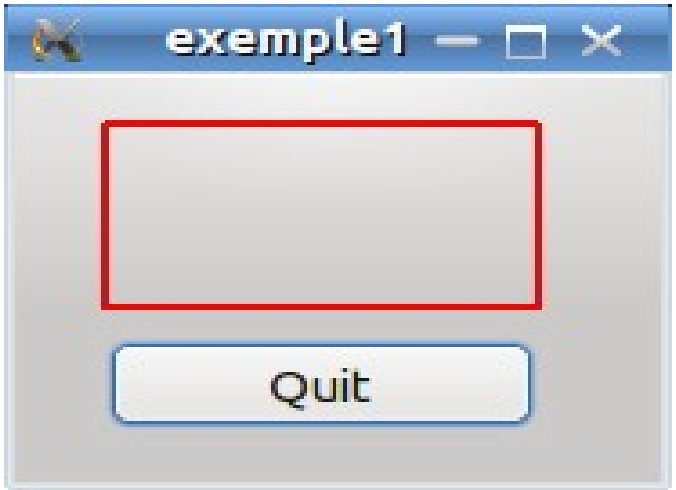
Graphique 2D

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



Exemple utilisant l'architecture Graphics View



Exemple utilisant QPainter

Le framework Graphics View (1/4)

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

Le framework Graphics View (2/4)

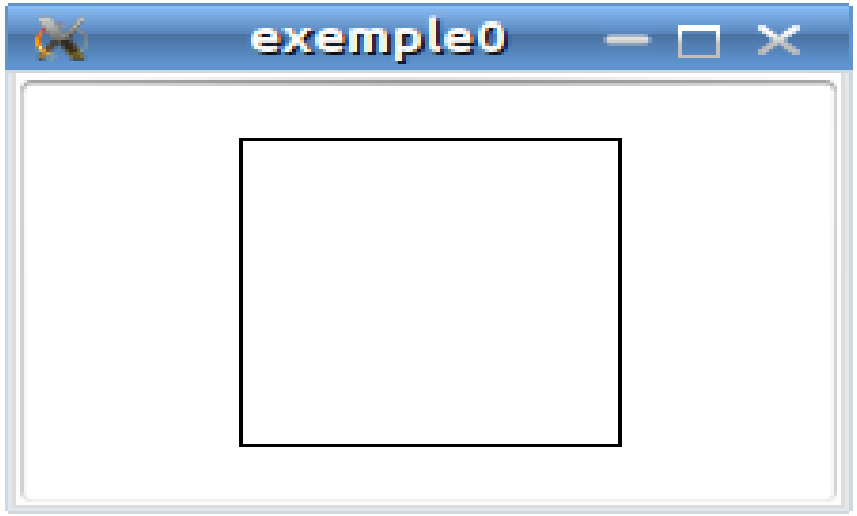
- La classe **QGraphicsScene** fournit la scène pour l'architecture Graphics View. La scène a les responsabilités suivantes :
  - fournir une interface rapide pour gérer un grand nombre d'éléments,
  - propager les événements à chaque élément,
  - gérer les états des éléments (telles que la sélection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

Le framework Graphics View (3/4)

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

Le framework Graphics View (4/4)

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




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

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

   QGraphicsScene scene;

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

   QGraphicsView view(&scene);

   view.show();

   return app.exec();
}
```

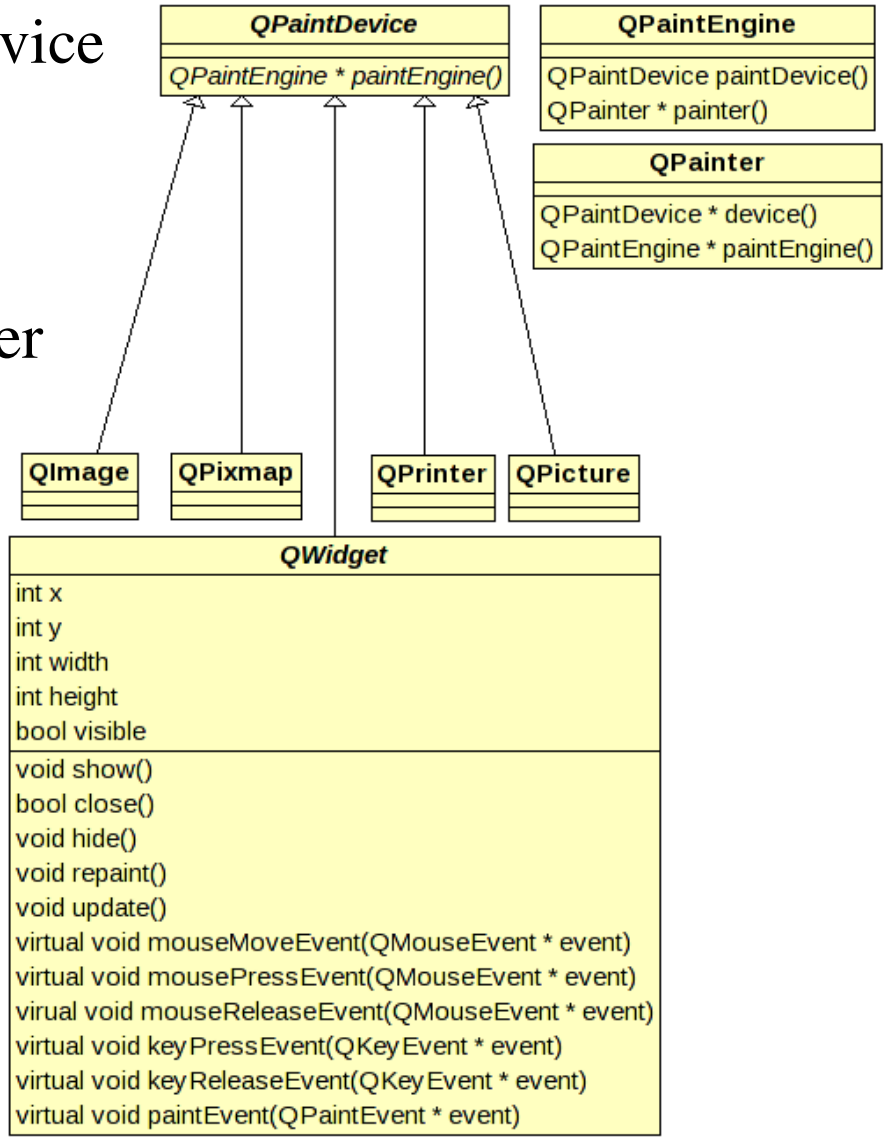
Dessin 2D

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

- **QPainter** est la classe utilisée pour effectuer les opérations de dessin.

- **QPaintDevice** représente un dispositif qui peut être peint en utilisant un QPainter.

- **QPaintEngine** fournit le moteur de rendu et l'interface (abstraite) que QPainter utilise pour dessiner sur différents types de dispositifs suivant la plate-forme utilisée.



QPainter (1/12)

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

QPainter (2/12)

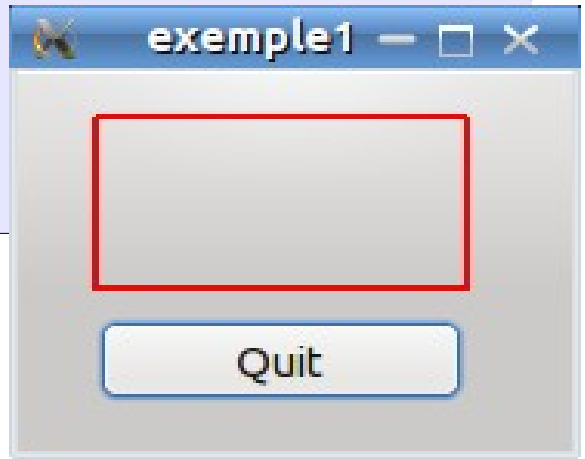
- Un **widget** est "repeint" :
  - Lorsque une fenêtre passe au dessus
  - Lorsque l'on déplace le composant
  - ...
  - Lorsque l'on le lui demande explicitement :
    - **`repaint()`** entraîne un rafraichissement immédiat
    - **`update()`** met une demande de rafraîchissement en file d'attente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

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

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

QPainter : exemple 1 (3/12)


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

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

     QPainter painter(this); // construire

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

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

   } // détruire
};
```

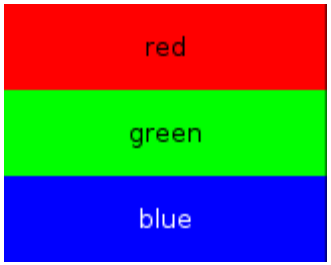


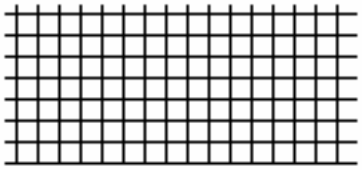
QPainter (4/12)

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

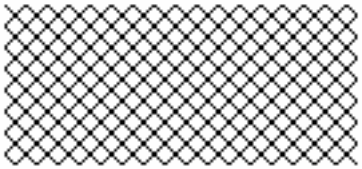
QPainter (5/12)

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

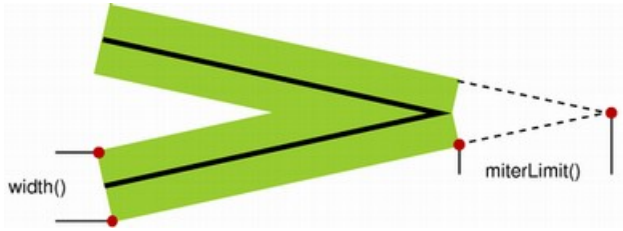




Qt::CrossPattern

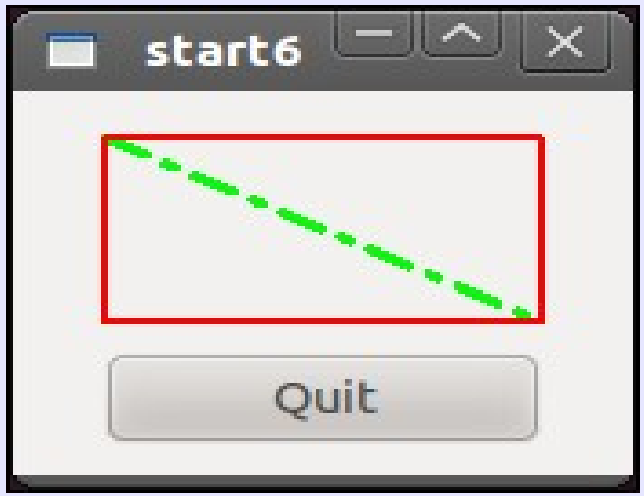


Qt::DiagCrossPattern

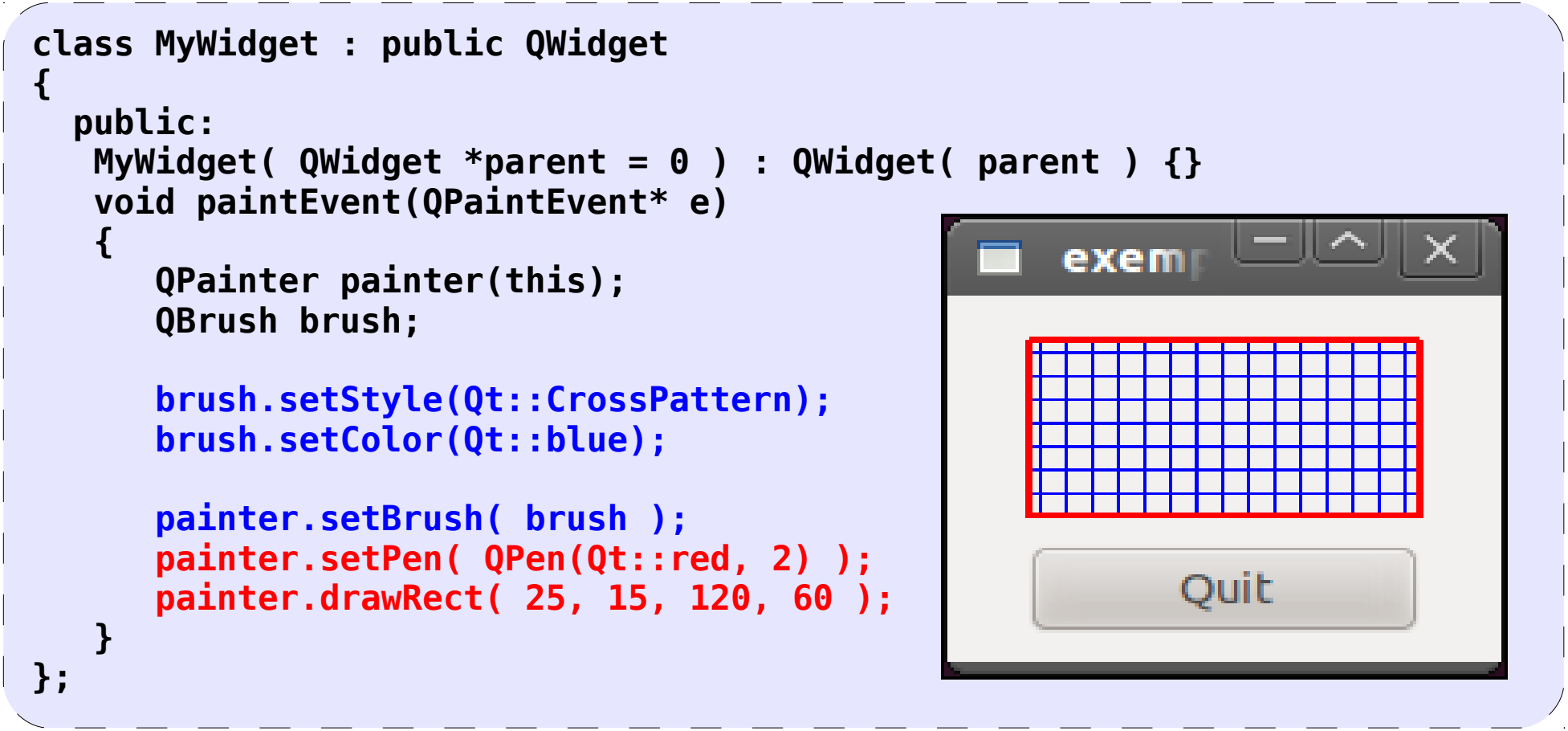


QPainter : exemple 2 (6/12)

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

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

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



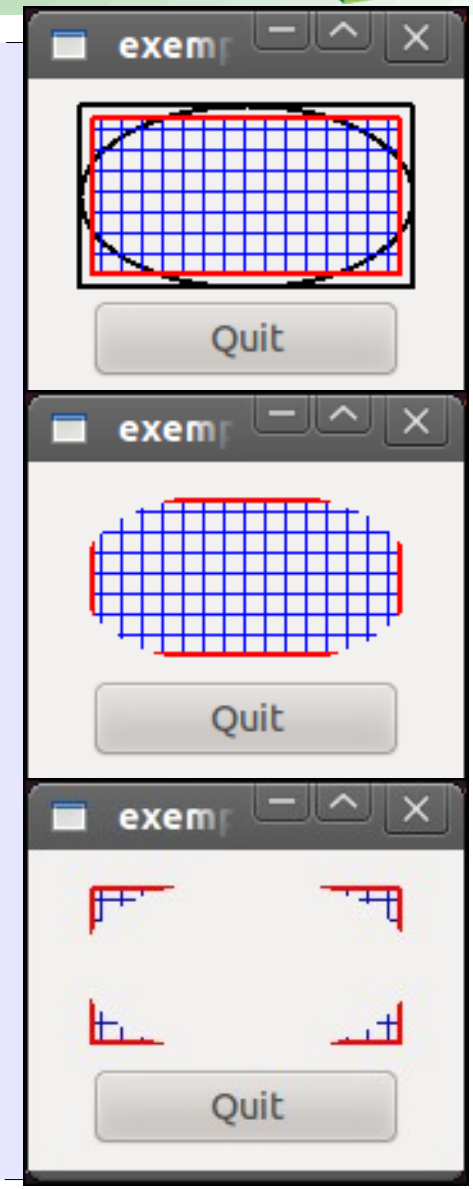
QPainter : exemple 3 (7/12)



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

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

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

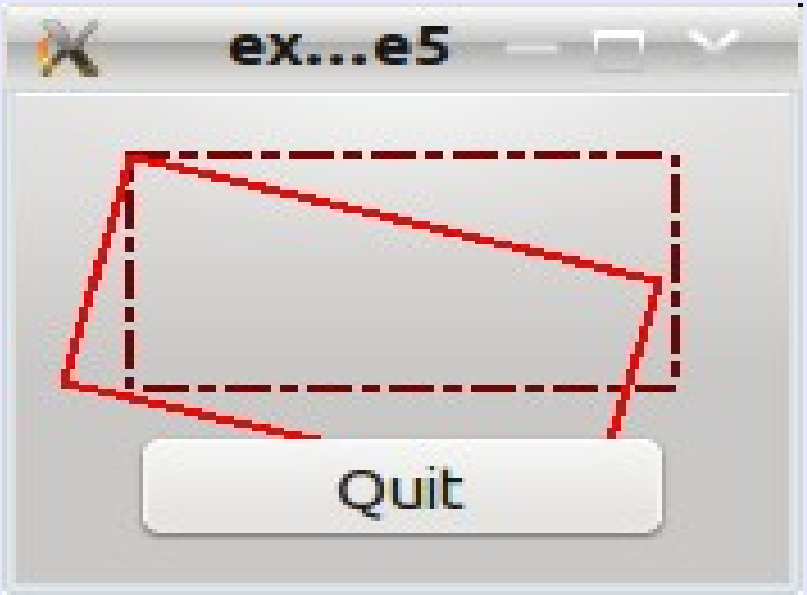
QPainter : exemple 4 (8/12)

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

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

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



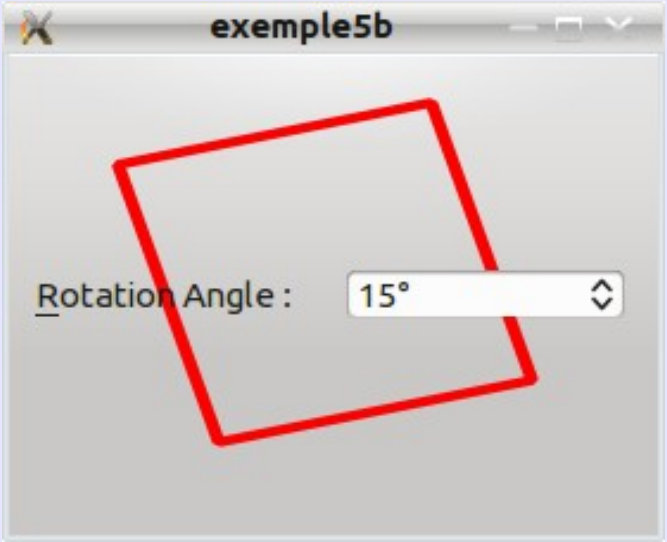
QPainter : exemple 5 (9/12)

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

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

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

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



QPainter : exemple 5b (10/12)

```
class MyWidget : public Qwidget {
   Q_OBJECT

   qreal rotationAngle;

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



QPainter : les images (11/12)

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

QPainter : exemple 6 (12/12)


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

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

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

Gestionnaire d'évènements souris

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

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

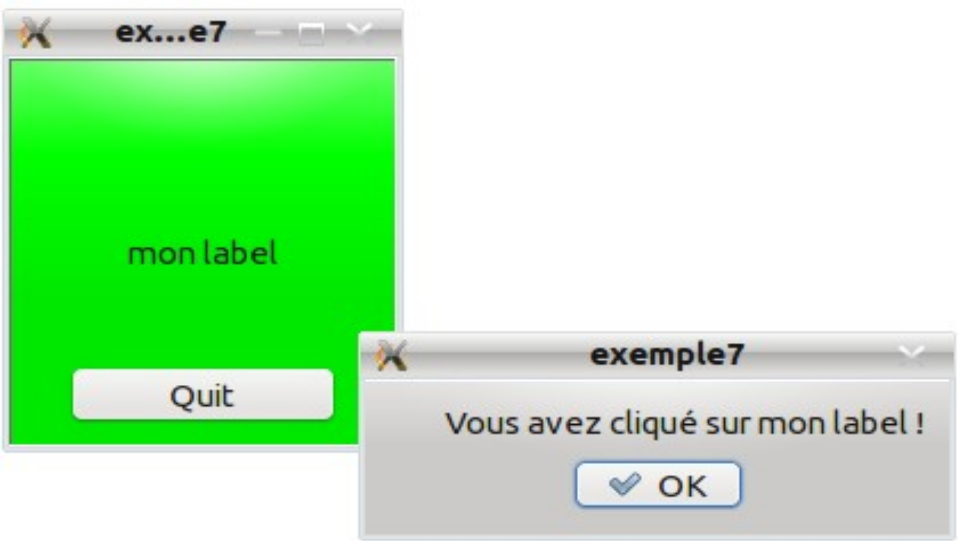
- Pour recevoir des événements de la souris dans ses propres widgets, il suffit donc de réimplémenter ces gestionnaires d'évènements (*event handler*).

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

Exemple 7 (1/2)

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

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



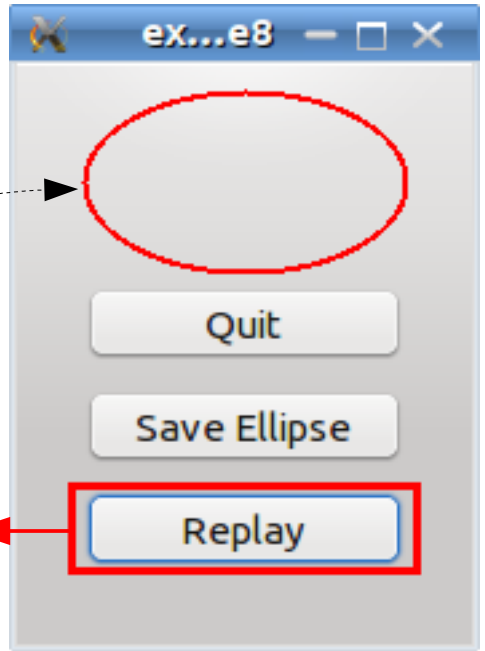
Exemple 7 (2/2)


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

QPicture : exemple 8


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


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



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

QImage : exemple 9

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

```
#include <QImage>

int main(int argc, char **argv) {
   QImage image(3, 3, QImage::Format_RGB32);
   QRgb value;

   value = qRgb(0, 0, 255); // 0x0000FF
   for(int i = 0; i < 3; i++)
      image.setPixel(0, i, value);

   value = qRgb(255, 255, 255); // blanc
   for(int i = 0; i < 3; i++)
      image.setPixel(1, i, value);

   value = qRgb(255, 0, 0);
   for(int i = 0; i < 3; i++)
      image.setPixel(2, i, value);

   image.save("france.png", "PNG");

   return 0;
}
```