Programmation événementielle

- Présentation
- Licences
- Environnement de Développement
- Structure générale de Qt
- Premier programme
- Principes
- Compilation
- Signaux et slots
- Gestion mémoire
- Propriétés
- Layout
- Internationalisation

Présentation (1/5)

- Qt est une **bibliothèque logicielle orientée objet** développée en C++ par Qt Development Frameworks, filiale de Nokia.
- Qt est une plateforme de développement d'interfaces graphiques (GUI - Graphical User Interface) fournie à l'origine par la société norvégienne Troll Tech, puis rachetée par Nokia en février 2008 (http://qt.nokia.com/).
- Qt permet la portabilité des applications (qui n'utilisent que ses composants) par simple recompilation du code source.
- Les environnements supportés sont les **Unix** (dont Linux) qui utilisent le système graphique X Window System, **Windows** et **Mac OS X**.

Présentation (2/5)

- Qt est principalement dédiée au développement d'interfaces graphiques en fournissant des éléments prédéfinis appelés widgets (pour *windows gadgets*) qui peuvent être utilisés pour créer ses propres fenêtres et des boîtes de dialogue complètement prédéfinies (ouverture / enregistrement de fichiers, progression d'opération, etc).
- Qt fournit également un ensemble de classes décrivant des éléments non graphiques : accès aux données, connexions réseaux (*socket*), gestion des fils d'exécution (*thread*), analyse XML, etc.
- Qt dispose d'un moteur de rendu graphique 2D performant.

Présentation (3/5)

- Depuis, Qt4 sépare la bibliothèque en modules :
  - **QtCore** : pour les fonctionnalités non graphiques utilisées par les autres modules ;
  - **QtGui** : pour les composants graphiques ;
  - QtNetwork : pour la programmation réseau ;
  - QtOpenGL : pour l'utilisation d'OpenGL ;
  - QtSql : pour l'utilisation de base de données SQL ;
  - QtXml : pour la manipulation et la génération de fichiers XML ;
  - QtDesigner : pour étendre les fonctionnalités de Qt Designer, l'assistant de création d'interfaces graphiques ;
  - QtAssistant : pour l'utilisation de l'aide de Qt ;
  - Qt3Support : pour assurer la compatibilité avec Qt 3.
  - et de nombreux autres modules, etc.
- Qt utilise Unicode 4.0 pour la représentation de ses chaînes de caractères.

Présentation (4/5)

- 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.

- 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.

- De plus en plus de développeurs utilisent Qt, y compris parmi de grandes entreprises. On peut notamment citer : Google, Adobe Systems, Asus, Samsung, Philips, ou encore la NASA et bien évidemment Nokia.

Présentation (5/5)

- L'environnement de Qt est aussi constitué de :
  - Qt Jambi : les possibilités de Qt pour le langage JAVA
  - Qtopia : une version de Qt destinée aux dispositifs portables (téléphones, PocketPC, etc..) et aux systèmes embarqués (ex Qt/Embedded)
  - QSA : Qt Script for Applications, ajout de scripts à ses applications.
  - Teambuilder : architecture de compilation distribuée pour les gros projets d'entreprise.

Licences

- La société Trolltech mit tout d'abord la version Unix/Linux de Qt sous licence **GNU GPL** lorsque l'application développée était également sous GNU GPL. Pour le reste, c'est la licence commerciale qui entre en application. Cette politique de double licence est appliquée pour tous les systèmes depuis la version 4.0 de Qt.
- Le 14 janvier 2009, Trolltech annonce qu'à partir de Qt 4.5, Qt sera également disponible sous licence **LGPL v2.1** (Licence publique générale limitée GNU). Cette licence permet ainsi des développements de logiciels propriétaires, sans nécessiter l'achat d'une licence commerciale auprès de Qt Development Frameworks.
  - Les licences Qt : http://qt.nokia.com/products/licensing/
  - Les licences GNU : http://www.gnu.org/licenses/licenses.fr.html

Environnement de Développement Intégré (EDI)

- **Qt Creator** 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, le complètement, l'indentation, etc... Qt Creator intègre en son sein les outils Qt Designer et Qt Assistant. Il intègre aussi un mode débuggage.

Remarque : 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 et développés indépendamment de Nokia, comme QDevelop et Monkey Studio.

- **Qt Designer** est un logiciel qui permet de créer des interfaces graphiques Qt dans un environnement convivial. L'utilisateur, par glisser-déposer, place les composants d'interface graphique et y règle leurs propriétés facilement. Les fichiers d'interface graphique sont formatés en XML et portent l'extension **.ui**. Lors de la compilation, un fichier d'interface graphique est converti en classe C++ par l'utilitaire **uic**.

Integrated Development Environment (IDE)

- **QDevelop** est un environnement de développement intégré libre pour Qt. Le but de QDevelop est de fournir dans les environnements les plus utilisés, Linux, Windows et Mac OS X d'un outil permettant de développer en Qt de la même manière avec un IDE unique. Il intègre également les outils Qt-Designer pour la création d'interface graphique et Qt-Linguist pour le support de l'internationalisation.

- **KDevelop** est un environnement de développement intégré (IDE) pour KDE. Il intègre également les outils Qt-Designer pour la création d'interface graphique et Qt-Linguist pour la gestion de l'internationalisation.

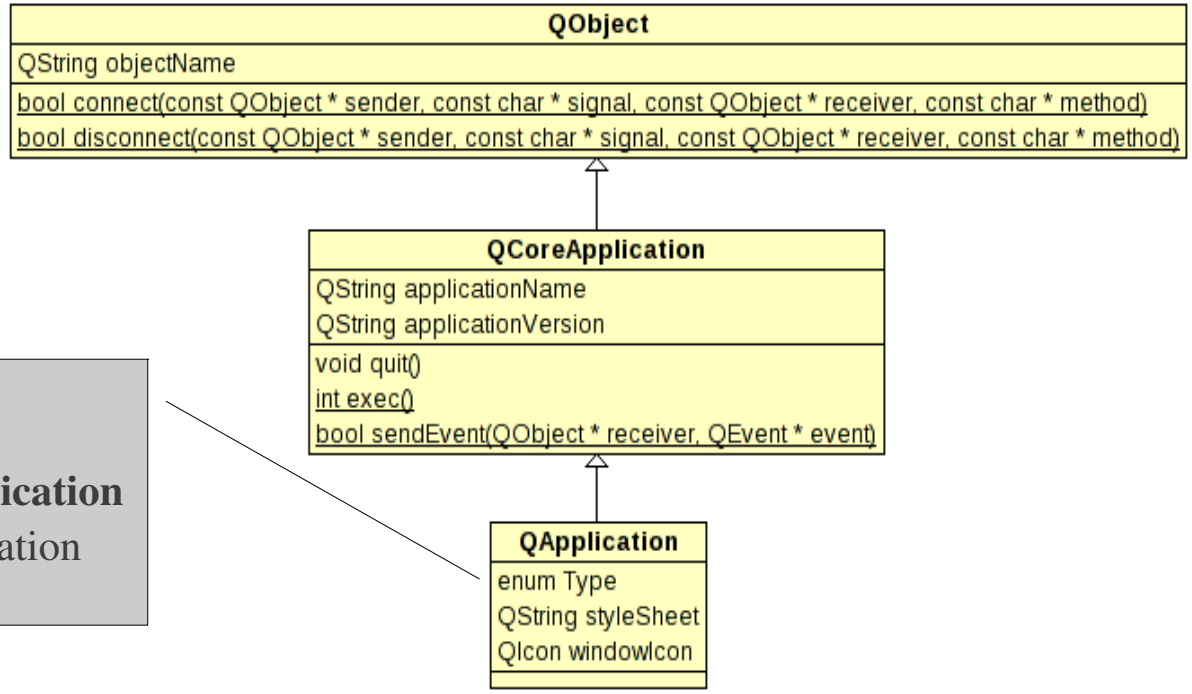
- Autres bibliothèques généralistes multi-plateformes, parmi les plus connus :
  - GTK+, utilisée par l'environnement graphique GNOME
  - wxWidgets (anciennement wxWindows)

Structure générale des classes Qt (1/3)

- L'API Qt est constituée de classes aux noms préfixés par Q et dont chaque mot commence par une majuscule (QLineEdit, QLabel, ...).
- La classe **QObject** est la classe mère de toutes les classes Qt.
- Arborescence des objets : Les objets Qt (ceux héritant de QObject) peuvent s'organiser d'eux-mêmes sous forme d'arbre. Ainsi, lorsqu'une classe est instanciée, on peut lui définir un objet parent.
- En dérivant de **QObject**, un certain nombre de spécificités Qt sont hérités :
  - signaux et slots : mécanisme de communication entre objets
  - une gestion simplifiée de la mémoire
  - un objet Qt peut avoir des propriétés
  - introspection : chaque objet de Qt a un méta-objet (une instance de la classe QMetaObject) pour fournir des informations au sujet de la classe courante.
- L'utilitaire **moc** permet l'implémentation de ces mécanismes.

Structure générale des classes Qt (2/3)

- **QObject** est le coeur du modèle objet de Qt. L'élément central de ce modèle est un mécanisme pour la communication entre objet appelé "signaux et les slots". On peut connecter un signal à un slot avec connect() et détruire ce lien avec disconnect().
- La classe **QCoreApplication** fournit une boucle d'événements pour les applications Qt console (non-GUI). QCoreApplication contient donc la boucle principale, où tous les événements du système d'exploitation et d'autres sources sont traités et expédiés. Elle gère également l'initialisation et la finalisation de l'application, ainsi que ses paramètres.

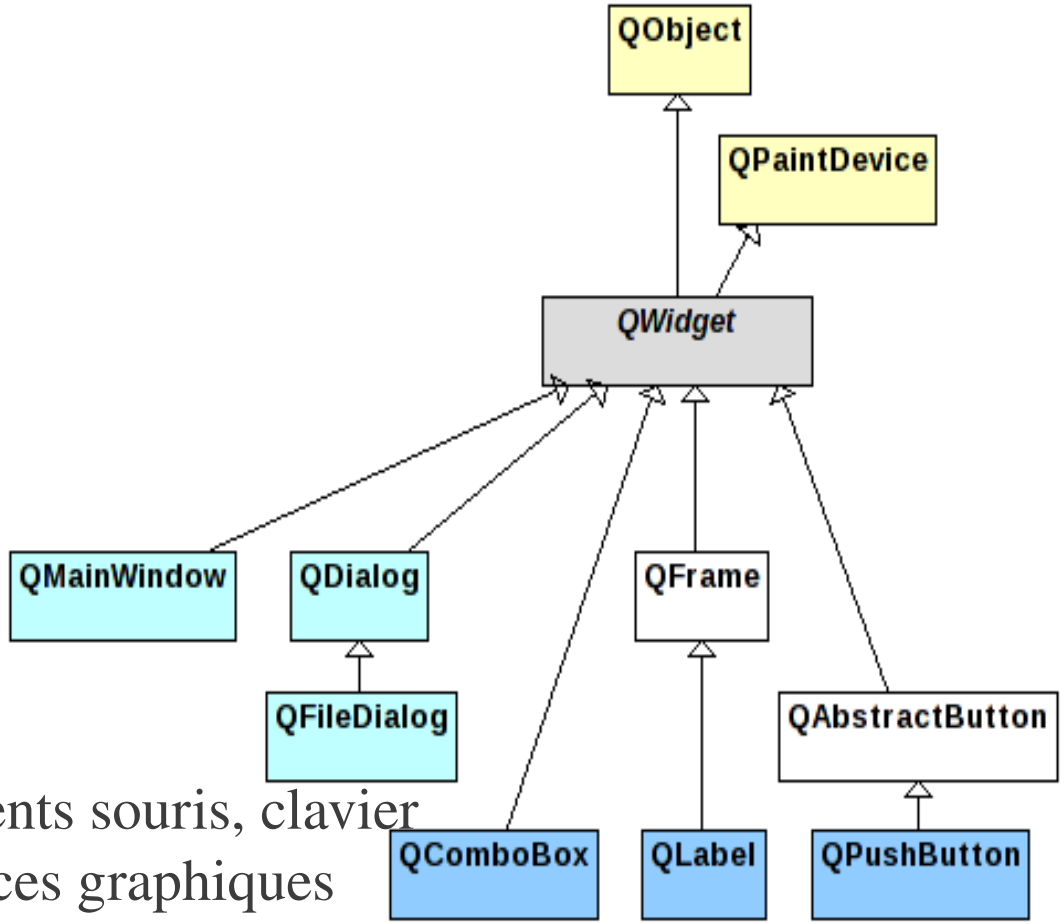


enum QApplication::Type :
- QApplication::Tty : a console application
- **QApplication::GuiClient : a GUI client application**
- QApplication::GuiServer : a GUI server application (for Qt for Embedded Linux)

Structure générale des classes Qt (3/3)

La classe **QWidget** :

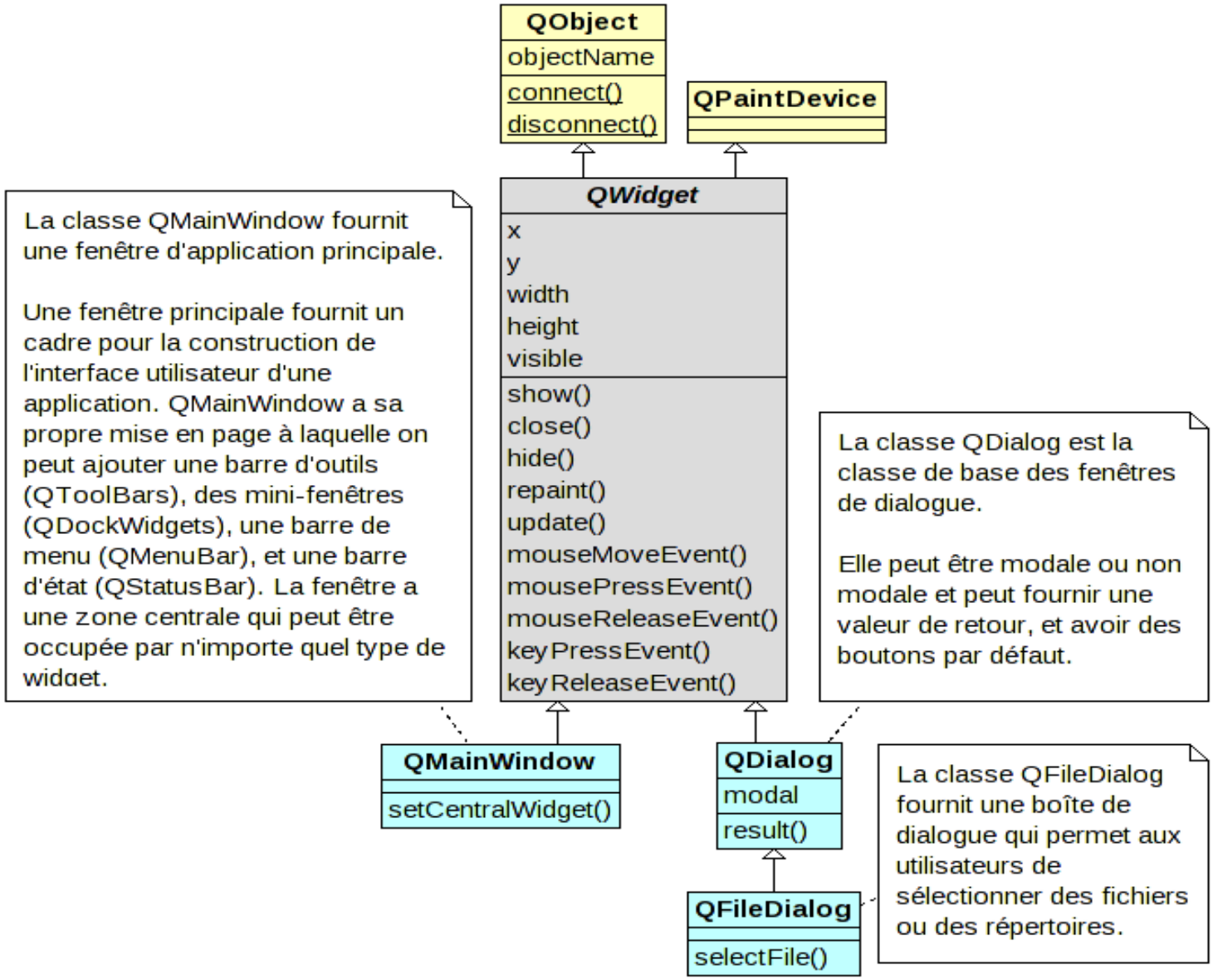
- elle hérite de QObject
- elle est la classe mère de toutes les classes servant à réaliser des interfaces graphiques



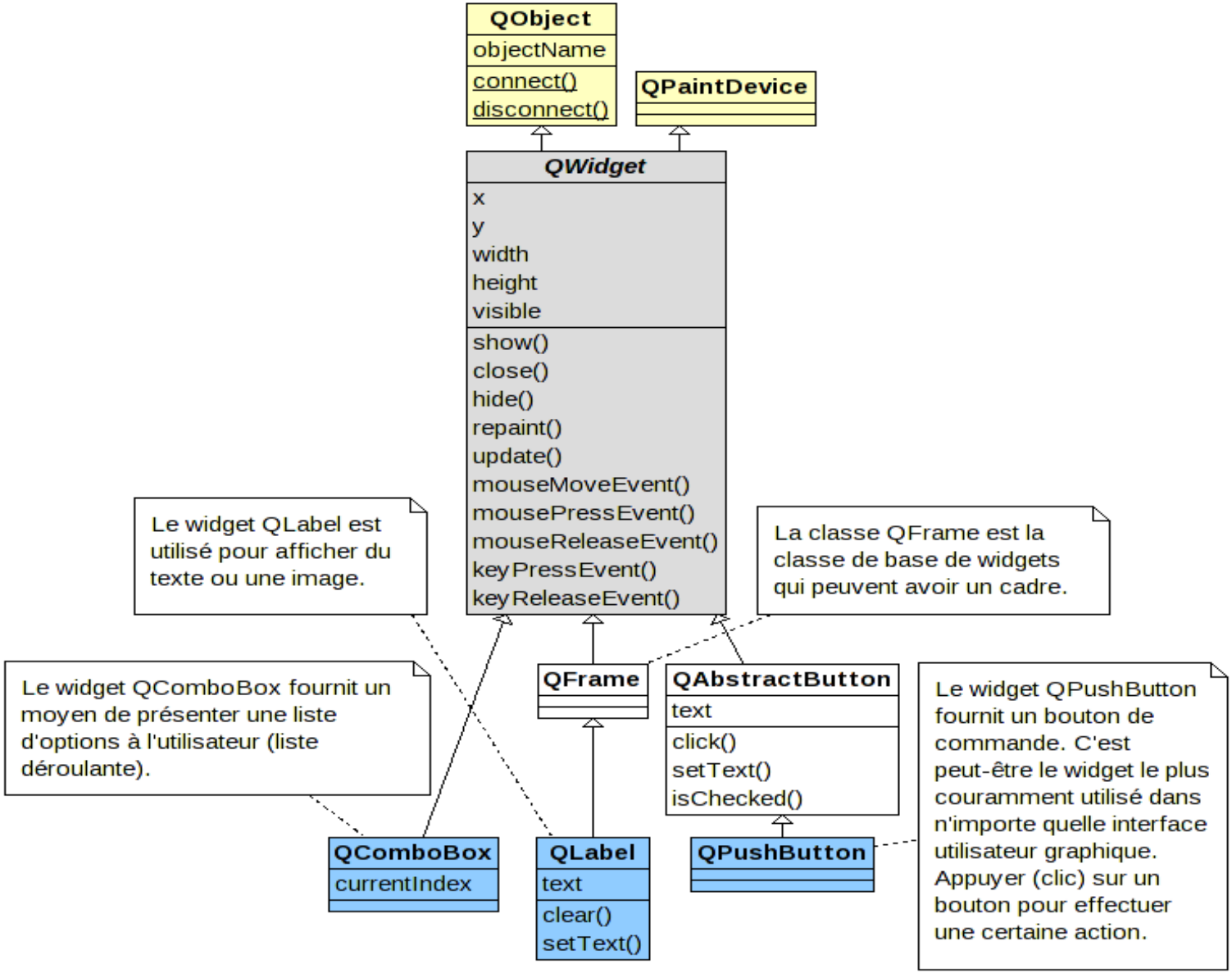
Les **widgets** :

- sont capable de se "peindre"
- sont capable de recevoir les évènements souris, clavier
- sont les éléments de base des interfaces graphiques
- sont tous rectangulaires
- ils sont ordonnés suivant l'axe z (gestion de la profondeur)
- ils peuvent avoir un widget parent

Widget de haut niveau : Les classes « fenêtre »



Widget de bas niveau : Les classes «widget»



Premier programme

- Qt4 pose comme principe que pour utiliser une classe, il faut inclure le fichier *header* du nom de la classe.

```
#include <QApplication>
#include <QLabel>

int main(int argc, char **argv)
{
    QApplication* app = new QApplication(argc, argv);
    QLabel* hello = new QLabel("Hello Qt !");

    hello->show();

    return app->exec();
}
```

Affichage du widget

Boucle d'attente des événements jusqu'à la fermeture du dernier widget

- Un objet de type QApplication doit être créé avant la création des widgets. La classe QApplication gère les options standards d'un programme Qt et les propriétés globales de l'application (style des fenêtres, couleur du fond, taille de la fenêtre principale).

Principes de fonctionnement

- Instanciation de la classe **QApplication**
  - Elle doit être créée avant tout objet graphique et reçoit tous les paramètres transmis à la fonction main (argc, argv). Elle s'occupe de toute la gestion des événements (et les envoie aux widgets concernés). Cet objet est toujours accessible dans le programme grâce au pointeur global nommé qApp.
  - Les applications doivent se terminer proprement en appelant QApplication::quit(). Cette méthode est appelée automatiquement lors de la fermeture du dernier widget.
  - Un widget est toujours créé caché, il est donc nécessaire d'appeler la méthode **show()** pour l'afficher.

Compilation : ligne de commande

- La génération d'une application se fait en plusieurs étapes :
  - **création d'un répertoire et des sources** (dépend de l'EDI utilisé)
    le nom initial du répertoire détermine le nom du projet et donc de l'exécutable qui sera produit (par défaut).
  - `qmake -project`
    génère un fichier **.pro** qui décrit comment générer un Makefile pour compiler ce qui est présent dans le dossier courant.
  - `qmake`
    génère un **Makefile** à partir des informations du fichier .pro
  - `make` ou `mingw32-make`
    appel classique à l'outil make, par défaut il utilise un fichier appelé Makefile

Qmake

- Qt se voulant un environnement de développement portable et ayant l'utilitaire **moc** comme étape intermédiaire avant la phase de compilation/édition de liens, il a été nécessaire de concevoir un moteur de production spécifique. C'est ainsi qu'est conçu le programme **qmake**.
- Ce dernier prend en entrée un fichier (avec l'extension **.pro**) décrivant le projet (liste des fichiers sources, dépendances, paramètres passés au compilateur, etc...) et génère un fichier de projet spécifique à la plateforme. Ainsi, sous les systèmes UNIX/Linux, qmake produit un **Makefile**.
- Le fichier de projet est fait pour être très facilement éditable par un développeur. Il consiste en une série d'affectations de variables.
- Manuel : http://doc.trolltech.com/4.x/qmake-manual.html

Exemple Fichier Projet (.pro)

```
QT += core gui              # modules Qt requis

TEMPLATE = app              # type de projet
TARGET = testWiimote        # nom de l'exécutable

DESTDIR = bin
OBJECTS_DIR = build
MOC_DIR = build
UI_DIR = build

FORMS = WiimoteIHM.ui       # fiche de Qt Designer

HEADERS += WiimoteIHM.h Wiimote.h
SOURCES += WiimoteIHM.cpp main.cpp Wiimote.cpp

LIBS += -lm -lwiiuse
INCLUDEPATH += .
```

Deuxième programme

- D'une manière générale, les widgets sont hiérarchiquement inclus les uns dans les autres. Le principal avantage est que si le parent est déplacé, les enfants le sont aussi.
- Lors de la création d'un widget, on indique le widget parent (ici **box**). Les widgets enfants sont alors dessinés à l'intérieur de l'espace attribué à leur parent.

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>

int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QWidget box;
    QPushButton *quitBtn = new QPushButton("Quit",
&box);
    QObject::connect(quitBtn, SIGNAL(clicked()),
&app, SLOT(quit()));
    box.show();
    return app.exec();
}
```

Signaux et slots (1/6)

- Pour la gestion des événements, Qt utilise un mécanisme de communication d'objets faiblement couplés appelé **signal / slot**. Faiblement couplé signifie que l'émetteur d'un signal ne sait pas quel objet va le prendre en compte (d'ailleurs, il sera peut être ignoré). De la même façon, un objet interceptant un signal ne sait pas quel objet a émis le signal.
- Tout objet dont la classe hérite (directement ou non) de la classe **QObject** peut émettre et recevoir un signal. L'utilitaire **moc** implémentera ce mécanisme.
- Un signal peut être connecté à un autre signal. Dans ce cas, lorsque le premier signal est émis, il entraîne l'émission du second.
- Émission d'un signal peut être "automatique" par exemple lorsqu'on appui sur un bouton, le signal existe déjà dans la classe utilisée (QPushButton).

Signaux et slots (2/6)

- On peut aussi forcer l'émission d'un signal à l'aide de la méthode emit :

```
emit nomSignal(parametreSignal);
```

- La déclaration d'un signal personnalisé est possible, il faut savoir qu'il n'a :
  - pas de valeur de retour (donc void) et
  - pas de définition de la méthode (donc pas corps)
- Pour déclarer un signal, on utilise le mot clé **signals** dans le fichier d'entête (.h) comme public, private...

```
signals :
  void nomSignal(parametreSignal);
```

Signaux et slots (3/6)

- Les slots étant des méthodes, ils peuvent être appelés explicitement (sans présence d'un événement) comme toute méthode.
- Les slots peuvent donc être private, protected ou public avec les conséquences habituelles pour leur connexion (private pour la classe ...). De même, ils peuvent être virtuels et/ou surchargés.
- Les slots ont seulement la propriété d'être connectés à un signal et dans ce cas, il sera automatiquement appelé lorsque le signal sera émis.
- **signal et slot doivent être compatibles (avoir la même signature) pour être connectés.**
- Un signal peut être connecté à plusieurs slots. Attention : les slots sont activés dans un ordre arbitraire.
- Plusieurs signaux peuvent être connectés à un seul slot.

Signaux et slots (4/6)

- Une connexion signal / slot peut être réalisée par la méthode :

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

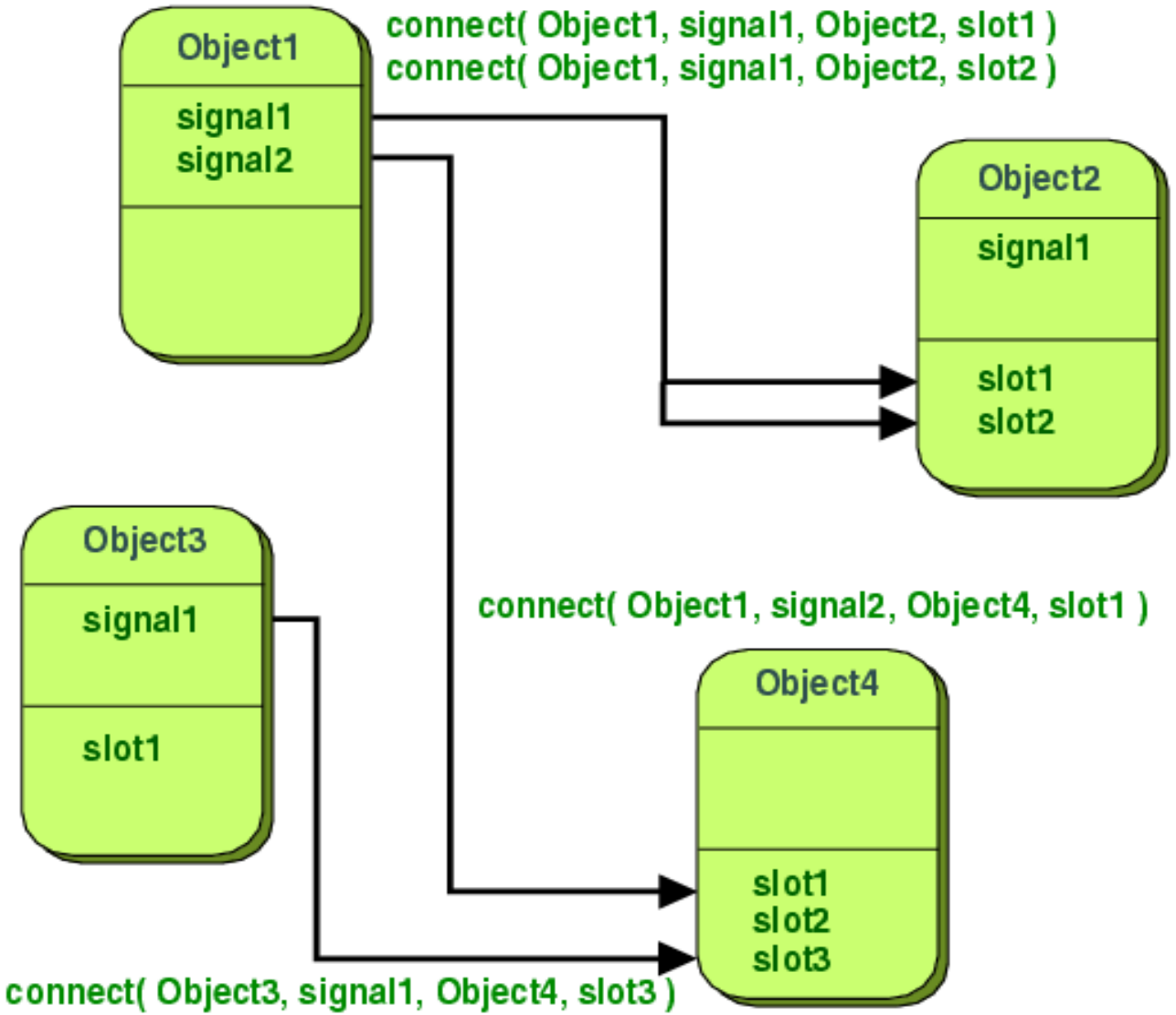
- Une connexion signal/slot peut être supprimée par la méthode :

```
bool QObject::disconnect ( const QObject * sender, const char
* signal, const QObject * receiver, const char * method )
```

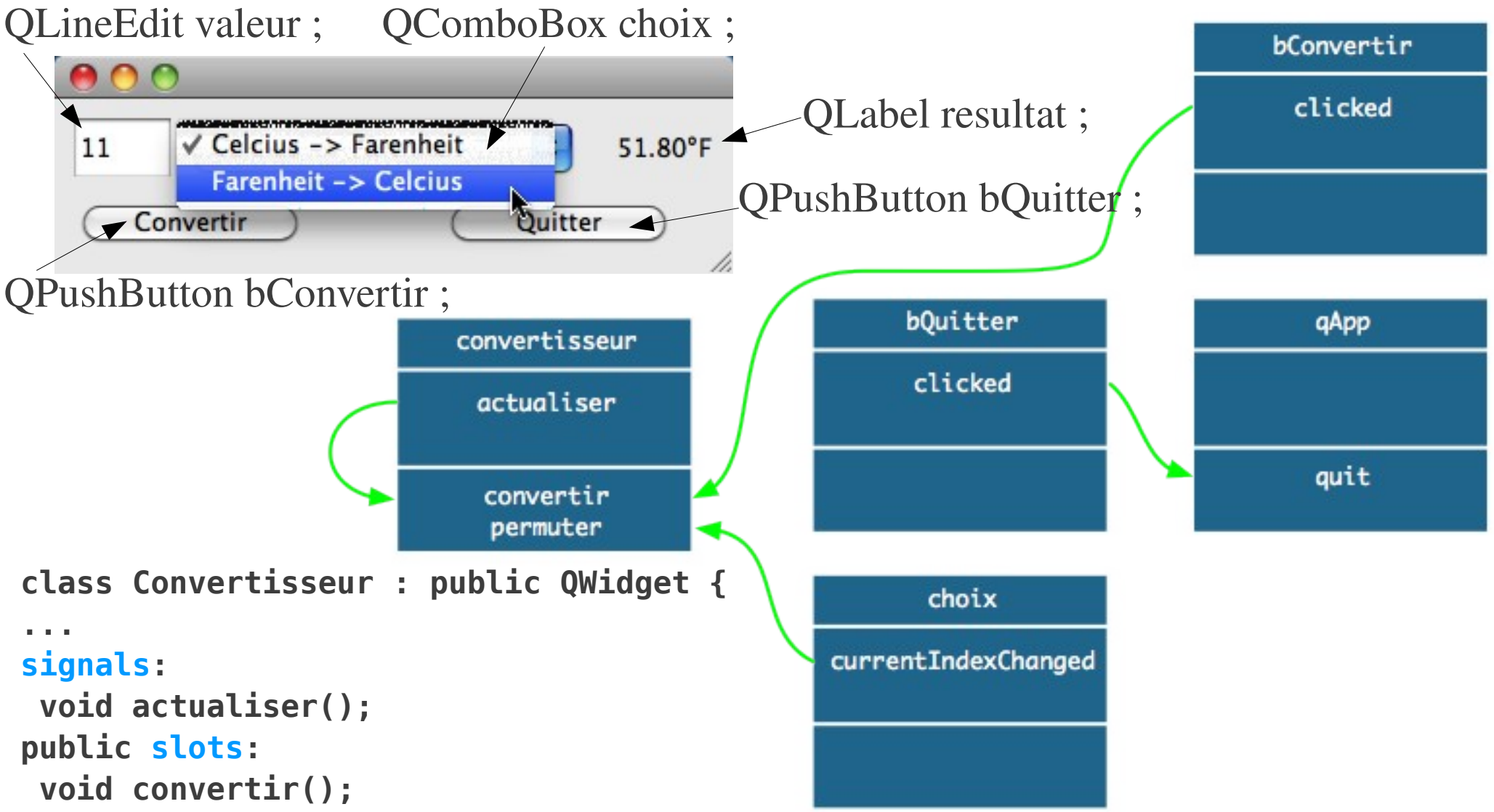
Exemple pour connecter le **signal** **clicked**() d'un bouton au **slot** **quit**() de l'application :

```
QObject::connect(quitBtn, SIGNAL(clicked()), &app, SLOT(quit()));
```

Signaux et slots (5/6)



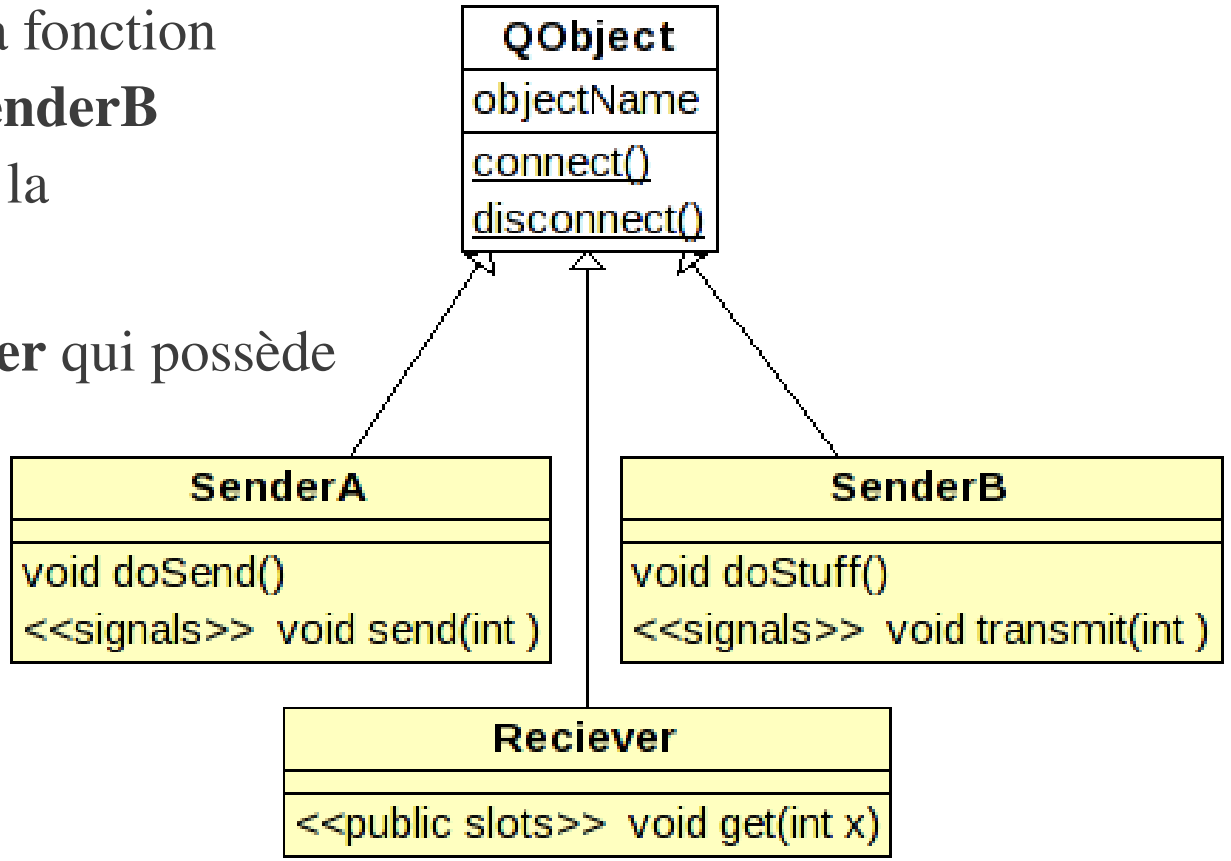
Signaux et slots (6/6)




```
class Convertisseur : public QWidget {
...
signals:
 void actualiser();
public slots:
 void convertir();
 void permuter(int index); };
```

Exemple : signaux et slots (1/6)

- On peut utiliser le mécanisme signal/slot pour ses propres classes en héritant de la classe **QObject**.
- On va utiliser deux classes émettrices : la classe **SenderA** met en application le signal **send** qui sera émis par la fonction membre **doSend** et la classe **SenderB** émet le signal **transmit** depuis la fonction membre **doStuff**.
- Et une classe receveuse **Reciever** qui possède un slot **get**.



Exemple : signaux et slots (2/6)

```
#ifndef SENDER_H
#define SENDER_H
#include <QObject>
#include <iostream>

class SenderA : public QObject {
   Q_OBJECT
   public:
    SenderA( QObject *parent=0 );
    void doSend();
   signals:
    void send( int );
};

class SenderB : public QObject {
   Q_OBJECT
   public:
    SenderB( QObject *parent=0 );
    void doStuff();
   signals:
    void transmit( int );
};

#endif
```

- Le mot clé **Q_OBJECT** est nécessaire dès qu'un mécanisme Qt est utilisé.
- La section **signals** déclare les signaux de la classe.
- Les méthodes **send**() et **transmit**() ne sont pas définies.

- Ce fichier en-tête **`Sender.h`** (*header*) sera précompilé par l'utilitaire **moc** et produira un fichier cpp **`moc_sender.cpp`**.

- *Rappel : une application Qt est 100% C++.*

Exemple : signaux et slots (3/6)

```
#include "sender.h"

SenderA::SenderA( QObject
*parent/*=0*/ ) : QObject( parent )
{
}

void SenderA::doSend()
{
   emit( send( 7 ) );
}

SenderB::SenderB( QObject
*parent/*=0*/ ) : QObject( parent )
{
}

void SenderB::doStuff()
{
   emit( transmit( 5 ) );
}
```

- Dans la méthode doSend : le mot clé **emit** est provoque l'émission du signal **send** avec la valeur **7**.
- Dans la méthode doStuff : le mot clé **emit** est provoque l'émission du signal **transmit** avec la valeur **5**.

Exemple : signaux et slots (4/6)

```
#ifndef RECIEVER_H
#define RECIEVER_H

#include <QObject>
#include <iostream>

class Reciever : public QObject
{
    Q_OBJECT

    public:
     Reciever( QObject *parent=0 );

    public slots:
     void get( int x );
};

#endif
```

- Le mot clé **Q_OBJECT** est nécessaire dès qu'un mécanisme Qt est utilisé.
- La section **slots** déclare les slots de la classe.
- La méthode **get()** sera définie.

- Ce fichier en-tête **Reciever.h** (*header*) sera précompilé par l'utilitaire **moc** et produira un fichier cpp **moc_reciever.cpp**.

- *Rappel : une application Qt est 100% C++.*

Exemple : signaux et slots (5/6)

```
#include "reciever.h"
Reciever::Reciever( QObject *parent/*=0*/
) : QObject( parent ) {}
void Reciever::get( int x ) {
   std::cout << "Recieved: " << x <<
std::endl;
}
```

- La méthode **get()** est un **slot** mais aussi une méthode comme une autre : ici elle affiche la valeur x reçue.
- **SIGNAL** et **SLOT** sont des macros pour la phase de précompilation.

```
#include <QApplication>
#include "sender.h"
#include "reciever.h"

int main( int argc, char **argv ) {
   QApplication a( argc, argv );
   Reciever r; SenderA sa; SenderB sb;
   QObject::connect( &sa, SIGNAL(send(int)), &r, SLOT(get(int)) );
   QObject::connect( &sb, SIGNAL(transmit(int)), &r, SLOT(get(int)) );
   sa.doSend();
   sb.doStuff();
   return 0;
}
```

```
[tv@alias signaux-slots-1]$ ./sisl
Recieved: 7
Recieved: 5
[tv@alias signaux-slots-1]$ _
```

Connexion des signaux aux slots

Exemple : signaux et slots (6/6)

- En respectant la convention Qt pour les slots, il est possible de connecter automatiquement les signaux aux slots en appelant **connectSlotsByName()**.

```
class Reciever : public Qobject {
 ...
 public slots:
   void on_Sender_send( int x );
   void on_Sender_transmit( int x );
};
```

```
int main( int argc, char **argv ) {
   QApplication a( argc, argv );
   Reciever r;
   SenderA *sa = new SenderA(&r);
   sa->setObjectName("Sender");
   SenderB *sb = new SenderB(&r);
   sb->setObjectName("Sender");
   QMetaObject::connectSlotsByName(&r);
   sa->doSend();
   sb->doStuff();
   return 0;
}
```

```
[tv@alias signaux-slots-2]$ ./sisl2
Recieved from Sender: 7
Recieved from Sender: 5
[tv@alias signaux-slots-2]$ _
```

Connexion automatique des signaux aux slots

Gestion mémoire (1/3)

- *Rappel : Qt est basé autour du modèle d'objet de Qt. Cette architecture est entièrement basée autour de la classe QObject et de l'outil moc. Les classes graphiques dérivent toutes de QObject. Les QObject s'organisent eux mêmes en arbre d'objets (parent-enfants). Quand on crée un QObject (un objet graphique par exemple) avec un autre QObject comme parent, l'objet parent ajoute le nouvel objet créé à sa liste d'enfants.*
- Conséquences :
  - Si le parent est détruit (appel de son destructeur), il **détruira automatiquement tous ses enfants**.
  - On ne détruira (opérateur **`delete`**) donc que les QObject créés par l'opérateur **`new`** qui n'ont pas de parent.

Gestion mémoire (2/3)

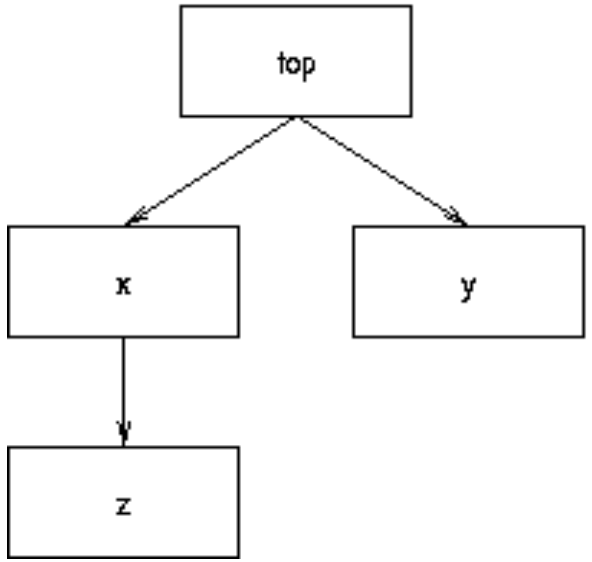
- Exemple : une classe **VerboseObject** qui hérite de la classe **QObject**

```
#include <QApplication>
#include <QDebug>

// Un objet "verbeux"
class VerboseObject : public QObject
{
 public:
  VerboseObject(QObject *parent=0, QString name=0):QObject(parent) {
     setObjectName( name );
     qDebug() << "Created: " << objectName();
  }
  ~VerboseObject() {
     qDebug() << "Deleted: " << objectName();
  }
  void doStuff() {
     qDebug() << "Do stuff " << objectName();
  }
};
```

Gestion mémoire (3/3)

- On instancie quelques objets « verbeux » :

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

  // Création des  instances
  VerboseObject top( 0, "top" );
  VerboseObject *x = new VerboseObject( &top, "x" );
  VerboseObject *y = new VerboseObject( &top, "y" );
  VerboseObject *z = new VerboseObject( x, "z" );

  top.doStuff();
  x->doStuff();
  y->doStuff();
  z->doStuff();

  return 0;
}
```



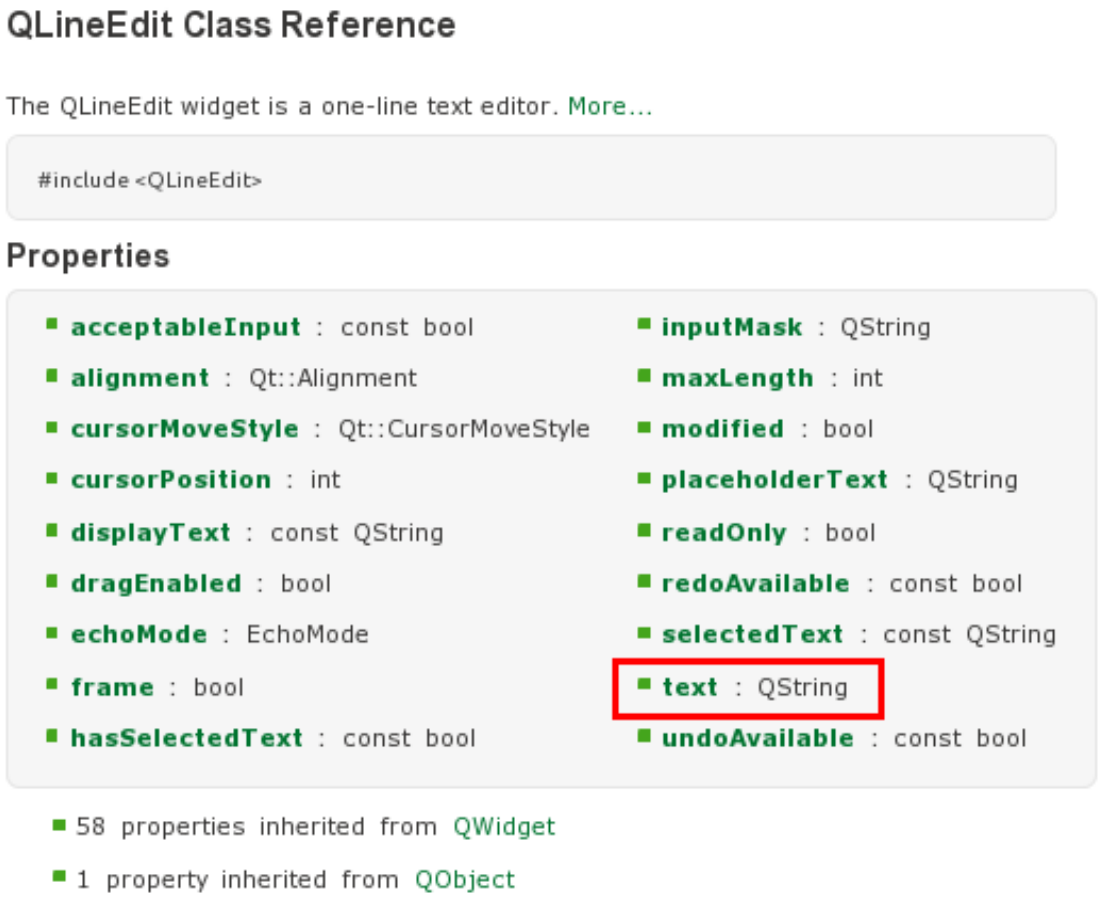

```
[tv@alias memoire]$ ./memoire
Created:  "top"
Created:  "x"
Created:  "y"
Created:  "z"
Do stuff  "top"
Do stuff  "x"
Do stuff  "y"
Do stuff  "z"
Deleted:  "top"
Deleted:  "x"
Deleted:  "z"
Deleted:  "y"
[tv@alias memoire]$ _
```

Propriétés (1/5)

- Un objet Qt peut avoir des **propriétés**.
- Toutes les propriétés sont des attributs de la classe que l'on peut lire et éventuellement modifier.
- Qt suit cette convention pour le nom des accesseurs :

`propriete()` : c'est la méthode qui permet de lire la propriété

`setPropriete()` : c'est la méthode qui permet de modifier la propriété



QLineEdit Class Reference

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

#include <QLineEdit>

Properties

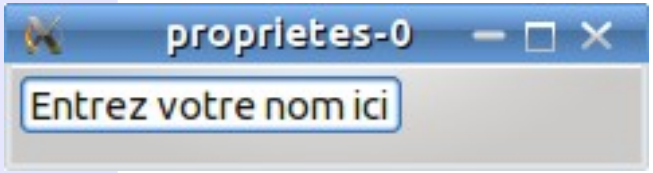
- acceptableInput : const bool
- alignment : Qt::Alignment
- cursorMoveStyle : Qt::CursorMoveStyle
- cursorPosition : int
- displayText : const QString
- dragEnabled : bool
- echoMode : EchoMode
- frame : bool
- hasSelectedText : const bool
- inputMask : QString
- maxLength : int
- modified : bool
- placeholderText : QString
- readOnly : bool
- redoAvailable : const bool
- selectedText : const QString
- text : QString
- undoAvailable : const bool

- 58 properties inherited from QWidget
- 1 property inherited from QObject

Propriétés (2/5)

- Exemple : comme indiqué dans la documentation, la propriété **text** d'un QLineEdit est de type QString et on peut la lire avec la méthode **text()**. Pour modifier le texte présent dans le QLineEdit, on utilisera la méthode **setText()** :

```
#include <QApplication>
#include <QString>
#include <QLineEdit>
#include <QWidget>

int main(int argc, char **argv) {
   QApplication app(argc, argv);
   QWidget box;
   QLineEdit lineEdit("Contenu du lineEdit", &box);

   QString contenu = lineEdit.text();

   lineEdit.setText("Entrez votre nom ici");

   box.show();
   return app.exec();
}
```



Propriétés (3/5)

```
#ifndef BALLON_H
#define BALLON_H
#include <QObject>
#include <QVariant>

class Ballon : public QObject {
   Q_OBJECT
   Q_PROPERTY( Couleur couleur READ couleur WRITE setCouleur )
   Q_ENUMS( Couleur )
   Q_PROPERTY( Taille taille READ taille )
   Q_ENUMS( Taille )

   public:
      Ballon( QObject *parent=0 );
      enum Couleur { Bleu, Blanc, Rouge };
      void setCouleur( Couleur c );
      Couleur couleur() const;
      enum Taille { Petit, Moyen, Grand };
      Taille taille() const;
   private:
      Couleur _couleur;
      Taille _taille;
};

#endif
```

Propriétés (4/5)

```
#include "ballon.h"

Ballon::Ballon( QObject *parent/*=0*/ ) : QObject( parent )
{
   _taille = Petit;
   _couleur = Bleu;
}

void Ballon::setCouleur( Couleur c ) {
   _couleur = c;
}

Ballon::Couleur Ballon::couleur() const {
   return _couleur;
}

Ballon::Taille Ballon::taille() const {
   return _taille;
}
```

Propriétés (5/5)

```
#include <QApplication>
#include <QDebug>
#include "ballon.h"

int main( int argc, char **argv ) {
   QApplication a( argc, argv );
   QObject *ballon1 = new Ballon();

   qDebug() << "couleur = " << ballon1->property( "couleur" ).toString();
   ballon1->setProperty( "couleur", "Rouge" );
   qDebug() << "couleur = " << ballon1->property( "couleur" ).toString();
   ballon1->setProperty( "taille", "Moyen" );
   qDebug() << "taille = " << ballon1->property( "taille" ).toString();

   Ballon *ballon2 = new Ballon();
   qDebug() << "couleur = " << ballon2->couleur();
   ballon2->setCouleur(Ballon::Rouge);
   qDebug() << "couleur = " << ballon2->couleur();
   qDebug() << "taille = " << ballon2->taille();
   return 0;
}
```

```
[tv@alias proprietes-2]$ ./proprietes-2
couleur =  "0"
couleur =  "2"
taille =  "0"
couleur =  0
couleur =  2
taille =  0
[tv@alias proprietes-2]$ _
```

Les *layouts* (1/5)

- Qt fournit un système de disposition (*layout*) pour l'organisation et le positionnement automatique des widgets enfants dans un widget. Ce gestionnaire de placement permet l'agencement facile et le bon usage de l'espace disponible.
- Qt inclut un ensemble de classes **QxxxLayout** qui sont utilisés pour décrire la façon dont les widgets sont disposés dans l'interface utilisateur d'une application.
- Toutes les sous-classes de QWidget peuvent utiliser les *layouts* pour gérer leurs enfants. **QWidget::setLayout()** applique une mise en page à un widget.
- Lorsqu'un *layout* est défini sur un widget de cette manière, il prend en charge les tâches suivantes :
  - Positionnement des widgets enfants
  - Gestion des tailles (minimale, préférée)
  - Redimensionnement
  - Mise à jour automatique lorsque le contenu change

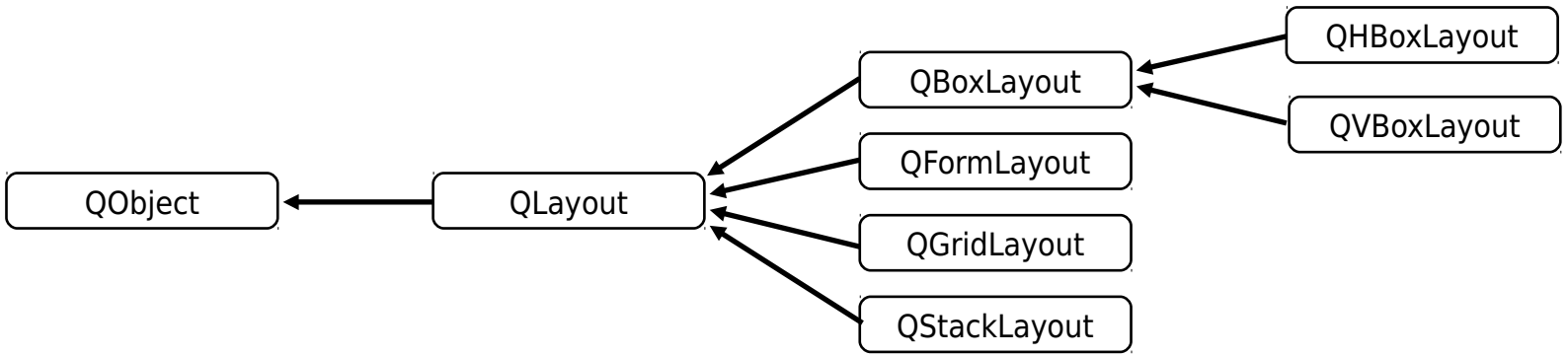
Les *layouts* (2/5)

- D'une manière générale, les widgets sont hiérarchiquement inclus les uns dans les autres. Le principal avantage est que si le parent est déplacé, les enfants le sont aussi.
- Les gestionnaires de disposition (les classes Layout) simplifient ce travail :
  - On peut ajouter des widgets dans un layout

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

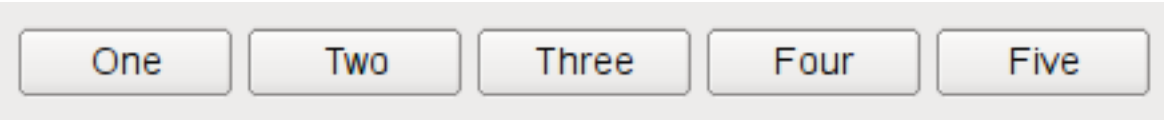
  - On peut associer un layout à un widget qui devient alors le propriétaire du layout et parent des widgets inclus dans le layout

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

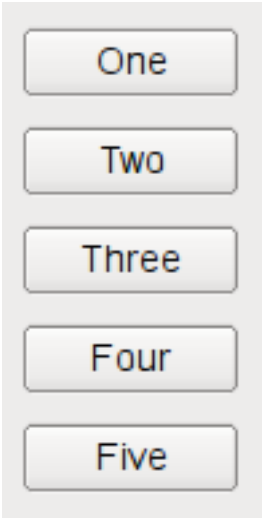


Les *layouts* (3/5)

- Quelques layouts : horizontal, vertical, grille, formulaire ...



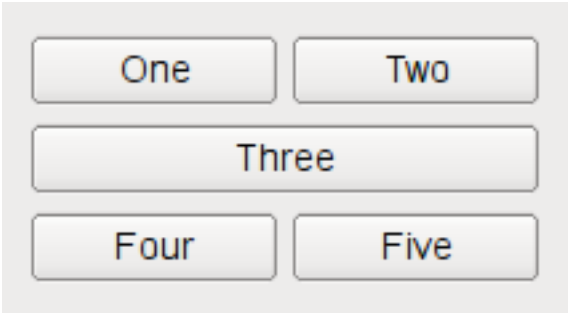
QHBoxLayout



QVBoxLayout

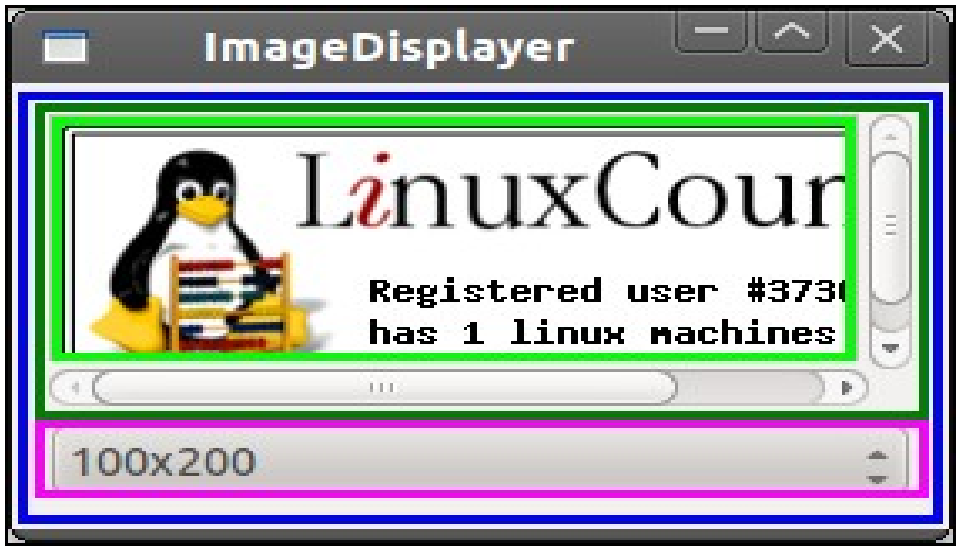
QFormLayout

QGridLayout

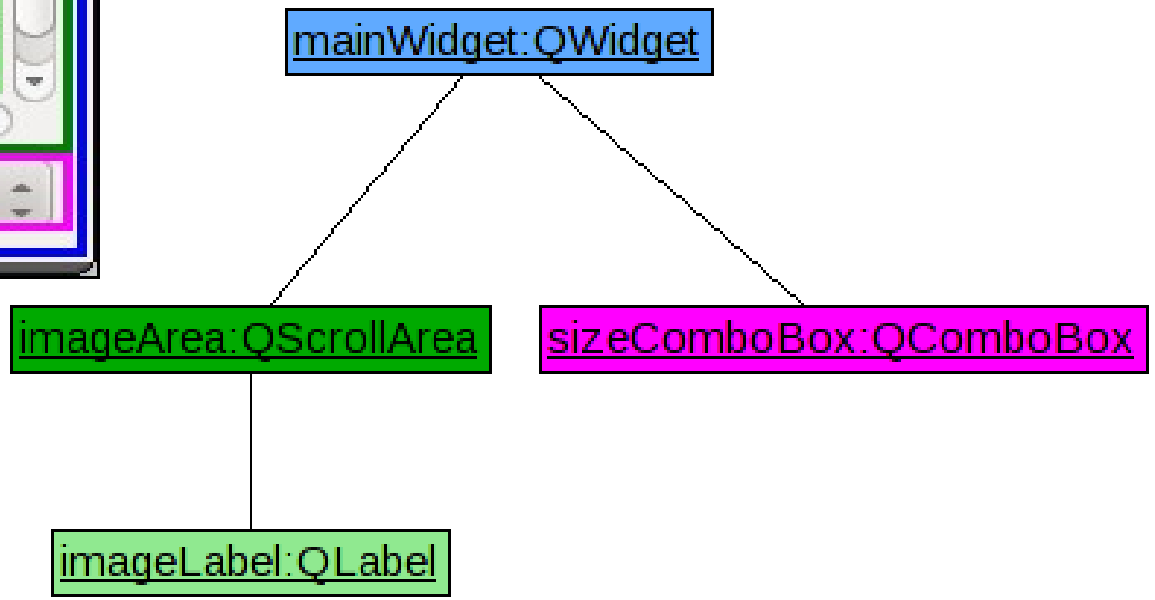


Les *layouts* (4/5)

- Exemple : on utilise le gestionnaire de disposition **`QVBoxLayout`** que l'on associe au widget parent (**`mainWidget`**). La classe **QScrollArea** fournit une zone de défilement utilisée pour afficher le contenu d'un widget enfant dans un cadre.



Hierarchie des objets :



Les *layouts* (5/5)

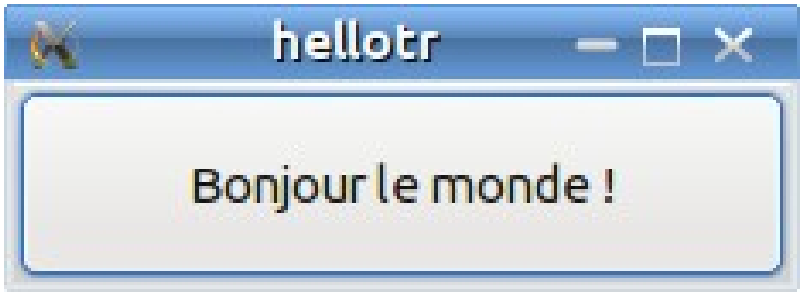
- La destruction de mainWidget entraîne la destruction successive de sizeComboBox, imageArea et de imageLabel.

```
int main(int argc, char *argv[]) {
   QApplication a(argc, argv);
   QLabel * imageLabel = new QLabel;
   imageLabel->setPixmap(QPixmap("373058.png"));
   QScrollArea * imageArea = new QScrollArea;
   imageArea->setWidget(imageLabel);
   // imageArea parent de imageLabel
   QComboBox * sizeComboBox = new QComboBox;
   sizeComboBox->addItem("100x200");
   sizeComboBox->addItem("200x400");
   QVBoxLayout * layout = new QVBoxLayout;
   layout->addWidget(imageArea);
   layout->addWidget(sizeComboBox);
   QWidget mainWidget;
   mainWidget.setLayout(layout);
   // mainWidget parent de imageArea et de sizeComboBox
   mainWidget.show();
   return a.exec();
}
```

Internationalisation (1/6)

- Qt intègre son propre système de traduction. La prise en charge de plusieurs langues est extrêmement simple dans des applications Qt et ajoute peu de surcharge de travail pour le programmeur.

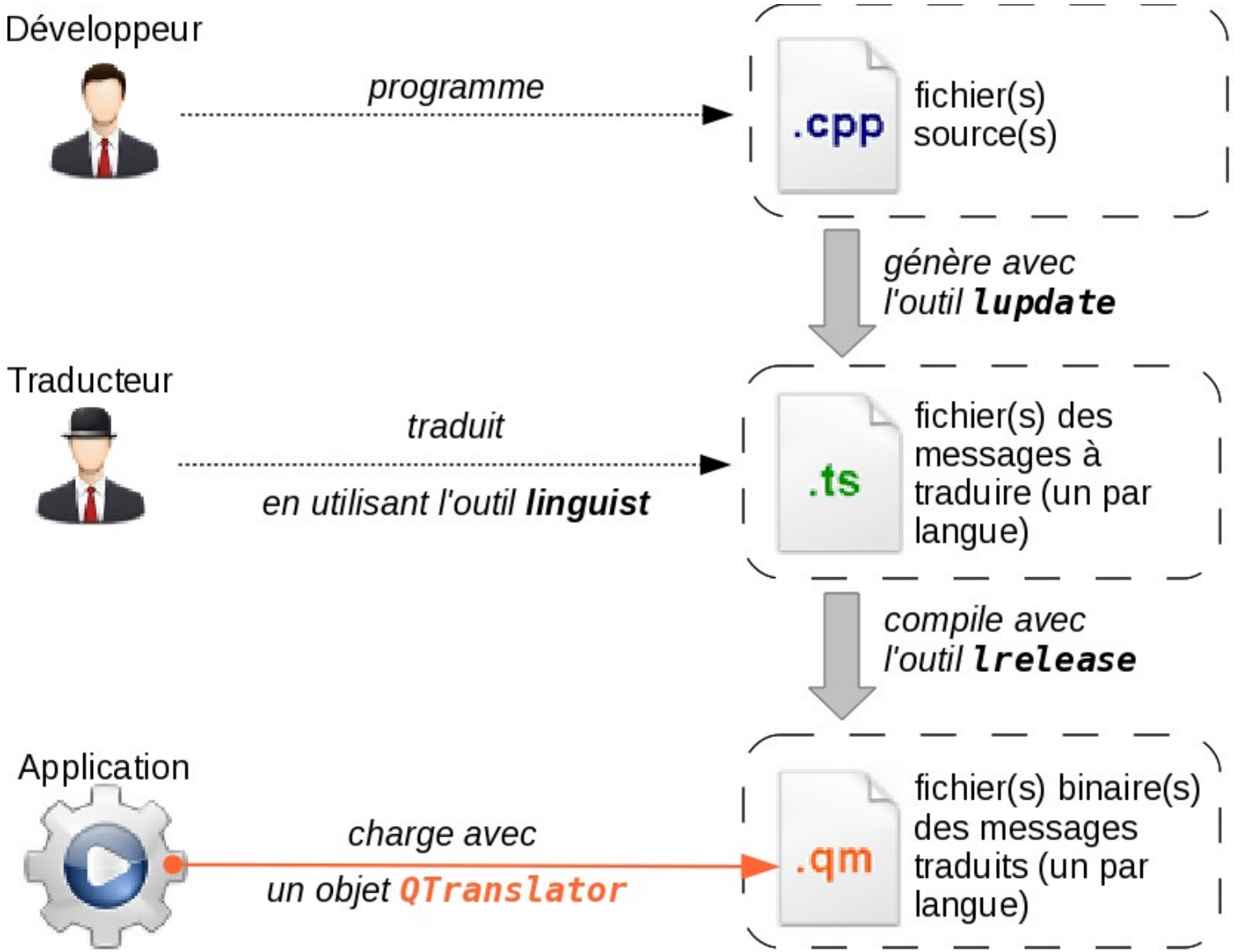
- Selon le manuel de **Qt Linguist**, l'internationalisation est assurée par la collaboration de trois types de personnes : les développeurs, le chef de projet et les traducteurs.



Internationalisation (2/6)

- Dans leur code source, les **développeurs** entrent des chaînes de caractères dans leur propre langue. Ils doivent permettre la traduction de ces chaînes grâce à la méthode **`tr()`**. En cas d'ambiguïté sur le sens d'une expression, ils peuvent également indiquer des commentaires destinés à aider les traducteurs.
- Le **chef de projet** déclare les fichiers de traduction (un pour chaque langue) dans le fichier de projet (.pro). L'utilitaire **`lupdate`** parcourt les sources à la recherche de chaînes à traduire et synchronise les fichiers de traduction avec les sources. Les fichiers de traductions sont des fichiers XML portant l'extension **.ts**.
- Les **traducteurs** utilisent **Qt Linguist** pour renseigner les fichiers de traduction. Quand les traductions sont finies, le chef de projet peut compiler les fichiers .ts à l'aide de l'utilitaire **`lrelease`** qui génère des fichiers binaires portant l'extension **.qm**, exploitables par le programme. Ces fichiers sont lus à l'exécution et les chaînes de caractères qui y sont trouvées remplacent celles qui ont été écrites par les développeurs.

Internationalisation (3/6)



Internationalisation (4/6)

- Le **développeur**

hellotr.cpp

```
QPushButton hello(QPushButton::tr("Hello world!"));
```

- Le **chef de projet**

hellotr.pro

```
SOURCES      = hellotr.cpp
TRANSLATIONS = hellotr_fr.ts
```

```
$ lupdate -verbose hellotr.pro
```

- Le **traducteur**

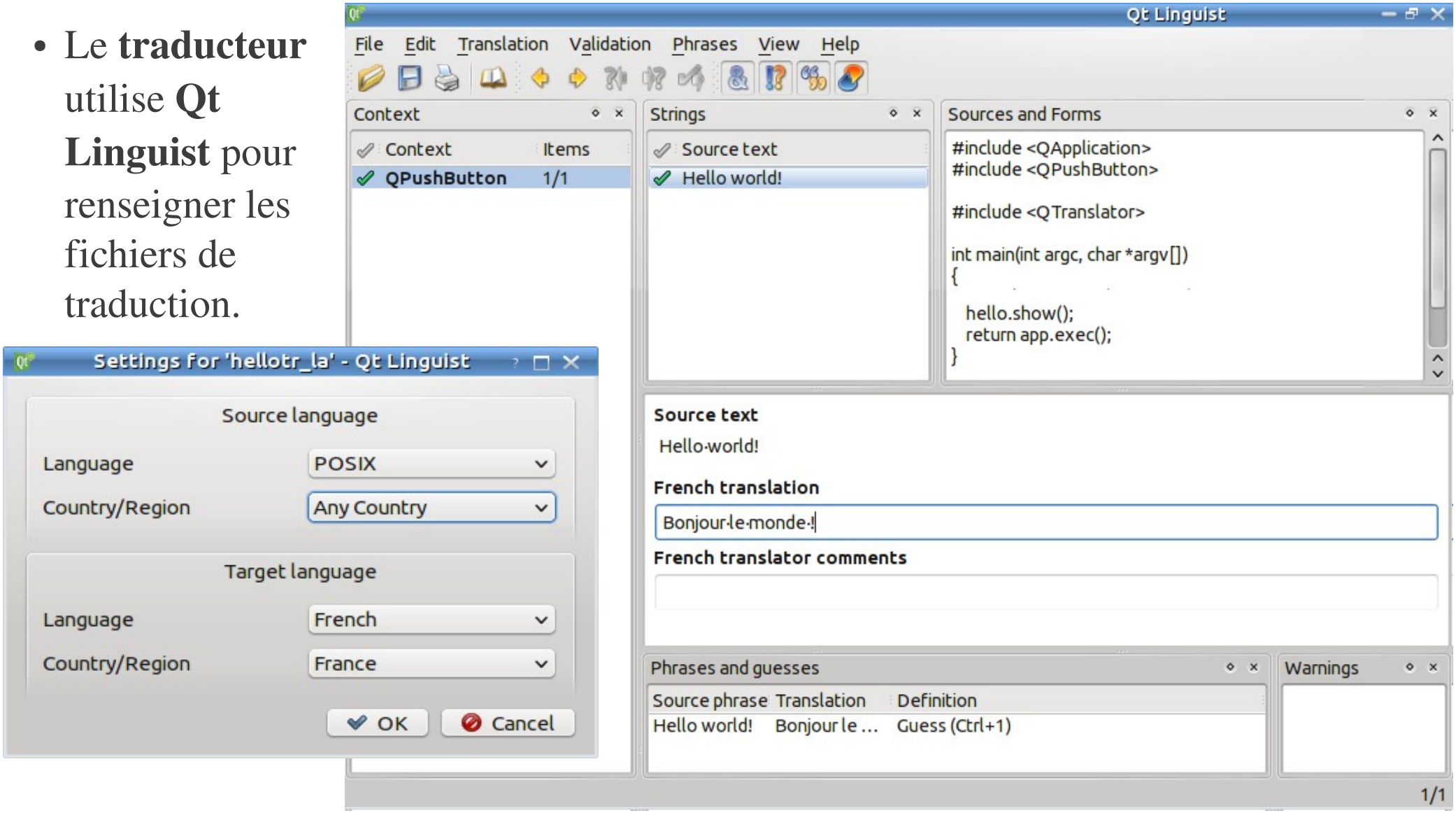
```
$ linguist hellotr_fr.ts
```

- Le **chef de projet**

```
$ lrelease hellotr_fr.ts
```

Internationalisation (5/6)

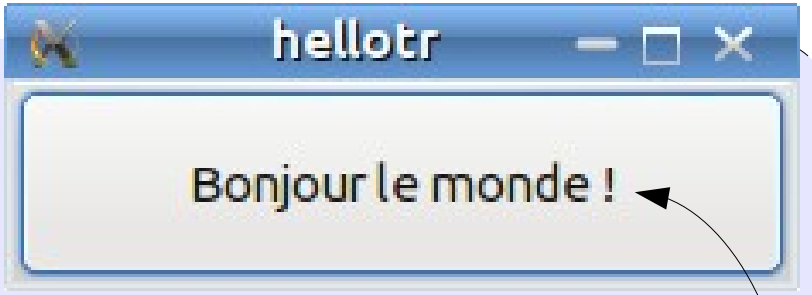
- Le **traducteur** utilise **Qt Linguist** pour renseigner les fichiers de traduction.

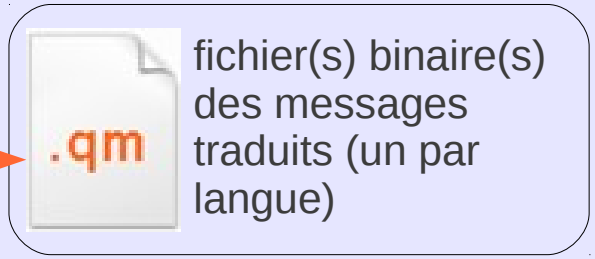


Internationalisation (6/6)

- Exemple :

```
#include <QApplication>
#include <QPushButton>
#include <QTranslator>

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

 QTranslator translator;
 translator.load("hello_fr");
 app.installTranslator(&translator);

 QPushButton hello(QPushButton::tr("Hello world!"));

 hello.resize(100, 30);

 hello.show();
 return app.exec();
}
```





/ Qt indexe chaque chaîne traduisible dans le contexte de traduction qui est généralement le nom de la sous-classe QObject utilisé. Un contexte de traduction est défini pour les nouvelles classes héritant de QObject et en utilisant la macro Q_OBJECT. */*

Documentation (http://doc.qt.nokia.com/)

- Tutoriel : http://doc.trolltech.com/4.7/tutorials.html

- La référence Qt4 : http://doc.trolltech.com/4.7/index.html, et plus particulièrement les concepts de base (http://doc.trolltech.com/4.7/qt-basic-concepts.html) :
  - Signals and Slots ;
  - Main Classes ;
  - Main Window Architecture ;
  - Internationalization (pensez à écrire votre programme en anglais puis à le traduire en français) ;
  - OpenGL Module.

- Les documentations des outils indispensables :
  - qmake (http://doc.trolltech.com/4.7/qmake-tutorial.html) ;
  - Qt Designer (http://doc.trolltech.com/4.7/designer-manual.html) ;
  - Qt Linguist (http://doc.trolltech.com/4.7/linguist-manual.html) ;
  - Qt Creator (http://doc.qt.nokia.com/qtcreator-2.0/index.html).

Documentation

- Quelques autres sources d'informations méritant le détour :
  - Qt Centre : http://www.qtcentre.org/
  - Qt Forum (en anglais) : http://www.qtforum.org/
  - QtFr, un site français, http://www.qtfr.org/
  - L'Independant Qt Tutorial qui est plein d'exemples (et disponible en français) : http://www.digitalfanatics.org/projects/qt_tutorial/
  - Qt-Apps.org propose un annuaire de programmes libres construits sur Qt : http://www.qt-apps.org/
  - Et Qt-Prop.org, l'équivalent pour les programmes non-libres (comme Skype et GoogleEarth) : http://www.qt-prop.org/

- **Liste des Tps : http://tvaira.free.fr/info1/**