Architecture Modèle / Vue

- Présentation
- Pattern MVC
- Architecture Modèle/Vue
- Les modèles
- Les vues
- Les délégués

Présentation

Qt 4 a introduit un nouvel ensemble de classes qui utilisent une architecture **modèle / vue (*model/view*)** afin de gérer la relation entre les données et la façon dont elles sont présentées à l'utilisateur.

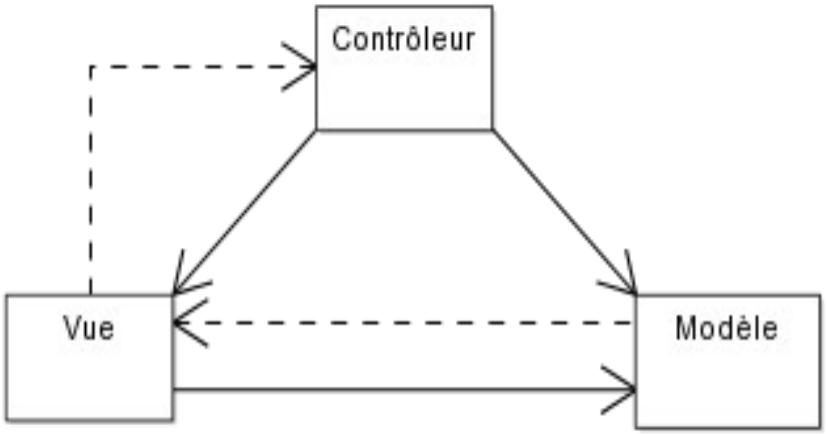
L'architecture modèle / vue (*model/view*) de Qt est inspiré du modèle de conception (*design pattern*) **MVC (Modèle-Vue-Contrôleur)** provenant de *Smalltalk* et qui est très souvent utilisé lors de la construction d'interfaces utilisateur (IHM).

Le MVC (*Model-View-Controller*) divise l'IHM en un modèle (**modèle de données**), une vue (**présentation des données**, interface utilisateur) et un contrôleur (**logique de contrôle**, gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface.

Le pattern MVC

Ce patron d'architecture impose la séparation entre les données, la présentation et les traitements, ce qui donne trois parties fondamentales dans l'application finale :

- le **modèle** contient les données. Il s'occupe du traitement et de l'interaction des données avec, par exemple, une base de données ;
- la **vue** est la représentation graphique.
  Elle correspond à la partie affichage des données, de ce que l'utilisateur peut voir ;
- le **contrôleur** prend en charge l'interaction avec l'utilisateur, recevant tous les événements déclenchés par l'utilisateur (clic, sélection...) et mettant par la suite à jour les données.



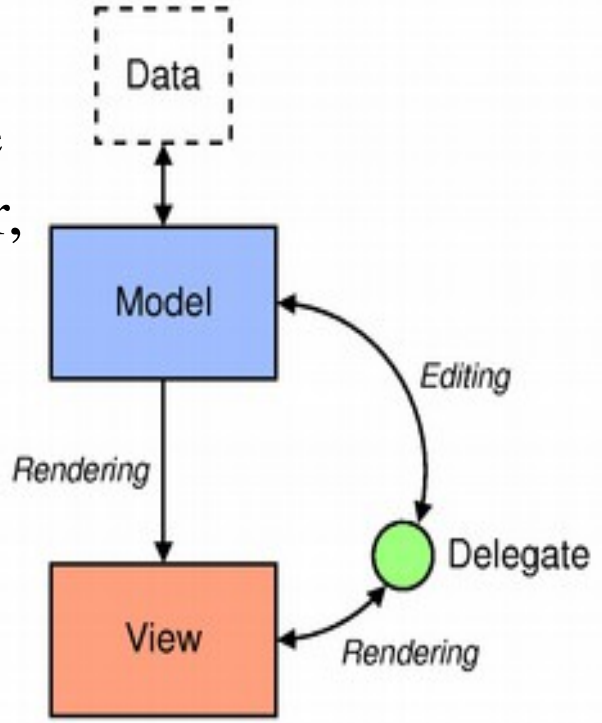
Cette architecture permet donc de séparer les différentes entités d'une application, aboutissant à une architecture flexible, claire et maintenable.

Architecture Modèle-Vue de Qt

Qt utilise en réalité une architecture **modèle-vue** (model-view) entourée d'un **délégué** (*delegate*).

Qt ne fournit pas de réel composant permettant de gérer les interactions avec l'utilisateur. De ce fait, celles-ci sont gérées par la vue elle-même.

Cependant, pour des raisons de flexibilité, l'interaction et les entrées utilisateurs ne sont non pas prises en compte par un composant totalement séparé, à savoir le contrôleur, mais par un composant « interne » à la vue : le délégué.



Ce composant est responsable de deux choses :

- personnaliser l'édition des éléments au moyen d'un éditeur ;
- personnaliser le rendu des éléments à l'intérieur d'une vue.

Les modèles (1/2)

Tous les modèles fournis par Qt sont basés sur la classe **`QAbstractItemModel`**. Cette classe est la plus abstraite et la plus haute dans la hiérarchie des classes des différents modèles. Elle fournit une interface que tous les modèles doivent respecter, afin d'être utilisés correctement avec une vue.

De base, Qt fournit un ensemble de modèles pouvant être directement utilisés comme :

- **`QStringListModel`** : stockage d'une liste de **`QString`** ;
- **`QFileSystemModel`** : un ensemble d'informations sur un fichier ou un répertoire du système de fichier local (anciennement QDirModel) ;
- **`QStandardItemModel`** : gestion de tout type de structure, complexe ou non ;
- **`QSqlTableModel`**, **`QSqlRelationalTableModel`** : accès à une base de données (SQLite, MySQL...).

Les modèles (2/2)

Si aucune de ces classes ne vous convient, vous devrez créer votre **propre modèle**.

Il suffit en effet de créer une classe dérivant d'une des classes suivantes :

- **`QAbstractItemModel`** : il s'agit de la classe la plus abstraite donc choix le plus flexible mais le plus complexe ;
- **`QAbstractListModel`** : classe abstraite fournissant une interface pour un modèle de type **liste** (associé à une **`QListView`**) ;
- **`QAbstractTableModel`** : classe abstraite fournissant une interface pour un modèle de type **tableau** (associé à une **`QTableView`**).

Remarque : il n'existe pas de classe séparée pour un modèle de type hiérarchique (**`QTreeView`**). Il s'agit simplement de **`QAbstractItemModel`**.

Les vues

Une fois que l'on possède un modèle (déjà existant ou personnalisé), on dispose de trois types de **vue** :

- **`QListView`** : **liste** d'éléments ;
- **`QTableView`** : **tableau** d'éléments ;
- **`QTreeView`** : représentation d'éléments sous forme hiérarchique (**arbre** d'éléments).

Une fois la vue choisie, le modèle y est affecté en utilisant la fonction :

```
void QAbstractItemView::setModel(QAbstractItemModel *model)
```

Les délégués

Tous les délégués fournis par Qt sont basés sur la classe **`QAbstractItemDelegate`**. Qt fournit deux délégués :

- **`QItemDelegate`** ;
- **`QStyledItemDelegate`** (utilise le style courant pour le rendu des données)

Remarques :

Les vues sont déjà dotées d'un délégué par défaut **`QStyledItemDelegate`**.

Qt fournit en outre **`QSqlRelationalDelegate`**, permettant d'éditer et afficher des données d'un **`QSqlRelationalTableModel`**.

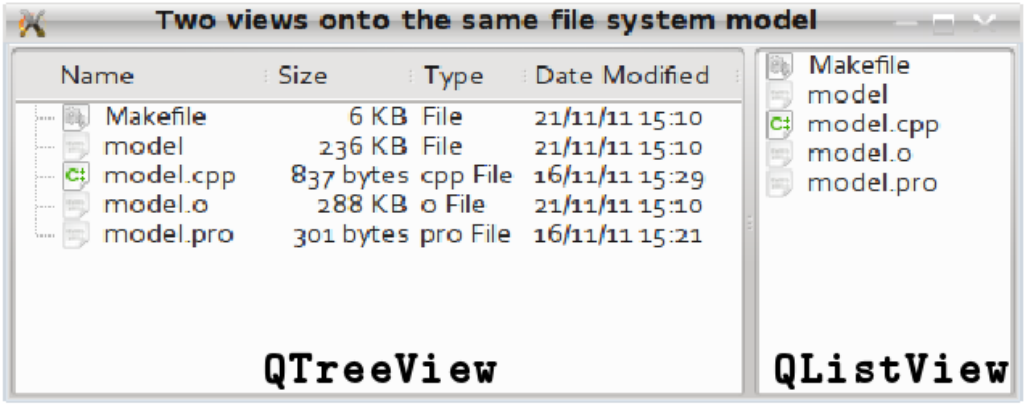
Cependant, même si les vues sont dotées d'un délégué par défaut, il est bien entendu possible de modifier et paramétrer celui-ci grâce à la fonction

`void QAbstractItemView::setItemDelegate(QAbstractItemDelegate * delegate)`

Exemple simple n°1

Un modèle (QFileSystemModel) pour plusieurs vues (QTreeView et QListView)

```
#include <QApplication>
#include <QFileSystemModel>
#include <QTreeView>
#include <QListView>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  QSplitter *splitter = new QSplitter;

  QFileSystemModel *model = new QFileSystemModel;
  model->setRootPath(QDir::currentPath());
  QModelIndex parentIndex =
    model->index(QDir::currentPath());

  QTreeView *tree = new QTreeView(splitter);
  tree->setModel(model);
  tree->setRootIndex(parentIndex);

  QListView *list = new QListView(splitter);
  list->setModel(model);
  list->setRootIndex(parentIndex);

  splitter->setWindowTitle("Two views onto
                the same file system model");
  splitter->show();
  return app.exec();
}
```



L'index de modèle

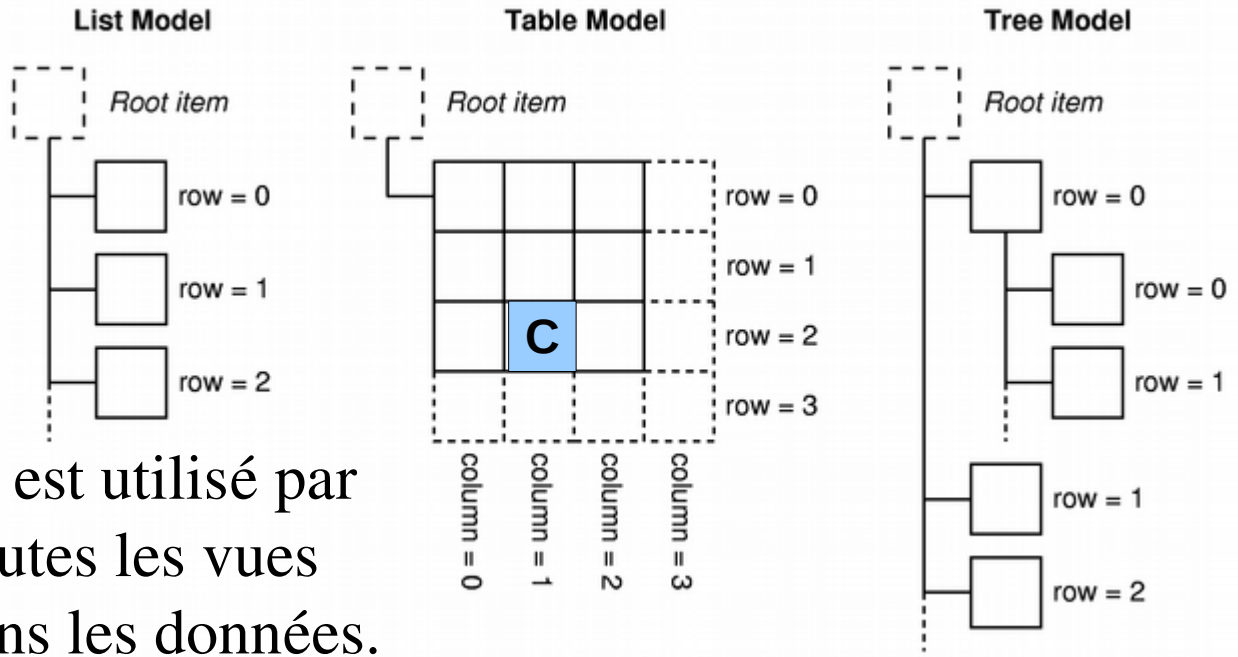
Afin de s'assurer que la représentation des données est séparée de la façon dont il est consulté, le concept d'un **index de modèle** est introduit.

Chaque élément d'information qui peuvent être obtenus via un modèle est représenté par un **index de modèle**. Les **Vues** (et les Délégués) utilisent ces **index** pour demander des éléments de données à afficher.

Concept de modèle

- Dans l'architecture modèle / vue, le modèle fournit une interface standard que les vues et les délégués utilisent pour accéder aux données.

- Dans Qt, l'interface standard est défini par la classe **`QAbstractItemModel`**. Peu importe comment les éléments de données sont stockés dans une structure sous-jacente des données, toutes les sous-classes de **`QAbstractItemModel`** représentent les données comme une structure hiérarchique contenant des tableaux d'objets.




```
QModelIndex indexC = model->index(2, 1,
QModelIndex());
```

- Le mécanisme signaux / slots est utilisé par les modèles pour notifier à toutes les vues attachées tout changement dans les données.

Mise en oeuvre du modèle/vue

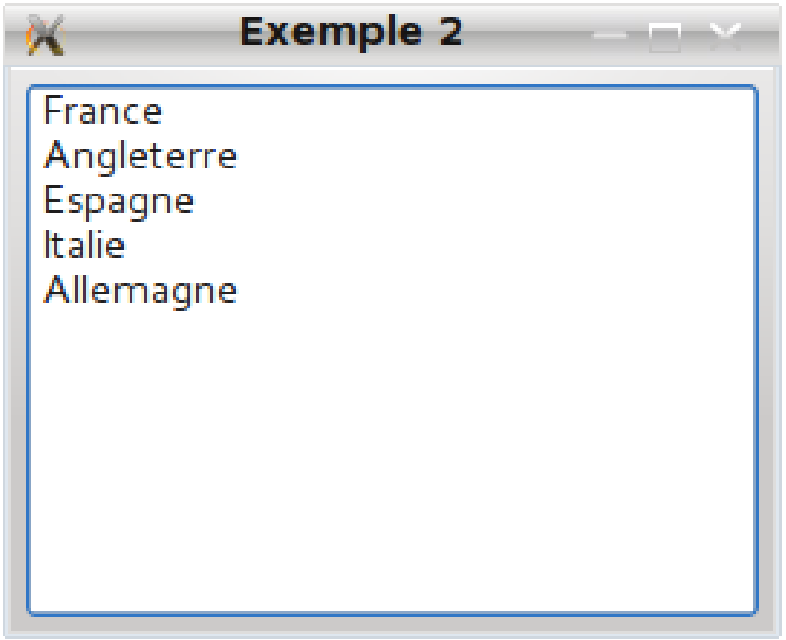
Lorsqu'on utilise l'architecture modèle/vue avec Qt, cela se passe toujours en 3 étapes :

- Créer le modèle
- Créer la vue
- Associer la vue et le modèle

Exemple n°2 : mise en oeuvre du modèle/vue

Un modèle (**QStringListModel**) pour une vue (**QListView**)

```
#include <QApplication>
#include <QStringListModel>
#include <QListView>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv); QMainWindow  mainWindow; QWidget resultatWindow;

  // Créer Les données
  QStringList listePays;
  listePays << "France" << "Angleterre"
    << "Espagne" << "Italie" << "Allemagne";

  // Créer le modèle
  QStringListModel *modele
   = new QStringListModel(listePays);

  // Créer la vue
  QListView *vueListe = new QListView;

  // Associer la vue et le modèle
  vueListe->setModel(modele);

  // Affichage
  ...
  mainWindow.show();
  return app.exec();
}
```



Créer son propre modèle

Il suffit en effet de créer une classe dérivant d'une des classes suivantes :

- **`QAbstractItemModel`** : comme évoqué plus haut, il s'agit de la classe la plus abstraite ;
- **`QAbstractListModel`** : classe abstraite fournissant une interface pour un modèle de type liste (associé à une `QListView`) ;
- **`QAbstractTableModel`** : classe abstraite fournissant une interface pour un modèle de type tableau (associé à une `QTableView`).

Une fois que l'on a choisi la classe de base convenant le mieux à notre utilisation, il ne reste plus qu'à redéfinir certaines méthodes virtuelles comme :

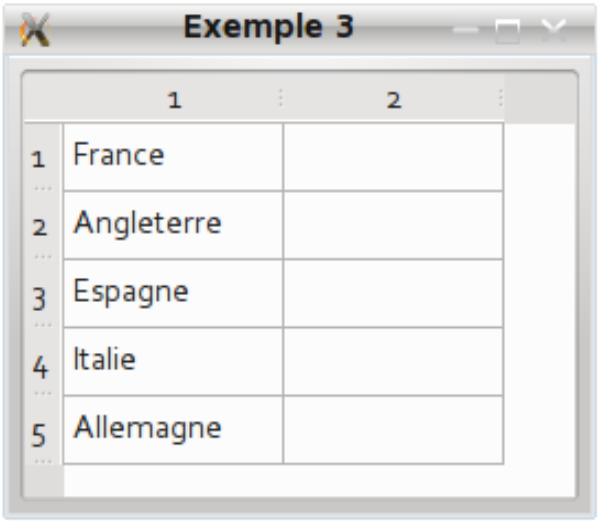
- La fonction retournant le nombre de lignes du modèle :
  ```
  int QAbstractItemModel::rowCount(const QModelIndex & parent = QModelIndex()) const;
  ```
- La fonction renvoyant la donnée associée au rôle role pour l'index index :
  ```
  QVariant QAbstractItemModel::data(const QModelIndex & index, int role = Qt::DisplayRole) const;
  ```

Exemple n°3 : modèle personnel

```
class PaysTableModel : public QabstractTableModel {
    private:
        QStringList listePays;
    public:
        PaysTableModel(const QStringList &listePays, QObject *parent = 0
):QAbstractTableModel(parent)
        {if (listePays.count() > 0) this->listePays = listePays;}
        int rowCount(const QModelIndex &parent = QModelIndex()) const
        { return this->listePays.count(); }
        int columnCount(const QModelIndex &parent = QModelIndex()) const
        { return 2; // pour insérer une (future) capitale }
        QVariant data(const QModelIndex &index, int role) const {
          if (! index.isValid())
            return QVariant();
          if (((unsigned int)index.row()>=this->listePays.count())||(index.column()>= 2))
            return QVariant();
          if ((role == Qt::DisplayRole)) {
            if(index.column() == 0) // la première colonne = pays
               return this->listePays.at(index.row()); // retourne le nom du pays demandé
           }
           else return QVariant();
        }
};
```

Exemple n°3 : suite

```
int main(int argc, char *argv[]) {
     QApplication app(argc, argv); QMainWindow  mainWindow; QWidget tableWidget;

     QStringList listePays;
     listePays << "France" << "Angleterre" << "Espagne" << "Italie" << "Allemagne";

     PaysTableModel *modele = new PaysTableModel(listePays);

     QTableView *vueTable = new QTableView;
     vueTable->setModel(modele);

     QHBoxLayout *layout = new QHBoxLayout;
     layout->addWidget(vueTable);
     tableWidget.setLayout(layout);
     mainWindow.setCentralWidget(&tableWidget);
     mainWindow.setWindowTitle("Exemple 3");
     mainWindow.show();
     return app.exec();
}
```



Remarques : Pour des modèles modifiables, il est aussi nécessaire de redéfinir la fonction **`bool QAbstractItemModel::setData(const QModelIndex & index, const QVariant & value, int role = Qt::EditRole)`**. Il serait aussi utile de redéfinir la méthode **`Qt::ItemFlags flags(const QModelIndex & index) const`** pour décider quelles parties de notre modèle seront éditables ou non.

Exemple 4 : modèle modifiable

```
class PaysTableModel : public QAbstractTableModel {
 private:
  QStringList listePays; QStringList listeCapitales;
  enum Colonnes { Pays = 0, Capitale = 1, NbColonnes = Capitale + 1};
 public:
  PaysTableModel(const QStringList &listePays, QStringList &listeCapitales, QObject
*parent = 0):QAbstractTableModel(parent)
  { if (listePays.count() > 0) this->listePays = listePays;
    if (listeCapitales.count() > 0) this->listeCapitales = listeCapitales; }
  int rowCount(const QModelIndex &parent = QModelIndex()) const
  { return parent.isValid() ? 0 : this->listePays.count(); }
  int columnCount(const QModelIndex &parent = QModelIndex()) const
  { return parent.isValid() ? 0 : NbColonnes; }
  QVariant data(const QModelIndex &index, int role) const {
   if (! index.isValid()) return QVariant() ;
   if(((unsigned int)index.row()>=this->listePays.count())||(index.column()>=NbColonnes))
      return QVariant();
   if ((role == Qt::DisplayRole || role == Qt::EditRole)) {
      if(index.column() == Pays)   return this->listePays.at(index.row());
      else if(index.column() == Capitale) return this->listeCapitales.at(index.row());
   }
   return QVariant();
  }
  ...
```

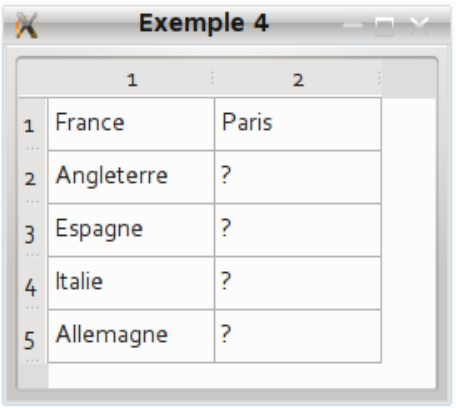
Les items d'un modèle peuvent jouer des **rôles** différents pour d'autres composants, permettant de fournir différents types de données pour des situations différentes. Par exemple, **Qt::DisplayRole** est utilisé pour accéder à une chaîne qui peut être affichée comme du texte dans une vue.

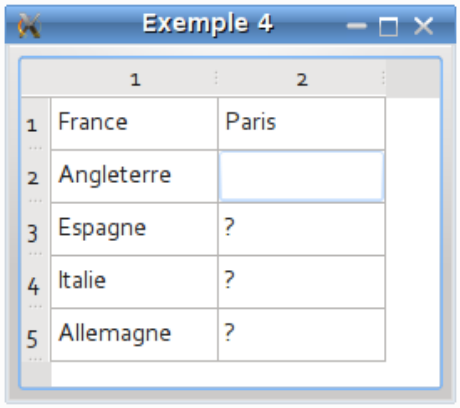
Exemple 4 : suite

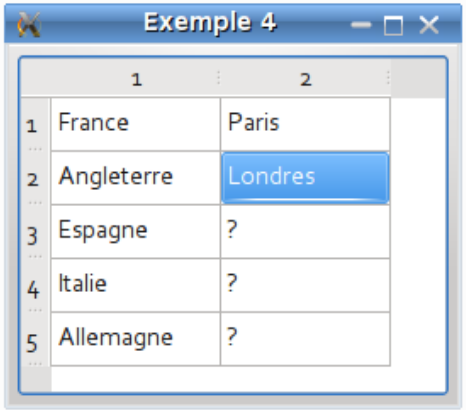
```
class PaysTableModel : public QAbstractTableModel {

 public:
  ...
  bool setData(const QModelIndex &index, const QVariant &value, int rôle) {
    if (index.isValid() && role == Qt::EditRole) {
      this->listeCapitales.replace(index.row(), value.toString());
      emit(dataChanged(index, index));
      return true;
    }
    return false;
  }
  Qt::ItemFlags flags(const QModelIndex &index) const {
    if (! index.isValid())
      return Qt::ItemIsEnabled;
    if (index.column() == Pays) // Un pays seulement selectionnable {
      return Qt::ItemIsEnabled | Qt::ItemIsSelectable;
    }
    else if (index.column() == Capitale) // Une capitale sera éditable {
      return Qt::ItemIsEnabled | Qt::ItemIsSelectable | Qt::ItemIsEditable;
    }
    return QAbstractTableModel::flags(index);
  }
};
```

Exemple 4 : fin

```
int main(int argc, char *argv[]) {
     QApplication app(argc, argv); QMainWindow mainWindow; QWidget tableWidget;

     QStringList listePays;
     listePays << "France" << "Angleterre" << "Espagne" << "Italie" << "Allemagne";

     QStringList listeCapitales;
     listeCapitales << "Paris" << "?" << "?" << "?" << "?";

     PaysTableModel *modele = new PaysTableModel(listePays,
                                                 listeCapitales);

     QTableView *vueTable = new QTableView ;

     vueTable->setModel(modele);

     ...
     mainWindow.show();

     return app.exec();
}
```





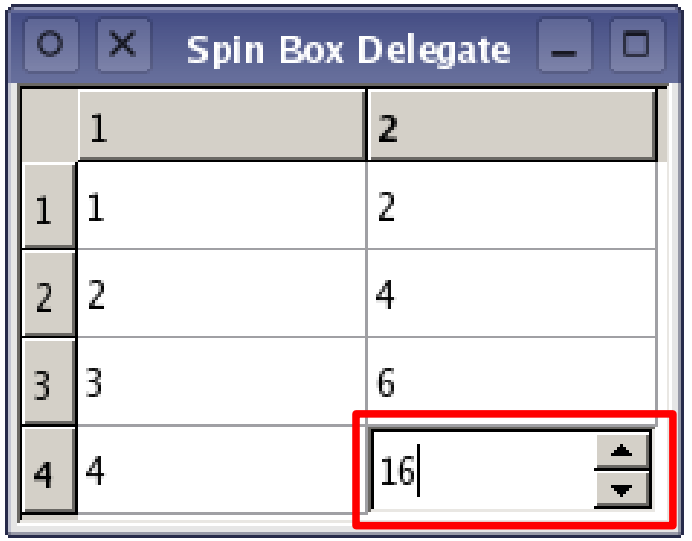


Responsabilité du délégué

Le délégué est responsable de deux choses :

- personnaliser l'édition des éléments au moyen d'un éditeur ;
- personnaliser le rendu des éléments à l'intérieur d'une vue.

Ainsi, grâce à un **délégué**, il vous est possible de personnaliser la manière dont les entrées utilisateurs seront gérées, ainsi que le rendu des éléments.



L'exemple est décrit dans la documentation Qt :

http://doc.qt.nokia.com/latest/model-view-programming.html#a-simple-delegate

Et le code source : http://doc.qt.nokia.com/latest/itemviews-spinboxdelegate.html

Créer son propre délégué (1/2)

Depuis Qt 4.4, il est recommandé d'utiliser **`QStyledItemDelegate`**. On l'utilisera donc comme classe de base de nos délégués :

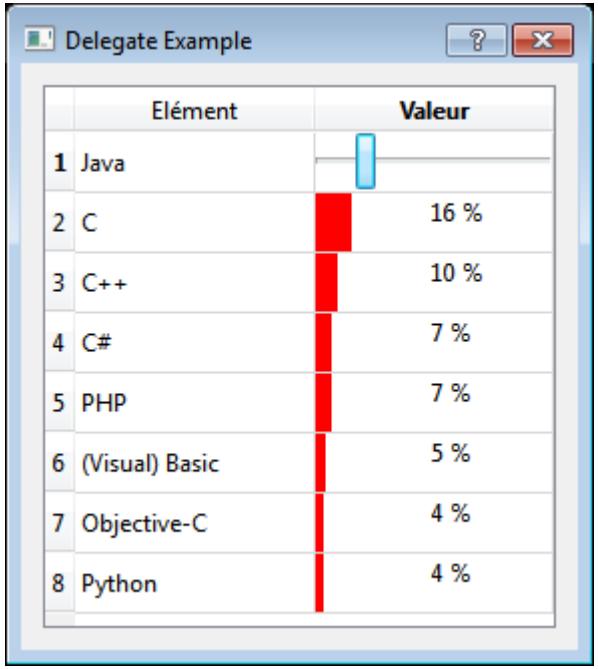
```
class MyDelegate : public QStyledItemDelegate { … };
```

Lorsque l'on souhaite uniquement personnaliser l'édition des éléments dans une vue et non le rendu, on doit redéfinir quatre méthodes :

- **`createEditor()`** : retourne le *widget* (éditeur) pour éditer l'item se trouvant à l'index index.
- **`setEditorData()`** : permet de transmettre à l'éditeur editor les données à afficher à partir du modèle se trouvant à l'index index.
- **`setModelData()`** : permet de récupérer les données de l'éditeur et de les stocker à l'intérieur du modèle, à l'index identifié par le paramètre index.
- **`updateEditorGeometry()`** : permet de redimensionner l'éditeur à la bonne taille lorsque la taille de la vue change

Créer son propre délégué (2/2)

```
class TableDelegate : public QStyledItemDelegate
{   Q_OBJECT
  public:
    TableDelegate(QObject *parent = 0);
    QWidget *createEditor(QWidget *parent, const
QStyleOptionViewItem &option, const QModelIndex &index)
const;
    void setEditorData(QWidget *editor, const QModelIndex
&index) const;
    void setModelData(QWidget *editor, QAbstractItemModel
*model, const QModelIndex &index) const;

    void paint(QPainter *painter, const QStyleOptionViewItem
&option, const QModelIndex &index) const;
    void updateEditorGeometry(QWidget *editor, const
QStyleOptionViewItem &option, const QModelIndex &index)
const;
};
```



L'exemple complet est disponible sur le site :
http://qt.developpez.com/tutoriels/mvc/apostille-delegates-mvd/

Documentation

- **La référence Qt4 : http://doc.qt.nokia.com/latest/model-view-programming.html**
- Un article sur les délégués : http://qt.developpez.com/tutoriels/mvc/apostille-delegates-mvd/
- MVC : http://fr.wikipedia.org/wiki/Mod%C3%A8le-Vue-Contr%C3%B4leur
- 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/lp_sil/tp_qt/