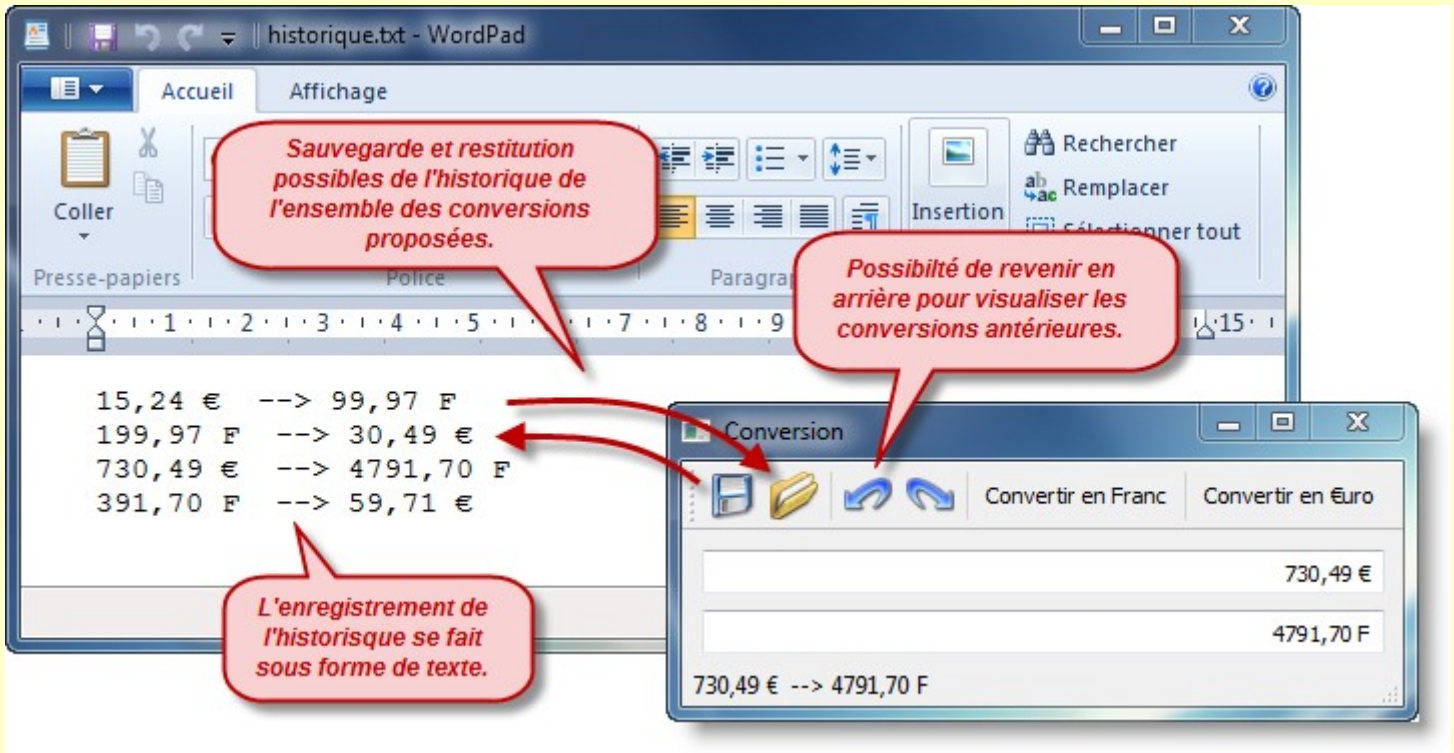
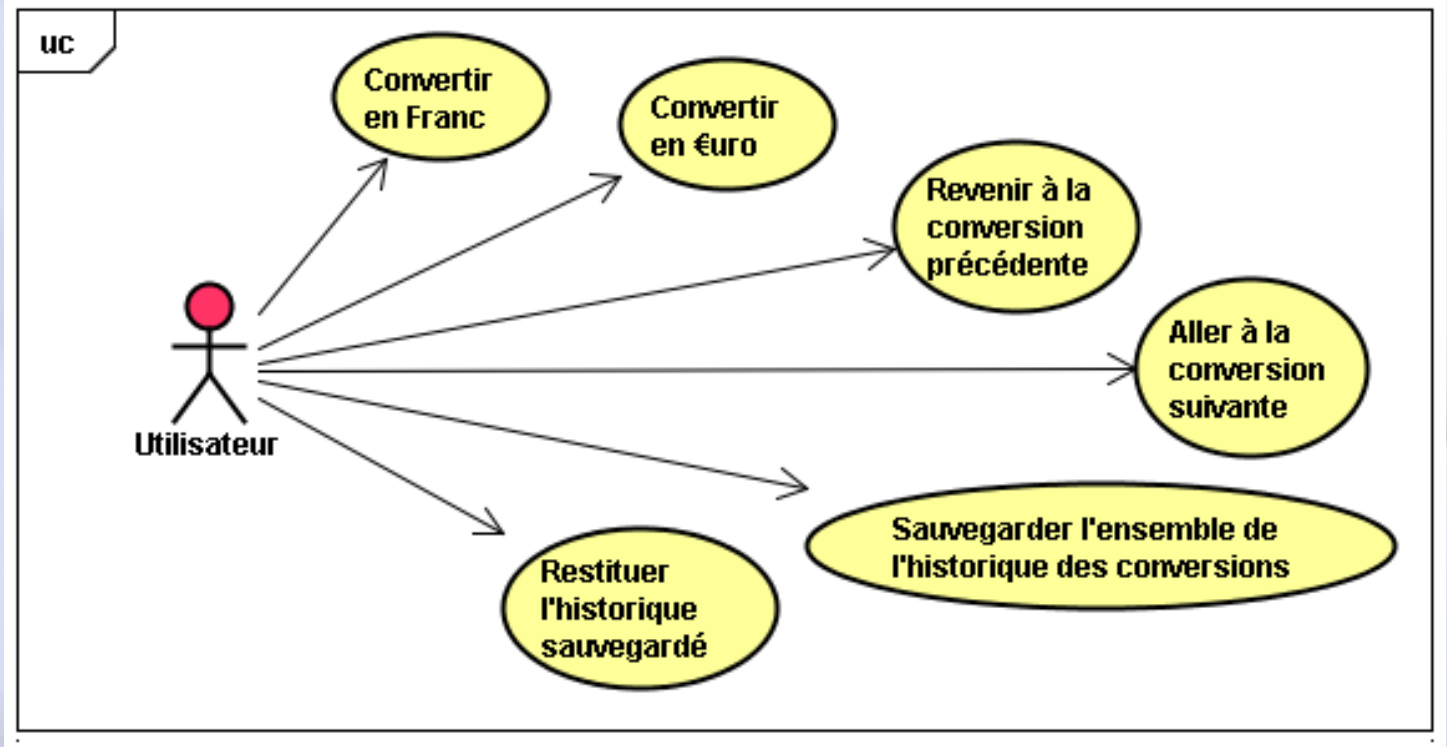
Lors de cette étude, nous allons nous intéresser plus particulièrement sur la création et la lecture de fichiers. Dans le même temps, nous aurons besoin de connaître la notion de flux. Nous allons évaluer toutes ces techniques au travers de deux projets différents.

PREMIER PROJET – MISE EN PLACE DES FLUX AU TRAVERS DE FICHIERS

Pour le premier, nous reprenons l'étude sur la conversion monétaire à travers laquelle nous rajoutons un historique qui recense tous les calculs effectués depuis le début du programme. Il sera ainsi possible de revenir sur une conversion déjà réalisée et de parcourir l'ensemble des calculs. Cet historique pourra être sauvegardé sous la forme d'un fichier texte qu'il sera ensuite possible de restituer.



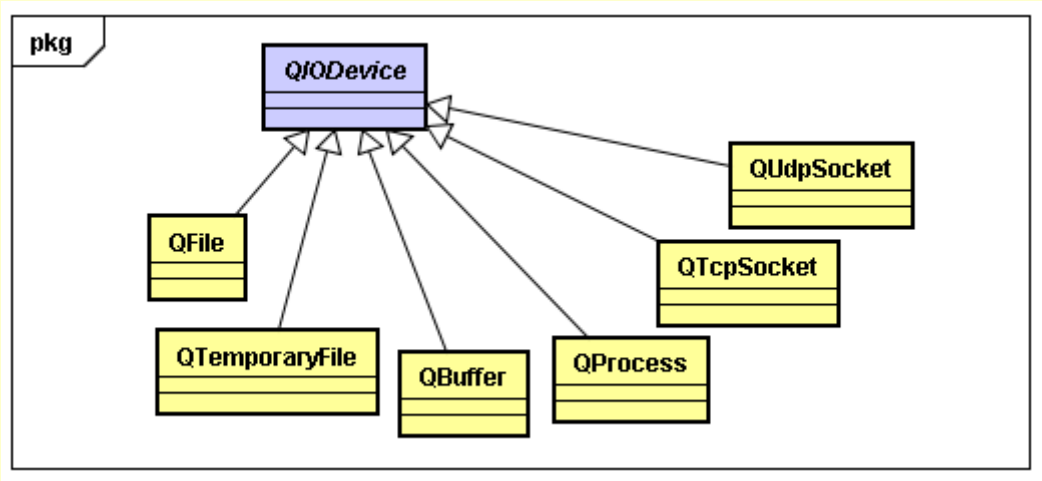
MODÉLISATION – DIAGRAMME DE CAS D'UTILISATION



LES FLUX ET LES FICHIERS

Le besoin d'effectuer des opérations de lecture ou d'écriture dans des fichiers ou sur un autre support est commun à presque toutes les applications.

*Qt fournit un excellent support pour ces opérations par le biais de la classe abstraite **QIODevice** qui encapsule des « périphériques différents » capables de lire et écrire une série d'octets. Nous disposons d'un certain nombre de classes concrètes qui hérite de cette classe abstraite **QIODevice** correspondant aux périphériques à prendre en compte.*



- **QFile : Accède aux fichiers du disque dur local ainsi que les ressources intégrées à l'application ou à d'autres périphériques de mémoire de masse, comme les clés USB, les disques optiques, etc.**
- **QTemporaryFile : Crée et accède à des fichiers temporaires du système de fichiers local.**
- **QBuffer : Effectue des opérations de lecture et d'écriture de données dans un QByteArray.**
- **QProcess : Exécute des programmes externes à l'application et gère la communication inter-processus. Cette classe permet ainsi de lancer des programmes externes et de communiquer avec ceux-ci par le biais de leurs canaux d'entrée, de sortie et d'erreur standard (cin, cout et cerr).**
- **QTcpSocket : Transfère un flux de données sur le réseau par le biais du protocole TCP.**
- **QUdpSocket : Envoi ou reçoit des datagrammes UDP sur le réseau.**

__QProcess__, __QTcpSocket__ et __QUdpSocket__ sont des classes séquentielles, ce qui implique un accès unique aux données, en commençant par le premier octet et en progressant dans l'ordre jusqu'au dernier octet. Cela correspond à un flux d'information.

__QFile__, __QTemporaryFile__ et __QBuffer__ sont des classes à accès aléatoire. Les octets peuvent donc être lus plusieurs fois à partir de n'importe quel emplacement. La méthode __seek()__ permet de repositionner le pointeur de fichier pour lire le prochain octet.

__QFile__ facilite l'accès aux fichiers individuels, qu'ils soient dans le système de fichier ou intégré dans l'exécutable de l'application en tant que ressources. Pour les applications ayant besoin d'identifier des jeux complets de fichiers sur lesquels travailler, __Qt__ fournit les classes __QDir__ et __QFileInfo__ qui, respectivement, gèrent les répertoires et fournissent des informations concernant leurs fichiers.

En plus de ces classes de périphérique, **Qt** fournit deux classes de flux de niveau plus élevé qui exécutent des opérations de lecture et d'écriture sur tous les périphériques d'entrée/sortie :

- **QDataStream : pour les données de type primitif.**
- **QTextStream : pour le texte.**

Ces classes gèrent les problèmes tels que le classement des octets et le codage de texte, de sorte que les applications __Qt__ s'exécutant sur d'autres plate-formes ou dans d'autres pays puissent effectuer des opérations de lecture et d'écriture sur leurs fichiers respectifs.

LA CLASSE QFILE

Un objet de la classe **QFile** représente un fichier spécifique du système de fichier local quelque soit la plate-forme utilisée. Vous spécifiez le nom du fichier désiré au moment de la construction. Par la suite, pour exploiter le contenu du fichier, vous devez l'ouvrir en spécifiant le mode d'ouverture requis, lecture ou écriture, tout ceci au moyen de la méthode ***open()***. La page suivante nous donne une liste plus exhaustive des méthodes qui vous seront bien souvent utiles :

- **QFile(nom du fichier) : construit un objet représentant le fichier spécifié en argument. Si vous spécifier un fichier au travers d'un répertoire, vous utilisez le séparateur « / » quelque soit le système de fichier, même pour Windows. ATTENTION, le symbole « \ » n'est pas du tout supporté.**
- **copy(nom du fichier source, nom du fichier destination) : copie un fichier et en crée un autre. Si l'opération s'est bien déroulée, la fonction retourne true, sinon false dans le cas contraire.**
- **exists() : permet de savoir si le fichier représenté par QFile existe vraiment.**
- **fileName() - setFileName(nom du fichier) : permet de connaître le nom du fichier représenté par QFile. Il est également possible de représenter un autre fichier au moyen du même objet QFile.**
- **open(mode d'ouverture) : ouvre réellement le fichier représenté par QFile. Nous pouvons ouvrir le fichier en lecture seule (QIODevice::ReadOnly), en écriture seule (QIODevice::WriteOnly), en lecture et écriture (QIODevice::ReadWrite). Vous pouvez rajoutez des spécifications supplémentaires pour indiquer par exemple que le fichier doit être un fichier texte au moyen de la constante suivante (QIODevice::Text).**
- **permissions() – setPermissions(permissions) : Il est possible de connaître ou de régler les permissions accordées pour chaque fichier. Ces permissions sont de la même nature que celles que vous rencontrez sous les systèmes Unix. La classe QFile possède en interne une énumération dénommée Permission qui propose l'ensemble des constantes correspondant à ce système de fichier. Nous retrouvons les quatre types d'utilisateurs, respectivement : propriétaire, Utilisateur, Groupe et Autre.**
- **enum Permission {ReadOwner, WriteOwner, ExeOwner, ReadUser, WriteUser, ExeUser, ReadGroup, WriteGroup, ExeGroup, ReadOther, WriteOther, ExeOther};**
- **remove() : supprime définitivement le fichier en cours et renvoie true si l'opération s'est déroulée correctement.**
- **rename(nouveau nom) : permet de changer le nom du fichier.**
- **close() : clôture le fichier si ce dernier est ouvert. Permet ainsi d'éviter de perdre des données. Ceci-dit, cette méthode est automatiquement appelée lorsque l'objet QFile est détruit, notamment lorsque nous sortons de la portée de la déclaration de l'objet.**
- **size() : retourne la taille du fichier en octets.**

LA CLASSE QDIR

De même que la classe **QFile** représente les fichiers, la classe **QDir** est spécialisée pour représenter les répertoires du système de fichier, ceci quelque soit la plate-forme utilisée. Cette classe est capable de travailler également au travers des ressources incluses dans l'application.

*Là aussi, **Qt** utilise le séparateur universel « / » que **Qt** transforme automatiquement pour se conformer au séparateur qui existe dans système de fichier de la plate-forme locale. **QDir** peut aussi bien prendre en compte le chemin complet du répertoire (positionnement absolu) ou se référencer à partir du répertoire courant (positionnement relatif au répertoire « home »).*

```
QDir("/home/user/Documents") // positionnement absolu
QDir("C:/Documents and Settings")

QDir("C:/Documents and Settings") // positionnement relatif
```

- **QDir(nom du répertoire) : construit un objet représentant le répertoire spécifié en argument. Suivant l'écriture proposée, il peut s'agir d'un répertoire absolu ou relatif (voir juste haut-dessus).**
- **absoluteFilePath(nom du fichier) : retourne le nom du chemin complet d'un fichier dont le nom est passé en argument en rapport avec le répertoire représenté par l'objet QDir.**
- **absolutePath() : retourne le nom du chemin complet du répertoire en cours.**
- **canonicalPath() : retourne le véritable nom du chemin complet du répertoire en cours, notamment si des liens symboliques sont proposés.**
- **cd(nom du nouveau répertoire) : permet de changer le répertoire courant à partir de celui où nous sommes.**
- **cdUp() : change également de répertoire, mais remonte d'un cran dans l'arborescence, équivalent à cd(« .. »).**
- **count() : renvoie le nombre total de répertoires et de fichiers dans le répertoire en cours.**
- **current() - setCurrent(chemin) : renvoie ou impose le répertoire courant (QDir) de l'application.**
- **currentPath() : renvoie le nom du répertoire courant en positionnement absolu de l'application.**
- **dirName() : renvoie uniquement le nom du répertoire sans prendre en compte tout le chemin.**
- **entryList(filtres de fichiers, types de filtre) : retourne la liste des fichiers et des répertoires contenus dans le répertoire en cours. Le premier argument est une liste de filtres de nom de fichiers. Ils peuvent contenir les caractères génériques « * » et « ? ». Nous pouvons réaliser un filtre qui ne prend en compte que les fichiers images par exemple. Le second argument spécifie le type d'entrée souhaité (fichiers normaux, répertoires, pas le répertoire courant et le répertoire parent, les fichiers cachés, etc.). Voici la liste exhaustive des types que nous pouvons prendre en compte.**
- **enum Filter {Dirs, AllDirs, Files, Drives, NoSymbLinks, NoDotAndDotDot, AllEntries, Readable, Writable, Executable, Modified, Hidden, System};**
- **entryList(types de filtre) : même méthode mais qui ne prend en compte le type d'entrée souhaité.**

- **entryInfoList(filtres de fichiers, types de filtre) :**
- **entryInfoList(types de filtre) : mêmes types de méthode que les précédentes avec l'avantage, cette fois-ci, de proposer directement une liste d'information de fichier sous la forme d'un QFileInfoList qui est en réalité un QList<QFileInfo>.**
- **exists(nom du fichier) : permet de savoir si le fichier passé en argument fait bien parti du répertoire en cours.**
- **exists() : permet de savoir si le répertoire représenté par QDir existe vraiment.**
- **filePath(nom du fichier) : renvoie le nom du chemin complet du fichier référencé en argument en rapport avec le répertoire en cours.**
- **filter() - setFilter(types de filtre) : renvoie la liste des types de filtre ou propose de nouveaux types.**
- **home() - homePath() : renvoie le répertoire courant de l'utilisateur, soit sous forme de QDir soit sous forme de chaîne.**
- **isAbsolute() - isReadable() - isRelative() - isRoot() : renseigne sur la qualité du répertoire en cours.**
- **makeAbsolute() : transforme le répertoire courant en entité absolue.**
- **makeDir(nom du sous-répertoire) : crée un sous-répertoire.**
- **makePath(nom du chemin) : crée un chemin supplémentaire.**
- **path() - setPath(nom du chemin) : retourne le chemin complet actuel ou en propose une nouvelle référence existante.**
- **refresh() : rafraîchi le contenu du répertoire actuel.**
- **relativeFilePath(nom du fichier) : renvoie le nom du chemin relatif suivant le nom du fichier proposé (ou le chemin) en rapport avec le chemin en cours représenté par QDir.**
- **remove(nom du fichier) : supprime définitivement le fichier spécifié dans le répertoire en cours et renvoie true si l'opération s'est déroulée correctement.**
- **rename(nom du fichier, nouveau nom) : permet de changer le nom du fichier spécifié dans le répertoire en cours.**
- **rmdir(nom du répertoire) - rmPath(nom du chemin) : supprime définitivement le répertoire ou le chemin spécifiés.**
- **root() - rootPath() : renvoie la racine (QDir) du système de fichier ou son nom.**
- **setNameFilter(filtres) : propose un filtre de fichiers, par exemple pour prendre en compte uniquement les fichiers images.**
- **sorting() - setSorting(ordre) : retourne ou propose l'ordre dans lequel les fichiers et les répertoires vont être récupérés dans la liste des entrées, par nom, par date, par taille, les répertoires d'abord, etc. Voici la liste des constantes prévues à cet effet ci-dessous.**
- **enum SortFlag {Name, Time, Size, Type, Unsorted, NoSort, DirsFirst, DirsLast, Reversed, IgnoreCase, LocaleAware};**

*Les opérateurs d'affectation, d'égalité et d'inégalité ont été redéfinis pour permettre le traitement entre deux objets **QDir**.*

LA CLASSE QFILEINFO

La classe **QFileInfo** nous permet d'accéder aux attributs d'un fichier, tels que la taille, les autorisations, le propriétaire et l'horodateur de ce fichier.

- **QFileInfo(nom du fichier) : construit un objet permettant de connaître les attributs du fichier donné en argument. Il est possible de préciser votre fichier en relatif ou au travers d'un chemin complet.**
- **QFileInfo(nom du répertoire, nom du fichier) : construit un objet permettant de connaître les attributs du fichier donné en argument dans le répertoire également précisé en argument.**
- **absoluteDir() : retourne le nom du chemin complet du fichier sous forme de QDir.**
- **absoluteFilePath() : même chose mais sous forme de chaîne de caractères.**
- **absolutePath() : même chose mais sans le nom du fichier.**
- **baseName() : retourne cette fois-ci uniquement le nom du fichier et surtout sans l'extension du fichier.**
- **canonicalPath() : retourne le véritable nom du chemin complet du fichier, notamment si des liens symboliques sont proposés.**
- **completeBaseName() : retourne uniquement le nom du fichier avec cette fois-ci son extension.**
- **completeSuffix() : propose uniquement l'extension complète du fichier.**
- **created() : renvoie la date et l'heure du fichier correspondant au moment de sa création.**
- **dir() : renvoie le répertoire complet où se situe le fichier concerné.**
- **exists() : permet de savoir si le fichier existe vraiment.**
- **fileName() : renvoie le nom du fichier complet sans le chemin.**
- **filePath() : renvoie le nom du fichier avec le chemin.**
- **group() : renvoie le nom du groupe auquel appartient ce fichier.**
- **isAbsolute() - isDir() - isExecutable() - isFile() - isHidden() - isReadable() - isRelative() - isRoot() - isSymLink() - isWritable() : renseigne sur la qualité du fichier en cours.**
- **lastModified() : retourne la date et l'heure de la dernière modification.**
- **lastRead() : renseigne sur l'instant de la dernière lecture effectuée.**
- **owner() : renseigne sur le propriétaire du fichier.**

- **path() : retourne le chemin complet actuel.**
- **permissions() : retourne les permissions accordées sur ce fichier.**
- **refresh() : rafraîchi le contenu du fichier actuel.**
- **setFile(nouveau fichier) : change de fichier de référence.**
- **size(): retourne la taille du fichier.**
- **suffix(): retourne l'extension du fichier.**
- **symLinkTarget(): retourne la position réelle du fichier représenté par le lien symbolique.**

LIRE ET ÉCRIRE DES DONNÉES BINAIRES

La façon la plus simple de charger et d'enregistrer des données binaires avec **Qt** consiste à prendre un objet de type **QFile**, à ouvrir le fichier et à y accéder par le biais d'un objet **QDataStream**. Ce dernier fournit un format de stockage indépendant de la plate-forme, qui supporte les types C++ de base tels que les **int** et **double**, de nombreux types de données **Qt**, dont **QByteArray**, **QFont**, **QImage**, **QPixmap** et **QString** ainsi que des classes conteneur telles que **QList<T>**.

*Un **QByteArray** est un simple tableau d'octets représenté sous la forme d'un décompte d'octets, suivi des octets eux-mêmes.*

- **A titre d'exemple, je vous propose de stocker dans un même fichier de données, le nom du fichier représentant la photo, la photo elle-même, sa largeur, sa hauteur et son ratio :**

```
QImage photo;
int largeur, hauteur;
double ratio;
QString nomFichier;
QFile fichier("exemple.image");
if (fichier.open(QIODevice::WriteOnly)) {
  QDataStream stockage(&fichier);
  stockage << nomFichier << photo << largeur << hauteur << ratio;
}
```

*Si le fichier s'ouvre avec succès, nous créons un flux **QDataStream** dans lequel nous plaçons successivement l'ensemble des informations de natures totalement différentes à l'aide du simple opérateur que nous connaissons bien « **<<** ». Ces informations sont bien enregistrées sous forme de suite d'octets et finalement stockées dans le fichier correspondant.*

- **L'intérêt des flux ici, c'est que nous travaillons directement avec des données de très haut niveau, ainsi qu'avec les types primitifs du langage C++, qui sont traduit automatiquement en une suite d'octets qui est par contre parfaitement adapté à un enregistrement dans un fichier ou pour être envoyée au travers d'un réseau.**
- **Pour s'y retrouver, les flux utilisent pour chaque variables, un en-tête de décompte d'octets qui sera bien utile par la suite pour permettre la lecture correcte des entités enregistrées.**
- **Ce flux d'octets peut naturellement être correctement interprété si nous utilisons la même version de flux, donc ici de nouveau un QDataStream, et surtout si nous lisons les données à récupérer dans le même ordre que lors de l'enregistrement. Utilisez, par contre, cette fois-ci l'opérateur « >> ».**

```
QImage photo;
int largeur, hauteur;
double ratio;
QString nomFichier;
QFile fichier("exemple.image");
if (fichier.open(QIODevice::ReadOnly)) {
  QDataStream stockage(&fichier);
  stockage >> nomFichier >> photo >> largeur >> hauteur >> ratio;
}
```

*QDataStream peut aussi être utilisé pour lire et écrire des octets bruts, sans en-tête de décompte d'octets, au moyen des méthodes **readRawBytes()** et **writeRawBytes()**.*

- **Lorsque nous utilisons QDataStream, Qt se charge de lire et d'écrire chaque type, dont les conteneurs avec un nombre arbitraire d'éléments. Cette caractéristique nous évite de structurer ce que nous écrivons et d'appliquer une conversion à ce que nous lisons. Notre seule obligation consiste à nous assurer que nous lisons tous les types dans leur ordre d'écriture, en laissant à Qt le soin de gérer tous les détails.**

LECTURE ET ÉCRITURE COMPLÈTE DE DONNÉES

Si nous souhaitons lire ou écrire dans un fichier en une seule fois, nous pouvons éviter l'utilisation de **QDataStream** et recourir à la place aux méthodes **write()** et **readAll()** de **QIODevice**.

```
bool copieFichier(const QString &source, const QString &destination)
{
  QFile fichierSource(source);
  if (fichierSource.open(QIODevice::ReadOnly))
  {
    QFile fichierDestination(destination);
    if (fichierDestination.open(QIODevice::WriteOnly))
      fichierDestination.write(fichierSource.readAll());
  }
  return fichierSource.error()==QFile::NoError && fichierDestination.error()==QFile::NoError;
}
```

*Sur la ligne de l'appel de **readAll()**, le contenu entier du fichier d'entrée est lu et placé dans un **QByteArray**. Il est alors transmis à la fonction **write()** pour être écrit dans le fichier de sortie. Le fait d'avoir toutes les données dans un **QByteArray** nécessite plus de mémoire que de les lire élément par élément, mais offre quelques avantages. Nous pouvons exécuter, par exemple, **qCompress()** et **qUncompress()** pour les compresser et les décompresser.*

LIRE ET ÉCRIRE DU TEXTE

Les formats de fichiers binaires sont généralement plus compacts que ceux basés sur le texte, mais ils ne sont pas lisibles ou modifiables par l'homme. Si vous désirez pouvoir consulter les données avec un simple éditeur, il est possible d'utiliser à la place les formats texte. **Qt** fournit la classe **QTextStream** pour lire et écrire des fichiers de texte brut, mais également d'autres formats de texte comme le HTML ou le XML ainsi que du code source.

***QTextStream** se charge automatiquement de la conversion entre Unicode et le codage local prévu par le système de fichier en cours, et gère de façon transparente les conventions de fin de ligne utilisées par les différents systèmes d'exploitation (« **\r\n** » sur Windows et « **\n** » sur Unix et Mac OS).*

*En plus des caractères et des chaînes, **QTextStream** prend en charge les types numériques de base du C++, qu'il convertit alors automatiquement en chaînes.*

Par exemple le code suivant écrit « Astérix le Gaulois, 47 pages à 8,45€ » :

```
QFile fichier("livre.txt");
double prix = 8.45;
if (fichier.open(QIODevice::WriteOnly)) {
  QTextStream sortie(&fichier);
  sortie << "Astérix le Gaulois, " << 47 << " pages à " << prix << "€" << endl;
}
```

L'écriture du texte est très facile, mais sa lecture peut représenter un véritable défit, car les données textuelles, contrairement aux données binaires écrites au moyen de **QDataStream**, sont fondamentalement ambiguës.

*Par exemple, si **sortie** est un **QTextStream** :*

sortie << « Astérix le » << « Gaulois »;

*Les données véritablement écrites dans **sortie** sont la chaîne **« Astérix le Gaulois »**. Rien ne permet de différencier une simple chaîne de deux chaînes consécutives. Ainsi si **entree** est également un **QTextStream** et si **chaine1** et **chaine2** sont deux chaînes e caractères :*

entrée >> chaine1 >> chaine2;

***chaine1** reçoit alors toute la chaîne **« Astérix le Gaulois »** alors que **chaine2** n'obtient rien du tout. Ce problème ne se pose pas avec un **QDataStream** puisque, comme nous l'avons vu, la longueur de chaque chaîne est stockée devant les données.*

Afin d'éviter ce genre d'inconvénient, il serait peut-être souhaitable d'enregistrer le texte ligne par ligne au moyen du terminateur **endl** et de faire de même lors de la lecture au moyen de la méthode **readLine()** de **QTextStream** qui récupère à chaque occurrence une seule ligne du texte enregistré.

*Il est également possible de traiter le texte en entier. Nous pouvons ainsi lire le fichier complet en une seule fois à l'aide de la méthode **readAll()** de **QTextStream**, si nous ne nous préoccupons pas bien sûr de l'utilisation de la mémoire, ou si nous savons que le fichier est petit.*

Comme **QDataStream**, **QTextStream** agit sur une sous-classe de **QIODevice**, qui peut être un **QFile**, un **QTemporaryFile**, un **QBuffer**, un **QProcess**, un **QTcpSocket** ou un **QUdpSocket**. En outre, il peut être utilisé directement sur un **QString**.

Par exemple, le code suivant permet de formater un message en prenant en compte simplement les valeurs numériques de différentes natures :

```
int largeur, hauteur;
double ratio;
QFileInfo fichier(nomFichier);
QString message;
QTextStream flux(&message);
flux << fichier.fileName() << " (" << largeur << ", " << hauteur << ") ratio = " << ratio;
```

*La classe **QString** possède des méthodes très utiles pour récupérer les valeurs numériques inscrites à l'intérieur d'une chaîne de caractères. Il existe effectivement une méthode qui transforme la chaîne en son équivalent numérique : **toInt()**, **toDouble()**, **toFloat()**, **toLong()**, etc.*

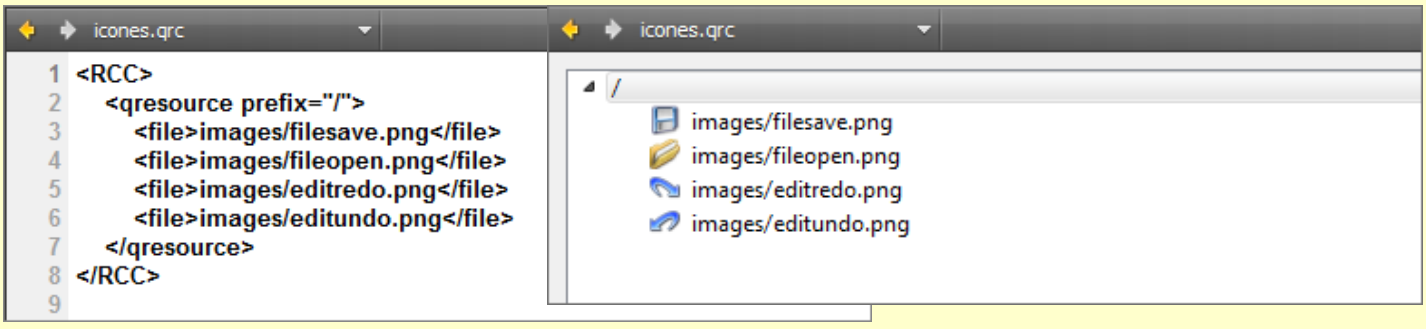
*Autre méthode intéressante de la classe **QString**, la méthode **split()** qui retourne une liste de chaînes de caractères qui est scindée à l'emplacement d'apparition du séparateur qui lui est fournit. Si nous choisissons par exemple le séparateur **« espace »**, nous récupérons ainsi chaque mot élémentaire de la phrase (ou de la ligne). Cela peut servir également pour séparer les valeurs numériques des autres mots dans la chaîne de caractères.*

✗ INTÉGRATION DES RESSOURCES

Jusqu'à présent, nous avons étudié l'accès aux données dans des périphériques externes, mais avec **Qt**, il est également possible d'intégrer du texte ou des données binaires dans l'exécutable de l'application. Pour ce faire, il convient d'utiliser le système de ressources de **Qt**.

- ***Il est possible d'intégrer tout type de fichier. C'est ce qu'il faut absolument faire pour les fichiers images si vous désirez que vos icônes soient toujours visibles lorsque vous déployez votre application sur un autre système.***
- ***Les fichiers intégrés peuvent être lus au moyen de QFile, exactement comme les fichiers normaux d'un système de fichiers.***

*Une ressource est un fichier XML, dont l'extension est **« *.qrc »** qui répertorie les fichiers à intégrer dans l'exécutable. Les ressources sont ensuite converties en code C++ par l'utilitaire standard intégré **rcc**, le compilateur de ressources de **Qt**. Depuis l'application, les ressources sont identifiées par le préfixe de chemin, ici « **/** ».*



- ***L'intégration de données dans l'exécutable présente plusieurs avantages : les données ne peuvent être perdues et cette opération permet la création d'exécutables véritablement autonomes.***
- ***Les inconvénients sont les suivants : si les données intégrées doivent être changées, il est impératif de remplacer l'exécutable entier, et la taille de ce dernier sera plus importante car il doit s'adapter aux données intégrées.***

conversion.h*

```
1  #ifndef CONVERSION_H
2  #define CONVERSION_H
3
4  #include <QtGui>
5  #include "ui_conversion.h"
6
7  class Conversion : public QMainWindow, public Ui::Conversion
8  {
9     Q_OBJECT
10 public:
11    Conversion(QWidget *parent = 0);
12 private slots:
13    void conversionEuroFranc();
14    void conversionFrancEuro();
15    void suivant();
16    void precedent();
17    void enregistrer();
18    void ouvrir();
19 protected:
20    void closeEvent(QCloseEvent *evt);
21 private:
22    void enregistrerCalcul(QString);
23    void retrouverCalcul(QString);
24 private:
25    const double TAUX;
26    QStringList historique;
27    int position;
28 };
29
30 #endif // CONVERSION_H
```

conversion.cpp

```
55 void Conversion::enregistrer()
56 {
57   QString nomFichier = QFileDialog::getSaveFileName(this, "Enregistrement de l'historique des conversions");
58   if (!nomFichier.isEmpty())
59   {
60     QFile fichier(nomFichier);
61     if (fichier.open(QIODevice::WriteOnly))
62     {
63       QTextStream ecriture(&fichier);
64       for (int i=0; i<historique.count(); i++) ecriture << historique[i] << endl;
65       statusBar()->showMessage("Enregistrement de l'historique effectué");
66     }
67     else statusBar()->showMessage("L'enregistrement n'a pas pu être effectué avec succès");
68   }
69 }
70
71 void Conversion::ouvrir()
72 {
73   QString nomFichier = QFileDialog::getOpenFileName(this, "Lire l'historique de conversion");
74   if (!nomFichier.isEmpty())
75   {
76     QFile fichier(nomFichier);
77     if (fichier.open(QIODevice::ReadOnly))
78     {
79       QTextStream lecture(&fichier);
80       QString ligne;
81       historique.clear();
82       while(!lecture.atEnd())
83       {
84         ligne = lecture.readLine();
85         historique.append(ligne);
86       }
87       position = historique.count();
88       suivant();
89     }
90   }
91 }
```

Création du fichier à partir du nom du fichier.

Récupération du nom du fichier à créer à partir du sélecteur de fichier.

Ouverture du fichier pour permettre l'écriture des données.

Création du flux pour écrire les données sous forme de texte dans le fichier choisi.

Ecriture de chaque conversion séparément qui prend une ligne de texte dans le fichier correspondant.

Création du fichier à partir du nom du fichier.

Récupération du nom du fichier à consulter à partir du sélecteur de fichier.

Ouverture du fichier pour permettre la lecture des données.

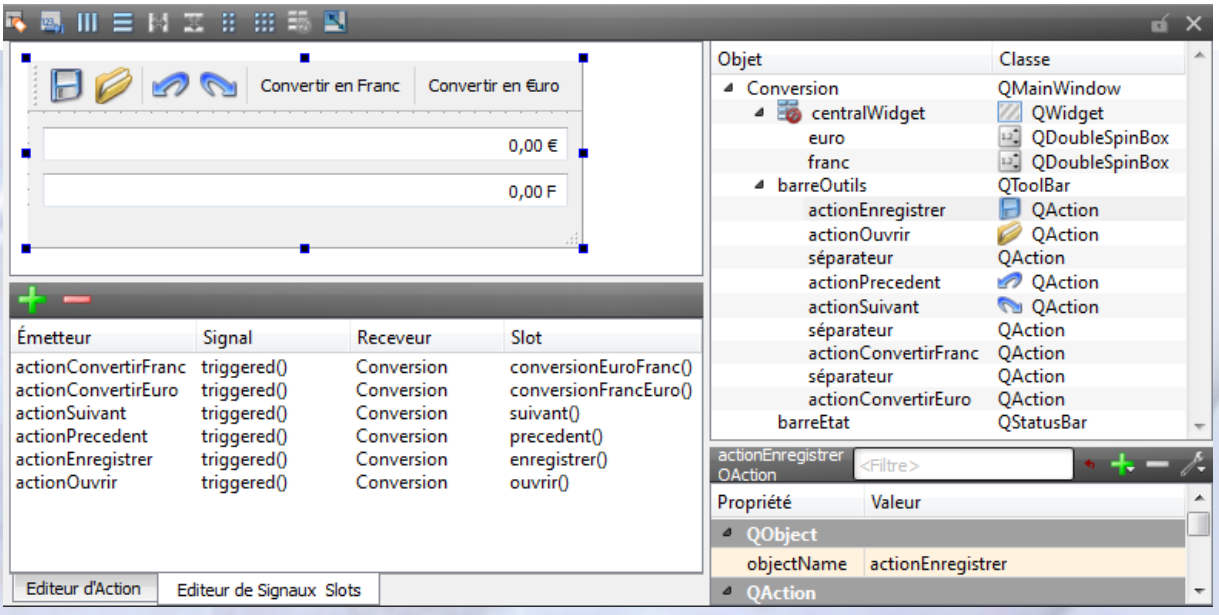
Création du flux pour lire les données sous forme de texte dans le fichier choisi.

Contrôle si la fin du fichier n'est pas atteinte.

Lecture du fichier texte ligne par ligne.

conversion.cpp | Conversion::Conversion(QWidget *) | Ligne : 4, Col : 1

```cpp
#include "conversion.h"
#include <QDataStream>

Conversion::Conversion(QWidget *parent) : QMainWindow(parent), TAUX(6.55957)
{
  setupUi(this);
  position = 0;
}

void Conversion::conversionEuroFranc()
{
  franc->setValue(euro->value()*TAUX);
  enregistrerCalcul(euro->text()+" --> "+franc->text());
}

void Conversion::conversionFrancEuro()
{
  euro->setValue(franc->value()/TAUX);
  enregistrerCalcul(franc->text()+" --> "+euro->text());
}

void Conversion::enregistrerCalcul(QString calcul)
{
  historique.append(calcul);
  position = historique.count();
  barreEtat->showMessage(calcul);
}

void Conversion::suivant()
{
  if (position < historique.count()) position++;
  retrouverCalcul(historique[position-1]);
}

void Conversion::precedent()
{
  if (position > 1) position--;
  retrouverCalcul(historique[position-1]);
}

void Conversion::retrouverCalcul(QString calcul)
{
  barreEtat->showMessage(calcul);
  QStringList liste = calcul.split(' ');
  if (liste[1]=="F") {
    franc->setValue(liste[0].toDouble());
    euro->setValue(liste[4].toDouble());
  }
  else {
    franc->setValue(liste[4].toDouble());
    euro->setValue(liste[0].toDouble());
  }
}
```



DEUXIÈME PROJET – MODÈLE-VUE-CONTRÔLEUR (MVC)

Au travers de ce projet, nous allons découvrir un pan important de la programmation qui consiste à séparer le traitement des données d'une part, et l'affichage de ces données d'autre part, au travers d'un standard dans ce domaine, le modèle **MVC**.

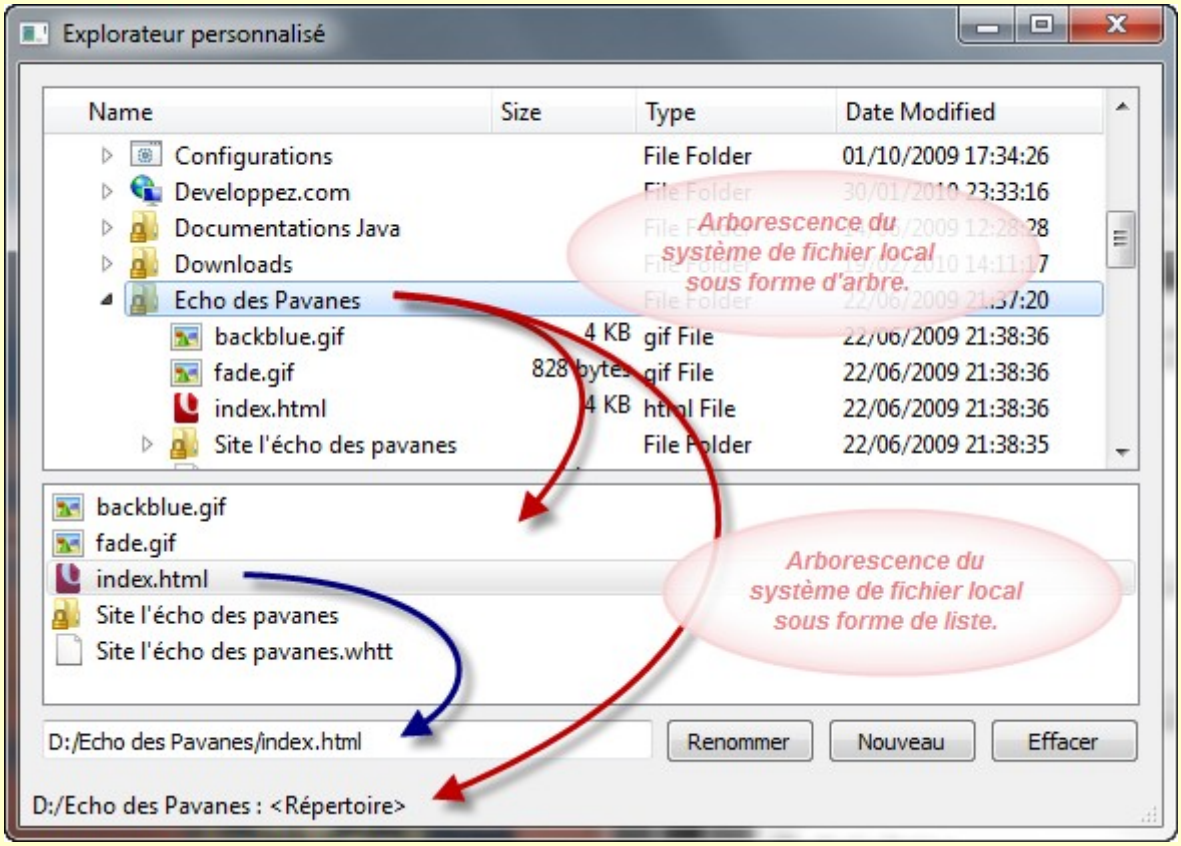
- ***Dans l'approche MVC, le modèle représente l'ensemble des données et il se charge de récupérer les données nécessaires pour afficher et enregistrer toute modification.***
- ***La vue présente les données à l'utilisateur. Seule une quantité limitée de données d'un grand ensemble sera visible en même temps, c'est-à-dire celles demandées par la vue.***
- ***Le contrôleur sert d'intermédiaire entre l'utilisateur et la vue ; il convertit les actions utilisateur en requêtes pour rechercher ou modifier des données, que la vue transmet ensuite au modèle si nécessaire.***

*Grâce à l'architecture **modèle/vue** de **Qt**, nous avons la possibilité d'utiliser des **modèles** qui ne récupèrent que les données nécessaires à l'affichage de la **vue**. Nous gérons donc de très grands ensembles de données beaucoup plus rapidement et nous consommons moins de mémoire qui si nous devions lire toutes les données.*

*De plus en enregistrant un **modèle** avec deux **vues** ou plus, nous donnons l'opportunité à utilisateur d'afficher et d'interagir avec les données de différentes manières, avec peu de surcharge. **Qt** synchronise automatiquement plusieurs **vues**, reflétant les changements apportés dans l'une d'elles dans toutes les autres.*

*L'architecture **modèle/vue** présente un autre avantage : si nous décidons de modifier la façon dont l'ensemble de données sous-jacent est enregistré, nous n'avons qu'à changer le **modèle** ; les **vues** continuerons à se comporter correctement.*

- ***Nous allons mettre en pratique toute cette théorie au travers d'un nouveau projet qui consiste à réaliser un explorateur personnalisé avec la présentation de l'arborescence du système de fichier local suivant deux vues différentes :***
- ***visualisation complète sous la forme d'un arbre,***
- ***visualisation du contenu d'un répertoire choisi sous forme d'une liste.***
- ***Par ailleurs, dans cet explorateur simplifié, vous aurez la possibilité de créer de nouveaux fichiers de les renommer ou de les détruire.***



*En général, nous ne devons présenter qu'un nombre relativement faible d'éléments à l'utilisateur. Dans ces cas fréquents, nous pouvons utiliser des classes spécifiques dans l'affichage d'éléments de **Qt** en association avec des modèles prédéfinis et également spécifiques suivant le type de données à présenter. Ces classes d'affichage et ces modèles spécifiques sont présenté dans la partie suivante.*

CLASSES D'AFFICHAGE ET MODÈLES PRÉDÉFINIS

S'agissant de grands ensembles de données, la duplication des données est souvent peu recommandée. Dans ce cas, nous pouvons utiliser les **vues** spécifiques de **Qt** en association avec un modèle de données, qui peut être un modèle personnalisé ou un des modèles prédéfinis de **Qt**. Voici la liste des vues les plus pratiques :

- **QListView : présentation des données sous forme de liste.**
- **QTableView : présentation des données sous forme de tableau.**
- **QTreeView : présentation des données sous forme d'arbre.**

*Par exemple, si l'ensemble de données se situe dans une base de données, nous pouvons combiner un **QTableView** avec un **QSqlTableModel**. Autre exemple courant, le système de fichier auquel nous pouvons associer un **QTreeView** avec un **QdirModel**.*

Qt propose ainsi plusieurs modèles prédéfinis à utiliser avec les classes d'affichage dont la liste est proposée ci-dessous :

- **QStringListModel : stocke une liste de chaînes.**
- **QStandardItemModel : stocke des données hiérarchiques arbitraire.**
- **QDirModel, : encapsule le système de fichiers local.**
- **QFileSystemModel : même chose que précédemment qui remplace d'ailleurs ce dernier.**
- **QSqlQueryModel : encapsule un jeu de résultats SQL.**
- **QSqlTableModel : encapsule une table SQL.**
- **QSqlRelationalTableModel : encapsule une table SQL avec des clés étrangères.**
- **QSortFilterProxyModel : Trie et/ou filtre un autre modèle.**

Notre projet exploite la classe **QFileSystemModel**, qui encapsule le système de fichier de l'ordinateur et qui peut afficher (et masquer) les divers attributs de fichiers. Ce modèle peut appliquer un filtre pour limiter les types d'entrées du système de fichiers qui sont affichés et peut organiser les entrées de plusieurs manières différentes.

- **QFileSystemModel() : construit le modèle représentant le système de fichier local.**
- **fileIcon(index) : retourne l'icône du fichier sélectionné désigné par l'index.**
- **fileInfo(index) : retourne les caractéristiques complètes du fichier sélectionné désigné par l'index.**
- **fileName(index) : retourne le nom du fichier sélectionné désigné par l'index.**
- **filePath(index) : retourne le chemin complet du fichier sélectionné désigné par l'index.**
- **filter() - setFilter(nouveau filtre) : retourne ou propose un nouveau filtre sur le modèle entier.**
- **index(chemin complet) : retourne l'index du modèle suivant le chemin spécifié en argument.**
- **isDir(index) : indique si l'élément sélectionné par index est un répertoire ou pas.**
- **lastModified(index) : retourne la date et l'heure de la dernière modification du fichier sélectionné désigné par l'index.**
- **mkdir(index, nom) : crée un sous-répertoire relatif à la position de index dont le nom est spécifié en argument**
- **nameFilters() - setNameFilters(noms des filtres) : renvoie ou propose la liste des noms des filtres du modèle entier.**
- **permissions(index) : retourne les permissions accordées du fichier sélectionné désigné par l'index.**
- **remove(index) : suppression définitive du fichier sélectionné désigné par l'index.**
- **rmdir(index) : suppression définitive du répertoire sélectionné désigné par l'index.**
- **rootDirectory() : renvoie le répertoire racine actuel sous forme de QDir.**
- **rootPath() - setRootPath(racine) : renvoie ou propose le nom du répertoire racine actuel sous forme de QString.**
- **size(index) : retourne la taille en octets du fichier sélectionné désigné par l'index.**
- **type(index) : retourne le type, comme « JPEG file » ou « Directory» de l'élément sélectionné désigné par l'index.**

*Vous remarquez que nous utilisons fréquemment la notion d'index dans les modèles. L'index représente l'entité sélectionné dans le modèle, correspondant à une action spécifique de l'utilisateur dans la classe d'affichage associée. Cet index n'est pas un nombre quelconque. Il est opérationnel sur tous les types de modèle de **Qt**. Une classe spécifique, **QModelIndex**, a donc été créée pour pouvoir parcourir et indicer n'importe quel type de modèle de **Qt**.*

x FICHIERS SOURCES

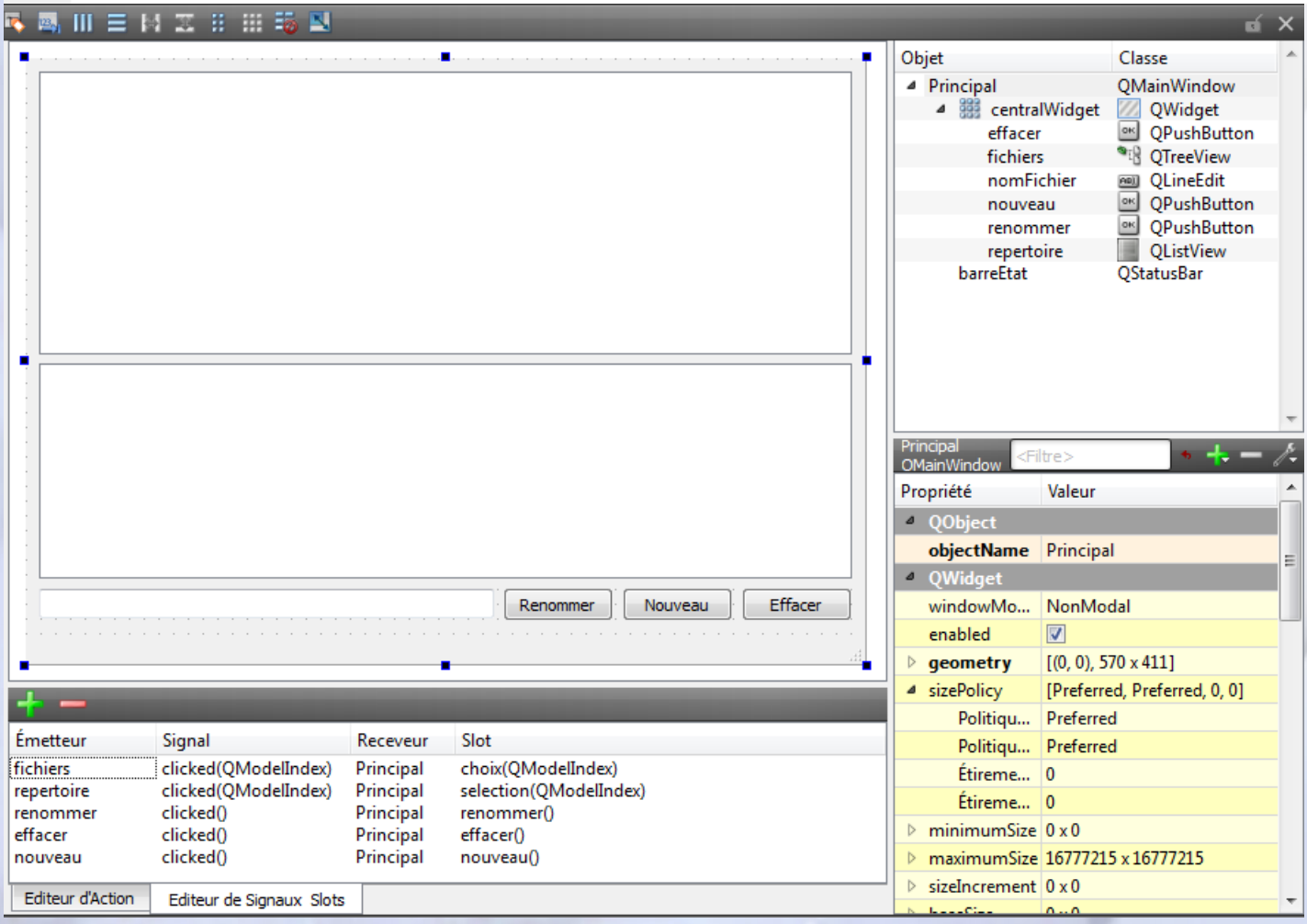
principal.h | <Selectionner un symbole>

```
1   #ifndef PRINCIPAL_H
2   #define PRINCIPAL_H
3
4   #include <QMainWindow>
5   #include <QFileSystemModel>
6   #include "ui_principal.h"
7
8   class Principal : public QMainWindow, public Ui::Principal
9   {
10    Q_OBJECT
11  public:
12    Principal(QWidget *parent = 0);
13  private slots:
14    void choix(QModelIndex);
15    void selection(QModelIndex);
16    void renommer();
17    void nouveau();
18    void effacer();
19  private:
20    QFileSystemModel modele;
21    QModelIndex index;
22  };
23
24  #endif // PRINCIPAL_H
```

Slot sollicité lorsque l'utilisateur sélectionne un fichier ou un répertoire dans la vue de type arbre.

Slot sollicité lorsque l'utilisateur sélectionne un fichier ou un répertoire dans la vue de type liste.

Création d'un seul modèle du système de fichier local pour deux vues différentes.



principal.cpp | Principal::Principal(QWidget *) | Ligne : 7, Col : 1

```
1   #include "principal.h"
2   #include <QtGui>
3   #include <QFileInfo>
4   #include <QTreeView>
5   #include <QModelIndex>
6
7   Principal::Principal(QWidget *parent) : QMainWindow(parent)
8   {
9     setupUi(this);
10    QModelIndex index = modele.setRootPath("C:/");
11    fichiers->setModel(&modele);
12    repertoire->setModel(&modele);
13    repertoire->setRootIndex(index);
14  }
15
16  void Principal::choix(QModelIndex index)
17  {
18    QString type = modele.fileInfo(index).isFile() ? "Fichier" : "Répertoire";
19    repertoire->setRootIndex(index);
20    barreEtat->showMessage(modele.filePath(index)+" : <"+type+">");
21    nomFichier->setText(modele.filePath(index));
22  }
23
24  void Principal::selection(QModelIndex index)
25  {
26    this->index = index;
27    nomFichier->setText(modele.filePath(index));
28  }
29
30  void Principal::renommer()
31  {
32    QFile fichier(modele.filePath(index));
33    fichier.rename(nomFichier->text());
34  }
35
36  void Principal::nouveau()
37  {
38    QFile fichier(nomFichier->text());
39    fichier.open(QIODevice::WriteOnly);
40  }
41
42  void Principal::effacer()
43  {
44    if (!modele.isDir(index)) modele.remove(index);
45  }
```

Association du modèle aux deux vues.

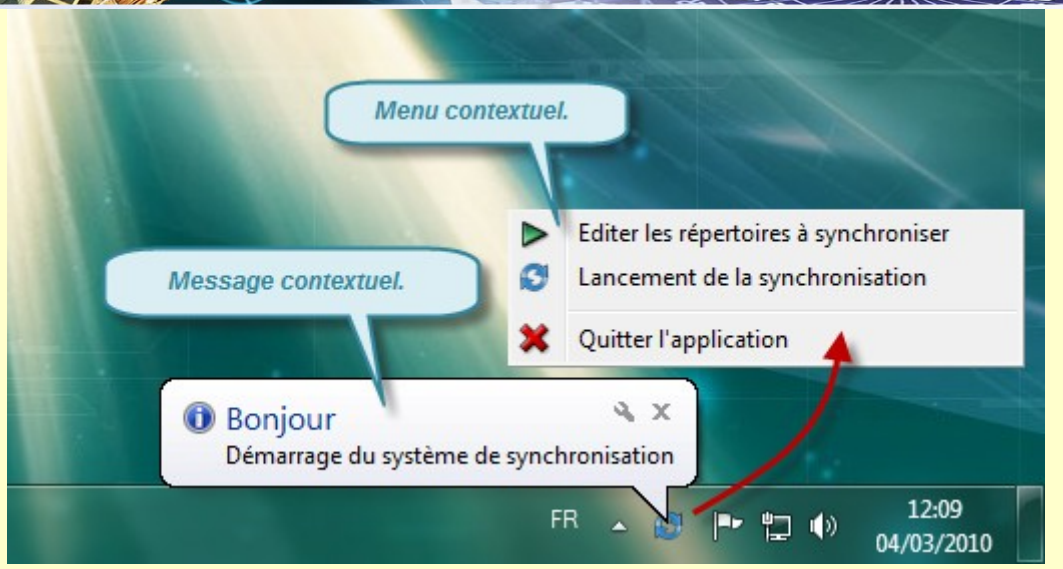
Sélection du répertoire de référence (répertoire racine) pour la vue répertoire.

SYSTÈME DE NOTIFICATION EN BARRE DES TÂCHES DU SYSTÈME D'EXPLOITATION

Les systèmes d'exploitation, quels qu'ils soient, disposent tous d'un système de notifications, représentées par les icônes des applications, dans la barre des tâches, juste à côté de l'horloge.

Ces différentes applications ont la particularité d'être actives dès le démarrage du Système d'exploitation. Il suffit alors de cliquer sur l'icône souhaité pour faire apparaître le menu correspondant et de lancer ainsi l'action désirée.

Par ailleurs, suivant le déroulement de l'application, des petites bulles d'aide peuvent apparaître pour notifier que certains événements ont bien été prise en compte. Dans **Qt Creator**, la classe qui représente une notification s'appelle **QSystemIconTray**.



*Dans votre application, si vous désirez mettre en œuvre une notification, vous devez donc créer un objet relatif à la classe **QSystemIconTray**. Durant la phase de création, vous devez indiquer quel est l'élément qui lance cette notification, généralement vous spécifiez le pointeur **this**. Vous pouvez également proposer à ce moment là, l'icône de notification. Vous devez ensuite proposer le menu contextuel (qui est à construire au préalable) au moyen de la méthode **setContextMenu()**. Il faut bien entendu rendre visible votre notification à l'aide de la méthode **show()**. Enfin, suivant des événements particuliers, vous pouvez visualiser des messages contextuels au moyen de la méthode **showMessage()**.*

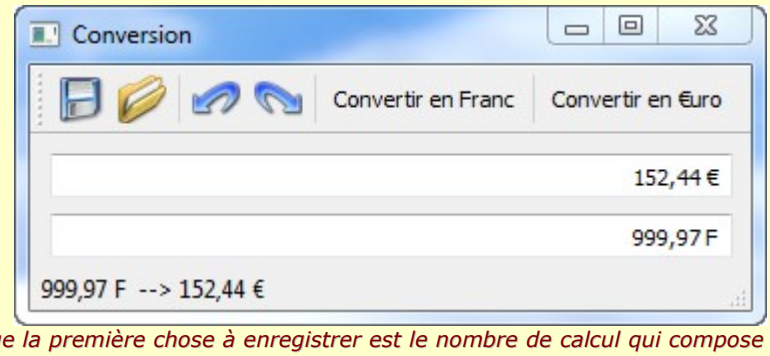
- ***QSystemIconTray(parent)***
- ***QSystemIconTray(icône, parent) : Création d'une notification en barre des tâche en relation avec celui qui le demande.***
- ***contextMenu() - setContextMenu(nouveau menu) : retourne ou propose un nouveau menu contextuel.***
- ***hide() - show() : enlève ou visualise l'icône de notification.***
- ***isSystemTrayAvailable() : indique si le système de notification existe dans votre système d'exploitation.***
- ***icon() - setIcon(icône) : retourne ou propose une nouvelle icône de notification.***
- ***toolTip() - setToolTip(message) : bulle d'aide qui apparaît automatiquement pendant quelques instants lorsque le curseur de la souris pointe sur l'icône de notification.***
- ***showMessage(titre, message, type d'icône, durée) : Message contextuel avec un titre qui apparaît pendant 10 secondes par défaut. Si vous désirez changer le temps d'affichage, vous devez spécifier le nombre de millisecondes sur l'argument durée. Dans ce message contextuel apparaît une petite icône qui indique le type de message, soit une simple information, soit un message d'avertissement ou enfin un message d'erreur. Prenez l'une des constantes suivantes pour choisir l'icône qui convient :***
- ***enum {NoIcon, Information, Warning, Critical};***

TRAVAUX PRATIQUES

Afin de confirmer l'ensemble de nos nouvelles connaissances, je vous invite comme d'habitude, à réaliser un certain nombre de projets.

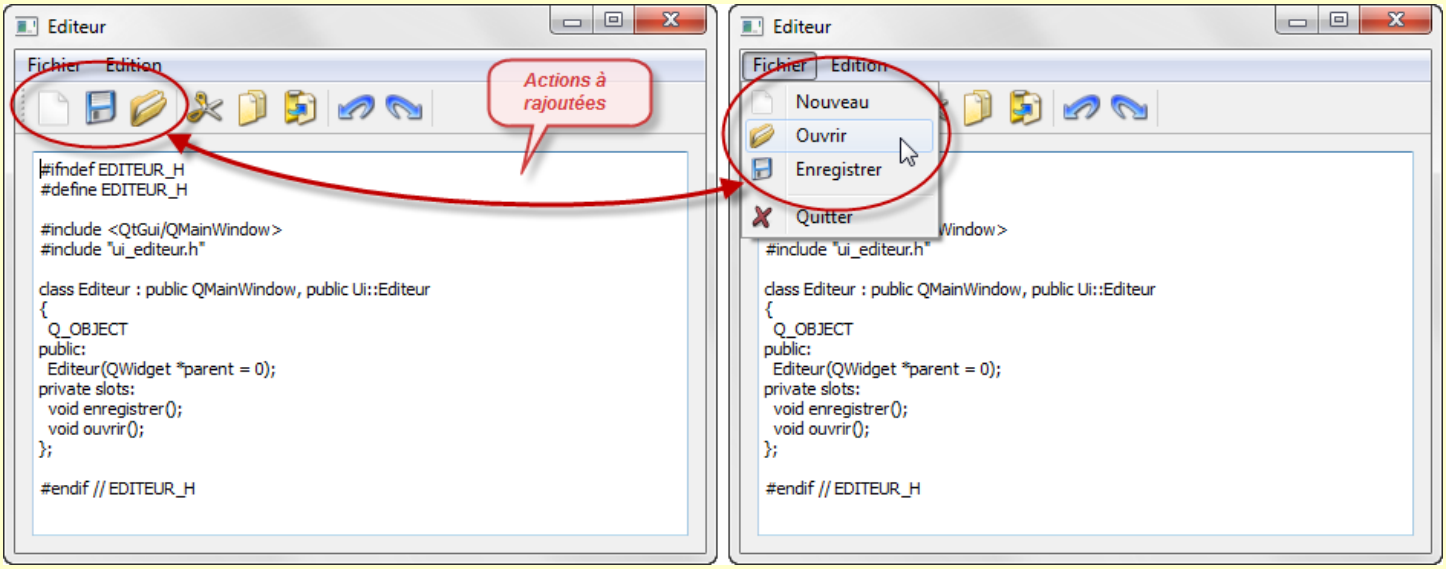
- ***Pour le premier d'entre eux, nous reprenons l'application sur la conversion et nous changeons la façon dont nous enregistrons l'historique des calculs. Cette fois-ci, nous désirons que le fichier produit ne soit plus un texte et donc qu'il ne puisse plus être lisible au travers d'un simple éditeur. Au contraire, nous désirons conserver un certain anonymat sur tous ces calculs réaliser, et nous souhaitons ainsi que l'enregistrement se fasse en binaire.***

- ***Vous redéfinirez donc respectivement les méthodes enregistrer() et ouvrir() en prenant cette fois-ci, un flux binaire QDataStream en lieu et place du flux de texte QTextStream.***

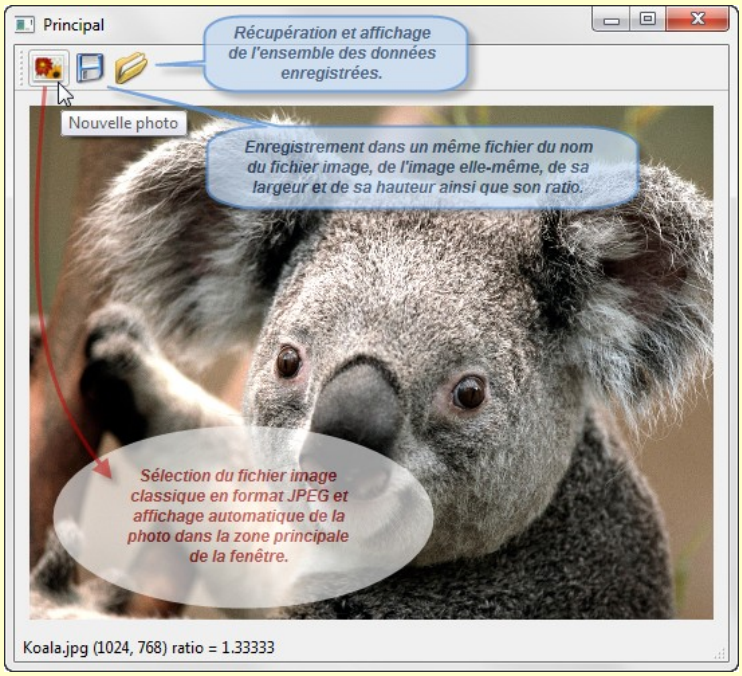


Pour vous aidez, pensez que la première chose à enregistrer est le nombre de calcul qui compose la liste des calculs. Cela vous aidera bien entendu pour la lecture, puisque vous serez alors au courant du nombre d'enregistrement effectué.

✗ Le deuxième projet consiste à reprendre l'éditeur dans lequel vous allez rajouter l'enregistrement et l'ouverture de document texte. Pour cela, vous devez proposer de nouvelles actions circonstanciées et écrite quelques lignes de code.



✗ Dans le projet suivant, vous allez vous servir du code source proposés en page 5 en l'adaptant à la situation. Cette application permet de récupérer et d'afficher une photo présente dans le système de fichier local. Par la suite, vous fabriquerez un fichier dont le contenu sera dans l'ordre, le nom du fichier image, la photo elle-même, sa largeur, sa hauteur et le ratio de l'image. Ce fichier peut à tout moment être lu pour revoir l'image enregistrée ainsi que l'ensemble de ses caractéristiques. Finalement, ce logiciel est capable de lire deux types de fichiers.



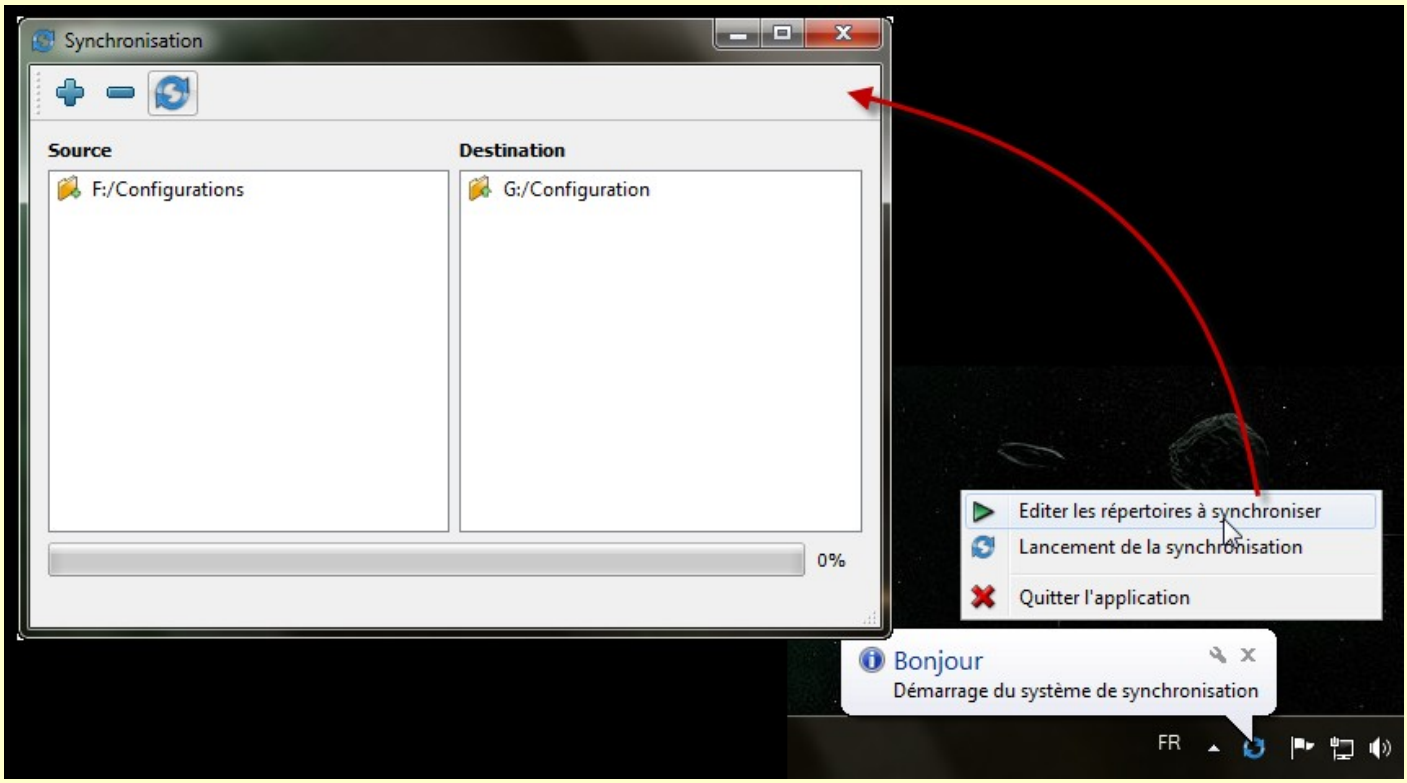
✗ ***Pour vous aider également dans ce projet, voici le code source du fichier en-tête :***

photooriginale.h | PhotoOriginale | Ligne : 6, Col : 1

```
1  #ifndef PHOTOORIGINALE_H
2  #define PHOTOORIGINALE_H
3
4  #include <QWidget>
5
6  class PhotoOriginale : public QWidget
7  {
8    Q_OBJECT
9  signals:
10   void envoyerMessage(QString);
11 public:
12   PhotoOriginale(QWidget *parent = 0) : QWidget(parent) {}
13 protected:
14   void paintEvent(QPaintEvent *);
15 private slots:
16   void changerPhoto();
17   void enregistrer();
18   void ouvrir();
19 private:
20   QString calculMessage();
21 private:
22   QImage photo;
23   int largeur, hauteur;
24   double ratio;
25   QString nomFichier;
26 };
27
28 #endif // PHOTOORIGINALE_H
```

✗

✗ ***Le dernier projet est plus conséquent puisqu'il s'agit de réaliser un logiciel de synchronisation entre deux unités de stockage entre, par exemple, des données sur le disque et la sauvegarde sur clé USB.***



Nous profitons de l'occasion de voir comment créer une icône de notification en barre des tâches qui est représentée par la classe ***QSystemTrayIcon****.*