Dt : 27 July 2019

Higher label programming

Chapter 7 : Libraries

Q1: What is a Library?

A library is a set of code which should be reuse in main stream applications.

A library is already camailed , tested framework or a package which can be reused by the application in different formats . Such as

1. Source code
2. Binary format lib formats
3. Packages are .lib and header files in linux platform

Q : What is a Forward Declaration ? And where it should be used ?

A forward declaration is way of declaring a class which is reused as a function parameter or a return type . For a limited use of class in the header the it always help compiler to parse quickly instead adding complete header files .

The significant use of it is when you use a pointer to a class as member variable of another class.

When Forward declaration of the class written , it can be used as pointer or reference not a value .

Example :

class Level;

Is a forward declaration, as the class are not have a body compailer will consider this as only as declartion .

You can then use pointers (Level *) and references (Level &) to this undefined class freely.

Note that you cannot directly instantiate Level since the compiler needs to know the **class's size to create variables.**

```
class Level;

class Entity{

   Level &level;  // legal

Level level;   // illegal

};
```

Q: What is Frameworks and What is Component ?

A framework is large collection of general purpose and domain specific class and conversations for consistency od application behavior .

Q : What is Design patterns ?

Design patterns are the solution of design problems .

As per the purpose the design patterns are subdivided to different catagories .

1. Creational pattern : - Which manages the creation of objects
2. Structural pattern :- How objects are related with each other
3. Behavioral pattern :- It deals with communication with each other

Q: Use of Iteration and the Visitor Pattern in QDir , QFile and QFileInfo?

Qt have two usefull calsses to get information about directory and files in file system .

QDir and QFileInfo :

```
/* This function searches for all Mp3 files in a directory

tree and adds them. */

void Mp3Db::recurseAddDir(QDir d) {

        d.setSorting( QDir::Name ); -> set the sorting

        d.setFilter( QDir::Files | QDir::Dirs ); -> find a filer

        QStringList qsl = d.entryList(); -> get list of entry in the directory

        foreach (QString entry, qsl) { -> iterate the list and read each entry as a recursive
        way

        QFileInfo finfo(entry); -> get information about the file

        if ( finfo.isSymLink () && !m_SymLink )

                return;

        if ( finfo.isDir() ) {

                if (!m_Recursive )

                        return;

                        QString dirname = finfo.fileName();

                        if ((dirname==".") || (dirname == ".."))

                        return;

                        QDir d(finfo.filePath());

                        if (skipDir(d))

                        return;
                        recurseAddDir(d);
                } else {
                if (fi->extension(false)=="mp3") {
                        addMp3File(fi->absFilePath());
                }
                }
        }
}
```

Q: What is a visitor pattor ?

A patter used in Qt to communicate beetween two resource objects .

It can be done in two different methods .

1. By Signal and Slot mechanisms .
2. By Inheritance .

Q: What is QObject ?

This is the class from which all the Qt Widget objects are derived .

An QIbject looks ad below

```
class QObject {
   public:
   QObject(QObject* parent=0);
   QObject * parent () const;
   QString objectName() const;
   void setParent ( QObject * parent );
   const ObjectList & children () const;
   // ... more ...
};
```

- The QObject Copy constructor is not in public that mean it is not allow to copy .
- QObject can not pass to another function by its value , but a QObject data member can be created .

Q: Why copying QObject is not permitted?

Qt disables copying of QObjects is that it manages the memory of the children of a QObject. When a QObject gets deleted, so do all of its children. This would be impractical to do properly if the QObject was copiable.

Each QObject parent manages its children. This means that the QObject

destructor automatically destroys all of its child objects.

Q: What is concept of reparenting in QObjects?

The Child list establish a Bidirectional One to many association of objects .

A QObject can set parent QObject as below

objA->setParent(objB);

ObjB become the parent of objA ,

Then if we change as below

objA->setParent(objC);

Then the pointer of ObjA remove from ObjB and added to ObjC , called reparenting

Q: Parent Objects versus Base Classes ?

The two behaviour should not be confused .

As parent children relationship is created to provide a management of object in run time .

Where in case if base derive relationship at the time of compile time .

Furthermore, if the copy gets destroyed (e.g., if the copy was a

value parameter in a function call), each child needs to be destroyed too. Even with

resource sharing methods, this approach would introduce some serious difficul-

ties. A deep copy of the child list could be a costly operation if the number of chil-

dren were large and the objects pointed to were large.

Q How a child and parent object are Destroyed ?

QObject parent object pointer can be passed to the creation of new child pointer as below.

MainWindow w(0,"w");

MainWindow *w1 = new MainWindow(&w,"w1");

Above w is the parent object and w1 is chiled pointer created from the address of w.

Then w destroctoe is call first then call child class destryed for w1.

If a new chuld pointer with no parent created then allocated memory will not free and memory leak will happen as below

MainWindow *x = new MainWindow(0,"c");

No destroctor will call for the pointer x because there is no parent availabe , it has to destry by application programmer by calling delete explicitly

Q: How Composit and Component design pattern involve to define parent and child relationship in QObject ?

A Composit object can contain a children

A Component can have a parent

Q: QApplication and QEvent Loop ?

Qt is an event based framework where messages are passed through events .

An observer Parten is involved while event based message passing occurs .

The event loop is started with a call to exec(). Long-running operations can call processEvents() to keep the application responsive.

The Qt class QEvent encapsulates the notion of an event.

QEvent is the base class for several specific event classes such as QActionEvent , QFileOpenEvent ,

QHoverEvent , QInputEvent , QMouseEvent

The type() member function returns an enum that has early a hundred specific values that can identify the particular kind of event

Q: What are the features are supported by Q_OBJECT

Children

Signals and slots

MetaObjects, metaproperties, metamethods

qobject_cast

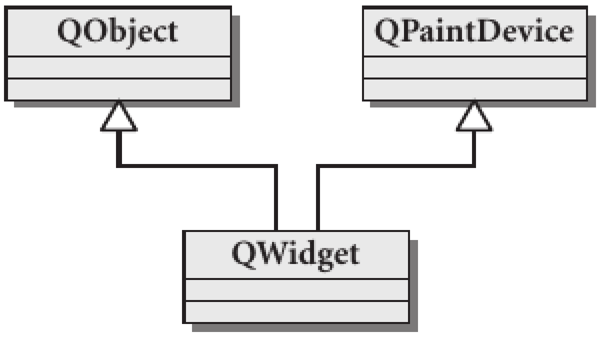
The Meta Object Compiler, moc , generates additional functions for each QObject - derived

class that uses the macro. Generated code can be found in files with names moc_filename.cpp .

Q: How the QWidget is created and use ?

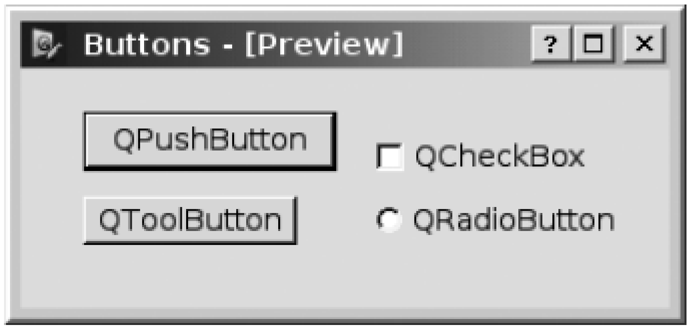
QWidget is visual representation of screens

- QWidget inherited from QObject and QPaintDevice
- A widget that has no parent is called a window.
- If one widget is a parent of another widget, the boundaries of the child widget lie completely within the boundaries of the parent.
- A QWidget can handle events by responding to signals from various entities in the window system (e.g., mouse, keyboard, other processes, etc.).

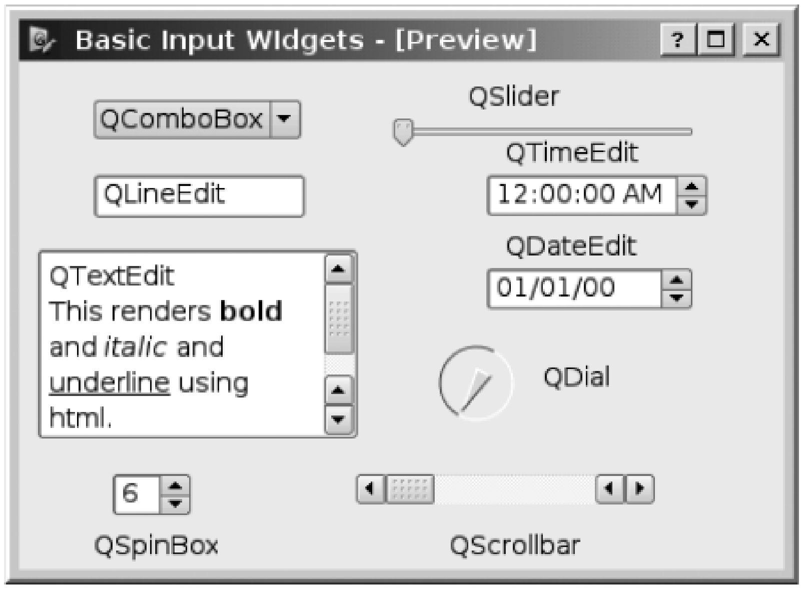


Q : Different Category's of QWidgets ?

Button widgets QPushButton, QToolButton, QRadioButton , QCheckBox



Input widgets QLineEdit, QSlider, QTextEdit, QTimeEdit, QDateEdit etc



Display widgets QLabel , QProgressBar ,and QPixMap .
Container widgets, such as the QMainWindow , QFrame , QToolBar , QTabWidget , and QStackedWidget
These widgets are used as building blocks to create other more complex widgets, such as:

- Dialogs for asking the user questions or popping up information, such as the QFileDialog , QInputDialog , and QErrorMessage .
- Views that provide displays of collections of data such as QListView , QTreeView , and QTableView

Q: What are the classes Which are not inherited from QWidget .

- **Qt Data types:** QPoint and QSize are popular types to use when working with graphical objects.
- **Controller classes:** QApplication and QAction are both objects that manage the GUI application's control flow.
- **Layouts:** These are objects that dynamically manage the layout of widgets.There are specific layout varieties: QHBoxLayout , QVBoxLayout ,QGridLayout , etc.
- **Models:** The QAbstractItemModel and its derived classes QAbstractListModel and QAbstractTableModel are part of Qt's model/view framework, and are used as base classes for classes that rep-resent data for a QListView , QTreeView , or QTableView .
- **Database models:** These are for use with QTableView (or other customized view classes) using SQL Databases as data sources: QSqlTableModel and QSqlRelationalModel .

Q: What is Monostate Pattern.

A Class is allow multiple instances to share the same state is called Monostate pattern.

Two instances of QSettings with the same organization/application name can be used to access the same persistent map data. This makes it easy for applications to access commonsettings from different source files.

QSettings is an implementation of the Monostate pattern.

Q: How to add a action into slot and trigger a menu bar action to do your operation ?

In mainwindow class :-

```
QAction *action;
QMenu *file;

action = new QAction(tr("New"),this);

action->setShortcuts(QKeySequence::AddTab);

action->setStatusTip(tr("new file creation"));

connect(action, &QAction::triggered, this, &MainWindow::newFile);


file = new QMenu("&Questions", this);

QMainWindow::menuBar()->addMenu(file);

file->addAction(action);
```

This Action works based on Command design pattern

Q: QProcess use?

QProcess is a very convenient (and cross-platform) class for starting and controlling other processes. It is derived from QObject and takes full advantage of signals and slots to make it easier to "hook up" with other Qt classes.

- A QProcess can spawn another process using the start() function.
- The new process is a child process in that it will terminate when the parent process does.

Q: What is Antipattern?

- Copy and paste programming: Copying and modifying existing code without creating more generic solutions.
- Hard coding: Embedding assumptions about the environment (such as constant numbers) in multiple parts of the software

- Interface bloat: Having too many methods, or too many arguments in functions; in general, refers to a complicated interface that is hard to reuse or implement.
- Reinventing the (square) wheel: Implementing some code when some-thing (better) already exists in the available APIs.
- God Object: An object that has too much information or too much responsibility. This can be the result of having too many functions in a single class. It can arise from many situations, but often happens when code for a model and view are combined in the same class.

Q: QMetaObject: The MetaObject Pattern ?

By abstracting the abstract data type itself, we achieve what is called a MetaObject.

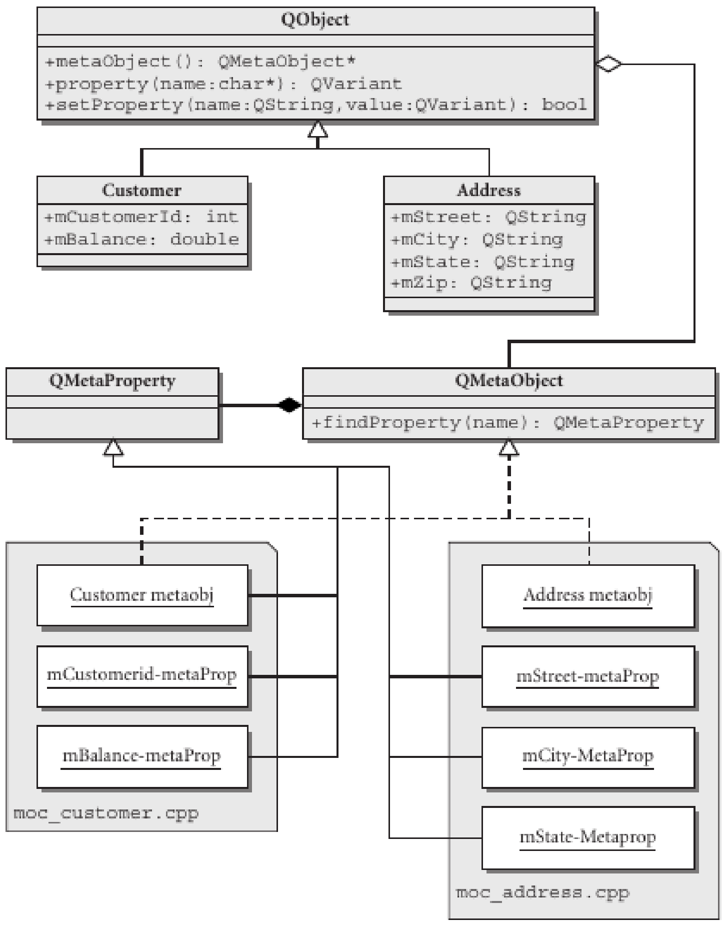
A MetaObject is an object that describes the structure of another object.

- The QMetaObject is Qt's implementation of the MetaObject pattern. It provides information about properties and methods of a QObject. The MetaObject pattern is sometimes known as the Reflection pattern.
- A single QMetaObject instance is created for each QObject subclass that is used in an application, and this instance stores all the meta-information for the QObject subclass. This object is available as QObject::metaObject().
- A QMetaObject can be used to invoke functions such as:

  className() , which returns the class name as a const char*

  superClass() , which returns a pointer to the QMetaObject of the base class if there is one (or 0 if there is not)

  methodCount() , which returns the number of member functions of the class.



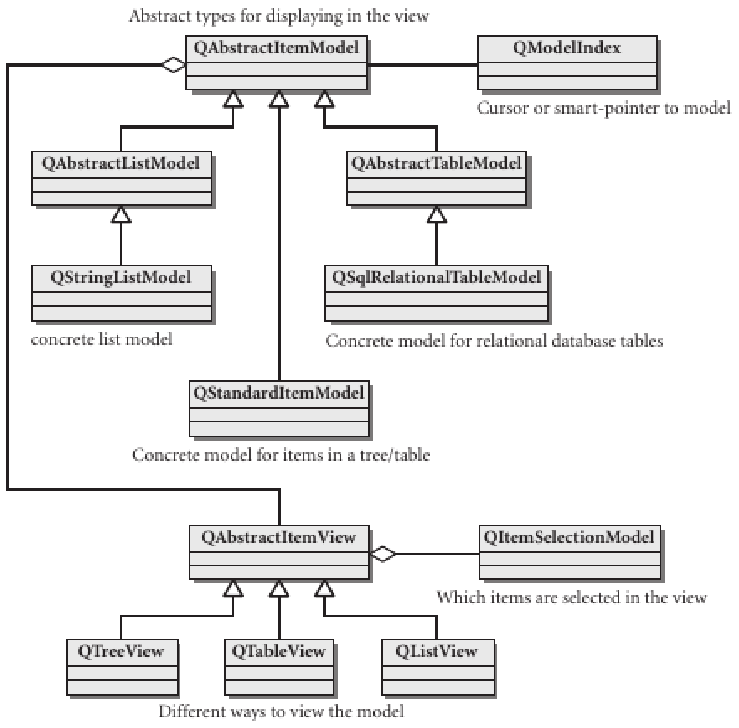
Q: QVariant?

QVariant is a union wrapper 5 for all the basic types, as well as all permitted Q_PROPERTY types. You can create a QVariant as a wrapper around another typed value. It remembers its type, and has member functions for getting and setting its value.

Q: What is Model and View Architecture?

The Model-View pattern describes techniques of separating the underlying data (the model) from the class that presents the user with a GUI (the view).

- separating model from view reduces the complexity of each model and view.
- Model code and view code have completely different maintenance imperatives—changes are driven by completely different factors—so it is much easier to maintain each of them when they are kept separate.
- view classes that can be reused with well-designed models is constantly growing.
- Controller code is code that manages the interactions among events, models, and views.


Abstract types for displaying in the view
QAbstractItemModel
QModelIndex
Cursor or smart-pointer to model
QAbstractListModel
QAbstractTableModel
QStringListModel
QSqlRelationalTableModel
concrete list model
Concrete model for relational database tables
QStandardItemModel
Concrete model for items in a tree/table
QAbstractItemView
QItemSelectionModel
Which items are selected in the view
QTreeView
QTableView
QListView
Different ways to view the model

A view obtain model index from the model

View can retrieve data from the data source , A deligity render data in the view , when a model want to communicate a deligity is used directly model for communication.

All model , view , controller and deligiti class are communicate using signal and slots.

Practical example :-

Created a controller view and model class to update Sensor information , signal and value into a visualization table and error panel to display the error value in the error table view.

Step1 : Create Controller , Model and view object In panel

Step2 :- model is derived from QAbstractTableModel ,

RowCount , ColumnCount, HeaderData, Data

These are the interface from the abstract class for the model which provide data to view.

Populate model is the interface to to model to refresh with new data which will reset the model with each new entry of data.

```
 this->beginResetModel();

this->endResetModel();
```

Step 3 : Creating table view

```
table_ = new QTableView(this);
```

table_->setModel(model_.get());

table_->resizeColumnsToContents();

Step4 :

Data change should be updated from controller

Q: Views

A View class encapsulates the components of a graphical user interface that

accesses the data in a model. Views come in a variety of different sizes and shapes.

In general, they are usually

- Lists in various arrangements
- Tables, perhaps with interactive elements
- Trees representing objects in a parent-child hierarchy
- Graphs and charts

Q: Model Index

The QModelIndex class provides a generic access system that works for all classes derived from QAbstractItemModel .

This system treats model data as if it were arranged in a rectangular array with row and column indices, regardless of what underlying data structure actually holds the data.

QModelIndex objects, created by the model, can be used by model, view, or delegate code to locate particular items in the data model. QModelIndex object have short life spans and can become invalid shortly after being created, so they should be used immediately and then discarded.

QModelIndex::isValid() should be called before using a QModelIndex

object that has existed for more than a few operations. QPersistentModelIndex objects have longer life spans but still should be checked with isValid() before being used

Q: Table Models

Extend and customize the QAbstractTable

QAbstractTableModel has a series of pure virtual functions, declared I which must be overridden, because they are invoked by QTableView to get and set data.

Q: Tree Models

To represent data for a hierarchy of widgets (parents and children), Qt offers two

choices of models:

1. QAbstractItemModel is a general-purpose, but very complex class, that can be used with QTreeView as well as QListView and QTableView .

2. QTreeWidgetItem is a simpler model, specifically for use with QTreeWidget .

The QTreeWidgetItem class is a tree node that can be instantiated or extended. The widget items need to be connected together in a tree-like fashion, similar to QObject children or QDomNodes (Section 14.3). In fact, QTreeWidgetItem is another implementation of the Composite pattern.