[ELO329] Desarrollo en Qt

Agustín González
Patricio Olivares

7 de julio de 2019

Table of contents

¿Qué es Qt?
¿Qué es Qt?

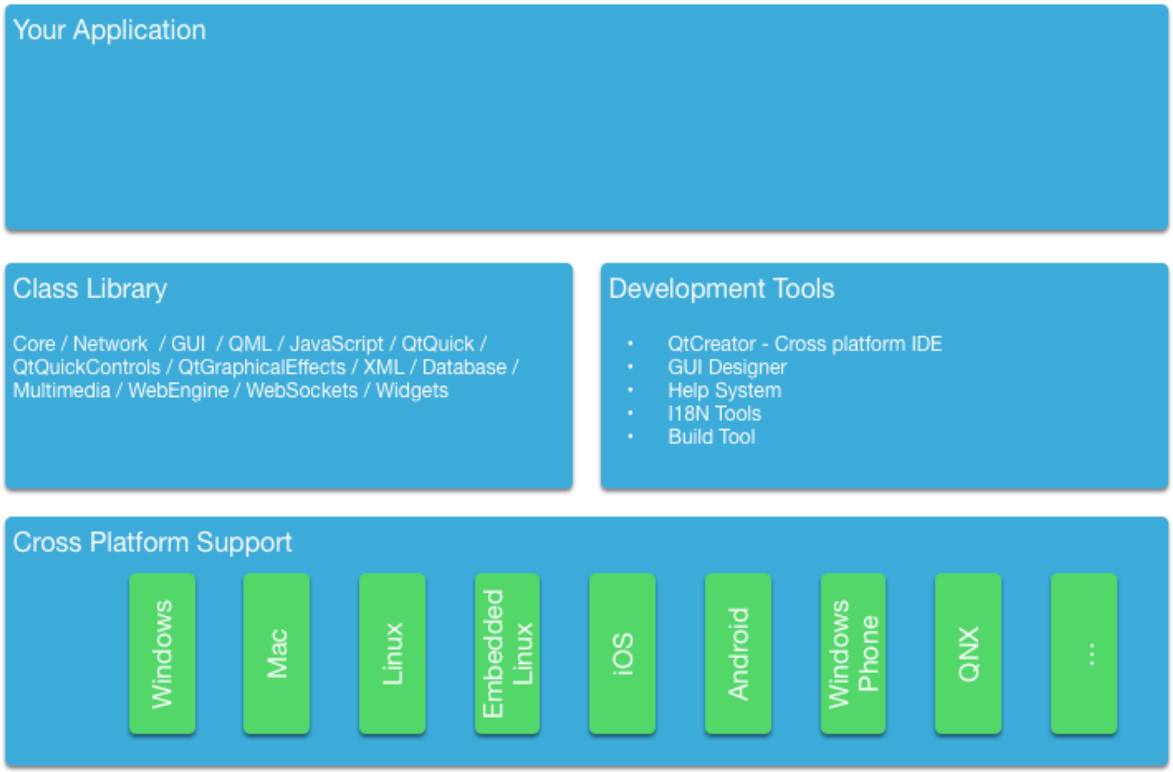
- Qt (pronunciado como çute") es un framework de desarrollo de aplicaciones con múltiples bibliotecas
- Creado por Haavard Nord y Erick Chambe-Eng en 1995 (primer release)
- Desarrollado sobre C++ inicialmente, pero en la actualidad es posible utilizarlo en otros lenguajes (Ej: PyQt para Python).
- Provee mejoras a las bibliotecas nativas de C++ y permite que códigos desarrollados sobre estas bibliotecas sean compatibles con múltiples plataformas (mismo concepto que se aplicaba en Java).
- Algunas de las plataformas compatibles con Qt
  - Windows
  - Linux
  - Unix
  - MacOS
  - Mobile (iOS, Windows Phone, Android)
- Última versión lanzada el Jun. 2019: Qt 5.13 (www.qt.io)

¿Qué es Qt?
Licencias de desarrollo

- Dependiendo del tipo de producto que se desee desarrollar, Qt se puede usar bajo diferentes licencias de desarrollo.
- Existen dos tipos de licencias de desarrollo:
  - Licencia comercial: Utilizada para desarrollo de aplicaciones de código cerrado
  - Licencia GNU Lesser General Public License (LGPL) v3: Licencia utilizada para desarrollo de aplicaciones de código abierto.
    Ojo ¡Que sean de código abierto **no** quiere decir que no se puedan comercializar!
- Qt además contiene códigos con licencias específicas asociadas al autor del mismo. Cuando se desarrollan aplicaciones con fines comerciales, es **crítico** revisar las licencias de las bibliotecas utilizadas (esto bajo cualquier contexto de desarrollo)

¿Qué es Qt?
Arquitectura

- El código se construye sobre bibliotecas Qt y sus distintas herramietas de desarrollo
- Éstas a su vez generan una abstracción para ejecución multiplataforma



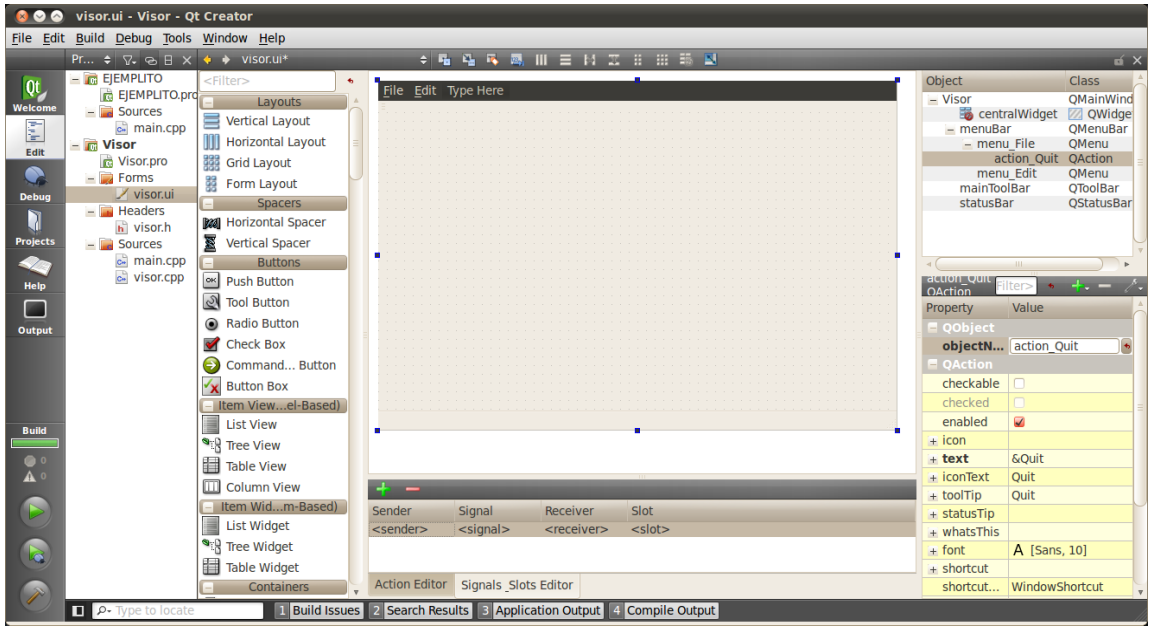
Existen dos tipos de módulos en Qt:

- Módulos Core-Essential: Son los que dan forma a Qt. Son parte del desarrollo de estas bibliotecas.
  Ejemplos de módulos core-essential: QtCore, QtGui, QtWidgets, etc.
- Módulos add-ons (agregados): Son códigos externos al desarrollo de Qt. Depende de las contribuciones de usuarios externos activos en el proyecto.
  Ejemplo de módulos add-ons: Qt3D, QtBluetooth, QtSensors, etc.

Elementos de Qt

Qt Creator

- Qt al ser un conjunto de bibliotecas, pueden ser integradas en cualquier IDE que lo permita. Incluso, es posible compilar programas Qt sin IDE.
- Qt Creator es un IDE multiplataforma para desarrollo de programas en C++ creado por el equipo de desarrollo de Qt. Está orientado principalmente a crear aplicaciones gráficas fácilmente utilizando las bibliotecas de desarrollo que provee Qt.

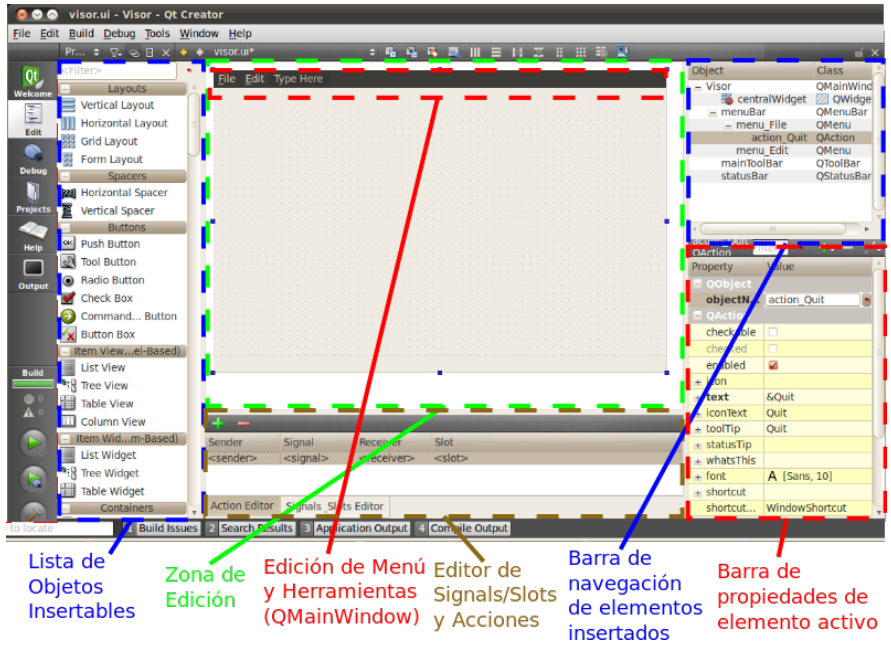


Elementos de Qt

Qt Creator

Qt Creator está compuesto por:

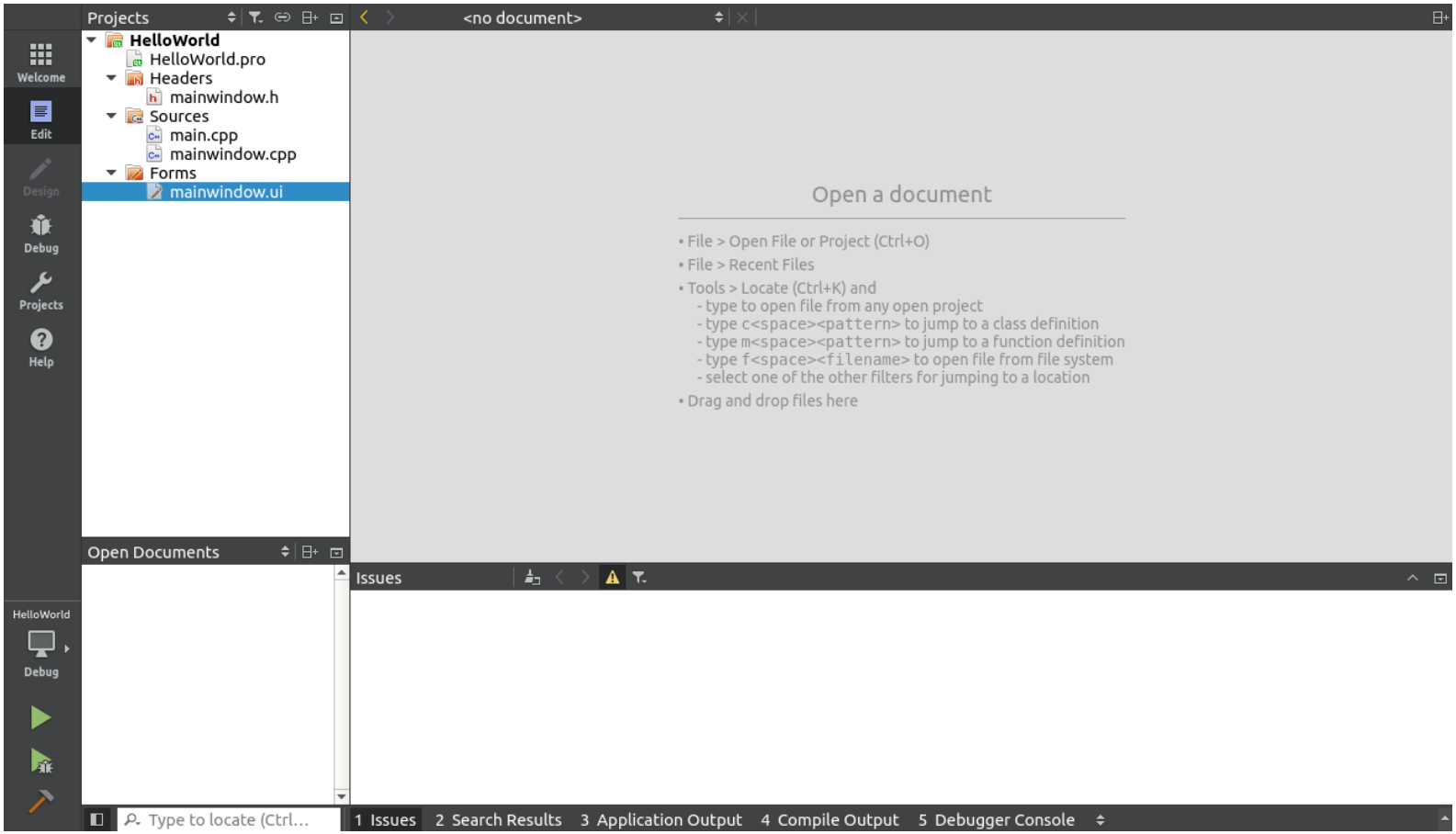
- Editor de texto: Éste posee varias funcionalidades, como autocompletado, revisión de sintaxis, identación automática, etc.
- Qt Designer: Para creación de interfaces gráficas de forma sencilla.
- Otras funcionalidades (Ej: Debug, conexión con sistemas de versionamiento (GIT, SVN, otros), etc.)



Un proyecto Qt está compuesto por los siguientes elementos:

- **Headers**: Conjunto de archivos con extensión .h que contienen las definiciones de las clases utilizadas en la aplicación.
- **Sources**: Conjunto de archivos con extensión .cpp que contienen las implementaciones de los métodos de clase y funciones utilizadas en el proyecto.
- **Forms**: Conjunto de archivos con extensión .ui. Son archivos con formato XML que contienen las características y disposición de los distintos elementos gráficos de cada pantalla del programa. Estos archivos pueden ser modificados utilizando el Qt Designer de forma gráfica o modificando directamente los archivos .ui.
- **Archivo .pro**: Es el archivo que contiene las directivas de compilación para el comando qmake, comando encargado de generar makefiles para desarrollos en Qt.

Programa: HelloWorld



HelloWorld.pro

```
#-------------------------------------------------
#
# Project created by QtCreator 2017-06-15T00:47:51
#
#-------------------------------------------------

QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = HelloWorld
TEMPLATE = app

# The following define makes your compiler emit warnings if you use
# any feature of Qt which as been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt 6.0.0


SOURCES += main.cpp\
        mainwindow.cpp

HEADERS  += mainwindow.h

FORMS    += mainwindow.ui
```

Headers/mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

// Se agrega la clase MainWindow actual al namespace Ui
namespace Ui {
class MainWindow;
}

// Clase que contiene definiciones de la ventana principal
// Hereda de QMainWindow
class MainWindow : public QMainWindow
{
    // Macro que convierte la clase en un QObject
    Q_OBJECT

public:
    // Constructor de ventana principal
    // explicit especifica que el contructor que en caso de usar casteo, es necesario que sea explícito
    // Una ventana, puede estar asociada a otra ventana/dialog/widget. Si existe esa asociación, parent !=0
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    // Puntero que se hará apuntar a la interfaz gráfica definida en mainwindow.ui
    // Definición de esta clase está en "ui_mainwindow.h"
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

Desarrollando en Qt

Ejemplo: Hola Mundo!

Sources/main.cpp

```cpp
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    // Se crea una instancia de aplicación
    QApplication a(argc, argv);
    // Dentro de la aplicación, se crea una instancia de MainWindow
    MainWindow w;
    // Se muestra la ventana principal
    w.show();

    // Se ejecuta la aplicación gráfica
    return a.exec();
}
```

Sources/mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

// Constructor de ventana principal, se inicializa con el contructor de
// QMainWindow y se inicializa el atributo privado ui
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    // Le agrega a la MainWindow, todos los elementos que se configuraron a través del QtDesig
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Desarrollando en Qt

Ejemplo: Hola Mundo!

Forms/mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget">
   <widget class="QLabel" name="label">
    <property name="geometry">
     <rect>
      <x>130</x>
      <y>20</y>
      <width>91</width>
      <height>17</height>
     </rect>
    </property>
    <property name="text">
     <string>Hola Mundo!</string>
    </property>
   </widget>
  </widget>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Desarrollando en Qt

Ejemplo: Hola Mundo!

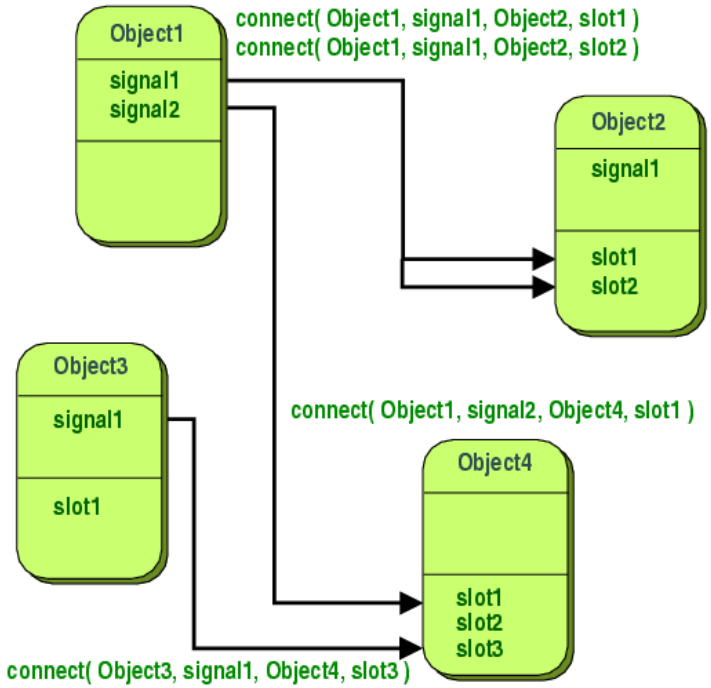
Forms/mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget">
   <widget class="QLabel" name="label">
    <property name="geometry">
     <rect>
      <x>130</x>
      <y>20</y>
      <width>91</width>
      <height>17</height>
     </rect>
    </property>
    <property name="text">
     <string>Hola Mundo!</string>
    </property>
   </widget>
  </widget>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Desarrollando en Qt

Signal y Slots

- Permiten la comunicación entre objetos/clases de tipo QObject.
- Una señal (signal) es **emitida** por una clase cuando un evento/cambio de estado particular ocurre.
- Una ranura (slot) es un método que es llamado en respuesta a una señal particular.
- Una clase puede tener tanto signals como slots.



Signal

- Son funciones de acceso público que **no deben ser implementadas y no retornan valores**
- Señales son emitidas por un objeto cuando su estado cambia

Slot

- Son funciones/métodos que pueden ser llamados normalmente. Su única diferencia es que se pueden conectar con signals.
- Un slot es llamado cuando la señal o las señales a las que está conectado son emitidas

Ejemplo de uso de Signal y Slot

```
#include <QObject>

class Counter : public QObject
{
    Q_OBJECT

public:
    Counter() { m_value = 0; }

    int value() const { return m_value; }

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};

void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}

Counter a, b;
    QObject::connect(&a, &Counter::valueChanged,
                     &b, &Counter::setValue);

    a.setValue(12);     // a.value() == 12, b.value() == 12
    b.setValue(48);     // a.value() == 12, b.value() == 48
```

Qt Widgets

- Equivalente a Swing de Java
- Permite acceder a los elementos gráficos incluidos en las bibliotecas Qt, tanto layouts como definición de ventanas, etc.
- Algunos elementos gráficos que heredan de QWidget:
  - QLabel
  - QPushButton
  - QGraphicsLayout
  - etc

Clase QPainter

- Permite realizar pintados de bajo nivel de elementos gráficos, generalmente sobre widgets.
- Su uso común es dentro del método paintEvent de un widget. paintEvent le dice a la clase qué debe hacer cuando se pinta el widget.

```
void SimpleExampleWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.setPen(Qt::blue);
    painter.setFont(QFont("Arial", 30));
    painter.drawText(rect(), Qt::AlignCenter, "Qt");
}
```

Ejemplo clase QPainter: Pintar un rectángulo

- Headers/widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QtGui>
#include <QtCore>

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = 0);
    void paintEvent(QPaintEvent *e);
    ~Widget();

private:
    Ui::Widget *ui;
};

#endif // WIDGET_H
```

- Sources/widget.cpp

```
#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
}

Widget::~Widget()
{
    delete ui;
}

void Widget::paintEvent(QPaintEvent *e)
{
    QPainter painter(this);
    painter.drawRect(10,10,100,30);

}
```

QGraphicsView y QGraphicsScene

- QPainter es una clase que permite dibujar a muy bajo nivel, por lo que no permite crear elementos gráficos con muchas propiedades.
- Podemos crear elementos gráficos con mayor autonomía al crearlos sobre una vista.
- QGraphicsView es donde se muestran los elementos gráficos.
- QGraphicsScene es la clase que controla cómo se desplegarán.
- Una QGraphicsView tiene asociada una QGraphicsScene que la controla.
- A la vista se le pueden agregar distintos elementos gráficos (QGraphicsItem) a través de la QGraphicsScene.
- Los QGraphicsItem agregados pueden ser los que provee Qt o personalizados heredando de QGraphicsItem.

Ejemplo QGraphicsScene: Pintar un elemento personalizado

- Headers/dialog.h

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>
#include <QtCore>
#include <QtGui>
#include <QGraphicsScene>
#include <QGraphicsRectItem>
#include "qascensor.h"

namespace Ui {
class Dialog;
}

class Dialog : public QDialog
{
    Q_OBJECT

public:
    explicit Dialog(QWidget *parent = 0);
    ~Dialog();

private slots:
    // Slots de botones usados en la clase Dialog
    void on_pushButton_clicked();
    void on_pushButton_2_clicked();

private:
    Ui::Dialog *ui;
    QGraphicsScene *scene;
    // Mi QGraphicItem
    QAscensor *asc;
    QTimer *timer = new QTimer(this);
};

#endif // DIALOG_H
```

- Sources/dialog.cpp

```
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog),timer(new QTimer(this))
{
    ui->setupUi(this);
    scene = new QGraphicsScene(this);
    ui->graphicsView->setScene(scene);
    asc = new QAscensor(10,10,100,30);
    scene->addItem(asc);
}
```

Método advance

- El método "advance" es un método slot de la clase QGraphicsScene y está asociado con la actualización de items en la escena.
- La actualización se hace en dos etapas:
  - En la primera etapa se notifica a todos los elementos de la escena que ésta va a cambiar (phase=0)
  - En la segunda etapa, se notifica a los items de la escena que se pueden mover (phase=1).
- El método advance, al ser llamado, llama a todos los métodos advance de cada uno de los elementos QGraphicsItem de la escena

Método advance

```
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog),timer(new QTimer(this))
{
    ui->setupUi(this);
    scene = new QGraphicsScene(this);
    ui->graphicsView->setScene(scene);
    asc = new QAscensor(10,10,100,30);
    scene->addItem(asc);
}

Dialog::~Dialog()
{
    delete ui;
    delete asc;
    delete scene;
    delete timer;
}

void Dialog::on_pushButton_clicked()
{
    connect(timer,SIGNAL(timeout()),scene,SLOT(advance()));
    timer->start(1000);
}

void Dialog::on_pushButton_2_clicked()
{
    timer->stop();
}
```

```
void QAscensor::advance(int phase){
    if(!phase) return;
    setPos(mapToParent(0,-(speed)));
}
```

Interfaces QML
Qt Quick

- QtQuick es el término utilizado para una tecnología de creación de interfaces de usuario utilizada en Qt
- Es alternativo a QWidget
- Se utiliza principalmente en el desarrollo de aplicaciones móviles
- Está compuesto principalmente por
  - Archivos QML: Archivos de etiquetado tipo JSON para definición de la interfaz
  - JavaScript: Para acciones producidas por la interfaz
  - Qt C++: Código detrás de la lógica del programa

Referencias I

[1] https://wiki.qt.io/Qt_for_Beginners/

[2] http://qmlbook.github.io

[3] https://www.qt.io/ide/

[4] http://doc.qt.io/qt-5/

[5] Ray Rischpater, Daniel Zucker. Beginning Nokia Apps Development, 2010

[6] J. Ryannel,J. Thelin. Qt5 Cadaques. Release 2015-03

[7] https://www.youtube.com/playlist?list=PL2D1942A4688E9D63 (Recomendado)

[8] Marcos Zúñiga, Eduardo García. Presentación Seminario de Programación - Interfaces en QT. Octubre, 2016