UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Electrónica

Desarrollos en Qt

ELO329: Diseño y Programación Orientados a Objetos

Departamento de Electrónica

Universidad Técnica Federico Santa María

Desarrollos en Qt

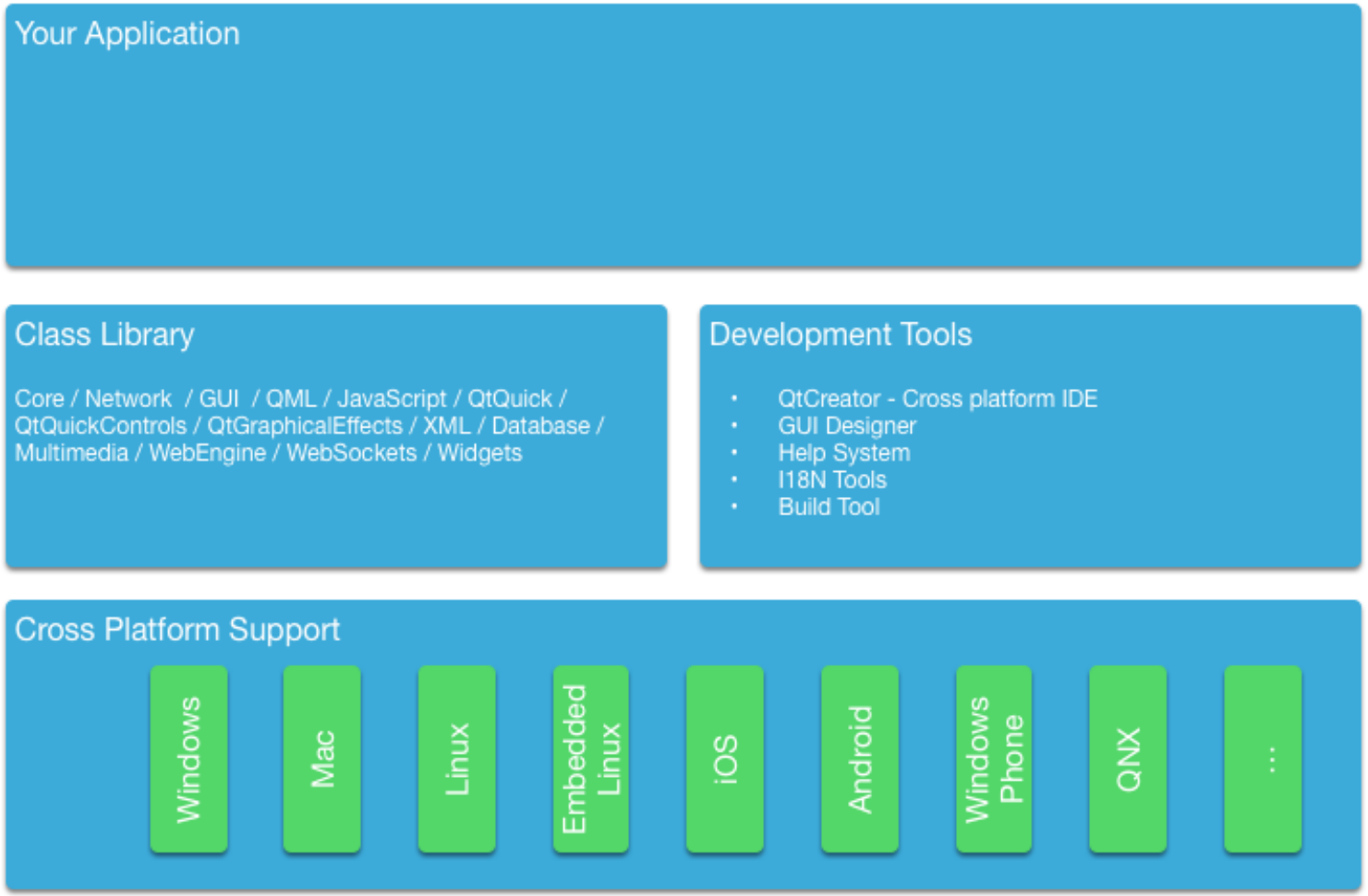
- 1.- Introducción
- 2.- Elementos de Qt
- 3.- Desarrollando en Qt
- 4.- Interfaces de usuario QML

¿Qué es Qt?

- Qt (pronunciado como “çute”) es un framework de desarrollo de aplicaciones con múltiples bibliotecas
- Su primera versión data de 1995
- 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)
- Puede encontrar información de la última versión en el siguiente enlace: https://www.qt.io/

¿Qué es Qt? Arquitectura

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



En este curso usaremos Qt5, LTS

Elementos de Qt: Módulos Qt

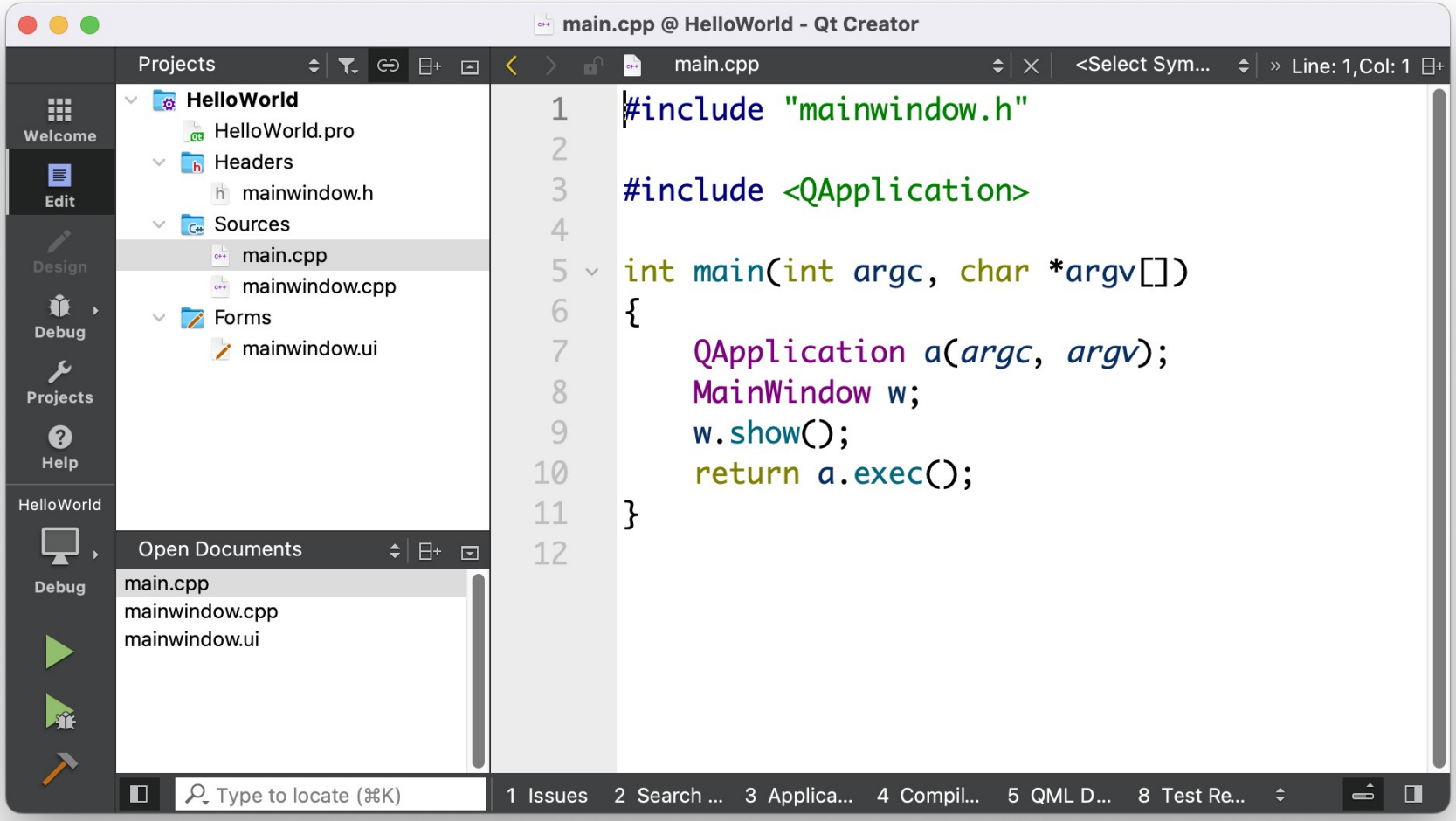
- Existen dos tipos de módulos en Qt:
  - Módulos Core-Essential: Son los que dan forma a Qt. Son parte del desarrollo de estas bibliotecas.
    Ejemplos de módulos core-essential: QtCore, QtGui, QtWidgets, etc.
  - Módulos add-ons (agregados): Son códigos externos al desarrollo de Qt. Depende de las 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 creado por el equipo de desarrollo de Qt para el desarrollo de programas en C++. Está orientado principalmente a crear aplicaciones gráficas fácilmente utilizando las bibliotecas de desarrollo que provee Qt.

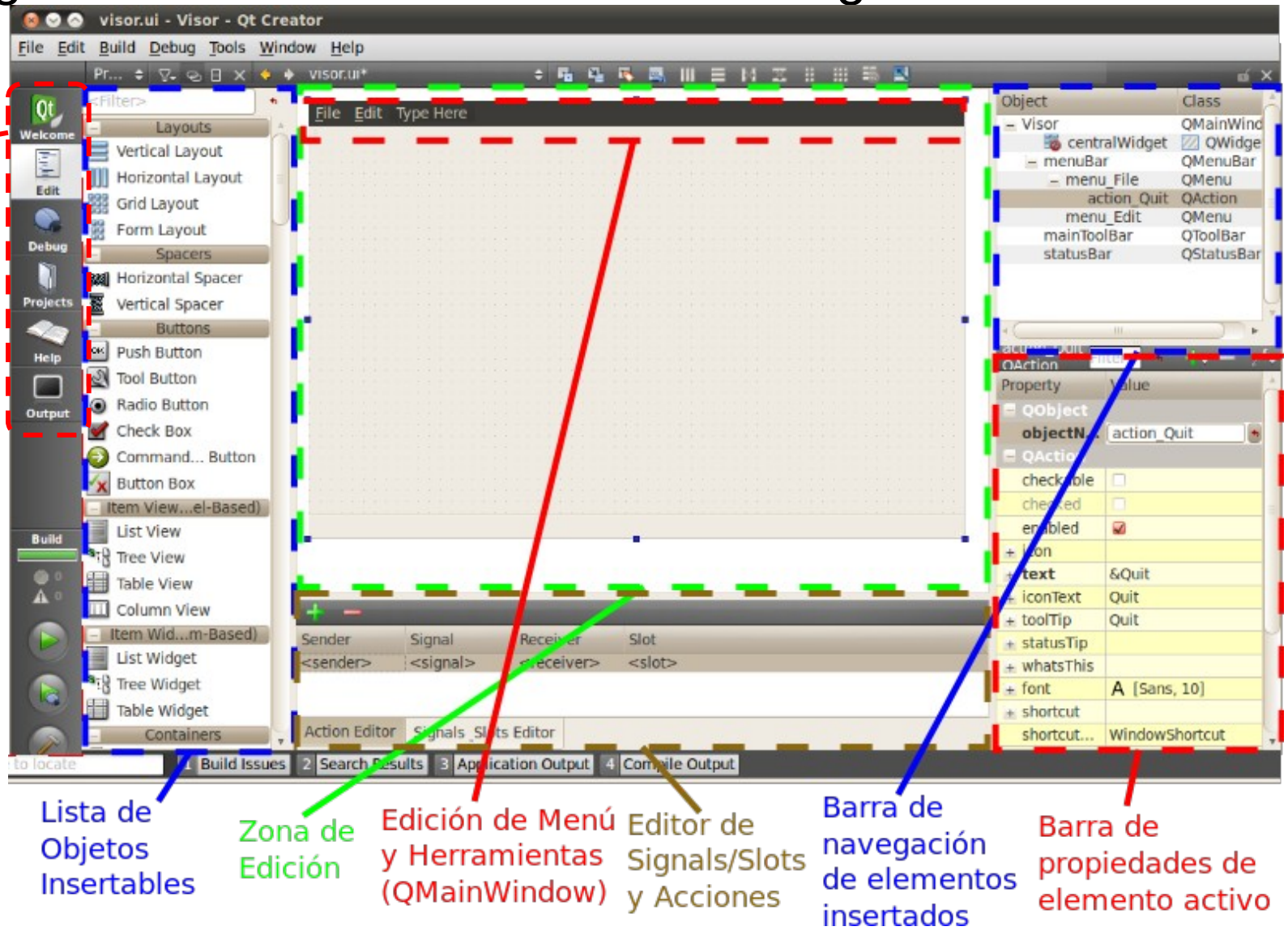
Selector de Modos de Qt: Editor

- Modo editor: Éste posee varias funcionalidades, como autocompletado, revisión de sintaxis, identación automática, etc.



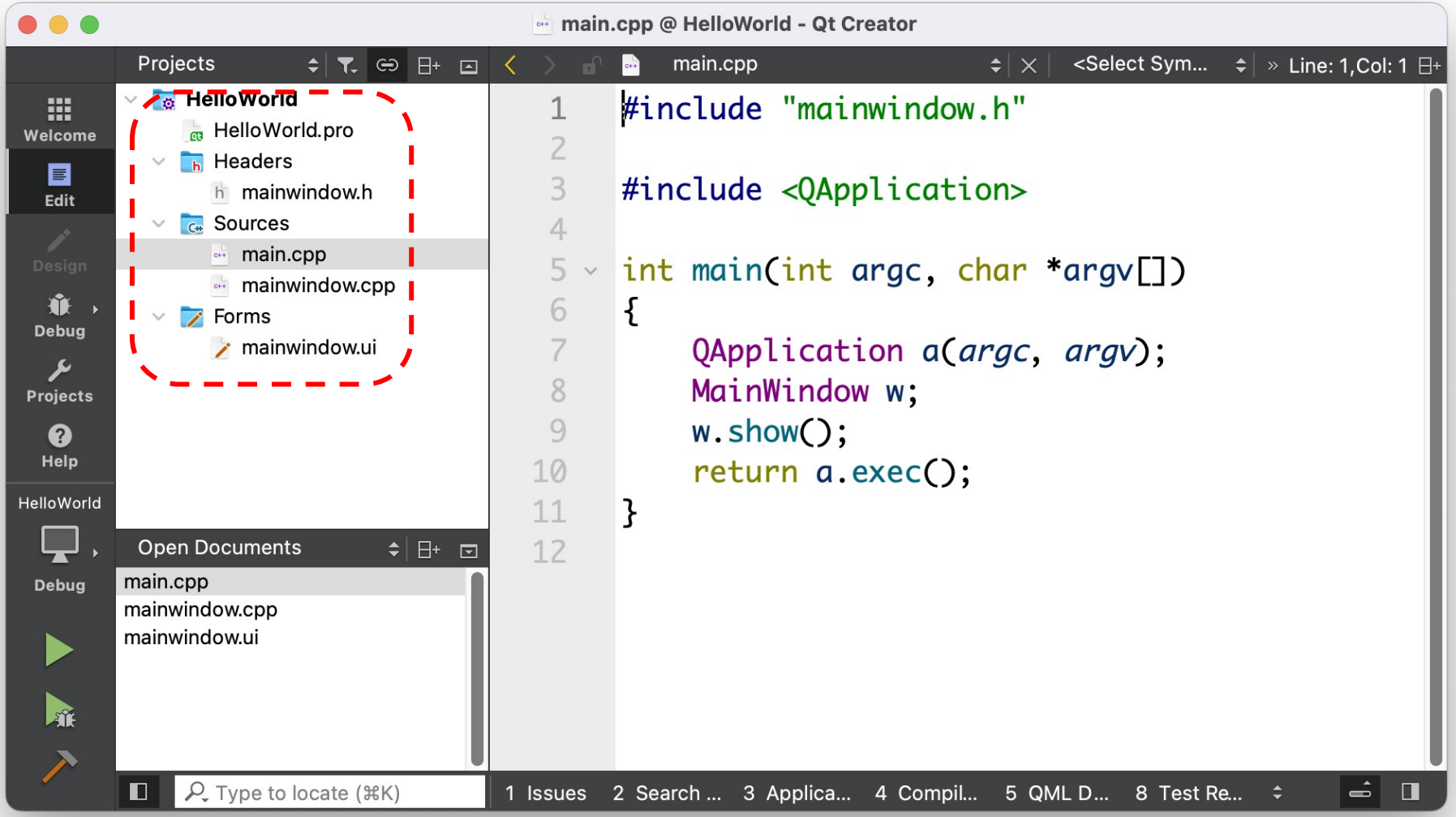
Selector de Modo de Qt: Diseño

- Qt Designer: Para creación de interfaces gráficas de forma sencilla.



Otras funcionalidades Ej: Debug, conexión con sistemas de versionamiento (GIT, SVN, otros), etc.

Ejemplo: Hola Mundo!



Creando un Proyecto

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.

Hola Mundo

- Revisar Qt Creator, en particular archivos:
- HelloWorld.pro
- Headers/mainwindow.h
- Sources/main.cpp
- Sources/mainwindow.cpp
- Forms/mainwindow.ui

Hola mundo: 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
```

Hola Mundo: source/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();
}
```

Hola Mundo: Source/mainwindows.cpp

```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 QtDesigner
    ui->setupUi(this);
}

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

Hola Mundo: Forms/mainwindow.ui

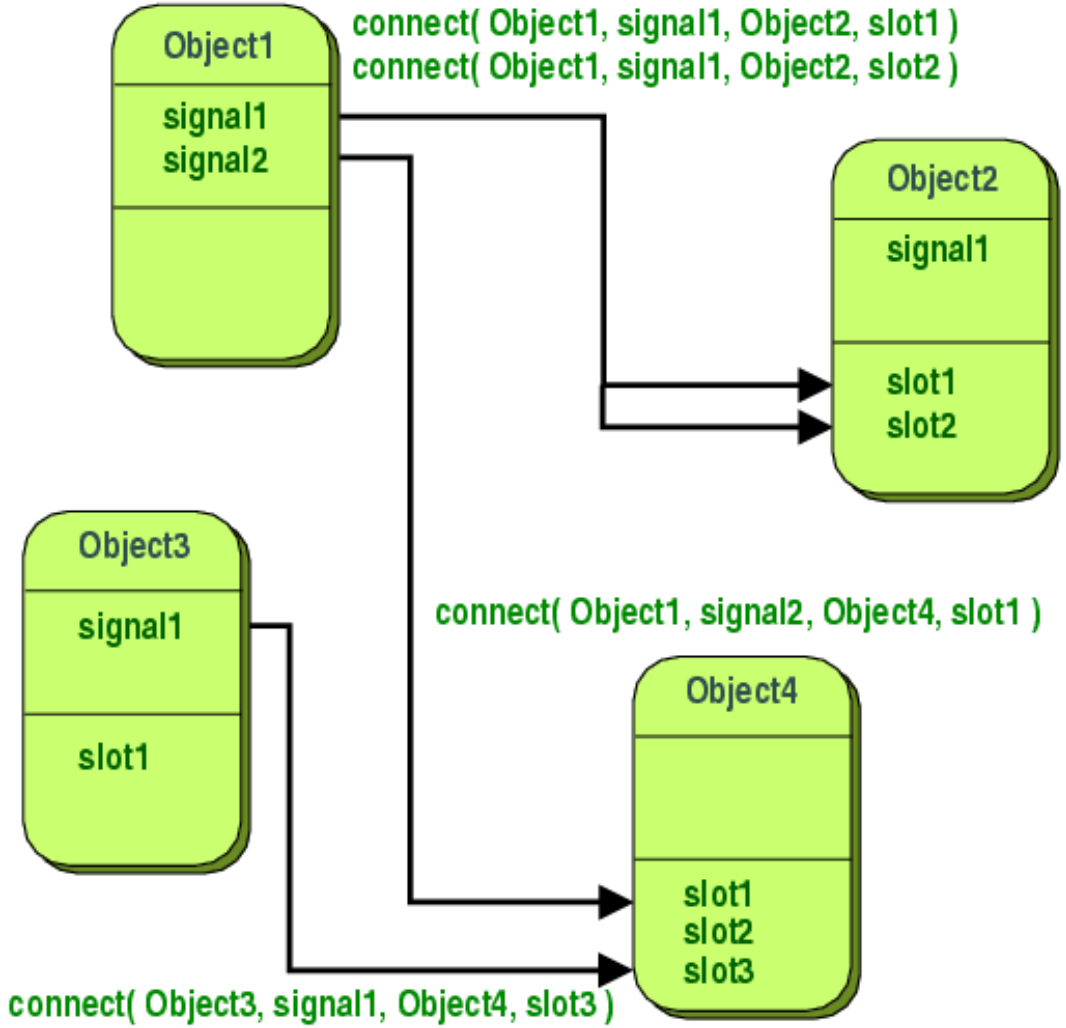
```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle">
   <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>
```

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>
```

Señales y Ranuras (Signals and Slots)

- Permiten la comunicación entre objetos de tipo QObject.
- Una señal (signal) es emitida por un objeto para señalar un evento o cambio de interés particular.
- Una ranura (slot) es un método que es llamado en respuesta a una señal particular.
- Una clase puede tener tanto signals como slots.
- Connect permite asociar slots a signals



Señales y Ranuras (Signals and Slots)

Signals

- Son funciones/métodos de acceso público que no deben ser implementadas y no retornan valores
- Señales son emitidas por un objeto para señalar algún evento de interés (definido por el programador).

Slots

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

Signals & Slots: Ejemplo vía programación

```
#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
```

Creando GUI en Qt

Qt Widgets

- Equivalente a Node de JavaFX
- 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

Creando objetos arbitrarios en una aplicación

- Una forma de hacerlo es vía la clase QGraphicsView
  - Esta clase provee un widget (objeto gráfico) para desplegar contenido de un QGraphicsScene
- La clase QGraphicsScene provee una superficie para manejar un gran número de ítems gráficos 2D.
  - Entre estos ítems gráficos que se pueden agregar vía su método addItem están QGraphicsPolygonItem, QGraphicsEllipseItem, QGraphicsLineItem, QGraphicsPathItem, QGraphicsRectItem, QGraphicsTextItem
  - Todas estas clases incluyen las funciones setBrush(..) y setPen(..) para definir cómo se dibujará su contorno e interior.
- Otra forma de dibujar objetos arbitrarios es heredando de Qwidget como en el próximo ejemplo.

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: Dibujar 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);

}
```

Referencias

- [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
- [8] Marcos Zúñiga, Eduardo García. Presentación Seminario de Programación - Interfaces en QT. Octubre, 2016
- [9] Aportes de Patricio Olivares en versiones previas de ELO329