Введение в сценарии

1. Для чего используют сценарии?

2. Поддержка JavaScript(JS) в Qt

JS - поддерживает:

- концепцию *ООП*,
- использует мета объектную модель,
- создание и использование классов,
- управление исключениями,
- синтаксис похож на *C++*.

JS встроен в модуль *QtQml* и является средой, обеспечивающей встроенную поддержку для сценариев в приложениях *Qt/C++.*

Благодаря тому, что поддержка языка сценариев встроена в библиотеку *Qt*, с ее помощью можно:

- управлять *Qt*-программами без их перекомпиляции,
- вносить в них изменения,
- реализовывать тестовые сценарии,
- выполнять настройки приложения для специфических запросов пользователей.

3. Принцип поддержки JS

Для реализации поддержки языка сценариев задействованы такие механизмы, как

- сигналы и слоты,
- объектные иерархии,
- свойства объектов (*properties*)

Чтобы сделать класс, унаследованный от *QObject*, доступным для использования в языке сценариев *JS*, необходимо наличие метаинформации, а, значит, в определении класса для его свойств должны использоваться макросы:

Q_ OBJECT,
Q_PROPERTY

Любой метод класса можно сделать видимым для языка сценария при помощи макроса ***Q_ INVOKABLE*** :

Q_INVOKABLE void scriptAccessableMethod();

4. Свойства объектов в Qt (Properties)

Свойства - это поля класса, для которых обязательно должны существовать методы чтения, с помощью которых можно получать доступ к атрибутам объектов извне (например, из языка сценариев). Свойства также широко задействованы в визуальной среде разработки пользовательского интерфейса *Qt Designer*.

«Свойство» задается при помощи макроса ***Q_PROPERTY*** и в общем виде выглядит так:

```
Q_PROPERTY ( type name          // тип и имя св-ва
             READ getfunction // имя метода чтения
            [WRITE setfunction]    // имя метода записи
            [RESET resetfunction]  // имя метода сброса значения
            [DESIGNABLE bool]      // отображаемость в QtDesigner
            [SCRIPTABLE bool]      // доступность для языка сценариев
            [STORED bool]          // сериализация, запоминаемость при сохранении объекта
)
```

*Из программы можно изменить значение свойства:

pobj->setProperty (“readOnly”, true);

А так можно получить текущее значение:

bool bReadOnly = pobj->property(“readOnly”).toBool();

*Чтобы узнать сразу все свойства любого объекта и их значения, необходимо получить при помощи метода **propertyCount()** класса **QMetaObject** количество свойств. Затем в цикле получить для каждого индекса объект свойства и вызвать из него метод **typeName()** - для типа свойства и метод **name()** - для имени свойства.

5. Пример класса Qt, доступного для JS

Зачем мы определили эти методы как слоты?

```
class MyClass : public QObject {

  Q_OBJECT
  Q_PROPERTY(bool readOnly WRITE setReadOnly READ isReadOnly)
private:
       bool m_bReadOnly;      // логический атрибут(поле)
public:
       MyClass(QObject* pobj = 0) : QObject(pobj) , m_bReadOnly(false){
      }
public slots:                          // слоты доступа к полю m_bReadObject
       void setReadOnly(bool bReadOnly) {
              m_bReadOnly = bReadOnly;
              emit readOnlyStateChanged();
       }
       bool isReadOnly() const {
            return m_bReadOnly;
        }
signals:
       void readOnlyStateChanged();
};
```

Чтобы методы класса были видимыми для языка сценариев - их необходимо определить как слоты.

6. Взаимодействие объектов приложения со сценариями

Манипулировать логическими значениями объекта класса из языка сценариев при помощи слотов можно так:

Стиль C++

```
myObject.setReadOnly(true);  // слот объекта класса
print("myCbject is read only:" + myObject.isReadOnly());
```

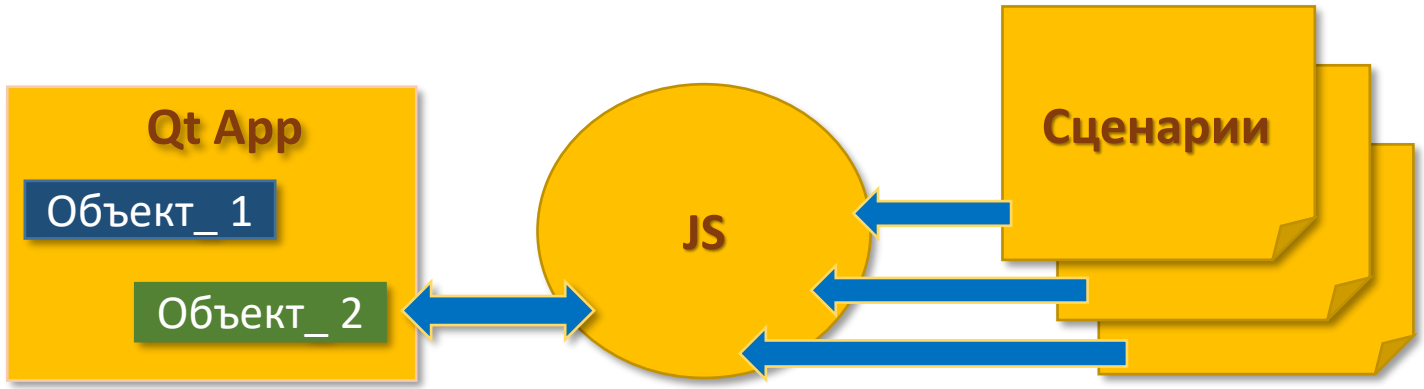
Манипулировать значением объекта класса из языка сценариев в «стиле JS» привычнее с помощью свойств (property):

Стиль JS

```
myObject.readOnly = true;  // свойство объекта класса
print("myObject is read only:" + myObject.readOnly);
```

Мы получаем доступ из языка сценариев **только к конкретным объектам приложения**, но не сразу ко всей функциональности библиотеки *Qt* или к функциональным особенностям всего приложения.

Сценарии получают доступ к *Объект_ 2*, как будто он встроен в сам язык *JS*. Этот объект может быть либо объектом приложения, либо интерфейсным объектом для управления приложением при помощи сценария.



7. Класс сторонней библиотеки

Для класса сторонней библиотеки (не *Qt*) необходимо реализовывать класс обертки (*wrapper*), унаследованный от *QObject*. Этот класс будет агрегировать объект нужного вам класса и предоставлять делегирующие методы.

```
class NonQtClass {
private:
        bool m_bReadOnly;  // атрибут класса
public:
       NonQtClass () :m_bReadOnly(false) {
       }
       void setReadOnly(bool bReadOnly) {
           m_bReadOnly = bReadOnly;
       }
       bool isReadOnly() const {
           return m_bReadOnly;
       }
};
```

```
class MyWrapper : public QObject {
   Q_OBJECT
   Q_PROPERTY(bool readOnly WRITE setReadOnly READ isReadOnly)
private:
       NonQtClass m_nonQtObject;  // агрегация nonQt-класса

public:
       MyWrapper(QObject* pobj = 0): QObject(pobj) {
        }

public slots:
       void setReadOnly(bool bReadOnly) {
            m_nonQtObject.setReadOnly(bReadOnly);
            emit readOnlyStateChanged();
        }
       bool isReadOnly() const {
            return m_nonQtObject.isReadOnly();
       }
signals:
      void readOnlyStateChanged();
};
```

8. “Hello, JavaScript”

```
#include <QtWidgets>
#include <QJSEngine>

int main(int argc, char** argv) {
   QApplication app(argc, argv);
   QLabel* plbl = new QLabel;

   QJSEngine scriptEngine;  //объект исполнения сценария
   QJSValue scriptLbl = scriptEngine.newQObject(plbl);
   scriptEngine.globalObject() .setProperty(“lbl”, scriptLbl);

   // передаем строку для исполнения интерпретатором яз. сценария
   // здесь мы работаем со свойствами и методами объекта класса QLabel
   QJSValue value = scriptEngine.evaluate(“lbl.text = ‘Hello, JavaScript! ‘ ”);
   value = scriptEngine.evaluate(“lbl.show()”);

  if (value.isError()) {
        qDebug() << "Error:" << value.toString();
  }

  return app.exec();
}
```

Среда для исполнения команд сценария

Контейнер для хранения типа данных JS (результата исполнения сценария)

Значение value имеет произвольный тип. (Qvariant)

QT += qml

#include <QJSEngine>

*Если *интерпретация* закончилась неуспешно - например, в случае синтаксической ошибки, возвращенное значение будет содержать ошибку.

*Принадлежность значения к конкретному типу можно проверить вызовом методов isT().

9. Резюме

Реализация приложений с поддержкой языка сценариев делает возможным *динамическое расширение* вашего приложения и его изменение под конкретные требования без необходимости перекомпиляции.

Библиотека *Qt* при помощи модуля *JavaScript* предоставляет возможность для организации поддержки языка сценариев в ваших программах. Этот модуль содержит интерпретатор языка сценариев и классы C++ для его поддержки.

За счет использования мета объектной модели *Qt* любой подкласс *QObject* можно сделать доступным для языка сценариев.

10. Практика

- Изучить работу приложения: «QtScriptBasicDemo»

11. Домашка #2

1. Создать пользовательский nonQt-класс.
2. Создать “Qt-обертку” nonQt-класса.
3. Определить в классе-обертке не менее 3-х свойств (Q_PROPERTY)
4. Создать объект пользовательского класса (задать его свойства при создании). Получить список свойств и их значения.

1. Придумать небольшое приложение Qt/C++ с элементами управления.
2. Расширить функционал приложения за счет поддержки скриптов JS в созданном вами приложении. (см. пример из «Практика» 10 слайд).