Programmation GUI avec Qt
Qt Quick et QML

Thierry Vaira

BTS SN Option IR

v.1.2 - 19 mars 2019

Sommaire

Préambule

- Cette présentation de QML a été réalisée à partir de la documentation de Qt (`http://doc.qt.io/`).
- Les notions de base présentées dans ce document contiennent les liens vers les parties à approfondir.

✍ Une liste d'exemples et de tutoriels est fournie à la fin de ce document.

La Salle Avignon

Sommaire

1 Introduction

- Qu'est-ce que QML ?
- Qu'est-ce que Qt QML ?
- Qu'est-ce que Qt Quick ?
- Qu'est-ce que Qt Quick Controls ?
- Les versions de Qt Quick
- Qu'est-ce que Qt Creator ?

Qu'est-ce que QML ?

- QML (*Qt Modeling Language*) est un langage de balisage d'interface utilisateur.
- C'est un langage déclaratif (similaire à CSS et JSON) pour la conception d'applications centrées sur l'interface utilisateur.
- Le langage QML et le moteur d'exécution sont fournis par le module *Qt QML*.
- Le module *Qt Quick* fournit une API QML pour créer des interfaces utilisateur avec le langage QML.
- Le module *Qt Quick Controls* fournit un ensemble de contrôles visuels pour créer des interfaces complètes dans *Qt Quick*.
- Lien : `http://doc.qt.io/qt-5/qmlapplications.html`

✍ QML est un langage déclaratif qui permet de décrire les interfaces utilisateur en termes de composants visuels et d'interaction entre elles. Les fichiers écrits en QML portent l'extension *.qml*.

Qu'est-ce que Qt QML ?

- Le module *Qt QML* fournit un cadre (*framework*) pour développer des applications avec le langage QML .
- Il fournit donc le langage QML et l'infrastructure du moteur pour l'exécution des applications QML.
- L'infrastructure QML permet au code QML de contenir du JavaScript et au code QML d'interagir avec le code C++.
- JavaScript permet au code QML de contenir la logique de l'application.
- Lien : `https://doc.qt.io/qt-5/qtqml-index.html`

Le moteur QML fournit un environnement JavaScript présentant certaines différences par rapport à celui fourni par un navigateur Web. Le code JavaScript peut être intégré directement dans les fichiers QML ou importé de fichiers `.js`.

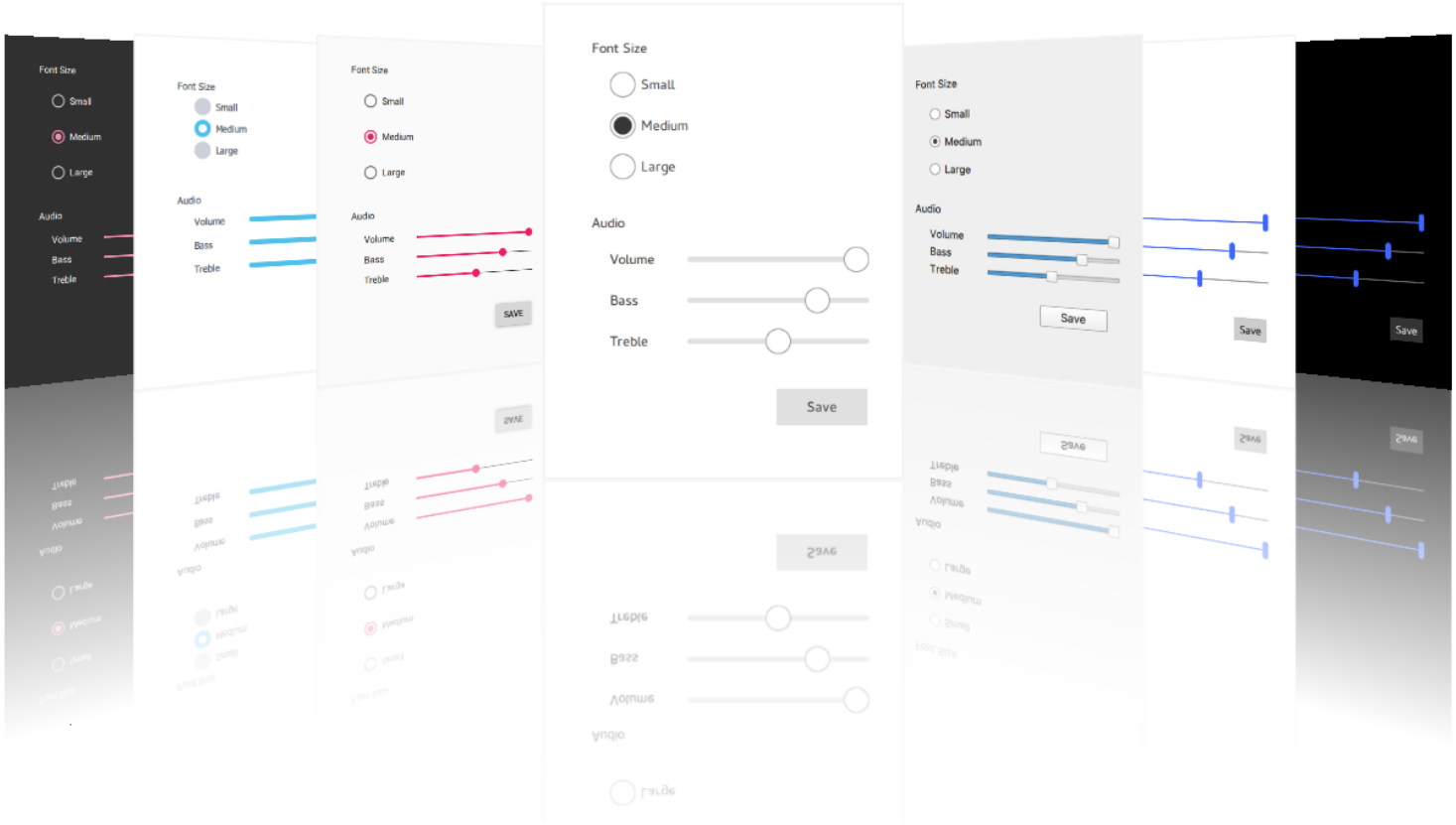
Qu'est-ce que Qt Quick ?

- Il fournit un moyen de créer une interface utilisateur graphique (GUI) personnalisables et dynamiques avec des effets de transition fluides.
- *Qt Quick* inclut un langage de script déclaratif appelé QML (*Qt Modeling Language*).
- *Qt Quick* est fait partie de Qt depuis la version Qt 4.7.
- Lien : `https://doc.qt.io/qt-5/qtquick-index.html`

✍ *Qt Quick* est souvent utilisé pour les applications mobiles où la saisie tactile, les animations fluides et l'expérience utilisateur sont essentielles.

Qu'est-ce que Qt Quick Controls ?

- *Qt Quick Controls* fournit un ensemble de contrôles pouvant être utilisés pour créer des interfaces complètes dans *Qt Quick*.
- *Qt Quick Controls* est livré avec une sélection de styles personnalisables.
- Lien : `https://doc.qt.io/qt-5/qtquickcontrols-index.html`



Les versions de Qt Quick

Qt	QtQuick	QtQuick.Controls, QtQuick.Controls.Material, QtQuick.Controls.Universal, QtQuick.Templates
5.7	2.7	2.0
5.8	2.8	2.1
5.9	2.9	2.2
5.10	2.10	2.3
5.11	2.11	2.4
...	...	...

Qu'est-ce que Qt Creator ?

- Qt Creator est un IDE (*Integrated Development Environment*) ou EDI (Environnement de Développement Intégré) dédié au développement d'applications Qt.
- Qt Creator intègre notamment l'outil *Qt Designer* qui permet de créer des interfaces graphiques.
- Il fournit aussi des assistants (*wizard*) pour créer des projets « types » : *Qt Widget* ou *Qt Quick*.
- Qt Creator lit en entrée un fichier de projet `.pro` qui décrit le contenu d'un projet Qt (modules utilisés, fichiers sources, ressources, options, ...).
- Lien : `http://doc.qt.io/qtcreator/`

Sommaire

2 Premier pas

- Premier document QML
- Créer une application Qt Quick
- Qt Quick Designer
- Fabriquer et exécuter

Premier document QML

- Chaque document QML comprend deux parties : une section d'importation de **modules** et une section de déclaration d'**objet**(s) (hierarchisés).
- On fixe ensuite des valeurs aux **propriétés** de chaque objet.
- Une simple déclaration d'objet peut être un **rectangle** avec du **texte** centré dessus :

```
import QtQuick 2.7 // importer le module Qt Quick

Rectangle { // définition d'un object Rectangle qui sera ici la fenêtre
    width: 200; height: 200 // 200x200 pixels
    color: "white" // couleur de fond en blanc
    Text {
        text: "Hello World" // définition d'un texte à l'intérieur du rectangle
        color: "#ff0000" // couleur du texte ici en rouge
        font.pointSize: 24 // taille de la police
        font.bold: true // en gras
        anchors.centerIn: parent // le texte sera centré dans la fenêtre
    }
}
```

Test QML n°1

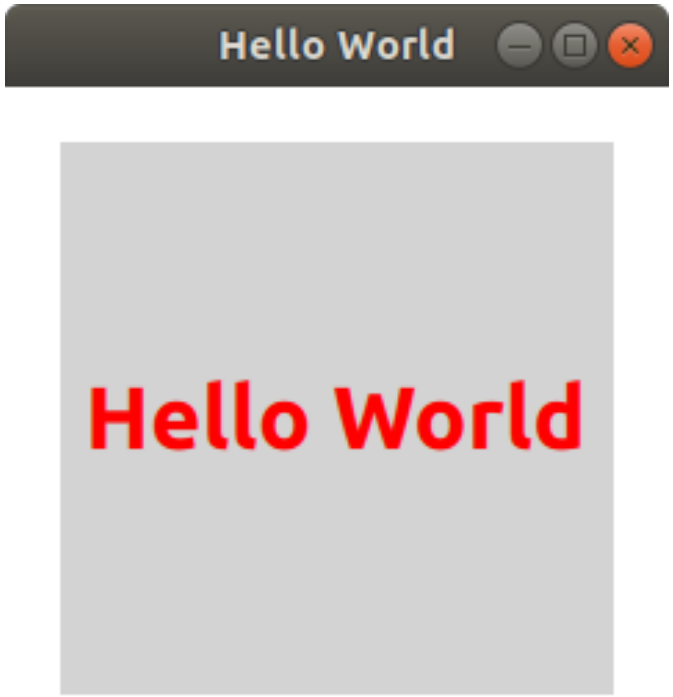
- Qt fournit un outil pour visualiser le rendu d'un document QML : `qmlviewer` pour Qt 4.x et `qmlscene` pour Qt 5.

Une hiérarchie d'objets : Fenêtre > Rectangle > Texte

- Un document QML définit une **hiérarchie d'objets** avec un seul objet racine (ici `Window`).
- On va maintenant créer une fenêtre `Window` et placer le `Rectangle` à l'intérieur :

```
import QtQuick 2.7
import QtQuick.Window 2.2 // pour le type Window

Window {
    visible: true
    width: 240
    height: 240
    title: qsTr("Hello World") // un titre pour la fenêtre
    Rectangle {
        width: 200
        height: 200
        color: "lightgray"
        anchors.centerIn: parent
        Text {...}
    }
}
```

Test QML n°2



- Lien : `https://doc.qt.io/qt-5/qmlfirststeps.html`

Gérer la souris

- La gestion de la souris est réalisée dans un élément `MouseArea`.
- Un objet `MouseArea` est un élément invisible généralement utilisé avec un élément visible afin de gérer la souris pour cet élément.

Qt est basée sur la programmation évènementielle. Ces événements sont des signaux et ces signaux sont traités par des gestionnaires de signaux (ici `onClicked`) :

```
Window { ...
    Rectangle {
        width: parent.width // propriété liée à son parent
        height: parent.height / 2
        Text {...}
        MouseArea { // élément invisible afin de gérer la souris pour l'objet
             Rectangle
            anchors.fill: parent
            onClicked: parent.color = "blue" // gestionnaire d'évènement pour le "
                clic"
        }
    }
}
```

Programmer en JavaScript

On peut aussi utiliser du JavaScript dans les gestionnaires de signaux :

```
Window { ...
    Rectangle {
        width: parent.width // propriété liée à son parent
        height: parent.height / 2
        Text {...}
        MouseArea { // élément invisible afin de gérer la souris pour l'objet
             Rectangle
            anchors.fill: parent
            onClicked: { // gestionnaire d'évènement pour le "clic"
                if(parent.color == "#0000ff")
                    parent.color = "#ffffff" // blanc ou "white"
                else
                    parent.color = "#0000ff" // bleu ou "blue"
            }
        }
    }
}
```

Intégrer des composants prédéfinis

Qt Quick Controls fournit un ensemble de composants à intégrer dans les interfaces, comme le **Button** :

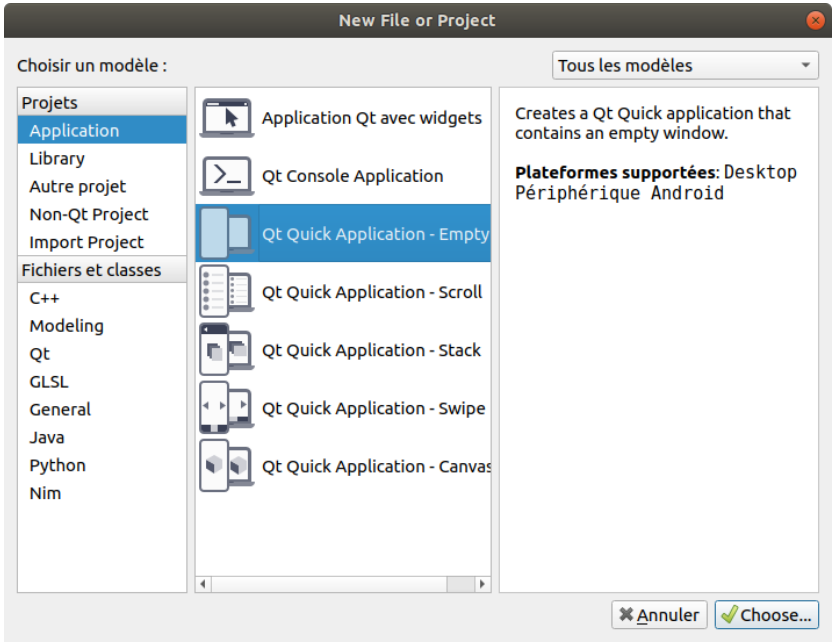
```
...
import QtQuick.Controls 2.1 // pour Button

Window { ...
    Rectangle {
        ...
        Button {
            text: qsTr("Valider")
            anchors.bottom: parent.bottom
            anchors.horizontalCenter: parent.horizontalCenter
            onClicked: {
                console.log(qsTr('Vous avez cliqué sur le bouton'))
                ... // affiche le texte "Ok" en vert
            }
        }
    }
}
```

Test QML n°3

- Lien : `http://doc.qt.io/qt-5/qtquickcontrols-overview.html`

Créer une application Qt Quick

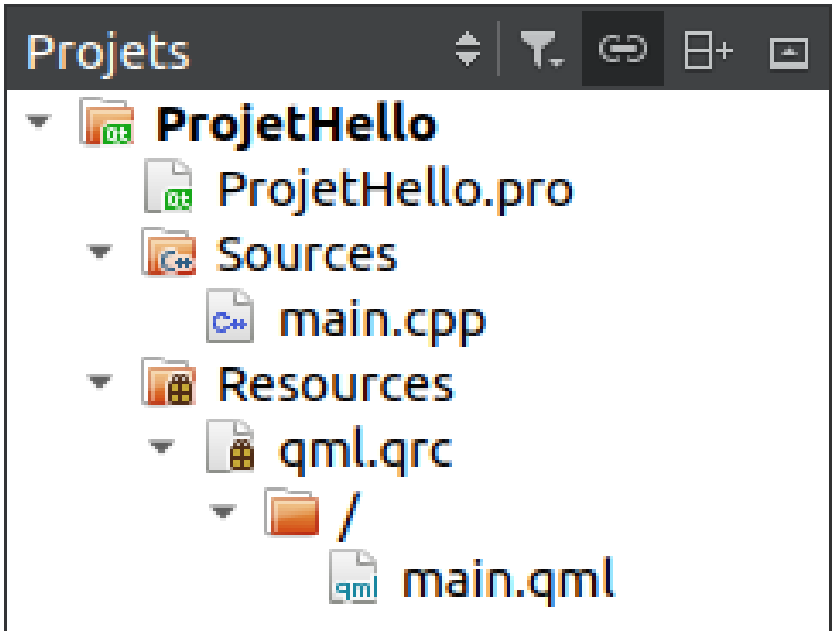
- Avec *Qt Creator*, Sélectionnez Fichier > Nouveau fichier ou projet > Application > **Qt Quick Application - Empty** > Choose.
- Lien : `http://doc.qt.io/qtcreator/quick-projects.html`



✍ L'assistant va vous guider pour créer un projet d'application *Qt Quick* pouvant contenir du code QML et du code C++. Ici par exemple, l'application pourra être déployée sur des plates-formes cibles de bureau (*Desktop*) ou *Android*.

Arborescence d'un projet Qt Quick

- Une fois le projet créé, on obtient le squelette d'une application *Qt Quick* :



✍ Le fichier `main.qml` correspond au document QML.

Le fichier de projet `.pro`

- Le fichier de projet a été généré automatiquement et il contient au minimum :

```
TEMPLATE = app

QT += qml quick

CONFIG += c++11

SOURCES += main.cpp

RESOURCES += qml.qrc
```

✍ Par la suite, on ajoute souvent des modules complémentaires dans la variable `QT`, comme le module `quickcontrols2` par exemple.

Le fichier main.cpp

- QQmlApplicationEngine fournit un moyen pratique de charger une application à partir d'un seul fichier QML.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    return app.exec();
}
```

✍ Contrairement à QQuickView, QQmlApplicationEngine ne crée pas automatiquement une fenêtre racine Window.

Le fichier main.qml

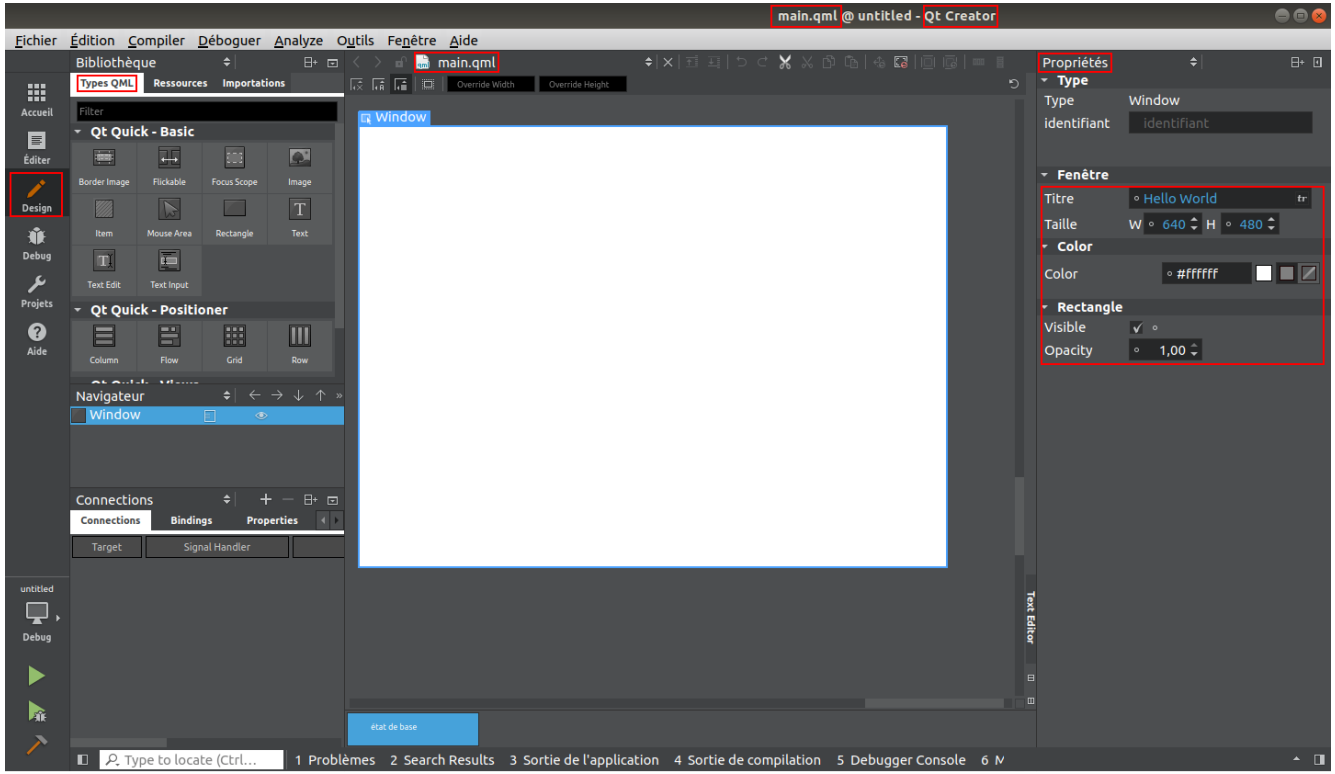
- Le document QML de base généré par l'assistant contient (pour l'instant) un simple objet `Window` :

```
import QtQuick 2.9
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
}
```

Qt Quick Designer

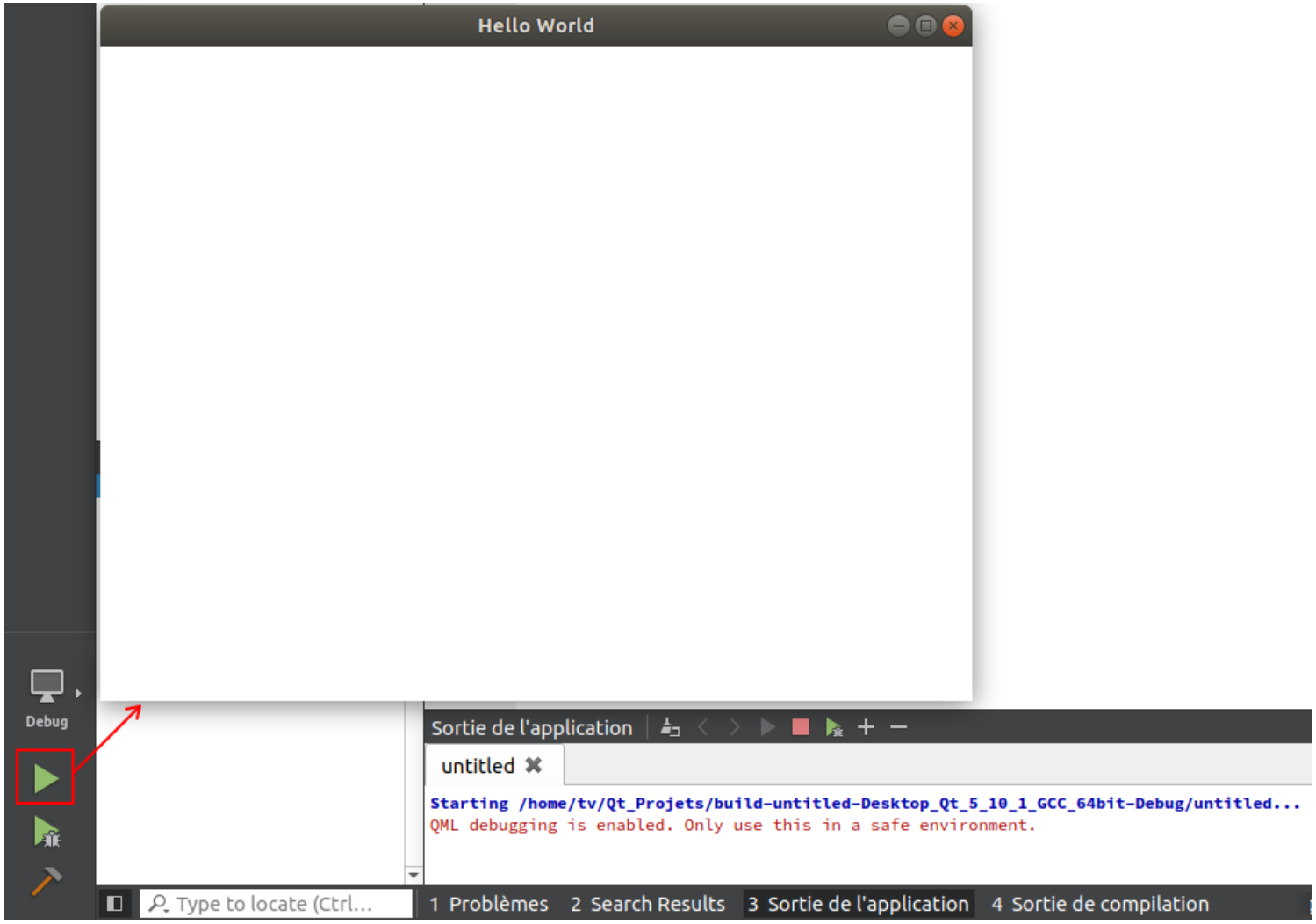
L'outil *Designer* de *Qt Creator* est adapté pour réaliser les interfaces en QML. On trouve :

- à gauche les types d'objets QML à intégrer au document par glisser-déposer
- au centre le rendu du document QML (ici la fenêtre)
- à droite les propriétés à éditer des objets QML du document



Fabriquer et exécuter

Dans *Qt Creator*, il suffit de cliquer sur la flèche verte (Exécuter) ou Ctrl-R :



Sommaire

3 Notions de base

- Type d'objet QML
- Les éléments visuels
- Les éléments de texte
- Le positionnement
- Javascript
- Les sous-modules Qt Quick
- Qt Quick Controls
- Les layouts
- Les boîtes de dialogue
- Les animations
- Les états
- Fenêtre d'application
- Chargement dynamique (`Loader`) et JavaScript

Type d'objet QML

- Un type d'objet QML est un type à partir duquel un objet QML peut être instancié. Par exemple, `Rectangle` est un type d'objet QML.
- Les types d'objet QML sont dérivés de `QtObject` et sont fournis par les modules QML.
- Les applications peuvent importer (`import`) ces modules pour utiliser les types d'objets qu'ils fournissent.
- Le module *QtQuick* fournit les types d'objet les plus courants nécessaires à la création d'interfaces utilisateur dans QML.
- Tous les types d'objet fournis par *QtQuick* sont basés sur le type `Item`, lui-même dérivé de `QtObject`.

☞ Liste : `http://doc.qt.io/qt-5/qtquick-qmlmodule.html`

Les éléments visuels dans QML

Le module *QtQuick* fournit des types primitifs graphiques :

- `Item` : le type visuel de base QML
  `http://doc.qt.io/qt-5/qml-qtquick-item.html`
- `Rectangle` : peint un rectangle
  `http://doc.qt.io/qt-5/qml-qtquick-rectangle.html`
- `Image` : affiche une image
  `http://doc.qt.io/qt-5/qml-qtquick-image.html`
- `Canvas` : fournit un élément de dessin 2D
  `http://doc.qt.io/qt-5/qml-qtquick-canvas.html`

☞ `http://doc.qt.io/qt-5/qtquick-usecase-visual.html`

L'objet Rectangle

```
Rectangle {
    width: 100
    height: 100
    //color: "steelblue"
    border.color: "black"
    border.width: 5
    radius: 10
    rotation: 90
    gradient: Gradient {
        GradientStop { position: 0.0; color: "lightsteelblue" }
        GradientStop { position: 1.0; color: "blue" }
    }
}
```

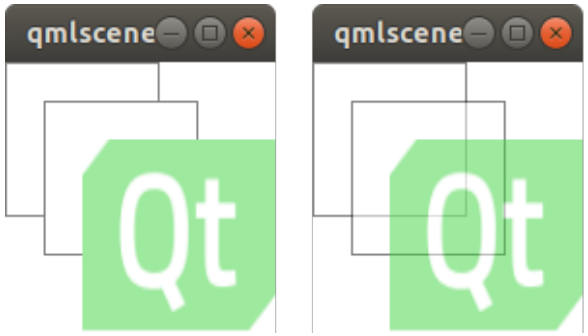
L'objet Image

```
Image {
    x: 40
    y: 20
    width: 61
    height: 73
    source: "http://codereview.qt-project.org/static/logo_qt.png"
}
```

L'objet Item

- Tous les éléments visuels de *Qt Quick* héritent de `Item`.
- Bien qu'un objet `Item` n'ait pas d'apparence visuelle, il définit tous les attributs communs aux éléments visuels, tels que la position x et y, la largeur et la hauteur, ...
- `Item` peut être utile pour regrouper plusieurs éléments sous un élément visuel racine unique. Par exemple :

```
Item {
    width: 140; height: 140; opacity: 0.5;
    layer.enabled: true; // ou false
    Rectangle { width: 80; height: 80; border.width: 1 }
    Rectangle { x: 20; y: 20; width: 80; height: 80; border.width: 1 }
    Image { x: 40; y: 40; width: 100; height: 100; source: "logo_qt.png" }
}
```



Définir ses propres éléments

Il est possible de définir des éléments dans des fichiers `.qml` et les utiliser dans un document QML :

Bouton.qml

```
import QtQuick 2.9

Rectangle {
    width: 100; height: 50;
    ...
    Text {
        text: "Valider"
        ...
    }
    ...
}
```

main.qml

```
import QtQuick 2.9
import QtQuick.Window 2.2

Window {
    visible: true;
    width: 640; height: 480;
    title: qsTr("Hello World")

    Bouton {
        ...
    }

}
```

Les éléments de texte dans QML

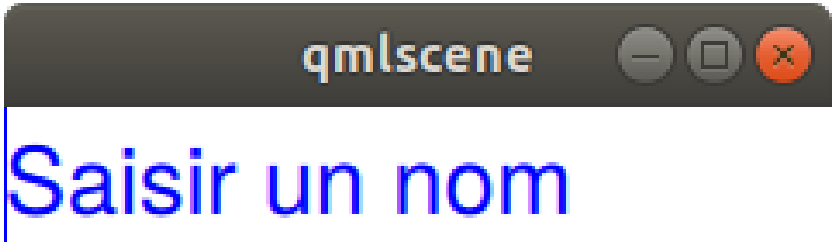
- `Text` : affiche un texte mis en forme
  `http://doc.qt.io/qt-5/qml-qtquick-text.html`
- `TextEdit` : affiche plusieurs lignes de texte formaté modifiable
  `http://doc.qt.io/qt-5/qml-qtquick-textedit.html`
- `TextInput` : affiche une ligne de texte modifiable
  `http://doc.qt.io/qt-5/qml-qtquick-textinput.html`

L'objet Text

```
Text {
    textFormat: Text.RichText
    text: "Un <b>super</b> <i>site</i> : <a href=\"http://tvaira.free.fr\">
         tvaira.free.fr</a>."
    font.family: "Helvetica"
    font.pointSize: 24
    color: "steelblue"
    style: Text.Outline;
    onLinkActivated: console.log("lien activé : " + link)
}
```

L'objet TextEdit

```
TextEdit {
    width: 240
    text: "Saisir un nom"
    font.family: "Helvetica"
    font.pointSize: 20
    color: "blue"
    focus: true
}
```



L'objet TextInput

```
TextInput {
    text: "Saisir une valeur"
    validator: IntValidator{bottom: 1; top: 100;}
    focus: true
    cursorVisible: false
    onAccepted: console.log("Accepted")
}
```

Le positionnement (notion d'ancre)

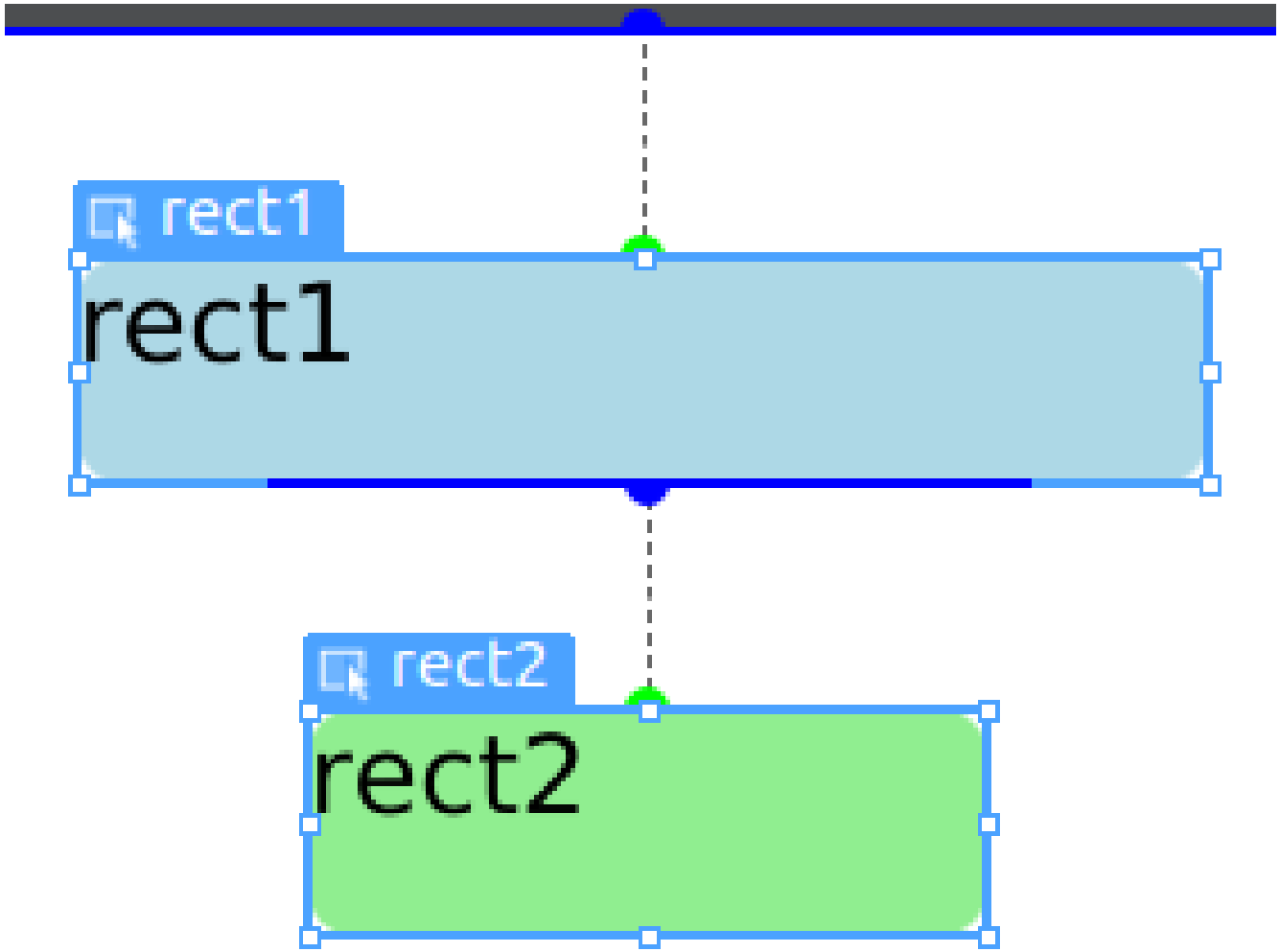
- Les éléments peuvent se positionner par leur attributs `x` et `y` mais ce n'est pas la technique conseillée.
- On positionne habituellement les éléments les uns par rapport aux autres.
- On peut le faire par un système d'**ancres** (`anchors`). Une ancre se rattache à un autre élément par son point haut (`top`), bas (`bottom`), droit (`right`) ou gauche (`left`). Comme les éléments sont souvent emboîtés les uns dans les autres, on fera souvent référence à son élément parent (`parent`).
- Un élément emboîté dans un autre élément peut aussi remplir tout son espace : `anchors.fill: parent`.
- On peut aussi appliquer des marges (`anchors.margins`) ou spécifiquement (`anchors.leftMargin`, etc ...).
- Les ancres permettent aussi de se centrer horizontalement (`anchors.horizontalCenter`) et/ou verticalement (`anchors.verticalCenter`).

Positionnement avec des ancres

```
Window {
    Rectangle { id: rect1
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: parent.top
        anchors.topMargin: 50
        Text { text: "rect1" }
    }
    Rectangle { id: rect2
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: rect1.bottom
        anchors.topMargin: 50
        Text { text: "rect2" }
    }
}
```

☞ https://doc.qt.io/qt-5/qtquick-positioning-anchors.html

Design



☞ https://doc.qt.io/qt-5/qtquick-positioning-anchors.html

Les éléments de positionnement

☞ `http://doc.qt.io/qt-5/qtquick-positioning-layouts.html`

- Les éléments de positionnement sont des conteneurs qui gèrent les positions des éléments dans l'interface utilisateur.
- Ils se comportent de la même manière que les layouts utilisés avec les *widgets* Qt.
- Le sous-module *Qt Quick Layouts* peut également être utilisé pour organiser des éléments QML (voir plus loin).

Un ensemble de « positionneurs » est fourni dans *Qt Quick*, les plus simples sont :

- `Column` ou `Row` : positionne les éléments enfants dans une colonne ou une ligne
- `Flow` : positionne les éléments enfants côte à côte, en les enveloppant si nécessaire
- `Grid` : positionne les éléments enfants dans une grille

✍ Voir aussi : `LayoutMirroring` et `Positioner`

Positionnement en colonne

```
Item {
    width: 310; height: 170
    Column {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        spacing: 5
        Rectangle { color: "lightblue"; radius: 10.0; width: 300; height: 50
                    Text { anchors.centerIn: parent; font.pointSize: 24; text: "
                        Books" } }
        Rectangle { color: "gold"; radius: 10.0; width: 300; height: 50
                    Text { anchors.centerIn: parent; font.pointSize: 24; text: "
                        Music" } }
        Rectangle { color: "lightgreen"; radius: 10.0; width: 300; height: 50
                    Text { anchors.centerIn: parent; font.pointSize: 24; text: "
                        Movies" } }
    }
}
```

ATTENTION : les éléments placés dans un `Row` et/ou un `Column` ne peuvent plus se positionner les uns par rapport aux autres par leurs ancres ! C'est logique.

Test : Positionnement en colonne

Positionnement avec Screen

Il est possible d'afficher des éléments en fonction de l'orientation et des dimensions de l'écran en utilisant le type `Screen` :

```
import QtQuick 2.7
import QtQuick.Window 2.2

Window {
    visible: true
    // Plein écran
    width: Screen.width // ou Screen.desktopAvailableWidth
    height: Screen.height // ou Screen.desktopAvailableHeight

    Rectangle {
        width: 200
        height: 100
        // Au milieu de l'écran
        x: (Screen.width - width) / 2
        y: (Screen.height - height) / 2
        color: "blue"
    }
}
```

L'orientation de l'écran

On peut distinguer deux type d'orientation : le mode paysage (`Qt.LandscapeOrientation`) et le mode portrait (`Qt.PortraitOrientation`). La propriété `Qt.primaryOrientation` permet de connaître l'orientation au lancement de l'application.

```
import QtQuick 2.7
import QtQuick.Window 2.2

Window {
    //...

    // Adapter la largeur en fonction de l'orientation
    Rectangle {
        width: Screen.orientation === Qt.PortraitOrientation ? parent.width : 200
        //...
    }
}
```

Intégration du Javascript

- Le moteur JavaScript fourni par QML permet d'exécuter les instructions standard telles que les tests conditionnels, les boucles et l'utilisation des variables comme les tableaux ...
- En plus, le JavaScript de QML inclut un certain nombre de fonctions qui simplifient la création d'interfaces utilisateur et l'interaction avec l'environnement QML.
- Le code JavaScript peut être utilisé :
  - directement dans l'initialisation des propriétés
  - dans les corps des gestionnaires de signaux
  - pour définir des fonctions personnalisées
  - dans des fichiers de ressources

☞ Liens : `http://doc.qt.io/qt-5/qtqml-javascript-topic.html` et `http://doc.qt.io/qt-5/qtqml-javascript-expressions.html`

Du JavaScript dans les propriétés

Le code JavaScript peut être utilisé directement dans l'initialisation des propriétés (ici `color`) :

```
Rectangle {
    width: 200; height: 200

    //color: mousearea.pressed ? "steelblue" : "lightsteelblue"
    color: if(mousearea.pressed)
             "steelblue"
           else
             "lightsteelblue"

    MouseArea {
        id: mousearea
        anchors.fill: parent
    }
}
```

Du JavaScript dans les gestionnaires de signaux

Le code JavaScript peut être utilisé dans les corps des gestionnaires de signaux (ici `onClicked`) :

```
Rectangle {
    width: 200; height: 200
    color: "white"

    MouseArea {
        anchors.fill: parent
        onClicked: { // gestionnaire d'évènement pour le clic
            if(parent.color == "#0000ff")
                parent.color = "white"
            else
                parent.color = "blue"
        }
    }
}
```

Du JavaScript pour définir ses fonctions

Le code JavaScript permet de définir des ses propres fonctions puis les utiliser dans un document QML :

```
import QtQuick 2.7

Rectangle {
    id: button
    width: 200; height: 200
    color: "blue"

    function foo(couleur) { // définition d'une fonction foo()
            button.color = couleur;
    }

    MouseArea {
        anchors.fill: parent
        onClicked: foo("red"); // appel de la fonction foo()
    }
}
```

Du JavaScript dans des fichiers de ressources I

Le code JavaScript peut être utilisé à partir de fichiers `.js`. Pour cela, il faudra soit l'importer soit l'inclure.

- Lorsqu'un fichier JavaScript est importé, il doit être importé avec un qualificateur. Les fonctions de ce fichier sont alors accessibles à partir du script d'importation via le qualificatif (c'est-à-dire `qualificateur.nomFonction(parametres)`)

```
import "script.js" as MyScript // MyScript est le qualificateur
...
onClicked: {
    MyScript.showCalculations(10)
    MyScript.factorial(10))
}
```

Parfois, il est souhaitable que les fonctions soient disponibles dans le contexte d'importation sans avoir à les qualifier.

Du JavaScript dans des fichiers de ressources II

- Dans ce cas, la fonction `Qt.include()` peut être utilisée dans un fichier JavaScript pour inclure un autre fichier JavaScript.

```
// script.js
Qt.include("factorial.js")

function showCalculations(value) {
    console.log("Call factorial() from script.js:", factorial(value));
}
```

```
// factorial.js
function factorial(a) {
    ...
}
```

Les sous-modules de Qt Quick

Qt Quick comprend plusieurs sous-modules qui contiennent des types supplémentaires, comme :

- *Controls* : fournit un ensemble de contrôles pour créer des interfaces complètes
- *Layouts* : fournit des dispositifs d'organisation pour les éléments de Qt
- *Window* : contient des types pour créer des fenêtres de niveau supérieur
- *Dialogs* : contient les types pour créer des boîtes de dialogue et interagir avec le système

Et aussi (non traités) :

- *XML List Model* : contient des types pour créer des modèles à partir de données XML
- *Local Storage* : un sous-module contenant une interface JavaScript pour une base de données SQLite
- *Particles* : fournit un système de particules pour *Qt Quick*
- *Tests* : contient des types pour écrire des tests unitaires

Qt Quick Controls

Qt Quick Controls fournit un ensemble de composants pour créer des interfaces très complètes :

- Application fenêtrée : `ApplicationWindow`, `MenuBar`, `StatusBar`, `ToolBar`, `MenuBar` et `Action`
- Navigation et vues : `ScrollView`, `SplitView`, `StackView`, `TabView`, `TableView` et `TreeView`
- Contrôles : `Button`, `CheckBox`, `ComboBox`, `Label`, `ProgressBar`, `Slider`, `SpinBox`, `BusyIndicator`, ...
- Menus : `Menu`, `MenuItem` et `MenuSeparator`

☞ `http://doc.qt.io/qt-5/qtquickcontrols-index.html`

Button

Qt Quick Controls fournit un ensemble de composants à intégrer dans les interfaces, comme le **Button** :

```
...
import QtQuick.Controls 2.1 // pour Button

Window { ...
    Rectangle {
        ...
        Button {
            text: qsTr("Valider")
            anchors.bottom: parent.bottom
            anchors.horizontalCenter: parent.horizontalCenter
            onClicked: {
                console.log(qsTr('Vous avez cliqué sur le bouton'))
                ... // affiche le texte "Ok" en vert
            }
        }
    }
}
```

Les layouts

Qt Quick Layouts est un ensemble de types QML utilisés pour organiser les éléments dans une interface utilisateur. Contrairement aux positionneurs, les layouts peuvent également redimensionner ses éléments. Comme les layouts sont des éléments, elles peuvent être imbriquées.

- `GridLayout` : permet d'organiser dynamiquement des éléments dans une grille
- `ColumnLayout` : identique à `GridLayout`, mais avec une seule colonne
- `RowLayout` : identique à `GridLayout`, mais avec une seule ligne
- `StackLayout` : sous forme de pile où un seul est visible à la fois
- `Layout` : fournit des propriétés pour les éléments insérés dans `GridLayout`, `RowLayout` ou `ColumnLayout`

☞ `http://doc.qt.io/qt-5/qtquicklayouts-index.html`

Positionnement RowLayout

```
import QtQuick 2.7
import QtQuick.Layouts 1.3

RowLayout {
    anchors.fill: parent
    spacing: 6
    Rectangle { width: 100; height: 100;
                radius: 20.0; color: "#024c1c" }
    Rectangle { width: 100; height: 100;
                radius: 20.0; color: "#42a51c" }
    Rectangle { width: 100; height: 100;
                radius: 20.0; color: "#c0c0c0" }
}
```

Test : Positionnement RowLayout

Les boîtes de dialogue

Le sous-module *Dialogs* fournit des boîtes de dialogue prédéfinies :

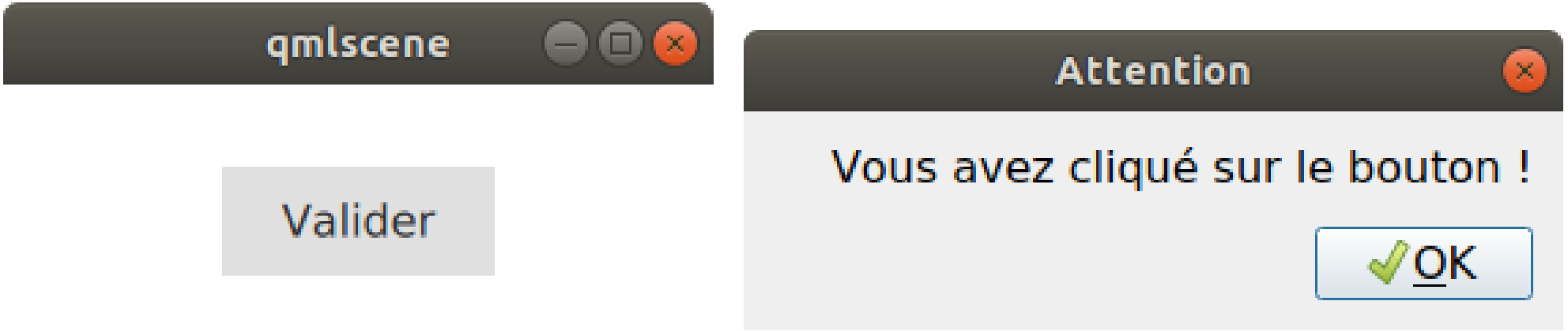
- `ColorDialog` : pour choisir une couleur
- `Dialog` : générique avec des boutons standards
- `FileDialog` : pour choisir des fichiers à partir d'un système de fichiers local
- `FontDialog` : pour choisir une police
- `MessageDialog` : pour afficher des messages contextuels

☞ `http://doc.qt.io/qt-5/qtquick-dialogs-qmlmodule.html`

MessageDialog

```
import QtQuick 2.7
import QtQuick.Controls 2.1
import QtQuick.Dialogs 1.2

Rectangle {
    width: 260; height: 100
    Button {
        text: qsTr("Valider")
        anchors.centerIn: parent
        onClicked: messageDialog.show(qsTr("Vous avez cliqué sur le bouton !"))
    }
    MessageDialog {
            id: messageDialog
            title: qsTr("Attention")
            function show(message) {
                messageDialog.text = message;
                messageDialog.open();
            }
    }
}
```

Test : MessageDialog



Les animations

Les animations sont créées en appliquant des types d'animation aux valeurs de propriété pour créer des transitions régulières.

- Pour créer une animation, on utilise un type d'animation approprié en fonction du type de propriété.
- Il existe plusieurs manières de définir une animation pour un objet.
- L'animation de propriété (`PropertyAnimation`) permet de modifier progressivement les valeurs des propriétés.
- Il existe des animations de propriété (`PropertyAnimation`) spécialisées comme `NumberAnimation` qui définit une animation à appliquer lorsqu'une valeur numérique change.
- On peut contrôler l'exécution des animations avec les méthodes `start()`, `stop()`, `resume()`, `pause()`, `restart()` et `complete()`.

☞ doc.qt.io/qt-5/qtquick-statesanimations-animations.html

PropertyAnimation

On peut par exemple utiliser `PropertyAnimation` sur les coordonnées x et y d'un objet pour le déplacer :

```
Rectangle {
    id: rect
    width: 100; height: 100
    color: "blue"
    PropertyAnimation on x { to: 100 }
    PropertyAnimation on y { to: 100 }
}
```

NumberAnimation

On peut par exemple utiliser NumberAnimation pour faire une rotation d'un texte :

```
Rectangle {
    Rectangle {
        property int d: 100
        width: d
        height: d
        anchors.centerIn: parent
        color: "red"
        NumberAnimation on rotation { from: 0; to: 360; duration: 2000; loops:
            Animation.Infinite; }
    }

    Text {
        anchors.centerIn: parent
        text: "Une animation"
    }
}
```

Contrôler l'exécution des animations

```
Rectangle {
    id: rect
    width: 75; height: 75
    color: "blue"
    opacity: 1.0
    MouseArea {
        anchors.fill: parent
        onClicked: animateColor.start()
    }
    PropertyAnimation {id: animateColor; target: rect; properties: "color"; to: "
         black"; duration: 1000}
}
```

Contrôle de synchronisation

Les animations de propriétés fournissent des contrôles de synchronisation et permettent différentes interpolations via des courbes prédéfinies. Ces courbes simplifient la création d'effets d'animation tels que les effets de rebond, l'accélération, la décélération et les animations cycliques. Par exemple, le rebound d'une balle :

```
Rectangle {
    width: 75; height: 150;
    Rectangle {
        width: 75; height: 75; radius: width; color: "salmon"; id: ball
        SequentialAnimation on y {
            loops: Animation.Infinite
            NumberAnimation {
                from: 0; to: 75; easing.type: Easing.OutExpo; duration: 300
            }
            NumberAnimation {
                from: 75; to: 0; easing.type: Easing.OutBounce; duration: 1000
            }
            PauseAnimation { duration: 500 }
        }
    }
}
```

Les états

Les états sont un ensemble de configurations de propriétés définies dans un type `State`. Il permettent, par exemple :

- d'afficher des composants d'interface utilisateur et en cacher d'autres
- de présenter différentes actions disponibles à l'utilisateur
- de démarrer, arrêter ou interrompre les animations
- d'exécuter un script requis dans le nouvel état
- de modifier une valeur de propriété pour un élément particulier
- d'afficher une vue ou un écran différent

☞ doc.qt.io/qt-5/qtquick-statesanimations-states.html

Notion d'état

- Tous les objets basés sur `Item` ont une propriété `state`.
- On peut spécifier des états supplémentaires en ajoutant de nouveaux objets `State` à la propriété `states` de l'élément.
- Chaque état d'un composant a un nom unique, une chaîne vide étant la valeur par défaut.
- L'ensemble des propriétés fixées par un nouvel état est réalisé avec le type `PropertyChanges`.
- Pour modifier l'état actuel d'un élément, il faut définir la propriété `state` sur le nom du nouvel état.
- Le type `State` a une propriété `when` qui permet de modifier l'état lorsque l'expression est vraie.

✍ Il est possible de réaliser des transitions sur les changements d'états : doc.qt.io/qt-5/qtquick-statesanimations-animations.html

Exemple

```
Rectangle {
    id: feu; width: 200; height: 200
    state: "NORMAL"
    states: [
        State {
            name: "NORMAL"
            PropertyChanges { target: feu; color: "green" }
        },
        State {
            name: "CRITIQUE"
            PropertyChanges { target: feu; color: "red" }
        }
    ]
    MouseArea {
        anchors.fill: parent
        onClicked: {
            if (feu.state == "NORMAL")
                feu.state = "CRITIQUE"
            else
                feu.state = "NORMAL"
        }
    }
}
```

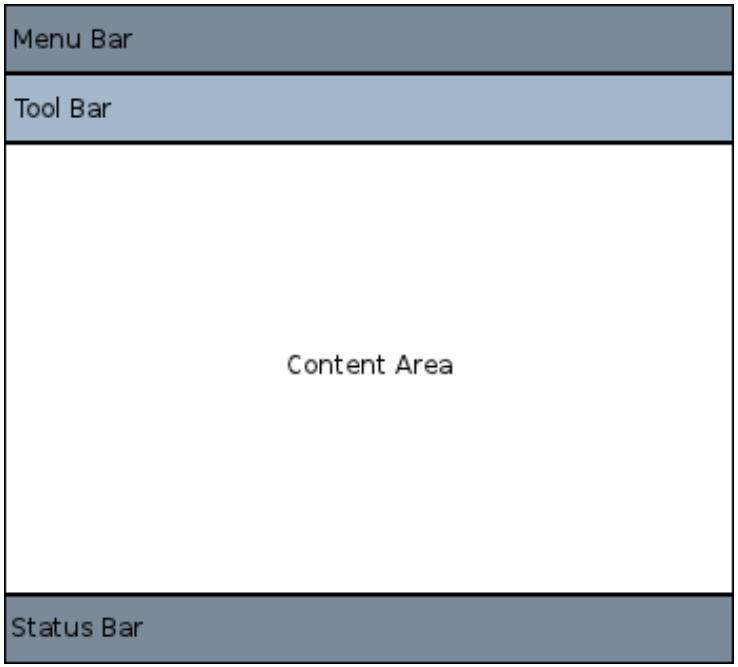
État et transition

Il est possible de réaliser des transitions sur les changements d'états : doc.qt.io/qt-5/qtquick-statesanimations-animations.html

```
Rectangle {
    id: feu; width: 200; height: 200
    ...
    transitions: [
        Transition {
            from: "NORMAL"
            to: "CRITIQUE"
            ColorAnimation { target: feu; duration: 1000 }
        },
        Transition {
            from: "CRITIQUE"
            to: "NORMAL"
            ColorAnimation { target: feu; duration: 1000 }
        }
    ]
    ...
}
```

ApplicationWindow (Qt Quick Controls 1)

- Une fenêtre d'application version 1 contient le plus souvent un menu, une barre d'outils et une barre de statut.
- Pour créer ce type de fenêtre, il faut instancier un objet `ApplicationWindow` (qui hérite de `Window`) du module *Qt Quick Controls 1*.

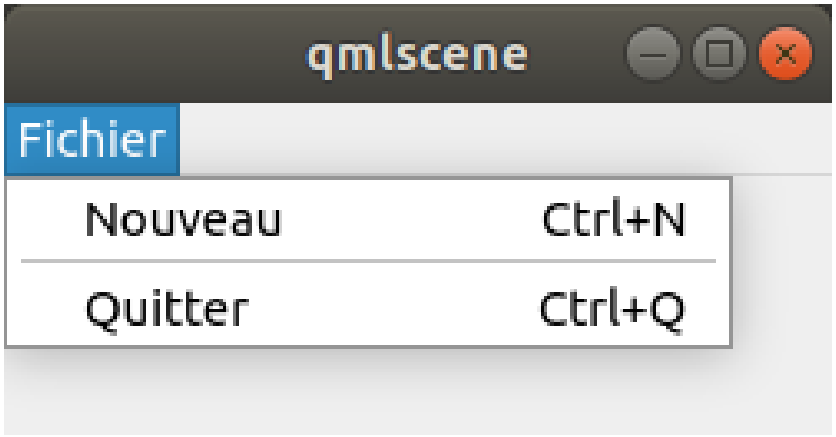


☞ `http://doc.qt.io/qt-5/qml-qtquick-controls-applicationwindow.html`

Exemple ApplicationWindow (Qt Quick Controls 1) : Menu

```
import QtQuick.Controls 1.4
ApplicationWindow {
    visible: true

    menuBar: MenuBar {
        Menu {
            title: "Fichier"
            MenuItem {
                text: "Nouveau"
                shortcut: "Ctrl+N"
                //iconSource:
            }
            MenuSeparator {}
            MenuItem {
                text: "Quitter"
                shortcut: "Ctrl+Q"
                onTriggered: Qt.quit()
            }
        }
    }
}
```

Exemple ApplicationWindow (Qt Quick Controls 1)

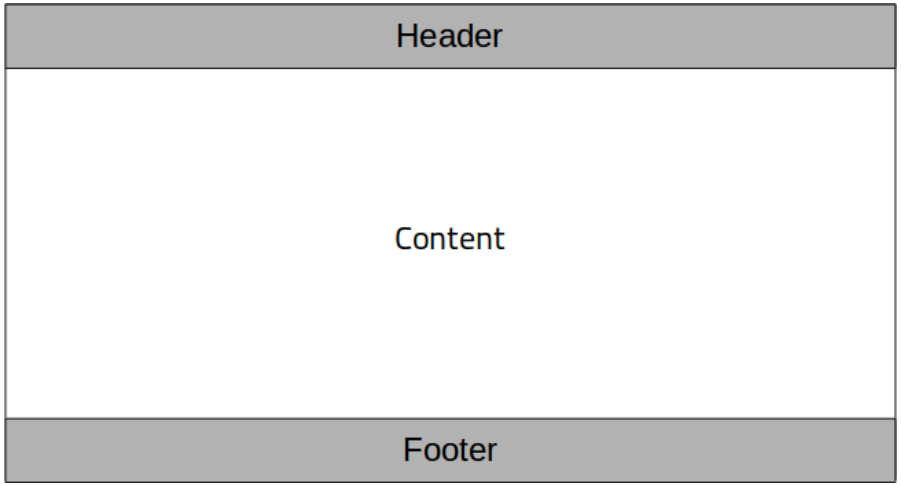


Et aussi :

```
ApplicationWindow {
    ...
    toolBar: ToolBar {
        RowLayout {
            anchors.fill: parent
            ToolButton {}
        }
    }
    TabView {
        anchors.fill: parent //...
    }
}
```

ApplicationWindow (Qt Quick Controls 2)

- Une fenêtre d'application version 2 est une fenêtre qui permet d'ajouter un élément d'en-tête et de pied de page.
- Pour créer ce type de fenêtre, il faut instancier un objet `ApplicationWindow` (qui hérite de `Window`) du module *Qt Quick Controls 2*.



☞ `https://doc.qt.io/qt-5.9/qml-qtquick-controls2-applicationwindow.html`

Exemple ApplicationWindow (Qt Quick Controls 2)

```
import QtQuick.Controls 2.1

ApplicationWindow {
    visible: true

    header: ToolBar {
        // ...
    }

    footer: TabBar {
        // ...
    }

    StackView {
        anchors.fill: parent
    }
}
```

Chargement dynamique (Loader)

L'élément `Loader` est utilisé pour charger dynamiquement des composants QML. L'élément à charger est contrôlé via la propriété `source` ou la propriété `sourceComponent`.

`Loader` peut charger :

- propriété `source` : charge l'élément à partir d'un fichier QML,
- propriété `sourceComponent` : instancie un composant.

✍ l'élément chargé sera redimensionné à la taille du chargeur ou à sa taille si elle est spécifiée explicitement.

☞ `http://doc.qt.io/qt-5/qml-qtquick-loader.html`

Loader : chargement d'un composant

```
Item { id: root
    width: parent.width; height: parent.height
    Loader { id: loader
        //anchors.fill: parent; // de la taille de l'élément parent
        anchors.centerIn: parent; // de la taille de l'élément chargé (ici 50x50)
              et centré dans l'élément parent
        sourceComponent: rect1
    }
    Component { id: rect1
        Rectangle { width: 50; height: 50; color: "red";
            MouseArea { anchors.fill: parent
                onClicked: loader.sourceComponent = rect2;
            }
        }
    }
    Component { id: rect2
        Rectangle { width: 50; height: 50; color: "green";
            MouseArea { anchors.fill: parent
                onClicked: loader.sourceComponent = rect1;
            }
        }
    }
}
```

Loader : en utilisant un état

```
Item { ...
    Loader { id: loader
        anchors.centerIn: parent;
    }
    Component { id: rect1
        Rectangle { width: 50; height: 50; color: "red";
            MouseArea { anchors.fill: parent
                onClicked: root.state = "on";
    } } }
    Component { id: rect2
        Rectangle { width: 50; height: 50; color: "green";
            MouseArea { anchors.fill: parent
                onClicked: root.state = "off";
    } } }
    state: "off"
    states: [
        State { name: "on"
            PropertyChanges { target: loader; sourceComponent: rect2; } },
        State { name: "off"
            PropertyChanges { target: loader; sourceComponent: rect1; } }
    ]
}
```

Loader : chargement d'un fichier QML

```
Item {
    id: root
    width: parent.width; height: parent.height
    Loader {
        id: loader
        anchors.centerIn: parent;
    }
    state: "on"
    states: [
        State {
            name: "on"
            PropertyChanges { target: loader; source: "on.qml"; }
        },
        State {
            name: "off"
            PropertyChanges { target: loader; source: "off.qml"; }
        }
    ]
}
```

Les possibilités du Loader

En utilisant `Loader` :

- On peut accéder aux propriétés de l'élément chargé :
  `Text { text: "color = " + loader.item.color; }`
- On peut recevoir des signaux de l'élément chargé en utilisant `Connections`
- On peut lier les propriétés de l'élément racine avec celui de l'élement chargé avec `Binding`
- On peut donner le *focus* à l'élément chargé en fixant la propriété `focus` à `true` du `Loader`

Chargement dynamique en JavaScript

```
var component = Qt.createComponent("Releves.qml");
if (component.status == Component.Ready)
{
        var page = component.createObject(vueReleve);
        if (page == null)
        {
            console.log("Erreur création Releves.qml !");
        }
        else
        {
        }
}
else if (component.status == Component.Error)
{
        console.log("Erreur chargement Releves.qml :", component.errorString());
}
...
Item {
    id: vueReleve
    anchors.fill: parent
}
```

Sommaire

4 Interaction QML/C++

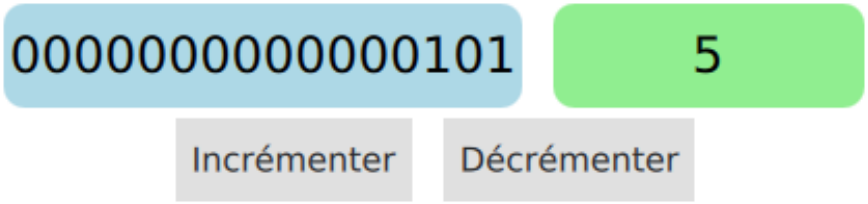
Interaction QML/C++

- QML est conçu pour être extensible en intégrant du code C++.
  - Qt permet d'appeler les fonctionnalités C++ directement à partir de QML.
  - Qt QML permet de charger et de manipuler des objets QML à partir de C++.
- Cela permet le développement d'applications hybrides contenant du code QML, JavaScript et C++.

☞ `http://doc.qt.io/qt-5/qtqml-cppintegration-overview.html` et `http://doc.qt.io/qt-5/qtqml-cppintegration-topic.html`

Exemple : Un compteur binaire

- On va réaliser un compteur binaire dont on affichera sa valeur en binaire et en décimal et que l'on pourra incrémenter ou décrémenter.



- On créera l'interface graphique en QML et on implémentera une classe C++ `CompteurBinaire` pour le comptage.
- On intégrera ensuite la classe C++ avec QML.

Exemple : l'interface graphique en QML I

On va utiliser un positionnement en lignes (`Row`) et en colonnes (`Column`) :

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.1
Window {
    visible: true; title: qsTr("Compteur")
    Column {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        spacing: 5
        Row {
            anchors.horizontalCenter: parent.horizontalCenter
            spacing: 15
            Rectangle {
                color: "lightblue"; radius: 10.0; width: 250; height: 50
                Text {
                    id: valeurBinaire
                    anchors.centerIn: parent; font.pointSize: 18;
                    text: "0"
                }
            }
```

Exemple : l'interface graphique en QML II

```
        //...
        Rectangle {
            color: "lightgreen";
            radius: 10.0;
            width: 150; height: 50
            Text {
                id: valeur
                anchors.centerIn: parent;
                font.pointSize: 18;
                text: "0"
            }
        }
    } // <- Row
```

Exemple : l'interface graphique en QML III

```
        // ...
        Row {
            anchors.horizontalCenter: parent.horizontalCenter
            spacing: 15
            Button {
                id: boutonIncrement
                text: qsTr("Incrémenter")
                //onClicked:
            }
            Button {
                id: boutonDecrement
                text: qsTr("Décrémenter")
                //onClicked:
            }
        } // <- Row
    } // <- Column
} // <- Window
```

Exemple : la classe CompteurBinaire

```
#include <QObject>

class CompteurBinaire : public QObject
{
    Q_OBJECT
private:
    unsigned short _compteur;

public:
    explicit CompteurBinaire(QObject *parent = nullptr);
    Q_INVOKABLE unsigned short getValeur() const; // pour être appelé à partir
    Q_INVOKABLE QString getValeurBinaire() const; // de QML

signals:
    void valueChanged() const;

public slots:
    void incrementer();
    void decrementer();
};
```

- Il faut hériter de la classe `QObject` pour intégrer les mécanismes propres à Qt.
- Les méthodes publiques que l'on veut rendre accessible à QML doivent être préfixées par `Q_INVOKABLE`.
- Les méthodes `incrementer()` et `decrementer()` émettront le signal `valueChanged()` à chaque fois que l'attribut `compteur` change.
- Le signal `valueChanged()` pourra être traité depuis QML à l'aide du gestionnaire `onValueChanged`.

☞ `http://doc.qt.io/qt-5/qtqml-cppintegration-topic.html`

Définition des types QML à partir de C++

- Les types QML peuvent être définis en C++, puis enregistrés comme un type QML avec `qmlRegisterType`.
- Cela permet à une classe C++ d'être instanciée en tant que type d'objet QML.

☞

`http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html`

Exemple : intégration C++

Il faut donc enregistrer enregistrer la classe `CompteurBinaire` en tant que type QML avec `qmlRegisterType`. Il sera accessible dans QML en faisant un `import`.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include "compteurbinaire.h"

int main(int argc, char *argv[])
{
  QGuiApplication app(argc, argv);

  qmlRegisterType<CompteurBinaire>("CompteurBinaire", 1, 0, "CompteurBinaire");

  QQmlApplicationEngine engine;
  engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
  if (engine.rootObjects().isEmpty())
      return -1;

  return app.exec();
}
```

Exemple : intégration dans QML

- Il faut tout d'abord importer `CompteurBinaire` puis créer une instance. Le signal `valueChanged()` de la classe `CompteurBinaire` est connecté au gestionnaire `onValueChanged`. Il est aussi possible d'appeler des méthodes publiques (si elles sont déclarées avec `Q_INVOKABLE`) :

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtQuick.Controls 2.1
import CompteurBinaire 1.0
Window { //...
    CompteurBinaire { // un objet QML CompteurBinaire
        id: compteurBinaire
        onValueChanged: {
            console.log("La valeur du compteur a changé !")
            valeurBinaire.text = compteurBinaire.getValeurBinaire();
            valeur.text = compteurBinaire.getValeur();
        }
    }
    // ...
}
```

Accès aux membres d'une classe C++

QML peut accéder aux membres suivants d'une instance d'une classe C++ qui hérite de `QObject` :

- Les propriétés déclarées avec `Q_PROPERTY`
- Les méthodes publiques marquées `Q_INVOKABLE`
- Les slots publiques
- Les signaux
- Les énumérations déclarées avec `Q_ENUMS`

☞ `http://doc.qt.io/qt-5/qtqml-cppintegration-exposecppattributes.html`

- On peut appeler un *slot* de la classe `CompteurBinaire` :

```
//...
Row {
    anchors.horizontalCenter: parent.horizontalCenter
    spacing: 15
    Button {
        id: boutonIncrement
        text: qsTr("Incrémenter")
        onClicked: compteurBinaire.incrementer(); // appel du slot
    }
    Button {
        id: boutonDecrement
        text: qsTr("Décrémenter")
        onClicked: compteurBinaire.decrementer(); // appel du slot
    }
}
//...
```

- Un objet QML peut se connecter à un signal d'une classe C++ avec un objet `Connections` :

```
Text {
    id: valeurBinaire
    anchors.centerIn: parent;
    font.pointSize: 18;
    text: compteurBinaire.getValeurBinaire()
    Connections {
        target: compteurBinaire
        onValueChanged: {
            console.log("signal recu")
            valeurBinaire.text = compteurBinaire.getValeurBinaire();
        }
    }
}
```

Exemple : déclaration d'une propriété en C++

- La macro `Q_PROPERTY` déclare une propriété accessible à partir de QML. On peut l'utiliser ici pour l'attribut `_compteur` en lecture (`compteur()`) et en écriture (`setCompteur()`) :

```
class CompteurBinaire : public QObject
{
    Q_OBJECT
    Q_PROPERTY(unsigned short compteur READ compteur WRITE setCompteur NOTIFY
        valueChanged)

private:
    unsigned short _compteur;

public:
    explicit CompteurBinaire(QObject *parent = nullptr);

    unsigned short compteur() const;
    void setCompteur(unsigned short valeur);

    // ...
};
```

Exemple : utilisation d'une propriété dans QML

- La propriété `compteur` est maintenant utilisable directement dans QML :

```
CompteurBinaire {
    id: compteurBinaire
    onValueChanged: {
        valeur.text = compteurBinaire.compteur; // appel CompteurBinaire::
            compteur()
    }
}
```

Accéder à un objet C++

- Il faut instancier un objet de la classe `CompteurBinaire` puis l'associer au document QML avec `setContextProperty()` :

```
int main(int argc, char *argv[])
{
  QGuiApplication app(argc, argv);

  qmlRegisterType<CompteurBinaire>("CompteurBinaire", 1, 0, "CompteurBinaire");

  CompteurBinaire compteurBinaire;
  QQmlApplicationEngine engine;

  engine.rootContext()->setContextProperty("compteurBinaire", &compteurBinaire);
  engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
  if (engine.rootObjects().isEmpty())
      return -1;

  return app.exec();
}
```

Emettre des signaux depuis QML

- On retire l'instanciation de l'objet QML `CompteurBinaire`.
- On donne un nom d'objet aux deux boutons.
- On ajoute la possibilité d'émettre un signal aux deux boutons (ici le *signal* `increment()`). L'émission du signal sera réalisé dans le gestionnaire `onClicked`.

```
//...
Button {
    id: boutonIncrement
    objectName: "boutonIncrement"
    text: qsTr("Incrémenter")
    signal increment()
    onClicked: boutonIncrement.increment();
}
//...
```

Connecter les signaux QML aux slots C++

- Il faut maintenant connecter les signaux QML aux slots C++ :

Dans main.cpp

```
QObject *boutonIncrement = getObject(engine, "boutonIncrement");
QObject *boutonDecrement = getObject(engine, "boutonDecrement");

if(boutonIncrement != NULL)
    QObject::connect(boutonIncrement, SIGNAL(increment()), &compteurBinaire, SLOT
        (incrementer()));
if(boutonDecrement != NULL)
    QObject::connect(boutonDecrement, SIGNAL(decrement()), &compteurBinaire, SLOT
        (decrementer()));
```

Récupérer les objets QML

- Il faut pouvoir récupérer les objets dans le document QML, pour cela on utilise la fonction `getObject()` :

```
QObject* getObject(const QQmlApplicationEngine &engine, QString objectName)
{
    QObject *object = NULL;
    QList<QObject*> lObj = engine.rootObjects();
    for (int i = 0; i < lObj.size(); ++i)
    {
        QObject *obj = lObj.at(i);
        if(obj != NULL)
        {
            object = obj->findChild<QObject*>(objectName);
            if(object != NULL)
                break;
        }
    }
    return object;
}
```

Passer des objets QML au C++

- Il est possible de passer en argument des objets QML.
- Exemple pour un signal : `signal qmlSignal(var anObject)`.
- Ils seront déclarés comme `QVariant` en C++. Il faudra ensuite les transtyper en `QQuickItem`.
- On pourra modifier ses propriétés en utilisant `setProperty()`.

```
void Classe::cppSlot(QVariant &v)
{
    QQuickItem *item = qobject_cast<QQuickItem*>(v.value<QObject*>());

    qDebug() << Q_FUNC_INFO << "Item dimensions :" << item->width() << item->
        height();

    item->setProperty("text", "ok");
}
```

Association d'un objet C++ au document principal QML

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>

#include "maclasse.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;

    engine.rootContext()->setContextProperty("monObjet", new MaClasse());

    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec();
}
```

Accède à la déclaration de la **classe C++**

Ajoute un **objet C++** au contexte **QML**

Instancie un **objet C++**

Charge le **document principal QML** dans l'application

Appel d'une méthode d'un objet C++ à partir de QML

maclasse.h

```
#ifndef MACLASSE_H
#define MACLASSE_H

#include <QObject>
#include <QString>

class MaClasse : public QObject
{
  Q_OBJECT

public:
  explicit MaClasse(QObject *parent=nullptr);
  virtual ~MaClasse();

  Q_INVOKABLE bool lire(QString type, int nb=0);
};

#endif // MACLASSE_H
```

La **méthode** doit être déclarée avec `Q_INVOKABLE` dans une classe qui hérite de `QObject` pour pouvoir l'appeler à partir de QML

main.qml

```
Button {
   id: btLire
   text: "Lire"
   onClicked: {
      if(monObjet.lire("a"))
      {
      }
   }
}
```

On peut appeler la **méthode** de l'objet monObjet de type MaClasse à partir de QML/JS.

Il est aussi possible d'appeler un ***slot***.

Déclarer une propriété C++ pour l'utiliser en QML

maclasse.h

```
#ifndef MACLASSE_H
#define MACLASSE_H

#include <QObject>
#include <QString>

class MaClasse : public QObject
{
  Q_OBJECT
  Q_PROPERTY(bool erreurConnexion MEMBER erreurConnexion NOTIFY erreurChanged)
  Q_PROPERTY(QString moyenne READ getMoyenne NOTIFY moyenneUpdated)

public:
  QString getMoyenne(); // accesseur

private:
    bool erreurConnexion;
    QString moyenne;

signals:
    void erreurChanged();
    void moyenneUpdated();
};

#endif // MACLASSE_H
```

Une **propriété** est déclarée Q_PROPERTY() dans une **classe** qui hérite de QObject.

On peut exporter un **attribut** sous forme de **propriété** Qt avec MEMBER. L'attribut sera accessible en **lecture/écriture** sans avoir besoin d'accesseurs READ et WRITE.

Il faut toujours spécifier un ***signal*** avec NOTIFY pour autoriser la liaison de la **propriété** avec QML.

On peut exporter un **attribut** sous forme de **propriété** Qt accessible seulement en lecture avec l'accesseur READ sans utiliser MEMBER.

Utiliser les signaux C++ et les propriétés dans QML

maclasse.cpp

```
void MaClasse::connecter()
{
    erreurConnexion = ...
    emit erreurChanged();
}

void MaClasse::calculerMoyenne()
{
    moyenne = ...;
    emit moyenneUpdated();
}

QString MaClasse::getMoyenne()
{
    return moyenne;
}
```

Émet un ***signal*** Qt

L'élément **Connections** crée une connexion aux ***signaux*** de Qt.

main.qml

```
Connections {
  target: monObjet
  onMoyenneUpdated: {
   console.log("Moyenne : " + monObjet.moyenne);
  }
  onErreurChanged: {
   console.log("Erreur : " + monObjet.erreurConnexion);
  }
}
```

L'**objet** Qt qui émet le ***signal***

Il est aussi possible ce créer un signal dans QML et de le connecter à un slot C++ (voir Cours QML).

Les ***signaux*** Qt sont préfixés par **on** : **onSignal**

Accès à la **propriété** C++ d'un **objet**.

Sommaire

Modèles et vues

- Qt Quick fournit les notions de modèles, de vues et de délégués pour afficher des données.
  - `Model` : il contient les données et leur structure. Il existe plusieurs types QML pour créer des modèles.
  - `View` : un conteneur qui affiche les données. La vue peut afficher les données dans une liste ou une grille.
  - `Delegate` : il gère comment les données doivent apparaître dans la vue.

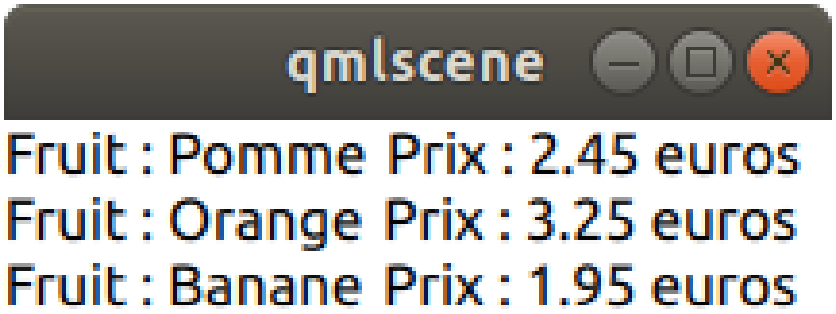
☞

`http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html`

Les vues

- Les vues sont des conteneurs pour des collections d'éléments.
- Un ensemble de vues standard est fourni dans Qt Quick :
  - `ListView` : organise les éléments dans une liste horizontale ou verticale
  - `GridView` : organise les éléments dans une grille
  - `PathView` : organise les éléments sur un chemin

☞ `http://doc.qt.io/qt-5/qtquick-views-example.html`



Les délégués

- Les vues ont besoin d'un délégué pour représenter visuellement un élément dans une liste.
- Les éléments d'un modèle sont accessibles via la propriété `index` ainsi que les propriétés de l'élément.
- On utilisera le type `Component` pour créer un délégué.

Les modèles

- Les données sont fournies au délégué via des rôles de données nommés auxquels le délégué peut se connecter.
- QML fournit plusieurs types de modèles de données :
  - `ListModel` est une hiérarchie simple des types spécifiés dans QML. Les rôles disponibles sont spécifiés par les propriétés `ListElement`.
  - `XmlListModel` permet la construction d'un modèle à partir d'une source de données XML. Les rôles sont spécifiés via le type `XmlRole`. Le type doit être importé : `import QtQuick.XmlListModel 2.0`
  - `ObjectModel` contient les éléments visuels à utiliser dans une vue. Lorsqu'un objet `ObjectModel` est utilisé dans une vue, la vue ne nécessite pas de délégué car `ObjectModel` contient déjà le délégué visuel.
- De plus, les modèles peuvent être créés en C++, puis mis à la disposition de QML avec `QQmlEngine`.

Exemple : une liste de fruits

- On définit un `Item` (ou un `Rectangle`) qui comprend un modèle `ListModel`, un délégué `Component` et une vue `ListView` :

```
import QtQuick 2.0
Item {
        ListModel {
            id: monModele
            // ...
        }
        Component {
            id: monDelegue
            // ...
        }
        ListView {
            id: maVue
            model: monModele
            delegate: monDelegue
            anchors.fill: parent
        }
}
```

Exemple : une liste de fruits (le modèle)

```
import QtQuick 2.0
Item {
        ListModel {
            id: monModele
            ListElement {
                nom: "Pomme"
                cout: 2.45
            }
            ListElement {
                nom: "Orange"
                cout: 3.25
            }
            ListElement {
                nom: "Banane"
                cout: 1.95
            }
        }
        // ...
}
```

Exemple : une liste de fruits (le délégué)

```
import QtQuick 2.0
Item {
        // ...
        Component {
            id: monDelegue
            Row {
                id: fruit
                spacing: 5
                Text { text: "Fruit : " + nom }
                Text { text: "Prix : " + cout + " euros" }
            }
        }
        // ...
}
```

Exemple : une liste de fruits (un modèle en C++)

```
QStringList monModele;

monModele.append("Pomme");
monModele.append("Orange");
monModele.append("Banane");

engine.rootContext()->setContextProperty("monModele", QVariant::fromValue(
    monModele));
```

Exemple : une liste de fruits (QML)

```
Item {
        width: 250; height: 250

        Component {
            id: monDelegue
            Rectangle {
                width: 250; height: 25
                Text { text: "Fruit : " + modelData }
            }
        }

        ListView {
            id: maVue
            model: monModele
            delegate: monDelegue
            anchors.fill: parent
        }
}
```

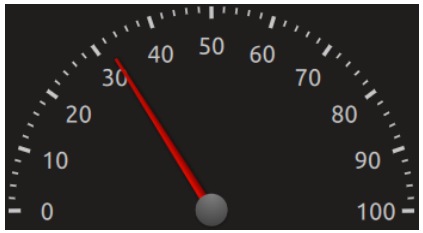
Sommaire

QQuickWidget

Il est possible d'utiliser des interfaces utilisateurs en QML dans une application *Qt Widget*.

- La classe `QQuickWidget` fournit un *widget* pour afficher une interface utilisateur *Qt Quick*.
- On utilise la méthode `setSource()` pour charger un fichier `.qml` dans le *widget*.
- On ajoute le module `quickwidgets` dans le fichier de projet :
  `QT += widgets quickwidgets`
- Lien : `http://doc.qt.io/qt-5/qquickwidget.html`

☞ Exemple : On va tout d'abord créer un composant QML à partir d'un type fourni (une jauge circulaire : `https://doc.qt.io/qt-5/qml-qtquick-extras-circulargauge.html` puis l'intégrer dans une application `QMainWindow`.



circular-gauge.qml

```
import QtQuick 2.3
import QtQuick.Extras 1.4 // CircularGauge
import QtQuick.Controls.Styles 1.4 // CircularGaugeStyle


Rectangle
{
    color: "#201d1d"
    width: 300
    height: 300
    CircularGauge {
        objectName: "circular-gauge"
        minimumValue: 0
        maximumValue: 100
        value: 0
        anchors.centerIn: parent
        style: CircularGaugeStyle {
                minimumValueAngle: -90
                maximumValueAngle: 90
        }
    }
}
```

mainwindow.h

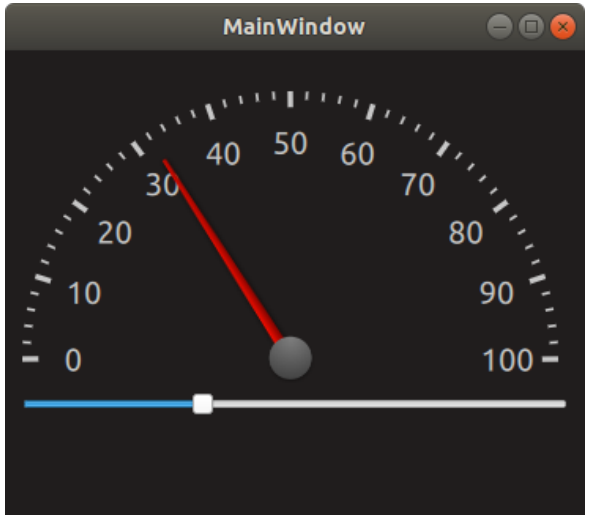
```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

public slots:
    void horizontalSliderValueChanged(int value);
};

#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QQuickWidget>
#include <QQuickItem>
#include <QQuickView>
#include <QVariant>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::
     MainWindow) {
    ui->setupUi(this);
    ui->quickWidget1->setSource(QUrl(QStringLiteral("qrc:/circular-gauge.qml")))
         ;
    ui->quickWidget1->show();
    QObject::connect(ui->slider1, SIGNAL(valueChanged(int)), this, SLOT(
         horizontalSliderValueChanged(int)));
}
void MainWindow::horizontalSliderValueChanged(int value) {
    QObject* qmlObject = ui->quickWidget1->rootObject();
    QObject* child = qmlObject->findChild<QObject*>("circular-gauge");
    if (child)
        child->setProperty("value", value);
}
MainWindow::~MainWindow() { delete ui; }
```

main.cpp

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setStyleSheet("QMainWindow {background: #201d1d;}");
    w.show();
    return a.exec();
}
```



Test QQuickWidget

Qt : Tutoriels et exemples

Qt fournit de nombreux tutoriels et exemples détaillés :

- Liste des tutoriels et exemples :
  `https://doc.qt.io/qt-5/qtquick-codesamples.html`
- La création d'un éditeur de texte :
  `https://doc.qt.io/qt-5/gettingstartedqml.html`
- Le jeu *Same Game* :
  `https://doc.qt.io/qt-5/qml-advtutorial.html`
- ...

➠ Et aussi : `https://qmlbook.github.io/` ...

Exemples https://tvaira.free.fr/

- Liste des exemples : http://tvaira.free.fr/dev/qt-android/qt-android.html
  - Application avec Qt Quick Controls
  - Base de données SQLite
  - Base de données MySQL
  - Dessiner des graphiques
  - Géolocalisation et cartes
  - Bluetooth BLE
  - Bluetooth
  - MQTT