
# Langage QML

Introduction à la syntaxe

## Le langage QML : Introduction à la syntaxe QML

- QML : Qt Modeling Language
  - La syntaxe du langage est inspirée de deux autres langages déclaratifs :
  - XAML (associé à WPF) et surtout JSON (lui-même inspiré de JavaScript).
  - Il s'agit d'un langage déclaratif permettant de décrire les éléments constitutifs d'une application QtQuick.
  - Il est possible de commenter une partie d'un fichier QML en utilisant les mêmes syntaxes de commentaires que le langage C++.

```
/* Commentaire multi-lignes */
// Commentaire sur une unique ligne
```

# Le langage QML : Introduction à la syntaxe QML

- On introduit un objet QML par le nom de sa classe suivit d'une paire d'accolades.

```
Window {  // Commencent par une majuscule
}
```

- Un objet est constitué de propriétés (entre les accolades). Pour introduire une propriété, on commence par son nom, puis on utilise un caractère « : », suivit de sa valeur.

```
Window {
     width: 320    // Commencent par une minuscule
     height: 240
}
```

- Note : les habitués de JavaScript/JSON, noteront qu'un « ; » n'est pas requis à la fin d'une propriété, sauf si vous cherchez à en mettre deux sur la même ligne.

```
Window { width: 320 ; height: 240 }
```


# Le langage QML : Introduction à la syntaxe QML

- Pour qu'un objet QML puisse être instancié, il doit :
  - Soit être déclaré en QML dans le projet.
  - Soit être importé (il est possible d'y spécifier la version minimale requise).

```
import QtQuick
import QtQuick.Controls
```

- Tout objet QML peut être nommé et l'on pourra se référer à ce nom plus tard dans le fichier QML.

```
Window {
     id: window
     width: 320
     height: 240
     /* Suite */
}
```

# Le langage QML : Introduction à la syntaxe QML

- Il est possible d'utiliser le moteur d'internationalisation de QtQuick dans vos fichiers QML.
- Pour introduire une chaîne internationalisée, il faut utiliser la fonction « ***qsTr*** ».

```
Window {
    id: window
    width: 320
    height: 240
    visible: true
    title: qsTr("Hello World")
}
```

- Pour le reste, il faut utiliser les mêmes outils (Qt Linguist...) que pour l'approche Qt traditionnelle.


# Le langage QML : Introduction à la syntaxe QML

- Une application QML reste un programme C++

```
#include <QGuiApplication>
#include <QQuickView>

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

    QQuickView view;
    view.setSource(QUrl("main.qml"));
    view.show();

}
return app.exec();
```

# Premiers composants

- Quelques premiers composants QML :
  - Text : l'équivalent d'un label.

```
Text {
      id: helloText
      text: "Hello world!"
      x: 30; y: 30
      font.pointSize: 24; font.bold: true
}
```

  - TextInput : un champ de saisie sur une unique ligne de texte.
  - TextEdit : un champ de saisie sur plusieurs lignes de texte.

```
TextEdit {
      anchors.centerIn: parent
      color: "red"
      width: 200
}
```


# Premiers composants

- Button : un bouton classique et cliquable.

```
Button {
      text: "Click Me"
      x: 300; y: 300
      width: 100
}
```

- Image : pour introduire une image.
  - Il est possible de prendre l'image sur Internet.

```
Image {
      source : "qrc:/folder/image.png"
      width: 300; height: 200
      fillMode: Image.PreserveAspectCrop
}
```

# Quelques autres types QML de base

- Item : une zone rectangulaire neutre
- Rectangle : un Item avec certaines caractéristiques supplémentaires (color, radius...)
- Timer : permet de déclencher un signal à intervalle régulier.

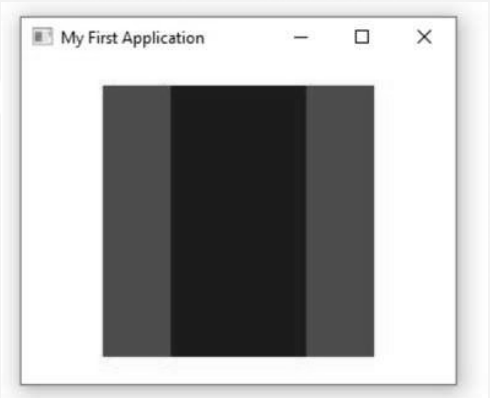
- Pour obtenir la liste complète des types QML supportés consulter le lien suivant :

**https://doc.qt.io/qt-6/qmltypes.html**


# Le langage QML : Les bindings dynamiques

- Il est possible de définir une propriété d'un composant QML à partir de celle d'un autre composant.
  - On parle d'un « binding dynamique ».

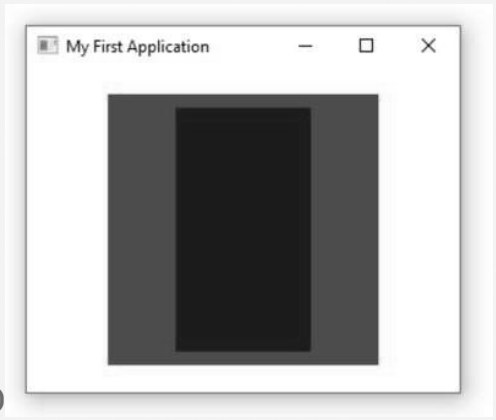
```
Rectangle {
    x: 60; y: 20
    width: 200; height: 200
    color: "red"
    Rectangle {
        x: 50
        width: 100
        height: parent.height
        color: "blue"
    }
}
```




# Le langage QML : Les bindings dynamiques

- Il est possible d'effectuer des calculs dans un binding dynamique.
  - Il est à noter que le positionnement d'un composant QML se fait toujours relativement à son parent.

```
Rectangle {
   x: 60; y: 20
   width: 200; height: 200
   color: "red"
   Rectangle {
        x: 50; y: 10
        width: 100
        height: parent.height - 20
        color: "blue"
}
```





# Le langage QML : Les bindings dynamiques

- Voici un autre exemple d'utilisation du binding : un cadre s'adapte à la taille de l'image qu'il contient

```
Rectangle {
   width: logo.width + 20
   height: logo.height + 20
   anchors.centerIn: parent
   color: "red"
   Image {
        id: logo
        x: 10; y: 10
        source: "qrc:/LogoQML.png"
   }
}
```


# Le langage QML : Les bindings dynamiques

- Si vous souhaitez obtenir des zones scrollables, vous pouvez utiliser un composant « Flickable ».
  - Un binding est donc encore nécessaire pour lier la zone scrollable à son contenu.

```
Flickable {
    anchors.centerIn: parent
    width: logo.height
    height: logo.height
    contentWidth: logo.width
    contentHeight: logo.height
    clip: true
    Image {
        id: logo
        source: "qrc:/LogoQML.png"
    }
}
```



# Exemple

- Il définit deux objets : un rectangle et un texte contenu dans le rectangle,

```
import QtQuick
Rectangle {
    id: page
    width: 320; height: 480
    color: "lightgray"
    Text {
        id: helloText
        text: "Hello world!"
        y: 30
        anchors.horizontalCenter: page.horizontalCenter
        font.pointSize: 24; font.bold: true
    }
}
```

# Le langage QML : Les signaux et les « signal handlers »

- Les différents composants QML peuvent, bien entendu, déclencher des signaux (des événements).
  - Ces signaux peuvent être pris en charge directement dans le QML : on parle alors de « signal handler ».
  - Un « signal handler » (un gestionnaire d'événements) se reconnaît car son nom commence par « on » et chaque nouveau mot reprend par une majuscule.

```
Button {
      text: "Cliquer moi"
      anchors.centerIn: parent
      width: 200
      onClicked: console.log("Clic sur bouton« )
}
```

- Le « console.log » est produit sur la console de debug de QtCreator.


# Le langage QML : Les signaux et les « signal handlers »

- Un signal handler peut exécuter plusieurs instructions : pour ce faire, il faut définir un bloc d'instructions via une paire d'accolades.
  - Dans ce cas, il est courant de retrouver les « ; » en fin d'instructions, même si dans les faits ils ne sont pas obligatoires.

```
Button {
      text: « Cliquer ici"
      anchors.centerIn : parent
      width: 200
      onClicked: {
          this.width = 100;
          this.scale += 0.1;
          console.log("Clic sur le bouton !");
          if (this.scale > 0.3) {
              console.log(« Trop de clics !!");
          }
      }
}
```

# Les signaux et les « signal handlers » MouseArea

- Certains éléments QML ne supportent pas directement un gestionnaire «***onClicked*** ».
  - On peut alors y mettre une zone ***« MouseArea »*** pour y intercepter des événements souris.

```
Rectangle {
     id: rect
     width: 200
     height: 30
     anchors.centerIn: parent
     color: "red"
     MouseArea {
          anchors.fill: parent
          onClicked:
              rect.color = "blue"
     }
}
```


# Les signaux et les « signal handlers » MouseArea hoverEnabled

- Attention : certains signaux relatifs à la souris ne déclenchent pas systématiquement sur un ***« MouseArea ».***
  - Il est nécessaire de les configurer (***hoverEnabled*** par exemple).

```
Rectangle {
     x: 10; y: 300
     width: 100; height: 30
     color: "red"
     MouseArea {
         anchors.fill: parent
         hoverEnabled: true
         onEntered: console.log("Entrée")
         onExited: console.log("sortie")
     }
}
```

# Les signaux et les « signal handlers » preventStealing

- ATTENTION : dans certains cas subtils, le ***« MouseArea »*** pourrait ne pas recevoir certains événements.
  - Imaginons que le ***« MouseArea »*** est placée dans un élément qui filtre les événements de la souris enfant, tels que Flickable, les événements de la souris peuvent être capturés par le ***« Flickable »***, si un geste est reconnu par l'élément parent.
  - Par défaut, la propriété ***« preventStealing »*** du MouseArea est fixée à false.
  - Si ***« preventStealing »*** est défini sur true, votre MouseArea recevra bien les événements de la souris.

```
MouseArea {
// . . .
     hoverEnabled: false
     preventStealing: true
}
```


# Les signaux et les « signal handlers » TapHandler

- Dans le cas d'une application tactile, vous pouvez opter pour l'élément TapHandler (en lieu et place du MouseArea).

```
Rectangle {
     id: rect
     width: 100; height: 100
     TapHandler {
          onPressedChanged: {
               console.log("Ecran tactile touché?",pressed)
          }
          onLongPressed: {
               console.log("Ecran touché longuement")
          }
     }
}
```

# Les signaux et les « signal handlers » DragHandler

- L'élément DragHandler permet de prendre en charge les événements relatifs à du drag'n drop.
  - L'élément visuel sur lequel est placé DragHandler sera déplacé automatiquement.

```
ApplicationWindow {
      id: window
      width: 600; height: 300
      visible: true; title: qsTr("DragHandler sample« )
      Rectangle {
            id: first
            x: 10; y: 10; width: 100; height: 100; color: "red"
            DragHandler {}
      }
      Rectangle {
            x: first.x; y: first.y + 140; width: 100; height: 100
            color: "green";
      }
}
```


# Les signaux et les « signal handlers » Timer

- Le composant Timer permet d'effectuer des traitements périodiques.
  - Le signal handler ***« onTriggered »*** permet de spécifier le code à exécuter à chaque intervalle.

```
Timer {
      running: true
      interval: 1000
      repeat: true
      onTriggered: console.log("bip!")
}
```

# Les signaux et les « signal handlers » Signal implicite et alias

- Toute propriété QML est associée à un signal de « changement de valeur » : on parle de signal implicite.

```
Button {
      x: 10; y: 10
      width: 100; height: 20
      text: "Lancer la démo"
      onClicked: width += 10
      onWidthChanged: console.log("largeur changée")
}
```

- Il est possible de définir des alias sur des propriétés.

```
Button {
      id : me
      property alias content: me.text
      x: 10; y: 10; width: 100; height: 20
      text: "Lancer la démo"
      onClicked: content = "Trop fort !"
}
```


# Les signaux et les « signal handlers » Component.onCompleted

- Il est possible de déclencher un gestionnaire d'événement, pour chaque élément graphique, une fois que l'environnement QML complet a été initialisé.
- A ce stade, l'instance associée sera pleinement initialisée.
- Pour se faire, il faut s'accrocher au signal **Component.onCompleted**.

```
Rectangle {
      id: aRectangle
      color: "red"
      Component.onCompleted = {
            console.log('Rectangle instancié avec la couleur
${color}');
      }
}
```

- **Component.onCompleted** est un signal attaché : on peut l'invoquer sur n'importe quel élément.
- Ne cherchez pas à vous connecter directement à onCompleted : cela ne marchera pas.

# Les signaux et les « signal handlers » Component.onCompleted

- Attention, l'ordre de déclenchement des gestionnaires **Component.onCompleted** n'est pas garanti.

```
Rectangle {
      Component.onCompleted: {
          console.log("External rectangle is completed");
}
Rectangle {
      color: "red"
      Component.onCompleted: {
           console.log("Internal rectangle is completed -"+
                       this.color);
      }
}
```

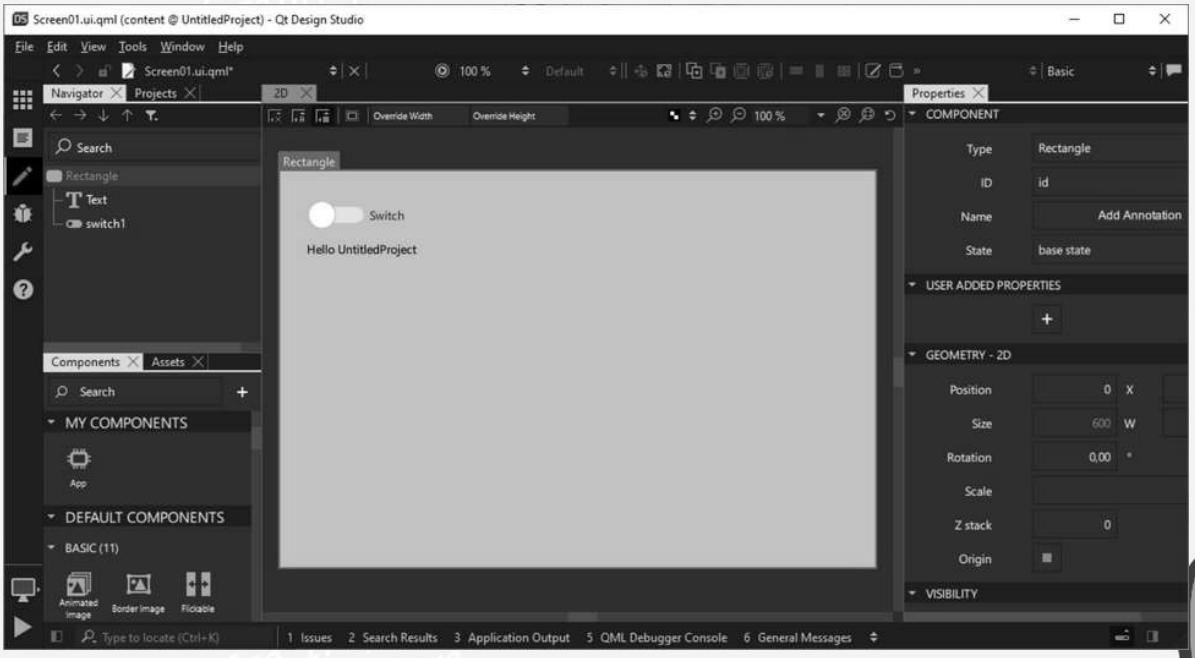

# Les signaux et les « signal handlers » binding dynamique

- Il est possible de définir un binding dynamique via JavaScript dans un « signal handler » :

```
Row {
      TextField {
          id: valuelnput; width: 200
      }
      Button {
           text: « Premier"
           onClicked: function() {
                secondButton.text = Qt.binding(function() {
                     return valuelnput.text;
                });
           }
      }
      Button {
           id: secondButton; text: "Second"
      }
}
```

# Le langage QML : Utilisation du Qt Design Studio

- Voici l'éditeur QML en mode « Design »




# Le langage QML : Utilisation du Qt Design Studio

- Et voici l'éditeur en mode QML avec le code produit pour l'exemple précédent.