QML Programming

Second Edition, Ver 2.0

Qt Korea Developer Community

www.qt-dev.com

QML can implement GUI, functions can be implemented in C++,
so you can improve code reusability.

Jesus loves you

QML Programming

Release version 2.0 (13 June, 2024)

Homepage URL www.qt-dev.com

Preface

We distribute this book free of charge to anyone interested in Qt.

If there is one thing I pray for, it is that those who are still unbelievers will hear the gospel and turn to our Father.

In addition, Jesus, His only begotten Son, came to this earth and took our sins upon Himself. By the precious blood of Jesus, our sins have been forgiven, and God the Father is waiting for all unbelievers to return to Him out of love for His children.

If any of you who have read this book are still unbelievers, we pray and bless you with our earnest hope that you will accept Jesus as your Saviour and be born again as a child of faith. We also pray that God's good grace will be with you. Amen.

37. Jesus replied: " 'Love the Lord your God with all your heart and with all your soul and with all your mind.'

38. This is the first and greatest commandment.

39. And the second is like it: 'Love your neighbor as yourself.'

Matthew 22:37~39

Table of Contents

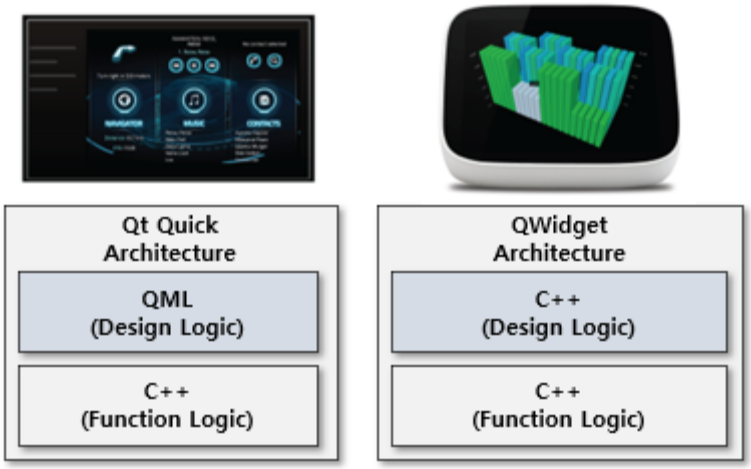
1. What is Qt Quick and QML

Qt Quick is intended to make it easy to design and implement modern GUI interfaces. Qt Quick does not use C++ to implement GUIs. Qt Quick uses an interpreter language called QML. QML stands for Qt Modeling Language.

It is not always necessary to use Qt Quick to implement a GUI. You can use C++ or you can use QML to implement your application with Qt. Whatever you use, there are advantages and disadvantages.

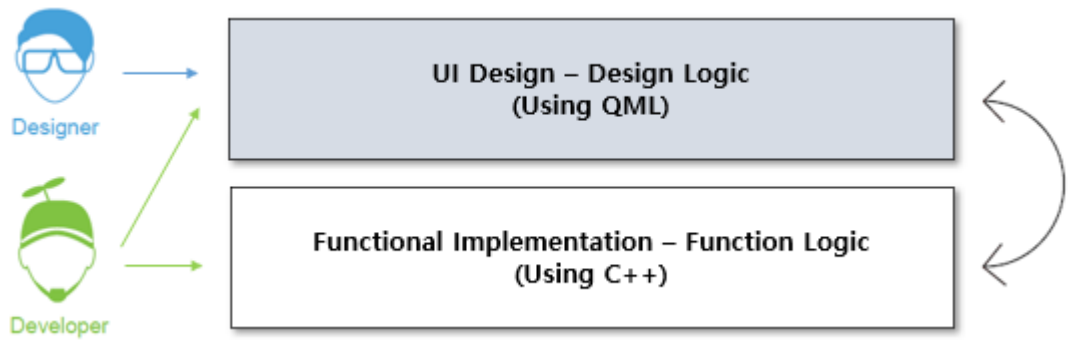
The advantage of using Qt Quick is that you can separate the design logic from the functional logic. For example, design logic refers to the GUI. Functional logic is the functionality. Suppose you have a GUI that is design logic and a function that is executed when a button is clicked. The button on the screen is the design logic, and what happens when you click the button is the functional logic.

The advantage of implementing the design logic using QML provided by Qt Quick is that it is completely separated from the functional logic implemented in C++, so even if the GUI, which is the design logic, is changed, the functional logic, which is the function implementation module, can be reused.



Another advantage of using Qt Quick is that it is suitable for use in environments such as embedded devices with limited user interfaces. For example, a desktop PC can use a variety of user interfaces, such as a keyboard and mouse, for user input. However, Qt

Quick is better suited for situations where the user interface is limited, such as on smartphones or industrial embedded computers where only touch is available.



Conversely, if the computer or embedded device on which your application runs has low or very limited hardware resources, such as CPU, memory, etc. For example, if your GUI needs to be fast and responsive in a constrained environment, it is preferable to implement it in C++.

This means that using QML provided by Qt Quick is not necessarily a good way to develop applications. You should consider whether your application will run on a hardware-intensive computing environment, such as a desktop, or not, and what the different user input interfaces are, before deciding whether to use Qt Quick. Qt Quick can be summarized as follows.

① Qt Quick is used to easily implement modern GUI interfaces.
② Qt Quick uses an interpreter language called QML to implement the GUI.
③ QML is a procedural language. (declarative or interpreter language)

QML used by Qt Quick is an interpreter language. Since it is not compiled like C++ but interpreted sequentially, it is somewhat slower in terms of performance than implementing a GUI in C++.

For example, just as JAVA uses a software stack such as the JAVA Virtual Machine to run, Qt Quick is characterized by the fact that it uses a separate stack called Qt Declarative. An example of a QML-like interpreter language is HTML. It is the same interpreter language that is parsed by a web browser, and QML is the same interpreter language. So far, we've covered Qt Quick and QML.

To summarize Qt Quick, the advantage is that when you develop an application using Qt Quick, the design logic and function logic are separated, so if you utilize function logic except design logic in a project developed with Qt Quick, you can increase reusability.

On the downside, Qt Quick uses an interpreter language called QML rather than C++ for Design Logic, so it is slower than C++. However, the slowness of Qt Quick compared to C++ can be overcome if sufficient hardware resources are available.

So far, we've discussed the features and benefits of Qt Quick. Using Qt Quick is not always a good thing. In fact, there are times when it is better not to use it. Before using Qt Quick, it is recommended that you review whether the software you want to develop meets the features and advantages of Qt Quick mentioned above.

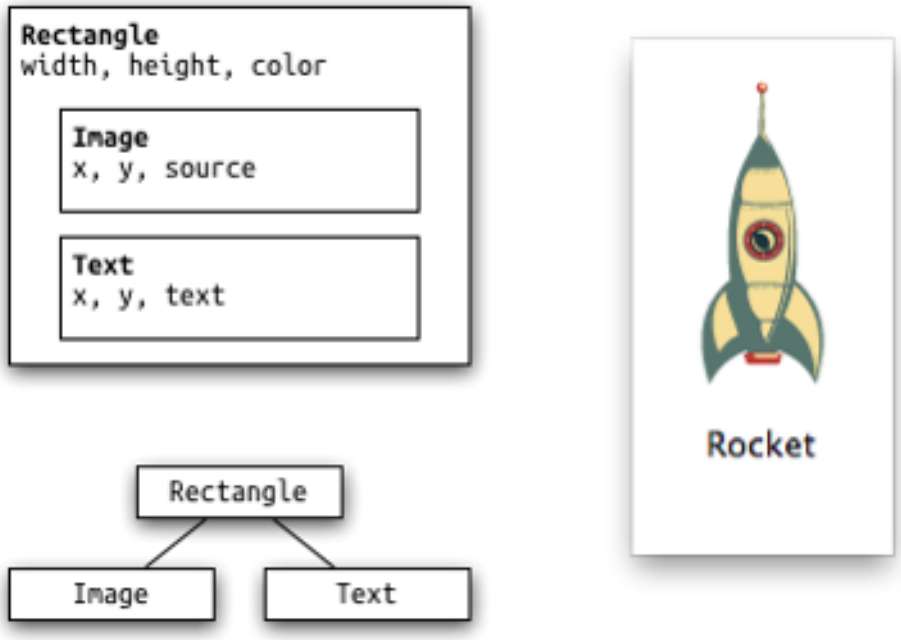
2. QML Basic

In this chapter, you will learn the essentials of implementing a GUI using QML, including QML syntax, data types, QML elements, event handling, and components. This chapter is organized as shown in the following table.

Title	Description
QML Basic syntax	A description of the syntax and constructs used by QML.
Types	Provide a QML type, for example, to implement a GUI using QWidget in C++.
Event	Handling user input events
Dynamic GUI Configuration with Loader Types	Provide functionality for dynamically calling another QML file or type from QML
Canvas	Provide functionality for drawing within a QML region, such as the QPainter class used by the QWidget class.
Graphics Effects	Use effects like blending, masking, blurring, etc.
QML Module Programming	Provide modularity so that frequently used custom types in QML can be reused like a library
GML에서 JavaScript 사용	Using JavaScript within QML
Dialog	Provide a variety of dialogs, including simple dialogs, color dialogs, file dialogs, font dialogs, etc.
Layout	Layout provided by QML, such as ColumnLayout, GridLayout, etc.
Type Positioning	Provides the ability to get information about a type at a specific location among multiple types.

2.1. QML Basic

QML is similar to HTML structure, and QML has a basic hierarchical structure. For example, suppose you want to output an image inside a rectangular area.



In the figure above, if you want to display the image of a rocket on the right and the text "Rocket" below in a rectangular area, you can declare a Rectangle first and display the image and text as shown in the figure above. In other words, QML has a hierarchical structure like HTML, unlike grammars such as C/C++, which has a hierarchy of elements. The following source code is an example of QML.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    id: root
    width: 120
    height: 240
    color: "#D8D8D8"
```

```
    Image {
        id: rocket
        x: (parent.width - width) / 2
        y: 40
        source: 'assets/rocket.png'
    }

    Text {
        y: rocket.y + rocket.height + 20
        width: root.width
        horizontalAlignment: Text.AlignHCenter
        text: 'Rocket'
    }
}
```

In the QML example above, the topmost import statement is similar to the "#include<...>" statement in C++ syntax. The following is the syntax structure of the import statement <ModuleIdentifier> specifies the module name and <Version.Number> indicates the version.

```
import <ModuleIdentifier> <Version.Number> [as <Qualifier>]
```

<Qualifier> allows the user to specify a pseudonym or alias to replace the module name. This will be covered in more detail later. If no version information is specified, the latest version is used.

In the source code of the QML example above, the Window type (also called Element) is the type that creates the highest level window.

The items specified under the Window type (inside) are called Properties. And within the Window type, there can be a hierarchy of subtypes, such as Image and Text, that are internal to it.

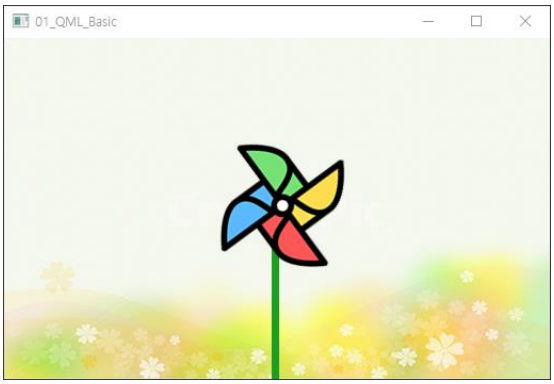
As shown in the example, the visible property of the Window type determines whether it is visible on the screen. The Id property specifies the unique name or ID of the type. The width and height are the horizontal and vertical dimensions. The Color property can specify a color within the Window type.

The Image type can display image resources within a Window type. The parent.width used within Image refers to the width of the Window type. x and y are the coordinates of the starting position to be displayed in the Window type of the Image type. The source

property specifies the location and filename of the image resource. The rocket.y value used in a Text type can refer to the y value of an Image type whose id property is rocket.

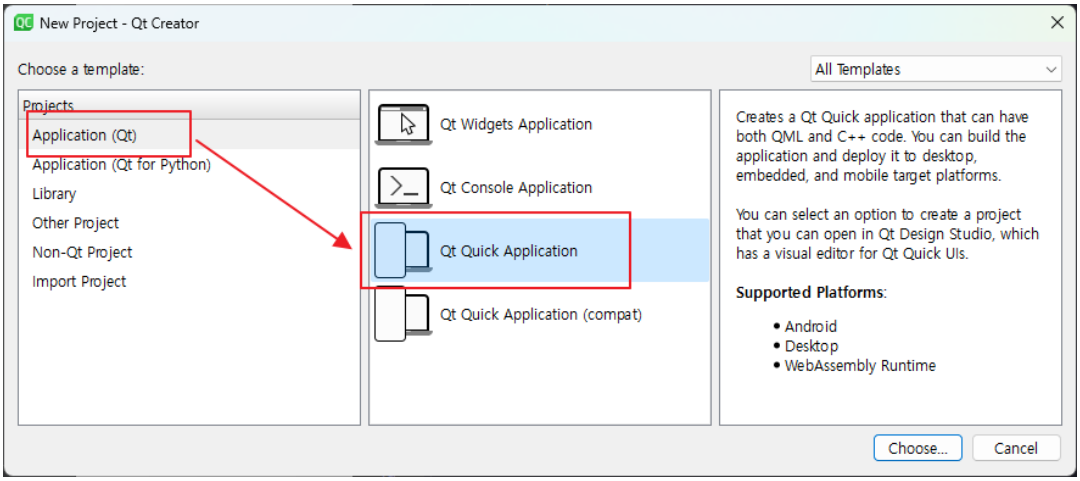
- QML example with image and mouse

This example rotates an image when it is clicked while holding down the left and right mouse and keyboard keys inside a Window type region.

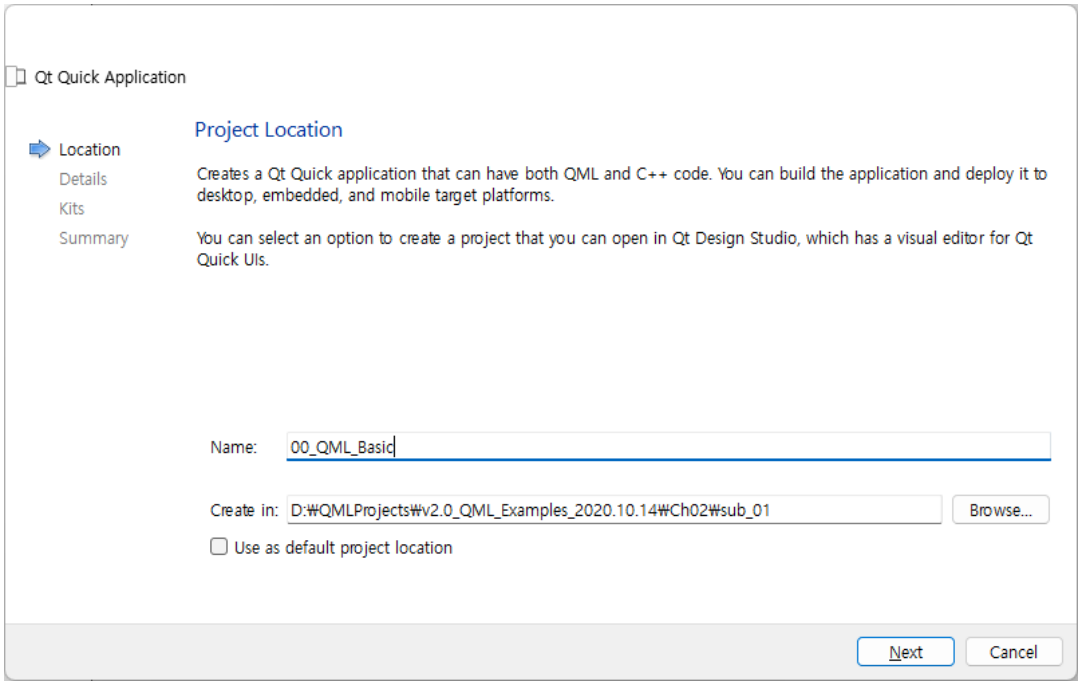


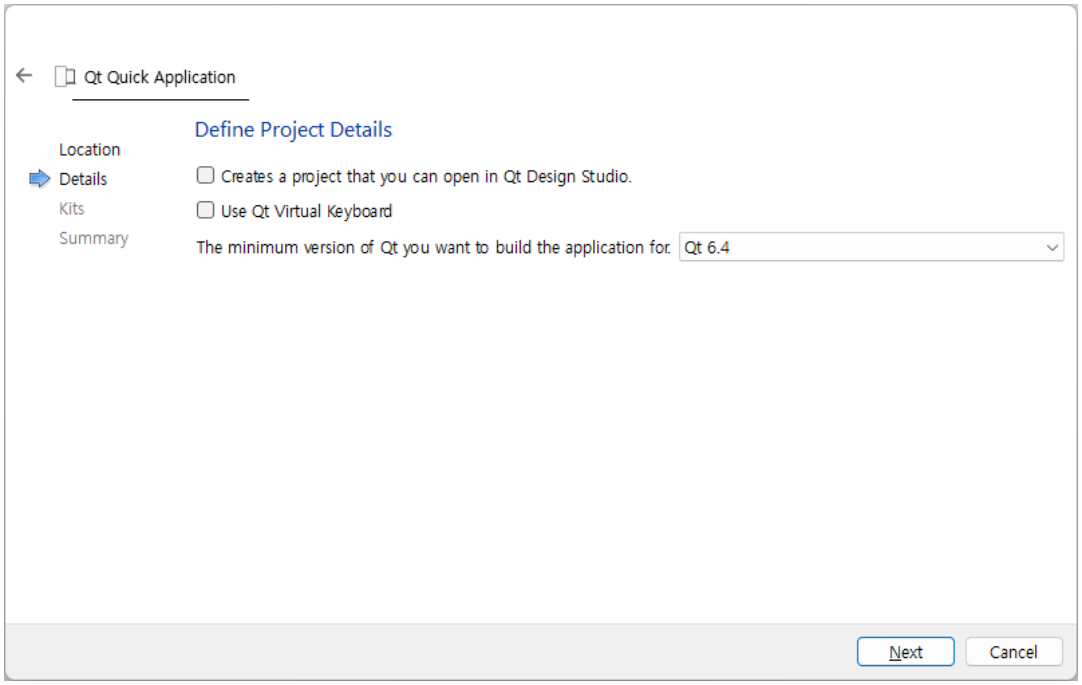
As shown in the figure above, clicking the mouse on the screen will rotate the pinwheel image, and clicking the left keyboard key will rotate the pinwheel image to the left, and clicking the right keyboard key will rotate the pinwheel image to the right. To develop an application using Qt Quick, create a new project in Qt Creator.

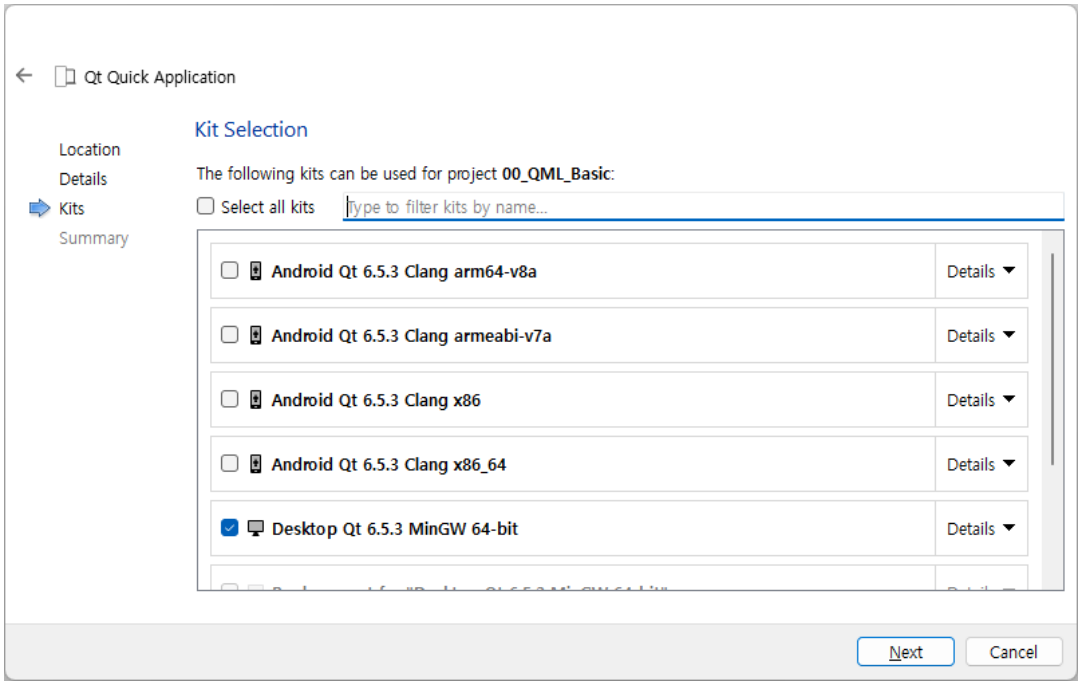
To create a new project, click [New Project] under the [File] menu. Then select [Qt Quick Application].

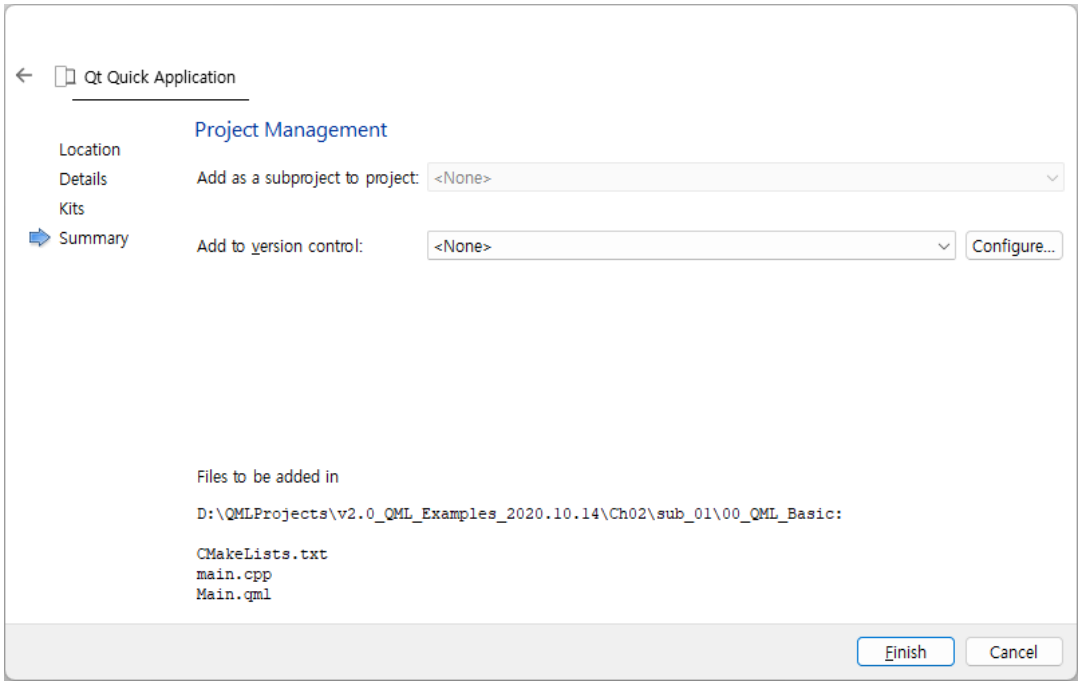


Next, enter a name for your project in the Name field.


Qt Quick Application
Location
Details
Kits
Summary
Project Location
Creates a Qt Quick application that can have both QML and C++ code. You can build the application and deploy it to desktop, embedded, and mobile target platforms.
You can select an option to create a project that you can open in Qt Design Studio, which has a visual editor for Qt Quick UIs.
Name:
00_QML_Basic
Create in:
D:₩QMLProjects₩v2.0_QML_Examples_2020.10.14₩Ch02₩sub_01
Browse...
Use as default project location
Next
Cancel


Qt Quick Application
Location
Details
Kits
Summary
Define Project Details
Creates a project that you can open in Qt Design Studio.
Use Qt Virtual Keyboard
The minimum version of Qt you want to build the application for.
Qt 6.4
Next
Cancel





Click the [Finish] button in the dialog window to complete the project creation, as shown in the figure above.

Starting with Qt Creator 10.x, qmake is not an option when creating Qt Quick-related

(QML) projects. Therefore, CMake is always used when creating projects. However, it is possible to use qmake directly. For example, projects created with qmake (with a .pro extension) can be used in Qt Creator.

However, if you use qmake manually or have an existing qmake, you can import it and use it. Therefore, you should use CMake as your build tool when creating your project. Once the project has been created, let's take a look at main.cpp.

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


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

    QQmlApplicationEngine engine;
    const QUrl url(u"qrc:/00_QML_Basic/Main.qml"_qs);
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreationFailed,
        &app, []() { QCoreApplication::exit(-1); },
        Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

As shown in the example source code above, the QQmlApplicationEngine class is responsible for loading the QML file. You can load the QML file using the load( ) member function at the bottom of the source. Next, create the main.qml file as shown below.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    id: root
    width: 512; height: 320; color: "#D8D8D8"

    property int rotationStep: 90
```

```
    BorderImage {
        source: "images/background.png"
    }

    Image {
        id: pole
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.bottom: parent.bottom
        source: "images/stick.png"
    }

    Image {
        id: wheel
        anchors.centerIn: parent
        source: "images/wheel.png"
        Behavior on rotation {
            NumberAnimation {
                duration: 250
            }
        }
    }

    MouseArea {
        anchors.fill: parent
        onPressed: {
            wheel.rotation += rotationStep
        }
    }

    Item {
        anchors.fill: parent
        focus: true
        Keys.onLeftPressed: {
            console.log("move left")
            wheel.rotation -= root.rotationStep
        }

        Keys.onRightPressed: {
            console.log("move right")
```

```
            wheel.rotation += root.rotationStep
        }
    }
}
```

The Window type is the highest-level type in QML. Its size is specified using width and height to specify the horizontal and vertical dimensions. The color property is the background color of the Window type.

The BorderImage type specifies an image to use as the background image. Of type Image, an Image with id property pole is a pinwheel pole, and an Image with id property wheel is a pinwheel image. Each image looks like this

background.png

stick.png

wheel.png

The MouseArea type handles mouse events that occur within the Window area. When the mouse button is clicked, it rotates 90 degrees as defined by the onPressed type of the MouseArea type. During the 90-degree rotation, the NumberAnimation type declared by the Image type of the wheel id is animated.

This animation is applied when a mouse event occurs in the MouseArea and rotates 90 degrees. This means that the 90-degree rotation does not happen instantaneously, but takes 250 milliseconds to rotate 90 degrees.

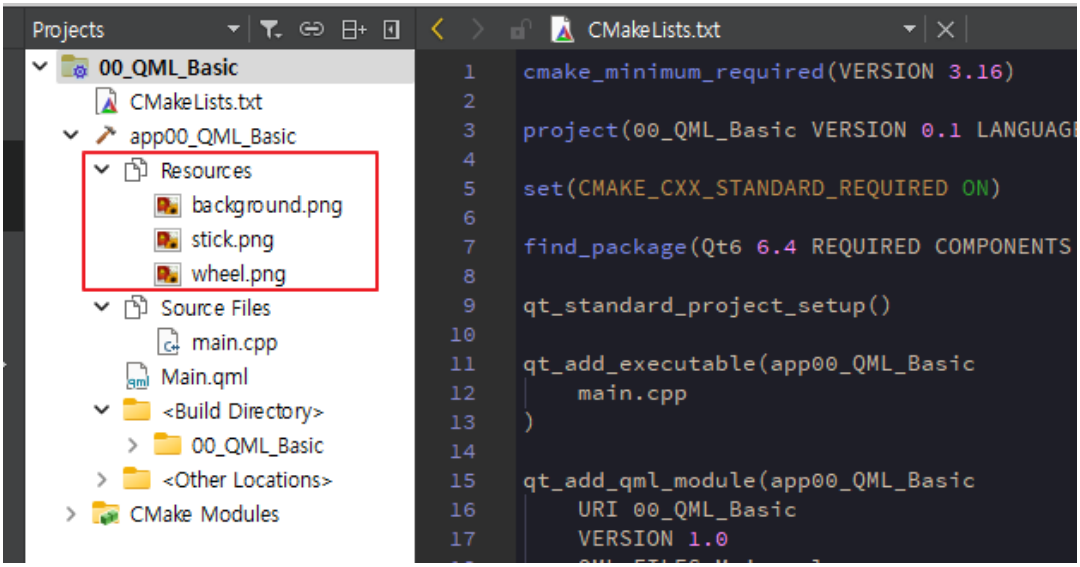
The Item type rotates -90 degrees when the left arrow key is pressed on the keyboard. To add the images to the project, create an image directory in the project subdirectory and copy the three images. The image files are attached to the example project.

After copying the images to the project directory, open the CMakeLists.txt file and add the following to it

```
...
qt_add_executable(app00_QML_Basic
```

```
    main.cpp
)

qt_add_qml_module(app00_QML_Basic
    URI 00_QML_Basic
    VERSION 1.0
    QML_FILES Main.qml
    RESOURCES
        "images/background.png"
        "images/stick.png"
        "images/wheel.png"
)
...
```

After adding and saving as shown above, you can see the Resources item added to the project list as shown below.



Next, build and run the source, then try clicking the mouse to see if the pinwheel spins. Also, check to see if the pinwheel rotates when you press the left or right arrow keys on your keyboard.

The source code for this example can be found in the 00_QML_Basic directory.

2.2. Types

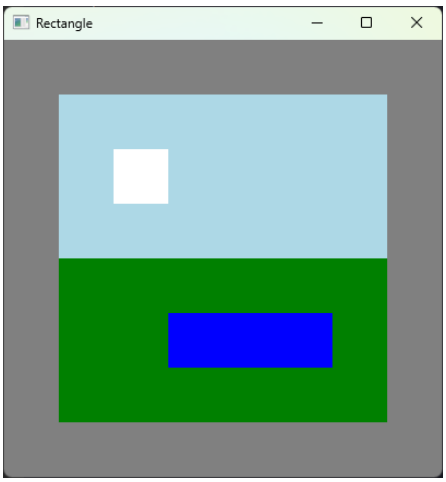
In QML, a Type is an object that you provide to implement a GUI. Window, Image, MouseArea, Item, etc. used in the previous section are called types in QML. QML provides a variety of types, and the most commonly used types are shown in the following table.

Type name	Description
Rectangle	Items of type Rectangular
Image	Items for displaying images
BorderImage	Show background image in a specified area
AnimatedImage	Items for displaying items like animated GIFs
AnimatedSprite	Items for displaying animations using consecutive frames
SpriteSequence	Display multiple items using consecutive frames
Text	Items for displaying strings
Window	Items for creating parent windows
Item	Custom items
Anchors	Provide features like layouts
Screen	Information about the area where the item is displayed
Sprite	Specifying animations for displayed items
Repeater	Provide multiple items you create to apply to your model
Loader	Show separated items, such as modules or files
Transform	Moving items
Scale	Controlling the size of an item
Rotate	Rotate items
Translate	Define an item's movement properties

- Rectangle

The Rectangle type provides functionality for displaying items in a rectangular area. And the Rectangle type can be used in nested form.

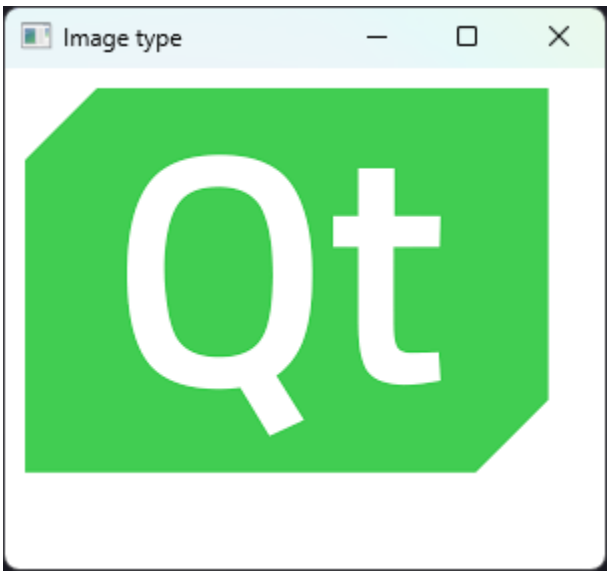
```
import QtQuick
import QtQuick.Window

Window {
    title: "Repeater"
    visible: true
    width: 400; height: 100

    Row {
        Repeater {
            model: 3
            anchors.top: parent.top
            Rectangle {
                width: 100; height: 40
                border.width: 1
                color: "yellow"
            }
        }
    }
}
```



- Image

The example below is for displaying an image using the Image Type.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 300; height: 250
    title: "Image type"

    Rectangle
    {
        width: 300; height: 250
        color: "white"

        Image {
            x: 10; y: 10
            source: "images/qtlogo.png"
        }
    }
}
```



- AnimatedImage

The AnimatedImage type can render an animated GIF image on the screen and provides start and stop functions.

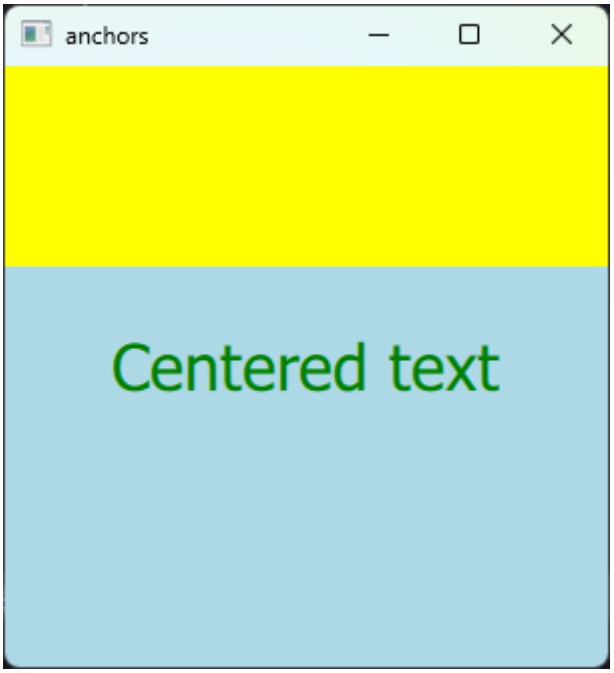
```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: ani.width
    height: ani.height + 8
    title: "AnimatedImage"

    AnimatedImage {
        id: ani
        source: "images/ani.gif"
    }

    Rectangle {
        property int frames: ani.frameCount
        width: 4
        height: 8
        x: (ani.width - width) * ani.currentFrame / frames
        y: ani.height
        color: "red"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            if( ani.paused === true )
                ani.paused = false
            else
                ani.paused = true
        }
    }
}
```

- anchors

Just as the Qt C++ API uses layouts like QHBoxLayout, QVBoxLayout, etc. to arrange widgets, QML provides anchors. The following is an example of using anchors to position QML types.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 300; height: 300
    title: "anchors"

    Rectangle {
        width: 300
        height: 300
        color: "lightblue"
        id: rectangle1

        Rectangle {
            id: subRect
            width: 300
            height: 100
            color: "yellow"
        }
```

```
        Text {
            text: "Centered text";
            color: "green"
            font.family: "Helvetica";
            font.pixelSize: 32
            anchors.top: subRect.bottom
            anchors.centerIn: rectangle1
        }
    }
}
```



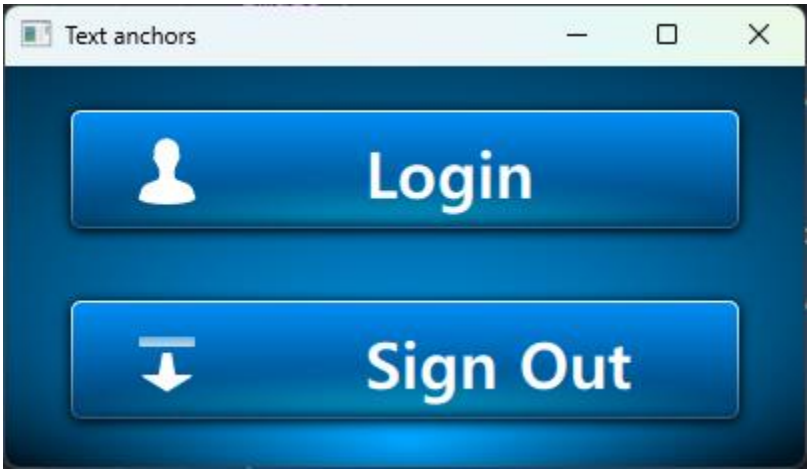
- Example of placing multiple Types using anchors

The following QML example source code is an example of using anchors to place QML Types.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
```

```
    width: 400; height: 200
    title: "Text anchors"

    Rectangle {
        width: 400; height: 200

        Image {
            source: "./images/bluebackground.png"
        }

        BorderImage {
            source: "./images/bluebutton.png"
            border { left: 13; top: 13; right: 13; bottom: 13 }
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.top: parent.top
            anchors.topMargin: 15
            width: 350; height: 75
            Image {
                anchors.left: parent.left
                anchors.leftMargin: 40
                anchors.verticalCenter: parent.verticalCenter
                source: "./images/login.png"
            }
            Text {
                anchors.left: parent.horizontalCenter
                anchors.leftMargin: -20
                anchors.verticalCenter: parent.verticalCenter
                text: "Login"
                font.bold: true
                color:"white"
                font.pixelSize: 32
            }
        }

        BorderImage {
            source: "./images/bluebutton.png"
            border { left: 13; top: 13; right: 13; bottom: 13 }
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.bottom: parent.bottom
            anchors.bottomMargin: 15
            width: 350; height: 75
            Image {
```

```
                anchors.left: parent.left
                anchors.leftMargin: 40
                anchors.verticalCenter: parent.verticalCenter
                source: "./images/signout.png"
            }
            Text {
                anchors.left: parent.horizontalCenter
                anchors.leftMargin: -20
                anchors.verticalCenter: parent.verticalCenter
                text: "Sign Out"
                font.bold: true
                color:"white"
                font.pixelSize: 32
            }
        }
    }
}
```

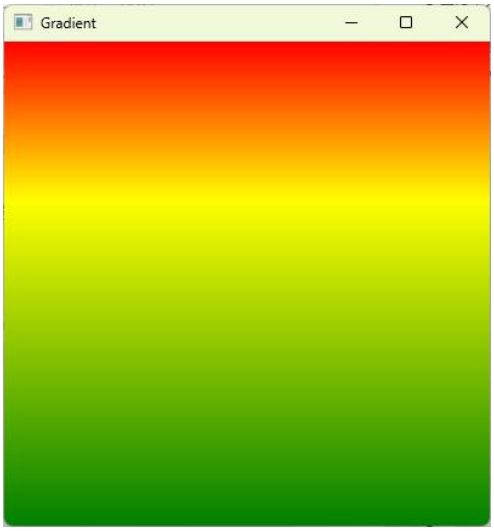


- Gradient

Gradient provides the ability to specify a range of colors for a given area. You can specify a range of values between 0.0 and 1.0. The following example shows an example of using Gradient.

```
import QtQuick
import QtQuick.Window
```

```
Window {
    visible: true
    width: 400; height: 400
    title: "Gradient"

    Rectangle {
        width: 400; height: 400
        gradient: Gradient {
            GradientStop { position: 0.0; color: "red" }
            GradientStop { position: 0.33; color: "yellow" }
            GradientStop { position: 1.0; color: "green" }
        }
    }
}
```



- SystemPalette

SystemPalette provides functionality for accessing the Qt application palette, i.e., information about the standard colors used in the application windows. The following example shows an example of using the SystemPalette.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
```

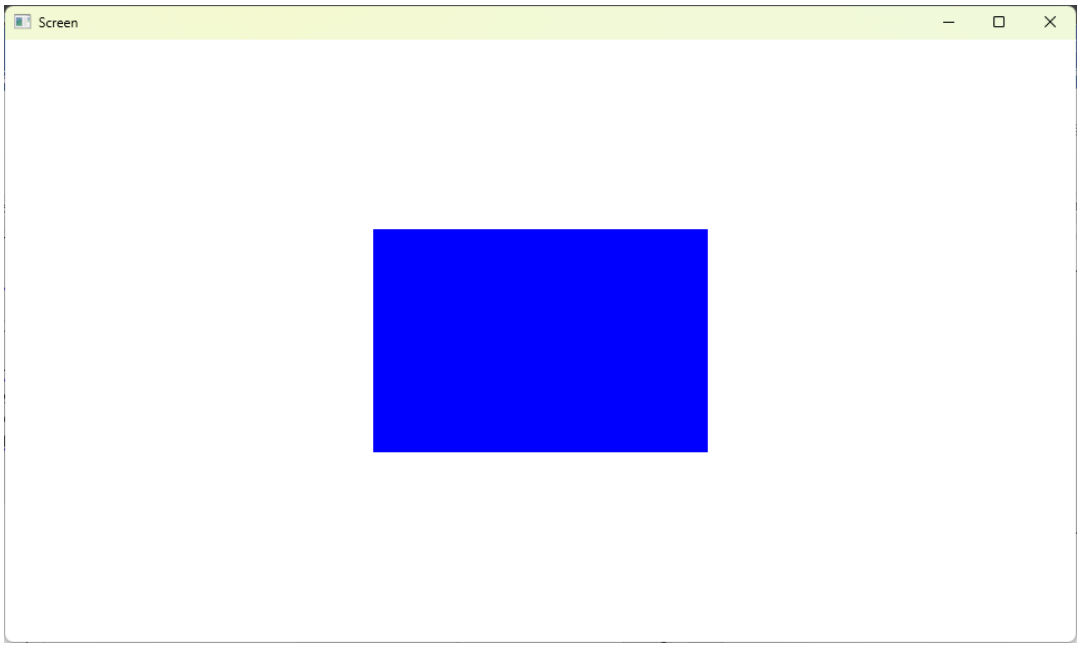
```
    width: 250; height: 100
    title: "SystemPalette"

    Rectangle {
        width: 200; height: 80
        color: myPalette.window

        SystemPalette {
            id: myPalette
            colorGroup: SystemPalette.Active
        }

        Text {
            id: myText
            anchors.fill: parent
            text: "Hello!";
            font.pixelSize: 32
            color: myPalette.windowText
        }
    }
}
```

- Screen

Screen provides information about the screen on which the GUI is loaded. The following is an example of getting the system's screen resolution information.

```
import QtQuick
import QtQuick.Window

Window {
    title: "Screen"
    visible: true
```

```
    width : Screen.width / 2;
    height : Screen.height / 2;

    Rectangle {
        width : 300
        height: 200
        color: "blue"

        anchors.centerIn: parent
    }
}
```

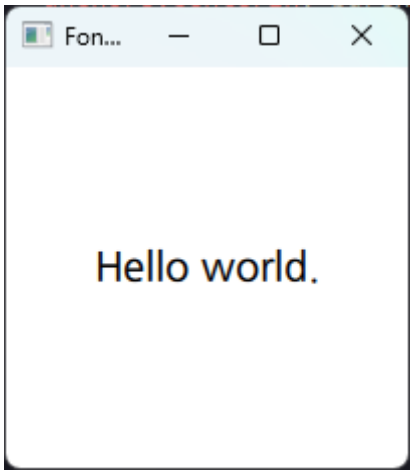


- FontLoader

FontLoader allows you to specify the fonts you want to use.

```
import QtQuick
import QtQuick.Window

Window {
    title: "FontLoader"
    visible: true
```

```
    width: 200; height: 200

    Rectangle {
        width: parent.width
        height: parent.height

        FontLoader {
            id: myFont
            source: "NanumGothic.ttf"
        }

        Text {
            text: "Hello world.";
            font.family: myFont.name
            font.bold: true
            font.pixelSize: 20
            anchors.centerIn: parent
        }
    }
}
```

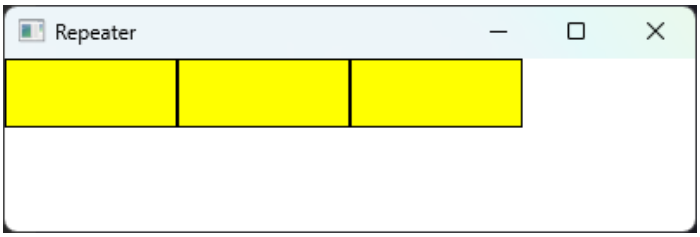


- Repeater

The Repeater type can place the same QML type back-to-back. The following example is an example of using a Repeater.

```
import QtQuick
```

```
import QtQuick.Window

Window {
    title: "Repeater"
    visible: true
    width: 400; height: 100

    Row {
        Repeater {
            model: 3
            anchors.top: parent.top
            Rectangle {
                width: 100; height: 40
                border.width: 1
                color: "yellow"
            }
        }
    }
}
```



- Transformation and Rotation

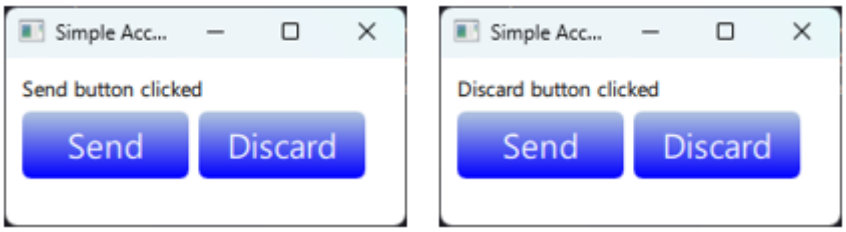
Image provides functionality for displaying images. The following is example source code that uses the Image type to display an image on the screen.

```
import QtQuick
import QtQuick.Window

Window {
    title: "transformation"
    visible: true; width: 1000;  height: 220

    Row {
        x: 10; y: 10; spacing: 10
```

```
        Image {
            source: "images/qtlogo.png"
        }

        Image {
            source: "images/qtlogo.png"
            transform: Rotation {
                origin.x: 30; origin.y: 30
                axis { x: 0; y: 1; z: 0 } angle: 18
            }
        }

        Image {
            source: "images/qtlogo.png"
            transform: Rotation {
                origin.x: 30; origin.y: 30
                axis { x: 0; y: 1; z: 0 } angle: 36
            }
        }

        Image {
            source: "images/qtlogo.png"
            transform: Rotation {
                origin.x: 30; origin.y: 30
                axis { x: 0; y: 1; z: 0 } angle: 54
            }
        }
    }
}
```

- Implementing button functionality with Accessible

This example shows how to create and place a button using Accessible. The following figure shows the example running.



As shown in the image above, clicking the [Send] button displays the message "Send button clicked" in the Text type at the top. And when you click the [Discard] button, it displays the message "Discard button clicked". Let's take a look at Button.qml, the first of the two QML files in this example.

```
import QtQuick

Rectangle {
    id: button

    property alias text : buttonText.text
    Accessible.name: text
    Accessible.description: "This button does " + text
    Accessible.role: Accessible.Button
    Accessible.onPressAction: {
        button.clicked()
    }

    signal clicked

    width: buttonText.width + 20
    height: 50
    gradient: Gradient {
        GradientStop { position: 0.0; color: "lightsteelblue" }
        GradientStop { position: 1.0;
            color: button.focus ? "red" : "blue" }
    }

    radius: 5
    antialiasing: true
    Text {
        id: buttonText
```

```
        text: parent.description
        anchors.centerIn: parent
        font.pixelSize: parent.height * .5
        color: "white"
        styleColor: "black"
    }
    MouseArea {
        id: mouseArea
        anchors.fill: parent
        onClicked: parent.clicked()
    }
}
```

The above example source code is saved as Button.qml. The filename becomes the QML type name when you try to use a QML type that implements a button in another QML file.

Accessible is used to implement custom GUI interfaces such as buttons, text fields, check boxes, etc. Accessible.role allows you to select the type of customized GUI interface. In this example, Button is specified. You can also specify CheckBox, RadioButton, Slider, SpinBox, Dial, ScrollBar, etc.

Accessible.onPressAction specifies the event to fire when the button is clicked. Here we use the signal clicked. This is the same signal we used in C++ in our header file: we defined that we want to use the signal clicked. radius is used to round the corners of the button. antialiasing is used to apply a vector to the backer to display on the screen. Let's take a look at an example that uses Button.qml.

```
import QtQuick
import QtQuick.Window
import "content"

Window {
    title: "Simple Accessible"
    visible: true
    id: window; width: 240; height: 100
    color: "white"

    Column {
        id: column; spacing: 6
        anchors.margins: 10
```

```
        anchors.fill: parent
        width: parent.width

        Text {
            id : status
            width: column.width
        }

        Row {
            spacing: 6
            Button {
                id: sendButton
                width: 100; height: 40; text: "Send"
                onClicked: {
                    status.text = "Send button clicked"
                }
            }
            Button { id: discardButton
                width: 100; height: 40; text: "Discard"
                onClicked: {
                    status.text = "Discard button clicked"
                }
            }
        }
    }
}
```

Column types are placed within Column in the order in which they are declared vertically. The Row type places the types declared within Row in the order they are declared in the horizontal direction. And the Button type is the name of the user-defined type defined in Button.qml. The filename becomes the type name.

2.3. Event

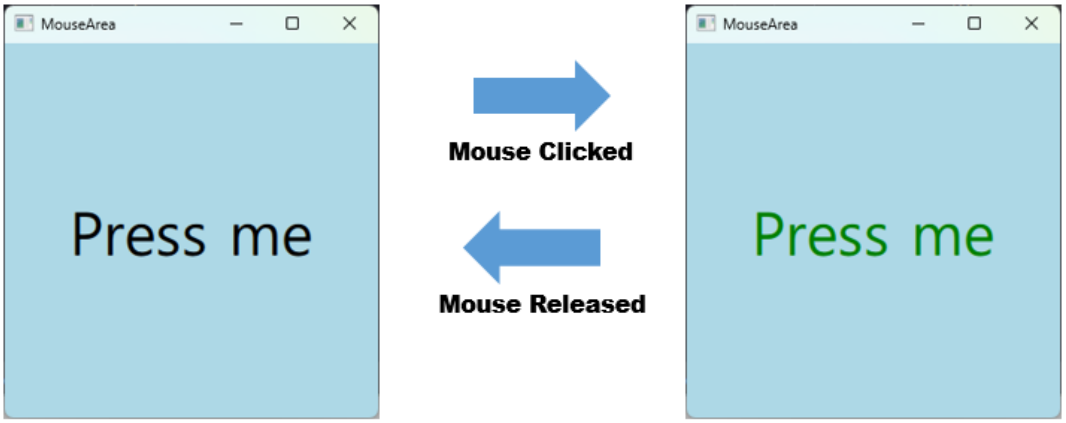
Qt Quick 은 사용자 입력과 같은 이벤트를 처리하기 위해서 다음 표에서 보는 것과 같은 다양한 타입을 제공한다.

Type name	Description
MouseArea	Handling mouse events within specific QML types
Keys	Additional properties provided to manipulate keystrokes
KeyNavigation	Support for arrow key navigation
FocusScope	Adjust keyboard focus
Flickable	Flickable provides a GUI for dragging and dropping items
PinchArea	PinchArea is an invisible item and is typically used in conjunction with a visible item to provide pinch gesture handling for that item.
MultiPointTouchArea	Activation handling for manipulating multi-touch pointers
Drag	Types for specifying drag and drop of items
DropArea	Type for specifying drag and drop in a specific area
TextInput	Handle keystrokes from users
TextEdit	Text Editor to edit text
TouchPoint	A type that contains information about the coordinates of the touch
PinchEvent	Handling events in the Pinch area
WheelEvent	Mouse wheel events
MouseEvent	Mouse Button Events
KeyEvent	Key Events
DragEvent	Events related to Drag events

- MouseArea

In C++-based applications, you can use properties to handle mouse events, such as executing the associated Slot function when a mouse event signal is raised on a QWidget. The following is an example source code that handles mouse events on a MouseArea type.

```
import QtQuick
import QtQuick.Window

Window {
    title: "MouseArea"
    visible: true
    width: 300
    height: 300

    Rectangle {
        width: parent.width
        height: parent.height
        color: "lightblue"

        Text {
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.verticalCenter: parent.verticalCenter
            text: "Press me"; font.pixelSize: 48

            MouseArea {
                anchors.fill: parent
                onPressed: {
                    parent.color = "green"
                    console.log("Press")
                }
                onReleased: {
                    parent.color = "black"
                    console.log("Release")
                }
            }
        }
    }
}
```



In the example source code above, anchors.fill sets the area in which the MouseArea will operate. Since we have specified a Parent, the area of type Text is set as the MouseArea area.

The onPressed property is fired when the mouse button is clicked, and the onReleased property is fired when the mouse button is released.

The source code for this example can be found in the 00_MouseArea directory.

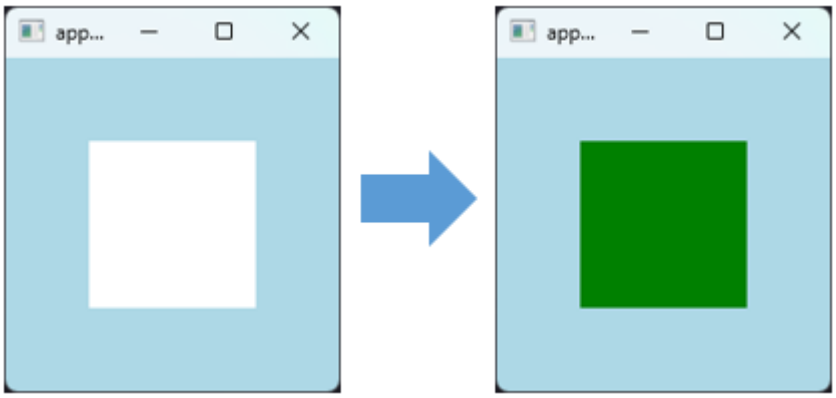
- Using the containsMouse property in the MouseArea area

The containsMouse property can be used to handle events where the mouse is inside the Rectangle.

In order to use the containsMouse property, the hoverEnabled property must be set to true.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 200
    height: 200

    Rectangle {
        width: 200; height: 200; color: "lightblue"
        Rectangle {
            x: 50; y: 50; width: 100; height: 100
            color: mouseArea.containsMouse ? "green" : "white"
```

```
            MouseArea {
                id: mouseArea
                anchors.fill: parent;
                hoverEnabled: true
            }
        }
    }
}
```



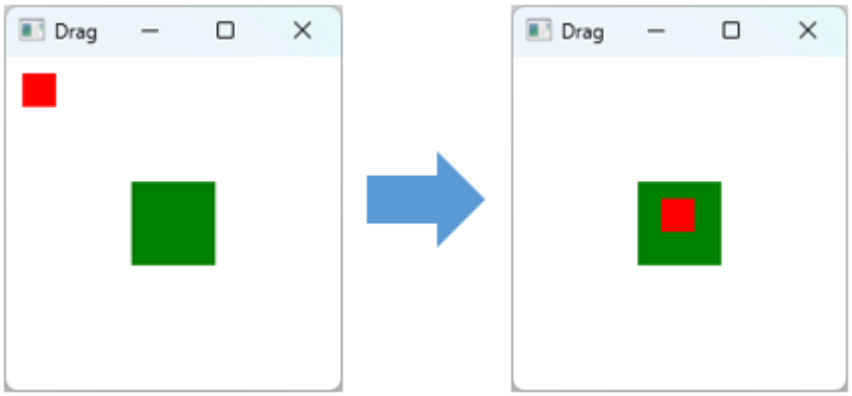
You can find the source code for this example in the 01_containsMouse directory.

- Drag and DropArea

This example shows how to change the background color of the DropArea to green when a Rectangle is dragged inside the DropArea by dragging the mouse inside the Rectangle region.

```
import QtQuick
import QtQuick.Window

Window {
    title: "Drag"
    visible: true
    width: 200
    height: 200

    DropArea {
        x: 75; y: 75
```

```
        width: 50; height: 50
        Rectangle {
            anchors.fill: parent
            color: "green"
        }
    }

    Rectangle {
        x: 10; y: 10
        width: 20; height: 20
        color: "red"

        Drag.active: dragArea.drag.active
        Drag.hotSpot.x: 10
        Drag.hotSpot.y: 10

        MouseArea {
            id: dragArea
            anchors.fill: parent
            drag.target: parent
        }
    }
}
```



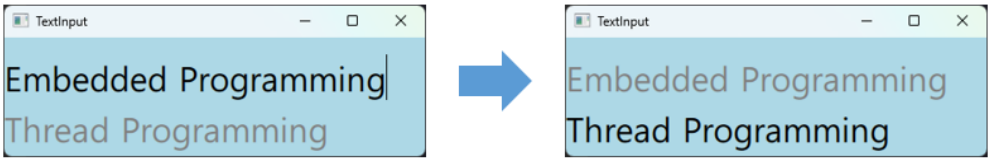
You can find the source code for this example in the 02_drag directory.

- Event Handling with the TextInput Type

The TextInput type can handle keyboard input events. The following example is sample

source code that uses the TextInput type to handle focus events.

```
import QtQuick
import QtQuick.Window

Window {
    title: "TextInput"
    visible: true
    width: 400; height: 112

    Rectangle {
        width: 400; height: 112; color: "lightblue"
        TextInput {
            anchors.left: parent.left; y: 16
            anchors.right: parent.right

            text: "Embedded Programming";
            font.pixelSize: 32
            color: focus ? "black" : "gray"
            focus: true
        }

        TextInput {
            anchors.left: parent.left; y: 64
            anchors.right: parent.right

            text: "Thread Programming";
            font.pixelSize: 32
            color: focus ? "black" : "gray"
        }
    }
}
```



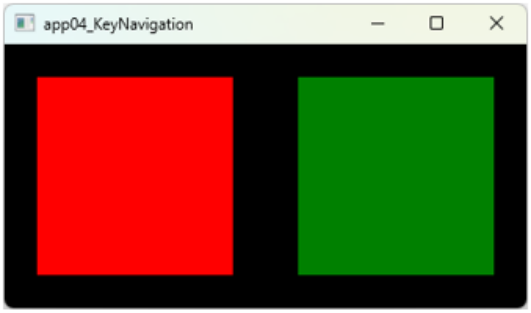
You can find the source code for this example in the 03_TextInput directory.

- KeyNavigation

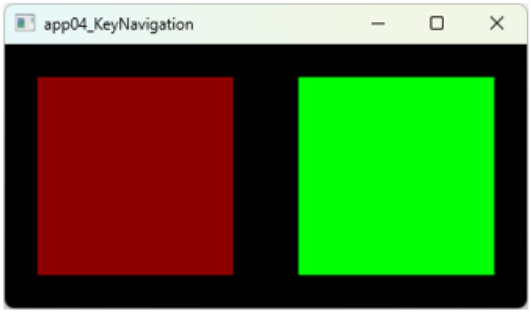
The KeyNavigation type can handle key events for the up, down, left, and right direction of the key. In addition to direction, you can use the KeyNavigation.tab property for TAB key events.

And to handle "Shift + Tab" key events, you can use the KeyNavigation.backtab property.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 400; height: 200

    Rectangle {
        width: 400; height: 200; color: "black"

        Rectangle {
            id: leftRect
            x: 25; y: 25; width: 150; height: 150
            color: focus ? "red" : "darkred"
            KeyNavigation.right: rightRect
            focus: true
        }

        Rectangle {
            id: rightRect
            x: 225; y: 25; width: 150; height: 150
            color: focus ? "#00ff00" : "green"
            KeyNavigation.left: leftRect
        }
    }
}
```



KeyNavigation.left



KeyNavigation.right

You can find the source code for this example in the 04_KeyNavigation directory.

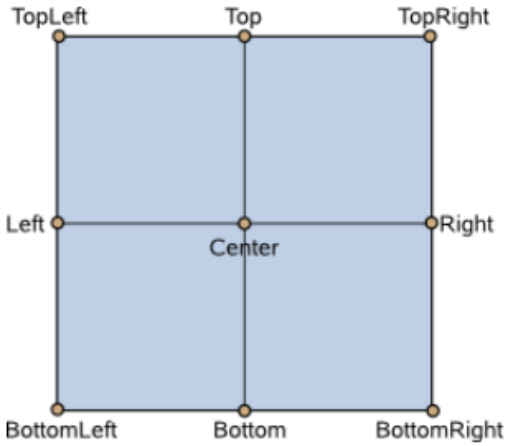
- Keys

The Keys event can handle keyboard events. Keys can use the onPressed and onReleased signal properties.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 240; height: 200

    Rectangle {
        width: 230; height: 190; color: "white"

        Image {
            id: logo
            x: 30; y: 30
            source: "images/qtlogo.png"
            transformOrigin: Item.Center
            //transformOrigin: Item.Left
        }

        Keys.onPressed: (event)=> {
            if (event.key === Qt.Key_Left) {
                logo.rotation = (logo.rotation - 10) % 360
            } else if (event.key === Qt.Key_Right) {
                logo.rotation = (logo.rotation + 10) % 360
            }
        }
```

```
        focus: true
    }
}
```

On Image types, the transformOrigin property allows you to select the direction of the center point for rotating an image of type Image. For example, using Item.Left as the transformOrigin value will cause the rotation center to rotate in the Left direction.



You can find the source code for this example in the 05_Keys directory.

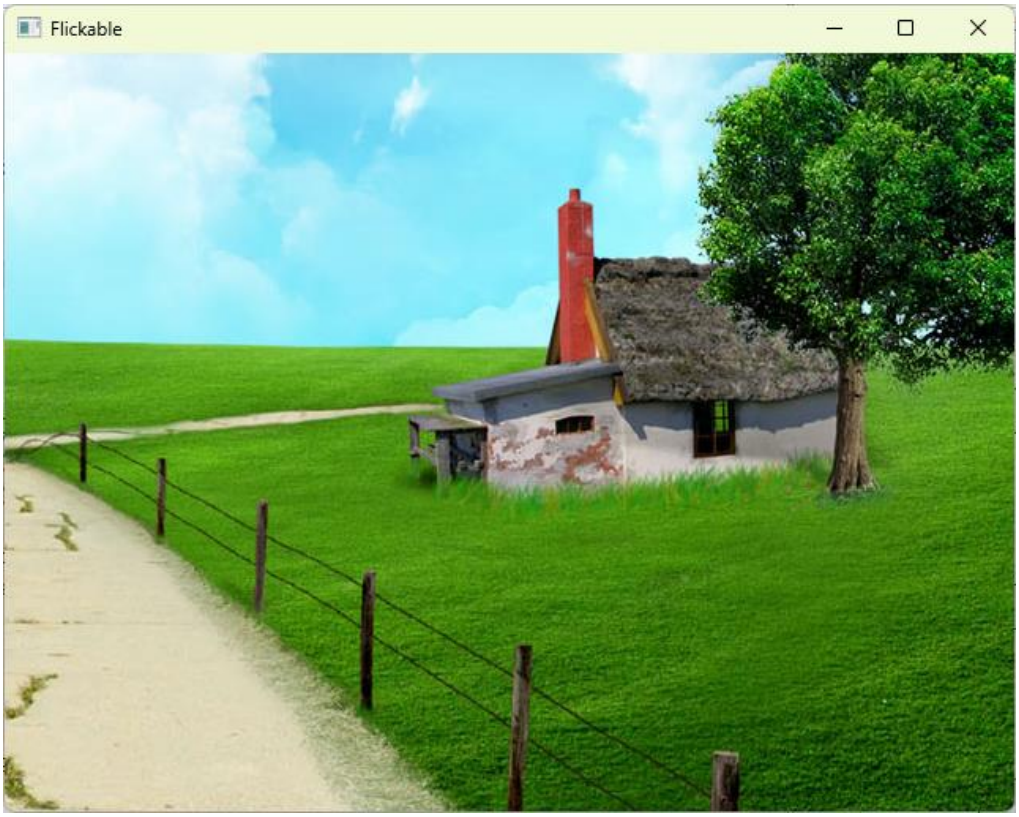
- Flickable

Flickable provides the ability to move types such as Rectangle and Item on the screen with Drag and Flicked events. The following example source code is the Flickable example source code.

```
import QtQuick
import QtQuick.Window
```

```
Window
{
    title: "Flickable"
    visible: true
    width: 640; height: 480

    Rectangle {
        width: 640
        height: 480

        Flickable {
            id: view
            anchors.fill: parent
            contentWidth: picture.width
            contentHeight: picture.height

            Image {
                 id: picture
                 source: "images/background.jpg"
            }
         }
    }
}
```

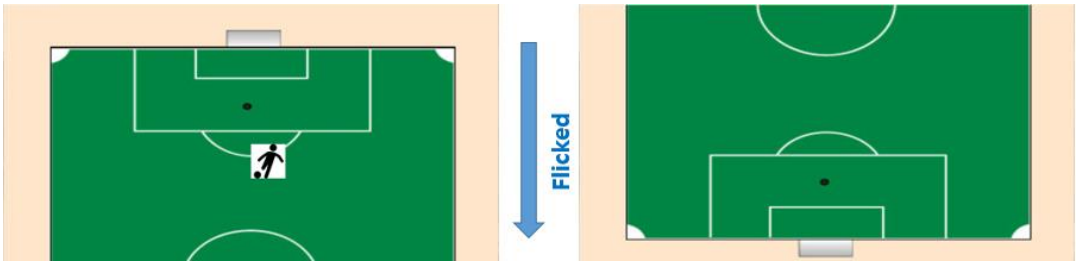
The figure below shows the example source code run screen. Let's try dragging the loaded image with the mouse on the run screen.



You can find the source code for this example in the 06_Flickable directory.

- Example of handling the Flicked image up and down event

In this example, we'll use the Flickable from the previous example to handle events when an image is flicked up, down, left and right. The image is an image of a soccer field. As you can see in the image below, when the Down Flicked event is fired, the image is dragged downward.




```
import QtQuick
import QtQuick.Window
```

```
Window {
    visible: true
    width: Screen.width
    height: Screen.height

    Flickable {
        width : Screen.width
        height : Screen.height
        contentWidth: Screen.width
        contentHeight: Screen.height * 2
        interactive: true // event on/off

        Image {
            id: ground
            anchors.fill: parent
            source : "images/ground.jpg"
            sourceSize.width: Screen.width
            sourceSize.height: Screen.height * 2
        }

        Image {
            id: player
            source : "images/player.png"
            x: Screen.width / 2
            y: Screen.height / 2
        }
    }
}
```

You can find the source code for this example in the 07_Flickable_Ground directory.

- Signal and Signal Handler

In QML, a Signal occurs in a type and a Signal Handler is associated with the Signal to provide functionality for handling the event that the Signal occurs.

For example, as shown in the example source code below, when a Signal named clicked is fired, a Signal Handler named onClicked performs the event.

```
import QtQuick
```

```
import QtQuick.Window

Window {
    visible: true; width: 360; height: 360
    Rectangle {
        id: rect
        width: 360; height: 360; color: "blue"

        MouseArea {
            anchors.fill: parent
            onClicked: {
                rect.color = Qt.rgba(Math.random(),
                                     Math.random(),
                                     Math.random(), 1);

                console.log("Clicked mouse at", mouseX, mouseY)
            }
        }
    }
}
```

The source code for this example can be found in the 08_SignalHandler_exam1 directory.

- Connections

The Connections type allows Signal Handlers to be handled within the Connections type.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 360; height: 360
    Rectangle {
        id: rect
        width: 360; height: 360; color: "blue"

        MouseArea {
            id: mouseArea
            anchors.fill: parent
        }

        Connections {
```

```
            target: mouseArea
            onClicked: {
                rect.color = Qt.rgba(Math.random(),
                                     Math.random(),
                                     Math.random(), 1);
            }
        }
    }
}
```

The source code for this example can be found in the 08_SignalHandler_exam2 directory.

- Add a custom signal

In QML, the syntax for adding a user-defined signal is as follows

```
signal <name>[([<type> <parameter name>[, ...]])]
```

In QML types, when a Signal is emitted (an event occurs), a specific function associated with the Signal can be executed as follows.

The following example is an example of a Signal added by the user. This example consists of two example source codes.

The first is SquareButton.qml in a directory called content. The second is called main.qml. The QML type defined by the user in SquareButton.qml is used in main.qml. Let's take a look at the SquareButton.qml example source code.

```
import QtQuick

Rectangle {
    id: root
    signal activated(real xPosition, real yPosition)
    signal deactivated
    width: 100; height: 100

    MouseArea {
        anchors.fill: parent
        onPressed: root.activated(mouseX, mouseY)
        onReleased: {
            root.deactivated()
        }
```

```
    }
}
```

Here is the source code for the main.qml example.

```
import QtQuick
import QtQuick.Window
import "content"

Window {
    visible: true; width: 360; height: 360
    SquareButton
    {
        width: 360; height: 360
        onActivated: console.log("Activated at " + xPosition + "," + yPosition)
        onDeactivated: console.log("Deactivated!")
    }
}
```

The source code for this example can be found in the 08_SignalHandler_exam3 directory.

- Connecting a Signal and a Method (or Function)

The following example source code is an example source code for connecting a Signal and a Method.

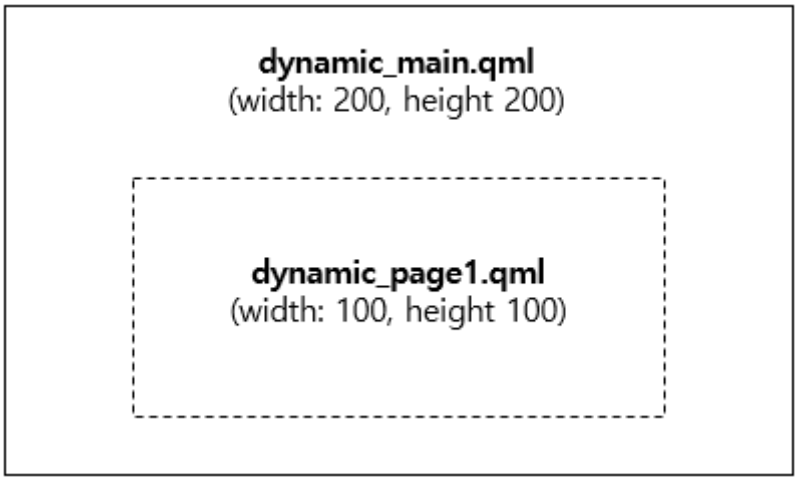
```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 360; height: 360

    id: relay
    signal messageReceived(string person, string notice)

    Component.onCompleted: {
        relay.messageReceived.connect(sendToPost)
        relay.messageReceived.connect(sendToTelegraph)
        relay.messageReceived.connect(sendToEmail)
    }

    function sendToPost(person, notice) {
        console.log("Sending to post: " + person + ", " + notice)
```

```
    }
    function sendToTelegraph(person, notice) {
        console.log("Sending to message: " + person + ", " + notice)
    }
    function sendToEmail(person, notice) {
        console.log("Sending to email: " + person + ", " + notice)
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            relay.messageReceived("Tom", "Happy Birthday")
        }
    }
}
```

In this example, when a Signal is raised, the Method registered in Component.onCompleted is called. The source code for this example can be found in the 08_SignalHandler_exam4 directory.

2.4. Implementing Dynamic UI with Loader type

In this section, we will look at the ability to dynamically call another QML in a specific area within a QML. To load another QML file inside a QML, we provide a Loader type.



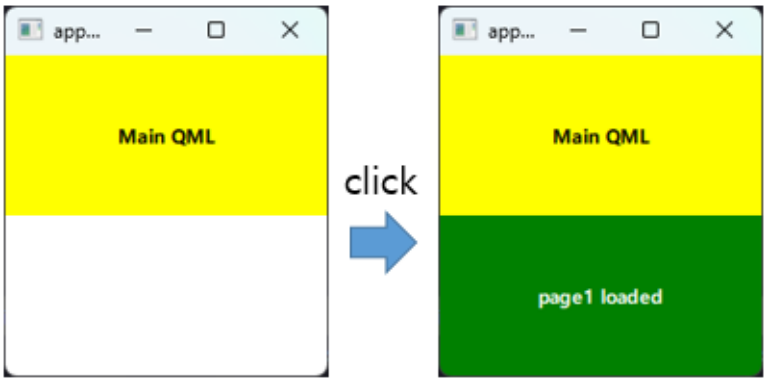
As shown in the figure above, the Loader type allows you to dynamically load a QML inside a QML. The following example loads the dynamic_page1.qml file on top of dynamic_main.qml. First, let's take a look at the dynamic_main.qml example source code.

```
import QtQuick
import QtQuick.Window

Window {
    width: 200; height: 200; visible: true
    Loader {
        id: pageLoader
        anchors.top: myRect.bottom
    }

    Rectangle {
        id: myRect
        width: 200; height: 100
        color: "yellow"
        Text {
            anchors.centerIn: parent
            text : "Main QML"; font.bold: true
```

```
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
                pageLoader.source = "dynamic_page1.qml"
            }
        }
    }
}
```

In the example source code above, we can load the page1.qml file from the area where the Loader type is declared. In the Loader type, we have specified the id as pageLoader. Then, when the mouse is clicked, the QML file to be loaded is specified using the source property whose id is pageLoader. Therefore, the page1.qml file is loaded when the mouse is clicked within the Rectangle type area with the id MyRect. The following is dynamic_page1.qml

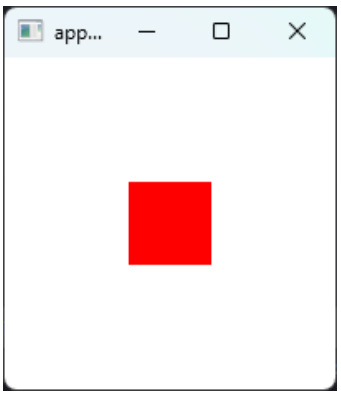
```
import QtQuick

Rectangle {
    width: 200; height: 100; color: "green"
    Text {
        anchors.centerIn: parent
        text: "page1 loaded"
        font.bold: true;
        color: "#FFFFFF"
    }
}
```



- Using the Component type to load within the Loader type area

The Component type can specify the type that is loaded in the Loader area. The following example source is an example using the Component type.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 200; height: 200

    Loader {
        anchors.centerIn: parent
        sourceComponent: rect
    }

    Component {
        id: rect
        Rectangle {
            width: 50
            height: 50
            color: "red"
        }
    }
}
```



- Handling events in the Loader type

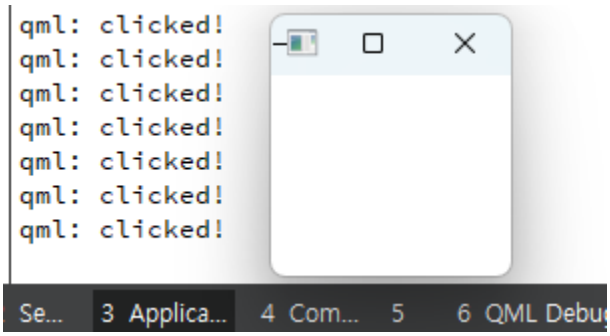
You can use the Connections type to handle events generated by the Loader type. The following example shows an example of connecting signals using Connections.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 100
    height: 100

    Loader {
       id: myLoader
       source: "application_myitem.qml"
    }

    Connections {
        target: myLoader.item
        function onMessage(msg) {
            console.log(msg)
        }
    }
}
```

In the example source code above, the Loader type loads the application_myitem.qml file. And in the Connection type, the property named onMessage is the signal registered in application_myitem.qml.

To use the signal named message in application_myitem.qml in the Connections type, change the first letter m to M before the message signal name and add the string on to the signal string. The following is the source code for the application_myitem.qml example.

```
import QtQuick

Rectangle {
   id: myItem
   signal message(string msg)
   width: 100; height: 100

   MouseArea {
       anchors.fill: parent
       onClicked: myItem.message("clicked!")
   }
}
```

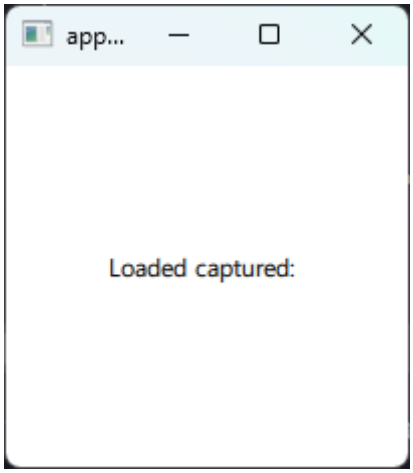


- Handling Key Events in the Loader Area

The focus property must be assigned a value of true to handle key events from QML files loaded by the Loader type. The following example shows the source code for the key_main.qml example.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 200
    height: 200

    Loader {
        id: loader
        focus: true
        source: "key_keyreader.qml"
        anchors.centerIn: parent
    }
}
```

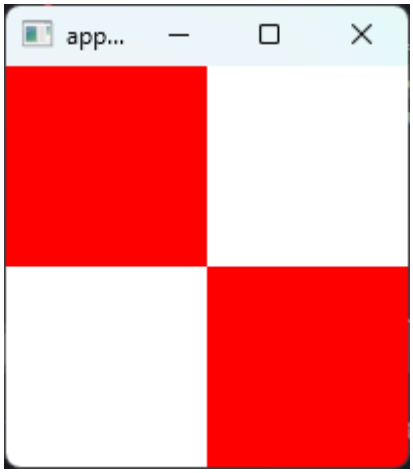
As shown in the example above, the Loader type loads the key_keyreader.qml source file. The following is the key_keyreader.qml example source code.



- Calling more than one Loader

This example calls using two Loader types.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 200
    height: 200

    Component {
        id: redSquare
        Rectangle {
           color: "red"; width: 100; height: 100
        }
    }

    Loader {
        sourceComponent: redSquare
    }

    Loader {
        sourceComponent: redSquare; x: 100; y: 100
    }
}
```



- status property of type Loader

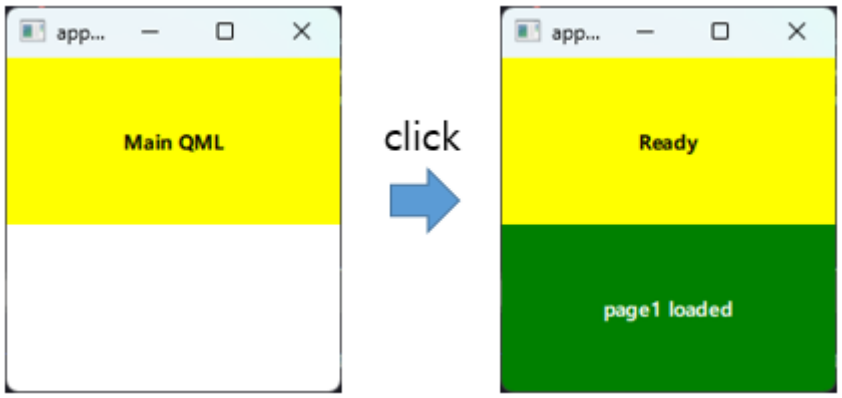
The status property provided by the Loader type provides the status of the Loader type.

Kind	Description
Null	If the QML source you call from the Loader area is disabled or doesn't exist
Ready	When the QML source is ready to load
Loading	When the QML source is loading modern
Error	When the QML source encountered an error while loading

The following is an example source code that uses state in the Loader type.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 200; height: 200

    Loader {
        id: pageLoader
        anchors.top: myRect.bottom
        onStatusChanged: {
            if (pageLoader.status == Loader.Ready)
                myText.text = "Ready"
        }
```

```
    }
    Rectangle {
        id: myRect
        width: 200; height: 100;
        color: "yellow"

        Text {
            id: myText
            anchors.centerIn: parent
            text : "Main QML"; font.bold: true
        }

        MouseArea {
            anchors.fill: parent
            onClicked: pageLoader.source = "dynamic_page1.qml"
        }
    }
}
```



In the image above, if you click the area labeled Main QML, the page1.qml file loads at the bottom and outputs the Ready string in the console window.

- Changing the value of a property in a QML file loaded with the Loader type

Let's see how to change the value of a property declared in a QML file loaded with the Loader type. This example consists of value_change.qml and ExComponent.qml. This example shows how to load ExComponent.qml as a Loader type from value_change.qml. First, let's see the example source code of the value_change.qml file.

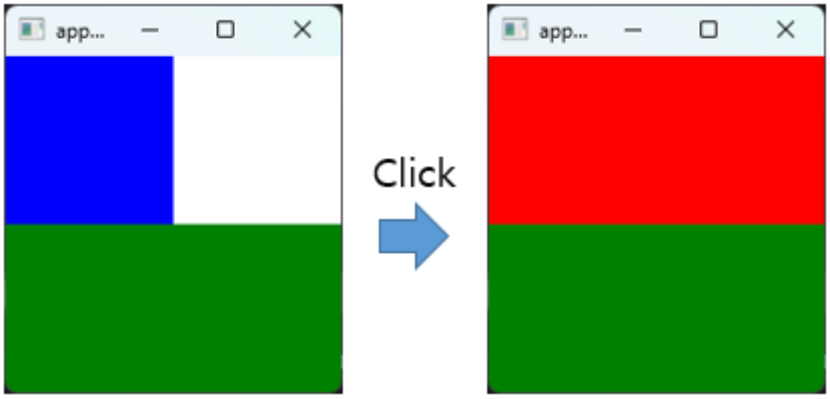
```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 200; height: 200

    Loader {
        id: squareLoader
        onLoaded: {
            console.log("Width : " + squareLoader.item.width);
        }
    }

    Component.onCompleted: {
        squareLoader.setSource("ExComponent.qml", { "color": "blue" });
    }

    Rectangle {
        anchors.top: squareLoader.bottom
        width: 200; height: 100
        color: "green"
        MouseArea {
            anchors.fill: parent
            onClicked: {
                squareLoader.setSource("ExComponent.qml", {"width": 200})
            }
        }
    }
}
```

Load the ExComponent.qml loaded from the Loader and the Rectangle at the bottom, as shown in the example source code above. The following is the ExComponent.qml source code.

```
import QtQuick

Rectangle {
    id: rect
    color: "red"
    width: 100
    height: 100
```

```
}
```

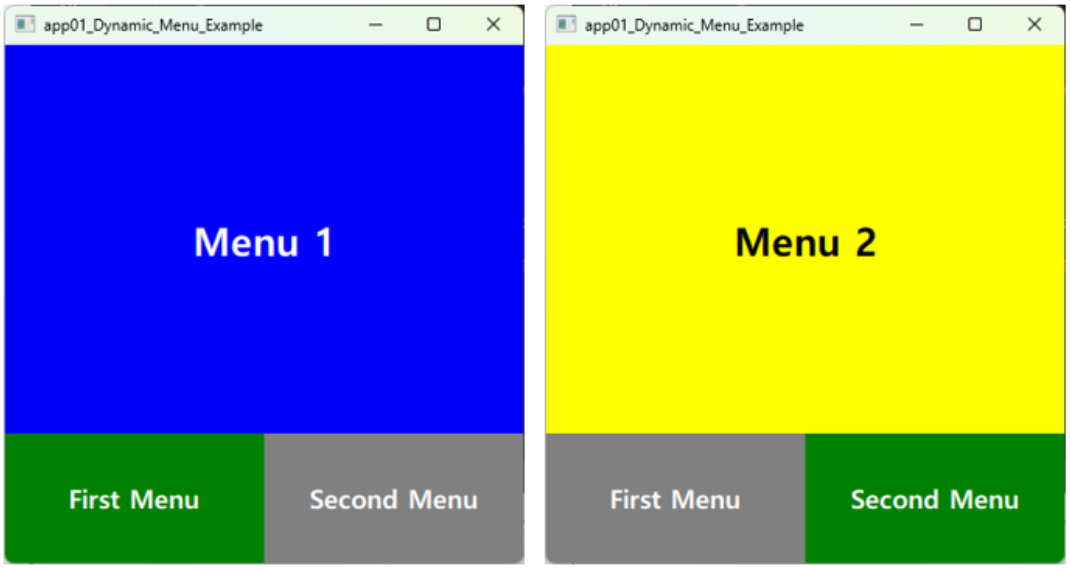
In value_change.qml, ExComponent.qml is loaded using the Loader, so two rectangles of type Rectangle are loaded as shown in the following figure.



As shown in the image above, clicking on the Rectangle with the color "green" below changes the value of the width property of the top Rectangle from 100 to 200, as shown on the right. It also changes the value of the Color to "red".

- Example of dynamic QML loading using the Loader type

This example shows how to change the QML at the top when clicking on the first menu at the bottom and the second menu area in the loaded QML. The following is the example run screen.



As shown in the image above, clicking the [Second Menu] on the bottom right will change the top menu to yellow and the bottom right to green, as shown in the image on the right. This example consists of three QML files as follows.

File name	Description
main.qml	Load the menu1.qml and menu2.qml files using the Loader type.
menu1.qml	White color Rectangle
menu2.qml	Yellow color Rectangle

As you can see in the table above, this example consists of three QML files. Let's start with the source code for the main.qml file.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 400; height: 400

    Rectangle {
        id: root
        width: 400; height: 400

        Loader {
            id: myLoader
            anchors.left: parent.left; anchors.right: parent.right
            anchors.top: parent.top; anchors.bottom: menu1Button.top
            onLoaded: { source: "menu1.qml" }
        }

        Rectangle {
            id: menu1Button
            anchors.left: parent.left; anchors.bottom: parent.bottom
            color: "gray"
            width: parent.width / 2
            height: 100

            Text {
                anchors.centerIn: parent;
                text: "First Menu";
                font.bold: true; font.pixelSize: 20; color: "white"
            }
```

```
            MouseArea {
                anchors.fill: parent
                onClicked: root.state = "menu1";
            }
        }

        Rectangle {
            id: menu2Button

            anchors.right: parent.right
            anchors.bottom: parent.bottom

            color: "gray"

            width: parent.width/2
            height: 100

            Text {
                anchors.centerIn: parent
                text: "Second Menu";
                font.bold: true; font.pixelSize: 20; color: "white"
            }

            MouseArea {
                anchors.fill: parent
                onClicked: root.state = "menu2";
            }
        }

        state: "menu1"

        states: [
            State {
                name: "menu1"
                PropertyChanges { target: menu1Button; color: "green"; }
                PropertyChanges { target: myLoader; source: "menu1.qml"; }
            },
            State {
                name: "menu2"
                PropertyChanges { target: menu2Button; color: "green"; }
                PropertyChanges { target: myLoader; source: "menu2.qml"; }
            }
```

```
        ]
    }
}
```

c The State type at the bottom is State Machine, which means that if the current Status is "menu1", the QML of the Rectangle at the top is loaded with the menu1.qml QML file and the Color is changed to "green".

And if the Status is "menu2", change the QML of the top Rectangle to the menu2.qml file and change the Color to "green". The following is the source code for the menu1.qml example.

```
import QtQuick

Rectangle {
    width: 400; height: 300; color: "blue"
    Text {
        anchors.centerIn: parent
        text: "Menu 1"
        font.bold: true
        font.pixelSize: 30
        color: "white"
    }
}
```

Next, write the menu1.qml example source code like below.

```
import QtQuick

Rectangle {
    width: 400; height: 300; color: "yellow"
    Text {
        anchors.centerIn: parent
        text: "Menu 2"
        font.bold: true
        font.pixelSize: 30
        color: "black"
    }
}
```

You can find the source code for this example in the 01_Dynamic_Menu_Example directory.

2.5. Canvas

Canvas provides functionality for drawing within a QML region, similar to the QPainter class used by the QWidget class. You can render drawings and images of elements such as lines, shapes, gradients, etc. within the QML area, and you can specify the size of the drawing area using the width and height properties.

```
import QtQuick

Canvas {
    id: mycanvas
    width: 100
    height: 200
}
```

When you use QPainter in a QWidget, you can use the onPaint property in Canvas, such as update( ) or repaint( ).

```
import QtQuick
Canvas {
    id: mycanvas; width: 100; height: 200
    onPaint: {
        ...
    }
}
```

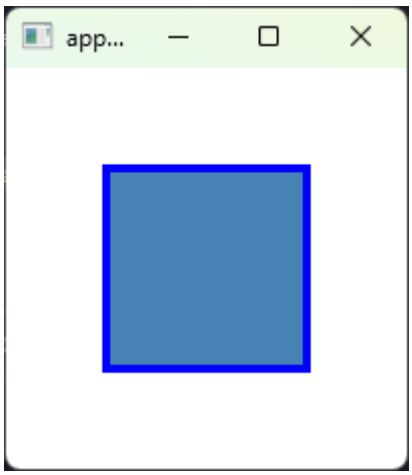
- Draw a simple rectangle inside a Canvas-type area

This example shows how to draw a rectangle in the Canvas area.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width:200; height: 200

    Canvas {
        id: root
```

```
        width: 200; height: 200

        onPaint:
        {
            var ctx = getContext("2d")

            ctx.lineWidth = 4
            ctx.strokeStyle = "blue"
            ctx.fillStyle = "steelblue"

            ctx.beginPath()
            ctx.moveTo(50,50)
            ctx.lineTo(150,50)
            ctx.lineTo(150,150)
            ctx.lineTo(50,150)
            ctx.closePath()

            ctx.fill()
            ctx.stroke()
        }
    }

}
```

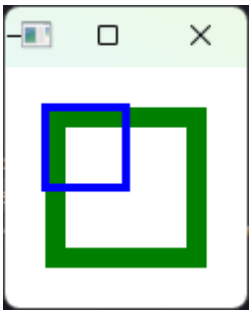
In the source code above, onPaint provides the same functionality as the QPainter class used by QWidget. Inside onPaint, we have specified the "2d" argument to the getContext( ) function. In Canvas, you can only select "2d". Therefore, you cannot specify "3d" in the Canvas area.



- Drawing Different Shapes of Rectangles

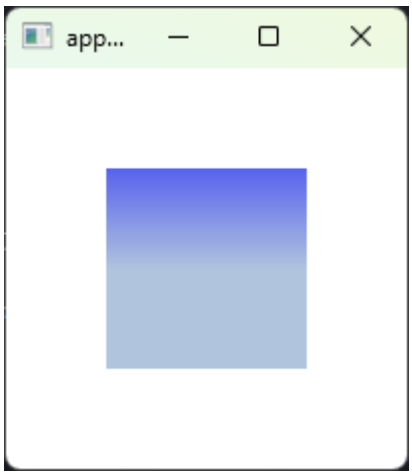
You can draw various rectangles in the Canvas area, as shown in the following example source code.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 120
    height: 120

    Canvas {
        id: root
        width: 120
        height: 120

        onPaint: {
            var ctx = getContext("2d")
            ctx.fillStyle = 'green'
            ctx.strokeStyle = "blue"
            ctx.lineWidth = 4

            ctx.fillRect(20, 20, 80, 80)
            ctx.clearRect(30,30, 60, 60)
            ctx.strokeRect(20,20, 40, 40)
        }
    }
}
```



- Using Gradients for Canvas Areas

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 200; height: 200

    Canvas {
        id: root
        width: 200; height: 200

        onPaint: {
            var ctx = getContext("2d")

            var gradient = ctx.createLinearGradient(100,0,100,200)
            gradient.addColorStop(0, "blue")
            gradient.addColorStop(0.5, "lightsteelblue")

            ctx.fillStyle = gradient
            ctx.fillRect(50,50,100,100)
        }
    }
}
```



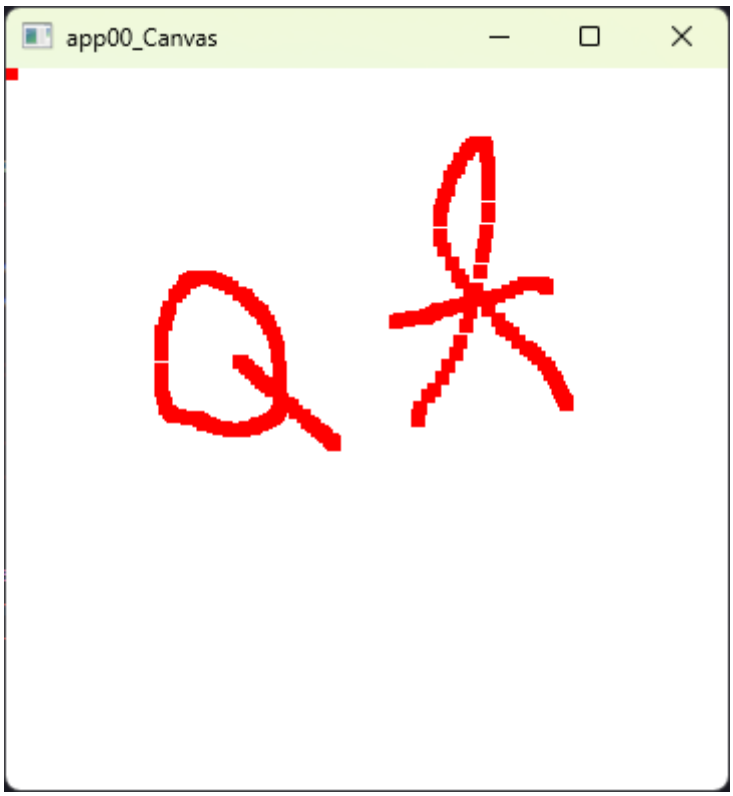
- Drawing in the Canvas area

In this example, we'll be drawing by holding down the mouse button and dragging in

the Canvas area.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 360; height: 360

    Rectangle {
        id: root
        width: 360
        height: 360

        Canvas {
            id: myCanvas
            anchors.fill: parent

            property int xpos
            property int ypos

            onPaint: {
                var ctx = getContext('2d')
                ctx.fillStyle = "red"
                ctx.fillRect(myCanvas.xpos-1, myCanvas.ypos-1, 7, 7)
            }

            MouseArea{
                anchors.fill: parent
                onPressed: {
                    myCanvas.xpos = mouseX
                    myCanvas.ypos = mouseY
                    myCanvas.requestPaint()
                }
                onMouseXChanged: {
                    myCanvas.xpos = mouseX
                    myCanvas.ypos = mouseY
                    myCanvas.requestPaint()
                }
                onMouseYChanged: {
                    myCanvas.xpos = mouseX
                    myCanvas.ypos = mouseY
                    myCanvas.requestPaint()
                }
```

```
            }
        }
    }
}
```


app00_Canvas

2.6. Graphics Effects

In Qt, you can use effects such as blending, masking, blurring, coloring, and so on.

● **Blend Effect**

Source

Foreground Source

Result

● **Colorize Effect**

Source

hue: 0.0
saturation: 0.5
lightness: -0.2

hue: 0.8
saturation: 1.0
lightness: 0

To use the Graphic Effects provided by Qt Quick in QML, we need to import them into QML as follows.

```
import Qt5Compat.GraphicalEffects
```

The QtGraphicalEffects module provides a variety of effects, as shown in the following table.

Kind	Type	Description
Blend	Blend	Blend processing by merging two images
Color	BrightnessContrast	Brightness and contrast
	ColorOverlay	Changing the color of a source item with an overlay color applied to it
	Colorize	Setting colors to the HSL color space

	Desaturate	Reducing the saturation of colors
	GammaAdjust	Adjust the brightness of the original source item
	HueSaturation	Adjusting the original item color for the HSL color space
	LevelAdjust	Adjusting color levels for the RGBA color space
Gradient	ConicalGradient	Seamlessly blend two or more colors. Colors start at a specified angle and display up to 360 degrees.
	LinearGradient	Seamless blending of two or more colors. Colors are displayed from a specified start to an end point
	RadialGradient	Seamlessly blend two or more colors. Colors start in the middle and go to the end of the border.
Distortion	Displace	Move pixels of the source item according to the Displacement map
DropShadow	DropShadow	Drop shadow effects on source images
	InnerShadow	Blur inside the source image
Blur	FastBlur	Apply a blur effect to one or more source items
	GaussianBlur	Apply a higher quality blur
	MaskedBlur	Controlling the blur intensity for each pixel using maskSource so that some parts of the source are blurrier than others
	RecursiveBlur	Using a recursive feedback loop to blur the source multiple times to soften the image
Motion Blur	DirectionBlur	Apply Blur to a specific direction
	RadialBlur	Blur the item's center point in a circle around it
	ZoomBlur	Apply Blur based on the center point

Glow	Glow	Used to create a colored and blurred effect on the original image and to highlight darker areas in the original image
	RectangularGlow	Use to create a colored and blurred effect on a rectangular area and highlight the dark side area
Mask	OpacityMask	Mask the source image with another item
	ThresholdMask	Apply a mask to the original item and apply the threshold area

- Blend Effect

Blend can apply an effect that blends the Source image with the Foreground Source image. To improve the performance of rendering, you can use Cache. To use Cache, you can set the cached property to true.

Source

Foreground Source

Result

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 200
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc
            source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
```

```
            visible: false
        }

        Image {
            id: butterfly
            source: "images/butterfly.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        Blend {
            anchors.fill: bgc
            source: bgc
            foregroundSource: butterfly
            mode: "average"
            cached: true
        }
    }
}
```

- BrightnessContrast

BrightnessContrast 는 Source 이미지의 Brightness 와 Contrast를 조정할 수 있다.

Source

brightness: 0.5
contrast: 0.5

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 200

    Item {
        width: parent.width
```

```
        height: parent.height

        Image {
            id: bgc
            source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        BrightnessContrast {
            anchors.fill: bgc
            source: bgc
            brightness: 0.5
            contrast: 0.5
        }
    }
}
```

- ColorOverlay

c ColorOverlay allows you to set a Colorize effect on the Source image.

Source

color: "#8000ff00"

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 300

    Image {
        id: bug
        source: "images/butterfly.png"
```

```
        sourceSize: Qt.size(parent.width, parent.height)
        smooth: true
        visible: false
    }

    ColorOverlay {
        anchors.fill: bug
        source: bug
        color: "#8000ff00"
    }
}
```

- Colorize

The Colorize effect can set a color in the HSL color space of the Source image item.

Source

hue: 0.0
saturation: 0.5
lightness: -0.2

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 200
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc
            source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
```

```
            visible: false
        }

        Colorize {
            anchors.fill: bgc
            source: bgc
            hue: 0.0
            saturation: 0.5
            lightness: -0.2
        }
    }
}
```

- Desaturate

Provides the ability to desaturate the RGB values of the source image. This can be done, for example, by displaying it in black and white.

Source

desaturation: 0.5

desaturation: 1.0

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 300
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bug
            source: "images/bug.jpg"
```

```
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        Desaturate {
            anchors.fill: bug
            source: bug
            desaturation: 1.0
        }
    }
}
```

- GammaAdjust

GammaAdjust can apply an effect to each pixel according to a predefined curve, such as a power-law expression, which means it uses gamma values to apply the effect.

Source Gamma: 2.0

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 200
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc; source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }
```

```
        GammaAdjust {
            anchors.fill: bgc
            source: bgc
            gamma: 2.0
        }
    }
}
```

- HueSaturation

HueSaturation is similar to the Colorize effect. The difference with Colorize is that you can specify the hue and saturation properties.

Source

hue: 0.3
saturation: 0
lightness: 0

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 200
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc; source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true; visible: false
        }
        HueSaturation {
            anchors.fill: bgc; source: bgc
            hue: 0.3
```

```
            saturation: 0
            lightness: 0
        }
    }
}
```

- LevelAdjust

LevelAdjust can apply effects to separate colors for each Color channel in the Source image.

Source

minimumOutput: "#00ffffff"
maximumOutput: "#ff000000"

minimumInput: "#00000070"
maximumInput: "#ffffff"
minimumOutput: "#000000"
maximumOutput: "#ffffff"

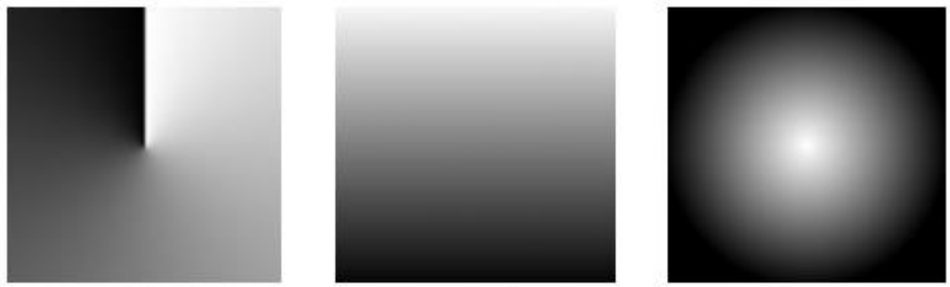
```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 300
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: butterfly; source: "images/butterfly.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true; visible: false
        }

        LevelAdjust {
            anchors.fill: butterfly
```

```
            source: butterfly
            minimumInput: "#00000070"
            maximumInput: "#ffffff"
            minimumOutput: "#000000"
            maximumOutput: "#ffffff"
        }
    }
}
```

- Gradient

Gradient provides an effect that displays a seamless blend of two or more colors. ConicalGradient displays a color starting at a specified angle and going through 360 degrees.

LinearGradient displays a color from a specified point to a specified point. RadialGradient starts in the middle and goes to the end of the border.




```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 600; height: 200

    Row
    {
        ConicalGradient {
            width: 200; height: 200
            angle: 0.0
            gradient: Gradient {
                GradientStop { position: 0.0; color: "white" }
```

```
                GradientStop { position: 1.0; color: "black" }
            }
        }

        LinearGradient {
            width: 200; height: 200
            start: Qt.point(0, 0)
            end: Qt.point(0, 300)
            gradient: Gradient {
                GradientStop { position: 0.0; color: "white" }
                GradientStop { position: 1.0; color: "black" }
            }
        }

        RadialGradient {
            width: 200; height: 200
            gradient: Gradient {
                GradientStop { position: 0.0; color: "white" }
                GradientStop { position: 0.5; color: "black" }
            }
        }
    }
}
```

- Displace

Displace provides the ability to move pixels in a source image item according to a displacement map.

Source

Displacement Source

Result

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
```

```
    visible: true
    width: 300; height: 200
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc
            source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        Rectangle {
            id: displacement
            color: Qt.rgba(0.5, 0.5, 1.0, 1.0)
            anchors.fill: parent
            visible: false

            Image {
                anchors.centerIn: parent
                source: "images/glass_normal.png"
                sourceSize: Qt.size(parent.width/2, parent.height/2)
                smooth: true
            }
        }

        Displace {
            anchors.fill: bgc
            source: bgc
            displacementSource: displacement
            displacement: 0.1
        }
    }
}
```

- DropShadow

DropShadow provides functionality for specifying a drop shadow effect on a source image. The radius property can specify the softness of the shadow.

The larger the value of this property, the more blurry and jagged the edges of the shadow can appear. You can also use the color property to specify the color of the shadow.

Source

color: "#aa000000"
radius: 8.0
samples: 16
horizontalOffset: 0
verticalOffset: 20
spread: 0

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 300
    Item {
        width: parent.width
        height: parent.height

        Image {
            width:300; height: 300
            id: ball
            source: "images/ball.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        DropShadow {
            anchors.fill: ball
            radius: 8.0
            samples: 16
            horizontalOffset: 0
            verticalOffset: 20
            spread:0
            source: ball
```

```
            color: "#aa000000"
        }
    }
}
```

- FastBlur

c FastBlur can apply a blur effect on the Source image. It provides a lower blur effect than GaussianBlur, but renders faster than GaussianBlur.

Source

radius: 32

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true
    width: 300; height: 200
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc; source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        FastBlur {
            anchors.fill: bgc
            source: bgc
            radius: 32
        }
    }
```

```
}
```

- GaussianBlur

c Higher quality than FastBlur, but slower to render than FastBlur.

Source

deviation: 4
radius: 8
samples: 16

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true
    width: 300; height: 200
    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc
            source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        GaussianBlur {
            anchors.fill: bgc
            source: bgc
            deviation: 4
            radius: 8
            samples: 16
        }
    }
```

```
}
```

- DirectionBlur

DirectionBlur 는 특정 방향으로 Blur Effect를 적용할 수 있다.

Source

angle: 90
length: 32
samples: 24

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true
    width: 300
    height: 200

    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc; source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true
            visible: false
        }

        DirectionalBlur {
            anchors.fill: bgc; source: bgc
            angle: 90
            length: 32
            samples: 24
        }
    }
```

```
}
```

- RadialBlur

You can apply a blur effect, such as rotating to the center of the Source image.

Source

samples: 24
angle: 30

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 200

    Item {
        width: parent.width
        height: parent.height

        Image {
            id: bgc; source: "images/background.png"
            sourceSize: Qt.size(parent.width, parent.height)
            smooth: true; visible: false
        }

        RadialBlur {
            anchors.fill: bgc
            source: bgc
            samples: 24
            angle: 30
        }
    }
}
```

- Glow

Source Used to create a colored and blurred effect on an image and highlight dark areas.

Source

Result

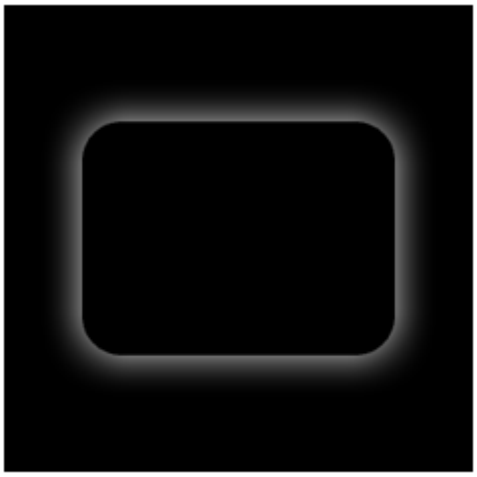
```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 300
    Rectangle {
        anchors.fill: parent
        color: "black"
    }

    Image {
        id: ball
        source: "images/ball.png"
        sourceSize: Qt.size(parent.width, parent.height)
        smooth: true
        visible: false
    }
    Glow {
        anchors.fill: ball
        radius: 30
        samples: 16
        color: "green"
        source: ball
    }
```

```
}
```

- RectangularGlow

Colorize the Rectangle region and create a Blur. We then provide a Glow Effect by positioning the Rectangle item.

Source



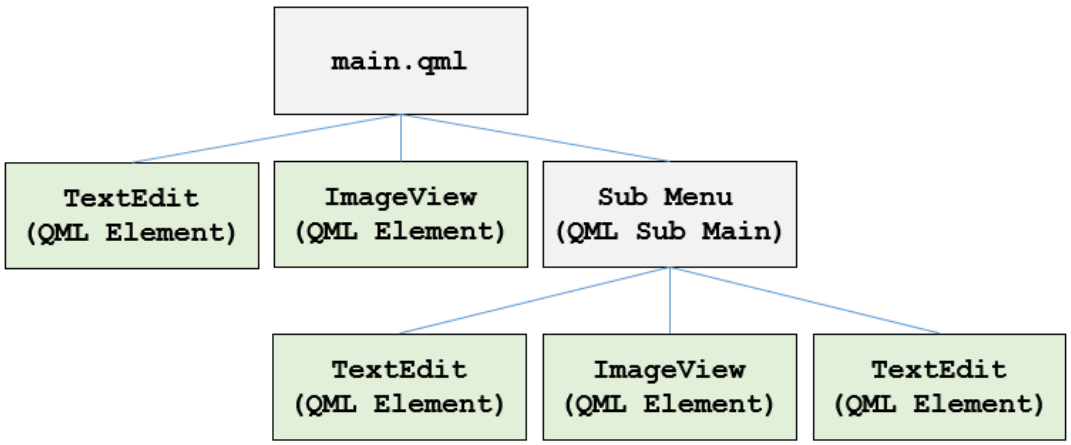
Result

```
import QtQuick
import QtQuick.Window
import Qt5Compat.GraphicalEffects

Window {
    visible: true; width: 300; height: 300

    Rectangle {
        id: background; anchors.fill: parent; color: "black"
    }

    RectangularGlow {
        id: effect
        anchors.fill: rect
        glowRadius: 10
        spread: 0.2
        color: "white"
        cornerRadius: rect.radius + glowRadius
    }
```

```
    Rectangle {
        id: rect
        color: "black"
        anchors.centerIn: parent
        width: Math.round(parent.width / 1.5)
        height: Math.round(parent.height / 2)
        radius: 25
    }
}
```

2.7. Module programming

Just as you can create and use frequently used classes as libraries in C++, you can also modularize frequently used custom types in QML so that they can be reused as libraries, as shown in the figure below.



- Custom types

You can write user-defined types in separate QML files to be imported into other QMLs. For example, suppose you have a main.qml and a LineEdit.qml QML source code file. To call the LineEdit.qml QML from main.qml, the name "LineEdit", which is the filename without the extension of the file, is used as the name of the module that can be called. The following example shows an example of calling the LineEdit.qml file from main.qml.

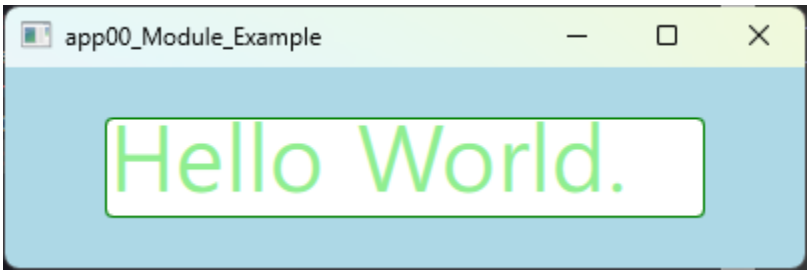
The example QML source code below is Main.qml.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 400; height: 100
    color: "lightblue"

    LineEdit
    {
        anchors.horizontalCenter: parent.horizontalCenter
```

```
        anchors.verticalCenter: parent.verticalCenter
        width: 300; height: 50
    }
}
```

The following example is LineEdit.qml.

```
import QtQuick

Rectangle
{
  border.color: "green";
  color: "white"; radius: 4; smooth: true

  TextInput {
    anchors.fill: parent
    anchors.margins: 2
    text: "Hello World."
    color: focus ? "black" : "lightgreen"
    font.pixelSize: parent.height - 4
  }
}
```



- Custom Property

In addition to the properties used by the built-in QML types, you can define your own properties for use within a QML type.

```
Syntax: property <type> <name>[: <value>]
```

Here's an example of using a custom Property

```
property string message: "qt-dev.com"
property int num: 123
```

```
property real numfloat: 123.456
property bool boolcondi: true
property url address: "http://www.qt-dev.com"
```

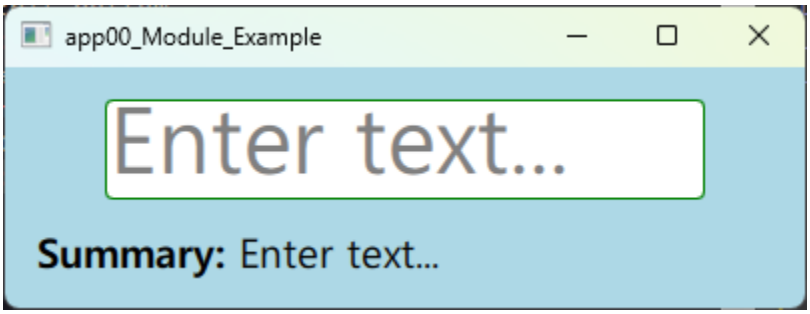
c The following example source is an example using custom properties. And the source code for the example below is new_main.qml.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 400; height: 120
    color: "lightblue"

    NewLineEdit {
        id: lineEdit
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: parent.top
        anchors.topMargin: 16
        width: 300; height: 50
    }

    Text {
        anchors.top: lineEdit.bottom
        anchors.topMargin: 12
        anchors.left: parent.left
        anchors.leftMargin: 16
        font.pixelSize: 20
        text: "<b>Summary:</b> " + lineEdit.text
    }
}
```

아래 예제 소스코드는 NewLineEdit.qml 이다.

```
import QtQuick

Rectangle {
    border.color: "green"
    color: "white"
    radius: 4; smooth: true

    TextInput {
        id: textInput
```

```
        anchors.fill: parent
        anchors.margins: 2
        text: "Enter text..."
        color: focus ? "black" : "gray"
        font.pixelSize: parent.height - 4
    }

    property string text: textInput.text
}
```



- Custom Signal

The signal keyword is provided to enable the use of custom signals and slots in QML, just like custom signals and slots in Qt/C++. The following example shows the syntax structure for using signal.

```
Signal syntax: signal <name>[(<type> <value>, ...)]
```

✓ Example for defining a custom signal

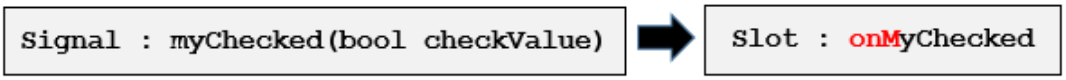
```
MouseArea {
    anchors.fill: checkImage
    onClicked: if (parent.state === "checked") {
                   parent.state = "unchecked";
                   parent.myChecked(false);
               } else {
                   parent.state = "checked";
                   parent.myChecked(true);
               }
}
```

```
...
signal myChecked(bool checkValue)
```

✓ Example of a slot associated with a signal

```
...
onMyChecked: checkValue ? parent.color = "red":  parent.color = "lightblue "
...
```

As shown in the example above, where you define a Signal, you define the Signal name and the Arguments (function arguments) that you want to pass to the Signal. If there are no arguments, you can just specify the signal name. Slot then capitalizes the first letter of the signal name. The signal name must be prefixed with "on".



This time, we will implement a CheckBox using custom signals. The QML file is an example where the checked_main.qml file calls NewCheckBox.qml, which is located under the "content" directory.

Let's start by creating the checked_main.qml file.

```
import QtQuick
import QtQuick.Window
import "content"

Window {
    id: root
    visible: true; width: 250; height: 100
    color: "lightblue"

    NewCheckBox {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter

        onMyChecked: checkValue ? root.color = "red"
                                :  root.color = "lightblue"
    }
}
```

This time, create a directory called content and inside it, write the NewCheckBox.qml file

like below.

```
import QtQuick

Item {
    width: textItem.width + checkImage.width

    Image {
        id: checkImage
        anchors.left: parent.left
        anchors.verticalCenter: parent.verticalCenter
    }

    Text {
        id: textItem
        anchors.left: checkImage.right
        anchors.leftMargin: 4
        anchors.verticalCenter: checkImage.verticalCenter
        text: "Option"
        font.pixelSize: checkImage.height - 4
    }

    states: [
        State {
            name: "checked"
            PropertyChanges {
                target: checkImage;
                source: "../images/checked.svg"
            }
        },
        State {
            name: "unchecked"
            PropertyChanges {
                target: checkImage;
                source: "../images/unchecked.svg"
            }
        }
    ]

    state: "unchecked"

    MouseArea {
        anchors.fill: checkImage
```

```
        onClicked: if (parent.state === "checked") {
                       parent.state = "unchecked";
                       parent.myChecked(false);
                   } else {
                       parent.state = "checked";
                       parent.myChecked(true);
                   }
    }

    signal myChecked(bool checkValue)
}
```

- Alias

Alias provides a pseudonymization feature to replace the module name used by QML. The following example shows an example of using Alias.

```
import QtQuick
import QtQuick.Window
import "content" as MyContent

Window {
     visible: true; width: 250; height: 100; color: "lightblue"

     MyContent.NewCheckBox {
          anchors.horizontalCenter: parent.horizontalCenter
          anchors.verticalCenter: parent.verticalCenter
     }
}
```

Alias is also available in the official QML module provided by Qt Quick, as shown in the following example.

```
import QtQuick as MyQt
```

```
import QtQuick.Window

MyQt.Window {
     visible: true; width: 250; height: 100; color: "lightblue"

    MyQt.Rectangle {
           width: 250; height: 100; color: "lightblue"
           MyQt.Text {
               anchors.centerIn: parent
               text: "Hello Qt!"; font.pixelSize: 32
           }
    }
}
```

2.8. How to use JavaScript in QML

QML has a syntax like JSON, XML, or HTML. As such, it does not provide a structured syntax like JavaScript or C++. For example, QML does not provide syntax such as If statements, functions, variable substitution, etc.

Therefore, JavaScript can be used with QML to enable structured programming in QML. You can also use JavaScript files as modules within QML. In this chapter, you will learn how to use JavaScript with QML.

- Property Binding

Property Binding can assign a value to a Property named color using ternary operator syntax, as shown in the following example.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    title: qsTr("Hello World")

    id: colorbutton;
    width: 200;
    height: 80;
    color: mousearea.pressed ? "steelblue" : "lightsteelblue"
    MouseArea {
        id: mousearea; anchors.fill: parent
    }
}
```

The following example QML source code uses a JavaScript function. You can call a JavaScript function from a function using the binding provided by QML.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
```

```
    id: colorbutton
    width: 200; height: 80; color: "red"

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

    Component.onCompleted:
    {
        color = Qt.binding( function() {
            return mousearea.pressed ? "steelblue" : "lightsteelblue"
        } );
    }
}
```

- Calling a JavaScript Method using Signal Handler

In QML, we used MouseArea to handle the event when the mouse button is clicked. We called a predefined slot such as onClicked provided by MouseArea and used it like below.

```
...
Rectangle {
    id: relay

    signal messageReceived(string person, string notice)

    Component.onCompleted: {
        relay.messageReceived.connect(sendToPost)
        relay.messageReceived.connect(sendToTelegraph)
        relay.messageReceived.connect(sendToEmail)
        relay.messageReceived("Tom", "Happy Birthday")
    }

    function sendToPost(person, notice) {
        console.log("Sending to post: " + person + ", " + notice)
    }
    function sendToTelegraph(person, notice) {
        console.log("Sending to telegraph: " + person + ", " + notice)
    }
    function sendToEmail(person, notice) {
        console.log("Sending to email: " + person + ", " + notice)
```

```
    }
}
...
```

- Using JavaScript functions in QML types

To use a JavaScript function in a QML type, you can call the JavaScript function as shown in the example below.

```
import QtQuick
import QtQuick.Window

Window {
    visible: true
    title: qsTr("Hello World")

    id: colorButton;

    width: 200;
    height: calculateHeight();

    color: mousearea.pressed ? "steelblue" : "lightsteelblue"

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

    function calculateHeight()
    {
        return colorButton.width / 2;
    }
}
```

The following example shows example source code using the arguments of a JavaScript function.

```
import QtQuick
import QtQuick.Window

Window {
   width: 200; height: 200; visible: true
```

```
    MouseArea {
        anchors.fill: parent
        onClicked: label.moveTo(mouseX, mouseY)
    }

    Text {
        id: label
        function moveTo(newX, newY) {
            label.x = newX;
            label.y = newY;
        }

        text: "Move me!"
    }
}
```

다음 예제는 MouseArea 에서 JavaScript 함수를 호출하고 함수 결과를 리턴 하는 예제이다.

```
...
Item {
    function factorial(a) {
        a = parseInt(a);
        if (a <= 0)
            return 1;
        else
            return a * factorial(a - 1);
    }
    MouseArea {
        anchors.fill: parent
        onClicked: console.log(factorial(10))
    }
}
```

- **Importing and Using JavaScript Files**

In QML, you can call JavaScript files (files with the .js extension) where JavaScript functions are defined.

```
import QtQuick
import QtQuick.Window
```

```
import "script.js" as MyScript

Window {
    visible: true
    title: qsTr("Hello World")

    id: item; width: 200; height: 200

    MouseArea {
        id: mouseArea; anchors.fill: parent
    }
    Component.onCompleted: {
        mouseArea.clicked.connect(MyScript.jsFunction)
    }

}
```

As shown in the example source code above, when a mouse button click event occurs, we can call a function defined in a JavaScript file called script.js, as shown below.

```
// script.js

function jsFunction()
{
    console.log("Called JavaScript function!")
}
```

Libraries can also be defined and used in JavaScript using the Pragma keyword, just like Library. The following example uses the pragma keyword.

```
import QtQuick
import QtQuick.Window
import "script.js" as MyLib

Window {
    visible: true
    id: item; width: 500; height: 200

    Text {
        width: parent.width
        height: parent.height
        property int input: 17

        text: "The Number of " + input
```

```
            + " is: " + MyLib.factorial(input)
    }
}
```

The following example source code is the script.js source code.

```
.pragma library

var factorialCount = 0;
function factorial(a)
{
    a = parseInt(a);

    // factorial recursion
    if (a > 0)
        return a * factorial(a - 1);

    // shared state
    factorialCount += 1;

    // recursion base-case.
    return 1;
}

function factorialCallCount()
{
    return factorialCount;
}
```

In QML, you can use Qt.include( ) to call JavaScript files, as shown in the example below.

```
import QtQuick
import QtQuick.Window
import "script.js" as MyScript

Window {
    visible: true
    width: 100; height: 100
    MouseArea {
        anchors.fill: parent
        onClicked: {
            MyScript.showCalculations(10)
            console.log("Call from QML:", MyScript.factorial(10))
        }
```

```
    }
}
```

```
// script.js

Qt.include("factorial.js")

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

```
// factorial.js

function factorial(a)
{
    a = parseInt(a);

    if (a <= 0)
        return 1;

    else
        return a * factorial(a - 1);
}
```

To call the above example JavaScript file, facrotyal.js, from script.js, we can use the Qt.include( ) function to call the JavaScript file.

- **Example of using JavaScript in QML**

This example is for calling a JavaScript function from QML.

If you enter two values as shown in the figure above and click the [Calculation] button, the result of summing the two values is shown as shown in the figure above. Create the Main.qml file like below.

```
import QtQuick
import QtQuick.Window
import QtQuick.Controls
import QtQuick.Layouts

import "js/jslib.js" as JUtils

Window {
    id: root; visible: true; width: 300; height: 300
    title: qsTr("JavaScript Example")

    property string val: "";

    ColumnLayout {
        anchors.fill: parent; anchors.margins: 9
        RowLayout {
            Layout.fillWidth: true
            TextField {
                id: input1;
                Layout.fillWidth: true;
```

```
            focus: true
        }
        Label {
            text: " + "
        }
        TextField {
            id: input2;
            Layout.fillWidth: true;
            focus: true
        }
        Button {
            text: " Calculation "
            onClicked: {
                root.val = input1.text.trim() + "," + input2.text.trim();
                root.jsCall(val);
            }
        }
    }
    Item {
        Layout.fillWidth: true;  Layout.fillHeight: true
        Rectangle {
            anchors.fill: parent
            color: '#333'; opacity: 0.2; radius: 5
            border.color: Qt.darker(color)

        }

        ScrollView {
            id: scrollView
            anchors.fill: parent; anchors.margins: 9
            ListView {
                id: resultView
                model: ListModel {
                    id: outputModel
                }
                delegate: ColumnLayout {
                    width: ListView.view.width
                    Label {
                        Layout.fillWidth: true
                        color: 'green'
                        text: "> " + model.expression
                    }
```

```
                        Label {
                            Layout.fillWidth: true
                            color: 'blue'
                            text: "= " + model.result
                        }
                        Rectangle {
                            height: 1
                            Layout.fillWidth: true
                            color: '#333'
                            opacity: 0.2
                        }
                    }
                }
            }
        }
    }

    function jsCall(exp) {

        var data = JUtils.addCall(exp);
        console.log(data);

        outputModel.insert(0, data)
    }
}
```

다음 예제는 위의 QML에서 호출한 JavaScript 파일의 소스코드 이다.

```
.pragma library

function addCall(msg)
{
    var str = msg.toString();
    var res = str.split(",");

    var src1 = parseInt(res[0]);
    var src2 = parseInt(res[1]);
    var result = src1 + src2;

    var data = {
        expression : result
    }
```

```
    data.expression = src1 + ' + ' + src2;
    data.result = result;

    return data;
}
```

For the above example source code, refer to the 00_JavaScript_Example directory.

2.9. Dialog

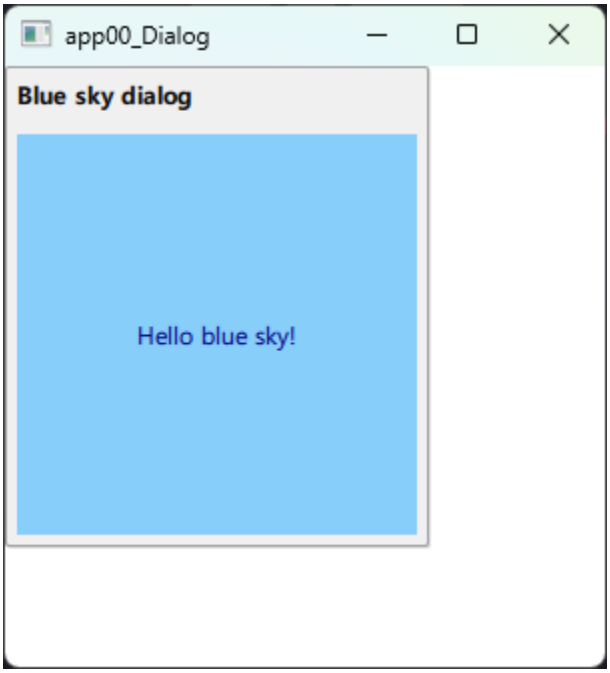
To use dialogs in QML, you need to import Dialogs as follows.

```
import QtQuick.Dialogs
```

- **Dialog**

Qt Quick provides the Dialog type to use dialogs in QML. The following example uses Dialog.

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

Window {
    width: 300; height: 300; visible: true

    Button {
        text: "Dialog loading"
        anchors.centerIn: parent
        onClicked: { myDial.visible = true }
    }

    Dialog {
        id: myDial; visible: false
        title: "Blue sky dialog"

        contentItem: Rectangle {
            color: "lightskyblue"
            implicitWidth: 200; implicitHeight: 200
            Text {
                text: "Hello blue sky!"; color: "navy"
                anchors.centerIn: parent
            }
        }
    }
}
```

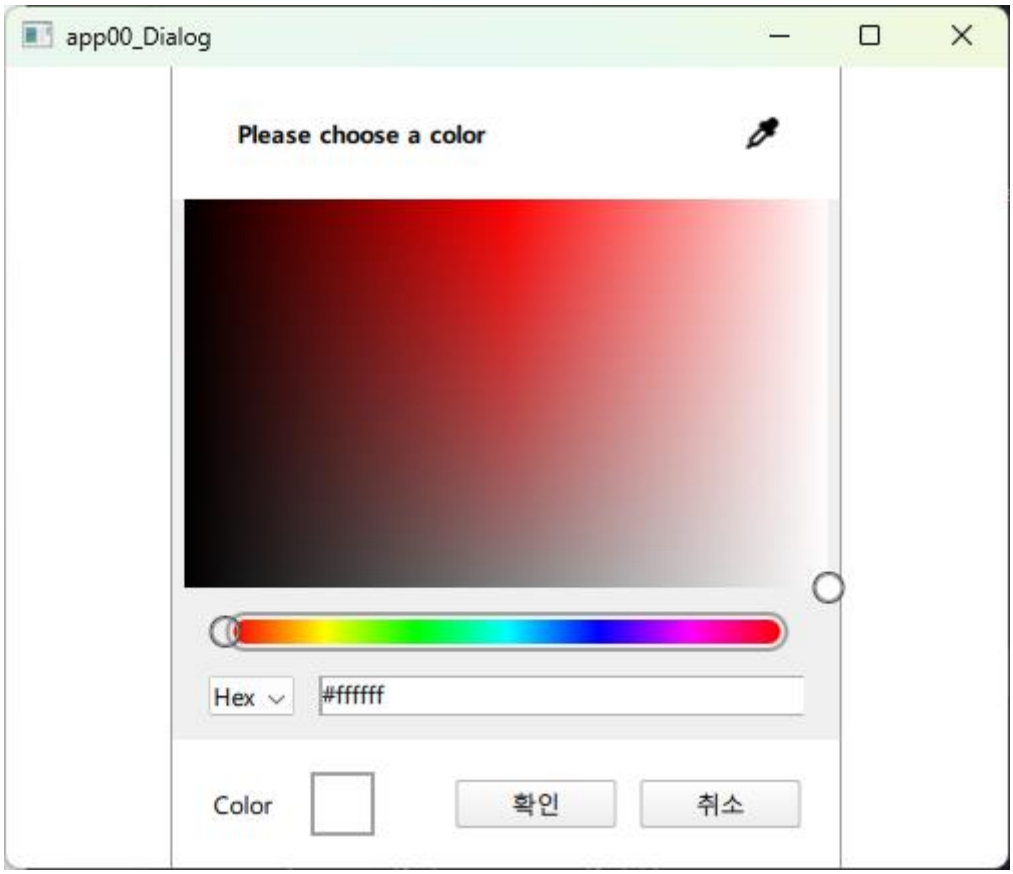


- **ColorDialog**

The ColorDialog type provides a dialog that allows the user to select a color.

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

Window {
    width: 500; height: 400; visible: true
    Button {
        text: "ColorDialog loading";
        anchors.centerIn: parent
        onClicked: {
            colorDialog.visible = true
        }
    }

    ColorDialog {
        id: colorDialog
        title: "Please choose a color"
        onAccepted: {
            console.log("You chose: " + colorDialog.selectedColor)
        }
```

```
        onRejected: {
            console.log("Canceled")
        }

        Component.onCompleted: visible = false
    }
}
```



- **FileDialog**

The FileDialog type provides a dialog that allows the user to select a file.

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

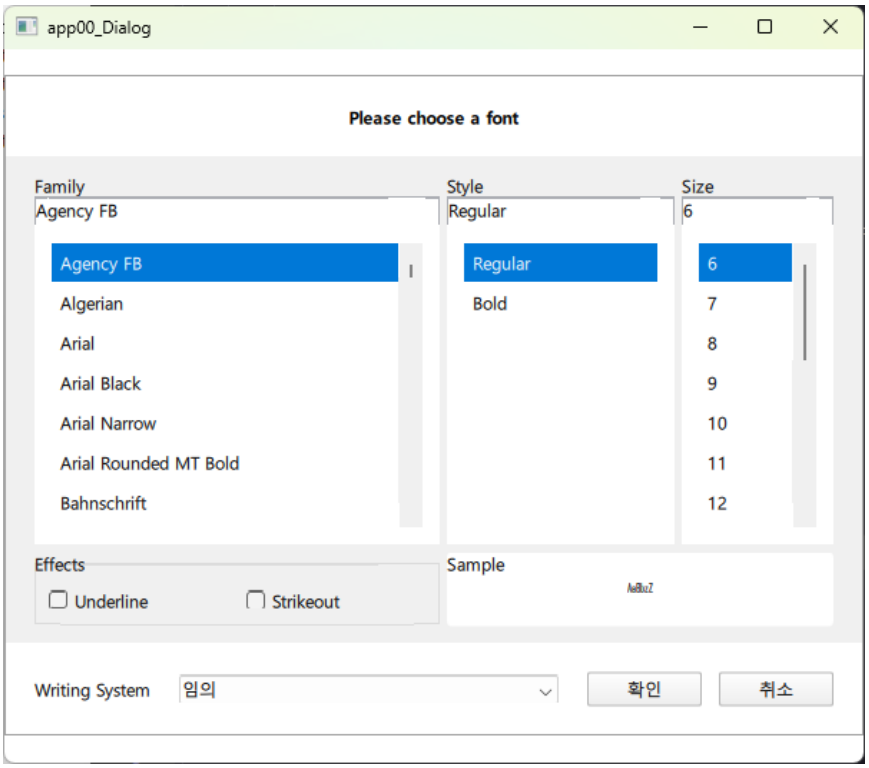
```
Window {
    width: 300; height: 300; visible: true

    Button {
        text: "FileDialog loading"
        anchors.centerIn: parent
        onClicked: {
            fileDialog.visible = true
        }
    }

    FileDialog
    {
        id: fileDialog
        title: "Please choose a file"
        onAccepted: {
            console.log("You chose: " + fileDialog.selectedFile)
        }
        onRejected: {
            console.log("Canceled")
        }

        Component.onCompleted: visible = false
    }
}
```

- **FontDialog**

The FontDialog type provides a dialog that allows the user to select a font.

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

Window {
    width: 600; height: 500; visible: true
    Button {
        text: "FontDialog loading";
        anchors.centerIn: parent
        onClicked: {
            fontDialog.visible = true
        }
```

```
    }

    FontDialog {
        id: fontDialog
        title: "Please choose a font"

        onAccepted: {
            console.log("You chose: " + fontDialog.selectedFont)
        }
        onRejected: {
            console.log("Canceled")
        }
        Component.onCompleted: visible = true
    }
}
```



- **MessageDialog**

The FontDialog type provides a dialog that allows the user to select a font.

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

Window {
    width: 500; height: 400; visible: true

    Button {
        text: "MessageDialog loading";
        anchors.centerIn: parent
        onClicked: {
            dialog.visible = true
        }
    }

    MessageDialog {
        id: dialog
        text: qsTr("The document has been modified.")
        informativeText: qsTr("Do you want to save your changes?")
        buttons: MessageDialog.Ok | MessageDialog.Cancel

        onButtonClicked: function (button, role) {
            switch (button) {
            case MessageDialog.Ok:
                document.save()
                break;
            }
        }
    }
}
```

2.10. Layout

In order to use the layout in QML, it must be imported into the QML file as follows

```
import QtQuick.Layouts
```

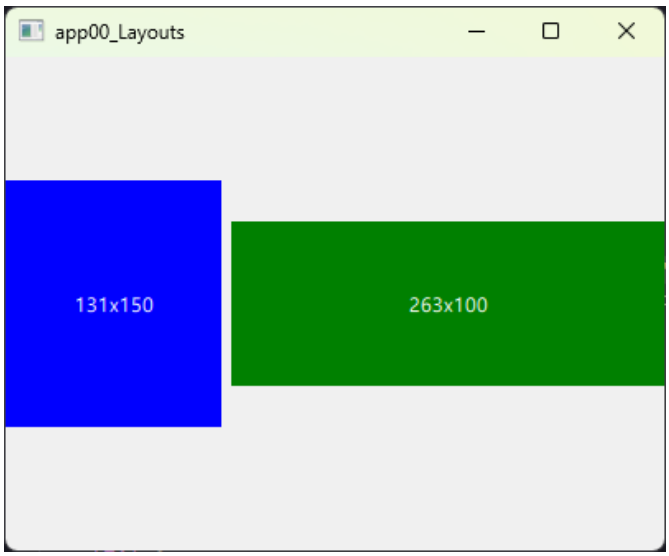
Qt offers the following types of layouts

Layout	Description
RowLayout	Positioning types horizontally on a single row
ColumnLayout	Positioning types in a vertical orientation
GridLayout	Place types like cells
Layout	Available by specifying one of the following: GridLayout, RowLayout, or ColumnLayout.

- RowLayout

```
import QtQuick
import QtQuick.Layouts

Window {
    width: 400; height: 300; visible: true

    RowLayout {
        id: layout
        anchors.fill: parent
        spacing: 6
        Rectangle {
            color: 'blue'
            Layout.fillWidth: true
            Layout.minimumWidth: 50
            Layout.preferredWidth: 100
            Layout.maximumWidth: 300
            Layout.minimumHeight: 150
            Text {
                anchors.centerIn: parent; color: "white"
                text: parent.width + 'x' + parent.height
            }
```

```
        }
        Rectangle {
            color: 'green'
            Layout.fillWidth: true
            Layout.minimumWidth: 100
            Layout.preferredWidth: 200
            Layout.preferredHeight: 100
            Text {
                anchors.centerIn: parent; color: "white"
                text: parent.width + 'x' + parent.height
            }
        }
    }
}
```



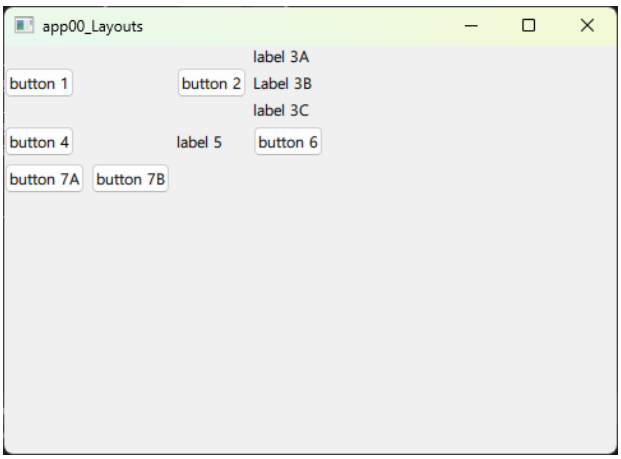
- ColumnLayout

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

Window {
    width: 400; height: 300; visible: true

    ColumnLayout{
        spacing: 2
```

```
        Rectangle {
            Layout.alignment: Qt.AlignCenter
            color: "red"
            Layout.preferredWidth: 40
            Layout.preferredHeight: 40
        }

        Rectangle {
            Layout.alignment: Qt.AlignRight
            color: "green"
            Layout.preferredWidth: 40
            Layout.preferredHeight: 70
        }

        Rectangle {
            Layout.alignment: Qt.AlignBottom
            Layout.fillHeight: true
            color: "blue"
            Layout.preferredWidth: 70
            Layout.preferredHeight: 40
        }
    }
}
```

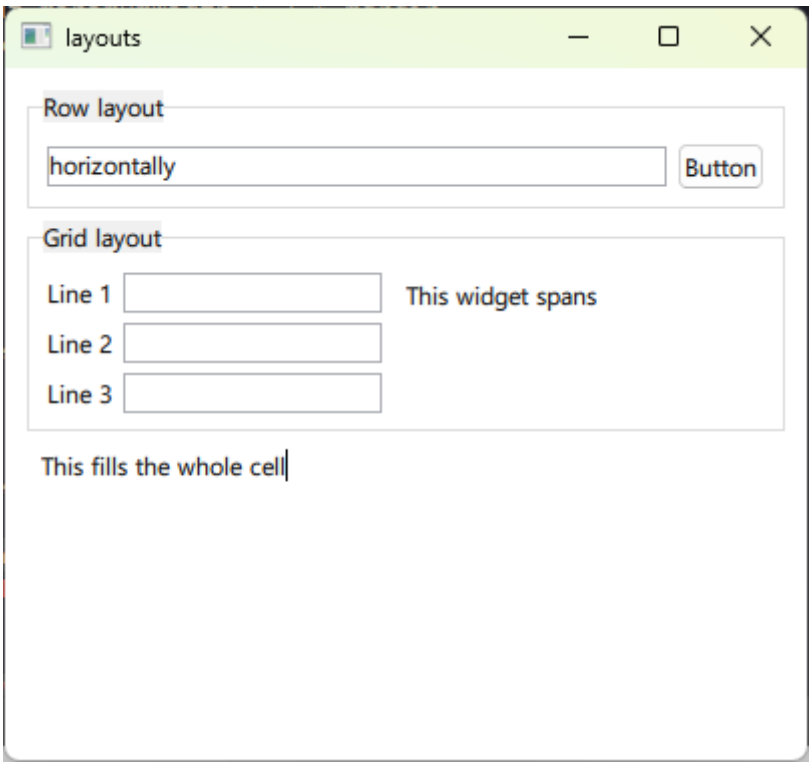
- GridLayout

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

Window {
    id: myWindow; width: 480; height: 320; visible: true

    GridLayout{
        columns: 3
        Button { text: "button 1"}
        Button { text: "button 2"}
        ColumnLayout{
            Label { text: "label 3A"}
            Label { text: "Label 3B"}
            Label { text: "label 3C"}
        }
        Button { text: "button 4"}
        Label{ text: "label 5"}
        Button { text: "button 6"}
        RowLayout{
            Button { text: "button 7A"}
            Button { text: "button 7B"}
        }
    }
}
```



- Layout example

In this example, we'll use the Layout provided by Qt Quick to implement a widget that places types as shown in the image below.



In this example, we've used ColumnLayout, RowLayout, GridLayout, and Layout to place types in a nested Layout structure. Here is the source code for the example

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

Window
{
    width: 400; height: 300
    visible: true
    title: "layouts"
    property int margin: 11
    minimumWidth: mainLayout.Layout.minimumWidth + 2 * margin
    minimumHeight: mainLayout.Layout.minimumHeight + 2 * margin

    ColumnLayout {
        id: mainLayout
```

```
        anchors.fill: parent
        anchors.margins: margin
        GroupBox {
            id: rowBoxr
            title: "Row layout"
            Layout.fillWidth: true

            RowLayout {
                id: rowLayout
                anchors.fill: parent
                TextField {
                    text: "horizontally"
                    Layout.fillWidth: true
                }
                Button {
                    text: "Button"
                }
            }
        }

        GroupBox {
            id: gridBox
            title: "Grid layout"
            Layout.fillWidth: true

            GridLayout {
                id: gridLayout
                rows: 3
                flow: GridLayout.TopToBottom
                anchors.fill: parent

                Label { text: "Line 1" }
                Label { text: "Line 2" }
                Label { text: "Line 3" }

                TextField { }
                TextField { }
                TextField { }

                TextArea {
                    text: "This widget spans";
                    Layout.rowSpan: 3
```

```
                    Layout.fillHeight: true
                    Layout.fillWidth: true
                }
            }
        }
        TextArea {
            id: t3
            text: "This fills the whole cell"
            Layout.minimumHeight: 30
            Layout.fillHeight: true
            Layout.fillWidth: true
        }
    }
}
```

2.11. Type Positioning

이번 장에서는 여러 개의 타입 중에서 특정 위치의 타입의 정보를 얻기 위한 방법에 대해서 살펴보자.

예를 들어 QML에서 100개 이상의 Rectangle 타입이 있다고 가정해 보자. 만약 100개 이상의 타입 중에서 특정 조건에 부합하는 타입의 위치 정보를 얻기 위해서 QML 에서 Positioner를 제공한다.

따라서 이번 장에서는 Positioner를 이용해 특정 타입의 위치 정보를 알아내는 방법에 대해서 살펴보도록 하자.

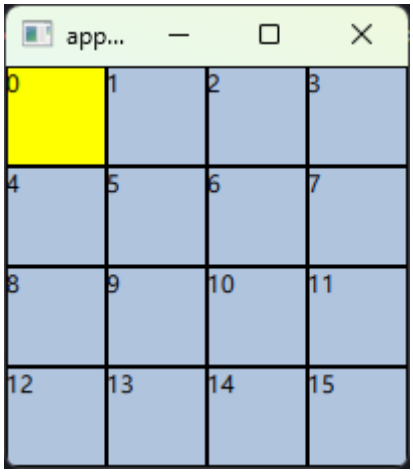
- **Positioner**

A Positioner item can know the position of a specific type in a child type such as Column, Row, or Grid. The index property tells you how many types a particular type is, and the isFirstItem property tells you if it is the first type.

c If it is the first type, the isFirstItem property returns true; otherwise, it returns false. Here is an example using the Positioner

```
import QtQuick
import QtQuick.Window

Window {
    visible: true; width: 200; height: 200

    Grid {
        Repeater {
            model: 16

            Rectangle {
                id: rect
                width: 50; height: 50
                border.width: 1
                color: Positioner.isFirstItem ? "yellow" : "lightsteelblue"

                Text { text: rect.Positioner.index }
            }
```

```
        }
    }
}
```



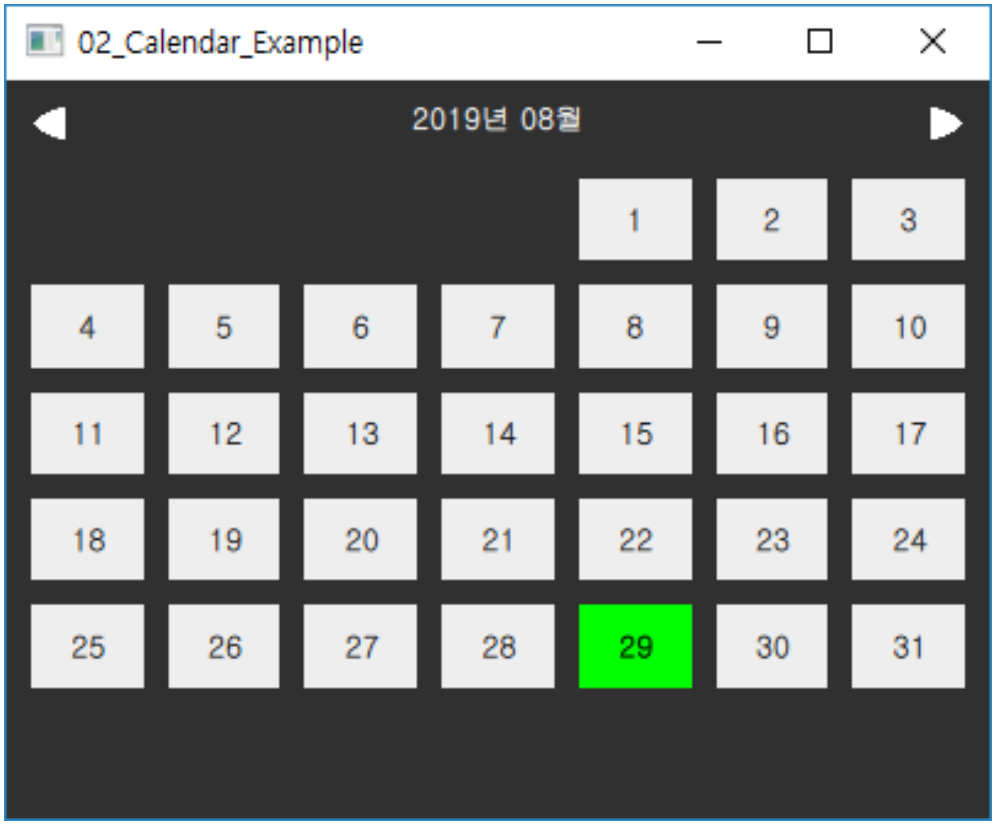
- **Calendar example**

In this example, we'll implement a calendar using Grid and Repeater. To implement the calendar, we'll use a class that provides a library for calculating dates in JavaScript.

```
property date today: new Date()
property date showDate: new Date()

property int daysInMonth:
new Date(showDate.getFullYear(), showDate.getMonth() + 1, 0).getDate()

property int firstDay:
new Date(showDate.getFullYear(), showDate.getMonth(), 1).getDay()
```

daysInMonth returns the number of days in the current month. firstDay outputs the num value of the first day of the week for the year, month, and day.

The getDay( ) function returns one of the num values (0 through 6) of the day of the week. For example, if it is Tuesday, it returns 2. Here is the example running with the calendar function complete.




```
import QtQuick
import QtQuick.Window

Window {
    id: calendar; width: 400; height: 300; visible: true; color: "#303030"

    property date today: new Date()
    property date showDate: new Date()
    property int daysInMonth:
       new Date(showDate.getFullYear(), showDate.getMonth() + 1, 0).getDate()

    property int firstDay:
       new Date(showDate.getFullYear(), showDate.getMonth(), 1).getDay()

    Item {
        id: title
        anchors.top: parent.top
        anchors.topMargin: 10
        width: parent.width
```

```
        height: childrenRect.height

        Image {
            source: "images/left.png"
            anchors.left: parent.left
            anchors.leftMargin: 10

            MouseArea {
                anchors.fill: parent
                onClicked: {
                    var tYear = showDate.getFullYear()
                    var tMonth = showDate.getMonth()
                    showDate = new Date(tYear, tMonth, 0)
                }
            }
        }

        Text {
            color: "white"
            text: Qt.formatDateTime(showDate, "MM, yyyy")
            font.bold: true
            anchors.horizontalCenter: parent.horizontalCenter
        }

        Image {
            source: "images/right.png"
            anchors.right: parent.right
            anchors.rightMargin: 10

            MouseArea {
                anchors.fill: parent
                onClicked: {
                    var tYear = showDate.getFullYear()
                    var tMonth = showDate.getMonth()
                    showDate = new Date(tYear, tMonth + 1, 1)
                }
            }
        }
    }

    function isToday(index) {
        if (today.getFullYear() != showDate.getFullYear())
```

```
            return false;
        if (today.getMonth() != showDate.getMonth())
            return false;

        return (index === today.getDate() - 1)
    }

    Item {
        id: dateLabels
        anchors.top: title.bottom
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.margins: 10

        height: calendar.height - title.height - 20 - title.anchors.topMargin
        property int rows: 6

        Item {
            id: dateGrid
            width: parent.width
            anchors.top: parent.top
            anchors.topMargin: 5
            anchors.bottom: parent.bottom

            Grid {
                columns: 7
                rows: dateLabels.rows
                spacing: 10

                Repeater {
                    model: firstDay + daysInMonth

                    Rectangle {
                        color: {
                            if (index < firstDay)
                                calendar.color;
                            else
                                isToday(index - firstDay) ? "#00ff00":"#eeeeee";
                        }
                        width: (calendar.width - 20 - 60)/7
                        height: (dateGrid.height -
                                    (dateLabels.rows - 1)*10)/dateLabels.rows
```

```
                        Text {
                            anchors.centerIn: parent
                            text: index + 1 - firstDay
                            opacity: (index < firstDay) ? 0 : 1
                            font.bold: isToday(index - firstDay)
                        }
                    }
                }
            }
        }
    }
}
```

You can find the source code for this example in the 01_Calendar_Example directory.

2.12. Qt Quick Controls

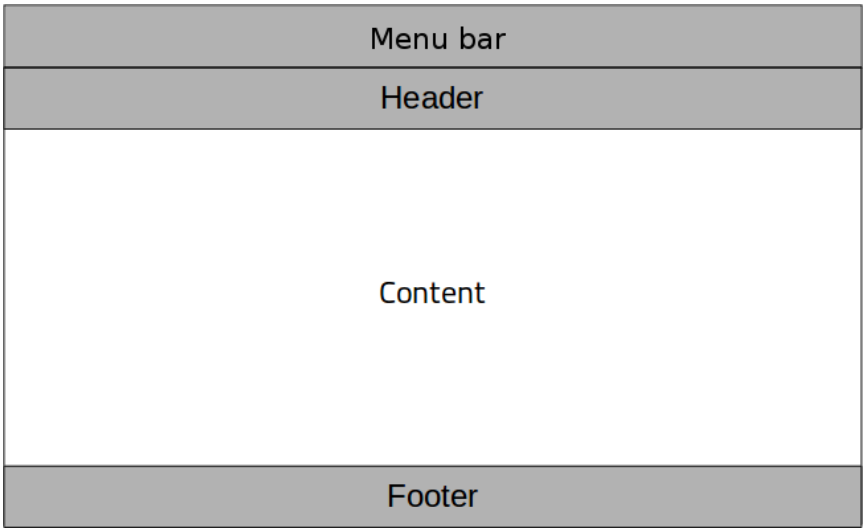
Qt Quick Controls provides basic UI controls for implementing GUIs in QML. For example, it provides types such as Button, CheckBox, Tab, Combo, SpinBox, etc. that are required for GUI implementation.

Qt Quick Controls was released in Qt 5.7, so Qt Quick Controls can be used with Qt 5.7 or later. And using Qt 5.X.X is somewhat different than using Qt 6.5.X. We will focus on Qt 6.5.X and later. If you want to refer to Qt Quick Controls in Qt 5.15.X, you can refer to the 2.0 and earlier versions of this documentation.

Also, the controls described here are the most commonly used control types. For more controls, you can refer to the Qt Assistant or the official help.

- ApplicationWindow

This type is provided to use the basic window GUI, such as the MenuBar, Action, StatusBar, ToolBar, etc. of the main window (top-level element).




```
import QtQuick.Controls

ApplicationWindow {
```

```
        visible: true

        menuBar: MenuBar {
            // ...
        }

        header: ToolBar {
            // ...
        }

        footer: RowLayout{
            // ...
        }

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

- Navigation 과 Views

ScrollView, SplitView, TableView, TreeView 등과 같은 Navigation 과 View 기능의 타입을 제공한다.

QML타입	설명
ScrollView	View that provides scrolling within a QML type
SplitView	Provide dragging functionality between QML types with Splitter
StackView	Features used to organize UIs with multiple screens like Stack
TableView	Items that provide TABLE-like list UI functionality
TreeView	Items that provide tree-like UI functionality, such as directory structures.

- **Controls**

It provides types such as Button, ComboBox, GroupBox, ProgressBar, CheckBox, Label, SpinBox, etc.

For example source code related to Controls, see the 00_Controls directory.

- **Menus**

Provides the Menu, MenuItem, and MenuSeparator QML types.

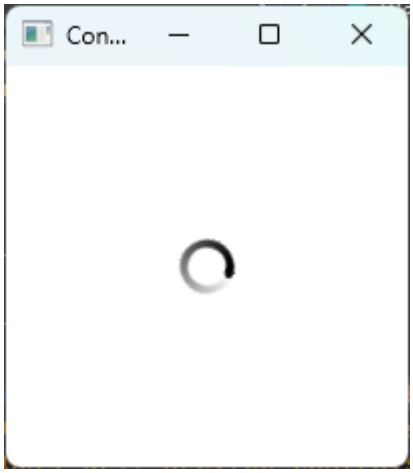
Type	Description
Menu	Popup menus, Context menus
MenuItem	Items to add to a menu or menubar
MenuSeparator	Provide UI functionality to differentiate between menu items

- BusyIndicator

Similar to the ProgressBar. It can be used to show uncertain progress. It can also be used to show background activity.

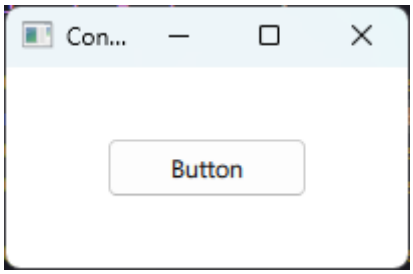
```
import QtQuick
import QtQuick.Window
import QtQuick.Controls
```

```
Window {
    width: 300; height: 300; visible: true
    title: qsTr("Controls")

    BusyIndicator {
        anchors.centerIn: parent
        running: true
    }
}
```



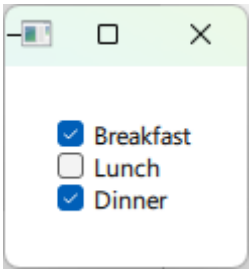
- Button

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

Window {
    width: 200; height: 200; visible: true
    title: qsTr("Controls")

    Button {
        width: 100; height: 30
        anchors.centerIn: parent
        text: "Button"
    }
}
```



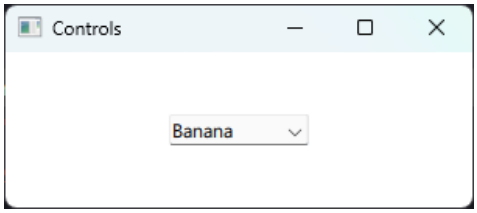
- CheckBox

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

Window {
    width: 100; height: 100; visible: true
    title: qsTr("Controls")

    Column {
        anchors.centerIn: parent

        CheckBox {
            text: qsTr("Breakfast")
            checked: true
        }
        CheckBox {
            text: qsTr("Lunch")
        }
        CheckBox {
            text: qsTr("Dinner")
            checked: true
        }
    }
}
```



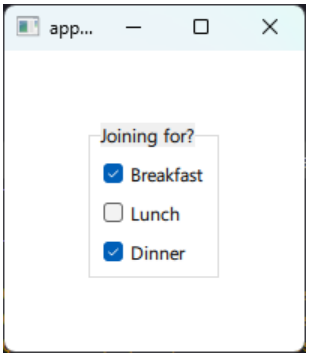
- ComboBox

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

Window {
    width: 300; height: 100; visible: true;
    title: qsTr("Controls")

    ComboBox {
        anchors.centerIn: parent

        editable: true
        model: ListModel {
            id: model
            ListElement { text: "Banana" }
            ListElement { text: "Apple"}
            ListElement { text: "Coconut"}
        }
        onAccepted: {
            // Add only when no item with the same name exists
            if (find(currentText) === -1) {
                model.append({text: editText})
            }
        }
    }
}
```



- GroupBox

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

Window {
    width: 200; height: 200
    visible: true

    GroupBox {
        anchors.centerIn: parent
        title: "Joining for?"

        Column {
            spacing: 10
            CheckBox { text: "Breakfast"; checked: true }
            CheckBox { text: "Lunch"; checked: false    }
            CheckBox { text: "Dinner"; checked: true    }
        }
    }
}
```



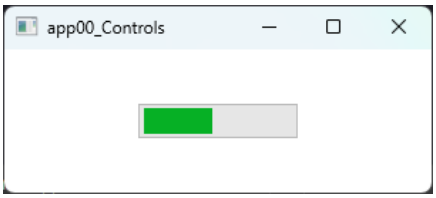
- Label

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

Window {
    width: 300
    height: 100
    visible: true

    Label {
        anchors.centerIn: parent
        text: "Hello world"
        font.pixelSize: 22
        font.italic: true
        color: "steelblue"
    }
}
```

- ProgressBar

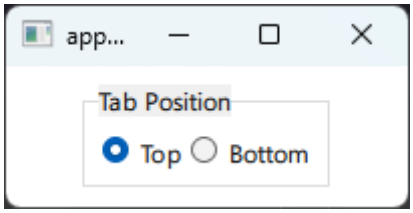
```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

Window {
    width: 300; height: 300; visible: true
    Column {
        anchors.centerIn: parent
        spacing: 10
        ProgressBar {
```

```
            value: 0.5
        }
    }
}
```



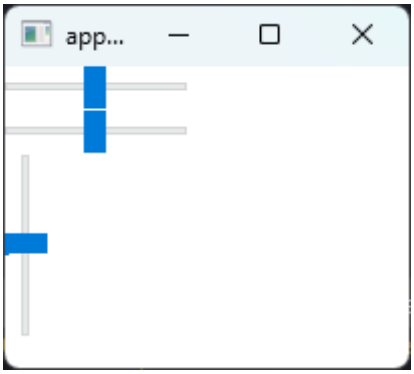
- RadioButton

```
import QtQuick
import QtQuick.Window
import QtQuick.Controls
import QtQuick.Layouts

Window {
    width: 200; height: 70; visible: true
    GroupBox {
        title: "Tab Position"
        anchors.centerIn: parent
        RowLayout {
            RadioButton {
                text: "Top"
                checked: true
            }
            RadioButton {
                text: "Bottom"
            }
        }
    }
}
```



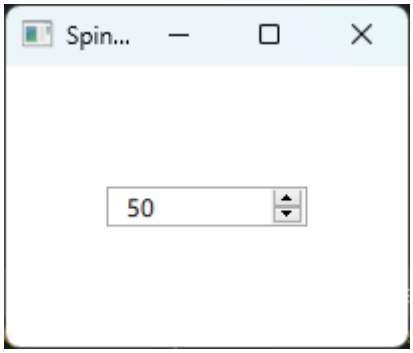
- Slider

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

Window {
    width: 400; height: 260; visible: true
    Column{
        Label { id: label }
        Slider{
            id: slider1; to: 100; from: 0
            value: 50; onValueChanged: label.text = "slider1: " + parseInt(value)
        }
        Slider{
            id: slider2; to: 100; from: 0
            value: 50; stepSize: 10
            onValueChanged: label.text = "slider2: " + parseInt(value)
        }
        Slider{
            id: slider3; to: 100; from: 0
            value: 50; stepSize: 25
            orientation: Qt.Vertical
            onValueChanged: label.text = "slider3: " + parseInt(value)
        }
    }
}
```



- SpinBox

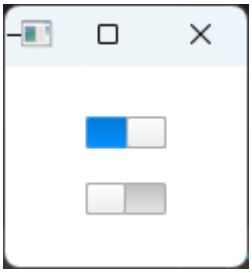
```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

Window {
    title: qsTr("SpinBox")
    width: 200; height: 140; visible: true

    Column{
        anchors.centerIn: parent
        spacing: 5

        Label {
            id: label
        }

        SpinBox {
            id: spinbox1; width: 100
            from: 100; to: 0; value: 50
            onValueChanged: label.text = "spinbox1: " + value
        }
    }
}
```



- Switch

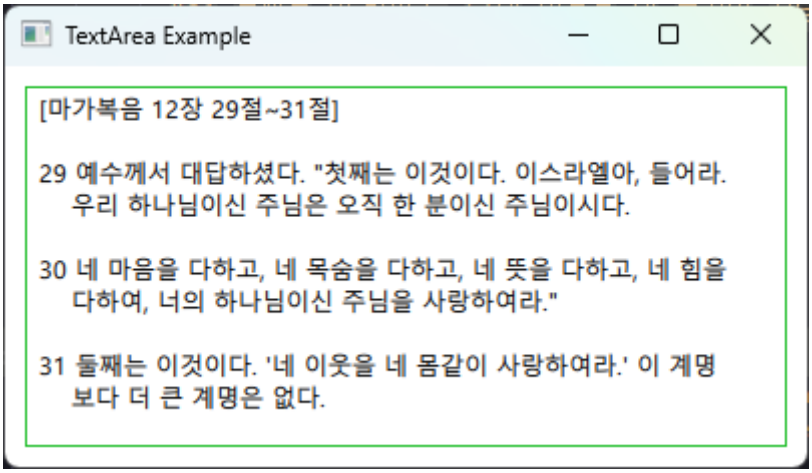
```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

Window {
    width: 100
    height: 100
    visible: true

    Column {
        anchors.centerIn: parent
        spacing: 5
        Switch {
            id: first
            checked: true
            onClicked: {
                if(checked)
                    console.log("checked true");
                else
                    console.log("checked false")
            }
        }

        Switch {
            checked: false
        }
    }
```

```
}
```



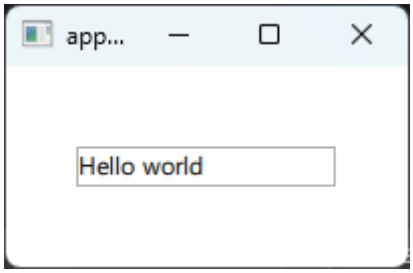
- TextArea

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

Window {
    width: 400; height: 200; visible: true
    title: "TextArea Example"

    TextArea {
        anchors.centerIn: parent
        background: Rectangle {
            border.color: "#21be2b"
        }

        width: 380; height: 180
        text:
          "[마가복음 12장 29절~31절] \n\n" +
          "29 예수께서 대답하셨다. \"첫째는 이것이다. 이스라엘아, 들어라.\n" +
          "    우리 하나님이신 주님은 오직 한 분이신 주님이시다.\n\n" +
          "30 네 마음을 다하고, 네 목숨을 다하고, 네 뜻을 다하고, 네 힘을\n" +
          "    다하여, 너의 하나님이신 주님을 사랑하여라.\" \n\n" +
          "31 둘째는 이것이다. \'네 이웃을 네 몸같이 사랑하여라.\' 이 계명\n" +
          "    보다 더 큰 계명은 없다."
    }
}
```



- TextField

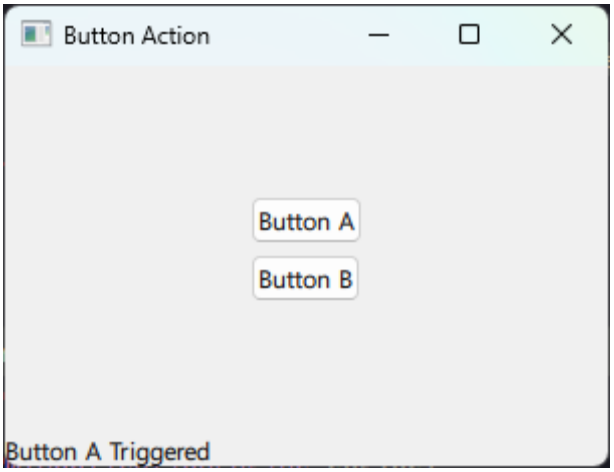
```
import QtQuick
import QtQuick.Window
import QtQuick.Controls

Window {
    width: 200
    height: 100
    visible: true

    TextField {
        anchors.centerIn: parent
    }
}
```



- Button and Action example

In this example, we'll use an Action to handle an event from a Button type.

```
import QtQuick
import QtQuick.Controls
import QtQuick.Window
import QtQuick.Layouts

ApplicationWindow {
    title: qsTr("Button Action")
    width: 300; height: 200; visible: true

    ColumnLayout{
        anchors.centerIn: parent
        Button { action: actionButtonA }
        Button { action: actionButtonB }
    }

    Action{
        id: actionButtonA
        text: "Button A"
        onTriggered: statusLabel.text = "Button A Triggered "
    }

    Action{
        id: actionButtonB
        text: "Button B"
        checkable: true
        onCheckedChanged:
            statusLabel.text = "Button B checked: " + checked
    }

    footer: Label {
        id: statusLabel; text: ""
    }
}
```



You can find the source code for this example in the 01_Button_Action directory.

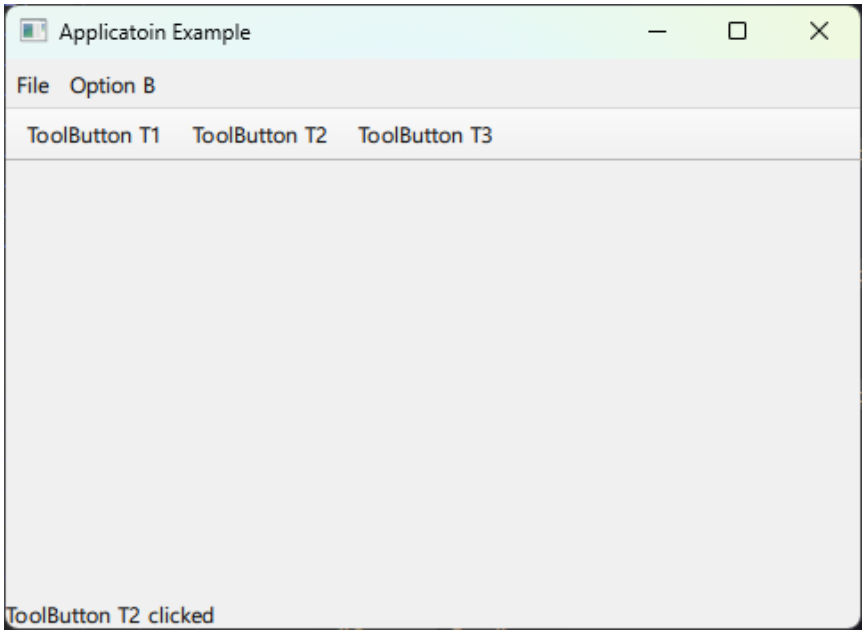
- Example using the ApplicationWindow type

In this example, we implemented a window GUI screen using the ApplicationWindow, MenuBar, ToolBar, and StatusBar types.

```
import QtQuick
import QtQuick.Controls
import QtQuick.Window
import QtQuick.Layouts

ApplicationWindow {
    id: myWindow; width: 480; height: 320
    title: "Applicatoin Example"
    visible: true

    menuBar: MenuBar {
        Menu {
            title: qsTr("File")
            MenuItem {
                text: qsTr("Exit")
                onTriggered: Qt.quit();
            }
            MenuItem {
                text: "File Item2"
                onTriggered: statusLabel.text = "File Item2 Click"
```

```
            }
        }
        Menu {
            title: "Option B"
            MenuItem {
                text: "Opt B1"
                onTriggered: statusLabel.text = "Opt B1 Click"
            }
            MenuItem {
                text: "Opt B2"; checkable: true
                onCheckedChanged:
                    statusLabel.text = "Opt B2 Checked - " + checked
            }
            MenuItem {
                text: "Opt B3"; checkable: true
                onCheckedChanged:
                    statusLabel.text = "Opt B3 Checked - " + checked
            }
        }
    }

    header: ToolBar{
        RowLayout{
            ToolButton{
                text: "ToolButton T1"
                onClicked: statusLabel.text = "ToolButton T1 clicked"
            }
            ToolButton{
                text: "ToolButton T2"
                onClicked: statusLabel.text = "ToolButton T2 clicked"
            }
            ToolButton{
                text: "ToolButton T3"
                onClicked: statusLabel.text = "ToolButton T3 clicked"
            }
        }
    }

    footer: RowLayout{
```

```
        Label{
            id: statusLabel
            text: "Status Bar"
        }
    }
}
```



You can find the source code for this example in the 02_ApplicationWindow directory.

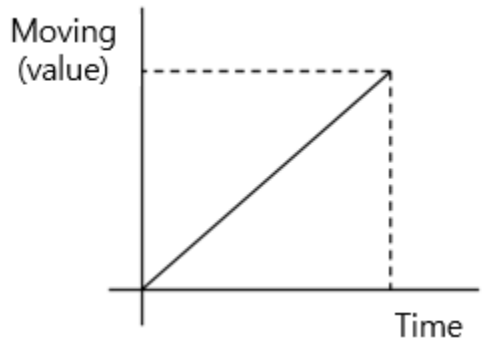
3. Animation Framework

To use animations, such as smooth screen transitions, we provide the Animation Framework.

You can use animation elements such as transparency, movement, zooming, etc. as animation elements. For example, suppose you have a windowed screen with a width and height of 600 x 400 pixels when your application is running. You can have the window pane increase in size from an initial size of 100 x 100 to 600 x 400 for a specified amount of time.

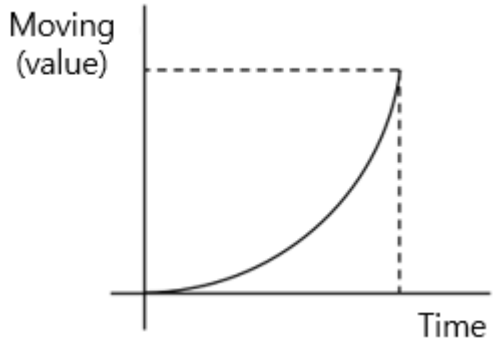
At the same time, you can add an Animation effect using Opacity. 0.0 is fully transparent and 1.0 is fully opaque. You can animate the value to change from 0.0 to 1.0 over a specified time period.

In addition, Easing Curves can be used for time values in animations. For example, suppose you want the X,Y position of a GUI button to move from the position of 100,100 to the coordinates of 200,200 in 1.0 seconds, and you specify a travel time of 1.0 seconds. It will move at a constant speed from the X,Y start coordinate to the destination coordinate.

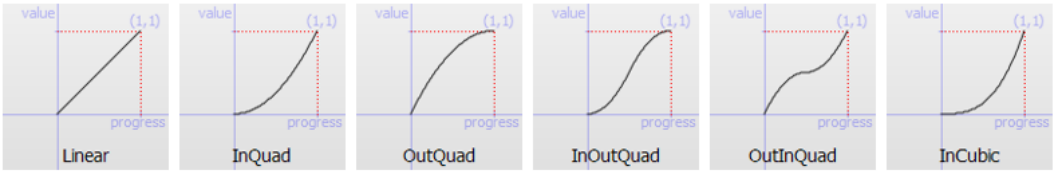


As shown in the figure above, the Y-axis labeled Move (Value) is a graph that shows that it takes a constant 1.0 second time for the X,Y position coordinates of the GUI button to move from 100,100 to the value 200,200.

In other words, the time increases linearly with the movement value. However, if you use Easing Curves for Time, you can change the value of Time as a function of the movement value, as shown in the following figure.



The value of the time to travel to the final X, Y coordinates can be applied as shown in the figure above. In addition, Easing Curve can apply about 46 different Easing Curves.



QML can also use a technique called State Transition for Animation elements. An example of this is an ON/OFF switch. Not only can you use ON/OFF to change a single value, but you can use multiple options together.

In other words, you can use State Transition, which is a technique that allows you to apply multiple values simultaneously depending on the situation. In this chapter, we'll take a look at what we've discussed so far.

3.1. Animation

In Qt, there are several commonly used Animation hits, as shown in the table below.

Type	Description
NumberAnimation	Animating with numbers, such as coordinates X, Y, or transparent values like Opacity.
PropertyAnimation	Animating properties like RGB, width, height, etc.
RotationAnimation	Apply a Rotation Animation
PauseAnimation	Functions for Stopping Animation
SmoothedAnimation	Features for using gravity acceleration weighting
SpringAnimation	Provided for use with animations such as springs
Behavior	An animation that occurs when a specific value changes, such as running an animation when the width changes.
ScriptAction	Provided for the purpose of executing a specific Method or Script when the Animation runs.
PropertyAction	Provided to change the value of an Element's properties during animation behavior.

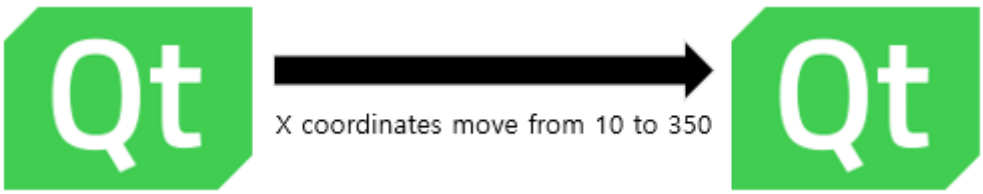
The following provides types that Animation can be grouped into.

Type	Description
SequentialAnimation	Running Multiple Animations in Sequence
ParrellelAnimation	Running Multiple Animations in Parallel

- NumberAnimation

c NumberAnimation can animate using numbers. For example, the Image type specified by the X and Y coordinates changes the X coordinate from 10 to the value 250 for 2,000 milliseconds. If you don't specify a value for the duration property, it defaults to 250 milliseconds.

```
import QtQuick
import QtQuick.Window

Window {
    width: 630; height: 230; visible: true

    Image {
        source: "images/qtlogo.png"
        x: 10; y: 20;

        NumberAnimation on x
        {
            from: 10; to: 350
            duration: 2000
        }
    }
}
```



- PropertyAnimation

PropertyAnimation can animate a property of a type. The following example uses the values of the width and height properties of an Image type as the values for an animation.

```
import QtQuick
import QtQuick.Window

Window {
    width: 230; height: 230; visible: true

    Image {
        id: proAni
        source: "images/qtlogo.png"
        x: 50; y: 40;
```

```
        width: 50; height: 50;
    }

    PropertyAnimation {
        target: proAni
        properties: "width, height"
        from: 0; to: 100; duration: 1000
        running: true
        //loops : Animation.Infinite
    }
}
```

- RotationAnimation

The RotationAnimation type can animate a start and end value. The start and end values are the values of the angle. So the from property is the starting angle and the to property is the ending angle.

```
import QtQuick
import QtQuick.Window

Window {
    width: 330; height: 330; visible: true

    Image {
        id: rotAni
        source: "images/qtlogo.png"
        anchors.centerIn: parent
```

```
        smooth: true

        RotationAnimation on rotation {
            from: 45; to: 315
            direction: RotationAnimation.Shortest
            duration: 3000
        }
    }
}
```

In the example above, the direction property of a RotationAnimation can specify whether it should rotate clockwise or counterclockwise. The following constants can be used as values for the direction property

Contant	Description
RotationAnimation.Numerical	Linear 보간 알고리즘 사용. 방향은 시계 방향
RotationAnimation.Clockwise	시계 방향으로 회전
RotationAnimation.Counterclockwise	반 시계 방향으로 회전
RotationAnimation.Shortest	STEP 은 20도씩 회전. 회전 방향은 반 시계 방향.

- Events in Animation

Events can be fired the moment the Animation starts, when the Animation stops, or when the value of a property of the Animation changes.

```
import QtQuick
import QtQuick.Window

Window {
    width: 330; height: 230; visible: true
    Image {
        source: "images/qtlogo.png"
        x: 10; y: 20;
        id : logo

        NumberAnimation on scale
        {
            id : scaleAni
            from: 0.1; to: 1.0
            duration: 2000
            running: true

            onStarted : console.log("started")
            onStopped : console.log("stopped")
        }

        onScaleChanged: {
            if(scale > 0.5) {
                scaleAni.complete()
            }
        }
    }
}
```

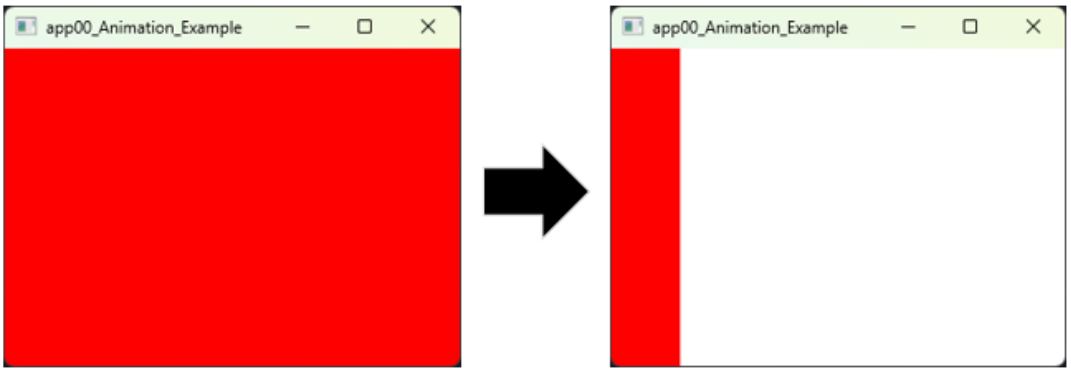
- Behavior

Behavior 는 타입의 프로퍼티의 값이 변경 되었을 때 동작 시키기 위한 목적으로 사용한다.

```
import QtQuick
import QtQuick.Window

Window {
    width: 330; height: 230; visible: true
```

```
    Rectangle {
        width: 330; height: 230;
        id: rect
        color: "red"

        Behavior on width {
            NumberAnimation {
                duration: 1000
            }
        }

        MouseArea {
            anchors.fill: parent
            onClicked: rect.width = 50
        }
    }
}
```

In the example above, when the width value of the Rectangle type changes to 50 when the mouse is clicked, the width value changes from the current value of 330 to 50, which can be animated.

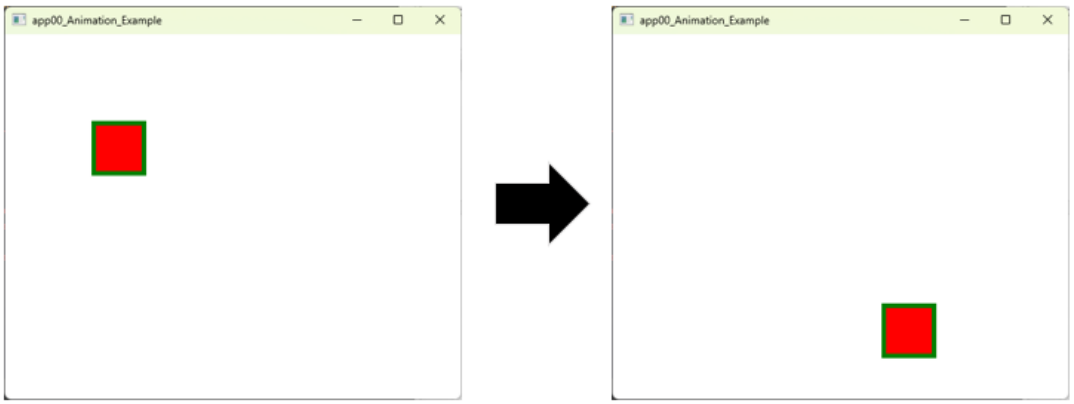


- SmoothedAnimation

The SmoothedAnimation type provides the ability to handle animation using the value of the velocity property. The following example shows a Rectangle moving in response to keyboard arrow keys.

```
import QtQuick
```

```
import QtQuick.Window

Window {
    width: 800; height: 600; visible: true

    Rectangle {
        width: 800; height: 600
        Rectangle {
            width: 60; height: 60
            x: rect1.x - 5; y: rect1.y - 5
            color: "green"

            Behavior on x { SmoothedAnimation { velocity: 200 } }
            Behavior on y { SmoothedAnimation { velocity: 200 } }
        }

        Rectangle {
            id: rect1; width: 50; height: 50; color: "red"
        }

        focus: true
        Keys.onRightPressed: rect1.x = rect1.x + 100
        Keys.onLeftPressed: rect1.x = rect1.x - 100
        Keys.onUpPressed: rect1.y = rect1.y - 100
        Keys.onDownPressed: rect1.y = rect1.y + 100
    }
}
```



- SpringAnimation

The SpringAnimation type can use animations like springs. The spring property can reflect effects such as the spring's elasticity.

For example, the property can have the effect of stretching the spring and then releasing it, causing it to contract quickly. Therefore, a large value for the spring property indicates a high degree of elasticity or cohesion.

The damping property can be used for effects like braking. The larger the value, the faster the brake is applied. The value can be set between 0.0 and 1.0.

```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true

    Rectangle {
        id: rect
        width: 50; height: 50
        color: "red"

        Behavior on x {
            SpringAnimation {
                spring: 14; damping: 0.2
            }
        }

        Behavior on y {
            SpringAnimation {
                spring: 14; damping: 0.2
            }
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
```

```
            rect.x = mouseX - rect.width/2
            rect.y = mouseY - rect.height/2
        }
    }
}
```

- SequentialAnimation

The SequentialAnimation type provides functionality for running multiple Animations in sequence.

```
import QtQuick
import QtQuick.Window

Window {
    width: 300
    height: 200
    visible: true

    Image {
        id: aniSeq
        anchors.centerIn: parent
        source: "images/qtlogo.png"
    }

    SequentialAnimation {
        NumberAnimation {
            target: aniSeq; properties: "scale"
            from: 1.0; to: 0.5; duration: 1000
        }
        NumberAnimation {
            target: aniSeq; properties: "opacity"
            from: 1.0; to: 0.5; duration: 1000
        }
        running: true
    }
}
```

- **ParallelAnimation**

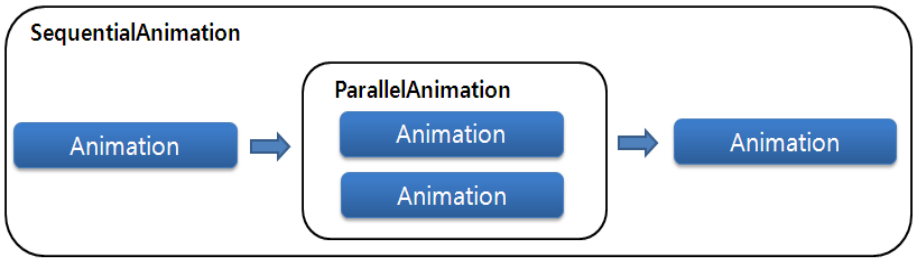
ParallelAnimation provides the ability to perform all of the Animations used simultaneously.

For example, the previous SequentialAnimation runs the Animations in sequence, while ParallelAnimation runs them simultaneously (in parallel).

```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true

    Image {
        id: aniPara
        anchors.centerIn: parent
        source: "images/qtlogo.png"
    }

    ParallelAnimation {
        NumberAnimation {
            target: aniPara; properties: "scale"
            from: 1.0; to: 0.5; duration: 1000
        }
        NumberAnimation {
            target: aniPara; properties: "opacity"
            from: 1.0; to: 0.5; duration: 1000
        }
        running: true
    }
}
```

- Nested examples of SequentialAnimation and ParallelAnimation

SequentialAnimation and ParallelAnimation can be performed in a nested structure. For example, you can perform them in a nested structure as shown below.

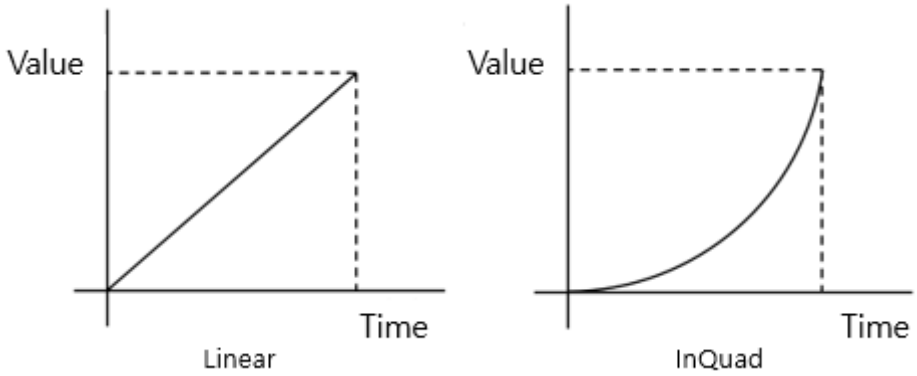


The following example uses SequentialAnimation and ParallelAnimation in a nested structure, as shown in the figure above.

```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true

    Image {
        id: qtlogo; anchors.centerIn: parent
        source: "./images/qtlogo.png"
    }

    SequentialAnimation {
        NumberAnimation {
            target: qtlogo; properties: "scale"
            from: 1.0; to: 0.5; duration: 1000
        }
        SequentialAnimation {
            NumberAnimation {
                target: qtlogo; properties: "rotation"
                from: 0.0; to: 360.0; duration: 1000
            }
            NumberAnimation {
                target: qtlogo; properties: "opacity"
                from: 1.0; to: 0.0; duration: 1000
            }
        }
        running: true
    }
}
```

- Easing Curve

The Easing Curve allows you to apply a variety of curves, such as shifting coordinates, transparency, zooming in and out, etc. without the animation running in a linear pattern between the start and end of the animation. For example, you can change the rate at which the path progresses from the start coordinate of a type to the last coordinate in the following pattern.



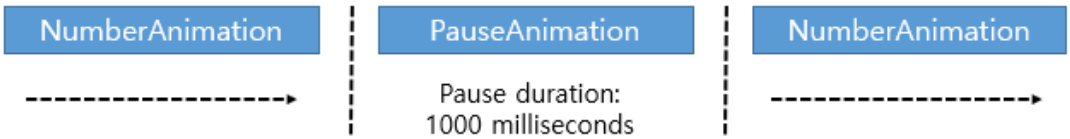
The following example source code is an example of using the OutExpo of an Easing curve.

```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true
    Image {
        id: easCur
        anchors.centerIn: parent
        source: "images/qtlogo.png"

        NumberAnimation {
            target: easCur; properties: "scale"
            from: 0.1; to: 1.0; duration: 1000
            easing.type: "OutExpo"
            running: true
        }
    }
}
```

In the example above, the easing.type property can specify one of the easing curves to use. You can understand the difference by comparing the difference between not using an easing curve and using one.

- PauseAnimation

A PauseAnimation can pause an Animation between sequentially executing Animations in a SequentialAnimation for the time specified by the value of the duration property.



The following example uses the PauseAnimation type.

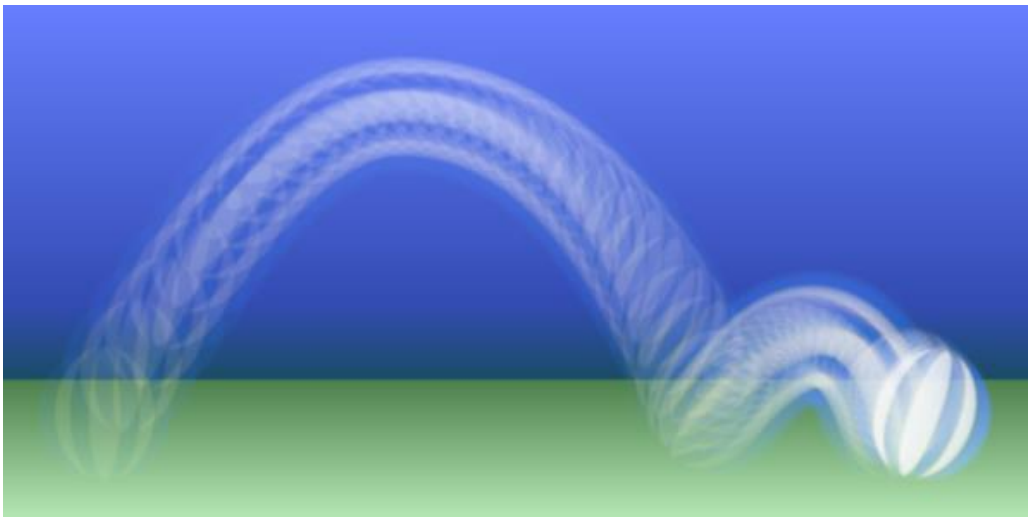
```
import QtQuick
import QtQuick.Window

Window {
    width: 300
    height: 200
    visible: true

    Image {
        id: logo
        anchors.centerIn: parent; source: "images/qtlogo.png"
    }

    SequentialAnimation {
        NumberAnimation {
            target: logo; properties: "scale"
            from: 0.0; to: 1.0; duration: 1000
        }
        PauseAnimation {
            duration: 1000
        }
        NumberAnimation {
            target: logo; properties: "scale"
            from: 1.0; to: 0.0; duration: 1000
        }
```

```
        running: true
    }
}
```

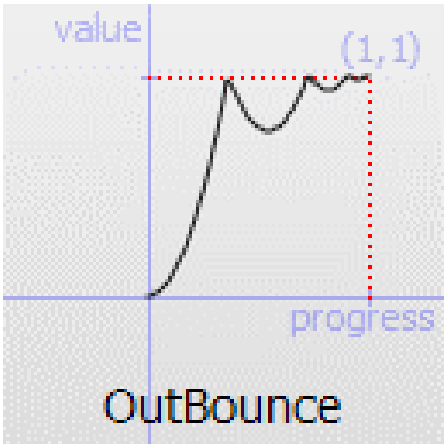
For the example source code covered in this chapter, see the 00_Animation_Example directory.

3.2. Example with Animation and Easing curve

In this section, we will create an example using Animation and an Easing curve.



Let's create an example where a Ball bounces on the floor as shown in the image above. First, let's use the Image type to display the Ball image and then animate the Ball image to bounce as it falls from the top to the bottom as shown in the following example. To animate the bounce of the Ball image as it hits the floor, use the OutBounce of the Easing curve.




```
import QtQuick
import QtQuick.Window
```

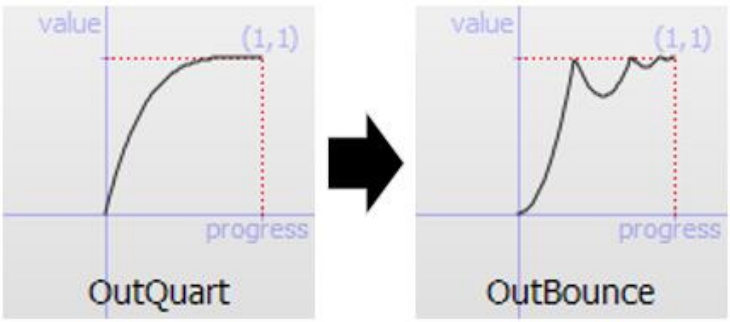
```
Window {
    width: 600; height: 300; visible: true

    Image {
        source: "images/ball.png"
        anchors.horizontalCenter: parent.horizontalCenter

        NumberAnimation on y
        {
            from: 20; to: 200
            easing.type: "OutBounce"
            duration: 1000
        }
    }
}
```

When created and run as above, the Ball image should behave as if it is falling from below and bouncing around. Let's add another NumberAnimation, this time a NumberAnimation.

```
import QtQuick
import QtQuick.Window

Window {
    width: 600; height: 300; visible: true

    Image {
        source: "images/ball.png"
        anchors.horizontalCenter: parent.horizontalCenter

        SequentialAnimation on y {
            NumberAnimation {
                from: 200; to: 20
                easing.type: "OutQuad"
                duration: 250
            }
            NumberAnimation {
                from: 20; to: 200
                easing.type: "OutBounce"
```

```
                duration: 1000
            }
        }
    }
}
```

The animation we added animates the Ball image as if it is being thrown upwards, so the animation is performed sequentially as shown in the image below.



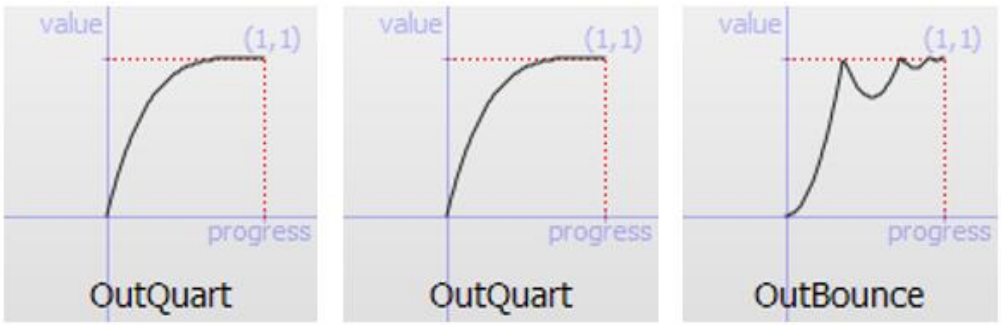
So far, the Ball image has moved along the Y-axis. Let's animate it along the X-axis. To move along the X-axis, add a NumberAnimation as follows.

```
import QtQuick
import QtQuick.Window

Window {
    width: 600; height: 300; visible: true

    Image {
        source: "images/ball.png"

        NumberAnimation on x {
            from: 20; to: 500
            easing.type: "OutQuad"
            duration: 1250
        }

        SequentialAnimation on y {
            NumberAnimation {
                from: 200; to: 20
                easing.type: "OutQuad"
                duration: 250
```

```
            }
            NumberAnimation {
                from: 20; to: 200
                easing.type: "OutBounce"
                duration: 1000
            }
        }
    }
}
```

When you add a NumberAnimation as shown above, the Ball image moves from left to right. When you run the example, the Ball image moves from left to right, and at the same time, the ball bounces.

For example, if we throw our Ball, it will bounce on the floor and the animation will behave as if it is bouncing. In this example, we used three Animations, as shown in the following image.



Next, let's add a RotationAnimation. The RotationAnimation rotates the Ball image 360 degrees in a clockwise direction.

```
import QtQuick
import QtQuick.Window

Window {
    width: 600; height: 300; visible: true

    Image {
        source: "images/ball.png"

        NumberAnimation on x {
            from: 20; to: 500
```

```
            easing.type: "OutQuad"
            duration: 1250
        }

        SequentialAnimation on y {
            NumberAnimation {
                from: 200; to: 20
                easing.type: "OutQuad"
                duration: 250
            }
            NumberAnimation {
                from: 20; to: 200
                easing.type: "OutBounce"
                duration: 1000
            }
        }

        RotationAnimation on rotation {
            from: 0; to: 360
            direction: RotationAnimation.Clockwise
            duration: 1000
        }
    }
}
```

Next, let's use ParallelAnimation and SequentialAnimation for Animation.

```
import QtQuick
import QtQuick.Window

Window {
    width: 600; height: 300; visible: true

    Image {
        id: ball
        source: "images/ball.png"
        x: 20; y: 200
        smooth: true
        MouseArea {
            anchors.fill: parent
```

```
            onClicked: ballAnimation.running = true
        }

        ParallelAnimation {
            id: ballAnimation

            NumberAnimation {
                target: ball
                property: "x"
                from: 20; to: 500
                easing.type: "OutQuad"
                duration: 1250
            }

            SequentialAnimation {
                NumberAnimation {
                    target: ball
                    property: "y"
                    from: 200; to: 20
                    easing.type: "OutQuad"
                    duration: 250
                }
                NumberAnimation {
                    target: ball
                    property: "y"
                    from: 20; to: 200
                    easing.type: "OutBounce"
                    duration: 1000
                }
            }

            SequentialAnimation {
                RotationAnimation {
                    target: ball
                    property: "rotation"
                    from: 0; to: 360
                    direction: RotationAnimation.Clockwise
                    duration: 1000
                }
```

```
                RotationAnimation {
                    target: ball
                    property: "rotation"
                    from: 360; to: 380
                    direction: RotationAnimation.Clockwise
                    duration: 250
                }
            }
        }
    }
}
```

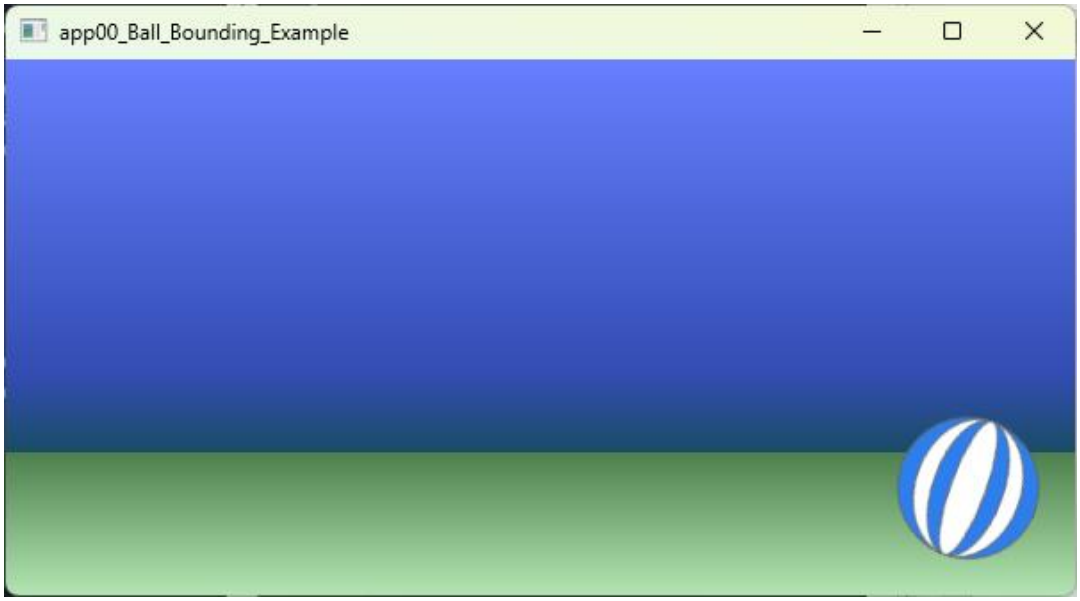
Finally, let's use a Gradient to paint the background color. Add the Rectangle and Gradient types to the code written as shown in the following example.

```
import QtQuick
import QtQuick.Window

Window {
    width: 600; height: 300; visible: true

    Rectangle {
        x: 0; y:0
        width: parent.width; height: 220
        gradient: Gradient {
            GradientStop { position: 0.0; color: Qt.rgba(0.4,0.5,1.0,1) }
            GradientStop { position: 0.8; color: Qt.rgba(0.2,0.3,0.7,1) }
            GradientStop { position: 1.0; color: Qt.rgba(0.1,0.3,0.4,1) }
        }
    }
    Rectangle {
        y: 220
        width: parent.width; height: 80
        gradient: Gradient {
            GradientStop { position: 1.0; color: Qt.rgba(0.7,0.9,0.7,1) }
            GradientStop { position: 0.0; color: Qt.rgba(0.3,0.5,0.3,1) }
        }
    }

    Image {
```

```
        id: ball
        source: "images/ball.png"
        x: 20; y: 200
        smooth: true

        MouseArea {
            anchors.fill: parent
            onClicked: ballAnimation.running = true
        }

        ParallelAnimation {
            id: ballAnimation

            NumberAnimation {
                target: ball
                property: "x"
                from: 20; to: 500
                easing.type: "OutQuad"
                duration: 1250
            }

            SequentialAnimation {
                NumberAnimation {
                    target: ball
                    property: "y"
                    from: 200; to: 20
                    easing.type: "OutQuad"
                    duration: 250
                }
                NumberAnimation {
                    target: ball
                    property: "y"
                    from: 20; to: 200
                    easing.type: "OutBounce"
                    duration: 1000
                }
            }

            SequentialAnimation {
```

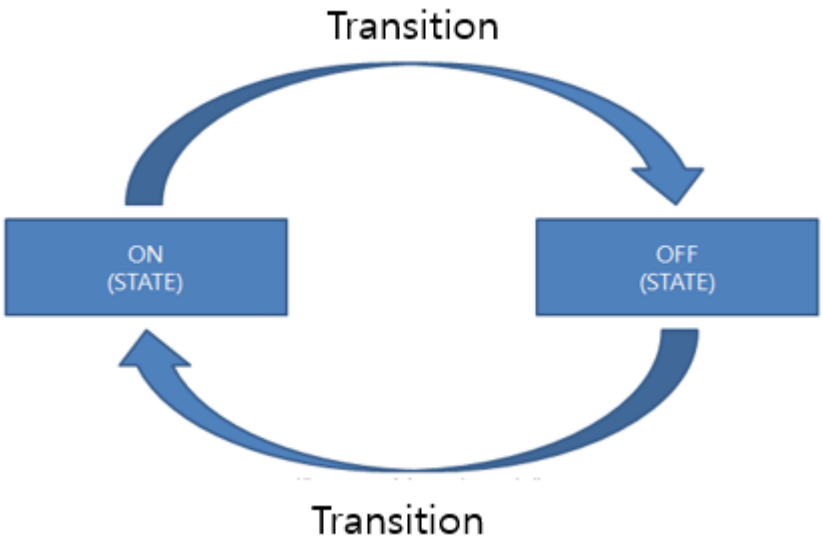
```
                RotationAnimation {
                    target: ball
                    property: "rotation"
                    from: 0; to: 360
                    direction: RotationAnimation.Clockwise
                    duration: 1000
                }
                RotationAnimation {
                    target: ball
                    property: "rotation"
                    from: 360; to: 380
                    direction: RotationAnimation.Clockwise
                    duration: 250
                }
            }
        }
    }
}
```

If you write the code like above, you can see that the Ball image is animated to bounce. Run the example and click on the Ball image with the mouse to activate the animation.



3.3. State and Transition

States and Transitions can apply state machine techniques to types in QML. For example, suppose you have an ON/OFF switch. We define the ON or OFF state as a State. And the change of state from ON to OFF or from OFF to ON is defined as an action, or Transition. In QML, a State is a state of a certain type, and we can define a Transition as an event that causes a QML type to be animated and change its property attributes.

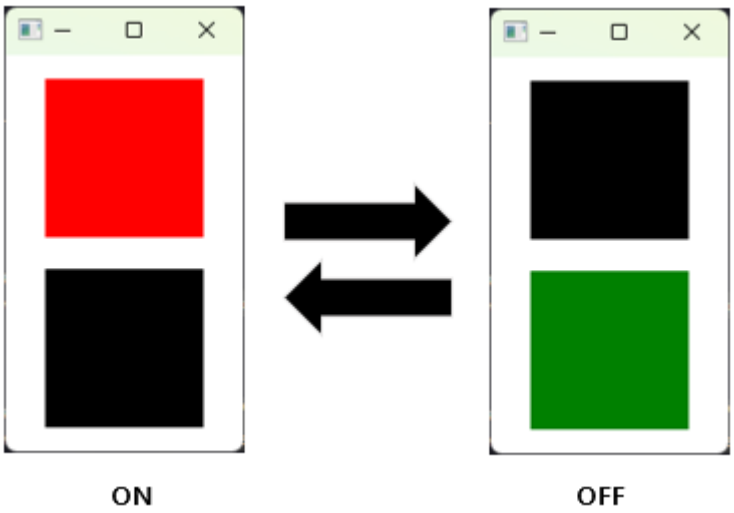


- Example using State

For example, a Color of type Rectangle that is Red can be called a State. When the color of a Rectangle type changes from Red to Black, we can define it as a Transition. The following example shows how to implement the function of changing the color of a Rectangle using State.

```
import QtQuick
import QtQuick.Window

Window {
    width: 150; height: 250; visible: true

    Rectangle {
```

```
        width: parent.width; height: parent.height

        Rectangle {
            id: onElement
            x: 25; y: 15; width: 100; height: 100
        }

        Rectangle {
            id: offElement
            x: 25; y: 135; width: 100; height: 100
        }

        states: [
            State {
                name: "on"
                PropertyChanges { target: onElement;  color: "red" }
                PropertyChanges { target: offElement; color: "black" }
            },
            State {
                name: "off"
                PropertyChanges { target: onElement; color: "black" }
                PropertyChanges { target: offElement;  color: "green" }
            }
        ]

        state: "on"

        MouseArea
        {
            anchors.fill: parent
            onClicked: parent.state === "on" ?
                            parent.state = "off" : parent.state = "on"
        }
    }
}
```



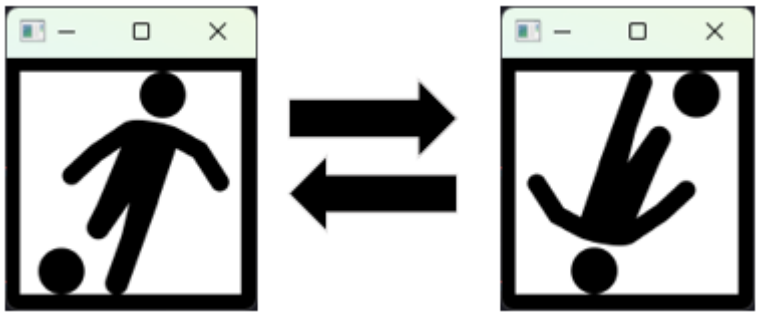
- Example using State and Transition

For this example, we'll use State and Transition to implement the example. As shown in the image below, the image will be rotated when the example is executed.

If the State has two states, "up" and "down", then when the Transition from "up" to "down" is executed, the image will rotate 180 degrees. Conversely, if a transition from "down" to "up" is performed, the image is rotated 180 degrees.

```
import QtQuick
import QtQuick.Window

Window {
  width: 150; height: 150; visible: true

    Rectangle {
        width: 150; height: 150; color: "black"

        Image {
            id: player
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.verticalCenter: parent.verticalCenter
            source: "images/player.png"
        }

        states: [
```

```
            State {
                name: "up"
                PropertyChanges { target: player; rotation: 0 }
            },
            State {
                name: "down"
                PropertyChanges { target: player; rotation: 180 }
            }
        ]

        state: "up"

        transitions: [
            Transition {
                from: "*"; to: "*"
                PropertyAnimation {
                    target: player
                    properties: "rotation"; duration: 1000
                }
            }
        ]
        MouseArea {
            anchors.fill: parent
            onClicked: parent.state === "up" ?
                    parent.state = "down" : parent.state = "up"
        }
    }
}
```

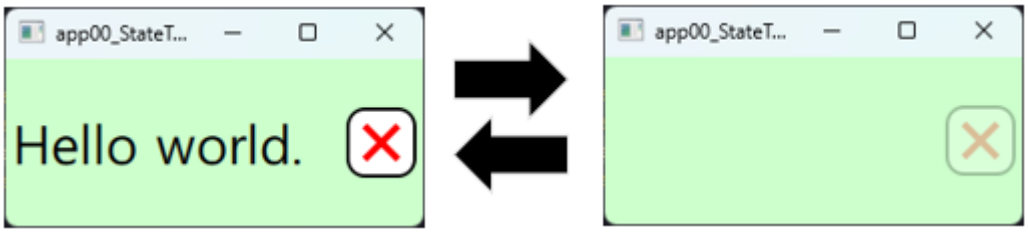


- State and when

The when property of a State is executed when certain conditions are met. You can use the when property as shown in the following example.

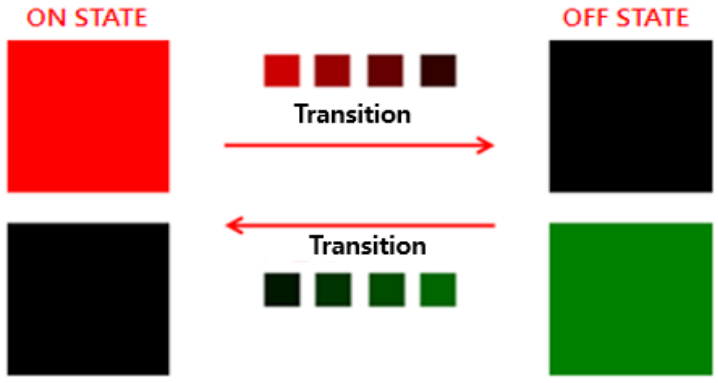
```
import QtQuick
import QtQuick.Window

Window {
  width: 250; height: 100; visible: true

    Rectangle {
        width: 250; height: 100; color: "#ccffcc"

        TextInput {
            id: textField
            text: "Hello world."
            font.pointSize: 24
            anchors.left: parent.left
            anchors.leftMargin: 4
            anchors.verticalCenter: parent.verticalCenter
        }

        Image {
            id: clearButton
            source: "images/clear.svg"
            anchors.right: parent.right
            anchors.rightMargin: 4
            anchors.verticalCenter: textField.verticalCenter

            MouseArea {
                anchors.fill: parent
                onClicked: textField.text = ""
            }
        }

        states: [
            State {
                name: "with text"
                when: textField.text !== ""
```

```
                PropertyChanges { target: clearButton; opacity: 1.0 }
            },
            State {
                name: "without text"
                when: textField.text === ""
                    PropertyChanges { target: clearButton; opacity: 0.25 }
                    PropertyChanges { target: textField; focus: true }
            }
        ]
    }
}
```



- Using Animation in Transition

In this example, we will use a PropertyAnimation when a Transition is performed. By setting the property of PropertyAnimation to the color property and the duration property to 1000, we can use an animation effect, such as a sharpening of the color value for 1000 milliseconds. The first Transition is specified to change from the "off" state to the "on" state, and the second Transition is specified to change from the "on" state to the "off" state.

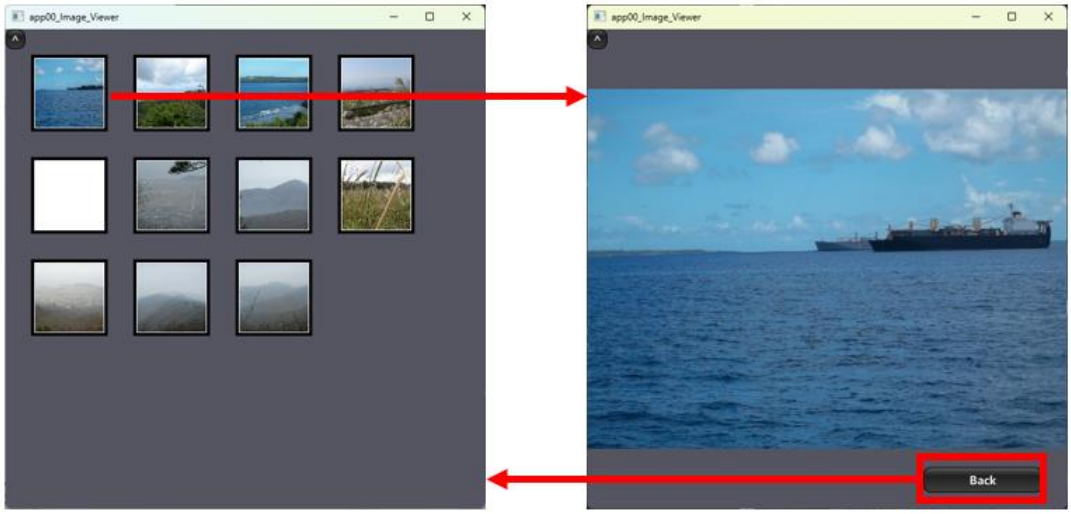


```
import QtQuick
import QtQuick.Window

Window {
    width: 150; height: 250; visible: true

    Rectangle {
        width: 150; height: 250
        Rectangle {
            id: onElement; x: 25; y: 15; width: 100; height: 100
        }

        Rectangle {
            id: offElement; x: 25; y: 135; width: 100; height: 100
        }

        states: [
            State {
                name: "on"
                PropertyChanges { target: onElement;  color: "red" }
                PropertyChanges { target: offElement; color: "black" }
            },
            State {
                name: "off"
                PropertyChanges { target: onElement; color: "black" }
                PropertyChanges { target: offElement;  color: "green" }
            }
        ]

        state: "on"

        MouseArea {
            anchors.fill: parent
            onClicked: parent.state === "on" ?
                    parent.state = "off" : parent.state = "on"
        }

        transitions: [
            Transition {
```

```
                from: "off"; to: "on"
                PropertyAnimation {
                    target: onElement
                    properties: "color"
                    duration: 1000
                }
            },
            Transition {
                from: "on"; to: "off"
                PropertyAnimation {
                    target: offElement
                    properties: "color"
                    duration: 1000
                }
            }
        ]
    }
}
```

3.4. Implementing the Image Viewer

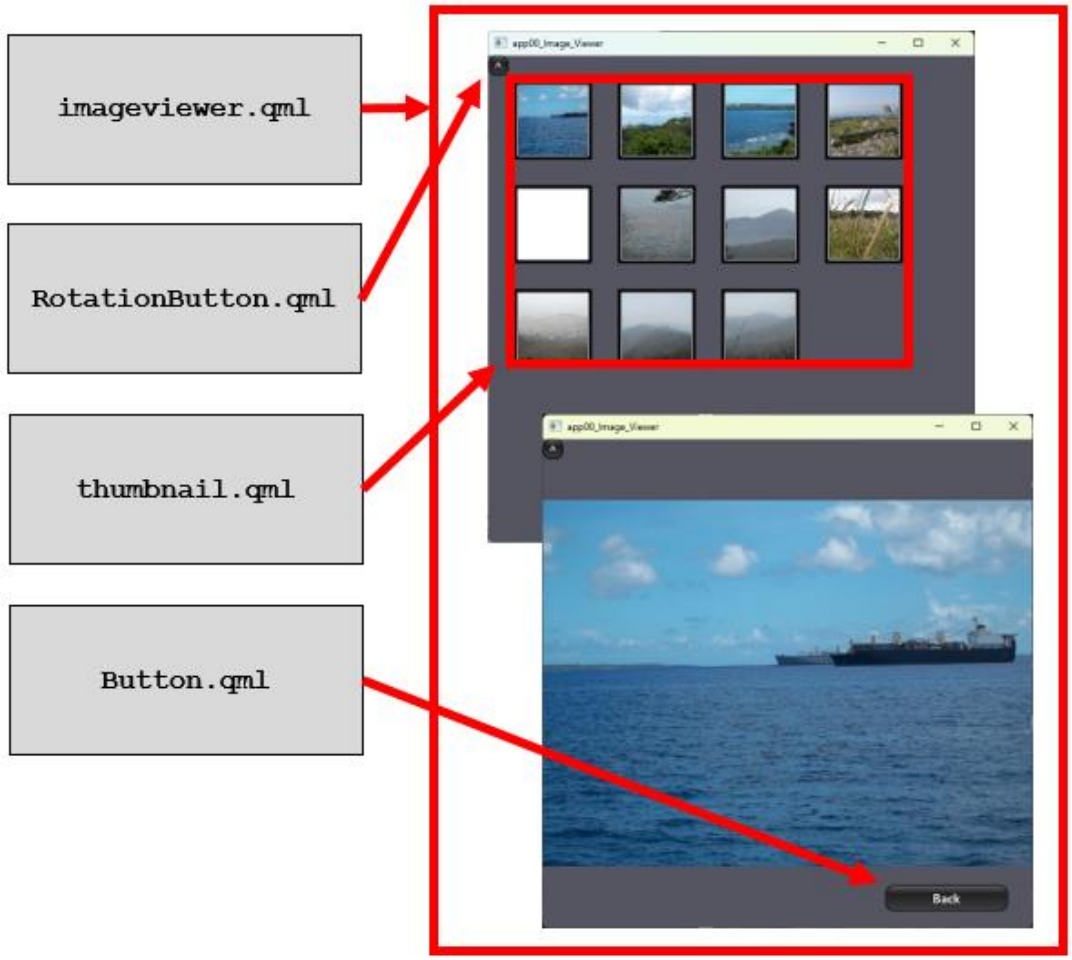
For this example, we will implement an Image Viewer. The Image Viewer is a screen where you can view thumbnail images on the main screen. If you click on an image you want to see larger, you can enlarge the image as shown in the image below.



If you look at the top left of the screen, you'll see an image displayed. This image can rotate the screen 90 degrees. For example, if we look at our phone in portrait or landscape orientation, the screen will rotate 90 degrees to the left or right.

This is an example of an Image Viewer example that provides similar functionality. The Image Viewer example consists of four QML source files, as shown in the following figure.

QML File	Description
ImageViewer.qml	The main QML source code where you placed the QML type.
RotationButton.qml	Ability to rotate screen -90 degrees and click again to rotate to 0 degrees
Button.qml	button, which returns to the previous screen when clicked.
Thumbnail.qml	A custom type called Thumbnail



ImageViewer.qml calls Thumbnail, RotationButton, and Button to use them. The following example is the source code for the ImageViewer.qml example.

```
import QtQuick
import QtQuick.Window
import "module"

Window {
    width: 600; height: 600; visible: true

    Item {
        id: screen; width: 600; height: 600
        property int animDuration: 500

        RotationButton {
            id: rotationButton
```

```
            duration: screen.animDuration
            z: 100
            anchors.top: screen.top
            anchors.left: screen.left
        }

        Rectangle {
            id: background
            anchors.centerIn: parent; color: "#555560";
            width: rotationButton.angle == 0 ? parent.width : parent.height
            height: rotationButton.angle ==0 ? parent.height : parent.width
            rotation: rotationButton.angle

            Behavior on rotation {
                RotationAnimation {
                    duration: screen.animDuration
                    easing.type: Easing.InOutQuad
                }
            }

            Item {
                id: grid
                anchors.fill: parent

                function displayPicture(path) {
                    picture.source = path
                    screen.state = "displayPicture"
                }

                Thumbnail {
                    column: 0; row: 0; image: "images/101.JPG"
                    onClicked: parent.displayPicture(image)
                }
                Thumbnail {
                    column: 1; row: 0; image: "images/102.JPG"
                    onClicked: parent.displayPicture(image)
                }
                Thumbnail {
                    column: 2; row: 0; image: "images/103.JPG"
```

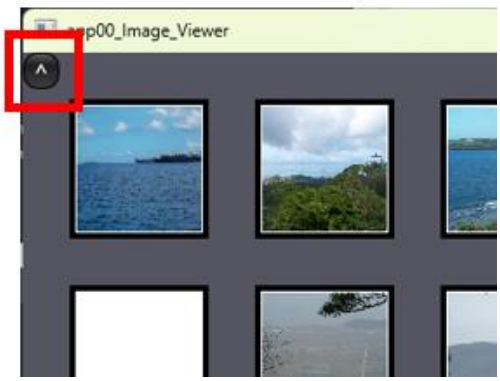
```
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 3; row: 0; image: "images/104.JPG"
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 0; row: 1; image: "images/105.JPG"
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 1; row: 1; image: "images/106.JPG"
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 2; row: 1; image: "images/107.JPG"
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 3; row: 1; image: "images/108.JPG"
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 0; row: 2; image: "images/109.JPG"
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 1; row: 2; image: "images/110.JPG"
            onClicked: parent.displayPicture(image)
        }
        Thumbnail {
            column: 2; row: 2; image: "images/111.JPG"
            onClicked: parent.displayPicture(image)
        }
    }

    Image {
        id: picture
        z: 2
```

```
            x: 2 * parent.width; y: 0
            width: parent.width; height: parent.height
            smooth: true
            fillMode: Image.PreserveAspectFit
        }

        Button {
            id: backButton
            width: 150
            height: 32

            x: parent.width - width - 30
            y: parent.height + 3 * height
            z: 5

            text: "Back"
            onClicked: screen.state = "displayGrid"
            visible: false
        }
    }

    state: "displayGrid"

    states: [
        State {
            name: "displayGrid"
            PropertyChanges { target: background; color: "#555560" }
        },
        State {
            name: "displayPictures"
            PropertyChanges { target: background; color: "black" }
        }
    ]

    transitions: [
        Transition {
            from: "displayGrid"; to: "displayPicture"
            PropertyAnimation {
                target: backButton; properties: "visible"; to: true
```

```
            }
            NumberAnimation {
                target: grid
                properties: "scale"; to: 0.5
            }
            NumberAnimation {
                target: grid
                property: "opacity"; to: 0.0
                duration: screen.animDuration
                easing.type: Easing.InOutQuad
            }
            NumberAnimation {
                target: picture
                properties: "x"; to: 0
                duration: screen.animDuration
                easing.type: Easing.InOutQuad
            }
            NumberAnimation {
                target: backButton
                properties: "y"
                to: background.height - backButton.height - 20
                duration: screen.animDuration * 2
                easing.type: Easing.OutBounce
            }
        },
        Transition {
            from: "displayPicture"; to: "displayGrid"
            SequentialAnimation {
                ParallelAnimation {
                    NumberAnimation {
                        target: picture
                        properties: "x"; to: 2 * screen.width;
                        easing.type: Easing.InOutQuad
                    }
                    NumberAnimation {
                        target: backButton
                        properties: "y"
                        to: background.height + 3 * backButton.height
                        duration: screen.animDuration * 2
```

```
                        easing.type: Easing.InBack
                    }
                }
                PauseAnimation { duration: screen.animDuration / 2 }
                ParallelAnimation {
                    NumberAnimation {
                        target: grid
                        properties: "scale"; to: 1.0
                        duration: screen.animDuration
                        easing.type: Easing.InOutQuad
                    }
                    NumberAnimation {
                        target: grid
                        properties: "opacity"; to: 1.0
                        duration: screen.animDuration
                        easing.type: Easing.InOutQuad
                    }
                }
            }
        }
    ]
}
}
```

The RotationButton.qml source file is located at the top left. Clicking this button will rotate the view by -90 degrees. If you click this button while rotated -90 degrees, it will rotate back to its original state of 0 degrees.



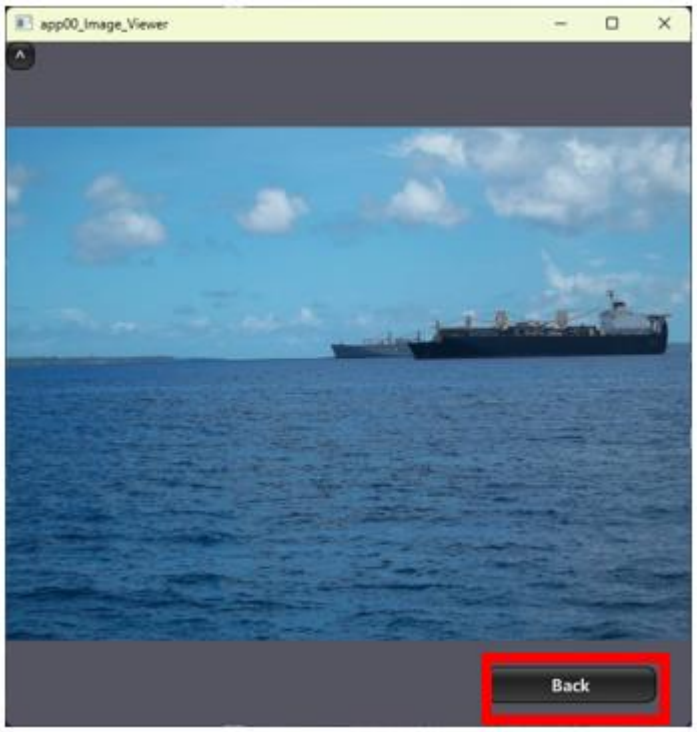
The following example is the source code for the RotationButton.qml example.

```
import QtQuick
```

```
Item {
    id: container; width: 50; height: 50

    property int angle: 0
    property int duration: 250

    Button {
        id: button
        text: "^"
        width: 25; height: 25;
        x: 0; y: 0
        onClicked: {
            container.angle = (container.angle == 0 ? -90 : 0)
        }

        states: [
            State {
                name: "normal"
                when: container.angle == 0
            },
            State {
                name: "rotated"
                when: container.angle == -90
            }
        ]

        state: "normal"

        transitions: [
            Transition {
                from: "normal"; to: "rotated"
                NumberAnimation {
                    targets: button
                    properties: "rotation"; to: -90
                    duration: 200
                }
            },
            Transition {
```

```
                from: "rotated"; to: "normal"
                NumberAnimation {
                    targets: button
                    properties: "rotation"; to: 0
                    duration: 200
                }
            }
        ]
    }
}
```

Button.qml returns to the Thumbnail.qml screen from the image enlargement screen.

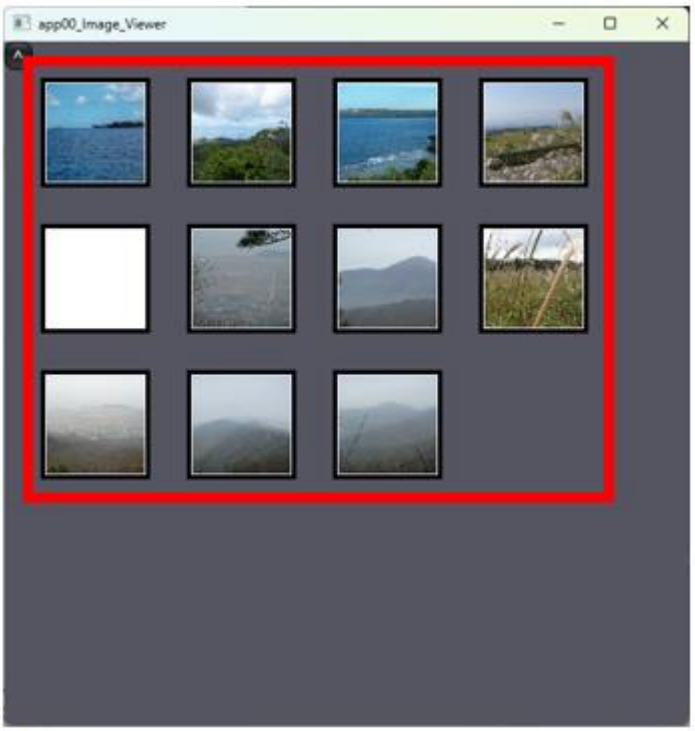


다음 예제는 Button.qml 예제 소스코드 이다.

```
import QtQuick

Item {
    id: container

    signal clicked
```

```
    property string text

    Rectangle {
        id: background
        anchors.fill: parent
        border.color: mouseRegion.pressed ? "gray" : "black"
        smooth: true
        radius: 10
        gradient: Gradient {
            GradientStop { position: 0.0; color: "#606060" }
            GradientStop { position: 0.33; color: "#202020" }
            GradientStop { position: 1.0; color: "#404040" }
        }
    }
    Text {
        color: "white"
        anchors.centerIn: background;
        font.bold: true;
        font.pixelSize: 15
        text: container.text; style: Text.Raised; styleColor: "black"
    }
    MouseArea {
        id: mouseRegion
        anchors.fill: parent
        onClicked: container.clicked()
    }
}
```

The Thumbnail.qml source file is what you see in the image below, and it's a way to display multiple images on a single screen by reducing the original image size.



The following example source code is the Thumbnail.qml source.

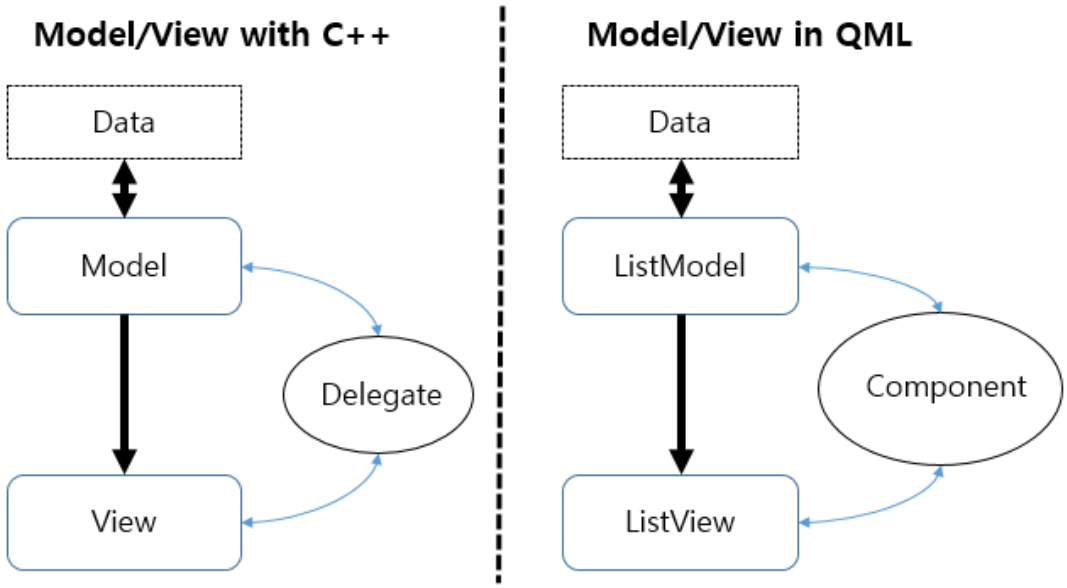
```
import QtQuick

Item {
    id: container

    signal clicked

    property int column
    property int row
    property string image

    width : 96
    height : 96

    x: 32 + column * (width + 32)
    y: 32 + row * (width + 32)

    Rectangle {
        color: "black"
```

```
        anchors.fill: parent
    }

    Rectangle {
        x: 4
        y: 4
        width: parent.width - 8
        height: parent.height - 8
        color: "white"
        smooth : true
    }

    Image {
        x: 5; y: 5
        width: parent.width - 10
        height: parent.height - 10

        source: "../" + container.image
        fillMode: Image.PreserveAspectCrop
    }

    MouseArea {
        anchors.fill: parent
        onClicked: parent.clicked()
    }
}
```

For the examples in this chapter, you can refer to the 00_Image_Viewer directory.

4. Model and View

QML uses Model/View in the same way that Qt/C++ uses Model/View. The following figure shows a comparison between the Model/View approach used in C++ and the Model/View approach in QML.



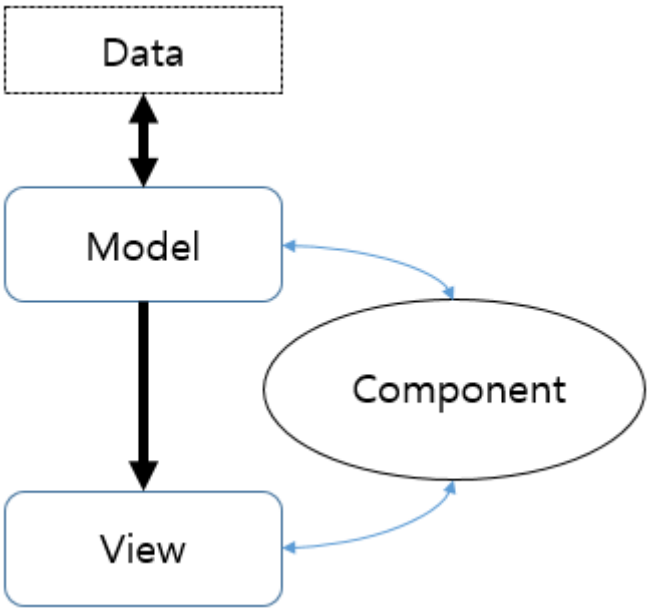
As shown in the figure above, the C++ and QML approaches are identical in concept and usage. In this chapter, we'll see how to use Model /View, and then we'll see how to use it in detail with a simple application example.

4.1. Representing data with model and view

Just as we use tables to represent large amounts of data, QML uses Model / View to represent large amounts of data in a tabular or list-like form to the user.

A Model can be thought of as a container for data. And a View is like a table or list-like tool that shows the user the container containing the data.

For example, in the Model / View concept, when you insert data, you don't insert it directly into the View. Also, when you modify data, you don't modify the data displayed in the View. The insertion/modification/deletion of data is done in the Model. The View is connected to the Model to show the data contained in the Model and to forward user events to the Model.



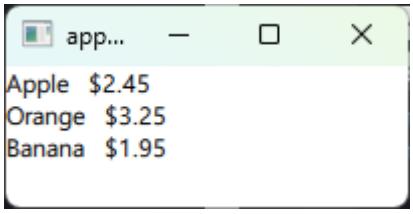
- ListModel and ListView

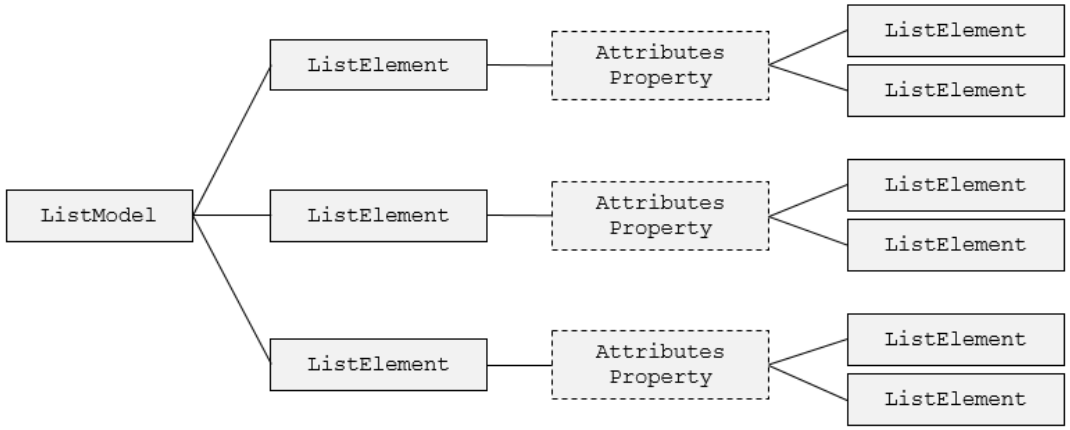
A ListModel provides the same functionality as a Container to hold the data you want to display in a ListView. A ListModel can have one or more ListElements. And between the

ListModel and the ListView, we use a Component. The Component can give the ListView a shape or style to represent the data. The following example uses a ListModel, a ListView, and a Component.

```
import QtQuick
import QtQuick.Window

Window {
    width: 200; height: 70; visible: true

    ListModel {
        id: fruitModel
        ListElement { name: "Apple";  cost: 2.45 }
        ListElement { name: "Orange"; cost: 3.25 }
        ListElement { name: "Banana"; cost: 1.95 }
    }
    Component {
        id: fruitDelegate
        Row {
            spacing: 10
            Text { text: name }
            Text { text: '$' + cost }
        }
    }
    ListView {
        anchors.fill: parent
        model: fruitModel
        delegate: fruitDelegate
    }
}
```

As shown in the example above, a ListModel can use one or more ListElements to add data.



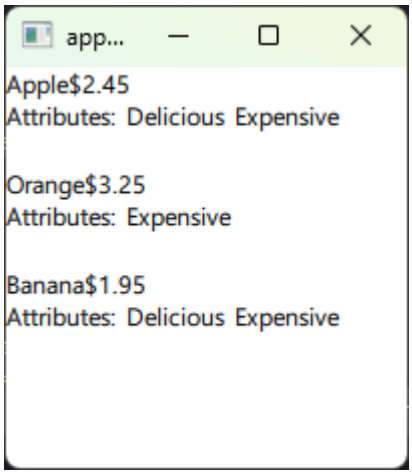


- How to Use the attributes Property of ListElement

The following example shows the use of attributes to use a ListElement as a child of a ListElement.

```
import QtQuick
import QtQuick.Window

Window {
    width: 200; height: 200; visible: true

    ListModel {
        id: fruitModel

        ListElement {
            name: "Apple"; cost: 2.45
            attributes: [
                ListElement { description: "Delicious" },
                ListElement { description: "Expensive" }
            ]
        }
        ListElement {
            name: "Orange"; cost: 3.25
            attributes: [
                ListElement { description: "Expensive" }
            ]
        }
        ListElement {
```

```
            name: "Banana"; cost: 1.95
            attributes: [
                ListElement { description: "Delicious" },
                ListElement { description: "Expensive" }
            ]
        }
    }

    Component {
        id: fruitDelegate
        Item {
            width: 200; height: 50
            Text { id: nameField; text: name }
            Text { text: '$' + cost; anchors.left: nameField.right }
            Row {
                anchors.top: nameField.bottom
                spacing: 5
                Text { text: "Attributes:" }
                Repeater {
                    model: attributes
                    Text { text: description }
                }
            }
        }
    }

    ListView {
        anchors.fill: parent
        model: fruitModel
        delegate: fruitDelegate
    }
}
```



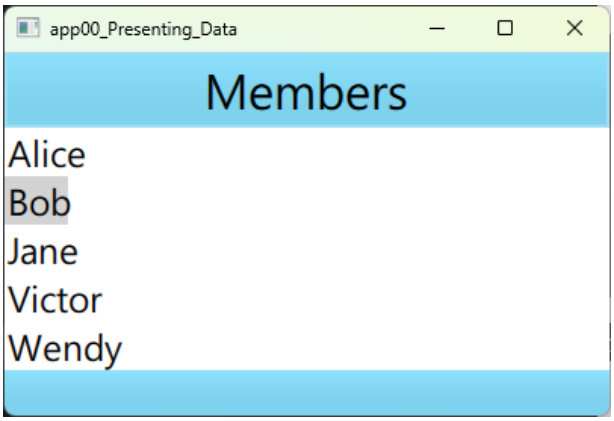
- ListView's header and footer properties

In a ListView, the header property can be used to style the top directly. footer can be used to style the bottom. The following example shows example source code using header and footer.

```
import QtQuick
import QtQuick.Window

Window {
    width: 400; height: 240; visible: true; color: "white"
    id: root

    ListModel {
        id: nameModel
        ListElement { name: "Alice"; }
        ListElement { name: "Bob";   }
        ListElement { name: "Jane";  }
        ListElement { name: "Victor"; }
        ListElement { name: "Wendy";  }
    }
    Component {
        id: nameDelegate
        Text {
            text: name;
            font.pixelSize: 24
            anchors.left: parent.left
```

```
            anchors.leftMargin: 2
        }
    }

    Component {
        id: bannercomponent
        Rectangle {
            id: banner
            width: root.width; height: 50
            gradient: clubcolors
            border {color: "#9EDDF2"; width: 2}
            Text {
                anchors.centerIn: parent
                text: "Members"
                font.pixelSize: 32
            }
        }
    }

    Gradient {
        id: clubcolors
        GradientStop { position: 0.0; color: "#8EE2FE"}
        GradientStop { position: 0.66; color: "#7ED2EE"}
    }

    ListView {
        anchors.fill: parent
        clip: true
        model: nameModel
        delegate: nameDelegate
        header: bannercomponent
        footer: Rectangle {
            width: parent.width; height: 30;
            gradient: clubcolors
        }
        highlight: Rectangle {
            color: "lightgray"
        }
        focus: true // Properties for moving menus with keyboard events
```

```
    }
}
```

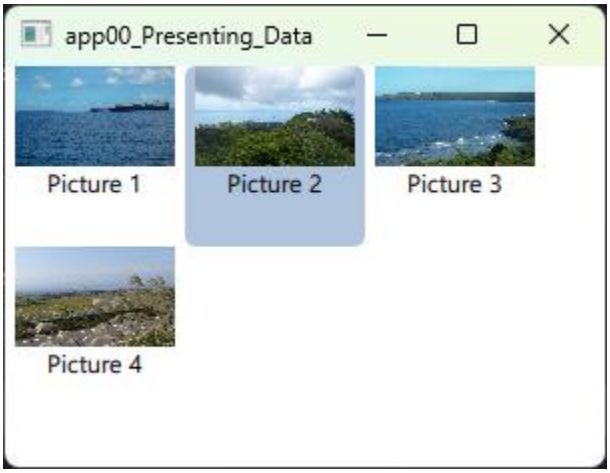


- GridView

A GridView can display ListElements in a Grid style. The following example shows an example using a GridView.

```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true

    ListModel {
        id: gridModel
        ListElement {
            name: "Picture 1"; frame: "images/101.JPG"
        }
        ListElement {
            name: "Picture 2"; frame: "images/102.JPG"
        }
        ListElement {
            name: "Picture 3"; frame: "images/103.JPG"
        }
        ListElement {
            name: "Picture 4"; frame: "images/104.JPG"
```

```
        }
    }

    Component {
        id: contactDelegate
        Item {
            width: grid.cellWidth; height: grid.cellHeight
            Column {
                anchors.fill: parent
                Image {
                    width: 80; height: 50
                    source: frame;
                    anchors.horizontalCenter: parent.horizontalCenter
                }
                Text {
                    text: name;
                    anchors.horizontalCenter: parent.horizontalCenter
                }
            }
        }
    }

    GridView {
        id: grid
        anchors.fill: parent
        cellWidth: 90; cellHeight: 90

        model: gridModel
        delegate: contactDelegate
        highlight:
            Rectangle {
                color: "lightsteelblue"; radius: 5
            }
        focus: true
    }
}
```



As you can see in the image above, when the example is run, you can see the inverted region move as you move the keyboard arrow keys.

- Using Animation in GridView

You can use Animation when you move the arrow keys on the keyboard. The following is an example of using Animation on a GridView.

```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true

    ListModel {
        id: gridModel
        ListElement {
            name: "Picture 1"; frame: "images/101.JPG"
        }
        ListElement {
            name: "Picture 2"; frame: "images/102.JPG"
        }
        ListElement {
            name: "Picture 3"; frame: "images/103.JPG"
        }
        ListElement {
```

```
        name: "Picture 4"; frame: "images/104.JPG"
    }
}

Component {
    id: highlight
    Rectangle {
        width: view.cellWidth; height: view.cellHeight
        color: "lightsteelblue"; radius: 5
        x: view.currentItem.x
        y: view.currentItem.y
        Behavior on x { SpringAnimation { spring: 3; damping: 0.2 } }
        Behavior on y { SpringAnimation { spring: 3; damping: 0.2 } }
    }
}

GridView {
    id: view
    width: 300; height: 200
    cellWidth: 80; cellHeight: 80

    model: gridModel
    delegate: Column {
        Image {
            width: 60; height: 50; source: frame;
            anchors.horizontalCenter: parent.horizontalCenter
        }
        Text {
            text: name;
            anchors.horizontalCenter: parent.horizontalCenter
        }
    }
    highlight: highlight

    // Set the current item's size to match
    highlightFollowsCurrentItem: true

    focus: true
}
```

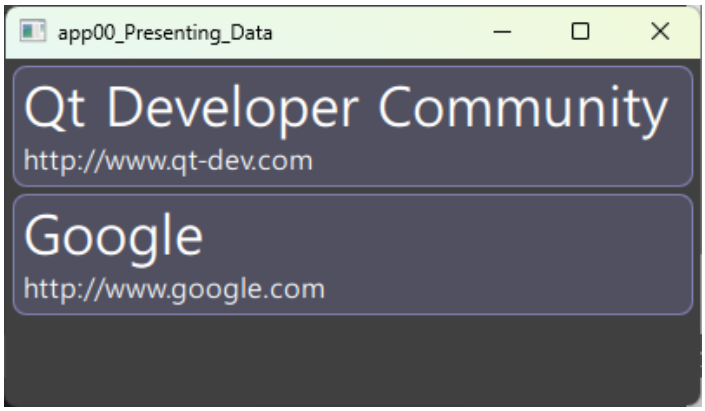
```
}
```

- **XML Model**

QML provides an XmlListModel to represent XML data. An XmlListModel can get XML data remotely by specifying an xml file using the source property or by specifying an Internet URL. The following example reads and displays data from an XML file.

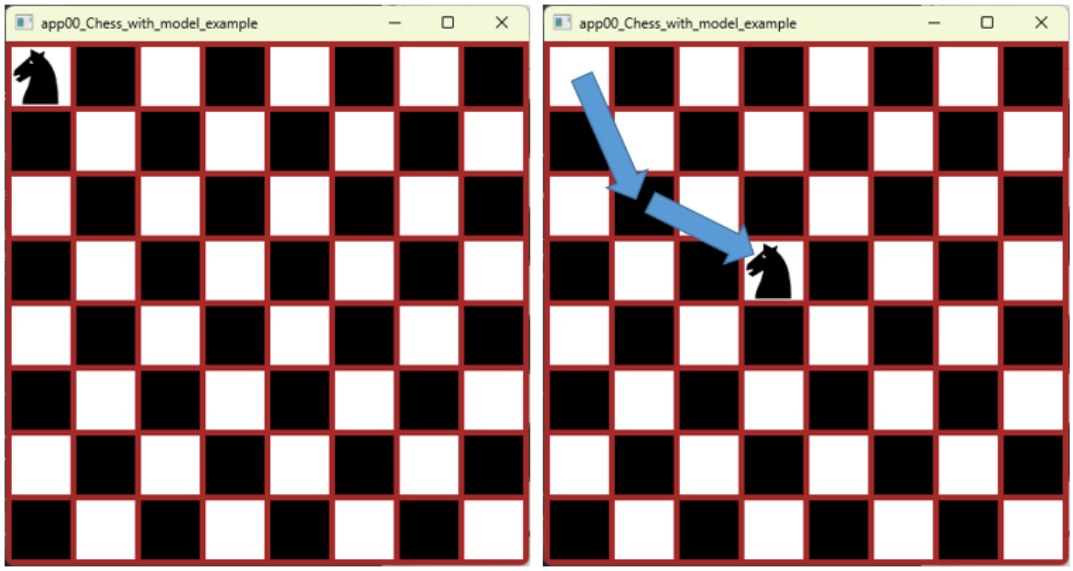
```
import QtQuick
import QtQuick.Window
import QtQml.XmlListModel

Window {
    width: 400; height: 200; visible: true; color: "#404040"

    XmlListModel {
            id: xmlModel
            source: "item.xml"
            query: "/items/item"

            XmlListModelRole { name: "title"; attributeName: "title" }
            XmlListModelRole { name: "link"; attributeName: "link" }
    }

    Component {
       id: xmlDelegate
       Item {
          width: parent.width; height: 74
          Rectangle {
             width: Math.max(childrenRect.width + 16, parent.width)
             height: 70; clip: true
             color: "#505060"; border.color: "#8080b0"; radius: 8
             Column {
                 Text {
                     x: 6; color: "white"
                     font.pixelSize: 32; text: title
                 }
                 Text {
                     x: 6; color: "white"
```

```
                    font.pixelSize: 16; text: link
                }
            }
        }
    }

    ListView {
        anchors.fill: parent; anchors.margins: 4; model: xmlModel
        delegate: xmlDelegate
    }
}
```



4.2. Chess Knight

In this chapter, we will implement chess using Repeater and Mode/View. The following figure shows how the Knight can move according to the rules of movement in chess.

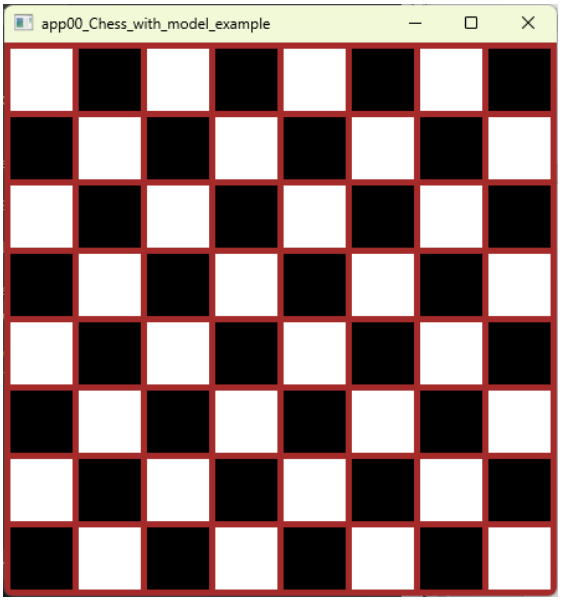


- **[Step 1] Implement Chess**

To implement the chess board shape, we'll use Repeater to create 8 rows by 8 columns. The following is an example source code that implements the Chess board shape.

```
import QtQuick
import QtQuick.Window

Window {
    width: 445; height: 445; color: "brown"; visible: true

    Grid {
        x: 5; y: 5
        rows: 8; columns: 8; spacing: 5

        Repeater {
            model: parent.rows * parent.columns
```

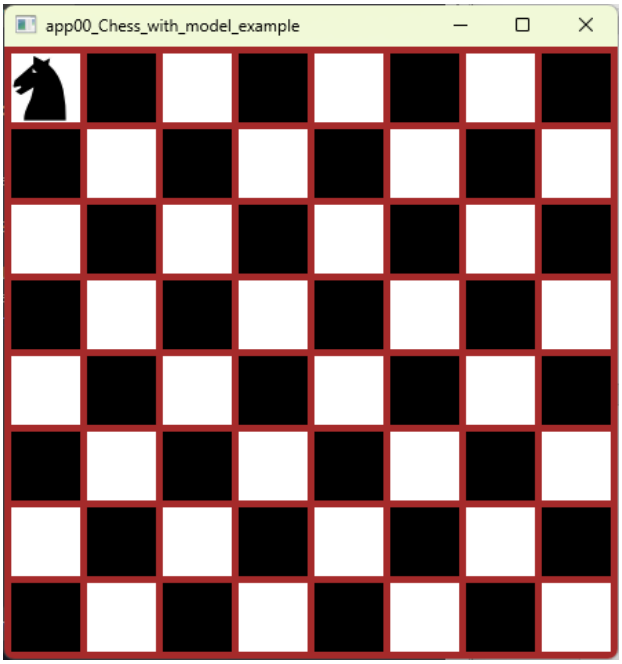
```
            Rectangle {
                width: 50; height: 50
                color: {
                    var row = Math.floor(index / 8);
                    var column = index % 8

                    // Even 1, otherwise 0
                    if ((row + column) % 2 == 1)
                        "black";
                    else
                        "white";
                }
            }
        }
    }
}
```



- **[Step 2] Implement Knight**

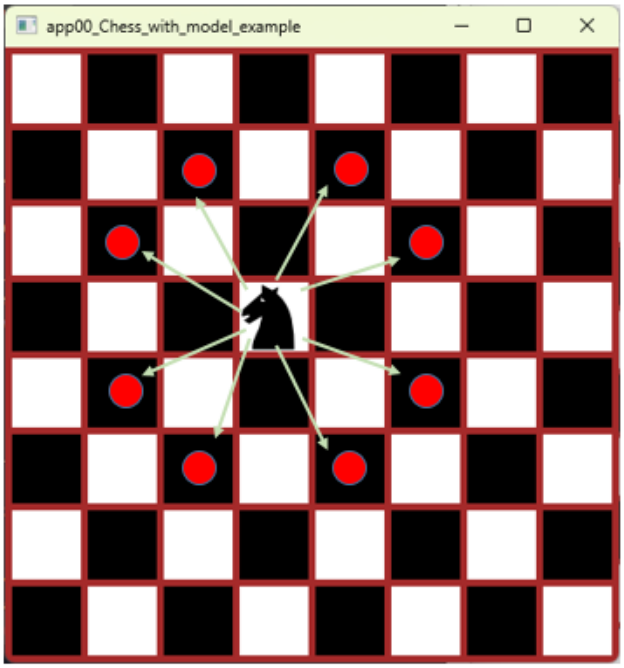
This time, let's place a Knight on a chessboard. To place the Knight on the chessboard, we use the Image type. The following example source code shows an example of how to place a Knight.

```
import QtQuick
import QtQuick.Window

Window {
    width: 445; height: 445; color: "brown"; visible: true

    Grid {
        x: 5; y: 5
        rows: 8; columns: 8; spacing: 5

        Repeater {
            model: parent.rows * parent.columns
            Rectangle {
                width: 50; height: 50
                color: {
                    var row = Math.floor(index / 8);
                    var column = index % 8
                    if ((row + column) % 2 == 1)
                        "black";
                    else
                        "white";
                }
            }
        }
    }

    Image {
        id: knight
        property int cx
        property int cy

        source: "./images/knight.png"

        x: 5 + 55 * cx
        y: 5 + 55 * cy
    }
}
```



- **[Step 3] Move the Knight**

In this step, we will implement the ability to move a placed Knight using the mouse. As shown in the following figure, the Knight can be moved 2 spaces forward and 1 space sideways, or 1 space sideways and 2 spaces forward, with a rule that allows it to move.



- **Calculate the values needed to implement rules and features that Knight can move to**
  - cx, cy is Knight's location
  - x, y is Knight's current location
  - Can be moved 2 spaces forward, 1 space sideways or 1 space sideways, 2 spaces forward
  - (x-cx) = 2 , (y-cy) =1 or (x-cx) =1 , (y-cy) = 2
  - We need to make sure that only positive values appear in the above calculation expression, as negative values can occur.

Based on the value calculations required for the above implementation, we could implement the following to determine if a location is possible for Knight to move to

```
if ((Math.abs(x - knight.cx) == 1 && Math.abs(y - knight.cy) == 2) ||
   (Math.abs(x - knight.cx) == 2 && Math.abs(y - knight.cy) == 1)) {
      knight.cx = x;
      knight.cy = y;
}
```

The following example source code is a complete example of applying the Knight movement rules from chess.

```
import QtQuick
import QtQuick.Window

Window {
    width: 445
    height: 445
    color: "brown"; visible: true

    Grid {
        x: 5; y: 5
        rows: 8; columns: 8; spacing: 5

        Repeater {
            model: parent.rows * parent.columns
            Rectangle {
```

```
            width: 50; height: 50
            color: {
                var row = Math.floor(index / 8);
                var column = index % 8
                if ((row + column) % 2 == 1)
                    "black";
                else
                    "white";
            }
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    var x = index % 8;
                    var y = Math.floor(index/8);

                    // Knight moves x, y = 2, 1 or x, y = 1, 2
                    if ((Math.abs(x - knight.cx) == 1 &&
                         Math.abs(y - knight.cy) == 2) ||
                        (Math.abs(x - knight.cx) == 2 &&
                         Math.abs(y - knight.cy) == 1)) {

                        knight.cx = x;
                        knight.cy = y;
                    }
                }
            }
        }
    }
}

Image {
    id: knight
    property int cx
    property int cy

    source: "./images/knight.png"

    x: 5 + 55 * cx
    y: 5 + 55 * cy
```

```
    }
}
```

For the examples covered in this chapter, see the 00_Chess_with_model_example directory.

5. Integration QML and C++

In this chapter, you will learn how to pass data and events between C++ and QML.

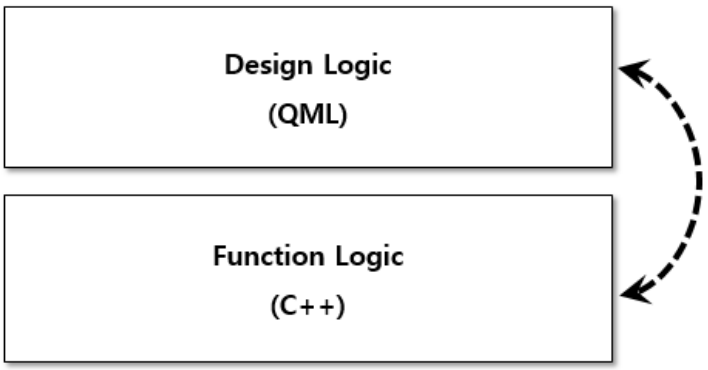
Interaction between C++ and QML Suppose you need to dynamically display data in QML. For example, when a new user joins a chat program, you need to display the new user's information in a GUI implemented in QML, and you need to pass the new user's information from C++ to QML.

In this case, you need to pass information from C++ to QML or from QML to C++. We will also see how a member function of a certain class in C++ should be called when a certain button is clicked on the QML.

So in this chapter, we'll look at how to pass data and events between QML and C++.

5.1. Overview

Interaction between QML and C++ refers to a way to pass data communication and events between QML and Qt as shown in the figure below. Design Logic (GUI) is implemented in QML and Function Logic is implemented in C++ to provide functions for interaction between QML and C++.



Qt Meta-Object System provides data communication between Design Logic written in QML and Function Logic written in C++.

Q_PROPERTY( ) is used to communicate properties of QML types with C++ as follows. Q_INVOKABLE is used to call member functions of C++ classes from QML.

And when an event occurs in QML, you can execute the SLOT function defined in C++ as an accessor public slot.

- **Implementing functionality for communicating QML-type properties to C++**

```
Q_PROPERTY(...)
```

- **Calling Member Functions of a C++ Class from a QML Type**

```
Q_INVVOKABLE
```

- **Calling a C++ Slot function when an event occurs in QML**

```
public slots:
```

```
    void refresh()
```

There are four cases for passing data communication events between QML and C++, which can be summarized as follows

① Calling a C++ function on a QML type to get the resulting value

```
Rectangle
{
    width : 300; height: 300
    Text {
        text: (Calling a function that returns a QString implemented in C++ code)
        ...
    }
...
```

② Calling a C++ function with a specific value from a QML type to get a return value

```
Text {
      ...
      Component.onCompleted:
      {
          // Call the author(QString str) function in C++ code
          msg.author = "Hello"
      }
  }
```

③ Calling the Slot function implemented in C++ when an event occurs in QML

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        var str = "Who are you ?"

        // In C++ code, public slots
        // Call the SLOT function defined by the accessor
        var result = msg.postMessage(str)
```

```
        ...
}
```

④ Calling a member function of a C++ class from QML

```
MouseArea
{
    anchors.fill: parent
    onClicked:
   {
        ...
        msg.refresh(); // Calling public's defined function in C++ code
    }
}
```

- **Syntax of Q_PROPERTY**

Q_PROPERTY is used in C++ and is used to implement functions for communicating QML-type properties with C++. The syntax of Q_PROPERTY is as follows.

```
Q_PROPERTY( type name
            (READ getFunction [WRITE setFunction] ¦
            MEMBER memberName [(READ getFunction ¦ WRITE setFunction)])
            [RESET resetFunction]
            [NOTIFY notifySignal]
            [REVISION int]
            [DESIGNABLE bool]
            [SCRIPTABLE bool]
            [STORED bool]
            [USER bool]
            [CONSTANT]
            [FINAL] )
```

For example, suppose you used Q_PROPERTY as follows.

```
Q_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)
```

READ specifies the name of a function that returns the value of author, which is a QString in C++. WRITE assigns the value of the author variable, which is a QString value. For example, the function setAuthor(QString author) defines a function that sets the value to the value of the first argument. NOTIFY specifies a SIGNAL function. In C++, the declaration of a Q_PROPERTY is done in the following places

```
#include <QObject>

class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY( QString author
                READ author
                WRITE setAuthor
                NOTIFY authorChanged)
...
```

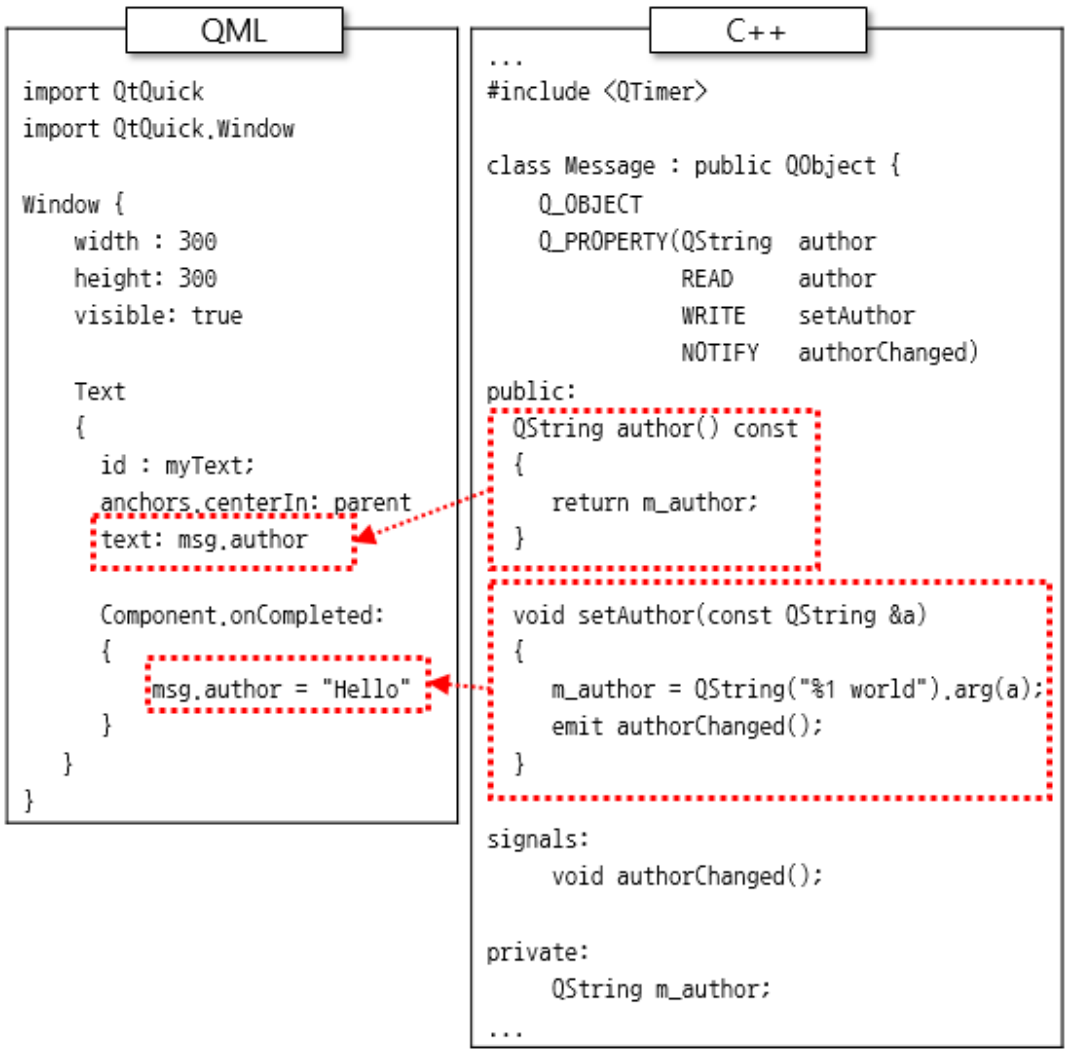
- **Q_INVOKABLE**

Q_INVOKABLE is used in QML as a way to call member functions that are declared as public in C++ in the Access Modifier. The following is an example source code using Q_INVOKABLE.

```
#include <QObject>

class Message : public QObject
{
    Q_OBJECT
    ...
public:
    Q_INVOKABLE bool postMessage(const QString &msg)
    {
        ...
        return true;
    }
```

- Calling the value of a specific property of a QML type (Exporting)

To export the value of a specific property of a QML type, you can call a member function of a C++ class by defining it using Q_PROPERTY( ) provided by the QObject class. For example, to assign the value of the text property of a QML type of type Text, you can use the following code.



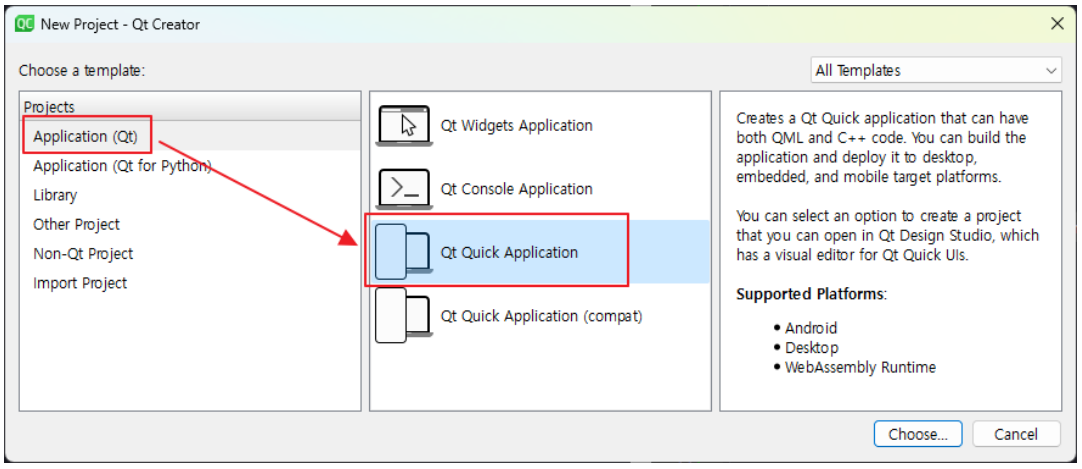
As shown in the source code above, you can use the QQuickView class to call member functions of the C++ Message class in QML. For example, the QQuickView class can be used as follows.

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

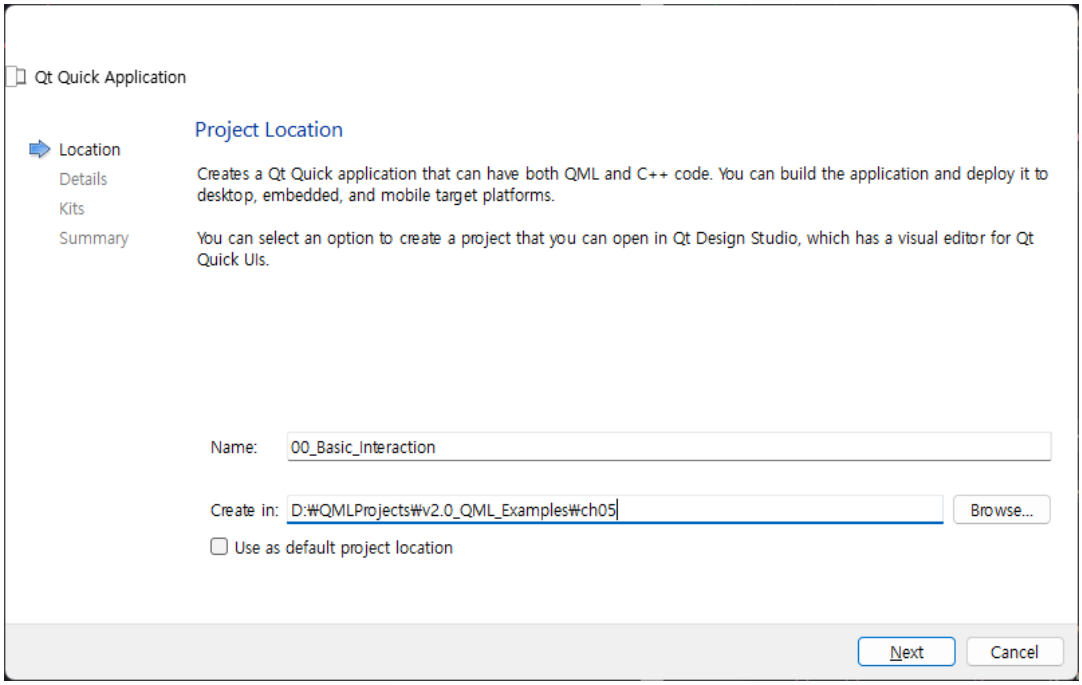
```
#include "message.h"

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

    QQmlApplicationEngine engine;

    Message msg;
    engine.rootContext()->setContextProperty("msg", &msg);

    const QUrl url("qrc:/00_Basic_Interaction/Main.qml");
    engine.load(url);

    return app.exec();
}
```

- **Examples for Interacting between QML and C++**

In this example, let's take a look at an example of a winning interaction between QML and C++. Let's set the value of the text property of type Text in QML to the return value of the author( ) member function of the C++ class as described in the example source code above. Create a project as shown in the following figure.



Select [Qt Quick Application] as shown in the image above and click the [Choose...] button at the bottom.



Specify a project name and the directory where the project will be located, as shown in the image above. Once the project creation is complete, add the Message class to the project and create the message.h header file as shown below.

```
#ifndef MESSAGE_H
#define MESSAGE_H

#include <QObject>

class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString  author
               READ     author
               WRITE    setAuthor
               NOTIFY   authorChanged)
public:
    explicit Message(QObject *parent = nullptr);

    void setAuthor(const QString &a)
    {
        m_author = QString("%1 world.").arg(a);
```

```
        emit authorChanged();
    }

    QString author() const
    {
        return m_author;
    }

signals:
    void authorChanged();

private:
    QString m_author;

};

#endif // MESSAGE_H
```

Next, we don't need to write anything in the message.cpp source code file, so we only need to see the constructor function as shown below.

```
#include "message.h"

Message::Message(QObject *parent)
    : QObject{parent}
{
}
```

Next, create the Main.qml file as shown below.

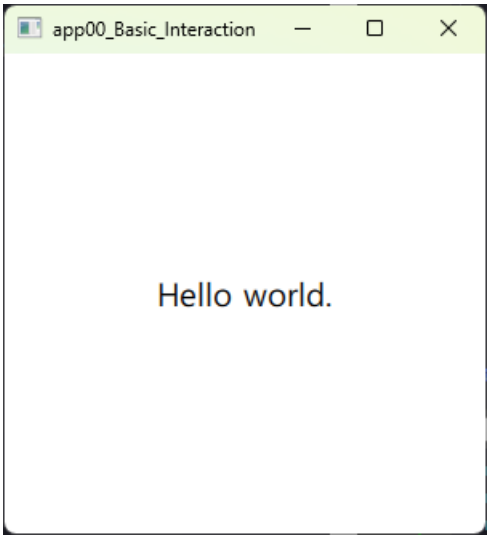
```
import QtQuick
import QtQuick.Window

Window {
    width : 300; height: 300; visible: true
    Text {
        id : myText;
        anchors.centerIn: parent
        text: msg.author
        font.pixelSize: 20
```

```
        Component.onCompleted: {
            msg.author = "Hello"
            myText.text = msg.author
        }
    }
}
```

Next, to enable communication between QML and C++, write the main.cpp file as follows.

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

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

    QQmlApplicationEngine engine;

    Message msg;
    engine.rootContext()->setContextProperty("msg", &msg);

    const QUrl url("qrc:/00_Basic_Interaction/Main.qml");
    engine.load(url);

    return app.exec();
}
```

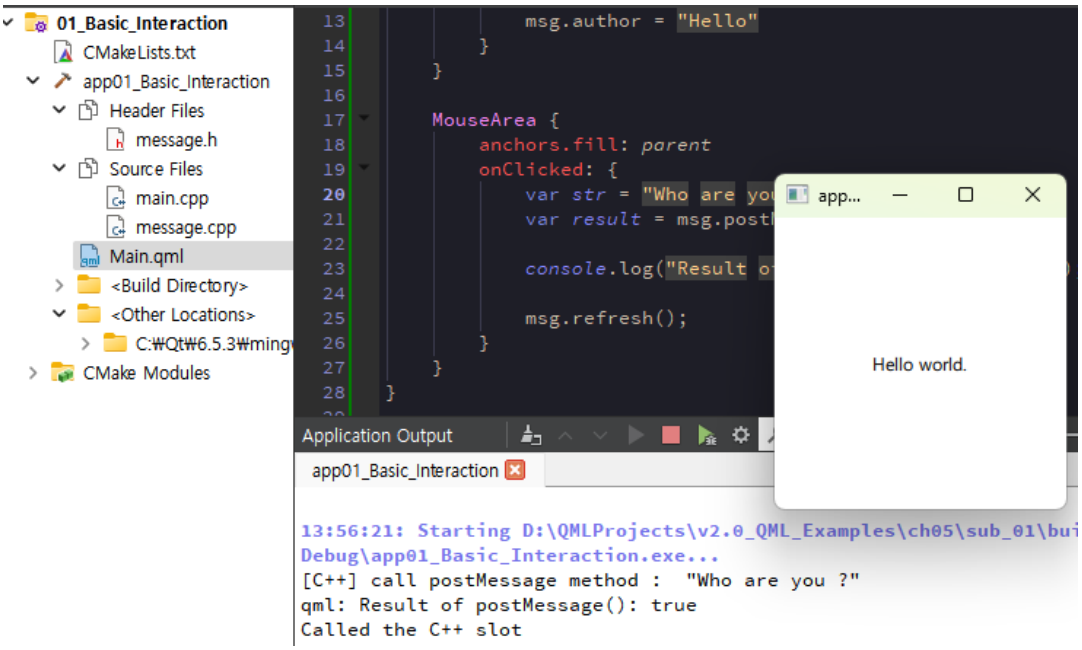
Once you've written the source code as above, let's build and run the project.



You can find the source code for this example in the 00_Basic_Interaction directory.

- **Example of using the Q_INVOKABLE function**

In this example, we will write an example in QML that executes the Slot function defined by Q_INVOKABLE of the C++ Message class when the mouse is clicked. When you click on the screen, it prints a message to the console debugging window, as shown in the following figure.



For this project, create the project in the same way as the previous project. Then, add the Message class. Once you've added the Message class, write the message.h header file like this

```
#ifndef MESSAGE_H
#define MESSAGE_H

#include <QObject>
#include <QDebug>

class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString  author
               READ     author
               WRITE    setAuthor
               NOTIFY   authorChanged)
public:
    explicit Message(QObject *parent = nullptr);
    Q_INVOKABLE bool postMessage(const QString &msg) {
        qDebug() << "[C++] call postMessage method : "
                 << msg;

        return true;
    }

    void setAuthor(const QString &a) {
        m_author = QString("%1 world.").arg(a);
        emit authorChanged();
    }

    QString author() const {
        return m_author;
    }

signals:
    void authorChanged();

private:
```

```
    QString m_author;

public slots:
    void refresh() {
        qDebug() << "Called the C++ slot";
    }
};

#endif // MESSAGE_H
```

Next, the message.cpp source code only has a constructor function. So, make sure it looks like this

```
#include "message.h"

Message::Message(QObject *parent)
    : QObject{parent}
{
}
```

Next, create the Main.qml file as shown below.

```
import QtQuick
import QtQuick.Window

Window {
    width : 200; height: 200; visible: true

    Text {
        id : myText;
        anchors.centerIn: parent
        text: msg.author

        Component.onCompleted: {
            msg.author = "Hello"
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
```

```
            var str = "Who are you ?"
            var result = msg.postMessage(str)
            console.log("Result of postMessage():", result);

            msg.refresh();
        }
    }
}
```

Next, open the main.cpp file and write something like this

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

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

    QQmlApplicationEngine engine;
    Message msg;

    engine.rootContext()->setContextProperty("msg", &msg);

    const QUrl url("qrc:/01_Basic_Interaction/Main.qml");
    engine.load(url);

    return app.exec();
}
```

Let's write the source code as shown in the example above and run the example. You can see that the onClicked property of the MouseArea type in QML executes the function defined by Q_INVOKABLE in C++.

You can find the source code for this example in the 01_Basic_Interaction directory.

5.2. Implementing QML Type in C++

In this chapter, we will implement QML types using C++. To use a QML type implemented in C++ in QML, you can use the qmlRegisterType( ) function, and you can use it in two ways as shown below.

```
template<typename T>
int qmlRegisterType(const char * uri, int versionMajor, int versionMinor,
                    const char * qmlName);

template<typename T, int metaObjectRevision>
int qmlRegisterType(const char *uri, int versionMajor, int versionMinor,
                    const char *qmlName);
```

The following is an example source code of how to use the qmlRegisterType( ) function. The first argument specifies the name to import when importing QML. The second argument is the major version, the third is the minor version, and the fourth is the type name to be used in the QML.

```
qmlRegisterType<Message>( "Message", 1, 0, "Msg" );
```

And to define a QML module called Message as a C++ class, we can implement it as follows.

```
class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)
    ...
```

The following example can be used in QML as shown in the following example, assuming that you have implemented a QML module called Message as above.

```
import QtQuick
import Message 1.0

Window {
    width : 300; height: 300
```

```
    ...
    Msg {
        ...
    }
...
```

The second argument when using the qmlRegisterType( ) function is used to specify the Revision version. The following is an example of using the Revision version

```
qmlRegisterType<Message, 1>( " Message", 1, 1, " Msg")
```

If you used the revision as above, you need to add the revision information in Q_PROPERTY.

```
class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString author READ author WRITE setAuthor
               NOTIFY authorChanged REVISION 1)
signals:
    Q_REVISION(1) void authorChanged();
...
```

In QML, this can be implemented in the form of subtypes existing inside a newly implemented QML type.

```
MessageBoard
{
    Message { author: "Eddy" }
    Message { author: "Candy" }
}
```

In addition to the above, it can also be used in the following ways

```
MessageBoard
{
    messages: [
        Message { author: "Eddy" },
        Message { author: "Candy" }
    ]
}
```

To use it as shown in the two QML examples above, you can use the Q_CLASSINFO( ) macro and the QQmlListProperty( ) function.

```
class MessageBoard : public QObject {
    Q_OBJECT
    Q_PROPERTY(QQmlListProperty<Message> messages READ messages)
    Q_CLASSINFO("DefaultProperty", "messages")

public:
    QQmlListProperty<Message> messages() const;

private:
    QList<Message *> messages;
};
```

- Example of implementing a custom QML type using C++ class

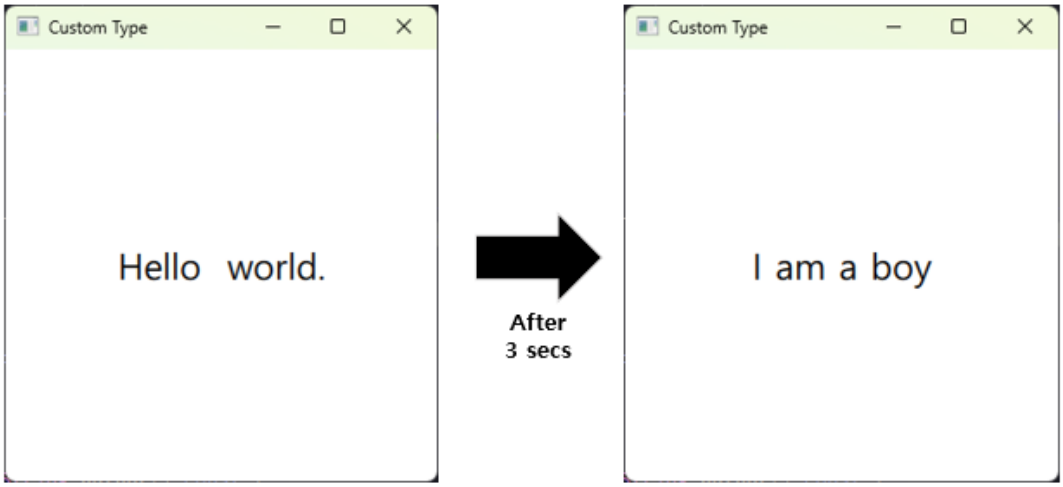
In this example, we will directly implement a new type for use in QML. The module we will implement will be named Message. The following example shows how to run a program and have the following Slot function called by the QTimer defined in the Message class after 3 seconds.

```
void timerTimeout()
{
   emit newMessagePosted("I am a boy");
}
```

When the Slot function above is called, the Slot property of the QML called onNewMessagePosted of type Msg is called, as shown below. The following is part of the example QML

```
...
import Message 1.0

Window {
    ...
    Text {
        ...
        Msg {
```

```
            id: myMsg
            onNewMessagePosted: {
                console.log("[QML] New Message received: ", subject);
                myText.font.pixelSize = 25
                myText.text = subject;
            }
        }
    }
```



As you can see in the image above, the string "Hello world" printed in the window is changed to "I am a boy". The following source code is the header file source code for the Message class.

```
#ifndef MESSAGE_H
#define MESSAGE_H

#include <QObject>
#include <QTimer>
#include <QDebug>

class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString  author
               READ     author
               WRITE    setAuthor
```

```
               NOTIFY   authorChanged)
public:
    explicit Message(QObject *parent = nullptr) {
        QTimer::singleShot(3000, this, SLOT(timerTimeout()));
    }

    Q_INVOKABLE bool postMessage(const QString &msg) {
        qDebug() << "[C++ Layer] call postMessage method : "
                 << msg;
        return true;
    }

    void setAuthor(const QString &a) {
        m_author = QString("%1 world.").arg(a);
        emit authorChanged();
    }

    QString author() const {
        return m_author;
    }

signals:
    void authorChanged();
    void newMessagePosted(const QString &subject);

private:
    QString m_author;

public slots:
    void refresh() {
        qDebug() << "Called the C++ slot";
    }

    void timerTimeout() {
        emit newMessagePosted("I am a boy");
    }

};
```

```
#endif // MESSAGE_H
```

Next, write Main.qml as shown below.

```
import QtQuick
import QtQuick.Controls
import Message 1.0

Window {
    width : 300; height: 300; visible: true
    title: "Custom Type"

    Text {
        id : myText;
        anchors.centerIn: parent
        text: myMsg.author
        font.pixelSize: 25
        Component.onCompleted: {
            myMsg.author = "Hello "
        }

        Msg {
            id: myMsg
            onNewMessagePosted: function(subject) {
                console.log("Message received: ", subject);
                myText.font.pixelSize = 25
                myText.text = subject;
            }
        }
    }
}
```

Next, create the main.cpp source code file as shown below.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "message.h"

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

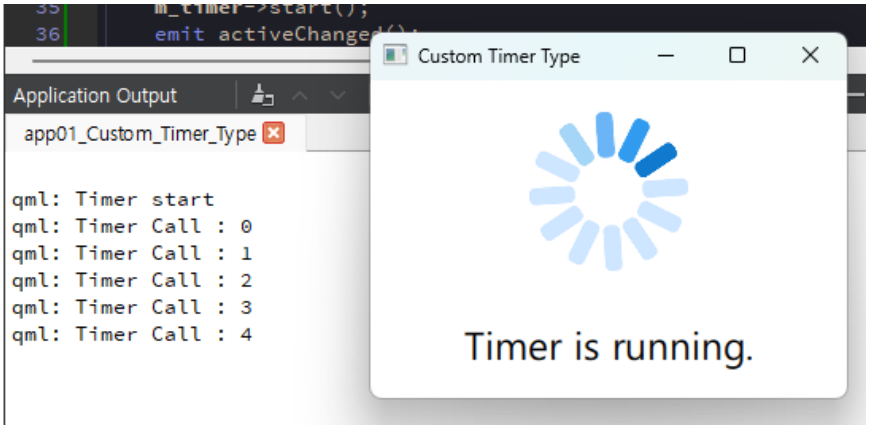
```
    qmlRegisterType<Message>( "Message", 1, 0, "Msg" );

    QQmlApplicationEngine engine;

    const QUrl url("qrc:/00_Custom_Type_Basic/Main.qml");
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreationFailed,
        &app, []() { QCoreApplication::exit(-1); },
        Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

You can find the source code for this example in the 00_Custom_Type_Basic directory.

- Timer type implementation example

This is an example of implementing a timer that works in QML. The following figure shows the example running.

As shown in the image above, the timer is stopped when the QML is loaded. Clicking on the QML area starts the timer. Then, the debugging window (Application Output) of the Qt Creator IDE tool prints a debugging message as shown below.



The images used in the project are located in the example source code directory. The example source code directory is located in a directory called images inside the 01_Custom_Timer_Type directory.

Add a class called Timer to your project, and then create a timer.h header file like this

```
#ifndef TIMER_H
#define TIMER_H

#include <QObject>
#include <QTimer>

class Timer : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int      interval
               READ     interval
               WRITE    setInterval
               NOTIFY   intervalChanged )

    Q_PROPERTY(bool     active
               READ     isActive
               NOTIFY   activeChanged )

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

    void setInterval( int msec );
```

```
    int interval();

    bool isActive();

public slots:
    void start();
    void stop();

signals:
    void timeout();
    void intervalChanged();
    void activeChanged();

private:
    QTimer* m_timer;
};

#endif // TIMER_H
```

다음은 timer.cpp 소스코드 파일이다. 아래에서 보는 것과 같이 소스코드를 작성한다.

```
#include <QTimer>
#include <QDebug>
#include "timer.h"

Timer::Timer( QObject* parent ) : QObject( parent ),
    m_timer( new QTimer( this ) )
{
    connect( m_timer, SIGNAL(timeout()), this, SIGNAL(timeout()));
}

void Timer::setInterval( int msec )
{
    if ( m_timer->interval() == msec )
        return;
    m_timer->setInterval(msec);
    emit intervalChanged();
}

int Timer::interval()
```

```
{
    return m_timer->interval();
}

bool Timer::isActive()
{
    return m_timer->isActive();
}

void Timer::start()
{
    if ( m_timer->isActive() )
        return;
    m_timer->start();
    emit activeChanged();
}

void Timer::stop()
{
    if ( !m_timer->isActive() )
        return;
    m_timer->stop();
    emit activeChanged();
}
```

Next, write the main.cpp file as shown below.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "timer.h"

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

    qmlRegisterType<Timer>("MyCustomTimer", 1, 0, "MyTimer");

    QQmlApplicationEngine engine;

    const QUrl url("qrc:/01_Custom_Timer_Type/Main.qml");
```

```
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreationFailed,
        &app, []() { QCoreApplication::exit(-1); },
        Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

In the example source code above, the first argument to the qmlRegisterType( ) function, "MyCustomTimer", is used as the name of the QML module to import from QML.

The second and third arguments are the version information and finally the fourth argument, "MyTimer", is the name of the type used by QML. Here is the Main.qml file. Create it as shown below.

```
import QtQuick
import QtQuick.Controls
import MyCustomTimer 1.0

Window {
    width: 300; height: 200; visible: true
    title: "Custom Timer Type"

    property int timerCnt: 0

    Image {
        id: loadImage
        source: "images/loading.png"
        width:100; height: 100
        anchors.top: parent.top
        anchors.topMargin: 20
        anchors.horizontalCenter: parent.horizontalCenter
    }

    PropertyAnimation {
        id: loadAni
        target: loadImage
        loops: Animation.Infinite
        from: 0;
        to: 360
```

```
        property: "rotation"
        duration: 2000
        running: false
    }

    Text {
        text: timer.active ? "Timer is running." : "Timer stopped."

        font.pixelSize: 24
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.top: loadImage.bottom
        anchors.topMargin: 30
    }

    MyTimer {
        id: timer
        interval: 1000
        onTimeout: {
            console.log( "Timer Call :", timerCnt++);
        }
    }

    MouseArea {
        anchors.fill: parent

        onClicked: {
            if ( timer.active == false ) {
                console.log( "Timer start" );
                timer.start();
                loadAni.start();
            } else {
                console.log( "Timer stop" );
                timer.stop();
                loadAni.stop();
            }
        }
    }
}
```

In the QML source code above, onTimeout is a Signal property. This property is fired

when the timeout( ) signal is called in the C++ Timer class.

And the start( ) and stop( ) functions of the Slot function in the C++ Timer class are mapped to the timer.start( ) and timer.stop( ) functions in the QML source code above.

For example, when timer.start( ) is called in QML, the start( ) member function is called in the C++ Timer class.

5.3. How to use QQuickPaintedItem class in QML

The QQuickPaintedItem class provides the same functionality as the QPainter class. The difference is that the QPainter class is used for the purpose of displaying 2D graphic elements (lines, lines, circles, curves, gradients, image displays, etc.) within the GUI window area implemented by the QWidget class.

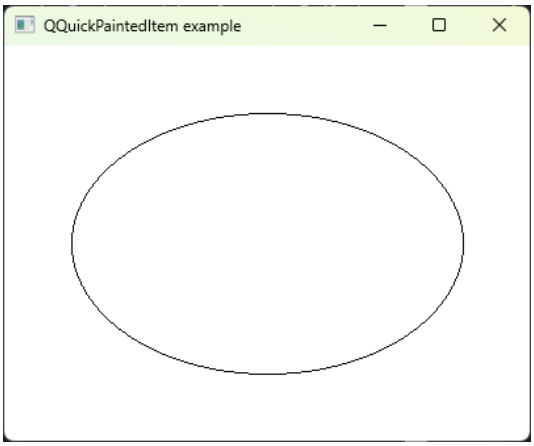
However, the QQuickPaintedItem class can display 2D graphic elements within a QML type area. Therefore, you can display the result of a 2D graphic drawn with the QQuickPaintedItem class within a QML type area.

QML

```
import QtQuick
import Shapes 1.0

Item {
    width: 300; height: 200

    Ellipse
    {
        x: 50; y: 50
        width: 200; height: 100
    }
}
```

C++

```
#include <QQuickPaintedItem>

class EllipseItem : public QQuickPaintedItem
{
    Q_OBJECT

public:
    EllipseItem( QQuickItem *parent = 0 );
    void paint(QPainter *painter);
};
```



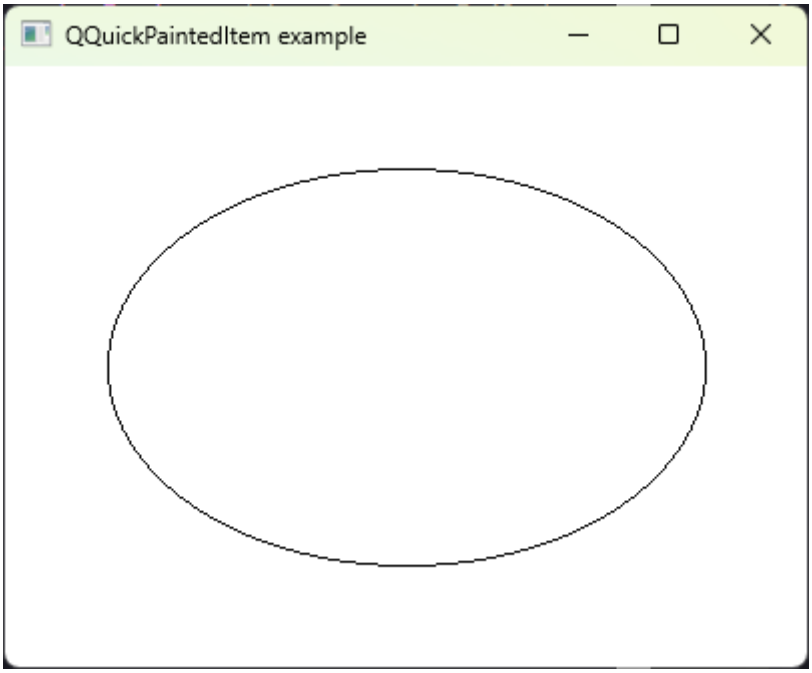
The figure above is an example run screen. To display the ellipse as shown in the example run screen, the result of drawing an ellipse in the paint( ) member function of the

EllipseItem class in the example source code on the left is displayed in the Ellipse type area of QML.

QQuickPaintedItem 클래스는 QQuickItem 클래스로부터 상속 받는다. QQuickItem 클래스는 Qt Quick 에서 제공하는 모든 그래픽 표시 관련 QML 타입을 구현하는 데 부모 클래스로 사용된다. QQuickItem 클래스는 그래픽 표시 관련된 것 이외에도 키보드, 터치, 마우스 등 이벤트를 처리 할 수 있다.

- **Example using the QQuickPaintedItem class**

In this example, we will implement the example from the above example. In the QQuickPaintedItem class, you can display the result of drawing an ellipse, a 2D graphic element, in a QML area.



We will use the Qt C++ class QQuickPaintedItem to draw an ellipse, as shown in the example run screen above.

After creating the project, add the EllipseItem class to the project. Then open the ellipseItem.h header file and write the following code

```
#ifndef ELLIPSEITEM_H
#define ELLIPSEITEM_H
```

```
#include <QQuickPaintedItem>

class EllipseItem : public QQuickPaintedItem
{
    Q_OBJECT
public:
    EllipseItem(QQuickItem *parent = nullptr);
    void paint(QPainter *painter);
};

#endif
```

As you can see in the example source code above, the paint( ) function is the member function that implements the source code to draw an ellipse, which is a 2D graph element.

Next, open the ellipseItem.cpp source code file and write the following code.

```
#include "ellipseitem.h"
#include <QPainter>

EllipseItem::EllipseItem(QQuickItem *parent)
    : QQuickPaintedItem(parent)
{
}

void EllipseItem::paint(QPainter *painter)
{
    const qreal halfPenWidth = qMax(painter->pen().width() / 2.0, 1.0);

    QRectF rect = boundingRect();

    rect.adjust(halfPenWidth,
                halfPenWidth,
                -halfPenWidth,
                -halfPenWidth);

    painter->drawEllipse(rect);
}
```

Next, open the Main.qml file and write the following code

```
import QtQuick
import QtQuick.Window
import Shapes 1.0

Window {
    width: 400
    height: 300
    visible: true
    title: "QQuickPaintedItem example"

    Ellipse {
        anchors.centerIn: parent
        width: 300
        height: 200
    }
}
```

The following example source code is from main.cpp. In main.cpp, qmlRegisterType( ) is used to register the EllipseItem class to be available as an Ellipse QML type in QML.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "ellipseitem.h"

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

    qmlRegisterType<EllipseItem>("Shapes", 1, 0, "Ellipse");

    QQmlApplicationEngine engine;
    const QUrl url("qrc:/00_QQuickPaintedItem_Example/Main.qml");
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreationFailed,
        &app, []() { QCoreApplication::exit(-1); },
        Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

You can find the source code for this example in the 00_QQuickPaintedItem_Example directory.

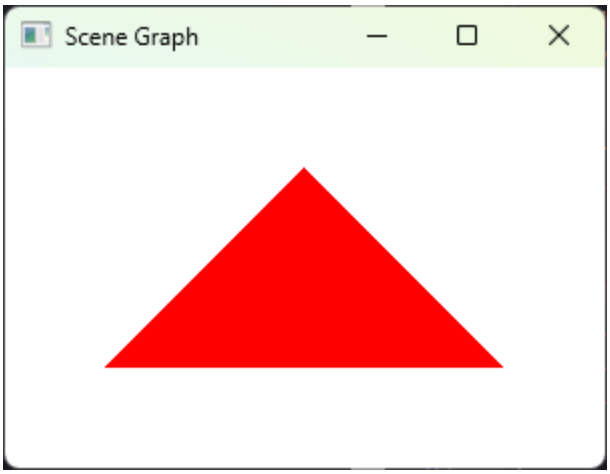
5.4. Scene Graph

The Scene Graph provides the same functionality as the QPainter provided by Qt C++. To use the Scene Graph, you can use the QSGNode class, which is implemented by inheriting from the QQuickItem class. This class provides the same functionality as the QGraphicsItem class provided by the Qt Graphics View Framework.

The Scene Graph class uses OpenGL ES 2.0 or OpenGL 2.0 by default. In addition, the QSGNode class provides classes such as QSGGeometry, QSGMaterial(Texture), and others.

- Triangle example using the Scene Graph class

In this example, we will use the QSGNode, QSGGeometry, and QSGFlatColorMaterial classes to display a 2D graph element, a Triangle.



The QSGNode class is the class that manages all graph units, or Nodes, in the Scene Graph. The QSGGeometry class is the class that stores geometry information to be rendered on the Scene Graph. The QSGFlatColorMaterial class is the class used to represent Color within the Scene Graph.

Create a new project and add the TriangleItem class, and then create the triangleitem.h header file as shown below.

```
#ifndef TRIANGLEITEM_H
#define TRIANGLEITEM_H
```

```
#include <QQuickItem>
#include <QSGGeometry>
#include <QSGFlatColorMaterial>

class TriangleItem : public QQuickItem
{
    Q_OBJECT
    QML_ELEMENT
public:
    TriangleItem(QQuickItem *parent = nullptr);

protected:
    QSGNode *updatePaintNode(QSGNode *node, UpdatePaintNodeData *data);

private:
    QSGGeometry m_geometry;
    QSGFlatColorMaterial m_material;
};

#endif // TRIANGLEITEM_H
```

다음으로 triangleitem.cpp 소스코드 파일을 아래와 같이 작성한다.

```
#include "triangleitem.h"
#include <QSGGeometryNode>

TriangleItem::TriangleItem(QQuickItem *parent)
    : QQuickItem(parent),
        m_geometry(QSGGeometry::defaultAttributes_Point2D(), 3)
{
    setFlag(ItemHasContents);
    m_material.setColor(Qt::red);
}

QSGNode *TriangleItem::updatePaintNode(QSGNode *n, UpdatePaintNodeData *)
{
    QSGGeometryNode *node = static_cast<QSGGeometryNode *>(n);
    if (!node) {
        node = new QSGGeometryNode();
```

```
    }

    QSGGeometry::Point2D *v = m_geometry.vertexDataAsPoint2D();
    const QRectF rect = boundingRect();
    v[0].x = (float)rect.left();
    v[0].y = (float)rect.bottom();

    v[1].x = (float)rect.left() + (float)rect.width()/2;
    v[1].y = (float)rect.top();

    v[2].x = (float)rect.right();
    v[2].y = (float)rect.bottom();

    node->setGeometry(&m_geometry);
    node->setMaterial(&m_material);

    return node;
}
```

The ItemHasContents ENUM value specified in the setFlag() function is the Flag ENUM value provided by QQuickItem.

The m_material object of the QSGFlatColorMaterial class contained in the Scene Graph can be colored using the setColor( ) member function. Next, open the main.cpp file and write the following code.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "triangleitem.h"

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

    qmlRegisterType<TriangleItem>("Shapes", 1, 0, "Triangle");

    QQmlApplicationEngine engine;
    const QUrl url("qrc:/00_Scene_Graph_Example/Main.qml");
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreationFailed,
        &app, []() { QCoreApplication::exit(-1); },
```

```
        Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

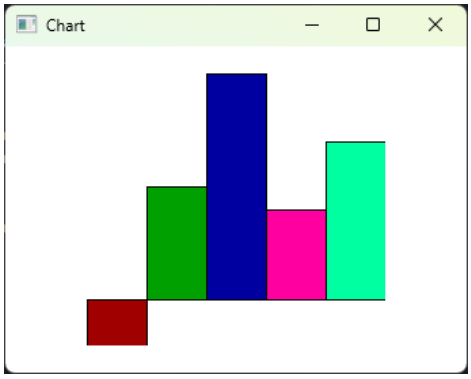
The following is the QML source code. Write the source code as shown below.

```
import QtQuick
import QtQuick.Window
import Shapes 1.0

Window {
    width: 300; height: 200; visible: true
    title: "Scene Graph"

    Triangle {
        anchors.centerIn: parent
        width: 200; height: 100
    }
}
```

The source code for this example can be found in the 00_Scene_Graph_Example directory.

- Bar chart implementation example

In this example, we'll implement a Bar chart graph, as shown in the following figure.



As shown in the figure above, we implemented two QML types to display a Bar chart. The Chart type acts like a Container. For example, as shown in the figure above, it provides the Container function for displaying five bar graphs in one chart. And the Bar type is

the actual element of the bar graph. For example, as you can see in the image above, there are five bars, so we used five Bar types.

Create a new project and then create a BarItem class in the project. Then create the baritem.h header file as shown below.

```
#ifndef BARITEM_H
#define BARITEM_H
#include <QColor>
#include <QObject>

class BarItem : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
    Q_PROPERTY(qreal value READ value WRITE setValue NOTIFY valueChanged)

public:
    BarItem(QObject *parent = 0);
    QColor color() const;
    void setColor(const QColor &newColor);
    qreal value() const;
    void setValue(qreal newValue);

signals:
    void colorChanged();
    void valueChanged();

private:
    QColor m_color;
    qreal m_value;
};

#endif
```

Next, create the baritem.cpp source code file as shown below.

```
#include "baritem.h"

BarItem::BarItem(QObject *parent)
    : QObject(parent)
```

```
{
}

QColor BarItem::color() const
{
    return m_color;
}

void BarItem::setColor(const QColor &newColor)
{
    if (m_color != newColor) {
        m_color = newColor;
        emit colorChanged();
    }
}

qreal BarItem::value() const
{
    return m_value;
}

void BarItem::setValue(qreal newValue)
{
    if (m_value != newValue) {
        m_value = newValue;
        emit valueChanged();
    }
}
```

Next, add the ChartItem class, and write the chartitem.h header file like this

```
#ifndef CHARTITEM_H
#define CHARTITEM_H

#include <QQuickPaintedItem>

class BarItem;
class ChartItem : public QQuickPaintedItem
{
    Q_OBJECT
```

```
    Q_PROPERTY(QQmlListProperty<BarItem> bars READ bars NOTIFY barsChanged)

public:
    ChartItem(QQuickItem *parent = nullptr);
    void paint(QPainter *painter);
    QQmlListProperty<BarItem> bars();

signals:
    void barsChanged();

private:
    static void append_bar(QQmlListProperty<BarItem> *list, BarItem *bar);
    QList<BarItem*> m_bars;
};
#endif
```

Next, create the chartitem.cpp source code file as shown below.

```
#include <QPainter>
#include "baritem.h"
#include "chartitem.h"

ChartItem::ChartItem(QQuickItem *parent)
    : QQuickPaintedItem(parent)
{
}

void ChartItem::paint(QPainter *painter)
{
    if (m_bars.count() == 0)
        return;

    qreal minimum = m_bars[0]->value();
    qreal maximum = minimum;

    for (int i = 1; i < m_bars.count(); ++i) {
        minimum = qMin(minimum, m_bars[i]->value());
        maximum = qMax(maximum, m_bars[i]->value());
    }
```

```
    if (maximum == minimum)
        return;

    painter->save();

    const QRectF rect = boundingRect();

    qreal scale = rect.height()/(maximum - minimum);
    qreal barWidth = rect.width()/m_bars.count();

    qDebug() << scale;
    qDebug() << barWidth;

    for (int i = 0; i < m_bars.count(); ++i) {
        BarItem *bar = m_bars[i];
        qreal barEdge1 = scale * (maximum - bar->value());
        qreal barEdge2 = scale * maximum;
        QRectF barRect(rect.x() + i * barWidth,
                       rect.y() + qMin(barEdge1, barEdge2),
                       barWidth, qAbs(barEdge1 - barEdge2));

        painter->setBrush(bar->color());
        painter->drawRect(barRect);
    }

    painter->restore();
}

QQmlListProperty<BarItem> ChartItem::bars()
{
    return QQmlListProperty<BarItem>(this, &m_bars);
}

void ChartItem::append_bar(QQmlListProperty<BarItem> *list, BarItem *bar)
{
    ChartItem *chart = qobject_cast<ChartItem *>(list->object);
    if (chart) {
        bar->setParent(chart);
        chart->m_bars.append(bar);
```

```
    }
}
```

Next, open the main.cpp file and write something like this

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "chartitem.h"
#include "baritem.h"

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

    qmlRegisterType<ChartItem>("Shapes", 1, 0, "Chart");
    qmlRegisterType<BarItem>("Shapes", 1, 0, "Bar");

    QQmlApplicationEngine engine;
    const QUrl url(u"qrc:/01_ChartItem_Example/Main.qml");
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreationFailed,
        &app, []() { QCoreApplication::exit(-1); },
        Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

Next, write Main.qml as shown below.

```
import QtQuick
import QtQuick.Window
import Shapes 1.0

Window {
    width: 340
    height: 240
    visible: true
    title: qsTr("Chart")


    Chart {
```

```
        width: 220; height: 200
        anchors.centerIn: parent
        bars: [
            Bar { color: "#a00000"; value: -20 },
            Bar { color: "#00a000"; value: 50  },
            Bar { color: "#0000a0"; value: 100 },
            Bar { color: "#ff00a0"; value: 40  },
            Bar { color: "#00ffa0"; value: 70  }
        ]
    }
}
```

As shown in the example above, the Chart type is implemented in class ChartItem and the Bar type is implemented in class BarItem.

The BarItem defines the properties of the bar graph, and the actual drawing of the bar graph elements is done by the ChartItem class.

The source code for this example can be found in the 01_ChartItem directory.

5.5. Interaction and variable mapping between C++ and QML

Interaction refers to handling QML types in C++, accessing or setting property values of QML types, and connecting to SIGNALs in QML.

Therefore, in this chapter, we will discuss interactions and variable mappings between Qt C++ and QML.

✓ **Interaction between C++ and QML using QQmlComponent class in C++**

The QQmlComponent class provides functionality for interacting with QML. Create a QML file like the one below.

```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 300; visible: true
    title: qsTr("Interaction Example")

    Rectangle {
        id: myRect
        anchors.centerIn: parent
        width: 200; height: 100
        color: "#0000FF"
    }
}
```

Let's say we have a QML file like the one above, we need to use the QQmlComponent class to access the above QML types.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlComponent>
#include <QQmlProperty>
#include <QDebug>
```

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

    QQmlApplicationEngine engine;
    QQmlComponent component(&engine, "qrc:/00_Interaction/Main.qml");
    QObject *object = component.create();

    qDebug() << "Width : " << QQmlProperty(object, "width").write(400);
    return app.exec();
}
```

✓ How to Access a Property or Object within a QML Type Using C++

To change a property of a QML type using the QObject used in the example above, you can use the setProperty( ) member function of the QObject class or QQmlProperty, as follows

```
object->setProperty("width", 500);
QQmlProperty(object, "width").write(500);
```

✓ How to WRITE/READ QML Type Properties with C++

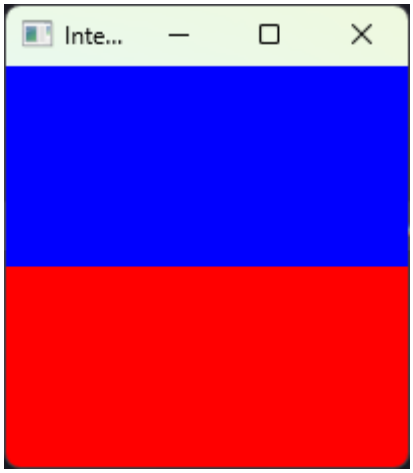
To access a QML type in C++, you can use the findChild( ) member function of the QObject class to access the properties of the QML type. The following is the source code for a QML example.

```
import QtQuick
import QtQuick.Window

Window {
    width: 200; height: 200; visible: true
    title: qsTr("Interaction Example")

    Rectangle {
        id: myRect
        width: 200; height: 100
        color: "#0000FF"
```

```
    }

    Rectangle {
        objectName: "secondRect"
        anchors.left: myRect.left
        anchors.top: myRect.bottom
        width: 200; height: 100
        color: "#00FF00"
    }
}
```

To access the color property of the Rectangle type in the above QML example source code, you can change the value of the color property in C++ as follows.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlComponent>
#include <QQmlProperty>

#include <QDebug>

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

    QQmlApplicationEngine engine;

    QQmlComponent component(&engine, "qrc:/00_Interaction/Main.qml");
    QObject *object = component.create();
    QObject *rect = object->findChild<QObject*>("secondRect");
    rect->setProperty("color", "#FF0000");

    return app.exec();
}
```



The following example source code shows how to reference a property value of a QML type.

```
import QtQuick
import QtQuick.Window

Window {
    width: 200; height: 200; visible: true
    title: qsTr("Interaction Example")

    property int someNumber: 100
}
```

As a way to reference the someNumber property above in C++, you can use the property( ) member function provided by the QObject class, as shown in the following C++ example source code.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlComponent>
#include <QQmlProperty>
#include <QDebug>

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

```
    QQmlApplicationEngine engine;

    QQmlComponent component(&engine, "qrc:/00_Interaction/Main.qml");
    QObject *object = component.create();

    qDebug() << "Property value:"
             << QQmlProperty::read(object, "someNumber").toInt();

    QQmlProperty::write(object, "someNumber", 5000);

    qDebug()  << "Property value:"
              << object->property("someNumber").toInt();

    return app.exec();
}
```

✓ How to call a Method( ) written in QML from C++.

In C++, you can use the invokeMethod( ) member function provided by the QMetaObject class to call a Method implemented in QML using the QVariant class. The following is the source code of an example Method function implemented in QML.

```
import QtQuick
import QtQuick.Window

Window {
    width: 200; height: 200; visible: true
    title: qsTr("Interaction Example")

    function myQmlFunction(msg) {
        console.log("Got message:", msg)
        return "some return value"
    }

}
```

To call myQmlFunction( ) implemented in QML above from C++, you can use it as shown in the following example.

```
#include <QGuiApplication>
```

```
#include <QQmlApplicationEngine>
#include <QQmlComponent>
#include <QQmlProperty>

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

    QQmlApplicationEngine engine;

    QQmlComponent component(&engine, "qrc:/00_Interaction/Main.qml");
    QObject *object = component.create();

    QVariant returnedValue;
    QVariant msg = "Hello from C++";
    QMetaObject::invokeMethod(object,
                              "myQmlFunction",
                              Q_RETURN_ARG(QVariant, returnedValue),
                              Q_ARG(QVariant, msg) );
    return app.exec();
}
```

✓ How to use the connection() function in C++ to connect to a Signal in QML

To connect a Signal in QML using the connection( ) function for connecting a Signal and a Slot in C++, we need to register a signal as shown in the following QML example.

```
import QtQuick
import QtQuick.Window

Window {
    width: 200; height: 200; visible: true
    title: qsTr("Signal Example")

    id: item
    signal qmlSignal(string msg)

    MouseArea {
        anchors.fill: parent
```

```
        onClicked: {
            item.qmlSignal("Hello from QML")
        }
    }
}
```

The following example is an example source code to connect the Signal in QML above with the Slot function provided in C++.

After creating the C++ class as shown below, create a header file as shown below.

```
#ifndef MYCLASS_H
#define MYCLASS_H

#include <QObject>
#include <QDebug>

class MyClass : public QObject
{
    Q_OBJECT
public:
    explicit MyClass(QObject *parent = nullptr);

public slots:
    void cppSlot(const QString &msg) {
        qDebug() << "Called the C++ slot with message:"
                 << msg;
    }
};

#endif // MYCLASS_H
```

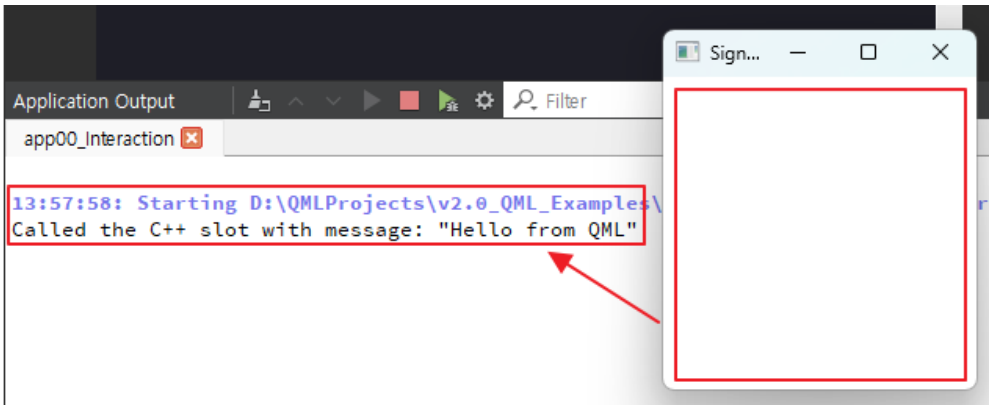
Next, create a source code file.

```
#include "myclass.h"

MyClass::MyClass(QObject *parent)
    : QObject{parent}
{
}
```

Next, write main.cpp as shown below.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlComponent>
#include <QDebug>
#include "myclass.h"


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

    QQmlApplicationEngine engine;

    QQmlComponent component(&engine, "qrc:/00_Interaction/Main.qml");
    QObject *object = component.create();

    MyClass myClass;
    QObject::connect(object, SIGNAL(qmlSignal(QString)),
                     &myClass, SLOT(cppSlot(QString)));

    return app.exec();
}
```

Click on the QML area to see the message as shown in the image below.



✓ Data Exchange and Variable Type Mapping between C++ and QML

For data exchange between C++ and QML, the following types can be converted to the following types.

Qt C++ Variables	QML Variable Types
bool	bool
unsigned int 또는 int	int
double	double
float, qreal	real
QString	string
QColor	color
QFont	font
QDate	date
QTime	time
QPoint, QPointF	point
QSize, QSizeF	size
QRect, QRectF	rect
QMatrix4x4	matrix4x4
QQuaternion	quaternion
QVector2D	vector2d
QVector3D	vector3d
QVector4D	vector4d

Classes such as QColor, QFont, QQuaternion, etc. that you used in Qt C++ can be used in QML, as shown in the following example.

```
Item {
    Image { sourceSize: "100x200" }
    Image { sourceSize: Qt.size(100, 200) }
}
```

- Example of Data Exchange between C++ and QML using Model and View

In this example, we will see how to use the values of the QStringList class in Qt C++ in QML.

After creating the project, create a QML as shown below.

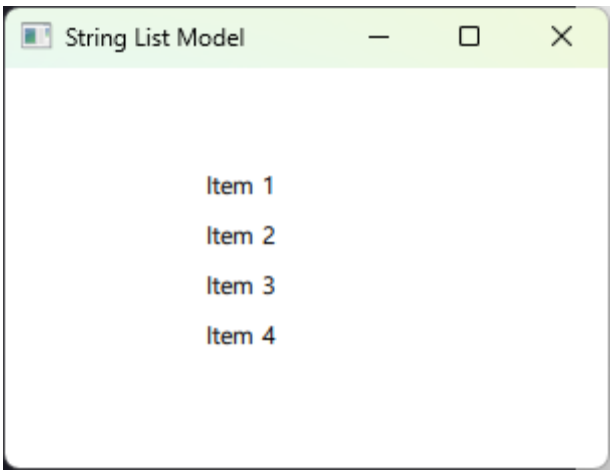
```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true
    title: qsTr("String List Model")



    ListView {
        anchors.centerIn: parent
        width: 100; height: 100

        model: MyModel
        delegate: Rectangle
        {
            height: 25
            width: 100
            Text { text: modelData }
        }
    }
}
```

Next, open the main.cpp file and write something like this

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

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

    QStringList dataList;
    dataList.append("Item 1");
    dataList.append("Item 2");
    dataList.append("Item 3");
    dataList.append("Item 4");
```

```
    QQmlApplicationEngine engine;

    QQmlContext *ctx = engine.rootContext();
    ctx->setContextProperty("MyModel", QVariant::fromValue(dataList));

    const QUrl url("qrc:/00_StringListModel/Main.qml");
    engine.load(url);

    return app.exec();
}
```

As shown in the example above, in order to use the data array stored in the QStringList class in QML, we used the Model/View provided by QML. The following is a QML example



The source code for the example above can be found in the 00_StringListModel directory.

- C++ Class Interchange with Model and View

In the previous example, we passed string data to QML in a QList class. In this example, we will pass a class to QML. First, create the C++ class that you want to pass to QML. Create a project and create a DataObject class. Then create the dataobject.h header file.

```
#ifndef DATAOBJECT_H
#define DATAOBJECT_H

#include <QObject>

class DataObject : public QObject
```

```
{
    Q_OBJECT
    Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
    Q_PROPERTY(QString color READ color WRITE setColor NOTIFY colorChanged)
public:
    explicit DataObject(QObject *parent = nullptr);
    DataObject(const QString &name,
               const QString &color,
               QObject *parent=0);

    QString name() const;
    void setName(const QString &name);
    QString color() const;
    void setColor(const QString &color);

signals:
    void nameChanged();
    void colorChanged();

private:
    QString m_name;
    QString m_color;
};

#endif // DATAOBJECT_H
```

Next, create the dataobject.cpp file as shown below.

```
#include "dataobject.h"

DataObject::DataObject(QObject *parent)
    : QObject{parent}
{
}

DataObject::DataObject(const QString &name,
                       const QString &color,
                       QObject *parent )
    : QObject(parent),
    m_name(name),
```

```
    m_color(color)
{
}

QString DataObject::name() const {
    return m_name;
}
void DataObject::setName(const QString &name)
{
    if (name != m_name) {
        m_name = name;
        emit nameChanged();
    }
}
QString DataObject::color() const {
    return m_color;
}

void DataObject::setColor(const QString &color)
{
    if (color != m_color) {
        m_color = color;
        emit colorChanged();
    }
}
```

The color( ) member function of class DataObject returns the value of m_color stored in this class in QML. Therefore, the member functions of the DataObject class can be used in QML.

Next, create the Main.qml file as shown below.

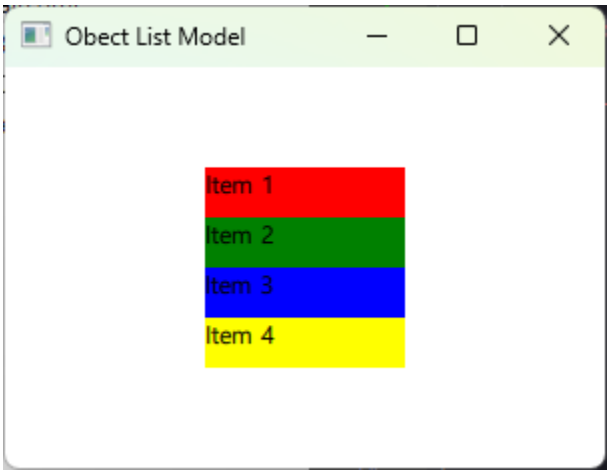
```
import QtQuick
import QtQuick.Window

Window {
    width: 300; height: 200; visible: true
    title: qsTr("Obect List Model")

    ListView {
```

```
        anchors.centerIn: parent
        width: 100; height: 100
        model: MyModel

        delegate: Rectangle
        {
            height: 25
            width: 100
            color: model.modelData.color

            Text {
                text: name
            }
        }
    }
}
```

Next, let's see how to store objects of class DataObject in a QList and pass values to QML using class QQmlContext. Open the main.cpp file and write the following code.

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

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

    QList<QObject*> dataList;
    dataList.append(new DataObject("Item 1", "red"));
    dataList.append(new DataObject("Item 2", "green"));
    dataList.append(new DataObject("Item 3", "blue"));
    dataList.append(new DataObject("Item 4", "yellow"));

    QQmlApplicationEngine engine;

    QQmlContext *ctx = engine.rootContext();
    ctx->setContextProperty("MyModel", QVariant::fromValue(dataList));
```

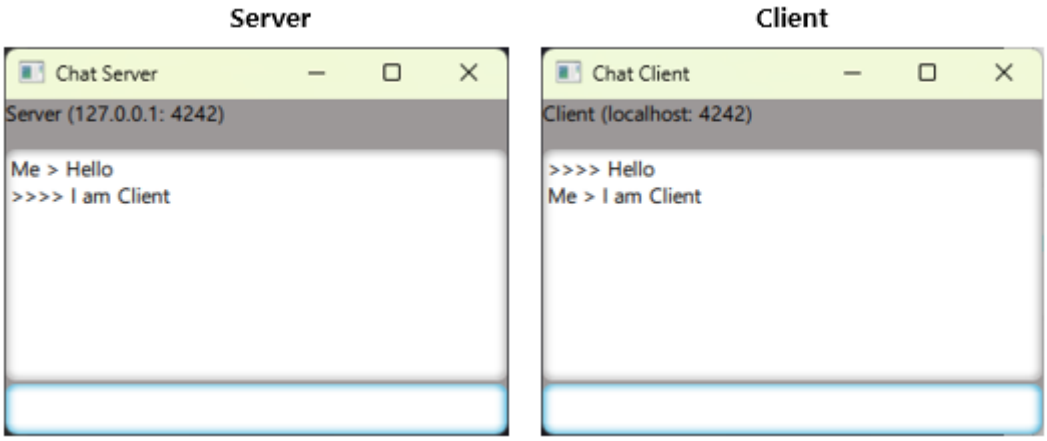
```
    const QUrl url("qrc:/01_ObjectListModel/Main.qml");
    engine.load(url);

    return app.exec();
}
```



You can find the source code for this example in the 01_ObjectListModel directory.

5.6. Implementing a chat application based on TCP

In this chapter, we will develop a chat application based on the TCP protocol using C++ and QML. The example application we will implement is divided into Server and Client.



The two examples, Server and Client, consist of the following source code files. The table below shows the C++ and QML source code files for the Server example.

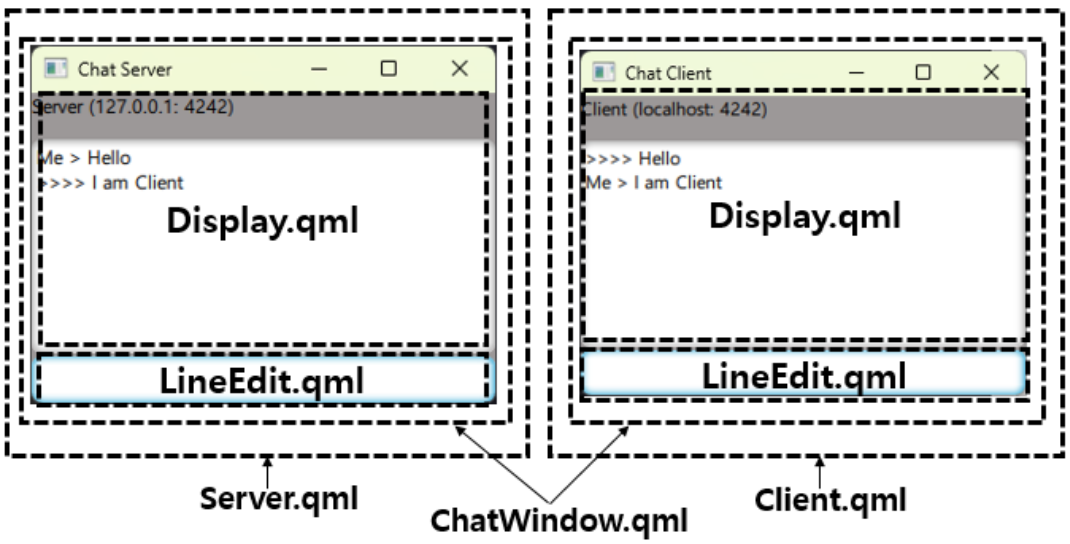
Kind	Source	Description
C++	main.cpp	Program starting point
	tcpconnection.h tcpconnection.cpp	The TcpConnection class implements the ability to send or receive messages over a network in QML.
QML	Server.qml	Set by calling the QML type implemented in ChatWindow.qml
	ChatWindow.qml	Configure the GUI with QML. Configure the Title, the chat content pane, and the screen for message entry at the bottom, as shown in the running example screen above.
	Display.qml	Implementing the chat transcript window QML type
	LineEdit.qml	Implementing Macy's Input QML Types

The table below shows the C++ and QML source files for the Client example.

Kind	Source	Description
C++	main.cpp	Program starting point
	tcpconnection.h tcpconnection.cpp	Implementing the ability to send or receive messages over a network in QML with the TcpConnection class
QML	Client.qml	Set by calling the QML type implemented in ChatWindow.qml
	ChatWindow.qml	Configure the GUI with QML Configure the Title, the chat content pane, and the screens for message entry at the bottom, as shown in the running example screen above
	Display.qml	Implementing the chat transcript window QML type
	LineEdit.qml	Implementing Macy's Input QML Types

c The two examples, Server and Client, use the same TcpConnection class. The only difference is that the server example uses Server.qml and the client example uses Client.qml.

So only Server.qml and Client.qml are different, and the ChatWindow.qml, Display.qml, and LineEdit.qml source code is the same for both Server and Client. The other difference is that the Server example calls Server.qml from main.cpp and the Client example calls Client.qml.



c The main.cpp below is the main.cpp of 00_Chatting_Server.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "tcpconnection.h"

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

    qmlRegisterType<TcpConnection>("TCP", 1, 0, "TcpConnection");

    QQmlApplicationEngine engine;
    const QUrl url("qrc:/00_Chatting_Server/Server.qml");
    engine.load(url);

    return app.exec();
}
```

As shown in the example above, the Server example calls Server.qml.

The TcpConnection class is a class that implements the TcpConnection QML type in QML. This class implements the message sending/receiving functionality required for network chat based on the TCP protocol using the QTcpServer class if it is a server.

In the case of a client, the QTcpSocket class is used to implement the message sending/receiving functions required for network chat based on the TCP protocol. Therefore, the TcpConnection class provides the network configuration in QML and the message sending/receiving functions for Server and Client. The following example shows the source code of the header tcpconnection.h of the TcpConnection class.

```
#ifndef TCPCONNECTIONELEMENT_H
#define TCPCONNECTIONELEMENT_H

#include <QObject>

class QTcpServer;
class QTcpSocket;
class TcpConnection : public QObject
{
    Q_OBJECT
```

```
    Q_PROPERTY(int port READ port WRITE setPort NOTIFY portChanged)

    Q_PROPERTY(QString hostname READ hostname WRITE setHostName
                NOTIFY hostNameChanged)

    Q_PROPERTY(ConnectionType type READ connectionType WRITE setConnectionType
                NOTIFY connectionTypeChanged)

public:
    enum ConnectionType { Server, Client };
    Q_ENUM(ConnectionType)

    TcpConnection(QObject *parent = nullptr);
    void setConnectionType(ConnectionType connectionType);
    ConnectionType connectionType() const;

    void setPort(int port);
    int port() const;

    void setHostName(const QString &hostName);
    QString hostName() const;

public slots:
    void initialize();
    void sendData(const QString &data);

signals:
    void dataReceived( const QString &data );
    void portChanged();
    void hostNameChanged();
    void connectionTypeChanged();

private slots:
    void receivedData();
    void slotConnection();
private:
    int m_port;
    QString m_hostName;
    ConnectionType m_connectionType;
```

```
    QTcpServer *m_tcpServer;
    QTcpSocket *m_tcpSocket;
};

#endif
```

The following is the tcpconnection.cpp source code file.

```
#include "tcpconnection.h"
#include <QHostAddress>

#include <QTcpServer>
#include <QTcpSocket>

TcpConnection::TcpConnection(QObject *parent)
    : QObject(parent), m_hostName("127.0.0.1")
{
}

void TcpConnection::sendData(const QString &data)
{
    m_tcpSocket->write( data.toUtf8() + "\n" );
}

int TcpConnection::port() const
{
    return m_port;
}

void TcpConnection::setPort(int port)
{
    if (m_port != port) {
        m_port = port;
        emit portChanged();
    }
}

QString TcpConnection::hostName() const
{
```

```
    return m_hostName;
}

void TcpConnection::setHostName(const QString &hostName)
{
    if (m_hostName != hostName) {
        m_hostName = hostName;
        emit hostNameChanged();
    }
}

TcpConnection::ConnectionType TcpConnection::connectionType() const
{
    return m_connectionType;
}

void TcpConnection::setConnectionType(ConnectionType connectionType)
{
    if (m_connectionType != connectionType) {
        m_connectionType = connectionType;
        emit connectionTypeChanged();
    }
}

void TcpConnection::initialize()
{
    if ( m_connectionType == Server ) {
        m_tcpServer = new QTcpServer;
        m_tcpServer->listen( QHostAddress(m_hostName), m_port );
        connect( m_tcpServer, SIGNAL( newConnection() ),
                 this, SLOT( slotConnection() ) );
    }
    else {
        m_tcpSocket = new QTcpSocket(this);
        m_tcpSocket->connectToHost( m_hostName, m_port );
        connect( m_tcpSocket, SIGNAL(readyRead()), this, SLOT(receivedData()) );
    }
}
```

```
void TcpConnection::slotConnection()
{
    m_tcpSocket = m_tcpServer->nextPendingConnection();
    connect( m_tcpSocket, SIGNAL(readyRead()), this, SLOT(receivedData()) );
}

void TcpConnection::receivedData()
{
    const QString txt = QString::fromUtf8(m_tcpSocket->readAll());
    emit dataReceived( txt );
}
```

The sendData( ) member function provides the ability to send a message. port( ) returns the current network port number. setPort( ) allows you to set the network port number.

The initialize( ) member function implements functions for server functionality using the QTcpServer class if it is a server. If it is a Client, it uses the QTcpClient class.

The slotConnection( ) Slot function is a Slot function that is called when a new connection event occurs. This Slot function uses the connection( ) function to handle the readyRead( ) signal event so that the receivedData( ) Slot function can be called when the new user receives a message. The readyRead( ) signal is fired when a message is received from the network. Let's take a look at the Server.qml example used by Server.

```
import QtQuick
import QtQuick.Window
import TCP 1.0

Window {
    width: 300; height: 200; visible: true
    title: "Chat Server"

    ChatWindow {
        width: 300
        height: 200
        type : TcpConnection.Server
        port : 4242
    }
}
```

In the Server.qml above, we used TcpConnection.Server as the value of the type property.

In Client.qml, we use TcpConnection.Client. The following is the ChatWindow.qml file.

```
import QtQuick
import TCP 1.0

Item {
    property alias type : tcpConnection.type
    property alias port : tcpConnection.port
    property alias hostName : tcpConnection.hostName

    TcpConnection {
        id : tcpConnection
        onDataReceived : {
            output.text += ">>>> " + data
        }
    }

    Component.onCompleted: tcpConnection.initialize()
    Rectangle {
        color: "#9c9898"; border.width: 0;
        border.color: "#000000"; anchors.fill: parent
    }
    Text {
        id : title
        text: titleText()
        anchors {
            top : parent.top
            left : parent.left
            right : parent.right
        }
        height : 30
    }
    Display {
        id: output
        height:300
        anchors.bottomMargin: 2
        anchors {
            top : title.bottom
            left : parent.left
            right : parent.right
```

```
            bottom : entryRect.top
        }
    }

    LineEdit {
        id: entryRect
        x: 0; y: 83; width: 100; height: 30
        focus: true
        anchors {
            bottom : parent.bottom
            left : parent.left
            right : parent.right
        }
        onTextEntered: function(text) {
            output.text += "Me > " + text + "\n"
            tcpConnection.sendData(text)
        }
    }

    function titleText() {
        return (tcpConnection.type == TcpConnection.Server ? "Server" : "Client")
                + " (" + hostName + ": " + port + ")"
    }
}
```

ChatWindow.qml organizes the screens of the GUI. Server and Client use the same source code. The following is the source code for the Display.qml example.

```
import QtQuick

Rectangle {
    property alias text: output.text
    color: "transparent"
    border.color: "transparent"
    BorderImage {
        anchors.fill: parent
        border { left: 5; top: 5; right: 5; bottom: 6 }
        horizontalTileMode: BorderImage.Stretch
        verticalTileMode: BorderImage.Stretch
        source: "./images/textoutput.png"
```

```
    }

    TextEdit {
        id: output
        anchors { margins: 3; fill: parent }
        selectionColor: "transparent"
    }
}
```

The Display.qml above displays the contents of the chat. For example, any messages sent by a remote user or by you will be displayed. The following is the source code for an example LineEdit.qml that provides a message input window.

```
import QtQuick

Rectangle {
    signal textEntered(string text)

    height: entryField.height+6
    color: "transparent"
    border.color: "transparent"
    anchors {
        rightMargin: 0; leftMargin: 0; bottomMargin: 0
    }

    BorderImage {
        anchors.fill: parent
        border { left: 5; top: 5; right: 5; bottom: 6 }
        horizontalTileMode: BorderImage.Stretch
        verticalTileMode: BorderImage.Stretch
        source: "./images/textinput.png"
    }
    TextInput {
        id : entryField
        anchors { margins: 3; fill: parent }
        focus: true

        Keys.onReturnPressed : {
            textEntered(text)
            text = ""
```

```
        }
    }
}
```

For server examples, see the 00_Chatting_Server directory. And for client examples, see the 00_Chatting_Client directory.