
Qt
Qt Quick Controls

Contents

- Qt Quick Controls Introduction
- Window and Application Window
- Controls
- Localization
- Container Controls
- Views
- Qt Quick Layouts
- Styling

Qt Quick Controls Introduction

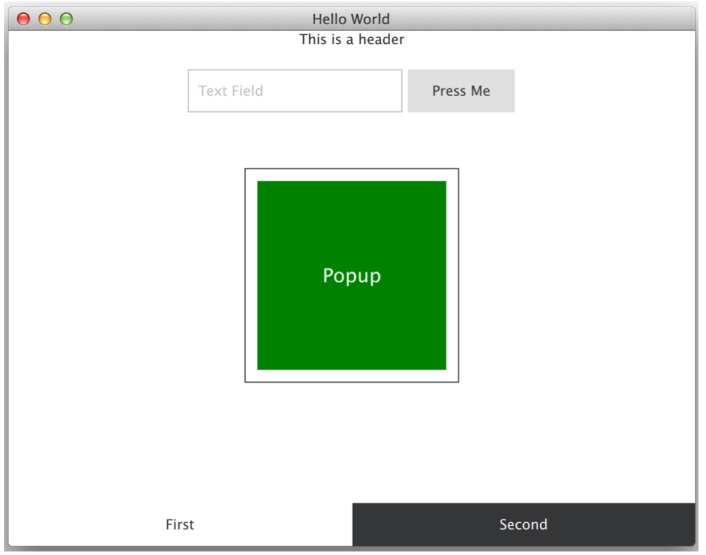
Ready-made UI control QML types in two versions

- Qt Quick Controls 1
  - Implemented in QML, by extending existing QML types (`Button` -> `FocusScope`)
  - Rather heavy in terms of memory consumption
    - E.g. `Button` allocated 60 `QObjects`, while Qt Quick Controls 2 `Button` allocates 7
  - Customization sometimes difficult as more specialized than Qt Quick 2 Controls
  - Styled with control-specific style types
- Qt Quick Controls 2
  - Easier to customize as control content can be any `Item`
  - Event handling in C++, no need for a separate `MouseArea`, for example
  - More memory efficient
  - Styled with application-global, configurable style
  - High-DPI support

Basic Building Blocks

- More than just UI controls

- `ApplicationWindow`
- Views
  - Ready-made layout and/or navigation
  - Scroll, stack, and swipe views
  - Split, tab, and table views only in Qt Quick Controls 1
- Layouts
  - Can dynamically shrink/expand items in the layout
- Controls
  - `Button, BusyIndicator, ComboBox, Drawer, Page, Pane, ScrollBar, Slider, SpinBox, Tumbler, RangeSlider, DelayButton`



Window and Application Window

- `Window` QML type instantiates `QQuickWindow` C++ class
  - Basic window management: geometry, visibility, window flags, background color
  - Syncs with scene graph to render items on the scene
- Multiple screen support
  - Qt supports multiple screens
  - Easy to define with which screen the window is associated with – use property `screen`
  - Useful `Screen` properties: `screen orientation, screen physical dimensions and pixels, pixel density`

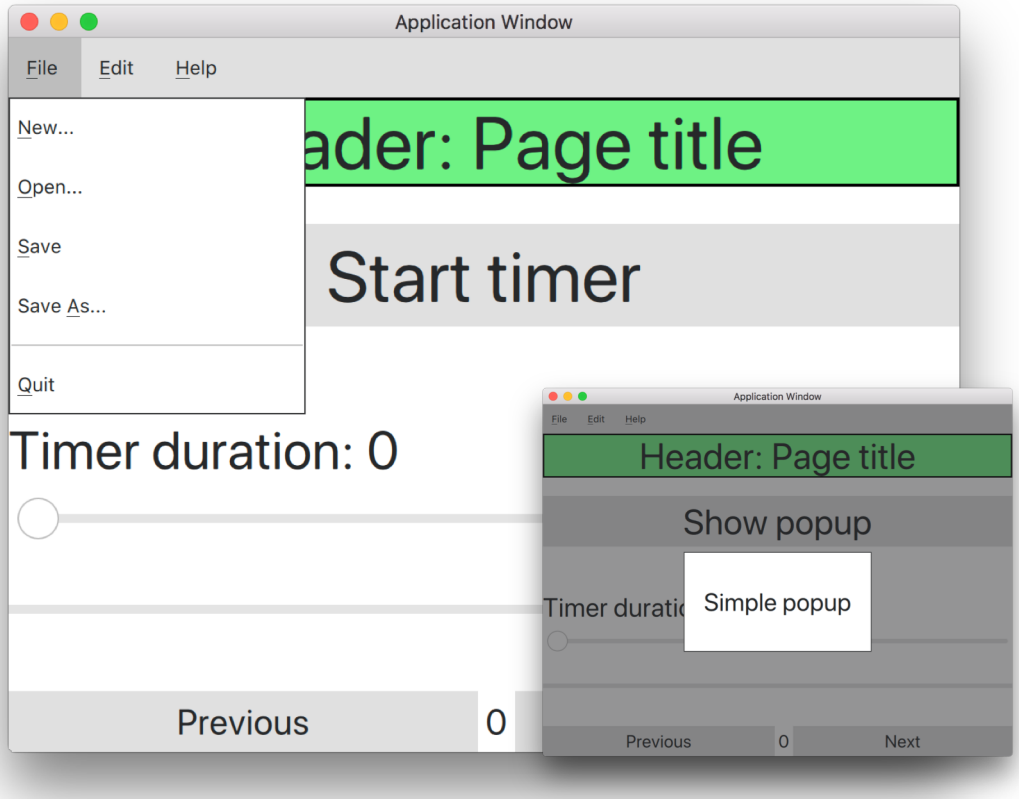
```
property int orientation: Screen.orientation
Screen.orientationUpdateMask: Qt.PortraitOrientation
// The default mask value is 0
```

- `ApplicationWindow` extends `Window`

Demo: controls/applicationWindow

Application Window

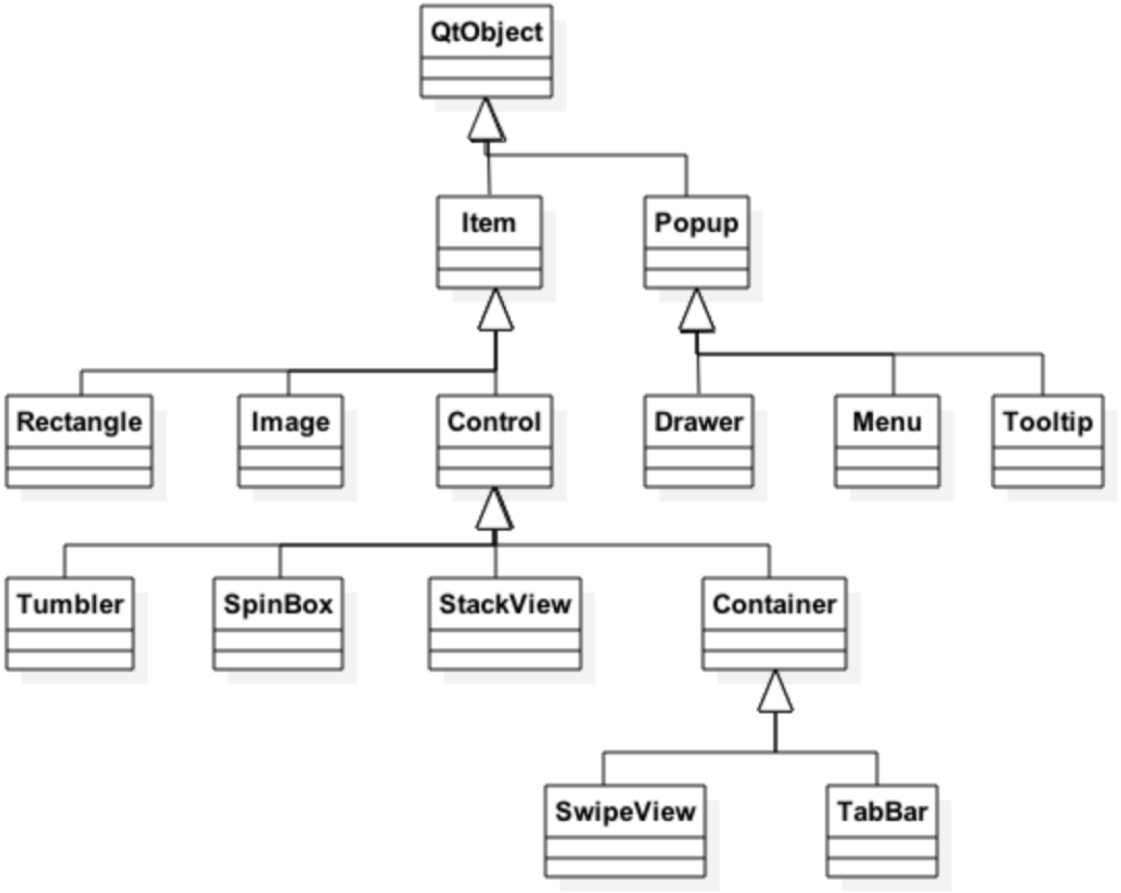
- Header and footer items
- Menubar
  - With menus, sub-menus, separators, and actions
- Content
  - Window children
  - For example, a view, a container, a control, an item
- Background
  - Any item
- Overlay
  - Modal or non-modal popup
  - Modal popup dims the window
- Window-specific palette



Application Window

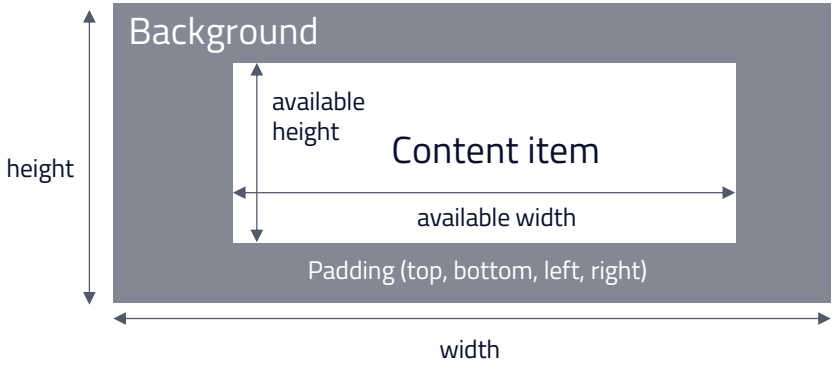
```
ApplicationWindow {
    visible: true; width: 640; height: 480; title: qsTr("Hello World")
    header: Label {
        horizontalAlignment: Qt.AlignHCenter
        text: qsTr("This is a header") }
    footer: TabBar {
        TabButton { text: qsTr("Open a popup 1")
            onClicked: popup.open();
        }
    }
    menuBar: MenuBar {
        Menu { title: qsTr("&File")
        Action { text: qsTr("&New...") }
        MenuSeparator { }
    Popup { id: popup
        width: parent.width * 0.5; height: parent.height * 0.5
        x: (parent.width - width) / 2; y: (parent.height - height) / 2
        modal: true
        Text { anchors.centerIn: parent text: qsTr("Text in popup") }
    }
    Container { id: container }
```

Controls



Controls

- Extend `Control`
- Generic content – property `contentItem`
- Event handling
  - Interactive controls do not pass mouse click and touch events to the item below them
- Locale aware
- Background
- Layout
  - Control's implicit size is the background's implicit size by default
- Palette
- Focus
  - Any item may request to get active focus (property `focus`)



Palette

- Basic QML type `palette`
- Defines color of various roles
- Application wide
  - Maintained by the application window
  - Can be set in `QGuiApplication`
  - Propagated to all window children
- Can be customized in controls
  - All control children will use the customized palette
- Some color properties
  - `palette.base`
  - `palette.brightText`
  - `palette.text`

```
ApplicationWindow {
    id: window visible
    // All child controls will have blue foreground
text
    palette.text: "blue"
```

Localization

- Unicode support for strings
- On-screen user visible texts
  - Can be marked localizable
  - Plural handling (1 file vs 2 files)
- Retranslation support
  - `QQmlEngine::retranslate()`
- Locale awareness
  - Number and date formats
  - Locale-based resources (`icon_de_DE.png`)
- Translation tool - Qt Linguist
- LTR and RTL text, positioners, list and grid views

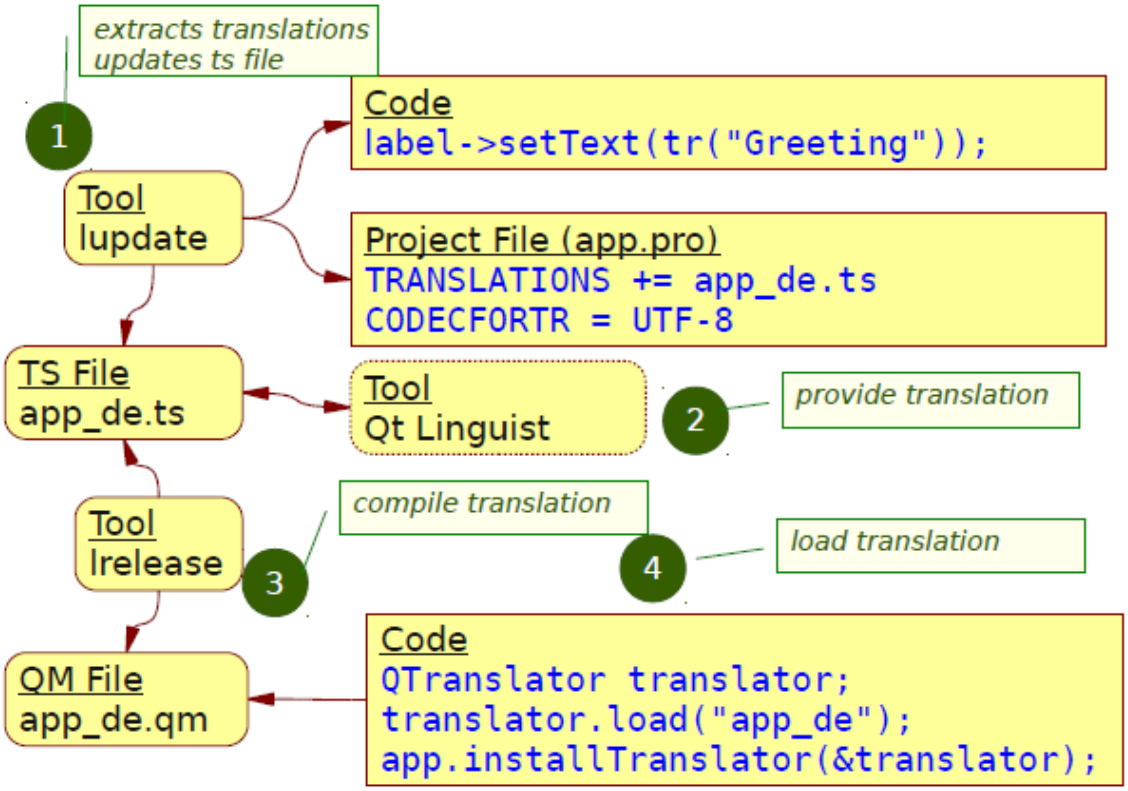
Localized Strings

- **lupdate** - scans C++ and `.ui` files for strings and creates or updates the translation script `.ts` file
  - Uses `SOURCES` and `FORMS` in `.pro` file
  - Must add QML files into `SOURCES`

```
lupdate_only {
    SOURCES += *.qml }
```

- **linguist** – graphical tool to localize strings
- **lrelease** - reads `.ts` files and creates `.qm` binary files
- **`QTranslator`** – allows adding any number of translations
  - Call `retranslate()` after a new language has been added – refreshes ALL bindings

Translation Process



String Localization in QML

- Strings are localized with translation functions
  - `string qsTr(string sourceTest, string disambiguation, int n)`
  - `string qsTranslate(string context, string sourceTest, string disambiguation, int n)`
  - `string qsTrId(string id)`
    - `//% "I'm a template %n"    // Possible to define templates`
    - `property string localizedText: qsTrId("someId", 7);`
    - String ids require `lrelease -idbased` option
  - Note! Functions do not work in .js files
- Context and comments may be used to help the translator
  - // : Comment
  - //~Context The text is used in 3D graphics programming
- If the same text must be used in different ways, disambiguation can be used
  - Otherwise, the same single translation used everywhere, where the same `sourceText` used

String Localization in QML

- To localize string arrays, use `QT_TR_NOOP()`, `QT_TRANSLATE_NOOP()` or `QT_TRID_NOOP()` functions
  - They Identify a string, requiring translation, but uses `qs*()` functions to do the actual translation
  - All the strings can be managed in one place
    - `var stringArray[] = [ QT_TR_NOOP("hello"), QT_TR_NOOP("world") ]`
    - `Text { text: qsTr(stringArray(0)) }`
- Translation functions look for translations from all installed translation modules, starting from the last one installed
  - The first translation found (if any), will be returned
  - Otherwise, the `sourceText` is returned
- `QQmlApplicationEngine` tries to find and install a system default translation automatically
  - If provided in `qml` folder adjacent to `main.qml` file and named as `qml_language_country.qm` (e.g., `qml_fi_FI.qm`)

LTR and RTL Support

- `Text`, `TextInput`, and `TextEdit` automatically align the text according to `QInputMethod::inputDirection()`, which uses the system locale
  - Can be overridden with `horizontalAligment` property or `LayoutMirroring.enabled` attached property
  - The latter changes the effective layout direction

```
Text {
    text: "خامل"
    horizontalAlignment: Text.AlignLeft
    LayoutMirroring.enabled: true
}
```

- `Row`, `Grid`, `Flow`, `ListView`, `GridView`
  - By default horizontal direction is left to right
  - Change using `layoutDirection` property (`Qt.RightToLeft`) or `LayoutMirroring`

LTR and RTL Support

- `LayoutMirroring` can be used to mirror left and right anchors
  - Again actual anchors do not change, but the effective layout direction changes
- `LayoutMirroring` does not inherit to child items by default
  - Use `LayoutMirroring.childrenInherit` to change this
- Current layout direction may be queried with
  - `Qt.application.layoutDirection` property
  - And changed with `QGuiApplication::setLayoutDirection()`
- It's also possible to make the layout direction locale aware by defining
  - `QT_TRANSLATE_NOOP("QGuiApplication", "QT_LAYOUT_DIRECTION");`
  - And defining RTL or LTR as a "translation"

Locale-Aware Applications

- `Locale` makes your applications locale-aware
  - Supports converting numbers to strings with different locale formatting
  - Possible to get the system locale and set the default locale
  - `var fin = Qt.locale("fi_FI");`
- Number formats
  - Use %L1 to format a parameter according to the current locale
  - `qsTr("%L1 ").arg(someNumber);`
- Date and time format
  - `qsTr("Today is %1").arg(Date().toLocaleString(Qt.locale()));`
- Currencies
  - `var hugeAmountOfMoney = 12.97;`
  - `var moneyString = hugeAmountOfMoney.toLocaleCurrencyString(fin, Locale.currencySymbol(Locale.CurrencySymbol)); // CurrencyIsoCode, CurrencyDisplayName`

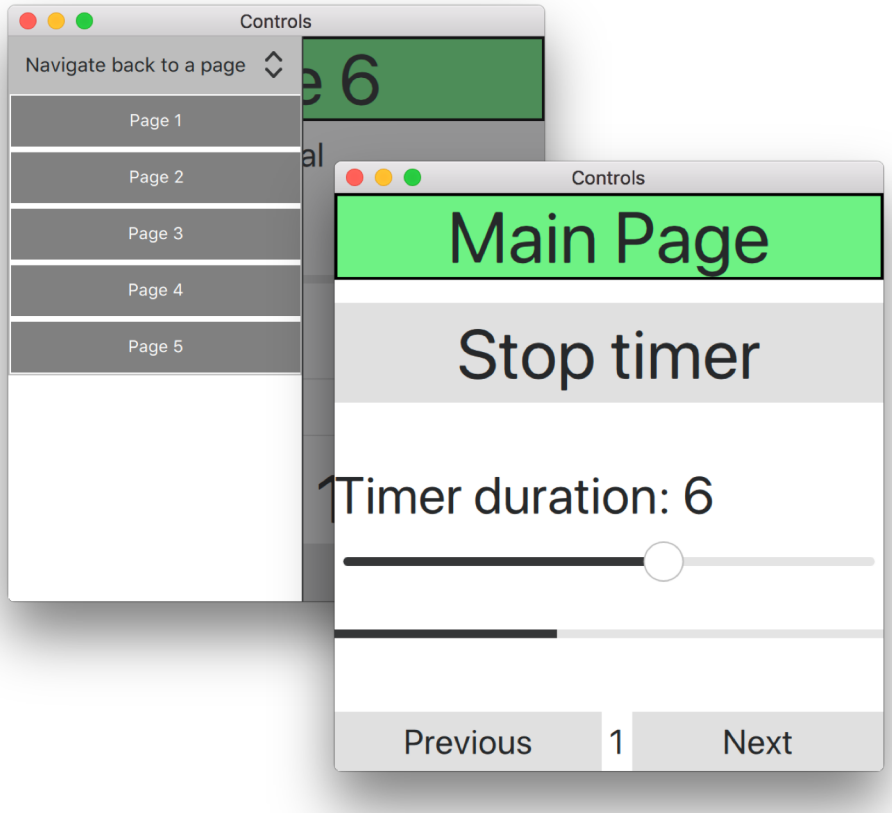
Best Practices

- Mark all your strings localizable
  - Localization functions apply to strings
  - For example role names are not just strings => cannot localize with localization functions
- To support automatic translation installation
  - Use `QQmlApplicationEngine` and provide the .qm modules
- To make deployment easier, put translations into the resource system
- `LayoutMirroring` is useful to localize existing applications
  - Make new applications locale-aware

Some Controls

Demo: controls/controls
https://doc.qt.io/qt-5.10/qtquickcontrols2-guidelines.html

- `Button`
  - Can be clicked or touched
  - Properties: `action` with shortcuts, `autoExclusive`, `display` (icon, text or both)
  - Multi-touch support, like in `Slider`
- `Slider`
  - Used to select a value by sliding a handle
  - Properties: `from`, `to`, `position`, `value`, `handle`, `snapMode`
  - Live value update (default), like in `RangeSlider` and `Dial`
- `ComboBox`
  - Combined button and popup list for selection options
  - Can be editable



Container Controls

- Support adding, inserting, moving, and removing items
  - Additional properties: `currentIndex` and `currentItem`
- No visual presentation
  - Defined by the `contentItem` property
- Container items are defined using `contentModel` default property
  - All children are assigned to the `contentModel` default property
- `Page` is another container (not `Container` sub-type) having a header and a footer

```
Container {
    id: container
    contentItem: ListView {
        model: container.contentModel
    }
    Image { source: "qrc:/images/page1_image" }
    Image { source: "qrc:/images/page2_image" }
```

Dynamic Management of Container Items

```
footer: TabBar {
    id: tabBar
    currentIndex: container.currentIndex
    TabButton {
        text: qsTr("+")
        onClicked: tabBar.addItem(tabButton.createObject(tabBar));
    }
    Component {
        id: tabButton
        TabButton {
            text: qsTr("I'm removed by clicking")
            onClicked: tabBar.removeItem(tabBar.currentIndex);
        }
    }
}
```

Demo: controls/controls

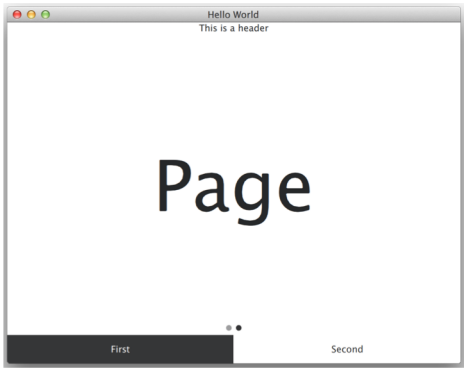
Views – StackView

- Allows user to push, pop, and replace pages in the stack
- Only the top-most item visible
- Several pages may be pushed in one function call, only the topmost created
- Custom animations may be defined for view transitions

```
StackView {
    id: stackView
    initialItem: page
    pushEnter: Transition {
        ParallelAnimation {
            RotationAnimation { from: 0; to: 360 }
            NumberAnimation { properties: "opacity"; from: 0.0; to: 1.0;
                              easing.type: Easing.InOutQuad }
    }
}
Component {
    id: page
```

Views – SwipeView

- Swipe trigged page navigation – horizontal or vertical
- Pages may be dynamically added and removed
  - As extends `Container`
- Page indicator helps user to see there are multiple pages
  - Another control added by the developer




```
SwipeView {
    id: swipeView; anchors.fill: parent
    currentIndex: tabBar.currentIndex
    Page { Label { text: qsTr("Page"); anchors.centerIn: parent } } }
    PageIndicator { id: indicator
        count: swipeView.count; currentIndex: swipeView.currentIndex
        anchors.bottom: swipeView.bottom;
        anchors.horizontalCenter: parent.horizontalCenter
    }
}
```

Qt Quick Controls 1 Views

- Split view
  - Lays out items horizontally or vertically using draggable splitters
  - Compare to widget's `QSizePolicy::Expanding`
- Tab view
  - Allows user to select one of the stacked items
  - For example, Settings application
- Scroll view
  - Used to replace `Flickable` or decorate `Flickable`
  - Three item properties: `contentItem, viewport, flikacbleItem`
  - Sub-items `TableView, TextArea`
- Table View

Demo: modelView/gridViewVSTableView

TableView

```
TableView {
    TableViewColumn {
        title: "Btn"
        role: "btnChecked"
        delegate: tableViewDelegate }
    model: simpleModel }
Component {
    id: tableViewDelegate
    Item {
        RadioButton {
            checked: (styleData.value === "false") ? false : true } } }
ListModel { id: simpleModel }
```

- Provides scroll bars as inherits from `ScrollArea`
- Item, row, and column delegates
  - Different delegates are exposed different data using the `styleData` property
- Based on `ListView`
  - No item index selections, for example

Qt Quick Layouts

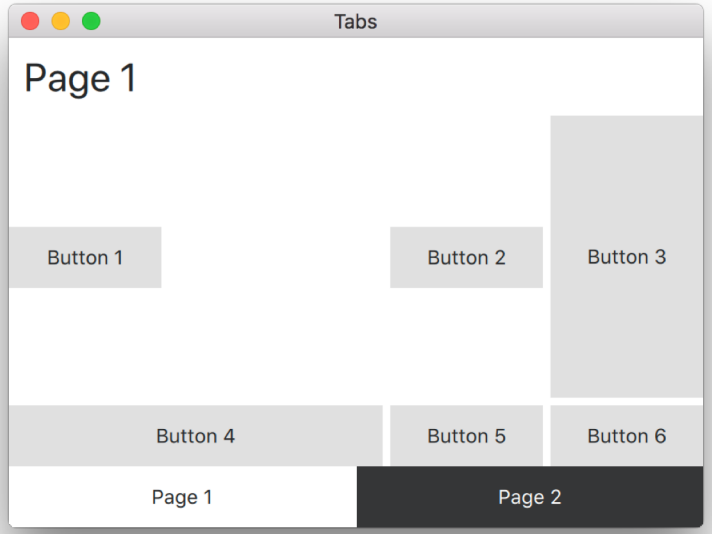
- Default behavior is similar to positioners
- However, can be used in the same way as `QLayout` works for widgets
  - The layout automatically defines the size of the items – no anchors or explicit width/height needed
- Just set the `Layout.fillHeight` or `Layout.fillWidth` to
  - `false` – if you do not want the layout to use all extra space for the item
  - `true` – if you want the extra space to be used to expand the item
  - Compare to `QSizePolicy::Expanding`

Demo: controls/layouts

Layouts Example

- Two buttons expand vertically and horizontally
- Other button dimensions are based on the text and font properties

```
GridLayout {
    columns: 3
    …
    Button {
        text: qsTr("Btn 2")
        Layout.fillHeight: true
        … }
    Button {
        text: qsTr("Btn 4")
        Layout.fillWidth: true
        … }
```



Styling Qt Quick Controls

- Cross-platform icon themes
- Stylable control properties
  - `background`
  - `contentItem`
- Qt Quick templates
  - Stylable control-specific properties
- Configuration file
  - Define, which style and which style-specific property values used

Icon Themes

- Buttons, item delegates, and menu items support icons
- Default icon size and color defined in a style
  - Can be overridden with icon properties

```
icon.name: "File-open"; icon.source: "open.png"
```

- Icon themes allow using the same icons everywhere in the application
  - Set the theme name before loading the QML file

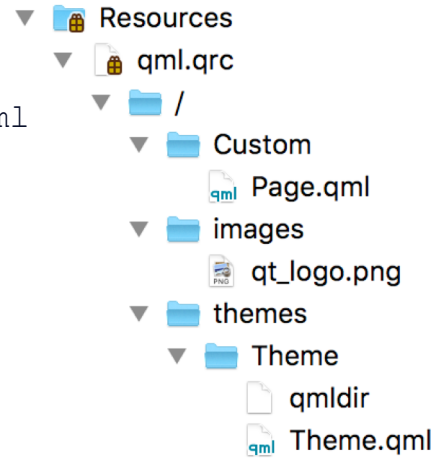
```
QIcon::setThemeName("clusterTheme");
```

```
[Icon Theme]
Name=clusterTheme
Comment=Cluester Icon Theme
Directories=32x32,32x32@2
[32x32]
Size=32
Type=Fixed
[32x32@2]
Size=32
```

```
<RCC>
  <qresource prefix="/">
    <file>icons/desktopTheme/index.theme</file>
    <file>icons/desktopTheme/32x32/open.png</file>
    <file>icons/desktopTheme/32x32@2/open.png</file>
  </qresource>
</RCC>
```

Qt Quick Templates

- Non-visual implementation of control's logic and behavior
- Style agnostic
- Style properties defined in `$QTDIR/qml/QtQuick/Controls.2/ControlType.qml`
  - Implicit dimension, `padding`, `spacing`, icon properties, `contentItem`, `background`



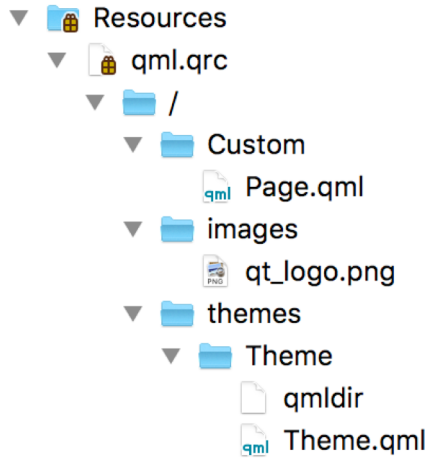
- To create a custom style
  - Create a style folder
  - Copy controls, you want to style, from `$QTDIR/qml/QtQuick/Controls.2`
  - Apply existing style property values, if applicable
  - Implement customizations
  - Apply the style using the configuration file, a command line switch `-style`, an environment variable or set with `QQuickStyle::setStyle("CustomStyle")`

Demo: controls/styling

Custom Page Example

```
import QtQuick 2.9
import QtQuick.Templates 2.2 as T
import QtQuick.Controls.Universal 2.2
import Theme 1.0
T.Page {
    id: control
    property alias backgroundVisible: backgroundImg.visible

    background: Rectangle {
        color: control.Universal.background
        Image {
            id: backgroundImg
            anchors.centerIn: parent
            // All stylable properties in the same file
            source: Theme.backgroundImage
        }
    }
}
```

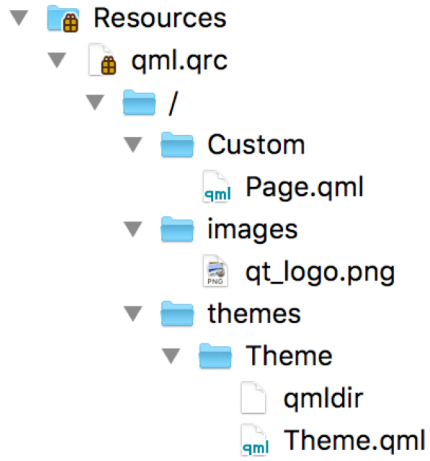


Demo: controls/styling

Theme

- Custom set of stylable properties as `QtObject` children
- Define as singleton
- Typically deployed as a module
  - `module Theme`
  - `singleton Theme 1.0 Theme.qml`

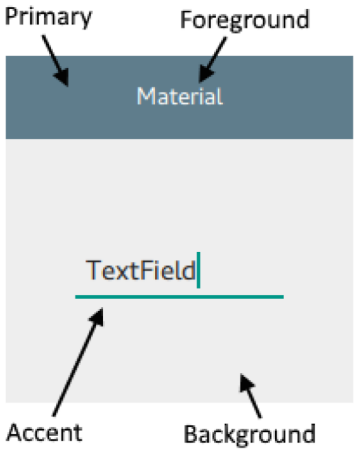
```
pragma Singleton
import QtQuick 2.9
QtObject {
    // Window
    readonly property int windowWidth: 640
    readonly property int windowHeight: 480
    readonly property int windowMinWidth: 380
    readonly property int windowMinHeight: 400
    // Font
    property font defaultFont
    defaultFont.pointSize: 24
```



Qt Quick Controls Styles

Define attached properties, used in controls styling

- Default style
  - Light-weight default style
  - Used as a fallback, if the style does not implement a control
- Imaging style
  - Property `path` defines the image assets location
- Material and Universal styles
  - Google Material and MS Universal Design Guidelines –based styles
  - Properties: accent, background, foreground, theme, primary (only in Material style)
- Fusion style
  - Platform-agnostic style, providing desktop look'n'feel




```
import QtQuick.Controls.Material 2.0
Button {
    text: "Stop"; highlighted: true
    Material.accent: Material.Red
    Material.theme: Material.Dark
}
```

Custom Style Properties

- Declare a `QObject` subclass with
  - a style property `Q_PROPERTY(int styleProperty...)`
  - a factory: `static CoolStyle *qmlAttachedProperties(QObject *object);`
  - `QML_DECLARE_TYPEINFO(CoolStyle, QML_HAS_ATTACHED_PROPERTIES)`
- Register your C++ type for the QML engine
  - `qmlRegisterUncreatableType<CoolStyle>("StyleModule", 42, 0, "CoolStyleName", "Error message")`
- Use the property in custom styling

```
import StyleModule 42.0

Button {
    text: "Button 2"
    CoolStyleName.styleProperty: 23 }
```

Configuration File

- Deployed in resources similarly to QML files
- Configure style and style properties
- Configure default font properties
- Configure palette

```
[Controls]
Style=Custom

[Universal]
Theme=Dark

Font\Family=Courier New
Font\PointSize=24
Font\Style=StyleItalic

[Custom]
Palette\Text=#abcdef
```

Platform, Locale, and Style Variants

- Built-in support for selecting different variants of QML files
  - Based on the file selectors
- Platform

```
:/CustomControl.qml
:/+linux/CustomControl.qml
```

- Locale

```
:/+fi_FI/CustomControl.qml
```

- Style

```
:/+custom/CustomControl.qml
```

- Variants may be combined as well
  - `:/+custom/+fi_FI/CustomControl.qml`

Summary

- Qt Quick Controls provide ready-made UI controls
- `ApplicationWindow` provides a `QQuickWindow` with header, footer, menu bar, and popups
- Window contains a layout of views, containers, and controls
  - Split view, stack view, tab view, scroll view
  - Button, slider, label etc.
- Controls are locale-aware
  - UI strings can be dynamically retranslated
  - Layout direction, date formats, and currency unit change, when the locale changes
- Controls may be styled in three ways
  - With custom background and content items
  - By changing existing style properties using Qt Quick Control templates
  - By modifying the configuration file