Qt Quick Structures Module

Qt Essentials - Training Course

Produced by Nokia, Qt Development Frameworks

Material based on Qt 4.7, created on January 18, 2011

http://qt.nokia.com

Module: Qt Quick Structures

- Components
- Modules

Objectives

- Difference between Custom Items and Components
- How to define Custom Items
- How to define Components
- Properties, Signal/Slots in Components
- Grouping Components to Modules
- Module Versioning
- Using Namespaces

Module: Qt Quick Structures

- Components
- Modules

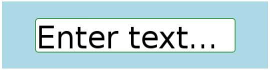
Custom Items and Components

Two ways to create reusable user interface components:

- Custom items
  - defined in separate files
  - one main element per file
  - used in the same way as standard items
  - can have an associated version number
- Components
  - used with models and view
  - used with generated content
  - defined using the `Component` item
  - used as templates for items

Defining a Custom Item

```
import QtQuick 1.0

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

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



- Simple line edit
  - based on undecorated `TextInput`
  - stored in file `LineEdit.qml`

Using a Custom Item

```
import QtQuick 1.0

Rectangle {
    width: 400; height: 100; color: "lightblue"

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

- `LineEdit.qml` is in the same directory
  - item within the file automatically available as `LineEdit`

Demo qml-modules-components/ex-modules-components/lineedit/use-lineedit.qml

Adding Custom Properties

- `LineEdit` does not expose a `text` property
- The text is held by an internal `TextInput` item
- Need a way to expose this text
- Create a custom property

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

Examples:

```
property string product: "Qt Quick"
property int count: 123
property real slope: 123.456
property bool condition: true
property url address: "http://qt.nokia.com/"
```

See Extending types from QML Documentation

Custom Property Example

```
// NewLineEdit.qml
Rectangle {
    ...
    TextInput {
        id: text_input
        ...
        text: "Enter text..."
        ...
    }
    property string text: text_input.text
}
```

- Custom `text` property *binds to* `text_input.text`
- Setting the custom property
  - changes the binding
  - no longer refers to `text_input.text`

Demo qml-modules-components/ex-modules-components/custom-property/NewLineEdit.qml

Property Aliases

```
// AliasLineEdit.qml
Rectangle {
    ...
    TextInput {
        id: text_input
        ...
        text: "Enter text..."
        ...
    }
    property alias text: text_input.text
}
```

- Custom `text` property *aliases* `text_input.text`
- Setting the custom property
  - changes the `TextInput`'s `text`
- Custom property acts like a proxy

Demo qml-modules-components/ex-modules-components/alias-property/AliasLineEdit.qml

Adding Custom Signals

- Standard items define signals and handlers
  - e.g., `MouseArea` items can use `onClicked`
- Custom items can define their own signals

Signal syntax: **`signal <name>[(<type> <value>, ...)]`**
Handler syntax: **`on<Name>: <expression>`**

Examples of signals and handlers:

```
signal clicked
```

- handled by `onClicked`

```
signal checked(bool checkValue)
```

- handled by `onChecked`
- argument passed as `checkValue`

Demo qml-modules-components/ex-modules-components/items/NewCheckBox.qml

Defining a Custom Signal

```
// NewCheckBox.qml
Item {
    ...
    MouseArea {
        ...
        onClicked: if (parent.state == "checked") {
                       parent.state = "unchecked";
                       parent.checked(false);
                   } else {
                       parent.state = "checked";
                       parent.checked(true);
                   }
    }
    signal checked(bool checkValue)
}
```

- `NewCheckBox` item has a `checked` signal
- Communicates a boolean value called `checkValue`

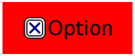
Emitting a Custom Signal

```
// NewCheckBox.qml
Item {
    ...
    MouseArea {
        ...
        onClicked: if (parent.state == "checked") {
                       parent.state = "unchecked";
                       parent.checked(false);
                   } else {
                       parent.state = "checked";
                       parent.checked(true);
                   }
    }
    signal checked(bool checkValue)
}
```

- `MouseArea`'s `onClicked` handler emits the signal
- Calls the signal to emit it

Receiving a Custom Signal

```
import QtQuick 1.0
import "items"

Rectangle {
    width: 250; height: 100; color: "lightblue"

    NewCheckBox {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        onChecked: checkValue ? parent.color = "red"
                              : parent.color = "lightblue"
    }
}
```



- checked signal is handled where the item is used
  - by the `onChecked` handler
  - `on*` handlers are automatically created for signals
  - value supplied using name defined in the signal (`checkValue`)

Demo qml-modules-components/ex-modules-components/use-custom-signal.qml

Module: Qt Quick Structures

- Components
- Modules

Modules

Modules hold collections of elements:

- Contain definitions of new elements
- Allow and promote re-use of elements and higher level components
- Versioned
  - allows specific versions of modules to be chosen
  - guarantees certain features/behavior
- Import a directory name to import all modules within it

See QML Modules Documentation

Custom Item Revisited

```
import QtQuick 1.0

Rectangle {
    width: 400; height: 100; color: "lightblue"

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

- Recall this example from earlier
- `LineEdit.qml` is in the same directory
    - item within the file automatically available as `LineEdit`
- We would like to make different versions of this item
    - to do this, we need to learn about collections of items

Demo qml-modules-components/ex-modules-components/lineedit/use-lineedit.qml

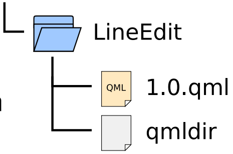
Collections of Items

```
import QtQuick 1.0
import "items"

Rectangle {
    width: 250; height: 100; color: "lightblue"

    CheckBox {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
    }
}
```

- Importing `"items"`
  - imports all files in `items` directory
  - including file `items/CheckBox.qml`
  - the item within is available as `CheckBox`
- Useful for splitting up an application
- Provides the mechanism for versioning of modules

Demo qml-modules-components/ex-modules-components/use-collection-of-items.qml

Versioning Modules



- Create a directory called `LineEdit` containing
  - `1.0.qml` – implementation of the custom item
  - `qmldir` – version information for the module
- The `qmldir` file contains a single line:

```
LineEdit 1.0 LineEdit-1.0.qml
```

- Describes the name of the item exported by the module
- Relates a version number to the file containing the implementation

Using a Versioned Module

```
import QtQuick 1.0
import LineEdit 1.0

Rectangle {
    width: 400; height: 100; color: "lightblue"

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

- Now explicitly import the `LineEdit`
  - using a relative path
  - and a version number

Demo qml-modules-components/ex-modules-components/versioned/use-lineedit-version.qml

Running the Example

- Locate the `declarative-uis/modules-components` directory
- Launch the example:

```
qmlviewer -I versioned versioned/use-lineedit-version.qml
```

- Normally, the module would be installed on the system
  - within the Qt installation's `imports` directory
  - so the `-I` option would not be needed for `qmlviewer`

Supporting Multiple Versions

- Imagine that we release version 1.1 of `LineEdit`
- The example imports version 1.0
  - only the features from that version are required
  - we need to ensure backward compatibility
- `LineEdit` needs to include support for multiple versions
- Version handling is done in the `qmldir` file

```
LineEdit 1.1 LineEdit-1.1.qml
LineEdit 1.0 LineEdit-1.0.qml
```

- Each implementation file is declared
  - with its version
  - in decreasing version order (newer versions first)

Importing into a Namespace

```
import QtQuick 1.0 as MyQt

MyQt.Rectangle {
    width: 150; height: 50; color: "lightblue"

    MyQt.Text {
        anchors.centerIn: parent
        text: "Hello Qt!"
        font.pixelSize: 32
    }
}
```

- `import` ... `as` ...
    - all items in the `Qt` module are imported
    - accessed via the `MyQt` namespace
- Allows multiple versions of modules to be imported

Demo qml-modules-components/ex-modules-components/use-namespace-module.qml

Importing into a Namespace

```
import QtQuick 1.0
import "items" as Items

Rectangle {
    width: 250; height: 100; color: "lightblue"

    Items.CheckBox {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
    }
}
```

- Importing a collection of items from a path
- Avoids potential naming clashes with items from other collections and modules

Demo qml-modules-components/ex-modules-components/use-namespace.qml