Qt Quick Best Practices Part 2

Langston Ball
ICS Consulting Engineer
ICS, Inc.

Agenda

- Creating New Item
- State and Transitions
- Dynamic Creation of Items

Creating New Items

Extending Items

- Any instance of a QML item is effectively a subclass
  - Add member properties
  - Add member functions
  - Add signals
    - There is no reason to add signals to an instance

Extending Items

```
Rectangle{
  Text {
    id: label
    property int count: 1

    text: “Count = ” + count

    function incrementCount() {
        count = count+1
    }
  }

  Timer {
    running: true
    onTriggered: label.incrementCount()
  }
}
```

Property Types

- Properties can be almost any type
  - Basic: int, real, string, date, time, point
    - Copies are stored
  - Any valid QML type: Rectangle, Text
    - These are actually pointers
  - The “var” type is equivalent to Qvariant
    - Use explicit types as much as you can
      - They are faster

Property Types

```
Rectangle{
  id: button

  property int anInt: 1
  property real aDouble: 50.5
  property bool aBool: false
  property string aString: “”
  property var anything: 50.5
  property list<point> points: [ Qt.point(0,0), Qt.point(100,100) ]
}
```

Dividing Code Into Components

- Often a devs to put too much code in one QML file
  - Common issue for all programming languages
  - QML makes it easy to componentize your code
- Component refers to an item that can be instanced multiple times

Creating New Items

- Simply create a new .qml file
  - Type is named after the filename
    - Must begin with a capital letter
  - Implement
    - Properties
    - Signals
    - Functions

Inline Button Code

```
Rectangle{ // Main.qml
  id: toplevel
  color: “black”

  Rectangle {
    id: button
    width: 100; height: 50
    color: “blue”

    Text {
        text: “Click Me”
        }

        MouseArea {
          anchors.fill: parent
          onPressedChanged: button.color = (pressed) ? “red” : “blue”
          onClicked: toplevel.color = “white”
        }
  }
}
```

Componentized Button

```
Rectangle{ // Main.qml
  id: toplevel
  color: “black”

  Button {
     text: “Click Me”
     onClicked: toplevel.color = “white”
  }
}
```

Componentized Button

```
Rectangle{ // Button.qml
  id: button
  property alias text: label.text
  signal clicked()

  color: “blue”
  width: 100; height: 50

  Text {
    id: label
    anchors.centerIn: parent
  }

  MouseArea{
     id: ma
     anchors.fill: parent
     onClicked: button.clicked()
  }
}
```

Alias Properties

- Proxies properties to child items
  - Allows hiding of implementation details
  - Saves memory and binding recalculations

Property Scope

- Public Scope
  - All public properties of the root item
    - Custom properties defined on the root item
- Private Scope
  - All child items and their properties

Public Members

```
Rectangle{ // Button.qml
  id: button
  property alias text: label.text
  signal clicked()

  color: “blue”

  Text {
    id: label
    anchors.centerIn: parent
  }

  MouseArea{
     id: ma
     anchors.fill: parent
     onClicked: button.clicked()
  }
}
```

Private Members

```
Rectangle{ // Button.qml
  id: button
  property alias text: label.text
  signal clicked()

  color: “blue”

  Text {
    id: label
    anchors.centerIn: parent
  }

  MouseArea{
     id: ma
     anchors.fill: parent
     onClicked: button.clicked()
  }
}
```

Private Properties

```
Rectangle { // Button.qml
  id: button
  property alias text: label.text
  signal clicked()

  QtObject {
     id: internal
     property int centerX: button.width()/2
  }

  Text {
     x: internal.centerX
  }
}
```

Avoid Inheriting Public Members

```
// Inherit from the basic Item type and hide everything else
Item { // Button.qml
  id: button
  property alias text: label.text
  signal clicked()

  Rectangle {
     id: background
     color: “blue”
  }

  Text {
    id: label
    anchors.centerIn: parent
  }
  ...
}
```

States and Transitions

States

- State Machines can make your code “more declarative”
  - A basic state machine is built into every Item
    - No parallel states or state history

States

- Every Item has a states property
  - States contain
    - Name
    - When Clause
    - List of PropertyChanges{} objects

Setting States

- Item can be set to a give state two ways
  - 1) “state” property is set to the name of the State
    - item.state = “Pressed”
  - 2) The when clause of the State is true
    - When clauses must be mutually exclusive
      - They are evaluated in creation order

Button States

```
Item {
  Rectangle { id: bkg; color: “blue” }
  MouseArea { id: ma }

  states: [
    State {
        name: “Pressed”
        when: ma.pressed
        PropertyChanges { target: bkg; color: “red” }
    },
    State {
        name: “Disabled”
        when: !(ma.enabled)
        PropertyChanges { target: bkg; color: “grey” }
    }
  ]
}
```

Default State

- The initial bindings are the “Default State”
  - The name of the default state is “”
  - Default state is in effect when
    - No when clauses are satisfied
    - “state” property is set to “”

Properties When in a State

- The bindings of a QML document is defined as
  - The default state bindings
  - Overlaid with PropertyChanges from the current state
  - This will save you a ton of typing
    - States do not need to be unwound
    - Set common properties in the default state
      - Avoids writing duplicate PropertyChanges

Transitions

- Run animations on a state change
  - Control how properties will change
    - Qt will automatically interpolate values
  - Control in which order properties change

Transitions

```
[ ... ]
  transitions: [
    Transition {
      from: “”; to: “Pressed”
      PropertyAnimation { target: bkg
                          properties: “color”
                          duration: 500
       },
    Transition {
      from: “*”; to: “Disabled”
         PropertyAnimation { target: bkg
                          properties: “color”
                          duration: 250
    }
  ]
[ ... ]
```

Transition Defaults

- Transition{} defaults to
  - from: “*”; to: “*”
  - That Transition will apply to all state changes
- PropertyAnimation
  - When a target is not specified
    - That animation will apply to all items

Button Transition

```
Item {
  Rectangle { id: bkg; color: “blue” }
  MouseArea { id: ma }

  states: [
    State { name: “Pressed”; when: ma.pressed
              PropertyChanges { target: bkg; color: “red” }
    },
    State { name: “Disabled”; when: !(ma.enabled)
              PropertyChanges { target: bkg; color: “grey” }
    }
  ]
  transitions: [
    Transition {
       PropertyAnimation { properties: “color”; duration: 500 }
    }
  ]
}
```

Dynamic Creation of Items

Creating Items Dynamically

- Procedural Way
  - Component createObject(parent, bindings) function
- Declarative Way
  - Loader Item
  - Repeater Item
  - ListView / GridView Items

Procedural Creation

```
Item {
  id: screen
  property SettingDialog dialog: undefined

  Button {
     text: “Settings...”
     onClicked: {
        var component = Qt.createComponent(“SettingsDialog.qml”)
        screen.dialog = component.createObject(screen,
                                               { anchors.centerIn: screen
})
        screen.dialog.close.connect(screen.destroySettingsDialog)
     }

  function destroySettingsDialog()
  {
    screen.dialog.destroy()
    screen.dialog = undefined
  }
}
```

Procedural / Declarative Creation

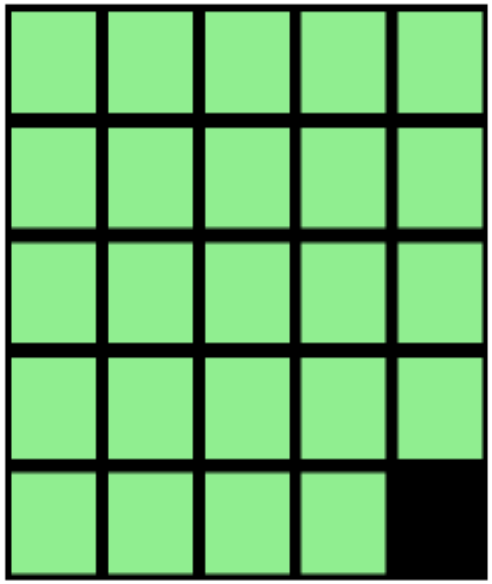
```
Item {
  id: screen

  Button {
     text: “Settings...”
     onClicked: {
            dialogComponent.createObject(screen)
     }

  Component {
    id: dialogComponent

    SettingsDialog {
        anchors.centerIn: parent
        onClose: destroy()
    }
  }
}
```

Declarative Creation

```
Item {

  Button {
     text: “Settings...”
     onClicked: loader.sourceComponent = dialogComponent

  Loader {
     id: loader
     anchors.fill: parent
  }

  Component {
    id: dialogComponent
    SettingsDialog {
        anchors.centerIn: parent
        onClose: loader.sourceComponent = undefined
    }
  }
}
```

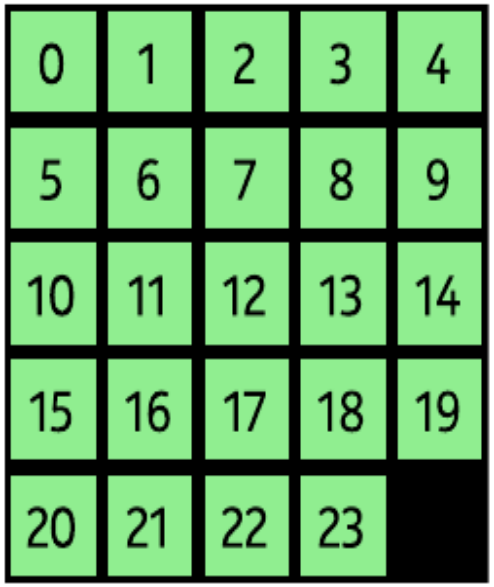
Creating Multiple Items

```
Item {
  width: 400; height: 400
  color: “black”

  Grid {
    x: 5; y:5
    rows: 5; columns: 5

    Repeater {
      model: 24
      Rectangle {
          width: 70; height: 70
          color: “lightgreen”
      }
    }
  }
}
```



Creating Multiple Items

```
Item {
  width: 400; height: 400
  color: “black”

  Grid {
    x: 5; y:5
    rows: 5; columns: 5

    Repeater {
      model: 24
      Rectangle {
          width: 70; height: 70
          color: “lightgreen”
          Text {
              anchors.centerIn: parent
              text: index
          }
      }
    }
  }
}
```



Repeater

- Repeaters can use all type of data models
  - ListModel
  - JSON Data
  - property list<type>
  - QList<QObject*>
  - QAbstractItemModel
- Model data is accessed via attached properties

Thank You!

Part III: July 23rd, 2015

Http://www.ics.com/webinars

Langston Ball
ICS Consulting Engineer
ICS, Inc.