ICS

QML Programming Fundamentals and Beyond

States and Transitions

Material based on Qt 5.12

Course Outline

Session 1: April 28, Introduction to QML
- About QML
- Properties
- Basic Types

Session 2: May 1, QML Item Placement
- How to correctly size and place items
- When to use Anchors, Layouts and Positioners

Session 3: May 5, Touch Interaction
- QML Signals
- Touch Events
- Single and Multi-Touch
- Swipe and Pinch Gestures

Session 4: May 8, States & Transitions
- Creating and defining states
- Using Transitions

Session 5: May 15, Custom Items & Components
- Creating your own Components
- Creating a Module

Session 6: May 19, Model / View
- Model / View
- QML Models
- QML Views

Session 7: May 22, C++ Integration
- Why expose C++ to QML
- Exposing C++ Objects
- Exposing C++ Classes

About ICS

ICS Designs User Experiences and Develops Software for Connected Devices

- Largest source of independent Qt expertise in North America since 2002
- Headquartered in Waltham, MA with offices in California, Canada, Europe
- Includes Boston UX, ICS' UX design division

- Embedded, touchscreen, mobile and desktop applications
- Exclusive Open Enrollment Training Partner in North America

Qt Service Partner

BOSTON UX

UX/UI Design and Development for Connected Devices Across Many Industries

Agenda

- Defining and using States in QML
- How to add Transition behavior between states

States

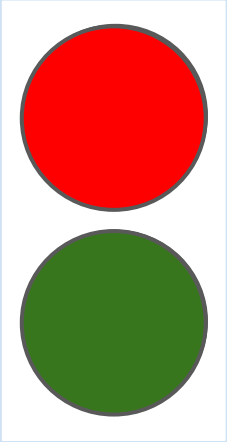
States manage named items

- Represented by the **State** QML type
- Each item can define a set of states
  - With the **states** property
  - Current state is set with the **state** property
- Properties are set when a state is entered
  - Can also modify anchors
  - Change the parents of items
  - Run scripts

States Example

```
Item {
    id: rootItem
    anchors.fill: parent;

    Rectangle {
        id: stopLight
        ...
    }
    Rectangle {
        id: goLight
        ...
    }
```



- Prepare each item with an **id**
- Set up properties not modified by states

Defining States

```
states: [
    State {
        name: "stop"
          ...
    },
    State {
        name: "go"
          ...
    }
]
```


- Define states with names: “**stop**” and “**go**”
- Set up properties for each state with **PropertyChanges**
  - Defining differences from the default values

Changing Properties

Properties are changed with the **PropertyChanges** QML type:

```
State {
    name: "go"
    PropertyChanges { target: stopLight; color: "black" }
    PropertyChanges { target: goLight; color: "green" }
}
```


- Acts on a **target** item named using the target property
  - The **target** refers to an **id** of the intended item
- Applies the many property definitions to the target item
  - One **PropertyChanges** instance can redefine multiple properties
- Property definitions are evaluated when a state is entered
  - **PropertyChanges** describes the new property values for an item
  - New values are assigned to items when the state is entered
  - Properties left unspecified are assigned their default values
- **AnchorChanges** / **ParentChange** is used to change anchors/parent.

Example Continued

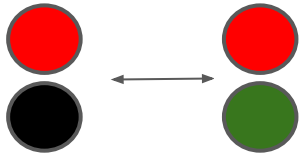
```
// continuing rootItem body

states: [
    State {
        name: "stop"
        PropertyChanges { target: stopLight; color: "red" }
        PropertyChanges { target: goLight; color: "black" }
    },
    State {
        name: "go"
        PropertyChanges { target: stopLight; color: "black" }
        PropertyChanges { target: goLight; color: "green" }
    }
]
```

Setting the State

- Define an initial state of the `rootItem`:

```
state: "stop"
```



- Add a **MouseArea** to the `rootItem` to switch between states:

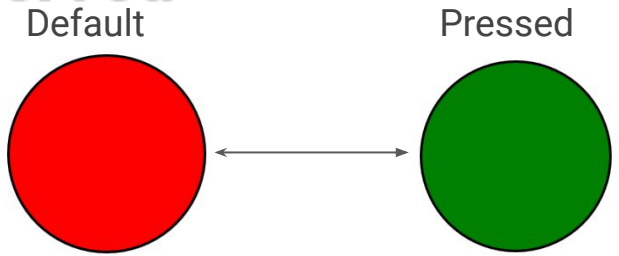

```
MouseArea {
    anchors.fill: parent
    onClicked: parent.state == "stop" ?
               parent.state = "go" : parent.state = "stop"
}
```

- MouseArea reacts to a click on the user interface
- Toggles the parent's **state** property between "**stop**" and "**go**" states

State Conditions

Another way to use states:

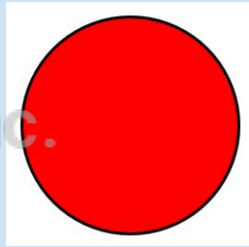
- Define the **`when`** property
  - Using an expression that evaluates to true or false
  - Allows the **`State`** to decide when to be active using conditions evaluated with **`when`**
- Only one state in a **`states`** list should be active
  - Ensure **`when`** is true for only one state



State Conditions Example

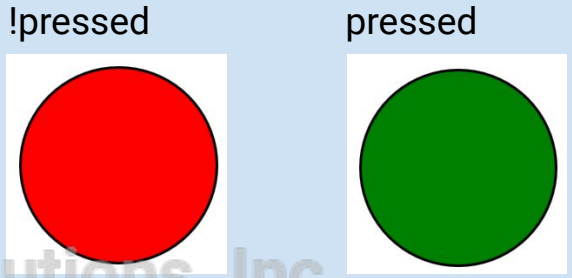
- Define default property values and actions

```
Rectangle {
    id: redCircle
    anchors.centerIn: parent; radius: 360
    width: boxHeight; height: boxHeight
    border.color: "black"; border.width: 3
    color: "red"
    MouseArea {
        id: clickArea
        anchors.fill: parent
    }
}
```



State Conditions Example

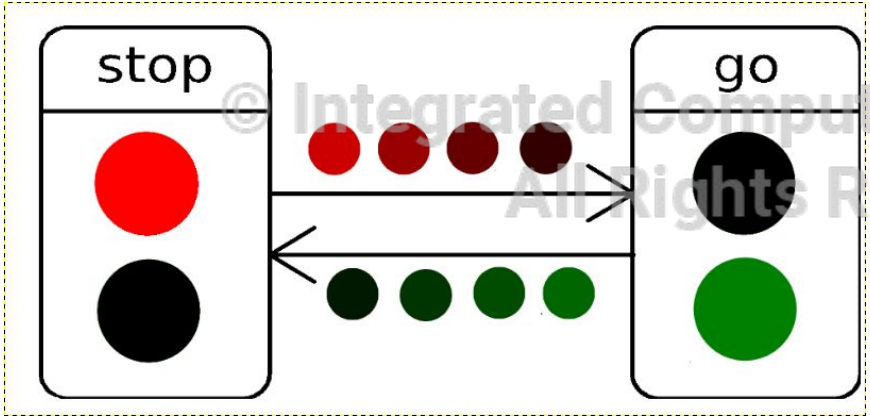
```
states: [
    State {
        name: "greenState"
        when: clickArea.pressed
        PropertyChanges {
            target: redCircle
            color: "green"
        }
    }
]
```

- Only turns green when pressed.
- Do not need to define **state** or a name for this state
- States does not require a name

Transitions

- Define how items change when switching states
- Applies to two or more states
- Usually describes how items are animated



- Let's add transitions to the previous example...

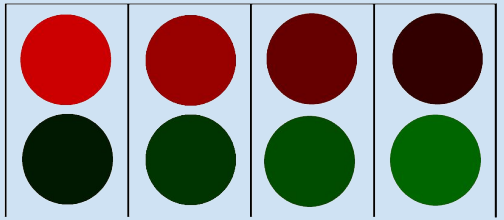
Transitions Example

```
transitions: [
    Transition {
        from: "stop"; to: "go"
        PropertyAnimation {
            targets: [stopLight, goLight]
            properties: "color"; duration: 1000
        }
    },
    Transition {
        from: "go"; to: "stop"
        PropertyAnimation {
            targets: [stopLight, goLight]
            properties: "color"; duration: 1000
        }
    }
]
```


- The **`transitions`** property defines a list of transitions
- A **`Transition`** defines the two states changing **`from`** and **`to`** and the animation to apply for that change.

Reversible Transitions

```
transitions: [
    Transition {
        from: "stop"; to: "go"
        reversible: true
        PropertyAnimation {
            target: stopLight
            properties: "color";
            duration: 1000 }
        PropertyAnimation {
            target: goLight
            properties: "color";
            duration: 1000 }
    }
]
```

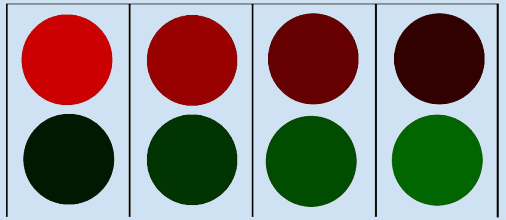


Useful when two transitions operate on the same properties

- Transition applies from "stop" to "go" and back
- Removes the need to define two separate transitions

Wildcard Transitions

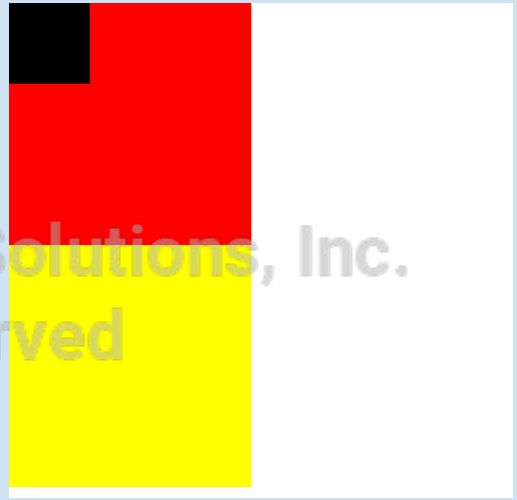
```
transitions: [
    Transition {
        from: "*"; to: "*"
        PropertyAnimation {
            target: stopLight
            properties: "color"; duration: 1000 }
        PropertyAnimation {
            target: goLight
            properties: "color"
            duration: 1000 }
    }
]
```



- Use "*" to represent any state (this is the default for from and to)
- Now the same transition is used whenever the state changes
- Both lights fade at the same time

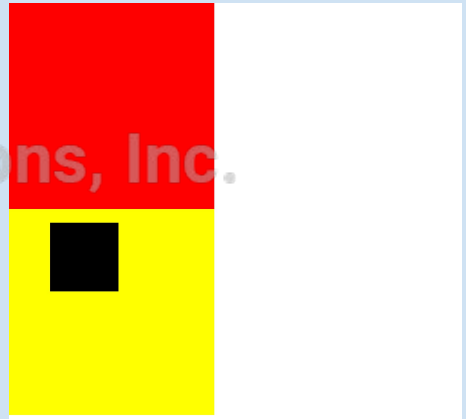
Parent & Anchor Changes

```
Rectangle {
     id: redRectangle
     width: 400; height: 400
     color: "red"
     Rectangle {
          id: blackRectangle
          width: 100; height: 100
          color: "black"
     }
}
Rectangle {
     id: yellowRectangle
     y: redRectangle.height
     width: 400; height: 400
     color: "yellow"
}
```



Parent Changes

```
states: State {
            name: "reanchored"
            ParentChange {
                target: blackRectangle // used to be child of redRectangle
                parent: yellowRectangle
                x: 60; y: 20
            }
        }
transitions: Transition {
            ParentAnimation {
                NumberAnimation {
                    properties: "x,y"
                    duration: 1000
                }
            }
        }
```



- Used to animate an item when its parent changes
- QML type **`ParentAnimation`** applies only when changing the parent with **`ParentChange`** in a state change

Anchor Changes

```
states: State {
                    name: "reanchored"
                    AnchorChanges {
                        target: blackRectangle
                        anchors.left: parent.left
                        anchors.right : parent.right
                    }

                }
transitions: Transition {
                    AnchorAnimation {
                        duration : 1000
                    }
                }
```

- Used to animate an item when its anchors change
- QML type **`AnchorAnimation`** applies only when changing the anchors with **`AnchorChanges`** in a state change

Using States and Transitions

- Avoid defining complex state charts
  - Use more than one state chart to manage the entire UI
  - Define the charts individually for each component
  - Link together components with internal states
- Setting state with script code
  - Easy to do, but might be difficult to manage
- Setting state with state conditions
  - More declarative style
  - Can be difficult to specify conditions
- Using animations in transitions
  - Do not specify `from` and `to` properties
  - Use `PropertyChanges` in state definitions

Q&A Session

If you have additional questions or feedback, please contact us at QtTraining@ics.com