Qt Quick

Team Emertxe

Introduction to Qt Quick

Objectives

- Overview of the Qt library
  - Qt framework presentation
  - Qt Quick inside the Qt framework
- Understanding of QML syntax and concepts
  - Elements and identities
  - Properties and property binding
- Basic user interface composition skills
  - Familiarity with common elements
  - Understanding of anchors and their uses
  - Ability to reproduce a design

Cross-Platform
Framework

Cross-Platform Class Library

One Technology for All Platforms

Integrated Development Tools

Shorter Time-to-Market

Cross-Platform IDE, Qt Creator

Productive development environment

EMERTXE

Qt UI

Qt Quick

C++ on the back, declarative UI design
(QML) in the front for beautiful, modern
touch-based User Experiences.

Qt Widgets

Customizable C++ UI controls for traditional desktop look-and-feel. Also good for more static embedded UIs for more limited devices / operating systems.

Web / Hybrid

Use HTML5 for dynamic web documents,
Qt Quick for native interaction.

EMERTXE

Qt Quick
Requirements

- Platform must support OpenGL ES2
- Needs at least QtCore, QtGui, QtQml, and QtQuick modules
- Other modules can be used to add new features:
  - QtGraphicalEffects: add effects like blur, dropshadow…
  - Qt3D:3D programming in QML
  - QtMultimedia: audio and video items, camera
  - QtWebEngine: web view
  - …

Qt Modules

The Qt framework is split into modules:

- Examples: QtCore, QtGui, QtWidgets, QtNetwork, QtMultimedia...
- Modules contain libraries, plugins and documentation.
- Libraries are linked to your applications
- Libraries group a set of common features (xml, dbus, network...)
- Qt Core is mandatory for all Qt applications

What is Qt Quick?

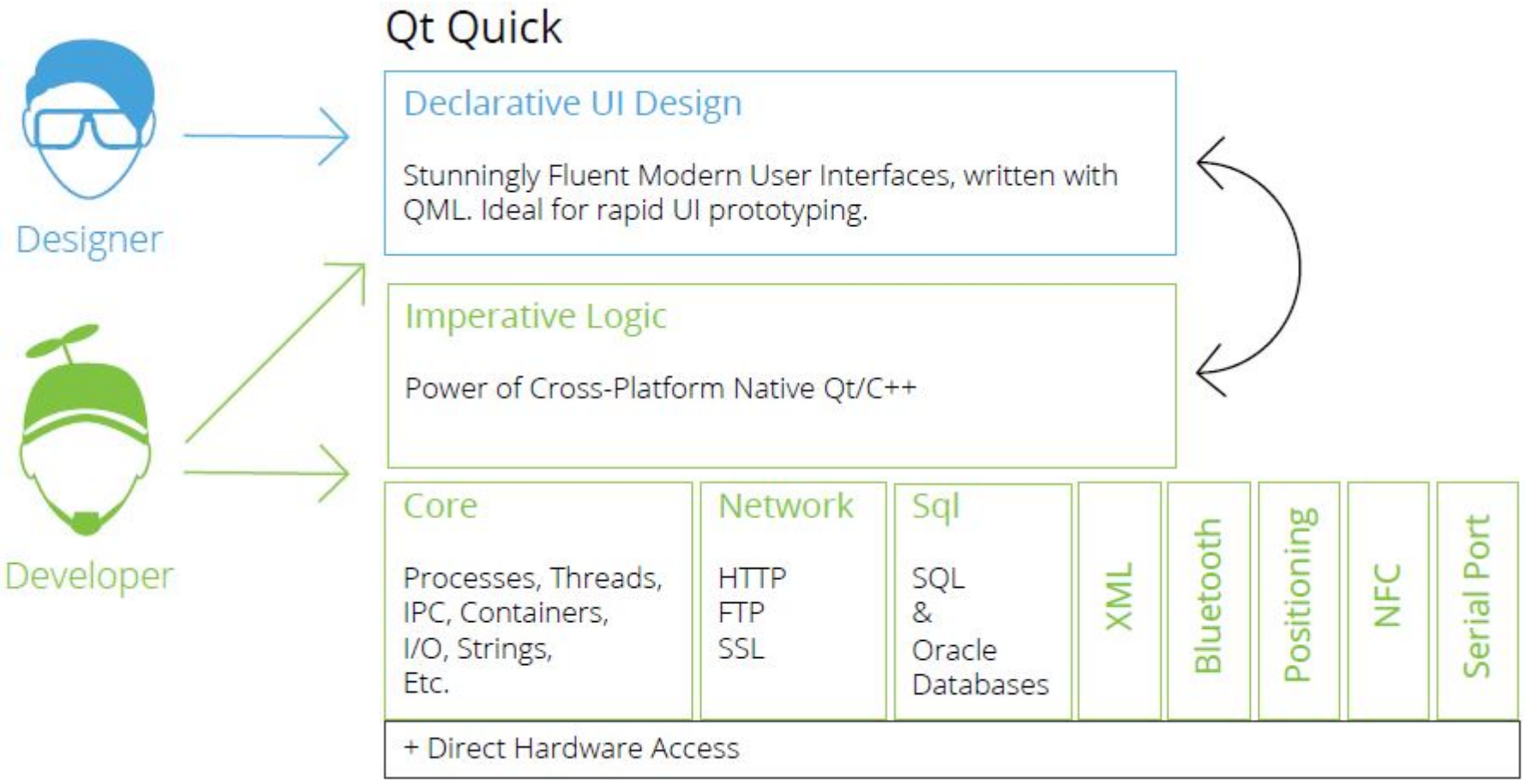
A set of technologies including:

- Declarative markup language: QML
- Imperative Language: JavaScript
- Language runtime integrated with Qt
- C++ API for integration with Qt applications
- QtCreator IDE support for the QML language

Why Qt Quick?

- Intuitive User Interfaces
- Design-Oriented
- Rapid Prototyping and Production
- Easy Deployment
- Enable designer and developers to work on the same sources

Qt Quick
Workflow



EMERTXE

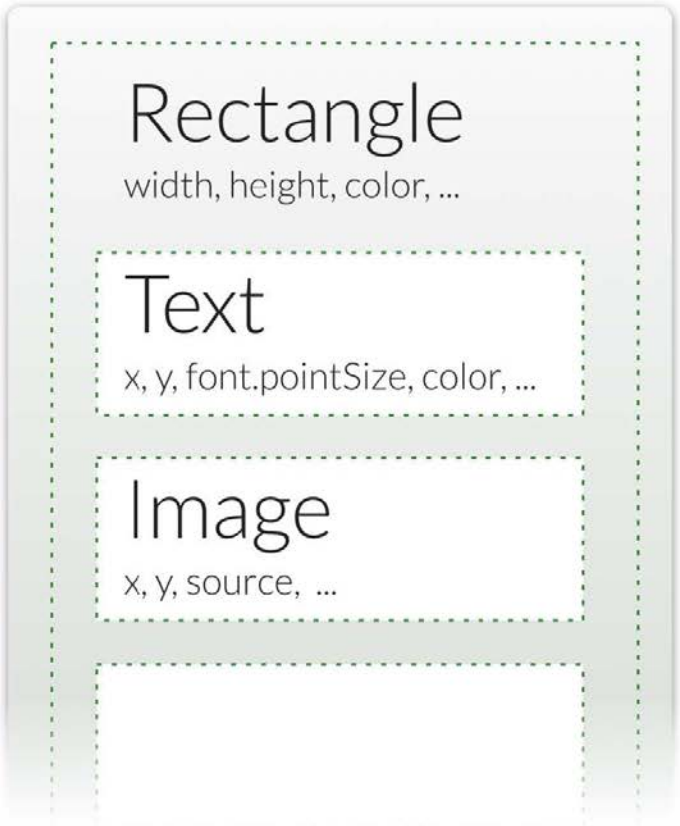
Qt Quick - Concepts

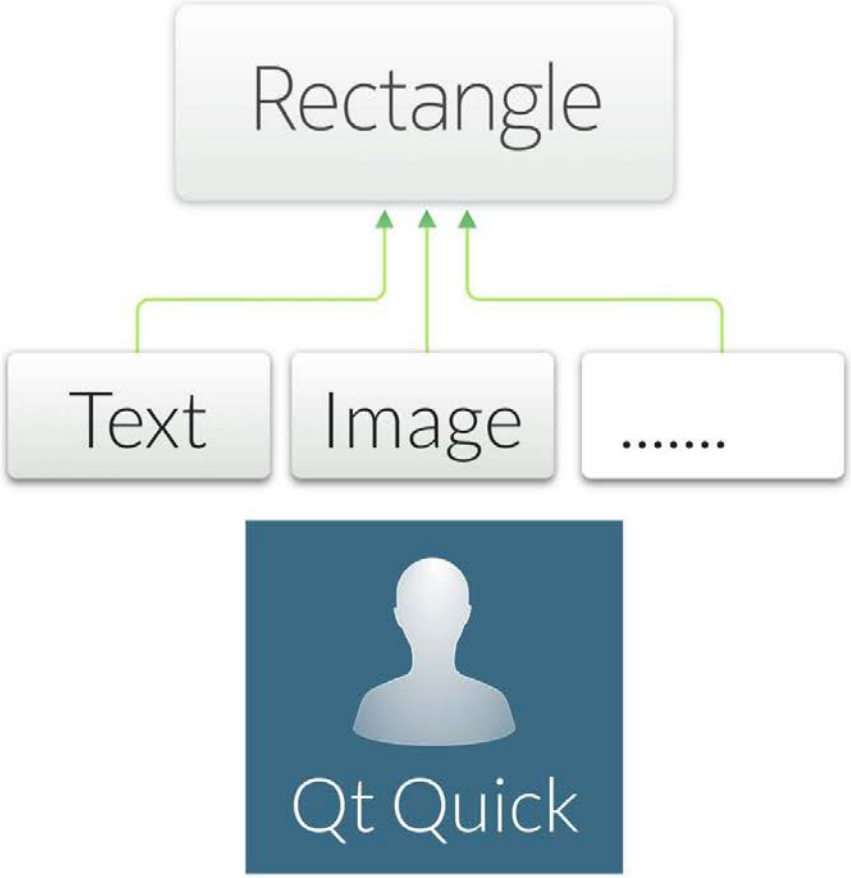
What is QML?

Declarative language for User Interface elements:

- Describes the user interface
  - What elements look like
  - How elements behave
- UI specified as tree of elements with properties

Tree of elements





EMERTXE

Example

```
import QtQuick 2.4

Rectangle {
    width: 400;
    height:
    400 color: “lightblue”"
}
```

- Locate the example: rectangle.qml
- Launch the QML runtime:

  qmlscene rectangle.qml
- Demo

Elements

- Elements are structures in the markup language
  - Represent visual and non-visual parts
- Item is the base type of visual elements
  - Not visible itself
  - Has a position, dimensions
  - Usually used to group visual elements
  - Rectangle, Text, TextInput,...
- Non-visual elements:
  - States, transitions,...
  - Models, paths,...
  - Gradients, timers, etc.
- Elements contain properties
  - Can also be extended with custom properties

Properties

Elements are described by properties:

- Simple name-value definitions
  - width, height, color,…
  - With default values
  - Each has a well-defined type
  - Separated by semicolons or line breaks
- Used for
  - Identifying elements (id property)
  - Customizing their appearance
  - Changing their behavior

Property
examples

- Standard properties can be given values:

```
Text {
    text: "Hello world"
    height: 50
}
```

- Grouped properties keep related properties together:

```
Text {
    font.family: "Helvetica"
    font.pixelSize: 24
    // Prefferred syntax
    // font { family: "Helvetica";
    pixelSize: 24 }
}
```

Property
examples

- Identity property gives the element a name:
  - Identifying elements (id property)
  - Customizing their appearance
  - Changing their behavior

```
Text {
    id: label
    text: "Hello  world"
    height: 50
}
```

- KeyNagivation.tab is not a standard property of TextInput
- Is a standard property that is attached to elements

```
TextInput {
    text: "Hello world"
    KeyNavigation.tab: nextInput
}
```

Property
examples

- Custom properties can be added to any element:

```
Rectangle {
    property real mass:100.0
}
Circle {
    property real radius: 50.0
}
```

EMERTXE

Binding
properties

```
Item {
    width: 400; height: 200
    Rectangle {
        x: 100; y: 50; width: height * 2; height: 100
        color: "lightblue"
    }
}
```

- Properties can contain expressions
  - See above: width is twice the height
- Not just initial assignments
- Expressions are re-evaluated when needed
- Demo

Identifying
Elements

The id property defines an identity for an element

- Lets other elements refer to it
  - For relative alignment and positioning
  - To use its properties
  - To change its properties (e.g., for animation)
  - For re-use of common elements (e.g., gradients, images)
- Used to create relationships between elements

Using Identities

```
Item {
        width: 300; height: 115
        Text {
                id: title x: 50; y: 25 text: "Qt Quick"
                font.family: "Helvetica"; font.pixelSize: 50
        }
        Rectangle {
                x: 50; y: 75; height: 5 width: title.width
                color: "green"
        }
}
```

- Property Text element has the identity, title
- Property width of Rectangle bound to width of title

Demo

Basic types

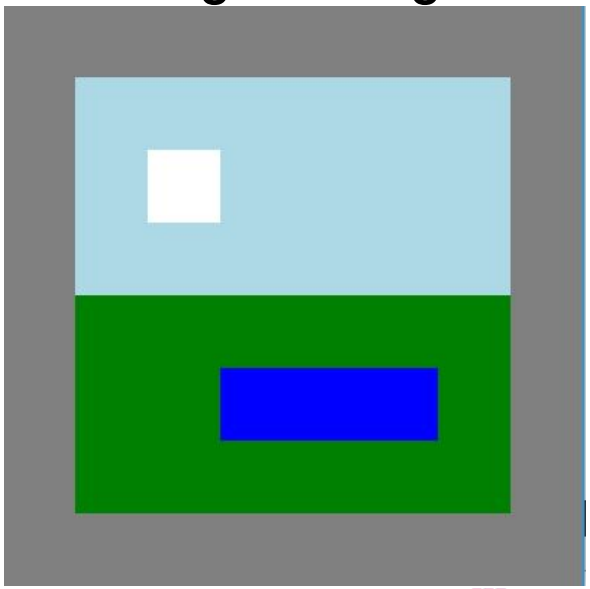
Property values can have different types:

- Numbers (int and real): 400 and 1.5
- Boolean values: true and false
- Strings: “HelloQt”
- Constants: AlignLeft
- Lists:[…]
  - One item lists do not need brackets
- Scripts:
  - Included directly in property definitions
- Other types:
  - colors, dates, rects, sizes, 3Dvectors,…
  - Usually created using constructors

Hands-on

The image on the right shows two items and two child items inside a 400 × 400 rectangle.

1. Recreate the scene using Rectangle items.
2. Can items overlap? Experiment by moving the light blue or green rectangles.
3. Can child items be displayed outside their parents? Experiment by giving one of the child items negative coordinates.



EMERTXE

Summary

- QML defines user interfaces using elements and properties
  - Elements are the structures in QML source code
  - Items are visual elements
- Standard elements contain properties and methods
  - Properties can be changed from their default values
  - Property values can be expressions
  - Id properties give identities to elements
- Properties are bound together
  - When a property changes, the properties that reference it are updated
- Some standard elements define methods
- A range of built-in types is provided

Questions?

- How do you load a QML module?
- What is the difference between Rectangle and width?
- How would you create an element with an identity?
- What syntax do you use to refer to a property of another element?

User interaction

Contents

- Knowledge of ways to receive user input
  - Mouse / touch input
  - Keyboard input
- Awareness of different mechanisms to process input
  - Signal handlers
  - Property bindings

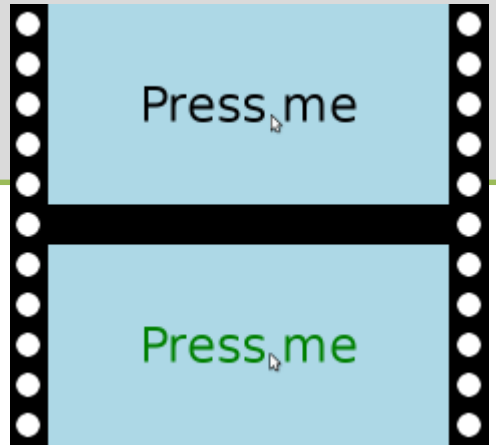
Mouse input

Mouse areas

- Placed and resized like ordinary items
  - Using anchors if necessary
- Two ways to monitor mouse input:
  - Handle signals
  - Dynamic property bindings

Clickable mouse area

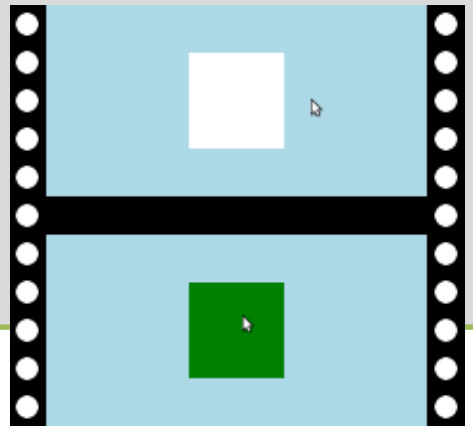
```
Rectangle {
        width: 400; height: 200; color: "lightblue"
        Text {
                anchors.horizontalCenter: parent.horizontalCenter
                anchors.verticalCenter: parent.verticalCenter
                text: "Press me"; font.pixelSize: 48
                MouseArea {
                        anchors.fill: parent
                        onPressed: parent.color = "green"
                        onReleased: parent.color = "black"
                }
        }
}
```



Demo

Mouse Hover
and Properties

```
Rectangle {
    width: 400; height: 200; color: "lightblue"
    Rectangle {
        x: 150; y: 50; width: 100; height: 100
        color: mouseArea.containsMouse ? "green" : "white"
        MouseArea {
            id: mouseArea
            anchors.fill: parent
            hoverEnabled: true
        }
    }
}
```



Demo

Mouse Area Hints
and Tips

- A mouse area only responds to its acceptedButtons
  - The handlers are not called for other buttons, but
  - Any click involving an allowed button is reported
  - The pressedButtons property contains all buttons
  - Even non-allowed buttons, if an allowed button is also pressed
- With hoverEnabled set to false
  - Property containsMouse can be true if the mouse area is clicked

Signals vs
Property Bindings

- Signals can be easier to use in some cases
- When a signal only affects one other item
- Property bindings rely on named elements
- Many items can react to a change by referring to a property
- Use the most intuitive approach for the use case
- Favor simple assignments over complex scripts

Touch Events

- Single-touch (MouseArea)
- Multi-touch (MultiPointTouchArea)
- Gestures
  - Tap and Hold
  - Swipe
  - Pinch

Multi-Touch Events

```
MultiPointTouchArea {
    anchors.fill: parent
    touchPoints: [
        TouchPoint { id: point1 },
        TouchPoint { id: point2 },
        TouchPoint { id: point3 }
    ]
}
```

- TouchPoint properties:
  - int x
  - int y
  - bool pressed
  - int pointId

MultiPointTouchArea

Signals

- onPressed(list<TouchPoint> touchPoints)
- onReleased(...)
  - touchPoints is list of changed points.
- onUpdated(...)
  - Called when points is updated (moved)
  - touchPoints is list of changed points.
- onTouchUpdated(...)
  - Called on any change
  - touchPoints is list of all points.

MultiPointTouchArea

Signals

- onGestureStarted(GestureEvent gesture)
  - Cancel the gesture using gesture.cancel()
- onCanceled(list<TouchPoint> touchPoints)
  - Called when another element takes over touch handling.
  - Useful for undoing what was done on onPressed.
- Demo

Gestures

- Tap and Hold (MouseArea signal onPressAndHold)
- Swipe (ListView)
- Pinch (PinchArea)

Swipe Gestures

- Build into ListView
- `snapMode: ListView.SnapOneItem`
- The view settles no more than one item away from the first visible item at the time the mouse button is released.
- `orientation: ListView.Horizontal`

Pinch Gesture

- Automatic pinch setup using the target property:

```
Image {
    source: "qt-logo.jpg”
    PinchArea {
        anchors.fill: parent
        pinch.target: parent
        pinch.minimumScale: 0.5; pinch.maximumScale: 2.0
        pinch.minimumRotation: -3600; pinch.maximumRotation: 3600
        pinch.dragAxis: Pinch.XAxis
    }
}
```

Demo

Pinch Gesture

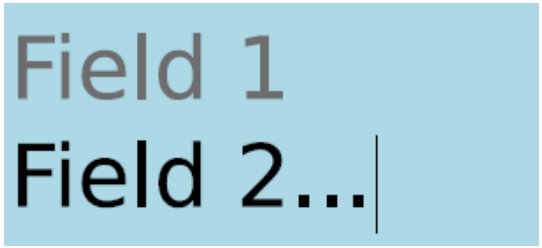
- Signals for manual pinch handling
  - `onPinchStarted(PinchEventpinch)`
  - `onPinchUpdated(PinchEventpinch)`
  - `onPinchFinished()`
- PinchEvent properties:
  - point1, point2, center
  - rotation
  - scale
  - accepted
    - set to false in the onPinchStarted handler if the gesture should not be handled

Keyboard input

- Basic keyboard input is handled in two different use cases:
- Accepting text input
  - Elements TextInput and TextEdit
- Navigation between elements
  - Changing the focused element
  - directional(arrow keys), tab and backtab
- On Slide 28 we will see how to handle raw keyboard input.

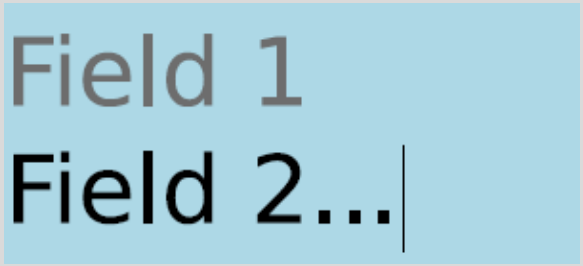
Assigning Focus

- Uis with just one TextInput
  - Focus assigned automatically
- More than one TextInput
  - Need to change focus by clicking
- What happens if a TextInput has no text?
  - No way to click on it
  - Unless it has a width or uses anchors
- Set the focus property to assign focus



Using TextInputs

```
TextInput {
        anchors.left: parent.left; y: 16
        anchors.right: parent.right
        text: "Field 1"; font.pixelSize: 32
        color: focus ? "black" : "gray"
        focus: true
}
TextInput {
        anchors.left: parent.left; y: 64
        anchors.right: parent.right
        text: "Field 2"; font.pixelSize: 32
        color: focus ? "black" : "gray"
}
```



Demo

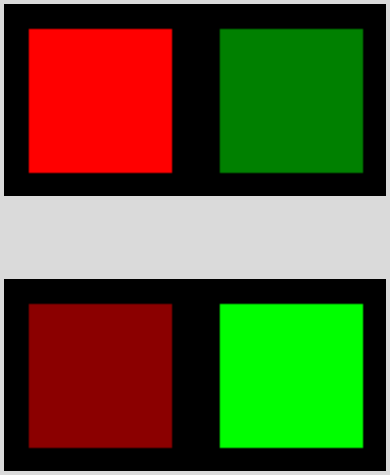
Focus Navigation

```
TextInput {
    id: nameField
    focus: true
    KeyNavigation.tab: addressField
}
TextInput {
    id: addressField
    KeyNavigation.backtab: nameField
}
```

- The name_field item defines KeyNavigation.tab
  - Pressing Tab moves focus to the address_field item
- The address_field item defines KeyNavigation.backtab
  - Pressing Shift+Tab moves focus to the name_field item
- Demo

Key Navigation

```
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
}
```



- Using cursor keys with non-text items
- Non-text items can have focus, too
- Demo

Summary

Mouse and cursor input handling:

- Element MouseArea receives clicks and other events
- Use anchors to fill objects and make them clickable
- Respond to user input:
  - Give the area a name and refer to its properties, or
  - Use handlers in the area and change other named items

Key handling:

- Elements TextInput and TextEdit provide text entry features
- Set the focus property to start receiving key input
- Use anchors to make items clickable
  - Lets the user set the focus
- Element KeyNavigation defines relationships between items
  - Enables focus to be moved
  - Using cursor keys, tab and backtab
  - Works with non-text-input items

Questions?

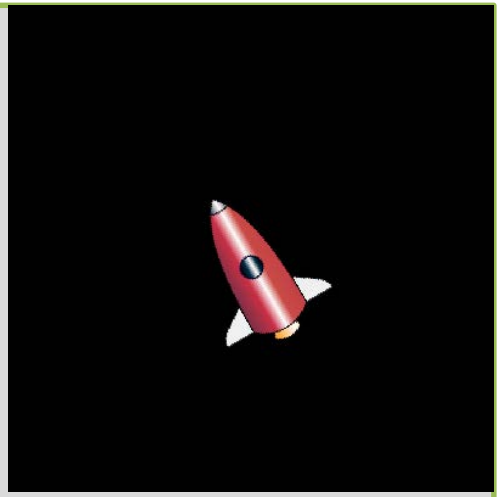
- Which element is used to receive mouse clicks?
- Name two ways TextInput can obtain the input focus.
- How do you define keyboard navigation between items?

Hands-on

Raw Keyboard
Input

- Raw key input can be handled by item
  - With predefined handlers for commonly used keys
  - Full key event information is also available
- The same focus mechanism is used as for ordinary text input
  - Enabled by setting the focus property
- Key handling is not an inherited property of items
  - Enabled using the Keys attached property
- Key events can be forwarded to other objects
  - Enabled using the Keys.forwardTo attached property
  - Accepts a list of objects

Raw Keyboard
Input



```
Rectangle {
    width: 400; height: 400; color: "black"
    Image {
        id: rocket
        x: 150; y: 150
        source: "../images/rocket.svg"
        transformOrigin: Item.Center
    }
    Keys.onLeftPressed: rocket.rotation = (rocket.rotation - 10) % 360
    Keys.onRightPressed: rocket.rotation = (rocket.rotation + 10) % 360
    focus: true
}
```

Raw Keyboard
Input

- Can use predefined handlers for arrow keys:

```
Keys.onLeftPressed: rocket.rotation = (rocket.rotation - 10) % 360
Keys.onRightPressed: rocket.rotation = (rocket.rotation + 10) % 360
```

- Or inspect events from all key presses:

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

Focus Scopes

- Focus scopes are used to manage focus for items
- Property FocusScope delegates focus to one of its children
- When the focus scope loses focus
  - Remembers which one has the focus
- When the focus scope gains focus again
  - Restores focus to the previously active item

Composing UI’s

Objectives

- Elements are often nested
  - One element contains others
  - Manage collections of elements
- Colors, gradients and images
  - Create appealing UIs
- Text
  - Displaying text
  - Handling text input
- Anchors and alignment
  - Allow elements to be placed in an intuitive way
  - Maintain spatial relationships between elements

Why Use?

- Concerns separation
- Visual grouping
- Pixel perfect items placing and layout
- Encapsulation
- Reusability
- Look and feel changes
- Example

Nested elements

```
Rectangle {
    width: 400; height: 400
    color: "lightblue"
    Rectangle {
        x: 50; y: 50; width: 300; height: 300
        color: "green"
        Rectangle {
            x: 200; y: 150; width: 50; height: 50
            color: "white"
        }
    }
}
```

- Each element positioned relative to its parents
- Demo

Graphical elements
Colors

- Specifying colors
  - Named colors (using SVG names): "red", "green", "blue",...
  - HTML style color components: "#ff0000", "#008000", "#0000ff",...
  - Built-in function: Qt.rgba(0,0.5,0,1)
- Changing items opacity:
  - Using the opacity property
  - Values from 0.0 (transparent) to 1.0 (opaque)

Graphical elements

Colors

```
Item {
    width: 300; height: 100
    Rectangle {
        x: 0; y: 0; width: 100; height: 100; color: "#ff0000"
    }
    Rectangle {
        x: 100; y: 0; width: 100; height: 100 color: Qt.rgba(0,0.75,0,1)
    }
    Rectangle {
        x: 200; y: 0; width: 100; height: 100; color: "blue"
    }
}
```

Demo

Graphical elements

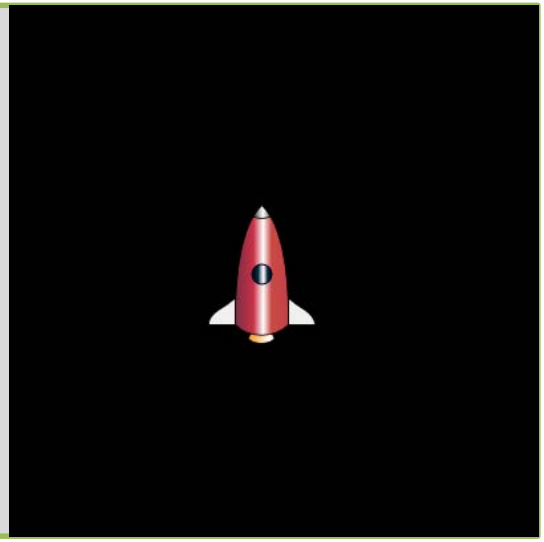
Images

- Represented by the Image element
- Refer to image files with the source property
  - Using absolute URLs
  - Or relative to the QML file
- Can be transformed
  - scaled, rotated
  - About an axis or central point

Graphical elements

Images

```
Rectangle {
    width: 400; height: 400
    color: "black”
    Image {
        x: 150; y: 150
        source: "../images/rocket.png"
    }
}
```



- Property source contains a relative path
- Properties width and height are obtained from the image file
- Demo

ΣMERTXE

Graphical elements

Image scaling

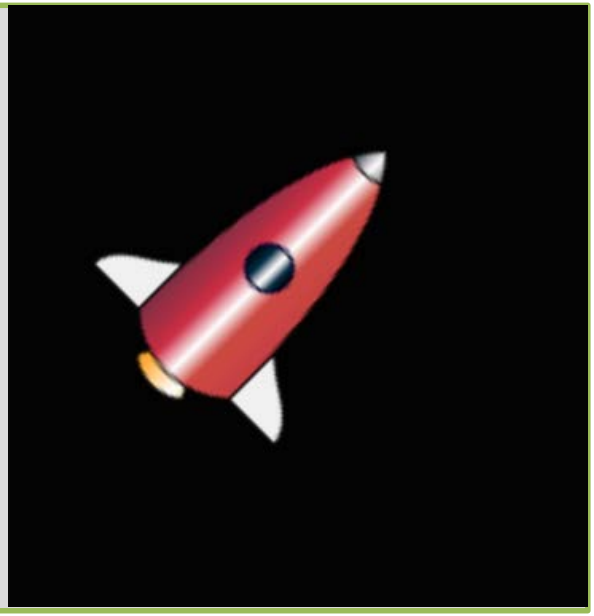
```
Rectangle {
    width: 400; height: 400
    color: "black”
    Image {
        x: 150; y: 150
        source: "../images/rocket.png"
    }
}
```

- Property source contains a relative path
- Properties width and height are obtained from the image file
- Demo

Graphical elements

Image rotation

```
Rectangle {
    width: 200; height: 200
    color: "black”
    Image {
        x: 50; y: 30
        source: "../images/rocket.png“
        rotation: 45.0
    }
}
```

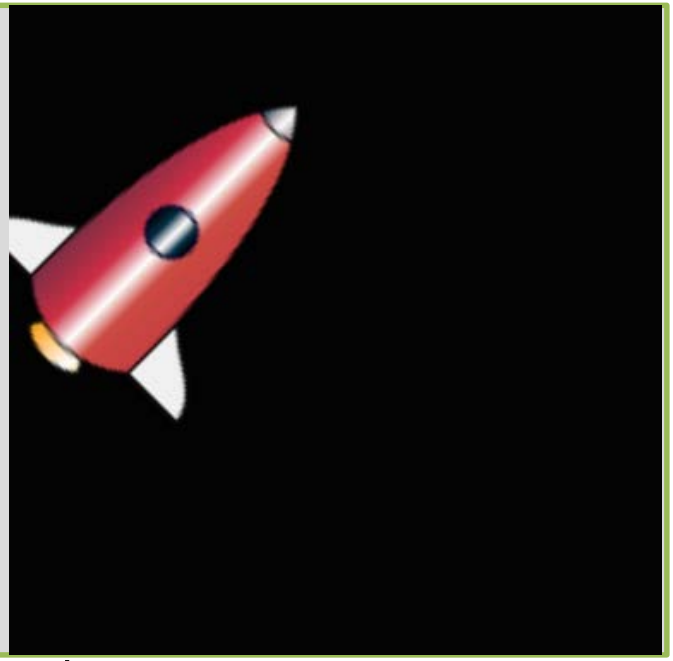


- Set the rotate property
- By default, the center of the item remains in the same place
- Demo

Graphical elements

Image rotation

```
Rectangle {
    width: 200; height: 200
    color: "black”
    Image {
        x: 50; y: 30
        source: "../images/rocket.png“
        rotation: 45.0
        transformOrigin: Item.Top
    }
}
```



- Set the `transformOrigin` property
- Now the image rotates about the top of the item
- Demo

ΣMERTXE

Gradients

Define a gradient using the gradient property:

- With a Gradient element as the value
- Containing GradientStop elements, each with
  - A position: a number between 0 (startpoint) and 1 (endpoint)
  - A color
- The start and end points
  - Are on the top and bottom edges of the item
  - Cannot be repositioned
- Gradients override color definitions
- Alternative to gradients: A simple background image.

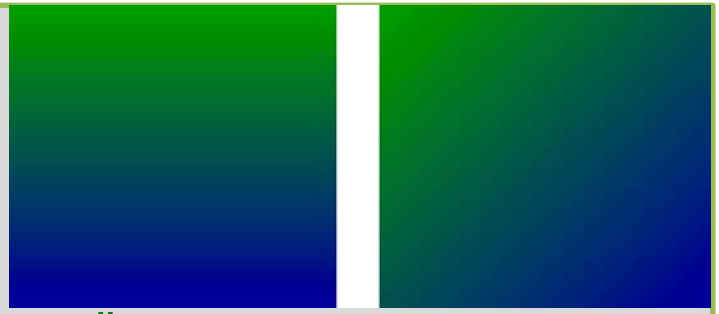
Gradients

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

- Note the definition of an element as a property value
- Demo

Gradient Images

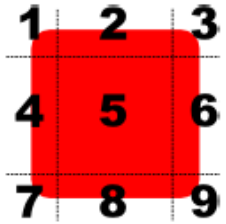
```
Rectangle {
    width: 425; height: 200
    Image {
        x: 0; y: 0
        source: "../images/vertical-gradient.png"
    }
    Image {
        x: 225; y: 0; source: "../images/diagonal-gradient.png"
    }
}
```



- It is often faster to use images instead of real gradients
- Artists can create the desired gradients
- Demo

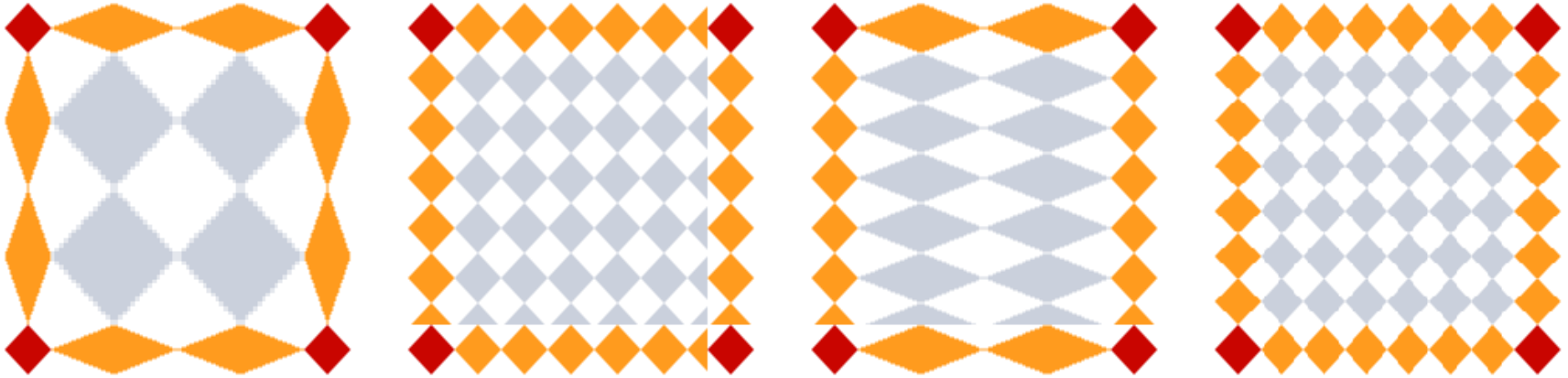
ΣMERTXE

Border Images



- Create border using part of an image:
  - Corners (region 1,3,7,9) are not scaled
  - Horizontal borders (2 and 8) are scaled according to `horizontalTileMode`
  - Vertical borders (4 and 6) are scaled according to `verticalTileMode`
  - Middle region (5) is scaled according to both modes
- There are 3 different scale modes
  - `Stretch:` scale the image to fit to the available area.
  - `Repeat:` tile the image until there is no more space.
  - `Round:` like `Repeat`, but scales the images down to ensure that the last image is not cropped

Border Images



```
BorderImage {
        source: "content/colors.png"
        border { left: 30; top: 30; right: 30; bottom: 30; }
        horizontalTileMode: BorderImage.Stretch
        verticalTileMode: BorderImage.Repeat
        // ...
}
```

Demo

Text elements

```
Rectangle {
    width: 400; height: 400
    color: "lightblue”
    Text {
        x: 100; y: 100
        text: "Qt Quick"
        font.family: "Helvetica”; font.pixelSize: 32
        }
}
```

- Width and height determined by the font metrics and text
- Can also use HTML tags in the text:
  - `"<html><b>Qt Quick</b></html>“`
- Demo

ΣMERTXE

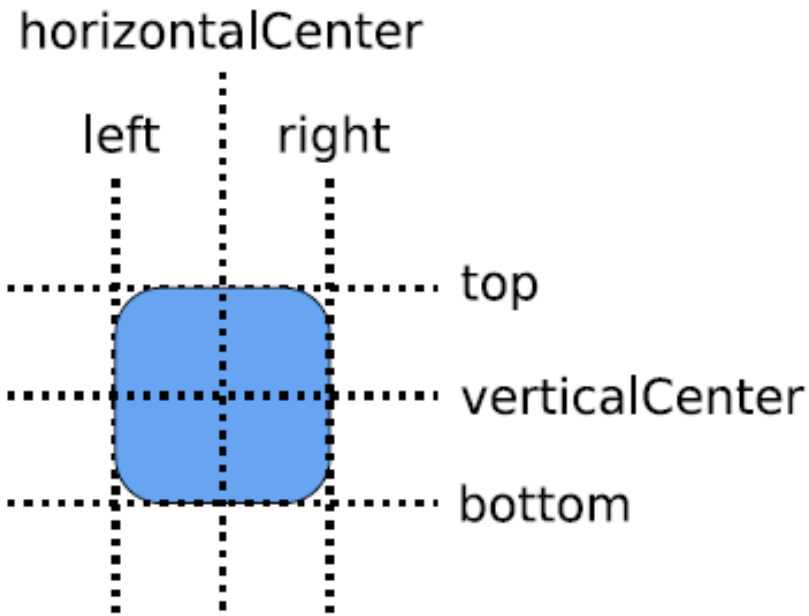
TextInput

```
TextInput {
        x: 50; y: 100; width: 300
        text: "Editable text"
        font.family: "Helvetica"; font.pixelSize: 32
}
```

- No decoration (not a QLineEdit widget)
- Gets the focus when clicked
  - Need something to click on
- Property text changes as the user types
- Demo

Anchor layout

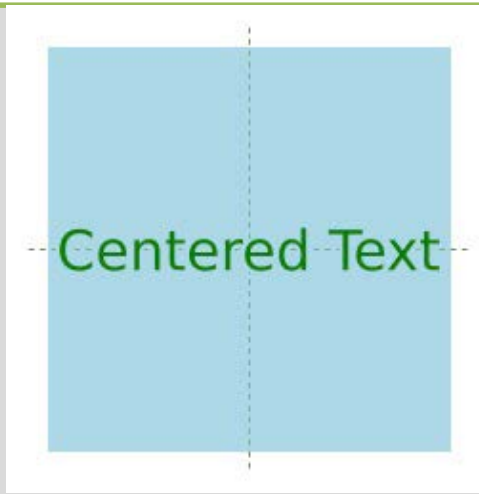
- Used to position and align items
- Line up the edges or central lines of items
- Anchors refer to
  - Other items (centerIn, fill)
  - Anchors of other items (left, top)



ΣMERTXE

Anchors

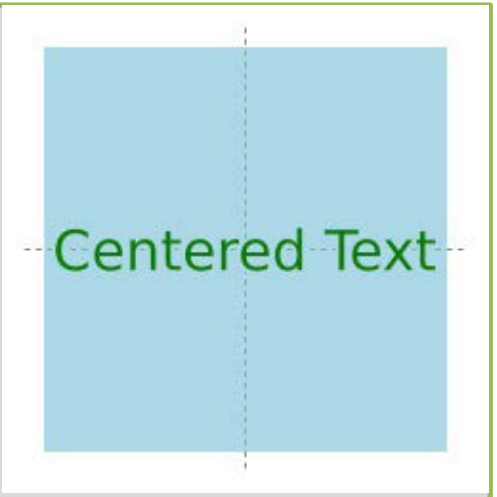
```
Rectangle {
    width: 400; height: 400
    color: "lightblue"
    id: rectangle1
    Text {
        text: "Centered text"; color: "green"
        font.family: "Helvetica"; font.pixelSize: 32
        anchors.centerIn: rectangle1
    }
}
```



- `anchors.centerIn` centers the `Text` element in the Rectangle
  - Refers to an item not an anchor
- Demo

Anchors

```
Text {
        text: "Centered text";
        color: "green"
        font.family: "Helvetica";
        font.pixelSize: 32
        anchors.centerIn: parent
        }
}
```



- Each element can refer to its parent element
  - Using the parent ID
- Can refer to ancestors and named children of ancestors
- Demo

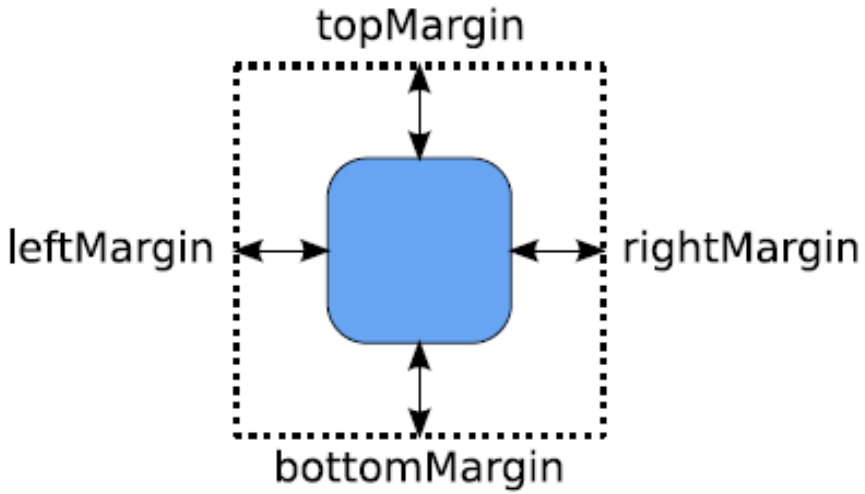
Anchors

```
Text {
    y: 34
    text: "Right-aligned text"; color: "green"
    font.family: "Helvetica"; font.pixelSize: 32
    anchors.right: parent.right
}
```

- Connecting anchors together
- Anchors of other items are referred to directly
  - Use `parent.right`
  - Not `parent.anchors.right`
- Demo

Margins

- Used with anchors to add space
- Specify distances
  - In pixels
  - Between elements connected with anchors



Margins

```
Rectangle {
    width: 400; height: 200; color: "lightblue"
    Image {
        id: book; source: "../images/book.svg"
        anchors.left: parent.left
        anchors.leftMargin: parent.width/16
        anchors.verticalCenter: parent.verticalCenter
    }
    Text {
        text: "Writing"; font.pixelSize: 32
        anchors.left: book.right anchors.leftMargin: 32
        anchors.baseline: book.verticalCenter
    }
}
```

Writing

Demo

Hints and Tips

- Anchors can only be used with parent and sibling items
- Anchors work on constraints
  - Some items need to have well-defined positions and sizes
  - Items without default sizes should be anchored to fixed or well-defined Items
- Anchors create dependencies on geometries of other items
  - Creates an order in which geometries are calculated
  - Avoid creating circular dependencies
    - e.g.,parent → child→parent
- Margins are only used if the corresponding anchors are used
  - e.g., leftMargin needs left to be defined

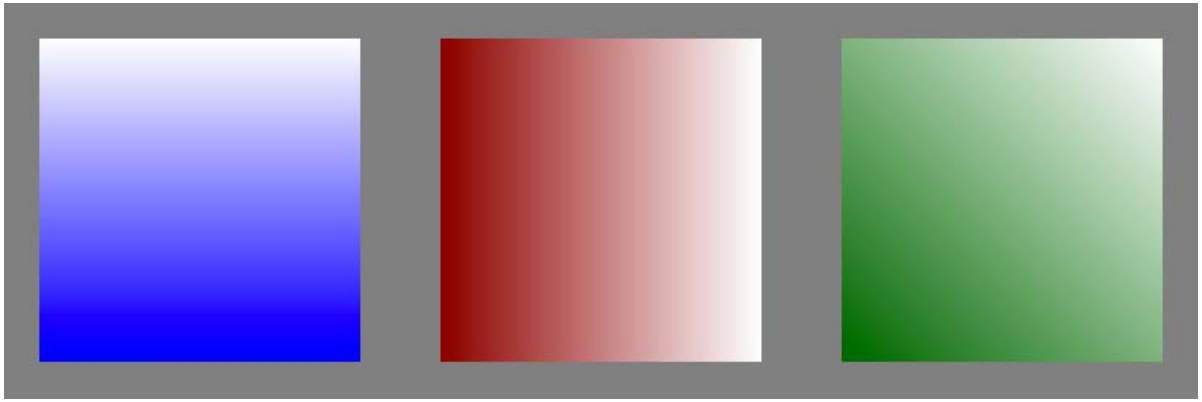
Strategies for Use

Identify item with different roles in the user interface:

- Fixed items
  - Make sure these have id properties defined
  - Unless these items can easily be referenced as parent items
- Items that dominate the user interface
  - Make sure these have id properties defined
  - Items that react to size changes of the dominant items
  - Give these anchors that refer to the dominator fixed items

Questions?

1. How else can you write these colors?
   - "blue"
   - "#ff0000"
   - Qt.rgba(0,0.5,0,1)
2. How would you create these items using the gradient property?



3. Describe another way to create these gradients?

Questions?

1. When creating an Image, how do you specify the location of the image file?
2. By default, images are rotated about a point inside the image. Where is this point?
3. How do you change the text in a Text element?

Hands-on

QML Structures

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

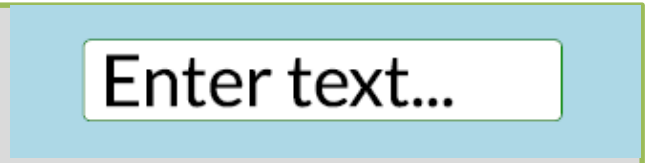
Custom Items and Components

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

```
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

```
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

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>]`**

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

Custom Property
Example

```
Rectangle {
    ...
    TextInput {
        id: textInput
        ...
        text: "Enter text..."
    }
    property string text: textInput.text
}
```

- Custom `text` property binds to `text_input.text`
- Setting the custom property
  - Changes the binding
  - No longer refer to `text_input.text`
- Demo

Property aliases

```
Rectangle {
    ...
    TextInput {
        id: textInput
        ...
        text: "Enter text..."
    }
    property alias text: textInput.text
}
```

- Custom text property aliases text_input.text
- Setting the custom property
  - Changes the TextInput's text
- Demo

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`

Defining a Custom Signal

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

Demo

ΣMERTXE

Emitting a Custom Signal

```
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 "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"
    }
}
```

- Signal checked is handled where the item is used
  - By the `onCheckedhandler`
  - `on*` handlers are automatically created for signals
  - Value supplied using name defined in the signal (checkValue)
- Demo

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

Custom Item
Revisited

```
Rectangle {
    width: 400; height: 100; color: "lightblue"
    LineEdit {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        width: 300; height: 50
    }
}
```

- Element `LineEdit.qml` is in the same directory
- We would like to make different versions of this item so we need collections of items
- Demo

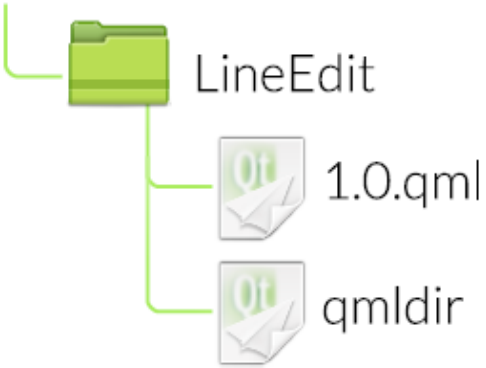
Collections of
Items

```
import "items"
    Rectangle {
        width: 250; height: 100; color: "lightblue"
        CheckBox {
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.verticalCenter: parent.verticalCenter
        }
}
```

- Importing “items” directory
- Includes all the files (e.g. items/CheckBox.qml)
- Useful to organize your application
- Provides the mechanism for versioning of modules
- Demo

Versioning Modules

- Create a directory called LineEdit containing
  - `LineEdit-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 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

Running the
Example

- Locate `qml-modules-components/ex-modules-components`
- Launch the example:
  - `qmlscene -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 `qmlscene`

Supporting
Multiple Versions

- Imagine that we release version `1.1` of `LineEdit`
- 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 2.4 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

Importing into a Namespace

```
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

State & transitions

Objectives

Can define user interface behavior using states and transitions:

- Provides a way to formally specify a user interface
- Useful way to organize application logic
- Helps to determine if all functionality is covered
- Can extend transitions with animations and visual effects
- States and transitions are covered in the Qt documentation

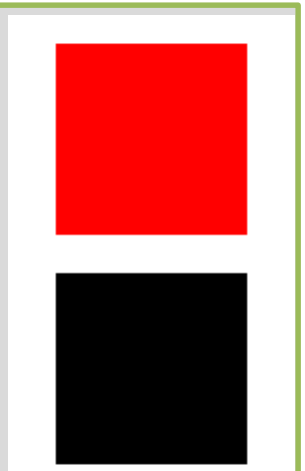
States

States manage named items

- Represented by the State element
- 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

State example

```
Rectangle {
    width: 150; height: 250
    Rectangle {
        id: stopLight
        x: 25; y: 15; width: 100; height: 100
    }
    Rectangle {
        id: goLight
        x: 25; y: 135; width: 100; height: 100
    }
}
```



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

ΣMERTXE

Defining states

```
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" }
    }
]
```

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

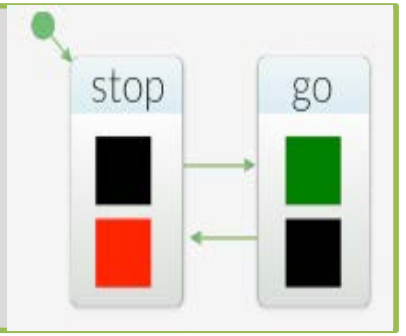
Setting the State

- Define an initial state:

```
state: "stop"
```

- Use a `MouseArea` to switch between states:

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



- Reacts to a click on the user interface
  - Toggles the parent's `state` property between `“stop”` and `“go”` states

Changing
Properties

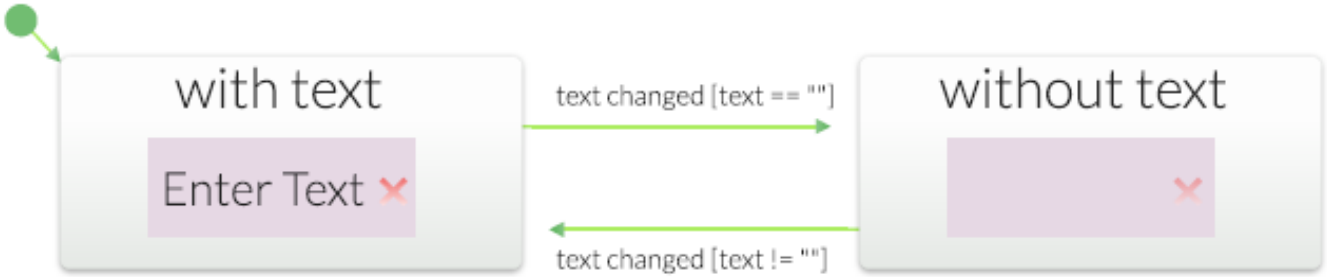
- States change properties with the PropertyChanges element:

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

- Acts on a target element named using the target property
  - The target refers to an id
- Applies the other property definitions to the target element
  - One PropertyChanges element can redefine multiple properties
- Property definitions are evaluated when the state is entered
- PropertyChanges describes 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

State conditions

- Another way to use states:
- Let the State decide when to be active
- Using conditions to determine if a state is active
- Define the when property
- Using an expression that evaluates to true or false
- Only one state in a states list should be active
- Ensure when is true for only one state
- Demo



ΣMERTXE

State Conditions

Example

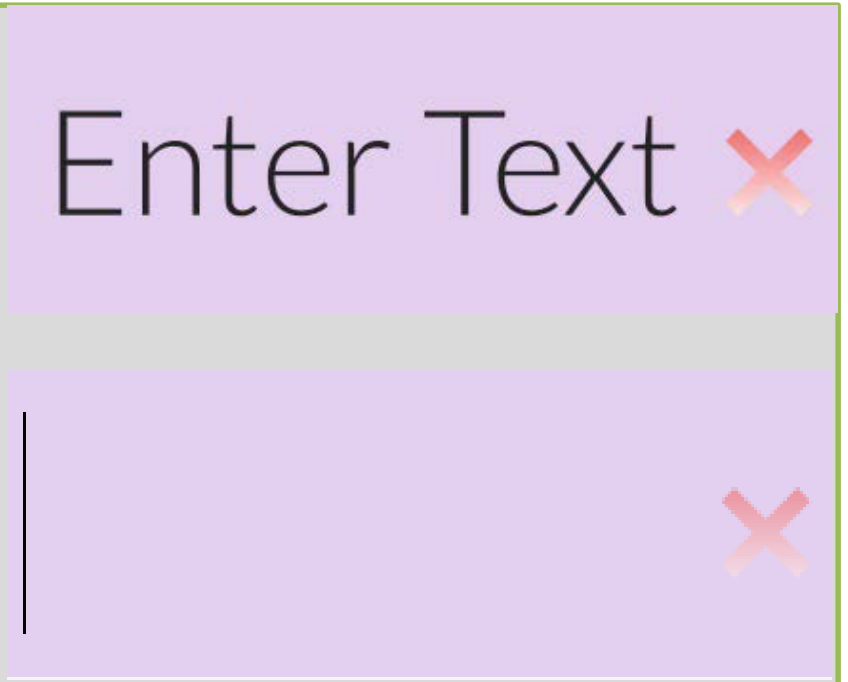
```
TextInput {
        id: textField
        text: "Enter text..."
…           }
Image {
        id: clearButton
        source: "../images/clear.svg”
        …
        MouseArea { anchors.fill: parent
                        onClicked: textField.text = "" }
}
```

- Define default property values and actions

State Conditions

Example

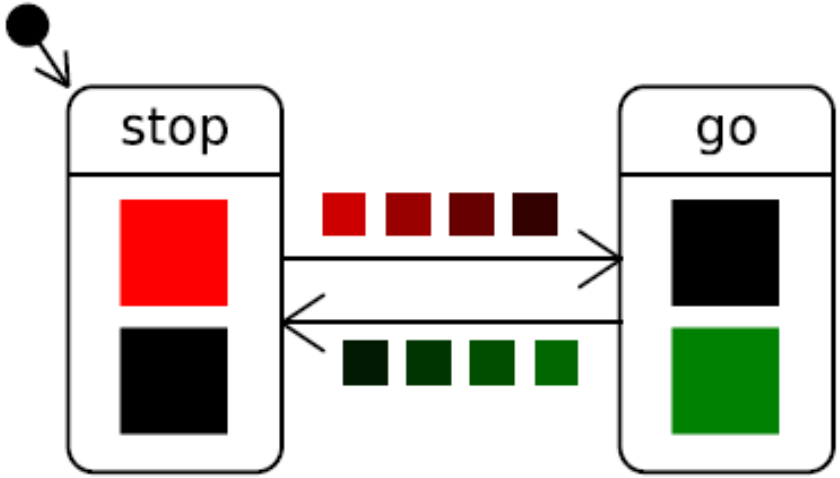
```
states: [
        State {
                name: "with text"
                when: textField.text != ""
                PropertyChanges {
                target: clearButton; opacity: 1.0
                }
        },
        State {
                name: "without text"
                when: textField.text == ""
                PropertyChanges {
                target: clearButton; opacity: 0.25 }
                PropertyChanges {
                target: textField; focus: true }
                }
]
```



- A clear button that fades out when there is no text
- Do not need to define `state`

Transitions

- Define how items change when switching states
- Applied to two or more states
- Usually describe how items are animated



- Let's add transitions to a previous example…
- Demo

Transition

Example

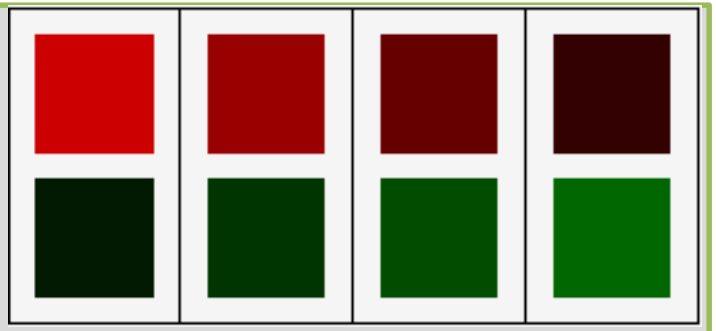
```
transitions: [
        Transition {
                from: "stop"; to: "go"
                PropertyAnimation {
                        target: stopLight
                        properties: "color"; duration: 1000
                }
        },
        Transition {
                from: "go";
     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

- The `transitions` property defines a list of transitions
- Transitions between “`stop`” and “`go`” states

Wildcard

Transitions

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

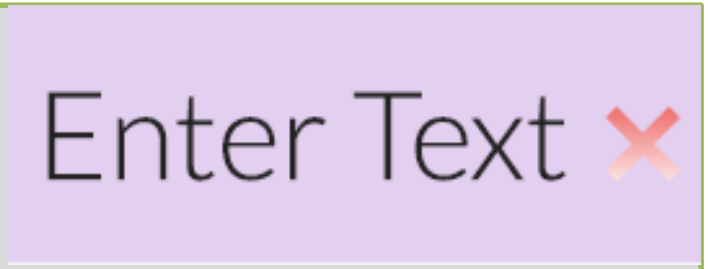


- Use “*” to represent any state
- Now the same transition is used whenever the state changes
- Both lights fade at the same time
- Demo

Reversible

Transitions

```
transitions: [
    Transition {
        from: "with text"; to: "without text"
        reversible: true
        PropertyAnimation {
            target: clearButton
            properties: "opacity";
            duration: 1000
        }
    }
]
```



- Useful when two transitions operate on the same properties
- Transition applies from “with text” to “without text”
  - And back again from “without text” to “with text”
- No need to define two separate transitions
- Demo

Parent Changes

```
states: State {
        name: "reanchored"
        ParentChange {
                target: myRect
                parent: yellowRect
                x: 60; y: 20 }
}
transitions: Transition {
        ParentAnimation {
                NumberAnimation {
                        properties: "x,y"
                        duration: 1000 }
                }
}
```

- Used to animate an element when its parent changes
- Element `ParentAnimation` applies only when changing the parent with `ParentChange` in a state change
- Demo

Anchor Changes

```
states: State {
        name: "reanchored"
        AnchorChanges {
                target: myRect
                anchors.left: parent.left
                anchors.right : parent.right }
        }
        transitions: Transition { AnchorAnimation {
        duration : 1000 }
}
```

- Used to animate an element when its anchors change
- Element `AnchorAnimation` applies only when changing the anchors with `AnchorChanges` in a state change
- Demo

Using States and Transitions

- Avoid defining complex state charts
  - Not just one state chart to manage the entire UI
  - Usually defined 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` elements in state definitions

Summary

States

`State` items manage properties of other items:

- Items define states using the `states` property
  - Must define a unique `name` for each state
- Useful to assign `id` properties to items
  - Use `PropertyChanges` to modify items
- The `state` property contains the current state
  - Set this using JavaScript code, or
  - Define a `when` condition for each state

Summary

Transitions

`Transition` items describe how items change between states:

- Items define transitions using the `transitions` property
- Transitions refer to the states they are between
  - Using the `from` and `to` properties
  - Using a wildcard value, `"*"`, to mean any state
- Transitions can be reversible
  - Used when the `from` and `to` properties are reversed

Questions?

- How do you define a set of states for an item?
- What defines the current state?
- Do you need to define a name for all states?
- Do state names need to be globally unique?
- Remember the thumbnail explorer page? Which states and transitions would you use for it?

Hands-on

QML Animations

Objectives

Can apply animations to user interfaces:

- Understanding of basic concepts
  - Number and property animations
  - Easing curves
- Ability to queue and group animations
  - Sequential and parallel animations
  - Pausing animations
- Knowledge of specialized animations
  - Color and rotation animations

Why use?

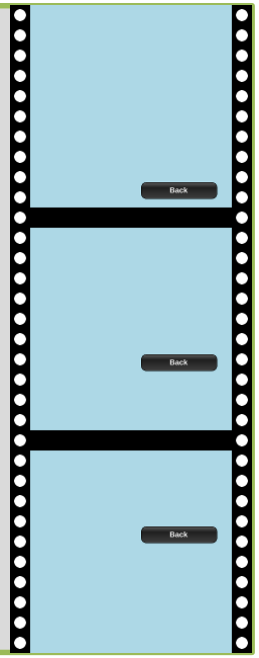
- Handle form factor changes
- Outline application state changes
- Orchestrate high level logic
- Natural transitions
- Our brain expects movement
- Helps the user find its way around the GUI
- Don't abuse them!
- Demo

Animations

- Animations can be applied to any element
- Animations update properties to cause a visual change
- All animations are property animations
- Specialized animation types:
  - `NumberAnimation` for changes to numeric properties
  - `ColorAnimation` for changes to color properties
  - `RotationAnimation` for changes to orientation of items
  - `Vector3dAnimation` for motion in 3D space
- Easing curves are used to create variable speed animations
- Animations are used to create visual effects

Number
Animations

```
Rectangle {
        width: 400; height: 400
        color: "lightblue"
        Image {
                x: 220 source: "../images/backbutton.png"
                NumberAnimation on y {
                        from: 350; to: 150
                        duration: 1000
                        }
        }
}
```



Demo

EMERTXE

Number
Animations

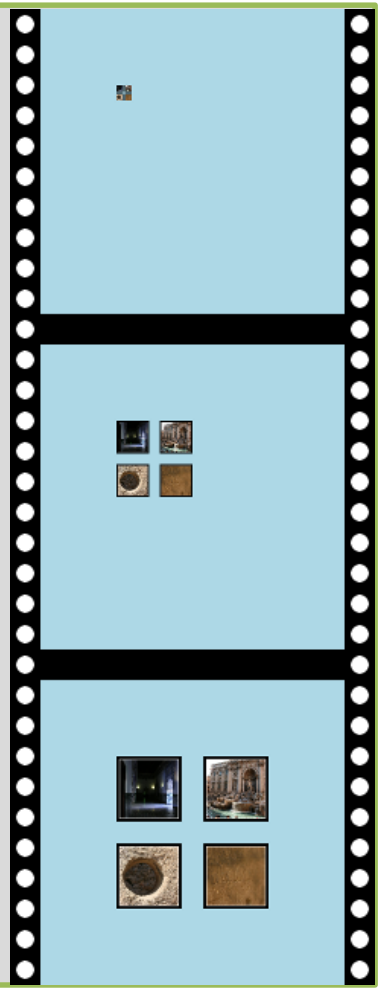
- Number animations change the values of numeric properties

```
NumberAnimation on y {
    from: 350;
    to: 150
    duration: 1000
}
```

- Applied directly to properties with the on keyword
- The y property is changed by the `NumberAnimation`
- Starts at 350
- Ends at 150
- Takes 1000 milliseconds
- Can also be defined separately
- Demo

Property
Animations

```
Rectangle {
        width: 400;
        height: 400;
        color: "lightblue"
        Image {
                id: image
                x: 100; y: 100
                source: "../images/thumbnails.png" }
                PropertyAnimation {
                        target: image
                        properties: "width,height"
                        from: 0; to: 200;
                        duration: 1000
                        running: true
                }
        }
}
```



Demo

ΣMERTXE

Property
Animations

- Property animations change named properties of a target

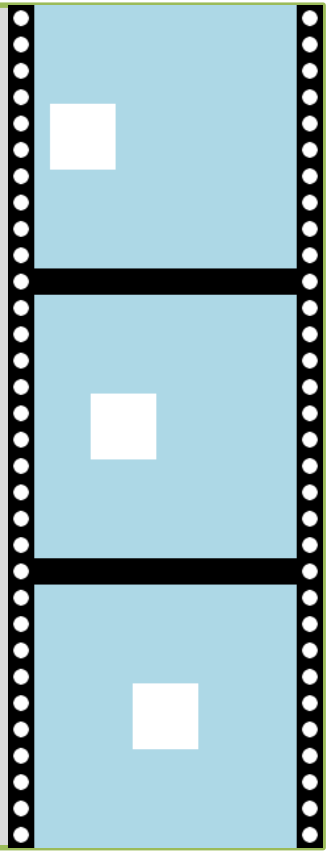
```
PropertyAnimation {
    target: image
    properties: "width,height"
    from: 0; to: 200; duration: 1000
    running: true
}
```

- Defined separately to the target element
- Applied to properties of the `target`
  - Property `properties` is a comma-separated string list of names
- Often used as part of a `Transition`
- Not run by default
- Set the `running` property to true

Number

Animations revisited

```
Rectangle {
        width: 400; height: 400; color: "lightblue"
        Rectangle {
                id: rect
                x: 0; y: 150; width: 100; height: 100
        }
        NumberAnimation {
                target: rect
                properties: "x"
                from: 0; to: 150; duration: 1000
                running: true
        }
}
```



Demo

Number

Animations revisited

Number animations are just specialized property animations

```
NumberAnimation {
    target: rect
    properties: "x"
    from: 0; to: 150; duration: 1000
    running: true
}
```

- Animation can be defined separately
- Applied to properties of the target
  - Property properties contains a comma-separated list of property names
- Not run by default
  - Set the running property to true

The Behavior
Element

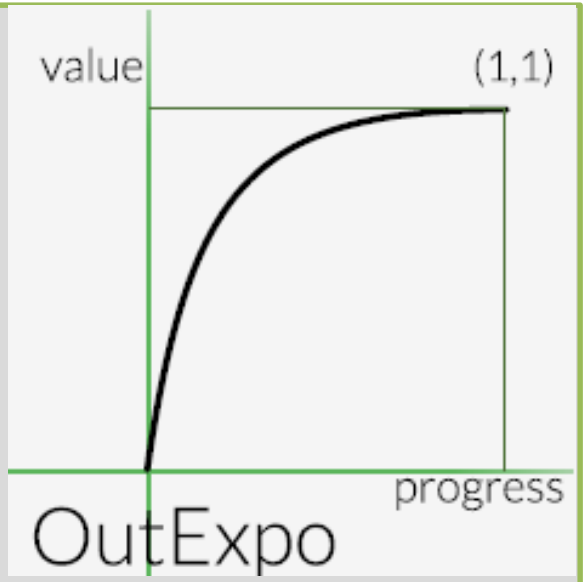
- Behavior allows you to set up an animation whenever a property changes.

```
Behavior on x {
    SpringAnimation { spring: 1; damping: 0.2 }
}
Behavior on y {
    SpringAnimation { spring: 2; damping: 0.2 }
}
```

- Demo

Easing Curves

```
Rectangle {
        width: 400; height: 400
        color: "lightblue"
        Image {
                x: 220
                source: "../images/backbutton.png"
                NumberAnimation on y {
                        from: 0; to: 350
                        duration: 1000
                        easing.type: "OutExpo"
                }
        }
}
```



- Demo

ΣMERTXE

Easing Curves

Apply an easing curve to an animation:

```
NumberAnimation on y {
        from: 0; to: 350
        duration: 1000
        easing.type: "OutExpo"
}
```

- Sets the `easing.type` property
- Relates the elapsed time
- To a value interpolated between the `from` and `to` values
- Using a function for the easing curve
- In this case, the `"OutExpo"` curve

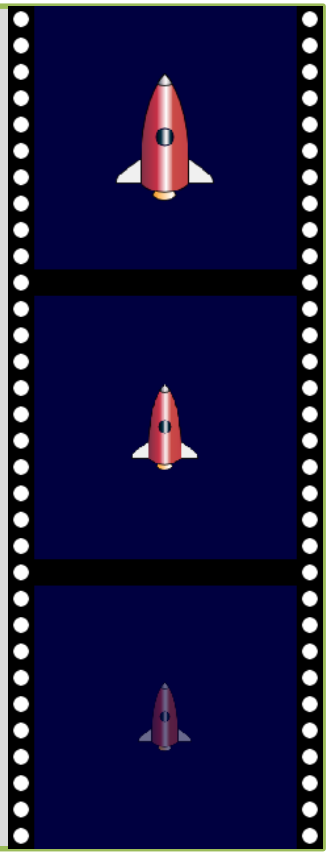
Sequential and Parallel Animations

Animations can be performed sequentially and in parallel

- `SequentialAnimation` defines a sequence
  - With each child animation run in sequence
- For example:
  - A rescaling animation, followed by an opacity changing animation
- `ParallelAnimation` defines a parallel group
  - With all child animations run at the same time
- For example:
  - Simultaneous rescaling and opacity changing animations
- Sequential and parallel animations can be nested

Sequential
Animations

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

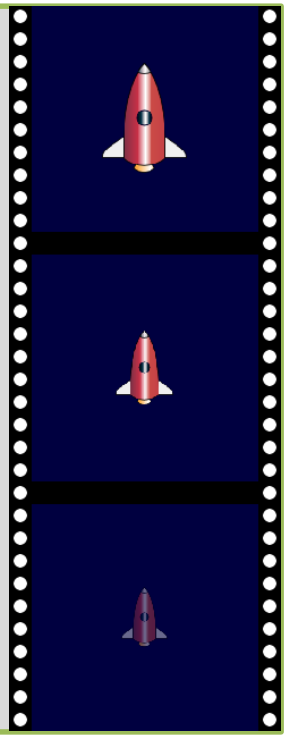


Demo

EMERTXE

Sequential
Animations

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



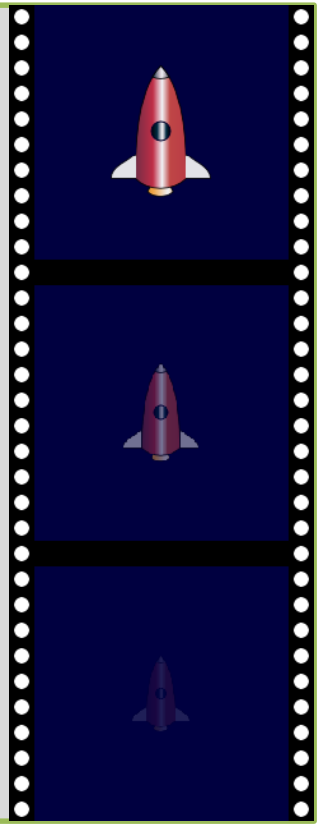
- Child elements define a two-stage animation:
  - First ,the rocket is scaled down and then it fades out
- `SequentialAnimation` does not itself have a target
  - It only groups other animations

Pausing between
Animations

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

Parallel Animations

```
ParallelAnimation {
        NumberAnimation {
                target: rocket; properties: "scale"
                from: 1.0; to: 0.5; duration: 1000
        }
        NumberAnimation {
                target: rocket;
                properties: "opacity"
                from: 1.0; to: 0.0; duration: 1000
        }
        running: true
}
```



- Demo

ΣMERTXE

Other Animations

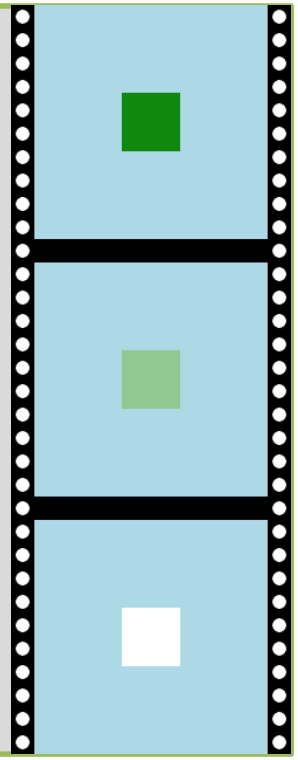
Other animations

- `ColorAnimation` for changes to color properties
- `RotationAnimation` for changes to orientation of items
- `Vector3dAnimation` for motion in 3D space
- `AnchorAnimation` animate an anchor change
- `ParentAnimation` animates changes in parent values.
- `SpringAnimation` allows a property to track a value in a spring-like motion
- `PropertyAction` allows immediate property changes during animation
- `ScriptAction` allows scripts to be run during an animation

Color animations

- `ColorAnimation` describes color changes to items
- Component-wise blending of RGBA values

```
ColorAnimation {
        target: rectangle1
        property: "color"
        from: Qt.rgba(0,0.5,0,1)
        to: Qt.rgba(1,1,1,1)
        duration: 1000
        running: true
}
```



EMERTXE

Rotation Animation

- `RotationAnimation` describes rotation of items
- Easier to use than `NumberAnimation` for the same purpose
- Applied to the `rotation` property of an element
- Value of `direction` property controls rotation:
  - `RotationAnimation.Clockwise`
  - `RotationAnimation.Counterclockwise`
  - `RotationAnimation.Shortest` – the direction of least angle between `from` and `to` values

Rotation Animation

```
Image {
    id: ball
    source: "../images/ball.png"
    anchors.centerIn: parent
    smooth: true
    RotationAnimation on rotation {
        from: 45; to: 315
        direction: RotationAnimation.Shortest
        duration: 1000
    }
}
```

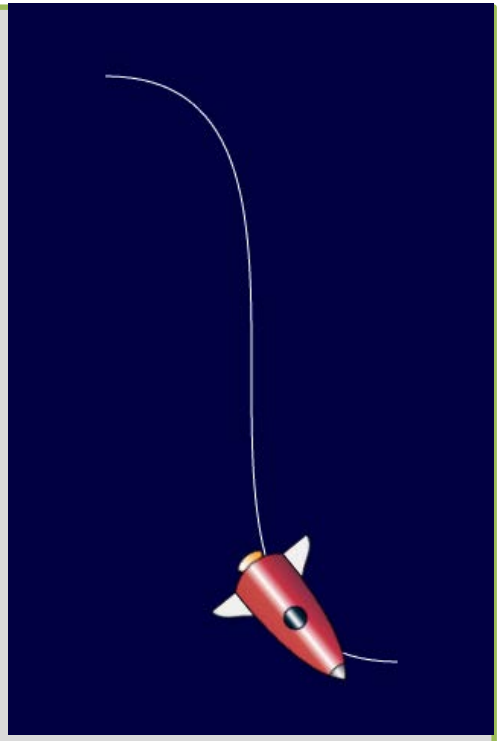
- 1 second animation
- Counter-clockwise from 45° to 315°
  - Shortest angle of rotation is via 0°

Path Animation

- Element `PathAnimation` animates an item along a path
- Manipulates the `x, y` and `rotation` properties of an element
- The `target` element will be animated along the `path`
- Value of orientation property controls the target rotation:
  - `PathAnimation.Fixed`
  - `PathAnimation.RightFirst`
  - `PathAnimation.LeftFirst`
  - `PathAnimation.TopFirst`
  - `PathAnimation.BottomFirst`
- Value of `path` is specified using `Path` element and its helpers
- `PathLine, PathQuad, PathCubic, PathCurve, PathArc, PathSvg`

Path Animation

```
PathAnimation {
        id: pathAnim
        duration: 2000
        easing.type: Easing.InOutQuad
        target: rocket
        orientation: PathAnimation.RightFirst
        anchorPoint: Qt.point(rocket.width/2, rocket.height/2)
        path: Path {
                startX: rocket.width/2; startY: rocket.height/2
                PathCubic {
                        x: window.width - rocket.width/2
                        y: window.height - rocket.height/2
                        control1X: x; control1Y: rocket.height/2
                        control2X: rocket.width/2; control2Y: y
                }
        }
}
```



- Demo

Hands-on

Presenting Data

Contents

- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

Objectives

Can manipulate and present data:

- Familiarity with positioners and repeaters
  - Rows, columns, grids, flows
  - Item indexes
- Understanding of the relationship between models
  - Pure models
  - Visual models
  - XML models
- Ability to define and use list models
  - Using pure models with repeaters and delegates
  - Using visual models with repeaters
- Ability to use models with views
  - Using list and grid views
  - Decorating views
  - Defining delegates

Why Use Model/view

Separation?

- Easily change the UI later
- Add an alternative UI
- Separation of concerns
- Leads to easier maintenance
- Easily change the data source
  - (XML? JSON? Other?)
- Allows the use of 'dummy' data during development
- Many Qt APIs to consume the common data structures

Contents

- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

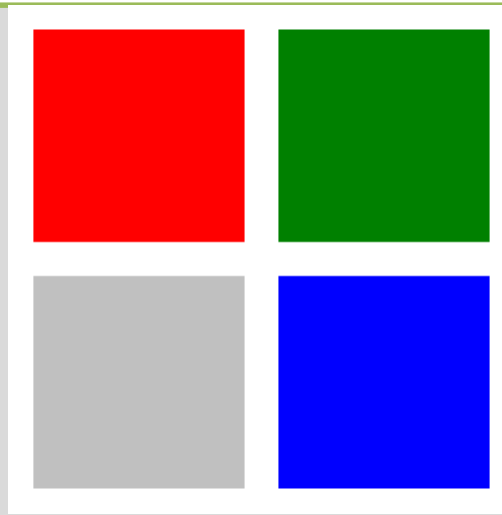
Arranging Items

Positioners and repeaters make it easier to work with many items

- Positioners arrange items in standard layouts
  - In a column: Column
  - In a row: Row
  - In a grid: Grid
  - Like words on a page: Flow
- Repeaters create items from a template
  - For use with positioners
  - Using data from a model
- Combining these make it easy to layout lots of items

Positioning Items

```
Grid {
        x: 15; y: 15; width: 300; height: 300
        columns: 2; rows: 2; spacing: 20
        Rectangle { width: 125; height: 125; color: "red" }
        Rectangle { width: 125; height: 125; color: "green" }
        Rectangle { width: 125; height: 125; color: "silver" }
        Rectangle { width: 125; height: 125; color: "blue" }
}
```



- Items inside a positioner are automatically arranged
  - Ina 2 by 2 Grid
  - With horizontal/vertical spacing of 20 pixels
  - `x, y` is the position of the first item
- Like layouts in Qt
- Demo

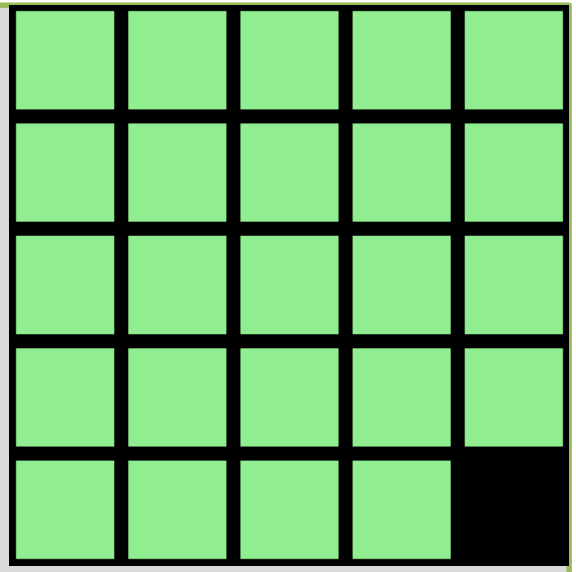
Repeating Items

```
Rectangle { width: 400; height: 400; color: "black"
        Grid { x: 5; y: 5 rows: 5; columns: 5; spacing: 10
                Repeater {
                        model: 24
                        Rectangle { width: 70; height: 70 color: "lightgreen" }
                }
        }
}
```

- The Repeater creates items
- The Grid arranges them within its parent item
- The outer Rectangle item provides
  - The space for generated items
  - A local coordinate system

Repeating Items

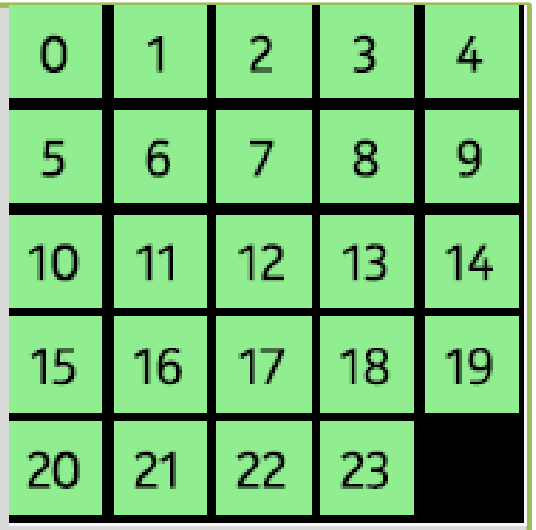
```
Rectangle { width: 400; height: 400; color: "black"
        Grid { x: 5; y: 5 rows: 5; columns: 5; spacing: 10
        Repeater {
                model: 24
                Rectangle {
                        width: 70; height: 70 color: "lightgreen" }
                }
        }
}
```



- `Repeater` takes data from a model
  - Just a number in this case
- Creates items based on the template item
  - A light green rectangle
- Demo

Indexing Items

```
Rectangle { width: 400; height: 400; color: "black"
    Grid { x: 5; y: 5 rows: 5; columns: 5; spacing: 10
    Repeater {
        model: 24
        Rectangle {
            width: 70; height: 70 color: "lightgreen"
            Text {
                text: index
                font.pointSize: 30
                anchors.centerIn: parent }
            }
        }
    }
}
```



- `Repeater` provides an index for each item it creates
- Demo

Positioner

Hints and Tips

- Anchors in the `Row`, `Column` or `Grid`
  - Apply to all the items they contain

Hands-on

Contents

- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

Models and Views

Models and views provide a way to handle data sets

- Models hold data or items
- Views display data or items
  - Using delegates

Models

Pure models provide access to data:

- ListModel
- XmlListModel

Visual models provide information about how to display data:

- Visual item model: ObjectModel (replaces VisualItemModel)
  - Contains child items that are supplied to views
- Visual data model: DelegateModel (replaces VisualDataModel)
  - Contains an interface to an underlying model
  - Supplies a delegate for rendering
  - Supports delegate sharing between the views

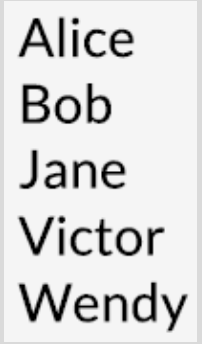
List Models

- List models contain simple sequences of elements
- Each `ListElement` contains
  - One or more pieces of data
  - Defined using properties
  - *No information* about how to display itself
- `ListElement` does not have pre-defined properties
  - All properties are custom properties

```
ListModel {
    id: nameModel
    ListElement { ... }
    ListElement { ... }
    ListElement { ... }
}
```

Defining a List Model

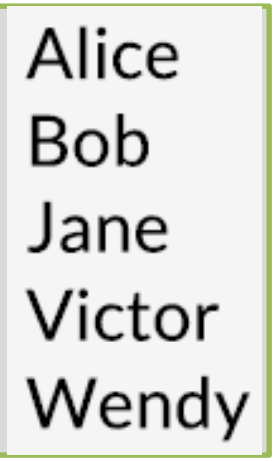
```
ListModel {
        id: nameModel
        ListElement { name: "Alice" }
        ListElement { name: "Bob" }
        ListElement { name: "Jane" }
        ListElement { name: "Victor" }
        ListElement { name: "Wendy" }
}
```



- Define a `ListModel`
  - With an id so it can be referenced
- Define `ListElement` child objects
  - Each with a `name` property
  - The property will be referenced by a delegate
- Demo

Defining a Delegate

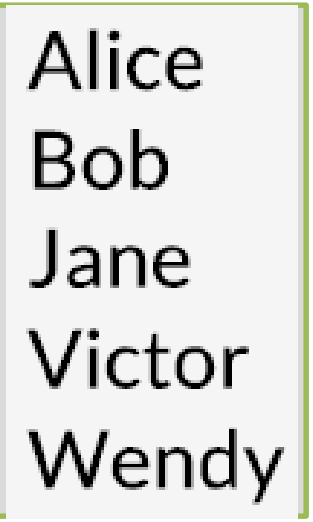
```
Component {
    id: nameDelegate
    Text {
        text: name;
        font.pixelSize: 32
    }
}
```



- Define a `Component` to use as a delegate
  - With an `id` so it can be referenced
  - Describes how the data will be displayed
- Properties of list elements can be referenced
  - Use a `Text` item for each list element
  - Use the value of the `name` property from each element
- In the item inside a `Component`
  - The `parent` property refers to the view
  - A `ListView` attached property can also be used to access the view

Using a List Model

```
Column {
    anchors.fill: parent
    Repeater {
        model: nameModel
        delegate: nameDelegate
    }
}
```



- A `Repeater` fetches elements from `nameModel`
  - Using the delegate to display elements as `Text` items
- A `Column` arranges them vertically
  - Using anchors to make room for the items

Working with
Items

- `ListModel` is a dynamic list of items
- Items can be appended, inserted, removed and moved
  - Append item data using JavaScript dictionaries:
  - `bookmarkModel.append({"title": lineEdit.text})`
  - Remove items by index obtained from a ListView
  - `bookmarkModel.remove(listView.currentIndex)`
  - Move a number of items between two indices:
  - `bookmarkModel.move(listView.currentIndex, listView.currentIndex + 1, number)`

List Model Hints

- Note: Model properties cannot shadow delegate properties:

```
ListModel {
    ListElement { text: "Alice" }
}
Component {
    Text {
        text: text; // Will not work
    }
}
```

Defining an Object Model

```
Rectangle {
    width: 400; height: 200; color: "black"
    ObjectModel {
        id: labels
        Rectangle { color: "#cc7777"; radius: 10.0
            width: 300; height: 50
            Text { anchors.fill: parent
                font.pointSize: 32; text: "Books"
                horizontalAlignment: Qt.AlignHCenter } }
        Rectangle { color: "#cccc55"; radius: 10.0
            width: 300; height: 50
            Text { anchors.fill: parent
                font.pointSize: 32; text: "Music"
                horizontalAlignment: Qt.AlignHCenter } }
} }
```

- Define a `ObjectModel` item
  - With an `id` so it can be referenced
  - Import `QtQml.Models 2.1`

Defining an Object Model

```
Rectangle {
    width: 400; height: 200; color: "black"
    ObjectModel {
        id: labels
        Rectangle { color: "#cc7777"; radius: 10.0
            width: 300; height: 50
            Text { anchors.fill: parent
                font.pointSize: 32; text: "Books"
                horizontalAlignment: Qt.AlignHCenter } }
        Rectangle { color: "#cccc55"; radius: 10.0
            width: 300; height: 50
            Text { anchors.fill: parent
                font.pointSize: 32; text: "Music"
                horizontalAlignment: Qt.AlignHCenter } }
} }
```

- Define child items
  - These will be shown when required

Using an Object Model

```
Rectangle {
    width: 400; height: 200; color: "black"
    ObjectModel {
        id: labels
        ....
    }
    Column {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        Repeater { model: labels }
    }
}
```

- A `Repeater` fetches items from the labels model
- A `Column` arranges them vertically

Contents

- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

Views

- `ListView` shows a classic list of items
  - With horizontal or vertical placing of items
- `GridView` displays items in a grid
  - Like an file manager's icon view

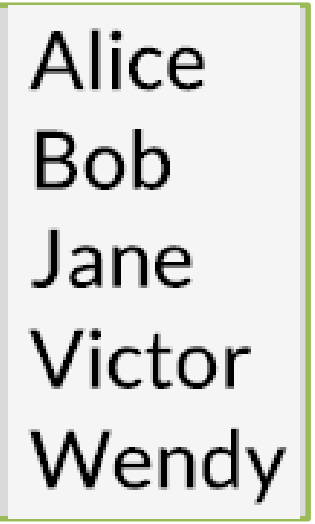
List Views

Take the model and delegate from before:

```
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: 32
        }
}
```

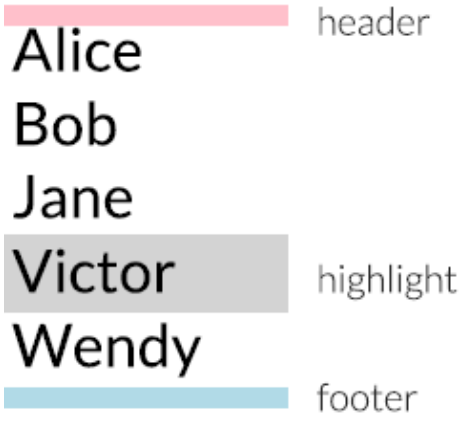
List Views

```
ListView {
        anchors.fill: parent
        model: nameModel
        delegate: nameDelegate
        clip: true
}
```



- No default delegate
- Unclipped views paint outside their areas
  - Set the clip property to enable clipping
- Views are positioned like other items
  - The above view fills its parent
- Demo

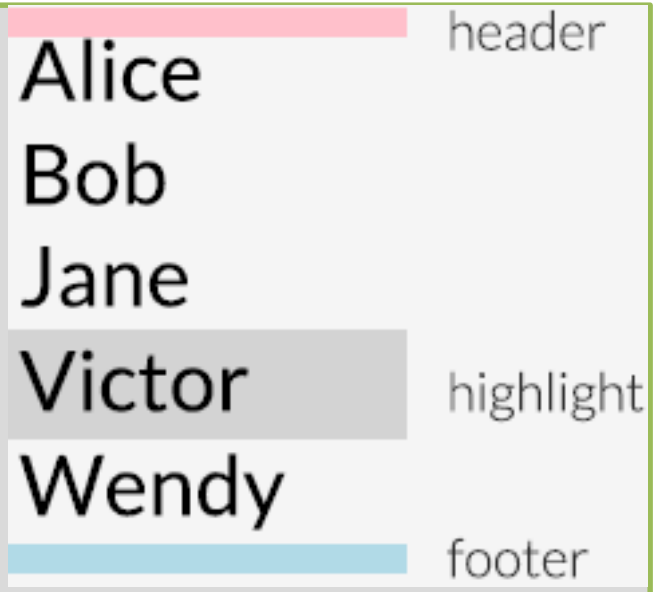
Decoration and Navigation



- By default, ListView is
  - Undecorated
  - A flickable surface (can be dragged and flicked)
- To add decoration:
  - With a header and footer
  - With a highlight item to show the current item
- To configure for navigation:
  - Set focus to allow keyboard navigation
  - Property highlight also helps the user with navigation
  - Unset interactive to disable dragging and flicking
- Demo

Decoration and Navigation

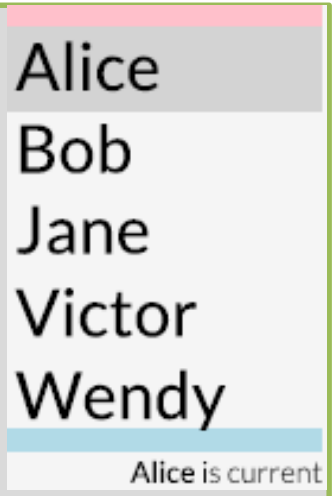
```
ListView {
        anchors.fill: parent
        model: nameModel
        delegate: nameDelegate
        focus: true
        clip: true
        header: Rectangle {
                width: parent.width; height: 10;
                color: "pink" }
        footer: Rectangle {
                width: parent.width; height: 10;
                color: "lightblue" }
        highlight: Rectangle {
        width: parent.width
        color: "lightgray" }
}
```



Decoration and
Navigation

- Each `ListView` exposes its current item:

```
ListView {
    id: listView
}
Text {
    id: label
    anchors.bottom: parent.bottom
    anchors.horizontalCenter: parent.horizontalCenter
    text: "<b>" + listView.currentItem.text + "</b> is current"
    font.pixelSize: 16
}
```

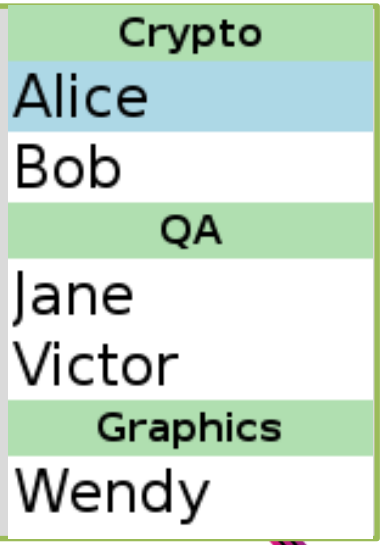


- Recall that, in this case, each item has a `text` property
  - re-use the listView's `currentItem's text`
- Demo

Adding Sections

- Data in a `ListView` can be ordered by section
- Categorize the list items by
  - Choosing a property name; e.g. `team`
  - Adding this property to each `ListElement`
  - Storing the section in this property

```
ListModel {
        id: nameModel
        ListElement { name: "Alice"; team: "Crypto" }
        ListElement { name: "Bob"; team: "Crypto" }
        ListElement { name: "Jane"; team: "QA" }
        ListElement { name: "Victor"; team: "QA" }
        ListElement { name: "Wendy"; team: "Graphics" }
}
```



Displaying Sections

Using the `ListView`

- Set `section.property`
  - Refer to the `ListElement` property holding the section name
- Set section.criteria to control what to show
  - `ViewSection.FullString` for complete section name
  - `ViewSection.FirstCharacter` for alphabetical groupings
- Set `section.delegate`
  - Create a delegate for section headings
  - Either include it inline or reference it

Displaying Sections

```
ListView {
        model: nameModel
        section.property: "team"
        section.criteria: ViewSection.FullString
        section.delegate: Rectangle {
        color: "#b0dfb0"
        width: parent.width
        height: childrenRect.height + 4
        Text { anchors.horizontalCenter: parent.horizontalCenter
                font.pixelSize: 16
                font.bold: true
                text: section }
        }
}
```

- The `section.delegate` is defined like the `highlight` delegate

Grid Views

```
ListModel {
        id: nameModel
        ListElement { file: "../images/rocket.svg" name: "rocket" }
        ListElement { file: "../images/clear.svg" name: "clear" }
        ListElement { file: "../images/arrow.svg" name: "arrow" }
        ListElement { file: "../images/book.svg" name: "book" }
}
```

- Set up a list model with items:
- Define string properties to use in the delegate
- Demo

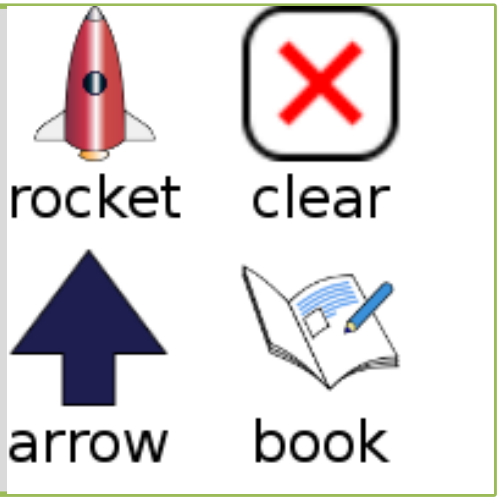
Grid Views

- Set up a delegate:

```
Component {
    id: nameDelegate
    Column {
        Image {
            id: delegateImage
            anchors.horizontalCenter: delegateText.horizontalCenter
            source: file; width: 64; height: 64; smooth: true
            fillMode: Image.PreserveAspectFit
        }
        Text {
            id: delegateText
            text: name; font.pixelSize: 24
        }
    }
}
```

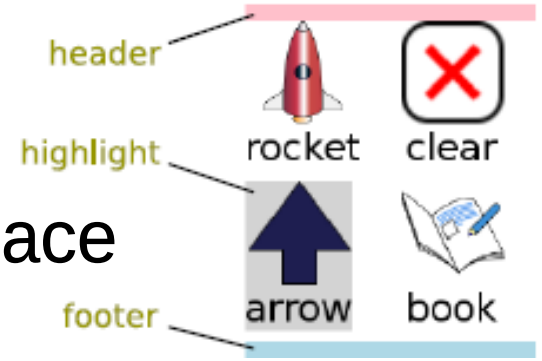
Grid Views

```
GridView {
        anchors.fill: parent
        model: nameModel
        delegate: nameDelegate
        clip: true
}
```



- The same as ListView to set up
- Uses data from a list model
  - Not like Qt's table view
  - More like Qt's list view in icon mode

Decoration and
Navigation

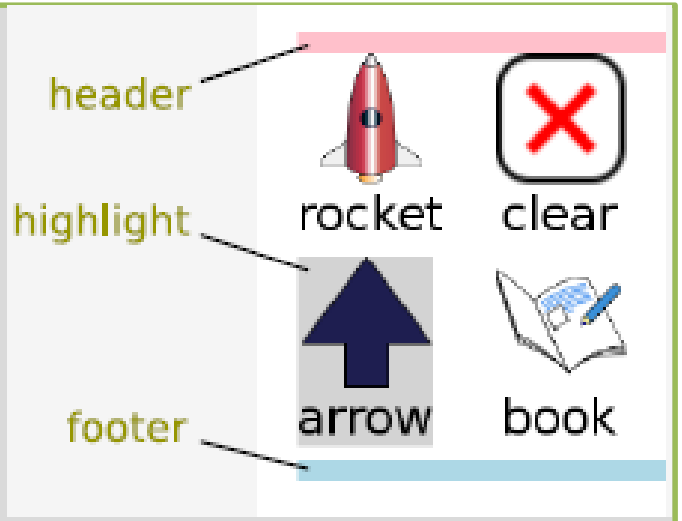


Like `ListView`, `GridView` is

- Undecorated and a flickable surface
- To add decoration:
  - Define header and footer
  - Define highlight item to show the current item
- To configure for navigation:
  - Set focus to allow keyboard navigation
  - Highlight also helps the user with navigation
  - Unset interactive to disable dragging and flicking
- Demo

Decoration and Navigation

```
GridView {
…
        header: Rectangle {
                width: parent.width; height: 10
                color: "pink"
        }
        footer: Rectangle {
                width: parent.width; height: 10
                color: "lightblue"
        }
        highlight: Rectangle {
                width: parent.width
                color: "lightgray"
        }
        focus: true clip: true
}
```



Hands-on

Contents

- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

XML List Models

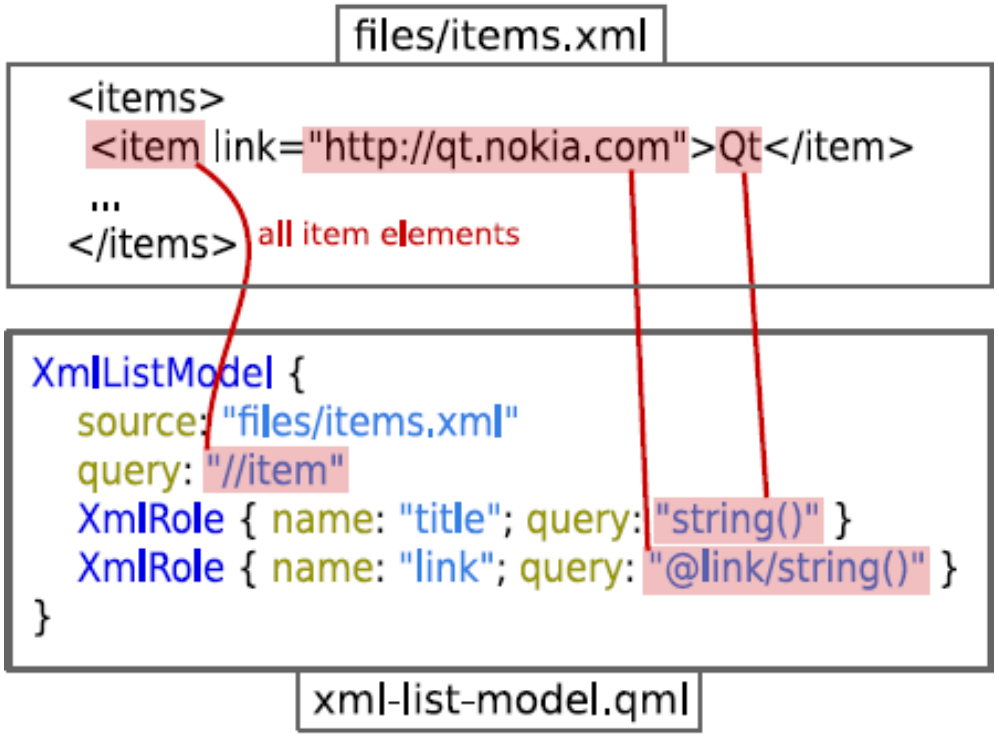
- Many data sources provide data in XML formats
- Element `XmlListModel` is used to supply XML data to views
  - Using a mechanism that maps data to properties
  - Using XPath queries
- Views and delegates do not need to know about XML
  - Use a `ListView` or `Repeater` to access data

Defining an XML
List Model

```
XmlListModel {
        id: xmlModel
        source: "files/items.xml"
        query: "//item"
        XmlRole { name: "title"; query: "string()" }
        XmlRole { name: "link"; query: "@link/string()" } }
}
```

- Set the id property so the model can be referenced
- Specify the source of the XML
- The query identifies pieces of data in the model
- Each piece of data is queried by XmlRole elements
- Demo

XML Roles



Result

title: "Qt"
link: "http://qt.nokia.com"

- Element XmlRole associates names with data obtained using Xpath queries
- Made available to delegates as properties
  - Properties title and link in the above example

Using an XML
List Model

```
TitleDelegate {
        id: xmlDelegate
}
ListView {
        anchors.fill: parent
        anchors.margins: 4
        model: xmlModel
        delegate: xmlDelegate
}
```

- Specify the `model` and `delegate` as usual
- Ensure that the view is positioned and given a size
- Element `TitleDelegate` is defined in `TitleDelegate.qml`
- Must be defined using a `Component` element
- Demo

Defining a Delegate

```
Component {
    Item {
        width: parent.width; height: 64
        Rectangle {
            width: Math.max(childrenRect.width + 16, parent.width)
            height: 60; 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 }
            }
        }
    }
}
```

- Property `parent` refers to the view where it is used
- Properties `title` and `link` are properties exported by the model

Contents

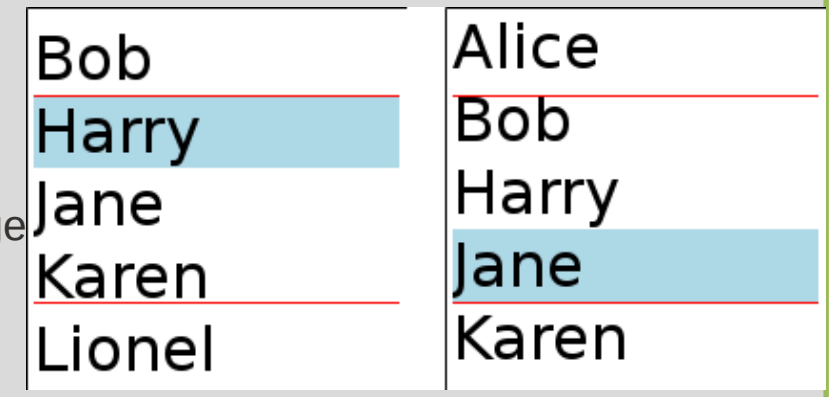
- Arranging Items
- Data Models
- Using Views
- XML Models
- Views Revisited

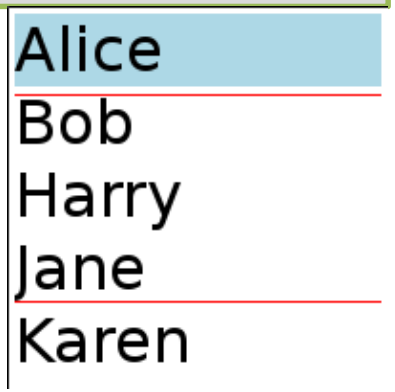
Customizing Views

- All views are based on the `Flickable` item
- Key navigation of the highlighted item does not wrap around
  - Set `keyNavigationWraps` to true to change this behavior
- The highlight can be constrained
  - Set the `highlightRangeMode` property
  - Value `ListView.ApplyRange` tries to keep the highlight in a given area
  - Value `ListView.StrictlyEnforceRange` keeps the highlight stationary, moves the items around it

Customizing Views

```
ListView {
        preferredHighlightBegin: 42
        preferredHighlightEnd: 150
        highlightRangeMode: ListView.ApplyRange
…
}
```



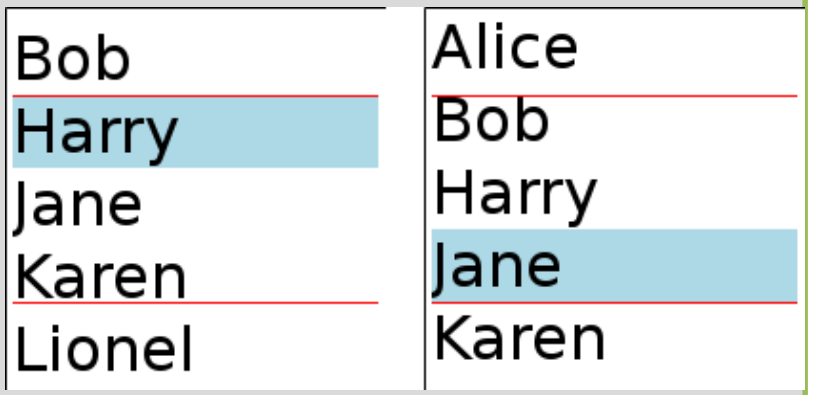


- View tries to keep the highlight within range
- Highlight may leave the range to cover end items
- Properties `preferredHighlightBegin` and `preferredHighlightEnd` should
  - Hold coordinates within the view
  - Differ by the height/width of an item or more
- Demo

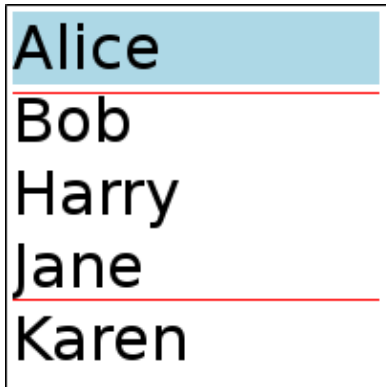
ΣMERTXE

Customizing Views

```
ListView {
        preferredHighlightBegin: 42
        preferredHighlightEnd: 150
        highlightRangeMode:
                ListView.StrictlyEnforceRange
        ...
}
```



- View always keeps the highlight within range
- View may scroll past its end to keep the highlight in range
- Properties `preferredHighlightBegin` and
- `preferredHighlightEnd` should
- Hold coordinates within the view
- Differ by the height/width of an item or more
- Demo



ΣMERTXE

Optimizing Views

- Views create delegates to display data
  - Delegates are only created when they are needed
  - Delegates are destroyed when no longer visible
  - This can impact performance
- Delegates can be cached to improve performance
  - Property `cacheBuffer` is the maximum number of delegates to keep (calculated as a multiply of
  - the height of the delegate)
  - Trades memory usage for performance
  - Useful if it is expensive to create delegates; for example
    - When obtaining data over a network
    - When delegates require complex rendering

Integrating QML with C++

Contents

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types Plug-ins

Objectives

- The QML runtime environment
  - Understanding of the basic architecture
  - Ability to set up QML in a C++ application
- Exposing C++ objects to QML
  - Knowledge of the Qt features that can be exposed
  - Familiarity with the mechanisms used to expose objects

Contents

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types Plug-ins

Overview

Qt Quick is a combination of technologies:

- A set of components, some graphical
- A declarative language: QML
  - Based on JavaScript
  - Running on a virtual machine
- A C++ API for managing and interacting with components
  - The QtQuick module

Setting up a
QtQuick Application

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

int main(int argc, char *argv[])
{
        QGuiApplication app(argc, argv);
        QQmlApplicationEngine engine;
        engine.load(QUrl(QStringLiteral("qrc:/animation.qml")));
        return app.exec();
}
```

Demo

Setting up QtQuick

```
QT += quick
RESOURCES = simpleviewer.qrc
SOURCES = main.cpp
```

```
import QtQuick 2.0
import QtQuick.Window 2.2
Window {
        visible: true
        width: 400; height: 300
```

Demo

Contents

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types Plug-ins

Exporting C++
Objects to QML

- C++ objects can be exported to QML

```
class User : public QObject {
        Q_OBJECT
        Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
        Q_PROPERTY(int age READ age WRITE setAge NOTIFY ageChanged)
public:
        User(const QString &name, int age, QObject *parent = 0); ... }
```

- The notify signal is needed for correct property bindings!
- Q_PROPERTY must be at top of class

Exporting C++
Objects to QML

- Class `QQmlContext` exports the instance to QML.

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

	AnimalModel model; model.addAnimal(Animal("Wolf", "Medium"));
	model.addAnimal(Animal("Polar bear", "Large"));
	model.addAnimal(Animal("Quoll", "Small"));

	QQmlApplicationEngine engine;
	QQmlContext *ctxt = engine.rootContext();
	ctxt->setContextProperty("animalModel", &model);

	engine.load(QUrl(QStringLiteral("qrc:/view.qml")));

	return app.exec();
}
```

Using the Object
in QML

- Use the instances like any other QML object.

```
Window {
        visible: true
        width: 200; height: 250
        ListView {
                width: 200; height: 250
                model: animalModel
                delegate: Text { text: "Animal: " + type + ", " + size }
        }
}
```

What Is Exported?

- Properties
- Signals
- Slots
- Methods marked with `Q_INVOKABLE`
- Enums registered with `Q_ENUMS`

```
class IntervalSettings : public QObject
{
	Q_OBJECT
	Q_PROPERTY(int duration READ duration WRITE setDuration
	                                        NOTIFY durationChanged)
	Q_ENUMS(Unit)
	Q_PROPERTY(Unit unit READ unit WRITE setUnit NOTIFY unitChanged)
public:
	enum Unit { Minutes, Seconds, MilliSeconds };
```

Contents

- Declarative Environment
- Exporting C++ objects to QML
- Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting QPainter based GUI Classes
  - Exporting Scene Graph based GUI Classes
- Using Custom Types Plug-ins

Overview

Steps to define a new type in QML:

- In C++: Subclass either `QObject` or `QQuickItem`
- In C++: Register the type with the QML environment
- In QML: Import the module containing the new item
- In QML: Use the item like any other standard item
- Non-visual types are `QObject` subclasses
- Visual types (items) are `QQuickItem` subclasses
- `QQuickItem` is the C++ equivalent of `Item`

Step 1:
Implementing the Class

```
#include <QObject>

class QTimer;
class Timer : public QObject {
	Q_OBJECT

public:
	explicit Timer( QObject* parent = 0 );

private:
	QTimer* m_timer;
}
```

Implementing
the Class

- Element `Timer` is a `QObject` subclass
- As with all `QObjects`, each item can have a parent
- Non-GUI custom items do not need to worry about any painting

Step 1:
Implementing the Class

```
#include "timer.h"
#include <QTimer>
Timer::Timer( QObject* parent ): QObject( parent ),
                    m_timer( new QTimer( this ) )
{
    m_timer->setInterval( 1000 );
    m_timer->start();
}
```

Step 2:
Registering the Class

```
#include "timer.h"
#include <QGuiApplication>
#include <qqml.h> // for qmlRegisterType
#include <QQmlApplicationEngine>

int main(int argc, char **argv) {
      QGuiApplication app( argc, argv );
      // Expose the Timer class
      qmlRegisterType<Timer>( "CustomComponents", 1, 0, "Timer" );
      QQmlApplicationEngine engine;
      engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
      return app.exec();
}
```

- `Timer` registered as an element in module "CustomComponents"
- Automatically available to the `main.qml` file

Reviewing the Registration

```
qmlRegisterType<Timer>(
"CustomComponents", 1, 0, "Timer" );
```

- This registers the `Timer` C++ class
- Available from the CustomComponents QML module
  - version1.0 (first number is major; second Is minor)
- Available as the `Timer` element
  - The `Timer` element is an non-visual item
  - A subclass of `QObject`

Step 3+4

Importing and Using the Class

- In the main.qml file:

```
import CustomComponents 1.0
Window {
    visible: true; width: 500; height: 360
    Rectangle { anchors.fill: parent
        Timer { id: timer }
    }
    …
}
```

- Demo

Adding Properties

- In the `main.qml` file:

```
Rectangle {
    …
    Timer {
        id: timer
    interval: 3000
    }
    …
}
```

- A new `interval` property
- Demo

Declaring a Property

- In the `timer.h` file:

```
class Timer : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int interval READ interval WRITE setInterval
               NOTIFY intervalChanged) // Or use MEMBER
    ....
```

- Use a `Q_PROPERTY` macro to define a new property
  - Named `interval` with `int` type
  - With getter and setter, `interval()` and `setInterval()`
  - Emits the `intervalChanged()` signal when the value changes
- The signal is just a notification
  - It contains no value
  - We must emit it to make property bindings work

Declaring Getter, Setter and Signal

- In the *timer.h* file:

```
public:
    void setInterval( int msec );
    int interval();
signals:
    void intervalChanged();
private:
    QTimer* m_timer;
```

- Declare the getter and setter
- Declare the notifier signal
- Contained `QTimer` object holds actual value

Implementing
Getter and Setter

- In the *timer.cpp* file:

```
void Timer::setInterval( int msec )
{
	if ( m_timer->interval() == msec )
		return;
	m_timer->stop();
	m_timer->setInterval( msec );
	m_timer->start();
	Q_EMIT intervalChanged();
}
int Timer::interval() {
	return m_timer->interval();
}
```

- Do not emit notifier signal if value does not actually change
- Important to break cyclic dependencies in property bindings

Summary of
Items and Properties

- Register new QML types using `qmlRegisterType`
  - New non-GUI types are subclasses of `QObject`
- Add QML properties
  - Define C++ properties with `NOTIFY` signals
  - Notifications are used to maintain the bindings between items
  - *`Only`* emit notifier signals if value actually changes

Adding Signals

- In the *main.qml* file:

```
Rectangle {
    …
    Timer {
        id: timer
        interval: 3000
        onTimeout : {
            console.log( "Timer fired!" );
        }
    }
}
```

- A new `onTimeout` signal handler
  - Outputs a message to stderr.
- Demo

Declaring a Signal

- In the *timer.h* file:

```
Q_SIGNALS:
    void timeout();
    void intervalChanged();
```

- Add a `timeout()` signal
- This will have a corresponding `onTimeout` handler in QML
- We will emit this whenever the contained `QTimer` object fires

Emitting the Signal

- In the *timer.cpp* file:

```
Timer::Timer( QObject* parent ): QObject( parent ),
                                 m_timer( new QTimer( this ) )
{
    connect( m_timer, &QTimer::timeout, this, &Timer::timeout );
}
```

- Change the constructor
- Connect `QTimer::timeout()` signal to `Timer::timeout()` signal

Handling the Signal

- In the *main.qml* file:

```
Timer {
    id: timer
    interval: 3000
    onTimeout: {
        console.log( "Timer fired!" );
    }
}
```

- In C++:
  - The `QTimer::timeout()` signal is emitted
  - Connection means `Timer::timeout()` is emitted
- In QML:
  - The `Timer` item's `onTimeout` handler is called
  - Outputs message to stderr

Adding Methods
to Items

Two ways to add methods that can be called from QML:

- Create C++ slots
  - Automatically exposed to QML
  - Useful for methods that do not return values
- Mark regular C++ functions as invokable
  - Allows values to be returned

Adding Slots

- In the *main.qml* file:

```
Timer {
    id: timer
    interval: 1000
    onTimeout: {
        console.log( "Timer fired!" );
    }
}
MouseArea {
    anchors.fill: parent
    onClicked: {
        if (timer.active == false) {
            timer.start();
        } else {
            timer.stop();
        }
    }
}
```

Adding Slots

- Element Timer now has `start()` and `stop()` methods
- Normally, could just use properties to change state...
- For example a `running` property
- Demo

Declaring Slots

- In the *timer.h* file:

```
….
public Q_SLOTS:
    void start();
    void stop();
….
```

- Added `start()` and `stop()` slots to public slots section
- No difference to declaring slots in pure C++ application

Implementing
Slots

- In the *timer.cpp* file:

```
void Timer::start() {
    if ( m_timer->isActive() )
        return;
    m_timer->start();
    Q_EMIT activeChanged();
}
void Timer::stop() {
    if ( !m_timer->isActive() )
        return;
    m_timer->stop();
    Q_EMIT activeChanged();
}
```

- Remember to emit notifier signal for any changing properties

Adding Methods

- In the *`main.qml`* file:

```
Timer {
    id: timer
    interval: timer.randomInterval(500, 1500)
    onTimeout: {
        console.log( "Timer fired!" );
    }
}
```

- Timer now has a `randomInterval()` method
- Obtain a random interval using this method
- Accepts arguments for min and max intervals
- Set the interval using the `interval` property
- Demo

Declaring a Method

- In the *timer.h* file:

```
public:
    explicit Timer( QObject* parent = 0 );
    Q_INVOKABLE int randomInterval( int min, int max )
const;
```

- Define the `randomInterval()` function
- Add the `Q_INVOKABLE` macro before the declaration
- Returns an `int` value
- *Cannot* return a `const` reference

Implementing a Method

THANK YOU