# Qt Training – QML Edition

Based on Qt 5.8

The Qt Company, 2017

# Contents

| | |
|---|---|
| Introduction to QtQuick and QML | Meet Qt Quick<br>Concepts |
| Composing Uis | Nested Items<br>Graphical QML Types<br>Text Type<br>Anchor Layout |
| User Interactions | Mouse Input<br>Touch Input<br>Keyboard Input |
| Structures | Components<br>Modules |
| States and Transitions | States<br>State Conditions<br>Transitions |

# Contents

| | |
|---|---|
| Animations | Animations<br>Easing Curves<br>Animation Groups |
| Presenting Data | Arranging Items<br>Data Models<br>Using Views<br>XML Models<br>Views Revisited |
| QtQuick Controls | Qt Quick Designer<br>Qt Quick Controls<br>Application Window<br>Controls and Views<br>Layouts<br>Styling |

# Contents

| | |
|---|---|
| C++ Integration | Declarative Environment<br>Exporting C++ Objects to QML<br>Exporting Classes to QML<br>Exporting Non-GUI Classes<br>Exporting QPainter based GUI Classes<br>Exporting Scene Graph -based GUI Classes<br>Using Custom Types Plug-ins |
| Graphics Effects | Canvas<br>Particles<br>Shaders |

# Contents

- Meet Qt Quick
- Concepts

# Objectives

- Understanding of QML syntax and concepts
  - QML types and identities
  - Properties and property binding
- Basic user interface composition skills
  - Familiarity with common QML types
  - Understanding of anchors and their uses
  - Ability to reproduce a design

# Qt Quick Requirements

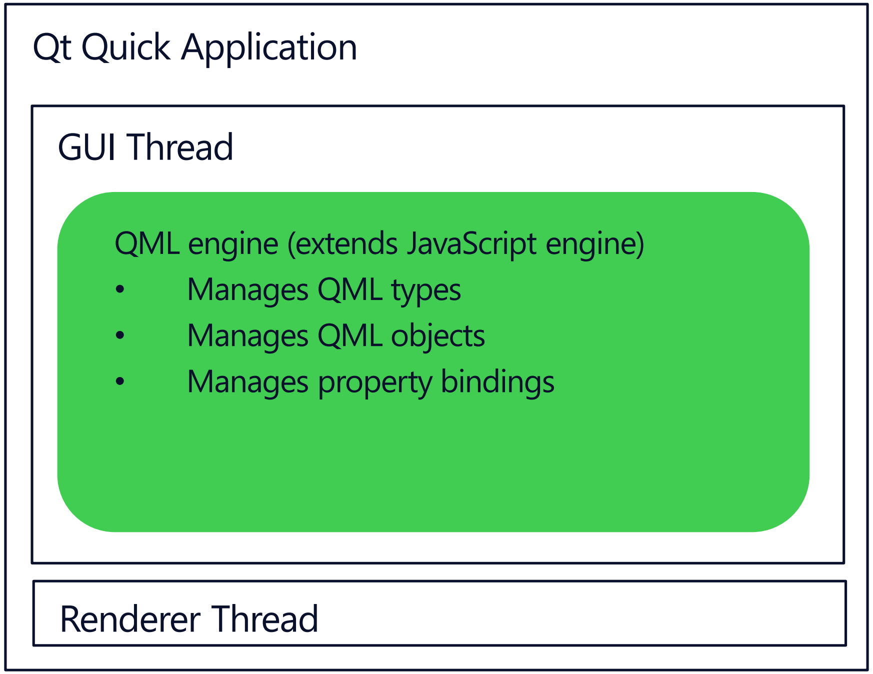
- Graphics API for rendering
  - OpenGL ES 2.0 or higher
  - Qt Quick 2D renderer for SW rendering
  - Direct3D 12

- Other modules can be used to add new features:
  - Qt Graphical Effects: add effects like blur, dropnshadow...
  - Qt 3D: 3D simulations and games in QML
  - Qt Multimedia: audio and video items, camera
  - Qt WebEngine / Qt WebView: web view
  - Qt Sensors: compass, orientation, tilt, proximity…
  - Qt Positioning and Location
  - Qt Bluetooth
  - …

# 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
  - Qt Quick Designer
  - Debugger
  - QML Profiler



Qt Quick Application

GUI Thread

QML engine (extends JavaScript engine)

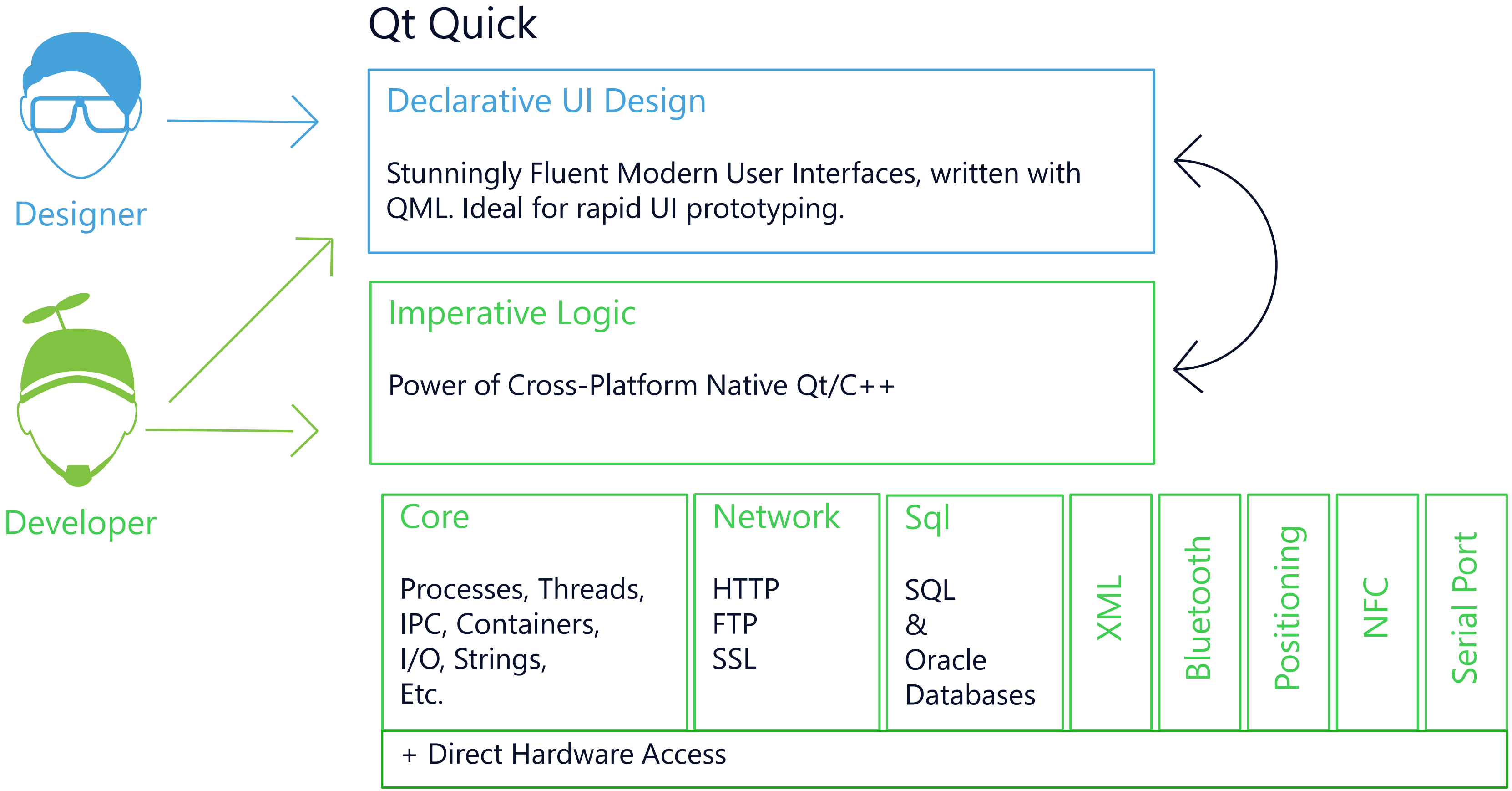
- Manages QML types
- Manages QML objects
- Manages property bindings

Renderer Thread

# Philosophy of Qt Quick

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

# Rapid Workflow with Qt Quick



Designer

Developer

Qt Quick

Declarative UI Design

Stunningly Fluent Modern User Interfaces, written with QML. Ideal for rapid UI prototyping.

Imperative Logic

Power of Cross-Platform Native Qt/C++

Core

Processes, Threads,
IPC, Containers,
I/O, Strings,
Etc.

Network

HTTP
FTP
SSL

Sql

SQL
&
Oracle
Databases

XML

Bluetooth

Positioning

NFC

Serial Port

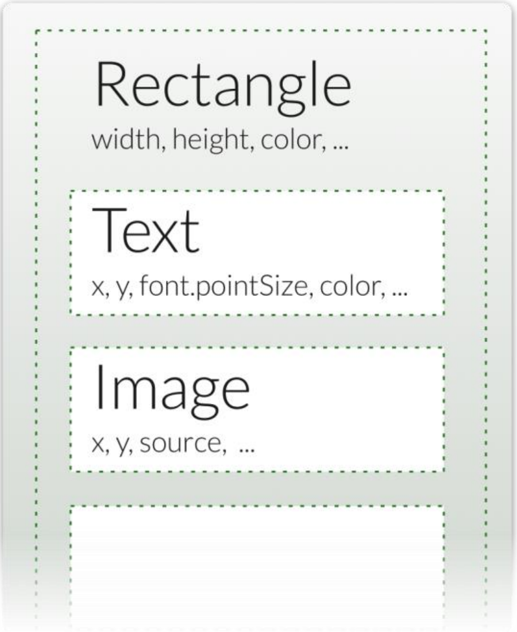
+ Direct Hardware Access

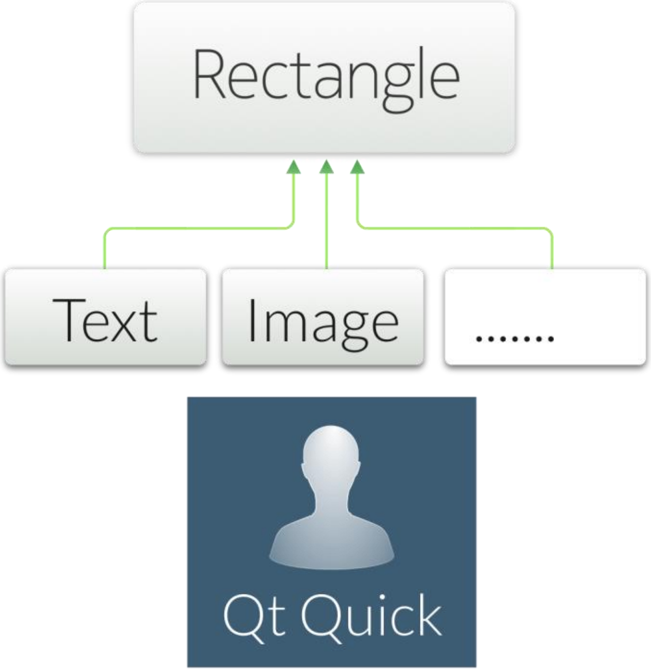
# What Is QML?

Declarative language for User Interface building blocks

- Describes the user interface
  - What UI building blocks look like
  - How they behave
- UI specified as tree of QML objects with properties

# A Tree of QML Objects





- Let's start with an example...

# Viewing an Example

```
import QtQuick 2.7

Rectangle {
    width: 400
    height: 400
    color: "lightblue"
}
```

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

```
qmlscene rectangle.qml
```

- Or open `qml-intro.qmlproject` in Qt Creator
  - Open `rectangle.qml` in editor
  - Click on the *Run* Button – `qmlscene` uses the current file as the main QML file

Demo: ex-concepts/rectangle.qml

# QML Types

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

Documentation: Visual QML Types

# Properties

QML types 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 QML objects (`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.pointSize: 24
    // Preferred syntax
    // font { family: "Helvetica"; pixelSize: 24 }
}
```

- Identity property gives the object a name:
  - Identifying objects (`id` property)
  - Customizing their appearance
  - Changing their behavior

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

# Property Examples

- Attached properties are applied to QML objects without object creation:

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

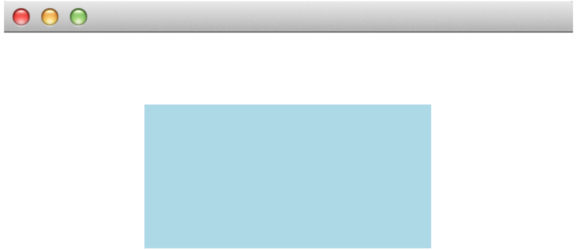
- `KeyNagivation.tab` is not a standard property of `TextInput`
- Is a standard property that is attached to objects

- Custom properties can be added to any object:

```
Rectangle {
    property real mass: 100.0
}

Circle {
    property real radius: 50.0
}
```

# Binding Properties

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



- Properties can contain JavaScript expressions
  - See above: `width` is twice the `height`
- Not just initial assignments
- Expressions are re-evaluated when needed
- Note! JavaScript assignment operator ‘=‘ is not a binding
  - Assignment: `width = height * 2 // No re-evaluation`
  - Assignment to a binding: `width = Qt.binding(function() { return height * 2; } )`

qml-intro/ex-concepts/expressions.qml

# Identifying Objects

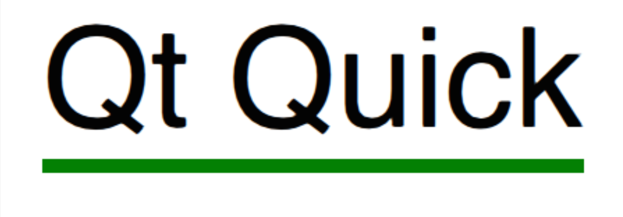
The `id` property defines an identity for a QML object

- Lets other objects 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 types (e.g., gradients, images)
- Used to *create relationships* between objects

Documentation: Property binding

# Using Identities

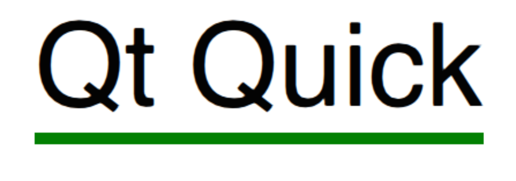
```
Item {
    width: 300; height: 115
    Text {
        id: title
        x: 50; y: 25
        text: "Qt Quick"
        font { family: "Helvetica"; pointSize: parent.width * 0.1 }
    }

    Rectangle {
        x: title.x; y: title.y + title.height - height; height: 5
        width: title.width
        color: "green"
    }
}
```



qml-intro/ex-concepts/identity.qml

# Viewing an Example

```
Text {
    id: title
    x: 50; y: 25
    text: "Qt Quick"
    font { family: "Helvetica"; pointSize: parent.width * 0.1 }
}

Rectangle {
    x: title.x; y: title.y + title.height - height; height: 5
    width: title.width
    color: "green"
}
```



- `Text` item has the identity, `title`
- Properties `width`, `x`, `y` of `Rectangle` bound to `width` of title

# 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

Documentation: QML types

# QML File Structure

- Identifier
- Property declarations
- Signal declarations
- JavaScript functions
- Object properties
- Child objects
- States
- Transitions

```
Item {
    id: exampleItem
    property var exampleProperty: ListView.view
    signal exampleSignal(var variantArgument)
    function example() { return 0; }
    width: window.width; height: window.height
    Text { }
    states: [ State {} ]
    transitions: [ Transition {} ]
```

# Questions

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

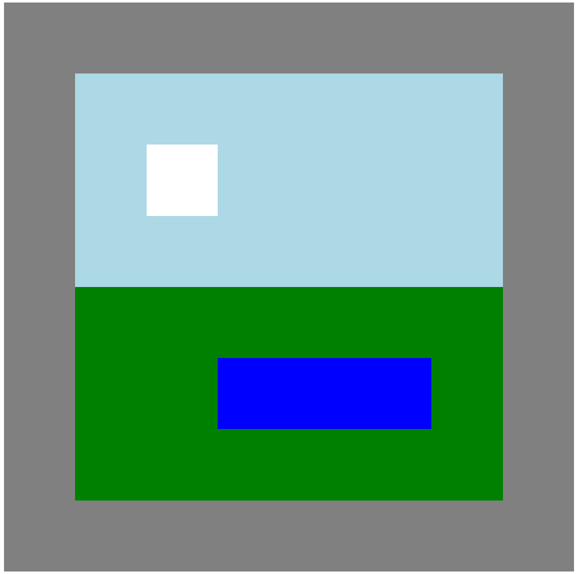
# Summary

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

# Lab – Nested Items

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

1. Recreate the scene using Rectangle items. Make item sizes scalable. Positions can be fixed.
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.



# Contents

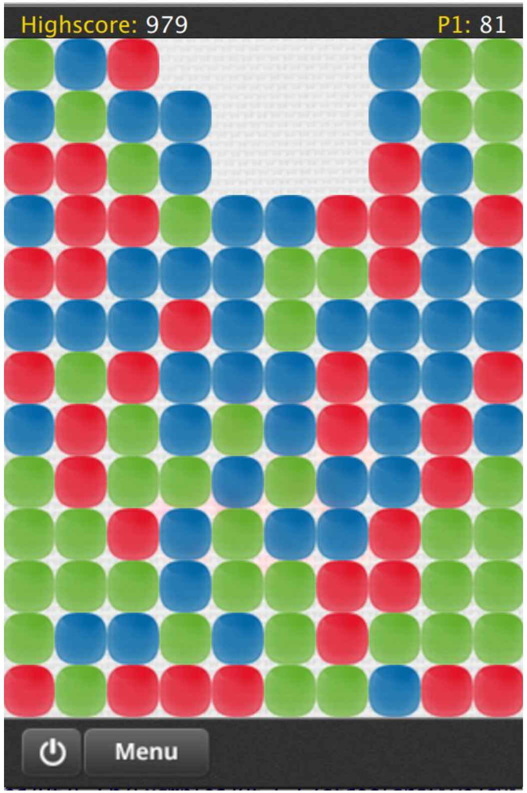
- Nested Items
- Graphical QML Types
- Text Type
- Anchor Layout

# Objectives

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

# Why Use Nested Items, Anchors and Components?

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



Demo: Qt Quick Demo . Same Game

# Nested Items

```
Rectangle {
    width: 400; height: 400
    color: "lightblue”
    Rectangle {
        x: 50; y: 50
        width: parent.width - 2 * x; height: parent.height - 2 * y
        color: "green”
        Rectangle {
            x: parent.width - 2 * width; y: parent.height - 3 * height
            width: 50; height: 50
            color: "white"
        }
    }
}
```

- Each item positioned relative to its parents

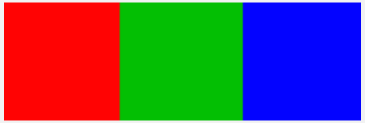
qml-composing-uis/ex-elements/nested2.qml

# 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)

Documentation: QML basic type colors

# Colors



```
Rectangle {
    id: rectangle1
    x: 0; y: 0;
    width: parent.width / 3; height: parent.height; color: "#ff0000"
}
Rectangle {
    id: rectangle2
    x: rectangle1.width; width: parent.width / 3
    height: parent.height
    color: Qt.rgba(0,0.75,0,1)
}
Rectangle {
    x: rectangle1.width + rectangle2.width;
    width: parent.width / 3;
    height: parent.height;
    color: "blue"
}
```

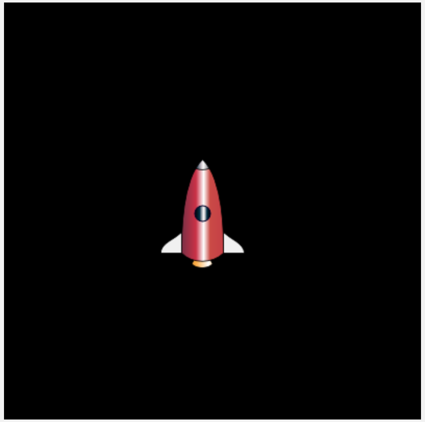
qml-composing-uis/ex-elements/colors.qml

# Images

- Represented by the `Image` QML type
- 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

# Images

```
aRectangle {
    width: 400; height: 400
    color: "black"
    Image {
        x: (parent.width - width) / 2
        y: (parent.height - height) / 2
        source: "../images/rocket.png"
    }
}
```

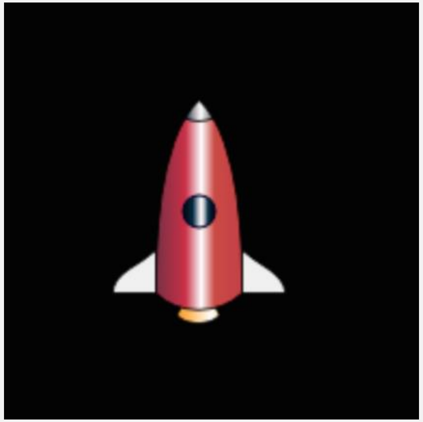


- Property `source` contains a relative path
- Properties `width` and `height` are obtained from the image file

qml-composing-uis/ex-elements/images.qml

# Image Scaling

```
Rectangle {
    width: 400; height: 400
    color: "black"
    Image {
        x: (parent.width - width) / 2
        y: (parent.height - height) / 2
        source: "../images/rocket.png"
        scale: 2.0
    }
```

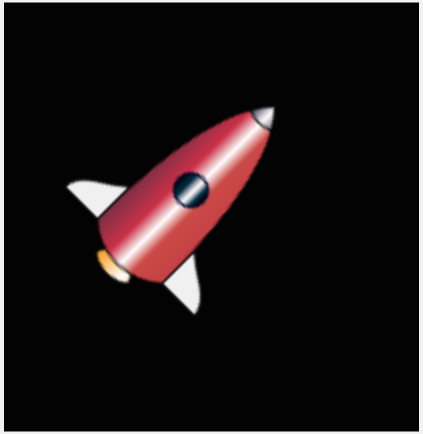


- Property `source` contains a relative path
- Properties `width` and `height` are obtained from the image file
  - Image has non-zero implicit size – pixel dimension
  - Explicit size can override implicit size
  - Properties `sourceWidth` and `sourceHeight` define image size in memory

qml-composing-uis/ex-elements/image-scaling.qml

# Image Rotation

```
Rectangle {
    width: 200; height: 200; color: "black"
    Image {
        x: (parent.width - width) / 2;
        y: (parent.height - height) / 2
        source: "../images/rocket.png"
        rotation: 45.0
    }
}
```

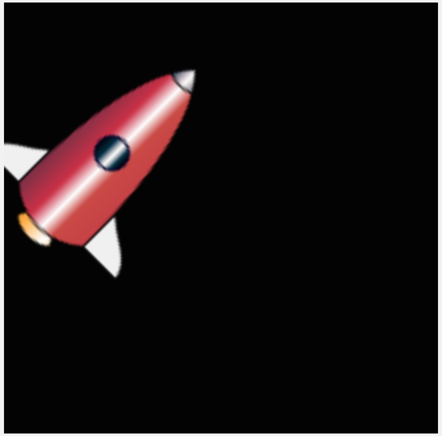


- Set the rotate property
- By default, the center of the item remains in the same place

qml-composing-uis/ex-elements/image-rotation.qml

# Image Rotation

```
Rectangle {
    width: 200; height: 200; color: "black"
    Image {
        x: (parent.width - width) / 2;
        y: (parent.height - height) / 2
        source: "../images/rocket.png"
        rotation: 45.0
        transformOrigin: Item.Top
    }
}
```



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

# Gradients

Define a gradient using the gradient property:

- With a `Gradient` QML type as the value
- Containing `GradientStop` objects, 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.

Documentation: QML Gradient Type

# Gradients

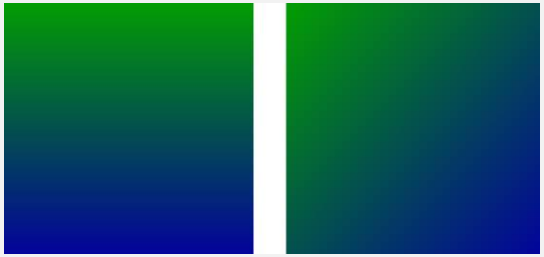
```
Rectangle {
    width: 400; height: 400
    gradient: Gradient {
        GradientStop {
            position: 0.0; color: "green"
        }
        GradientStop {
            position: 1.0; color: "blue"
        }
    }
}
```

- Note the definition of an item as a property value
- Radial and conical gradients are available in `QtGraphicalEffects` module

qml-composing-uis/ex-elements/gradients.qml

# Gradient Images

```
Rectangle {
    property real margin: 25
    width: 425; height: 200
    Image {
        id: image1
        width: (parent.width - margin) / 2
        height: parent.height
        source: "../images/vertical-gradient.png"
    }
    Image {
        x: image1.width + margin
        width: (parent.width - margin) / 2; height: parent.height
        source: "../images/diagonal-gradient.png"
    }
}
```

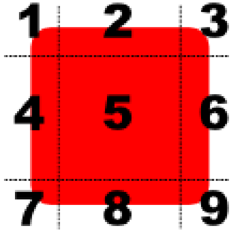


- It is often faster to use images instead of real gradients
- Artists can create the desired gradients

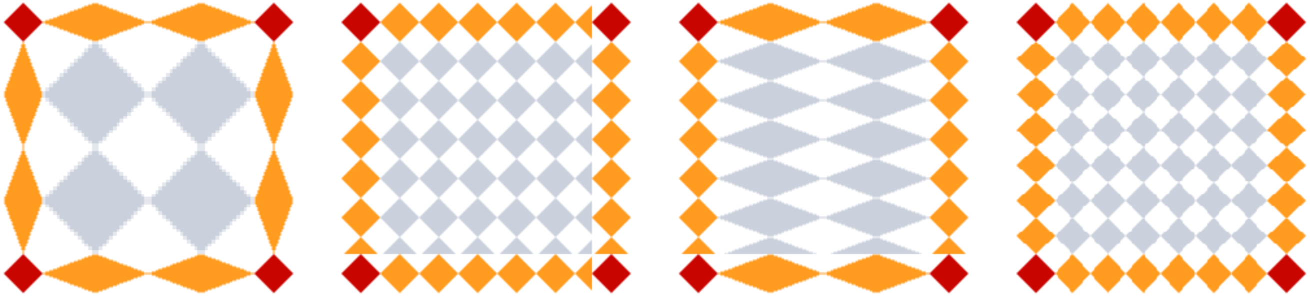
qml-composing-uis/ex-elements/image-gradients.qml

# 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

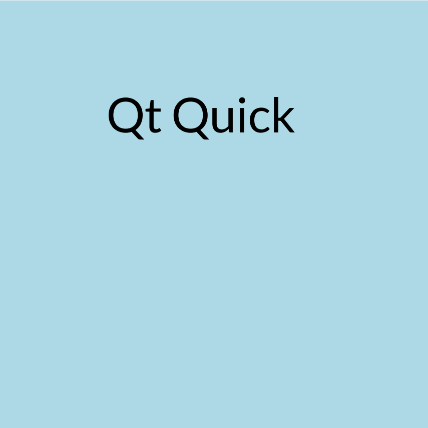


```
BorderImage {
    source: "content/colors.png"
    border { left: 30; top: 30; right: 30; bottom: 30; }
    horizontalTileMode: BorderImage.Stretch
    verticalTileMode: BorderImage.Repeat
    // ...
}
```

Demo: <Qt Examples>/declarative/imageelements/borderimage

# Text Type

```
Rectangle {
    width: 400; height: 400; color: "lightblue"
    Text {
        x: parent.width * 0.25; y: parent.height * 0.25
        text: qsTr("Qt Quick”)
        font { family: "Helvetica";
               pixelSize: parent.width * 0.1 }
    }
}
// fontSizeMode property is another way to do sclaing
```

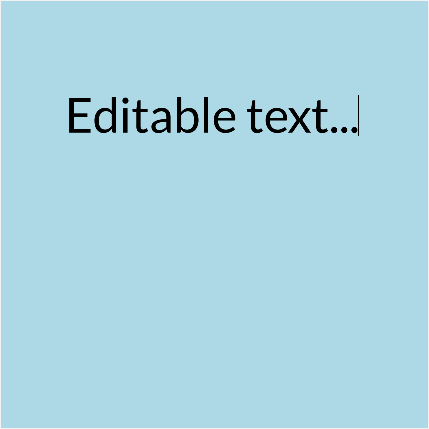


- Width and height determined by the font metrics and text
- Can also use HTML tags in the text:
  - `"<html><b>Qt Quick</b></html>"`
- Rectangle size could depend on the font size
  - `FontMetrics { id: metrics: font.family: “Courier” }`
  - `Rectangle { height: metrics.height * nofRows`

qml-composing-uis/ex-elements/text.qml

# TextInput

```
Rectangle {
    width: 400; height: 400; color: "lightblue"
    TextInput {
        x: parent.width * 0.25
        y: parent.height * 0.25
        width: parent.width * 0.75
        text: qsTr("Editable text")
        font { family: "Helvetica";
               pixelSize: parent.height * 0.1 }
        wrapMode: Text.WordWrap
    }
}
```

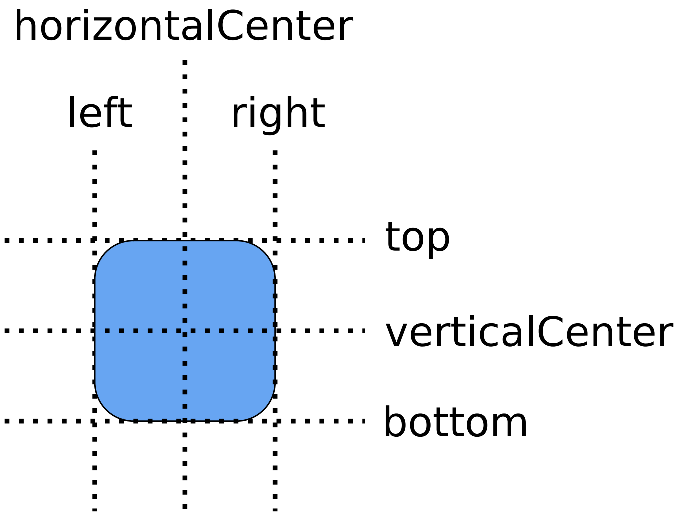


- No decoration (not a `QLineEdit` widget)
- Gets the focus when clicked
  - Need something to click on
- Property `text` changes as the user types
- Method `qsTr()` marks the string translatable

qml-composing-uis/ex-elements/textInput.qml

# Anchors

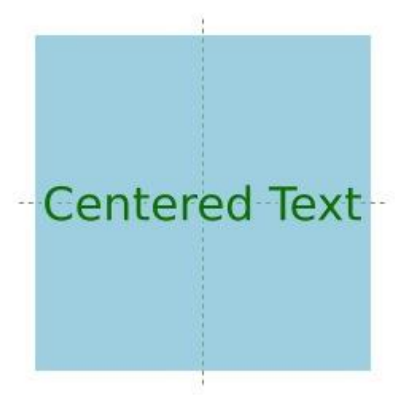
- Used to position and align items
- Line up the edges or central lines of items
- Anchors refer to
  - Other items (`centerIn`, `fill`)
  - Anchors of other items (`left`, `top`)



Documentation: [Positioning and anchors]

# Anchors

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

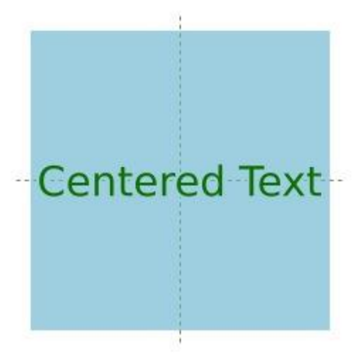


- anchors.centerIn centers the Text item in the Rectangle
    - Refers to an item not an anchor

qml-composing-uis/ex-anchor-layout/anchors.qml

# Anchors

```
Text {
    text: qsTr("Centered text”)
    color: "green"
    font { family: "Helvetica"; pixelSize: … }
    anchors.centerIn: parent
    }
}
```

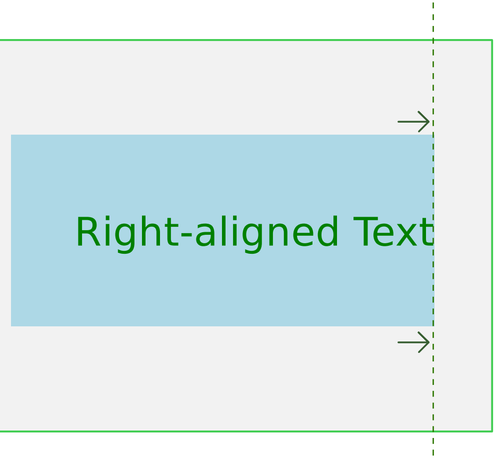


- Each item can refer to its parent item
  - Using the parent ID
- Can refer to ancestors and named children of ancestors

qml-composing-uis/ex-anchor-layout/anchors2.qml

# Anchors

```
Text {
    y: 34
    text: qsTr("Right-aligned text”)
    color: "green"
    font { family: "Helvetica"; pixelSize: … }
    anchors.right: parent.right
```

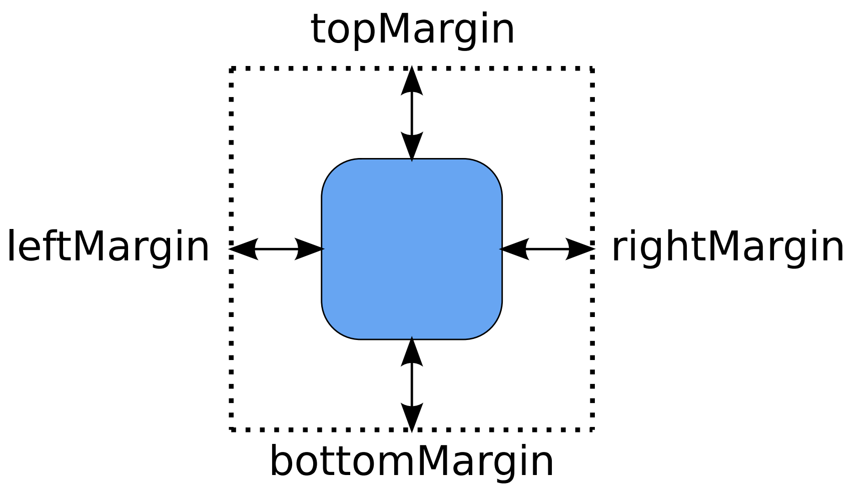


- Connecting anchors together
- Anchors of other items are referred to directly
    - Use `parent.right`
    - Not `parent.anchors.right`

qml-composing-uis/ex-anchor-layout/anchor-to-anchor.qml

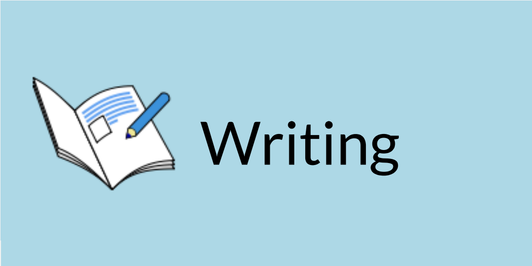
# Margins

- Used with anchors to add space
- Specify distances
  - In pixels
  - Between items 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: qsTr("Writing”); font.pixelSize: 32
        anchors.left: book.right anchors.leftMargin: 32
        anchors.baseline: book.verticalCenter
    }
}
```



qml-composing-uis/ex-anchor-layout/alignment.qml

# 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

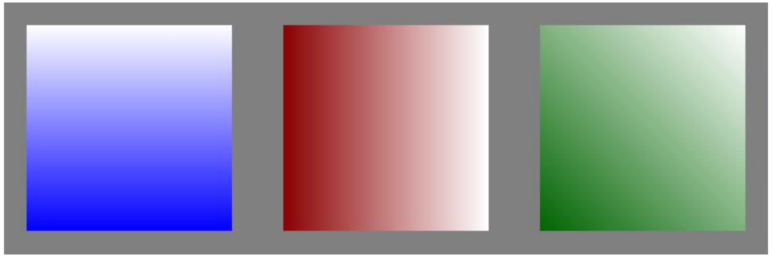
# Strategies for Use

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

# Lab – Color and Gradients

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?
   - The effect

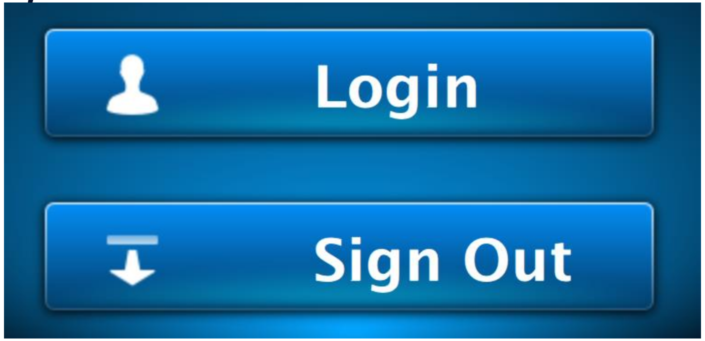


3. Describe another way to create these gradients?

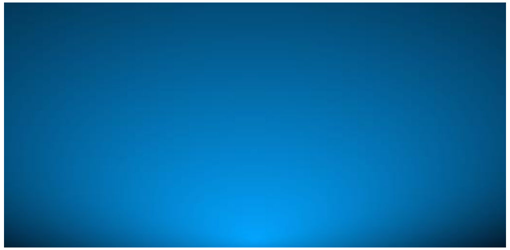
# Lab – Images and Text

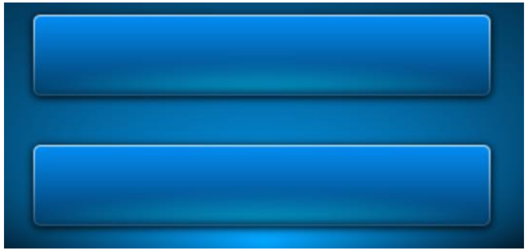
1. When creating an `Image`, how do you specify the location of the image file?
2. By default, images are rotated about a point inside the image. Where is this point?
3. How do you change the text in a `Text` QML type?

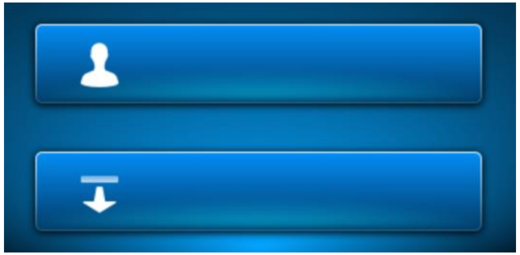
# Lab – Images, Text, and Anchors



- Create a user interface similar to the one shown above.
- Hint: Use the background image supplied in the common images directory.







qml-composing-uis/lab-text-images-anchors

# Contents

- Mouse Input
- Touch Input
- Keyboard Input

# Objectives

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

Demo: <Qt Examples>/declarative/toys/corkboards

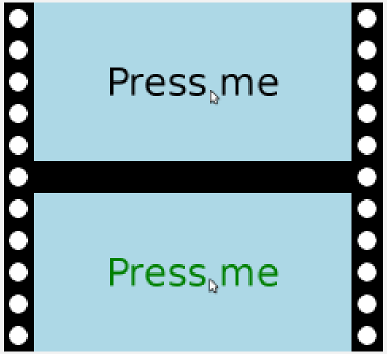
# Mouse Areas

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

Documentation: MouseArea QML Type

# Clickable Mouse Area

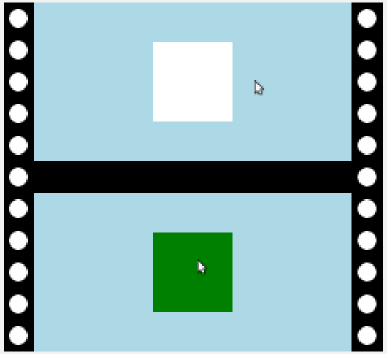
```
Rectangle {
    width: 400; height: 200; color: "lightblue”
    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: qsTr("Press me”); font.pixelSize: 48
        MouseArea {
            anchors.fill: parent
         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



qml-user-interaction/ex-mouse-input/mouse-pressed-signals.qml

# 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
          }
     }
}
```



qml-user-interaction/ex-mouse-input/hover-property.qml

# 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 objects
  - 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:
  - `real x, y`
  - `real prviousX, previousY`
  - `bool pressed`
  - `int pointId`
  - `real pressure`

# 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 item takes over touch handling.
  - Useful for undoing what was done on `onPressed`.

qml-user-interaction/ex-multi-touch/main.qml

# 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`

Demo: <Qt Examples>/declarative/toys/corkboards

# 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
    }
}
```

qml-user-interaction/ex-pinch

# Pinch Gestures

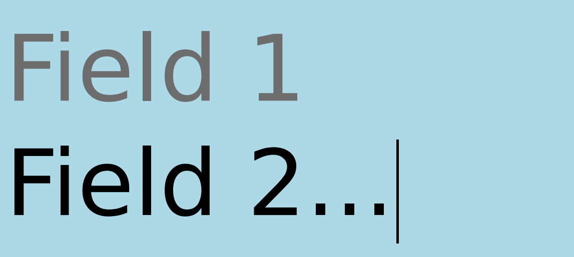
- Signals for manual pinch handling
  - `onPinchStarted(PinchEventpinch)`
  - `onPinchUpdated(PinchEventpinch)`
  - `onPinchFinished()`
- `PinchEvent` properties:
  - point1, point2, center
  - rotation
  - scale
  - accepted
    - set to false in the `onPinchStarted` handler if the gesture should not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

# Keyboard Input

- Basic keyboard input is handled in two different use cases:
- Accepting text input
  - QML types `TextInput` and `TextEdit`
- Navigation between items
  - Changing the focused item
  - directional(arrow keys), tab and backtab

# 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

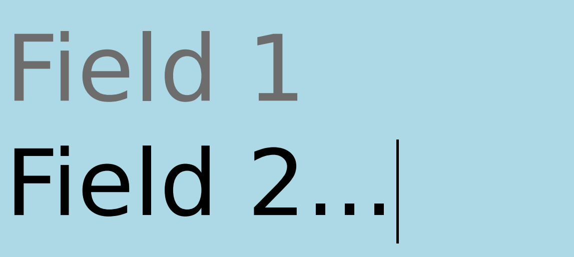


Field 1

Field 2...

# Using TextInputs

```
TextInput {
    id: upperTextInput
    anchors.left: parent.left
    anchors.right: parent.right
    text: "Field 1"; font.pixelSize: 32
    color: focus ? "black" : "gray"
    text: qsTr("Field") }
TextInput {
    anchors.left: parent.left
    anchors.top: upperTextInput.bottom
    anchors.right: parent.right
    text: qsTr("Field 2”); font.pixelSize: 32
    color: focus ? "black" : "gray"
}
```



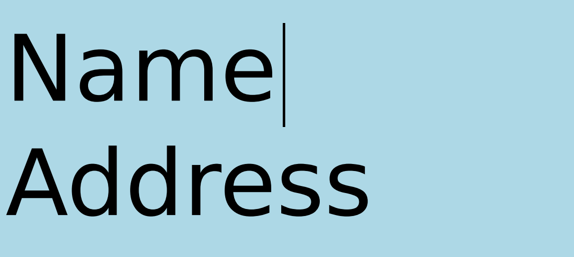
Field 1

Field 2...

qml-user-interaction/ex-key-input/textinputs.qml

# Focus Navigation

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



Name

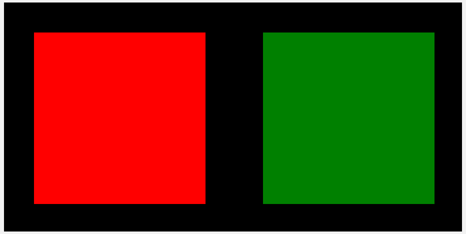
Address

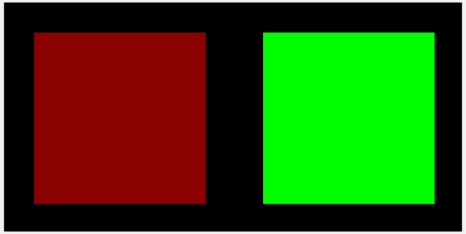
- The `name_field` item defines `KeyNavigation.tab`
  - Pressing Tab moves focus to the address_field item
- The `address_field` item defines `KeyNavigation.backtab`
  - Pressing **Shift+Tab** moves focus to the name_field item

qml-user-interaction/ex-key-input/tab-navigation.qml

# Key Navigation

```
Rectangle { id: leftRect
            anchors { top: … }
            color: focus ? "red" : "darkred"
            KeyNavigation.right: rightRect
            focus: true
}
Rectangle { id: rightRect
            anchors { top: … }
            color: focus ? "#00ff00" : "green"
            KeyNavigation.left: leftRect
}
```





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

qml-user-interaction/ex-key-input/key-navigation.qml

# Summary

Mouse and cursor input handling:

- QML type `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

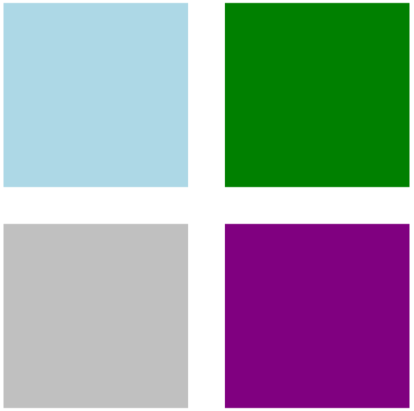
Key handling:

- QML types `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
- QML type `KeyNavigation` defines relationships between items
  - Enables focus to be moved
  - Using cursor keys, tab and backtab
  - Works with non-text-input items

# Lab – User Input

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

# Lab – Menu Screen



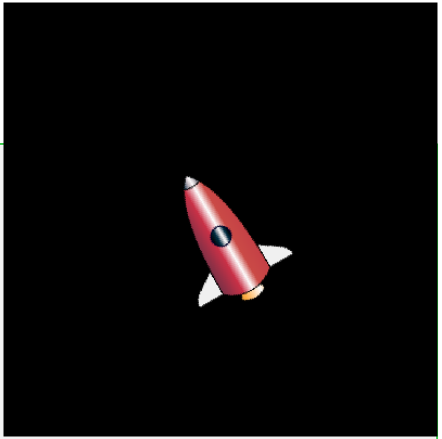
- Using the partial solution as a starting point, create a user interface similar to the one shown above with these features:
  - Items that change color when they have the focus
  - Clicking an item gives it the focus
  - The current focus can be moved using the cursor keys

qml-user-interaction/lab-menu-screen

# 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
        anchors.centerIn: parent
        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
}
```

qml-user-interaction/ex-key-input/key-press.qml

# 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
  - Useful, when several focusable instances created, e.g. button array
  - Without focus scope, the focus is given to the last instanced item

- 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

qml-user-interaction/ex-key-input/focus-scope.qml

# Contents

- Components
- Modules

# Objectives

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

# Custom Items and Components

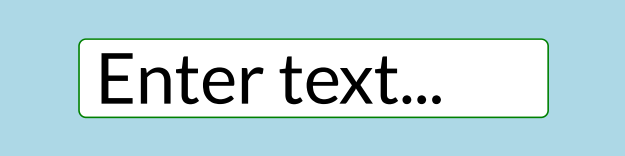
Two ways to create reusable user interface components:

- Custom items
  - Defined in separate files
  - One main item 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: qsTr("Enter text...”)
        color: focus ? "black" : "gray"
        font.pixelSize: parent.height - 4
    }
}
```



Enter text...

- Simple line edit
  - Based on undecorated `TextInput`
  - Stored in file `LineEdit.qml`

qml-modules/ex-modules-components/lineedit/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`

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

# Adding Custom Properties

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

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

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

Documentation: QML Object Attributes

# Custom Property Example

```
Rectangle {
     property string text: textInput.text // alias property preferred
     …
     TextInput {
          id: textInput
          …
          text: qsTr("Enter text...”)
     }
}
```

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

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

# Property Aliases

```
Rectangle {
    property alias text: textInput.text
    …
    TextInput {
        id: textInput
        …
        text: qsTr("Enter text...”)
    }
}
```

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

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

# Property Visibility Scope

- Defines the visibility rules for properties
- JavaScript has its own scope
  - QML does not interfere with that

```
Item {
    function return2() {
        var x = 2; // Does not interfere with item's x coordinate
        return x;
}
```

- Binding scope
  - Binding scope object’s properties may be accessed without qualification

```
Item {
    property int aProperty: 120
    x: aProperty // Item is a binding scope object
                 // Its properties can be accessed without qualification
```

# Property Visibility Scope

- Component scope
  - A union of object ids within the component and the component's root object's properties

```
delegate: Component {
    Rectangle {
        MouseArea {
            anchors.fill: parent
            console.log(qsTr("Item clicked"));
```

- Component instance hierarchy
  - Component instances can access the component scopes of their ancestors

```
Repeater { // Example QML type, which has a Component delegate
    delegate: Component {
        Rectangle {
            color: ancestorObjectId.color
```

# Adding Custom Signals

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

- Custom items can define their own signals
- Signal syntax: **`signal <name>[(<type> <value>, ...)]`**
- Handler syntax: **`on<Name>: <expression>`**

- Examples of signals and handlers:
  - Signal `clicked`
    - Handled by `onClicked`
  - Signal `checked(bool checkValue)`
    - Handled by `onChecked`
    - Argument passed as `checkValue`

# Defining a Custom Signal

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

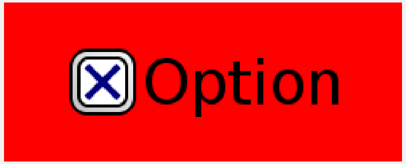
qml-modules/ex-modules-components/items/NewCheckBox.qml

# Emitting a Custom Signal

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

- `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`)

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

# Modules

Modules hold collections of QML types:

- Contain definitions of new types
- Allow and promote re-use of types 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

Documentation: QML Modules

# Custom Item Revisited

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

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

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

# 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

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

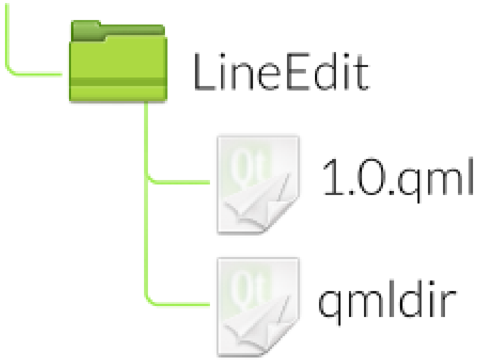
# Versioning Modules

- 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

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

# 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

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

# 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

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

# Contents

- States
- State Conditions
- 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

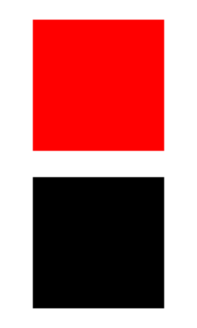
States manage named items

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

Documentation: QML States

# States 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

# 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

qml-states-transitions/ex-states/states.qml

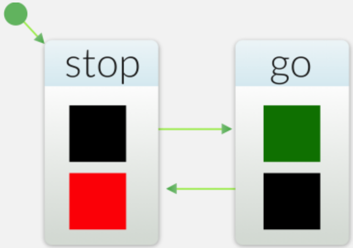
# Setting the State

- 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` QML type:

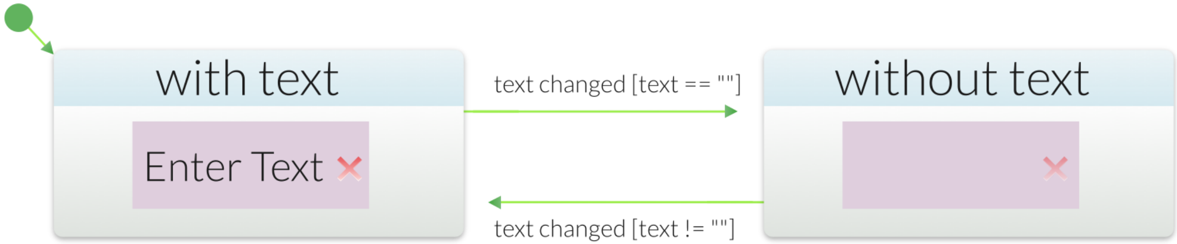
```
State {
    name: "go"
    PropertyChanges { target: stopLight; color: "black” }
    PropertyChanges { target: goLight; color: "green" }
}
```

- Acts on a target item named using the target property
  - The target refers to an id
- Applies the other property definitions to the target item
  - One `PropertyChanges` instance 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:

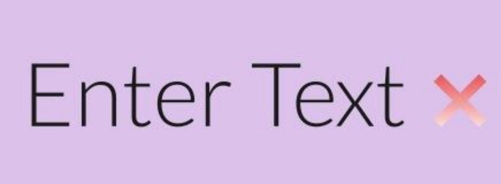
- Let the State decide when to be active
  - Using conditions to determine if a state is active
- Define the when property
  - Using an expression that evaluates to true or false
- Only one state in a states list should be active
  - Ensure when is true for only one state



qml-states-transitions/ex-states/states-when.qml

# State Conditions Example

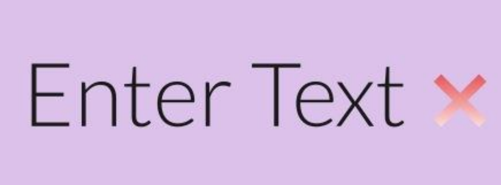
```
TextInput { id: textField
            text: "Enter text..."
            … }
Image { id: clearButton
        source: "../images/clear.svg”
        …
        MouseArea { anchors.fill: parent
                    onClicked: textField.text = "" }
}
```

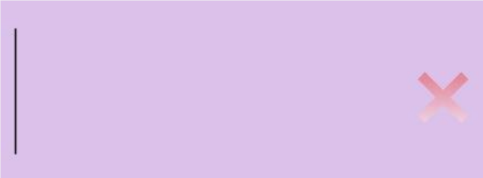


- Define default property values and actions

# 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 }
    }
]
```

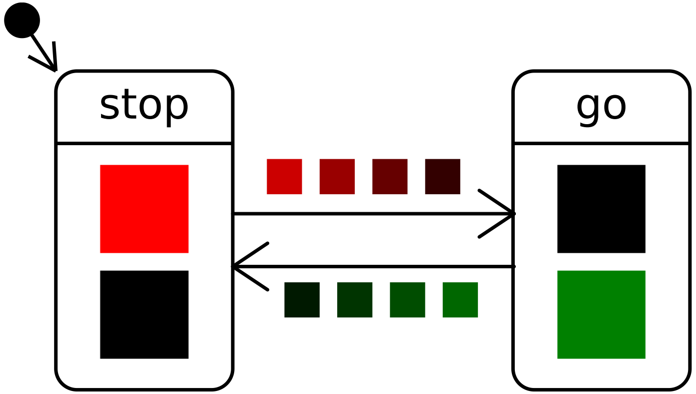




- A clear button that fades out when there is no text
- Do not need to define `state`

# 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...

qml-states-transitions/ex-transitions/transitions.qml

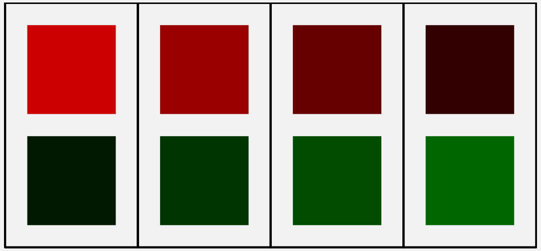
# Transitions Example

```
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 }
} ]
```

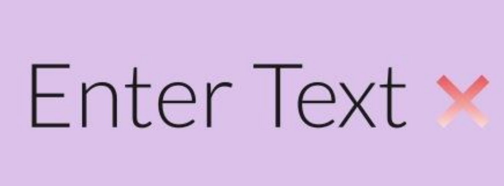


- Use "*" to represent any state
- Now the same transition is used whenever the state changes
- Both lights fade at the same time

qml-states-transitions/ex-transitions/transitions-multi.qml

# 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

qml-states-transitions/ex-transitions/transitions-reversible.qml

# 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 item when its parent changes
- QML type `ParentAnimation` applies only when changing the parent with `ParentChange` in a state change

qml-states-transitions/ex-animations/parent-animation.qml

# 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 item when its anchors change
- QML type `AnchorAnimation` applies only when changing the anchors with `AnchorChanges` in a state change

qml-states-transitions/ex-animations/anchors-animation.qml

# 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` 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

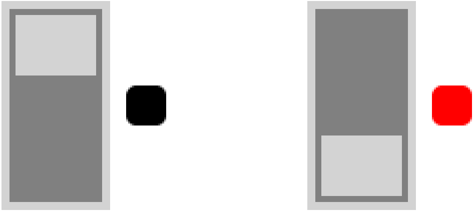
- Transitions refer to the states they are between
  - Using the `from` and `to` properties
  - Using a wildcard value, `"*",` to mean any state

- Transitions can be reversible
  - Used when the `from` and `to` properties are reversed

# Questions – States and Transitions

- 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?

# Lab – Light Switch



- Using the partial solutions as hints, create a user interface similar to the one shown above.
- Adapt the reversible transition code from earlier and add it to the example.

qml-states-transitions/lab-switch

# Contents

- Animations
- Easing Curves
- Animation Groups

# Objectives

Can apply animations to user interfaces:

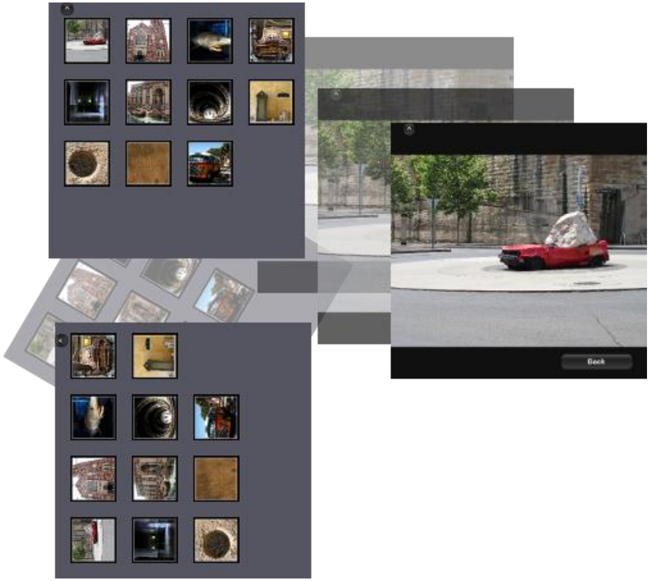
- Understanding of basic concepts
  - Number and property animations
  - Easing curves

- Ability to queue and group animations
  - Sequential and parallel animations
  - Pausing animations

- Knowledge of specialized animations
  - Color and rotation animations

# Why Use Animations, States and Transitions?

- 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!



qml-animations/ex-thumbnailexplorer/thumbnailexplorer.qml

# Animations

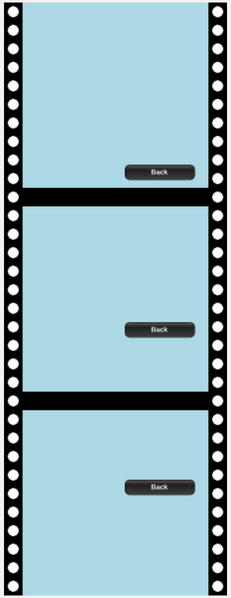
Animations can be applied to any item

- 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

Documentation: Animations in QML

# 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
        }
    }
}
```



qml-animations/ex-animations/number-animation.qml

# Number Animations

Number animations change the values of numeric properties

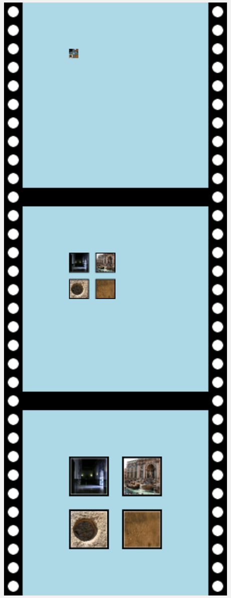
```
NumberAnimation on y {
      from: 350;
      to: 150
      duration: 1000
}
```

- 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

qml-animations/ex-animations/number-animation.qml

# 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
        }
    }
}
```



qml-animations/ex-animations/property-animation.qml

# Property Animations

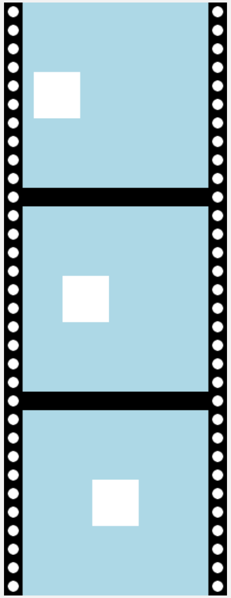
Property animations change named properties of a target

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

- Defined separately to the target item
- 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
    }
}
```



qml-animations/ex-animations/number-animation2.qml

# 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 QML Type

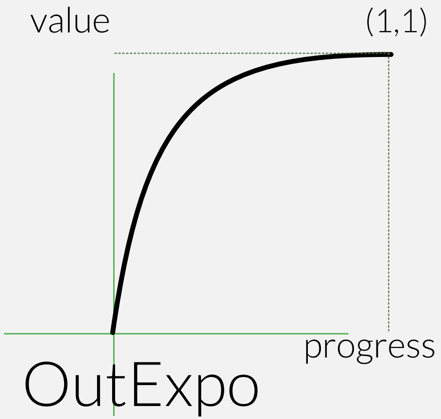
- Behavior allows you to set up an animation whenever a property changes.

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

qml-animations/ex-animations/spring-animation.qml

# 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"
        }
    }
}
```



qml-animations/ex-animations/easing-curve.qml

# 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

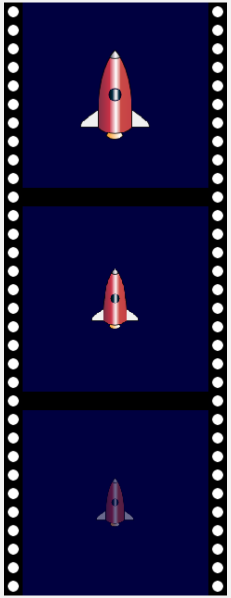
- `SequentialAnimation` defines a sequence
  - With each child animation run in sequence
- For example:
  - A rescaling animation, followed by an opacity changing animation

- `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

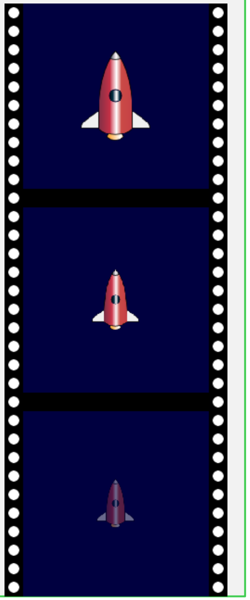
```
SequentialAnimation {
    NumberAnimation {
        target: rocket;
        properties: "scale"
        from: 1.0; to: 0.5; duration: 1000
    }
    NumberAnimation {
        target: rocket;
        properties: "opacity"
        from: 1.0; to: 0.0; duration: 1000
    }
    running: true
}
```



qml-animations/ex-animations/sequential-animation.qml

# 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
}
```



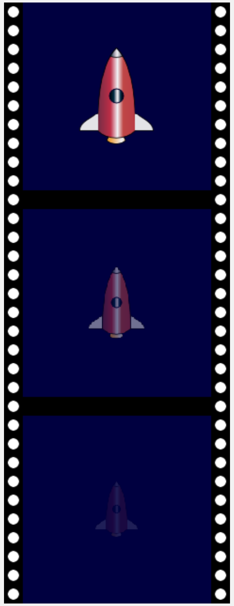
- Child objects define a two-stage animation:
  - First ,the rocket is scaled down and then it fades out
- `SequentialAnimation` does not itself have a `target`
  - It only groups other animations

# 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
}
```



qml-animations/ex-animations/parallel-animation.qml

# Other Animations

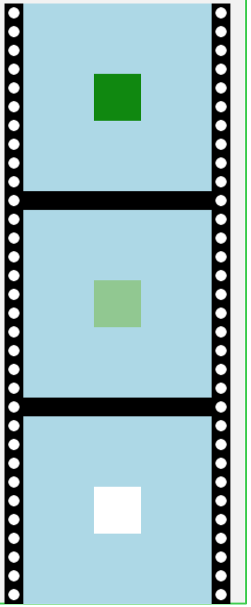
Other animations

- `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 Animation

- 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
}
```

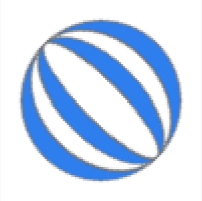


# Rotation Animation

- `RotationAnimation` describes rotation of items
- Easier to use than `NumberAnimation` for the same purpose
- Applied to the `rotation` property of an item
- 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
    }
}
```



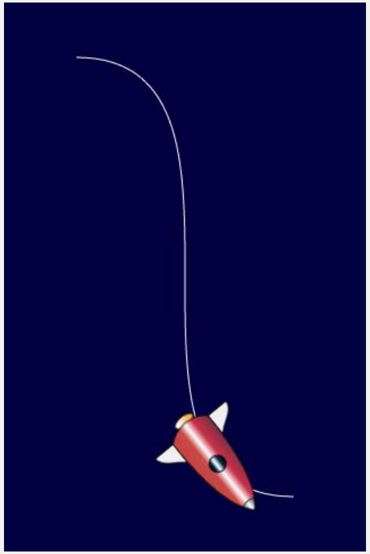
- 1 second animation
- Counter-clockwise from 45° to 315°
  - Shortest angle of rotation is via 0°

# Path Animation

- QML type `PathAnimation` animates an item along a path
- Manipulates the `x`, `y` and `rotation` properties of an item
- The `target` QML type 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` QML type and its helpers
  - `PathLine`, `PathQuad`, `PathCubic`, `PathCurve`, `PathArc`, `PathSvg`

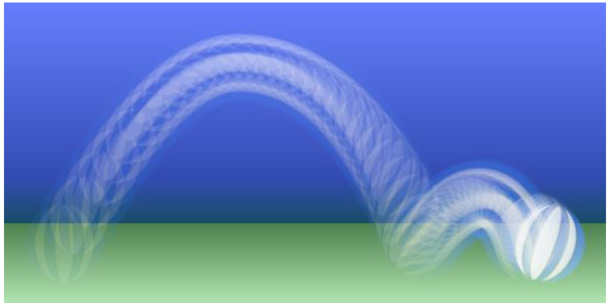
# Path Animation

```
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
        }
    }
}
```



qml-animations/ex-animations/path-animation.qml

# Lab: Bouncing Ball



Starting from the first partial solution:

- Make the ball start from the ground and return to the ground.
- Make the ball travel from left to right
- Add rotation, so the ball completes just over one rotation
- Reorganize the animations using sequential and parallel animations
- Make the animation start when the ball is clicked
- Add decoration (ground and sky)

qml-animations/lab-animations

# Contents

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

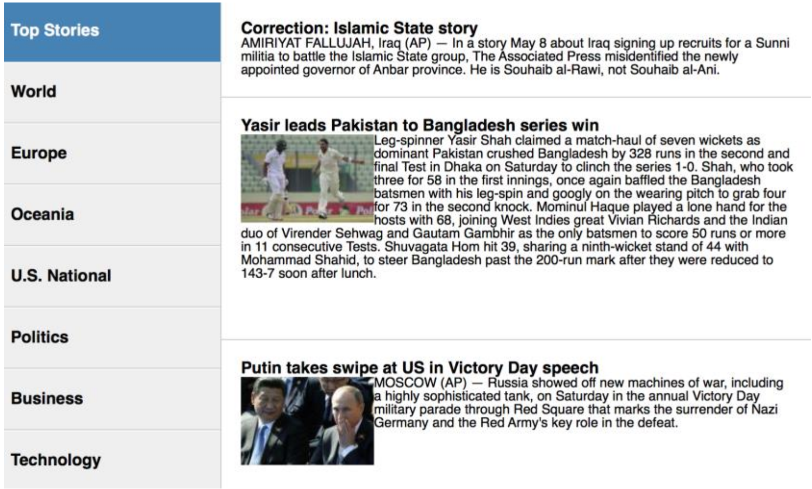
# Objectives

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



**Correction: Islamic State story**
AMIRIYAT FALLUJAH, Iraq (AP) — In a story May 8 about Iraq signing up recruits for a Sunni militia to battle the Islamic State group, The Associated Press misidentified the newly appointed governor of Anbar province. He is Souhaib al-Rawi, not Souhaib al-Ani.

**Yasir leads Pakistan to Bangladesh series win**
Leg-spinner Yasir Shah claimed a match-haul of seven wickets as dominant Pakistan crushed Bangladesh by 328 runs in the second and final Test in Dhaka on Saturday to clinch the series 1-0. Shah, who took three for 58 in the first innings, once again baffled the Bangladesh batsmen with his leg-spin and googly on the wearing pitch to grab four for 73 in the second knock. Mominul Haque played a lone hand for the hosts with 68, joining West Indies great Vivian Richards and the Indian duo of Virender Sehwag and Gautam Gambhir as the only batsmen to score 50 runs or more in 11 consecutive Tests. Shuvagata Hom hit 39, sharing a ninth-wicket stand of 44 with Mohammad Shahid, to steer Bangladesh past the 200-run mark after they were reduced to 143-7 soon after lunch.

**Putin takes swipe at US in Victory Day speech**
MOSCOW (AP) — Russia showed off new machines of war, including a highly sophisticated tank, on Saturday in the annual Victory Day military parade through Red Square that marks the surrender of Nazi Germany and the Red Army's key role in the defeat.

Demo: <Qt Examples>/declarative/demos/rssnews/rssnews.pro

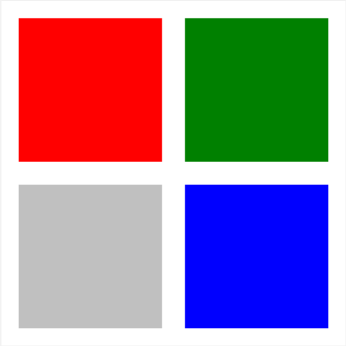
# Arranging Items

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

qml-presenting-data/ex-arranging-items/grid-rectangles.qml

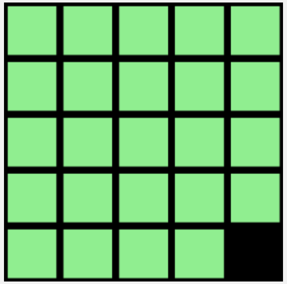
# Repeating Items

```
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 { id: grid
        x: 5; y: 5 rows: 5; columns: 5; spacing: 10
        Repeater {
            model: 24
            Rectangle {
                width: root.width / grid.columns - grid.spacing
                height: root.height / grid.rows - grid.spacing
                color: "lightgreen" }
        }
    }
}
```

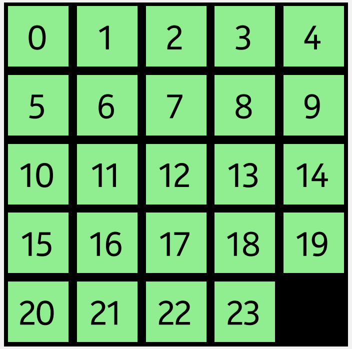


- `Repeater` takes data from a model
  - Just a number in this case
- Creates items based on the template item
  - A light green rectangle

qml-presenting-data/ex-arranging-items/repeater-gird.qml

# 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: …; height: …; color: "lightgreen"
                Text {
                    text: index
                    font.pointSize: 30
                    anchors.centerIn: parent }
            }
        }
    }
}
```



| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | |

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

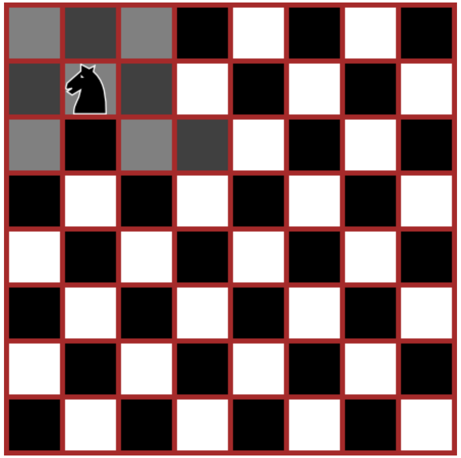
qml-presenting-data/ex-arranging-items/repeater-gird-index.qml

# Positioner Hints and Tips

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

# Lab – Chess Board

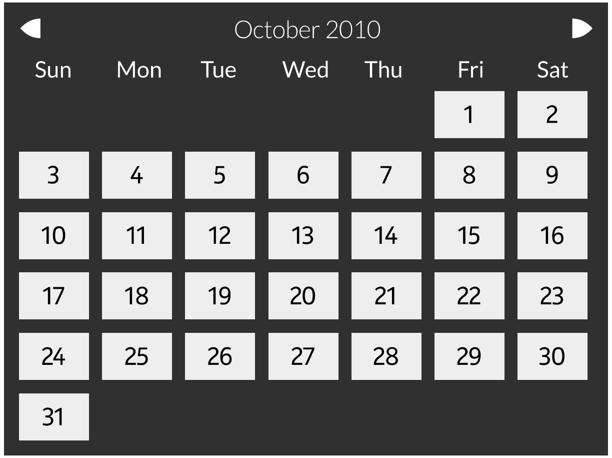
- Start by creating a chess board using a `Grid` and a `Repeater`
  - Use the `index` to create a checker pattern
- Use the `knight.png` image to create a piece that can be placed on any square
  - Bind its `x` and `y` properties to custom `cx` and `cy` properties
- Make each square clickable
  - Move the piece when a suitable square is clicked
- Make the model an `Array` that records which squares have been visited
- Make the board and piece separate components



qml-presenting-data/lab-chess-boardl

# Lab – Calendar

- Start by creating a chess board using a `Grid` and a `Repeater`
  - Put the grid inside an `Item`
  - Use the `index` to give each square a number
- Place a title above the grid
- Ensure that the current date is highlighted
- Use the `left.png` and `right.png` images to create buttons on each side of the title
- Make the buttons navigate to the next and previous months
- Add a header showing the days of the week



◀ October 2010 ▶

| Sun | Mon | Tue | Wed | Thu | Fri | Sat |
|---|---|---|---|---|---|---|
| | | | | | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | | | | | | |

qml-presenting-data/lab-calendar

# 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`
  - Contains child items that are supplied to views
- Visual data model: `DelegateModel`
  - Contains an interface to an underlying model
  - Supplies a delegate for rendering
  - Supports delegate sharing between the views

Documentation: Data Models

# List Models

- List models contain simple sequences of list 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" }
}
```

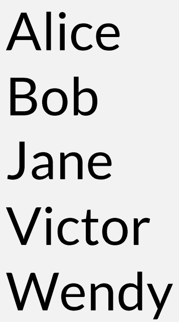
```
Alice
Bob
Jane
Victor
Wendy
```

- 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

qml-presenting-data/ex-models-views/list-model-list-view.qml

# Defining a Delegate

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



```
Alice
Bob
Jane
Victor
Wendy
```

- 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 list element

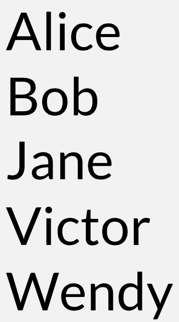
# Delegates, Contexts, and Attached Properties

- Each property is exposed in one context
  - Defines how the property can be accessed together with the scope rules
- Views, `Repeater`, `Instantiator`, expose properties to delegate instances in sub-contexts
  - This allows the parent to expose properties, visible in the sub-context only (`index, modelData`)
  - `modelData` is exposed, if the model is a string or object list
- Views also provide attached properties to delegates

```
Component {
    id: nameDelegate
    Text {
        property var listView: ListView.view
        text: name; font.pixelSize: 32
        color: (listView.currentIndex === index) ? "red" : "black"
    }
}
```

# Using a List Model

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



Alice
Bob
Jane
Victor
Wendy

- A `Repeater` fetches elements from `nameModel`
  - Using the delegate to display model 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)`
- Roles (item types) may be dynamic – `dynamicRoles` property set to true
  - Strongly discouraged
  - Using dynamic roles is 4-6 times slower than using static ones
  - Use for example `QVariantMap` instead

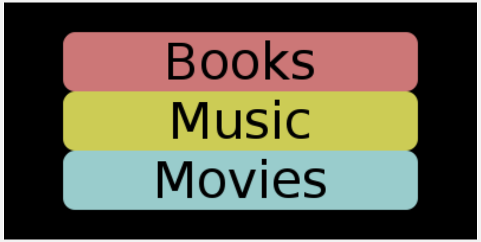
# List Model Hints

- **Note:** Model properties cannot shadow delegate properties:

```
ListModel {
    ListElement { text: "Alice" }
}

Component {
    Text {
        text: text; // Will not work
    }
}
```

# Defining an Object Model

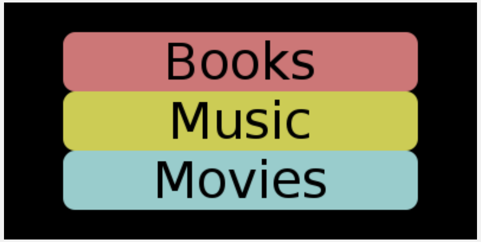
```
Rectangle {
    width: 400; height: 200; color: "black"
    ObjectModel {
        id: labels
        Rectangle { color: "#cc7777"; radius: 10.0
                    width: 300; height: 50
                    Text { anchors.fill: parent
                           font.pointSize: 32; text: "Books"
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

# Defining an Object Model

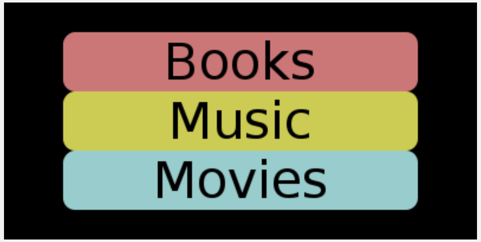
```
Rectangle {
    width: 400; height: 200; color: "black"
    ObjectModel {
        id: labels
        Rectangle { color: "#cc7777"; radius: 10.0
                    width: 300; height: 50
                    Text { anchors.fill: parent
                           font.pointSize: 32; text: "Books"
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

# Hierarchical Models

- QML models have named properties , used by views
- Hierarchical C++ models can be used with `DelegateModel` from `QtQml.Models`
- Provides access to `QAbstractItemModel` model index (`rootIndex`) also persistent ones
- Provides navigation functions to access child items and parent items in the model hierarchy
  - Navigate down - `modelIndex()`
  - Navigate up - `parentModelIndex()`

# Hierarchical Models

```
ListView {
    anchors.fill: parent
    model: DelegateModel {
        model: delegateModel
        delegate: Item {
            property var view: ListView.view
            width: childrenRect.width
            height: childrenRect.height

            MouseArea {
                anchors.fill: parent
                acceptedButtons: Qt.RightButton | Qt.LeftButton
                onClicked: {
                    if (mouse.button === Qt.LeftButton) {
                        if (model.hasModelChildren) {
                            ++level;
                            view.model.rootIndex =
                            view.model.modelIndex(index);
```

qml-presenting-data/ex-models-views/hierarchical-model/hierarchical-model.pro

# 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

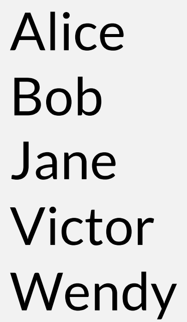
# List Views

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
}
```



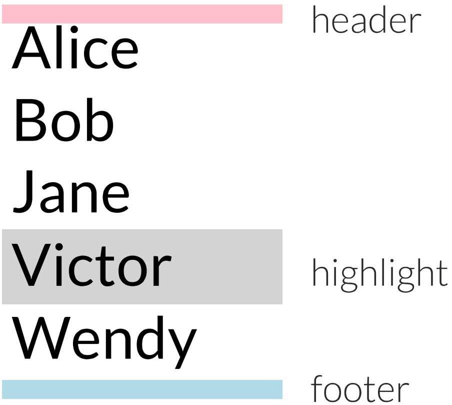
Alice
Bob
Jane
Victor
Wendy

- 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

qml-presenting-data/ex-models-views/list-model-list-view.qml

# Decoration and Navigation

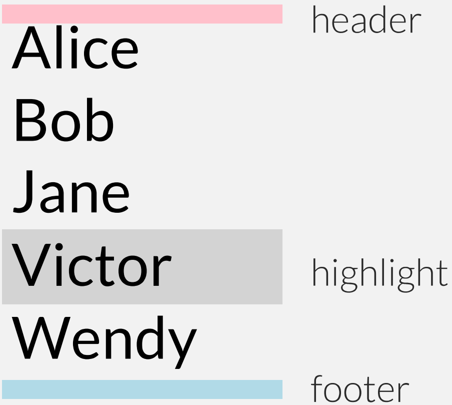
- By default, `ListView` is
  - Undecorated
  - A flickable surface (can be dragged and flicked)
- To add decoration:
  - With a `header` and `footer`
  - With a `highlight` item to show the current item
- To configure for navigation:
  - Set `focus` to allow keyboard navigation
  - Property `highlight`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



qml-presenting-data/ex-models-views/list-view-decoration.qml

# Decoration and Navigation

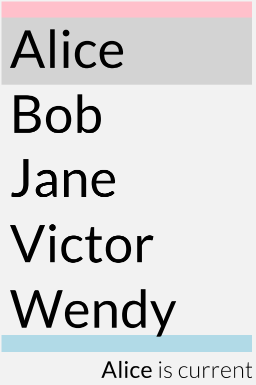
```
ListView {
    anchors.fill: parent
    model: nameModel
    delegate: nameDelegate
    focus: true
    clip: true
    header: Rectangle {
        width: parent.width; height: 10;
       color: "pink” }
    footer: Rectangle {
        width: parent.width; height: 10;
        color: "lightblue" }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



Alice
Bob
Jane
Victor
Wendy

**Alice** is current

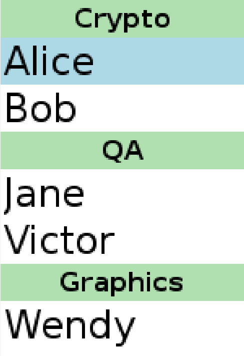
- Recall that, in this case, each item has a `text` property
  - re-use the listView's `currentItem's text`

qml-presenting-data/ex-models-views/list-view-current.item.qml

# 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

- Set up a list model with items:

```
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" }
}
```

- Define string properties to use in the delegate

ml-presenting-data/ex-models-views/list-model-grid-view.qml

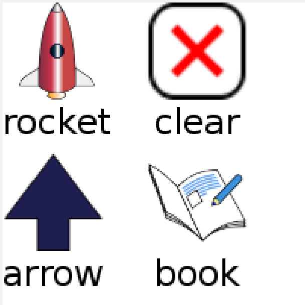
# Grid Views

- 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

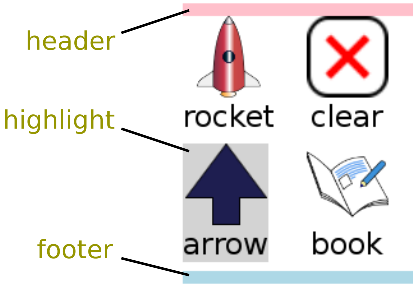
# Decoration and Navigation

Like `ListView`, `GridView` is

- Undecorated and a flickable surface

- To add decoration:
  - Define header and footer
  - Define highlight item to show the current item

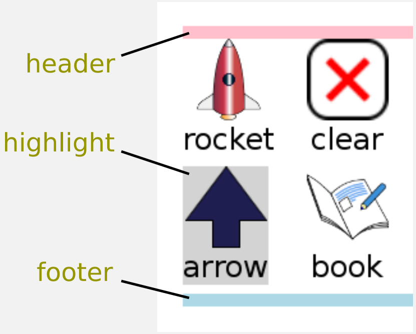
- To configure for navigation:
  - Set focus to allow keyboard navigation
  - Highlight also helps the user with navigation
  - Unset interactive to disable dragging and flicking



qml-presenting-data/ex-models-views/grid-view-decoration.qml

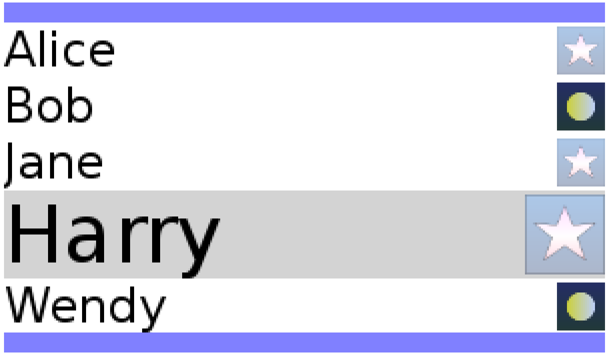
# Decoration and Navigation

```
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
}
```



# Lab – Contacts

- Create a ListItemModel, fill it with ListElement objects, each with
  - A name property
  - A file property referring to an image
- Add a ListView and a Component to use as a delegate
- Add header, footer and highlight properties to the view
- Add states and transitions to the delegate
  - Activate the state when the delegate item is current
  - Use a state condition with the ListView.isCurrentItem attached property
  - Make a transition that animates the height of the item



Alice
Bob
Jane
Harry
Wendy

# XML List Models

- Many data sources provide data in XML formats
- QML type `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

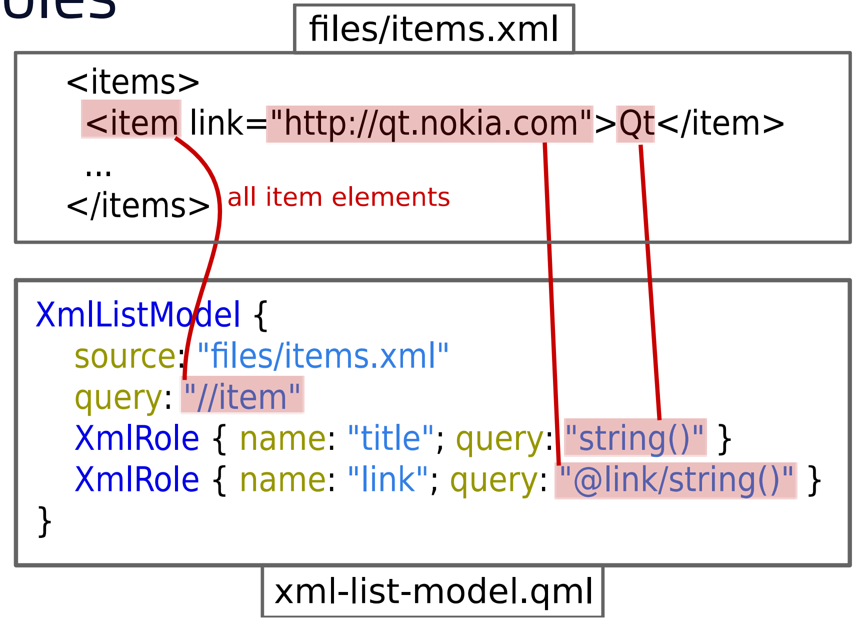
# Defining an XML List Model

```
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` instances

qml-presenting-data/ex-models-views/xml-list-model.qml

# XML Roles



files/items.xml

```
<items>
  <item link="http://qt.nokia.com">Qt</item>
  ...
</items>
```

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

xml-list-model.qml

**Result**

title: "Qt"

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

- QML type `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
- QML type `TitleDelegate` is defined in `TitleDelegate.qml`
  - Must be defined using a `Component` type

qml-presenting-data/ex-models-views/TitleDelegate.qml

# 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 }
            }
        }
    }
}
```

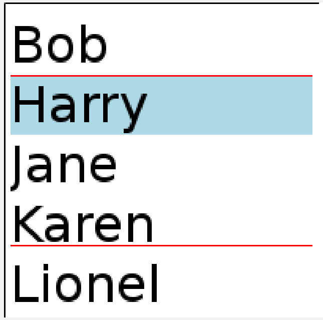
- Property `parent` refers to the view where it is used
- Properties `title` and `link` are properties exported by the model

# Customizing Views

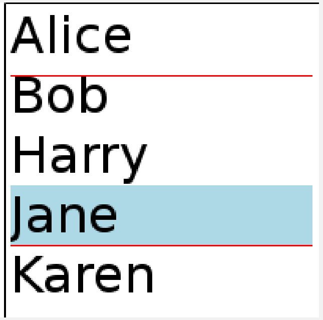
- All views are based on the `Flickable` item
  - Children will be parented to `contentItem` property
- Define the flicking behavior or disable it completely with `interactive: false`
  - `flickDirection, flickDeceleration, horizontalVelocity, verticalVelocity, boundsBehavior(StopAtBounds, DragOverBounds), pixelAligned, rebound, …`
- Key navigation of the highlighted item does not wrap around
  - Set k`eyNavigationWraps` 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

    …
}
```

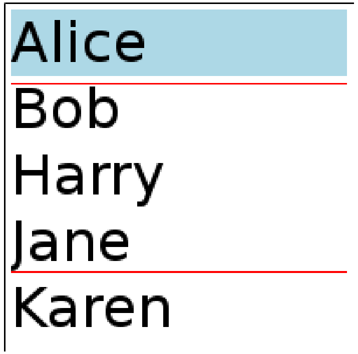


Bob
Harry
Jane
Karen
Lionel



Alice
Bob
Harry
Jane
Karen

- 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

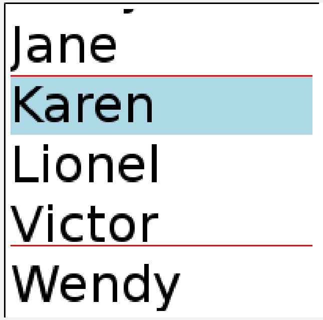


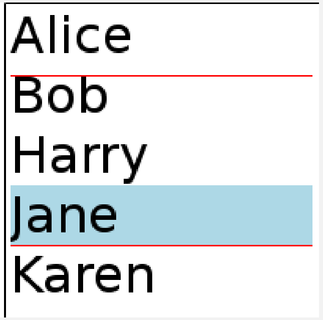
Alice
Bob
Harry
Jane
Karen

qml-presenting-data/ex-models-views/list-view-highlight-range-apply.qml

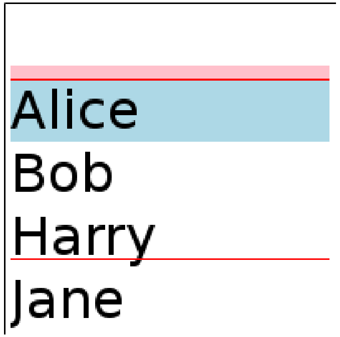
# Customizing Views

```
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



qml-presenting-data/ex-models-views/list-view-highlight-range-strict.qml

# 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
- Avoid heavy and complicated delegates
  - Heavy part of the delegate can be loaded on demand using `Loader`

# Contents

- Qt Quick Designer
- Qt Quick Controls
- Application Window
- Controls and Views
- Layouts
- Styling

# Objectives

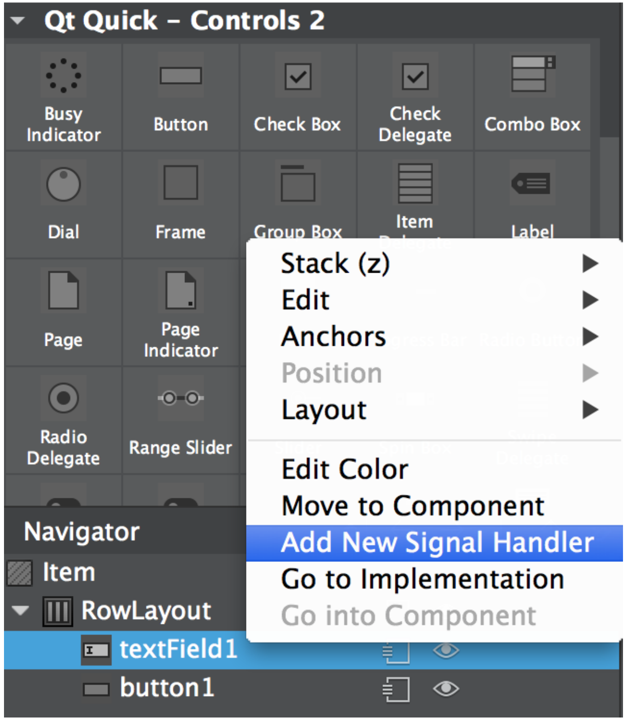
- Qt Quick Designer
  - UI design and layout
  - Forms and components
- Essential controls
  - Application window
  - Controls
  - Containers
  - Views
- Layouts
  - How to create expandable controls
  - Differences between anchors, positioners and QtQuick Layouts
- Styles
  - Learn how to style controls

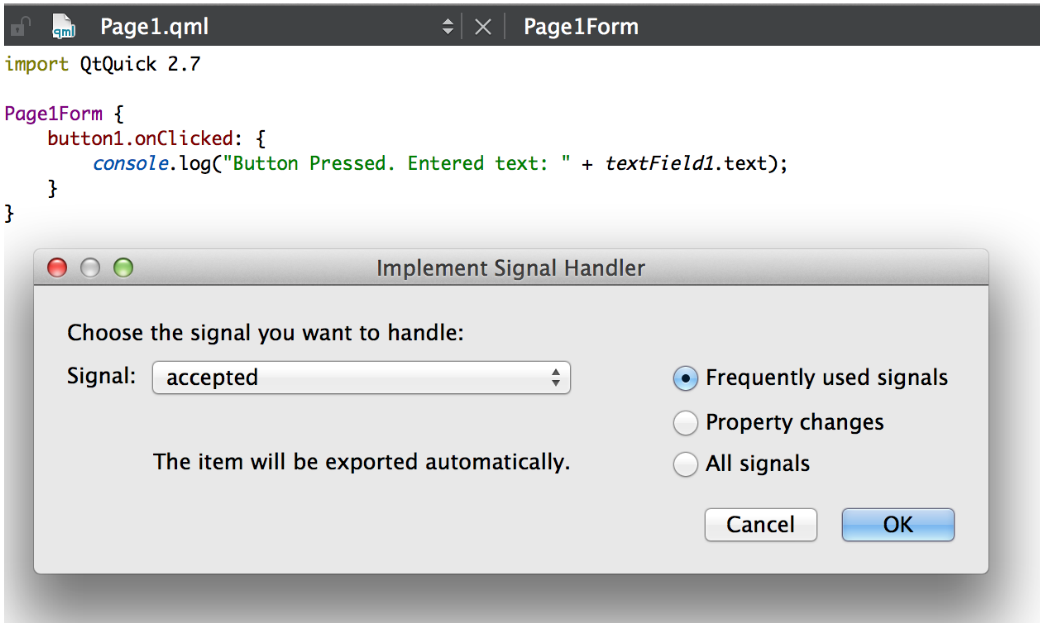
# Qt Quick Designer

- Allows composing UIs from QML types, Qt Quick Controls, and custom types
- Allows defining UI layout, creating properties and property bindings
- Clear separation between UI and business logic
  - UI designed in form files with `ui.qml` extension – do not use JavaScript code in forms
  - Business logic implemented in `.qml` and `.cpp` files

- Qt Quick Designer creates empty signal handlers in component files for selected signals

# Separation between the UI and Business Logic





```
import QtQuick 2.7

Page1Form {
    button1.onClicked: {
        console.log("Button Pressed. Entered text: " + textField1.text);
    }
}
```

Implement Signal Handler

Choose the signal you want to handle:

Signal: accepted

● Frequently used signals

○ Property changes

○ All signals

The item will be exported automatically.

- Drag and drop UI controls and specify their layout
- Add signal handlers into the component file – not in the form

# Qt Quick Controls

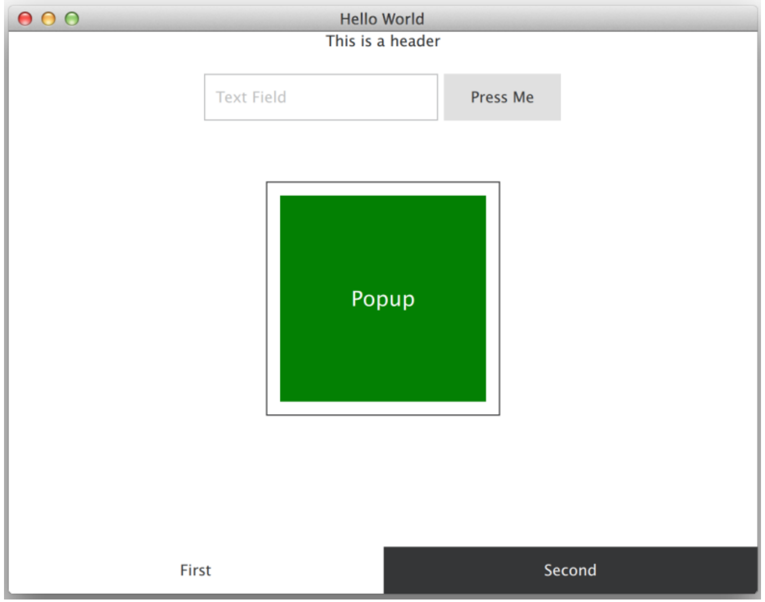
- Ready-made UI control QML types
- Two versions
  - Prefer using Qt Quick Controls 2
  - Some controls are only available in Qt Quick Controls 1 (`TableView`, `TreeView`)

- Qt Quick Controls 1 extend existing QML types (`Button` -> `FocusScope`)
  - Not optimal for memory consumption or performance point of view
  - Button: 15 `QQuickItems` (including 4 `Loaders`), totally 60 `QObjects`
  - Button in Qt Quick Controls 2: 4 `QQuickItems`, 7 `QObjects`, 7 times faster to create
- Biggest differences in event handling and styling
  - Qt Quick Controls 2 event handling in C++
  - Qt Quick Controls styled with control-specific style types
  - Qt Quick Controls 2 styled with application-global, configurable style

# Qt Quick Controls

- More than just UI controls

- `ApplicationWindow`
  - Window with header, footer, overlay (read-only popup window)
- Views
  - Layout and navigation
  - Scroll, split, stack, tab, table, and tree views
  - Stack and swipe views in Qt Quick Controls 2
- Layouts
  - Can shrink/expand items in the layout
- Deployed in several modules



# Application Window

- Window QML type instantiates QQuickWindow C++ class
  - Basic window management: geometry, visibility, window flags, background color
  - Scene management
  - Syncs with scene graph to render items on the scene
- Window belongs to one Screen (in QtQuick.Window  module) at a time
  - Screen cannot be instantiated in QML- would not make sense to create new screens
  - Useful Screen properties: screen orientation, screen physical dimensions and pixels, pixel density

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

# Application Window

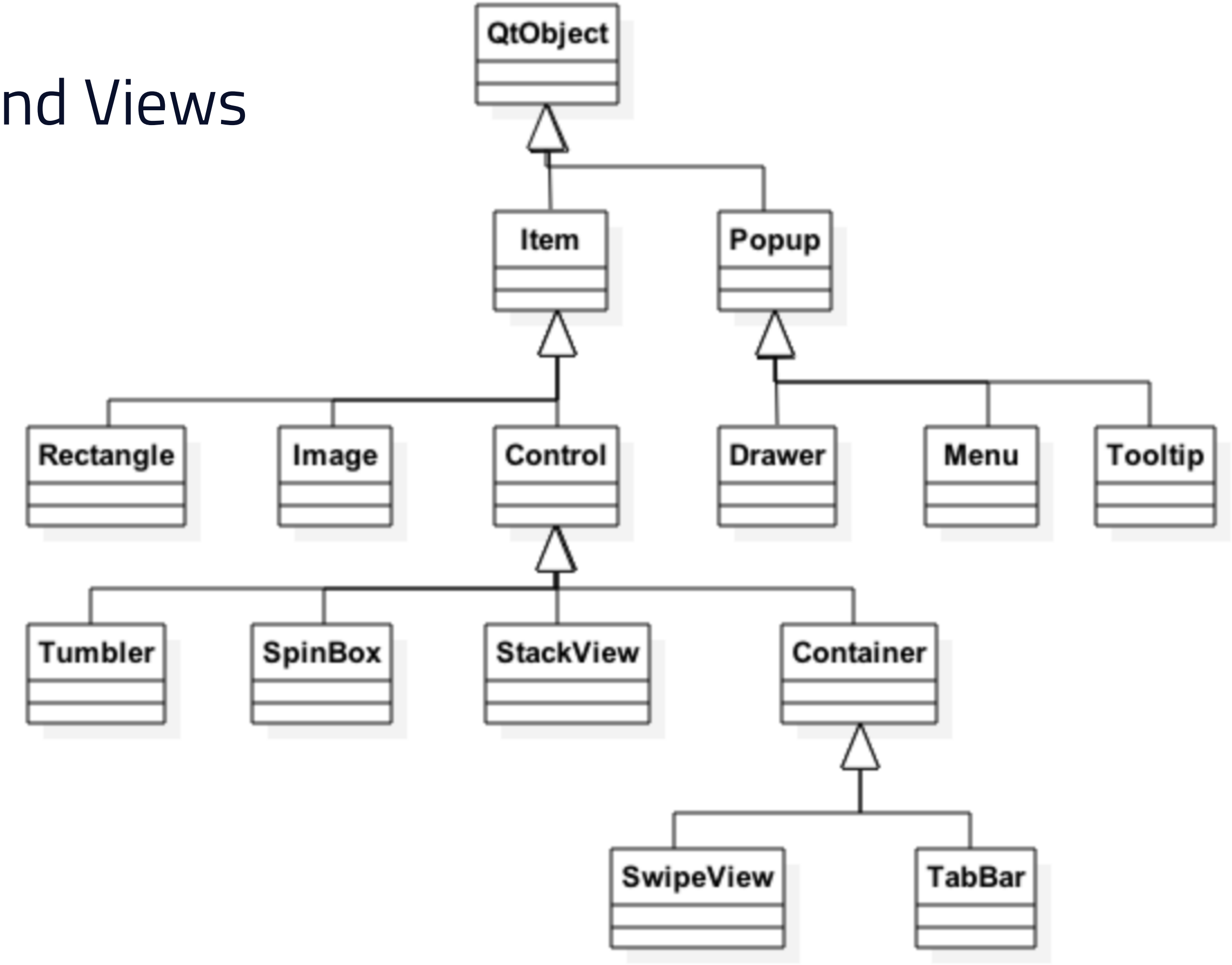
- Header and footer items
- Content
  - Window children
  - For example, a view, a container
- Background
  - Any item
- Overlay
  - Modal or non-modal popup
  - Modal popup dims the window

# Application Window

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

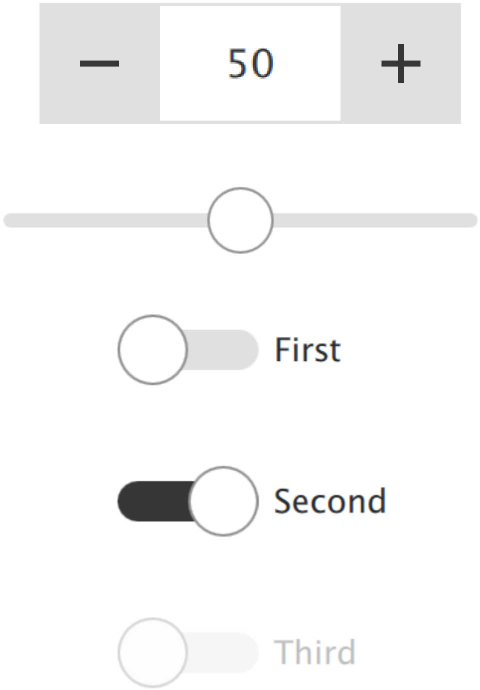
qml-controls/ex-application-window

# Controls and Views



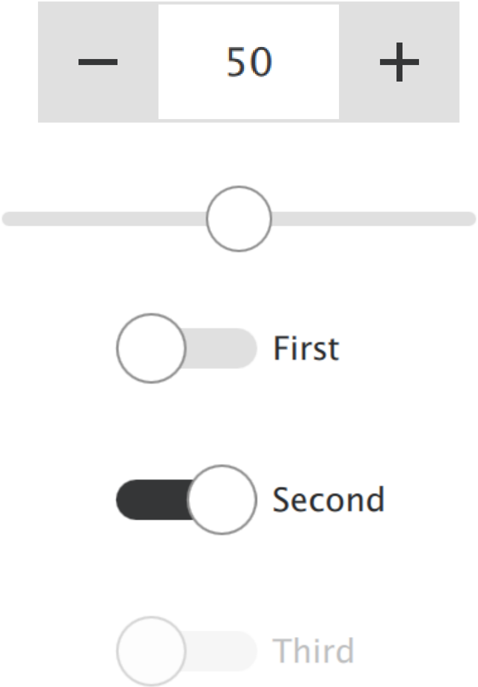
# Controls

- Receive input events and paint themselves on screen
- Define the layout and use a control in a window, a view or a container
- Focus
  - Any item may request to get active focus (property `focus`)
  - Focus policy (tab focus, click focus, strong focus) can be set for each control



# Controls

- Event handling
  - Many control 2 subtypes handle the events in C++ and provide signals to QML
  - Use property value change signal handlers or event related signal handlers
- Localization
  - Property `locale` can be set for a control. If not set, locale is inherited from the parent
  - Defines the layout direction, for example



qml-controls/ex-application-window

# Container Control

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

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

qml-controls/ex-container

# Adding Items Dynamically

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

qml-controls/ex-dynamic-tab-bar

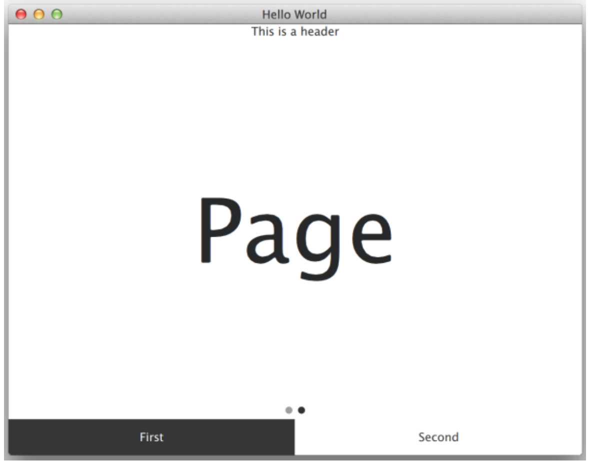
# Views – StackView

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

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

qml-controls/ex-views/ex-stackview

# Views – SwipeView



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

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

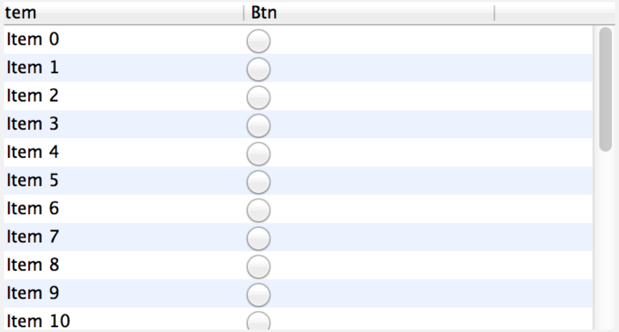
qml-controls/ex-views/ex-swipeview

# Qt Quick Controls 1 Views

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

# TableView

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



| tem | Btn |
|---|---|
| Item 0 | ○ |
| Item 1 | ○ |
| Item 2 | ○ |
| Item 3 | ○ |
| Item 4 | ○ |
| Item 5 | ○ |
| Item 6 | ○ |
| Item 7 | ○ |
| Item 8 | ○ |
| Item 9 | ○ |
| Item 10 | ○ |

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

qml-controls/ex-views/ex-tableview

# Layouts

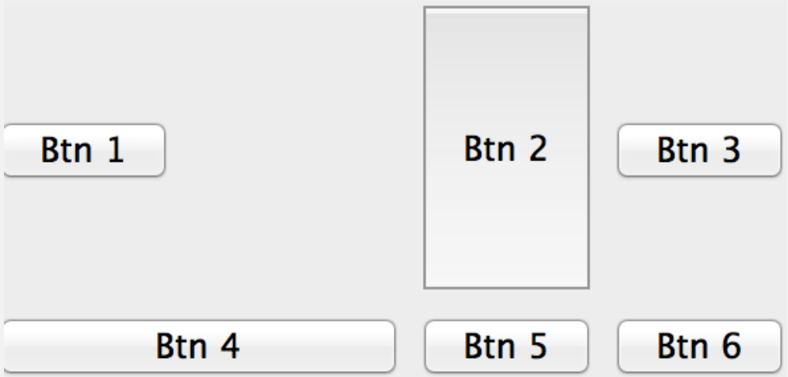
- Default behavior is similar to positioners

- However, can be used in the same way as `QLayout` works for widgets
  - The layout automatically defines the size of the items – no anchors or explicit width/height needed

- Just set the `Layout.fillHeight` or `Layout.fillWidth` to
  - `false` – if you do not want the layout to use all extra space for the item
  - `true` – if you want the extra space to be used to expand the item
  - Compare to `QSizePolicy::Expanding`

# Layouts Example

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

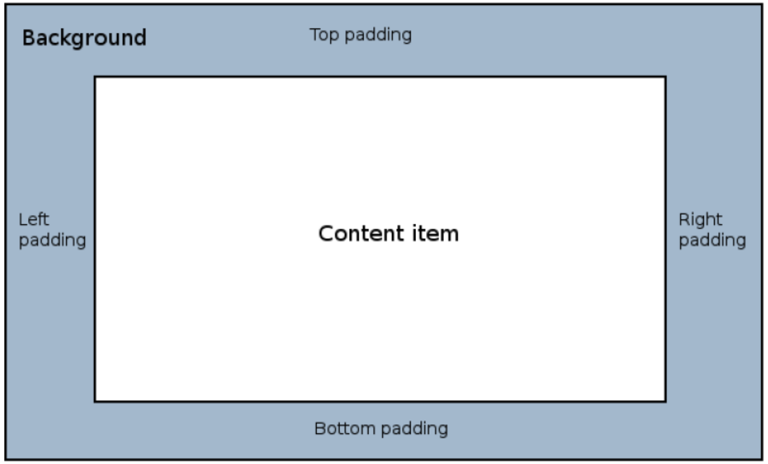


- Two buttons expand vertically and horizontally
- Other button dimensions are based on the text and font properties

qml-controls/ex-views/ex-layouts

# Styling

- It is possible to style one or a few controls or just define a custom style, applied to every control
- Each control has two customizable properties
  - `background`
  - `contentItem`
  - Customization reference defines what kind of property assignments each control expects
  - Alternative way is to copy existing style from `$QTDIR/qml/QtQuick/Controls.2/ControlType.qml` and modify that
- Deploy the customize control with your app
  - Provide a new, styled control type, such as `CustomSlider`

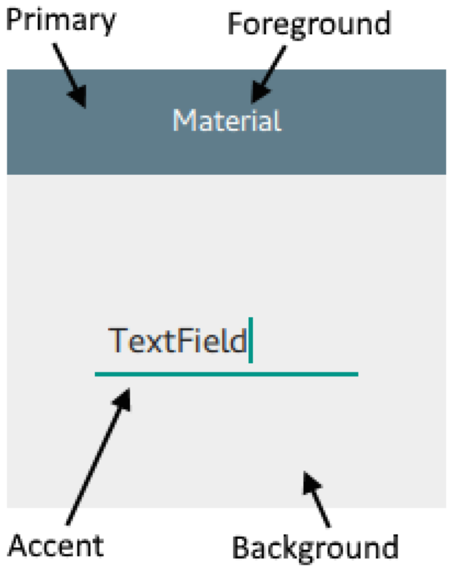


# Custom Styles

- To custom several controls, put the style files into a separate folder
- Style file names correspond to control type names, e.g. `Slider.qml`
- If there is no custom style for a control, the default one will be used
- Style folder may be defined using command line switch –style, environment variable or set with `QQuickStyle::setStyle()`

# Custom Styles

- Stylable properties, applied to several control types are defined in C++ as attached properties
  - For example, existing `Material` and `Universal` styles have a `theme` attached property, configurable via `qtquickcontrols2.conf`



```
[Controls]
Style=Material
[Universal]
Theme=Dark
[Material]
Accent=Red
```

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

qml-controls/ex-views/ex-styles

# Custom Style Property

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

```
import StyleModule 42.0

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

# Summary

- Qt Quick Controls provide ready-made UI controls
- `ApplicationWindow` provides a `QQuickWindow` with header and footer
- Window may contain any views, controls, items
  - Split view, stack view, tab view, scroll view
  - Button, slider, label etc.
- Controls may be styled in three ways
  - With custom background and content items
  - By changing existing style properties
  - By creating completely new styles

# 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

qml-cpp-integration/ex-clock

# 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();
}
```

qml-cpp-integration/ex-simpleviewer

# Setting up QtQuick

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

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 400; height: 300
```

qml-cpp-integration/ex-simpleviewer

# 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.
- All properties of an object may be exposed with `setContextObject(QObject *)`

```
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 };
```

# 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 = Q_NULLPTR);

private:
    QTimer *m_timer;
}
```

# Implementing the Class

- QML type `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 QML type 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
  - version 1.0 (first number is major; second Is minor)

- Available as the `Timer` type
  - The `Timer` type is an non-visual item
  - A subclass of `QObject`

# Other Registration Functions

- Singletons
  - No need to instantiate objects in QML as with QML register type
  - `QObject` or `QJSValue`
  - `qmlRegisterSingletonType<Type>(“module”, 1, 7, ”QMLType”, creationFunc);`
- Interfaces
  - For inheritance and object coercing
  - `template<type T>int qmlRegisterInterface(const char *typename)`
- Unavailable types
  - `qmlRegisterTypeNotAvailable()`
- Uncreatable type
  - For providing enumerations and attached properties
  - `qmlRegisterUncreatableType()`

# 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 }
    }
    ...
}
```

qml-cpp-integration/ex-simple-timer

# Adding Properties

- In the `main.qml` file:

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

- A new `interval` property

qml-cpp-integration/ex-timer-properties

# 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();
Q_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.

qml-cpp-integration/ex-timer-signals

# 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

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

qml-cpp-integration/ex-timer-slots

# 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

qml-cpp-integration/ex-methods

# Declaring a Method

- In the *timer.h* file:

```
public:
    explicit Timer(QObject* parent = Q_NULLPTR);

    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

- In the *timer.cpp* file:

```
int Timer::randomInterval(int min, int max) const
{
    int range = max - min;
    int msec = min + qrand() % range;
    qDebug() << "Random interval =" << msec << "msecs";
    return msec;
}
```

- Define the new `randomInterval()` function
  - The pseudo-random number generator has already been seeded
  - Simply return an `int`
  - Do not use the `Q_INVOKABLE` macro in the source file

# Summary of Signals, Slots and Methods

- Define signals
  - Connect to Qt signals with the `onSignal` syntax

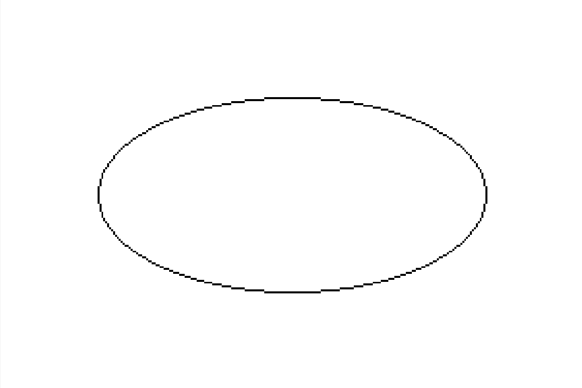
- Define QML-callable methods
  - Reuse slots as QML-callable methods
  - Methods that return values are marked using `Q_INVOKABLE`

# Exporting a QPainter based GUI Class

- Derive from `QQuickPaintedItem`
- Implement `paint(...)`
- Similar to non GUI classes:
  - Export object from C++
  - Import and use in QML
  - Properties, signals/slots, `Q_INVOKABLE`

# Exporting a QPainter based GUI Class cont'd.

```cpp
#include <QQuickPaintedItem>

class EllipseItem : public QQuickPaintedItem
{
    Q_OBJECT

public:
    EllipseItem(QQuickItem *parent = Q_NULLPTR);
    void paint(QPainter *painter);
};
```



# Exporting a QPainter based GUI Class cont'd.

```
EllipseItem::EllipseItem(QQuickItem *parent) :
    QQuickPaintedItem(parent)
{
}

void EllipseItem::paint(QPainter *painter)
{
    const qreal halfPenWidth = qMax(painter->pen().width() / 2.0, 1.0);

    QRectF rect = boundingRect();
    rect.adjust(halfPenWidth, halfPenWidth, -halfPenWidth, -halfPenWidth);

    painter->drawEllipse(rect);
}
```

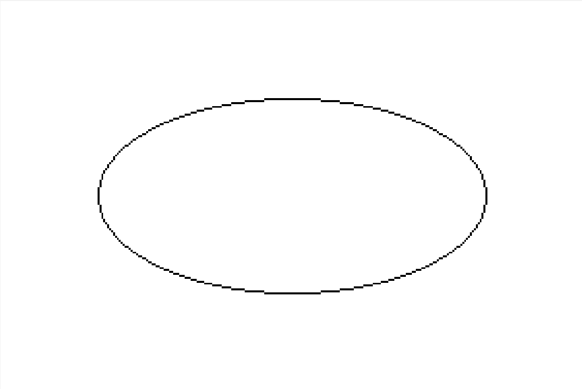
# Exporting a QPainter based GUI Class cont'd.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "ellipseitem.h"

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

    qmlRegisterType<EllipseItem>("Shapes", 1, 0, "Ellipse");

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/ellipse1.qml")));
    return app.exec();
}
```

# Exporting a QPainter based GUI Class cont'd.

- In the `ellipse1.qml` file:

```
import Shapes 1.0

Window {
    visible: true
    width: 300; height: 200
    Item {
        anchors.fill: parent
        Ellipse {
            x: 50; y: 50
            width: 200; height: 100
        }
    }
}
```



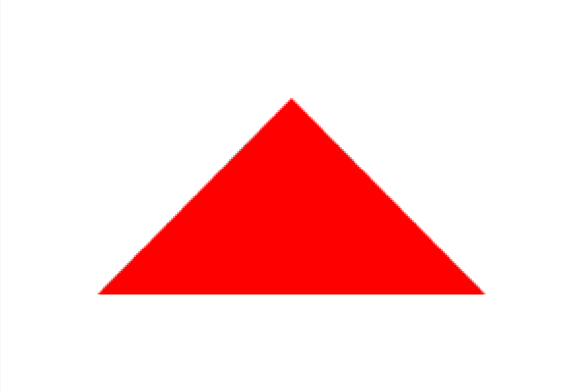
qml-cpp-integration/ex-simple-item

# Exporting a Scene Graph based GUI Class

- Derive from `QQuickItem`
- Implement `updatePaintNode(...)`
- Create and initialize a `QSGNode` subclass (e.g. `QSGGeometryNode`)
  - `QSGGeometry` to specify the mesh
  - `QSGMaterial` to specify the texture
- Similar to non GUI classes:
  - Export object from C++
  - Import and use in QML
  - Properties, signals/slots, `Q_INVOKABLE`

# Exporting a Scene Graph based GUI Class cont'd.

```
#include <QQuickItem>
#include <QSGGeometry>
#include <QSGFlatColorMaterial>

class TriangleItem : public QQuickItem {
    Q_OBJECT

public:
    TriangleItem(QQuickItem *parent = Q_NULLPTR);

protected:
    QSGNode *updatePaintNode(QSGNode *node, UpdatePaintNodeData *data);

private:
    QSGGeometry m_geometry;
    QSGFlatColorMaterial m_material;
};
```



# Exporting a Scene Graph based GUI Class cont'd.

```
#include "triangleitem.h"
#include <QSGGeometryNode>

TriangleItem::TriangleItem(QQuickItem *parent) :
    QQuickItem(parent),
    m_geometry(QSGGeometry::defaultAttributes_Point2D(), 3)
{
    setFlag(ItemHasContents); m_material.setColor(Qt::red);
}
```

# Exporting a Scene Graph based GUI Class cont'd.

```
QSGNode *TriangleItem::updatePaintNode(QSGNode *n, UpdatePaintNodeData *)
{
    QSGGeometryNode *node = static_cast<QSGGeometryNode *>(n);
    if (!node) { node = new QSGGeometryNode(); }
    QSGGeometry::Point2D *v = m_geometry.vertexDataAsPoint2D();
    const QRectF rect = boundingRect();
    v[0].x = rect.left();
    v[0].y = rect.bottom();
    v[1].x = rect.left() + rect.width()/2;
    v[1].y = rect.top();
    v[2].x = rect.right();
    v[2].y = rect.bottom();
    node->setGeometry(&m_geometry);
    node->setMaterial(&m_material);
    return node;
}
```

qml-cpp-integration/ex-simple-item-scenegraph

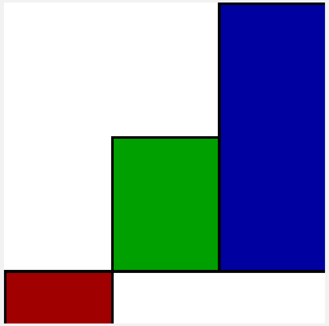
# Defining Custom Property Types

- Enums
- Custom types as property values

```
Timer {
    id: timer
    interval { duration: 2; unit: IntervalSettings.Seconds }
}
```

- Collection of custom types

```
Chart {
    anchors.fill: parent
    bars: [
        Bar { color: "#a00000" value: -20 },
        Bar { color: "#00a000" value: 50 },
        Bar { color: "#0000a0" value: 100 }
    ]
}
```



# Defining Custom Property Types

- Custom classes can be used as property types
  - Allows rich description of properties
  - Subclass `QObject` or `QQuickItem` (as before)
  - Requires registration of types (as before)

- A simpler way to define custom property types:
  - Use simple enums and flags
  - Easy to declare and use

- Collections of custom types:
  - Define a new custom item
  - Use with a `QQmlListProperty` template type

# Using 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};
```

```
Timer {
    id: timer
    interval {
        duration: 2;
        unit: IntervalSettings.Seconds
   }
}
```

# Custom Classes as Property Types

- Use the subtype as a pointer

```
class Timer : public QObject
{
    Q_OBJECT
    Q_PROPERTY(IntervalSettings* interval READ interval WRITE setInterval
                                          NOTIFY intervalChanged)
public:
    IntervalSettings *interval() const;
    void setInterval(IntervalSettings *);

private:
    QTimer *m_timer;
    IntervalSettings *m_settings;
}
```

# Custom Classes as Property Types cont'd.

- Instantiate `m_settings` to an instance rather than just a null pointer:

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

# Custom Classes as Property Types cont'd.

- Instantiating allow you this syntax:

```
Timer {
    id: timer
    interval {
        duration: 2
        unit: IntervalSettings.Seconds
    }
}
```

- Alternatively you would need this syntax:

```
Timer {
    id: timer
    interval: IntervalSettings {
        duration: 2
        unit: IntervalSettings.Seconds
    }
}
```

# Custom Classes as Property Types cont'd.

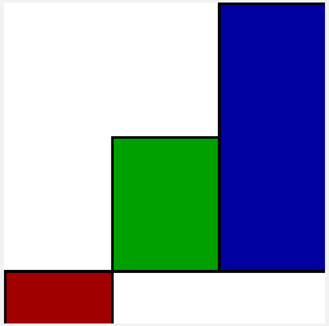
- Both classes must be exported to QML

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

qml-cpp-integration/ex-timer-custom-types

# Collections of Custom Types

```
Chart {
    anchors.fill: parent
    bars: [
        Bar { color: "#a00000" value: -20 },
        Bar { color: "#00a000" value: 50 },
        Bar { color: "#0000a0" value: 100 }
    ]
}
```



- A chart item
  - With a `bars` list property
  - Accepting custom `Bar` items

qml-cpp-integration/ex-custom-collection-types

# Declaring the List Property

- In the *chartitem.h* file:

```
class ChartItem : public QQuickPaintedItem
{
    Q_OBJECT
    Q_PROPERTY(QQmlListProperty<BarItem> bars READ bars NOTIFY barsChanged)

public:
    ChartItem(QQuickItem *parent = Q_NULLPTR);
    void paint(QPainter *painter) Q_DECL_OVERRIDE;
    QQmlListProperty<BarItem> bars();
    …
}
```

- Define the bars property
    - In theory, read-only but with a notification signal
    - In reality, writable as well as readable

# Declaring the List Property

- In the *chartitem.h* file:

```
    QQmlListProperty<BarItem> bars();
    …
Q_SIGNALS:
    void barsChanged();

private:
    static void append_bar(QQmlListProperty<BarItem> *list, BarItem *bar);
    QList<BarItem*> m_bars;
```

- Define the getter function and notification signal
- Define an append function for the list property

# Defining the Getter Function

- In the *chartitem.cpp* file:

```
QQmlListProperty<BarItem> ChartItem::bars()
{
    return QQmlListProperty<BarItem>(this, 0, &ChartItem::append_bar,
                                           0, 0, 0);
}
```

- Defines and returns a list of `BarItem` objects
  - With an append function
- Possible to define count, at and clear functions as well

# Defining the Append Function

```
void ChartItem::append_bar(QQmlListProperty<BarItem> *list, BarItem *bar)
{
    ChartItem *chart = qobject_cast<ChartItem *>(list->object);
    if (chart) {
        bar->setParent(chart);
        chart->m_bars.append(bar);
        chart->barsChanged();
    }
}
```

- Static function, accepts
  - The list to operate on
  - Each `BarItem` to append
- When a `BarItem` is appended
  - Emits the `barsChanged()` signal

# Summary of Custom Property Types

- Define classes as property types:
  - Declare and implement a new `QObject` or `QQuickItem` subclass
  - Declare properties to use a pointer to the new type
  - Register the item with `qmlRegisterType`
- Use enums as simple custom property types:
  - Use `Q_ENUMS` to declare a new enum type
  - Declare properties as usual
- Define collections of custom types:
  - Using a custom item that has been declared and registered
  - Declare properties with `QQmlListProperty`
  - Implement a getter and an append function for each property
  - read-only properties, but read-write containers
  - read-only containers define append functions that simply return

# Default Property

- One property can be marked as the default

```
class ChartItem : public QQuickPaintedItem {
    Q_OBJECT
    Q_PROPERTY(QQmlListProperty<BarItem> bars READ bars NOTIFY barsChanged)
    Q_CLASSINFO("DefaultProperty", "bars")
```

- Allows child-item like syntax for assignment

```
Chart {
    width: 120; height: 120
    Bar { color: "#a00000" value: -20 }
    Bar { color: "#00a000" value: 50 }
    Bar { color: "#0000a0" value: 100 }
}
```

qml-cpp-integration/ex-default-property

# Creating Extension Plugins

- Declarative extensions can be deployed as plugins
  - Using source and header files for a working custom type
  - Developed separately then deployed with an application
  - Write QML-only components then rewrite in C++
  - Use placeholders for C++ components until they are ready
- Plugins can be loaded by the `qmlscene` tool
  - With an appropriate `qmldir` file
- Plugins can be loaded by C++ applications
  - Some work is required to load and initialize them

qml-cpp-integration/ex-extension-plugin

# Defining an Extension Plugin

```
#include <QQmlExtensionPlugin>

class EllipsePlugin : public QQmlExtensionPlugin {
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "org.qt-project.Qt.QQmlExtensionInterface/1.0")

public:
    void registerTypes(const char *uri) Q_DECL_OVERRIDE;
};
```

- Create a `QQmlExtensionPlugin` subclass
  - Add type information for Qt's plugin system
  - Only one function to re-implement

# Implementing an Extension Plugin

```
#include “ellipseplugin.h”
#include “ellipseitem.h”

void EllipsePlugin::registerTypes(const char *uri)
{
    qmlRegisterType<EllipseItem>(uri, 9, 0, "Ellipse");
}
```

- Register the custom type using the `uri` supplied
  - The same custom type we started with

# Building an Extension Plugin

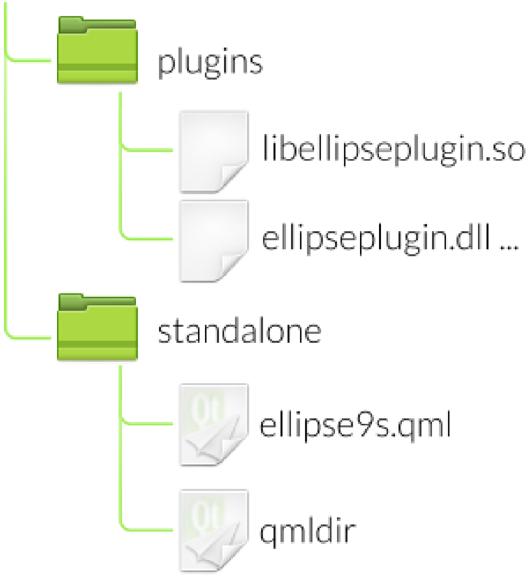
```
TEMPLATE = lib
CONFIG += qt plugin
QT += quick

HEADERS += ellipseitem.h ellipseplugin.h

SOURCES += ellipseitem.cpp ellipseplugin.cpp

DESTDIR = ../plugins
```

- Ensure that the project is built as a Qt plugin
- QtQuick module is added to the configuration
- Plugin is written to a `plugins` directory

# Using an Extension Plugin

To use the plugin with the `qmlscene` tool:

- Write a `qmldir` file
  - Include a line to describe the plugin
  - Stored in the `standalone` directory
- Write a QML file to show the item
  - File `ellipse9s.qml`
- The `qmldir` file contains a declaration
  - `plugin ellipseplugin ../plugins`
- Plugin followed by
  - The plugin name: `ellipseplugin`
  - The plugin path relative to the `qmldir` file: `../plugins`



# Using an Extension Plugin

- In the *ellipse9s.qml* file:

```
Item {
    anchors.fill: parent
    Ellipse {
        x: 50; y: 50
        width: 200;
        height: 100
    }
}
```

- Use the custom item directly
- No need to import any custom modules
  - Files `qmldir` and `ellipse9s.qml` are in the same project directory
  - QML type `Ellipse` is automatically imported into the global namespace

# Loading an Extension Plugin

To load the plugin in a C++ application:

- Locate the plugin
  - Perhaps scan the files in the `plugins` directory
- Load the plugin with `QPluginLoader`
  - `QPluginLoader loader(pluginsDir.absoluteFilePath(fileName));`
- Cast the plugin object to a `QQmlExtensionPlugin`

  -
    ```
    QQmlExtensionPlugin *plugin =
                      qobject_cast<QQmlExtensionPlugin *>(loader.instance());
    ```
- Register the extension with a URI

  -
    ```
    if (plugin)
              plugin->registerTypes("Shapes");
    ```
  - In this example, `Shapes` is used as a URI

# Using an Extension Plugin

- In the `ellipse9s.qml` file:

```
import Shapes 9.0

Item {
    Ellipse {
        x: 50; y: 50
        width: 200;
        height: 100
    }
}
```

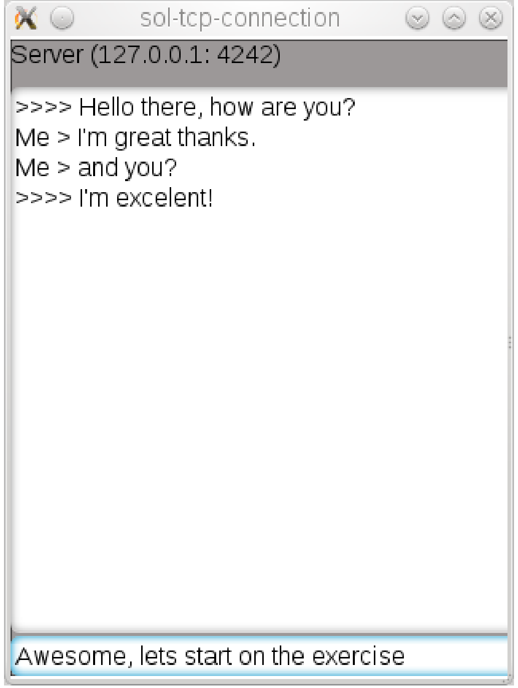
- The `Ellipse` item is part of the `Shapes` module
- A different URI makes a different import necessary; e.g.,
  - `plugin->registerTypes("com.theqtcompany.examples.Shapes");`
  - corresponds to `import com.theqtcompany.examples.Shapes 9.0`

# Summary of Extension Plugins

- Extensions can be compiled as plugins
  - Define and implement a `QQmlExtensionPlugin` subclass
  - Define the version of the plugin in the extension
  - Build a Qt plugin project within the quick option enabled
- Plugins can be loaded by the `qmlscene` tool
  - Write a `qmldir` file
  - Declare the plugin's name and location relative to the file
  - No need to import the plugin in QML
- Plugins can be loaded by C++ extensions
  - Use `QPluginLoader` to load the plugin
  - Register the custom types with a specific URI
  - Import the same URI and plugin version number in QML

# Lab – Chat Program

- The handout contains a partial solution for a small chat program
- One side of the chat will be a server (using `QTcpServer`) and the other end connect to it
- The TCP connection is already implemented in C++
- The GUI is implemented in QML
- Missing: The glue which makes the two parts work together
- STEPS aree available in the file `readme.txt`



Server (127.0.0.1: 4242)

```
>>>> Hello there, how are you?
Me > I'm great thanks.
Me > and you?
>>>> I'm excelent!
```

Awesome, lets start on the exercise

# Contents

- Canvas
- Particles
- Shaders

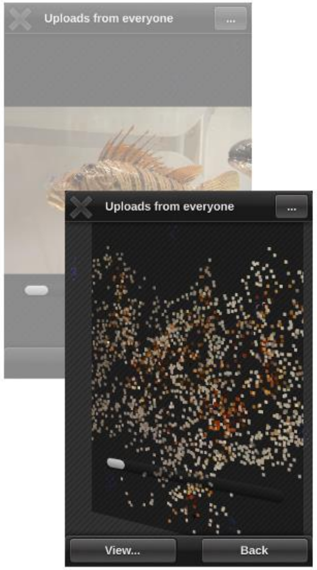
# Objectives

- Knowledge on how to create items with your own painting code
- Use canvas with user interaction of animations
- Create a complete particle system
  - Specify the particles
  - Provide velocity, acceleration or other physics traits
- Use shaders to modify items rendering
  - Fragment shaders for pixel manipulation
  - Vertex shaders for shape manipulation

# Why Use Canvas, Particles and Shaders?

- Custom painting of components
- Graphs and plots

- Complex visual effects
- Simulate some physics during animations
- Benefit as much as possible from the GPU



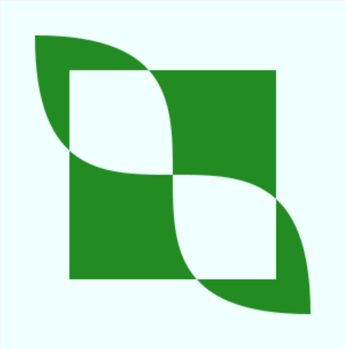
qml-graphics-effects/ex-canvas/piechart.qml

# Canvas

- QML type `Canvas` is used to insert an item in which to paint
- Handler `onPaint` will contain the painting code
- API somewhat similar to the old `QPainter` API
- API compatible with the HTML5 Canvas API
  - Need to request a `Context2D` instance first
- Method `requestPaint()` to schedule repainting

# Path Rendering

```
Canvas {
    anchors.fill: parent
    onPaint: {
        var context = getContext("2d");
        context.clearRect(0, 0, width, height);
        context.fillRule = Qt.OddEvenFill;
        context.fillStyle = "forestgreen";
        context.beginPath();
        context.moveTo(width * 0.1, height * 0.1);
        context.bezierCurveTo(width * 0.9, width * 0.1, width * 0.1,
                              height * 0.9, height * 0.9, height * 0.9);
        context.bezierCurveTo(width * 0.9, width * 0.1, width * 0.1,
                              height * 0.9, height * 0.1, height * 0.1);
        context.closePath();
        context.rect(width * 0.2, height * 0.2,
                     width * 0.6, height * 0.6);
        context.fill(); } }
```



qml-graphics-effects/ex-canvas/path-painting.qml

# Scribble Area

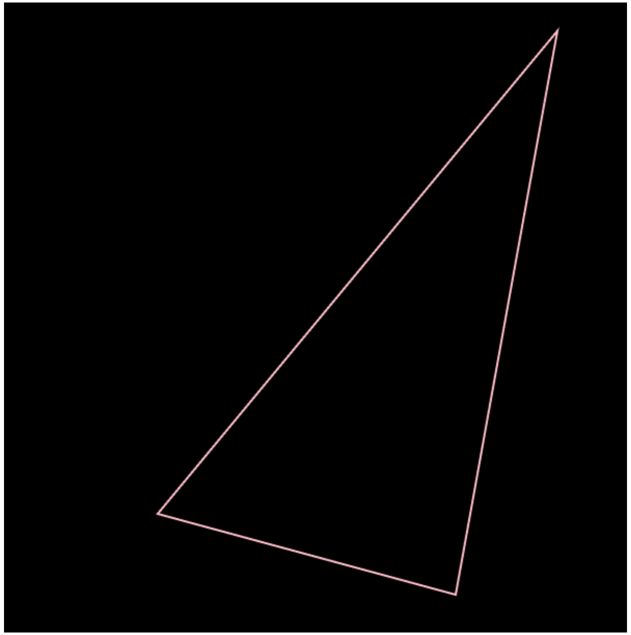
```
Rectangle {
    width: 600; height: 600
    color: "white"
    MouseArea {
        anchors.fill: parent
        onPressed: canvas.requestPaint();
    }
    Canvas {
        id: canvas
        anchors.fill: parent
        onPaint: {
            var context = getContext("2d");
        }
    }
}
```

qml-graphics-effects/ex-canvas/scribble-area.qml

# Lab: Screen Saver

Starting from the partial solution:

- Get the lines to be rendered
- Have the points forming the lines animated



qml-graphics-effects/lab-screensaver/screensaver.qml

# Particle System

- A `ParticleSystem` requires
  - At least a particles source
  - The description of how particles look

- Particles sources are `Emitter` instances
  - They emit the logical particles
  - Provide initial attributes: `emitRate`, `lifeSpan`, `size`, `speed`,...
  - The flow can be controlled using `enabled`, `pulse()` and `burst()`

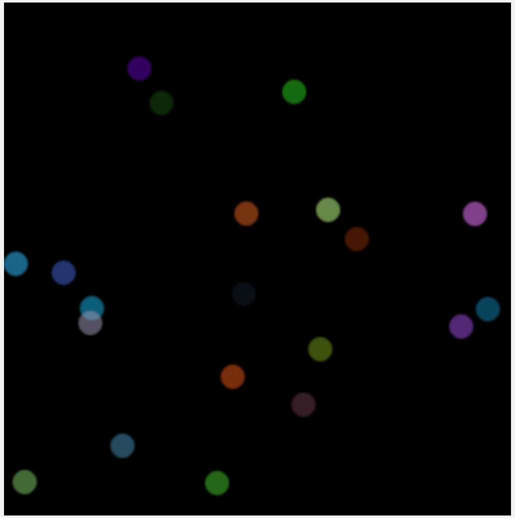
- Particle appearance is controlled by a `ParticlePainter` instance
  - `ImageParticle` uses an image as `source`, it can be rotated, colorized, etc.
  - `ItemParticle` uses an `Item` delegate to render particles
  - `CustomParticle` uses shaders to render particles

# Emitter

- Emit the particles using `Emitter` methods
  - Emit particles immediately using `burst(int count, int x, int y)`
    - Coordinates may be omitted, in which case the emitter emits particles randomly inside the emitter area
    - Use `shape` property (`EllipseShape`, `LineShape`, `MaskShape`) to provide a non-rectangular emitter area
  - Enable the disabled emitter for a `duration` milliseconds `pulse(int duration)`
- Provide the initial particle attributes
  - `emitRate(10)` – number of particles emitted per second
  - `lifeSpan(1000)` – life span of each particle in milliseconds. Values > 10 mins are treated as infinite
  - `size(16)` – particle size in pixels. May be linearly interpolated to `endSize`
  - `acceleration, velocity` – defined as stochastic `Direction`

# Christmas Lights

```
import QtQuick 2.5
import QtQuick.Particles 2.0

Rectangle { …
    Timer { onTriggered: emitter.burst(20) }
    ParticleSystem {
        anchors.fill: parent
        Emitter {
            id: emitter
            anchors.fill: parent
            enabled: false
            lifeSpan: 1000 size: 32
        }
        ImageParticle {
            source: "../images/particle.png"
            sizeTable: "../images/sizeTable.png"
            redVariation: 100 greenVariation: 100 blueVariation: 100 }
    }
}
```



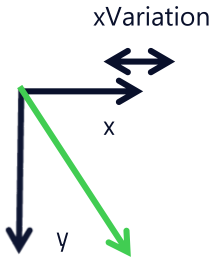
qml-graphics-effects/ex-particles/christmas-lights.qml

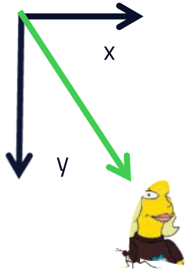
# Physics: Speed & Acceleration

- Both initial `velocity` and `acceleration` are specified using a `Direction`

- A `Direction` is a vector space of possible directions for a particle
  - Value intervals are specified using `*Variation` properties
  - Each particle gets a random vector of the vector space

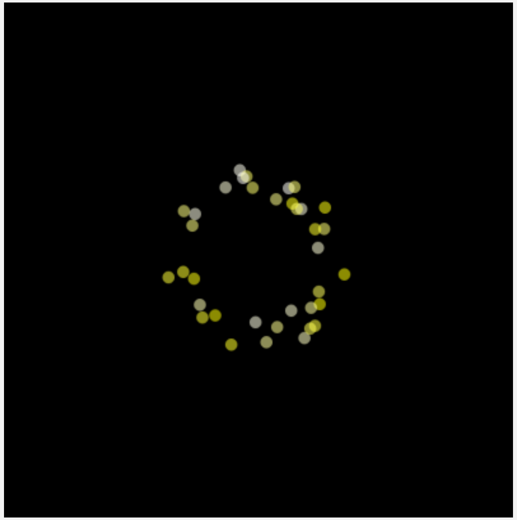
- `Direction` is never used, it has subclasses
  - `AngleDirection` for directions varying in `angle`
  - `PointDirection` for directions varying in `x` and `y` components
  - `TargetDirection` for directions toward a `targetItem`
  - `CumulativeDirection` acts as a direction that sums the directions within it





# Explosion

```
import QtQuick 2.5
import QtQuick.Particles 2.0

Rectangle { …
    Timer { onTriggered: emitter.pulse(50) }
    ParticleSystem {
        anchors.fill: parent
        Emitter {
            id: emitter
            anchors.centerIn: parent
            enabled: false
            emitRate: 700; lifeSpan: 500 size: 16
            velocity: AngleDirection {
                magnitude: 500; angleVariation: 360 } }
            ImageParticle {
                source: "../images/particle.png"
                sizeTable: "../images/sizeTable.png"
                blueVariation: 100 } } }
```



qml-graphics-effects/ex-particles/explosion.qml

# ParticlePainter – Particle Visualizer

- `ImageParticle` uses an image as `source` property. The image can be
  - colorized
  - rotated
  - deformed:: `xVector` (`Direction`), `yVector`
  - a sprite-based animation: `sprites` (a list of `Sprite` objects)
- `ItemParticle` uses an `Item delegate` to render particles
  - Grab the item from the logical item `give(Item)` and associate it back with the logical item `take(Item, bool)`
  - Control the item life time progressions yourself `freeze(Item)` or let the particle system control it `unfreeze(Item)`
- `CustomParticle` uses vertex and fragment shaders to render particles
  - Template code provided in the documentation

qml-graphics-effects/ex-particles/custom-particle.qml

# Physics: Force fields

- Affect particle attributes after the particle has been emitted
  - May affect in a rectangular area or in any shape, if `shape` property is defined
  - `Affector` provides useful properties, but use sub-types in your QML code
- May be disabled/enabled: `enabled`(true)
- May have arbitrary shape: `shape`
- Provides collision checking: `whenCollidingWith`: a list of particle groups
  - `Affector` affects a particle only, if the particle collides with another particle in one of the groups
- May be applied once (`once: true`) or any number of times

- Provides a signal, when a particle is affected
  - `affected(x, y)`

# Affector Sub-Types

- `Age { lifeLeft: 2000 }`
  - Defines the remaining life time

- `Attractor { affectedParameter: Attractor.Position/Velocity/Acceleration; proportionalToDistance: Attractor.Constant/Linear/InverseLinear/Quadratic/InverseQuadratic }`
  - Specifies a point of zero size, attracting particles

- `Friction { factor: 0.1; threshold: 0.0 }`
  - Slows down moving particles by a factor of their velocity, until the threshold has been achieved

- `Gravity { angle: -90; magnitude: 23.5 }`
  - Accelerates particles to a vector of the specified magnitude in the specified angle

# Affector Sub-Types

- `Turbulence { noiseSource: “qrc:/smoothBlackAndWhite.png”; strength: 100 }`
  - Applies a chaos map of force vectors to the particles
  - A default image exists
  - The magnitude of the velocity vector varies in a range [0, √2]

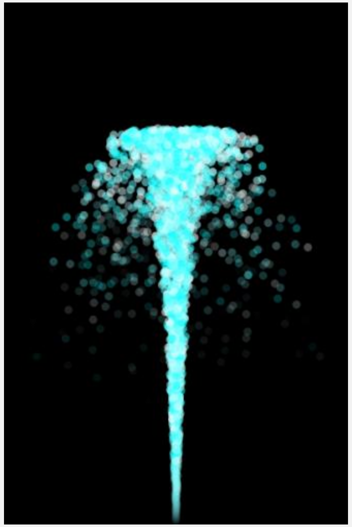
- `Wander { pace: 7.2; affectedParameter: PointAttractor.Position/Velocity/Acceleration; xVariance: 99.9 }`
  - Applies random particle trajectory

- `GroupGoal { goalState: “Group_72”;  jump: false }`(`SpriteGoal`)
  - Changes the state of a group of a particle
  - Groups may have defined durations and transitions between groups
  - Setting `goalState` will cause head down the path which will reach the state quickest
  - Setting `jump: true` will cause the goal state to be reached immediately without finishing the current state and using any transition path

qml-graphics-effects/ex-particles/explosion-implosion.qml

# Fountain

```
ParticleSystem {
    anchors.fill: parent
    Emitter {
        id: emitter
        anchors.centerIn: parent
        enabled: false
        emitRate: 700; lifeSpan: 500; size: 16
        velocity: AngleDirection {
            magnitude: 500; angleVariation: 360 } }
    ImageParticle {
        source: "../images/particle.png"
        sizeTable: "../images/sizeTable.png"
        blueVariation: 100 }
    Gravity {
        magnitude: 200”
    } }
```



qml-graphics-effects/ex-particles/fountain.qml

# Particle Groups

- Can be used for simple grouping
  - Emitter in “group1” uses particles in “group1” and affectors in “group1”
  - Particle group defined implicitly using the `group` property
  - `Emitter { group: “rocket”; emitRate: 100 }`
- Use `ParticleGroup` explicitly, if you wish to define timed transitions between the groups
  - After five seconds an emitted rocket explodes and generates two other groups: smoke and pieces of the rocket
- Define the weighted transitions using `to` property
  - `to: { “group1”: 22, “group2”:15, “group3”: 63 }`
- Use e.g. `GroupGoal` affector to define, when the transition occurs
- Use `TrailEmitter(s)` to emit particles in another group (particle positions based on previous particle positions)

# Lab: Make It Snow!

Starting from the partial solution:

- Get the snow flakes to slowly fall
- Make sure they're not all going exactly in the same direction/speed
- Optionally: Get the snow flakes to rotate as they fall

qml-graphics-effects/lab-makeitsnow/makeitsnow.qml

# Shaders

- A shader is a program used to calculate rendering effects on the GPU
- Two types of shader are available in QtQuick
  - Fragment shaders
    - Operate on each pixel
    - Cannot be complex as it has no knowledge of the scene geometry
    - Used for color manipulation, bump mapping, shadows, etc.
  - Vertex shaders
    - Operate on each vertex
    - Can change position, color and texture coordinate
    - Cannot create new vertices
- Their execution is heavily parallelized in the GPU pipeline
- Extremely efficient
- Written using OpenGL Shading Language (GLSL)

# Fragment Shaders

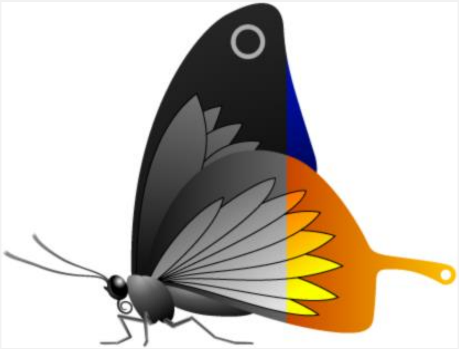
- QML type `ShaderEffect` is a rectangle displaying the result of a shader program
- The `fragmentShader` property is a string with the fragment shader code
- Often such shaders use textures as inputs

- QML type `ShaderEffectSource` allows to render an item as a texture
- Property `sourceItem` holds the Item to be rendered

- Note that `ShaderEffectSource` is an invisible item aimed at consumption in `ShaderEffect` instances

# Saturation Filter

```
ShaderEffectSource { id: effectSource
    sourceItem: Image { id: butterfly … }
    ShaderEffect {
        width: butterfly.width
        height: butterfly.height
        property variant source: effectSource
        property real filterPosition: 0.0
        SequentialAnimation on filterPosition { … }
        fragmentShader: "
            uniform sampler2D source;
            uniform float filterPosition;
            varying highp vec2 qt_TexCoord0;
            void main() {
                highp vec4 color = texture2D(source, qt_TexCoord0);
                if (qt_TexCoord0.s < filterPosition) {
                    gl_FragColor = vec4( … );
                } else {
                    gl_FragColor = color; } }" } }
```



qml-graphics-effects/ex-shaders/saturation-filter.qml

# Vertex Shaders

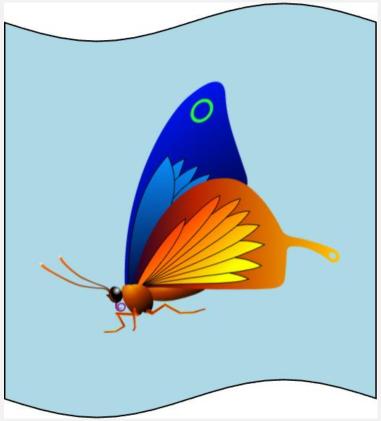
- Works similarly to fragment shaders

- Use the `vertexShader` property for the vertex shader code

- Pay attention to the `mesh` property
  - Specifies the number of vertices of the `ShaderEffect` QML type
  - It must be fine enough to resolve the transformation

# Saturation Filter

```
ShaderEffectSource { id: effectSource; … }
ShaderEffect { …
    property variant source: effectSource
    property real pi: Math.PI
    property real offset: 0
    NumberAnimation on offset { … }
    mesh: Qt.size(20, 20)
    vertexShader: "
        uniform highp float offset;
        uniform highp mat4 qt_Matrix;
        attribute highp vec4 qt_Vertex;
        attribute highp vec2 qt_MultiTexCoord0;
        varying highp vec2 qt_TexCoord0;
        void main() {
            qt_TexCoord0 = qt_MultiTexCoord0;
            highp vec4 pos = qt_Vertex;
            pos.y = …
            gl_Position = qt_Matrix * pos; }" } }
```



qml-graphics-effects/ex-shaders/flag.qml

# Chaining Shaders

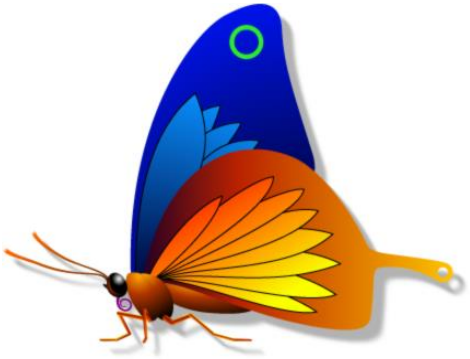
- QML type `ShaderEffectSource` can have any `Item` as `sourceItem`
- Even a `ShaderEffect`
- Allows to create complex effects by chaining shader programs

# Drop Shadow

A drop shadow is a combination of:

- A blur operation
- A darkening of the result of the blur
- A composition of the original on top of the created shadow with an offset

- Drop shadow can be applied with Qt graphical effects as well



qml-graphics-effects/ex-shaders/drop-shadow.qml + drop-shadow2.qml

# Shaders and Particles

- Everything about the particle systems is still valid
- Allows for less CPU intensive particle rendering
- Use `CustomParticle` instead of `ImageParticle`
- Use the `vertexShader` and `fragmentShader` properties

qml-graphics-effects/ex-shaders/fountain-shaders.qml

# QtGraphicalEffects

- QML module, providing more than 20 reusable types with ready-made shaders
- Can be applied to any item, can be combined
- Blend
  - Item composition (blending with several modes)
- Color
  - `BrightnessContrast`, `Colorize`, `GammaAdjust`, …
- Gradient
  - `LinearGradient`, `RadialGradient`, `ConicalGradient`
- Distortion
  - `Displace`
- Drop Shadow

# QtGraphicalEffects

- Blur
  - `FastBlur`, `MaskedBlur`, `GaussianBlur`, …
- Motion Blur
  - `DirectionalBlur`, `RadialBlur`, `ZoomBlur`
- Glow
  - `Glow`, `RectangularGlow`
- Mask
  - `OpacityMask`, `ThresholdMask`

qml-graphics-effects/ex-ex-effects/

# QtGraphicalEffects

- Blur

```
Image {
    id: blurSrc
    source: "clarice.gif"
    width: parent.width
    height: parent.height
    smooth: true
    visible: false
}
FastBlur {
    anchors.fill: blurSrc
    source: blurSrc
    radius: 8
}
```

- Opacity mask

```
Image {
    id: mask
    // Exactly similar to blurSrc }
Image {
    id: maskSrc
    source: "butterfly.png"
    width: parent.width
    height: parent.height
    smooth: true
    visible: false }
OpacityMask {
    anchors.fill: mask
    source: mask
    maskSource: maskSrc
}
```

qml-graphics-effects/ex-ex-effects/

# Lab: Dissolve Effect

Starting from the partial solution:

- Create an alpha gradient effect
- Animate it so the item fades out from top to bottom and back in again

qml-graphics-effects/lab-dissolve/dissolve.qml

# Thank You!

www.qt.io