Qml QtQuick Essentials Training

3-day session

Overview	Qt Quick scene graph concept Understanding Qml Component/Document/Module Architecute Understanding QtQuick classes Exporting C++ class to Qml as object and class type Three layers software development with qml and c++ Dynamic Properties
Duration	Three days - 24 hours 50% of lecture, 50% of practical labs.
Trainer	http://www.linkedin.com/in/pravinkumarsinha
Prerequisite	**Knowledge of Qt programming** Knowledge of Qt programming is important. Technically Qml is javascript counter part of C++ Qt. Knowing Qt will help fast understanding of Qml. Qt slides can browsed at http://www.minhinc.com/training/qt/advance-qt-slides.php Pdf document can be downloaded from http://www.minhinc.com/training/advance-qt-slides.pdf **Knowledge of GUI and other concepts** Qt/Qml is used for GUI development and many other technologies including Networks, operating systems, Database, scripting. Basic knowledge of these domains are required as per the Qt/Qml is developed for the particular domain.
Setup	Ubuntu LTS 16.04, Qt 5.[>6]

Lecture

Lecture session will be course content presentation through the trainer.
Any source code example related to the topic will be demonstrated, it would include executing the binaries. Lecture content can be downloaded at http://www.minhinc.com/training/advance-qml-slides.pdf

Labs

Labs session would be completely hands on session where each example (with example data and execution instruction) would be provided to the students. Students can verify their results with the results provided in the material.

Day 1 Morning

Lecture - Introduction to Qt Quick and QML

- QObject Meta Object System
- Defining Qml Component
- Tree of QML Objects
- Qml Types
  - Visual
  - Non Visual
- Qt Quick classes

Lecture - Qt Properties

- Combination of Get/Set/Notify
- Declaration of a Qt Property
- Qt Property with Enum
- Getting & Setting Qt Properties
- Dynamic Properties
- Signal and Slots

Lecture - Building Blocks of QML

- QuickItem
- Variant
  - QVariant and QML
- Variant Containers
  - QVariantList
  - QList
  - QqmlListProperty

Day 1 Afternoon

Lecture - Composing UIs	Lab
• Nested Items • Graphical QML Types • Text Type • Anchor Layout	• Write a QML document • Import QML document in another QML document as a package • Use anchor layout to place current window with imported window in some layout fashion.

Day 2 Morning

Lecture - User Input	Lecture - Components and Structures
• Keyboard Navigation and Focus • Input Focus - Focus Order - Focus property - Active Focus - FocusScope • Handling Keyboard Input - Key-Specific Handlers - All Keys Handler - Key Event Propagation - Event Propagation is Convenient • Multi-Touch - Common Multi-Touch Gestures - Handling Overlapping Touch Areas • Mouse/Single-Touch - Tap - Double-Tap - Tap and Hold	• Components - QQmlComponent • Dividing code into Components - Creating new Items through new .qml file - Creating Component dynamically • Modules • qmldir file - directory import - Module import

Lecture - Qt Quick Controls
• Qt Quick Designer • Qt Quick Controls • Application Window • Contorls and Views • Layouts • Styling

Day 2 Afternoon

Lecture - State and Transitions	Lab
• States • State Conditions • Transitions	• Emit signal when mouse moves over an area • Write qml component as a separate .qml file • Place Anchor layout inside Vertical layout

Day 3 Morning

Lecture - Dynamic Creation of Items	Lecture - C++ Integration
• Creating Items Dynamically - Procedural Method - Declarative Method • Procedural Creation • Procedural/Declarative Creation • Declarative Creation • Creating Multiple Items • Repeaters	• Declartive Envrionment -QQmlApplicationEngine -QQuickView • Exporting C++ Objects to QML -QQmlContext • Exporting Classes to QML • Exporting Non-GUI Classes • Exporting Qpainter based GUI Classes -QQuickPaintedItem • Exporting Scene Graph based GUI Classes • Using Custom Types Plugins • Building an application as a Library

Lecture - Graphical Effects
• Canvas • Particles • Shaders

Day 3 Afternoon

Lecture - Animations	Lecture - Presenting Data
• Animations • Easing Curves • Animation Groups	• Arranging Items • Data Models • Using Views • Using Delegate • XML Models

Lab

- Define component dynamically
- Implement Ecllipse as Qml Class type
- Place the Ecllipse class type as Library
- Implement List data model view through delegate

Qml Essentials

Qml Essenstials- Training Course

Minh, Inc.

DISCLAIMER

Text of this document is written in Bembo Std Otf(13 pt) font.

Code parts are written in Consolas (10 pts) font.

This training material is provided through **Minh, Inc.**, B'lore, India
Pdf version of this document is available at http://www.minhinc.com/training/advance-qml-slides.pdf
For suggestion(s) or complaint(s) write to us at sales@minhinc.com

Document modified on Sep-30-2019

Document contains 73 pages.

Day 1 Morning

1. Introduction to Qt Quick and QML

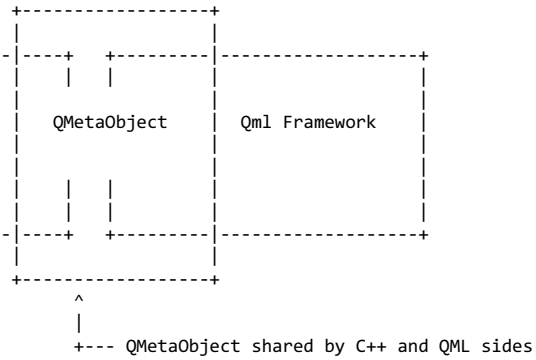
- QObject Meta Object System
  - Defining Qml Component
  - Tree of QML Objects
  - Qml Types
    - Visual
    - Non Visual
  - Qt Quick classes
  - Data Type Conversion Between QML and C++

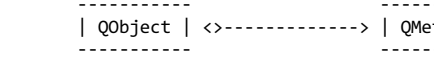
QObject has two parts. One is QObject declared in class and another is meta object associated to the QObject. Meta Object is referred by QMetaObject. Methods in meta object is referred by QMetaProperty.

```
QMetaObject *mo=::QObject->metaObject();
```

Meta object property can be executed through QObject as

```
::QObject->setProperty("abc",20);
```






```
const QMetaObject* metaObject=obj->MetaObject();
QStringList properites;
for(int i = metaObject->propertyOffset();i<metaObject->propertyCount();++i)
properties<<QString::fromLatin1(metaObject->property(i).name());
```

A Property can be added to meta system through

```
Q_PROPERTY(type name
          (READ getFunction [WRITE setFunction] |
           MEMBER memberName [(READ getFunction | WRITE setFunction)])
          [RESET resetFunction]
          [NOTIFY notifySignal]
          [REVISION int]
          [DESIGNABLE bool]
          [SCRIPTABLE bool]
          [STORED bool]
          [USER bool]
          [CONSTANT]
          [FINAL])
class Person : public QObject
{
 Q_OBJECT
 Q_PROPERTY(QString name READ name WRITE setName)
 Q_PROPERTY(int shoeSize READ shoeSize WRITE setShoeSize)
 .
 .
```

A methods can be added to meta system by

```
Q_INVOKABLE
class Window:public QWidget{
Q_OBJECT
public:
 Window();
 void normalMethod();
 Q_INVOKABLE void invokableMethod();
};
```

Methods in Qml can be called from c++ side

Qml

```
// MyItem.qml
Item {
 function readValues(anArray, anObject) {
  for (var i=0; i<anArray.length; i++)
  console.log("Array item:", anArray[i])
  for (var prop in anObject) {
  console.log("Object item:", prop, "=", anObject[prop])
  }
 }
}
```

C++

```
QQuickView view(QUrl::fromLocalFile("MyItem.qml"));
QVariantList list;
list << 10 << QColor(Qt::green) << "bottles";
QVariantMap map;
map.insert("language", "QML");
map.insert("released", QDate(2010, 9, 21));
QMetaObject::invokeMethod(view.rootObject(), "readValues",
 Q_ARG(QVariant, QVariant::fromValue(list)),
 Q_ARG(QVariant, QVariant::fromValue(map)));
```

For function returning object types

Qml

```
// MyItem.qml
Item {
 function readDate(dt) {
 console.log("The given date is:", dt.toUTCString());
 return new Date();
 }
}
```

C++

```
QQuickView view(QUrl::fromLocalFile("MyItem.qml"));
QDateTime dateTime = QDateTime::currentDateTime();
QDateTime retValue;
QMetaObject::invokeMethod(view.rootObject(), "readDate",
 Q_RETURN_ARG(QVariant, retValue),
 Q_ARG(QVariant, QVariant::fromValue(dateTime)));
qDebug() << "Value returned from readDate():" << retValue;
```

Day 1 Morning

1. Introduction to Qt Quick and QML

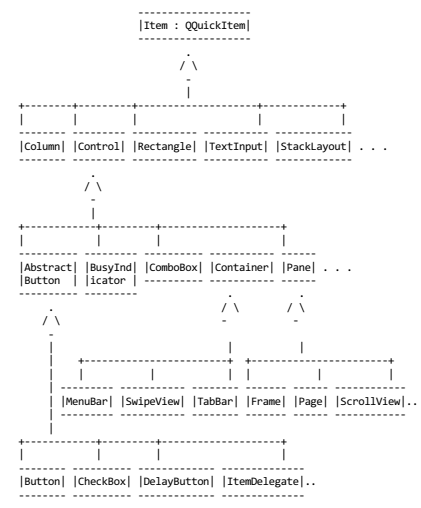
- QObject Meta Object System
- Defining Qml Component
- Tree of QML Objects
- Qml Types
  - Visual
  - Non Visual
- Qt Quick classes
- Data Type Conversion Between QML and C++

A .qml file is a component which is similar to a c++ header file having single class declration. A Qml component can exist as a .qml file/document or can be embedded in a Qml document as a 'Component' type.

Qml file contains
a) A import statement (not exatly same as c/c++ incude directive)
b) A single root obect delcraton (Item or its deriviative)

- import statement is to provides modules or type namespaces or javascript to enable the engine to load the QML Object types referenced within the doucment. Its not copy paste like #include in c/C++
- Qml object describes the hierarchy of single object which can be inherited, extended and instantiated. So it can have single root object hierarchy only. Once qml file in this fasion becomes a library object With new set of properties that can be resused in other component.

So one qml file creates one component or one new type that is resusable.
Base class of all qml type is "Item"
Item instantiates C++ class QquickItem



// MyQmlFile.qml

```
import QtQuick 2.0
Rectangle { width: 200; height: 200; color: "red" }
Rectangle { width: 200; height: 200; color: "blue" } // invalid two root object
```

Creating new Button type

// Button.qml

```
import QtQuick 2.0
Rectangle {
 width: 100; height: 100
 color: "red"
 MouseArea {
  anchors.fill: parent
```

```
  onClicked: console.log("Button clicked!")
 }
}
```

The Button type can then be used in an application:

```
// application.qml
import QtQuick 2.0
Column {
 Button { width: 50; height: 50 }
 Button { x: 50; width: 100; height: 50; color: "blue" }
 Button { width: 50; height: 50; radius: 8 }
}
```

```
         -------------------
         |Item : QQuickItem|
         -------------------
                .
               / \
                -
                |
           -----------
           |Rectangle|
           -----------
                .
               / \
                -
                |
           -----------
           | : Button| // Button inherits Rectangle in Button.qml and
           ----------- // gets instantiated in application.qml
```

A Component can exist in a qml document like

a)

```
import QtQuick 2.0
Item {
 width: 100; height: 100
 Component {
  id: redSquare
  Rectangle {
   color: "red"
   width: 10
   height: 10
  }
 }
 Loader { sourceComponent: redSquare }
 Loader { sourceComponent: redSquare; x: 20 }
```

where id is extra provided than file component

b)

```
var component = Qt.createComponent("Button.qml");
 if (component.status == Component.Ready)
  component.createObject(parent, {"x": 100, "y": 100});
```

c)

```
Item {
 id: root
 width: 500; height: 500
 Component {
  id: myComponent
  Rectangle { width: 100; height: 100; color: "red" }
 }
 Component.onCompleted: {
  myComponent.createObject(root)
  myComponent.createObject(root, {"x": 200})
 }
}
```

Day 1 Morning

1. Introduction to Qt Quick and QML

- QObject Meta Object System
- Defining Qml Component
- Tree of QML Objects
- Qml Types
  - Visual
  - Non Visual
- Qt Quick classes
- Data Type Conversion Between QML and C++

.qml file contains root Qml object (QQuickItem) where other sub objects aligned in parent child relationship making tree of objects. child objects are assigned to some property value of the parent object. Subobjects not assigned to parent's any property goes to parent default property.

for ex. xyz.qml

```
<Xyz.qml>
       Rectangle <>----+
                       |
```

```
                    |
                    |
                    |--> Rectangle
                    |--> TextArea
                    |--> TextInput
                    |--> QTimer
                    |--> Text
```

In this case Xyz is a new type that derived from Rectangle.

```
        Rectangle
          .
         / \
         ---
          |
         Xyz
```

Day 1 Morning

1. Introduction to Qt Quick and QML

- QObject Meta Object System
- Defining Qml Component
- Tree of QML Objects
- Qml Types
  - Visual
  - Non Visual
- Qt Quick classes
- Data Type Conversion Between QML and C++

Qml has mainly two types

Basic types-`bool, double, enumeration, int ,list, real, string, url, var`

Object Types

Non Gui Types-`QtObject, Connection, Component, Timer`

QtQuick Gui Types-`Item, Rectangle, TextArea, Text, TextInput`

Basic types

Basic types does not need any module to be imported by 'import' keyword.

Basic type provided through QtQuick module

date point, rect and size

'Qt' which is global variable provides subroutines to manipulate the basic types

For basic type, property change signal handler is invalid whereas signal handler on basic type itself is valid

```
Text {
// invalid!
 onFont.pixelSizeChanged: doSomething()
// also invalid!
 font {
  onPixelSizeChanged: doSomething()
 }
 onFontChanged: console.log("font changed")

 id: otherText
 focus: true
 Keys.onDigit1Pressed: font.pixelSize += 1
 Keys.onDigit2Pressed: font.b = !font.b
 Keys.onDigit3Pressed: font = otherText.font
}
```

javascript objects and arrays are also supported

with keyword `var`. `Date` and `Array` functors are available in Qml

```
import QtQuick 2.0
Item {
 property var theArray: new Array()
 property var theDate: new Date()
 Component.onCompleted: {
  for (var i = 0; i < 10; i++)
   theArray.push("Item " + i)
  console.log("There are", theArray.length, "items in the array")
  console.log("The time is", theDate.toUTCString())
 }
}
```

Types in Qml comes from three sources

a)Provided natively by qml compiler

b)Registered via c++ by QML modules

c)Provided as QML documents by QML module

Object type

All object types are derived from `QtQObject`

```
      QObject
        .
       / \
        -
```

```
      |
Qml Object Type
```

custom types can be made trough qml document types

Non gui Object types comes from QtQml module
`Component Date, Number, String, Component, Qt,QtObject, Locale, Binding, Connections, Instantiator, Timer`
List, Model Qml object types from QtQml.Model module
`DelegateModel, DelegateModelGroup, ListElement, ListModel, ObjectModel`

Whereas GUI object types come from QtQuick module

Day 1 Morning

1. Introduction to Qt Quick and QML

- QObject Meta Object System
- Defining Qml Component
- Tree of QML Objects
- Qml Types
  - Visual
  - Non Visual
- Qt Quick classes
- Data Type Conversion Between QML and C++

Qt Quick QML Types provided through QtQuick import
-`XML List Model`
-`Local Storage` - submodule containing javaScript interface for an SQLite database
-`Particles` - provides a particle system for Qt Quick
-`Layouts` - provides layout for arranging Qt Quick items
-`Window` - top-level windows and accessing screen information
-`Dialogs` - creating and interacting with system dialogs
-`Tests` - unit test for a QML application
-`Controls` - set of reusable UI componets

Day 1 Morning

1. Introduction to Qt Quick and QML

- QObject Meta Object System
- Defining Qml Component
- Tree of QML Objects
- Qml Types
  - Visual
  - Non Visual
- Qt Quick classes
- Data Type Conversion Between QML and C++

Qt Type	QML Basic Type
bool	bool
unsigned int, int	int
double	double
float, qreal	real
QString	string
QUrl	url
QColor	color
QFont	font
QDate	date
QPoint, QPointF	point
QSize, QSizeF	size
QRect, QRectF	rect
QMatrix4x4	matrix4x4
QQuaternion	quaternion
QVector2D, QVector3D, QVector4D	vector2d, vector3d, vector4d
Enums declared with Q_ENUM()/Q_ENUMS()	enumeration

2. Qt Properties

- Combination of Get/Set/Notifiy
  - Declaration of a Qt property
  - Qt Property with Enum
  - Getting & Setting Qt Properties
  - Dynamic Properties
  - Signal and Slots

Get, Set and Notify are getter, setter and setter notification signal function addition to Meta Object System. A new property is introducted in Meta Object System which is visible on QML side. Same thing can be achieved on Qml side when a new property and assiciated functions gets added to Meta Object System. As per Qml design those property as visible on C++ side also and signal handler to signal added in Qml side can be introduced in C++ side.

```
class Message : public QObject
{
 Q_OBJECT
 Q_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)
public:
 void setAuthor(const QString &a) {
  if (a != m_author) {
   m_author = a;
   emit authorChanged();
  }
 }
 QString author() const {
  return m_author;
 }
 signals:
  void authorChanged();
 private:
  QString m_author;
};
```

Here whenever author is modified in QML side, authorChanged signal is emitted. Qml engine attache a empty signal handler for it, named 'onAuthorChanged' in template on<propertyname>Changed.

This can be added to qml meta object system as object type or as instantiated object.

Instantiated object

```
int main(int argc, char *argv[]) {
 QGuiApplication app(argc, argv);
 QQuickView view;
 Message msg;
 view.engine()->rootContext()->setContextProperty("msg", &msg);
 view.setSource(QUrl::fromLocalFile("MyItem.qml"));
 view.show();
 return app.exec();
}
```

// MyItem.qml

```
import QtQuick 2.0
Text {
 width: 100; height: 100
 text: msg.author    // invokes Message::author() to get this value
 Component.onCompleted: {
  msg.author = "Jonah"  // invokes Message::setAuthor()
}
}
```

Registrered type

```
main(.....){
...
qmlRegisterType<Message>("Messaging", 1, 1, "Message")
...
}
```

// MyItem.qml

```
import QtQuick 2.0
import Messaging 1.1
Text {
 width: 100; height: 100
 Component.onCompleted: {
  text: Message{
   author="Jonah"
  }
 }
}
```

Day 1 Morning

2. Qt Properties

- Combination of Get/Set/Notify
- Declaration of a Qt property
- Qt Property with Enum
- Getting & Setting Qt Properties
- Dynamic Properties
- Signal and Slots

An object can have following attributes
a) The id attribute
b) property attributes
c) signal attributes
d) signal handler attributes
e) method attributes
f) attached properties and attached signal handler attributes

a) The id attrubute
The id attribute is unique for each instance of the class and it must start with lower case. Id is "component scoed". Id can be used to idenify the object as
if it is object name.

```
import QtQuick 2.0
 Column {
  width: 200; height: 200
  TextInput { id: myTextInput; text: "Hello World" }
  Text { text: myTextInput.text }
 }
```

b) property attributes
Qml has baisc types and object types. Object types can have property value that is static in value or bound to dynamic expression.

1) Defining property
A proprty can be inserted in QML object type through adding Q_PROPERY on C++ couter part class.
2) A property can be inserted in QML object through adding

```
'[default] property <propertyType> <propertyName>'
```

ex.

```
 Item {
//Basic types
  property int someNumber
  property string someString
  property url someUrl
//Var is takes any kind
  property var someNumber: 1.5
  property var someString: "abc"
  property var someBool: true
  property var someList: [1, 2, "three", "four"]
  property var someObject: Rectangle { width: 100; height: 100; color: "red" }
//Object type
  property Item someItem
  property Rectangle someRectangle
  property color nextColor: "blue" // declaration and initialization
//Custom type
  Property BasicButton bbtn // BasicButton.qml exists
 }
```

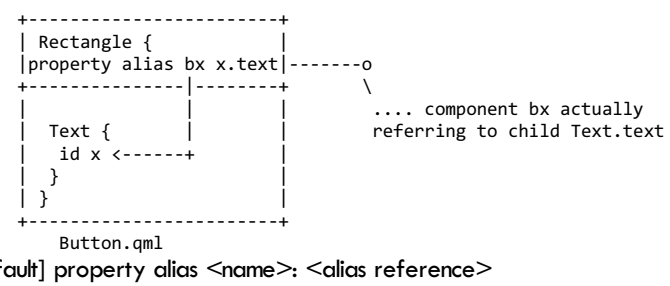
Property binding does not work on staticvalue returned through Javascript statement

```
import QtQuick 2.0
Rectangle {
 width: 100
 height: width * 2
 focus: true
 Keys.onSpacePressed: {
  height = width * 3 // once come here height is frozen
 }
}
```

Qt.binding() should be used instead

```
import QtQuick 2.0
Item {
width:500;height:500
Rectangle{
 id:rect
 width:100;  height:width*2
 focus:true
 Keys.onSpacePressed:{
  rect.height=Qt.binding(function() {return this.width*3})//'this' is component
                                                          // not rect
 }                                                 // this must be in Qt binding
```

```
}
```
3) property alias
Property alias is used to create alias to other property similar to reference in C++. It is mostly used in component property making alias to child items property. While creating object type only component level properties are visible outside.



[default] property alias <name>: <alias reference>

// Button.qml
```
import QtQuick 2.0
Rectangle {
property alias buttonText: textItem.text
width: 100; height: 30; color: "yellow"
Text { id: textItem }
}
```

Now Button.qml can be used in other qml document as
```
Button { buttonText: "Click Me" }
```

4) List property
List of valus can be declared as
```
[default] property list<<objectType>> propertyName: <value>
<<MyRectangle.qml>>
import QtQuick 2.0
Rectangle {
// declaration without initialization
 property list<Rectangle> siblingRects
// declaration with initialization
property list<Rectangle> childRects: [Rectangle { color: "red" },Rectangle { color: "blue"} ]
}
```

5) Grouped property
class member of object type can be field inititalized in curly braces
```
Text {
//dot notation
 font.pixelSize: 12;; font.b: true
}
```

font object properites can be grouped in curly braces
```
Text {
//group notation
font { pixelSize: 12; b: true }
}
```

4) Default property
Default property come into the picture when a object type is getting instantiated in a parent object type without gettig assigned to any of the parent properties. In this fashon parent default property holds the children.

c) Signal property
Signal is function that gets rasied by hardware or just by calling the function. Like unix signal, qml signal has handler associated to it.
signal can be declared as
```
  signal <signalName>[([<type> <parameter name>[, -]])]
```
<MySignal.qml>
```
import QtQuick 2.0
Item {
 signal clicked // with no argument, braces are not required
 signal hovered()
 signal actionPerformed(string action, var actionResult)
}
```

d)signal handler property
Qml also supports inbuilt handler for properites in the format
"on<Property>Changed"
<<MyTextInput.qml>>
```
import QtQuick 2.0
TextInput {
 text: "Change this!"
```

```
 onTextChanged: console.log("Text has changed to:", text)
}
```

Signal handler must be in nomenculture on<SignalName> or any method can also be nominated as signal handler.

<SquareButton.qml>

```
Rectangle {
id: root
signal activated(real xPosition, real yPosition)
signal deactivated
property int side: 100
idth: side; height: side
MouseArea {
anchors.fill: parent
onPressed: root.activated(mouse.x, mouse.y)//handler has not arguments declaration explictly
onReleased: root.deactivated()
}
}
```

class method can also be used as signal hander

```
Rectangle {
id: relay
signal messageReceived(string person, string notice)
Component.onCompleted: {
relay.messageReceived.connect(sendToPost)
relay.messageReceived.connect(sendToTelegraph)
relay.messageReceived.connect(sendToEmail)
relay.messageReceived("Tom", "Happy Birthday")
}
function sendToPost(person, notice) {
console.log("Sending to post: " + person + ", " + notice)
}
function sendToTelegraph(person, notice) {
console.log("Sending to telegraph: " + person + ", " + notice)
}
function sendToEmail(person, notice) {
console.log("Sending to email: " + person + ", " + notice)
}
}
```

e) Mehtod property

In c++ counterpart method can be declared through Q_PROPERTY() and Q_SLOT() for slots. In qml it through

```
function <functionName>([<parameterName>[, ...]]) { <body> }
```

<<MyText.qml>>

```
import QtQuick 2.0
Item {
width: 200; height: 200
MouseArea {
anchors.fill: parent
onClicked: label.moveTo(mouse.x, mouse.y)
}
Text {
id: label
function moveTo(newX, newY) {
label.x = newX;
label.y = newY;
}
text: "Move me!"
}
}
```

f) Attached property

attached properties are kind of static member in C++ where attributes are accessed through class. Here it is accessed through object type rather than instantiated objects. attached properties can be accessed only in current level and not to its children.

it is declared as

```
<AttachingType>.<propertyName>
<AttachingType>.on<SignalName>
<<MyListView.qml>>
import QtQuick 2.0
ListView {
width: 240; height: 320
model: 3
delegate: Rectangle {
width: 100; height: 30
color: ListView.isCurrentItem ? "red" : "yellow"
}
}
```

Day 1 Morning

2. Qt Properties

- Combination of Get/Set/Notifiy
- Declaration of a Qt property
- Qt Property with Enum
- Getting & Setting Qt Properties
- Dynamic Properties
- Signal and Slots

Enum from c++ side is availabe at qml side through Q_ENUMS() macro

c++

```
class Person:public QObject {
Q_OBJECT
Q_ENUMS(Height)
Q_PROPERTY(Height height READ height WRITE setHeight)
public:
Person(QObject *p=0);
enum Height{SHORT=0,MID,TALL};
Height height() const;
void setHeight(Height);
private:
Height _height;
};
#endif
```

qml

```
import QtQuick 2.0
import Person 1.0
Item{
property list<Person> personmodel:[Person{height:Person.MID}, Person{height:Person.TALL} ]
ListView{
width:200;height:400
model:personmodel
delegate:Text{
text:height
}
}
}
```

Day 1 Morning

2. Qt Properties

- Combination of Get/Set/Notifiy
- Declaration of a Qt property
- Qt Property with Enum
- Getting & Setting Qt Properties
- Dynamic Properties
- Signal and Slots

Qt properties can be get as set through READ and WRITE keyword in Q_PROPERTY macro

<person.h>

```
class Person:public QObject {
Q_OBJECT
Q_PROPERTY(QString name READ name WRITE setName)
public:
Person(QObject *p=0);
QString name() const;
void setName(const QString&);
private:
QString name;
};
```

person.cpp

```
QString Person::name() const{ return _name; }
void Person::setName(const QString& name){
if (name!=_name){
_name=name;
//emit nameChanged();
}
}
```

Day 1 Morning

2. Qt Properties

- Combination of Get/Set/Notifiy
- Declaration of a Qt property
- Qt Property with Enum
- Getting & Setting Qt Properties
- Dynamic Properties
- Signal and Slots

Properies in QMetaObject of QObject can be added dynamically. It is less type safe though.

```
void func(QObject *obj){
obj->setProperty("newintname",1);
```

```
obj->setProperty("newstringname","Qt");
int propint=obj->property("newintname").toInt();
QString propstring=obj->property("newstringname").toString();
QVariant propvariant=obj->property("newvariantname").toInt();
```

Day 1 Morning

2. Qt Properties

- Combination of Get/Set/Notify
- Declaration of a Qt property
- Qt Property with Enum
- Getting & Setting Qt Properties
- Dynamic Properties

- Signal and Slots

Signals are added through signals macro. A property provide binding when NOTIFY keyword is added in Q_PROPERTY

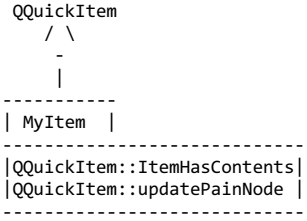
Day 1 Morning

3. Building Blocks of QML

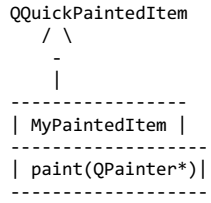
- QuickItem
- Variant
  - QVariant and QML
- Variant Containers
  - QVariantList
  - QVariantMap
  - QList
  - QQmlListProperty
- Import QML document in another QML document as a package

The QQuickItem is c++ class for Object type 'Item' in QtQuick. It provides most basic of all visual items. QQuickItem does not have any visual effect and it is item that is drawn on QQuickView.

For customised drawing



Use QPaintedItem for custom drawing.




```
import QtQuick 2.0
Item {
 Image {
 source: "tile.png"
 }
 Image {
 x: 80
 width: 100
 height: 100
 source: "tile.png"
 }
 Image {
 x: 190
 width: 100
 height: 100
 fillMode: Image.Tile
 source: "tile.png"
 }
}
```


Day 1 Morning

3. Building Blocks of QML

- QuickItem
- Variant
  - QVariant and QML
- Variant Containers
  - QVariantList
  - QVariantMap
  - QList
  - QQmlListProperty
- Import QML document in another QML document as a package

The var type is a generic property type that can refer to any data type. It is equivalent to QVariant in C++ side. It is equivalent to a regular JavaScript variable. For example, var properties can store numbers, strings, objects, arrays and functions:

```
Item {
  property var aNumber: 100
  property var aBool: false
  property var aString: "Hello world!"
  property var anotherString: String("#FF008800")
  property var aColor: Qt.rgba(0.2, 0.3, 0.4, 0.5)
  property var aRect: Qt.rect(10, 10, 10, 10)
  property var aPoint: Qt.point(10, 10)
  property var aSize: Qt.size(10, 10)
  property var aVector3d: Qt.vector3d(100, 100, 100)
```

```
  property var anArray: [1, 2, 3, "four", "five", (function() { return "six"; })]
  property var anObject: { "foo": 10, "bar": 20 }
  property var aFunction: (function() { return "one"; })
  property var car: new Object({wheels: 4})
  property var car1: {'wheels':4}
  property var car2: ({wheels:4})
}
```

Day 1 Morning

3. Building Blocks of QML

- QuickItem
- Variant
  - QVariant and QML
- Variant Containers
  - QVariantList
  - QVariantMap
  - QList
  - QQmlListProperty
- Import QML document in another QML document as a package

-QVariantList

-QVariantMap

Qml

```
//*.qml
Item {
 function readValues(anArray, anObject) {
  for (var i=0; i<anArray.length; i++)
  console.log("Array item:", anArray[i])
  for (var prop in anObject) {
  console.log("Object item:", prop, "=", anObject[prop])
  }
 }
}
```

C++

```
QQuickView view(QUrl::fromLocalFile("MyItem.qml"));
QVariantList list;
list << 10 << QColor(Qt::green) << "bottles";
QVariantMap map;
map.insert("language", "QML");
map.insert("released", QDate(2010, 9, 21));
QMetaObject::invokeMethod(view.rootObject(), "readValues",
 Q_ARG(QVariant, QVariant::fromValue(list)),
 Q_ARG(QVariant, QVariant::fromValue(map)));
```

-QQmlListProperty

The QQmlListProperty class allows applications to expose list-like properties to QML.

QML has many list properties, where more than one object value can be assigned.

The use of a list property from QML looks like this:

```
FruitBasket {
 fruits: [
 Apple {},
 Orange{},
 Banana{}
 ]
}
```

On c++ side it can be added to class like

```
class Fruit:public QObject{
}

class FruitBasket{
 Q_PROPERTY(QQmlListProperty<Fruit> fruits READ fruits);
public:
QQmlListProperty fruits()const;
private:
QList<Fruit*> _fruits;
}
```

it can not read and write, only read.

3. Building Blocks of QML

- QuickItem
- Variant
  - QVariant and QML
- Variant Containers
  - QVariantList
  - QVariantMap
  - QList
  - QQmlListProperty
- Import QML document in another QML document as a package

```
myapp
    |- mycomponents
        |- CheckBox.qml
        |- DialogBox.qml
        |- Slider.qml
    |- main
        |- application.qml
```

<application.qml>

```
<import "../mycomponents"

DialogBox {
    CheckBox {
        // ...
    }
    Slider {
        // ...
    }
}

import "../mycomponents" as MyComponents

MyComponents.DialogBox {
    // ...
}
```

for remotely located directory, directory must have a file 'qmldir'

<qmldir>

```
CheckBox CheckBox.qml
DialogBox DialogBox.qml
Slider Slider.qml
```

directory can be imported like

```
import "http://www.my-example-server.com/myapp/mycomponents"

DialogBox {
    CheckBox {
        // ...
    }
    Slider {
        // ...
    }
}
```

```
              <<parent/child>>
  QQuickItem <>---------------> QQuickItem
```

One QQuickItem is nested in other QQuickItem. Its like parent child relationship.

```
import QtQuick 2.0
Rectangle {            //<---------- Parent
 width: 400; height: 400
 color: "lightblue"
 Rectangle {           //<---------- Child
  x: 50; y: 50; width: 300; height: 300
  color: "green"
  Rectangle {
   x: 200; y: 150; width: 50; height: 50
   color: "white"
  }
 }
}
```


Day 1 Afternoon

4. Composing UIs

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

Colors

```
import QtQuick 2.0
Item {
width: 300; height: 100
Rectangle {
x: 0; y: 0; width: 100; height: 100; color: "#ff0000"
}
Rectangle {
x: 100; y: 0; width: 100; height: 100
color: Qt.rgba(0,0.75,0,1)
}
Rectangle {
x: 200; y: 0; width: 100; height: 100; color: "blue"
}
}
```

Images

```
import QtQuick 2.0
Rectangle {
 width: 400; height: 400
 color: "black"
 Image {
  x: 150; y: 150
  source: "../images/rocket.png"
  scale:2.0
  rotation:45.0
 }
}
```

Gradients

```
import QtQuick 2.0
Rectangle {
 width: 400; height: 400
 gradient: Gradient {
  GradientStop {
   position: 0.0; color: "green"
  }
  GradientStop {
   position: 1.0; color: "blue"
  }
 }
}
```

Gradient Images

```
import QtQuick 2.0
Rectangle {
 width: 425; height: 200
 Image {
  x: 0; y: 0
  source: "../images/vertical-gradient.png"
 }
 Image {
```

```
  x: 225; y: 0
  source: "../images/diagonal-gradient.png"
 }
}
```

Border Images

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

Day 1 Afternoon

4. Composing UIs

- Nested Items
- Graphical QML Types
- **Text Type**
- Anchor Layouts

Text Elements

```
import QtQuick 2.0
Rectangle {
 width: 400; height: 400
 color: "lightblue"
 Text {
  x: 100; y: 100
  text: "Qt Quick"
  font.family: "Helvetica"
  font.pixelSize: 32
 }
}
```

Text Input

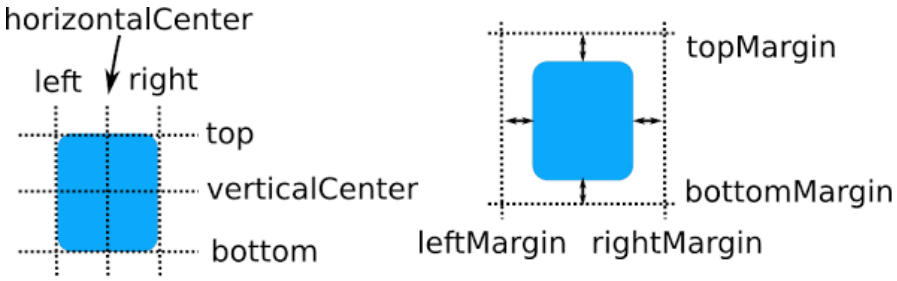
```
import QtQuick 2.0
Rectangle {
 width: 400; height: 400
 color: "lightblue"
 TextInput {
  x: 50; y: 100; width: 300
  text: "Editable text"
  font.family: "Helvetica"; font.pixelSize: 32
 }
}
```

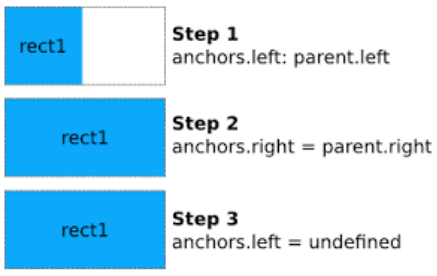
Day 1 Afternoon

4. Composing UIs

- Nested Items
- Graphical QML Types
- Text Type
- **Anchor Layouts**

Anchor layout works only among parent and children.





Various anchors attributes

```
anchors.top : AnchorLine
anchors.bottom : AnchorLine
anchors.left : AnchorLine
anchors.right : AnchorLine
anchors.horizontalCenter : AnchorLine
anchors.verticalCenter : AnchorLine
anchors.baseline : AnchorLine
anchors.fill : Item
anchors.centerIn : Item
anchors.margins : real
anchors.topMargin : real
anchors.bottomMargin : real
anchors.leftMargin : real
anchors.rightMargin : real
anchors.horizontalCenterOffset : real
anchors.verticalCenterOffset : real
anchors.baselineOffset : real
anchors.alignWhenCentered : bool
```

```
import QtQuick 2.0
Rectangle {
width: 400; height: 200
color: "lightblue"
 Image { id: book; source: "../images/book.svg"
 anchors.left: parent.left
 anchors.leftMargin: parent.width/16
 anchors.verticalCenter: parent.verticalCenter }
 Text { text: "Writing"; font.pixelSize: 32
 anchors.left: book.right
 anchors.leftMargin: 32
 anchors.baseline: book.verticalCenter }
}
```

Day 2 Morning

5. User Input

- Keyboard Navigation and Focus
- Input Focus
  - Focus property
  - Focus Order
  - Active Focus
  - Focus Scope
- Handling Keyboard input
  - Key-Specific Handlers
  - All keys Handler
  - Key Event Propogation
  - Event Propogation is Convenient
- Multi-Touch
  - Common Multi-Touch Gestures
  - Handling Overlapping Touch Areas
- Mouse/Single-Touch
  - Tap
  - Double-Tap
  - Top and Hold

When the user presses or releases a key, the following occurs:
1) Qt receives the key action and generates a key event.
2) If a QQuickWindow is the active window, the key event is delivered to it.
3) The key event is delivered by the scene to the Item with active focus. If no item has active focus, the key event is ignored.
4) If the QQuickItem with active focus accepts the key event, propagation stops. Otherwise the event is sent to the Item's parent until the event is accepted, or the root item is reached.

Rectangle has active focus and the A key is pressed, the event will not be propagated further. Upon pressing the B key, the event will propagate to the root item and thus be ignored.

```
Rectangle {
 width: 100; height: 100
 focus: true    <-----------------item with focus attribute
 Keys.onPressed: {
  if (event.key == Qt.Key_A) {
   console.log('Key A was pressed');
   event.accepted = true;
  }
 }
}
```

If the root Item is reached, the key event is ignored and regular Qt key handling continues.

Day 2 Morning

5. User Input

- Keyboard Navigation and Focus
- Input Focus
  - Focus property
  - Focus Order
  - Active Focus
  - Focus Scope
- Handling Keyboard input
  - Key-Specific Handlers
  - All keys Handler
  - Key Event Propogation
  - Event Propogation is Convenient
- Multi-Touch
  - Common Multi-Touch Gestures
  - Handling Overlapping Touch Areas
- Mouse/Single-Touch
  - Tap
  - Double-Tap
  - Top and Hold

- Focus property

SubItem containing the 'focus' property true recieves the focus

```
Rectangle {
 color: "lightsteelblue"; width: 240; height: 25
 Text { id: myText }
 Item {
  id: keyHandler
  focus: true
  Keys.onPressed: {
   if (event.key == Qt.Key_A)
    myText.text = 'Key A was pressed'
   else if (event.key == Qt.Key_B)
    myText.text = 'Key B was pressed'
   else if (event.key == Qt.Key_C)
    myText.text = 'Key C was pressed'
  }
 }
```

```
}
```

- Focus Order

Focus Order is decided among items for which 'focus' attributes are set true. If Item is reused in other component then last item with focus attribute would receive the focus.

```
//Window code that imports MyWidget
Rectangle {
 id: window
 color: "white"; width: 240; height: 150
 Column {
  anchors.centerIn: parent; spacing: 15
  MyWidget {
   focus: true             //set this MyWidget to receive the focus
   color: "lightblue"
  }
  MyWidget {
   color: "palegreen"
  }
 }
}
```

The MyWidget code:

```
Rectangle {
 id: widget
 color: "lightsteelblue"; width: 175; height: 25; radius: 10; antialiasing: true
 Text { id: label; anchors.centerIn: parent}
 focus: true
 Keys.onPressed: {
  if (event.key == Qt.Key_A)
   label.text = 'Key A was pressed'
  else if (event.key == Qt.Key_B)
   label.text = 'Key B was pressed'
  else if (event.key == Qt.Key_C)
   label.text = 'Key C was pressed'
 }
}
```

focus order can be shifted to other userinput through KeyNavigation.tab press.

```
import QtQuick 2.0
Rectangle{
width:200:height:200
color:"steelblue"
 TextInput{
 id:input1
 x:8;y:8
 width:96;height:20
 focus:true
 text:"Text Input 1"
 KeyNavigation.tab:input2
 }
 TextInput {
 id:input2
 x:8;y:32
 width:96;height:20
 text:"Text Input 2"
 KeyNavigation.tab:input1
 }
}
```

-Active focus

Which Item has the focus can be decided with property activeFocus.

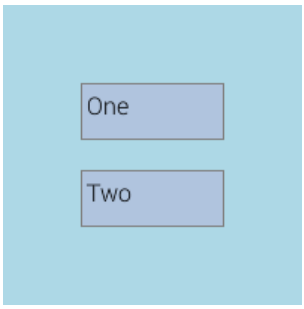
```
Text {
 text: activeFocus ? "I have active focus!" : "I do not have active focus"
}
```

-Focus Scope

two textinput inside an Item container can recieve KeyNavigation.tab but to textinput inside two Item container can not receive KyNavigation.tab. In this case Item container is placed in Focus scope.inside a focus scope last textinput with 'focus:true' would receive the key focus.

```
import QtQuick 2.5
FocusScope{
width:96;height:input.height+8
 Rectangle{
 anchors.fill:parent
 color:"lightsteelblue"
 border.color:"gray"
 }
 property alias text:input.text
 TextInput{
 id:input
 height:30
 anchors.fill:parent
```

```
 anchors.margins:4
 focus:true
 }
}
import QtQuick 2.0
Rectangle{
width:200;height:200
 Column{
 anchors.centerIn:parent
 spacing:20
  FocusScopeTextInput{
  id:input1
  text:"Input1"
  KeyNavigation.tab:input2
  }
  FocusScopeTextInput{
  id:input2
  text:"Input2"
  KeyNavigation.tab:input1
  }
 }
}
```



Day 2 Morning

5. User Input

- Keyboard Navigation and Focus
- Input Focus
  - Focus property
  - Focus Order
  - Active Focus
  - Focus Scope
- Handling Keyboard input
  - Key-Specific Handlers
  - All keys Handler
  - Key Event Propogation
  - Event Propogation is Convenient
- Multi-Touch
  - Common Multi-Touch Gestures
  - Handling Overlapping Touch Areas
- Mouse/Single-Touch
  - Tap
  - Double-Tap
  - Top and Hold

Any visual item can receive keyboard input through the Keys attached type. Keys can be handled via the onPressed and onReleased signal properties. The signal properties have a KeyEvent parameter, named event which contains details of the event. If a key is handled event.accepted should be set to true to prevent the event from propagating up the item hierarchy.

Kyes

Properties

```
enabled : bool
forwardTo : list<Object>
priority : enumeration
```

Signals

```
asteriskPressed(KeyEvent event)
backPressed(KeyEvent event)
backtabPressed(KeyEvent event)
callPressed(KeyEvent event)
cancelPressed(KeyEvent event)
context1Pressed(KeyEvent event)
context2Pressed(KeyEvent event)
context3Pressed(KeyEvent event)
context4Pressed(KeyEvent event)
deletePressed(KeyEvent event)
digit[0-9]Pressed(KeyEvent event)
downPressed(KeyEvent event)
enterPressed(KeyEvent event)
escapePressed(KeyEvent event)
flipPressed(KeyEvent event)
hangupPressed(KeyEvent event)
leftPressed(KeyEvent event)
menuPressed(KeyEvent event)
noPressed(KeyEvent event)
pressed(KeyEvent event)
released(KeyEvent event)
returnPressed(KeyEvent event)
rightPressed(KeyEvent event)
selectPressed(KeyEvent event)
spacePressed(KeyEvent event)
tabPressed(KeyEvent event)
upPressed(KeyEvent event)
volumeDownPressed(KeyEvent event)
```

```
volumeUpPressed(KeyEvent event)
yesPressed(KeyEvent event)
```

- Key-Specific Handlers
- All keys Handler

```
Item {
anchors.fill: parent
focus: true
 Keys.onPressed: {
  if (event.key == Qt.Key_Left) {
  console.log("move left");
  event.accepted = true;
  }
 }
}
```

-Key Event Propagation

```
forwardTo : list<Object>
```

This property provides a way to forward key presses, key releases, and keyboard input coming from input methods to other items. This can be useful when you want one item to handle some keys (e.g. the up and down arrow keys), and another item to handle other keys (e.g. the left and right arrow keys). Once an item that has been forwarded keys accepts the event it is no longer forwarded to items later in the list.
This example forwards key events to two lists:

```
Item {
 ListView {
 id: list1
 // ...
 }
 ListView {
 id: list2
 // ...
 }
 Keys.forwardTo: [list1, list2]
 focus: true
}
```

Day 2 Morning

5. User Input

- Keyboard Navigation and Focus
- Input Focus
  - Focus property
  - Focus Order
  - Active Focus
  - Focus Scope
- Handling Keyboard input
  - Key-Specific Handlers
  - All keys Handler
  - Key Event Propogation
  - Event Propogation is Convenient
- Multi-Touch
  - Common Multi-Touch Gestures
  - Handling Overlapping Touch Areas
- Mouse/Single-Touch
  - Tap
  - Double-Tap
  - Top and Hold

MultiPointTouchArea

Properties

```
maximumTouchPoints : int
minimumTouchPoints : int
mouseEnabled : bool
touchPoints : list<TouchPoint>
```

Signals

```
canceled(list<TouchPoint> touchPoints)
gestureStarted(GestureEvent gesture)
pressed(list<TouchPoint> touchPoints)
released(list<TouchPoint> touchPoints)
touchUpdated(list<TouchPoint> touchPoints)
updated(list<TouchPoint> touchPoints)
```

A MultiPointTouchArea is an invisible item that is used to track multiple touch points.

```
import QtQuick 2.0
Rectangle {
width: 400; height: 400
 MultiPointTouchArea {
 anchors.fill: parent
  touchPoints: [
  TouchPoint { id: point1 },
```

```
 TouchPoint { id: point2 }
 ]
}
Rectangle {
width: 30; height: 30
color: "green"
x: point1.x
y: point1.y
}
Rectangle {
width: 30; height: 30
color: "yellow"
x: point2.x
y: point2.y
}
}
```

Day 2 Morning

5. User Input

- Keyboard Navigation and Focus
- Input Focus
  - Focus property
  - Focus Order
  - Active Focus
  - Focus Scope
- Handling Keyboard input
  - Key-Specific Handlers
  - All keys Handler
  - Key Event Propogation
  - Event Propogation is Convenient
- Multi-Touch
  - Common Multi-Touch Gestures
  - Handling Overlapping Touch Areas
- Mouse/Single-Touch
  - Tap
  - Double-Tap
  - Top and Hold

Day 2 Morning

6. Components and Structures

- Components
  - Dividing code into Components
    - Creating new Items through new .qml file
    - Creating Component dynamically
  - Modules
  - Write qml component as a separate .qml file
  - qmldir file
    - Directory import
    - Module import

a component is a resuable, encapsulated QML type with well-defined interfaces. Components are often defined by component files - that is, .qml files. the component type essentially allows QML componens to be defined inline, within a QML document, rather than as a separate QML file.

```
import QtQuick 2.0
Item {
width: 100; height: 100
 Component {
 id: redSquare
  Rectangle {
  color: "red"
  width: 10
  height: 10
  }
 }
 Loader { sourceComponent: redSquare }
 Loader { sourceComponent: redSquare; x: 20 }
}
```


Day 2 Morning

6. Components and Structures

- Components
- Dividing code into Components
  - Creating new Items through new .qml file
  - Creating Component dynamically
- Modules
- Write qml component as a separate .qml file
- qmldir file
  - Directory import
  - Module import

-Creating new Items through new .qml file

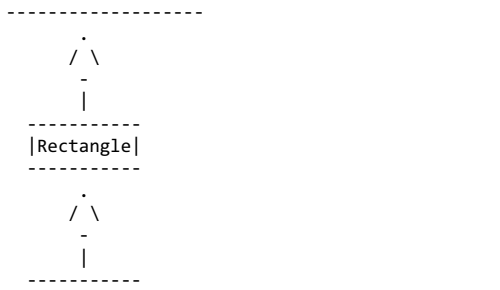
```
import QtQuick 2.0
Rectangle { width: 200; height: 200; color: "red" }
Rectangle { width: 200; height: 200; color: "blue" } // invalid two root object
```

Creating new Button type

```
// Button.qml
import QtQuick 2.0
Rectangle {
 width: 100; height: 100
 color: "red"
 MouseArea {
  anchors.fill: parent
  onClicked: console.log("Button clicked!")
 }
}
```

The Button type can then be used in an application:

```
// application.qml
import QtQuick 2.0
Column {
 Button { width: 50; height: 50 }
 Button { x: 50; width: 100; height: 50; color: "blue" }
 Button { width: 50; height: 50; radius: 8 }
}
```



```
          | : Button| // Button inherits Rectangle in Button.qml and
          ---------- // gets instantiated in application.qml
```

- Creating Component dynamically

a)

```
import QtQuick 2.0
Item {
 width: 100; height: 100
 Component {
  id: redSquare
  Rectangle {
   color: "red"
   width: 10
   height: 10
  }
 }
 Loader { sourceComponent: redSquare }
 Loader { sourceComponent: redSquare; x: 20 }
```

where id is extra provided than file component

b)

```
var component = Qt.createComponent("Button.qml");
 if (component.status == Component.Ready)
  component.createObject(parent, {"x": 100, "y": 100});
```

c)

```
Item {
 id: root
 width: 500; height: 500
 Component {
  id: myComponent
  Rectangle { width: 100; height: 100; color: "red" }
 }
 Component.onCompleted: {
  myComponent.createObject(root)
  myComponent.createObject(root, {"x": 200})
 }
}
```

Day 2 Morning

6. Components and Structures

- Components
- Dividing code into Components
  - Creating new Items through new .qml file
  - Creating Component dynamically
- **Modules**
- Write qml component as a separate .qml file
- qmldir file
  - Directory import
  - Module import

QML Modules

A QML module provides versioned types and JavaScript resources in a type namespace which may be used by clients who import the module. The types which a module provides may be defined in C++ within a plugin, or in QML documents. Modules make use of the QML versioning system which allows modules to be independently updated.

Defining of a QML module allows:

1)The sharing of common QML types within a project - for example, a group of UI components that are used by different windows

2)The distribution of QML-based libraries

3)The modularization of distinct features, so that applications only load the libraries necessary for their individual needs

4) Versioning of types and resources so that the module can be updated safely without breaking client code

Day 2 Morning

6. Components and Structures

- Components
- Dividing code into Components
  - Creating new Items through new .qml file
  - Creating Component dynamically
- Modules
- **Write qml component as a separate .qml file**
- qmldir file
  - Directory import
  - Module import

```
//ButtonC.qml
  Rectangle
    /
```

```
     -
     |
   ButtonC <>--+---> Text
               |
               +---> MouseArea
```

```
<ButtonC.qml>
import QtQuick 2.0
Rectangle {
id:button
property alias text: label.text
signal clicked()
color: "blue"
 Text {
 id: label
 anchors.centerIn: parent
 }
 MouseArea{
 anchors.fill: parent
 onClicked: button.clicked()
 }
}
```

Day 2 Morning

6. Components and Structures

- Components
- Dividing code into Components
  - Creating new Items through new .qml file
  - Creating Component dynamically
- Modules
- Write qml component as a separate .qml file
- qmldir file
  - Directory import
  - Module import

There are two distinct types of qmldir files:
QML document directory listing files
QML module definition files

-Directory import
all qml file in a directory,i.e. directory1 can be import using

```
import "./directory1"
```

or

```
import "./directory1" as mydirectory

mydirectory.Rectangle
...
```

An internal object type declaration allows a QML document to be registered as a type which becomes available only to the other QML documents contained in the directory import. The internal type will not be made available to clients who import the directory.
Example:

```
internal HighlightedButton HighlightedBtn.qml
```

A JavaScript resource declaration allows a JavaScript file to be exposed via the given identifier.
Example:

```
  MathFunctions mathfuncs.js
```

Additionally, JavaScript resources in the directory are not exposed to clients unless they are declared in a qmldir file.

Remotely located directory
A directory of QML files can also be imported from a remote location if the directory contains a directory listing qmldir file. For example, if the myapp directory in the previous example was hosted at "http://www.my-example-server.com", and the mycomponents directory contained a qmldir file defined as follows:

```
http://www.my-example-server.com/ myapp
                                      |- mycomponents
                                         |- CheckBox.qml
                                         |- DialogBox.qml
                                         |- Slider.qml
                                         |- qmldir
```

qmldir

```
  CheckBox CheckBox.qml
  DialogBox DialogBox.qml
  Slider Slider.qml
```

main.qml

```
import "http://www.my-example-server.com/myapp/mycomponents"
DialogBox {
 CheckBox {
 // ...
 }
 Slider {
 // ...
 }
}
```

-Module Import

Module import happens with versioning. A qmldir file must be present in the module directory

```
import MyModule 1.0
```

There must be directory called MyModule or MyModule.1 or MyModule.1.0 .
Directory containing module directory needs to be add to QQmlEngine::addImportPath.
qmldir in side MyModule directory

```
  module MyModule
  CustomButton 1.0 CustomButton.qml
  CustomButton 2.0 CustomButton20.qml
  CustomButton 2.1 CustomButton21.qml
  plugin examplemodule
  MathFunctions 2.0 mathfuncs.js
```

for

```
import a.b.c.d.e 1.0
```

qmldir file should be available in directory relatative to QQmlEngine::addImportPath("xyz") as
xyz/a/b/c/d/e

Day 2 Morning

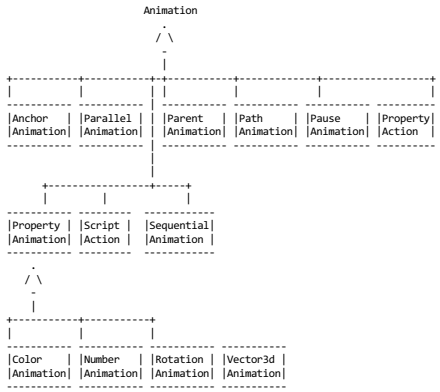
12. Qt Quick Controls

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

Day 2 Morning

12. Qt Quick Controls

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



Control is the base type of user interface controls. It receives input events from the window system, and paints a representation of itself on the screen.

Qt Quick Controls classes comes from module QtQuick.Controls

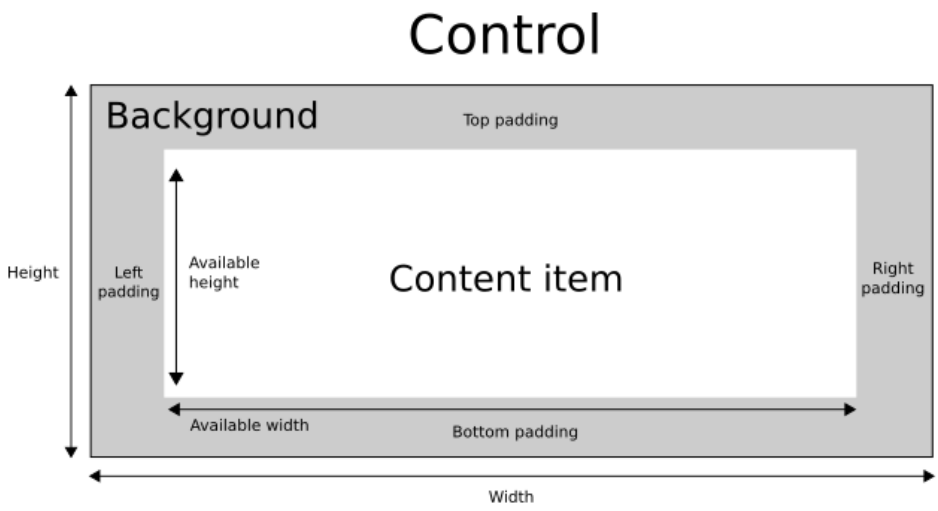
classes

```
ApplicationWindow : Provides a top-level application window
BusyIndicator : A busy indicator
Button : A push button with a text label
Calendar : Provides a way to select dates from a calendar
CheckBox : A checkbox with a text label
ComboBox : Provides a drop-down list functionality
GroupBox : Group box frame with a title
Label : A text label
Menu : Provides a menu component for use as a context menu, popup menu, or as part of a menu bar
MenuBar : Provides a horizontal menu bar
ProgressBar : A progress indicator
RadioButton : A radio button with a text label
ScrollView : Provides a scrolling view within another Item
Slider : Provides a vertical or horizontal slider control
SpinBox : Provides a spin box control
SplitView : Lays out items with a draggable splitter between each item
StackView : Provides a stack-based navigation model
StackViewDelegate : A delegate used by StackView for loading transitions
StatusBar : Contains status information in your app
Switch : A switch Tab Represents the content of a tab in a TabView
TabView : A control that allows the user to select one of multiple stacked items
TableViewColumn : Used to define columns in a TableView or in a TreeView
TextArea : Displays multiple lines of editable formatted text
TextField : Displays a single line of editable plain text
ToolBar : Contains ToolButton and related controls
ToolButton : Provides a button type that is typically used within a ToolBar
TableView : Provides a list view with scroll bars, styling and header sections
TreeView : Provides a tree view with scroll bars, styling and header sections
Action : Abstract user interface action that can be bound to items
ExclusiveGroup : Way to declare several checkable controls as mutually exclusive
MenuItem : Item to add in a menu or a menu bar
MenuSeparator : Separator for items inside a menu
Stack : Provides attached properties for items pushed onto a StackView
```

A Quick control has following property

Property

```
Properties
availableHeight : real
availableWidth : real
background : Item
bottomPadding : real
contentItem : Item
focusPolicy : enumeration
focusReason : enumeration
font : font
hoverEnabled : bool
hovered : bool
leftPadding : real
locale : Locale
mirrored : bool
padding : real
rightPadding : real
spacing : real
topPadding : real
visualFocus : bool
wheelEnabled : bool
```



e.x Button

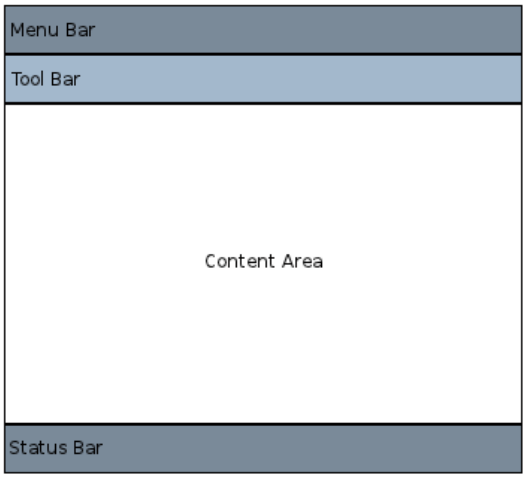
```
RowLayout {
 Button {
 text: "Ok"
 onClicked: model.submit()
 }
 Button {
 text: "Cancel"
 onClicked: model.revert()
 }
}
```

Day 2 Morning

12. Qt Quick Controls

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

ApplicationWindow is similar to QMainWindow on QWidget side which contains MeneBar ToolBar Statusbar like predefined gadgets.



example

```
ApplicationWindow {
 id: window
 visible: true
 menuBar: MenuBar {
  Menu { MenuItem {...} }
  Menu { MenuItem {...} }
 }
 toolBar: ToolBar {
  RowLayout {
   anchors.fill: parent
   ToolButton {...}
  }
 }
 TabView {
  id: myContent
  anchors.fill: parent
```

```
  ...
 }
}
```

On C++ side QtQuickView can not be used as C++ type as it instantiate ApplicationWindow, instead QQmlApplicationEngine would be used. To fetch the QQuickWindow, root object can be returned

```
int main(int argc,char *argv[]){
QGuiApplication(argc, argv);
QQmlApplicationEngine engine("main.qml");
QQuickWindow *win=engine.rootObjects()[0];
return app.exec();
}
```

Day 2 Morning

12. Qt Quick Controls

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

```
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
}
}
```

Using Views

List Views

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

Day 2 Morning

12. Qt Quick Controls

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

Grid Layout

```
import QtQuick 2.0
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" }
}
```

```
import QtQuick 2.0
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" } }
}
}
```

12. Qt Quick Controls

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

qml application can have many styling which can be set
a)in main.cpp

```
    QQmlApplicationEngine engine;
    QQuickStyle::setStyle("Fusion");
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;
```

b) qtquickcontrols2.conf. Qt Quick Controls 2 support a special configuration file, :/qtquickcontrols2.conf, that is built into an application's resources.

```
[Controls]
Style=Material

[Universal]
Theme=Dark
Accent=Red

[Material]
Theme=Light
Accent=Teal
Primary=BlueGrey
```

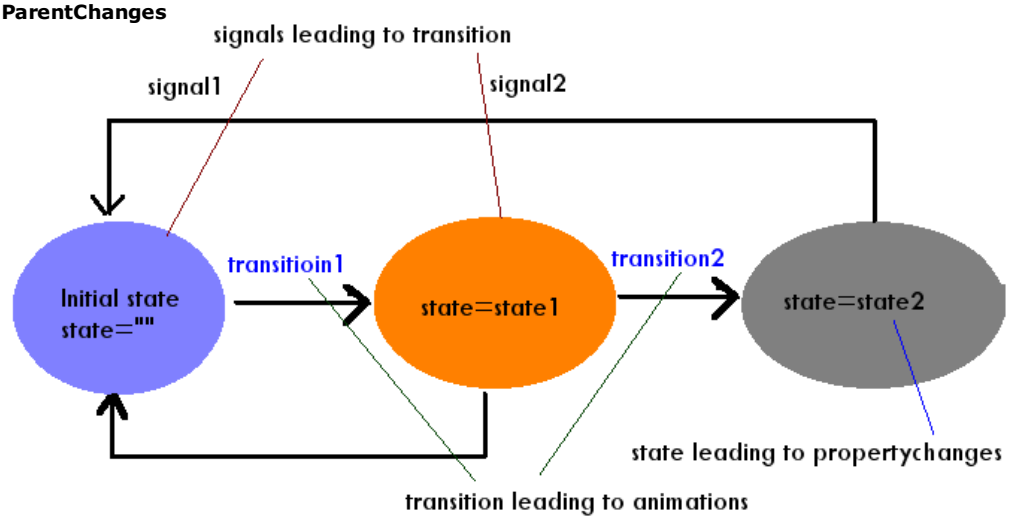
Day 2 Afternoon

7. State and Transitions

- States
  - State Conditions
  - Transitions

A state is a set of batched changes from the default configuration. All "Item" QtQuick type have a default state (state=="") that defines the default configuration of objects and property values. New states can be defined by adding State items to the states property to allow items to switch between different configurations.

Various changes that can be achieved through state are
PropertyChanges
AnchorChanges
ParentChanges



Item QtQuick type has two properties

```
state:string
states:list<State>
```

Sample Code of states property of Item

```
import QtQuick 2.0
Rectangle {
 id: root                  -------- state triggering in signal handler
 width: 100; height: 100  /
 MouseArea {              /
  id: mouseArea           |
  anchors.fill: parent    v
  onClicked: root.state == '' ? root.state = "red_color" : (root.state=='red_color' ? root.state = 'blue_color':root.state="");
 }
 states: [
  State {
   name: "red_color"
   PropertyChanges { target: root; color: "red" } <---Changing targets property
  },
  State {
   name: "blue_color"
   PropertyChanges { target: root; color: "blue" } <---Changing targets property
  }
 ]
}
```


Day 2 Afternoon

7. State and Transitions

- States
  - State Conditions
  - Transitions

State QtQuick type has following property

```
changes : list<Change>
extend : string
name : string
when : bool
```

State would take effect only when 'when' property holds true

```
Rectangle {
 id: myRect
 width: 100; height: 100
 color: "red"
 MouseArea { id: mouseArea; anchors.fill: parent }
 states: State {
  name: "hidden"; when: mouseArea.pressed
  PropertyChanges { target: myRect; opacity: 0 }
 }
```

}

Day 2 Afternoon

7. State and Transitions

- States
- State Conditions
- Transitions

Transition comes into effect when Item changes its state. Item sits on default state in initial phase.
Transition has following property

```
animations : list<Animation>
enabled : bool
from : string
reversible : bool
running : bool
to : string
```

Here transition triggers NumberAnimation.

```
import QtQuick 2.0
Rectangle {
 id: rect
 width: 100; height: 100
 color: "red"
 MouseArea {
  id: mouseArea
  anchors.fill: parent
 }
 states: State {
  name: "moved"; when: mouseArea.pressed
  PropertyChanges { target: rect; x: 50; y: 50 }
 }
 transitions: Transition {
  NumberAnimation { properties: "x,y"; easing.type: Easing.InOutQuad }
 }
}
```

Transitions During State Changes
Qt Quick States are property configurations where a property may have different values to reflect different states. State changes introduce abrupt property changes; animations smooth transitions to produce visually appealing state changes.

The Transition type can contain animation types to interpolate property changes caused by state changes. To assign the transition to an object, bind it to the transitions property.

<main.qml>

```
import QtQuick 2.0
Rectangle {
width: 75; height: 75
id: button
state: "RELEASED"
 MouseArea {
 anchors.fill: parent
 onPressed: button.state = "PRESSED"
 onReleased: button.state = "RELEASED"
 }
 states: [
  State {
  name: "PRESSED"
  PropertyChanges { target: button; color: "lightblue"}
  },
  State {
  name: "RELEASED"
  PropertyChanges { target: button; color: "green"}
  }
 ]
 transitions: [
  Transition {
  from: "PRESSED"
  to: "RELEASED"
  ColorAnimation { target: button; duration: 100}
  },
  Transition {
  from: "RELEASED"
  to: "PRESSED"
  ColorAnimation { target: button; duration: 100}
  },
  Transition {
  to: "*"
  PropertyAnimation { target: button; properties:"x"; duration: 100}
  }
 ]
```

```
}
```

MOUSE PRESS MOUSE RELEASE

Day 3 Morning

8. Dynamic Creation of Items

- Creating custom Item
  - Creating Items Dynamically
    - Procedural Method
    - Procedural/Declarative Creation
    - Declarative Method
  - Creating Multiple Items
  - Repeaters

Typically an Item placed in a file(document) as root element becomes a component. This new component behaves as subclass of the root Item. i.e. Rectangle is an item and when it is placed in Button.qml file, 'Button' becomes a new type. Editing Button.qml and adding attributes and methods to Rectangle is extending the class Rectangle where existing methods can be overriden.

```
//ButtonC.qml
  Rectangle
    / \
     -
     |
   ButtonC <>--+---> Text
               |
               +---> MouseArea
```

<ButtonC.qml>

```
import QtQuick 2.0
Rectangle {
id:button
property alias text: label.text
signal clicked()
color: "blue"
 Text {
 id: label
 anchors.centerIn: parent
 }
 MouseArea{
 anchors.fill: parent
 onClicked: button.clicked()
 }
}
```

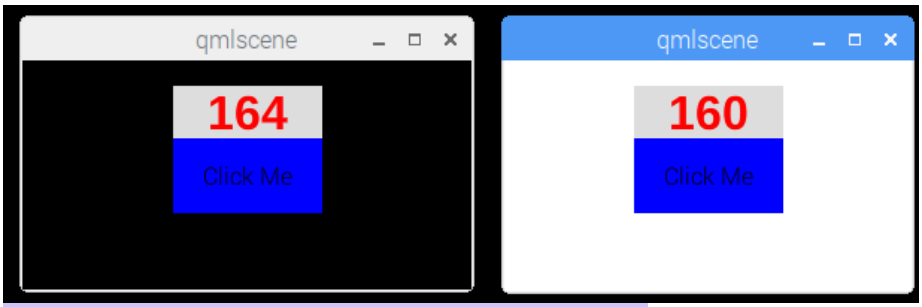
<LabelC.qml>

```
import QtQuick 2.7
Rectangle{
 color:"#dddddd"
 Text{
 id:label
 anchors.centerIn:parent
 anchors.verticalCenter:parent.verticalCenter
 property int count:1
 font.family: "Helvetica"
 font.pointSize: 24
 color: "red"
 height:parent.height
 text:""+this.count+""
  function incrementcount(){
   this.count=this.count+1
  }
 }
 Timer{
 interval:1000;repeat:true;running:true
 onTriggered:label.incrementcount()
 }
}
```

<main.qml>

```
import QtQuick 2.7
import QtQuick.Layouts 1.3
Rectangle{
id: toplevel
color: "black"
 ColumnLayout{
 spacing:20
 anchors.centerIn:parent
  LabelC{
  width:100;height:50
  }
  ButtonC{
  width:100;height:50
  text:"Click Me"
  onClicked:toplevel.color="white"
  anchors.centerIn:parent
   MouseArea {
   //  onPressedChanged: button.color = (pressed) ? "red" : "blue"
   }
  }
 }
```

```
}
```



Day 3 Morning

8. Dynamic Creation of Items

- Creating custom Item
- Creating Items Dynamically
  - Procedural Method
  - Procedural/Declarative Creation
  - Declarative Method
- Creating Multiple Items
- Repeaters

-Procedural Method

A document file (i.e. ButtonC.qml) can be crated as a Component, through Qt.createComponent(), with single root Item or through a componentbbject type that is declared in a document file itself. Component declared in a document file behave as an root Item 'declaration' which is 'defined' by <componentobject>.createObject() function. Component from a document file is first created through Qt.createComponent() function and then component object can be instantiated to Item objects through <componentobject>.createObject() function. When a document file is not available then Qt.createQmlObject function can be used to create object with document specified in a string.

> If Qt.createComponent() is used, the creation context is the QQmlContext in which this method is called
> If Qt.createQmlObject() is called, the creation context is the context of the parent object passed to this method
> If a Component{} object is defined and createObject() or incubateObject() is called on that object, the creation context is the context in which the Component is defined

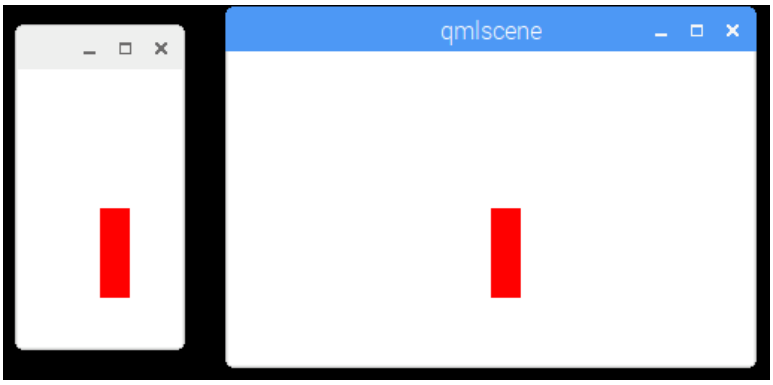
- Qt.createComponent() -> <componentobject>.createObject()

<Sprite.qml>
```
import QtQuick 2.0
Rectangle {
color: "red"
width: 20
height: 20
//x:appWindow.width/2
//y:appWindow.height/2
}
```

<main.qml>
```
import QtQuick 2.0
import "componentCreation.js" as MyScript

Rectangle {
id: appWindow
width: 300; height: 300
property var object:undefined
 Component.onCompleted: {
  object=MyScript.createSpriteObjects();
  object.height=60
 }
}
```

<componentCreation.js>
```
var component;
var sprite;

function createSpriteObjects() {
 component = Qt.createComponent("Sprite.qml");
 if (component.status == Component.Ready)
  return finishCreation();
 else
  component.statusChanged.connect(finishCreation);
}
```

```
function finishCreation() {
 if (component.status == Component.Ready) {
 //sprite = component.createObject(appWindow);
 sprite = component.createObject(appWindow,{"x":Qt.binding(function(){return appWindow.width/2}),"y":Qt.binding(function(){return
appWindow.height/2})});
  if (sprite == null) {
  console.log("Error creating object");
  }else return sprite;
 } else if (component.status == Component.Error) {
        console.log("Error loading component:", component.errorString());
        }
 return null;
}
```



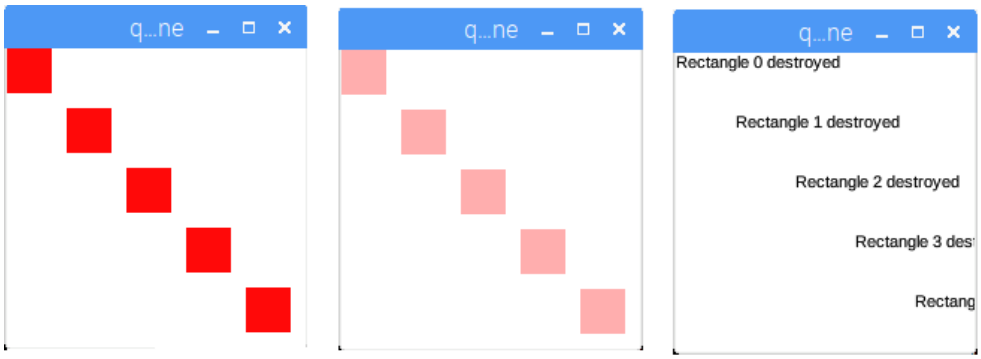
Object can be created without blocking using incubateObject().

Destroying.
Object can be destroyed using destroy() method.

```
<TextC.qml>
import QtQuick 2.0
Text{
font.family:"Helvetica"
font.pointSize:8
}
```

```
<SelfDestroyingRect.qml>
import QtQuick 2.0
Rectangle {
id: rect
width: 30; height: 30
property int count:0
color: "red"
 NumberAnimation on opacity {
 to: 0
 duration: 10000
  onRunningChanged: {
   if (!running) {
   //console.log("Destroying...")
   rect.destroy();
   textlist[count].text="Rectangle "+count+" destroyed";
   }
  }
 }
}
```

```
<main.qml>
import QtQuick 2.0
Item {
id: container
width: 200; height: 200
property var textlist:[]
 Component.onCompleted: {
 var component = Qt.createComponent("SelfDestroyingRect.qml");
 var componenttext = Qt.createComponent("TextC.qml");
  for (var i=0; i<5; i++) {
  var object = component.createObject(container,{"count":i});
  container.textlist.push(componenttext.createObject(container));
  container.textlist[i].x=object.x=(object.width + 10) * i;
  container.textlist[i].y=object.y=(object.height + 10) * i;
  }
 }
}
```



opacity = 1 Opacity -> 0 opacity = 0

- Qt.createQmlObject

```
var newObject = Qt.createQmlObject('import QtQuick 2.0; Rectangle {color: "red"; width: 20; height: 20}',
                                   parentItem,
                                   "dynamicSnippet1");
```

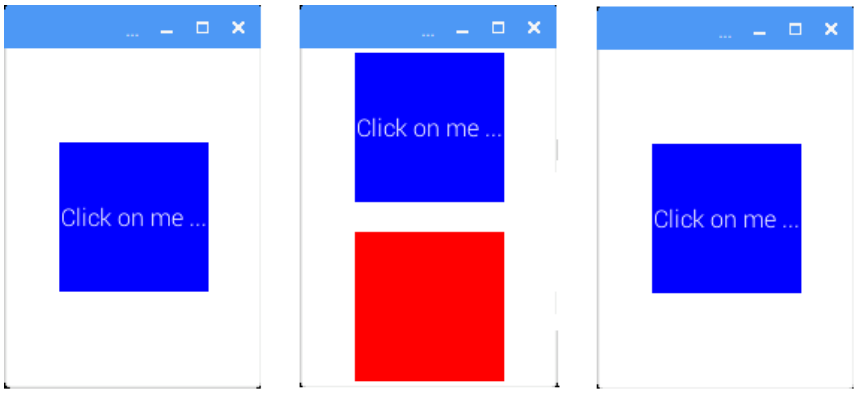
dynamicSnippet1 is file for error log.

-Procedural/Declrative creation

a Component type in a document can be directly instantiated through createObject() function call on the Component.

```
<ButtonC.qml>
import QtQuick 2.0
Rectangle {
id:button
property alias text: label.text
property alias textcolor:label.color
signal clicked()
color: "blue"
 Text {
 id: label
 anchors.centerIn: parent
 }
 MouseArea{
 anchors.fill: parent
 onClicked: button.clicked()
 }
}
```

```
<main.qml>
import QtQuick 2.0
Item{
id:root
//property var obj
 Column{
 id:columnid
 anchors.centerIn:parent
 spacing:20
  ButtonC{
  width:100;height:100
  //anchors{top:root.top;left:root.left}
  text:"Click on me ..."
  textcolor:"white"
   onClicked:{
   //root.obj=dialogComponent.createObject(root);
   componentid.createObject(columnid);
   }
  }
  Component{
  id:componentid
   Rectangle{
   //anchors{bottom:root.bottom;left:root.left}
   width:100;height:100
   color:"red"
    MouseArea{
    anchors.fill:parent
     onClicked:{
     //root.obj.destroy();
     parent.destroy();
     }
    }
   }
  }
 }
}
```



Only Button

Clicking on Button instantiate Rectangle

Cicking on Red Rectangle destroys itself

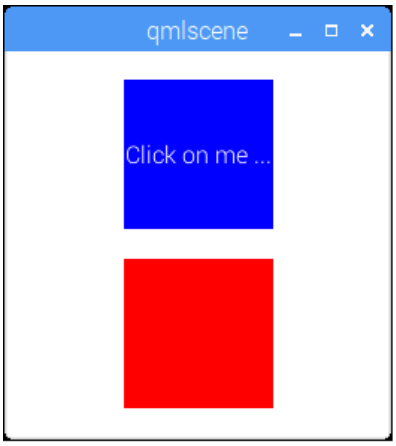
-Declarative creation

Like procedural creations a Loader type is used for creating object through a Component. if a Compoennt is a document file, i.e. ButtonC.qml, then source attribute can be specified or if Component is a type delcared in a document then sourceComponent attribute should be specified. Once Loader instantiate the Item, Item can be identifed through <Loader>.item,i.e. in Connections type. Like <ComponentObject>.createObject/<Component>.createObject, created Item object will have context of the Loader where it gets defined.

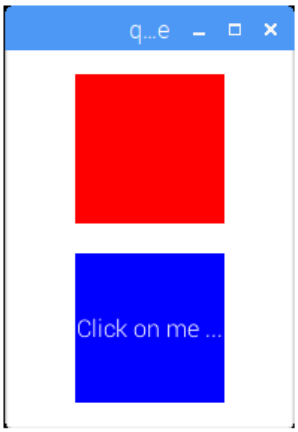
```
<ButtonC.qml>
import QtQuick 2.0
Rectangle {
id:button
property alias text: label.text
property alias textcolor:label.color
signal clicked()
color: "blue"
 Text {
 id: label
 anchors.centerIn: parent
 }
 MouseArea{
 anchors.fill: parent
 onClicked: button.clicked()
 }
}
```

```
<main.qml>
import QtQuick 2.0
Item{
id:root
//property var obj
 Column{
 id:columnid
 anchors.centerIn:parent
 spacing:20
  //Loader {
  //id:loaderid
  //}
  ButtonC{
  width:100;height:100
  //anchors{top:root.top;left:root.left}
  text:"Click on me ..."
  textcolor:"white"
   onClicked:{
   //root.obj=dialogComponent.createObject(root);
   //componentid.createObject(columnid);
   loaderid.sourceComponent=componentid
   }
  }
  Loader {
  id:loaderid
  }
  Component{
  id:componentid
   Rectangle{
   //anchors{bottom:root.bottom;left:root.left}
   width:100;height:100
   color:"red"
    MouseArea{
    id:mouseareaid
    anchors.fill:parent
     onClicked:{
     //root.obj.destroy();
     //parent.destroy();
     loaderid.sourceComponent=undefined
     }
    }
   }
```

```
  }
 }
}
```



Loader below ButtonC



Loader above ButtonC

Clicking on red sets sourceComponent undefined

Day 3 Morning

8. Dynamic Creation of Items

- Creating custom Item
- Creating Items Dynamically
  - Procedural Method
  - Procedural/Declarative Creation
  - Declarative Method
- Creating Multiple Items
- Repeaters

Multiple items generally gets created through type Model View Controller pattern.

Day 3 Morning

8. Dynamic Creation of Items

- Creating custom Item
- Creating Items Dynamically
  - Procedural Method
  - Procedural/Declarative Creation
  - Declarative Method
- Creating Multiple Items
- Repeaters

Repeater is a view which have delegate as repeating item and a datamodel as the model. Data model can be of all types.
-ListModel
-JSON Data
-property list<type>
-QList<QObject*>
-QAbstractItemModel

```
import QtQuick 2.7
Grid{
columns:3
rows:2
spacing:10
ListModel{
id:fruitcolormodelid
ListElement{ fruit:"mango";color:"green" }
ListElement{ fruit:"lichi";color:"pink" }
ListElement{ fruit:"apple";color:"green" }
ListElement{ fruit:"guava";color:"yellow" }
ListElement{ fruit:"cherry";color:"red" }
ListElement{ fruit:"grape";color:"black" }
}
Repeater{
//model:6
//model:["mango","lichi","apple","guava","cherry","grapes"]
//model:["yellow","green","red","blue","black","pink"]
model:fruitcolormodelid
Rectangle{
width:100;height:100
//color:modelData
color:model.color
Text{
text:model.fruit
color:"white"
anchors.centerIn:parent
}
}
```

```
}
}
```


mango
lichi
apple
guava
cherry
grape

Day 3 Morning

9. C++ Integration

- Declartive Envrionment
  -QQmlApplicationEngine
  -QQuickView
  - Exporting C++ Objects to QML
    -QQmlContext
  - Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting Qpainter based GUI Classes
    -QQuickPaintedItem
  - Building an application as a Library
  - Using Custom Types Plugins

Qml is a declarative language.It has two parts Qml and Javascript. Declarative part is qml and is the object initialization code with property name and value separated by colon ':'. It supports propery binding (LHS property modified when any RHS property changes).Javascript part is written in member functions and callback functions. Variable are assigned through '=' operator and so breaking the property binding. Binding or State Qml type can be used.

Qml part is interpreted through c++ classes (QQuickRectangle, QQuickText ...) where as Javascript part is interpreted through JIT interpreter.

```
import QtQuick 2.2
Item{
 Rectangle {
 id:toplevel      //<---- declarative
 color:"blue"
 height:500
 width:height
  Text{
  text:"Hello World"   //<---- declarative
  }
 }
 MouseArea{
  anchors.fill:parent
  onClicked:{
  toplevel.height=50;    //<---- javascript
  }
  onDoubleClicked:{
  toplevel.width=toplevel.height;
  }
  onHeightChanged:{
  toplevel.height=this.height;  //<------- breaking the binding
  }
 }
}
```

-QQmlApplicationEngine

```
          QObject
            / \
             -
             |
         QJSEngine
            / \
             -
         QQmlEngine
            / \
             -
             |
    QQmlApplicationEngine
```

Qml items are simple items of Graphics Views scene graph technology where all items belong to a scene and then scene can be drawn on one or more views/Widgets/Windows. Window or ApplicationWindow Quick type generates a window where Item (QQuickItem) gets finally drawn on it. When Window/ApplicationWindow is used on qml side then there is no need for QQuickView class on C++ side and just QQmlApplicationEngine works.

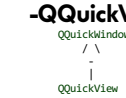
```
   Scene                      QQuickWindow/QQuickView
 ------------                 ------------------------
|          |  mapped to      |                        |
|   Item   | ---------------> |   QQuickItem          |
|          |                 |                        |
 ------------                 ------------------------
```

```
<main.qrc>
QT+=quick
TEMPLATE = app
TARGET = engine
INCLUDEPATH += .

MOC_DIR=build/moc
RCC_DIR=build/rcc
OBJECTS_DIR=build/obj
UI_DIR=build/uic

SOURCES += main.cpp
RESOURCES +=  main.qrc

<main.qrc>
<RCC>
 <qresource prefix="/">
   <file>main.qml</file>
 </qresource>
</RCC>
```

```
<main.cpp>
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QUrl>

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

```
<main.qml>
import QtQuick 2.0
import QtQuick.Window 2.2

Window {
visible:true
width:550
height:400
 Text{
 text:"Hello World"
 anchors.centerIn:parent
 }
}
```

-QQuickView



QQuickview provides window/widget to the top level Item in qml file. If qml file has Item as root element then cpp file must have QQuickView in order to show the root level Item.

```
<main.pro>
QT+=quick
TEMPLATE = app
TARGET = quickview
INCLUDEPATH += .

MOC_DIR=build/moc
RCC_DIR=build/rcc
OBJECTS_DIR=build/obj
UI_DIR=build/uic

# Input
SOURCES += main.cpp
RESOURCES +=
```

```
<main.qrc>
<RCC>
 <qresource prefix="/">
   <file>main.qml</file>
 </qresource>
</RCC>
```

```
<main.cpp>
#include <QGuiApplication>
#include <QQuickView>
#include <QUrl>

int main(int argc, char *argv[]) {
 QGuiApplication app(argc, argv);
 QQuickView* view = new QQuickView();
 QUrl source = QStringLiteral("qrc:/main.qml");
 view->setSource(source);
 view->show();
 return app.exec();
}
```

```
<main.qml>
import QtQuick 2.0

Rectangle{
width:400
height:300
color:"green"
 Text{
 text:"Hello World"
 anchors.centerIn:parent
 }
}
```

9. C++ Integration

- Declartive Envrionment
  -QQmlApplicationEngine
  -QQuickView
- Exporting C++ Objects to QML
  -QQmlContext
  - Exporting Classes to QML
  - Exporting Non-GUI Classes
  - Exporting Qpainter based GUI Classes
    -QQuickPaintedItem
  - Building an application as a Library
  - Using Custom Types Plugins

Qml document file has various context like root context root Item context and its children context and so on.. On c++ side QQmlcontext class refer a particular context on Qml side, i.e root context.
Contexts allow data to be exposed to the QML components instantiated by the QML engine.

```
QQmlEngine engine;
QStringListModel modelData;
QQmlContext *context = new QQmlContext(engine.rootContext());
context->setContextProperty("myModel", &modelData);

QQmlComponent component(&engine);
component.setData("import QtQuick 2.0
ListView { model: myModel }", QUrl());
QObject *window = component.create(context);
```

<main.cpp>

```
#include <QGuiApplication>
#include <QStringList>
#include <qqmlengine.h>
#include <qqmlcontext.h>
#include <qqml.h>
#include <QtQuick/qquickitem.h>
#include <QtQuick/qquickview.h>
int main(int argc, char ** argv) {
 QGuiApplication app(argc, argv);
 QStringList dataList;
 dataList.append("Item 1");
 dataList.append("Item 2");
 dataList.append("Item 3");
 dataList.append("Item 4");
 QQuickView view;
 QQmlContext *ctxt = view.rootContext();
 ctxt->setContextProperty("myModel", QVariant::fromValue(dataList));
 view.setSource(QUrl("qrc:view.qml"));
 view.show();
 return app.exec();
}
```

<view.qml>

```
import QtQuick 2.0
ListView {
width: 100; height: 100
model: myModel
 delegate: Rectangle {
 height: 25
 width: 100
  Text { text: modelData }
 }
}
```

main.pro

```
QT+=quick
TEMPLATE = app
TARGET = app2
INCLUDEPATH += .
SOURCES += main.cpp
RESOURCES += view.qrc
```

main.qrc

```
<RCC>
 <qresource>
  <file>view.qml</file>
 </qresource>
</RCC>
```

Day 3 Morning

9. C++ Integration

- Declartive Envrionment
  -QQmlApplicationEngine
  -QQuickView
- Exporting C++ Objects to QML
  -QQmlContext
- Exporting Classes to QML
- Exporting Non-GUI Classes
- Exporting Qpainter based GUI Classes
  -QQuickPaintedItem
- Building an application as a Library
- Using Custom Types Plugins

C++ classes declarare MetaObject properties and methods through macro Q_PROPERTY and Q_INVOKABLE. C++ classed is registered through `qmlRegisterType` to the qt meta object system in order to get is avaiable to qml side.

<main.qrc
```
<RCC>
 <qresource>
  <file>main.qml</file>
 </qresource>
</RCC>
```

<main.pro
```
QT+=quick
TEMPLATE = app
TARGET = binding
INCLUDEPATH += .
MOC_DIR=build
RCC_DIR=build
UI_DIR=build
OBJECTS_DIR=build
HEADERS += country.h river.h countryprint.h
SOURCES += country.cpp main.cpp river.cpp countryprint.cpp
RESOURCES += country.qrc
```

<country.h>
```
#ifndef COUNTRY_H
#define COUNTRY_H
#include <QObject>
#include <QDate>
#include <QDebug>
#include <qqml.h>
#include "river.h"
class CountryAttached : public QObject {
 Q_OBJECT
 Q_PROPERTY(QDate datadate READ datadate WRITE setDatadate NOTIFY datadateChanged)
public:
 CountryAttached(QObject *object);
 QDate datadate() const;
 void setDatadate(const QDate &);
signals:
 void datadateChanged();
private:
 QDate m_datadate;
};
class Country : public QObject {
 Q_OBJECT
 Q_PROPERTY(River *mainriver READ mainriver WRITE setMainriver NOTIFY mainriverChanged)
 Q_PROPERTY(QQmlListProperty<River> rivers READ rivers)
 Q_CLASSINFO("DefaultProperty", "rivers")
public:
 Country(QObject *parent = 0);
 River *mainriver() const;
 void setMainriver(River *);
 QQmlListProperty<River> rivers();
 int riverCount() const;
 River *river(int) const;
 static CountryAttached *qmlAttachedProperties(QObject *);
signals:
 void mainriverChanged();
 void announcementChanged();
private:
 River *m_mainriver;
 QList<River *> m_rivers;
};
QML_DECLARE_TYPEINFO(Country, QML_HAS_ATTACHED_PROPERTIES)
#endif
```

<country.cpp>
```
#include "country.h"
CountryAttached::CountryAttached(QObject *object) : QObject(object) { }
QDate CountryAttached::datadate() const {
 return m_datadate;
}
void CountryAttached::setDatadate(const QDate &d) {
 if (d != m_datadate) {
  m_datadate = d;
  emit datadateChanged();
 }
}
Country::Country(QObject *parent) : QObject(parent), m_mainriver(0) { }
River *Country::mainriver() const {
 return m_mainriver;
}
```

```
void Country::setMainriver(River *c) {
 if (c != m_mainriver){
  m_mainriver = c;
  emit mainriverChanged();
 }
}
QQmlListProperty<River> Country::rivers() {
 return QQmlListProperty<River>(this, m_rivers);
}
int Country::riverCount() const {
 return m_rivers.count();
}
River *Country::river(int index) const {
 return m_rivers.at(index);
}
CountryAttached *Country::qmlAttachedProperties(QObject *object) {
 return new CountryAttached(object);
}
```

```
<countryprinting.h>
#ifndef COUNTRYPRINTING_H
#define COUNTRYPRINTING_H
#include <QQmlPropertyValueSource>
#include <QQmlProperty>
#include <QStringList>
#include <river.h>
class CountryPrint : public QObject, public QQmlPropertyValueSource {
 Q_OBJECT
 Q_INTERFACES(QQmlPropertyValueSource)
public:
 CountryPrint(QObject *parent = 0);
 virtual void setTarget(const QQmlProperty &);
private slots:
 void advance();
private:
 QQmlProperty m_target;
};
#endif

<countryprint.cpp>
#include <QDebug>
#include <QTimer>
#include <typeinfo>
#include "country.h"
#include "river.h"
#include "countryprint.h"
CountryPrint::CountryPrint(QObject *parent) : QObject(parent) {
    QTimer *timer = new QTimer(this);
    QObject::connect(timer, &QTimer::timeout, this, &CountryPrint::advance);
    timer->start(5000);
}
void CountryPrint::setTarget(const QQmlProperty &p) {
    m_target = p;
}
void CountryPrint::advance() {
 QDate datadate;
 QObject *attached;
 River *river;
 river=m_target.object()->property(m_target.object()->metaObject()->property(m_target.object()->metaObject()-
>indexOfProperty("mainriver")).name()).value<River*>();
 attached= qmlAttachedPropertiesObject<Country>(river, false);
 if (attached) datadate = attached->property("datadate").toDate();
 if (datadate.isNull()) qWarning() <<endl<<"   " << river->name() << "No data date "<< river->about()->length()<<" "<<river->about()-
>states()<<" "<<river->about()->meet();
 else qDebug() <<endl<<"   " << river->name() << " "<< qPrintable(datadate.toString())<<river->about()->length()<<" "<<river->about()-
>states()<<" "<<river->about()->meet();
}
```

```
<main.cpp>
#include <QCoreApplication>
#include <QQmlEngine>
#include <QQmlComponent>
#include <QDebug>
#include "country.h"
#include "countryprint.h"
#include "river.h"

int main(int argc, char ** argv) {
 QCoreApplication app(argc, argv);
 qmlRegisterType<CountryAttached>();
 qmlRegisterType<Country>("Country", 1,0, "Country");
 qmlRegisterType<CountryPrint>("Country", 1,0, "CountryPrint");
 qmlRegisterType<RiverDescription>();
 qmlRegisterType<River>("Country", 1,0, "River");

 QQmlEngine engine;
 QQmlComponent component(&engine, QUrl("qrc:main.qml"));
 Country *country = qobject_cast<Country *>(component.create());
 if (country){
  for (int ii = 0; ii < country->riverCount(); ++ii) {
   River *river = country->river(ii);
```

```
   QDate datadate;
   QObject *attached = qmlAttachedPropertiesObject<Country>(river, false);
   if (attached) datadate = attached->property("datadate").toDate();
   qDebug()<<"------------"<<endl<<"River Name : "<<river->name()<<endl<<"datedate :"<<(datadate.isNull()?"NO
DATE":qPrintable(datadate.toString()))<<endl<<"Length : "<<river->about()->length()<<endl<<"States visiting : "<<river->about()->states()
<<endl<<"Meets : "<<river->about()->meet();
  }
 } else qWarning() << component.errors();
 return app.exec();
}
```

```
<river.h>
#ifndef RIVER_H
#define RIVER_H
#include <QObject>
#include <QStringList>
class RiverDescription : public QObject {
 Q_OBJECT
 Q_PROPERTY(qreal length READ length WRITE setLength NOTIFY lengthChanged)
 Q_PROPERTY(QStringList states READ states WRITE setStates NOTIFY statesChanged)
 Q_PROPERTY(QString meet READ meet WRITE setMeet NOTIFY meetChanged)
public:
 RiverDescription(QObject *parent = 0);
 qreal length() const;
 void setLength(qreal);
 QStringList& states();
 void setStates(QStringList&);
 QString state(int) const;
 int stateCount();
 QString meet() const;
 void setMeet(const QString &);
signals:
 void lengthChanged();
 void statesChanged();
 void meetChanged();
private:
 qreal m_length;
 QStringList m_states;
 QString m_meet;
};
class River : public QObject {
 Q_OBJECT
 Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
 Q_PROPERTY(RiverDescription *about READ about CONSTANT)
public:
 River(QObject *parent = 0);
 QString name() const;
 void setName(const QString &);
 RiverDescription *about();
signals:
 void nameChanged();
private:
 QString m_name;
 RiverDescription m_description;
};
#endif // RIVER_H

<river.cpp>
#include "river.h"
RiverDescription::RiverDescription(QObject *parent) : QObject(parent), m_length(0) { }
qreal RiverDescription::length() const {
 return m_length;
}
void RiverDescription::setLength(qreal s) {
 if (m_length!=s) {
  m_length = s;
  emit lengthChanged();
 }
}
QStringList& RiverDescription::states() {
 return m_states;
}
void RiverDescription::setStates(QStringList& l){
 if (m_states != l){
 m_states=l;
 emit statesChanged();
 }
}
QString RiverDescription::meet() const {
 return m_meet;
}
void RiverDescription::setMeet(const QString& p) {
 if (m_meet!=p){
  m_meet = p;
  emit meetChanged();
 }
}
QString RiverDescription::state(int i) const {
 return m_states.at(i);
}
RiverDescription *River::about() {
```

```
 return &m_description;
}
River::River(QObject *parent) : QObject(parent) { }
QString River::name() const {
 return m_name;
}
void River::setName(const QString &n) {
 if(m_name!=n) {
  m_name = n;
  emit nameChanged();
 }
}
```

```
<main.qml>
import Country 1.0
import QtQuick 2.0
Country {
 id: india
 CountryPrint on mainriver {}
 mainriver: River {
  id:indusriver
  name: "Indus"
  Country.datadate: "2009-07-06"
  about { length: 3180.0; states:["ladakh","punjab","sindh"]; meet: "sea"}
 }
 rivers:[River {
  name: "Cauvery"
  Country.datadate: "2010-04-26"
  about { length: 805.0; states:["karnataka","tamilnadu"]; meet: "sea"}
 },
 River {
  name: "Godavari"
  about { length: 1465.0; states:["maharashtra","telangana","ap"]; meet: "sea"}
 },
 River {
  name: "Ganga"
  Country.datadate: "2014-08-13"
  about { length: 2525.0; states:["uttrakhand","up","bihar","jharkhand","wb","bangladesh"]; meet: "sea"}
 },
 River {
  name: "Jhelum"
  about.length: 725.0
  about.states:["j&k","pok","punjab"]
  about.meet:indusriver.name
 }]
}
```

Output

```
$./binding
------------
River Name :  "Cauvery"
datedate : Mon Apr 26 2010
Length :  805
States visiting :  ("karnataka", "tamilnadu")
Meets :  "sea"
------------
River Name :  "Godavari"
datedate : NO DATE
Length :  1465
States visiting :  ("maharashtra", "telangana", "ap")
Meets :  "sea"
------------
River Name :  "Ganga"
datedate : Wed Aug 13 2014
Length :  2525
States visiting :  ("uttrakhand", "up", "bihar", "jharkhand", "wb", "bangladesh")
Meets :  "sea"
------------
River Name :  "Jhelum"
datedate : NO DATE
Length :  725
States visiting :  ("j&k", "pok", "punjab")
Meets :  "Indus"

    "Indus"   Mon Jul 6 2009 3180   ("ladakh", "punjab", "sindh")   "sea"

    "Indus"   Mon Jul 6 2009 3180   ("ladakh", "punjab", "sindh")   "sea"

    "Indus"   Mon Jul 6 2009 3180   ("ladakh", "punjab", "sindh")   "sea"
^C
```

Day 3 Morning

9. C++ Integration

- Declartive Envrionment
  -QQmlApplicationEngine
  -QQuickView
- Exporting C++ Objects to QML
  -QQmlContext
- Exporting Classes to QML
- Exporting Non-GUI Classes
- Exporting Qpainter based GUI Classes
  -QQuickPaintedItem
- Building an application as a Library
- Using Custom Types Plugins

Non Gui classess can be exported by subclassing QObject.

Day 3 Morning

9. C++ Integration

- Declartive Envrionment
  -QQmlApplicationEngine
  -QQuickView
- Exporting C++ Objects to QML
  -QQmlContext
- Exporting Classes to QML
- Exporting Non-GUI Classes
- Exporting Qpainter based GUI Classes
  -QQuickPaintedItem
- Building an application as a Library
- Using Custom Types Plugins

Gui classeses can be exported by subclassing/extending QQuickItem. If painting is required then QQuickPaintedItem needs to be subclassed and paint function needs to overrriden.

<main.pro>
```
QT+=quick
TEMPLATE = app
TARGET = ellipse
INCLUDEPATH += .

MOC_DIR=build
RCC_DIR=build
UI_DIR=build
OBJECTS_DIR=build

HEADERS += ellipse.h
SOURCES += ellipse.cpp main.cpp
RESOURCES += main.qrc
```

main.qrc
```
<RCC>
 <qresource>
  <file>main.qml</file>
 </qresource>
</RCC>
```

<ellipse.h>
```
#ifndef ELLIPSE_H
#define ELLIPSE_H
#include <QtQuick/QQuickPaintedItem>
#include <QColor>

class Ellipsesub:public QQuickPaintedItem {
 Q_OBJECT
 Q_PROPERTY(int angle READ angle WRITE setAngle)
 Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
 Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
public:
 Ellipsesub(QQuickItem *parent=0);
 int angle()const;
 void setAngle(const int);
 QColor color() const;
 void setColor(const QColor&);
 QString name() const;
 void setName(const QString&);
 void paint(QPainter *);
signals:
 void nameChanged();
 void colorChanged();
private:
 int _angle;
 QString _name;
 QColor _color;
};
class Ellipse : public QQuickPaintedItem {
 Q_OBJECT
 Q_PROPERTY(QQmlListProperty<Ellipsesub> ellipsesubs READ ellipsesubs)
// Q_PROPERTY(Ellipsesub *ellipsesub READ ellipsesub WRITE setEllipsesub)
 Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
 Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
public:
 Ellipse(QQuickItem *parent = 0);
 QString name() const;
 void setName(const QString &name);
 QColor color() const;
 void setColor(const QColor &color);
/* Ellipsesub* ellipsesub() const;
 void setEllipsesub(Ellipsesub*); */
 void paint(QPainter *painter);
 static void appendellipsesub(QQmlListProperty<Ellipsesub> *ellipsesublist,Ellipsesub *ellipsesub);

 QQmlListProperty<Ellipsesub> ellipsesubs();
 Q_INVOKABLE void clearEllipse();
signals:
 void nameChanged();
 void colorChanged();
 void ellipseCleared();
private:
 QString _name;
 QColor _color;
//Ellipsesub* _ellipsesub;
 QList<Ellipsesub*> _ellipsesublist;
};
#endif
```

<ellipse.cpp>

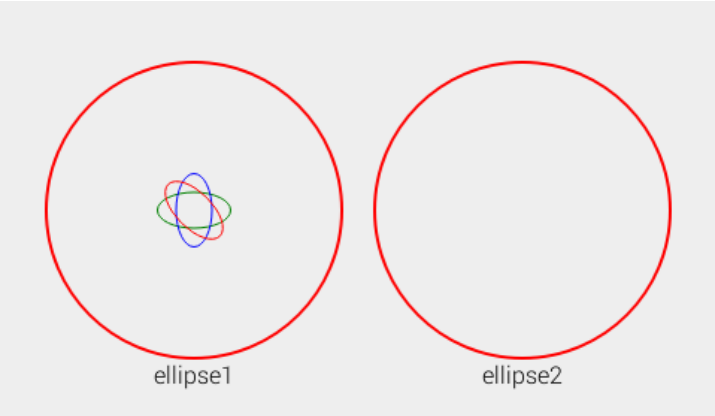
```
#include <QPainter>
#include "ellipse.h"
Ellipsesub::Ellipsesub(QQuickItem *parent):QQuickPaintedItem(parent),_angle(0){}
/*Ellipsesub* Ellipse::ellipsesub() const { return _ellipsesub; }
void Ellipse::setEllipsesub(Ellipsesub *ellipsesub){
_ellipsesub=ellipsesub;
_ellipsesub->setParentItem(this);
}*/
QString Ellipsesub::name() const { return _name; }
void Ellipsesub::setName(const QString& name){
 if (_name!=name){
  _name=name;
  emit nameChanged();
 }
}
QColor Ellipsesub::color() const { return _color; }
void Ellipsesub::setColor(const QColor& color){
 if(_color!=color){
  _color=color;
  update();
  emit colorChanged();
 }
}
int Ellipsesub::angle() const { return _angle; }
void Ellipsesub::setAngle(const int angle){
 if(_angle!=angle){
  _angle=angle;
//  emit angleChanged();
 }
}
void Ellipsesub::paint(QPainter *painter){
 int sidemin=qMin(width(),height());
 int sidemax=qMax(width(),height());
 painter->setViewport((width()-sidemin)/2,(height()-sidemin)/2,sidemin,sidemin);
 painter->setWindow(-sidemax/2,-sidemax/2,sidemax,sidemax);
 painter->setPen(QPen(_color,2));
 painter->setRenderHints(QPainter::Antialiasing,true);
 painter->save();
 painter->rotate(_angle);
 painter->drawEllipse(QRect(-width()/2,-height()/2,width(),height()).adjusted(1,1,-1,-1));
 painter->restore();
}

Ellipse::Ellipse(QQuickItem *parent):QQuickPaintedItem(parent){}
QString Ellipse::name() const { return _name; }
void Ellipse::setName(const QString& name){
 if (_name!=name){
  _name=name;
  emit nameChanged();
 }
}
QColor Ellipse::color() const { return _color; }
void Ellipse::setColor(const QColor& color){
 if(_color!=color){
  _color=color;
  update();
  emit colorChanged();
 }
}
void Ellipse::paint(QPainter *painter){
 painter->setPen(QPen(_color,2));
 painter->setRenderHints(QPainter::Antialiasing,true);
 painter->drawEllipse(boundingRect().adjusted(1,1,-1,-1));
}
void Ellipse::clearEllipse(){
 static QColor color=_color;
 if(_color!=Qt::transparent)
  setColor(QColor(Qt::transparent));
 else
  setColor(color);
 emit ellipseCleared();
}
QQmlListProperty<Ellipsesub> Ellipse::ellipsesubs(){
 return QQmlListProperty<Ellipsesub>(this,nullptr,&Ellipse::appendellipsesub,nullptr,nullptr,nullptr);
}
void Ellipse::appendellipsesub(QQmlListProperty<Ellipsesub> *ellipsesublist,Ellipsesub *ellipsesub){
 Ellipse *ellipse=qobject_cast<Ellipse*>(ellipsesublist->object);
 if(ellipse){
  ellipsesub->setParentItem(ellipse);
  ellipse->_ellipsesublist.append(ellipsesub);
 }
}
```

<main.cpp>

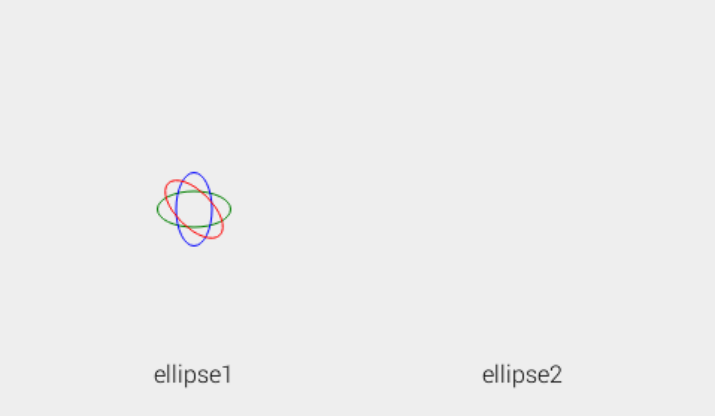
```
#include <QGuiApplication>
#include <QQuickView>
#include <QDebug>
#include "ellipse.h"
```

```
int main(int argc, char ** argv) {
 QGuiApplication app(argc, argv);
 qmlRegisterType<Ellipse>("EllipseM", 1,0, "Ellipse");
 qmlRegisterType<Ellipsesub>("EllipseM", 1,0, "Ellipsesub");
 QQuickView view;
 view.setResizeMode(QQuickView::SizeRootObjectToView);
 view.setSource(QUrl("qrc:main.qml"));
 view.show();
 return app.exec();
}
```

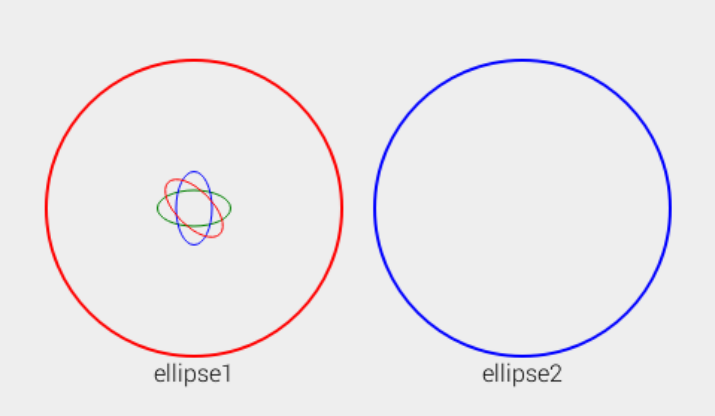
```
<main.qml>
import EllipseM 1.0
import QtQuick 2.0
Item{
width:400;height:400
 Row{
 anchors.centerIn:parent
 spacing:20
  Ellipse {
  id: ellipseid1
  width:200;height:200
  name:"ellipse1"
  color:"red"
  onEllipseCleared:{ console.log("Ellipse1 cleared"); }
  ellipsesubs:[Ellipsesub{
   width:100;height:50
   anchors.centerIn:parent
   color:"green"
   },
   Ellipsesub{
   width:100;height:50
   anchors.centerIn:parent
   color:"blue"
   angle:90
   },
   Ellipsesub{
   width:100;height:50
   anchors.centerIn:parent
   color:"red"
   angle:45
   }]
   Text{
   text:ellipseid1.name
   anchors{bottom:parent.bottom;horizontalCenter:parent.horizontalCenter;bottomMargin:-20}
   }
  }
  Ellipse {
  id: ellipseid2
  width:200;height:200
  name:"ellipse2"
  color:ellipseid1.color
  onEllipseCleared:console.log("Ellipse2 cleared")
   Text{
   text:ellipseid2.name
   anchors{bottom:parent.bottom;horizontalCenter:parent.horizontalCenter;bottomMargin:-20}
   }
  }
 }
 MouseArea{
 id:mousearea
 anchors{fill:parent}
 onClicked:ellipseid1.clearEllipse()
 onDoubleClicked:ellipseid2.color="blue";
 }
}
```



Ellipse and EllipseSubsets



Ellipse1 color become transparent and so Ellipse2 due to property binding



Ellipse2 breaks binding on mouse doubleclick

Day 3 Morning

9. C++ Integration

- Declartive Envrionment
  -QQmlApplicationEngine
  -QQuickView
- Exporting C++ Objects to QML
  -QQmlContext
- Exporting Classes to QML
- Exporting Non-GUI Classes
- Exporting Qpainter based GUI Classes
  -QQuickPaintedItem
- Building an application as a Library
- Using Custom Types Plugins

```
QQmlExtensionPlugin
      / \
       -
       |
       |
          ::registerTypes() //override
          ::initializeEngine() //override
```

Qml classes can be in application or can be pushed to a library to which application would link. In case of creating library, QQmlExtensionPlugin needs to be subclassed overriding its registerTypes and initializeEngine methods.

1. Subclass QQmlExtensionPlugin
  > Use the Q_PLUGIN_METADATA() macro to register the plugin with the Qt meta object system.
  > Override the registerTypes() method and call qmlRegisterType() to register the types to be exported by the plugin.
  > Override initializeEngine to initialize the extension from the uri using the engine. Here an application plugin may expose some data or objects to QML as context properties on the engine's root context.
2. Write a project file for the plugin.
3.Create a qmldir file to describe the plugin.

```
$ ls imports/EllipseM/
libqmlellipseplugin.so  qmldir
```

```
<*.pro>
QT+=qml quick
TEMPLATE = lib
CONFIG+=plugin
DESTDIR=imports/EllipseM
TARGET = qmlellipseplugin

INCLUDEPATH += .

HEADERS += ellipse.h ellipseplugin.h
SOURCES += ellipse.cpp ellipseplugin.cpp
```

```
<main.qrc>
<RCC>
 <qresource>
  <file>main.qml</file>
 </qresource>
</RCC>
```

```
<ellipseplugin.h>
#ifndef ELLIPSEPLUGIN_H
#define ELLIPSEPLUGIN_H
#include <QQmlExtensionPlugin>
class EllipsePlugin : public QQmlExtensionPlugin {
 Q_OBJECT
 Q_PLUGIN_METADATA(IID QQmlExtensionInterface_iid)
public:
 void registerTypes(const char *uri);
};
#endif
```

```
#include "ellipseplugin.h"
#include "ellipse.h"
#include <qqml.h>
void EllipsePlugin::registerTypes(const char *uri) {
 qmlRegisterType<Ellipse>(uri, 1, 0, "Ellipse");
 qmlRegisterType<Ellipsesub>(uri, 1, 0, "Ellipsesub");
}
```

Create a qmldir file to describe the plugin

```
<imports/EllipseM/qmldir>
module EllipseM
plugin qmlellipseplugin
```

```
<*.pro>
QT+=qml quick
TEMPLATE = lib
CONFIG+=plugin
DESTDIR=imports/EllipseM
TARGET = qmlellipseplugin

INCLUDEPATH += .

HEADERS += ellipse.h ellipseplugin.h
SOURCES += ellipse.cpp ellipseplugin.cpp

<ellipse.h>
#ifndef ELLIPSE_H
#define ELLIPSE_H
#include <QtQuick/QQuickPaintedItem>
#include <QColor>

class Ellipsesub:public QQuickPaintedItem {
 Q_OBJECT
 Q_PROPERTY(int angle READ angle WRITE setAngle)
 Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
 Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
public:
 Ellipsesub(QQuickItem *parent=0);
 int angle()const;
 void setAngle(const int);
 QColor color() const;
 void setColor(const QColor&);
 QString name() const;
 void setName(const QString&);
 void paint(QPainter *);
signals:
 void nameChanged();
 void colorChanged();
private:
 int _angle;
 QString _name;
 QColor _color;
};
class Ellipse : public QQuickPaintedItem {
 Q_OBJECT
 Q_PROPERTY(QQmlListProperty<Ellipsesub> ellipsesubs READ ellipsesubs)
// Q_PROPERTY(Ellipsesub *ellipsesub READ ellipsesub WRITE setEllipsesub)
 Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
 Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
public:
 Ellipse(QQuickItem *parent = 0);
 QString name() const;
 void setName(const QString &name);
 QColor color() const;
 void setColor(const QColor &color);
/* Ellipsesub* ellipsesub() const;
 void setEllipsesub(Ellipsesub*); */
 void paint(QPainter *painter);
 static void appendellipsesub(QQmlListProperty<Ellipsesub> *ellipsesublist,Ellipsesub *ellipsesub);

 QQmlListProperty<Ellipsesub> ellipsesubs();
 Q_INVOKABLE void clearEllipse();
signals:
 void nameChanged();
 void colorChanged();
 void ellipseCleared();
private:
 QString _name;
 QColor _color;
//Ellipsesub* _ellipsesub;
 QList<Ellipsesub*> _ellipsesublist;
};
#endif

<ellipse.cpp>
#include <QPainter>
#include "ellipse.h"
Ellipsesub::Ellipsesub(QQuickItem *parent):QQuickPaintedItem(parent),_angle(0){}
/*Ellipsesub* Ellipse::ellipsesub() const { return _ellipsesub; }
void Ellipse::setEllipsesub(Ellipsesub *ellipsesub){
_ellipsesub=ellipsesub;
_ellipsesub->setParentItem(this);
}*/
QString Ellipsesub::name() const { return _name; }
void Ellipsesub::setName(const QString& name){
 if (_name!=name){
  _name=name;
  emit nameChanged();
 }
}
QColor Ellipsesub::color() const { return _color; }
void Ellipsesub::setColor(const QColor& color){
 if(_color!=color){
  _color=color;
  update();
  emit colorChanged();
 }
}
int Ellipsesub::angle() const { return _angle; }
void Ellipsesub::setAngle(const int angle){
 if(_angle!=angle){
  _angle=angle;
//  emit angleChanged();
 }
}
void Ellipsesub::paint(QPainter *painter){
 int sidemin=qMin(width(),height());
 int sidemax=qMax(width(),height());
 painter->setViewport((width()-sidemin)/2,(height()-sidemin)/2,sidemin,sidemin);
 painter->setWindow(-sidemax/2,-sidemax/2,sidemax,sidemax);
 painter->setPen(QPen(_color,2));
 painter->setRenderHints(QPainter::Antialiasing,true);
 painter->save();
 painter->rotate(_angle);
 painter->drawEllipse(QRect(-width()/2,-height()/2,width(),height()).adjusted(1,1,-1,-1));
 painter->restore();
}

Ellipse::Ellipse(QQuickItem *parent):QQuickPaintedItem(parent){}
QString Ellipse::name() const { return _name; }
void Ellipse::setName(const QString& name){
 if (_name!=name){
  _name=name;
  emit nameChanged();
 }
}
QColor Ellipse::color() const { return _color; }
void Ellipse::setColor(const QColor& color){
 if(_color!=color){
  _color=color;
  update();
  emit colorChanged();
 }
}
void Ellipse::paint(QPainter *painter){
 painter->setPen(QPen(_color,2));
 painter->setRenderHints(QPainter::Antialiasing,true);
 painter->drawEllipse(boundingRect().adjusted(1,1,-1,-1));
}
void Ellipse::clearEllipse(){
 static QColor color=_color;
 if(_color!=Qt::transparent)
  setColor(QColor(Qt::transparent));
 else
  setColor(color);
 emit ellipseCleared();
}
QQmlListProperty<Ellipsesub> Ellipse::ellipsesubs(){
 return QQmlListProperty<Ellipsesub>(this,nullptr,&Ellipse::appendellipsesub,nullptr,nullptr,nullptr);
}
void Ellipse::appendellipsesub(QQmlListProperty<Ellipsesub> *ellipsesublist,Ellipsesub *ellipsesub){
 Ellipse *ellipse=qobject_cast<Ellipse*>(ellipsesublist->object);
 if(ellipse){
  ellipsesub->setParentItem(ellipse);
  ellipse->_ellipsesublist.append(ellipsesub);
 }
}

<ellipseplugin.h>
#ifndef ELLIPSEPLUGIN_H
#define ELLIPSEPLUGIN_H
#include <QQmlExtensionPlugin>
class EllipsePlugin : public QQmlExtensionPlugin {
 Q_OBJECT
 Q_PLUGIN_METADATA(IID QQmlExtensionInterface_iid)
public:
 void registerTypes(const char *uri);
};
#endif

<ellipseplugin.cpp>
#include "ellipseplugin.h"
#include "ellipse.h"
#include <qqml.h>
void EllipsePlugin::registerTypes(const char *uri) {
 qmlRegisterType<Ellipse>(uri, 1, 0, "Ellipse");
 qmlRegisterType<Ellipsesub>(uri, 1, 0, "Ellipsesub");
}

<imports/EllipseM/qmldir>
module EllipseM
plugin qmlellipseplugin
```

9. C++ Integration

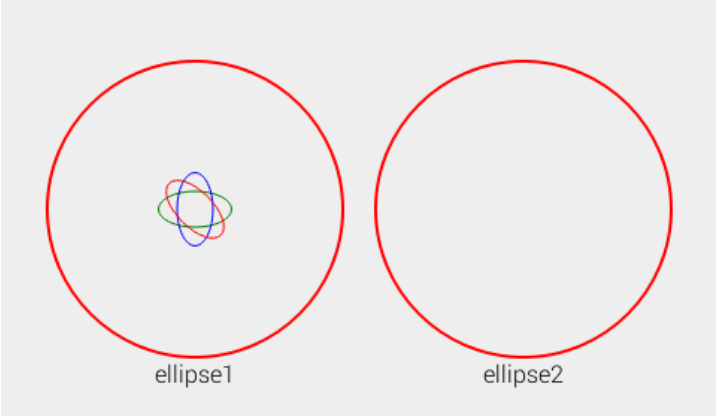
- Declartive Envrionment
  -QQmlApplicationEngine
  -QQuickView
- Exporting C++ Objects to QML
  -QQmlContext
- Exporting Classes to QML
- Exporting Non-GUI Classes
- Exporting Qpainter based GUI Classes
  -QQuickPaintedItem
- Building an application as a Library
- Using Custom Types Plugins

Application needs to add module directory which contains qmldir file i.e imports/EllipseM. Here EllipseM is module name so only 'imports' directory needs to be added to QQmlEngine::addImportPath. It is expected that plugin library is available in the directory where qmldir file exists, otherwise plugin path needs to be added in QQmlEngine::addPluginPath.

```
<*.pro>
QT+=quick
TEMPLATE = app
TARGET = ellipse
INCLUDEPATH += .

MOC_DIR=build
RCC_DIR=build
UI_DIR=build
OBJECTS_DIR=build

SOURCES += main.cpp
RESOURCES += main.qrc
```

```
<main.cpp>
#include <QGuiApplication>
#include <QQuickView>
#include <QQmlEngine>
#include <QDebug>

int main(int argc, char ** argv) {
 QGuiApplication app(argc, argv);
 QQuickView view;
 view.engine()->addImportPath(app.applicationDirPath()+"/imports");
 view.setResizeMode(QQuickView::SizeRootObjectToView);
 view.setSource(QUrl("qrc:main.qml"));
 view.show();
 return app.exec();
}
```

```
$ ls imports/
EllipseM
$ ls imports/EllipseM/
libqmlellipseplugin.so  qmldir
$ cat imports/EllipseM/qmldir
module EllipseM
plugin qmlellipseplugin
$./ellipse
```



Ellipse and EllipseSubsets

```
<*.pro>
QT+=quick
TEMPLATE = app
TARGET = ellipse
INCLUDEPATH += .

MOC_DIR=build
RCC_DIR=build
UI_DIR=build
OBJECTS_DIR=build

SOURCES += main.cpp
RESOURCES += main.qrc

<main.qrc>
<RCC>
 <qresource>
  <file>main.qml</file>
 </qresource>
</RCC>

<imports/EllipseM/qmldir>
module EllipseM
plugin qmlellipseplugin
```

```
<main.cpp>
#include <QGuiApplication>
#include <QQuickView>
#include <QQmlEngine>
#include <QDebug>

int main(int argc, char ** argv) {
 QGuiApplication app(argc, argv);
 QQuickView view;
 view.engine()->addImportPath(app.applicationDirPath()+"/imports");
 view.setResizeMode(QQuickView::SizeRootObjectToView);
 view.setSource(QUrl("qrc:main.qml"));
 view.show();
 return app.exec();
}

<main.qml>
import EllipseM 1.0
import QtQuick 2.0
Item{
width:400;height:400
 Row{
  anchors.centerIn:parent
```

```
spacing:20
 Ellipse {
 id: ellipseid1
 width:200;height:200
 name:"ellipse1"
 color:"red"
 onEllipseCleared:{ console.log("Ellipse1 cleared"); }
 ellipsesubs:[Ellipsesub{
  width:100;height:50
  anchors.centerIn:parent
  color:"green"
  },
  Ellipsesub{
  width:100;height:50
  anchors.centerIn:parent
  color:"blue"
  angle:90
  },
  Ellipsesub{
  width:100;height:50
  anchors.centerIn:parent
  color:"red"
  angle:45
  }]
  Text{
   text:ellipseid1.name
   anchors{bottom:parent.bottom;horizontalCenter:parent.horizontalCenter;bottomMargin:-20}
   }
 }
 Ellipse {
 id: ellipseid2
 width:200;height:200
 name:"ellipse2"
 color:ellipseid1.color
 onEllipseCleared:console.log("Ellipse2 cleared")
  Text{
  text:ellipseid2.name
  anchors{bottom:parent.bottom;horizontalCenter:parent.horizontalCenter;bottomMargin:-20}
  }
 }
}
MouseArea{
id:mousearea
anchors{fill:parent}
onClicked:ellipseid1.clearEllipse()
onDoubleClicked:ellipseid2.color="blue";
}
}
```

Day 3 Morning

10. Graphical Effects

- Canvas
  - Particles
  - Shaders

The Canvas item allows drawing of straight and curved lines, simple and complex shapes, graphs, and referenced graphic images. It can also add text, colors, shadows, gradients, and patterns, and do low level pixel operations.
Rendering to the Canvas is done using a Context2D object, usually as a result of the paint signal.

```
import QtQuick 2.0
Canvas {
id: mycanvas
width: 100
height: 200
 onPaint: {
 var ctx = getContext("2d");
 ctx.fillStyle = Qt.rgba(1, 0, 0, 1);
 ctx.fillRect(0, 0, width, height);
 }
}
```

Day 3 Morning

10. Graphical Effects

- Canvas
- Particles
  - Shaders

A particle system consists of three things
A ParticleSystem - which binds all elements to a simulation
An Emitter which emits particles into the system
A ParticlePainterderived element, which visualizes the particles

```
import QtQuick 2.0
import QtQuick.Particles 2.0

Rectangle {
id:root
width:480;height:160
color:"#1f1f1f"
ParticleSystem{
id:particlesystem
}
Emitter{
id:emitter
anchors.centerIn:parent
width:160;height:80
system:particlesystem
emitRate:10
lifeSpan:1000
lifeSpanVariation:500
size:16
endSize:32
}
ImageParticle{
source:"myparticle.png"
system:particlesystem
}
}
```

Day 3 Morning

10. Graphical Effects

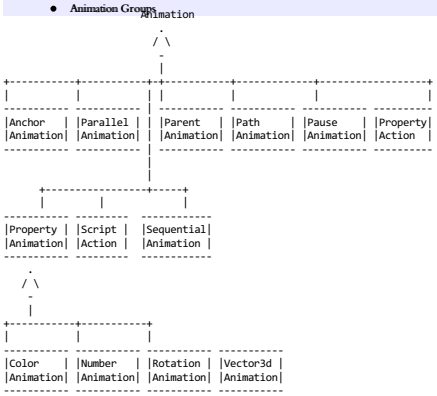
- Canvas
  - Particles
- Shaders

Day 3 Afternoon

11. Animations

- Animations
  - PropertyAnimation
  - ParallelAnimation
  - SequentialAnimation
  - PathAnimation

- Behaviour on
- Animation on
- StandAlone Animation
- Easing Curves
- Animation Groups



Various animations
Transition - Animates transitions during state changes
SequentialAnimation - Runs animations sequentially
ParallelAnimation - Runs animations in parallel
Behavior - Specifies a default animation for property changes
PropertyAction - Sets immediate property changes during animation
PauseAnimation - Introduces a pause in an animation
SmoothedAnimation - Allows a property to smoothly track a value
SpringAnimation - Allows a property to track a value in a spring-like motion
ScriptAction - Runs scripts during an animation

Types that animate properties based on data types
AnchorAnimation Animates changes in anchor values
ColorAnimation Animates changes in color values
NumberAnimation Animates changes in qreal-type values
ParentAnimation Animates changes in parent values
PathAnimation Animates an item along a path
PropertyAnimation Animates changes in property values
RotationAnimation Animates changes in rotation values
Vector3dAnimation Animates changes in QVector3d values

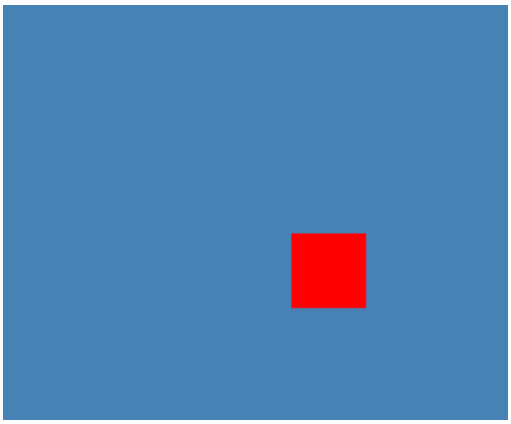
Properties	Signals	Methods
alwaysRunToEnd:bool	started()	complete()
loops:int	stopped()	pause()
paused:bool		restart()
running:bool		resume()
		start()
		stop()

-PropertyAnimation
PropertyAnimation provides a way to animate changes to a property's value. It can be used to define animations in a number of ways:

In a Transition

<main.qml
```
import QtQuick 2.0
Item{
id:itemid
width:200;height:200
 Rectangle {
 id: rect
 width: 50; height: 50
 color: "red"
  states: [
   State {
   name: "moved"
   PropertyChanges { target: rect; x:itemid.width-this.width;y:itemid.height-this.height }
```

```
   },
   State{
   name:""
   PropertyChanges{target:rect;x:0;y:0}
   }]
  transitions: Transition {
  PropertyAnimation { properties: "x,y"; easing.type: Easing.InOutQuad;duration:5000 }
  }
 }
focus:true
 Keys.onSpacePressed:rect.state="moved"
 Keys.onTabPressed:rect.state=""
}
```



In a Behavior

```
Rectangle {
width: 100; height: 100
color: "red"
 Behavior on x { PropertyAnimation {} }
 MouseArea { anchors.fill: parent; onClicked: parent.x = 50 }
}
```

As a property value source

For example, to repeatedly animate the rectangle's x property:

```
Rectangle {
width: 100; height: 100
color: "red"
 SequentialAnimation on x {
 loops: Animation.Infinite
 PropertyAnimation { to: 50 }
 PropertyAnimation { to: 0 }
 }
}
```

In a signal handler

```
MouseArea {
 anchors.fill: theObject
 onClicked: PropertyAnimation { target: theObject; property: "opacity"; to: 0 }
}
```

Standalone

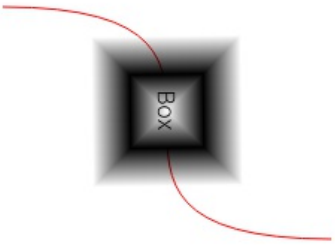
```
import QtQuick 2.0
Rectangle {
id: theRect
width: 100; height: 100
color: "red"
 // this is a standalone animation, it's not running by default
 PropertyAnimation { id: animation; target: theRect; property: "width"; to: 30; duration: 500 }
// MouseArea { anchors.fill: parent; onClicked: animation.running = true }
 MouseArea { anchors.fill: parent; onClicked: animation.start() }
}
```

-PathAnimation

pathanimation does animation on a set of pixels defined by Path Class.

```
<main.qml>
import QtQuick 2.0
Rectangle {
id: window
width: 320;height: 480
 Canvas {
 id: canvas
 anchors.fill: parent
 antialiasing: true
  onPaint: {
  var context = canvas.getContext("2d")
  context.clearRect(0, 0, width, height)
```

```
 context.strokeStyle = "red"
 context.path = pathAnim.path
 context.stroke()
 }
}
SequentialAnimation {
running: true
loops: -1
PauseAnimation { duration: 1000 }
 PathAnimation {
 id: pathAnim
 duration: 8000
 easing.type: Easing.InQuad
 target: box
 orientation: PathAnimation.RightFirst
 anchorPoint: Qt.point(box.width/2, box.height/2)
  path: Path {
  startX: 50; startY: 50
   PathCubic {
   x: window.width - 50
   y: window.height - 50
   control1X: x; control1Y: 50
   control2X: 50; control2Y: y
   }
  onChanged: canvas.requestPaint();
  }
 }
}
Rectangle {
 id: box
 x: 25; y: 25
 width: 50; height: 50
 border.width: 1
 antialiasing: true
 Text {
  anchors.centerIn: parent
  text: "Box"
 }
}
}
```



Day 3 Afternoon

11. Animations

- Animations
  - PropertyAnimation
  - ParallelAnimation
  - SequentialAnimation
  - PathAnimation
- Behaviour on
- Animation on
- StandAlone Animation
  - Easing Curves
  - Animation Groups

Default Animation as Behaviors

```
Rectangle {
width: 75; height: 75; radius: width
id: ball
color: "salmon"
 Behavior on x {
  NumberAnimation {
  id: bouncebehavior
   easing {
   type: Easing.OutElastic
   amplitude: 1.0
   period: 0.5
   }
  }
 }
 Behavior on y {
```

```
  animation: bouncebehavior
 }
 Behavior {
  ColorAnimation { target: ball; duration: 100 }
 }
}
```

Day 3 Afternoon

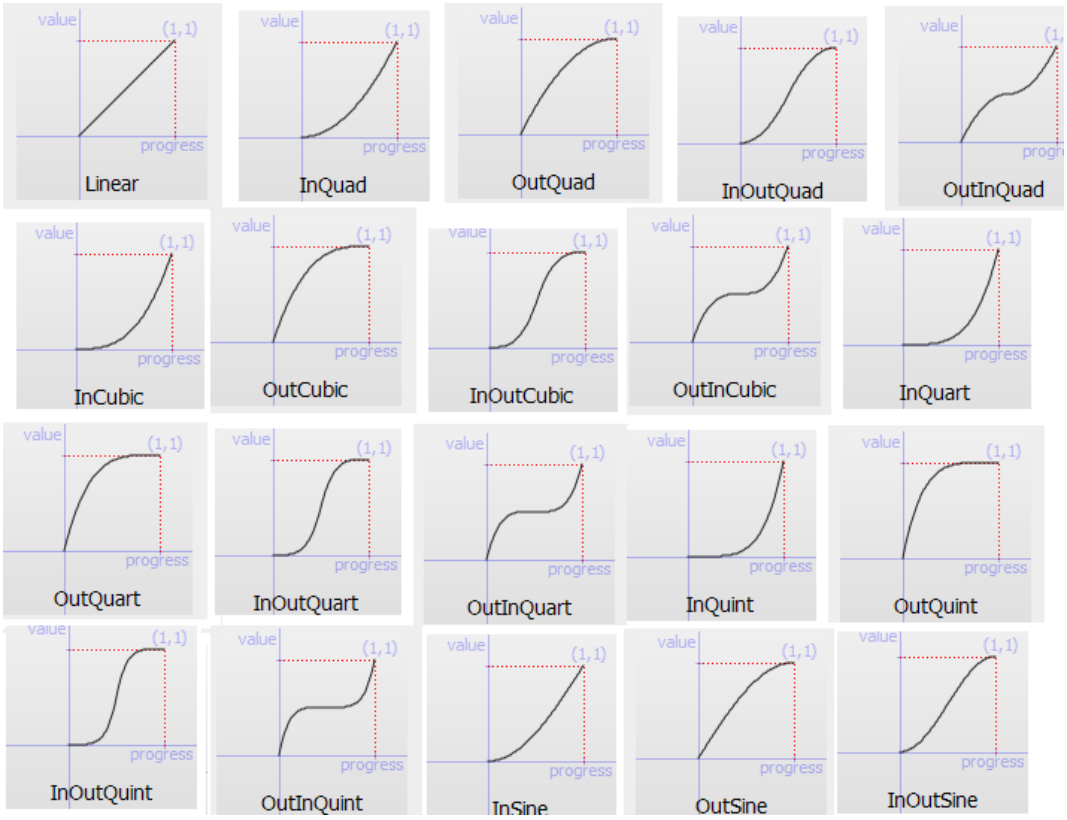
11. Animations

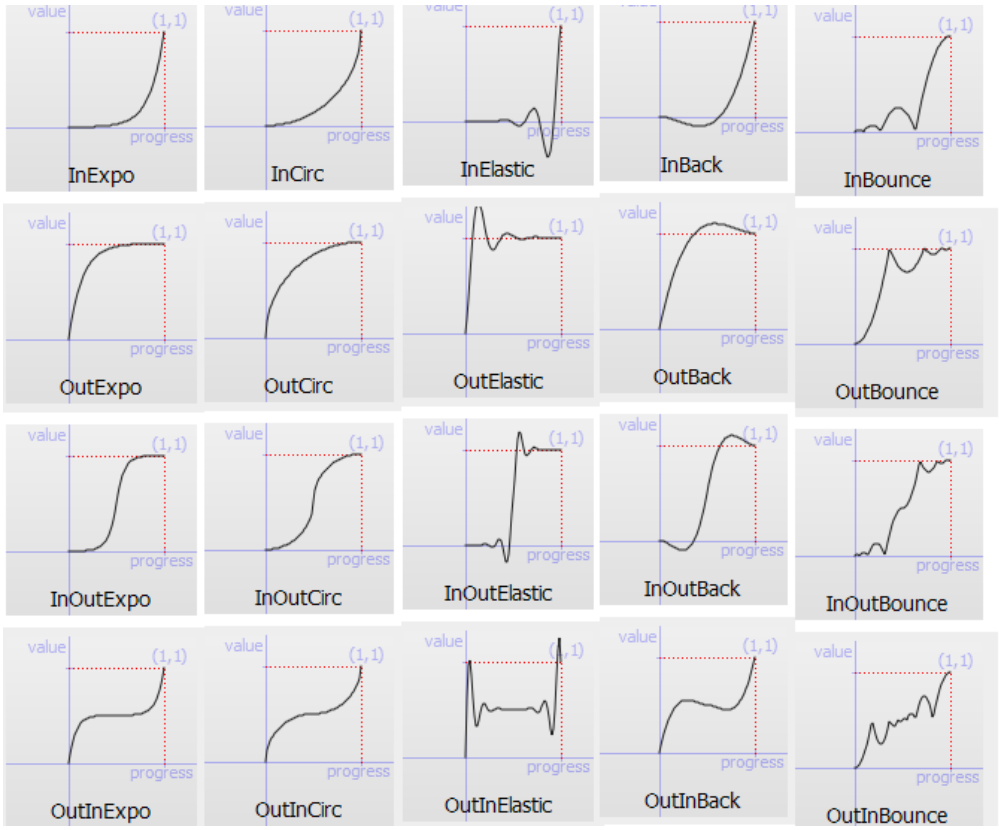
- Animations
  - PropertyAnimation
  - ParallelAnimation
  - SequentialAnimation
  - PathAnimation
- Behaviour on
- Animation on
- StandAlone Animation
- Easing Curves
- Animation Groups

Easing

Easing curves define how the animation will interpolate between the start value and the end value. Different easing curves might go beyond the defined range of interpolation. The easing curves simplify the creation of animation effects such as bounce effects, acceleration, deceleration, and cyclical animations,i.e Linear, Quad. For some curves,i.e. Easing.InOutElastic, amplitude, period and/or overshoot should also be mentioned.

```
PropertyAnimation { properties: "y";
 easing.type: Easing.InOutElastic;
 easing.amplitude: 2.0;
 easing.period: 1.5 }
```





Day 3 Afternoon

11. Animations

- Animations
  - PropertyAnimation
  - ParallelAnimation
  - SequentialAnimation
  - PathAnimation
- Behaviour on
- Animation on
- StandAlone Animation
- Easing Curves
- Animation Groups

There are two animation groups. SequentialAnimation and ParallelAnimation. They group contained animation sequentially and parallaly respectively.

```
import QtQuick 2.0
Rectangle {
width: 100; height: 100
color: "red"
 MouseArea{
 anchors.fill:parent
 onClicked:seqid.start()
 }
 SequentialAnimation on color {
 id:seqid
 ColorAnimation { to: "yellow"; duration: 1000 }
 ColorAnimation { to: "blue"; duration: 1000 }
 }
}
```

Once individual animations are placed into a SequentialAnimation or ParallelAnimation, they can no longer be started and stopped independently. The sequential or parallel animation must be started and stopped as a group.

Sharing Animation Instances

Sharing animation instances between Transitions or Behaviors is not supported, and may lead to undefined behavior.

```
Rectangle {
// NOT SUPPORTED: this will not work correctly as both Behaviors
// try to control a single animation instance
NumberAnimation { id: anim; duration: 300; easing.type: Easing.InBack }
Behavior on x { animation: anim }
Behavior on y { animation: anim }
}
```

Solution

```
// MyNumberAnimation.qml
NumberAnimation { id: anim; duration: 300; easing.type: Easing.InBack }
// main.qml
Rectangle {
```

```
Behavior on x { MyNumberAnimation {} }
Behavior on y { MyNumberAnimation {} }
}
```

Day 3 Afternoon

13. Presenting Data

- Data Item
  - Arranging Items
  - Positioners
  - Repeaters
- Scene Graph
  - Data Models
  - Using Views
  - Using Delegates
- Implement List data model view through delegate

Arranging Items

Positioners and repeaters make it easier to work with many items.

-Item Positioners

Positioner items are container items that manage the positions of items in a declarative user interface. Positioners behave in a similar way to the layout managers used with standard Qt widgets, except that they are also containers(Item subclassed) in their own right.

Set of Positioners

LayoutMirroring Property used to mirror layout behavior

Column Positions its children in a column

Flow Positions its children side by side, wrapping as necessary

Grid Positions its children in grid formation

Positioner Provides attached properties that contain details on where an item exists in a positioner

Row Positions its children in a row

LayoutMirroring and Positioner are attached property used to provide mirroring and give special information to Positioners Item containers,i.e. Rectangle, respectively.

LayoutMirroring

The LayoutMirroring attached property is used to horizontally mirror Item anchors, positioner types (such as Row and Grid) and views (such as GridView and horizontal ListView). Mirroring is a visual change: left anchors become right anchors, and positioner types like Grid and Row reverse the horizontal layout of child items.

Properties

```
childrenInherit : bool
enabled : bool
```

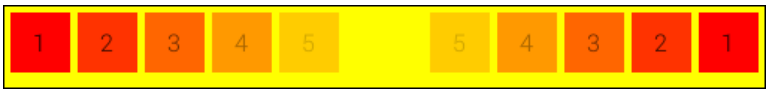
Positioner

An object of type Positioner is attached to the top-level child item within a Column, Row, Flow or Grid. It provides properties that allow a child item to determine where it exists within the layout of its parent Column, Row, Flow or Grid

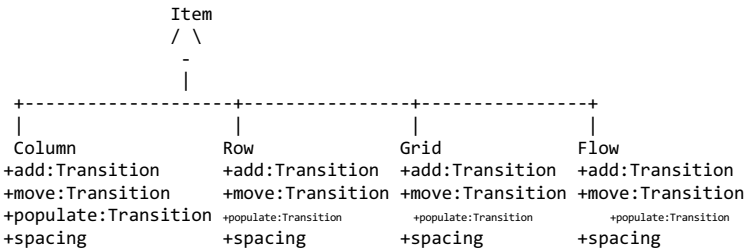
Properties

```
index : int
isFirstItem : bool
isLastItem : bool
```

```
<main.qml>
import QtQuick 2.0
Rectangle {
//LayoutMirroring.enabled: true
//LayoutMirroring.childrenInherit: true
color: "yellow"
border.width: 1
 Row {
// LayoutMirroring.enabled: true
// LayoutMirroring.childrenInherit: true
 anchors { left: parent.left; margins: 5 }
 y: 5; spacing: 5
  Repeater {
  model: 5
   Rectangle {
   color: "red"
   opacity: (5 - index) / 5
   width: 40; height: 40
    Text {
     text: parent.Positioner.index + 1
     anchors.centerIn: parent
    }
   }
  }
 }
 Row {
 LayoutMirroring.enabled: true    //LayoutMirroring Attached property
 LayoutMirroring.childrenInherit: true
 anchors { left: parent.left; margins: 5 }
 y: 5; spacing: 5
  Repeater {
  model: 5
   Rectangle {
   color: "red"
   opacity: (5 - index) / 5
   width: 40; height: 40
    Text {
     text: parent.Positioner.index + 1 //Positioner Attached property
     anchors.centerIn: parent
    }
   }
  }
 }
}
```



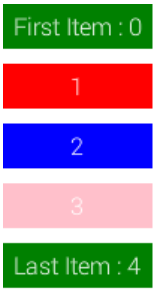
Column, Row, Grid and Flow are Item Positioners type which is subclass of Item type contains other Items



Column

Column is a type that positions its child items along a single column. It can be used as a convenient way to vertically position a series of items without using anchors.

```
<main.qml>
import QtQuick 2.0
Column{
spacing:10
 Repeater{
 model:["green","red","blue","pink","green"]
  Rectangle{
  width:100;height:30
  color:modelData
   Text{
   text:((parent.Positioner.isFirstItem)?"First Item : ":((parent.Positioner.isLastItem)?"Last Item : ":""))+parent.Positioner.index
   color:"white"
   anchors.centerIn:parent
   }
  }
 }
}
```



Flow

The Flow item positions its child items like words on a page, wrapping them to create rows or columns of items.

```
<main.qml>
import QtQuick 2.7
Item {
width: 600
height: 600
 Component {
 id: element
  Rectangle {
  width: Math.round(Math.random() * 100) + 50
  height: Math.round(Math.random() * 100) + 50
  color: Qt.rgba(Math.random(),Math.random(),Math.random(),1)
  }
 }
 Flow {
 id: flow
 spacing: 2
 anchors.fill: parent
  add: Transition {
   NumberAnimation { properties: "x,y"; easing.type: Easing.OutBack }
  }
 move: add
```

```
 }
/* Timer {
 id: timer
 property bool is_add: true
 interval: 1000
 repeat: true
 running: true
 onTriggered: {
  if(timer.is_add) {
   element.createObject(flow);
   if(flow.children.length > 20) timer.is_add = false;
  } else {
   flow.children[0].destroy();
   if(flow.children.length <= 1) timer.is_add = true;
  }
 }
}*/
focus:true
Keys.onSpacePressed:element.createObject(flow);
Keys.onEscapePressed:flow.children[0].destroy();
}
```

space keypress adds to it and escape delete at position 0

-Repeater

The Repeater type is used to create a large number of similar items. Like other view types, a Repeater has a model and a delegate: for each entry in the model, the delegate is instantiated in a context seeded with data from the model. A Repeater item is usually enclosed in a positioner type such as Row or Column to visually position the multiple delegate items created by the Repeater.

Properties

```
count : int
delegate : Component
model : any
```

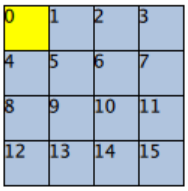
Signals

```
itemAdded(int index, Item item)
itemRemoved(int index, Item item)
```

Methods

```
Item itemAt(index)
```

```
Grid {   <-- Positioner Type
  Repeater {
  model: 16   <-------- Model
  Rectangle {   <------- Delegate
   id: rect
   width: 30; height: 30
   border.width: 1
   color: Positioner.isFirstItem ? "yellow" : "lightsteelblue"
   Text { text: rect.Positioner.index }
  }
 }
}
```



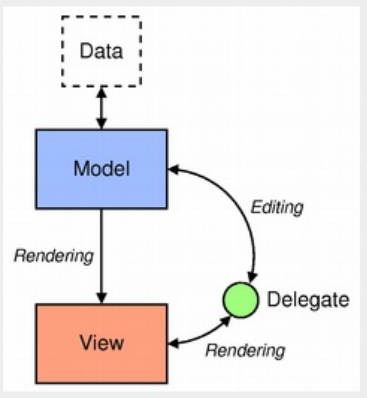
Day 3 Afternoon

13. Presenting Data

- Data Item
  - Arranging Items
    - Positioners
    - Repeaters
- Scene Graph
  - Data Models
  - Using Views
  - Using Delegates
- Implement List data model view through delegate

Data Models

Models and views provide a way to handle data sets
- Models hold data or items
- Views display data or items using delegates



Data is provided to the delegate via named data roles which the delegate may bind to

Mainly of following kind of models.
- ListModel
- XmlListModel
- VisualItemModel
- ObjectModel
- Integer as Model
- Object Instances as Models
- Repeaters

- ListModel

ListModel is a simple hierarchy of types specified in QML. The available roles are specified by the ListElement properties.

```
ListModel {
 id: fruitModel
 ListElement { name: "Apple";cost: 2.45 }
 ListElement { name: "Orange";cost: 3.25 }
 ListElement { name: "Banana";cost: 1.95 }
}
```

The above model has two roles, name and cost. These can be bound to by a ListView delegate, for example:

```
ListView {
 anchors.fill: parent
 model: fruitModel
 delegate: Row {
  Text { text: "Fruit: " + name }
  Text { text: "Cost: $" + cost }
 }
MouseArea {
 anchors.fill: parent
 onClicked: fruitModel.append({"cost": 5.95, "name":"Pizza"})
}
}
```

fruitModel can be appended through javascript code

- XmlListModel

XmlListModel allows construction of a model from an XML data source. The roles are specified via the XmlRole type. The type needs to be imported.

```
import QtQuick.XmlListModel 2.0
```

The following model has three roles, title, link and description:

```
XmlListModel {
 id: feedModel
 source: "http://rss.news.yahoo.com/rss/oceania"
 query: "/rss/channel/item"
 XmlRole { name: "title"; query: "title/string()" }
 XmlRole { name: "link"; query: "link/string()" }
```

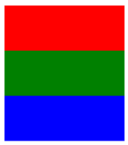
```
 XmlRole { name: "description"; query: "description/string()" }
}
```

- ObjectModel

A ObjectModel contains the visual items to be used in a view. When a ObjectModel is used in a view, the view does not require a delegate since the ObjectModel already contains the visual delegate (items).

An item can determine its index within the model via the index attached property.

```
import QtQuick 2.0
import QtQml.Models 2.1
Rectangle {
 ObjectModel {
  id: itemModel
  Rectangle { height: 30; width: 80; color: "red" }
  Rectangle { height: 30; width: 80; color: "green" }
  Rectangle { height: 30; width: 80; color: "blue" }
 }
 ListView {
  anchors.fill: parent
  model: itemModel
 }
}
```



-Integer as Model

An integer can be used as a model that contains a certain number of types. In this case, the model does not have any data roles.

The following example creates a ListView with five elements:

```
Item {
 width: 200; height: 250
 Component {
  id: itemDelegate
  Text { text: "I am item number: " + index }
 }
 ListView {
  anchors.fill: parent
  model: 5
  delegate: itemDelegate
 }
}
```

- Object Instances as Models

An object instance can be used to specify a model with a single object type. The properties of the object are provided as roles.

```
Rectangle {
 width: 200; height: 250
 Text {
  id: myText
  text: "Hello"
  color: "#dd44ee"
 }
 Component {
  id: myDelegate
  Text { text: model.color }  <--- Explicitly calling model.color
 }
 ListView {
  anchors.fill: parent
  anchors.topMargin: 30
  model: myText
  delegate: myDelegate
 }
}
```

-Repeaters

Repeaters create items from a template for use with positioners, using data from a model. Combining repeaters and positioners is an easy way to lay out lots of items.

A Repeater item is placed inside a positioner, and generates items that the enclosing positioner arranges. Each Repeater creates a number of items by combining each element of data from a model, specified using the model property.

```
import QtQuick 2.0
Rectangle {
 width: 400; height: 400; color: "black"
 Grid {
  x: 5; y: 5
  rows: 5; columns: 5; spacing: 10
  Repeater { model: 24
   Rectangle { width: 70; height: 70
    color: "lightgreen"
```

```
    Text { text: index
     font.pointSize: 30
     anchors.centerIn: parent } }
  }
 }
}
```

- Using Views

Views are containers for collections of items. They are feature-rich and can be customizable to meet style or behavior requirements.
Views allow visual customization through decoration properties such as the header, footer, and section properties. By binding an object, usually another visual object, to these properties, the views are decoratable. A footer may include a Rectangle type showcasing borders or a header that displays a logo on top of the list.

A set of standard views are provided in the basic set of Qt Quick graphical types:
ListView - arranges items in a horizontal or vertical list
GridView - arranges items in a grid within the available space
PathView - arranges items on a path

- ListView

```
ListModel {
 id: nameModel
 ListElement { name: "Alice" }
 ListElement { name: "Bob" }
 ListElement { name: "Jane" }
 ListElement { name: "Harry" }
 ListElement { name: "Wendy" }
}
Component {
 id: nameDelegate
 Text {
  text: name;
  font.pixelSize: 24
 }
}
ListView {
 anchors.fill: parent
 clip: true
 model: nameModel
 delegate: nameDelegate
 header: bannercomponent
 footer: Rectangle {
  width: parent.width; height: 30;
  gradient: clubcolors
 }
 highlight: Rectangle {
  width: parent.width
  color: "lightgray"
 }
}
Component {     //instantiated when header is processed
 id: bannercomponent
 Rectangle {
  id: banner
  width: parent.width; height: 50
  gradient: clubcolors
  border {color: "#9EDDF2"; width: 2}
  Text {
   anchors.centerIn: parent
   text: "Club Members"
   font.pixelSize: 32
  }
 }
}
Gradient {
 id: clubcolors
 GradientStop { position: 0.0; color: "#8EE2FE"}
 GradientStop { position: 0.66; color: "#7ED2EE"}
}
```

- Using Delegates

Views need a delegate to visually represent an item in a list. A view will visualize each item list according to the template defined by the delegate. Items in a model are accessible through the index property as well as the item's properties.

```
Component {
id: petdelegate
Text {
 id: label
 font.pixelSize: 24
 text: if (index == 0)
  label.text = type + " (default)"
 else
  text: type
 }
}
```

Cat (default)
Dog
Mouse
Rabbit
Horse

The list view to which the delegate is bound is accessible from the delegate through the ListView.view property. Likewise, the GridView GridView.view is available to delegates. The corresponding model and its properties, therefore, are available through ListView.view.model. In addition, any defined signals or methods in the model are also accessible.

```
Rectangle {
 width: 200; height: 200
ListModel {
 id: fruitModel
 property string language: "en"
 ListElement {
  name: "Apple"
  cost: 2.45
 }
 ListElement {
  name: "Orange"
  cost: 3.25
 }
 ListElement {
  name: "Banana"
  cost: 1.95
 }
}
Component {
 id: fruitDelegate
 Row {
  id: fruit
  Text { text: " Fruit: " + name; color: fruit.ListView.view.fruit_color }
  Text { text: " Cost: $" + cost }
  Text { text: " Language: " + fruit.ListView.view.model.language }
 }
}
ListView {
 property color fruit_color: "green"
 model: fruitModel
 delegate: fruitDelegate
 anchors.fill: parent
}
}
```

Day 3 Afternoon

13. Presenting Data

- Data Item
  - Arranging Items
    - Positioners
    - Repeaters
- Scene Graph
  - Data Models
  - Using Views
  - Using Delegates

- Implement List data model view through delegate

<main.pro>
```
QT+=quick
TEMPLATE = app
TARGET = modelapp
INCLUDEPATH += .
RCC_DIR=build
OBJECTS_DIR=build
MOC_DIR=build

# Input
HEADERS += person.h
SOURCES += main.cpp person.cpp
RESOURCES += main.qrc
```

<main.qrc>
```
<RCC>
 <qresource>
  <file>main.qml</file>
 </qresource>
</RCC>
```

<person.h>
```
#ifndef PERSON_H
#define PERSON_H
#include <QObject>
#include <QString>
#include <QList>
class Person:public QObject {
Q_OBJECT
Q_PROPERTY(QString name READ name WRITE setName)
```

```
Q_PROPERTY(int age READ age WRITE setAge)
Q_ENUMS(Height)
Q_PROPERTY(Height height READ height WRITE setHeight)
public:
Person(QObject *p=0);
enum Height{SHORT=0,MID,TALL};
QString name() const;
void setName(const QString&);
int age() const;
void setAge(int);
Height height() const;
void setHeight(Height);
private:
QString _name;
int _age;
Height _height;
};
#endif
```

<person.cpp>

```
#include "person.h"
Person::Person(QObject *p):QObject(p){}
QString Person::name() const{ return _name; }
void Person::setName(const QString& name){
if (name!=_name){
_name=name;
//emit nameChanged();
}
}
int Person::age() const{ return _age; }
void Person::setAge(int age){
if (age!=_age){
_age=age;
//emit ageChanged();
}
}
Person::Height Person::height()const { return _height; }
void Person::setHeight(Person::Height height){
if (_height!=height){
_height=height;
//emit heightChanged();
}
}
```

<main.cpp>

```
#include <QGuiApplication>
#include <QQmlEngine>
#include <QQuickView>
#include <QDebug>
#include "person.h"

int main(int argc, char ** argv) {
QGuiApplication app(argc, argv);
qmlRegisterType<Person>("Person", 1,0, "Person");
 QQuickView view;
 view.setResizeMode(QQuickView::SizeRootObjectToView);
 view.setSource(QUrl("qrc:main.qml"));
 view.show();
return app.exec();
}
```

<main.qml>

```
import QtQuick 2.0
import Person 1.0
Item{
property list<Person> personmodel:[Person{name:"John";age:20;height:Person.MID}, Person{name:"Rinku";age:10;height:Person.TALL} ]
ListView{
width:200;height:400
model:personmodel
delegate:Text{
text:name+":"+age+":"+height
}
}
}
```

John:20:21
Rinku:10:21