# 0.The QML Reference

https://doc.qt.io/qt-5/qmlreference.html

QML is a multi-paradigm language for creating highly dynamic applications. With QML, application building blocks such as UI components are *declared* and various properties set to define the application behavior. Application behavior can be further scripted through JavaScript, which is a subset of the language. In addition, QML heavily uses Qt, which allows types and other Qt features to be accessible directly from QML applications.

This reference guide describes the features of the QML language. Many of the QML types in the guide originate from the [Qt QML](https://doc.qt.io/qt-5/qtqml-index.html) or [Qt Quick](https://doc.qt.io/qt-5/qtquick-index.html) modules.

* [QML Syntax Basics](https://doc.qt.io/qt-5/qtqml-syntax-basics.html)
  + [Import Statements](https://doc.qt.io/qt-5/qtqml-syntax-imports.html)
  + [Object Declarations](https://doc.qt.io/qt-5/qtqml-syntax-basics.html" \l "object-declarations)
    - [Child Objects](https://doc.qt.io/qt-5/qtqml-syntax-basics.html" \l "child-objects)
  + [Comments](https://doc.qt.io/qt-5/qtqml-syntax-basics.html" \l "comments)
* [QML Object Attributes](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html)
  + [The](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "the-id-attribute)*[id](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "the-id-attribute)*[Attribute](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "the-id-attribute)
  + [Property Attributes](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "property-attributes)
  + [Signal Attributes](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "signal-attributes)
  + [Method Attributes](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "method-attributes)
  + [Attached Properties and Attached Signal Handlers](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "attached-properties-and-attached-signal-handlers)
  + [Enumeration Attributes](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "enumeration-attributes)
* [Property Binding](https://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html)
* [Signal and Handler Event System](https://doc.qt.io/qt-5/qtqml-syntax-signals.html)
* [Integrating QML and JavaScript](https://doc.qt.io/qt-5/qtqml-javascript-topic.html)
  + [Using JavaScript Expressions with QML](https://doc.qt.io/qt-5/qtqml-javascript-expressions.html)
  + [Dynamic QML Object Creation from JavaScript](https://doc.qt.io/qt-5/qtqml-javascript-dynamicobjectcreation.html)
  + [Defining JavaScript Resources In QML](https://doc.qt.io/qt-5/qtqml-javascript-resources.html)
  + [Importing JavaScript Resources In QML](https://doc.qt.io/qt-5/qtqml-javascript-imports.html)
  + [JavaScript Host Environment](https://doc.qt.io/qt-5/qtqml-javascript-hostenvironment.html)
* [The QML Type System](https://doc.qt.io/qt-5/qtqml-typesystem-topic.html)
  + [Basic Types](https://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html)
  + [JavaScript Types](https://doc.qt.io/qt-5/qtqml-typesystem-topic.html" \l "javascript-types)
  + [QML Object Types](https://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html)
    - [Defining Object Types from QML](https://doc.qt.io/qt-5/qtqml-documents-definetypes.html)
    - [Defining Object Types from C++](https://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html)
* [QML Modules](https://doc.qt.io/qt-5/qtqml-modules-topic.html)
  + [Specifying A QML Module](https://doc.qt.io/qt-5/qtqml-modules-qmldir.html)
  + [Supported QML Module Types](https://doc.qt.io/qt-5/qtqml-modules-topic.html" \l "supported-qml-module-types)
    - [Identified Modules](https://doc.qt.io/qt-5/qtqml-modules-identifiedmodules.html)
    - [Legacy Modules](https://doc.qt.io/qt-5/qtqml-modules-legacymodules.html)
  + [Providing Types and Functionality in a C++ Plugin](https://doc.qt.io/qt-5/qtqml-modules-cppplugins.html)
* [QML Documents](https://doc.qt.io/qt-5/qtqml-documents-topic.html)
  + [Structure of a QML Document](https://doc.qt.io/qt-5/qtqml-documents-structure.html)
  + [Syntax of the QML Language](https://doc.qt.io/qt-5/qtqml-documents-topic.html" \l "syntax-of-the-qml-language)
  + [Defining Object Types through QML Documents](https://doc.qt.io/qt-5/qtqml-documents-definetypes.html)
    - [Defining an Object Type with a QML File](https://doc.qt.io/qt-5/qtqml-documents-definetypes.html" \l "defining-an-object-type-with-a-qml-file)
    - [Accessible Attributes of Custom Types](https://doc.qt.io/qt-5/qtqml-documents-definetypes.html" \l "accessible-attributes-of-custom-types)
  + [Resource Loading and Network Transparency](https://doc.qt.io/qt-5/qtqml-documents-networktransparency.html)
  + [Scope and Naming Resolution](https://doc.qt.io/qt-5/qtqml-documents-scope.html)

# 1.QML Syntax Basics

https://doc.qt.io/qt-5/qtqml-syntax-basics.html

QML is a multi-paradigm language that enables objects to be defined in terms of their attributes and how they relate and respond to changes in other objects. In contrast to purely imperative code, where changes in attributes and behavior are expressed through a series of statements that are processed step by step, QML's declarative syntax integrates attribute and behavioral changes directly into the definitions of individual objects. These attribute definitions can then include imperative code, in the case where complex custom application behavior is needed.

QML source code is generally loaded by the engine through QML *documents*, which are standalone documents of QML code. These can be used to define [QML object types](https://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) that can then be reused throughout an application. Note that type names must begin with an uppercase letter in order to be declared as QML object types in a QML file.

## Import Statements

A QML document may have one or more imports at the top of the file. An import can be any one of:

* a versioned namespace into which types have been registered (e.g., by a plugin)
* a relative directory which contains type-definitions as QML documents
* a JavaScript file

JavaScript file imports must be qualified when imported, so that the properties and methods they provide can be accessed.

The generic form of the various imports are as follows:

* import Namespace VersionMajor.VersionMinor
* import Namespace VersionMajor.VersionMinor as SingletonTypeIdentifier
* import "directory"
* import "file.js" as ScriptIdentifier

Examples:

* import [QtQuick](https://doc.qt.io/qt-5/qtquick-module.html) 2.0
* import [QtQuick](https://doc.qt.io/qt-5/qtquick-module.html).LocalStorage 2.0 as Database
* import "../privateComponents"
* import "somefile.js" as Script

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

## Object Declarations

Syntactically, a block of QML code defines a tree of QML objects to be created. Objects are defined using *object declarations* that describe the type of object to be created as well as the attributes that are to be given to the object. Each object may also declare child objects using nested object declarations.

An object declaration consists of the name of its object type, followed by a set of curly braces. All attributes and child objects are then declared within these braces.

Here is a simple object declaration:

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 100

height: 100

color: "red"}

This declares an object of type [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html), followed by a set of curly braces that encompasses the attributes defined for that object. The [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type is a type made available by the QtQuick module, and the attributes defined in this case are the values of the rectangle's width, height and color properties. (These are properties made available by the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type, as described in the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) documentation.)

The above object can be loaded by the engine if it is part of a [QML document](https://doc.qt.io/qt-5/qtqml-documents-topic.html). That is, if the source code is complemented with *import* statement that imports the QtQuick module (to make the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type available), as below:

import QtQuick 2.0

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 100

height: 100

color: "red"}

When placed into a .qml file and loaded by the QML engine, the above code creates a [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object using the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) type supplied by the QtQuick module:



**Note:**If an object definition only has a small number of properties, it can be written on a single line like this, with the properties separated by semi-colons:

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) { width: 100; height: 100; color: "red" }

Obviously, the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object declared in this example is very simple indeed, as it defines nothing more than a few property values. To create more useful objects, an object declaration may define many other types of attributes: these are discussed in the [QML Object Attributes](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html) documentation. Additionally, an object declaration may define child objects, as discussed below.

### Child Objects

Any object declaration can define child objects through nested object declarations. In this way, **any object declaration implicitly declares an object tree that may contain any number of child objects**.

For example, the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object declaration below includes a [Gradient](https://doc.qt.io/qt-5/qml-qtquick-gradient.html) object declaration, which in turn contains two [GradientStop](https://doc.qt.io/qt-5/qml-qtquick-gradientstop.html) declarations:

import QtQuick 2.0

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 100

height: 100

gradient: Gradient {

[GradientStop](https://doc.qt.io/qt-5/qml-qtquick-gradientstop.html) { position: 0.0; color: "yellow" }

[GradientStop](https://doc.qt.io/qt-5/qml-qtquick-gradientstop.html) { position: 1.0; color: "green" }

}}

When this code is loaded by the engine, it creates an object tree with a [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object at the root; this object has a [Gradient](https://doc.qt.io/qt-5/qml-qtquick-gradient.html) child object, which in turn has two [GradientStop](https://doc.qt.io/qt-5/qml-qtquick-gradientstop.html) children.

Note, however, that this is a parent-child relationship in the context of the QML object tree, not in the context of the visual scene. The concept of a parent-child relationship in a visual scene is provided by the [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) type from the QtQuick module, which is the base type for most QML types, as most QML objects are intended to be visually rendered. For example, [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) and [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) are both [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html)-based types, and below, a [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object has been declared as a visual child of a [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object:

import QtQuick 2.0

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 200

height: 200

color: "red"

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) {

anchors.centerIn: parent

text: "Hello, QML!"

}}

When the [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object refers to its [parent](https://doc.qt.io/qt-5/qml-qtquick-item.html" \l "parent-prop) value in the above code, it is referring to its *visual parent*, not the parent in the object tree. In this case, they are one and the same: the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object is the parent of the [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object in both the context of the QML object tree as well as the context of the visual scene. However, while the [parent](https://doc.qt.io/qt-5/qml-qtquick-item.html" \l "parent-prop) property can be modified to change the visual parent, the parent of an object in the context of the object tree cannot be changed from QML.

(Additionally, notice that the [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object has been declared without assigning it to a property of the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html), unlike the earlier example which assigned a [Gradient](https://doc.qt.io/qt-5/qml-qtquick-gradient.html) object to the rectangle's gradient property. This is because the [children](https://doc.qt.io/qt-5/qml-qtquick-item.html" \l "children-prop) property of [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) has been set as the type's [default property](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "default-properties) to enable this more convenient syntax.)

See the [visual parent](https://doc.qt.io/qt-5/qtquick-visualcanvas-visualparent.html) documentation for more information on the concept of visual parenting with the [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) type.

## Comments

The syntax for commenting in QML is similar to that of JavaScript:

* Single line comments start with // and finish at the end of the line.
* Multiline comments start with /\* and finish with \*/

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) {

text: "Hello world!" //a basic greeting

/\*

We want this text to stand out from the rest so

we give it a large size and different font.

\*/

font.family: "Helvetica"

font.pointSize: 24}

Comments are ignored by the engine when processing QML code. They are useful for explaining what a section of code is doing, whether for reference at a later date or for explaining the implementation to others.

Comments can also be used to prevent the execution of code, which is sometimes useful for tracking down problems.

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) {

text: "Hello world!"

//opacity: 0.5}

In the above example, the [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object will have normal opacity, since the line opacity: 0.5 has been turned into a comment.

<https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html>

# 2.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](https://doc.qt.io/qt-5/qtqml-syntax-basics.html" \l "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
* enumeration 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](https://doc.qt.io/qt-5/qml-qtquick-textinput.html) object and a [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object. The [TextInput](https://doc.qt.io/qt-5/qml-qtquick-textinput.html) object's id value is set to "myTextInput". The [Text](https://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](https://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](https://doc.qt.io/qt-5/qml-qtquick-column.html) {

width: 200; height: 200

[TextInput](https://doc.qt.io/qt-5/qml-qtquick-textinput.html) { id: myTextInput; text: "Hello World" }

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) { 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](https://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](https://doc.qt.io/qt-5/qobject.html" \l "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] [required] [readonly] property <propertyType> <propertyName>

In this way an object declaration may [expose a particular value](https://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html" \l "defining-object-types-from-qml) to outside objects or maintain some internal state more easily.

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, required, and readonly keywords are optional, and modify the semantics of the property being declared. See the upcoming sections on [default properties](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "default-properties), [required properties](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "required-properties) and, [read-only properties](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "read-only-properties) for more information about their respective meaning.

Declaring a custom property implicitly creates a value-change [signal](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "signal-attributes) for that property, as well as an associated [signal handler](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "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](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "signal-handler-attributes) implemented for one of those new properties:

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

property [color](https://doc.qt.io/qt-5/qml-color.html) previousColor

property [color](https://doc.qt.io/qt-5/qml-color.html) nextColor

onNextColorChanged: console.log("The next color will be: " + nextColor.toString())}

##### Valid Types in Custom Property Definitions

Any of the [QML Basic Types](https://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html) aside from the [enumeration](https://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](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

property [int](https://doc.qt.io/qt-5/qml-int.html) someNumber

property [string](https://doc.qt.io/qt-5/qml-string.html) someString

property [url](https://doc.qt.io/qt-5/qml-url.html) someUrl}

(Enumeration values are simply whole number values and can be referred to with the [int](https://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](https://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html) documentation for more details.

Note the [var](https://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: 1.5property var someString: "abc"property var someBool: trueproperty var someList: [1, 2, "three", "four"]property var someObject: Rectangle { width: 100; height: 100; color: "red" }

Additionally, any [QML object type](https://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](https://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html" \l "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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

color: "red"

property [color](https://doc.qt.io/qt-5/qml-color.html) 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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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](https://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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

// both of these are static value assignments on initialization

width: 400

height: 200

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

// 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()](https://doc.qt.io/qt-5/qml-qtqml-qt.html" \l "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](https://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html) for more information.

#### 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](https://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](https://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](https://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](https://doc.qt.io/qt-5/qml-qtquick-item.html) type has a [states](https://doc.qt.io/qt-5/qml-qtquick-item.html" \l "states-prop) property that is used to hold a list of [State](https://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](https://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html) objects:

import QtQuick 2.0

[Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

states: [

[State](https://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html) { name: "loading" },

[State](https://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html) { name: "running" },

[State](https://doc.qt.io/qt-5/qml-qtqml-statemachine-state.html) { name: "stopped" }

]}

If the list contains a single item, the square brackets may be omitted:

import QtQuick 2.0

[Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

states: State { name: "running" }}

A [list](https://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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

// declaration without initialization

property list<[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html)> siblingRects

// declaration with initialization

property list<[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html)> childRects: [

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) { color: "red" },

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) { 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](https://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](https://doc.qt.io/qt-5/qml-qtquick-text.html) type has a [font](https://doc.qt.io/qt-5/qml-qtquick-text.html" \l "font.family-prop) group property. Below, the first [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object initializes its font values 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](https://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html) for more information.

#### 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 has the following restrictions:

* It can only refer to an object, or the property of an object, that is within the scope of the [type](https://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) within which the alias is declared.
* It cannot contain arbitrary JavaScript expressions
* It cannot refer to objects declared outside of the scope of its type.
* 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.
* It cannot refer to [attached properties](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "attached-properties-and-attached-signal-handlers).
* It cannot refer to properties inside a hierarchy with depth 3 or greater. The following code will not work:

property alias color: myItem.myRect.border.color

Item {

id: myItem

property Rectangle myRect

}

However, aliases to properties that are up to two levels deep will work.

property alias color: rectangle.border.color

Rectangle {

id: rectangle}

For example, below is a Button type with a buttonText aliased property which is connected to the text object of the [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) child:

// Button.qmlimport QtQuick 2.0

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

property alias buttonText: textItem.text

width: 100; height: 30; color: "yellow"

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) { id: textItem }}

The following code would create a Button with a defined text string for the child [Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) object:

[Button](https://doc.qt.io/qt-5/qml-qtquick-controls2-button.html) { 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](https://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.

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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html" \l "color-prop) property:

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: coloredrectangle

property alias color: bluerectangle.color

color: "red"

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html" \l "color-prop) property. Internally, however, the rectangle can correctly set its color property and refer to the actual defined property rather than the alias.

##### Property Aliases and Types

Property aliases cannot have explicit type specifications. The type of a property alias is the *declared* type of the property or object it refers to. Therefore, if you create an alias to an object referenced via id with extra properties declared inline, the extra properties won't be accessible through the alias:

// MyItem.qml[Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

property alias inner: innerItem

[Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

id: innerItem

property [int](https://doc.qt.io/qt-5/qml-int.html) extraProperty

}}

You cannot initialize *inner.extraProperty* from outside of this component, as inner is only an *Item*:

// main.qmlMyItem {

inner.extraProperty: 5 // fails}

However, if you extract the inner object into a separate component with a dedicated .qml file, you can instantiate that component instead and have all its properties available through the alias:

// MainItem.qmlItem {

// Now you can access inner.extraProperty, as inner is now an ExtraItem

property alias inner: innerItem

ExtraItem {

id: innerItem

}}

// ExtraItem.qmlItem {

property int extraProperty}

#### 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.qmlimport QtQuick 2.0

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) {

default property [var](https://doc.qt.io/qt-5/qml-var.html) someText

text: "Hello, " + someText.text}

The someText value could be assigned to in a MyLabel object definition, like this:

MyLabel {

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) { 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](https://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](https://doc.qt.io/qt-5/qml-qtquick-item.html)-based type without explicitly adding them to the [children](https://doc.qt.io/qt-5/qml-qtquick-item.html" \l "children-prop) property. This is because the default property of [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) is its data property, and any items added to this list for an [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) are automatically added to its list of [children](https://doc.qt.io/qt-5/qml-qtquick-item.html" \l "children-prop).

Default properties can be useful for reassigning the children of an item. See the [TabWidget Example](https://doc.qt.io/qt-5/qtquick-customitems-tabwidget-example.html), which uses a default property to automatically reassign children of the TabWidget as children of an inner [ListView](https://doc.qt.io/qt-5/qml-qtquick-listview.html). See also [Extending QML](https://doc.qt.io/qt-5/qtquick-codesamples.html" \l "extending-qml).

#### Required Properties

An object declaration may define a property as required, using the required keyword. The syntax is

required property <propertyType> <propertyName>

As the name suggests, required properties must be set when an instance of the object is created. Violation of this rule will result in QML applications not starting if it can be detected statically. In case of dynamically instantiated QML components (for instance via [Qt.createComponent()](https://doc.qt.io/qt-5/qml-qtqml-qt.html" \l "createComponent-method)), violating this rule results in a warning and a null return value.

It's possible to make an existing property required with

required <propertyName>

The following example shows how to create a custom Rectangle component, in which the color property always needs to be specified.

// ColorRectangle.qml[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

required color}

**Note:**You can't assign an initial value to a required property from QML, as that would go directly against the intended usage of required properties.

Required properties play a special role in model-view-delegate code: If the delegate of a view has required properties whose names match with the role names of the view's model, then those properties will be initialized with the model's corresponding values. For more information, visit the [Models and Views in Qt Quick](https://doc.qt.io/qt-5/qtquick-modelviewsdata-modelview.html) page.

and {[QQuickView::setInitialProperties](https://doc.qt.io/qt-5/qquickview.html" \l "setInitialProperties)} for ways to initialize required properties from C++.

#### 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](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

readonly property [int](https://doc.qt.io/qt-5/qml-int.html) someNumber: 10

Component.onCompleted: someNumber = 20 // doesn't work, causes an error}

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

#### Property Modifier Objects

Properties can have [property value modifier objects](https://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html" \l "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](https://doc.qt.io/qt-5/qtqml-syntax-basics.html" \l "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 NumberAnimation with 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](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type, for example, has a [clicked](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html" \l "clicked-signal) signal that is emitted when the user clicks within the mouse area.

An object can be notified through a [signal handler](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "signal-handler-attributes) whenever 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](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) object definition, and is invoked when the [MouseArea](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) is clicked, causing a console message to be printed:

import QtQuick 2.0

[Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

width: 100; height: 100

[MouseArea](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) {

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](https://doc.qt.io/qt-5/qobject.html" \l "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](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

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](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "defining-property-attributes) on this page.

To emit a signal, invoke it as a method. Any relevant [signal handlers](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "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](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "property-attributes). See the upcoming section on [property change signal handlers](https://doc.qt.io/qt-5/qtqml-syntax-signals.html" \l "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](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: root

signal activated(real xPosition, real yPosition)

signal deactivated

property [int](https://doc.qt.io/qt-5/qml-int.html) side: 100

width: side; height: side

[MouseArea](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) {

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.qmlSquareButton {

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

onDeactivated: console.log("Deactivated!")}

See the [Signal and Handler Event System](https://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](https://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](https://doc.qt.io/qt-5/qml-qtquick-textinput.html) has a [text](https://doc.qt.io/qt-5/qml-qtquick-textinput.html" \l "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](https://doc.qt.io/qt-5/qml-qtquick-textinput.html) {

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](https://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](https://doc.qt.io/qt-5/qobject.html" \l "Q_INVOKABLE) or by registering it as a [Q\_SLOT](https://doc.qt.io/qt-5/qobject.html" \l "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](https://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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) is clicked it invokes the moveTo() method which can then refer to the received newX and newY parameters to reposition the text:

import QtQuick 2.0

[Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

width: 200; height: 200

[MouseArea](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) {

anchors.fill: parent

onClicked: label.moveTo(mouse.x, mouse.y)

}

[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) {

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](https://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html" \l "providing-attached-properties)*[attaching type](https://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html" \l "providing-attached-properties)*[in C++](https://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html" \l "providing-attached-properties) 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](https://doc.qt.io/qt-5/qml-qtquick-listview.html) type has an attached property [ListView.isCurrentItem](https://doc.qt.io/qt-5/qml-qtquick-listview.html" \l "isCurrentItem-attached-prop) that is available to each delegate object in a [ListView](https://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](https://doc.qt.io/qt-5/qml-qtquick-listview.html) {

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](https://doc.qt.io/qt-5/qml-qtqml-component.html" \l "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](https://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](https://doc.qt.io/qt-5/qml-qtquick-listview.html) {

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](https://doc.qt.io/qt-5/qml-qtqml-component.html" \l "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](https://doc.qt.io/qt-5/qml-qtquick-item.html) and the colored [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) is a child of that item:

import QtQuick 2.0

[ListView](https://doc.qt.io/qt-5/qml-qtquick-listview.html) {

width: 240; height: 320

model: 3

delegate: Item {

width: 100; height: 30

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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](https://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](https://doc.qt.io/qt-5/qml-qtquick-listview.html) {

//....

delegate: Item {

id: delegateItem

width: 100; height: 30

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 100; height: 30

color: delegateItem.ListView.isCurrentItem ? "red" : "yellow" // correct

}

}}

Now delegateItem.ListView.isCurrentItem correctly refers to the isCurrentItem attached property of the delegate.

### Enumeration Attributes

Enumerations provide a fixed set of named choices. They can be declared in QML using the enum keyword:

// MyText.qml[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) {

enum TextType {

Normal,

Heading

}}

As shown above, enumeration types (e.g. TextType) and values (e.g. Normal) must begin with an uppercase letter.

Values are referred to via <Type>.<EnumerationType>.<Value> or <Type>.<Value>.

// MyText.qml[Text](https://doc.qt.io/qt-5/qml-qtquick-text.html) {

enum TextType {

Normal,

Heading

}

property [int](https://doc.qt.io/qt-5/qml-int.html) textType: MyText.TextType.Normal

font.bold: textType == MyText.TextType.Heading

font.pixelSize: textType == MyText.TextType.Heading ? 24 : 12}

More information on enumeration usage in QML can be found in the [QML Basic Types](https://doc.qt.io/qt-5/qtqml-typesystem-basictypes.html) [enumeration](https://doc.qt.io/qt-5/qml-enumeration.html) documentation.

The ability to declare enumerations in QML was introduced in Qt 5.10.

**See also**[QQmlComponent::createWithInitialProperties](https://doc.qt.io/qt-5/qqmlcomponent.html" \l "createWithInitialProperties) and [QQmlApplicationEngine::setInitialProperties](https://doc.qt.io/qt-5/qqmlapplicationengine.html" \l "setInitialProperties).

# 3.Property Binding

An object's property can be assigned a static value which stays constant until it is explicitly assigned a new value. However, to make the fullest use of QML and its built-in support for dynamic object behaviors, most QML objects use *property bindings*.

Property bindings are a core feature of QML that lets developers specify relationships between different object properties. When a property's *dependencies* change in value, the property is automatically updated according to the specified relationship.

Behind the scenes, the QML engine monitors the property's dependencies (that is, the variables in the binding expression). When a change is detected, the QML engine re-evaluates the binding expression and applies the new result to the property.

## Overview

To create a property binding, a property is assigned a JavaScript expression that evaluates to the desired value. At its simplest, a binding may be a reference to another property. Take the following example, where the blue [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html)'s height is bound to the height of its parent:

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 200; height: 200

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 100

height: parent.height

color: "blue"

}}

Whenever the height of the parent rectangle changes, the height of the blue rectangle automatically updates to be of the same value.

A binding can contain any valid JavaScript expression or statement, as QML uses a standards compliant JavaScript engine. Bindings can access object properties, call methods and use built-in JavaScript objects such as Date and Math. Below are other possible bindings for the previous example:

height: parent.height / 2

height: Math.min(parent.width, parent.height)

height: parent.height > 100 ? parent.height : parent.height/2

height: {

if (parent.height > 100)

return parent.height

else

return parent.height / 2}

height: someMethodThatReturnsHeight()

Below is a more complex example involving more objects and types:

[Column](https://doc.qt.io/qt-5/qml-qtquick-column.html) {

id: column

width: 200

height: 200

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: topRect

width: Math.max(bottomRect.width, parent.width/2)

height: (parent.height / 3) + 10

color: "yellow"

[TextInput](https://doc.qt.io/qt-5/qml-qtquick-textinput.html) {

id: myTextInput

text: "Hello QML!"

}

}

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: bottomRect

width: 100

height: 50

color: myTextInput.text.length <= 10 ? "red" : "blue"

}}

In the previous example,

* topRect.width depends on bottomRect.width and column.width
* topRect.height depends on column.height
* bottomRect.color depends on myTextInput.text.length

Syntactically, bindings are allowed to be of arbitrary complexity. However, if a binding is overly complex - such as involving multiple lines, or imperative loops - it could indicate that the binding is being used for more than describing property relationships. Complex bindings can reduce code performance, readability, and maintainability. It may be a good idea to redesign components that have complex bindings, or at least factor the binding out into a separate function. As a general rule, users should not rely on the evaluation order of bindings.

## Creating Property Bindings from JavaScript

A property with a binding is automatically updated as necessary. However, if the property is later assigned a static value from a JavaScript statement, the binding will be removed.

For example, the [Rectangle](https://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 height as 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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

width: 100

height: width \* 2

focus: true

Keys.onSpacePressed: {

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

}}

Now, after the space key is pressed, the rectangle's height will continue auto-updating to always be three times its width.

#### Debugging overwriting of bindings

A common cause of bugs in QML applications is accidentally overwriting bindings with static values from JavaScript statements. To help developers track down problems of this kind, the QML engine is able to emit messages whenever a binding is lost due to imperative assignments.

In order to generate such messages, you need to enable the informational output for the qt.qml.binding.removal logging category, for instance by calling:

[QLoggingCategory](https://doc.qt.io/qt-5/qloggingcategory.html)::setFilterRules([QStringLiteral](https://doc.qt.io/qt-5/qstring.html" \l "QStringLiteral)("qt.qml.binding.removal.info=true"));

Please refer to the [QLoggingCategory](https://doc.qt.io/qt-5/qloggingcategory.html) documentation for more information about enabling output from logging categories.

Note that is perfectly reasonable in some circumstances to overwrite bindings. Any message generated by the QML engine should be treated as a diagnostic aid, and not necessarily as evidence of a problem without further investigation.

### Using this with Property 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.

For example, the Component.onCompleted handler below is defined within the scope of the [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html). In this scope, width refers to the [Item](https://doc.qt.io/qt-5/qml-qtquick-item.html)'s width, not the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html)'s width. To bind the [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html)'s height to its own width, the binding expression must explicitly refer to this.width (or alternatively, rect.width):

[Item](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

width: 500

height: 500

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: rect

width: 100

color: "yellow"

}

Component.onCompleted: {

rect.height = Qt.binding(function() { return this.width \* 2 })

console.log("rect.height = " + rect.height) // prints 200, not 1000

}}

**Note:**The value of this is not defined outside of property bindings. See [JavaScript Environment Restrictions](https://doc.qt.io/qt-5/qtqml-javascript-hostenvironment.html" \l "javascript-environment-restrictions) for details.

**See also**[Positioning with Anchors](https://doc.qt.io/qt-5/qtquick-positioning-anchors.html).

# 4.Signal and Handler Event System

https://doc.qt.io/qt-5/qtqml-syntax-signals.html

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 a script 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 [Button](https://doc.qt.io/qt-5/qml-qtquick-controls2-button.html) type from the [Qt Quick Controls](https://doc.qt.io/qt-5/qtquickcontrols-index.html) module has a clicked signal, which is emitted whenever the button is clicked. In this case, the signal handler for receiving this signal should be onClicked. In the example below, whenever the button is clicked, the onClicked handler is invoked, applying a random color to the parent [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html):

import QtQuick 2.15import QtQuick.Controls 2.15

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: rect

width: 250; height: 250

[Button](https://doc.qt.io/qt-5/qml-qtquick-controls2-button.html) {

anchors.bottom: parent.bottom

anchors.horizontalCenter: parent.horizontalCenter

text: "Change color!"

onClicked: {

rect.color = Qt.rgba(Math.random(), Math.random(), Math.random(), 1);

}

}}

### 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](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html) type has a [pressed](https://doc.qt.io/qt-5/qml-qtquick-mousearea.html" \l "pressed-signal) property. To receive a notification whenever this property changes, write a signal handler named onPressedChanged:

import QtQuick 2.15

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: rect

width: 100; height: 100

[TapHandler](https://doc.qt.io/qt-5/qml-qtquick-taphandler.html) {

onPressedChanged: console.log("taphandler pressed?", pressed)

}}

Even though the [TapHandler](https://doc.qt.io/qt-5/qml-qtquick-taphandler.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](https://doc.qt.io/qt-5/qml-qtqml-connections.html) type for connecting to signals of arbitrary objects. A [Connections](https://doc.qt.io/qt-5/qml-qtqml-connections.html) object can receive any signal from its specified [target](https://doc.qt.io/qt-5/qml-qtqml-connections.html" \l "target-prop).

For example, the onClicked handler in the earlier example could have been received by the root [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) instead, by placing the onClicked handler in a [Connections](https://doc.qt.io/qt-5/qml-qtqml-connections.html) object that has its [target](https://doc.qt.io/qt-5/qml-qtqml-connections.html" \l "target-prop) set to the button:

import QtQuick 2.15import QtQuick.Controls 2.15

Rectangle {

id: rect

width: 250; height: 250

Button {

id: button

anchors.bottom: parent.bottom

anchors.horizontalCenter: parent.horizontalCenter

text: "Change color!"

}

Connections {

target: button

function onClicked(): {

rect.color = Qt.rgba(Math.random(), Math.random(), Math.random(), 1);

}

}}

### Attached signal handlers

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

For example, [Component.onCompleted](https://doc.qt.io/qt-5/qml-qtqml-component.html" \l "completed-signal) is an attached signal handler. It is often used to execute some JavaScript code when its creation process is complete. Here is an example:

import QtQuick 2.15

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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 a completed signal from the [Rectangle](https://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](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object by the QML engine. The engine emits this signal when the Rectangle object is 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, 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 extra code.

See [Attached properties and attached signal handlers](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "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, the code below is defined in a file named SquareButton.qml. The root [Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) object has an activated signal, which is emitted whenever the child [TapHandler](https://doc.qt.io/qt-5/qml-qtquick-taphandler.html) is tapped. In this particular example the activated signal is emitted with the x and y coordinates of the mouse click:

// SquareButton.qmlimport QtQuick 2.15

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: root

signal activated(real xPosition, real yPosition)

property [point](https://doc.qt.io/qt-5/qml-point.html) mouseXY

property [int](https://doc.qt.io/qt-5/qml-int.html) side: 100

width: side; height: side

[TapHandler](https://doc.qt.io/qt-5/qml-qtquick-taphandler.html) {

id: handler

onTapped: root.activated(mouseXY.x, mouseXY.y)

onPressedChanged: mouseXY = handler.point.position

}}

Now any objects of the SquareButton can connect to the activated signal using an onActivated signal handler:

// myapplication.qmlSquareButton {

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

See [Signal Attributes](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "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:

import QtQuick 2.15

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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 earlier, 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](https://doc.qt.io/qt-5/qtqml-javascript-dynamicobjectcreation.html).

There is a corresponding disconnect() method for removing connected signals:

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

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.

import QtQuick 2.15

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) {

id: forwarder

width: 100; height: 100

signal send()

onSend: console.log("Send clicked")

[TapHandler](https://doc.qt.io/qt-5/qml-qtquick-taphandler.html) {

id: mousearea

anchors.fill: parent

onTapped: console.log("Mouse clicked")

}

Component.onCompleted: {

mousearea.tapped.connect(send)

}}

Whenever the [TapHandler](https://doc.qt.io/qt-5/qml-qtquick-taphandler.html)'s tapped signal is emitted, the send signal will automatically be emitted as well.

output:

MouseArea clicked

Send clicked

# 5.Integrating QML and JavaScript

https://doc.qt.io/qt-5/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.

## JavaScript Expressions

QML has a deep JavaScript integration, and allows [signal handlers](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "signal-attributes) and [methods](https://doc.qt.io/qt-5/qtqml-syntax-objectattributes.html" \l "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](https://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html), which are also defined using JavaScript.

See the documentation page titled [JavaScript Expressions in QML Documents](https://doc.qt.io/qt-5/qtqml-javascript-expressions.html) for more information about using JavaScript expressions in QML.

## JavaScript Resources

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.

See the documentation page titled [Defining JavaScript Resources In QML](https://doc.qt.io/qt-5/qtqml-javascript-resources.html) for more information about defining JavaScript resources for QML.

## JavaScript Imports

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.

See the documentation page titled [Importing JavaScript Resources](https://doc.qt.io/qt-5/qtqml-javascript-imports.html) for more information on how to import JavaScript resources and how to use the functionality they provide.

## JavaScript Host Environment

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.

These limitations and extensions are documented in the description of the [JavaScript Host Environment](https://doc.qt.io/qt-5/qtqml-javascript-hostenvironment.html) provided by the QML engine.

## Fine Tuning the JavaScript engine

For specific use cases you may want to override some of the parameters the JavaScript engine uses for handling memory and compiling JavaScript. See [Fine Tuning the JavaScript engine](https://doc.qt.io/qt-5/qtqml-javascript-finetuning.html) for more information on these parameters.

# 6.The QML Type System

https://doc.qt.io/qt-5/qtqml-typesystem-topic.html

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.

See the [QML Basic Types](https://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](https://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](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

property [var](https://doc.qt.io/qt-5/qml-var.html) theArray: []

property [var](https://doc.qt.io/qt-5/qml-var.html) 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](https://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](https://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](https://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html) for in-depth information about object types.

# 6.1QML Object Types

https://doc.qt.io/qt-5/qtqml-typesystem-objecttypes.html

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](https://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](https://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++](https://doc.qt.io/qt-5/qtqml-cppintegration-definetypes.html). Note that in both cases, the type name must begin with an uppercase letter in order to be declared as a QML object type in a QML file.

## 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](https://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](https://doc.qt.io/qt-5/qtqml-documents-definetypes.html).

See [Defining Object Types through QML Documents](https://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](https://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](https://doc.qt.io/qt-5/qml-qtquick-item.html) {

id: root

width: 500; height: 500

[Component](https://doc.qt.io/qt-5/qml-qtqml-component.html) {

id: myComponent

[Rectangle](https://doc.qt.io/qt-5/qml-qtquick-rectangle.html) { 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](https://doc.qt.io/qt-5/qml-qtqml-component.html" \l "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](https://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](https://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](https://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](https://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](https://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++](https://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](https://doc.qt.io/qt-5/qtqml-modules-topic.html) for more information about how to create and deploy a QML extension module.

# 7.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

https://doc.qt.io/qt-5/qtqml-modules-topic.html

A module is defined by a [module definition qmldir file](https://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](https://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](https://doc.qt.io/qt-5/qtqml-syntax-imports.html" \l "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](https://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](https://doc.qt.io/qt-5/qtqml-modules-identifiedmodules.html)
* [Legacy Modules](https://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](https://doc.qt.io/qt-5/qtqml-syntax-imports.html" \l "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](https://doc.qt.io/qt-5/qtqml-modules-cppplugins.html) for more information.

# 8.QML Documents

https://doc.qt.io/qt-5/qtqml-documents-topic.html

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](https://doc.qt.io/qt-5/qml-qtqml-component.html) in QML code, or a [QQmlComponent](https://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](https://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](https://doc.qt.io/qt-5/qtqml-syntax-basics.html). An object declaration may include the specification of custom [object attributes](https://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](https://doc.qt.io/qt-5/qtqml-syntax-propertybinding.html).

Please see the documentation about the [syntax of QML](https://doc.qt.io/qt-5/qtqml-syntax-basics.html) for more information about valid syntax, and see the documentation about [integrating QML and JavaScript](https://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 }  } | IMG_256 |

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