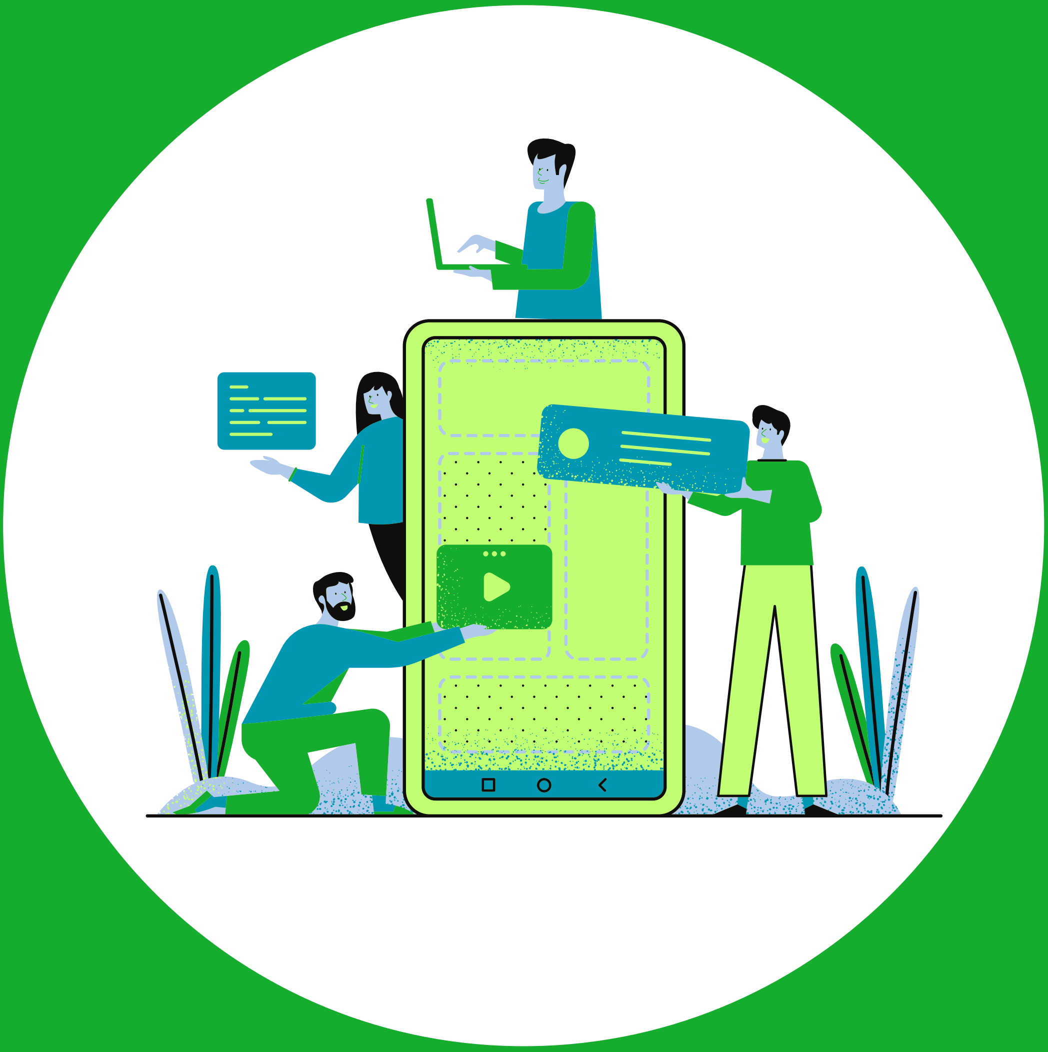
BUILD ONCE,DEPLOY EVERYWHERE.

Qt6 QML FOR BEGINNERS

FLUID AND DYNAMIC USER INTERFACES FOR DESKTOP,MOBILE AND EMBEDDED



DANIEL GAKWAYA

Qt6 QML For Beginners

Daniel Gakwaya

Version 1.0, 2023-04-28

Table of Contents

About the author

Daniel is a Senior Software Engineer at Blikoon Technologies. He has been writing software since 2011 and Qt is one of his favorite frameworks. Many of his cross-platform Qt-based projects are completed and out there on the market. Autodidact by nature, he has a passion for learning new things and loves sharing knowledge with others. Especially people new to the dark art of software development.

Preface

Building graphical user interfaces has always been a fascinating subject for me. My journey started back in the 2000s while in college and we were using a library called SDL to create games. But games weren't really my thing. I wanted to have some kind of GUI on a PC that I could use to control some electronic device. Nothing fancy though: you push a button, some light or LED goes on and you press it again and it goes off. The hardware wasn't hard as I was following an Electrical Engineering degree, but as good as SDL was it didn't quite cut it for me. I took a course on MFC, a GUI framework from Microsoft that was very popular at the time and started using it to create some things. I used it for a while but the API design seemed so convoluted for me. Naturally, I kept looking for something better. It was a Saturday afternoon and I stumbled on a section in a book I was reading. There was a chapter on Qt. I tried setting up a single button and I wanted to simply print out a message when the button was clicked. The button would emit a signal and we would connect that signal to a function that'll be triggered when the button is clicked. The syntax looked something like below

```
connect(button, SIGNAL(clicked()), this, SLOT(method()));
```

It worked right away! This seemed very intuitive for me and I felt like someone somewhere had listened to my pain and troubles and created a solution to address all of them. Qt was open source and I could even look at the code that created the magic that was having fun with. I kept using Qt in my personal, research and freelance projects for a few years that followed and in 2013 I landed my full-time paid contract as a Qt C++ developer. QML was starting to gain some traction around the time when Qt5 was released, but I was having too much fun with Qt Widgets and other cool stuff from the C++ side to give it any noticeable attention.

A few years later, I was back to doing things solo and my client wanted a "non-standard" GUI for some desktop application we were building. The pay was good and I was determined to see it through. QML was one of the options and I started trying things out. To my surprise, I was able to put together a prototype with some cool animations and transitions in a fairly short amount of time. The client liked it! Understandably, I started to learn as much as I could about QML. I started falling for the declarative nature of QML and I have to admit that it is the first declarative language I have ever used. Fast forward to the time I am writing this, and QML is a big part of my daily work.

Learning QML wasn't a straight-line path for me. I had to browse through pages and pages of the official Qt documentation, do lots of searches on stackoverflow and learn from mistakes through trial and error. I remember wishing there was a book or a video course that would make the job slightly easier. I promised myself I would document my learning journey through a book or a video course and I was able to deliver on the video course thing by creating a series of courses on QML that got a great number of good reviews from students. That surprised me! But it also told me that there are thousands of people out there, in need of good, practice-based learning material on QML, and Qt in general.

This book is my first attempt at sharing what I was honored to learn about QML, hoping to make the learning journey less daunting, a little more predictable and most importantly, fun! The author is well aware that you can't learn everything you need on a piece of technology from a book. That's why, just like he does in his video courses, he takes every chance he gets to lure you into using the documentation, a great way to learn about things the author couldn't fit in the book, or plainly doesn't know or care about, but that could be important to you.

In closing, I want to thank you for choosing this book and going with me, through this fascinating journey of learning to build fluid, dynamic and cross-platform graphical user interfaces with QML.

Target audience

Qt is one of the most documented software projects I have had the honor to work with in my decade-long career as a software developer. The docs can however seem overwhelming as each page has lots of links pointing you in dozens of different directions from the current subject of interest. This is even more true if you are a beginner and looking to find your way through QML development. This is where this book comes in. It's to be a step-by-step guide, covering QML from the fundamentals to intermediate concepts like animations and transitions, avoiding complicated jargon and explaining things in plain English. Each section is backed by a Qt Quick project we develop together in Qt Creator, allowing you to practice as we go. This book is for if

- you are a student or researcher looking to learn QML
- you are an aspiring developer looking to add QML to your stack of skills
- you are an existing developer looking to jump into QML development

What this book covers

Chapter 1, ***First Steps with Qt QML***, provides a brief introduction to Qt and QML, describes how to get Qt and Qt Creator installed on your host system and guides you through the process of building and understanding the building blocks of your first QML application.

Chapter 2, ***Dissecting the QML Syntax***, explores the QML syntax, guides you through the process of using QML basic QML data types in a live QML application, and introduces you to property bindings. It also gets your feet wet using property handlers and explores some features of the Qt global object.

Chapter 3, ***Basic QML Elements***, explores the QML elements: Item, Rectangle, Image and Text. It also shows you how you can compose these basic, built-in QML elements to build your own custom QML components.

Chapter 4, ***Signals and Handlers***, introduces the signals and slots mechanism in QML. It explores several facilities at our disposal, helping us communicate between different QML objects.

Chapter 5, ***User Input***, navigates through the user input facilities provided by QML, helping you add interactivity to your QML applications.

Chapter 6, ***Javascript***, sheds more light on where and how Javascript is used in your QML code

Chapter 7, ***Positioning***, explores the positioning mechanisms offered by QML. We look at anchors, positioners, layouts and a few others.

Get the most out of the book

There is no magic pill you can take to consume the knowledge in this book. One needs to take the time to read the content and make sure to try the code in Qt Creator. Once you have the thing running on your host computer, you can then try to change things a bit, read the docs to see if you can figure some other things out on your own. The book is not meant to give you everything. And it really can't! It's meant to get you started and help you pick up some momentum. Once you have the basics under your belt, and have a few applications running, you can even try to use the knowledge to build things I didn't necessarily do in the course, for example, why not build a calculator app, or a calory tracker, or really anything that happens to pick your interest. The more of these things you do the more momentum you'll be picking up and who knows where you'll end up? Here is a rule I try to follow these days: "Try to spend 20% of the time learning and consuming new knowledge and 80% of the time using the knowledge to build stuff."

Download the source code

The source code for the book is available in the book git repository

Chapter 1: First Steps with Qt QML

QML is a great piece of technology from the Qt framework. It can build great-looking, slick, fluid and dynamic user interfaces that work well even on tiny mobile and embedded devices, all that while taking advantage of the cross-platform nature of Qt. To master it, however, one has to start somewhere, lay a good foundation and gradually build on top of that until you reach higher ground. This chapter is just about that: laying a good foundation. We will start by exploring what exactly Qt and QML stand for. We'll then set up a development environment; this includes the Qt Creator IDE, and the actual Qt libraries that will allow us to carry out some magic. With the environment in place, we'll then build our very first Qt Quick/QML application and explore the main files, along with the syntax and main building blocks in those files. This will put us in a much better shape to start exploring the fundamentals of QML.

Introduction to Qt and QML

Qt is a cross-platform application framework used for developing software applications that run on various platforms such as Windows, macOS, Linux, and mobile operating systems like Android and iOS. It provides a variety of tools and libraries for building user interfaces, handling networking, accessing databases, and many other features that are necessary for developing modern applications.

Qt also offers the QML language, which allows developers to create rich, dynamic, fluid and interactive user interfaces for their applications. QML is a declarative language that is used for describing the user interface of an application. QML is particularly useful for building modern applications that require high-quality graphics, animations, and user experience. It is commonly used in industrial applications, car infotainment systems, medical applications and anywhere you need to go through a modern-looking user interface to talk to your device.

> A declarative programming language is a type of programming language that focuses on expressing what a program should accomplish, rather than how it should accomplish it. In a declarative programming language, you specify the desired outcome or result of the program, rather than providing explicit instructions for how to achieve that result. This is in contrast to imperative programming languages, where you specify the step-by-step instructions for the computer to follow in order to achieve the desired outcome.

The workflow for developing a Qt application typically involves two main stages: **front-end development** and **back-end development**. In the front-end development stage, developers use QML to create the user interface of their applications. This involves designing the visual elements of the application such as buttons, text fields, and images, as well as implementing the behavior of the user interface.

In the back-end development stage, developers use a programming language such as C++ or Python to implement the application logic and handle tasks such as data storage, networking, and communication with external services. Qt provides a variety of libraries and tools for back-end development, such as Qt Core, which provides basic functionality for handling events and data structures, and Qt Network, which provides tools for handling networking tasks.

Qt supports a wide range of target platforms, including desktop platforms such as Windows, macOS, and Linux, as well as mobile platforms such as Android and iOS. Qt developers may choose to work on any of the supported Qt development hosts: Windows, Linux and Mac. Once you have your project ready and tested on one platform, assuming you, the developer, are working on Windows for example, you have to move the project source code to a Linux machine and build it there, if you want a Linux binary executable out of your project. A more accurate motto for Qt, in my opinion, is "Develop Once, build everywhere".

It is important to understand that Qt provides two different ways to write your front end. You can either use **Qt Widgets** or **QML**. The term **Qt Widgets** refers to a collection of reusable components usually used to build desktop graphical user interfaces. To better understand where they stand, they were initially built back in the '90s and the only mainstream devices back then were desktop computers mainly running Windows, Linux, or OSX. So Qt widgets are extremely good at building graphical user interfaces for desktop computers. They were initially built with C++ and only provided a C++ user-facing API.

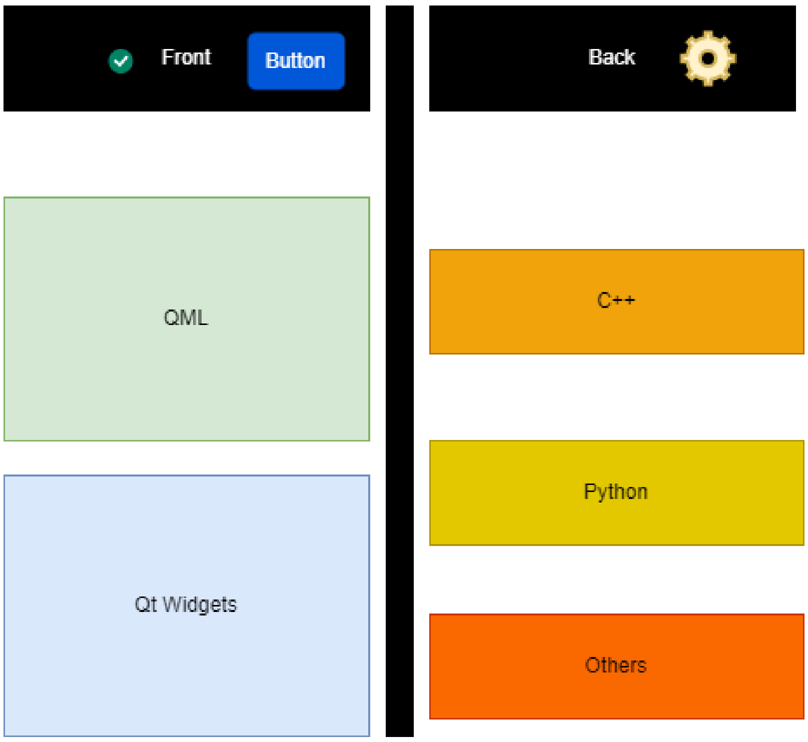


Figure 1. Qt Technologies

Widgets were only part of Qt offerings though, Qt also provided a host of other utility classes for common tasks like networking, threading, databases and lots of other facilities. Qt was so successful at its job that third parties started providing bindings for other languages, the most popular being PyQt and PySide for Python. These allow us to do what we would do with Qt through the Qt native language, C++, but using the Python programming language.

The mobile boom in the 2000s forced application developers to think about devices other than desktop computers. Devices with smaller screens started popping up everywhere, some running mainstream operating systems like Android and IoS, but we also started seeing Linux being ported to devices with smaller memory footprints and we had access to hardware like Raspberry PI and BeagleBone Black. Most of these new small devices had smaller screens and users expected the user interfaces running on them to be fluid and very dynamic with cool animations and transitions.

Qt needed to adapt the framework to accommodate these new platforms. They could have just adapted the Qt C++ API, but they took the chance to introduce a new declarative language, QML, in which you just describe what you want your user interface to look like without thinking too much about the logic that makes it work. Fast forward to the 2020s and QML is a mature technology used by companies big and small all over the world. Influenced by massive amounts of updates to QML in recent years and a seemingly dormant user-facing Qt Widget C++ API, you'll see people out there claiming that Qt Widgets are dead and that no one uses them anymore. That's a huge over-exaggeration in my opinion as there are tons of projects and desktop applications out there still relying on Qt Widgets. Here is my advice to people struggling to pick which Qt technology to use: If you're going to exclusively target desktop operating systems like Windows, Linux and Mac, then Qt Widgets are your best bet. If you think that you'll need to build your application for mobile targets like Android and IoS, or even for some embedded devices where people expect highly fluid and dynamic user interfaces, then QML is the way to go. But don't feel like you need to be on one side or the other, I have personally worked on projects where both Qt Widgets and QML Qt Quick were used together as a UI layer. We would pick which UI to activate based on the target where the application is built: If we're building for desktop for example, we would use a Qt Widgets Ui, and use QML for mobile and embedded targets.

But this book is about QML, and the focus is to help you grasp the fundamental concepts needed to start taking advantage of QML. QML was designed to appeal both to designers and developers. The intent was that it would easily be picked up by designers and make the split between front-end and back-end development with Qt even more clear. Once a designer using QML has the UI ready, the design is passed on to a back-end C++ or Python developer who then connects it to back-end heavy stuff. If you were designing an image processing application, for example, the button to initiate the processing should be done in QML and the actual processing should happen in C++ or any other back-end language supported by Qt. The split is so clear that today we even have two separate IDEs to do the job: Qt Design Studio is the tool you use to do your QML UI design. Once the design is ready, you export it and load it into Qt Creator, where it's connected to back-end heavy stuff. In this book, we'll ignore Qt Design Studio and focus on understanding the plumbing necessary to take as much advantage of QML as you can. Once you have the fundamentals under your belt, you'll then be armed properly to take better advantage of what Qt Design Studio allows you to do.

Now, we could go on and talk about Qt all day but I am not a big fan of talking in the air without some action going on. We'll wrap this introduction here and start installing a development environment in the next one, from where we'll then open Qt Creator and build something, learning new things about QML along the way.

Installing Qt on Your Host System

Now you should have an idea about what Qt and QML are and what they allow you to do. The next logical thing to do is to install Qt on our host system. Qt is flexible in that you can use it from any mainstream operating system like Windows, Linux or Mac. The author will be working from a Windows machine but the installation process is very similar on all operating systems. Head over to the Qt site and click on the button that says **Download.Try**. Qt is available under both a commercial and an open-source license, so you will be given the option to purchase a license if you want to. We will be using the open-source version in the book, so we'll click on the button that says **Go open source**. The Qt site is updated frequently and the locations of these buttons will change over time. If you're having trouble finding the **Go open source** button, you can just directly head to https://www.qt.io/download-open-source where you'll be introduced to your obligations, as a Qt open source user. At this point, we're just learning, we'll just scroll down until we see a button saying **Download the Qt Online Installer**.

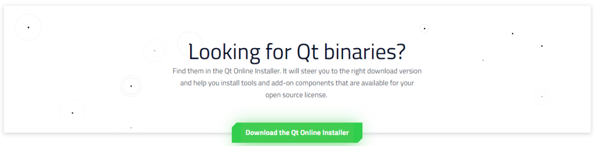
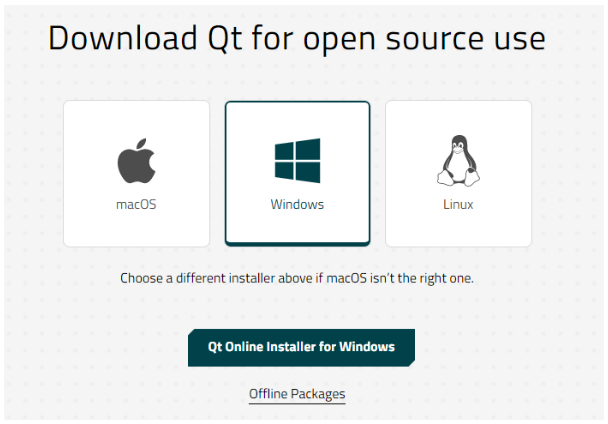

Looking for Qt binaries?
Find them in the Qt Online Installer. It will steer you to the right download version and help you install tools and add-on components that are available for your open source license.

Figure 2. Download the Qt Online Installer

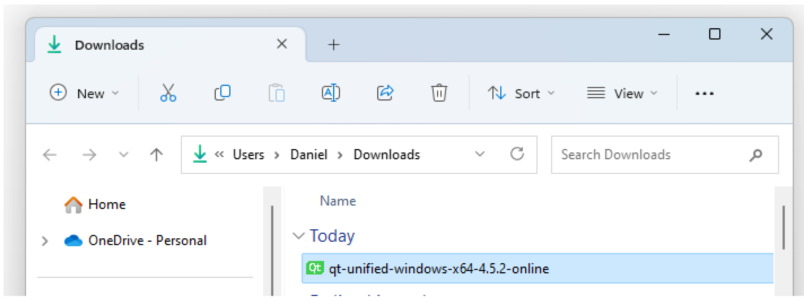
Click on the button, and you'll be taken to a screen with operating system options. They may detect your current operating system, but you can just click on which one you need the installer for.


Download Qt for open source use

Choose a different installer above if macOS isn't the right one.

Figure 3. Download Qt for Open Source Use

In my case, I'll select Windows and click on the button that says **Qt Online Installer for Windows** and that'll kick off the process to download my installer. Mine is stored in my Downloads folder by default and is named **qt-unified-windows-x64-4.5.2-online**.



qt-unified-windows-x64-4.5.2-online

Figure 4. Qt Online Installer in My Downloads Folder

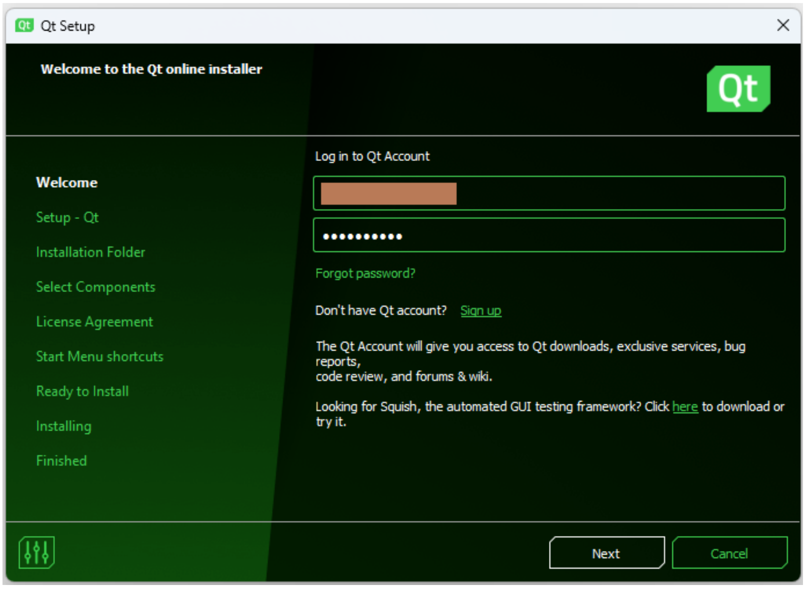
If you are on Linux or Mac, you may need to change the permissions of the downloaded online installer to allow it to execute on your machine.On Windows, you won't need to change these permissions. Below is the command you could run on Linux to change the permissions on the file

```
chmod +x name_of_the_downloaded_online_installer
```

Once you've made sure that the online installer is executable on your system, run it, either by double-clicking on it in the File Explorer graphical user interface or by executing it from a terminal application like below

```
PS C:\Users\Daniel\Downloads> .\qt-unified-windows-x64-4.5.2-online.exe
```

and hitting **Enter**. You'll be presented with the welcome screen for the Qt Online Installer. You'll need to key in your Qt Account credentials. If you don't have one, you can create a fresh one by clicking on the **Sign up** button. Once you have your email and password keyed in, you'll click on next and be presented with the **Qt Open Source Obligations** screen.



Qt Setup

Welcome to the Qt online installer

Welcome
Setup - Qt
Installation Folder
Select Components
License Agreement
Start Menu shortcuts
Ready to Install
Installing
Finished

Log in to Qt Account

[redacted]

••••••••••

Forgot password?

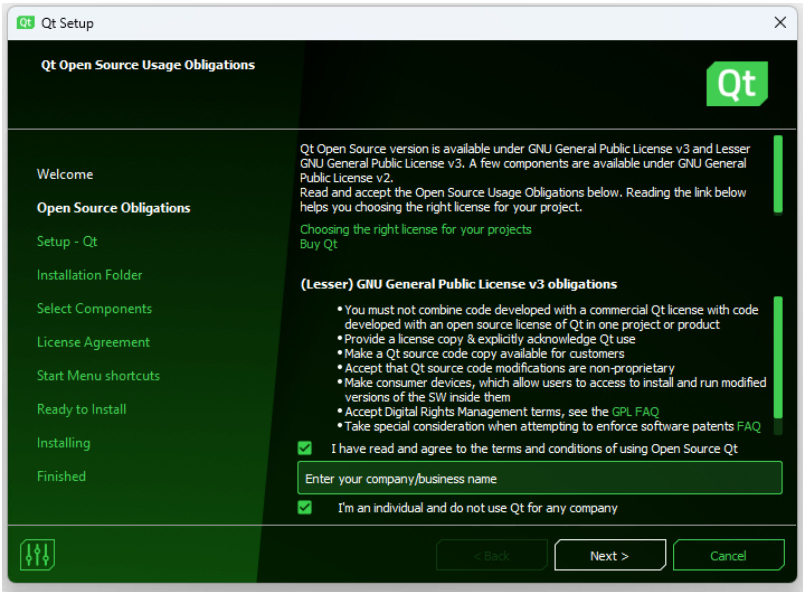
Don't have Qt account? Sign up

The Qt Account will give you access to Qt downloads, exclusive services, bug reports,
code review, and forums & wiki.

Looking for Squish, the automated GUI testing framework? Click here to download or try it.

Next | Cancel

Figure 5. Qt Online Installer Welcome Screen



Qt Setup

Qt Open Source Usage Obligations

Welcome
Open Source Obligations
Setup - Qt
Installation Folder
Select Components
License Agreement
Start Menu shortcuts
Ready to Install
Installing
Finished

Qt Open Source version is available under GNU General Public License v3 and Lesser GNU General Public License v3. A few components are available under GNU General Public License v2.
Read and accept the Open Source Usage Obligations below. Reading the link below helps you choosing the right license for your project.

Choosing the right license for your projects
Buy Qt

(Lesser) GNU General Public License v3 obligations

- You must not combine code developed with a commercial Qt license with code developed with an open source license of Qt in one project or product
- Provide a license copy & explicitly acknowledge Qt use
- Make a Qt source code copy available for customers
- Accept that Qt source code modifications are non-proprietary
- Make consumer devices, which allow users to access to install and run modified versions of the SW inside them
- Accept Digital Rights Management terms, see the GPL FAQ
- Take special consideration when attempting to enforce software patents FAQ

✓ I have read and agree to the terms and conditions of using Open Source Qt

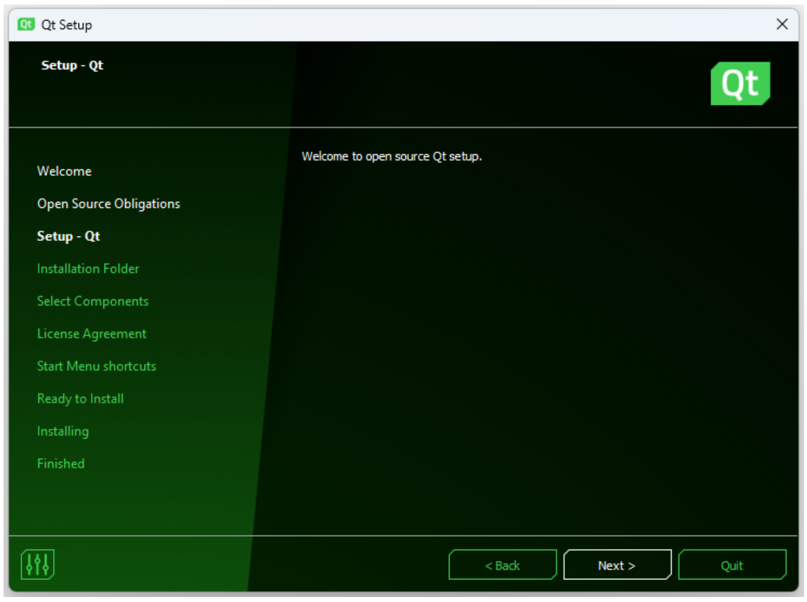
Enter your company/business name

✓ I'm an individual and do not use Qt for any company

< Back | Next > | Cancel

Figure 6. Qt Open Source Obligations

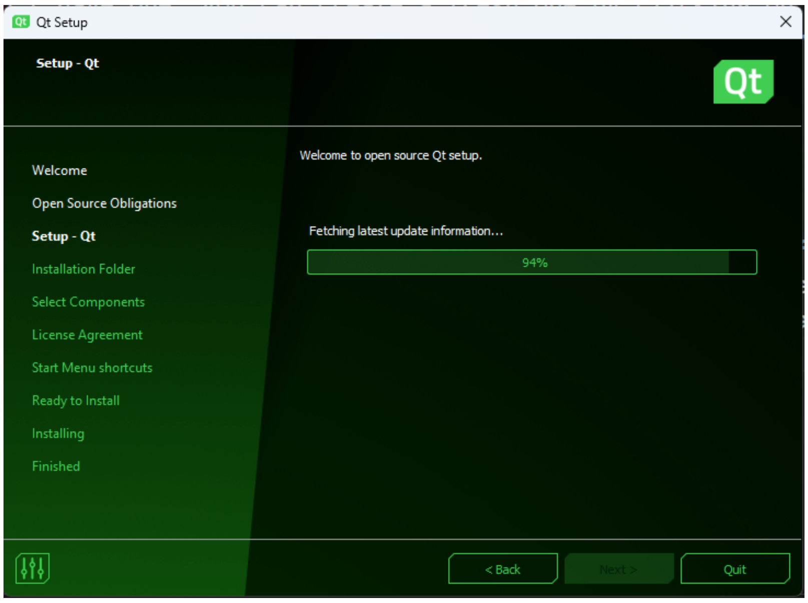
After you read through the obligations, make sure to check the **I have read and agree to the terms and conditions of using Open Source Qt** checkbox. If you are using Qt working for some company, you can key in the name of the company, but we're learning here and ticking the **I'm an individual and do not use Qt for any company** check box will do the job for us. Click next and you'll be given the **Qt Seup** screen on which you'll just click **Next**.



Welcome to open source Qt setup.

Figure 7. Qt Setup Screen

The installer will get some information from the Qt servers



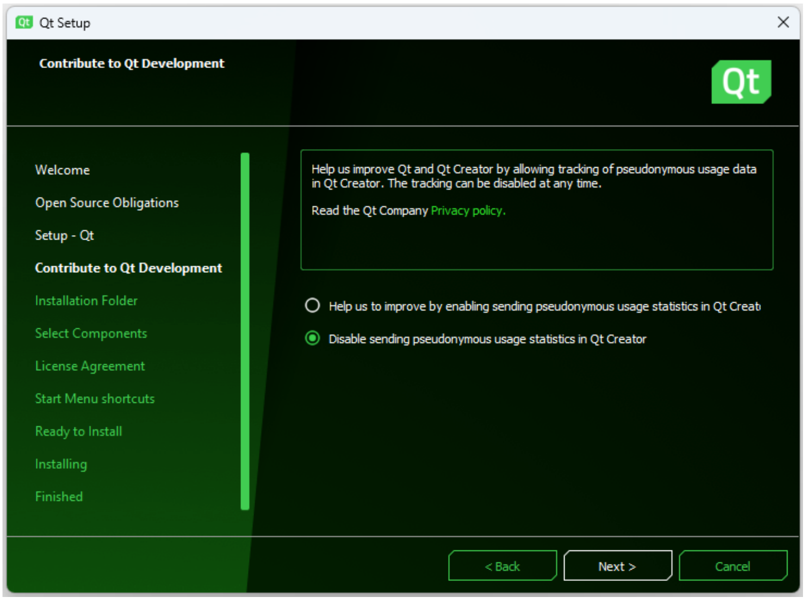
Welcome to open source Qt setup.

Fetching latest update information...

Figure 8. Getting Update Information

and prompt you to allow sending anonymous info about your Qt usage to Qt.

If allowed, Qt Creator collects anonymous usage statistics, such as how often the application is used, how many projects are created, how many files are opened, and how often the various features are used. This information is used to help improve the software by identifying areas that may require further development or bug fixes. It may also collect anonymous crash reports, which include information such as the operating system and version number, the type of hardware being used, and the version of Qt Creator that was running at the time of the crash. This information is used to help identify and fix bugs in the software.


Qt Setup

Contribute to Qt Development

Welcome
Open Source Obligations
Setup - Qt
Contribute to Qt Development
Installation Folder
Select Components
License Agreement
Start Menu shortcuts
Ready to Install
Installing
Finished

Help us improve Qt and Qt Creator by allowing tracking of pseudonymous usage data in Qt Creator. The tracking can be disabled at any time.

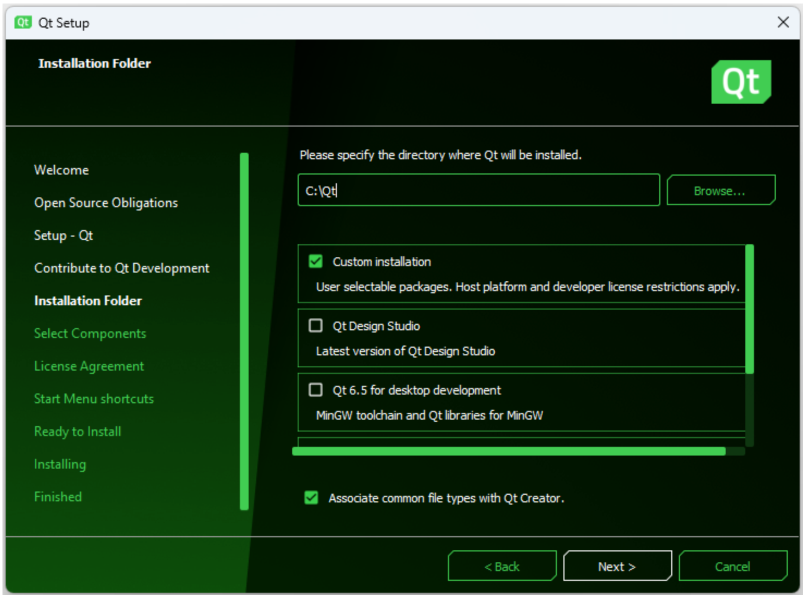
Read the Qt Company Privacy policy.

○ Help us to improve by enabling sending pseudonymous usage statistics in Qt Creat
[✓ Disable sending pseudonymous usage statistics in Qt Creator]

< Back | Next > | Cancel

Figure 9. Send Usage Information to the Qt Company

Make your choice here and click next. You'll be presented with the **Installation Folder Screen**.


Qt Setup

Installation Folder

Welcome
Open Source Obligations
Setup - Qt
Contribute to Qt Development
Installation Folder
Select Components
License Agreement
Start Menu shortcuts
Ready to Install
Installing
Finished

Please specify the directory where Qt will be installed.

C:\Qt | Browse...

[✓ Custom installation]
User selectable packages. Host platform and developer license restrictions apply.

☐ Qt Design Studio
Latest version of Qt Design Studio

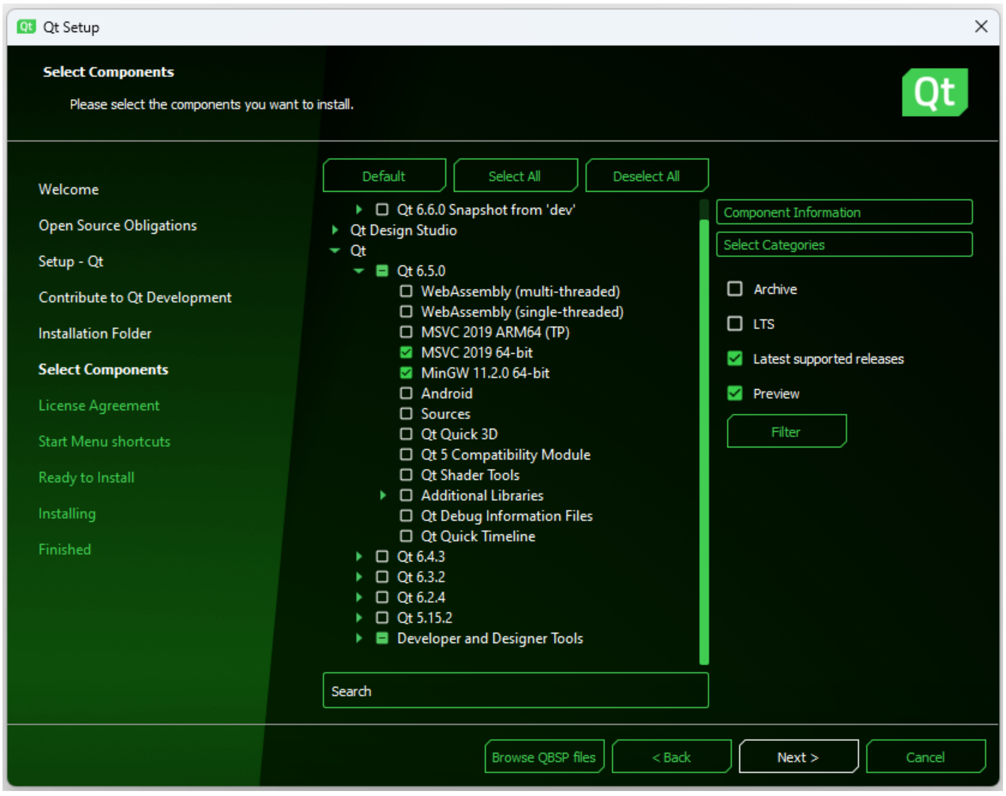
☐ Qt 6.5 for desktop development
MinGW toolchain and Qt libraries for MinGW

[✓ Associate common file types with Qt Creator.]

< Back | Next > | Cancel

Figure 10. Installation Folder

This is where you choose the location where your Qt files will live on your file system. Mine is set to **C:\Qt**. The installer also tries to be helpful here but that means they'll be making big decisions for you. For example, if you tick the **Qt6.x for desktop development** option, you'll install Qt to work with the MinGW compiler but many may not be aware that they could also use the Microsoft Visual C++ compiler on Windows. This is just one example of how these presets can hide the big picture from you. We're big boys and gals here, so we'll set up everything ourselves. We do that by ticking **Custom Installation** . Notice that, if you want, you can also **Associate common files with Qt Creator** which I find useful. Click next and you'll be presented with the screen to **Select Components**. The most important decision to make here is the Qt version to install. I always go for the latest supported release if I can. I do that by ticking **Latest supported releases** on the right side and clicking on **Filter**. The installer will re-populate the user interface with relevant files. In this case, I have an option to install **Qt Design Studio**. We won't be using that in the book but it's always good to have it around so you can tick the check box if you want.



- ☐ Qt 6.6.0 Snapshot from 'dev'
- Qt Design Studio
- Qt
  - ▣ Qt 6.5.0
    - ☐ WebAssembly (multi-threaded)
    - ☐ WebAssembly (single-threaded)
    - ☐ MSVC 2019 ARM64 (TP)
    - ✓ MSVC 2019 64-bit
    - ✓ MinGW 11.2.0 64-bit
    - ☐ Android
    - ☐ Sources
    - ☐ Qt Quick 3D
    - ☐ Qt 5 Compatibility Module
    - ☐ Qt Shader Tools
    - ☐ Additional Libraries
    - ☐ Qt Debug Information Files
    - ☐ Qt Quick Timeline
  - ☐ Qt 6.4.3
  - ☐ Qt 6.3.2
  - ☐ Qt 6.2.4
  - ☐ Qt 5.15.2
  - ▣ Developer and Designer Tools

- ☐ Archive
- ☐ LTS
- ✓ Latest supported releases
- ✓ Preview

Figure 11. Select Components

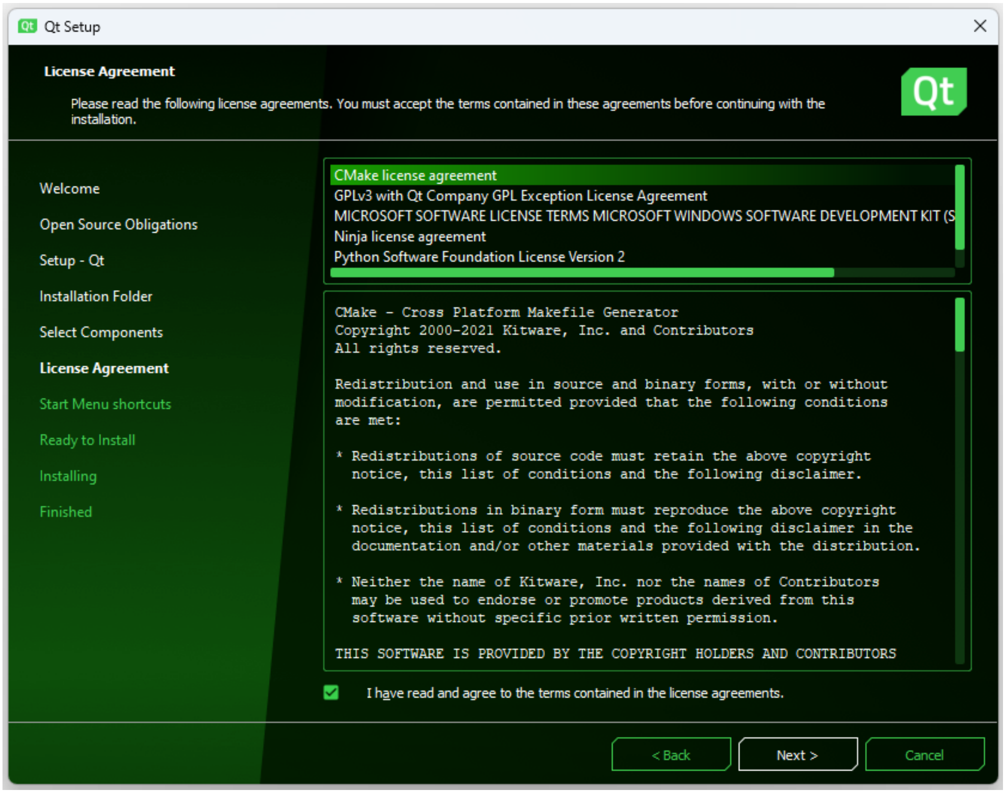
Most importantly, however, we have a **Qt** option that, when expanded, lists out Qt versions available to us. At the time of this writing, the latest Qt version is **Qt6.5.0** so it's what we'll expand. Nothing prevents you from installing multiple Qt versions at the same time on the same system if it's needed. We will stick to the latest Qt version in the book though. If you're reading in the future and there is a version newer than **Qt6.5.0** make sure to pick that. On Windows, we have the option to use Qt with two different compilers behind the scenes: Microsoft Visual C++, usually called MSVC or MinGW.

Ticking MSVC here will allow your Qt installation to pick up the compiler that comes with Microsoft Visual Studio for C++ and use that to build your Qt projects. The keyword here is **pick up**. You won't get the MSVC compiler installed by the Qt Online Installer. You'll have to install that compiler separately, and you get that by installing the Microsoft Visual Studio IDE. Ticking the MinGW checkbox will install Qt files that work with the MinGW compiler, but also download the compiler for you and you'll be ready to start using Qt without any further installs.

Now, some of may be thinking, why would I need to use MSVC if I could get all the files I need right away by using MinGW? Fair question. Sometimes your Qt application will be using other libraries and those libraries may only be working with the MSVC compiler. Another reason may be that the compiler choice may not be yours to make at all. I personally install both on Windows. Be aware, however, that installing both will consume more storage space on your drive. We will only be using MinGW in the book and you can install that only if you want.

Another thing worthy of note is that further down, we have a **Developer and Designer Tools** option that lists the tools that will come with our Qt installation. Expand the option and you'll see that **Qt Creator** is listed there. This is the Qt IDE we'll be using to type our QML code and do a host of other things with in the book. I usually leave this option to the default, but you can tweak it to your liking.

Here we are presented with compilers that work on Windows. If you're on Linux, you'll see a GCC compiler version listed, and if you're on Mac, a Clang version. Make your choices here and click next. The next screen will show you the **Licence Agreement**.


Qt Setup

License Agreement

Please read the following license agreements. You must accept the terms contained in these agreements before continuing with the installation.

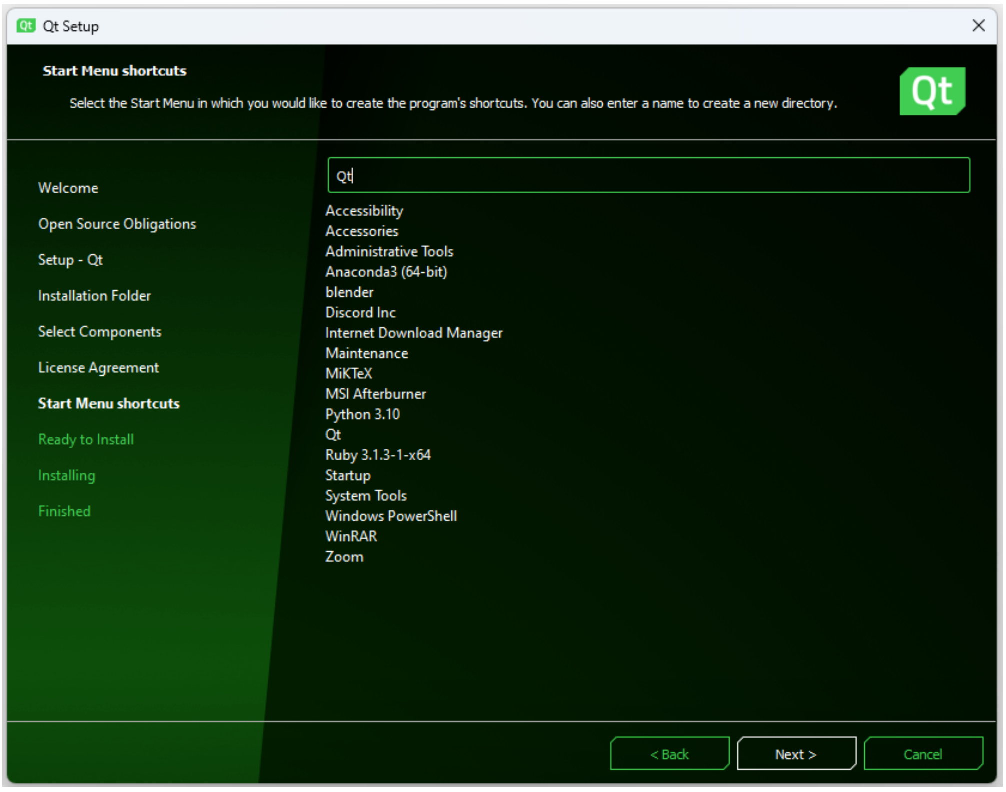
CMake license agreement
GPLv3 with Qt Company GPL Exception License Agreement
MICROSOFT SOFTWARE LICENSE TERMS MICROSOFT WINDOWS SOFTWARE DEVELOPMENT KIT (S
Ninja license agreement
Python Software Foundation License Version 2

```
CMake - Cross Platform Makefile Generator
Copyright 2000-2021 Kitware, Inc. and Contributors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.

* Neither the name of Kitware, Inc. nor the names of Contributors
  may be used to endorse or promote products derived from this
  software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

✓ I have read and agree to the terms contained in the license agreements.

Figure 12. License Agreement

You should read the license, tick the checkbox saying that you agree with the license and click **Next**.


Qt Setup

Start Menu shortcuts

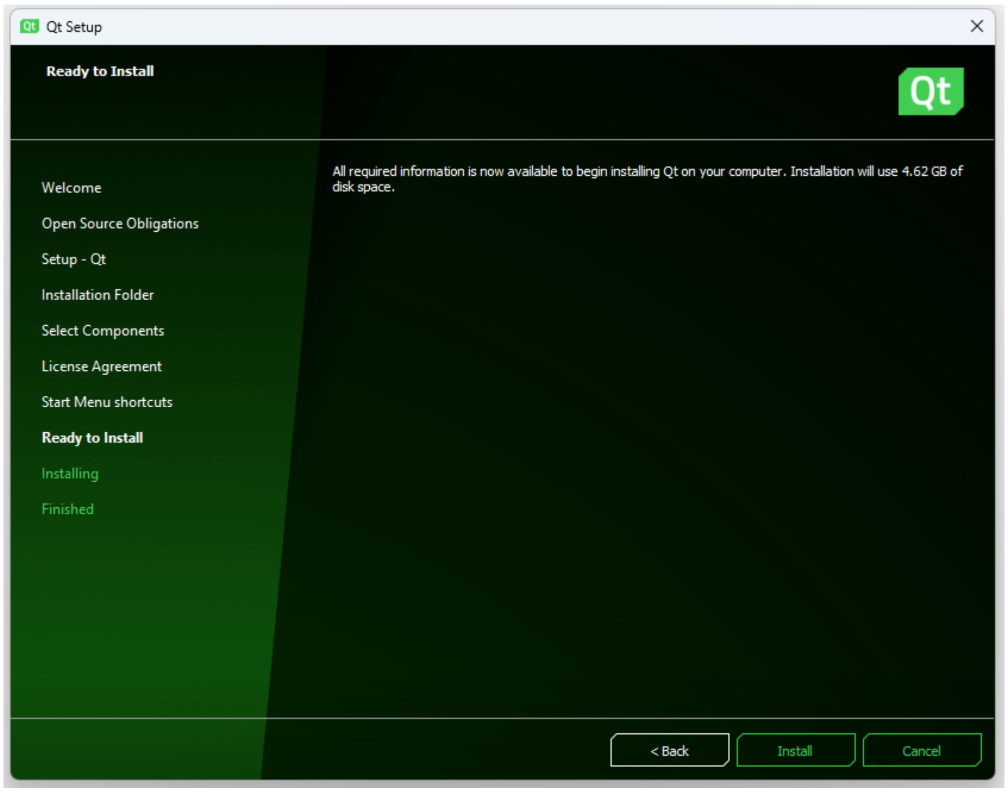
Select the Start Menu in which you would like to create the program's shortcuts. You can also enter a name to create a new directory.

Qt

Accessibility
Accessories
Administrative Tools
Anaconda3 (64-bit)
blender
Discord Inc
Internet Download Manager
Maintenance
MiKTeX
MSI Afterburner
Python 3.10
Qt
Ruby 3.1.3-1-x64
Startup
System Tools
Windows PowerShell
WinRAR
Zoom

Figure 13. Start Menu Shortcuts

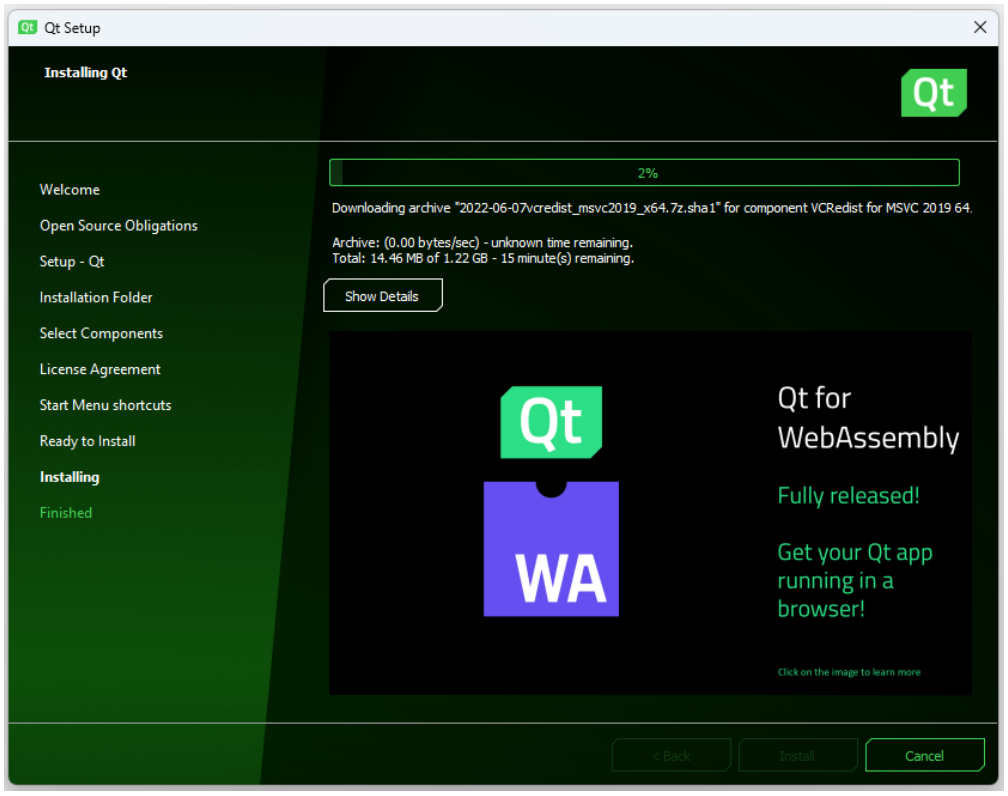
The **Start Menu Shortcuts** screen will come up. I usually leave this as is and click **Next**.



All required information is now available to begin installing Qt on your computer. Installation will use 4.62 GB of disk space.

Figure 14. Ready to Install

You'll be given a screen stating that everything is ready to install, giving you an estimate of how much space Qt will eat up from your drive. Click **Install** and you'll be given a screen showing the progress of the installation.



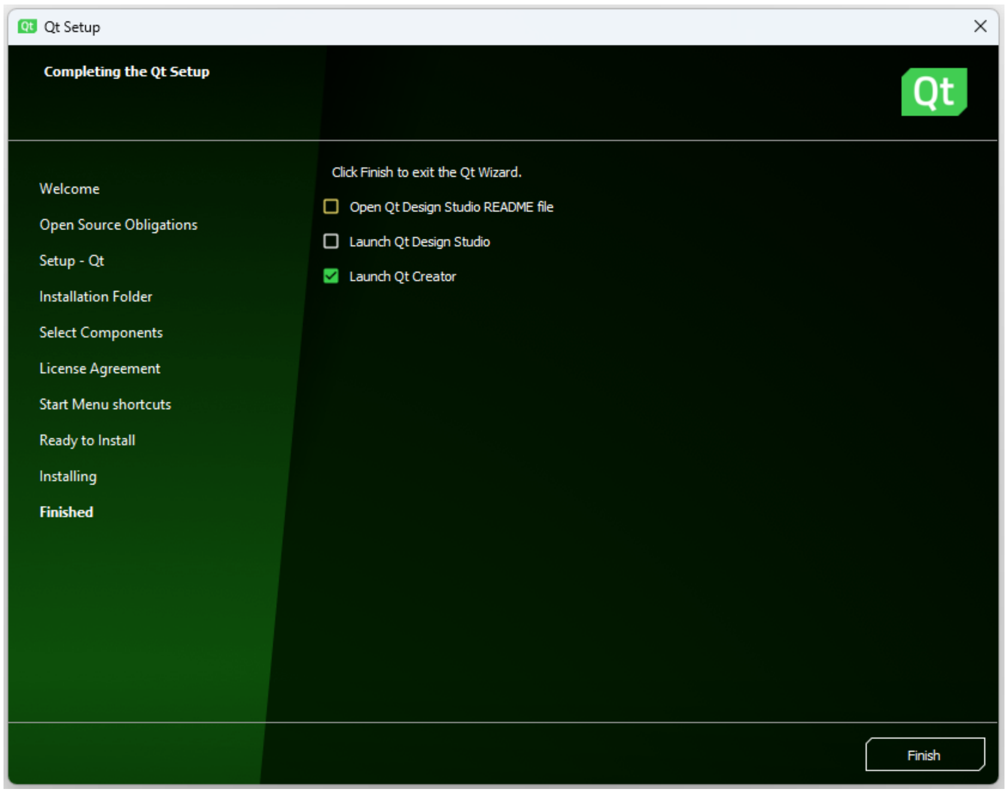
Downloading archive "2022-06-07vcredist_msvc2019_x64.7z.sha1" for component VCRedist for MSVC 2019 64.

Archive: (0.00 bytes/sec) - unknown time remaining.
Total: 14.46 MB of 1.22 GB - 15 minute(s) remaining.

Figure 15. Installing Qt

Some users have reported that in the process of the installation, the installer may occasionally throw errors with hashing issues. If you come across these, there usually is a window giving you a chance to **retry**. Clicking the **Retry** button fixes the issue in most cases.

When the installation is done, you'll see a screen like below


Qt Setup

Completing the Qt Setup

- Welcome
- Open Source Obligations
- Setup - Qt
- Installation Folder
- Select Components
- License Agreement
- Start Menu shortcuts
- Ready to Install
- Installing
- **Finished**

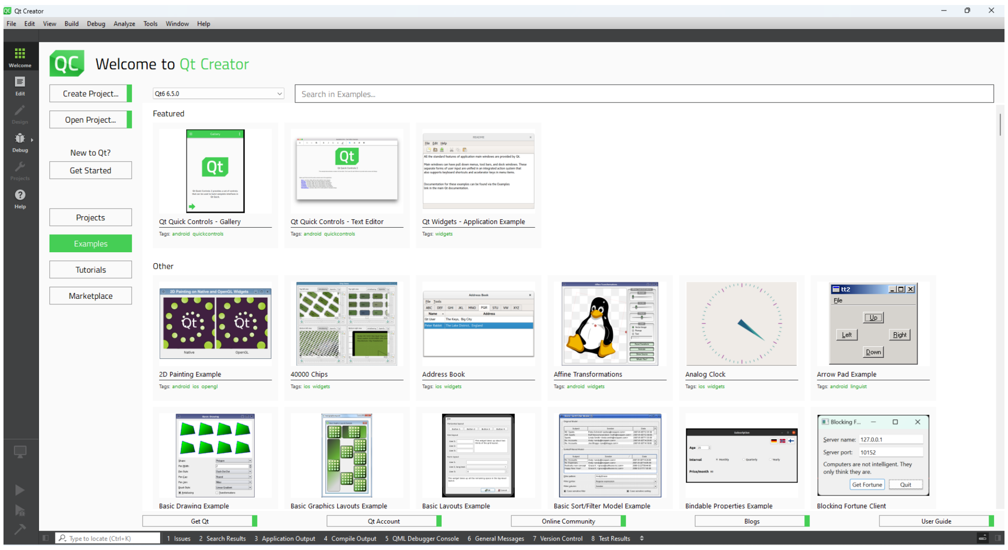
Click Finish to exit the Qt Wizard.

- [ ] Open Qt Design Studio README file
- [ ] Launch Qt Design Studio
- [✓] Launch Qt Creator

Finish

Figure 16. Completing the Qt Setup

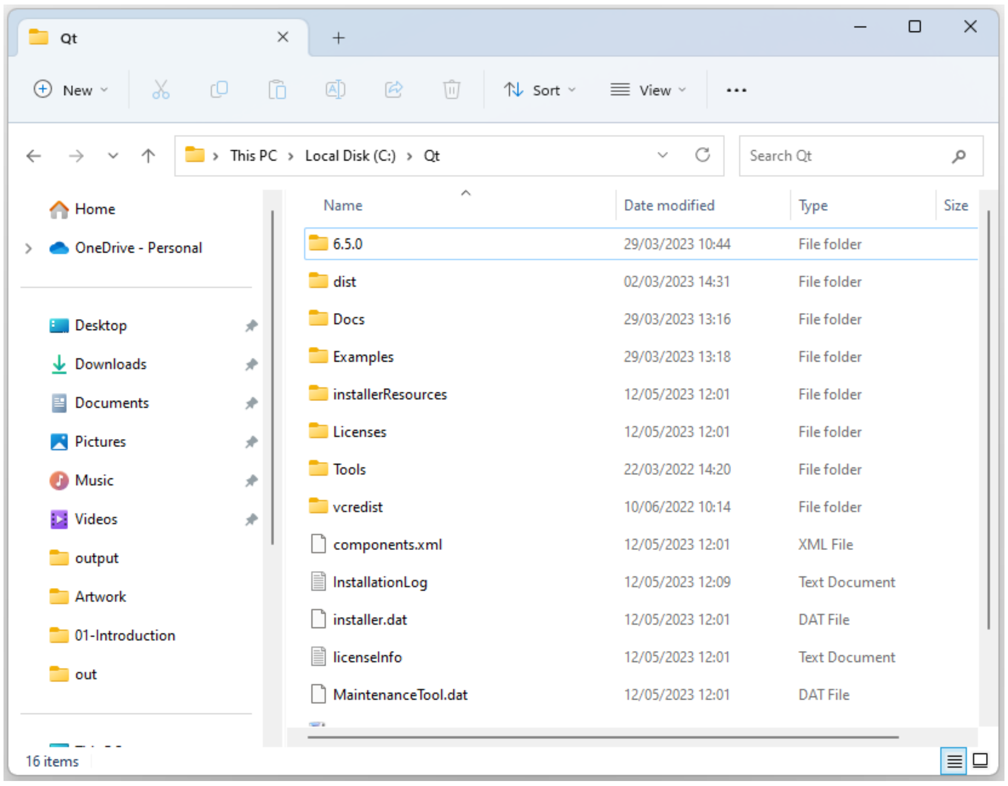
with a few options. Leave the **Launch Qt Creator** one ticked and click **Finish**. This will start the Qt Creator IDE. Wait a few seconds and you'll see it pop up on your screen.



Qt Creator
File Edit View Build Debug Analyze Tools Window Help
Welcome to Qt Creator
Create Project...
Open Project...
New to Qt?
Get Started
Projects
Examples
Tutorials
Marketplace
Qt 6.5.0
Search in Examples...
Featured
Qt Quick Controls - Gallery
Qt Quick Controls - Text Editor
Qt Widgets - Application Example
Other
2D Painting Example
40000 Chips
Address Book
Affine Transformations
Analog Clock
Arrow Pad Example
Basic Drawing Example
Basic Graphics Layouts Example
Basic Layouts Example
Basic Sort/Filter Model Example
Bindable Properties Example
Blocking Fortune Client
Get Qt
Qt Account
Online Community
Blogs
User Guide
1 Issues 2 Search Results 3 Application Output 4 Compile Output 5 QML Debugger Console 6 General Messages 7 Version Control 8 Test Results

Figure 17. Qt Creator Running

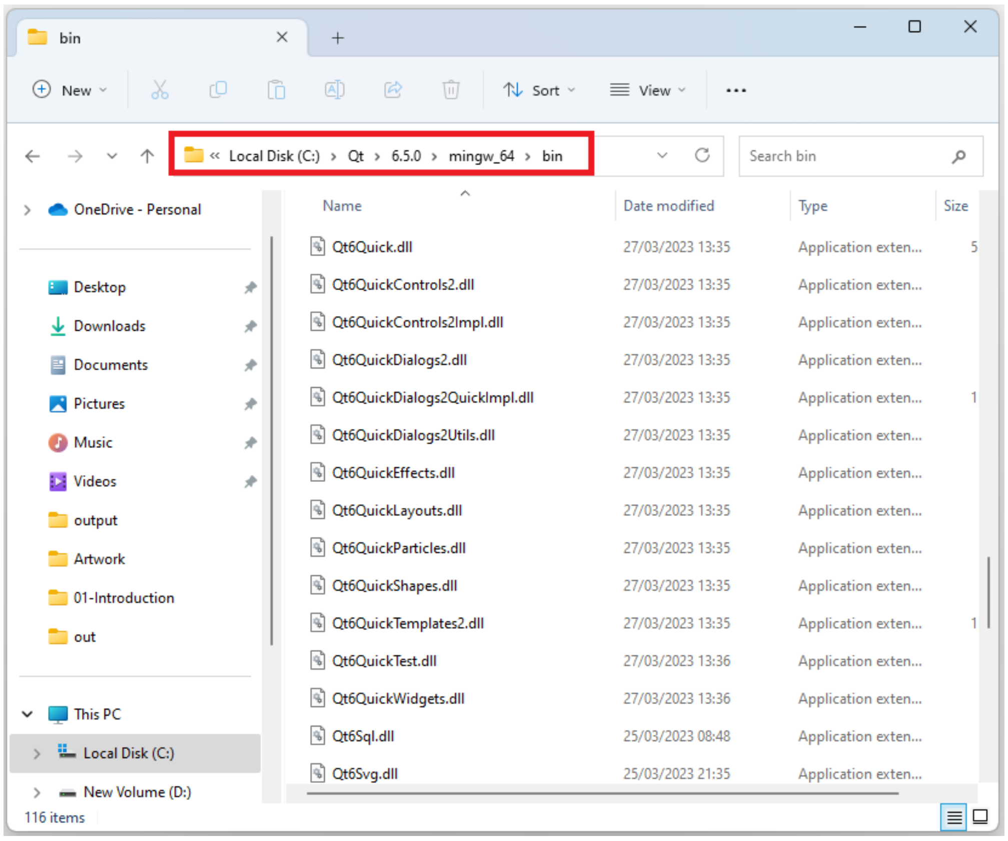
This means that you have a healthy Qt installation on your system. You can verify that the files are available in the path you specified earlier in the installation process. If you remember, mine was **C:\Qt** and going there, I can see my files.



Name	Date modified	Type	Size
6.5.0	29/03/2023 10:44	File folder	
dist	02/03/2023 14:31	File folder	
Docs	29/03/2023 13:16	File folder	
Examples	29/03/2023 13:18	File folder	
installerResources	12/05/2023 12:01	File folder	
Licenses	12/05/2023 12:01	File folder	
Tools	22/03/2022 14:20	File folder	
vcredist	10/06/2022 10:14	File folder	
components.xml	12/05/2023 12:01	XML File	
InstallationLog	12/05/2023 12:09	Text Document	
installer.dat	12/05/2023 12:01	DAT File	
licenseInfo	12/05/2023 12:01	Text Document	
MaintenanceTool.dat	12/05/2023 12:01	DAT File	

Figure 18. Qt Files on Local File System

Just browse through this folder to familiarize yourself with what files make up your Qt installation. For example, if you go in the **bin** folder, you'll see that we have folders related to the compilers we chose earlier. Go in the **bin** directory for each compiler and you'll see the actual binary files that make up the Qt framework!



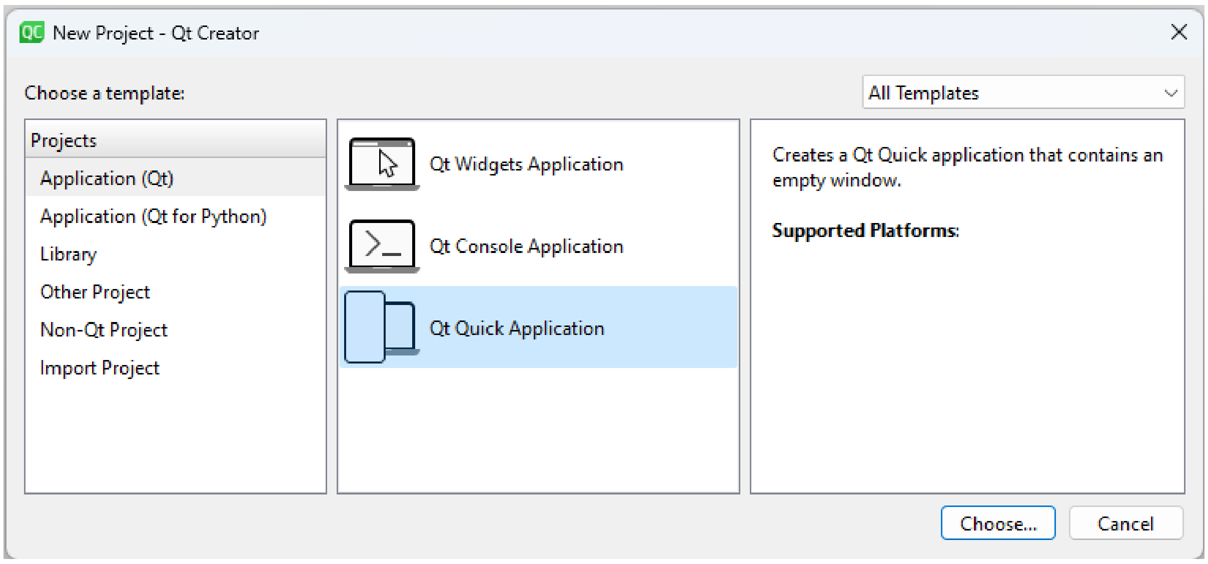
Name	Date modified	Type	Size
Qt6Quick.dll	27/03/2023 13:35	Application exten...	5
Qt6QuickControls2.dll	27/03/2023 13:35	Application exten...	
Qt6QuickControls2Impl.dll	27/03/2023 13:35	Application exten...	
Qt6QuickDialogs2.dll	27/03/2023 13:35	Application exten...	
Qt6QuickDialogs2QuickImpl.dll	27/03/2023 13:35	Application exten...	1
Qt6QuickDialogs2Utils.dll	27/03/2023 13:35	Application exten...	
Qt6QuickEffects.dll	27/03/2023 13:35	Application exten...	
Qt6QuickLayouts.dll	27/03/2023 13:35	Application exten...	
Qt6QuickParticles.dll	27/03/2023 13:35	Application exten...	
Qt6QuickShapes.dll	27/03/2023 13:35	Application exten...	
Qt6QuickTemplates2.dll	27/03/2023 13:35	Application exten...	1
Qt6QuickTest.dll	27/03/2023 13:36	Application exten...	
Qt6QuickWidgets.dll	27/03/2023 13:36	Application exten...	
Qt6Sql.dll	25/03/2023 08:48	Application exten...	
Qt6Svg.dll	25/03/2023 21:35	Application exten...	

Figure 19. Actual Qt Binary Files

Of course, these are specific to Windows, but you should see something similar if you happen to be on Linux or Mac. This completes our installation of Qt and we are ready to use it to build our first QML application in the next section.

Running your QML Application

In this section, we are going to build our first QML application and run it in Qt Creator. Along the way, we'll see some tips to go by when building your Qt QML applications. We'll also get to see all sorts of files that make up our project and get familiar with their syntax and structure. Let's get to work. Open Qt Creator and create a new project by going to **File › New Project**, choose **Application(Qt)→Qt Quick Application**.We'll start with this empty template.

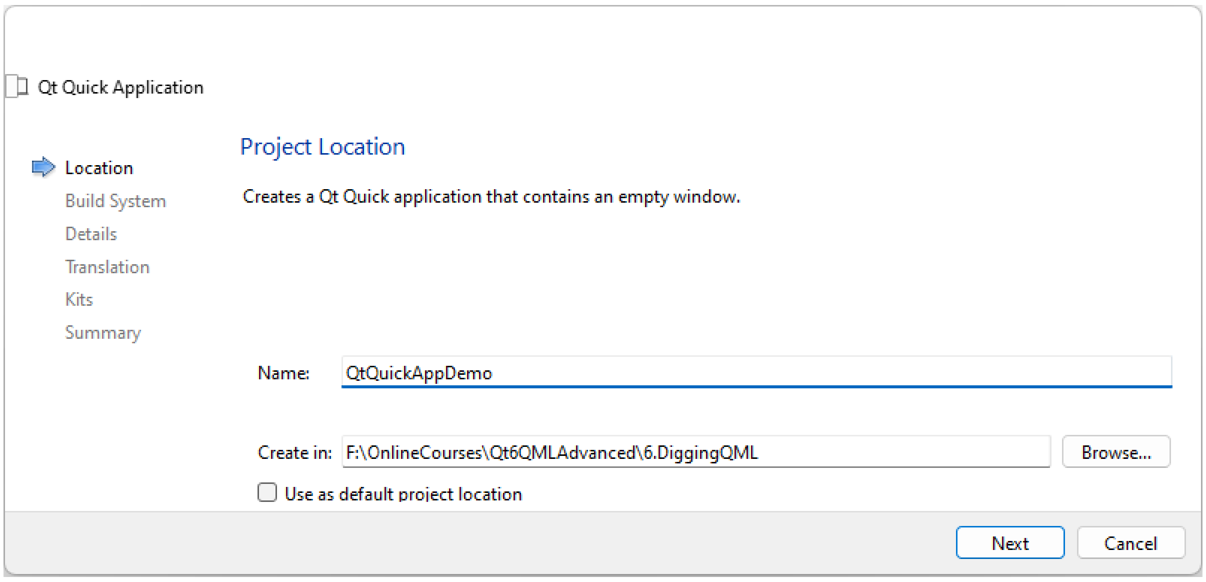


Creates a Qt Quick application that contains an empty window.

Supported Platforms:

Figure 20. Qt Creator Qt Quick Project Template

Click on **Choose**, and name the app **QtQuickAppDemo**.



Project Location

Creates a Qt Quick application that contains an empty window.

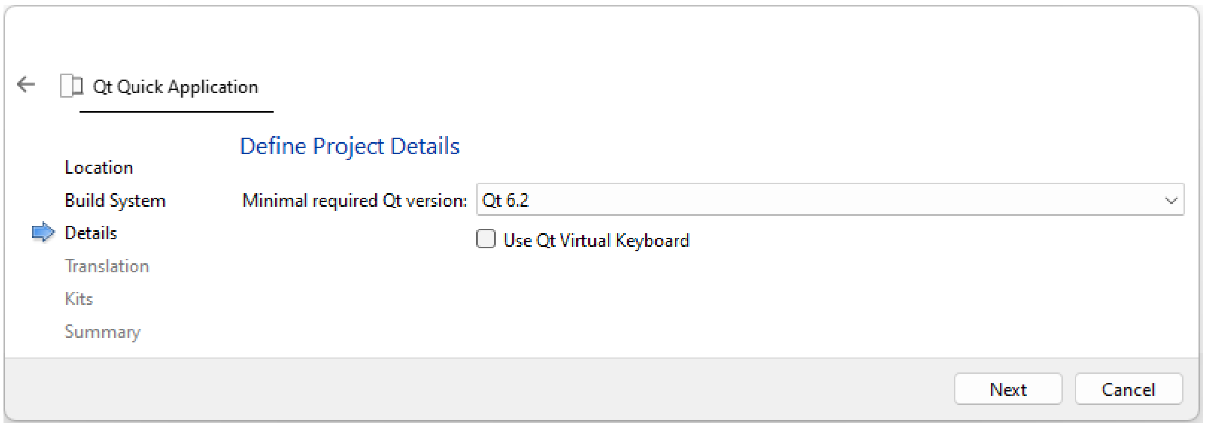
Name: QtQuickAppDemo

Create in: F:\OnlineCourses\Qt6QMLAdvanced\6.DiggingQML

Use as default project location

Figure 21. Qt Creator project name and location

Hit next, and choose CMake as the build system. The next screen may prompt you to choose the minimum required Qt version for your Qt project. If it does, leave that to the default and hit next. You should also leave the **Use Qt Virtual Keyboard** checkbox unchecked and hit next


Qt Quick Application

Location
Build System
Details
Translation
Kits
Summary

Define Project Details

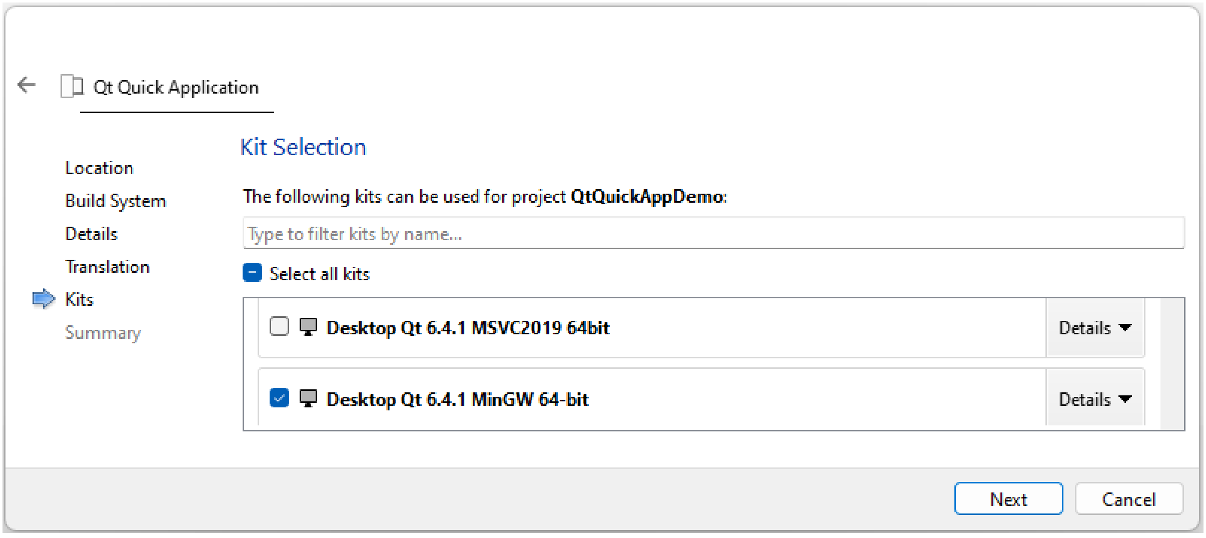
Minimal required Qt version: Qt 6.2

[ ] Use Qt Virtual Keyboard

Next Cancel

Figure 22. Qt Creator Project Minimum CMake Version

Leave the **Translation File** screen as is and hit next. You should be presented with the **Kit Selection** screen. This is where you specify the target device for your Qt Creator project and the set of tools needed to generate binaries for that device. Don't worry if all this sounds complicated for the moment. All this will be much more clear as we move forward in the book. We'll be mostly developing for desktop so I'll choose a kit with **Desktop** in its name. You may have more than one option depending on the kits installed on your system. The author is on Windows and chose to use **Desktop Qt 6.4.1 MinGW 64-bit** for the projects in the book.


Qt Quick Application

Location
Build System
Details
Translation
Kits
Summary

Kit Selection

The following kits can be used for project **QtQuickAppDemo**:

Type to filter kits by name...

Select all kits

[ ] **Desktop Qt 6.4.1 MSVC2019 64bit** Details

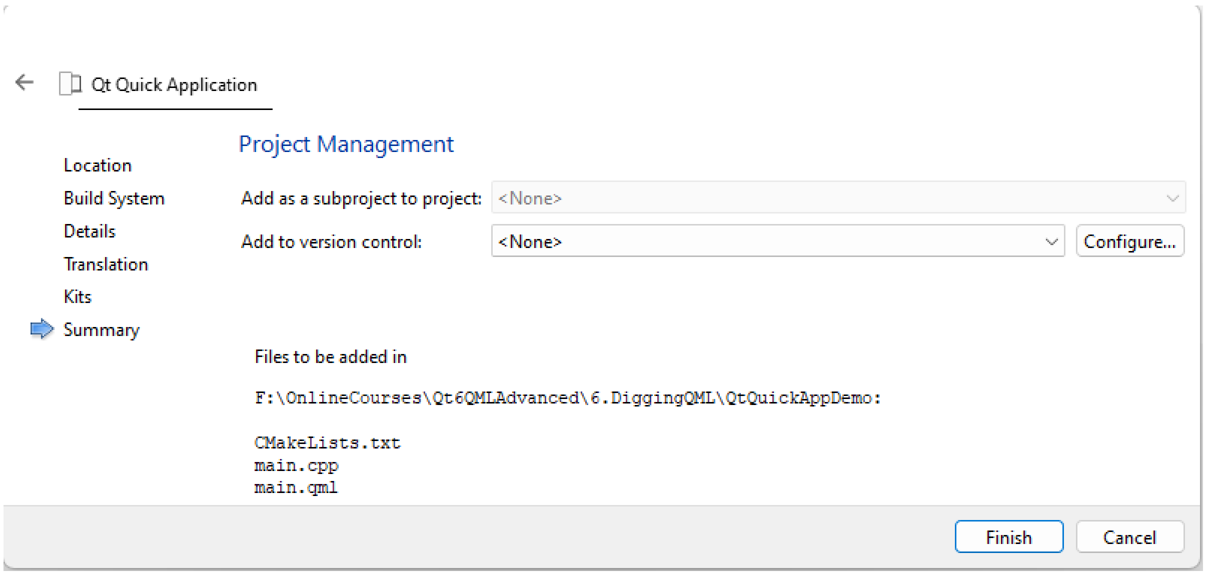
[✓] **Desktop Qt 6.4.1 MinGW 64-bit** Details

Next Cancel

Figure 23. Qt Creator Project Kit Selection

Hit next and you'll see a **Project Management** screen in which you get to choose whether some version control system will be used for the project. Qt supports several build systems and this is where you get to make your choice about that when creating the project. Git is the most popular one and you'll see it used a lot in your projects if you haven't already. It is tightly integrated into Qt Creator and you can take advantage of that to manage changes across files in your project, but we won't be doing that in this book. Choose **None** when it comes to the version control tool. The screen also shows an overview of the files that will be generated for your project. We can see three important files that will make up our project: `CMakeLists.txt`,`main.cpp` and `main.qml`.Keep these in mind as you'll be using them a lot in the book. It's also important to note the location where your project will be generated on your file system.

Some beginners will forget where their project files are saved and have trouble finding the files to work with the project later on. Don't be one of them!



Project Management

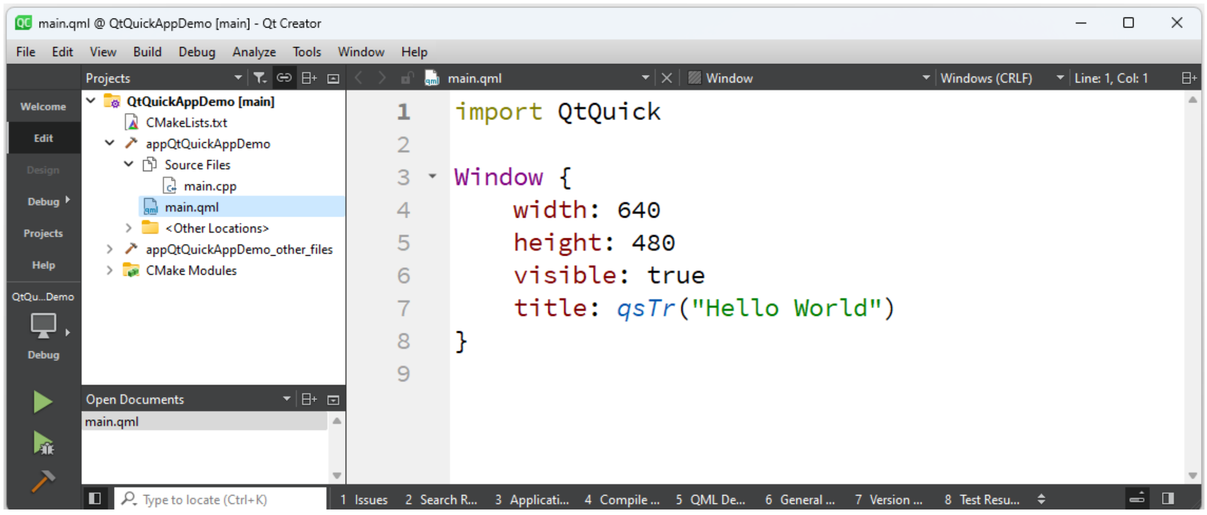
Add as a subproject to project: <None>

Add to version control: <None> Configure...

```
Files to be added in

F:\OnlineCourses\Qt6QMLAdvanced\6.DiggingQML\QtQuickAppDemo:

CMakeLists.txt
main.cpp
main.qml
```

Figure 24. Qt Creator Project Project Management Screen

Hit **Finish** and a project will be generated for you.



```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")
}
```

Figure 25. Qt Creator QML Project Opened

Now you have your QML project generated. The author invites you to click around expanding and collapsing things in the project explorer tab. Try to make it look like the figure above. Three files are of key importance here :

- CMakeLists.txt
- main.cpp
- main.qml

You don't need to know what they do at this moment but it won't hurt to just open these files and just peek at the content inside. In the next few sections, we'll be exploring each of these files in a bit more depth. We'll also be picking up a lot of tips and tricks you'll be using daily as a Qt/QML developer.

The CMakeLists.txt file

This is the main build system file. The build system is what takes all the files that make up your project and bundles them together before passing them to a C/C++ compiler that then generates an executable out of your project. Qt supports a bunch of build systems out of the box, the main ones being **CMake** and **QMake**. CMake is the dominant one currently and it's what is used by the developers of Qt themselves. QMake was favored by Qt developers in the past but it's on the path to deprecation as the Qt company (the guys behind Qt) chose CMake to go forward with. Long story short, if you're building a new Qt/QML app and have no reason to care about QMake, it's recommended to use CMake with Qt, and it's what we'll stick to in the book.

`CMakeLists.txt` is the first file CMake looks at to build your project and it contains info on what files make up your project, which compiler should be used for the project and other things we won't go into here. An example stripped-down version of our project's CMakeLists.txt file is shown below.

```
cmake_minimum_required(VERSION 3.16)

project(1-DemoApp VERSION 0.1 LANGUAGES CXX)

find_package(Qt6 6.2 COMPONENTS Quick REQUIRED)

qt_add_executable(app1-DemoApp
    main.cpp
)

qt_add_qml_module(app1-DemoApp
    URI 1-DemoApp
    VERSION 1.0
    QML_FILES main.qml
)

target_link_libraries(app1-DemoApp
    PRIVATE Qt6::Quick)
```

The `cmake_minimum_required(VERSION 3.16)` command sets up the minimum CMake version that should at least be installed on the computer where you're setting up your QML project. For example, if I had CMake 10.2 installed on my machine, I would get some errors if I tried to build this project.

Some of you may be wondering why we're spending all this time talking about CMake. This is important because CMake is such a key component of your QML project. And you'll often need to hop into the `CMakeLists.txt` file to apply the changes needed for your project to work.

The `project(1-DemoApp VERSION 0.1 LANGUAGES CXX)` command sets up the name of the project, which may be different from the name of the executable that'll be generated from your project. In this case, the name of the project is **1-DemoApp**. We also set up the major version and the minor version of the project. In our `CMakeLists.txt` file, the major version is 0 and the minor version is 1. The last part sets up the language that our project is using, and CXX means that our project is using C++.

Next up is

```
qt_add_executable(app1-DemoApp
    main.cpp
)
```

This is a Qt-specific command, which, among other things lets you set up the executable name your project will spit out when built, and the source files that make up your project. We'll get to learn about resource files later on in the book. If for example, our project was using other C++ files like `person.h` and `person.cpp`, they would show up in this command like so

```
qt_add_executable(app1-DemoApp
    main.cpp person.h person.cpp
)
```

The next Qt specific command we use is

```
qt_add_qml_module(app1-DemoApp
    URI 1-DemoApp
    VERSION 1.0
    QML_FILES main.qml
)
```

and for now, you can see it as the place where you'll be adding new QML and Javascript files to the project. For example, if we had another QML file in our project named `login_ui.qml`, we'd add it as shown below

```
qt_add_qml_module(app1-DemoApp
    URI 1-DemoApp
    VERSION 1.0
    QML_FILES main.qml login_ui.qml
)
```

The last two commands we need to look at are

```
find_package(Qt6 6.2 COMPONENTS Quick REQUIRED)
target_link_libraries(app1-DemoApp
    PRIVATE Qt6::Quick)
```

`find_package` will look for a Qt installation on your system and if it doesn't find it, your project will fail to build. Since our project will be using some Qt Quick features, we'll also tell it to look for the Qt Quick module specifically. Finding the Qt packages and Qt modules is one piece of the puzzle though; once `find_package` finds what it needs, we need to specifically tell our generated executable to link against Qt and the modules we need. You do that will the `target_link_libraries()` command. Notice that we want the `Qt6::Quick` module. If later on, we need other modules we'll be sure to edit these two commands in our `CMakeLists.txt` file. For example, later on in the book, we'll need to use the Qt Quick Controls Module, to give us ready-to-use visual components. For that, we'll need to bring the QuickControls2 module into our project with `find_package` and link against it with `target_link_libraries`.

```
find_package(Qt6 6.2 COMPONENTS Quick QuickControls2 REQUIRED)
target_link_libraries(app1-DemoApp
    PRIVATE Qt6::Quick Qt6::QuickControls2)
```

Just to recap, from our `CMakeList.txt` file, reproduced below for convenience,

```
cmake_minimum_required(VERSION 3.16)
project(1-DemoApp VERSION 0.1 LANGUAGES CXX)

find_package(Qt6 6.2 COMPONENTS Quick REQUIRED)

qt_add_executable(app1-DemoApp
    main.cpp
)

qt_add_qml_module(app1-DemoApp
    URI 1-DemoApp
    VERSION 1.0
    QML_FILES main.qml
)

target_link_libraries(app1-DemoApp
    PRIVATE Qt6::Quick)
```

we can gather that :

- Our project will be named **1-DemoApp**
- The executable that will be generated from the project will be named **app1-DemoApp**
- Our project is made up of a single C++ file: `main.cpp`
- Our project is made up of a single qml file: `main.qml`
- For our project to build successfully, we need a working Qt6 installation on our system, and our executable will be linking against the Qt6::Quick module.

What you should be aiming at now is to look at your `CMakeList.txt` file and pick up all this information instantly!

The main.cpp file

Now that we have some CMake basic powers under our belt, we can look at the `main.cpp` file that makes up our project

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

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

    QQmlApplicationEngine engine;
    const QUrl url(u"qrc:/1-DemoApp/main.qml"_qs);
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                     &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

Some of you may be already freaking out looking at this C++ code. No need to worry as this is almost the only point where you'll have to touch C++ in this book. This a QML book, but QML itself runs on top of C++, and most of what the code in our file here does is really handing over to your QML engine from C++. The file starts off including `QGuiApplication` and `QQmlApplicationEngine`. QGuiApplication is the wrapper class that represents our QML graphical user interface application. QQmlApplicationEngine is what we'll be using to load qml files and turn them into visual components on your screen.

It is possible to use Python as a backend language, in which case we would have main.py instead of main.cpp, but the backend language isn't important for this QML book. We went with C++ because support in Qt Creator is better compared to Python.

The main function is the entry point for any C/C++ application. In other words, when users run your QML application, execution will jump into the main function. Once into the main function, we create our QGuiApplication and objects

```
QGuiApplication app(argc, argv);
QQmlApplicationEngine engine;
```

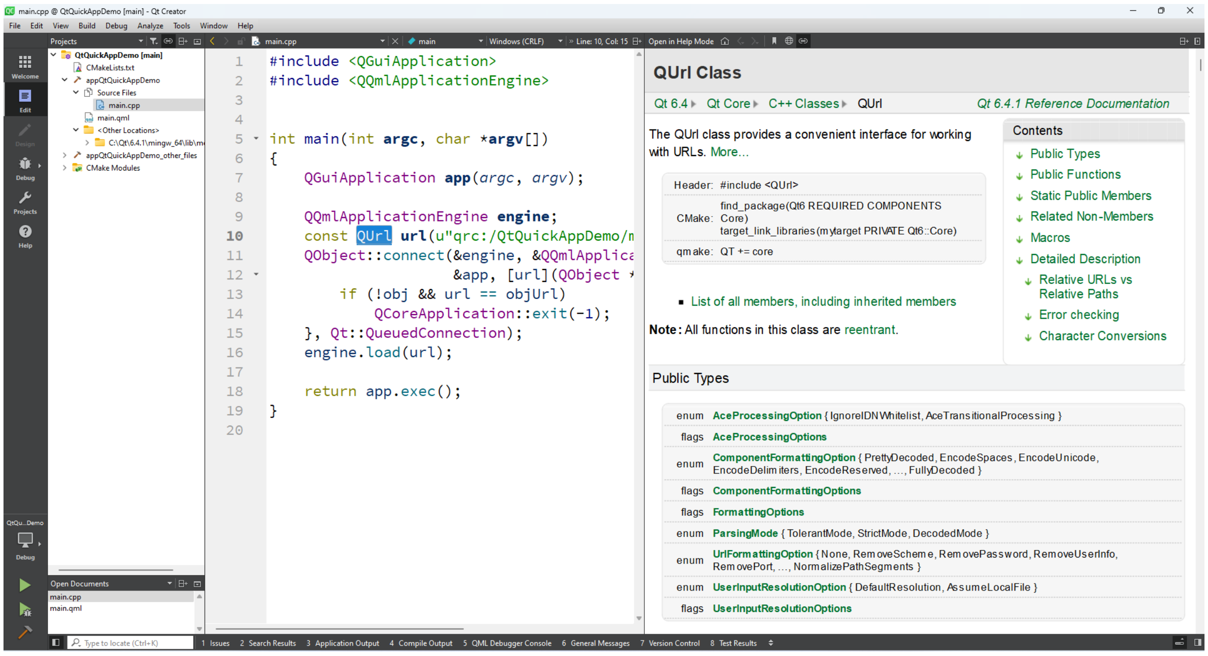
The QGuiApplication object is initialized with the parameters `argc`, and `argv` coming straight from the main function. Next up we set up the QUrl object which will be pointing to the location of the `main.qml` file in our project

```
const QUrl url(u"qrc:/1-DemoApp/main.qml"_qs);
```

Using the Documentation

Now that we are here, I'd like to bring your attention to one of the powers of working with Qt/QML; and that is **the documentation**! Qt is one of the best-documented projects the author has had to honor to work with, in their decade-long career as a professional developer. Using the docs is one of the best skills you'll need to develop to be an independent developer and be able to find solutions to problems faster and most importantly, **on your own**!

The Qt documentation is tightly integrated into Qt Creator and all you have to do in Qt Creator to learn more about any Qt type is to select it and hit `F1` on your keyboard. We just touched on the `QUrl` type and you might want to see with your own eyes what that type is all about. Just select `QUrl` in your main.cpp file and hit `F1`. You'll see another window open with the documentation you need.


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


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

    QQmlApplicationEngine engine;
    const QUrl url(u"qrc:/QtQuickAppDemo/m
    QObject::connect(&engine, &QQmlApplica
                     &app, [url](QObject *
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}
```

QUrl Class

Qt 6.4 ▸ Qt Core ▸ C++ Classes ▸ QUrl — *Qt 6.4.1 Reference Documentation*

The QUrl class provides a convenient interface for working with URLs. More...

Header:	#include <QUrl>
CMake:	find_package(Qt6 REQUIRED COMPONENTS Core) target_link_libraries(mytarget PRIVATE Qt6::Core)
qmake:	QT += core

- List of all members, including inherited members

Note: All functions in this class are reentrant.

Contents
- Public Types
- Public Functions
- Static Public Members
- Related Non-Members
- Macros
- Detailed Description
  - Relative URLs vs Relative Paths
  - Error checking
  - Character Conversions

Public Types

enum	AceProcessingOption { IgnoreIDNWhitelist, AceTransitionalProcessing }
flags	AceProcessingOptions
enum	ComponentFormattingOption { PrettyDecoded, EncodeSpaces, EncodeUnicode, EncodeDelimiters, EncodeReserved, ..., FullyDecoded }
flags	ComponentFormattingOptions
flags	FormattingOptions
enum	ParsingMode { TolerantMode, StrictMode, DecodedMode }
enum	UrlFormattingOption { None, RemoveScheme, RemovePassword, RemoveUserInfo, RemovePort, ..., NormalizePathSegments }
enum	UserInputResolutionOption { DefaultResolution, AssumeLocalFile }
flags	UserInputResolutionOptions

Figure 26. Qt Creator Documentation Page for QUrl

Another option, if you don't want to go through Qt Creator is to use the docs on the web. However the author personally finds it cumbersome to click through an endless chain of links to the qt docs website and they instead go through a search engine and just type in the type or class they're interested in. For example, if we type QUrl in the search engine, one of the first links to pop out is this pointing directly to the `QUrl` type we're interested in. From the docs, we can see that

> The QUrl class provides a convenient interface for working with URLs.

Another important piece of information to pick up from any docs page is the build system guide. From the docs, we can also see the block of text below

```
Header: #include <QUrl>
CMake:  find_package(Qt6 REQUIRED COMPONENTS Core)
    target_link_libraries(mytarget PRIVATE Qt6::Core)
qmake:  QT += core
```

Translated into English, this says that to use the QUrl type in your `.cpp` files, you need to have QUrl included. On top of that, if you're using CMake you need

```
find_package(Qt6 REQUIRED COMPONENTS Core)
target_link_libraries(mytarget PRIVATE Qt6::Core)
```

somewhere in your `CMakeLists.txt` file(s). If you're using QMake,you'll need

```
QT += core
```

somewhere in your .pro file.

Coming back to our `main.cpp` file, next we come across the next block of code:

```
QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                     &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
```

This provides a way for the engine object to let us know if it loaded the qml file properly. If the file was loaded properly, we move down in the qml file but if we fail to load it, we kill the application with the line

```
QCoreApplication::exit(-1);
```

The explanation above is what you should take if you don't know C and how Qt works with/uses C. If you need more however, the author invites you to join them as they read documentation for QQmlApplicationEngine::objectCreated.The docs say that

> This signal is emitted when an object finishes loading. If loading was successful, object contains a pointer to the loaded object, otherwise the pointer is NULL. The url to the component the object came from is also provided.
>
> — The Qt Docs

In English, we are connecting a slot to the QQmlApplicationEngine::objectCreated signal from our engine object. That signal is emitted when we successfully call the load method on our engine object

```
engine.load(url);
```

When `engine` finishes loading `url`,our signal is emitted and we respond in our lambda slot

```
[url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }
```

The body of the lambda does this test: if the obj pointer contains null, meaning that the loading was a failure and the url from the signal is the same as the one we originally loaded from, then we don't have a proper qml structure loaded in memory and we kill the application with

```
 QCoreApplication::exit(-1);
```

Want to know what this exit method does? Why not check it in the docs right now. The author can't encourage you enough to use the docs every chance you get. You'll even pick up new things, slowly cementing your knowledge and understanding of Qt and QML. The last part of the main.cpp file simply loads the actual qml file and starts the Qt event loop

```
engine.load(url);
return app.exec();
```

The call to `app.exec()` forces the app into a procedure that keeps looping around waiting for things to happen. For example, when we get to see our QML window on the screen, the event loop will be waiting for things to happen. This hopefully gives you an idea of what the main.cpp file in the project does. Next, we'll look at the main.qml file which is of special interest in this book.

The main.qml file

If you open your `main.qml` file you'll see code like below

```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")
}
```

It's what is generated by Qt Creator when you create your project from the Qt Quick project template. The code starts by importing the QtQuick module in the current qml file and then instantiating a Window object, which represents the window we'll see when we run this application. If we run the application, we'll see a window pop up on the screen like below

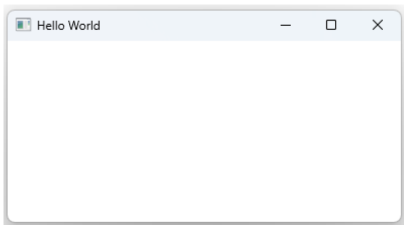


Figure 27. QML Starter Application Running

Looking at the code behind our window, you can get a glimpse at the QML syntax. Each QML file you'll work with will be made up of a number of elements; like `Window`, with a set of properties for the element nested in a pair of curly braces.

```
Window { // The window element
    width: 640 // The width property
    height: 480 // Another property
    visible: true
    title: qsTr("Hello World")
} // The closing curly brace
```

You can also move the curlies to the next line

```
Window // The window element
 { //The curly is on this line
    width: 640 // The width property
    height: 480 // Another property
    visible: true
    title: qsTr("Hello World")
} // The closing curly brace
```

From our code above,you can also gather that QML supports single-line comments through a pair of back slashes `//`.It also supports multi-line comments delimitated by `/*` and `*/`.

```
Window
 {
    //This is a single-line comment
    /*
    This is a multi-line comment that can
    span
    multiple
    lines
    */
    width: 640 // The width property
    height: 480 // Another property
    visible: true
    title: qsTr("Hello World")
}
```

Code that is commented out either through // or /* and */ is not processed by the QML engine and is meant to help you leave meaningful short descriptions of what your code does. Now that you've seen the application running, you might want to know more information on the Window element behind that app, or the properties like `width`, `height` and others in that element. What better way to do that fast, and on your own than browsing the documentation entry for that specific element! In the detailed description section of the docs, you see that

> The Window object creates a new top-level window for a Qt Quick scene. It automatically sets up the window for use with QtQuick graphical types.
>
> — The Qt Docs

The author encourages the reader to try and read the full description of this element just to familiarize themselves with the style used by the Qt docs. Further down the docs even state that

> When the user attempts to close a window, the closing signal will be emitted. You can force the window to stay open (for example to prompt the user to save changes) by writing an onClosing handler that sets close.accepted = false unless it's safe to close the window (for example, because there are no more unsaved changes).
>
> — The Qt Docs

A little trick you can use to frustrate your users and prevent your application from quitting when they close the window! On a serious note, don't do though. This is something you just learned by looking at the docs description, using a few seconds of your time! Qt even goes as far as trying to give you little snippets of code to help you understand a topic at hand. Here is the code from the docs for ease of reference

```
onClosing: (close) => {
    if (document.changed) {
        close.accepted = false
        confirmExitPopup.open()
    }
}
// The confirmExitPopup allows user to save or discard the document,
// or to cancel the closing.
```

Please note that you're not required to understand what this code does at this point in the book; the author is trying to emphasize that the docs are an invaluable resource you should learn to take full advantage of in your career as a Qt/QML developer. The docs go further and give an exhaustive list of properties supported by the element. You can click on each of these properties to have more information. On top of that, we also see the signals and methods we can use with the element.

Now that you have run your first basic QML application, have a basic understanding of the moving parts that make up your app, and most importantly know a bit about using the docs, let's modify the application to make it a proper `Hello World` application. We want it to display text in the middle of the window and the text should say `Hello World!`.The text should be red and have a fairly visible size.

QML provides a Text element we can use to display text in our QML applications.The docs say that

> Text items can display both plain and rich text. For example, red text with a specific font and size can be defined like this
>
> — The Qt Docs

```
Text {
    text: "Hello World!"
    font.family: "Helvetica"
    font.pointSize: 24
    color: "red"
}
```

The text property specifies the actual text that will be displayed by the Text element. `font.family` is a grouped property that helps us specify fonts for Text elements. We'll get a chance to talk more about how grouped properties work later on in the book.`font.pointSize` is also a grouped property we can use to control how big the text will be. The last property is used to control the color of the text. QML supports color names like `red`, `green` and `blue`, but it can also use color hex codes like `#123abc`.

We can grab the `Text` element snippet from the docs and put that in our code in the `main.qml` file, just below the title property like below

```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")
    Text {
        text: "Hello World!"
        font.family: "Helvetica"
        font.pointSize: 24
        color: "red"
    }
}
```

If we run the app, we'll see some `Hello World!` text in red, shown in the top-left corner of the window.

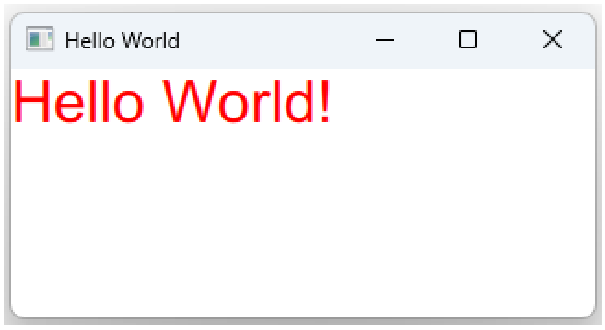


Figure 28. QML Hello World Application Running

This is closer to what we want, but we're not there quite yet. We want the text to show up in the middle of the screen. We can achieve that by instructing the Text element to position itself right in the middle of the window that contains it, r its parent. Anchors are one of the mechanisms in place to help us position elements relative to others. We modify our code like below

```
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")
    Text {
        text: "Hello World!"
        font.family: "Helvetica"
        font.pointSize: 24
        color: "red"
        anchors.centerIn: parent
    }
}
```

and run the application.

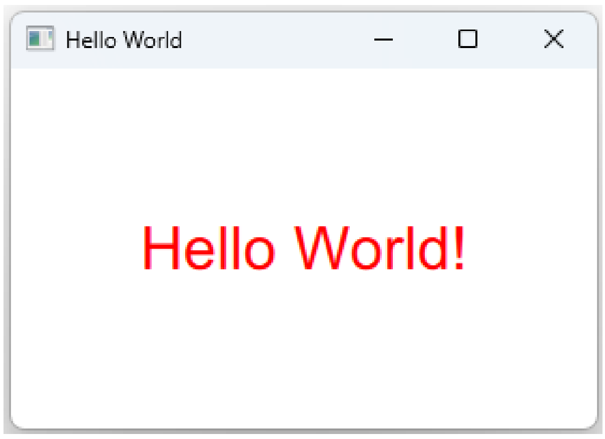


Figure 29. QML Hello World Application With Text In the Center

Now we can see our text in the middle because of the line `anchors.centerIn: parent` we added inside the Text element. We'll learn all about anchors later on in the book and all we want here is to have some interesting basic graphical user interface running and familiarize yourself with the moving parts involved in getting your QML project to run. We have seen some QML doing interesting things in this section but we need to learn about other constructs in the QML syntax to be able to build more practical applications. We'll do that in the next chapter.

Chapter Summary

You've made it this far. Congratulations! The main goal for the chapter was to get your feet wet and along the way build your very first QML application and see it running on your host environment. We saw that Qt is a cross-platform application development framework one can use to develop once and build for a host of devices. Qt can be used to target desktop platforms like Windows, Linux and Mac but also mobile and embedded devices. We saw that Qt provides two user-facing APIs to build graphical user interfaces: Qt Widgets through a C++ or Python API and QML. While Qt Widgets can technically run on mobile and embedded, they are commonly used to target desktop operating systems. QML was designed to be more appealing to designers with its declarative nature and it is currently used to build highly fluid and dynamic user interfaces for mobile and embedded applications. It's also important to remember that the designer-developer workflow: Designers cook up the UI and export it. Developers then plug the UI into a C++ or Python back-end that does the heavy lifting. With basic knowledge out of the way, installed Qt on our host environment and moved on to build our very first QML application through Qt Creator, the IDE of choice for Qt projects. We got introduced to the three main file types that make up your QML project: the `CMakeLists.txt` file, the `main.cpp` file and the `main.qml` file. `CMakeLists.txt` is the file used by the build system generator, CMake, to put together all the files that make up the project. `main.cpp` is the file where C++ hands control over to QML by loading the `main.qml` file. Last but not least, main.qml is where your QML code will live. We got a chance to look at the common commands in the `CMakeLists.txt` file, saw that the docs are an invaluable resource and you should take advantage of as much as possible and got to experience that QML syntax is just a number of elements that may be nested into each other, with properties nested in pairs of curly braces. Armed with this new knowledge, we're ready to start dissecting the QML syntax in the next chapter.

Chapter 2: Dissecting the QML Syntax

We have used some QML in the last chapter but the focus was on giving you a birds eye's view on your Qt Quick project. In this one, we'll zoom in on the specifics of the QML language syntax and basic constructs. You'll appreciate the declarative nature of QML, where each QML file is a hierarchy of elements nested into each other. We'll explore the basic data types offered by QML and let you play with them in a Qt Quick project. We will see that property bindings are a flexible way to tune in on the changes happening in a section of your application and respond in a different section of the application. Next up we'll look a the Qt global object and some of the interesting facilities it puts at your fingertips. We'll close the chapter by exploring property change handlers, another QML construct that may save your day for certain kinds of problems.

Syntax Overview

QML: A declarative language

QML is often described as a declarative language because it allows developers to specify what they want their user interface to look like, without having to worry about how it is implemented. In other words, QML focuses on describing the properties, behaviors, and relationships of the elements in a user interface, rather than on the procedural steps required to create it. This makes it easier for developers to create and maintain complex user interfaces since they can focus on the high-level design of the interface, rather than on the low-level details of how it is constructed.

QML achieves its declarative approach through the use of a hierarchical structure of elements, each of which can have properties and child elements. These elements can be combined and nested in a variety of ways to create complex, multi-layered user interfaces. Overall, the declarative nature of QML allows developers to write less code and create user interfaces more quickly and easily than with other approaches.

Basic Syntax

We have seen the basic files that make up your QML project in the last chapter. In this one, we'll explore the QML syntax in a little more depth. Rather than giving you a full lecture on QML, we'll build a simple project and explore some new ideas as we go along. The project is something like below

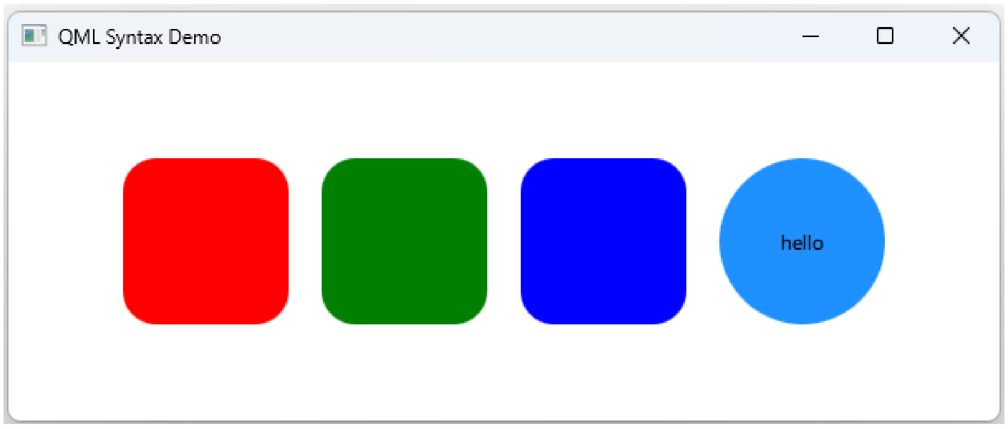


Figure 30. The Project We'll be Building Towards

Notice that we have a parent window with the window **tile QML Syntax Demo** and in the center of the window we have three rectangles with rounded corners and a circle. The elements are arranged in the window from left to right. Our job here is to explore the constructs QML provides to build something like this. Create a brand new QML project, give it a name and save it somewhere on your drive. The project should contain three main files: CMakeLists.txt, main.cpp and main.qml. Open your main.qml file and it should look something like

```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")
}
```

The `import QtQuick` line brings in the QtQuick module for use in the current QML file. You might already have the idea that QML code is organized into a hierarchy of elements, like **Window**, and each element will have properties nested in a pair of `{}`.In this case, the Window element has four properties nested in its pair of `{}`.`width` and `height` specify how wide and high the window will be. You should of course try to check the official documentation for any property of interest. And who knows, you might learn something new doing that. Let's look at the docs for the Window element and its properties.

Hop over to your favorite search engine and type **QML Window qt6** in the search bar. One of the first links to pop up should be pointing to the official docs for the Window element. Alternatively in Qt Creator, you can just select **Window** and hit F1.

Once you have the docs pane open after hitting F1, you can find a button saying **Open in Help Mode** in the top left corner of the docs window. If you click on that, the docs will show up in a full window for better visibility

We encourage you to read the full detailed description of the Window element right now. Even if some parts of it don't make sense at this moment, just develop the habit of reading the docs. Some of the things not making sense now will pop back into your mind as we go forward in the book, your brain will connect the dots and you'll develop a good understanding of the concepts.

Being lazy to read the docs in your learning journey will handicap your ability to find solutions to problems faster and on your own. Make an active effort to read the docs and you'll even learn more than any book or learning material can cover.

On our docs page for the Window element, if you go to the **Properties** section you'll see a list of properties for our element of interest. Click on `width` and look at what the docs have to say. In addition to the width and height properties, we learn about the x and y properties and are informed that these are measured relative to the screen size.

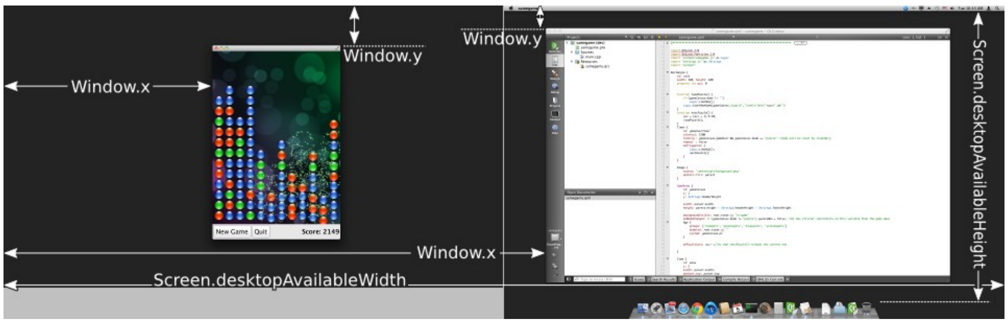


Figure 31. Screen and Window Dimensions from the Docs

The Window element also has a **title** property we can use to control the text showing up in the title bar of the Window. Let's play with that by changing our code like so.

```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("QML Syntax Demo"")
}
```

If you run the application now, you'll see that the title bar will be saying **QML Syntax Demo**.

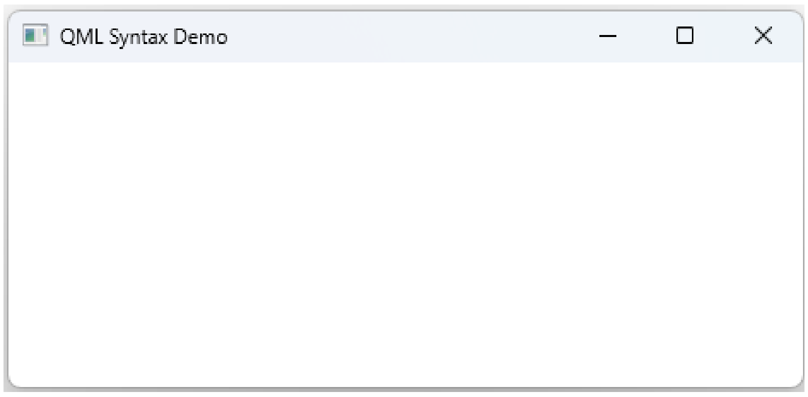
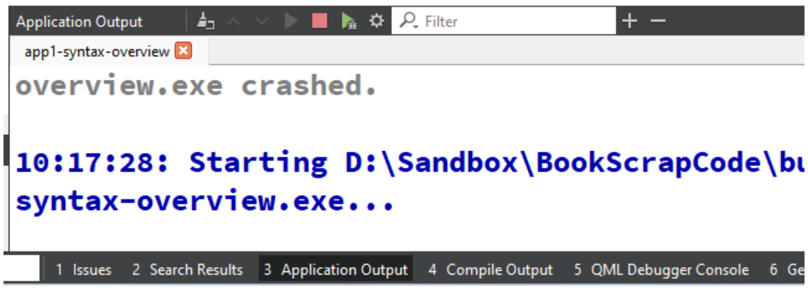


Figure 32. Window Element with Custom Title

Notice that we also have a `visible` property in our Window. Try to change its value to `false` and run the application. Nothing will show up on the screen. Why is that? Try to read the docs to figure that out. You'll have another problem at your hands after you run this **invisible** application of yours. How do you stop it? At the bottom side of your running Qt Creator instance, you'll see several tabs and one of them is **Application Output**.If you activate it you'll see a red square button. Hover on top of that button with your cursor and it'll say **Stop Running Application**. That's exactly what it does it'll stop any application you started from Qt Creator even if that app doesn't have any visible components to it. Use that to stop your **headless** application.



```
overview.exe crashed.

10:17:28: Starting D:\Sandbox\BookScrapCode\bu
syntax-overview.exe...
```

Figure 33. Stop Running Application from Qt Creator

Now we want to move a step further toward our goal application. We'll lay out the rectangles from left to right, inside our Window element. There is one tool we can use to do that, the Row element. If you look it up in the docs you'll see that it's used to lay things out horizontally or from left to right. We want three rectangles to start with, and they have background colors of red, green and blue respectively. Let's modify our main.qml file to add a Row inside the Window element

```
import QtQuick
Window {

    id : rootId
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Syntax Demo")

    Row {
        id : row1
        Rectangle {
            id : redRectId
            width: 100; height: 100
            color: "red"
        }

        Rectangle {
            id : greenRectId
            width: 100; height: 100
            color: "green"
        }
        Rectangle {
            id : blueRectId
            width: 100; height: 100
            color: "blue"
        }
    }
}
```

One would expect to run the application and see our rectangles inside the Window element, laid out from left to right. Let's run the app and see what happens.

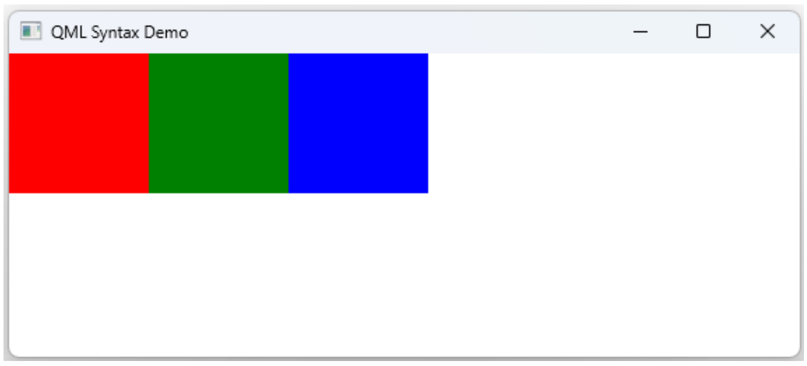


Figure 34. Row Element Showing up in Top Left Corner

The rectangles are showing up, but they're cramped in the top left corner of the Window element. This is the default behavior in QML; if you don't explicitly specify the position of the child element one way or another, QML will position that in the top left corner of the parent element. In other words, the top leftmost point of our Row element will be at the same position as the top-left point of the Window element.

> If you don't explicitly specify the position of the child element one way or another, QML will position that in the top left corner of the parent element.

We need to tell the Row element to position itself in the center of the Window element. One way to do that is through the **Anchor System**.If we add the line `anchors.centerIn: parent` in our Row element like below

```
import QtQuick
Window {

    id : rootId
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Syntax Demo")

    Row {
        id : row1
        anchors.centerIn: parent //The Row element will be centered in its parent,the
Window in this case.
        Rectangle {
            id : redRectId
            width: 100; height: 100
            color: "red"
        }

        Rectangle {
            id : greenRectId
            width: 100; height: 100
            color: "green"
        }
        Rectangle {
            id : blueRectId
            width: 100; height: 100
            color: "blue"
        }
    }
}
```

and run the application

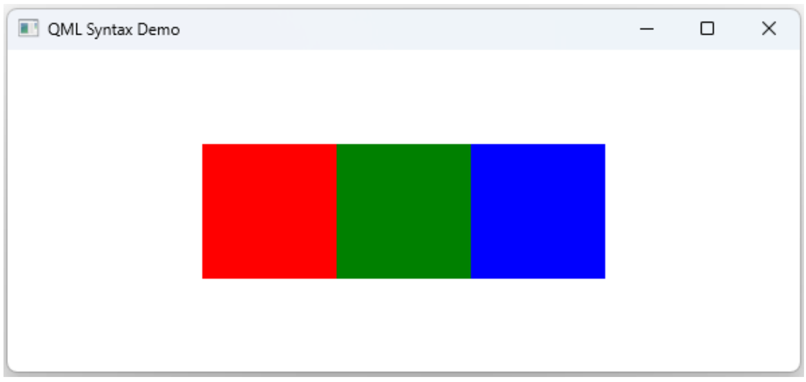


Figure 35. Row Element Showing up in the Center

we'll see our Row element with three rectangles centered in the Window. Anchors are a system for positioning elements relative to others with a very elegant syntax. For example, specifying that you want the right side of one element aligned with the left side of another element. But we're getting ahead of ourselves here at this point. For now, you can think that the line `anchors.centerIn: parent` centers the current element in the parent.

But we can do even better. At this point, I would even recommend you look at the docs page for the Row and Rectangle elements and see some properties you can play with just right now. The author personally likes round borders and thinks it would be nice to add some spacing between the rectangles. We'll use the `spacing` property from Row and the `radius` property from Rectangle. Here is our code with the modifications applied.

```
import QtQuick
Window {

    id : rootId
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Syntax Demo")

    Row {
        id : row1
        anchors.centerIn: parent
        spacing : 20 //Add a spacing of 20 between the elements in our Row
        Rectangle {
            id : redRectId
            width: 100; height: 100
            color: "red"
            radius : 20 // Make the borders of the rectangle round
        }

        Rectangle {
            id : greenRectId
            width: 100; height: 100
            color: "green"
            radius : 20 // Make the borders of the rectangle round

        }
        Rectangle {
            id : blueRectId
            width: 100; height: 100
            color: "blue"
            radius : 20 // Make the borders of the rectangle round

        }
    }
}
```

Running the app, you can see some space inserted between the rectangles and the rectangle borders are no longer sharp.

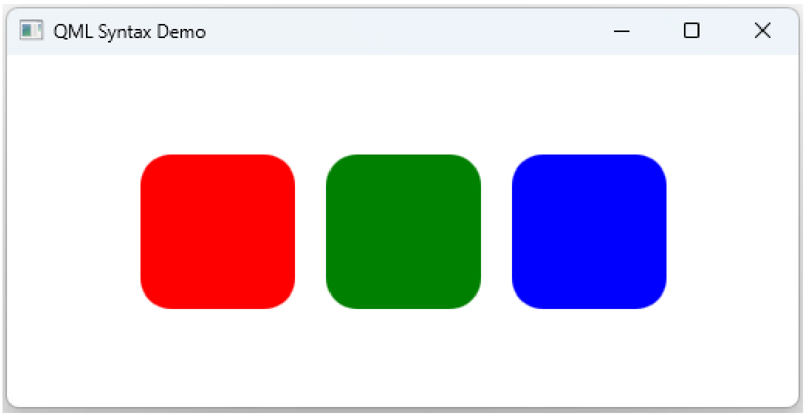


Figure 36. Row Spacing and Rounded Corners for Rectangle

Responding To Signals Through Handlers

We want to add some interactivity to our rectangles. How about responding in some way when somebody clicks on each rectangle? There is an element in QML that specializes in handling mouse events: `MouseArea`.Now is good to check it out in the docs **on your own**. You work with MouseArea elements by putting them inside visual elements you want to handle mouse events for. For example, if we insert one in the red rectangle like below

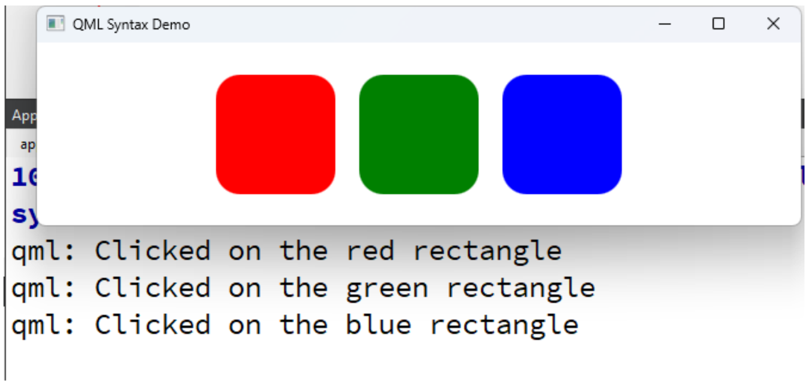
```
Rectangle {
    id : redRectId
    width: 100; height: 100
    color: "red"
    radius: 20
    MouseArea {
        anchors.fill: parent // Use anchors to make the MouseArea fill the Rectangle
        onClicked: {
            console.log("Clicked on the red rectangle")
        }
    }
}
```

we get the ability to respond to the clicked signal from MouseArea with our `onClicked` handler. A handler is a piece of code whose purpose is to respond to something happening in your application. Different QML elements can emit, or fire signals when things happen. You respond to these signals through handlers. Some of you may be asking the million-dollar question: "How do I know the signals that an element can emit?".The answer is **the documentation**. If you browse the docs page for MouseArea and take a look at the **Signals** section, you'll see a list of signals one can work with and one of them is `clicked`.We'll get a chance to learn more about signals and handlers later on in the book, for now, we just want the ability to print out some message when you click on the rectangles. Modify the other rectangles by putting in a MouseArea element in each, and printing a message in the onClicked handler. We print messages just like we do in regular Javascript syntax through `console.log()`.

From now on, we'll just show code snippets relevant to the current discussion at hand to keep the pages of the book reasonably readable. The full runnable source code will always be available in the git repository of the book.

```
Rectangle {
    id : greenRectId
    width: 100; height: 100
    color: "green"
    radius: 20
    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the green rectangle")
        }
    }

}
Rectangle {
    id : blueRectId
    width: 100; height: 100
    color: "blue"
    radius: 20
    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the blue rectangle")
        }
    }
}
```

Running the application, we see our messages printed out in the Application Output Pane when we click on the rectangles.



```
qml: Clicked on the red rectangle
qml: Clicked on the green rectangle
qml: Clicked on the blue rectangle
```

Figure 37. Clickable Rectangles

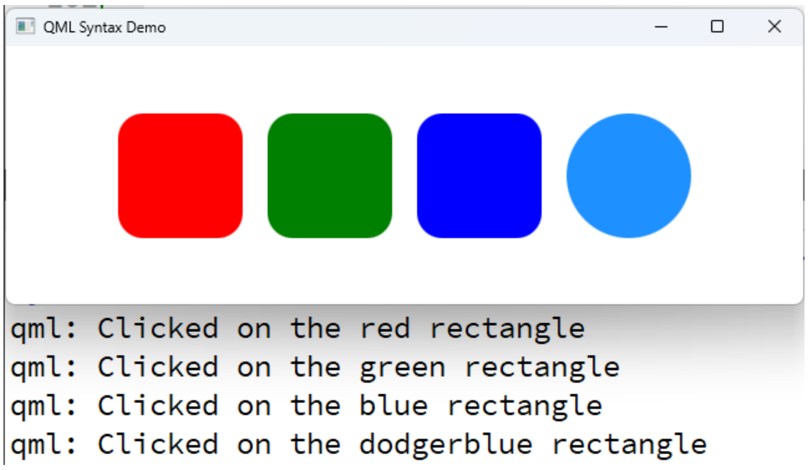
Messages you print out through console.log() will show up in the Application Output Pane in Qt Creator

Looking back at our target application, we need to add a circle after the blue rectangle. Some of you may be thinking of going through the docs to search for a Circle element but we don't have such a thing in QML. The trick we use to create circles in QML is to round the rectangle borders enough for the rectangles to turn into circles. To play with this, just pick one of the rectangles that you already have and change its radius value to 100, run the application and see what happens!

After you do this little experiment, you'll have no problem adding the circle to our Row element just like below

```
Row {
    id : row1
    Rectangle {
        id : redRectId
        //Rest of code ommitted for brevity
    }
    Rectangle {
        id : greenRectId
        //Rest of code ommitted for brevity
    }
    Rectangle {
        id : blueRectId
        //Rest of code ommitted for brevity
    }
    Rectangle {
        width: 100; height: 100
        color: "dodgerblue"
        radius: 100

        MouseArea {
            anchors.fill: parent
            onClicked: {
                console.log("Clicked on the dodgerblue circle")
            }
        }
    }
}
```

Running the application, we can see our dodgerblue fake circle, and you'll find that it's clickable, just like the other rectangles.



```
qml: Clicked on the red rectangle
qml: Clicked on the green rectangle
qml: Clicked on the blue rectangle
qml: Clicked on the dodgerblue rectangle
```

Figure 38. Adding a Circle

We also want to see some text inside the circle in our Row.QML offers a `Text` element we can use. You use it just like any other QML element, but putting that inside the element where you want it to show up, and specifying how it will be positioned. Remember that if you don't explicitly specify how your element should be positioned, it'll show up in the top left corner of its parent by default. Add a Text element inside the dodgerblue rectangle like below

```
Rectangle {
    id : textRectId
    width: 100; height: 100
    color: "dodgerblue"
    radius: 100

    Text {
        id : textId
        anchors.centerIn: parent
        text:"hello"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the dodgerblue circle")
        }
    }
}
```

and run the application, we'll have our text inside the circle. The text will be centered in the circle because will tell the Text element to position itself right in the center of its parent: the dodgerblue rectangle. You should also notice that we used the `text` property from the Text element. That's what you use to specify which text will show up in the Text element.

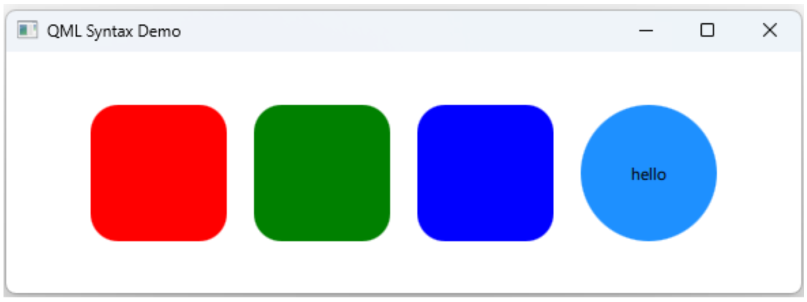


Figure 39. Text Centered in the Circle

Using The id Property in Foreign Elements

Now we want our click handlers to do something more interesting. How about getting the circle text showing the color of the rectangle we just clicked on? For example, if we click on red, the circle should say red, if we click on green the circle should say green and so on. Meet the `id` property. This is a property that uniquely identifies each QML element in your QML code. We have been using these already but didn't have any real need for them until now. Using the next piece of code as an example

```
Rectangle {
    id : blueRectId
    width: 100; height: 100
    color: "blue"
    radius: 20
    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the blue rectangle")
        }
    }

}

Rectangle {
    id : textRectId
    width: 100; height: 100
    color: "dodgerblue"
    radius: 100

    Text {
        id : textId
        anchors.centerIn: parent
        text:"hello"
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the dodgerblue circle")
        }
    }
}
```

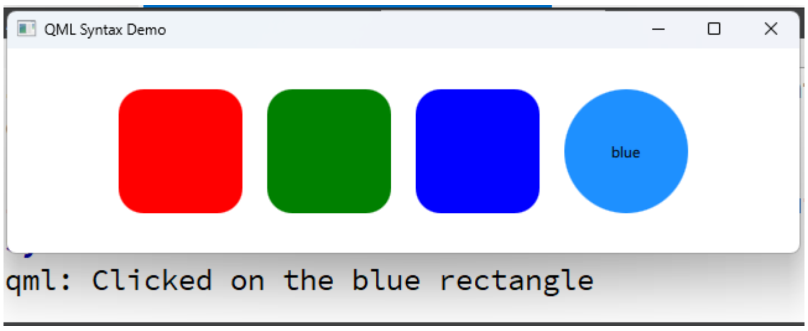
we can uniquely identify the blue rectangle anywhere in our qml file by `blueRectId`.Our good circle can now be identified using `textRectId`, alluding to the fact that its purpose is to show text. Do notice that our Text element also has an id: `textId`.

Do use descriptive and meaningful names for your IDs.It makes it easier to read and understand your code. You don't have to use IDs on every element in your QML file, but you'll need them especially if you need to manipulate the element from other elements in the QML element hierarchy.

Now we can go back to our question. How do we change the text of `textId`, from a click handler nested inside `blueRectId`? The `id` property comes in handy here. In our `blueRectId` click handler, we can say something like

```
textId.text = "blue"
```

when someone clicks on the blue rectangle. If you do that and run the application, you'll see that the text in the circle changes to "blue" when you click on the blue rectangle.



```
qml: Clicked on the blue rectangle
```

Figure 40. Using ID To Change Properties

I'll leave it to you to do the same for the red and green rectangles. Of course, you can always reference the git repository for the final runnable code.

Property Bindings

Our example is working fine and we're able to change the circle text from either of the rectangles. But there is an arguably better way to do this. And I'll take that chance and introduce you to **Property Bindings**. Property bindings are a mechanism in QML to make a property depend on another one. We'll get a chance to learn all about them later on in the course, but we'll look at the basics here nevertheless. We'll modify our code in three ways. First, we'll define a custom string property to hold the text that'll show up in our Text element.

```
Window {

    id : rootId
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Syntax Demo")

    //Define a custom property.Its tpye is string and the value is "hello"
    property string textToShow: "hello"
}
```

This is the syntax we use in QML to define properties. Second, we'll use that property to feed a value to the Text element

```
Text {
    id : textId
    anchors.centerIn: parent

    //Property binding : We're bounding the value of the text property to the
textToShow property
    text : textToShow
}
```

The key line here is `text : textToShow`.This is not just assigning the current value in `textToshow` to text. It is a "long-term" commitment saying that even later on in the running time of the program, if the value of textToShow changes, the changes will be reflected in the text value of textId and the text will change.

Lastly, well modify the value of `textToShow` whenever either of the rectangles is clicked.

```
Rectangle {
    id : blueRectId
    width: 100; height: 100
    color: "blue"
    radius: 20
    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the blue rectangle")
            textToShow = "blue"
        }
    }
}
```

To try out property binding here, you should remove the previous code that

> changed the text in textId directly through the id.

The current state of our code is reproduced below for ease of reference

```
import QtQuick

Window {

    id : rootId
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Syntax Demo")

    //Set up a custom property.The type is string and the value is "hello"
    property string textToShow: "hello"

    Row {
        id : row1
        anchors.centerIn: parent
        spacing: 20

        Rectangle {
            id : redRectId
            width: 100; height: 100
            color: "red"
            radius: 20
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    console.log("Clicked on the red rectangle")
                    //Change the value of the textToShow property.The changes will
show up in any other property that is bound to textToShow
                    textToShow = "red"
                }
            }
        }

        Rectangle {
            id : greenRectId
            width: 100; height: 100
            color: "green"
            radius: 20
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    console.log("Clicked on the green rectangle")
                    //Change the value of the textToShow property.The changes will
show up in any other property that is bound to textToShow
                    textToShow = "green"
                }
            }

        }
        Rectangle {
            id : blueRectId
            width: 100; height: 100
            color: "blue"
            radius: 20
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    console.log("Clicked on the blue rectangle")
                    //Change the value of the textToShow property.The changes will
show up in any other property that is bound to textToShow
                    textToShow = "blue"
                }
            }

        }

        Rectangle {
            id : textRectId
            width: 100; height: 100
            color: "dodgerblue"
            radius: 100

            Text {
                id : textId
                anchors.centerIn: parent
                text : textToShow
            }

            MouseArea {
                anchors.fill: parent
                onClicked: {
                    console.log("Clicked on the dodgerblue circle")
                }
            }
        }
    }

}
```

If you run the code, you'll see that the behavior is the same as we had by directly changing the value in `textId.text` through the id property. Click on red, the text says red. Click on green the text says green. And you can see that the changes in our rectangle click handlers are being automatically propagated to the `textId.text` property that is bound to `textToShow`.

We achieved the same thing by either changing the `textId.text` value in our click

> handlers or indirectly going through the `textToShow` property with property binding. Which one is better? I hear some of you ask. This is a design choice you'll have to make as a developer. If you don't need property binding, changing values directly makes your code a bit more readable, but property binding has its advantages and it's used all over the place in qml code out there. So you'll pick what works best for your project.

I wouldn't close off this section without showing you that you can go bad and break bindings! Yes. You can break a binding. And you do that by **assigning a static value to a property that was previously bound to another property**. I would read that at least twice if I were you. Let's break our binding when our circle is clicked.

```
Rectangle {
    id : textRectId
    width: 100; height: 100
    color: "dodgerblue"
    radius: 100

    Text {
        id : textId
        anchors.centerIn: parent
        text : textToShow
    }


    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the dodgerblue circle")
            //Break the binding by assigning a static value to textId.text
            textId.text = "broken"

        }
    }
}
```

If you run the application and click on `redRectId`, the text will change to "red".Click on `blueRectId` and the text changes to "blue". Click on `greenRectId` and the text changes to "green". Click on `textRectId` and the text changes to "broken". Now the fun begins: If you click back on either of the rectangles, nothing will happen because we broke the binding between `textId.text` and `textToShow` when we assigned "broken" to `textId.text`.You can say that after you click on the circle, `textId.text` no longer cares about `textToShow` and it'll fully ignore its changes. We'll explore property binding in more detail later on and see how to re-assign to a bound property without breaking the binding.

Exploring Data Types

QML Basic Types

QML supports a good number of data types. Most of the basic types you see in other languages like C, C++ and others. We'll explore a few of these and play with them to do some interesting things in our QML code. You can read about some of the available QML value types at the official docs page.

Start by creating a brand new Qt Quick project from Qt Creator. Give it a name and save it somewhere on your drive. We can define a few properties in our Window element like below

We won't always be able to show the full QML file and will only show code snippets relevant to the current discussion. The full final source code for the section can be found in the git repository for the book.

Content here is well consumed with a Qt Creator instance running, typing code, trying the concepts, and changing the code to make it your own. If you don't have Qt Creator around, at least be sure to make the time to grab the source code for this section, run it and try to modify a few things. While the author tries his best to explain the concepts, he is fully aware that you can't learn any technology by just reading a book. You'll have to get your hands dirty one way or the other.

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Datatypes Demo")

    //string : storing character sequences
    property string mString: "https://www.learnqt.guide"

    //int : storing numbers
    property int mInt: 45

    //bool : storing true/false values
    property bool isFemale: true

    //double : storing floating point numbers
    property double mDouble: 77.5

    //url : storing ressource location
    property url mUrl: "https://www.learnqt.guide"
}
```

The purpose for each of these is described in the comments accompanying the code. Let's play with our string a bit. We'll add a Text element and center it in the Window, and the text value is going to come from our mString property.

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Datatypes Demo")

    property string mString: "https://www.learnqt.guide"
    property int mInt: 45
    property bool isFemale: true
    property double mDouble: 77.5
    property url mUrl: "https://www.learnqt.guide"

    Text {
        id: mTextId
        anchors.centerIn: parent
        text: mString // Using our mString property
    }
}
```

We can even console.log() the value in our mString property when our Window finishes loading

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Datatypes Demo")

    property string mString: "https://www.learnqt.guide"
    property int mInt: 45
    property bool isFemale: true
    property double mDouble: 77.5
    property url mUrl: "https://www.learnqt.guide"

    Text {
        id: mTextId
        anchors.centerIn: parent
        text: mString // Using our mString property
    }

    Component.onCompleted: {
        //console.log("The value of mString is :" + mString)
        print("The value of mString is :"+mString)
    }
}
```

Looking back at our properties, you'll see that mString and mUrl contain the same value. This may mislead some beginners into thinking that string and url are the same and interchangeable. string, just like in many programming languages out there, is designed to represent a sequence of characters. url, however is designed to store sequences of characters but those characters should represent the location of some resource either on the local file system or on the web. It's a good idea to pause right now and go read the docs pages for both string and https://doc.qt.io/qt-6/qml-url.htmlurl to harden understanding of these and see some of the things you can do with them. We can prove that they are indeed different things with the following code snippet

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Datatypes Demo")

    property string mString: "https://www.learnqt.guide"
    property int mInt: 45
    property bool isFemale: true
    property double mDouble: 77.5
    property url mUrl: "https://www.learnqt.guide"

    Text {
        id: mTextId
        anchors.centerIn: parent
        text: mString // Using our mString property
    }

    Component.onCompleted: {

        if ( mString === mUrl)
        {
            console.log("They are the same")
        }else{
            console.log("They are NOT the same")
        }
    }
}
```

If you run your application, you'll see the message "They are NOT the same" printed out even if `mString` and `mUrl` seem to contain the same data. This is because the `===` operator in Javascript checks both for type and value. In other words, the only way to get the message "They are the same" printed out, is if `mString` and `mUrl` are of the same type and contain the same data. If you want to just check for the values, you can use the `==` operator, but we don't recommend that in your code as it may introduce errors that are hard to debug. You can also use the `bool` property to make decisions in your code :

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Datatypes Demo")

    property string mString: "https://www.learnqt.guide"
    property int mInt: 45
    property bool isFemale: true
    property double mDouble: 77.5
    property url mUrl: "https://www.learnqt.guide"

    Text {
        id: mTextId
        anchors.centerIn: parent
        text: mString // Using our mString property
    }

    Component.onCompleted: {
        if ( isFemale){
            console.log("You may wear a dress")
        }else{
            console.log("You may wear a suit")
        }
    }
}
```

Running this, you'll see

```
You may wear a dress
```

printed out in the Application Output Pane. Try to change the value of `isFemale` to `true` and see what happens. We can even use our bool property to decide whether the text in `mTextId` is bold.

```
Text {
    id: mTextId
    anchors.centerIn: parent
    text: mString
    /*
        If isFemale is true, the text will be bold
        otherwise it won't be.
    */
    font.bold: isFemale?true:false
}
```

This is using the ternary operator that is also found in many other programming languages out there. Run the application, and look at the font of the text. Change the value of isFemale and see how the font of the text changes accordingly. We can also do some things with the int and double properties. If we wrap our Text element in a Rectangle :

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Datatypes Demo")

    property string mString: "https://www.learnqt.guide"
    property int mInt: 45
    property bool isFemale: true
    property double mDouble: 77.5
    property url mUrl: "https://www.learnqt.guide"


    Rectangle {
        width: 200
        height: 100 + mInt //Use the int property
        anchors.centerIn: parent
        color: "yellow"

        Text {
            id: mTextId
            anchors.centerIn: parent
            text: mString
            font.bold: isFemale?true:false
        }
    }

    Component.onCompleted: {
        //Print out the values in our int and double properties
        console.log("The value of mInt is :"+mInt)
        console.log("The value of mDouble is :" + mDouble)
    }

}
```

We can make the height of the rectangle depend on our `int` property, and print the values in `console.log()` messages for the world to see.

var and other types

We have just played with a few QML data types, but you can always visit the docs page for QML Basic Types to see an exhaustive list of available properties. From that list, if we click on var, we'll see that *The var type is a generic property type that can refer to any data type*. On top of that, we also get a list of different properties you can define with var, reproduced below for convenience :

```
Item {
    property var aNumber: 100
    property var aBool: false
    property var aString: "Hello world!"
    property var anotherString: String("#FF008800")
    property var aColor: Qt.rgba(0.2, 0.3, 0.4, 0.5)
    property var aRect: Qt.rect(10, 10, 10, 10)
    property var aPoint: Qt.point(10, 10)
    property var aSize: Qt.size(10, 10)
    property var aVector3d: Qt.vector3d(100, 100, 100)
    property var anArray: [1, 2, 3, "four", "five", (function() { return "six"; })]
    property var anObject: { "foo": 10, "bar": 20 }
    property var aFunction: (function() { return "one"; })
}
```

var is discouraged for use, as it doesn't make it obvious what the type of the variable is, but you'll still see it in Javascript code out there. The code gives us several types defined and ready for use in our QML applications though. We'll try to use as many as we can. Copy the code inside the Item element and paste that into your Window element, just below the other properties we had declared, as shown below :

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Datatypes Demo")

    property string mString: "https://www.learnqt.guide"
    property int mInt: 45
    property bool isFemale: true
    property double mDouble: 77.5
    property url mUrl: "https://www.learnqt.guide"

    property var aNumber: 100
    property var aBool: false
    property var aString: "Hello world!"
    property var anotherString: String("#FF008800")
    property var aColor: Qt.rgba(0.2, 0.3, 0.4, 0.5)
    property var aRect: Qt.rect(10, 10, 10, 10)
    property var aPoint: Qt.point(10, 10)
    property var aSize: Qt.size(10, 10)
    property var aVector3d: Qt.vector3d(100, 100, 100)
    property var anArray: [1, 2, 3, "four", "five", (function() { return "six"; })]
    property var anObject: { "foo": 10, "bar": 20 }
    property var aFunction: (function() { return "one"; })


    Rectangle {
        width: 200
        height: 100 + mInt //Use the int property
        anchors.centerIn: parent
        color: "yellow"

        Text {
            id: mTextId
            anchors.centerIn: parent
            text: mString
            font.bold: isFemale?true:false
        }
    }

    Component.onCompleted: {
        //Print out the values in our int and double properties
        console.log("The value of mInt is :"+mInt)
        console.log("The value of mDouble is :" + mDouble)
    }

}
```

You can see the Color property declared as `aColor`. Want to learn more about this property? Why not type "QML Color Qt 6" in a search engine bar and start from there? This is just a way to define colors in your QML code. We have the color defined, and we can use it in our Rectangle, for example, to change the background color.

```
    Rectangle {
        width: 200
        height: 100 + mInt
        anchors.centerIn: parent
        color: aColor // Using our color property

        Text {
            id: mTextId
            anchors.centerIn: parent
            text: mString
            //font.bold: isFemale?true:false
            font : aFont
        }
    }
```

Run the application with this change applied and you'll see the color in your rectangle.

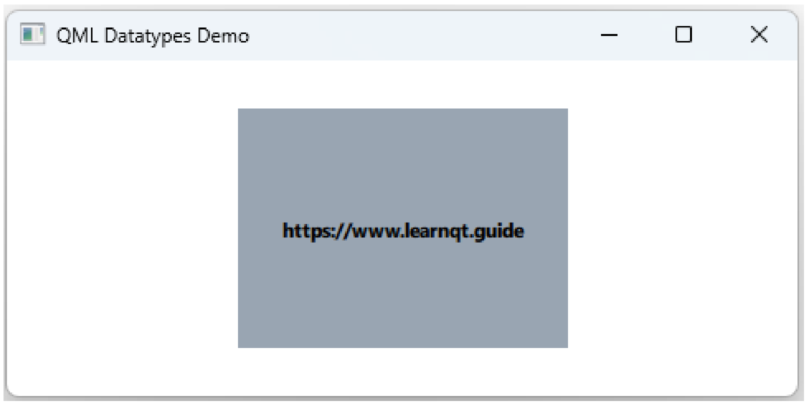


Figure 41. Use the Color Property

We can also play with the remainder of the properties by printing out their values in our Component.onCompleted handler.

```
    Component.onCompleted: {
        //Print out int,bool and string
        console.log("The value of aNumber is :"+ aNumber)
        console.log("The value of aBool is : " + aBool)
        console.log("The value of aString is : " + aString)
        console.log("The value of anotherString is : " + anotherString)

        //Print out the components of the rectangle : x, y, width and height.
        console.log("The components of aRect are : x : "+ aRect.x
                      + " y :"+ aRect.y + " width :" + aRect.width + " height :"+ aRect
.height)

        //Print out the length of the array
        console.log("The length of the array is :"+anArray.length)

        /*
        //One way to loop through the array printing data. We have a function
        //stored at index 5, so we're careful to call the function with a pair of
parentheses.
        anArray.forEach(function(value,index){
            if( index ===5){
              console.log(value())
            }else
            {
                console.log(value)
            }})
        */

        //Just another way to loop through the array
        for(var i = 0; i < anArray.length ; i++)
        {
            if ( i === 5)
            {
                console.log(anArray[i]())
            }else{
                console.log(anArray[i])
            }
        }

        //We make sure to call the function with a pair of parentheses
        console.log("The function value is: "+aFunction())

    }
```

Running your application with these changes applied, you'll see the relevant output in the application output pane, mine is reproduced below for convenience

```
qml: The value of aNumber is :100
qml: The value of aBool is : false
qml: The value of aString is : Hello world!
qml: The value of anotherString is : #FF008800
qml: The components of aRect are : x : 17 y :56 width :46 height :10
qml: The length of the array is :6
qml: 1
qml: 2
qml: 3
qml: four
qml: five
qml: six
qml: The function value is :Seven
```

Just for the fun of it, you can also define two more properties below aFunction,

```
property var aFont  : Qt.font({family: "Consolas", pointSize: 30, bold: false})
property date mDate: "2018-07-19"
```

You can use aFont to control text font in your application and use mDate where dates are expected.Be sure to use the docs if you need more information on any of these types.

The documentation for types already in Javascript like Date are best documented in the official Javascript documentation. Qt/QML documentation leaves more to be desired for these. Be sure to use your favorite Javascript reference when need arizes. MDN is a good one.

With these properties in place, you can change the font of our Text element

```
Text {
    id: mTextId
    anchors.centerIn: parent
    text: mString
    font : aFont
}
```

and print date infomation in our Component.onCompleted handler

```
Component.onCompleted: {
    console.log("The date is :"+ mDate)
}
```

This should get your feet wet using some of the available QML Basic types. Don't hesitate to check the documentation to learn more or even for helpful code snippets to get you started. I still use snippets from the Qt documentation as a starting point in my own Qt projects, even after using Qt for a decade in a professional setting.

> Remember to check the git repository for the book. It may help to see how any of the code snippets covered in the book fall under the big picture of the full runnable project for this section.

Property Bindings

One of the most powerful features of QML is property bindings. A property binding is a way to connect two properties in such a way that when one property changes, the other property is automatically updated to reflect the new value.

In QML, property bindings are created using the syntax of the signal and slot system. A signal is a message that is emitted when a property changes, and a slot is a function that is called when a signal is emitted. By connecting a signal to a slot, we can create a property binding that automatically updates one property when another property changes. For example, we can make the `height` of a rectangle depend on its `width`, and every time the `width` changes, the `height` will be updated automatically. The mechanism to push out change notifications is taken care of by the QML engine.

> The content in this section is best learned by having an instance of Qt Creator or your favorite IDE around to play with the concepts, break things, fix them back and make the code your own. The final source code is available in the book git repository.

Bindings in Action

Create a brand new Qt Quick project from Qt Creator, name it and save it somewhere on your drive. Change the code in the starter qml file from the Qt Quick Qt Creator project template to the code below:

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Property Binding Demo")

    Rectangle {
        id : redRectId
        width: 50
        height: width * 1.5 //Property binding in action
        color: "red"
    }
}
```

If you run the application, you should see a red rectangle in the top left corner of the Window. The important thing is happening on the line `height: width * 1.5`. **This line is a contract between the `width` and the `height` of the rectangle**, saying that this expression should hold true throughout the lifetime of the application. So if later on the width of the rectangle changes to 100, the QML engine has the responsibility to re-evaluate the height of the rectangle and change that to `100*1.5`. To prove that this is the case, add `blueRectId` under `redRectId` and make it show up in the bottom left corner of the Window :

```
//...
Rectangle {
    id : redRectId
    //...
}

Rectangle {
    id : blueRectId
    color: "blue"
    width: 100
    height: 100
    anchors.bottom: parent.bottom

    MouseArea {
        anchors.fill: parent
        onClicked: {
            redRectId.width = redRectId.width +10
        }
    }
}
//...
```

With these changes in your main.qml file, run the application and you should see the red rectangle in the top-left corner and the blue rectangle in the bottom-left corner of the window.

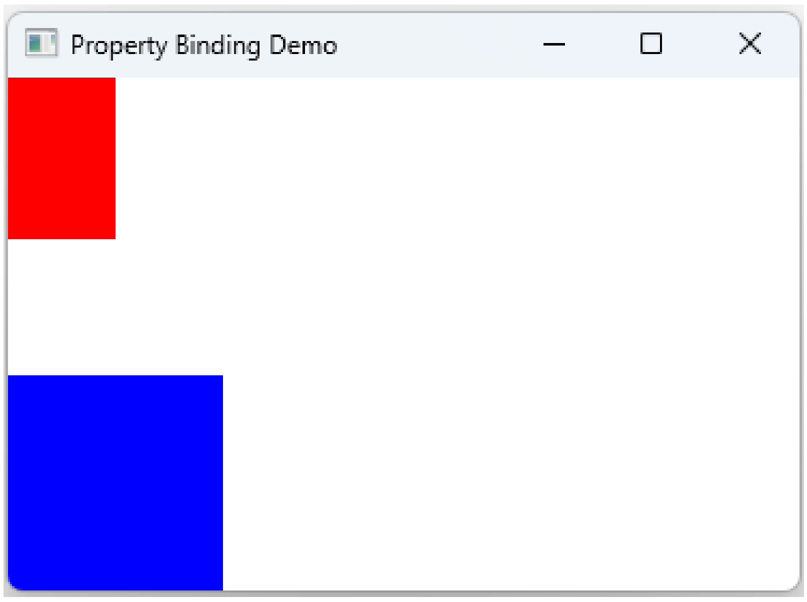


Figure 42. Blue Rectangle Changing the width of Red Rectangle

The click handler in blueRectId's MouseArea is just changing the width of `redRectId`. Because the height of `redRectId` is bound to its width however, it will follow and change every time we click on the blue rectangle. Try to click on the blue rectangle and you'll see that not only its `width` will change, but its `height` will also follow and change according to our binding expression in `redRectId` : `height: width * 1.5`.

Under the hood, the QML engine will be doing the heavy lifting for us. Whenever the width of the rectangle changes, there will be a signal emitted about that somewhere, and there will be a handler for that signal, whose job is to re-evaluate the height based on our binding expression and finally update the height of `redRectId` with the new value.

Breaking Bindings

Bindings are very powerful and they are used all over the place in QML. But if you're not careful, you can break them and introduce errors in your application. **You break a binding by assigning a static value to a property that was previously bound to another property**. Let's add a third green rectangle and assign a static value to redRectId.height when its MouseArea is clicked :

```
Window {
    //...
    Rectangle {
        id : redRectId
        width: 50
        height: width * 1.5
        color: "red"
    }

    //blueRectId is omitted here for brevity

    Rectangle {
        id : greenRectId
        color: "green"
        width: 100
        height: 100
        anchors.bottom: parent.bottom
        anchors.left: blueRectId.right
        MouseArea {
            anchors.fill: parent
            onClicked: {
            redRectId.height = 100 //Change the height to a static value. Breaks the
binding!
            }
        }
    }
}
```

The green rectangle is anchored to the bottom of the window but pushed to be to the right of the blue rectangle through anchors. Don't worry if this doesn't make sense yet. We'll get a chance to explore anchors in much more depth later on in the book. For now, they are just a way to position things nicely.

Our `greenRectId` MouseArea's click handler assigns a static value to `redRectId.height`, effectively replacing the previous expression that bound width and height with a boring static value of `100` for the height of the rectangle. When the QML engine sees you do that, it clears off all the infrastructure it had in place to propagate changes in width to the height of `redRectId`. Another way to look at this is that the height has its static value and doesn't need to listen in on changes from any other property in the application.

If you run the application and click on the blue rectangle, you'll see `redRectId` changing its width and height at the same time. Click on green and the height of the red rectangle will change. Now the fun begins: If you click back on blue, you'll see that only the width is going to change, and the height will remain at its static value of 100. **We have effectively broken the binding when we clicked on the green rectangle**!

Now we know this : **Assigning a static value to a property that was previously bound to another property will break the binding**. But what if we don't assign a simple static value, but assign an expression? Something like below

```
Rectangle {
    id : greenRectId
    //...
    MouseArea {
        anchors.fill: parent
        onClicked: {
            redRectId.height = redRectId.width * 1.5 // Replacing the binding with an
expression
        }
    }
}
```

The author invites you to try this in Qt Creator, click around on rectangles and see if the binding is broken. Do it! But this won't hold the binding in place. What the QML engine does here, is to evaluate the expression `redRectId.width * 1.5` and plug the result into redRectId.height as a **static** value, effectively breaking the binding just like we did before.

Now we also know that **you can't simply replace a previous binding with a new expression just like that**. But what if what we need is to change the binding to a new expression? What if, for some reason, we want the height to be twice the height? `Qt.binding` comes to the rescue here. `Qt.binding` allows us to specify a Javascript function that returns the new expression you want to be used for the binding. Let's look at a code example :

```
//...
Rectangle {
    id : greenRectId
    color: "green"
    width: 100
    height: 100
    anchors.bottom: parent.bottom
    anchors.left: blueRectId.right
    MouseArea {
        anchors.fill: parent
        onClicked: {
            redRectId.height = Qt.binding(function(){
            return redRectId.width * 2
            })

        }
    }
}
//...
```

Change your code accordingly and run the app. Click on the blue rectangle to see both width and height of the red rectangle changing. If you click on the green rectangle, however, this time we won't break the binding, but will merely replace it with a new binding expression that now makes sure the height will be twice the width. `Qt.binding()` effectively informs the QML engine that we want the binding expression changed to what is returned by the binding Javascript function.

The Qt Global Object

The Qt Global Object, also known as the Qt QML Type, is a singleton object that can be accessed from any QML file. It provides a number of useful properties and methods that can be used in QML. For example, it provides access to the current time, the current date, and the current locale. It also provides a way to display message boxes, access the clipboard, and perform other system-level operations.

One of the key benefits of using the Qt Global Object in QML is that it allows developers to easily share data and functionality between QML files. For example, a QML file that contains a custom control or widget can expose its properties and methods through the Qt Global Object, making them accessible from other QML files.

In addition to providing access to system-level functionality, the Qt Global Object can also be used to define global variables and functions that can be used throughout an application. This can help to simplify code and make it easier to maintain and modify over time. In this section, we'll try out some of the Qt Global Object features.

Facilities from the Qt Global Object

Create a brand new Qt Quick project from Qt Creator, name it and save it on your drive. Add a rectangle inside the window as shown below. The red rectangle contains a MouseArea with a click handler that is of interest to us.

```
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Qt Global Object Demo")
    Rectangle{
        width: 300
        height: 100
        color: "red"
        anchors.centerIn: parent
        MouseArea{
            anchors.fill: parent
            onClicked: {
                //Quit the application
                console.log("Quitting the App")
                Qt.quit()
            }
        }
    }
}
```

The click handler starts by printing out a debug output message through console.log() and then kills the application. Try running the application at this point; if you click on the rectangle, you'll see that the application is going to quit. And this is one of the things you can use the Qt Global Object for. You invoke things from the Qt Global Object by typing `Qt.` and specifying the facility you want to use. In this case, we wanted to call the `quit()` method. If you remember from the last lecture, we also used the` Qt.binding()` thing, which also came from the Qt Global object. Now comes the million-dollar question: "How do I get to know about available features in the Qt Global Object?". Well, the documentation my friend!

Now is a good chance to hone your research skills. Use your favorite search engine, and find out at least 3 properties and 15 methods you can call from the Qt global object. It may not be obvious, or the Qt docs may be using different wording for things. It helps a lot to try these things on your own, get your hands dirty and develop your preferences. Many people will read this, won't bother honing this skill and will mostly be dependent on others to find ideas or even fix problems in their projects. Don't be one of them! Learn to use the docs effectively. It really is a superpower in the author's opinion.

If somehow ignored the tip above, here is the documenation page for the Qt Global Object, sometimes also referred to as the Qt QML Type. Look at the properties, methods and there's a lot you can do with this type. If you look closely, you'll find that we have a fontFamilies method. We can use it to get _ a list of the font families available to the application_ as the documentation says. Let's try this in our application. The function returns a list of fonts, so we can loop through the list printing out each font :

```
import QtQuick
Window {
    //...
    //Get the list of fonts and store that in a property called fonts.
    property var fonts: Qt.fontFamilies()

    Rectangle{
        width: 300
        height: 100
        color: "red"
        anchors.centerIn: parent
        MouseArea{
            anchors.fill: parent
            onClicked: {
                //List the available fonts
                for( var i = 0; i < fonts.length ; i++){
                    console.log("["+ i + "] :" + fonts[i])
                }
            }
        }
    }
}
```

Notice that just below the title property of the window, we call our fontFamilies method and store its return value into a property, or variable called fonts. This is something you'll do a lot in your QML files; declaring properties at the root level or your top-level QML element, making those properties available to any element directly or indirectly nested in your top-level element. The click handler in our MouseArea simply sets up a simple loop, printing each font found by Qt.fontFamilies(). Below is a section of what I get when I run this application on my local Windows system and click on the red rectangle :

```
qml: [0] :Agency FB
qml: [1] :Algerian
qml: [2] :Arial
qml: [3] :Arial Black
qml: [4] :Arial Narrow
```

```
qml: [5] :Arial Rounded MT Bold
qml: [6] :Avignon Pro
qml: [7] :Avignon Pro Demi
qml: [8] :Avignon Pro Medium
qml: [9] :Avignon Pro Xlight
qml: [10] :Bahnschrift
qml: [11] :Bahnschrift Condensed
/...
```

If later on, you need to fetch font info from your system, this is something that may come in handy. We can also use the Qt.md5() method to hash strings :

```
MouseArea{
    anchors.fill: parent
    onClicked: {
        //Hash a string
        var mName = "Daniel Gakwaya"
        var mNameHash = Qt.md5(mName)
        console.log("The hash of the name is :"+ mNameHash)
    }
}
```

Running this on my system, I get the output below

```
qml: The hash of the name is :74bc9d4e0d66885a4de70a30f3fd1582
```

Let's try a few more interesting things you can do just to spice up your appetite. Looking at the docs, you can find the openUrlExternally method. That's exactly what it does, it uses your default browser on the system to open the url you pass in the method as a parameter. Below is a simple example that'll open our website page if you click on the red rectangle :

```
import QtQuick
MouseArea{
    anchors.fill: parent
    onClicked: {
        //Open url externally
        Qt.openUrlExternally("https://www.learnqt.guide/udemy-discounted-9/")
    }
}
```

You can also use Qt.openUrlExternally to open a file on the local system. The default program will be used. Here is a simple example to open an image from my local drive

```
//Open local files with the default program
Qt.openUrlExternally("file:///D:/artwork/LearnQt.png")
```

Notice that we provided the full path. There may be ways to use relative paths if that's what you need, but we'll leave that to the reader and the documentation. The last thing we'll show you here is that you can use Qt.platform.os to ask the application which operating system it's running on.

```
//Capture platform info
console.log("The current platform is : "+Qt.platform.os)
```

Run the application with this change the click handler of your MouseArea and you'll see your OS printed out on the console. Here is what we get on our Windows system :

```
qml: The current platform is : windows
```

The Qt QML Type, as some like to call it, will provide lots of handy features to make your life easier in your Qt QML development career. You don't have to memorize all these methods and properties, you just need to be aware of them, and they'll probably pop up in your mind when you need them to solve a problem at hand.

Property Change Handlers

We have had a chance to define custom properties in QML and we used the syntax below

```
property string firstName: "Murphy"
```

Once a property is defined, you can use it anywhere in your QML file, provided you use it beyond its point of definition. What we haven't seen, however, is the fact that when you define a property, QML also automatically generates handlers that are triggered whenever that property changes. For example, for our `firstname` property, QML will generate a handler named onFirstNameChanged, and you can use it like below

```
onFirstNameChanged: {
    console.log("The firtsname changed to :"+ firstName)
}
```

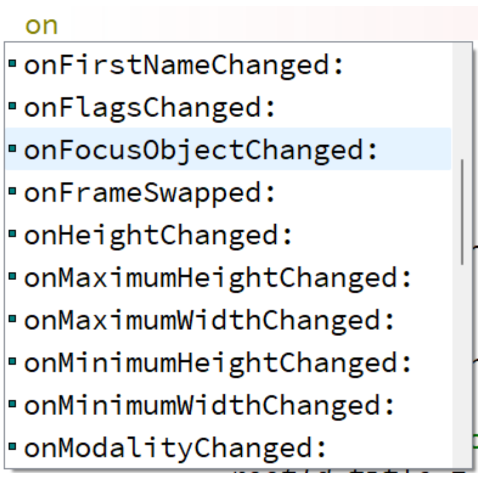
To play with this, please create a brand new Qt Quick project in Qt Creator, modify the generated main.qml file like below

```
import QtQuick
Window {
    id : rootId
    visible: true
    width: 640;height: 480
    title: qsTr("Properties and Handlers Demo")

    //Define a custom property
    property string firstName: "Murphy"

    //Use the handler for the custom property
    onFirstNameChanged: {
        console.log("The firstname changed to :"+ firstName)
    }
    Rectangle {
        width : 300;height: 100
        color: "greenyellow"
        anchors.centerIn: parent

        MouseArea{
            anchors.fill: parent
            onClicked: {
                firstName = "John"
            }
        }
    }
    Component.onCompleted: {
        console.log("The firstname is :"+firstName)
    }
}
```

The example sets up both the property and the property handler method. We change the firstname when the click handler in the MouseArea is triggered. If you run the application and click on the rectangle, you'll see the output message

```
The firstname changed to John
```

coming from the onFirstNameChanged handler, which is activated when the value in firstname changes. This is a behavior built into QML, and it still works even for built-in types. For example, in our Window, you can try to start typing the characters "on" and hit `Ctrl` + `Space` inside your Qt Creator editor, you'll see an intelliscence window pop up with suggestions on handlers you can use



```
on
onFirstNameChanged:
onFlagsChanged:
onFocusObjectChanged:
onFrameSwapped:
onHeightChanged:
onMaximumHeightChanged:
onMaximumWidthChanged:
onMinimumHeightChanged:
onMinimumWidthChanged:
onModalityChanged:
```

Figure 43. Property Change Handlers Inteliscence suggestions

in this list, you can see onHeightChanged(), which is triggered whenever the height of the window changes. This should act as proof that these property change handlers even work for built-in types. The code example below shows us using the onTitleChanged() handler. We are changing the title when the rectangle is clicked :

```
import QtQuick
Window {
    id : rootId
    visible: true
    width: 640
    height: 480
    title: qsTr("Properties and Handlers Demo")

    property string firstName: "Daniel"
    onFirstNameChanged: {
        console.log("The firtsname changed to :"+ firstName)
    }

    onTitleChanged: {
        console.log("The new title is : " + rootId.title)
    }
    Rectangle {
        width : 300
        height: 100
        color: "greenyellow"
        anchors.centerIn: parent

        MouseArea{
            anchors.fill: parent
            onClicked: {
                firstName = "John"
                rootId.title = "The sky is blue"
            }
        }
    }
}
```

Run the example and you should see the title of the window change when you click on the rectangle

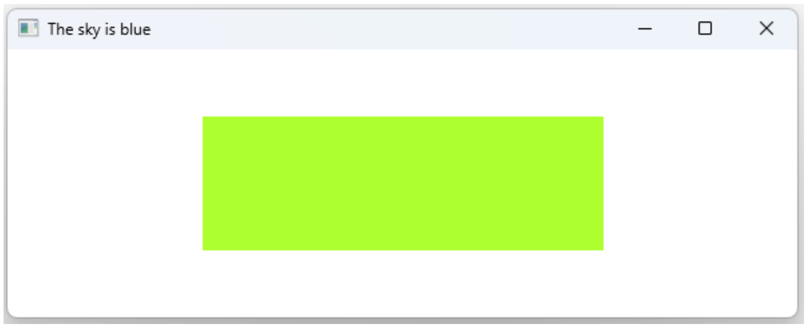


Figure 44. Window Title Property Change Handler

You should also see output below in the Application Output Pane in Qt Creator

```
qml: The firsname changed to :John
qml: The new title is : The sky is blue
```

Don’t stop here. Go ahead and play with as many property change handlers as you can. Over time your brain will develop an intuition for which handlers to use to solve a problem at hand.

Chapter Summary

Congratulations on making it this far in the book! In the chapter, we explored QML syntax in a little more detail. We started out by running a Qt Quick application made up of an empty `Window` element. We saw that the Window element, just like most QML elements, has a few properties like `width`, `height` and others. You use these properties to control some aspects of the QML element of interest. We attempted to lay out a few rectangles inside the Window element and from that, we saw that QML syntax is a hierarchy of elements nested into each other. QML elements can emit signals, and we get to respond to these signals through handlers. We took a chance to also see that the id property is present in any QML element, and we can use it to manipulate a QML element from the insides of another QML element. We learned about some of the basic property types like `string`, `int`, `bool`, `double` and `url`. We also saw them in action in a Qt Quick project. We explored property bindings and saw that it’s a mechanism to let changes in one property automatically affect the values of other properties. The Qt global object is omnipresent in your QML files and you can use the facilities it puts at your disposal. We closed the chapter by looking at property change handlers. In the next chapter, we’ll explore some basic QML elements in detail.

Chapter 3: Basic QML Elements

In this chapter, we'll focus on four basic QML elements: Rectangle, Item, Image and Text. We'll get to see that many elements in QML inherit properties from the Item element. We'll look at different ways to source images for your Image elements and have our first look at making use of the Qt resource system from your Qt Creator project. We'll wrap up the chapter by composing these basic elements to build our own custom elements. This will introduce us to the great, yet, sometimes confusing subject of exporting your QML properties for use from the outside.

Rectangle, Item and Text

Now that you have a good grasp on the QML basic syntax, we have enough information to start exploring some specific elements. We'll start with Rectangle, Item and Text. Before we even start talking about them, we invite you to check them out in the docs to have a basic idea of what you can do with them. `Rectangle` is what you use to create rectangles in your QML apps, it is a visual item and you can apply some visual properties like `color` to it. Item is not a visual element, but one common thing it's used for is to group other visual elements and manipulate them. You can't use properties like `color` on the Item element, but you can use properties like `x`, `y`, `with` and `height`.

Text is an element you use to display text in your QML user interfaces. Its most important property is text, but you can use other relevant properties like font and position your text with anchors. But we're not here to just talk about these elements. We need some action. Create a brand new Qt Quick project from Qt Creator, name it and save it somewhere on your drive. Change your main.qml file to contain code like below

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Basic Elements Demo")

    Item {
        id : containerItemId
        x : 50 ;y : 50
    }
}
```

If you run this application, you'll see a blank white screen. This is because `Item` is a nonvisual element by design. One technique one can use to see its location is to nest a Rectangle inside. But for that to work, we should also give a width and a height to our Item.

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Basic Elements Demo")

    Item {
        id : containerItemId
        x : 50 ;y : 50
        width: 600
        height: 300

        Rectangle{
            anchors.fill: parent
            color: "beige"
            border.color: "black"
        }
    }
}
```

Our Rectangle is nested in a way that it will fill the available space inside the Item. Unlike Item, Rectangle has a color property we can use to see where it's located in the Window. Since the rectangle is filling the Item, the location of the rectangle will be the location of the Item element. The effect is that we see a beige rectangle nested inside our Window element. Run the app with these changes applied to see this for yourself. We can also nest other smaller rectangles inside our Item element :

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Basic Elements Demo")

    Item {
        id : containerItemId
        x : 50 ;y : 50
        width: 600
        height: 300

        Rectangle{
            anchors.fill: parent
            color: "beige"
            border.color: "black"
        }

        Rectangle{
            x : 0
            y : 10
            width : 50
            height: 50
            color: "red"
        }

        Rectangle{
            x : 60
            y : 10
            width : 50
            height: 50
            color: "green"
        }
    }
}
```

Running the app gives us something like below

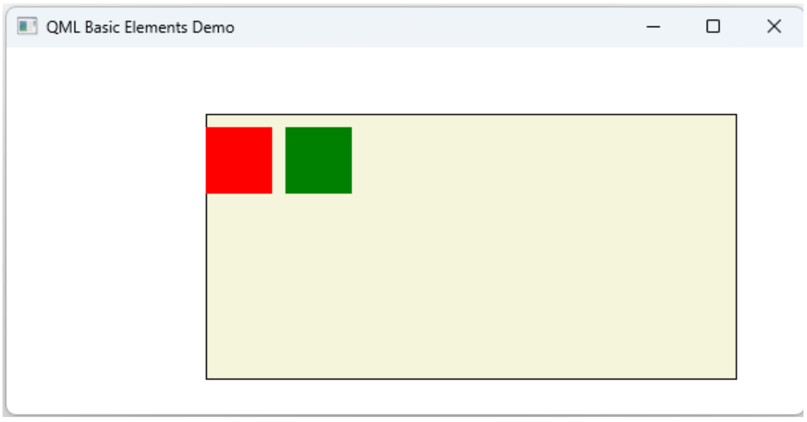


Figure 45. Rectangles contained in Item

One thing worthy of note here is that the red and green rectangles are positioned relative to the top left corner of `containerItemId`, **not relative to the Window element**. You should also notice that `x` values grow positive as you go towards the right and `y` values grow positive as you go down. As a specific example, the green rectangle is positioned 60 pixels from the top-left corner of `containerItemId` on `x`, and 10 pixels from the top-left corner of `containerItemId` on `y`.

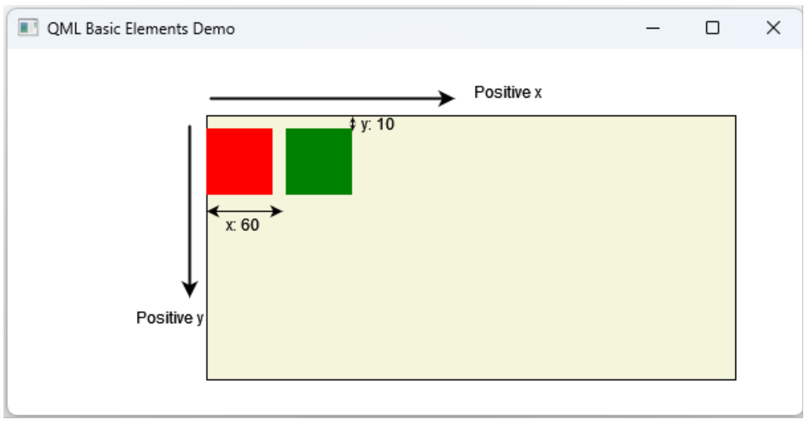


Figure 46. Relative Positions

Try to change the x and y values for containerItemId and see it moving together with its content. This is one of the applications for the Item element: Grouping elements and having a single place to manipulate them. Another thing you should keep in mind is that all visual items in QML inherit

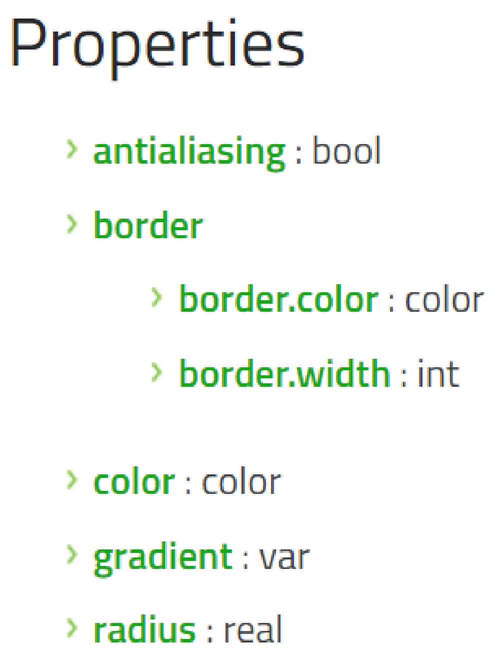
Item. In the docs page for Item, you can see a list of elements inheriting from Item in the **Inherited By** section. Just take a look at these and open the docs page for those that pick your attention, and see what they do. You'll pick up a lot of ideas just from occasionally browsing the docs page, even just by reading a few lines in the detailed description section.

Rectangle and Grouped Properties

You have seen several instances of the `Rectangle` element used in a couple of examples so far. What we haven't explored enough, however, is the fact that you can specify a border for your `Rectangle` and syntax variations that are available to achieve that. We'll take this chance and introduce you to grouped properties in QML. Take a good look at our background Rectangle :

```
Rectangle{
    anchors.fill: parent
    color: "beige"
    border.color: "black"
}
```

It has a border property we can use to control the look and feel of its border. It's not just a simple property though, **it's a grouped property**! Meaning that the property has sub-properties. In this case, the border has two sub-properties as specified by the docs.


Properties

- antialiasing : bool
- border
  - border.color : color
  - border.width : int
- color : color
- gradient : var
- radius : real

Figure 47. Rectangle border Grouped Property

The sub-properties are `width` and `color`. In other words, if we decide to set up a border for our Rectangle, we have the option to specify the color, the width, or both.

If you set up a Rectangle element without a color and without a border, the rectangle will just be sitting in memory but won't be visible, much like an Item element. Try to comment out the color and border properties in your background

Rectangle to see this with your own eyes.

We have already seen the first syntax to specify a border, and we can just add to it and also specify the border width, on top of the border color

```
Rectangle{
    anchors.fill: parent
    color: "beige"
    border.color: "black"
    border.width : 5

}
```

If you run the application with these changes applied, you'll see that our border has now put on some weight!

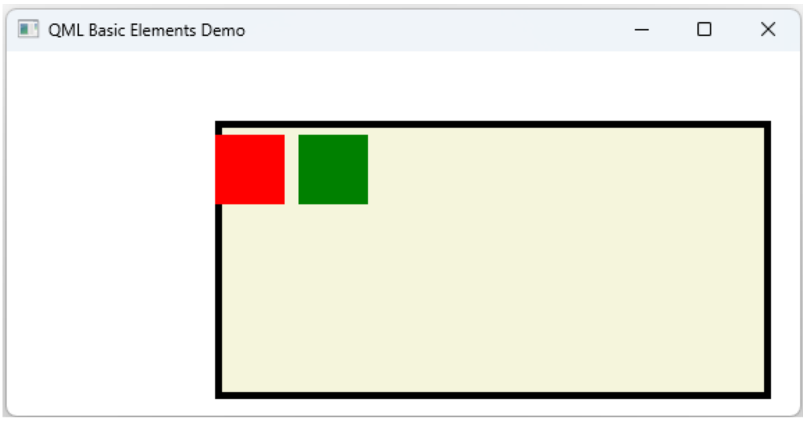


Figure 48. Rectangle border width and color

There are three syntax variations to specify grouped properties and what we have seen is just one of them. The second one is to wrap grouped properties in a pair of curly braces :

```
Rectangle{
    anchors.fill: parent
    color: "beige"

    /*
    //Grouped Properties Syntax Variation #1
    border.color: "black"
    border.width : 5
    */

    //Grouped Properties Syntax Variation #2
    border{
        color : "black"
        width : 5
    }
}
```

Comment out Variation #1 and activate Variation #2. If you run the application, you'll see that we have the same border width and color that we had previously. You can lay out the sub-properties on a single line and separate them with semi-colons.

```
Rectangle{
    anchors.fill: parent
    color: "beige"

    /*
    //Grouped Property Syntax Variation #1
    border.color: "black"
    border.width : 5
    */

    //Grouped Property Syntax Variation #2
    /*
    border{
        color : "black"
        width : 5
    }
    */

    //Grouped Property Syntax Variation #3
    border{
        color : "black";width : 5
    }
}
```

This is our third syntax variation to specify a grouped property. All these variations a laid out in a single code snipped to make it easier to compare them and see the differences. You should be familiar with all of them as you'll see them used all over the place in QML code out there.

The Text Element

The Text element is just for setting up text in your QML user interfaces. Let's just play with it in code. Change your code in the main.qml file like below :

```
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Basic Elements Demo")

    Item {
        id : containerItemId
        x : 150 ;y : 50
        width: 400;height: 200
        Rectangle{
            anchors.fill: parent
            color: "beige"
            border{
                color : "black";width : 5
            }
        }
        Rectangle{
            x : 0 ;y : 10
            width : 50
            height: 50
            color: "red"
        }
        Rectangle{
            x : 60 ;y : 10
            width : 50
            height: 50
            color: "green"
        }
        Text {
            x : 100 ;y : 100
            id : mTextId
            text: "Hello World!"
            font {
                family: "Helvetica"
                pointSize: 13
                bold: true
            }
            color: "red"
        }
    }
}
```

We have a `Text` element, with the id `mTextId`, positioned at 100 in `x` and 100 in `y`, relative to the top-left corner of `containerItemId`. `Text` has a font property, which you can recognize to be a grouped property by now. We have specified the font family, the size of the font and made it bold. Is this all there is to font property? I'll leave you to check the docs for more if you are interested. For now, we'll run the app and see our text saying "Hello World!" in red :

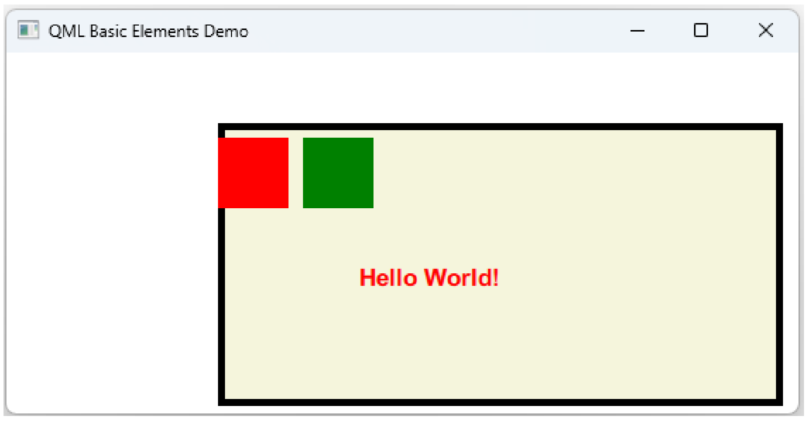


Figure 49. Text Element Hello World in Red

Image

The Image element is used to display images in your QML graphical user interfaces. This is one of those things that's easier to show in action than to talk about. So let's build something and explore the element along the way. Start a brand new Qt Quick project from Qt Creator. Name it and save it on your local drive. Change your main.qml file to contain code like below :

```
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("QML Basic Elements Demo")

    Item {
        id : containerItemId
        x : 150 ;y : 50
        width: 600
        height: 300

        Image {
            x : 10
            y : 50
            width: 100
            height: 100
            //Loading image from the working directory
            source: "file:LearnQt.png"
        }
    }
}
```

We have an `Image` element, nested inside an `Item`, which we'll use to manipulate several Image elements in a group. This is a trick you already know. An important property of the Image element is `source`. This is used to specify the location of the image that will be displayed. In this case, we want to load the image file from the local file system where you're building your QML project. How do I know that? It's because of the `file:` thing in front of the image file name. This is technically referred to as the file URI scheme. If you run the application as is at the moment, you'll see an empty screen and the image won't be anywhere to be seen. You'll instead see an error in your Qt Creator Application Output pane, saying something like

```
qrc:/2-Image/main.qml:39:9: QML Image: Cannot open: file:LearnQt.png
```

The question now becomes, where is the LearnQt.png file stored in the project? It's stored in a folder Qt Creator calls a working directory, and that's where the binaries for your QML files are generated. To find your working directory, (1) click on the project tab in Qt Creator, (2) on your active kit, click on the run configuration, and (3) on the right you'll see a text field labeled **Working Directory**.

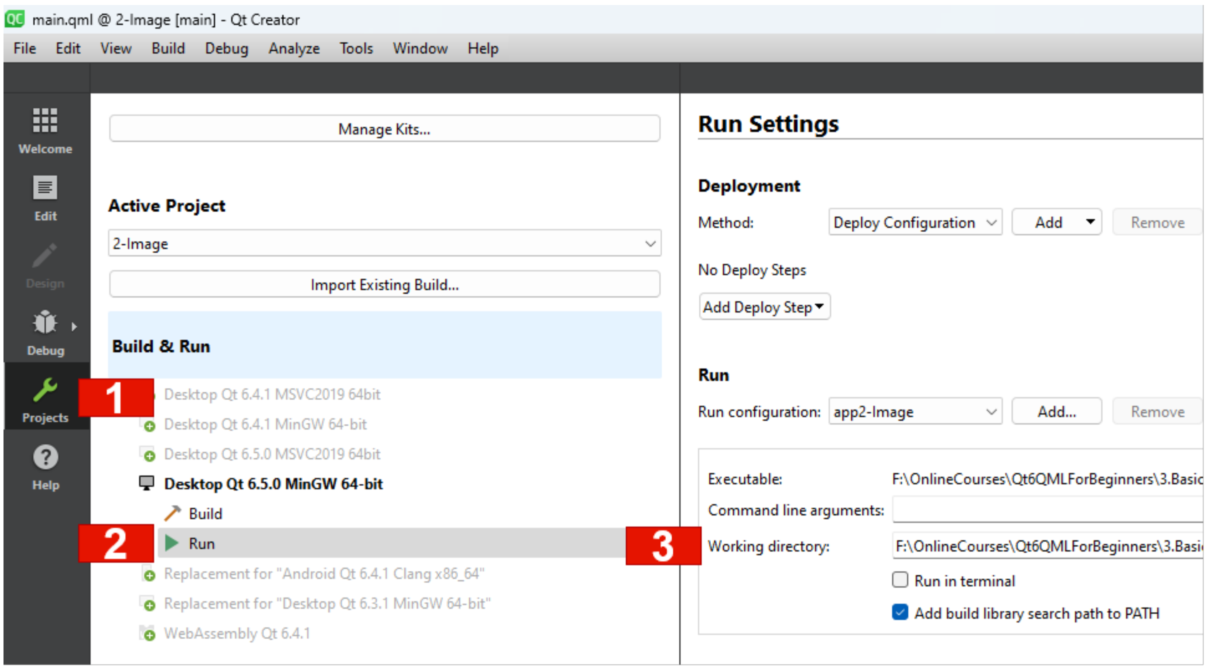

main.qml @ 2-Image [main] - Qt Creator
File Edit View Build Debug Analyze Tools Window Help
Welcome
Edit
Design
Debug
Projects
Help
Manage Kits...
Active Project
2-Image
Import Existing Build...
Build & Run
1
Desktop Qt 6.4.1 MSVC2019 64bit
Desktop Qt 6.4.1 MinGW 64-bit
Desktop Qt 6.5.0 MSVC2019 64bit
Desktop Qt 6.5.0 MinGW 64-bit
Build
2
Run
Replacement for "Android Qt 6.4.1 Clang x86_64"
Replacement for "Desktop Qt 6.3.1 MinGW 64-bit"
WebAssembly Qt 6.4.1
Run Settings
Deployment
Method: Deploy Configuration
Add
Remove
No Deploy Steps
Add Deploy Step
Run
Run configuration: app2-Image
Add...
Remove
3
Executable: F:\OnlineCourses\Qt6QMLForBeginners\3.Basic
Command line arguments:
Working directory: F:\OnlineCourses\Qt6QMLForBeginners\3.Basi
Run in terminal
Add build library search path to PATH

Figure 50. Qt Creator Project Working Directory

Open that location on your local file system, and paste in the LearnQt.png file you'll find in the book git repository for the book. Run the application and the image should show up on your user interface.

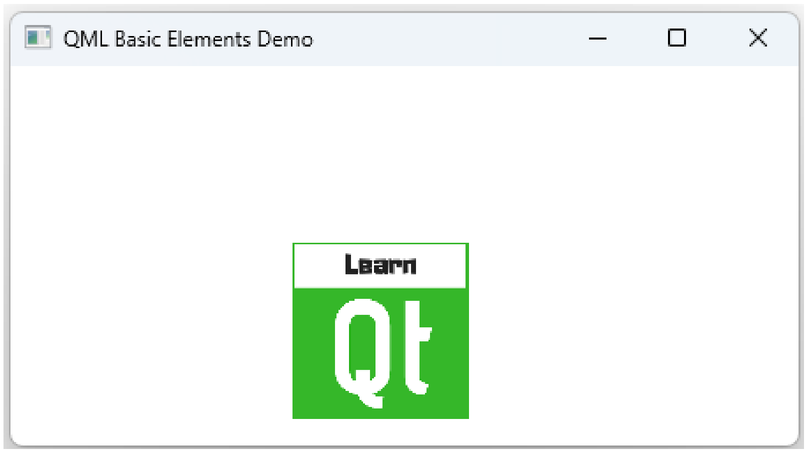


Figure 51. Image From Working Directory

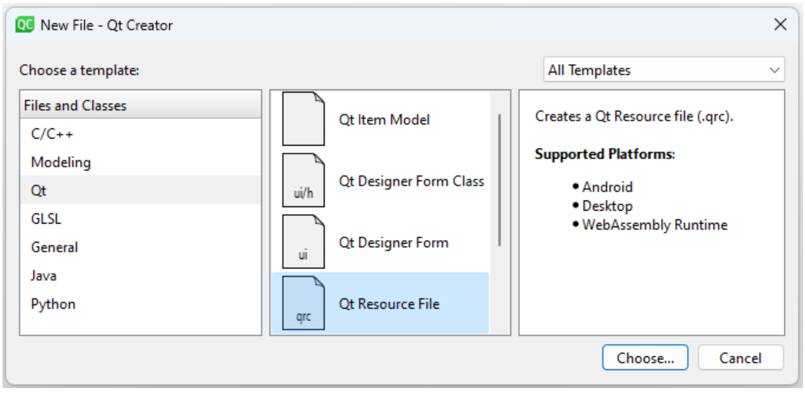
Loading Images From The Resource System

Getting the image from the working directory works as we have just seen. But it has a couple of drawbacks, the most important one being that the project can't easily move from one computer to another: If you move the project to another computer, the location of the project will probably change and the working directory will change with that, breaking things as the images won't be found if you run the application. Qt provides an easier way around that, and that's storing your images in a resource file. The Qt resource system works in such a way that files in your resource files are compiled into binary form, and packaged together with your application binary file and they will always be found by your QML application because they live inside it! The drawback with using the resource system is that the size of your executable binary file grows remarkably.

To work with the Qt resource system, the first thing one needs to do is store the images somewhere in your project directory. What we'll do here is create an images folder inside our project directory and store our LearnQt.png file inside. Below is the file structure in my project directory.

```
│    CMakeLists.txt
│    CMakeLists.txt.user
│    main.cpp
│    main.qml
│
└───────images
           LearnQt.png
```

You can see the usual files: `main.qml`, `main.cpp` and `CMakeLists.txt`. But now you should also have an images folder containing a file named LearnQt.png. At this point, you should have the image file in place. The next thing is to create the actual resource file. From Qt Creator issue the **File › New File** command. In the window that shows up, choose **Qt** template and select **Qt Resource File** in the window on the side.



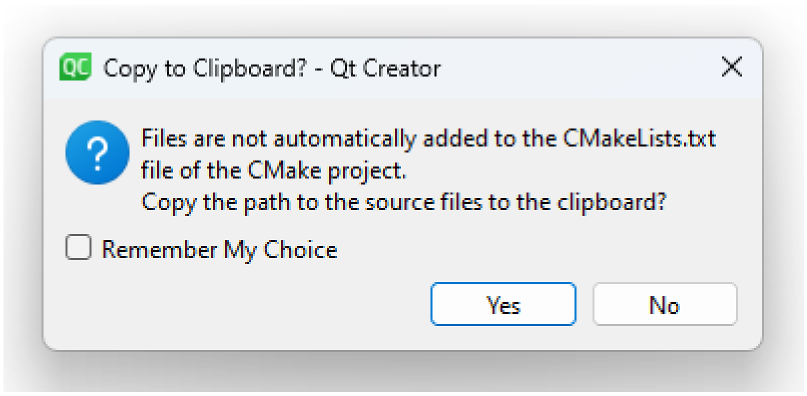
Creates a Qt Resource file (.qrc).

Supported Platforms:

- Android
- Desktop
- WebAssembly Runtime

Figure 52. Create Qt Resource File from Qt Creator

Click through the next set of windows that show up, specifying the name and location of your resource file, enabling or disabling Git for the file and your file will be generated in the project directory. Mine is named `resource.qrc`. When you're using the CMake build system like we are in the book, a window will pop up, prompting you to add the file to CMake.


Copy to Clipboard? - Qt Creator

Files are not automatically added to the CMakeLists.txt file of the CMake project.
Copy the path to the source files to the clipboard?

Remember My Choice

Figure 53. Add File to the CMakeLists.txt File

Adding a new file in the project directory like we just did, isn't enough for CMake to pick up the new file. You have to explicitly add the file to your CMakeLists.txt file. It's what the window here is trying to warn about. We can inform CMake about our new `resource.qrc` file by referencing it in the `qt_add_executable()` command in our CMakeLists.txt file. Modify your command like below

```
qt_add_executable(app2-Image
    main.cpp resource.qrc
)
```

and save your changes. CMake should do some processing and your resource file should show up in the Qt Creator file viewer. From all we just did here, you might have gathered that resource files in Qt Creator use the `.qrc` extension.

If the file structure gets confusing for some reason, please remember you can always reference the git repository for the book to see the final structure of the project for this section.

Here is my file structure at this point, for ease of reference

```
│    CMakeLists.txt
│    CMakeLists.txt.user
│    main.cpp
│    main.qml
│    resource.qrc
│
└───images
        LearnQt.png
```

We want to open the resource file and add our image file to it. This is one of the confusing parts of Qt Creator. My first intuition was to double-click on the resource file hoping it will open up ready for me to edit. But doing that just expands the file in the Qt Creator file viewer. To get the resource file opened up in the resource editor, which is what we want, you have to invoke the context menu by right-clicking on it and choosing **Open in Editor**.

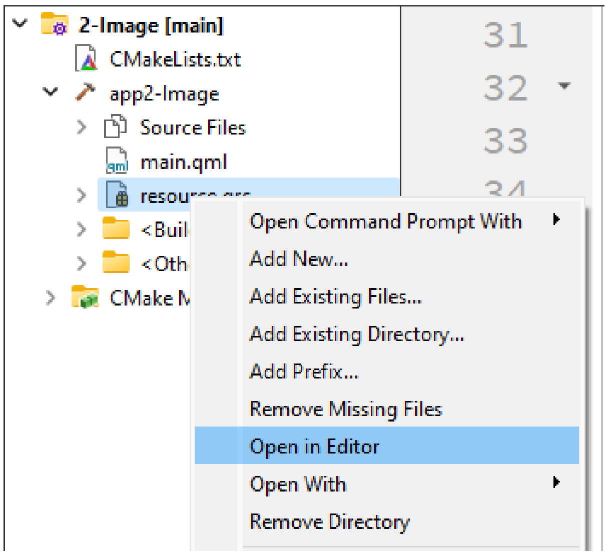


Figure 54. Open Resource File in Resource Editor

The file will open up as shown in the image below. Before we add files to the resource file, however, we need to first add a prefix. Prefixes are a way to logically organize your files in the Qt resource system. We won't go into details about that here, we'll just set up an empty prefix pointing to the root of our project, or the location where our `resource.qrc` file is hosted.

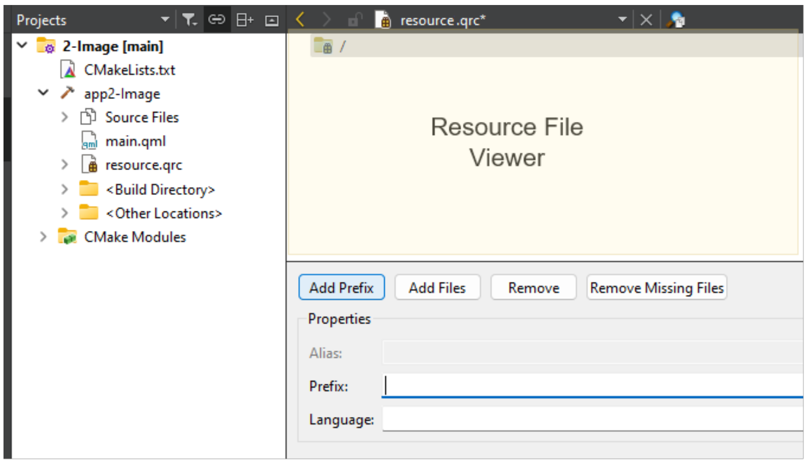


Figure 55. Adding Prefixes To Resource Files

Click on the **[Add Prefix]** button. Empty out the Prefix text field and hit **[Enter]**. This will create a root prefix, denoted by the `/` in the resource file viewer. With your `/` prefix selected, click on **[Add Files]**, go to your images folder and select the LearnQt.png file. Your image will be added and your opened `resource.qrc` file should look something like below:

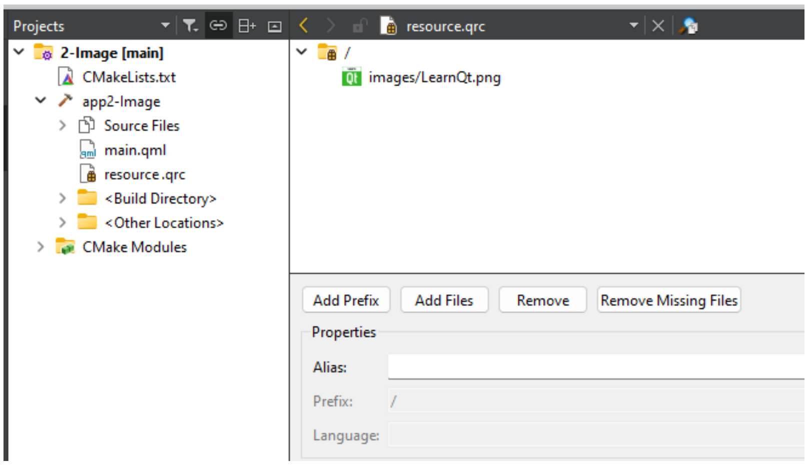


Figure 56. File Added to Resource File

We have our image properly added to the resource file. Let's take a step back and think about what we have at our disposal. When Qt sees your image referenced in a resource file like resource.qrc, it knows that when you issue the command to compile the application, it will also compile all files referenced in the resource file, including our LearnQt.png image, into binary form and package that together with the generated binary file, it doesn't matter where you move the binary, the image will be bundled into it. This makes your application highly portable. Time to use the file in the `Image` element inside our `main.qml` file. You could type in the URI of the file in your resource file, but there is an arguably better way, that personally helps me avoid typos.

In the Qt Creator file viewer, you can expand your `resource.qrc` file and it'll show all files inside. If you invoke the context menu by right-clicking on `LearnQt.png`, you'll see an entry that says Copy URL "qrc:/images/LearnQt.png", click on that and the path to the image will be copied into your clipboard.

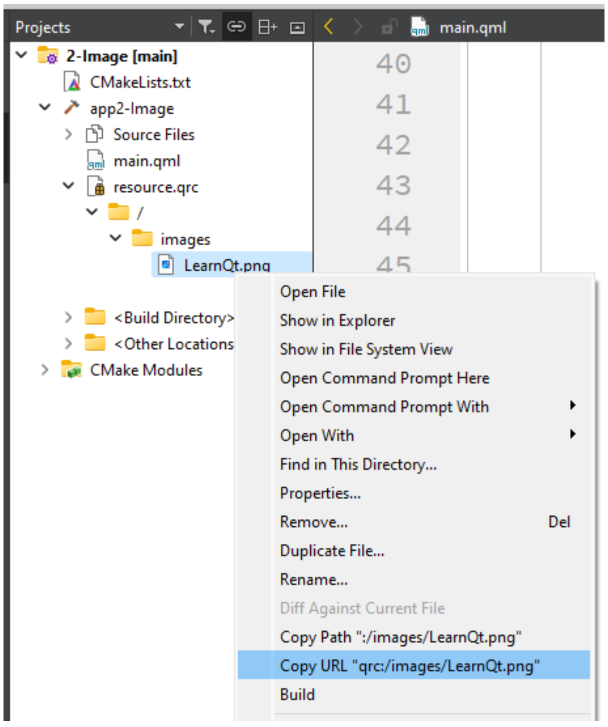


Figure 57. Copy URI Path to File

Paste that as a value to the source property of an Image element laid out just below the one we had previously:

```
//...
Image {
    x : 10;y : 50
    width: 100;height: 100
    //Loading image from the working directory
    source: "file:LearnQt.png"
}
Image {
    x : 150;y : 50
    width: 100;height: 100
    //Loading image from the resource file
    source: "qrc:/images/LearnQt.png"
}
//...
```

If you run the application with these changes applied, you'll see two images in your window, one coming from the working directory and the other coming from the resource file.

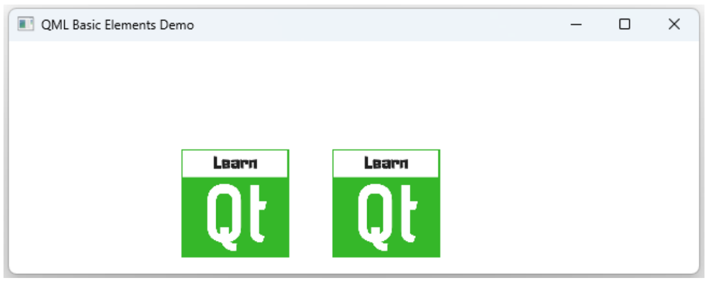


Figure 58. Image from Resource File

Using the resource system may seem complicated if this is your first time doing this, but it's the most flexible way to bundle data in your application and make sure it will be found without annoying issues. There are two more ways we can specify an image to be shown by the Image element : a full local path and the internet.

Loading Images From a Full Local Path

There isn't much to say here, you just specify a full path to the image as a value to the source property of your Image element. In our current Qt Creator project main.qml file, we can add a new Image element just below the last one

```
//...
Image {
    x : 150
    y : 50
    width: 100
    height: 100
    //Loading image from the resource file
    source: "qrc:/images/LearnQt.png"
}
```

```
//Load image from a full path
Image {
    x : 300
    y : 50
    width: 100
    height: 100
    //Specify the full path to the image
    source: "file:///D:/Artwork/LearnQt.png"
}
//...
```

If you run the application with these changes in place, you'll see three images: one coming from the working directory, one coming from the resource file and the last one coming from the full path.

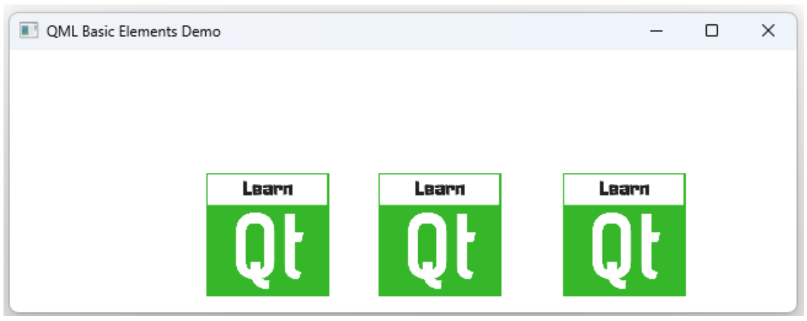


Figure 59. Image From Full Path

Notice that I used the file path convention of Windows because that's the system I am using. If you're on Linux or Mac, you'll have to adapt this to your system.

Loading Images From the Internet

You could also specify an http url as a value to the source property of your Image element. Add a new Image element to our current Qt Creator project main.qml file like below:

```
//...
//Load image from a full path
Image {
    x : 300
    y : 50
    width: 100
    height: 100
    //Specify the full path to the image
    source: "file:///D:/Artwork/LearnQt.png"
}

Image {
    x : 450
    y : 50
    width: 100
    height: 100
    //Specify the full web path to the image
    source: "https://www.learnqt.guide/images/qt_gui_intermediate.png"
}
//...
```

Run the application with these changes in place and you should see 4 images in your window: one coming from the working directory, one coming from the resource file, one coming from the full path and the last one coming from the internet.

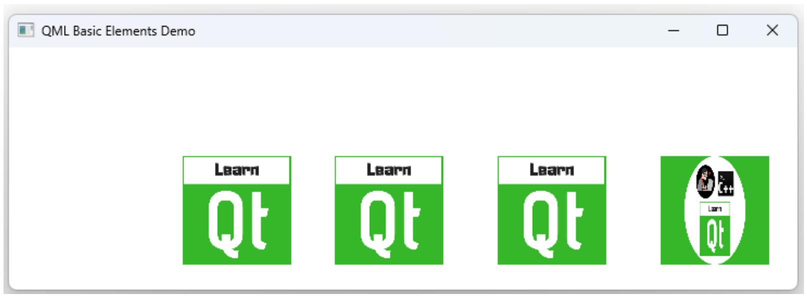


Figure 60. Image From The Internet

You have all these techniques to feed image data to your Image element and over time, you'll develop an intuition for which works best for your needs. The author almost exclusively uses the Qt resource system. This is because most of my projects run on devices with different configurations and the Qt resource system provides a unified way for data to be found by my application binary executables. Don't hesitate to experiment with all of these and find what you like best.

Custom Components

We have seen several built-in elements you can use in your QML graphical user interfaces like Rectangle, Text, Image, and built some applications to see them in action. In this section, we want to expose you to the idea that you can compose those elements and build your new custom elements.

Column

The Column element is used to lay things out vertically, from top to bottom. We are covering this element here because it will help us make a point later on in the section. Before we play with it in Qt Creator, I would encourage you to first look this element up in the documentation to develop your intuition about it. Create a brand new Qt Quick project from Qt Creator, name it and save it on your local drive. Change your main.qml file to be like below :

```
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    Column {
        Rectangle {
            width : 300;height: 100
            color: "red"
        }
        Rectangle {
            width : 300;height: 100
            color: "blue"
        }
    }
}
```

Based on what you read in the documentation, what do you think you will see if you run this application? If you haven't read the docs page for Column, the author encourages you to do so. We have two rectangles, one red, the other blue, laid inside a `Column` element. Notice that the rectangles are not positioned using anchors or explicitly through setting the `x` and `y` positions. Let's run the application to see what it'll look like :

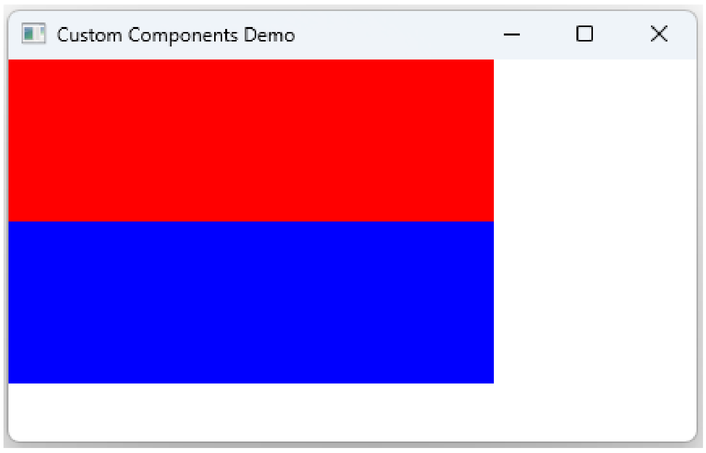


Figure 61. Rectangles in a Column Element

We can see that the rectangles are laid out vertically, the red one being on top and the blue one at the bottom. From this, we can gather that **the elements that show up first in the Column element are laid out on top of the elements that show up later on**. Notice that the Column element itself doesn't have any explicit positioning, so QML will position it in the top-left corner of the Window element. The Column element has several properties you can play with, if you look at the docs page, you'll see that we have a spacing property. Let's change our code like below

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    Column {
        spacing : 20
        Rectangle {
            width : 300
            height: 100
            color: "red"
        }
```

```
            Rectangle {
                width : 300
                height: 100
                color: "blue"
            }
        }
    }
```

and run the application. You'll see that we now have some space between the red and blue rectangles. That space is controlled by the `Column spacing` property.

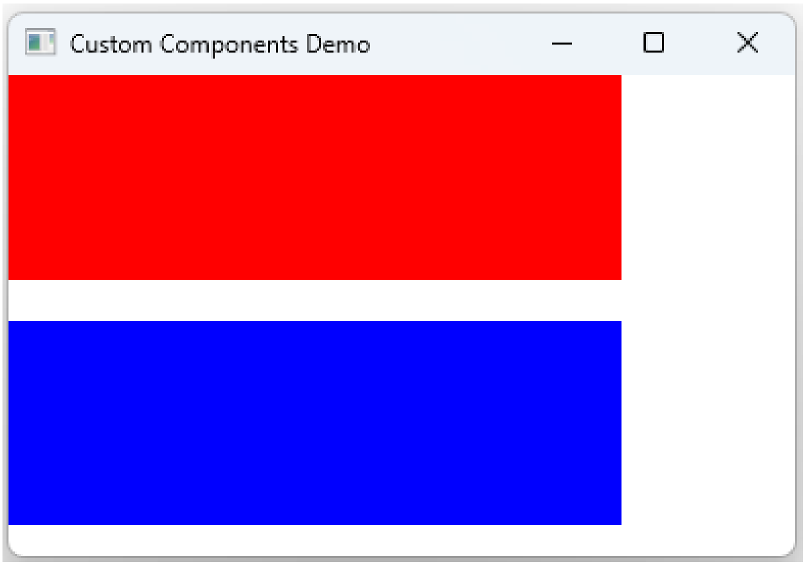


Figure 62. Column Spacing

Besides Column, we also have the Row element, which lays things out horizontally. Just like `Column`, it also has a spacing property you can use to control the space between elements. To see it in action, all you have to do is change Column to Row in our code and run the application. I leave that to you as an exercise but strongly encourage you to try this out. That's how you learn!

Building a Button

The goal of this section is to build a Button! We will model our button as a rectangle, but to make it visually appealing, we'll give it a color and a border. These are things we can achieve through a simple Rectangle element. Like any button that respects itself, however, the button should give us the ability to specify the text inside the button. We will use a Text element to control the text inside the button. Notice the keyword here: **inside**. In QML we will somehow put a **Text element inside a Rectangle** element. Let's start by changing our main.qml file to contain code below

```
import QtQuick


Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")
    Rectangle {
        id : containerRectId
        color: "red"
        border { color: "blue"; width : 3}

        //The width and height of the rectangle depend on the dimensions of the text
in buttonTextId
        width: buttonTextId.implicitWidth + 20
        height: buttonTextId.implicitHeight + 20

        Text {
            id : buttonTextId
            text : "Button"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
                console.log("Clicked on the button")
            }
        }
    }
}
```

Inside the Window, we have a Rectangle that visually represents the bounds of the button. The button is filled with a red color and has a 3 px wide blue border. The button should have some text, which we use the Text element to model. We want the text shown in the center of the red rectangle, so we anchor that in the center. Now comes the million-dollar question: what will be the dimensions of the rectangle? We want the size of the button to adapt to the size of the text in the button. In other words, we want the size of the button to depend on the size of the text inside the button. For that, we will take advantage of the implicitWidth and implicitHeight properties on our Text element :

```
//...
Rectangle {
    id : containerRectId
    //....
    //The width and height of the rectangle depend on the dimensions of text in
buttonTextId
    width: buttonTextId.implicitWidth + 20
    height: buttonTextId.implicitHeight + 20
}
//...
```

The implicitWidth and implicitHeight properties are inherited from the Item element, and they tell QML how wide and high the element would prefer to be. For the Text element, these properties take into account the size of the text, and Text prefers its size to be at least the width of the text inside, and as high as the text it contains. In our code, we rely on that and propagate that prefered size to the rectangle, adding a margin of 20 so our button has some breathing space. If you run the application, you should see the button in the top-left corner of the rectangle, and clicking on it should print out our console.log() message.

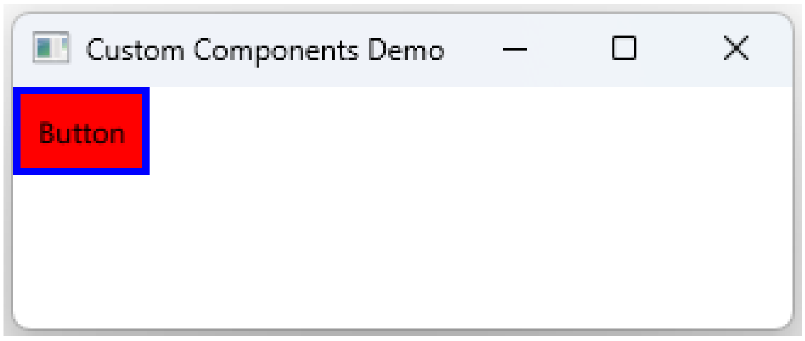
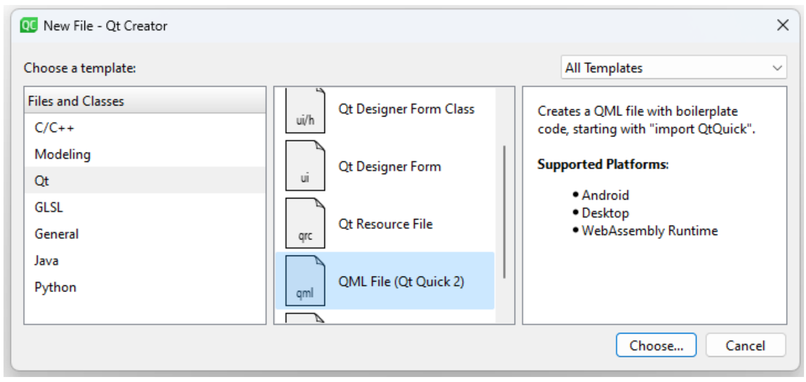


Figure 63. Red Button in main.qml

Our button is looking nice. But what if we need several of these buttons shown up on our user interface for example inside a Row or Column element? One option is to put all the code in main.qml and duplicate the button code to display all the buttons we need. But code duplication is a bad thing you should avoid as much as you can. QML provides a facility to wrap our button code in a custom component. Every time we need a button, we just instantiate that component and the button code will even live in its own .qml file, making its maintenance easy.

Adding a New QML File to The Project

Let's create a new QML file that is going to host our button code! This is going to be standard procedure but care should be taken to make sure CMake knows about the new file we're adding. From Qt Creator, issue the **File › New File** command. From the Qt template, choose QML File(Qt Quick X)



Creates a QML file with boilerplate code, starting with "import QtQuick".

Supported Platforms:

- Android
- Desktop
- WebAssembly Runtime

Figure 64. Creating a New QML File

Click "choose" and in the next window, specify the name for your new qml file. Mine is named `MButton`, I advise you to do the same for consistency in the book. Leave the path as is, and that'll save the qml file in the same location as the main.qml file of your project, which is what we want here. Click "Next". In the next window, you should be given the option to select a Version Control System like Git, I leave that to "None" in this project. Leave the rest to the defaults and click "Finish". The new file will open up in Qt Creator containing the starter code looking like below

```
import QtQuick

Item {

}
```

Qt Creator, when used with CMake is weird when adding new files to the project. After the file is added to the project, you have to explicitly edit the CMakeLists.txt file, to let CMake know about the added file. I haven't invested research effort to find out why but I am sure there is a good technical reason for this. We need to edit our CMakeLists.txt file and add a reference to our newly added MButton.qml file. We do that through the qt_add_qml_module() command :

```
qt_add_qml_module(app3-CustomComponents
    URI 3-CustomComponents
    VERSION 1.0
    QML_FILES main.qml MButton.qml
)
```

Just add `MButton.qml` after `main.qml`, save the file and CMake now knows about the `MButton.qml` file in our project. We now need to move our button code inside the MButton.qml file. Delete the Item element inside and move the Button code from main.qml to MButton.qml. After you do that, our MButton.qml file should look something like below. Now that we're starting to have multiple files in the project, the file of interest will be shown as a comment at the top of the code snipet.

```
//MButton.qml
import QtQuick

Rectangle {
    id : containerRectId
    color: "red"
    border { color: "blue"; width : 3}

    //The width and height of the rectangle depend on the dimensions of the text in
buttonTextId
    width: buttonTextId.implicitWidth + 20
    height: buttonTextId.implicitHeight + 20

    Text {
        id : buttonTextId
        text : "Button"
        anchors.centerIn: parent
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on the button")
        }
    }
}
```

Using the Custom Component

Having MButton.qml at the same location as the main.qml file in the project, and having the file registered to CMake as we did earlier, is enough to start using MButton as a type in our main.qml file. We can now change the main.qml file to contain the code below

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    MButton{

    }
}
```

and run the application to see the same old button. Now the fun begins! Change the main.qml file to contain two buttons like below

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    MButton{
    }
    MButton{
    }
}
```

I encourage you to take a moment and think about what you'll see if you run this application. Run the application, and you'll see one button, even if we have two buttons in our code. Can you figure out the issue here? The problem is an old one! If you don't explicitly position your elements, QML will stuff them in the top-left corner of your window, on top of each other. The second button is effectively laid out on top of the first button, hiding it from view. One way to make the two buttons show up is to tell the second button to position itself below the first button, using anchors. We will also set up IDs for our buttons to make our job easier :

```
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    MButton{
        id: button1
    }
    MButton{
        id: button2
        anchors.top : button1.bottom

    }
}
```

Run the application with these changes in place and you should see two buttons. Clicking on either will print the same console.log() message.

```
qml: Clicked on the button
```

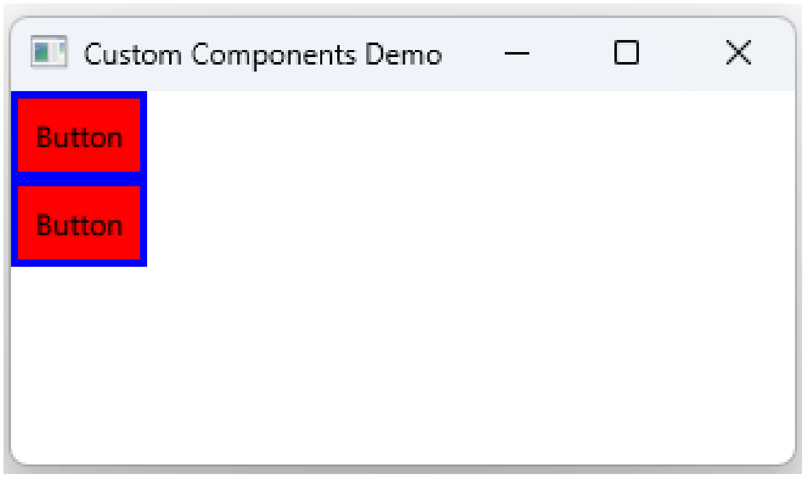


Figure 65. Two Confused Buttons

If you keep clicking on the two buttons, a problem will probably pop up in your mind: How do you tell these two buttons apart? We have two buttons instantiated in memory, but they show the same text, and they say the same console.log() message when clicked. We would like to exactly change that. They should show different text, for example, "button1" and "button2". When clicked they should console.log() a message saying which button was clicked. For example, "Clicked on button1".

Let’s first allow users of our MButton component to change the text in the button. We do that by exposing the text property of buttonTextId to the outside through a property alias

```
property alias buttonText: buttonTextId.text
```

We also change the click handler in the MouseArea to print the text of the button that's been clicked

```
MouseArea {
    anchors.fill: parent
    onClicked: {
        console.log("Clicked on " + buttonTextId.text)
    }
}
```

Putting it all together, our MButton.qml file should look like

```
//MButton.qml
import QtQuick

Rectangle {
    id : containerRectId
    color: "red"
    border { color: "blue"; width : 3}
    property alias buttonText: buttonTextId.text

    //The width and height of the rectangle depend on the dimensions of the text in
buttonTextId
    width: buttonTextId.implicitWidth + 20
    height: buttonTextId.implicitHeight + 20

    Text {
        id : buttonTextId
        text : "Button"
        anchors.centerIn: parent
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("Clicked on " + buttonTextId.text)
        }
    }
}
```

We can now change the main.qml file to be specific about the text in each button

```
//main.qml
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    MButton{
        id : button1
        buttonText: "Button1"
    }
    MButton{
        id: button2
        buttonText: "Button2"
        anchors.top : button1.bottom
    }
}
```

With these changes in place, run the application and you should see the buttons with our text. Clicking on either should print out which button was clicked.

```
qml: Clicked on Button1
qml: Clicked on Button2
```

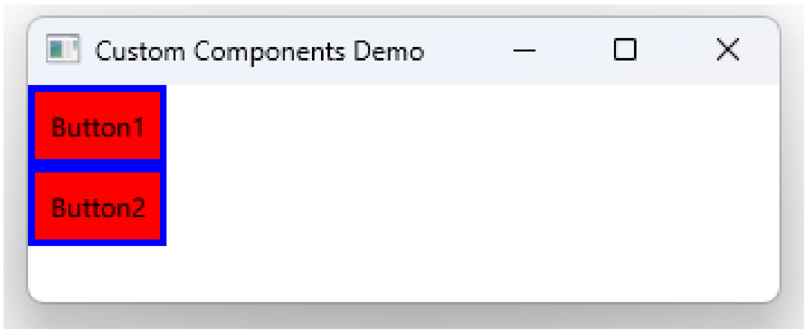


Figure 66. Buttons with Specific Text

Hiding Properties

Our application is using custom components just fine. But there is a problem. All properties of the rectangle modeling our button are exposed to the outside. This is the default behavior in qml.

The properties of the top-level element in your custom component QML file will be exposed to the outside world. They will be visible to the people instantiating and using your custom component. The common practice in QML is to wrap your custom component in an Item element, and explicitly expose the properties you want to be seen by the outside world.

For example, because to top-level element of MButton.qml is a Rectangle element, the color property is visible from the main.qml file and users of MButton can change the colors of the buttons

```
//main.qml
import QtQuick
```

```
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    MButton{
        id : button1
        buttonText: "Button1"
        color : "gray" // Change the color to gray
    }
    MButton{
        id: button2
        buttonText: "Button2"
        color : "yellow" // Change the color to yellow
        anchors.top : button1.bottom
    }
}
```

Running the application, you'll see the colors of the buttons changed accordingly.

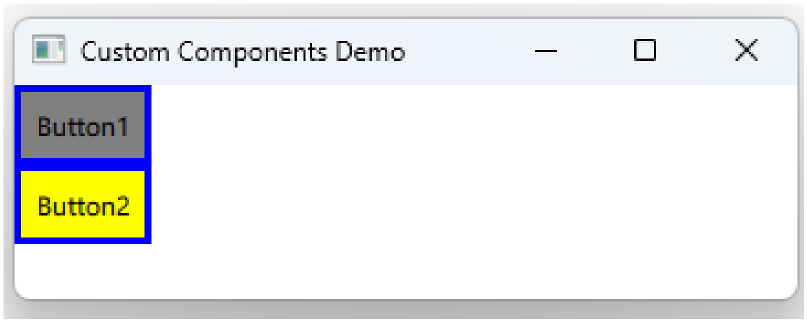


Figure 67. Changing Button Colors

This may be what you want depending on the design of your graphic user interface requirements. But for the sake of explanation, let's suppose we don't want the color property and other Rectangle properties exported to the outside. We can achieve that by making Item the top-level element in MButton.qml, and moving the buttonText property alias at Item root level. We also take the chance to set up meaningful IDs for our elements :

```
//MButton.qml
import QtQuick

Item{
    id: rootId
    property alias buttonText: buttonTextId.text
    Rectangle {
        id : containerRectId
        color: "red"
        border { color: "blue"; width : 3}

        //The width and height of the rectangle depend on the dimensions of the text
in buttonTextId
```

```
        width: buttonTextId.implicitWidth + 20
        height: buttonTextId.implicitHeight + 20

        Text {
            id : buttonTextId
            text : "Button"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
                console.log("Clicked on " + buttonTextId.text)
            }
        }
    }
}
```

If you try to compile and run the application with these changes applied in the MButton.qml file, you'll get an error

```
qrc:/path/to/main.qml:xx:xx: Cannot assign to non-existent property "color"
```

This is because the color property is no longer visible to the outside world and one can now say that it is private to the MButton component. This is a practice you'll see in a lot of QML code out there, you should take good note of it.

From the main.qml file, if we take out the code that tries to change the button color

```
//main.qml
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    MButton{
        id : button1
        buttonText: "Button1"
    }
    MButton{
        id: button2
        buttonText: "Button2"
        anchors.top : button1.bottom
    }
}
```

and run the application, we'll hope to see our buttons just like before. Surprise! You'll see just Button2. Button1 will be nowhere to be seen. Can you come up with the reason why? When we wrapped our Rectangle inside an Item element, size and position information was also hidden from the outside, and this will cause problems with QML positioning mechanisms like anchors, Row, and Column. You'll notice that even if we change the code in main.qml to use Column, we'll still only see Button2

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    Column{
        SButton{
            id : button1
            buttonText: "Button1"
        }
        SButton{
            id: button2
            buttonText: "Button2"
        }
    }
}
```

We still want to see our two buttons though. And make them play well with QML positioning mechanisms, while still making the color property private to the MButton component. One quick fix for this is to expose width and height information to the root level. Change your MButton.qml file like below

```
//MButton.qml
import QtQuick

Item{
    id: rootId
    property alias buttonText: buttonTextId.text

    //Expose width and height information to the outside
    //This makes MButton play well with QML positioning mechanisms
    width: containerRectId.width
    height: containerRectId.height

    Rectangle {
        id : containerRectId
        color: "red"
        border { color: "blue"; width : 3}
        //The width and height of the rectangle depend on the dimensions of the text
in buttonTextId
        width: buttonTextId.implicitWidth + 20
        height: buttonTextId.implicitHeight + 20

        Text {
            id : buttonTextId
            text : "Button"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
                console.log("Clicked on " + buttonTextId.text)
            }
        }
    }
}
```

Run the application with these changes applied to the MButton.qml file and you'll see your two buttons back, whether you're using anchors or the Column element to position the buttons vertically.

Get in the habit of making sure your custom components have width and height information exposed to the root level of the wrapping Item element.

Exporting the clicked signal

Our button is evolving nicely, but it lacks one thing buttons are supposed to do: responding when clicked. One could argue that our buttons are responding, but the response is hard-wired into the button logic itself. What if we don't want to print out which button was clicked but trigger some other mechanism somewhere else in our application? We can achieve that though the signal and slot mechanism in Qt. We'll get a chance to learn all about that mechanism in the next chapter, but right now we just want to make our button operational.

What we'll do here is declare a signal at the root level in our MButton.qml file, the signal name will be buttonClicked, and we'll trigger the signal when someone clicks in the MouseArea of the button. Users of MButton will then be able to set up handlers for that signal and do whatever they want. Change your MButton.qml file like below

```
//MButton.qml
import QtQuick

Item{
    id: rootId
    property alias buttonText: buttonTextId.text
    width: containerRectId.width
    height: containerRectId.height
    //Declare the signal
    signal buttonClicked

    Rectangle {
        id : containerRectId
        color: "red"
        border { color: "blue"; width : 3}

        //The width and height of the rectangle depend on the dimensions of the text
in buttonTextId
        width: buttonTextId.implicitWidth + 20
        height: buttonTextId.implicitHeight + 20

        Text {
            id : buttonTextId
            text : "Button"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {

                //Fire the signal
                rootId.buttonClicked()
            }
        }
    }
}
```

We also need to change the main.qml file to handle the signal for each of the buttons

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components Demo")

    Column{
        SButton{
            id : button1
            buttonText: "Button1"

            //handle the clicked signal in button1
            onButtonClicked: {
                console.log("Clicked on button1")
            }
        }
        SButton{
            id: button2
            buttonText: "Button2"

            //handle the clicked signal in button2
            onButtonClicked: {
                console.log("Clicked on button2")
            }
        }
    }
}
```

Apply these changes both in MButton.qml and main.qml. Run the application and you'll see your click handlers triggered when you click on either of the buttons. We are just printing out which button was clicked in main.qml but you can practically do anything within the body of the click handler.

Chapter Summary

Congratulations on making it this far in the book! We had a closer look at the basic QML elements: Item, Rectangle, Image and Text. The Rectangle element allows you to set up a box in your QML user interface. You can apply a host of properties to the box such as width, height, border and so on. We saw that many of the properties we have in Rectangle are inherited from Item. We saw that Item is the base type for all visual types in QML. From this, you can gather that Rectangle, Image and Text, all visual elements, pull a common set of properties from the Item element. Image is used to display Images and control how they look in your user interface. One of the most important properties of the Image element is the source property that you use to specify where the image to display will be coming from. We saw that there are four options: The image may be coming from the working directory, the Qt resource system, an internet URL, or from a full local path on your file system. The Text element is used to display text in your QML application. Just like other elements, it offers several properties you can use to control how your text looks. We also had a chance to explore how you can compose all these basic elements and build your own custom QML component. We saw how important it was to wrap your custom component QML code inside an Item element, allowing us to exactly control which properties are usable from the outside and which ones are not. Last but not least, don't forget to expose width and height information for your custom components. This helps them play well with Qt QML positioning mechanisms.

Chapter 4: Signals and Handlers

The signals and slots mechanism is one of the most important selling points of Qt. This mechanism allows you to communicate between different objects in your Qt project. The objects that want to share information send out a signal. Objects interested in the signal can tune in and be notified when the signal is emitted and do something through a slot, or a handler method. The signal and slot mechanism originated from the Qt C++ days back in the 90s but we can also use it in QML, and that's what we'll focus on in this chapter. We'll start out by getting your feet wet using handlers to respond to some things that happen in your QML application. We'll see that signals can send out parameters with additional information. We'll play with some property change handlers as they also are notifying us about some things changing in our app. We'll see that the Connections element allows us to hijack signals and handle them in places we originally wouldn't be able to handle them. Similar to the way we can have attached properties, we can also have attached signal handlers. We'll get a chance to play with them in the chapter. We'll show you how you can set up your own custom signals, show you that it's possible to connect a signal to a method or even connect a signal to another signal. We'll wrap out the chapter by looking at how one can send out signals across custom QML components, something that'll make your QML code very flexible.

Introduction

Signals and slots provide a powerful mechanism for communication and interaction between different components of a Qt application, enabling them to respond to events and signals emitted by other components in a flexible and decoupled manner.

At the core of the signals and slots mechanism is the concept of event-driven programming. Signals allow components in a Qt application to emit signals to notify other components of changes or events. Slots, on the other hand, are functions that can be registered with QML objects to handle these signals. Signals and slots facilitate loosely coupled communication between different parts of an application, allowing developers to create modular and extensible code.

In QML, signals are defined using the signal keyword followed by the name of the signal and its parameters, if any. For example, a QML object representing a button could define a signal called clicked that is emitted when the button is clicked. This is something we used in the Custom Components section without really understanding what is going on behind the curtains.

Slots, on the other hand, are implemented as functions in QML that can be connected to signals to handle them. Slots are declared using the function keyword and can be defined within QML objects or in external JavaScript files. For example, a QML object representing a button could have a slot called onClicked that handles the clicked signal.

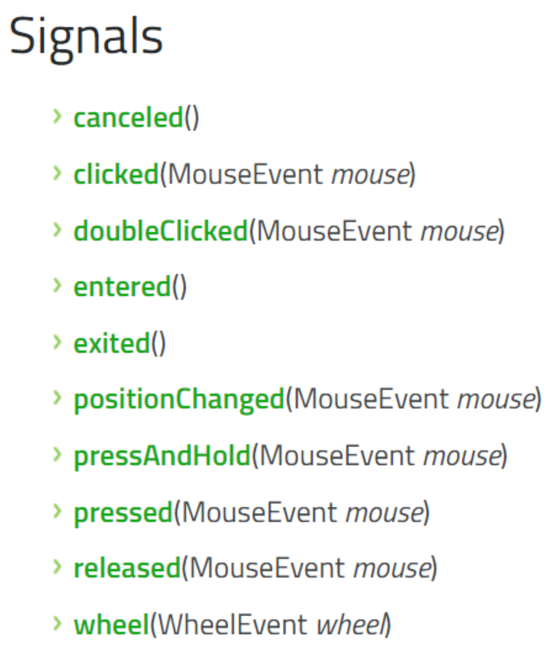
One way to connect a signal emitted by a QML object to a slot is to use the Connections object in QML. The Connections object acts as a mediator that connects signals and slots. You can specify the source object that emits the signal, the name of the signal, and the target object and slot to handle the signal. For example, you could connect the clicked signal of a button to the onClicked slot of another QML object. We'll get to learn about the Connections object later on in the chapter.

One of the powerful features of signals and slots in Qt is their ability to support multiple connections, allowing multiple slots to handle the same signal. This enables a highly decoupled and flexible architecture, where different parts of an application can react to the same event in their unique ways, without being tightly coupled to each other.

Signals and slots in QML provide a flexible and decoupled mechanism for handling events and signals, allowing developers to create modular and extensible code. In this section, we'll explore different facilities at our disposal, to take advantage of signals and slots in our QML applications.

Signal Handlers

A signal is something that is fired when something happens. For example, when you click on a button, the button can fire the `clicked` signal. For each signal, be it built-in or custom, QML generates an automatic handler, that you use to respond to the signal being fired. The handlers have a special naming convention in QML. They are named `on<SignalName>` with the first letter of the signal name capitalized. For example, the MouseArea element has a clicked signal. For that signal, there is an automatic handler named onClicked that you can use to respond to clicks. You can learn about available signals by looking at the docs page for a specific QML element. If you check out the docs page for MouseArea, you'll find a list of available signals, reproduced below for ease of reference :



Signals

- canceled()
- clicked(MouseEvent *mouse*)
- doubleClicked(MouseEvent *mouse*)
- entered()
- exited()
- positionChanged(MouseEvent *mouse*)
- pressAndHold(MouseEvent *mouse*)
- pressed(MouseEvent *mouse*)
- released(MouseEvent *mouse*)
- wheel(WheelEvent *wheel*)

Figure 68. MouseArea Signals

In your QML application, you can respond to any of these signals by setting up a handler named `on<SignalName>` with the first letter of the signal name capitalized. Below is a list of valid handlers for a MouseArea element.

```
onClicked: {
    console.log("Clicked on the rect")
}

onDoubleClicked: {
    console.log("Double clicked on the rect")
}
```

```
onEntered: {
    console.log("You're in!")
}
onExited: {
    console.log("You're out!")
}

onWheel: function(wheel) {
    console.log("Wheel : "+ wheel.x)
}
```

You can learn about each of these signals by clicking on them on the docs page for MouseArea. Let's play with a couple of signal handlers. Create a brand new Qt Quick project from Qt Creator. Name it and save it on your local drive. Change the code in your main.qml file to be like below

```
//main.qml
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Signal Handlers")

    Rectangle{
        id : rect
        width : 150
        height: 150
        color : "red"

        MouseArea{
            anchors.fill: parent
            onClicked: {
                console.log("Clicked on the rect")
            }
        }
    }
}
```

The code looks familiar. We've used the MouseArea element like 1000 times by now! But the concept is still important. MouseArea has a built-in signal named `clicked` and we can respond to the signal by putting our responding code in a handler named `onClicked`. If you compile and run the application, you'll see a red rectangle in the top-left corner of the window and it'll print out a console.log() message if you click on it. This is very powerful! Suppose that your QML graphical user interface is controlling a home automation system, and you can call a function to turn on all the lights in the house in your click handler. We also have a `doubleClicked` signal from MouseArea. You can play with it by adding an `onDoubleClicked` handler just under the `onClicked` handler:

```
//...
onDoubleClicked: {
    console.log("Double clicked on the rect")
}
//...
```

Run the application with these changes in place. Double-click on the rectangle and see the console.log() message printed out. We also have entered and exited signals. I invite you to read the docs for all these before we play with them. Change your main.qml file to add their handlers like below

```
//...
onEntered: {
    console.log("You're in!")
}
onExited: {
    console.log("You're out!")
}
//...
```

The entered signal is emitted or fired when the cursor pointer enters the area filled by the MouseArea. If you run the application and move the mouse on top of the rectangle without clicking, you'll notice that the signal isn't fired. Notice that you have to click inside the rectangle, then move in and out for the signals to be fired. If you want to fire the signals even without having clicked before, you can use the hoverEnabled property. I invite you to read about it in the docs.

```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Signal Handlers")

    Rectangle{
        id : rect
        width : 150
        height: 150
        color : "red"

        MouseArea{
            anchors.fill: parent
            hoverEnabled: true

            onClicked: {
                console.log("Clicked on the rect")
            }
            onDoubleClicked: {
                console.log("Double clicked on the rect")
            }
            onEntered: {
                console.log("You're in!")
            }
            onExited: {
                console.log("You're out!")
            }
        }
    }
}
```

Run the application with these changes in place, and you'll notice that the handlers are triggered when you move the mouse in and out of the rectangle. MouseArea also has a `wheel` signal. This is fired when you turn the wheel on your mouse. You can set up a handler like below

```
onWheel: function(wheel) {
    console.log("Wheel : "+ wheel.x)
}
```

to respond when the wheel is turned, with the mouse being on top of the area filled by the MouseArea element. This is the recommended syntax for signal handlers starting from Qt 6, especially when you are also processing parameters like we are doing here. We'll get to learn all bout signal parameters later on in the chapter, but handling the wheel signal makes this project more interesting and I couldn't resist! But signal parameters are nothing to be afraid of. They're just additional pieces of information that the signal can pass around. For example, the wheel signal can let us know the location in the rectangle, where the wheel was turned and we get access to that by calling `wheel.x` and `wheel.y`. Of course, you can learn about these signal parameters by reading the docs for the signal of interest. We just played with a couple of signals in this section, but I strongly encourage you to investigate signals for new QML elements you come in contact with in your career as a QML developer. Doing that slowly builds your intuition and it's incredibly helpful when you get to build practical projects. Over time, you'll get to know exactly where to look to solve a problem at hand.

Signal Parameters

Some signals can add more information to the signal when fired. For example when you click with the mouse, sometimes you want to know the exact position of the click. If you look up the clicked signal from MouseArea in the docs, you'll see that it has a parameter named `mouse`, whose Type is `MouseEvent`. You know what to do, look up MouseEvent in the docs. If you do, you'll see that, among others, it has `x` and `y` properties you can use to know the exact location of the click. Other signals may have one or more parameters of different types. Later on in the chapter, we'll even see how you can send out custom parameters with your custom signals. But for now, let's just explore the different syntax variations QML offers to process signal parameters in your QML applications. The goal is to know the exact `x` location for the click in the window. Here is the first variation

```
//Signal Parameter Processing: Syntax Variation #1
//...
MouseArea{
    onClicked: {
        //Deprecated
     console.log(mouse.x)
    }
}
//...
```

Notice that in the body of the handler, we just use the name of the MouseEvent parameter `mouse`. We know this parameter name because it's what is used in the signal docs entry. I always found this variation confusing as it always seemed like the mouse parameter name was coming out of the blue. Luckily, it was marked for deprecation in Qt 6 and you can think that it's on its way to extinction, meaning that it may be completely taken out in future versions of Qt. Let's look at the second variation.

```
//Signal Parameter Processing: Syntax Variation #2
//...
MouseArea{
    //Explicit Javascript function
    onClicked: function(mouse){
        console.log(mouse.x)
    }
}
//...
```

Here you see that the name of the signal parameter shows up as a function parameter. We even reserve the right to change the name of the parameter and QML will still take this.

```
//Signal Parameter Processing: Syntax Variation #2
//...
MouseArea{
    //Explicit Javascript function
    onClicked: function(mouse_param){
        console.log(mouse_param.x)
    }
}
//...
```

The third syntax variation takes advantage of Javascript arrow functions

```
//...
MouseArea{
    //Signal Parameter Processing: Syntax Variation #3
    //Javascript Arrow Function
    onClicked: (mouse_param) => console.log(mouse_param.x)

}
//...
```

With variation #3, you also reserve the right to change the name of the signal parameter. I find myself pivoting towards variation #2 in my own projects, but #2 and #3 are all fine. Just try to stay clear of #1 as it makes your code less readable. You don't have to memorize these things. I would just recommend being aware of them and using the documentation when you need the specifics. Let's see all these variations in action. Create a brand new Qt Quick project from Qt Creator. Name and save it somewhere on your drive. Change the code in your main.qml file to something like below

```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Signal Parameters")

    Rectangle{
        id : rect
        width : 150
        height: 150
        color : "red"

        MouseArea{
            anchors.fill: parent
            onClicked: {
                //Deprecated
                console.log(mouse.x)
            }

            /*
            //Explicit Javascript function
            onClicked: function(mouse_param){
                console.log(mouse_param.x)
            }
            */

            /*
             //Arrow function
            onClicked: (mouse_param) => console.log(mouse_param.x)
            */


        }
    }
}
```

You can see that all three variations are in, but you'll have to activate each one and disable the other two. In this case variation #1 is activated. If you run the application you'll see the console.log() message when you click somewhere in the red rectangle. Use block comments to comment out variation #1 and activate #2, and #3.

You can't have more than one variation activated at the same time. If we did that, we'd have two conflicting signal handlers trying to achieve the same thing. The QML engine wouldn't know which one to use and you'll get an error. Try this out!

Property Change Handlers

We covered signal handlers in a previous section of the chapter. In this section, the focus is to contrast signal handlers with property change handlers in a much more visible way. Signal handlers are triggered when a signal is emitted; be it a built-in signal or a custom one. Property change handlers are triggered when something about the property changes; be it a built-in property or a custom one. Let's take a closer look at this. Create a brand new Qt Quick project from Qt Creator. Name it and save it somewhere on your file system. Change your main.qml file like below

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Property change handlers")

    Rectangle{
        id : rect
        width : 300
        height : width
        color : "dodgerblue"

        property string description : "A rectangle to play with"

        onWidthChanged: function(){
            console.log("Width changed to :"+ rect.width)
        }
        onHeightChanged: function(){
            console.log("Height changed to :"+ rect.height)
        }
        onColorChanged: {}
        onVisibleChanged: {}
        onDescriptionChanged: {}

        MouseArea{
            anchors.fill: parent
            onClicked: {
                rect.width = rect.width+20
                }
            }
        }
    }
```

We know that the Rectangle Item has built-in properties like `width`, `height`, `color` and `visible`. For any property in your QML elements, be it built-in or custom, QML will automatically set up a handler that is triggered when the value of the property changes. The naming convention is `on<PropertyName>Changed` with the first letter of the property name capitalized. In our application here, we should have access to handlers such us `onWidthChanged`, `onHeightChanged`, `onVisibleChanged` and `onColorChanged`.The same applies to custom properties. We have `description` set up as a custom property, so we should have access to a handler named `onDescriptionChanged` as well.

The application is set up in a way that there is a property binding between the width and the height. If we change the width in the click handler of the MouseArea, the height will change as well, keeping our rectangle a square. As `width` and `height` change, the handlers for the respective property change handlers will be triggered. You should see output similar to the one below

```
qml: Height changed to :320
qml: Width changed to :320
qml: Height changed to :340
qml: Width changed to :340
//...
```

if you compile and run the application.

I often rely on IntelliSense from Qt Creator to quickly get a list of handlers available to me in a given context. Just start typing the name of the handler you think you need and hit `Ctrl` + `Space`.

Connections

The Connections element allows us to respond to signals outside of the context in which they were fired. In other words, it allows us to hijack signals and handle them anywhere we want in the application. All we need is the ID of the element emitting the signal. As always, I encourage you to open the docs page for this element and read about it a bit yourself. In this section, we will be hijacking the clicked signal of a MouseArea and handling it in the parent scope of the MouseArea. Something we wouldn't be able to do otherwise. Create a brand new Qt Quick project from Qt Creator. Name it and save it somewhere on your drive, and change your main.qml file like below :

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Connections")

    Rectangle{
        id : rectId
        width : 200
        height: 200
        color : "blue"

        MouseArea{
            id:mouseAreaId
            anchors.fill: parent
        }
    }

    Connections  {
        target : mouseAreaId
        function onClicked(){
            console.log("Hello")
        }
        function onDoubleClicked(mouse){
            console.log("Doubleclicked at : "+mouse.x)
        }
    }
}
```

The code looks familiar with a simple MouseArea filling a Rectangle. Usually, we would set up our handlers inside the MouseArea. But we're not doing that here. We'll use our `Connections` superpowers! **The Connections element hijacks handlers from the MouseArea by specifying the ID of the MouseArea as the value of its `target` property**. Once `target` is defined, the deal is done! You'll set up handler methods just like you would if you were inside the actual source of the signal.

Compile and run the application. You'll see a blue rectangle in the top-left corner of the window. Click and double-click in the window and you should see console.log() messages printed out

```
qml: Hello
qml: Doubleclicked at : 84
```

If you read the docs page for the Connections element, you've seen that it is crucial in cases such as when:

- Multiple connections to the same signal are required
- Creating connections outside the scope of the signal sender
- Connecting to targets not defined in QML

Connecting to targets not defined in QML is my favorite, as I get the ability to react to things coming from Python/C++, and do the response in QML. Communicating with Python/C++ is a subject beyond this book, so we won't go any further on the topic.

Attached Signal Handlers

Attached signal handlers are used to respond to things coming from attached objects. We have been using code like

```
Component.onCompleted : {
    //Do something
}
```

without really understanding what is going on. `Component` is a type that can be used to do a number of things in QML. Here we're interested in its attached signals. Attached signals come from attached objects. Attached objects provide a way to define properties and functions that can be attached to existing objects. They allow us to extend the functionality of objects without subclassing them.

`Component` is an attached object we have access to in visual elements like `Rectangle`

```
Rectangle{
    width : 200
    height: 200
    color : "green"
    anchors.left: parent.left

    Component.onCompleted: {
        console.log("Finished setting up the rectangle")
    }
}
```

even if `Component` is not defined anywhere in our QML files. The attaching is done by the QML engine. Attached objects may have properties and signals we can use to do things in QML. In our case, Component offers two attached signals, one of them being `onCompleted`. The docs state that the `onCompleted` signal from `Component` is emitted *when the object has been instantiated. This can be used to execute script code at startup, once the full QML environment has been established*. So anytime you need to know when some element in QML finishes instantiation in memory, you can use this attached signal handler, anywhere in your QML files. Hopefully, you can see how powerful this is. Let's play with this. Create a brand new Qt Quick project from Qt Creator, and change the code in your main.qml file to be like below

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Attached signal handlers")

    Rectangle{
        width : 200
        height: 200
        color : "green"

        Component.onCompleted: {
            console.log("Finished setting up the rectangle")
        }
    }
}
```

Compile and run the application, and you should have a green rectangle showing up on the screen. The most important thing to look out for in this example, however, is the **Application Output** pane in Qt Creator, which should have a message like below

```
qml: Finished setting up the rectangle
```

Please notice a few things here, our rectangle element can't know when it finishes being instantiated because it doesn't create itself. The QML engine which creates objects knows. A Component object is automatically attached to the rectangle, and it's through that that we're able to process the `Component.onCompleted` handler letting us know that the rectangle is fully created. The completed signal isn't emitted by the Rectangle here, it's emitted by some Component object hosted by the QML engine. I encourage to even set up another Component.onCompleted handler in the scope of the Window element and print out a message inside. This is often useful for logging and debugging your QML apps.

`Component` is one example of a type with attached signals for which we can set up attached signal handlers, but it's not the only one. There are lots of other types, such as `Keys` we can use to handle input in our QML applications. We'll get to learn about all these later on in the book. Please know that on top of built-in attached objects, QML also offers the ability to build your own attached types and their signals if you want to go down that road. We won't be covering that in this introductory book though. That's left for more advanced books on QML.

Custom Signals

We have been using all kinds of built-in signals from QML. In this section, we'll see how you can set up your own custom signal if the problem at hand requires it. Examples always speak better than convoluted explanations. Suppose you wanted to define a signal named `greet` in your QML file. The signal should also send out a string parameter containing the actual greeting message. We can set up such a signal with code like below

```
signal greet(string message)
```

The `signal` keyword is mandatory; it should be there. You have the flexibility to name the signal anything you want. For example you could have named it sayHi

```
signal sayHi(string message)
```

You could also set up zero or more signal parameters. If we wanted just the signal without any parameters, we could change our signal definition to be like

```
signal greet()
```

Please note that the pair of parentheses is still there. It just so happens to be empty. Now comes the million-dollar question: where, in the QML file, do you define the signal? You can really set that up anywhere you want, the answer will depend on the specific design of your application. For example, in our example below, the signal is set up inside a Rectangle element

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Custom Signals")

    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //Set up the signal
        signal greet(string message)

    }
}
```

You could move the signal definition a the root level in the Window element and it could work just as well. The specifics will depend on the actual design of your application.

Automatic Handlers

Once you have the signal, QML will set up an automatic handler following the convention `on<SignalName>` with the first letter of the signal name capitalized. In other words, once the greet `signal` is defined, there should be a handler named `onGreet` that we can activate from the scope where the signal was defined. Knowing this, we can change the code in our main.qml file to print out something and show the actual parameter that was sent from the signal

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Custom Signals")

    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //Set up the signal
        signal greet(string message)

        //Once you set up the signal, QML will setup a signal handler automatically
        onGreet: function(message){
            console.log("Greeting with message : "+ message)
        }

    }
}
```

Please note that the signal handler needs to show up in the same scope where the signal was defined, in our example, it should show up inside the rectangle because that's where the signal is defined. To make sure this is clear, change your code to move the signal handler out of the Rectangle, and put it inside the Window element

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Custom Signals")

    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //Set up the signal
        signal greet(string message)
    }

    //THE HANDLER ISN'T IN THE SAME SCOPE WHERE THE SIGNAL WAS DEFINED
    //YOU'LL GET AN ERROR IF YOU RUN THE APP
    onGreet: function(message){
        console.log("Greeting with message : "+ message)
    }
}
```

Qt Creator should give you hints that something is wrong. If you run the application, it won't run and show up. Instead, you'll see some error message hinting to the fact that the QML engine doesn't have a clue what onGreet is, despite having it defined inside the rectangle.

```
QQmlApplicationEngine failed to load component
qrc:/6-CustomSignals/main.qml:52:5: Cannot assign to non-existent property "onGreet"
```

Trying to go through the rectangle ID won't help either

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Custom Signals")

    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //Set up the signal
        signal greet(string message)
    }

    //Trying to go through the ID to handle the signal outside the scope
    //where it was defined. WON'T WORK EITHER.
    rectId.onGreet: function(message){
        console.log("Greeting with message : "+ message)
    }
}
```

The signal handler has to be inside the same element where the signal was defined.

Triggering the Signal

We have the signal defined, we have a handler ready to respond when the signal is fired, the missing piece of the puzzle is to trigger the signal. The syntax to trigger the signal is simple as well. You call it like you would call any Javascript function passing in any parameters where needed.

```
MouseArea{
    onClicked: {
        //Fire the signal by just calling it like you call a function
        rectId.greet("The sky is blue")
    }
}
```

Notice that because the we go through the rectangle ID, we have the flexibility to fire the signal anywhere we want from our QML file, provided we have access to the ID. To put all this together and have something fun running, create a Qt Quick project from Qt Creator and save it somewhere. Change the code in our main.qml file to be like below

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Custom Signals")

    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //Set up the signal
        signal greet(string message)

        //Once you set up the signal, Qt setup a signal handler automatically
        onGreet: function(message){
            console.log("Greeting with message : "+ message)
        }

        MouseArea{
            anchors.fill: parent
            onClicked: {
                //Fire the signal by just calling it like a function
                rectId.greet("The sky is blue")
            }
        }
    }
}
```

Notice that the MouseArea is filling the rectangle. When someone clicks in the area filled by the MouseArea, we fire our signal. We pass the text "The sky is blue" as a parameter to the signal. The text will be received by whatever piece of code is responsible for handling the signal. In our case, it's the `onGreet` handler, which prints out the message parameter. Running the application, you should see output like below

```
qml: Greeting with message : The sky is blue
qml: Greeting with message : The sky is blue
```

This should give you enough tools to

- define your own custom signals
- set up handlers for the signals
- fire the signals in your QML code where needed

Connecting a Signal to a Method

In the last couple of sections, we saw how signal handlers are triggered when a signal is fired. But automatic signal handlers are just one option. We also have the option to set up a custom function and trigger it when the signal is fired. The triggering is done by connecting the signal to a custom handler. This is often more helpful than going through regular automatic handlers because we get the ability to connect a signal to multiple handlers, or multiple handlers to a single handler. Of course, all these are options, you get to decide what makes the most sense for your project. Assuming you have your signal, let's use a custom one in the example code here, and your method like shown in the code below

```
//main.qml
import QtQuick
Window {
    //...
    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //The signal
        signal greet(string message)

        //The method, also called slot in Qt terminology
        function respond_your_way(message){
            console.log("Responding our way;Greeting with message : "+ message)
        }
    }
}
```

we can connect the signal to the slot(method) using the syntax below

```
//Make the connection explicitly
rectId.greet.connect(rectId.respond_your_way)
```

in English, you access the signal, call the connect method and specify the method in a pair of parentheses. Notice that the signal and the method could live anywhere in your QML application. You just need to be able to access them through IDs. Another thing worthy of note is that the signal parameters don't show up in your connection statement. QML will take care of that behind the scenes. If you need to process parameters, however, you need to take them into account in your custom method. We need to see this in action. Create a brand new Qt Quick project from Qt Creator and change your main.qml code to be like below

```
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Connect Signal to Method")

    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //Set up the signal
        signal greet(string message)

        //We want to connect, not to a built-in signal handler,
        // but to a custom regular function
        function respond_your_way(message){
            console.log("Responding our way;Greeting with message : "+ message)
        }

        MouseArea{
            anchors.fill: parent
            onClicked: {
                //Fire the signal by just calling it like a function
                rectId.greet("The sky is blue")
            }
        }
        Component.onCompleted: {
            //Make the connection explicitly
            rectId.greet.connect(rectId.respond_your_way)
        }
    }
}
```

It's nothing you haven't seen at this point, except for two things. We added the code to fire the signal. It's fired when someone clicks in the MouseArea that fills our rectangle. We also make the connection from the signal to the slot(method). An important question arises here. Where to make the connection? You could make it anywhere in your app. For example, you could make it in a method but the app will wait for the method to be called to make the connection. The common location, in my experience, is the attached `Component.onCompleted` handler of some element. Here, we set up the connection when the Rectangle is properly set up in memory. If you compile and run the application, you should see our good old dodgerblue rectangle. Click on it and the message should show up in the application output pane from Qt Creator.

```
qml: Responding our way; Greeting with the message: The sky is blue
```

The signal and slot mechanism is one of the powerful features of Qt, allowing us to transfer data among components in our application. This was just an introduction to the syntax. We'll be seeing more of them in use as we progress in the book.

Connecting a Signal to a Signal

Qt also provides the flexibility to connect a signal to another signal. The syntax is almost the same as for connecting signals to slots, but you specify another signal as a parameter to the connect method. Assuming we have two signals

```
//Set up the signals
signal greet(string message)
signal forward_greeting(string message)
```

we can set up a connection like below

```
//Connect a signal to another signal
rectId.greet.connect(rectId.forward_greeting)
```

For things to happen in your application, you somehow need to connect the last signal, in this case, `forward_greeting`, to an actual slot(method). You can chain as many signals as you want if it makes sense for your application. Let's see this in action. Create a brand new Qt Quick project from Qt Creator and change the code in main.qml like below

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Custom Signals")

    Rectangle{
        id : rectId
        width : 300
        height : 300
        color : "dodgerblue"

        //Set up the signals
        signal greet(string message)
        signal forward_greeting(string message)

        //The slot(method)
        function respond_your_way(message){
            console.log("Responding our way;Greeting with message : "+message)
        }

        MouseArea{
            anchors.fill: parent
            onClicked: {
                rectId.greet("The sky is blue")
            }
        }
        Component.onCompleted: {
            //Connect a signal to another signal
            rectId.greet.connect(rectId.forward_greeting)
            //Respond to the final signal
            rectId.forward_greeting.connect(rectId.respond_your_way)
        }
    }
}
```

The magic happens in the Component.onCompleted handler. We chain our signals and finally connect a slot to the final signal in the chain. You should also notice that the first signal in the chain is fired when we click on the rectangle. If you run the application and click on the rectangle, the `greet` signal will be fired. Our signal-to-signal connection will cause the `forward_greeting` signal to be fired as well. We have a slot specifically wired to respond to the `forward_greeting` signal and the code in the slot will execute, causing the message

```
qml: Responding our way; Greeting with the message: The sky is blue
```

to be printed in the **Output Pane** from Qt Creator. Connecting signals to signals isn't a thing I do often but it can come in handy when you have a lot of moving parts in your QML application. It's a good idea to just be aware of this feature and take advantage of it when necessary.

Signals and Slots Across Components

In this section, we want to tighten the screws a bit more and look at how one can use signals and handlers to transfer data across different QML components. As you already know, custom components can live in their own QML files. We will have two components: one called Sender, that'll send data and the other called Receiver, that'll be receiving data. The program will look like the image below when run.

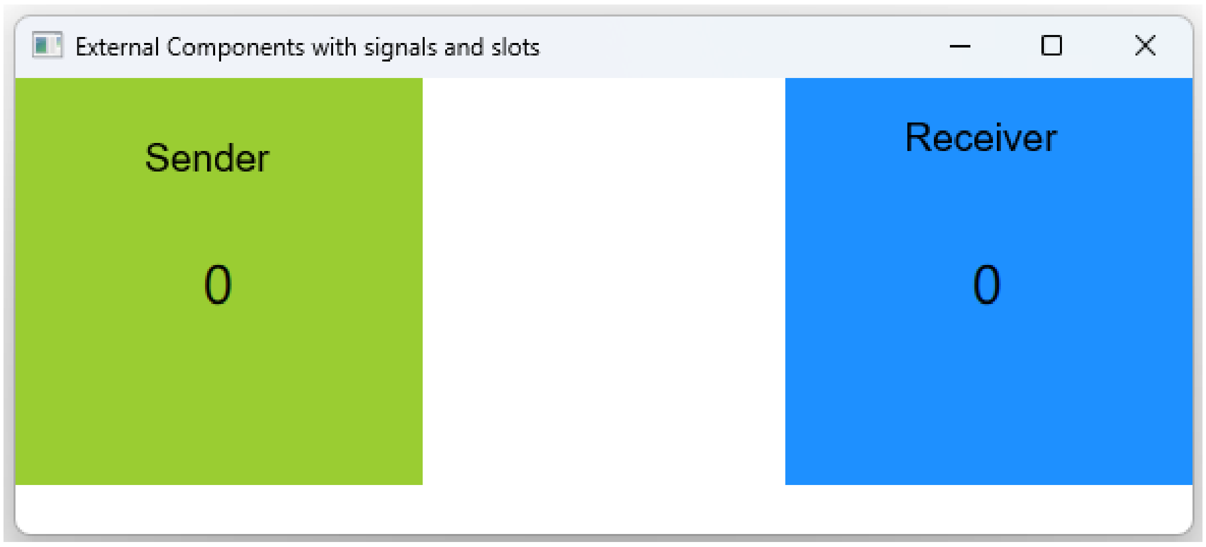


Figure 69. Signals Across Components

The rectangles will both start with a value of 0. When you click on Sender, it'll increment its value and send out a signal with the new value. The signal from Sender will be wired to a method from Receiver, that'll update the value to sync with what we have in the sender. In other words, the data will be traveling from Sender to Receiver through a signal slot connection. Our Sender and Receiver code will be hosted in external components, giving us the flexibility to have simple code in the main.qml file, something like code below

```
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("External Components with signals and slots")

    Sender{
        id : notifierId
    }

    Receiver {
        id : receiverId
        anchors.right: parent.right
    }
}
```

Looking at the application image, one can get an idea that `Sender` will be made up of a `Rectangle` with a `Text` element inside to display the current value. `Sender` should also have a `MouseArea` filling the rectangle, giving us the ability to process click events. Following the same logic, Receiver is also made up of a Rectangle with a Text element inside. `Receiver` doesn't need any `MouseArea` because it won't be processing click events. It'll get its data from the sender through a signal slot connection. Let's do this. Start a brand new Qt Quick project from Qt Creator and leave the main.qml file as is. You will need to add two new QML files to contain the code for Sender and the code for Receiver. The files will be named **Sender.qml** and **Receiver.qml**. You add a new QML file to the project by right-clicking on the app folder in the Qt Creator project viewer and selecting **Add New** and choosing **Qt** and **QML File (Qt Quick2)**. After that, you'll just follow the instructions and remember to name the files **Sender.qml** and **Receiver.qml**. After you go through the wizards to add the new QML files, you should remember to go back and register these in your CMakeLists.txt file. You do that by modifying the `qt_add_qml_module` command as shown below

```
qt_add_qml_module(app9-DifferentQMLComponentSignals
    URI 9-DifferentQMLComponentSignals
    VERSION 1.0
    QML_FILES main.qml Sender.qml Receiver.qml
)
```

What this says is that our application is made up of 3 QML files. Save your CMakeLists.txt file with the changes. Now the fund begins; we need to build our `Sender` and `Receiver` components. Change your Sender.qml file to contain code like below

```
//Sender.qml
import QtQuick
Item {
    width: notifierRectId.width
    height: notifierRectId.height
    property int count: 0
    signal notify( string count)//Declare signal

    Rectangle {
        id : notifierRectId
        width: 200
        height: 200
        color: "red"

        Text {
            id : displayTextId
            anchors.centerIn: parent
            font.pointSize: 20
            text : count
        }

        MouseArea{
            anchors.fill: parent
            onClicked: {
                count++
                notify(count)
            }
        }
    }
}
```

Our component is wrapped in an Item element because we want the flexibility to hide properties from the outside by default, and only expose those we need to be visible to the outside by hosting them a the root level of the wrapper Item. The Item element contains a Rectangle item. Inside `notifierRectId` we have 2 elements, a `Text` element to display the current count and a `MouseArea` to process click events. Notice that the text property of displayTextId is bound to `count`, `count` itself being a custom property we defined at the root level. We also have a signal named `notify` defined, whose job is to send the current count out to be received by receiver elements. When the MouseArea is clicked, we increment the count and fire the signal. Notice that our signal has a string parameter, but the count property is an integer. In simple cases like this, QML will transform the data from int to string. The last thing you should note is that we are exposing the width and height to the root level, to help our component play well with positioning mechanisms in QML.

The code for the Receiver element is similar and simpler. Change your Receiver.qml file to contain code like below

```
//Receiver.qml
import QtQuick
Item {
    width: receiverRectId.width
    height: receiverRectId.height

    Rectangle {
        id : receiverRectId
        width: 200
        height: 200
        color: "blue"

        Text {
            id : receiverDisplayTextId
            anchors.centerIn: parent
            font.pointSize: 20
            text : "0"
        }
    }
}
```

With these in place, we're ready to try and use these in our main.qml file. Open it up and change the code to be like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("External Components with signals and slots")

    Sender{
        id : notifierId
    }
    Receiver {
        id : receiverId
        anchors.right: parent.right
    }
}
```

The notifier object will be shown in the top-left corner of the window, the default behavior for any object without any explicit positioning mechanism. The receiver object is explicitly anchored to the right side of the window. If you run the application with the current changes applied, you should see a window like below

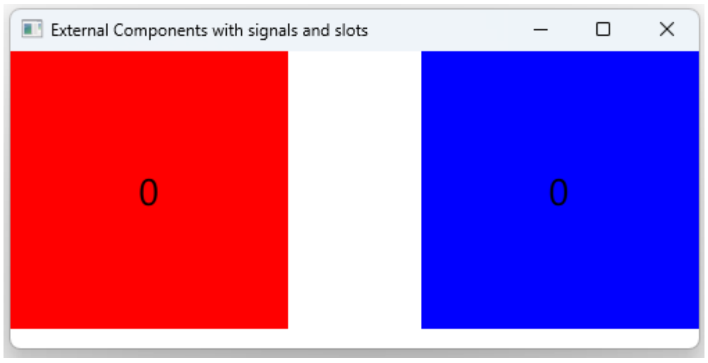


Figure 70. External Components Initial State

If you click on the sender, you'll see the value incrementing. This is because the `count` value is incremented in the click handler, and it's also bound to the text in the Text element. We also fire the signal in the click handler, but nothing is listening for the signal at the moment. The sender is shouting in the air but no one is there to listen. We need to change that.

Responding to the signal

At this point in the book, we are used to responding to signals from the same place where they originate, we have explored signal handlers that should be in the same element where the signal originates from and signal-to-slot connections that help us communicate between different elements. They also provide the flexibility to connect a single signal to multiple slots, or multiple signals to a single slot. In this section, the challenge is to use signals and slots to communicate between different QML components. There are three approaches we can take and we'll explore them one by one. Before we do that, though, let's set up the function to receive data in the receiver component.

Setting up the Receiver Slot

We will change our receiver component to add a slot(method) that will receive the data. Our modified Receiver.qml file should look like below

```
//Receiver.qml
import QtQuick
Item {

    width: receiverRectId.width
    height: receiverRectId.height
    function receiveInfo( count){
        receiverDisplayTextId.text = count
        console.log("Receiver received number : "+ count)
    }

    Rectangle {
        id : receiverRectId
        width: 200
        height: 200
        color: "blue"

        Text {
            id : receiverDisplayTextId
            anchors.centerIn: parent
            font.pointSize: 20
            text : "0"
        }
    }
}
```

We added the function to receive the information from the sender and the function takes a single parameter named count. In the body of the function, we use the data in the parameter to set the value to the Text element inside our receiver Rectangle element. With the receiver slot in place, we now need to make the connection from the signal to the slot. There are different approaches to this.

Making the connections in the main file

One way to get the data from Sender to Receiver is to make the connection in some Component.onCompleted handler in the main.qml Window element. Ours will be hosted at the root level of the Window element as shown in the code below.

```
//main.qml

import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("External Components with signals and slots")

    Sender{
        id : notifierId
    }

    Receiver {
        id : receiverId
        anchors.right: parent.right
    }

    Component.onCompleted: {
        notifierId.notify.connect(receiverId.receiveInfo)//Connect signal to slot
    }
}
```

We access the notifier signal and go through it to make the connection to the receiver's `receiveInfo` slot. If you compile and run the application, you'll see that as you click on the sender, the receiver is updated with the data we have in the sender.

Using a target property in Sender

One other arrangement we can do to make the connection is to hook the receiver to the sender inside the sender component code. This will rely on the fact that for every property in QML elements, there will be an `on<PropertyChanged>` handler you can use to respond to changes in the property. We can set up a receiver property in Sender.qml

```
//...
property Receiver target : null
//...
```

The property will be set up at the root level of the Item element inside Sender.qml. This will allow Sender elements instantiated in main.qml to be able to assign values to that property. With the property in place, we have automatic access to the property change handler that we set up as below.

```
//...
onTargetChanged: {
    notify.connect(target.receiveInfo)
}
//...
```

The handler does the most important thing here. Connecting the signal to the slot. The value of the `target` property is going to change when somebody assigns a value to it in instantiated `Sender` elements. Because `target` is of `Receiver` type, we can have access to the slot through what is assigned to `target` and make the connection as we do in the body of the property change handler. The assignment to the target property will happen somewhere in the main.qml file that instantiates both Sender and Receiver.

```
Notifier{
    id : notifierId
    target: receiverId
}

Receiver {
    id : receiverId
    anchors.right: parent.right
}
```

Below is the full code for Sender.qml, Receiver.qml and main.qml. This should help make it clear how the code we've seen so far fits into the big picture.

```
//Sender.qml
import QtQuick
Item {
    width: notifierRectId.width
    height: notifierRectId.height
    property int count: 0
    signal notify( string count)//Declare signal

    property Receiver target : null

    onTargetChanged: {
        notify.connect(target.receiveInfo)
    }

    Rectangle {
        id : notifierRectId
        width: 200
        height: 200
        color: "red"

        Text {
            id : displayTextId
            anchors.centerIn: parent
            font.pointSize: 20
            text : count
        }

        MouseArea{
            anchors.fill: parent
            onClicked: {
                count++
                notify(count)
            }
        }
    }
}
```

```
//Receiver.qml
import QtQuick

Item {
    width: receiverRectId.width
    height: receiverRectId.height

    function receiveInfo( count){
        receiverDisplayTextId.text = count
        console.log("Receiver received number : "+ count)
    }

    Rectangle {
        id : receiverRectId
        width: 200
        height: 200
        color: "blue"

        Text {
            id : receiverDisplayTextId
            anchors.centerIn: parent
            font.pointSize: 20
            text : "0"
        }
    }
}
```

```
//main.qml
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("External Components with signals and slots")

    Notifier{
        id : notifierId
        target: receiverId
    }

    Receiver {
        id : receiverId
        anchors.right: parent.right
    }
}
```

If you run the application with these changes applied and click on the red sender element, you should see the vales in the receiver element updating in sync.

Exposing properties: Two techniques

Our signal slot connection is working across external QML components. This is something you'll use all over the place in your QML projects. As a bonus, lets set up our components in such a way that we allow users to set colors to these components externally. Something like below

```
Notifier{
    id : notifierId
    rectColor: "yellowgreen"
    target: receiverId
}

Receiver {
    id : receiverId
    rectColor: "dodgerblue"
    anchors.right: parent.right
}
```

`rectColor` should somehow be exposed from the component and forward data inside the component when set externally. There are two ways to achieve this. One way is to rely on property change handlers, just like we did with the target property earlier in the Sender component. Change your Sender.qml file to put the code below under the target property change handler.

```
property color rectColor: "black"
onRectColorChanged: {
    notifierRectId.color = rectColor
}
```

When someone assigns a value to `rectColor` inside Sender, the property change handler is triggered and the current color is forwarded inside the component. If you change your main.qml to use a Sender object like below

```
Notifier{
    id : notifierId
    rectColor: "yellowgreen"
    target: receiverId
}
```

and run the application, your sender object should change the color accordingly. The second method to forward data from the outside and use that inside is to go through good old property aliases. We'll play with that using our receiver component. Change your Receiver.qml file to add a one-liner above the width property:

```
property alias rectColor: receiverRectId.color
```

You should also change your Receiver object in main.qml :

```
Receiver {
    id : receiverId
    rectColor: "dodgerblue"
    anchors.right: parent.right
}
```

and run the application. You'll see that the receiver rectangle changes color accordingly. These are two techniques you can use to expose properties to the outside in your external qml components. Which one you choose will be your design decision. My job here is to make you aware of the possibilities.

Multiple Signal Parameters

Signals in QML can send out more than one parameter. In practice, you can set up a signal like below

```
signal info(string last_name,string first_name,int age)
```

The signal has three parameters. In this section, we'll explore different ways we can set up a handler for this signal. First, the most intuitive way is to process all three parameters. You can then set up a handler like below

```
//Handle all three parameters
onInfo: function(l,f,a){
    print("last name : " +l + ", first name : " + f +", age : "+ a)
}
```

One could also only process the first two parameters and leave out the trailing one

```
//Only handle the first two
onInfo : function(l,f){
    print("last name : " +l + ", first name : " + f)
}
```

Following the same logic, one could also handle only the first one

```
//Only handle the first one
onInfo : function(l){
    print("last name : " +l)
}
```

From this, some readers may have figured out that we can only ignore trailing parameters, or, in other words, those coming last in the parameter list. For example, you can't ignore the last name and hope for a handler like below to process the first name and the age

```
//Can only omit training parameters. Can't only ignore the last name
onInfo: function(f,a){
    print(" first name : " + f +", age : "+ a)
}
```

If you run code with the handler above, QML will plug the last name in `f` and the first name in `a`. Probably not what you want! One practice I've seen people use to document that a non-trailing parameter is ignored is to use an underscore in place of that parameter. Something like below should indicate that the last name parameter is ignored in the body of our handler.

```
// Non-trailing parameters can be ignored through some hack
onInfo: function(_,f,a){
    print(" first name : " + f +", age : "+ a)
}
```

The underscore acts as a placeholder and there is no problem in how your parameters are processed. To play with this, I encourage you to create a brand new Qt Quick project from Qt Creator and change your main.qml file to contain the code below. The code contains several variations for the `onInfo` handler that we've talked about. At any moment in time, only one will be activated and the others will be commented out.

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Multiple Signal Parameters")

    signal info(string last_name,string first_name,int age)

    //Handle all three parameters
    onInfo: function(l,f,a){
        print("last name : " +l + ", first name : " + f +", age : "+ a)
    }

    //Only handle the first two
    /*
    onInfo : function(l,f){
        print("last name : " +l + ", first name : " + f)
    }
    */
    //Only handle the first one
    /*
    onInfo : function(l){
        print("last name : " +l)
    }
    */
    //Can only omit training parameters. Can't only ignore the last name
    /*
    onInfo: function(f,a){
        print(" first name : " + f +", age : "+ a)
    }
    */
    //Nontrailing parameters can be ignored through some hack
    /*
    onInfo: function(_,f,a){
        print(" first name : " + f +", age : "+ a)
    }
    */
    Rectangle{
        id : rectId
        width : 300;
        height : 300
        color : "blue"

        MouseArea{
            anchors.fill: parent
            onClicked: {
                info("John","Snow",33)
            }
        }
    }
}
```

The code is very familiar. It's just a rectangle that contains a MouseArea object filling it. In the click handler, we fire the `info` signal, which should trigger a call to the currently activated `onInfo` handler.

Chapter Summary

The signals and slots mechanism allows us to flexibly transfer data among objects in our QML application. The chapter started out by exploring signal handlers and the naming convention they use in QML files. Your handlers should be named `on<SignalName>` with the first letter of the signal name capitalized. Next, we saw that some signals use parameters to send out additional information. We looked that the parameters sent out by the `clicked` signal of the MouseArea element. We re-examined property change handlers and learned about the Connections element. It allows us to handle signals in places different from the one the signal originates from. Attached signal handlers, just like attached properties, add capabilities we wouldn't otherwise have to our QML code. We looked at how we can set up our own custom signals and ways one can connect a signal to a method or another signal. We saw that we can use the signals and handlers mechanism to transfer data even across custom QML components. We wrapped up by sending out multiple parameters with our signal.

Chapter 5: User Input

Your application will need to use data it gets from users one way or the other. QML provides a host of facilities one can use to collect data from the user; once the data is collected, you can use it to do whatever you want. In this chapter, we'll look at the TextInput element that one can use to collect a single line of text from the user. TextEdit will be used to collect multiple lines of text. MouseArea is the only user input element we've been using so far in the book. Yes! Clicks and double clicks can also be considered as input to your application. We'll see some other things we can do with it in the chapter. We'll look at the Keys attached property, a facility one can use to process key presses on your keyboard. KeyNavigation can be used to control what happens when the user presses the direction arrows on your keyboard. We'll wrap up the chapter with the FocusScope element and see how it allows you to control focus transfer among QML custom components.

TextInput

The `TextInput` element allows users to input text in a QML application, and it comes with various features and properties that can be customized to suit different use cases. In this section, we will explore some of its properties, signals and methods. Along the way, we'll build a simple form to collect first-name and last-name information from the user. To start with, I would recommend taking a peek at the docs page for the TextInput element. The end result from this section will be something like below

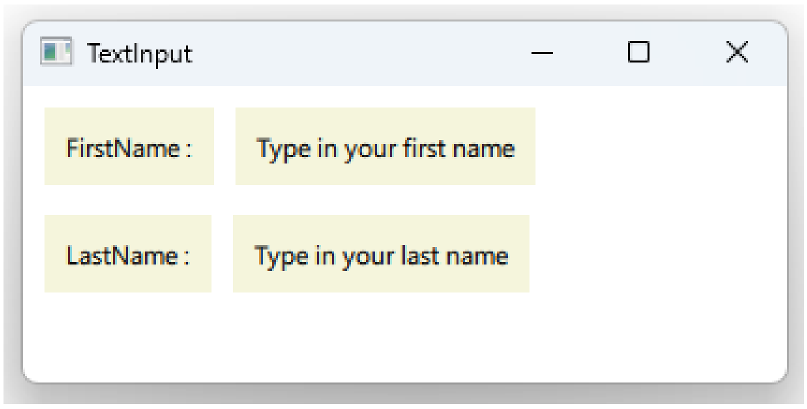


Figure 71. TextInput Form

On the left, we have a few labels and on the right we have TextEdit elements wrapped in a Rectangle with a beige background. TextInput provides the ability to edit text and it's built to handle one-liner pieces of text. It's not built to handle several lines of text. Let's play with this. Start a brand new Qt Quick project from Qt Creator and change your main.qml file to contain code like below

```
//main.qml
import QtQuick
Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("TextInput")
    Row {
        x : 10; y : 10
        spacing: 10

        Rectangle {
            id : firstNameRectId
            width : firstNameLabelId.implicitWidth + 20
            height: firstNameLabelId.implicitHeight + 20
            color : "beige"

            Text {
                id : firstNameLabelId
                anchors.centerIn: parent
                text : "FirstName :"
            }
        }

        Rectangle {
            id : firstNameTextRectId
            color : "beige"
            width: firstNameTextId.implicitWidth  + 20
            height: firstNameTextId.implicitHeight + 20

            TextInput {
                id : firstNameTextId
                anchors.centerIn: parent
                focus: true
                text : "Type in your first name"
                onEditingFinished: {
                    console.log("The first name changed to :"+ text)
                }
            }
        }
    }
}
```

The `TextInput` element is used inside a `Rectangle` element to capture the user's first name. The width and height of the TextInput element are determined by the `implicitWidth` and `implicitHeight` properties of its child Text element, plus an additional 20 units of padding on each side. The TextInput element has an id of "firstNameTextId", which can be used to refer to it in other parts of the code.

The `focus` property of the TextInput element is set to true, which means that it will automatically receive focus when the UI is loaded. This allows the user to start typing immediately without having to click on the TextInput element first. The initial text displayed in the TextInput element is set to "Type in your first name" using the text property.

The editingFinished signal of the TextInput element is connected to a JavaScript function that logs a message to the console when the user finishes editing the text. The message includes the updated text entered by the user, which can be accessed using the text property of the TextInput element.

The two rectangles are positioned horizontally using a Row positioner. If you build and run the application, you should see a window like below

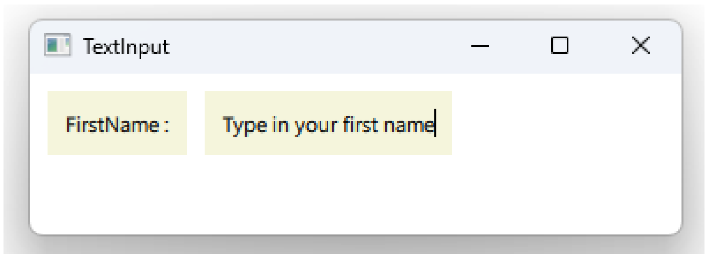


Figure 72. TextInput Form for First Name

You should have a blinking cursor in the TextInput element, prompting you to type things out. You could delete the data that is in by default or type in your data. If I delete everything and type in "Marlow" as the first name and hit Enter, I should see output like below in the Application Output pane of Qt Creator.

```
qml: The first name changed to: Marlow
```

This implies that the onEditingFinished handler is triggered when the user hits the Enter key after tying in some text. The exhaustive list of signals from TextInput are available from the Qt docs. Nothing prevents you from adding a second row below the first one, to handle the last name

```
//...
    Row {
        x : 10; y : 60
        spacing: 10

        Rectangle {
            id : lastNameRectId
            width : lastNameLabelId.implicitWidth + 20
            height: lastNameLabelId.implicitHeight + 20
            color : "beige"

            Text {
                id : lastNameLabelId
                anchors.centerIn: parent
                text : "LastName :"
            }
        }

        Rectangle {
            id : lastNameTextRectId
            color : "beige"
            width: lastNameTextId.implicitWidth  + 20
            height: lastNameTextId.implicitHeight + 20

            TextInput {
                id : lastNameTextId
                anchors.centerIn: parent
                focus: true
                text : "Type in your last name"
                onEditingFinished: {
                    console.log("The last name changed to :"+ text)
                }
            }
        }
    }
//...
```

If you run the application with the changes applied, you should see something like below

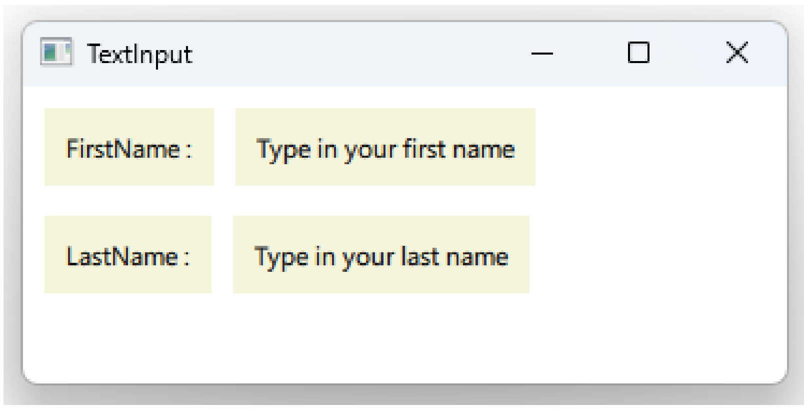


Figure 73. TextInput Form

It's encouraged to play with the application, and even try to add other rows for data like age and occupation to really make this your own. It's important to remember that the TextInput element's main purpose is to display *a single line of editable plain text*. You can't use it to display multiple lines of text. We'll get to see other elements you can use to do that as we progress in the book.

TextEdit

The TextEdit element can be used to display multiple lines of text and allow the user to edit the text. It's much like TextInput we have seen earlier but with the ability to handle multiple lines. Like TextInput, it has an editingFinished signal that is fired when the user hits Enter after editing the data. Unlike TextInput however, it can handle text in different formats like raw text or rich text. Take look at this element in the docs and start a brand new Qt Quick project to play with this. Change your main.qml code to be like below

```
//main.qml
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("TextEdit Demo")

    TextEdit {
        id : textInputId
        width: 240
        text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis
aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat
nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum."
        font.family: "Helvetica"
        font.pointSize: 20
        color: "blue"
        focus: true
    }

    Rectangle {
        id : mRectId
        width: 240
        height: 100
        color: "red"
        anchors.top: textInputId.bottom
        MouseArea{
            anchors.fill: parent
            onClicked: {
              console.log("The new text is :"+textInputId.text)
            }
        }
    }
}
```

The window contains a TextEdit element with a red Rectangle anchored to its bottom. Here, the rectangle is there to help us process click events and read data from the TextEdit element. If you run the application, you'll see something like below

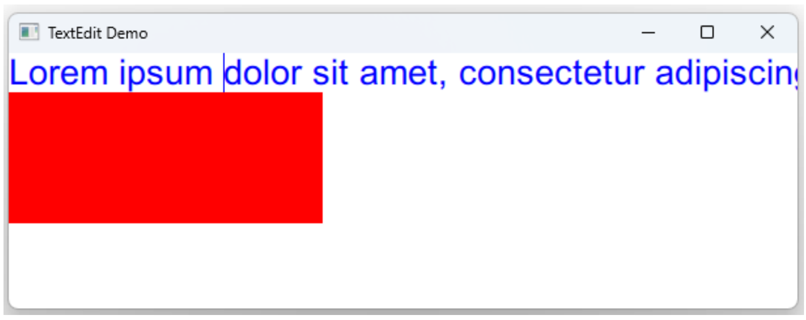


Figure 74. Raw TextEdit Element

We have a single line of blue text and our red rectangle on the bottom. If you click on the rectangle you should see output like below

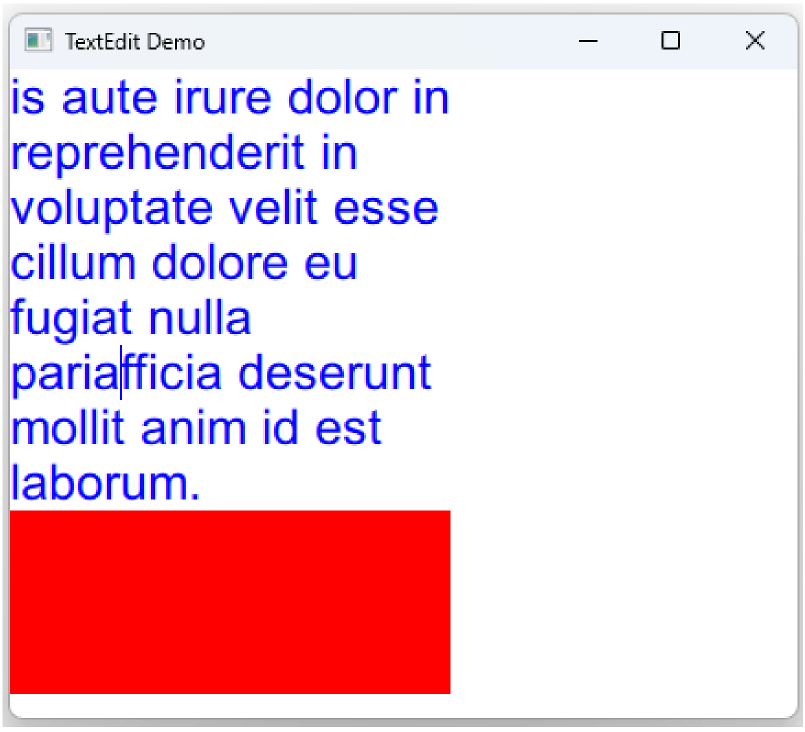
```
qml: The new text is :Lorem ipsum dolor sit amet,...
```

This is what we consoloe.log() from our click handler. Here is something I want you to pay attention to though. If you hover over the blue text, within the width of the rectangle, or 240 px, you'll see that we have a cursor icon hinting us to click and update the text. If you go outside the width of the rectangle, however, you'll see that our cursor will disappear and we have no way to manipulate text in our TextEdit element. This is because we set up our TextEdit element to be exactly 240 px wide and that's the range where it'll be effective.

The text in our TextEdit is obviously wider than it can handle. The default in this case is to smash all the text on a single line and hope for the best. We are, however, dealing with a TextEdit element that can handle multiple lines of text, so there should be a way to force the text to wrap over to the next line when we run out of horizontal space to deal with the text. We have a `wrapMode` property we can use to do exactly that. If you add this line

```
wrapMode: TextEdit.Wrap
```

below the id property of our TextEdit and run the application again, you'll see that the text will wrap at the width of the TextEdit and move the remainder of the text to the next line and keep doing that until all the text is exhausted. The result will be that our red rectangle, which is anchored to the bottom of the TextEdit, will be pushed out of the visible area of the window. Not to worry though, you can delete the text until the rectangle is back in the visible area of the window. Once the rectangle is visible, you can click on it to see the current text displayed in the Application Output pane.



is aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariafficia deserunt mollit anim id est laborum.

Figure 75. Wrapping TextEdit After Deleting Lots of Text

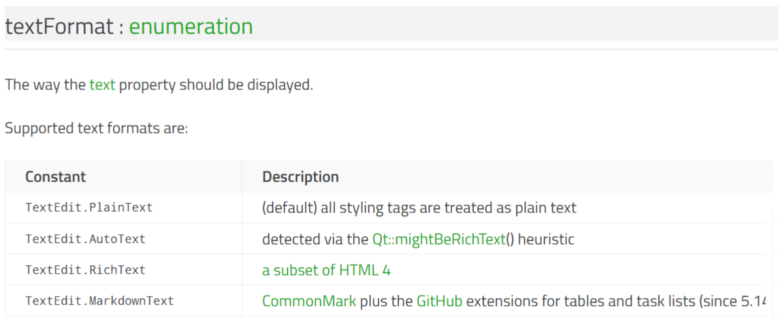
The detailed list of possible values for the `wrapMode` property is available in the documentation. TextEdit also has an editingFinished signal that is fired when the TextEdit loses focus after editing. I personally was confused by this because I somehow thought the signal is fired when one presses the "Enter" key after editing. Change your TextEdit to be like below

```
    TextEdit {
        id : textInputId
        wrapMode: TextEdit.Wrap
        width: 240
        text: "Lorem ipsum..."
        font.family: "Helvetica"
        font.pointSize: 20
        color: "blue"
        focus: true

        onEditingFinished: {
            console.log("The current text is :"+ text)
        }
    }
```

The value of the text property is chopped off here for space reasons in the book, but you should keep that as is in your code. Run the application and play with it, trying to get the code in the body of the onEditingFinished handler to run. You'll see that clicking in the TextEdit element and hitting "Enter" doesn't cause the text to print out. The text prints out when you leave your QML application window and click on another application, for example, an opened instance of Qt Creator. If you look in the Application Output pane, you should see a message like below

```
qml: The current text is :Lorem ipsum...
```

Clicking on the red rectangle should also show the current text in the TextEdit as per the login in our click handler. TextEdit has another property named textFormat that you use to control the formatting of the text. By default it uses the plain text format, but looking at the documentation for that property you can see that there are several possible values for this property.



textFormat : enumeration

The way the text property should be displayed.

Supported text formats are:

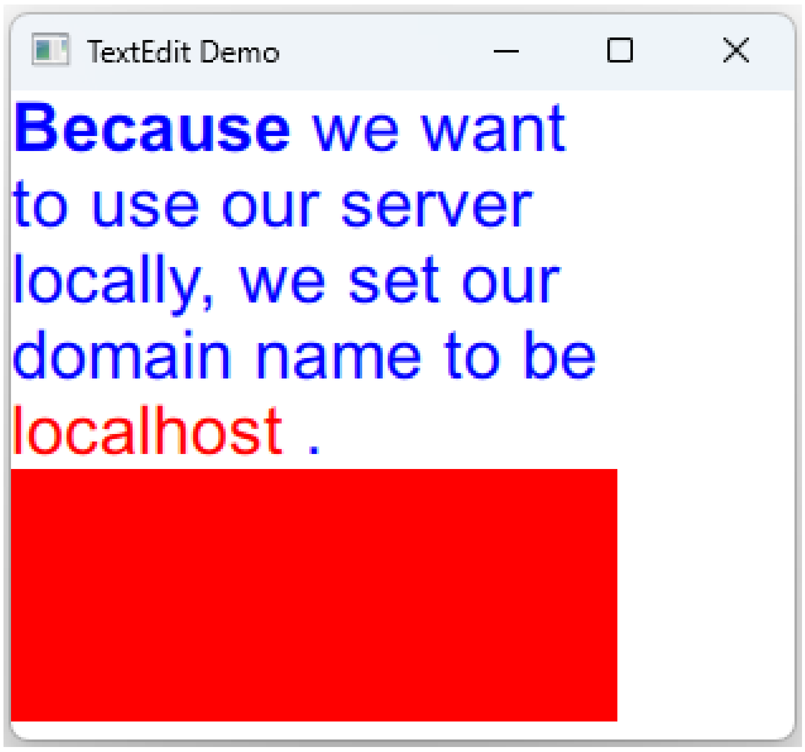
Constant	Description
`TextEdit.PlainText`	(default) all styling tags are treated as plain text
`TextEdit.AutoText`	detected via the Qt::mightBeRichText() heuristic
`TextEdit.RichText`	a subset of HTML 4
`TextEdit.MarkdownText`	CommonMark plus the GitHub extensions for tables and task lists (since 5.1

Figure 76. textFormat Property Possible Values

`TextEdit.RichText` is my favorite as it allows you to format your text using a subset of HTML that is documented here. Change your TextEdit to be like below.

```
    TextEdit {
        id : textInputId
        wrapMode: TextEdit.Wrap
        textFormat: TextEdit.RichText
        width: 240
        text: "<strong>Because</strong> we want to use our server locally, we set our
domain name \r to be <font color = 'red' >localhost </font>."
        font.family: "Helvetica"
        font.pointSize: 20
        color: "blue"
        focus: true

        onEditingFinished: {
            console.log("The current text is :"+ text)
        }
    }
```

First, we explicitly set the textFormat property to use RichText. We then changed the value of our text property to use some HTML tags like `strong` and `font`, and our TextEdit element will interpret them and visualize the text accordingly. If you run the application, you should "Because " displayed in bold and the "localhost" text displayed in red.



Because we want to use our server locally, we set our domain name to be localhost .

Figure 77. Displaying RichText

These are some of the things you can use a TextEdit element and it'll come in handy on many occasions in your QML applications. One shouldn't lose sight of one of its limitations though. It can't allow you to scroll through the text if the text happens to be higher than the window can display. We'll see different ways to handle this later on in the book.

MouseArea

MouseArea is also an element used to provide some kind of input to your QML application. We have largely been using it to process click events, and that's what it's mostly used for in practice, but you can use it to do much more. Apart from the clicked signal, it also provides a `wheel` signal that provides information about the wheel button on the mouse being turned, a `hoverChanged` signal, that is fired when you hover-enter the area filled by the mouse area and when you get out of the area. You can judge whether we are getting in or out by reading the `containsMouse` property. We won't only play with these signals, but we'll also take this chance to show you that you can drag a target element around when you click and move on the area filled by a MouseArea. Before we play with this, just like any good QML citizen, I would advise you to take a peek at the docs page for MouseArea and see some information to spice up your mind before we dive into this. Create a brand new Qt Quick project from Qt Creator and change your main.qml file to contain code like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("MouseArea Demo")

    Rectangle {
        id : containerRectId
        width : parent.width
        height: 200
        color: "beige"

        Rectangle {
            id : movingRectId
            width: 50
            height: width
            color: "blue"
        }

        MouseArea{
            anchors.fill: parent
            onClicked: function(mouse) {
                console.log(mouse.x)
                movingRectId.x =  mouse.x
            }

            onWheel:function(wheel) {
                console.log(" X : "+ wheel.x + " y : "+ wheel.y + " angleData :"+
wheel.angleDelta)
            }

            hoverEnabled: true
            onHoveredChanged: {
                if (containsMouse)
                {
                    containerRectId.color = "red"
                }else {
                    containerRectId.color = "green"
                }
            }
        }

    }
}
```

The window element contains a container rectangle whose ID is `containerRectId`. containerRectId contains a smaller blue rectangle that's supposed to move along the X axis as we click in the MouseArea filling `containerRectId`. Notice that the `onClicked` handler also processes a parameter named mouse. If you don't know about this parameter, I would strongly advise you to check it out in the docs right now if you can. But I'll tell you what it does in case it's not convenient for you. The docs page says that *The mouse parameter provides information about the click, including the x and y position of the release of the click, and whether the click was held.* Its type is MouseEvent, which contains much more information about the click. In this case, we're interested in the x position inside containerRectId where the user clicked. We're using that information to set the x position of the smaller blue rectangle in the click handler.

The `onWheel` handler also provides a wheel parameter that contains more information about the turning of the wheel button. It also contains the x and y position of where in the area filled by the MouseArea the wheel button was turned. We can also use the `angleDelta` property on the wheel parameter to decide whether the wheel was turned up or down. The last thing we look at in this code example is the `onHoverChanged` handler. We set the hoverEnabled property to true to be able to track down hover events even when the mouse is not pushed down.

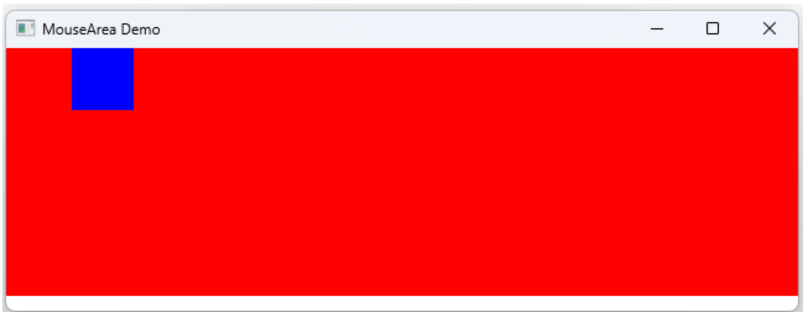


Figure 78. MouseArea Click Wheel and Hover Events

If you compile and run the application, you'll see a containerRectId with a beige color by default and a smaller blue rectangle inside. If you move the mouse on top of the container rectangle, you'll see it turn red because that's the logic we programmed it to do when it contains the mouse and the hover state changes. If you move the mouse out, it'll turn green.

You should also play with the wheel button and see what information gets printed in the Application Output pane in Qt Creator. Lastly, click in the container rectangle, and see the blue rectangle following you around!

You can also use MouseArea to drag target elements around. You can drag targets either on the x-axis or the y-axis. The docs page says that *drag.axis specifies whether dragging can be done horizontally (Drag.XAxis), vertically (Drag.YAxis), or both (Drag.XAndYAxis).* We also have a bunch of other grouped properties we'll learn about through a code example. Modify your main.qml file to add dragContaierId just under containerRectId with absolute positioning just as shown below

```
//main.qml
//...
Rectangle{
    id : containerRectId
    //...
}
Rectangle {
    id : dragContaierId
    width : parent.width
    height: 200
    color: "beige"
    y : 250

    Rectangle {
        id : draggableRect
        width: 50
        height: width
        color: "blue"

        onXChanged: {
            console.log("X Coordinate is : "+ x)
        }
    }

    MouseArea{
        anchors.fill : parent
        drag.target: draggableRect
        drag.axis: Drag.XAxis
        drag.minimumX: 0
        drag.maximumX: dragContaierId.width - draggableRect.width

    }
}
//...
```

We have a smaller blue rectangle that'll act as our draggable element and we've named the ID very conveniently. The MouseArea fills `dragContaierId` and is our point of interest here. We set draggableRect as our drag target, in other words, when we click and drag on the mouse area, it's draggableRect that'll be moving. We specify the drag axis to be the X axis and we want to be able to drag up to the end of dragContaierId in width. You can learn more about the drag grouped property and see some other interesting examples from the documentation. For now let's run the application and see this in action.

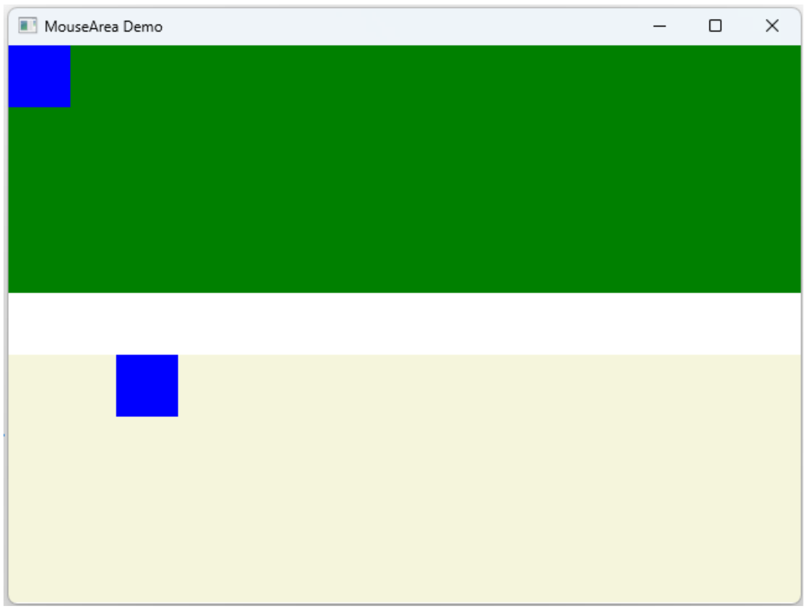


Figure 79. Drag Grouped Property

If you click, hold the mouse and move horizontally, you should see the blue rectangle following you in the beige rectangle on the bottom. These are some of the interesting things you can use the MouseArea element for in terms of user input. I hope you find some good use for these in some of your QML projects.

Keys Attached Property

The Keys attached property in QML provide a powerful way for key handling in all visual primitives. With support for `pressed` and `released` signals, developers can easily define keyboard shortcuts. The KeyEvent parameter provides details of the key event for convenient access. To prevent event propagation to parent items, setting `event.accepted` to true can be used. The Keys Attached property makes it easy to implement keyboard interactions and user interactions with proper event handling and propagation control. As always, it's recommended to read the docs page yourself to what useful information you may pick up. In this section the focus is giving you the ability to respond to keys being pressed on your keyboard and processing key modifiers like Control, Command or Shift.

Create a brand new Qt Quick project from Qt Creator and change your main.qml file to contain starter code like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Keys Attached Property")

    Rectangle {
        id : containedRect
        anchors.centerIn: parent
        width : 300
        height: 50
        color: "dodgerblue"
        focus: true // The rectangle needs focus for key events to fire properly

        Keys.onDigit5Pressed:function(event) {
            console.log("Specific Signal : Pressed on Key 5")
            event.accepted = true
        }
    }
}
```

The docs page says that *all visual primitives support key handling via the Keys attached property.* Here, we are using the Keys attached property to supercharge our Rectangle with Key handling capabilities. We are attaching those capabilities to it if you will. This is what attached properties do at the code if you remember. They bring capabilities within reach for elements and elements wouldn't have had those capabilities otherwise.

`Keys` has a `digit5Pressed` signal we can use the check whether the key 5 is pressed on our keyboard. The signal also sends out an event parameter with more info on what happened that caused the signal to be fired. If you are sure that you did all you needed to do in the handler and expect no one else to do more somewhere else in the app, you should set `event.accepted` to `true` in your handler. This signals to the QML system that the signal has properly been handled. If the event wasn't handled properly, may be some error occurred somewhere, you should set `event.accepted` to `false`. That gives other parts of your application to take give their shot at handling the event, maybe adding value to your application somehow. Of course, these are design decisions you'll have to make based on the requirements of your QML application.

You should also notice that we set `focus` to `true` in our rectangle. This is necessary if we want the rectangle to handle key events. If you run the application, you'll see our dodgerblue rectangle. Make sure your application has focus by clicking somewhere in the window. Press key 5 on your keyboard and you should see the expected message printed out in the Application Output pane in Qt Creator. `digit5Pressed` in just one of the signals, you can find the exhaustive list from the docs page.

With this out of the way, it shouldn't be hard to modify our handler to also process key modifiers like Ctrl if we wanted. Modify the onDigit5Pressed handler like below

```
Keys.onDigit5Pressed:function(event) {
    if ( event.modifiers === Qt.ControlModifier)
    {
        console.log("Pressed Control + 5")
    }else{
        console.log("Pressed regular 5")
    }
```

This is like a pair of nested levels of checks. We first check if key 5 is pressed. Once we're sure that it was pressed we proceed to check if it was pressed alone or if it was pressed together with the Control Modifier. If you run the application with the changes applied and press key 5 alone, you should see the message

```
Pressed regular 5
```

in your Application Output pane in Qt Creator. Press Ctrl+5 and you should see the message

```
Pressed Control + 5
```

in your Application Output pane. We also have a `pressed` signal whose handler we can use to process **any key press** on the keyboard. The trick is to ask the event parameter which key was pressed. We can also ask the event parameter if it detected any key modifiers. This is best explained with a code example. Comment out the previous `Keys.onDigit5Pressed `handler we had in place earlier, and put in place the following `onPressed` handler.

```
Keys.onPressed: function(event){
    if ( event.key ===Qt.Key_5)
    {
            console.log("General Signal: Pressed on Key 5")
    }
    if ((event.key === Qt.Key_5) && (event.modifiers & Qt.ControlModifier)){
        console.log("General Signal: Pressed Control + 5")
    }
}
```

The code does the usual, checking if key 5 was pressed alone or if it was coupled with the Control modifier. The example shows Qt.Key_5 but the exhaustive list of keys can be found at the official documentation. The expression `event.modifiers & Qt.ControlModifier` is a bitwise AND operation between the `event.modifiers` and `Qt.ControlModifier` values. The result of this operation is a bitwise combination of the modifier keys that are currently being pressed along with the Control key. This expression is used to check if the Control key is being pressed in combination with other modifiers during the occurrence of the event. If the result is non-zero, it means the Control key is currently being held down along with other modifier keys, and this condition can be used to trigger a specific action or behavior in your QML code. Keep this in mind as this trick often comes in handy. If you compile and run the application, and press key 5, you'll see the message

```
General Signal: Pressed on Key 5
```

and if you press Ctrl+5 you'll see both messages printed out

```
General Signal: Pressed on Key 5
General Signal: Pressed Control + 5
```

There are ways to make sure one of these branches is activated, for example by changing your handler to be like below

```
Keys.onPressed: function(event){
    if ( (event.key ===Qt.Key_5)&& !(event.modifiers & Qt.ControlModifier))
    {
            console.log("General Signal: Pressed on Key 5")
    }
    if ((event.key === Qt.Key_5) && (event.modifiers & Qt.ControlModifier)){
        console.log("General Signal: Pressed Control + 5")
    }
}
```

Event Forwarding

We have seen two techniques one can use to handle key events using the Keys attached property. But what if you have both handlers activated in your code at the same time? Something like below

```
Keys.onDigit5Pressed:function(event) {
    if ( event.modifiers === Qt.ControlModifier)
    {
        console.log("Pressed Control + 5")
    }else{
        console.log("Pressed regular 5")
    }
}

Keys.onPressed: function(event){
    if ( (event.key ===Qt.Key_5)&& !(event.modifiers & Qt.ControlModifier))
    {
            console.log("General Signal: Pressed on Key 5")
    }
    if ((event.key === Qt.Key_5) && (event.modifiers & Qt.ControlModifier)){
        console.log("General Signal: Pressed Control + 5")
    }
}
```

Which one will be triggered if we press key 5 here? I'll leave you to ponder on this for a minute, but nothing prevents you from running the application with these changes applied to see what happens. If run the app, you'll see the message

```
qml: Pressed regular 5
```

printed out, meaning that the specific handler `Keys.onDigit5Pressed` takes priority over the general handler `Keys.onPressed` if both happen to be activated at the same time. These priorities are described in the official documentation and I would advise you to at least take a moment to have an idea about them. Sometimes these little details can mean the difference between failure and success when faced with a problem at hand. We could instruct `Keys.onDigit5Pressed` not to handle the event alone and give `Keys.onPressed` a chance to also process the event. This is called event propagation, and Keys.onDigit5Pressed would literally propagate the event to Keys.onPressed. Events are propagated from the most specific handlers to the least specific handlers. If we add event.accepted = false as the last statement in the body of the Keys.onDigit5Pressed handler,

```
Keys.onDigit5Pressed:function(event) {
    if ( event.modifiers === Qt.ControlModifier)
    {
        console.log("Pressed Control + 5")
    }else{
        console.log("Pressed regular 5")
    }
    //Give other least specific handlers
    // a chance to also handle the event
    event.accepted = false

}

Keys.onPressed: function(event){
    if ( (event.key ===Qt.Key_5)&& !(event.modifiers & Qt.ControlModifier))
    {
            console.log("General Signal: Pressed on Key 5")
    }
    if ((event.key === Qt.Key_5) && (event.modifiers & Qt.ControlModifier)){
        console.log("General Signal: Pressed Control + 5")
    }
}
```

and run the application, we should see the messages

```
qml: Pressed regular 5
qml: General Signal: Pressed on Key 5
```

meaning that after the specific handler printed its message, Qt also gave a chance to the available least specific handler. It is up to you to decide whether you want the event propagated or not, but it is always a good idea to be aware of these possibilities. It s my hope that this broadened your horizon as to how to handle key events in your application through the Keys attached property. Don’t hesitate to use the official documentation to even learn more, especially the **Key Handling Priorities** section.

Key Navigation

KeyNavigation is a QML attached property that provides a simple way to navigate between focusable items using keyboard keys. It allows users to navigate through focusable items, such as buttons or text fields, using the Tab and Enter keys. KeyNavigation makes it easy to define the order in which items should receive focus and enables keyboard-based navigation in Qt Quick applications. It simplifies the implementation of keyboard navigation and enhances the usability of the user interface for keyboard users. Before we dive into this, please check the docs page yourself to have your own take on this attached property.

To play with this, we'll build an application made up of two rectangles. The currently focused rectangle will be red and the rectangle without focus will be grayed out. We will be able to move focus from one rectangle to another by using the left arrow or the right arrow buttons on your keyboard. The app will look something like below

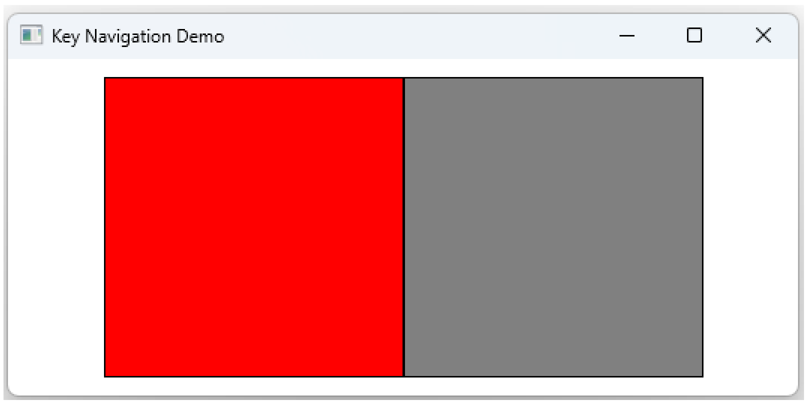


Figure 80. Key Navigation with Two Rectangles

In the example image above, the first rectangle is red, and in our design, that means that it has focus and can handle key events for example, through the Keys attached property. Let's play with this. Create a brand new Qt Quick project from Qt Creator and change your main.qml file to be like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Key Navigation Demo")

    Row {
        anchors.centerIn: parent

        Rectangle{
            id : firstRectId
            width: 200
            height: width
            border.color: "black"
            color: "red"
            focus: true

            onFocusChanged: {
                color = focus?"red":"gray"
            }

            Keys.onDigit5Pressed: {
                console.log("I am Rect1")
            }
            KeyNavigation.right: secondRectId
        }

        Rectangle{
            id : secondRectId
            width: 200
            height: width
            border.color: "black"
            color: "gray"
            onFocusChanged: {
                color = focus?"red":"gray"
            }

            Keys.onDigit5Pressed: {
                console.log("I am Rect2")
            }
            KeyNavigation.left: firstRectId
        }
    }
}
```

We have two rectangles laid out horizontally in a Row positioner. Each rectangle is set up to handle the Keys.digit5Pressed attached signal and print out which rectangle it is. This will help us figure out which rectangle currently holds focus. Notice that `firstRectId` holds focus by default when the application starts up. Now comes the important part, firstRectId is set up to pass focus to secondRectId when the right arrow button is pressed through the statement `KeyNavigation.right: secondRectId` and secondRectId is wired to pass focus to firstRecdtId when the left arrow button is pressed on the keyboard through the statement `KeyNavigation.left: firstRectId`. The rectangle currently holding focus will handle key events and tell us which one it is when key 5 is pressed. We have handled left and right in this example, but nothing prevents you from also handling other keys like tab, up, down and many others. The exhaustive list of possibilities can always be found in the official documentation. Compile and run the application and you should see something like below.

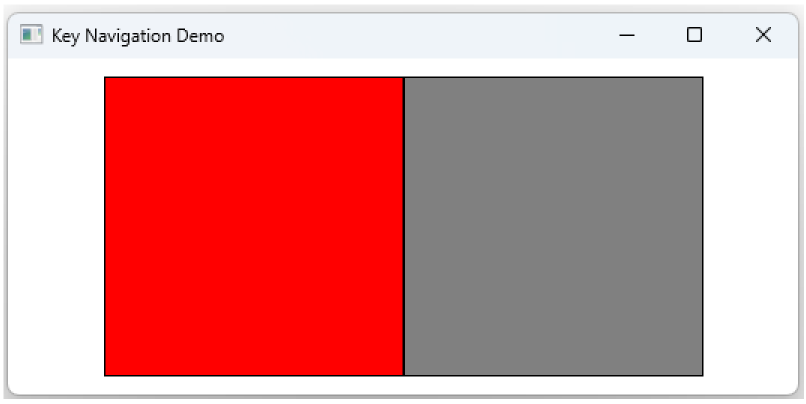


Figure 81. Key Navigation with Two Rectangles

which is what we set out to do in the first place. Click the right arrow button and the red color should move to the right, meaning that secondRectId now holds focus, and if you hit key 5, it'll be secondRectId that'll respond by printing "I am Rect2" in the Application Output pane from Qt

Creator. Click the left arrow button to pass focus back to firstRectId and hit key 5 to see it introduce itself! This is the KeyNavigation attached property in action, its main purpose is to specify which element will get focus at the press of a specific key on the keyboard.

To have some more fun with this and to show you that you can steal useful code snippets from the documentation, lets comment out the Row element in our main.qml file and replace it with the Grid element along with its contents as seen in the KeyNavigation docs page. Our code should look something like below after the change is applied.

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Key Navigation Demo")

    Grid {
          anchors.centerIn: parent
          columns: 2

          Rectangle {
              id: topLeft
              width: 100; height: 100
              color: focus ? "red" : "lightgray"
              focus: true
              KeyNavigation.right: topRight
              KeyNavigation.down: bottomLeft
          }
          Rectangle {
              id: topRight
              width: 100; height: 100
              color: focus ? "red" : "lightgray"
              KeyNavigation.left: topLeft
              KeyNavigation.down: bottomRight
          }
          Rectangle {
              id: bottomLeft
              width: 100; height: 100
              color: focus ? "red" : "lightgray"
              KeyNavigation.right: bottomRight
              KeyNavigation.up: topLeft
          }
          Rectangle {
              id: bottomRight
              width: 100; height: 100
              color: focus ? "red" : "lightgray"
              KeyNavigation.left: bottomLeft
              KeyNavigation.up: topRight
          }
    }
}
```

Run the application and you should see something like below

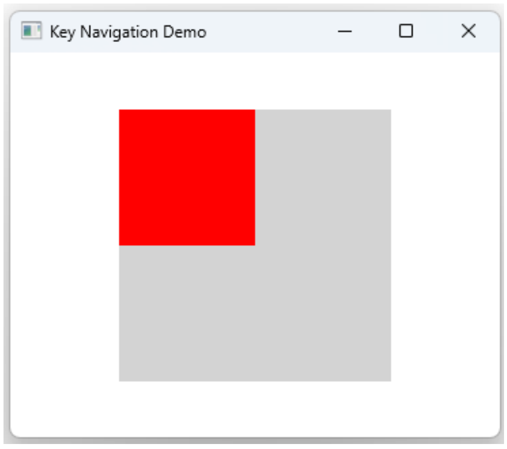


Figure 82. Key Navigation with Four Rectangles

Now you can navigate using the left, right, top and down arrow keys. More explanations on this can be found on the docs page, but I am sure you can figure this out on your own, pulling insights from the first example we did.

FocusScope

FocusScope is a nonvisual element that comes in handy when focus is involved while using external custom QML components. We have seen that for an element to handle key events, it needs to have active focus. This is best explained with an example. Create a brand new Qt Quick project and change your main.qml file to contain code like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("FocusScope Demo")

    Rectangle {
        id : containerRectId
        width: 300
        height: 50
        color: "beige"
        focus: true

        Text {
            id : textId
            anchors.centerIn: parent
            text: "Default"
        }
        Keys.onPressed:function(event) {
            if(event.key === Qt.Key_1)
            {
                textId.text = "Pressed on Key1"
            }else  if(event.key === Qt.Key_2)
            {
                textId.text = "Pressed on Key2"
            }else{
                textId.text = "Pressed on another key : "+ event.key
            }
        }
    }
}
```

The Window contains a Rectangle item that has active focus. **This means the Rectangle has key handling rights**. The Rectangle itself contains a Text element, which will display which key was pressed and an attached `Keys.onPressed` handler that tells our Text element which key was pressed. If you run the application, it'll behave just as expected.

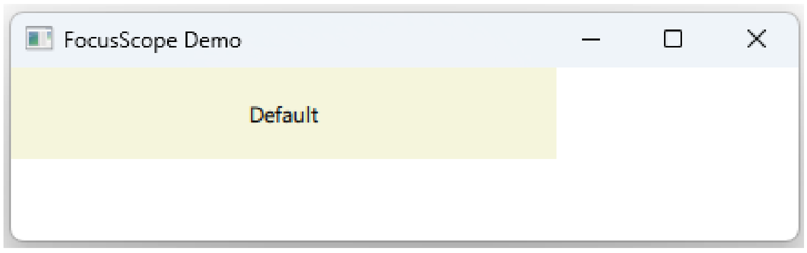


Figure 83. Grabing Focus in Main File

We'll see a beige rectangle saying "Default" by default. If you press key 1 on your keyboard, it'll say "Pressed on Key1", if you press key 2, it'll say "Pressed Key2" and if you press any other key it'll say the code for the key in the Text element. This is all cool and fine. But we like this focus-able rectangle of ours and plan on using it a lot, and come up with a plan to export this code in an external component named MButton. So off we go! We add a new QML file on our project and name it MButton.qml, remember to register the new QML file in the CMakeLists.txt file when you do that, the important part is

```
qt_add_qml_module(app6-FocusScope
    URI 6-FocusScope
    VERSION 1.0
    QML_FILES main.qml MButton.qml
)
```

The MButton.qml file will contain our external component. As good qml citizens, we'll wrap the focus-able rectangle in an item and expose width and height information to the root level to make it play well with positioning mechanisms. Were also making it possible to set the color of the focus-able rectangle from the outside. The code in MButton.qml should look like below

```
//MButton.qml
```

```
import QtQuick
Item {
    width: containerRectId.width
    height: containerRectId.height
    property alias color: containerRectId.color

    Rectangle {
        id : containerRectId
        width: 300
        height: 50
        color: "beige"
        focus: true

        Text {
            id : textId
            anchors.centerIn: parent
            text: "Default"
        }

        Keys.onPressed:function(event) {
            if(event.key === Qt.Key_1)
            {
               textId.text = "Pressed on Key1"
            }else  if(event.key === Qt.Key_2)
            {
               textId.text = "Pressed on Key2"
            }else{
                textId.text = "Pressed on another key : "+ event.key
            }
        }
    }

}
```

It is the same code we had in main.qml just adjusted to work well as an external qml component. With this in place, one could expect to change our main.qml file like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("FocusScope Demo")

    Column {
        MButton{
           color: "yellow"
           focus: true
        }
        MButton{
            color: "green"
        }
    }
}
```

Run the application and have the yellow rectangle handling key events. But to your surprise, you'll see that neither of the rectangles is going to respond when you press key 1 on the keyboard or any key for that matter. It seems like neither of the rectangles has focus. Let's change the code to give focus to the green rectangle and see what happens

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("FocusScope Demo")

    Column {
        MButton{
           color: "yellow"
        }
        MButton{
            color: "green"
            focus: true
        }
    }

}
```

Run the application and you'll see that not only the green rectangle won't have focus, but it will also be the yellow rectangle with focus even if we explicitly told the green one to grab focus!

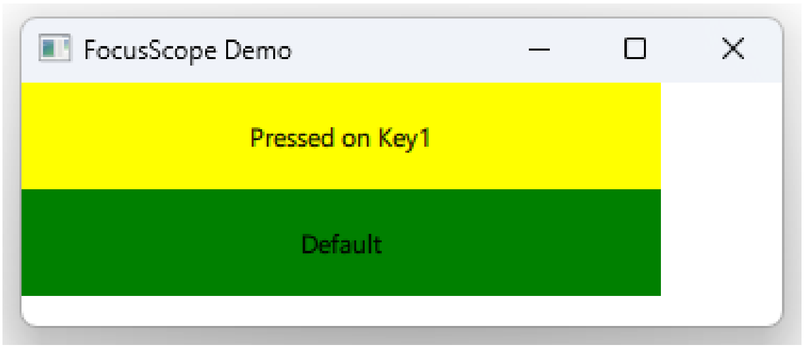


Figure 84. Focus in the wrong rectangle

This seems so unpredictable! But there is a reason for this. We are trying to grab focus from several places in this application: two times from instantiated MButton objects and once in the main.qml file. The QML engine will try its best to figure out who's the legitimate owner of focus. Sometimes it'll go for the wrong owner and sometimes it simply won't give focus to anyone as we've seen. If you want to dive deep into this, you’re welcome to read more on the topic.

Here, we’ll just be content with grabbing the solution and using it. And that’s the FocusScope element. When you wrap your focus-able rectangle in a FocusScope element instead of an Item element like we’ve done earlier, focus will go to the legitimate ower. Let’s try this. Change your MButton.qml to use FocusScope instead of Item like below

```
//MButton.qml
import QtQuick
FocusScope {
    width: containerRectId.width
    height: containerRectId.height
    property alias color: containerRectId.color

    Rectangle {
        id : containerRectId
        width: 300
        height: 50
        color: "beige"
        focus: true

        Text {
            id : textId
            anchors.centerIn: parent
            text: "Default"
        }

        Keys.onPressed:function(event) {
            if(event.key === Qt.Key_1)
            {
               textId.text = "Pressed on Key1"
            }else  if(event.key === Qt.Key_2)
            {
               textId.text = "Pressed on Key2"
            }else{
                textId.text = "Pressed on another key : "+ event.key
            }
        }
    }
}
```

and run the application. You’ll see that focus will go where you want it. If focus: true in the yellow rectangle, it will respond to key presses and say which key was pressed. If focus: true in the green rectangle, it will respond to key presses. There are advanced uses for FocusScope and they are described in the official documentation. Feel free to take a look there when the need arises.

Chapter Summary

User input is a big part of your application, especially when you are designing graphical user interfaces. This chapter explored QML offerings in terms of user input. We started by exploring TextInput, a raw element one can use to collect single lines of text from the user. We then explored TextEdit, a step further in that it can collect multiple lines of text. It has a limitation though: it doesn’t provide an easy way to scroll through the data if the text height happens to be longer than the TextEdit element can display. We’ll see a way around that later on in the book. Please keep in mind that these are raw text processing elements that are meant to be a starting point in building more user-facing, full-featured text collection elements. We saw that MouseArea is also a user input element that collects clicks from the user. The Keys attached property allows you to handle key events. You can even tweak it to process modifiers on top of your keys. KeyNavigation helps you respond when the direction arrow keys on your keyboard are pressed. The FocusScope element helps in making focus transfer work well across instantiated custom QML components.

Chapter 6: Javascript

Javascript is a language with roots in Web Development. Its main purpose is to add interactivity to web pages. QML supports a subset of the Javascript language, and the idea is the same: adding interactivity to QML user interfaces. It allows developers to add dynamic behavior to user interfaces and manipulate QML objects, making it a powerful tool for creating interactive applications. By combining the declarative syntax of QML with the dynamic capabilities of JavaScript, developers can create sophisticated user interfaces that are easy to use and highly responsive.

JavaScript code can be directly embedded into QML files, or it can be placed in separate .js files and imported into QML as a module. This provides developers with flexibility in how they structure their code and allows them to reuse code across multiple QML files. JavaScript can be used to create functions, manipulate QML objects, and connect to signals and slots, which we have already done without being aware that we're using Javascript.

Another benefit of using JavaScript in QML is the ability to interact with backend systems. Many applications require interaction with backend systems, such as databases or web services. JavaScript can be used to connect to these systems and retrieve data, which can then be displayed in the user interface. This allows developers to create applications that are highly functional and can integrate with a wide range of backend systems. We'll get a chance to use some of these facilities later on in the book.

The main purpose of the chapter is to shed some more light on how Javascript is used in QML. We will explore basic usage, look at how to import external Javascript files and work with Javascript modules in our QML files.

Javascript Usage

We have already been using Javascript for a while in our QML applications at this point in the course. The three main points where we've mainly used it are property bindings, signal handlers and custom functions. Lets take a look at this with a concrete example. Create a brand new Qt Quick project from Qt Creator and change the main.qml file to be like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("JS")

    Rectangle {
        id : containerRectId
        width : getHeight()  //JS in Function
        height: 100
        color: x > 300 ? "red" : "green" //property binding
        //JS in signal handler
        onXChanged: {
            console.log("Current value of x : "+ x)
        }

        //Custom function
        function getHeight()
        {
            return  height * 2
        }
    }

    MouseArea {
        anchors.fill: parent
        drag.target: containerRectId
        drag.axis: Drag.XAxis
        drag.minimumX: 0
        drag.maximumX: parent.width - containerRectId.width

    }
}
```

Looking at the code, you might recognize the main thing the application will be doing. It's a draggable green rectangle that'll be living in our Window element. The important thing here is to recognize Javascript code in our QML file here though. The first spot I would like to draw your attention on is the property binding that controls the color of the rectangle

```
color: x > 300 ? "red" : "green" //property binding
```

The color of the rectangle is bound to the expression `x > 300 ? "red" : "green"`. This expression is a ternary operator, making sure the color of the rectangle is red if the position of the rectangle in x is greater than 300, and green otherwise. You will see ternary operators used like this in QML all over the place because they arguably make the code more readable and concise. It is possible to get the same behavior using other Javascript constructs but the code may be living at different spots in your QML file or project and that'll hurt the readability of the code. What I am trying to say here is that ternary operators like this are something you should favor if you can.

The second spot where Javascript is used in our project is the custom function that controls the width of the rectangle

```
//Custom function
function getHeight()
{
    return  height * 2
}
```

If you have some complex logic to achieve some goal in your QML project, don't hesitate to wrap that into a Javascript function that you can call to give you the result. You can then plug the result in some QML property. The obvious benefit is that you get to reuse the function multiple times to do the heavy lifting for you.

The last part worthy of note is our signal handler

```
//JS in signal handler
onXChanged: {
    console.log("Current value of x : "+ x)
}
```

Which is also some Javascript code in action. One should be careful how much Javascript is used in your QML code. Javascript is a scripting language and it can drastically slow down your application if left to run wild. It's a good practice to limit it to things that relate to the interactivity of your QML user interface. Things like responding to buttons, and adding animations and transitions to your user interface. If you need to do something heavy like encrypting a file or processing an image to add some blur effect, doing that in Javascript is an obvious sign of poor design. You could do that in backend languages to QML like C++.

Be careful about the amount of Javascript you use in your QML user interfaces.

The example code in this section isn't teaching something you haven't seen before but is meant to open your eyes to how Javascript works with QML.

Function Scope

JavaScript functions have scope just like in regular JavaScript. The scope of a function defines where the function is accessible. There are two types of function scope in QML:

- Root level scope: Variables and functions defined in the global scope are accessible throughout the QML document. You can define a global function in a separate JavaScript file and import it into your QML document using the import statement.
- Local scope: Variables and functions defined within a QML item or component are only accessible within that item or component.

It's important to note that when you define a function within an item or component in QML, it becomes a property of that item or component. This means that you can access the function using dot notation. Let's make this clear through an example. Create a brand new Qt Quick project and change the code in your main.qml file like below

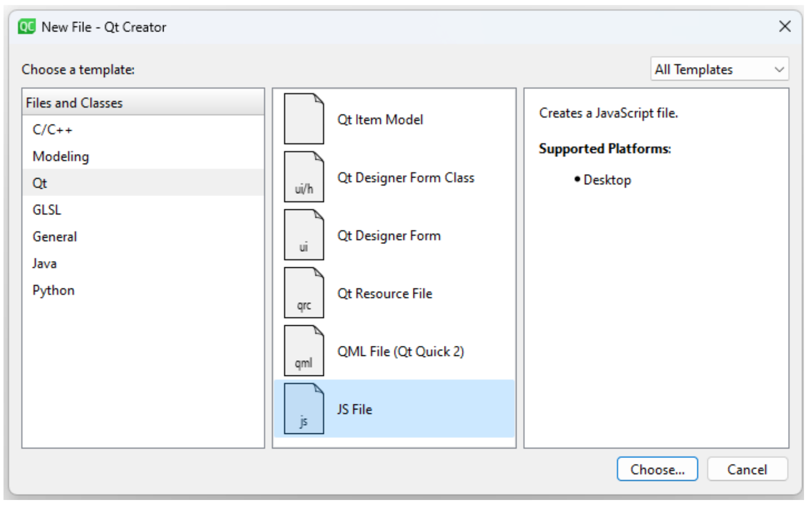
```
//main.qml
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Functions and Scope Demo")

    function min ( a ,b)
    {
        return Math.min(a,b)
    }

    Rectangle {
        id : mRectId
        width: min( 500,400)
        height: 100
        anchors.centerIn: parent
        color: "blue"
    }

    MouseArea {
        id : mMouseAreaId
        anchors.fill: parent

        function sayMessage()
        {
            console.log("Hello there")
        }

        onClicked: {
            sayMessage()
            console.log(min(10,12))

        }
    }

    Component.onCompleted: {
        console.log("The width of the rectangle is: "+ min(500,400))
        mMouseAreaId.sayMessage()
    }
}
```

`min` is a root level function in this case because it’s defined in the scope (inside) of the root level element in the QML file. We use it inside mRectId to set the width of the rectangle, we use it inside the click handler of `mMouseAreaId` and we use it inside the Component.onCompleted handler that is triggered when the window finishes loading. This illustrates that we access a root-level defined function anywhere in the element where it was defined, provided we try to access it past the point of its definition.

Functions defined locally in other elements should be qualified with the element ID when you access them outside the scope where they were defined. We use `sayMessage` as an example here. It is defined inside our MouseArea and to access it in the Component.onCompleted handler (outside its scope of definition), we have to qualify it with the MouseArea ID, `mMouseAreaId`.

Javascript Direct Imports

The last section was all about zooming in on some Javascript code that could be lurking in our QML files. But all the Javascript was hosted in our main.qml file. For organizational purposes, it is possible to host our Javascript code in external Javascript files and import them into our QML files if the need arises. To see this in action, create a brand new Qt Quick project and leave the main.qml as is at the moment. Add a new Javascript file to the project by summoning the File Menu and selecting **File › New File › Qt › JS File**



Creates a JavaScript file.

Supported Platforms:

• Desktop

Figure 85. New Javascript File Wizard

Hit next, name the file and leave it to be saved in the current project. Leave the rest to the defaults and finish. When using CMake with QML, just like when adding any other file type we've dealt with so far, we have to register the new file in our CMakeLists.txt. Javascript files go together with QML files and we need to modify our `qt_add_qml_module` command as follows.

```
qt_add_qml_module(app4-JsDirectImpoart
    URI 4-JsDirectImpoart
    VERSION 1.0
    QML_FILES main.qml utilities1.js
)
```

The Javascript file should now be visible in the Qt Creator file viewer. Open the file and change its content to be like below

```
//utilities1.js
function greeting()
{
    console.log("Hello there from external JS file utilities1.js")
}
```

The file contains a single Javascript function to print out some message. Now the job is making code in this file available for use in our main.qml file. Open the main.qml file and change its content be like below.

```
//main.qml
import QtQuick
import "utilities1.js" as Utilities1

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Javascript Direct Import")

    Rectangle {
        width : 300
        height: 100
        color: "yellowgreen"
        anchors.centerIn: parent

        Text {
            text : "Click Me"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
               Utilities1.greeting()
            }
        }
    }
}
```

The main purpose of the code is to call the `greeting` function from utilities1.js when the MouseArea is clicked. Before we call the function though, main.qml needs to be made aware of the code in utilities1.js. You do that by importing utilities1.js into main.qml. The statement

```
import "utilities1.js" as Utilities1
```

imports all the code from utilities1.js and makes it available under the Utilities1 name. To call the function, all we need to do is

```
Utilities1.greeting()
```

Anything from utilites1.js that we use in main.qml needs to be qualified with the Utilities1 name. With this infrastructure in place, you can go on and put other functions in the external Javascript file and even use it multiple times across different qml files if that makes sense for your project. For now just run the application and run it to prove that the setup here works. You should see a yellowgreen rectangle in the center of the window. If you click on it, you should see the message

```
Hello there from external JS file utilities1.js
```

coming from the external Javascript file. The example here imports Javascript code directly into our main.qml file. It is also possible to import the file indirectly, a subject we explore more of in the next section.

Javascript Indirect Imports

What we did in the last section is to directly import a Javascript file into a QML file. But it is also possible to import a Javascript file that imports another Javascript file and get access to the features in the second Javascript file from QML.

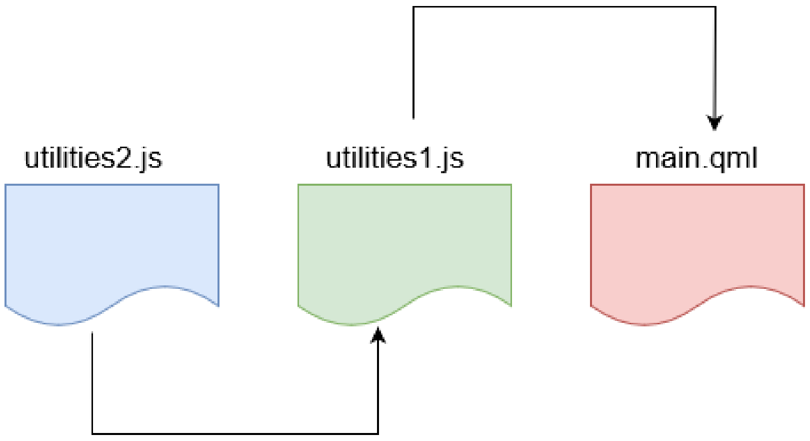


Figure 86. Javascript Indirect Import

In the illustration above utilities2.js contains some features that are imported into utilities1.js. QML then imports utilities1.js and uses its features, possibly indirectly using things coming from utilities1.js. This is the setup we're going for in this section and looking at the constructs QML offers to achieve it. Create a brand new Qt Quick project and add two Javascript files: one named utilities1.js and the other named utilities2.js. Please remember to register the new Javascript files in the CMakeLists.txt file. Leave the main.qml file as is and change the content of the Javascript files as follows.

```
//utilities1.js
.import "utilities2.js" as Utilities2

function greeting()
{
    console.log("Hello there from external JS file : utilities1.js")
}

function combineAges( age1, age2)
{
    return Utilities2.add(age1,age2)
}
```

```
//utilities2.js
function add(a,b) {
    console.log("Method from utilities2.js called")
    return a + b
}
```

utilities2.js has the `add` function that we need to use in utilities1.js. For that to happen utilities1.js needs to import utilities2.js and the syntax to do that is

```
.import "utilities2.js" as Utilities2
```

Notice the dot in front of import. With the correct import statement in place, the `combineAges` function can safely use the function from utilities2.js. combineAges itself is used in the main.qml file after directly importing utilities1.js just like we did in an earlier section. With this infrastructure, we can modify our main.qml file to be like below

```
import QtQuick
import "utilities1.js" as Utilities1
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Javascript Import Demo")

    Rectangle {
        width : 300
        height: 100
        color: "yellowgreen"
        anchors.centerIn: parent

        Text {
            text : "Click Me"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
                console.log("Our ages combined yield :" + Utilities1.combineAges(33,
17))
                // value = Utilities1.add(33,17) //Error
            }
        }
    }
}
```

Notice that in the click handler, we are calling the `combineAges` function which in turn uses the `add` function coming from utilities2.js. Be careful though, just because utilities2.js is imported into utilities1.js doesn't mean we can use the `add` function directly in the main.qml file. If you try to do that you'll get an error. Try to compile and run the application. If you click on the rectangle you should a console.log message proving that our `combineAges` function is working as expected.

```
qml: Method from utilities2.js called
qml: Our ages combined yield :50
```

Try to uncomment the line

```
// value = Utilities1.add(33,17) //Error
```

and run the application. You should get some kind of error saying that the add function is not known in the context. This is my preferred way to import Javascript files in QML projects as the implementation details in utilities2.js don't leak in the main.qml file. And if you really wanted, you could directly import utilities2.js in main.qml. But ultimately, my job here is to make you aware of the possibilities. Over time, you'll develop an intuition for what works best for your QML projects. If you really want indirectly imported code to be available in the main.qml file, Qt offers a way to do that through Qt.include and we'll look at that in the next section.

Qt.Include

The setup in the last section had a limitation in that code in files you include using the syntax

```
.import "utilities2.js" as Utilities2
```

isn't available for use in the final QML file. This section explores a facility Qt offers to remediate that. All we have to do is to replace the previous import statement in utilities1.js with the following Qt.include statement

```
Qt.include("utilities2.js")
```

and if you directly import utilities1.js in main.qml, the code from utilities2.js will be available for use. This best illustrated with a project we can play with. Create a brand new Qt Quick project from Qt Creator and add our trusty two Javascript files: utilities1.js and utilities2.js. Leave main.qml as is and populate the Javascript files as shown below.

```
//utilities2.js
function add(a,b) {
    console.log("Method from utilities2.js called")
    return a + b
}
```

```
//utilities1.js
Qt.include("utilities2.js")

function greeting()
{
    console.log("Hello there from external JS file : utilities1.js")
}

function combineAges( age1, age2)
{
    return add(age1,age2)
}
```

The only difference from the last section is that utilities1.js is using the `Qt.include` facility to get access to the `add` function. What` Qt.include` does is like literally copying the content from utilities2.js and pasting in utilities1.js. This makes it possible for the add function to be used directly in main.qml even if it's not available in utilities1.js. Change your main.qml file to be like below

```
//main.qml
import QtQuick
import "utilities1.js" as Utilities1
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Javascript Import Demo")

    Rectangle {
        width : 300
        height: 100
        color: "yellowgreen"
        anchors.centerIn: parent

        Text {
            text : "Click Me"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
            //console.log("Our ages combined yield :" + Utilities1.combineAges(33,17))
            console.log("Our ages combined yield :" + Utilities1.add(33,17))
            }
        }
    }
}
```

You can see that we're using the add function, even though it wasn't directly present in utilities1.js, which is what we directly imported into main.qml. If you run the application and click on the rectangle, you should see our console.log message printed out

```
qml: Method from utilities2.js called
qml: Our ages combined yield :50
```

proving that the add function is available in main.qml, something that triggered an error in the last section. Again, just because the facility is available doesn't mean you have to use it. I am showing you the possibilities here and will pick those that help you out and leave the rest.

Javascript Modules

JavaScript has support for modules. A facility for organizing and encapsulating code in separate files, making it easier to maintain, reuse, and collaborate with others. It's a new way to do what we've already done with external Javascript files but with the ability to control what's exported and what's not. To make functions and variables in a module available to other modules or scripts, they must be explicitly marked for export using the `export` keyword. These exported items can then be imported into other scripts or modules using the `import` statement and accessed using dot notation. Javascript module files have the extension `.mjs`. Modules are a Javascript feature and they are documented here. QML supports a subset of what is available in browsers and we'll cover the basics here.

The first issue when trying to use Javascript modules in QML is that Qt Creator doesn't provide a wizard to create a .mjs file. I personally use other editors like Notepad++ or VS Code to do that and put them somewhere in the path of my Qt Quick application. The second problem is that some import and export keywords are marked with squiggly lines while they are perfectly valid syntax. In that case, I just ignore them and compile, run and test the application to make sure it works as expected.

To play with Javascript modules we will reorganize our utilities1.js and utilities2.js files into modules and get them used in our main.qml file. Create a brand new Qt Quick project and leave the main.qml file as is at the moment. Now, use your favorite editor (I use Notepad++ and Visual Studio Code) to add two Javascript module files: utilities1.mjs and utilities2.mjs. Notice the extension here. With the files in place, register them in your CMakeLists.txt file by modifying your `qt_add_qml_module` command

```
qt_add_qml_module(app7-JsModules
    URI 7-JsModules
    VERSION 1.0
    QML_FILES main.qml utilities1.mjs utilities2.mjs
)
```

After you save the CMakeLists.txt file, the module files should be visible in the Qt Creator project file viewer. Open utilities2.mjs and change the content as shown below

```
//utilities2.mjs
export function add(age1,age2){
    return age1 + age2
}

function subtract(age1,age2){
    return age1 - age2;
}
```

One of the core features of Javascript modules is to have fine-grained control over what is usable from the outside and what's not. If you want a function to be usable by other files that import our file, you explicitly mark it for export using the `export` keyword. If a function is not marked with the export keyword, it won't be usable from the outside. Following that logic, the `add` function is usable from the outside but the `subtract` function is not.

Open the utilities1.mjs and change the content as shown below

```
//utilities1.mjs
import * as Utilities2 from "utilities2.mjs";

export function combineAges(age1,age2){
    return Utilities2.add(age1,age2)
}

export function ageDiff(age1,age2){
    return Utilities2.subtract(age1,age2)
}
```

The first thing you see is that we are importing code from the utilities2.mjs module file through the statement

```
import * as Utilities2 from "utilities2.mjs";
```

In English, the statement means that we want to import everything from the utilities2.mjs file and make it available in the current file under the name Utilities2. The asterisk means that we want everything. After the import statement, utilities1.mjs will set up its own functions and export them all for use in other files that import it. We have imported everything from utilities2.mjs but Javascript has the ability to only import specific names. For example, we could only import the `add` function from utilities2.mjs. You can read more about this in the official docs for Javascript but I couldn't get that to work in Qt Creator and will leave it to that in the book. We can now open our main.qml file and change its content as shown below.

```
//main.qml
import QtQuick
import "utilities1.mjs" as Utilities1
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Javascript Import Demo")

    Rectangle {

        width : 300
        height: 100
        color: "yellowgreen"
        anchors.centerIn: parent
        Text {
            text : "Click Me"
            anchors.centerIn: parent
        }

        MouseArea {
            anchors.fill: parent
            onClicked: {
               //Calling a properly exported mathod
               console.log("Our ages combined yield :" + Utilities1.combineAges(33,
17))
            }
        }
    }
}
```

We import the module file just like we did for regular Javascript files

```
import "utilities1.mjs" as Utilities1
```

and call the `combineAges` function from utilities1.mjs. Remember that this function used the add function from utilties2.mjs in its body. If you compile and run the application, you should see a yellowgreen rectangle and if you click on it you should see the message

```
qml: Our ages combined yield :50
```

in the Application Output pane from Qt Creator. Do remember that, from the outside, you can only use explicitly exported functions. In our code, if you take a look at utilities2.mjs, you'll notice that the `subtract` function is not exported. Let's see what happens if we try to execute code that uses it indirectly from main.qml. Modify the click handler inside our rectangle to look like code below.

```
onClicked: {
    //Can't call a method that's not exported: subtract isn't exported from
utilities2.mjs
    console.log("Age diff : " + Utilities1.ageDiff(33,17))
}
```

If you compile and run the application with these changes applied, you'll see the green rectangle. If you click on it, you'll see an error in the Application Output pane. Mine says something like below

```
qrc:/7-JsModules/utilities1.mjs:8: TypeError: Type error
```

The error doesn't say exactly what's wrong but it at least points to the suspect line in utilities1.mjs. If you look closely, it'll be the line that calls the subtract function, which isn't exported from utilties2.mjs and therefore not accessible from utilities1.mjs. I wish the error here was more descriptive of the problem but this is what we get from Qt Creator at the moment.

The last thing I want to make clear is that just because the add function is available for use in utilties1.mjs, doesn't mean that main.qml will get access to it by just importing utilities1.mjs. To make this clear, modify your click handler to be like below

```
onClicked: {
    //Just because add is usable from utilities1.js doesn't mean main.qml
        // can use it.
        console.log ("The sum is : " + Utilities1.add(33,17))//Error
}
```

If you compile and run the application, you'll see your good old yellowgreen rectangle. Clicking on it, you'll also get an error message

```
qrc:/7-JsModules/main.qml:70: TypeError: Property 'add' of object TypeError: Type
error is not a function
```

meaning that the main.qml file doesn't see add as a function because it was exported **to** utilities1.mjs but **not to** main.qml. Javascript modules provide a way to re-export the add function and make it visible and usable by files importing utilties1.mjs but again, I couldn't get this to work with QML and I'll leave it to that here in the book.

Chapter Summary

Javascript is the language that is natively used by QML to add interactivity to your user interfaces. It's used to set up handlers, for example, to respond when a button is clicked, property bindings, functions and much more. The chapter was focused on bringing the Javascript in your QML to the surface for everybody to see. We looked at how functions are scoped in your QML files: sometimes, you'll need to go through IDs to access the function if it's nested deep in your QML element hierarchy. Next, we looked at our options when it comes to partitioning our Javascript code into separate files: Javascript direct imports, indirect imports and Qt.include. Finally, we explored the basic usage of Javascript modules if you need to go down that path.

Chapter 7: Positioning

Positioning is a big thing in graphical user interface design. The more tools and flexibility a GUI technology provides when it comes to laying out user interface components, the better the technology. QML provides a host of tools you can use to control the layout of your elements in the window, and we had a chance to use a few of them already, without going into the details of how they work. You have seen things like anchors, Row and Column. All these come from some positioning mechanisms offered by QML. One can position a visual element directly by assigning values to its `x` and \`y\`properties. That's hard positioning though. It's not hard to see that doing things this way will break your user interface if the window is resized, or if things are added or removed from the user interface dynamically. QML allows us to position elements using the anchor mechanism, in which elements are positioned relative to others. If anchors prove hard to maintain, QML provides wrapper positioners like Row, Column and Grid, that you can use to arrange items horizontally, vertically and in a grid structure. If you need resizing, QML also provides a number of layouts you can use to lay things out. This chapter will incrementally explore all these positioning mechanisms along with the properties they provide to tune your user interface just like you want it.

Anchors

Anchors are one of the positioning mechanisms offered by QML. They work by letting you position your QML element's anchor line relative to another element's anchor lines. Each QML visual element has 6 pre-defined anchor lines: top, bottom, left, right, horizontal-center and vertical-center.

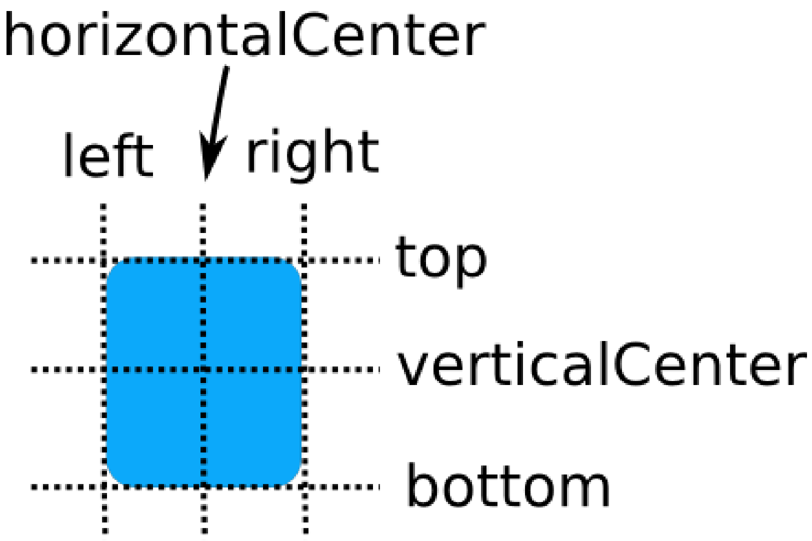


Figure 87. QML Anchor Lines

On top of these, we also have the baseline anchor line, which is the line on top of which text sits if our element happens to display some text. If there is no text, the baseline anchor line is the same as the top anchor line. We'll set up an example to help play with anchors and learn on the go. The goal for this section is to build something like below

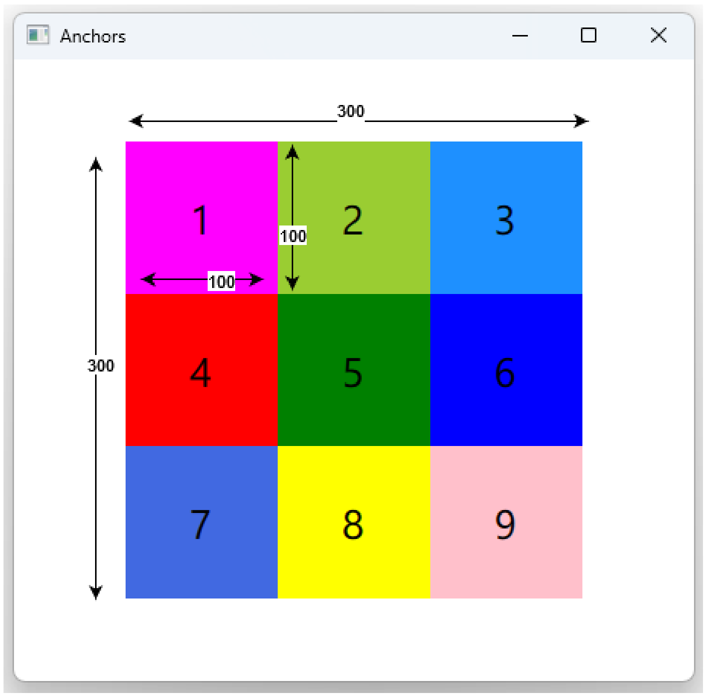


Figure 88. The Goal

We will have a container rectangle with a size of 300 in width and 300 in height. The rectangle is meant to contain nine rectangles, each 100 in width and 100 in height, and we should end up with a 9x9 grid-like structure. The rectangles are labeled with numbers and colored to make them easy to follow around. Let's describe our positioning design in words, and then we'll implement this in QML using anchors.

- Rectangle "1" will have no explicit positioning mechanism inside the container rectangle. QML will stick it in the top left corner of the container rectangle. In other words, its top-left corner will coincide with the top-left corner of the container.
- We want Rectangle "2" to be aligned to the right of Rectangle "1". In other words, the left side of Rectangle "2" should coincide with the right side of Rectangle "1".
- We want Rectangle "3" to be positioned to the right of Rectangle "2". The left side of Rectangle "3" will be aligned with the right side of Rectangle "2"

With this, we should have the first row of our 9x9 grid taken care of. Let's work on the second row.

- We want the left side of Rectangle "4" to coincide with the left side of the container rectangle. We also want it to be positioned below one of the rectangles in the first row.
- We want Rectangle "5" to be to the right of Rectangle "4" and below one of the rectangles in the first row
- We want Rectangle "6" to be to the right of Rectangle "5" and below one of the rectangles in the first row

This takes care of the second row in our 9x9 grid. We could position the third row relative to the second row like we just did for row 2 relative to row 1, but let's do the third row exclusively relative to Rectangle "5" which is in the center of the grid. This will help us learn about some behaviors of anchor positioning mechanism you need to know about.

- We want the right side of Rectangle "7" to coincide with the left side of Rectangle "5" and the top side of Rectangle "7" to coincide with the bottom side of Rectangle "5"

- We want the left side of Rectangle "8" to coincide with the left side of Rectangle "5" and the top side of Rectangle "8" to coincide with the bottom side of Rectangle "5"
- We want the left side of Rectangle "9" to coincide with the right side of Rectangle "5" and the top side of Rectangle "9" to coincide with the bottom side of Rectangle "5"

This may seem verbose, but it’s the exact same thing you’ll be describing in QML code using anchors. Let’s create a brand new Qt Quick application and change the main.qml file to contain code like below.

```
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Rectangle {
        id : containerRectId
        width: 300
        height: width
        border.color: "black"
        anchors.centerIn: parent

        Rectangle {
            id : topLeftRectId
            width: 100
            height: width
            color: "magenta"
            Text{
                anchors.centerIn: parent
                text : "1"
                font.pointSize: 20
            }
        }
    }
}
```

The code sets up the container rectangle inside our Window element under the ID `containerRectId`. The container just contains one rectangle at this point and it is the one with a label of "1". You can see that the label is implemented as a Text element centered in the first rectangle with an ID of `topLeftRectId`. topLeftRectId has no explicit positioning. QML will just stick it in the top left corner of the container rectangle. If you compile and run the application at this point, you’ll see our magenta rectangle with the label "1" in the container rectangle

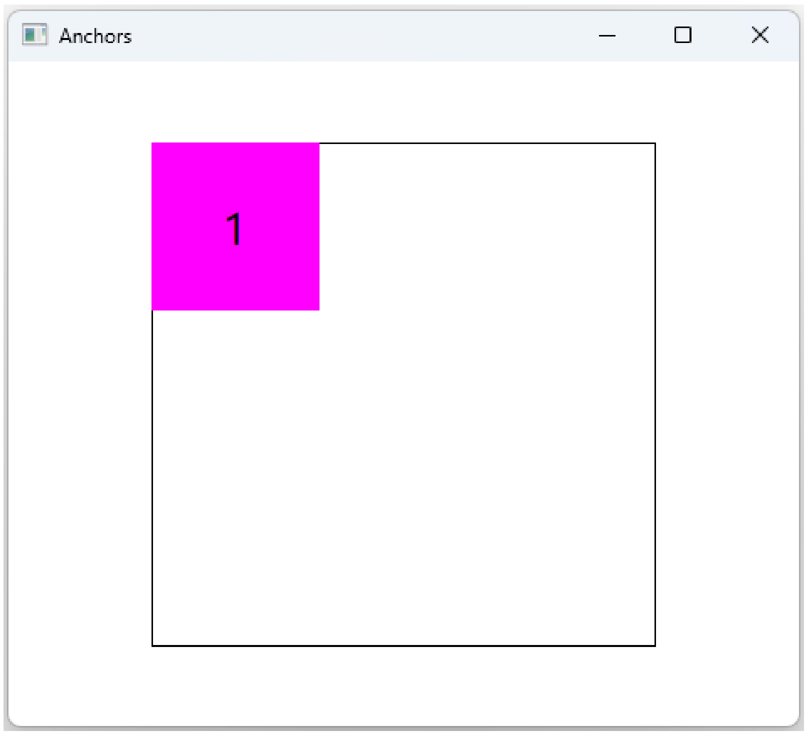


Figure 89. Rectangle 1

Notice that our container rectangle is just a black outline because we only specified the `border.color` grouped property. Rectangle "1" is stuck in the top-left corner of containerRectId. We didn't do any active positioning using anchors to get Rectangle "1" in place but we'll have to do that to get Rectangle "2" where it's wanted. Visiting our English description of what we want : * We want Rectangle "2" to be aligned to the right of Rectangle "1". In other words, the left side of Rectangle "2" should coincide with the right side of Rectangle "1".

we can translate this into anchor language like below

```
//Inside containerRectId
Rectangle {
    id : topCenterRectId
    width: 100
    height: width
    color: "yellowgreen"
    anchors.left: topLeftRectId.right
    Text{
        anchors.centerIn: parent
        text : "2"
        font.pointSize: 20
    }
}
```

The most important statement here being `anchors.left: topLeftRectId.right`. Translated into English, it means that the left side of topCenterRectId should coincide with the right side of topLeftRectId. If you put this code below topLeftRectId in our Qt Quick project the updated code should look like below

```
//main.qml
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Rectangle {
        id : containerRectId
        width: 300
        height: width
        border.color: "black"
        anchors.centerIn: parent

        Rectangle {
            id : topLeftRectId
            width: 100
            height: width
            color: "magenta"
            Text{
                anchors.centerIn: parent
                text : "1"
                font.pointSize: 20
            }
        }

        Rectangle {
            id : topCenterRectId
            width: 100
            height: width
            color: "yellowgreen"
            anchors.left: topLeftRectId.right
            Text{
                anchors.centerIn: parent
                text : "2"
                font.pointSize: 20
            }
        }
    }
}
```

Run the application and you should see two rectangles labeled "1" and "2" inside the container rectangle.

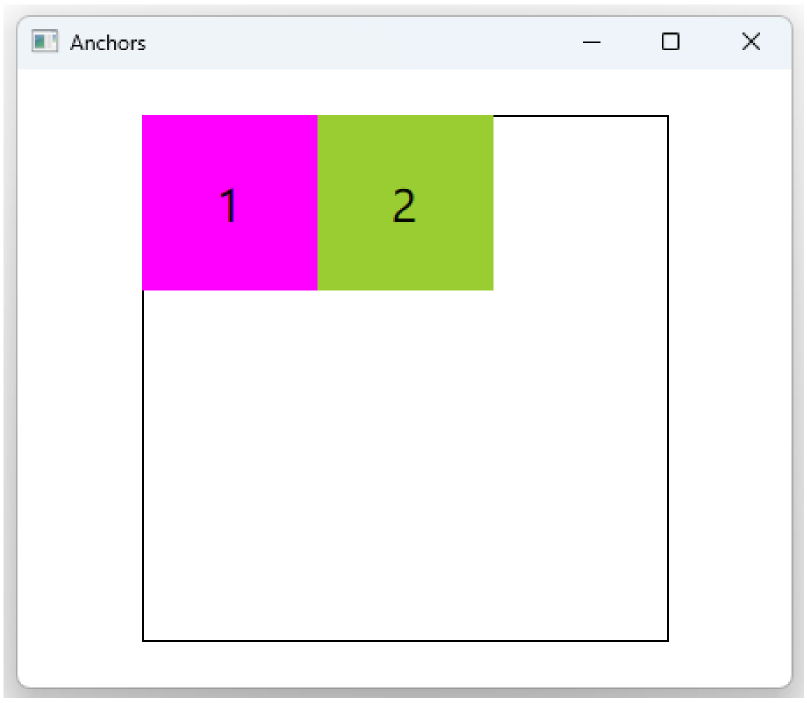


Figure 90. Two rectangles in the first row

One thing worthy of note here is that we didn't explicitly specify the position of `topCenterRectId` relative to the `containerRectId` top anchor line. You can think that by default `topCenterRectId` was stuck in the top-left corner of `containerRectId` and we forced it to move towards the right and stop at the right side of `topLeftRectId`, keeping its default vertical position relative to the top anchor line for `containerRectId`.

You can add code for the third rectangle labeled "3" below the `topCenterRectId` rectangle in the main.qml file. The logic is the same:

- We want Rectangle "3" to be positioned to the right of Rectangle "2". The left side of Rectangle "3" will be aligned with the right side of Rectangle "2".

```
//Inside containerRectId, after topCenterRectId
Rectangle {
    id : topRightRectId
    width: 100
    height: width
    color: "dodgerblue"
    anchors.left: topCenterRectId.right
    Text{
        anchors.centerIn: parent
        text : "3"
        font.pointSize: 20
    }
}
```

If you run the application with the changes applied, you should see three rectangles in the first row of our 9x9 grid.

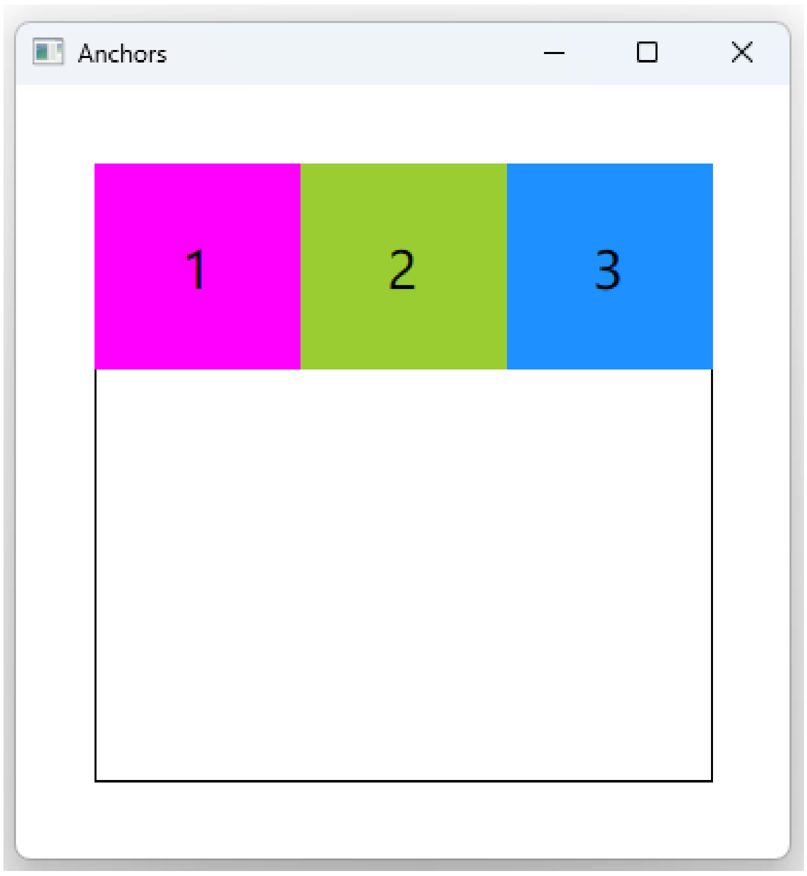


Figure 91. Three rectangles in the first row

Now that you are starting to get the hang of it, we can do the second row in one go. Just for reference, remember our goal:

- We want the left side of Rectangle "4" to coincide with the left side of the container rectangle. We also want it to be positioned below one of the rectangles in the first row.
- We want Rectangle "5" to be to the right of Rectangle "4" and below one of the rectangles in the first row
- We want Rectangle "6" to be to the right of Rectangle "5" and below one of the rectangles in the first row

Translated in QML anchor code, we can position rectangles "4", "5" and "6" inside containerRectId like below

```
//Inside containerRectId, after topRightRectId
//...
Rectangle {
    id : centerLeftRectId
    width: 100
    height: width
    color: "red"
    anchors.top: topLeftRectId.bottom
    Text{
        anchors.centerIn: parent
        text : "4"
        font.pointSize: 20
    }
}

Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom
    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }

}
Rectangle {
    id : centerRightRectId
    width: 100
    height: width
    color: "blue"
    anchors.left: centerCenterRectId.right
    anchors.top: topRightRectId.bottom
    Text{
        anchors.centerIn: parent
        text : "6"
        font.pointSize: 20
    }
}
//...
```

Rectangle "4" is forced to be below Rectangle "1" through the anchor statement `anchors.top: topLeftRectId.bottom`, Rectangle "5" is our rectangle in the center. It is forced to the right of Rectangle "4" and below any of the rectangles in the first row, we chose topRightRectId through the anchor statements

```
anchors.left: centerLeftRectId.right
anchors.top: topRightRectId.bottom
```

and Rectangle "6" is positioned to the right of the center rectangle, Rectangle "5", and below any rectangle in the first row, again we chose topRightRectId. The anchor code guilty for that is

```
anchors.left: centerCenterRectId.right
anchors.top: topRightRectId.bottom
```

If you run the code with these changes applied, you should see the second row of our 9x9 grid populated with rectangles labeled "4", "5" and "6"

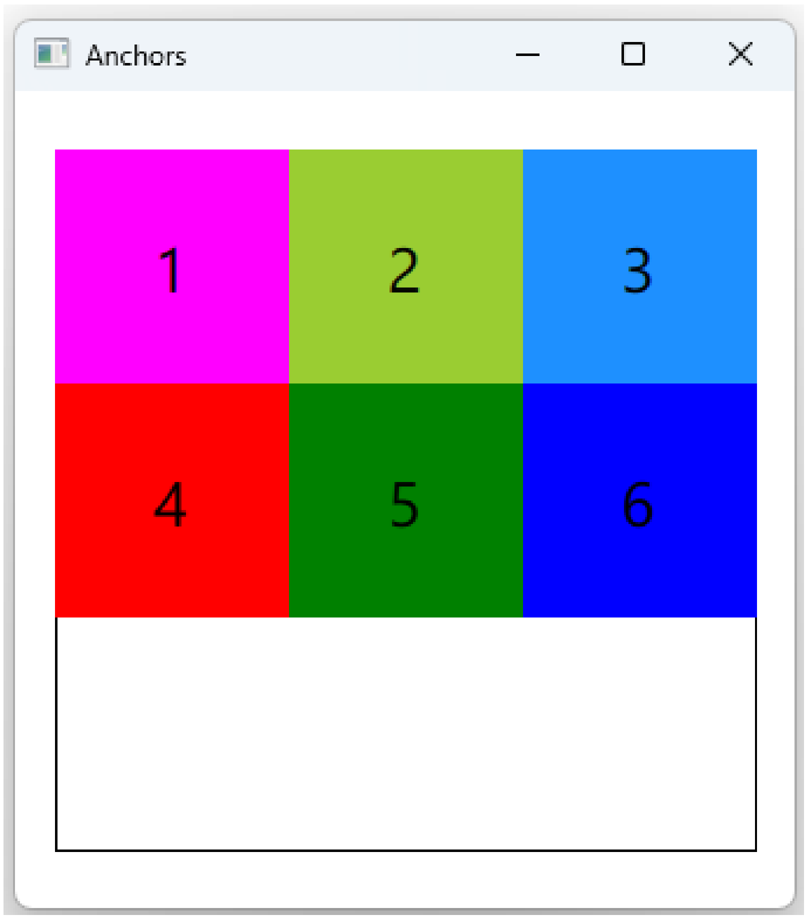


Figure 92. Three rectangles in the second row

We have the third row left, and if you remember we want every single rectangle in the third row to be positioned relative to the rectangle in the center: centerCenterRectId. Our goals are reproduced here for reference:

- We want the right side of Rectangle "7" to coincide with the left side of Rectangle "5" and the top side of Rectangle "7" to coincide with the bottom side of Rectangle "5"
- We want the left side of Rectangle "8" to coincide with the left side of Rectangle "5" and the top side of Rectangle "8" to coincide with the bottom side of Rectangle "5"
- We want the left side of Rectangle "9" to coincide with the right side of Rectangle "5" and the top side of Rectangle "9" to coincide with the bottom side of Rectangle "5"

As an exercise, I advise you to try and translate this into QML anchor code that goes after centerRightRectId inside the container rectangle. After trying you can reference our QML code below.

```
//The bottom row will be positioned in terms of centerCenterRectId
//...
Rectangle {
    id : bottomLeftRectId
    width: 100
    height: width
    color: "royalblue"
    anchors.right: centerCenterRectId.left
    anchors.top: centerCenterRectId.bottom
    Text{
        anchors.centerIn: parent
        text : "7"
        font.pointSize: 20
    }
}
Rectangle {
    id : bottomCenterRectId
    width: 100
    height: width
    color: "yellow"
    anchors.left: centerCenterRectId.left
    anchors.top: centerCenterRectId.bottom
    Text{
        anchors.centerIn: parent
        text : "8"
        font.pointSize: 20
    }
}

Rectangle {
    id : bottomRightRectId
    width: 100
    height: width
    color: "pink"
    anchors.left: centerCenterRectId.right
    anchors.top: centerCenterRectId.bottom
    Text{
        anchors.centerIn: parent
        text : "9"
        font.pointSize: 20
    }
}
//...
```

Relative to centerCenterRectId, Rectangle "7" is on the left and below. This translates into anchor code below for Rectangle "7"

```
anchors.right: centerCenterRectId.left
anchors.top: centerCenterRectId.bottom
```

In other words, the right side of Rectangle "7" coincides with the left side of centerCenterRectId and the top side of Rectangle "7" coincides with the bottom side of centerCenterRectId. I leave the translation of anchor code for Rectangles "8" and "9" to you as an exercise. If your run the code with the changes applied, you should see the third row of our 9x9 grid populated with rectangles "7", "8" and "9".

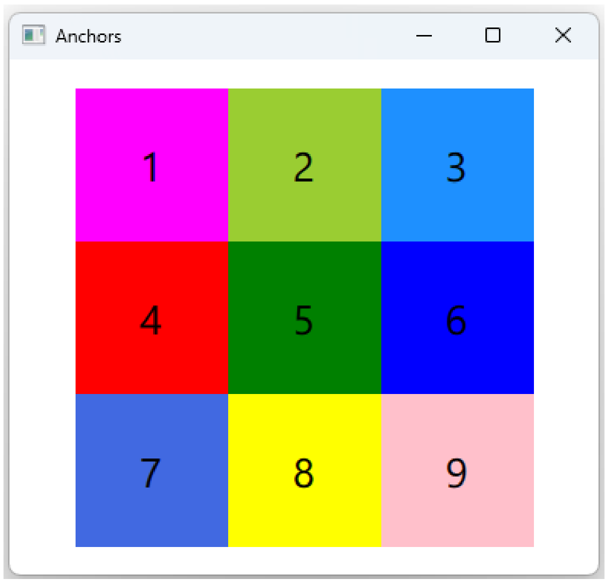


Figure 93. Three rectangles in the third row

This should give you a good grasp of how the anchor positioning system works. If you need more info, I would suggest you read the official docs page on the topic.

HorizontalCenter and Vertical Center

I wouldn’t leave this section without touching on the horizontalCenter and verticalCenter anchor lines. They allow you to position your elements relative to the horizontal and vertical center anchor lines from the parent element. In our 9x9 grid of rectangles, remember that we want Rectangle "5" to be perfectly centered in the parent container rectangle. Another way to achieve that would be to position centerCenterRectId relative to the center anchor lines. Modify your code for centerCenterRectId like below

```
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

     /*
     //Position relative to other rectangles
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom
    */

    //Position the rectangle relative to parent center lines
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.verticalCenter: parent.verticalCenter
    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

In other words, we want the horizontal center line for centerCenterRectId to coincide with the horizontal center line for the container rectangle and the vertical center line for centerCenterRectId to coincide with the vertical center line for the container rectangle. This would perfectly center Rectangle "5" in the container rectangle. If you run the application with these changes applied, you should see the same visual result but the implementation is different.

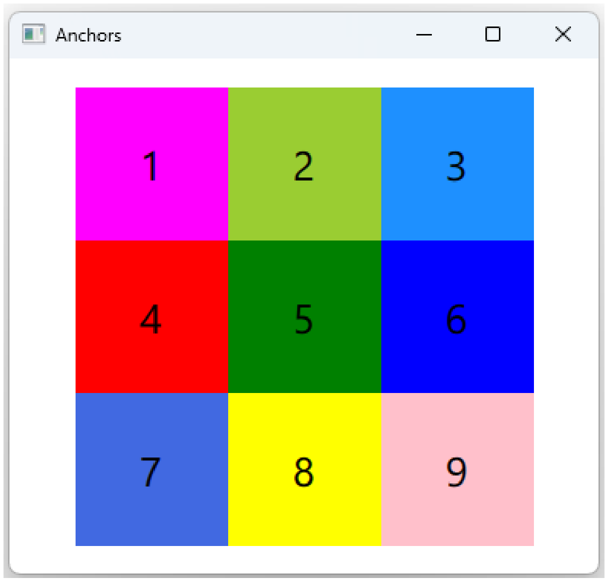


Figure 94. Three rectangles in the third row

anchor.centerIn: parent

Now that you have seen the center-line anchors in action, it's time to unveil what the `anchor.centerIn: parent` setting does. It's just a shorthand for setting both horizontalCenter and verticalCenter anchors. To prove our point we can modify the positioning code of our centerCenterRectId rectangle to something like below

```
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"
    //Position relative to other rectangles
    /*
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom
    */

    //Position relative to parent center lines
    /*
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.verticalCenter: parent.verticalCenter
    */
    //Position using the anchors.centerIn shorthand
    anchors.centerIn: parent

    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

When the QML engine sees the statement `anchors.centerIn: parent` in centerCenterRectId, it expands it to

```
anchors.horizontalCenter: parent.horizontalCenter
anchors.verticalCenter: parent.verticalCenter
```

behind the scenes, giving us exactly the same visual result when we run the application. This should give you a firm grasp on using the anchor positioning system to lay out your elements in QML applications.

Anchor Margins and Offsets

When using the anchor positioning system, one often needs to leave some breathing space around elements. That's where margins and offsets come in. You use margins to add some space around border-side anchor lines like top, bottom, left and right. You use offsets to add some breathing space around center anchor lines and the baseline anchor line. One thing to keep in mind is that **for a margin or offset to work, there needs to be a related anchor line already set**. Let's clarify all this with an example. Create a brand new Qt Quick project and change the main.qml file to contain the last code we had in the project on anchors, that set up an 9x9 grid of rectangles. If you run the application, it should look something like below

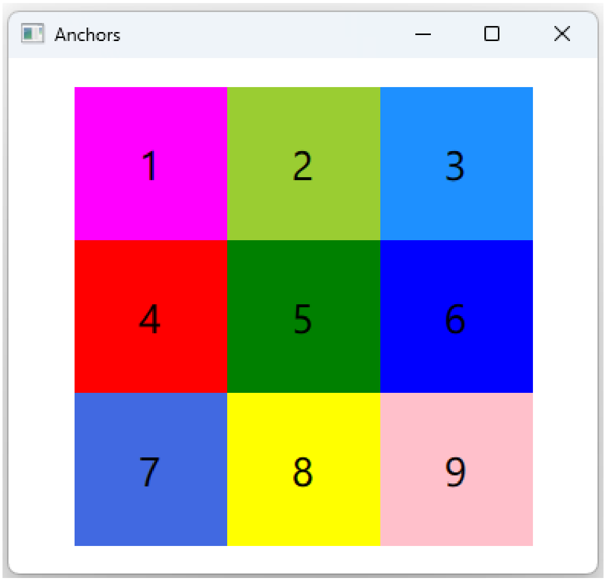


Figure 95. The 9x9 grid of rectangles

The full final source code for this section is available in the book git repository. Sometimes we won't be able to show the full code for space reasons and will only show relevant parts.

Margins

The code responsible for `centerCenterRectId` is shown below. Also do remember that the third row in our 9x9 grid is anchored relative to `centerCenterRectId`.

Make sure centerCenterRectId is centered by anchoring relative to other rectangles like below for margins to work properly. That won't give you problems with the code and we'll explain why shortly.

```
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Anchor relative to other rectangles
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom
    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

Suppose we wanted to push centerCenterRectId down by 10 px. One way to do that would be to add a top-margin of 10. After we do that, there should be a distance of 10 px between the top anchor line (which is the bottom of topRightRectId) and the actual top of centerCenterRectId. Change the code for centerCenterRectId to add a top margin like below

```
//Adding a top margin
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Anchor relative to other rectangles
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom

    //Set up the top margin
    anchors.topMargin: 10
```

```
        Text{
            anchors.centerIn: parent
            text : "5"
            font.pointSize: 20
        }
    }
```

If you run the code with these changes applied, you should see centerCenterRectId pushed down by 10 px.

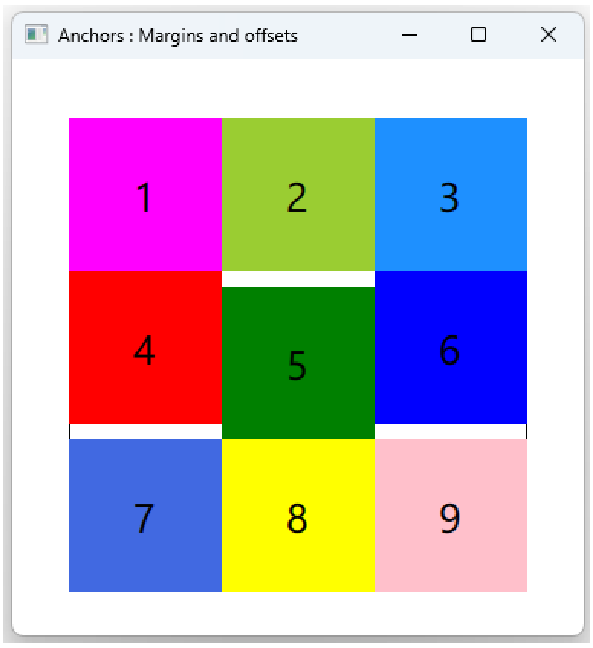


Figure 96. Center rectangle pushed down

Wait. The entire third row moved down together with centerCenterRectId! That's right. Remember that the third row is anchored relative to centerCenterRectId. So if centerCenterRect is moved by a margin, anything anchored relative to it also moves. This is important to understand.

> If an element is moved by a margin, anything anchored to the element will also be moved by the margin.

Margins Depend on Anchor lines

Another important thing to keep in mind is that, for a margin to work, you first need an anchor set for the line where you want the margin applied. Again, this is best explained by an example. Looking at the code for centerCenterRectId,

```
//The code behind centerCenterRectId
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Anchor relative to other rectangles
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom

    //Set up the top margin
    anchors.topMargin: 10
    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

you see that we have a left anchor (the left of centerCenterRectId) and a top anchor (the top of centerCenterRectId). Because we have these anchors, we have the ability to set a top margin and a left margin. But we can't set a right margin and expect it to work. To prove this, change the code and attempt to add a right margin to centerCenterRectId that's supposed to push the rectangle towards the left.

```
//right margin won't work because we don't have a right anchor set
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Using margins
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom

    anchors.topMargin: 10

    //The right margin won't work because we don't have a right anchor
    anchors.rightMargin: 10
    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

If you run the application with the changes applied, you'll be surprised to see that QML will simply silently ignore your right margin setting.

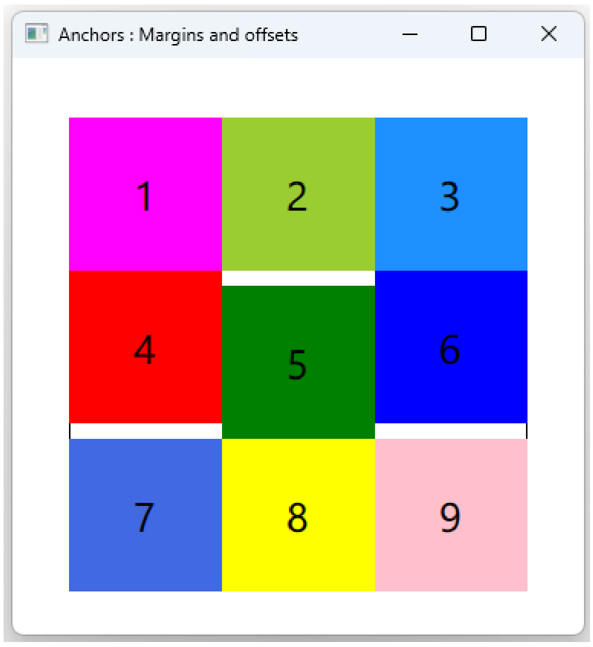


Figure 97. The right margin is ignored

But if you set the left margin it will work because we have a left anchor set

```
//Left margin will work because we have a left anchor set
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Anchor relative to other rectangles
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom

    //margins
    anchors.topMargin: 10
    //anchors.rightMargin: 10  // This margin won't work because we have no right
anchor
    anchors.leftMargin: 10

    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

Run the application with these changes applied and you should see the left margin in action

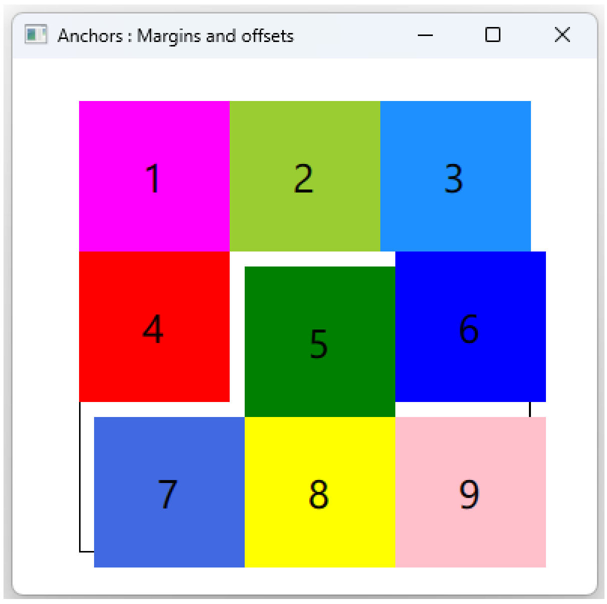


Figure 98. The left margin works

This is an important thing to keep in mind if you're not careful, you'll set up margins and they won't have any effect because you don't have anchors on the lines where you want the margin to apply. This can be very hard to fix if it's part of a relatively big QML project. You've been warned!

Offsets

Just like margins, offsets are used to add space around center anchor lines and the baseline anchor line. To play with this we will change centerCenterRectId to be centered using horizontalCenter and verticalCenter anchors and take out any margins we had previously set

```
//Center using center anchor lines
//Take out any previous margins
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Anchor relative to center anchor lines
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.verticalCenter: parent.verticalCenter

    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

If you run the application with these changes applied, you should go back to our 9x9 grid with no margins set

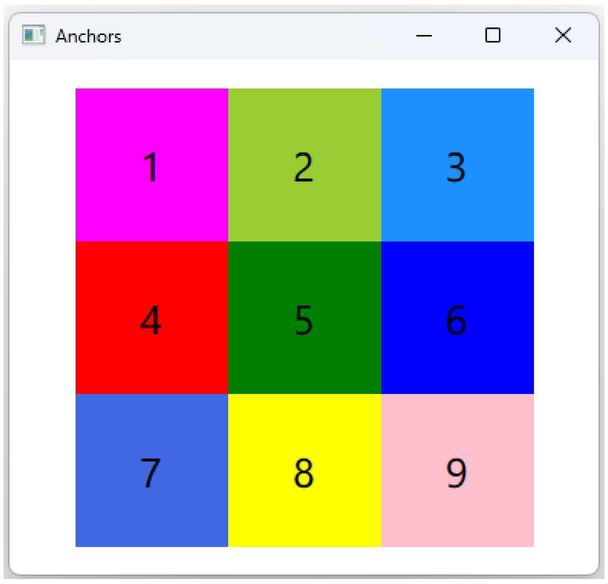


Figure 99. The 9x9 grid of rectangles with no margins

but now it's centered around horizontalCenter and verticalCenter anchor lines. We can't set margins on these center lines but we can set offsets to move centerCenterRectId relative to these anchor lines. For example, we can set an offset of 10 both vertically and horizontally

```
//Set an offset of 10 from center lines
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Anchor relative to center anchor lines
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.verticalCenter: parent.verticalCenter

    //Offsets
    anchors.horizontalCenterOffset: 10
    anchors.verticalCenterOffset: 10

    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

Run the application with these changes and you should see centerCenterRectId pushed to the right and to the bottom by 10 px.

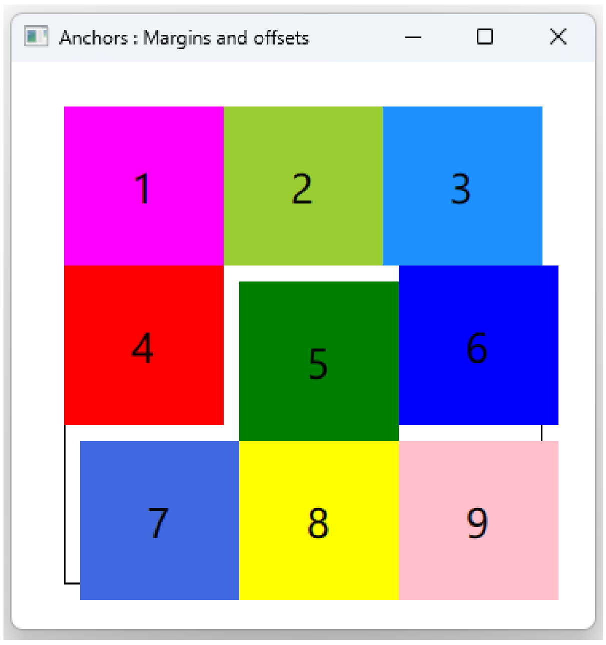


Figure 100. The center rectangle moved by offsets from center lines

Just like for margins, offsets only work if you have an anchor set on the line where you want the offset to apply. In our example here, the offsets work because we have the horizontalCenter and verticalCenter anchors set. As a proof, you can take out the code that centers centerCenterRectId relative to center anchor lines and activate the code that centers relative to other rectangles.

```
//Take out center line anchors and try to use center line offsets
//Won't work. Center line offsets only work if you have center line anchors set
Rectangle {
    id : centerCenterRectId
    width: 100
    height: width
    color: "green"

    //Anchor relative to other rectangles
    anchors.left: centerLeftRectId.right
    anchors.top: topRightRectId.bottom

    //Offsets. Won't work because we don't have center anchor lines
    anchors.horizontalCenterOffset: 10
    anchors.verticalCenterOffset: 10

    Text{
        anchors.centerIn: parent
        text : "5"
        font.pointSize: 20
    }
}
```

If you run the application with these changes applied, your offsets should be ignored.

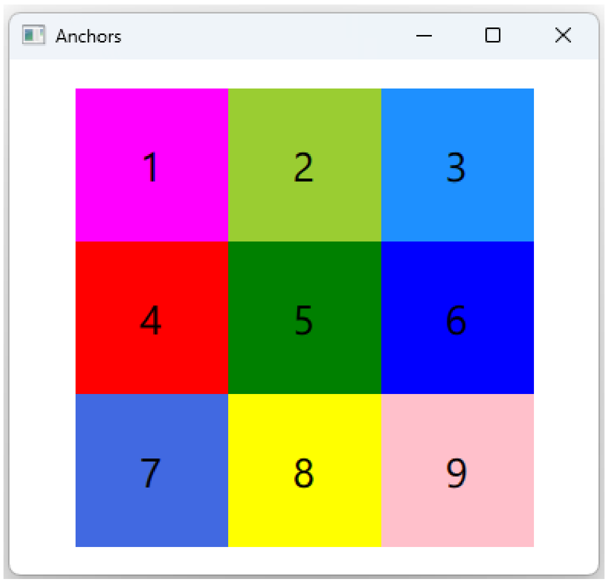


Figure 101. Offsets ignored because we don't have center anchors

It is possible to set negative margins or offsets to push things in the other direction. Use this trick if it makes sense for your project.

Anchors : Parents and Siblings

QML anchors have a simple rule that should be followed: *You can only anchor to a parent or sibling element*. If you violate this rule your anchors won't work and you'll get errors when your invalid anchor is processed. This is best explained with an example. Create a brand new Qt Quick project and change the main.qml file to contain code like below

```
//main.qml
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors : Parents and Siblings")

    Rectangle {
        id : containerRectId
        width: 300
        height: width
        border.color: "black"
        anchors.centerIn: parent

        Rectangle {
            id : topLeftRectId
            width: 100;height: width
            color: "magenta"
            //This anchor won't work because siblingRect isn't either a ibling or a
parent
            anchors.top: siblingRect.bottom
        }
    }
    Rectangle {
        id : siblingRect
        width: 200;height: 200
        color : "black"
        anchors.right: containerRectId.left
    }
}
```

The Window element contains two rectangles, a container rectangle that contains `topLeftRectId`, and a sibling rectangle named `siblingRect`. `siblingRect` is anchored against `containerRectId` because they are siblings. One thing you can't do, however, is to try to anchor `topLeftRectId` relative to `siblingRect`, **because these two rectangles don't have a parent or sibling relationship**. Suppose we wanted to align the top of `topLeftRectId` with the bottom of `siblingRect`.

If you run the application with these changes in place, you'll see our rectangles.

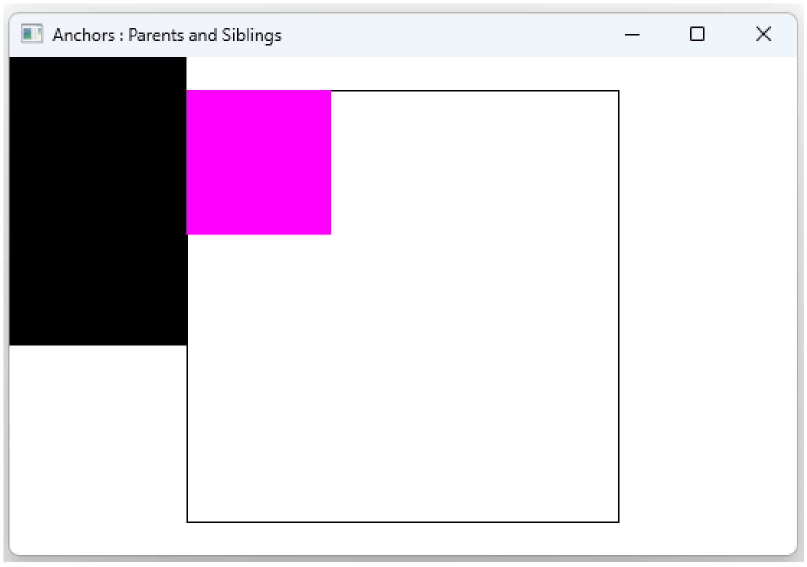


Figure 102. Anchors only work between parent and sibling

But the top of `topLeftRectId` is not aligned with the bottom of `siblingRect` as we wanted. You'll also get an error in the **Application Output** pane

```
qrc:/4-AnchorParentSibling/main.qml:32:9: QML Rectangle: Cannot anchor to an item that
isn't a parent or sibling.
```

The error is self-explanatory. Please keep this in mind as you use anchors to position your elements.

QML Positioners

Besides the anchor positioning system, QML also provides a few positioner elements you can use to lay elements out horizontally, vertically or in a grid. The elements are Row, Colum and Grid. Let's see them in action. Create a brand new Qt Quick project from Qt Creator and change the main.qml file to contain code like below

```
//main.qml
//Grid Positioner
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Rectangle {
        id : containerRectId
        width: 300;height: width
        border.color: "black"

        Grid {
           columns: 3
            Rectangle {
                id : topLeftRectId
                width: 100;height: width
                color: "magenta"
                Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
            }
            Rectangle {
                id : topCenterRectId
                width: 100;height: width
                color: "yellowgreen"
                Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
            }
            Rectangle {
                id : topRightRectId
                width: 100;height: width
                color: "dodgerblue"
                Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}
            }
            Rectangle {
                id : centerLeftRectId
                width: 100;height: width
                color: "beige"
                Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
            }
        }
    }
}
```

Inside our Window element, we create a Grid element that contains 4 rectangles and we set the columns property of our Grid to 3 to force the Grid to have 3 columns. The Grid element will take however many rectangles we put inside, and organize them, row by row, in 3 columns. This is best seen when you run the application. So give it a go!

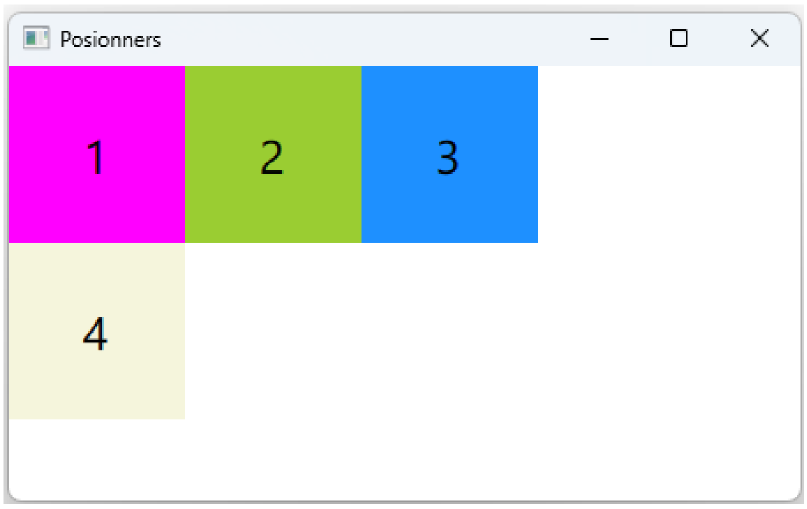


Figure 103. Four rectangles in a Grid element

We have rectangles "1", "2" and "3" in the first row and rectangle "4" is forced to the second row. Can you see why? The guilty line is `columns: 3`. We instructed the Grid element to organize its content into 3 columns, so it'll start from the top left corner and position rectangles "1", "2" and "3", that'll fill up three columns in the first row and it'll jump to the next row and position whatever rectangles are remaining. If we wanted the rectangles to make a square, we could change the columns property to `columns: 2`. With that in place, the first two rectangles will be on the first row and the remaining two will be on the second row.

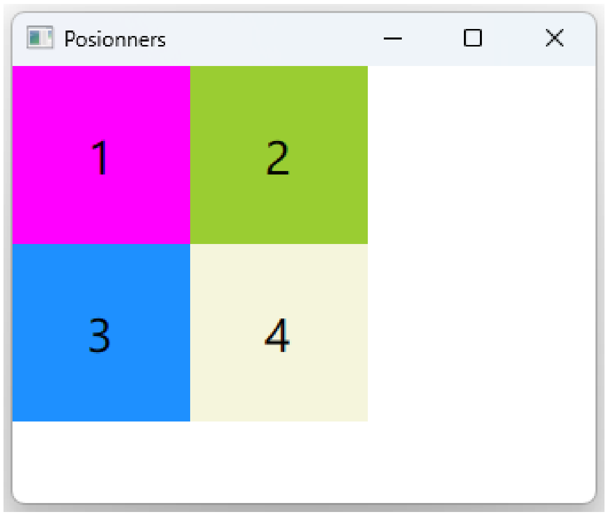


Figure 104. Grid with 2 columns

As an exercise, I would challenge you to recreate the 9x9 grid of rectangles we made with anchors, but using the Grid positioner. Grid is just one of the positioners offered by QML though. We also have Row and Column. Change the code in main.qml to be like below

```
//main.qml
//Row Positioner
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Positionners")
    Row {
        Rectangle {
            id : topLeftRectId
            width: 100;height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100;height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100;height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}

        }
        Rectangle {
            id : centerLeftRectId
            width: 100;height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

Notice that the only change we made was to replace Grid with Row and take out the columns property. Run the application and you should see the rectangles laid out from left to right.

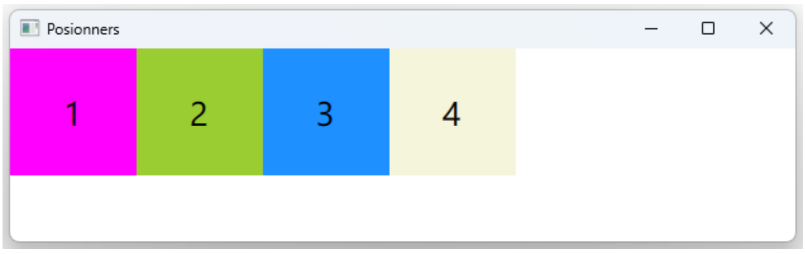


Figure 105. Rectangles in a Row Positioner

The Row positioner does just that. It arranges visual elements inside from left to right in a **row**. Of course, you can read more about it from the official documentation. We also have a Column positioner that lays its content out from top to bottom. Let's see it in action. Change the content of your main.qml file to be like below

```
//main.qml
//Column Positioner
import QtQuick
```

```
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Column {
        Rectangle {
            id : topLeftRectId
            width: 100
            height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100
            height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100
            height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}

        }
        Rectangle {
            id : centerLeftRectId
            width: 100
            height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

Run the application with these changes applied and you'll see your rectangles organized from top to bottom.

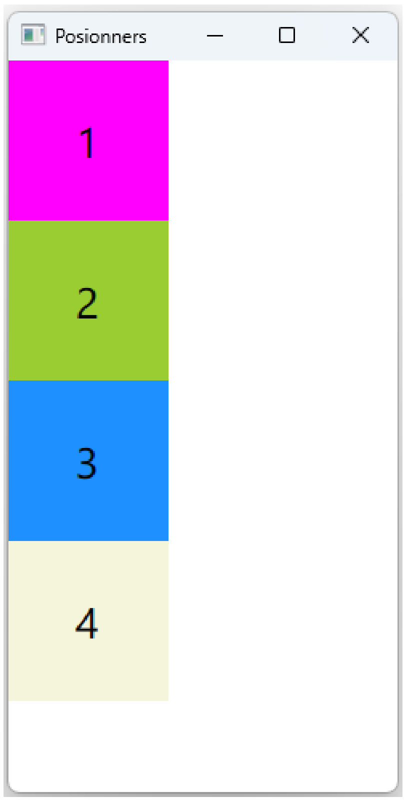


Figure 106. Rectangles in a Column Positioner

Row spacing and Column spacing

It's encouraged to check the documentation for the properties offered by these positioners. Lets turn back to the Grid element and apply some horizantal spacing between rectangles.

```
//main.qml
//Grid: spacing the content out
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Grid {
        columns : 2
        rowSpacing: 10 // Space the rows out by 10 px.
        Rectangle {
            id : topLeftRectId
            width: 100
            height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100
            height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100
            height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}

        }
        Rectangle {
            id : centerLeftRectId
            width: 100
            height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

The `rowSpacing: 10` statement adds a 10 px space between rows. Run the app with the changes applied and you should see the spacing applied.

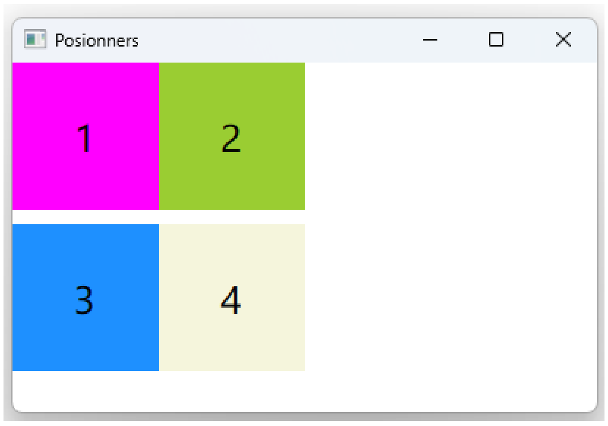


Figure 107. Grid row spacing

You can also add a vertical spacing.

```
//main.qml
//Grid. Vertical spacing
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Grid {
        columns : 2
        rowSpacing: 10
        columnSpacing: 10

        Rectangle {
            id : topLeftRectId
            width: 100;height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100;height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100;height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : centerLeftRectId
            width: 100;height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

Run the application with these changes, and you should see the columns spaced out by 10 px.

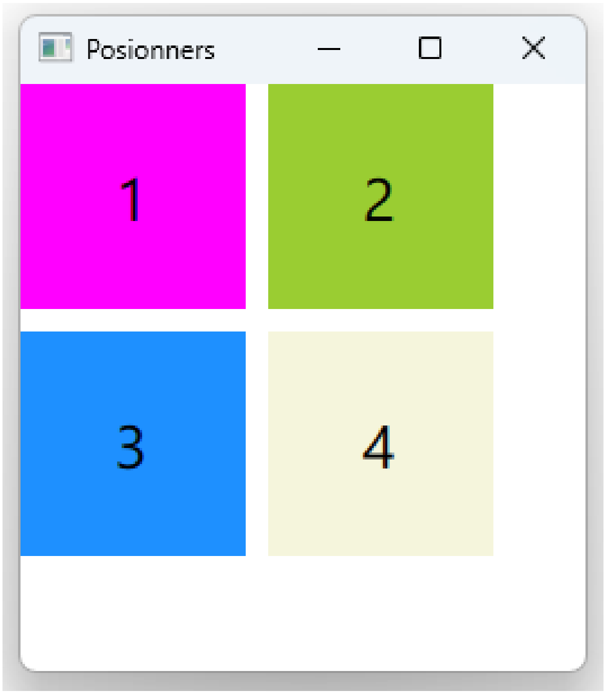


Figure 108. Grid column spacing

There is a shorthand `spacing` property that adds both rowSpacing and columnSpacing at the same time. Take out the lines

```
rowSpacing: 10
columnSpacing: 10
```

and replace them with

```
spacing: 10
```

Run the application with these changes applied and you should see the same thing as earlier when we set rowSpacing and columnSpacing individually.

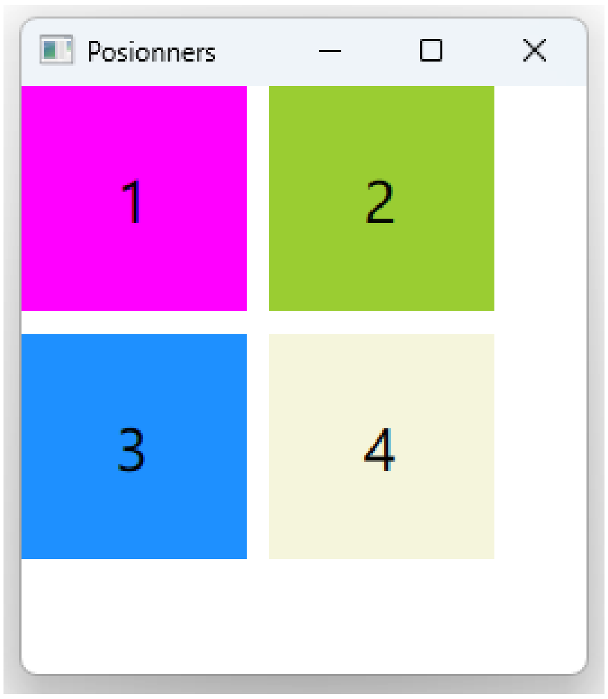


Figure 109. Grid spacing property

Aligning Items

We can also control how things items are aligned within grid cells. To see this in action change the `topLeftRectId` rectangle to have a width and height of 60.

```
//main.qml
//topLeftRectId has a smaller size
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Grid {
        columns : 2
        spacing: 10

        Rectangle {
            id : topLeftRectId
            width: 60;height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100;height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100;height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : centerLeftRectId
            width: 100;height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

Our grid has three squares with a side of 100 and one with a side of 60. What do you expect to see when you run the app? The best way to see if your expectations are right is to run the app. Give it a go!

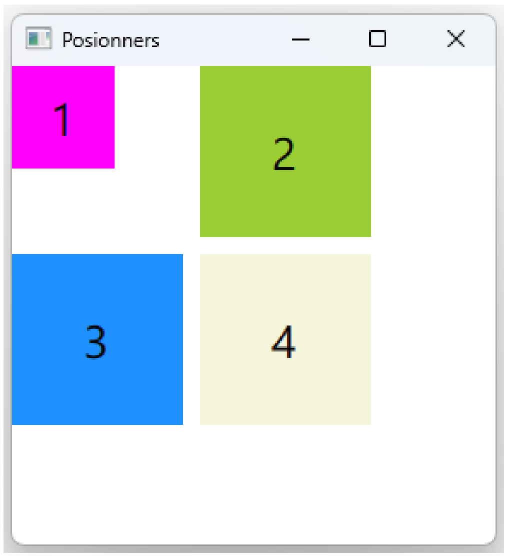


Figure 110. Left Aligned by Default

Because in each row, the largest item has a side of 100, each cell will be set up to have a size of 100 on each side. Rectangles "2","3" and "4" will fully fit in the allocated cell space, but "1" is much smaller in size. The million-dollar question pops up then. Where, in the 100x100 cell do we put our 60x60 rectangle? This is controlled by the alignment properties, and content is aligned in the top-left corner by default as we see with our rectangle labeled "1". You can change this using the `horizontalItemAlignment` and `verticalItemAlignment` properties. If we wanted content in our Grid to be aligned in the top-right corner, all we need to do is change our code like below

```
//main.qml
//Align grid content in the top-right corner
import QtQuick

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Grid {
        columns : 2
        spacing: 10
        //Align right horizontally. Keep the default vertically
        horizontalItemAlignment: Grid.AlignRight

        Rectangle {
            id : topLeftRectId
            width: 60
            height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100
            height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100
            height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}

        }
        Rectangle {
            id : centerLeftRectId
            width: 100
            height: width
```

```
                color: "beige"
                Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}

            }
        }
    }
```

The `horizontalItemAlignment: Grid.AlignRight` statement specifies that we want content aligned to the right of the cell horizontally. Since we don't set the vertical alignment vertically, the default will be kept which is to the top. Run the application with the changes applied and you should see rectangle "1" aligned to the top-right of the cell.

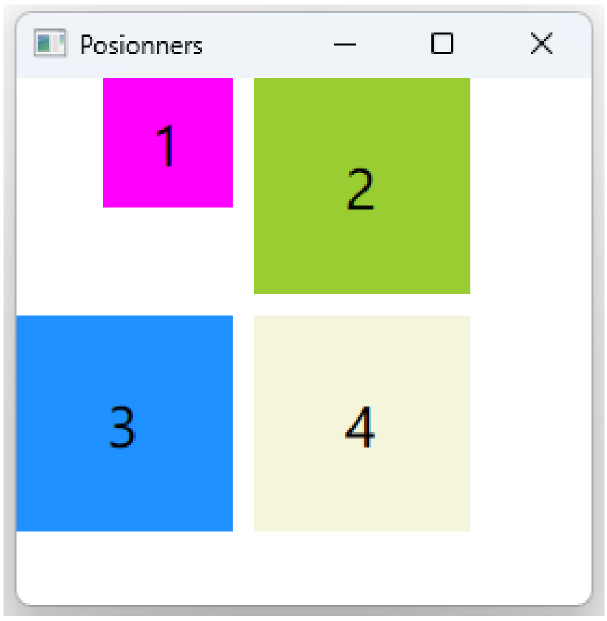


Figure 111. Align in top-right

We also have a verticalAlignment property. Reading the docs about it, you can see that the possible values are `Grid.AlignTop`, `Grid.AlignBottom` and `Grid.AlignVCenter`. Let's align content in the vertical center of the cell. Just because we can!

```
//main.qml
//Align grind content in the vertical center of the cell
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Grid {
        columns : 2
        spacing: 10

        horizontalItemAlignment: Grid.AlignRight
        //Align content in vertical-center of the cell
        verticalItemAlignment: Grid.AlignVCenter
        Rectangle {
            id : topLeftRectId
            width: 60;height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100;height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100;height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : centerLeftRectId
            width: 100;height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

`verticalItemAlignment: Grid.AlignVCenter` is the most important statement here, forcing our Grid content to be aligned vertically in the center of the cell. Run the application with the changes in place and you should see the results below.

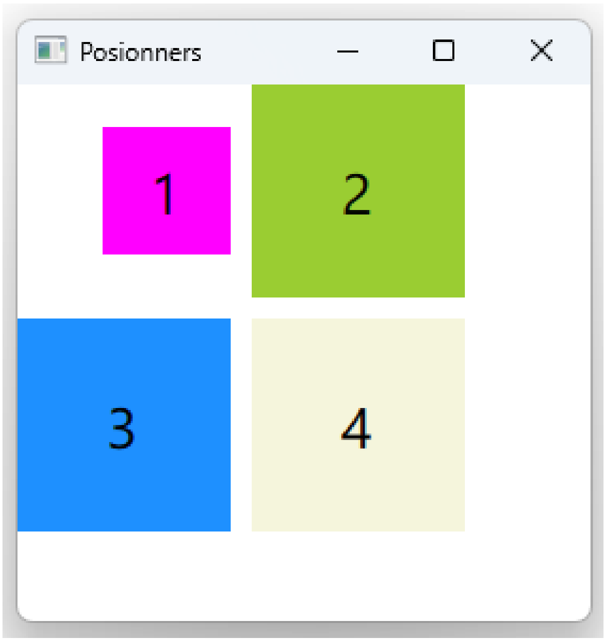


Figure 112. Align in vertical center of the cell

You are encouraged different values for horizontalItemAlignment and verticalAlignment to really drill this in your brain.

LayoutMirroring

Let's take this chance and introduce you to the LayoutMirroring attached property. Before we even do anything with it, I encourage you to check it out in the docs yourself. As the docs page states, it is used *to horizontally mirror Item anchors, positioner types (such as Row and Grid) and views (such as GridView and horizontal ListView)*. We use the Grid element to do a couple of things so far, so let's mirror our layout and see what happens. Change your main.qml file to contain code like below.

```
//main.qml
//LayoutMirroring
import QtQuick
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Posionners")

    Grid {
        columns : 2
        spacing: 10
        horizontalItemAlignment: Grid.AlignRight
        verticalItemAlignment: Grid.AlignVCenter

        //LayoutMirroring attached property
        LayoutMirroring.enabled: true
        LayoutMirroring.childrenInherit: true

        Rectangle {
            id : topLeftRectId
            width: 60;height: width
            color: "magenta"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topCenterRectId
            width: 100;height: width
            color: "yellowgreen"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100;height: width
            color: "dodgerblue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : centerLeftRectId
            width: 100;height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

The most important part being

```
//LayoutMirroring attached property
LayoutMirroring.enabled: true
LayoutMirroring.childrenInherit: true
```

We are enabling mirroring for the Grid element and forcing all children of the grid to inherit this mirroring setting. Based on what you read from the docs page of LayoutMirroring, read it if you haven't done so yet, what do you think we'll see if we run the application with these changes applied? Run the app and see for yourself.

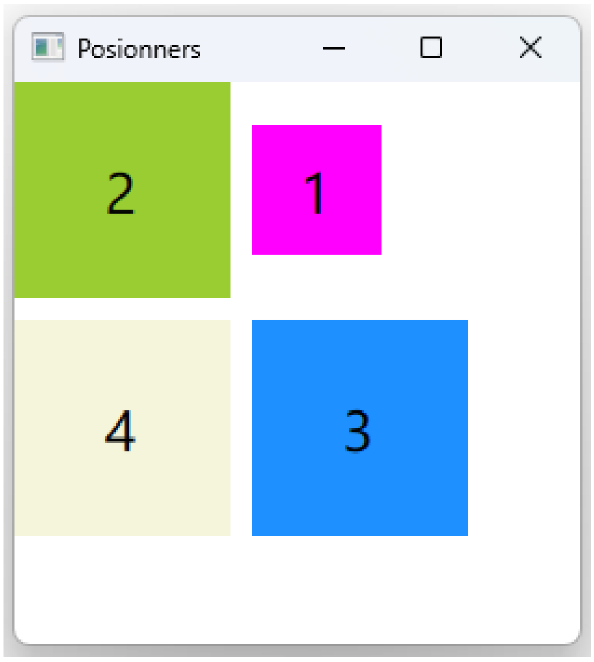


Figure 113. LayoutMirroring

You can see that things are mirrored horizontally relative to the left side of the window in this case. The rectangles were going from left to right before and now they're going from right to left. If you need this kind of thing in your QML application, you know where to look. We just scratched the surface on these positioners but hacking through some code and seeing the visual effects of the changes you make to your code, as we did throughout this section, should give you a good grasp on using them to get the look you may be after in your QML projects. As much as I'd like to help, I can't explore every single property for QML elements we touch on in different chapters. The docs are your best tool to find things on your own, and faster. Learn to use them. Using the documentation effectively is a superpower!

Layouts

On top of positioner items like Row, Column and Grid, QML also offers layout items. The advantage of these items is that they resize the user interface when the parent window is resized. Their exhaustive list is available from the official docs, but in this section, we'll play with `RowLayout`, `ColumnLayout` and `GridLayout`. We will also take the chance and compare them to what we already know about Row, Column and Grid positioners. Let's explore this through a live example. Create a brand new Qt Quick project from Qt Creator and change your main.qml file to contain code like shown below.

```
//main.qml
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3

        Rectangle {
            id : topLeftRectId
            width: 70;height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}

        }
        Rectangle {
            id : topCenterRectId
            width: 100;height: width
            color: "red"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : topRightRectId
            width: 100;height: width
            color: "blue"
            Text{text: "3"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : centerLeftRectId
            width: 100;height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : centerCenterRectId
            width: 100;height: width
            color: "pink"
            Text{text: "5"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : centerRightRectId
            width: 100;height: width
            color: "yellow"
            Text{text: "6"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : bottomLeftRectId
            width: 100;height: width
            color: "magenta"
            Text{text: "7"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : bottomCenterRectId
            width: 100;height: width
            color: "yellowgreen"
            Text{text: "8"; anchors.centerIn: parent;font.pointSize: 20}
        }
        Rectangle {
            id : bottomRightRectId
            width: 100;height: width
            color: "dodgerblue"
            Text{text: "9"; anchors.centerIn: parent;font.pointSize: 20}
        }
    }
}
```

Notice the statement `import QtQuick.Layouts` whose job is to import the Layouts module that contains the elements we're interested in. Inside the Window element, we have a GridLayout element that contains 9 rectangles labeled "1", "2" all the way to "9". The rectangles are also colored to make them easy to spot. Notice that except for putting the rectangles inside a GridLayout element to specifying the columns count as 3, all the positioning is taken care of the GridLayout element, much like the Grid positioner we've learned about.

GridLayout can resize if the parent window resizes. To make the changes in size automatically propagate to the Window element, we bind the width and height of the window to the implicit size of the GridLayout element.

```
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")
```

```
        GridLayout{
            id : mGridLayoutId
        }
    }
```

The implicit size is the size we get after taking into account the content of the GridLayout element. With this in place, if we add something to the layout, the container window will resize to accommodate the newly added item. The rest is just like what we saw with the Grid element: a bunch of rectangles with labels and different colors. If you run the application with these changes in place, you should see something like below.

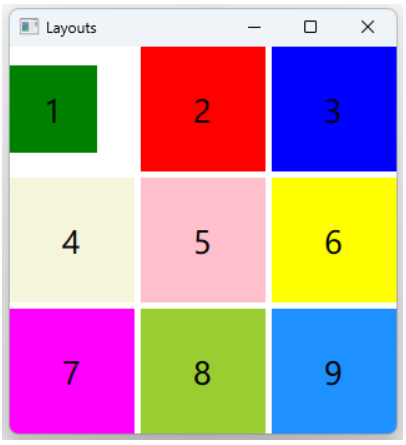


Figure 114. GridLayout Default Size

You can see that the container window perfectly wraps around the content of the grid layout. This is because of the binding we did to map the implicit size of the GridLayout element to the size of the window. Try to resize the window and you'll see that the content of the GridLayout element will also resize, trying to fill the available space.

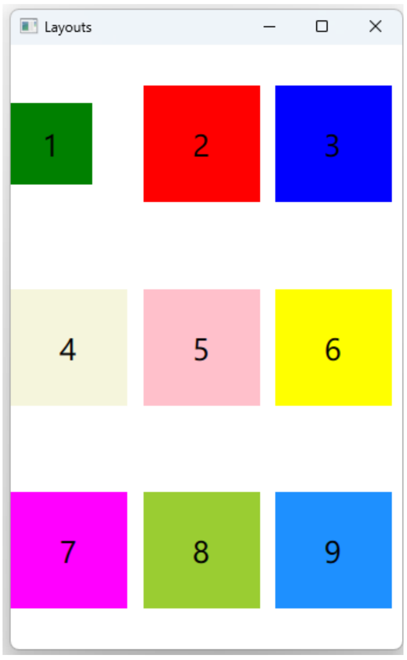


Figure 115. GridLayout Resizing

What is happening here is that the Window element is changing its size, the GridLayout element is noticing new empty space and the response is to try and fill it up as much as possible, leaving empty spaces between rectangles. The effect is the implicit size of the GridLayout element changes, taking into account the spaces between the rectangles and that implicit size is propagated back to the window because of the property binding we have set up for the width and height of the window. This is something you can take advantage of if you need this kind of resizing for your QML graphical user interfaces. Try to compare this application to the one we did with the Grid element and notice the differences. We can also play with RowLayout which aligns our rectangles from left to right by default. Replace GridLayout with RowLayout and take out the column property as shown below

```
//main.qml
//Using RowLayout
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    RowLayout{
        id : mGridLayoutId
        anchors.fill: parent
        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        //The rest of the rectangles omitted for space reasons
    }
}
```

If you run the application with these changes in place, you should see the rectangles aligned in a row, again, with the container window element perfectly wrapping around the content.

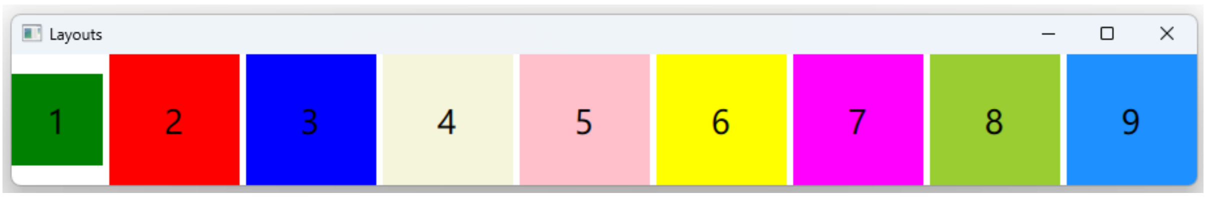


Figure 116. RowLayout Default Size

You can try to resize the window and you'll see the content trying to fill up the available space, with empty space added between the rectangles. Replacing RowLayout in the code with ColumnLayout will lay the rectangles out vertically, from top to bottom.

```
//main.qml
//Using ColumnLayout
```

```
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    ColumnLayout{
        id : mGridLayoutId
        anchors.fill: parent
        Rectangle {
            id : topLeftRectId
            width: 70;height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        //The rest of the rectangles omitted for space reasons
    }
}
```

Run the code and you'll see something like below

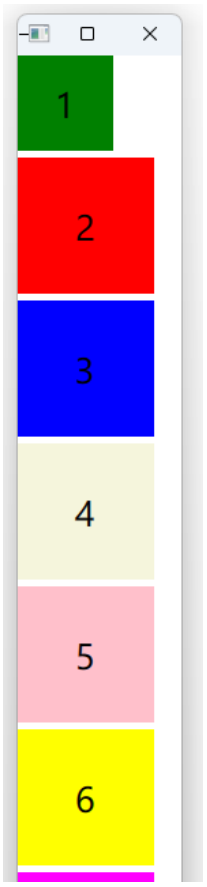


Figure 117. ColumnLayout

If you try to resize the window horizontally, you'll see that the ColumnLayout element won't try to resize to fill the new horizontal space. This is just how ColumnLayout is wired. Let's bring our code back to using the GridLayout element and re-enable to columns property.

```
//main.qml
//Using GridLayout
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3
        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
        }
        //The rest of the rectangles omitted for space reasons
    }
}
```

Running the app should give us something like below

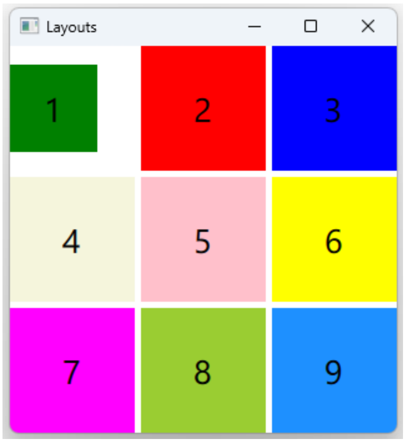


Figure 118. GridLayout

Notice that the green rectangle, labeled "1" is smaller than others and is aligned to the left of the cell. We can change this using the `Layout.alignment` attached property inside our Rectangle element of interest. Change the code to add this property

```
//main.qml
//Layout.alignment attached property
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3
        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
            Layout.alignment: Qt.AlignRight
        }
        //The rest of the rectangles omitted for space reasons
    }
}
```

Run the application with these changes applied, and you should see the "1" green rectangle pushed to the right of the cell.

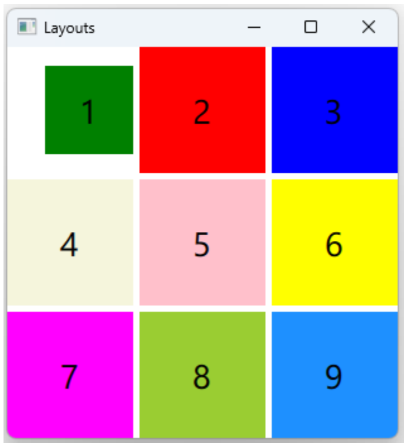


Figure 119. Layout Alignment Attached Property

Looking at the docs page for the Layout.alignment attached property, you can see that we can even combine these alignments using the bitwise OR operator. For example, if we wanted the small "1" green rectangle pushed to the top-right of the cell, we can change the alignment to be like in the code below.

```
//main.qml
//Layout.alignment  combined values
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3
        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
            Layout.alignment: Qt.AlignRight|Qt.AlignTop
        }
        //The rest of the rectangles omitted for space reasons
    }
}
```

Running the app with the changes applied, you should see our "1" green rectangle pushed to the top-right corner of the cell.

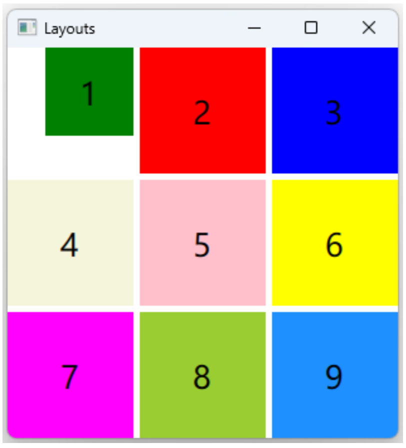


Figure 120. Rectangle Aligned to Top-Right

The exhaustive list of possible values for these alignments is shown on the official docs page.

Despite the "1" green rectangle being small, we can instruct it to fill the available cell space. We can do that through the `Layout.fillWidth` and `Layout.fillHeight` attached properties. These properties take `true/false` values. Since in this case we want the rectangle to fill the space, we'll set them to true.

```
//main.qml
//Layout.fillWidth and Layout.fillHeight
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3
        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
            Layout.alignment: Qt.AlignRight|Qt.AlignTop
            Layout.fillWidth: true
            Layout.fillHeight: true
        }
        //The rest of the rectangles omitted for space reasons
    }
}
```

If you run the application with these changes in place, you should see the "1" green rectangle filling the cell space.

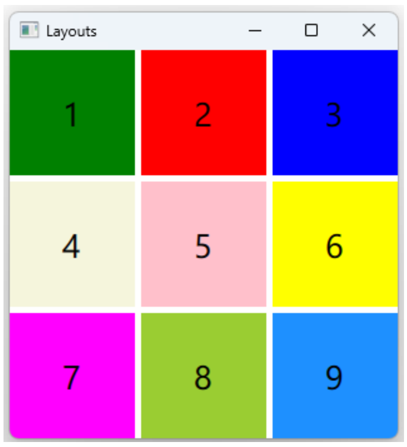


Figure 121. Layout fillWidth and Layout fillHeight

Of course, at this point the `Layout.alignment` thing is not helping out, because the rectangle is filling the cell anyway, so you cat take it out if you want. But there is a problem: if you resize the window, you'll notice that the "1" green rectangle is greedy and pushing other rectangles around, trying to take up as much space as possible.

This is happening because we are telling rectangle "1" to take up as much space as possible but other rectangles just keep their default assigned size by default. If there is new space available, the layout just adds empty space around them, giving us the effect you see in the figure below.

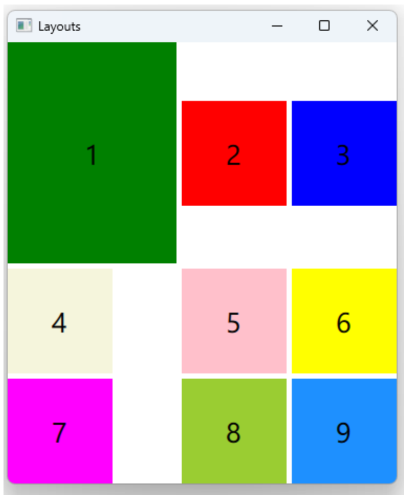


Figure 122. Rectangle 1 Getting Greedy

We can fix this by also instructing other rectangles to fill up the available width and height space.

```
//main.qml
//Tell other rectangles to also fill up the space
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3
        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
            Layout.alignment: Qt.AlignRight|Qt.AlignTop
            Layout.fillWidth: true
            Layout.fillHeight: true
        }

        Rectangle {
            id : topCenterRectId
            width: 100
            height: width
            color: "red"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}
            //Add this to all other rectangles in the GridLayout element.
            Layout.fillWidth: true
            Layout.fillHeight: true
        }
        //The rest of the rectangles omitted for space reasons
    }
}
```

It’s important to note that we’ve only shown code for only two rectangles here for space reasons. If you’re trying out the code in Qt Creator, you should apply the changes to all the rectangles. The final code for the section is always shared in the git repository of the book, and you can use that as a reference if you lose context for what we’re doing here. Run the code with the changes in place and you should see all the rectangles uniformly resizing to fill up the space when you resize the window.

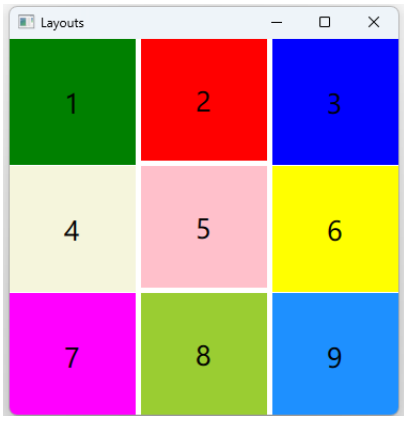


Figure 123. Rectangles Resizing Uniformly

We can also use the Layout.maximumWidth and Layout.maximumHeight attached properties to control the maximum size of the rectangle when resizing to fill up the available space. If the maximum is reached, the rectangle will stop resizing and empty white spaces will be left around the rectangle. Change your code to be like below.

```
//main.qml
//Tell other rectangles to also fill up the space
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3
        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}
```

```
                Layout.alignment: Qt.AlignRight|Qt.AlignTop
                Layout.fillWidth: true
                Layout.fillHeight: true
                Layout.maximumWidth: 150
                Layout.maximumHeight: 150
            }
            //The rest of the rectangles omitted for space reasons
        }
    }
```

Run the application with these changes in place. Resize the window and you'll see the rectangles expanding trying to fill up the space, but at some point, the "1" green rectangle will stop growing and stop at the 150x150 size, the rest of the cell will be just empty white space, and the rectangle will align in the top-right corner of the cell because of the `Layout.alignment: Qt.AlignRight|Qt.AlignTop` statement we still have in our code.

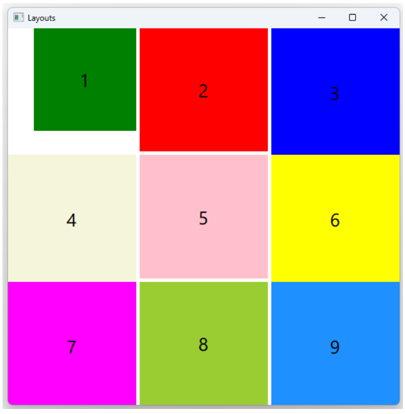


Figure 124. Maximum Width and Height

The next thing we'll cover about GridLayout is that you can instruct a cell to span several rows or columns. Taking our current grid as an example we can instruct **rectangle "2" to span two columns and fill up the space occupied by rectangle "3"** and instruct rectangle "4" to span two rows and fill up the space occupied by rectangle "7".

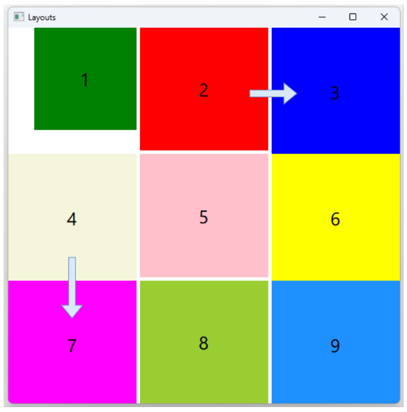


Figure 125. Rowspan and Columnspan Concepts

Of course for that to work, we'll have to take out rectangles "3" and "7" because we would get conflicting things otherwise. Change your code to be like below

```
//main.qml
```

```
//Row span and column span
import QtQuick
import QtQuick.Layouts
Window {
    visible: true
    width: mGridLayoutId.implicitWidth
    height: mGridLayoutId.implicitHeight
    title: qsTr("Layouts")

    GridLayout{
        id : mGridLayoutId
        anchors.fill: parent
        columns: 3

        Rectangle {
            id : topLeftRectId
            width: 70
            height: width
            color: "green"
            Text{text: "1"; anchors.centerIn: parent;font.pointSize: 20}

            Layout.alignment: Qt.AlignRight|Qt.AlignTop
            Layout.fillWidth: true
            Layout.fillHeight: true
            Layout.maximumWidth: 150
            Layout.maximumHeight: 150
        }
        Rectangle {
            id : topCenterRectId
            width: 100
            height: width
            color: "red"
            Text{text: "2"; anchors.centerIn: parent;font.pointSize: 20}

            Layout.fillWidth: true
            Layout.fillHeight: true
            Layout.columnSpan: 2
        }

        //Rectangle "3" taken out . Its space will be taken by Rectangle "2"

        Rectangle {
            id : centerLeftRectId
            width: 100
            height: width
            color: "beige"
            Text{text: "4"; anchors.centerIn: parent;font.pointSize: 20}

            Layout.fillWidth: true
            Layout.fillHeight: true
            Layout.rowSpan: 2
        }
        Rectangle {
            id : centerCenterRectId
            width: 100
            height: width
            color: "pink"
            Text{text: "5"; anchors.centerIn: parent;font.pointSize: 20}

            Layout.fillWidth: true
            Layout.fillHeight: true

        }
        Rectangle {
            id : centerRightRectId
            width: 100
            height: width
            color: "yellow"
            Text{text: "6"; anchors.centerIn: parent;font.pointSize: 20}

            Layout.fillWidth: true
            Layout.fillHeight: true

        }

        //Rectangle "7" Taken out. Its space will be taken by Rectangle 4

        Rectangle {
            id : bottomCenterRectId
            width: 100
            height: width
            color: "yellowgreen"
            Text{text: "8"; anchors.centerIn: parent;font.pointSize: 20}

            Layout.fillWidth: true
            Layout.fillHeight: true

        }
        Rectangle {
            id : bottomRightRectId
            width: 100
            height: width
            color: "dodgerblue"
            Text{text: "9"; anchors.centerIn: parent;font.pointSize: 20}

            Layout.fillWidth: true
            Layout.fillHeight: true
        }
    }
}
```

The full code is shown here for context. We instruct rectangle "2" to span two columns through the statement `Layout.columnSpan: 2` and instruct rectangle "4" to span two rows through the statement `Layout.rowSpan: 2`. What you have to be careful about is that the space you span into is free. We made sure of that by taking out rectangles "3" and "7". Run the application with these changes in place, and you should have a graphical user interface like below.

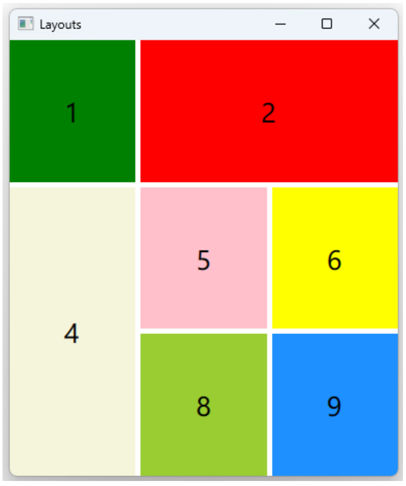


Figure 126. Rowspan and Columnspan

I know this has been a long section, but one last thing and we'll call it a day. The GridLayout element has a layoutDirection property you can use to control whether content flows from left to right or from right to left. The default is from left to right, but we can change that. To play with this, just the code below

```
layoutDirection: Qt.RightToLeft
```

inside the GridLayout element, just under the columns property and run the application. You'll see that the content now goes from right to left.

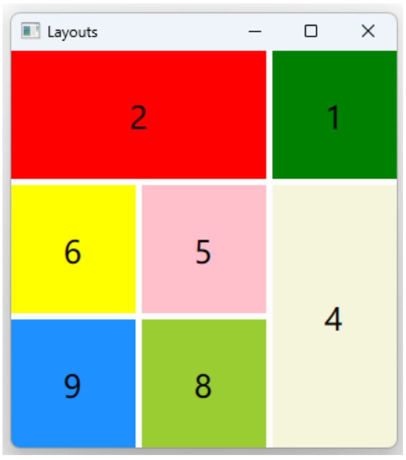


Figure 127. Layout Direction

Of course, you can explore more on these elements and properties yourself in the official documentation. The goal here was to get your feet wet through practical examples for you to have an intuition on using these layout elements.

Flow

The Flow element lays out its content like text flows on a page. Things are first arranged on a row and when the row fills up, content wraps on the next row until we run out of content. Let's see it in action. Create a brand new Qt Quick project from Qt Creator and change your main.qml file to contain code like below.

```
//main.qml
//Flow element
import QtQuick
import QtQuick.Window

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Flow")

    Flow {
        id : containerFlowId
        width : parent.width
        height: parent.height

        Rectangle {
            id : topLeftRectId
            width : 70;height: 70
            color: "green"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "1"
            }
        }
        Rectangle {
            id : topCenterRectId
            width : 100;height: 100
            color: "beige"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "2"
            }
        }
        Rectangle {
            id : topRightRectId
            width : 100;height: 100
            color: "dodgerblue"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "3"
            }
        }
        Rectangle {
            id : leftCenterRectId
            width : 100;height: 100
            color: "magenta"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "4"
            }
        }
        Rectangle {
            id : centerRectId
            width : 100;height: 100
            color: "red"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "5"
            }
        }
        Rectangle {
            id : rightCenterId
            width : 100;height: 100
            color: "yellow"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "6"
            }
        }
        Rectangle {
            id : bottomLeftRectId
            width : 100;height: 100
            color: "royalblue"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "7"
            }
        }
        Rectangle {
            id : bottomCenterRect
            width : 100;height: 100
            color: "greenyellow"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "8"
            }
        }
        Rectangle {
            id : bottomRightRectId
            width : 100;height: 100
            color: "blue"
            Text {
                anchors.centerIn: parent
                color :"black"
                font.pointSize: 30
                text : "9"
            }
        }
    }
}
```

The Flow element inside our window contains 9 labeled rectangles and has the same size as the window. The rectangles inside will flow in the window like text flows on a page, fist filling the available width and then wrapping to the next line when we run out of horizontal space. From this you can pick up the idea that the default flow is left-to-right and top-to-bottom. If you run the application with these changes in place, you'll see something like below.

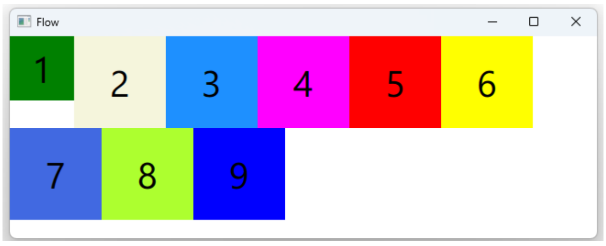


Figure 128. Flow element

We have rectangles "1" through "6" in the first row and the rest wrap to the next line. The first 6 rectangles have a width of 560. If we added another one on the first row, the width would be 660, which exceeds the 640 width we have in the window: so rectangle "7" is pushed to the next row. Try to resize the window up and down in width and you'll see that the rectangles inside the Flow element will re-arrange themselves to use as much space as possible on the first row. If we run out space in the first row, they'll wrap to the next line. If we resize down enough, we can even get elements to make a 9x9 grid of rectangles.

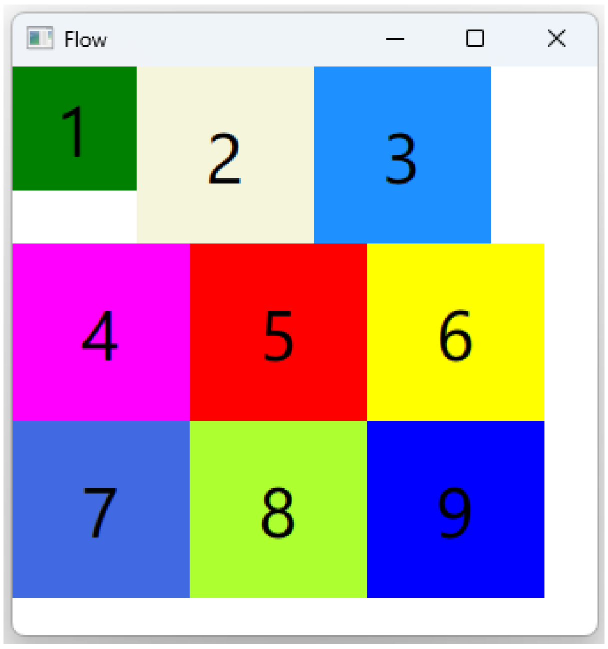


Figure 129. 9x9 Grid in Flow Element

The Flow element also has properties you can use to manipulate its behavior. For example, we can use the `flow` property to control whether the content flows from left to right or from top to bottom. If you add the

```
flow : Flow.TopToBottom
```

statement just below the height property of the Flow element in our current project and run the application, you'll see that the rectangles will flow from top to bottom inside the window. When we run out of vertical space, we wrap to the next column.

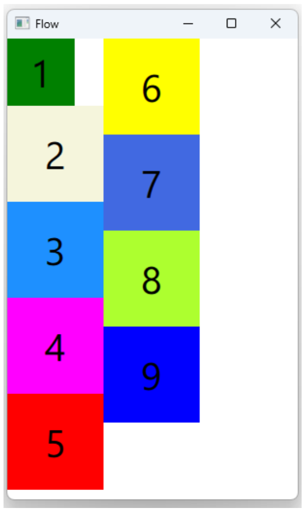


Figure 130. The flow Property

We also have a `layoutDirection` property that holds the layout direction of the layout. Rephrasing the official docs, possible values are:

- `Qt.LeftToRight (default)` - Items are positioned from the top to bottom, and left to right. The flow direction is dependent on the Flow::flow property.
- `Qt.RightToLeft` - Items are positioned from the top to bottom, and right to left. The flow direction

is dependent on the Flow::flow property.

To play with this property, add the following statement

```
layoutDirection: Qt.RightToLeft
```

just below the flow property in our current project. If you run the code with the change applied, you should see the rectangles still flowing from top to bottom but they also go from right to left this time.

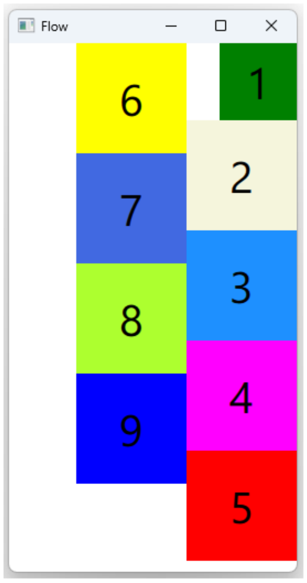


Figure 131. The layoutDirection Property

You can play with different combinations of these properties to see what you end up with. The Flow element also has a spacing property you can use to add some space between the rectangles. I'll leave you to play with that in the current project. The main goal of the chapter was to introduce you to different positioning mechanisms in Qt. We started out by looking at anchors together with their margins and offsets. We moved on and looked at raw positioner elements like Row, Column and Grid. We saw that QML also offers a collection of layout items you can use if you care about resizing the content when the container window grows, and we closed off by looking at the Flow element you can use if you want your content to flow as text does on a page. These are just tools in your arsenal, use what makes sense for the problem at hand.

Chapter Summary

QML provides a rich set of positioning mechanisms one can use to layout out elements in the windows. No one is generally better than the other, and in practice, you'll use some combination of a few of them. Which one you say? It depends on the situation at hand. Over time, you'll develop an intuition for what works best for you. The chapter started out exploring anchors and used them to position a bunch of rectangles. We could use margins and offsets to add breathing space around our positioned rectangles. For margins and offsets to work, you need to previously have an anchor set on the same anchor line. We also got a chance to see that anchors work against an item with which you have a parent or sibling relationship. Next up we learned about Row, Column and Grid positioners. These do what their name says. They allow to lay content out horizontally, vertically and in a grid structure. They also have properties like padding that you use to further tune their behavior. QML also offers a set of layout elements you can use if care about resizing if the parent window is resized. We have RowLayout, ColumnLayout and GridLayout. With layouts, we could instruct each contained cell to fill the available width or height and we could even instruct a cell to span a number of rows and columns. We wrapped the chapter up looking at the Flow element one can use to lay content out in a window like text flows on a page. Of course there are a few properties one can use to further control the behavior, the exhaustive list of which can be found in the official documentation.