Introduction to Application Development with Qt Quick

Release 1.0

Digia, Qt Learning

February 28, 2013

Contents

CHAPTER 1

Introduction

1.1 Who should read this tutorial and why

This tutorial explains the basics of developing Qt Quick applications with the help of code walkthroughs of complete applications. The tutorial extends the standard Qt Quick documentation provided with Qt and relies on it as a source of more detailed information about Qt Quick APIs and fundamental concepts.

This tutorial is thought to precede in-depth content as a prerequisite for those of you who are new to Qt Quick. Even though it starts with the basics, you still have to be familiar with principles of programming. Some basic knowledge of JavaScript is highly recommended.

After completing this tutorial, you should be able to write your own Qt Quick applications and start discovering more by reading advanced materials or analyzing the code of complex applications.

1.2 The journey is the target

This tutorial starts with the traditional "Hello World" application and ends with a full-featured Qt Quick application that can be used on a daily basis. The features of this final application covers the major Qt Quick programming aspects.

Features are never enough

No doubt many more features could have been added, but it had to be finished at a certain level to keep the size of this tutorial in limits. Feel free to take the source code of the final application and extend it. You will know how to do it!

The tutorial is a journey through these aspects. We will stop at famous places, watch some remarkable scenery, and use the gained knowledge to enhance the application more and more. At each of those steps we will look into code samples, and discuss what is happening there and how we use it in our application. At the end, you might be surprised that the application is fairly simple compared to the length of the tutorial. This is on purpose: our target is not to develop the application quickly, but to learn through the journey.

Right after "Hello World", we're going to develop a simple application that shows the current time and date in the nostalgic format of an old LED clock. The next step is another application which reads, parses, and displays weather forecasts from the Internet. Later, we try to combine these two applications as components into a larger application: a clock with integrated weather forecasting.

Note that some advanced aspects are not discussed in this tutorial to keep it simple. Those aspects are mentioned in the last chapter. One piece of advice that we have for you is to remember the old developer saying: "Documentation is your friend"[1]...

1.3 Downloads

A .zip file that contains the source code of each chapter is provided so you can compare it with your code:

Source code[2]

This tutorial is available in the following formats:

PDF[3]

ePub[4]

Qt Help[5].

..note: The example application referred in this guide is developed with Qt 4.8.x and Qt Quick 1.1 runn

Porting to Qt5 (page 91) for details about how to get this example application running with Qt5.

1.4 Help us help you

We would greatly appreciate any feedback or comments from you that can help us improve the content of this guide.

Use the Qt Bug Tracker to give us feedback.

1.5 Related material

Other guides

[1]http://qt-project.org/doc/qt-4.8/index.html
[2]http://releases.qt-project.org/learning/developerguides/qtquickappdevintro/qt_quick_app_dev_intro_src.zip
[3]http://releases.qt-project.org/learning/developerguides/qtquickappdevintro/QtQuickAppDevIntro.pdf
[4]http://releases.qt-project.org/learning/developerguides/qtquickappdevintro/QtQuickAppDevIntro.epub
[5]http://releases.qt-project.org/learning/developerguides/qtquickappdevintro/QtQuickAppDevIntro.qch

As mentioned before, the following Qt Learning Guides for Qt Quick could be useful if you're developing for desktop and mobile devices on Symbian and MeeGo:

- Programming with Qt Quick for Symbian and MeeGo Harmattan Devices
- Qt Quick Application Developer Guide for Desktop

Check this link[6] to download these and other guides.

Qt documentation

There are two major sets of documents in Qt that we will be referring to. We recommend reading them to learn all the key details about Qt Quick:

- Qt Quick landing page in the Qt documentation[7]
- Introduction to the QML Language[8]

Training materials

You may consider looking into the training materials published on the Qt Training web page[9] . In addition to training slides, training materials contain a lot of useful examples.

*Videos**

The recordings of training sessions and technical talks given at Qt Developer Days are another interesting learning resource. They are available in the Qt's video collection[10] .

*Examples and demos**

Qt Quick comes with a wide range of demos and examples. You can access all of them either from the Qt Creator welcome page or from the Qt Quick Code Samples[11] page in Qt documentation.

The wiki on the Qt Project website[12] has two listings of examples and demos:

- Demos and Examples[13]
- Example Applications for Qt Quick[14]

1.6 License

[6]http://qt-project.org/wiki/Developer-Guides/
[7]http://qt-project.org/doc/qt-4.8/qtquick.html
[8]http://qt-project.org/doc/qt-4.8/qdeclarativeintroduction.html
[9]http://qt.digia.com/Product/Learning/Topics/QML-Qt-Quick/
[10]http://qt-project.org/videos
[11]http://qt-project.org/doc/qt-4.8/qdeclarativeexamples.html
[12]http://qt-project.org/
[13]http://qt-project.org/wiki/Category:Learning::Demos_and_Examples
[14]http://qt-project.org/wiki/qml_examples_directory

What's Next?

The next chapter covers how to set up the development environment and run your first Qt Quick application.

CHAPTER 2

Work Environment Setup

If this is your first Qt Quick project, it makes sense to take a look at the tools you need, how a new project can be started and what are the most important needs of the daily development workflow.

2.1 Installing the tools

We recommend using the latest Qt libraries for working with this tutorial. You can download the libraries for your platform on *www.qt-project.org/downloads*. If you already have a working Qt development environment, make sure that you use Qt version Qt 4.7.4 or 4.8.x, which include QtQuick 1.1.

2.2 Creating Qt Quick applications

In this tutorial we will be using QtCreator[1] as an IDE. You can download the latest Qt Creator IDE and configure it to use the Qt 4.x libraries. For more information about configuring Qt Creator, refer to *Adding Qt Versions <http://doc-snapshot.qt-project.org/qtcreator-2.6/creator-project-qmake.html>*.

The wizard under the menu *File* → *New File or Project* creates not only project files, but also some initial application code. The wizard provides two choices relevant to Qt Quick:

Qt Quick Application Qt Quick UI

The major difference between these two options is how the application code is executed. This difference reveals quite a few interesting facts of how Qt Quick works under the hood. Let's create a project of each type and take a look at the files generated by the wizard.

Qt Quick UI* project type

---

[1] http://qt-project.org/wiki/Category:Tools::QtCreator

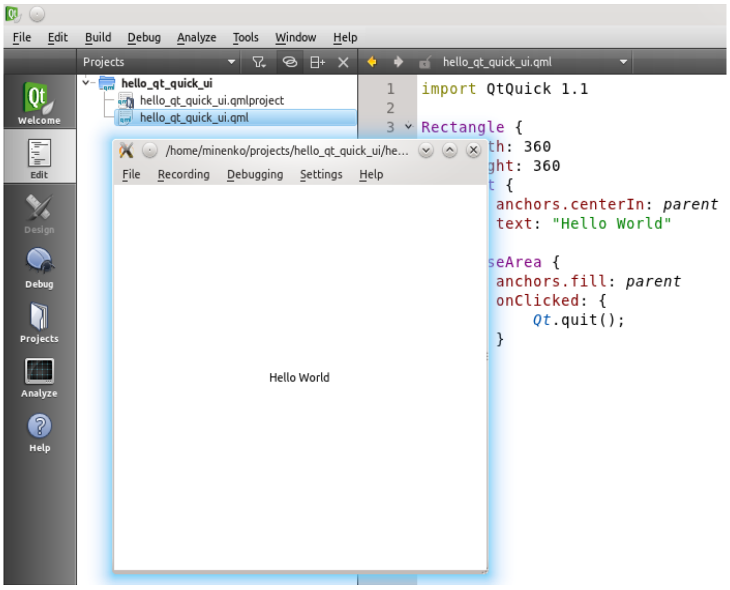
If you select Qt Quick UI*, let's name it `hello_qt_quick_ui`, you will get just one `qml` file and two project files in the project directory:

- `hello_qt_quick_ui.qml` - application code you start with
- `hello_qt_quick_ui.qmlproject` - the project file. You do not need to touch it for now
- `hello_qt_quick_ui.qmlproject.user` - your project settings. Automatically generated and changed by Qt Creator. You need not edit this file, nor check it in into your version control system

`hello_qt_quick_ui.qml` looks like this:

```
import QtQuick 1.1

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Hello World"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

You can execute it by pressing `CTRL-R` or by selecting *Build* → *Run* in Qt Creator. This shows the following on the screen:



```
import QtQuick 1.1

Rectangle {
th: 360
ght: 360
t {
 anchors.centerIn: parent
 text: "Hello World"

seArea {
 anchors.fill: parent
 onClicked: {
     Qt.quit();
 }
```

But wait! There is no compilation step! How does the application run?

Qt Quick is script-based, just like Python or Perl. A script needs to be interpreted and executed by an engine. This engine in Qt Quick is called Qt Declarative UI Runtime[2]. Qt Quick applications can use this engine in two different ways:

1. passing a `qml` file as a command-line option to the engine application, qmlviewer[3]
2. integrating Qt Declarative UI Runtime[4] in C++ code and loading the qml files

Our `hello_qt_quick_ui` uses the first way. The qmlviewer[5] application located in the `bin` folder of the Qt installation. It contains code which uses Qt Declarative UI Runtime[6] to load the `qml` file specified on the command line. In the *Qt Quick UI* projects, Qt Creator automatically runs qmlviewer[7] to load the main `qml` file when you press `CTRL-R` or select *Build* → *Run*. No compilation step is required.

The qmlviewer[8] application also provides debugging interfaces and many other cool goodies. Check its documentation page or call `qmlviewer --help` to see what is available.

This project type and the way it is integrated in Qt Creator is very simple and handy for discovering the functionality of Qt Quick. We use it most of the time. Nevertheless, in the next section we are going to learn about the *Qt Quick Application* project type to understand how you can use Qt Quick in a standard Qt application developed in C++.

Before we start with it, a short remark about the name *Qt Quick UI* given to a project type with `qml` files only. This underlines the major purpose of Qt Quick to serve as a script-based programming environment for application UI. Complex application logic and heavy processing should stay on the C++ side and expose its APIs to Qt Quick.

2.3 Qt Quick Application* project type

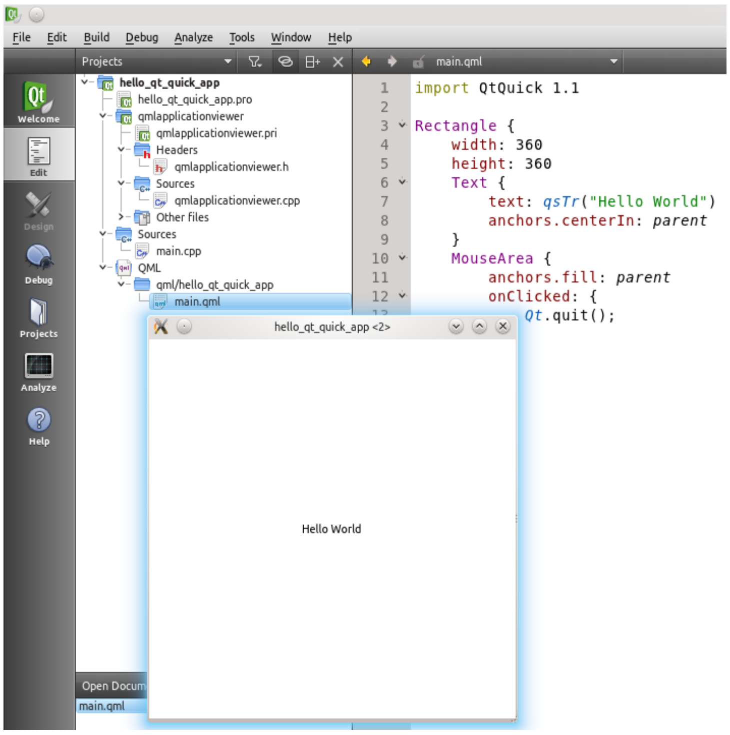
Another project type called *Qt Quick Application* allows you to unleash the power of Qt Quick. If you create a `hello_qt_quick_app` project in the wizard with this type, you get a Qt application in C++:

- `hello_qt_quick_app*.png` and `.svg` - desktop icons for different platforms
- `hello_qt_quick_app*.desktop` - desktop description files for different platforms
- `hello_qt_quick_app.pro` - Qt project file. You need not touch it for now.
- `hello_qt_quick_app.pro.user` - your local project settings. Automatically generated and changed by Qt Creator. You need not edit this file, nor check it in into your version control system
- `main.cpp` - the main file of your application starting your own `qmlviewer` implemented using the `QmlApplicationViewer` C++ class

[2]http://qt-project.org/doc/qt-4.8/qmlruntime.html
[3]http://qt-project.org/doc/qt-4.8/qmlviewer.html
[4]http://qt-project.org/doc/qt-4.8/qmlruntime.html
[5]http://qt-project.org/doc/qt-4.8/qmlviewer.html
[6]http://qt-project.org/doc/qt-4.8/qmlruntime.html
[7]http://qt-project.org/doc/qt-4.8/qmlviewer.html
[8]http://qt-project.org/doc/qt-4.8/qmlviewer.html

- `qml`- a folder where the *Hello World* `qml` file resides. Add other `qml` files here
- `qmlapplicationviewer` - a folder with implementation of the `QmlApplicationViewer` C++ class which initializes and starts Qt Declarative UI Runtime[9] with the main `qml` file

Your application now has its own qmlviewer-like module along with the same *Hello World* `qml` file. This is a Qt C++ application which provides the basic functionality of `qmlviewer` plus some additional code for integration on non-desktop platforms . If you run this project, Qt Creator compiles and builds your application like any other C++ application, and starts it. `QmlApplicationViewer` loads and executes Qt Quick code the same way as `qmlviewer`:



```
import QtQuick 1.1

Rectangle {
    width: 360
    height: 360
    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
```

Your new `QmlApplicationViewer` is another possible instance of Qt Declarative UI Runtime[10]. This way of handling Qt Quick applications opens many interesting possibilities for integrating Qt Quick code with classic C++ applications and even making C++ code available as new Qt Quick items. This is a more advanced technique beyond the scope of this tutorial. You can read more about this in Qt Declarative UI Runtime[11] documentation, as well as in the "Extending QML Functionality using C++"[12] article in the Qt documentation.

2.4 Tracing what is going on

In the course of this tutorial you might want to trace what your application code does and might get interested to stop the application to inspect it at run-time. Let's take a look at how

[9] http://qt-project.org/doc/qt-4.8/qmlruntime.html
[10] http://qt-project.org/doc/qt-4.8/qmlruntime.html
[11] http://qt-project.org/doc/qt-4.8/qmlruntime.html
[12] http://qt-project.org/doc/qt-4.8/qml-extending.html

debugging and tracing works in Qt Quick. Debugging is feature rich and is well documented in the Debugging Qt Quick Projects[13] article. When using it for the first time, you must set up the Debugging Helpers[14].

In your first steps with Qt Quick, you will mostly use tracing as it is a simple and easy way to follow what happens in an application. You can add tracing statements by using the `console.log()`, `console.debug()` or just `print()` methods provided by JavaScript.

For example, if we choose to trace the place where our Hello World* application quits upon a mouse click, the code would look like this:

(`helloqml/helloqml.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
/****************************************************************************
**
** Copyright (C) 2012 Digia Plc and/or its subsidiary(-ies).
** Contact: http://www.qt-project.org/legal
**
** $QT_BEGIN_LICENSE:BSD$
** You may use this file under the terms of the BSD license as follows:
**
** "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 Digia Plc and its Subsidiary(-ies) nor the names
**     of its 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
** "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
** LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
** A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
** OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
** SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
** LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
** DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
** THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
** (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
** OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."
**
** $QT_END_LICENSE$
**
****************************************************************************/
import QtQuick 1.1

Rectangle {
```

[13] http://qt-project.org/doc/qtcreator-2.6/creator-debugging-qml.html
[14] http://qt-project.org/doc/qtcreator-2.6/creator-debugging-helpers.html

```
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Hello World"
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log("I was \"" + parent + "\"!")
            console.log("Bye for now!")
            Qt.quit();
        }
    }
}
```

This code produces the following in the `QML Viewer` tab in Qt Creator:

```
I was "QDeclarativeRectangle(0x23705d0)"!
Bye for now!
```

What's Next?

By now you should have a working development environment and a simple project with a "Hello World" application which you can manipulate, run and inspect how it works. In the next chapter, we are going to discuss some of the core concepts of Qt Quick and take a look at what is available to do more than a simple *Hello World* example.

CHAPTER 3

Qt Quick Core Principles for Application Development

3.1 Qt Quick Compared to Classical Qt

Being a part of Qt, Qt Quick (or Qt User Interface Creation Kit*) provides a totally new way of creating UIs and even programming in general. Most use cases for Qt Quick are in areas where users expect non-widget based UIs with rich animations, effects and graphical resources. It allows turning virtually any graphical element into an interactive UI component with minimal effort.

Using Qt Quick, you can move fast from sketching ideas on to a working prototype. You can continue using the prototyped code in later development phases with almost no rewriting. You can involve graphic, interaction and animation designers during the entire development process and skip the painful steps of exchanging ideas and requirements on paper or slides.

Developing with Qt Quick does not require any C++ knowledge. It is a script language inspired by CSS and JavaScript. Moreover, it also uses JavaScript a lot in its syntax and functionality. It uses a JavaScript engine to execute your code. Those of you who are familiar with QtScript will feel at home. Though no C++ is needed to develop in Qt Quick, you can extend Qt Quick with own modules written in C++. This can also be used to exchange data with existing Qt applications. This allows starting with Qt Quick without breaking your code if you are interested. And you will :-)

Qt Quick does not replace classical Qt...

Though you can implement complete applications with Qt Quick only, the role of classical Qt and C++ changes, but does not get less important. In complex real world applications, Qt Quick is used for UI and user interaction, application's business logic and system interfaces are still written in C++.

3.2 Declarative vs imperative programming

Qt Quick can be seen as one of several implementations of declarative programming[1]. Qt Quick programs describe which items an application UI consists of, how they look and how they change upon various user actions. This is very different from the traditional imperative programming[2] with Qt C++, which puts algorithms and statements in the foreground. This difference is actually the major reason why Qt Quick can directly be used for designing application UIs. UI design deals with the content on a screen and interactions with users, assuming that application logic is somewhere in the back. If needed, Qt Quick still allows the use of imperative elements and you can even write fairly complex algorithms in JavaScript in the same QML file.

3.3 Four cornerstones

There are four fundamental aspects used in Qt Quick:

1. UI is composed of nested elements ordered in a hierarchical tree structure
2. Elements are described by properties
3. A property can be bound to another property and keep the same value
4. A notification is generated and processed by a handler on each property change or a signal sent

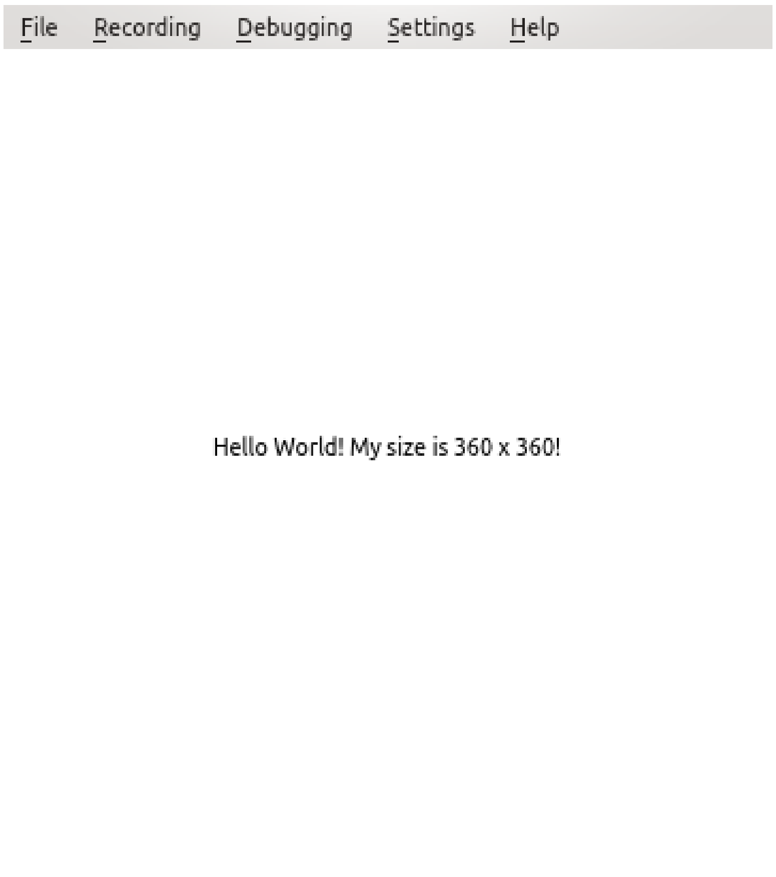
You write a Qt Quick application by adding elements one after the other or by nesting them. You customize the appearance and the behavior of the elements by changing their properties. If a property of one element should follow the value of a property of another element on each change, you bind those properties. If you need to react on a property change, you add some handler code which is automatically executed on each change.

Let's find the four cornerstones listed above in a slightly modified version of our first Qt Quick application:

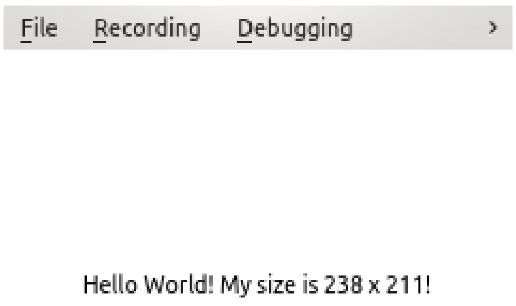
```
import QtQuick 1.1

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: "Hello World! My size is " +
              parent.width + " x " + parent.height + "!"
    }
    onHeightChanged: print ("new size: ", width, "x", height)
    MouseArea {
        anchors.fill: parent
```

[1]http://en.wikipedia.org/wiki/Declarative_programming
[2]http://en.wikipedia.org/wiki/Imperative_programming

```
        onClicked: {
            Qt.quit();
        }
    }
}

*Concerning point 1:** Our application consists of three elements ordered in a t

*Concerning point 2:** We assign new values to the properties ``width`` and ``he

*Concerning point 3:** We bind the text content of the ``Text`` element to the d
```



but if you resize its window with the mouse the text will change and the console output will keep posting the current dimensions of `Rectangle`:



Hello World! My size is 238 x 211!

If it had been just a traditional assignment the text would have not changed and would have kept its first value. This is a very powerful technique which is used very intensively in any Qt Quick application.

> *Concerning point 4:** An `on` notification is generated each time a property change including changes of `width` and `height`. We add a handler for notification `onHeightChanged`. Each time its height changes, the size of the `Rectangle` is printed on the console. It is also possible to send a signal when you need to notify other elements about something. Our simple code segment does not use any signals, but we will talk about them later. Generally, the fourth cornerstone is very similar to properties and signals-and-slots in classical Qt.

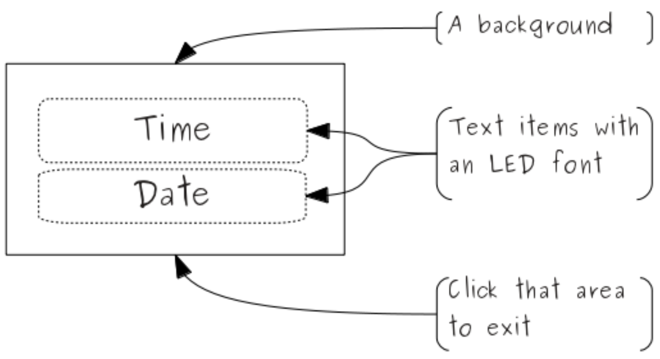
Let's take a look at the application concept phase and let's analyze how we proceed implementing it with Qt Quick. We will take a closer look at these four cornerstones and learn about other important details.

3.4 Moving from a concept to a real application

There is a reason why the concept phase of the application development is important in Qt Quick. Your application UI is based on Qt Quick elements, and most of them are rectangles or something similar to it. You can use property binding and notification handlers to make a functional system with those elements. There is an easy way to replicate and modularize its functionality. The first version of your application runs very soon, even though there might be not much application logic implemented. You can add more application logic and then realize that something should be enhanced in the UI design. This turnarounds are known on any platform with any application development framework. Qt Quick makes going through these turnarounds less time-consuming and less error-prone. You just need to pay a bit more attention to the decomposition of the initial application concept.

We are going to develop a digital clock with an integrated weather forecast. The use case for the application is more of a decorative kind. Imagine waking up in the middle of the night wanting to quickly check what time it is and then continue sleeping. If it is already time to start your day, you might want to find out what's the weather forecast so you can decide what to wear. Our application shows the current time of the day and the weather forecast for the next days fetched from the Internet. Additionally, we need another top level window to store a few basic settings (e.g. the city where the user and his device are currently located). This gives three main components: a clock, weather forecast, and possibly settings. The clock and weather forecast are shown on the same screen, whereas settings can pop-up and dismiss.

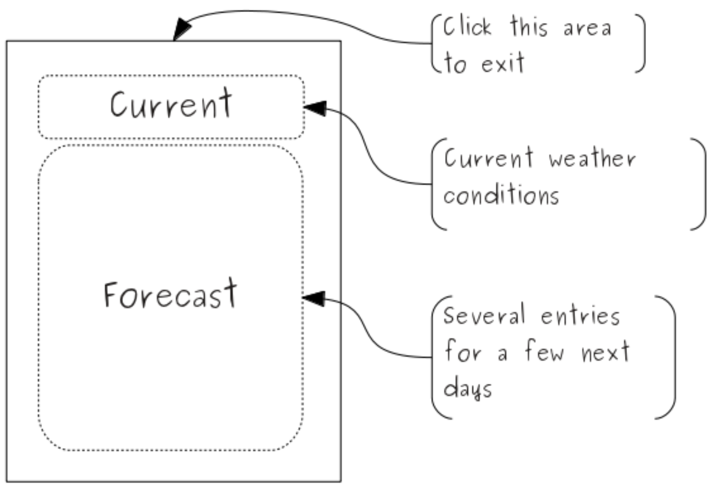
The clock element looks something like this:



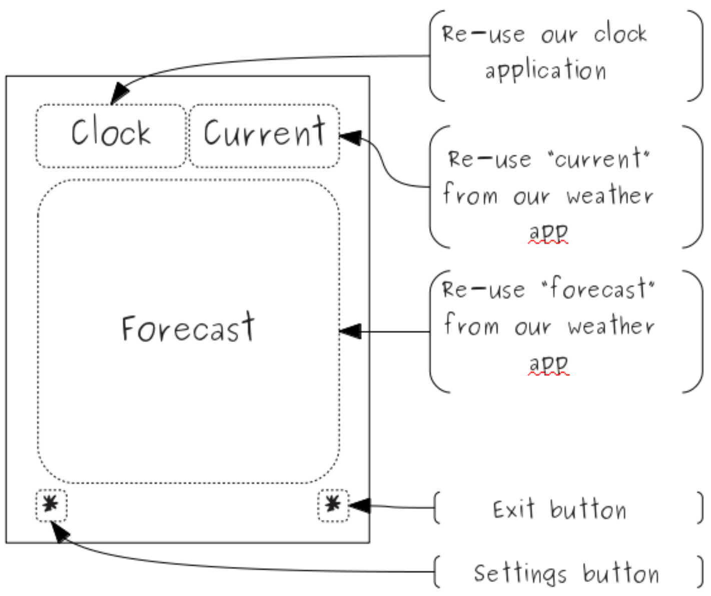
This just shows the current time and date, which can be seen as elements on their own as well.

A weather forecast usually displays information about the weather for the current day and the actual forecast section covering a few days in the future. This information is repeated for all days, showing just weather conditions and temperatures. We plan to get the weather data from the Internet. This makes weather elements tightly linked to the weather data. We should keep this in mind when selecting Qt Quick elements to use.

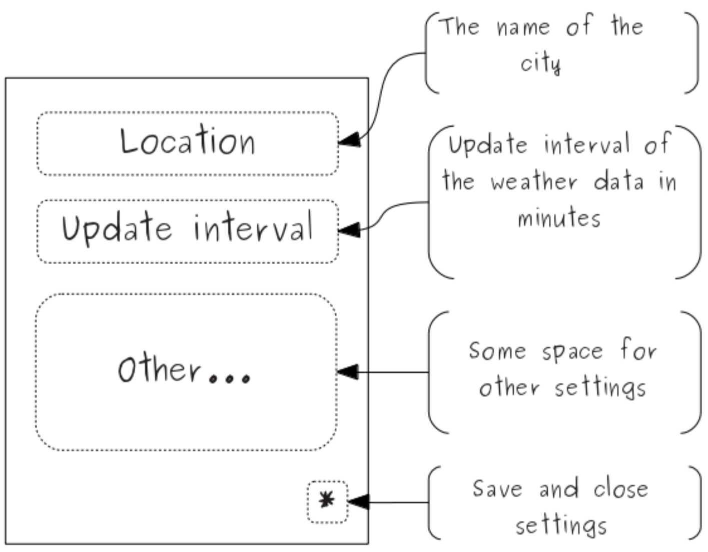
A root element containing all weather related elements can look something like this:


Click this area to exit
Current
Current weather conditions
Forecast
Several entries for a few next days

The clock and weather parts are quite independent and it makes sense to develop them separately and use them as components in the final application:


Clock
Current
Forecast
Re-use our clock application
Re-use "current" from our weather app
Re-use "forecast" from our weather app
Exit button
Settings button

Settings will pop-up as a separate window:


Location
Update interval
Other...
The name of the city
Update interval of the weather data in minutes
Some space for other settings
Save and close settings

This element can be developed separately as well. We just need to take care that there is a way to transfer the settings data from and into the application core.

Additionally, we probably need some basic UI features such as text input fields, check boxes and simple buttons.

Visual appearance is important! We will spend some extra time exploring possibilities in Qt Quick to enhance our application.

What's Next?

In the next chapter we are going to explore how to use Qt Quick elements to compose an UI.

CHAPTER 4

Elements as building blocks

We start with our simple *Hello World* application and use it as a basis to expand and learn what happens.

As we already know, Qt Quick applications can been seen as a hierarchical tree of elements. This tree is composed into a dynamic scene on a screen. All elements can be seen as objects with parent-child relationships which can also be seen as a kind of inheritance.

Some elements are not visible on the screen, but rather used under the hood to control or transform other elements. Due to this, a generic name for all Qt Quick building blocks is items*. There is also *the item* - Item[1] which is the parent of most (21 in Qt Quick 1.1) other items. Being a parent of so many other items, Item[2] has very important properties inherited by many other items. Take some time to check its documentation. We will use the term *item* most of the time, but we will use *element* when we want to underline that an item is visible on the screen.

Visual elements can be manipulated by changing their properties such as position, size, visibility, opacity, and so on. This can be done with or without animations. There are special elements for capturing user input or visualizing data from different sources. Qt documentation provides a list of all items sorted in groups[3]. Take some time to get an overview of all available items.

When you write a Qt Quick application, you should think about its UI in terms of elements. Changes in the UI become transitions, where elements move to another place, hide or even change their form. This is quite different compared to traditional programming, where you used to think in terms of algorithms and spent a lot of time making changes in the UI. Developing UIs with Qt Quick is more like creating a cartoon...

4.1 Composing a basic UI with nested elements

Like in any other tree structure, there is always exactly one root item which contains all others in any *qml* file. Let's take the rectangle from our *Hello World* example, and put some other

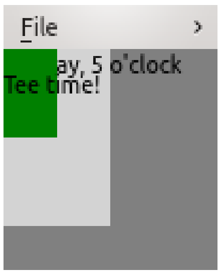
[1] http://qt-project.org/doc/qt-4.8/qml-item.html
[2] http://qt-project.org/doc/qt-4.8/qml-item.html
[3] http://qt-project.org/doc/qt-4.8/qml-groups.html

elements in it:

(`many_elements/many_elements.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Rectangle {
    width: 100
    height: 100
    color: "grey"
    Rectangle {
        width: 50
        height: 80
        color: "lightgrey"
        Text {
            text: "Sunday, 5 o'clock"
        }
    }
    Rectangle {
        width: 25
        height: 40
        color: "green"
        Text {
            anchors.verticalCenter: parent.verticalCenter
            text: "Tee time!"
        }
    }
}
```

This is how it looks on the screen:



Although the application is not any closer to being useful in real life, it's interesting to further analyze what happens on the screen.

One Rectangle[4] is the root item. Two others contain a Text[5] with different text. All items are rendered as a stack according their order in the tree:

> the root rectangle first
>
> then the grey rectangle with its text
>
> the green rectangle comes on top and covers a part of the previous text
>
> the Text[6] element contained in the green rectangle comes as the last one on top of all others.

[4]http://qt-project.org/doc/qt-4.8/qml-rectangle.html
[5]http://qt-project.org/doc/qt-4.8/qml-text.html
[6]http://qt-project.org/doc/qt-4.8/qml-text.html

We could add more elements, all of them are added to this stack and drawn on the upper left corner which is the *(0,0)* position. This is the default rendering position, if you do not specify `x` and `y` properties.

This default behavior would make in this example appear too overlapped on the screen. We add `anchors.verticalCenter: parent.verticalCenter` to move the last text a bit below the text in the gray rectangle. Anchoring allows positioning of elements relatively to each other. We will talk about this a bit later.

4.2 Ordering elements on the screen

It is possible to control the position and order of the elements on the screen in absolute `(x,y)` coordinates or relatively to other elements.

The start of the coordinate system is the upper left corner of the application window. Coordinate units are actual pixels on the screen. In addition to the `(x,y)` coordinates, positioning of the items has a kind of 3rd dimension defining the stacking order of sibling items. This is defined by the z property[7]. By default, its value is `0`. If you set it to `1` it rises the item to the next level. Lets try this out on our small application:

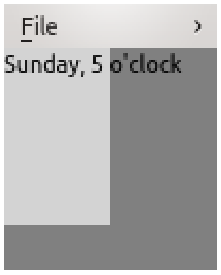
(`ordered_elements/ordered_elements.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Rectangle {
    width: 100
    height: 100
    color: "grey"

    Rectangle {
        width: 50
        height: 80
        color: "lightgrey"
        // puts this rect in the front.
        // Uncomment next line or set it to -1 and see what happens
        //z: 1
        // this rect will clip the text element
        // Uncomment this line and see what happens
        // clip: true
        Text {
            text: "Sunday, 5 o'clock"
        }
    }
    Rectangle {
        width: 25
        height: 40
        color: "green"
        Text {
            anchors.baseline: parent.verticalCenter
            text: "Tee time!"
```

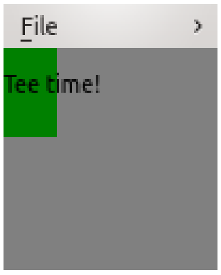
[7]http://qt-project.org/doc/qt-4.8/qml-item.html#z-prop

```
        }
    }
}
```

This looks like this:

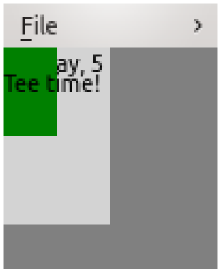


Setting `z` to `-1` will put an item into the background of its parent:



If you noticed, the text elements are not clipped by the boundaries of their parents - rectangles. This is the default behavior for performance reasons. If needed, this can be changed by setting the `clip` property to true: `clip: true`. `clip` is a property of Item[8] and available in all visual elements inheriting from it.

If we set `clip` to true in our application it will look like this:



4.3 Arranging application elements on the screen

On the next step we will need to arrange the elements so that the build up an application UI. Arranging items brings up another key aspect: identification of items.

Though the use of the IDs is optional in Qt Quick. We have to use IDs if items have to be arranged in relation to each other. Generally, you should strongly consider using IDs all the time. This greatly improves readability of the code and prevents weird side effects. Be advised to use consistent IDs for all root items in your project, for example, just `root`. This helps you keep track of items used and avoid side effects.

[8]http://qt-project.org/doc/qt-4.8/qml-item.html

We already saw some use of anchoring in previous examples. Lets take a closer look on this and use it to place elements so that they start getting closer to a clock.

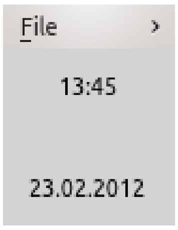
We just take two Text[9] elements and place them inside our `root` element. Anchoring uses so called anchoring lines which are provided as properties. When you anchor items, you just bind an anchoring line of one item to an anchoring line of another one. You basically stitch items to each other. Additionally, you can also set anchor margins. Margins are zero by default and define a distance between anchored items. We place `timeText` and `dateText` centered inside of `root` and add margins of 10 pixels:

(`static_clock/static_clock.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Rectangle {
    id: root
    width: 80
    height: 80
    color: "lightgrey"

    Text {
        id: timeText
        anchors.top: root.top
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.margins: 10
        text: "13:45"
    }
    Text {
        id: dateText
        anchors.bottom: root.bottom
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.margins: 10
        text: "23.02.2012"
    }

    MouseArea {
        anchors.fill: root
        onClicked: {
            Qt.quit();
        }
    }
}
```

The top edge of `timeText` stitches to the top edge of the `root` with a margin of 10 pixels. `dateText` does the same at the bottom. If you run this application and resize the application window with the mouse, you will see that both text items keep their anchored positions at the edges. Just the space between them changes.

[9] http://qt-project.org/doc/qt-4.8/qml-text.html



The item MouseArea[10] fills the entire surface of `root` and even reacts to mouse click events for both text items. We will talk about handling mouse events later.

Anchoring is fairly simple, but you need to pay a bit more attention if you have many elements on the screen. Otherwise you might lose orientation of where your elements are located. A useful approach is to anchor to the dominant elements and keep anchoring in a hierarchical order. A detailed description of anchoring is available in this article[11] in Qt documentation.

Another approach to handle positioning of many elements which should be placed in a regular order is using Positioners[12]. Positioners[13] do all the ordering for you and have some additional features.

Lets try migrate our application to use positioners. It makes the code shorter even with our two items. We now place both Text[14] elements inside a Column[15] positioner. In order to keep them apart from each other we define 20 pixels spacing and anchor the Column[16] in the center of root:

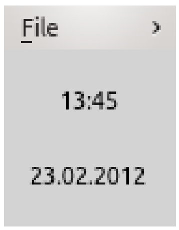
(`static_clock1/static_clock1.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Rectangle {
    id: root
    width: 80
    height: 80
    color: "lightgrey"

    Column {
        id: clockText
        anchors.centerIn: root
        spacing: 20

        Text {
            id: timeText
            anchors.horizontalCenter: parent.horizontalCenter
            text: "13:45"
        }
        Text {
            id: dateText
            anchors.horizontalCenter: parent.horizontalCenter
```

[10]http://qt-project.org/doc/qt-4.8/qml-mousearea.html
[11]http://qt-project.org/doc/qt-4.8/qml-anchor-layout.html
[12]http://qt-project.org/doc/qt-4.8/qml-positioners.html
[13]http://qt-project.org/doc/qt-4.8/qml-positioners.html
[14]http://qt-project.org/doc/qt-4.8/qml-text.html
[15]http://qt-project.org/doc/qt-4.8/qml-column.html
[16]http://qt-project.org/doc/qt-4.8/qml-column.html

```
            text: "23.02.2012"
        }
    }

    MouseArea {
        anchors.fill: root
        onClicked: {
            Qt.quit();
        }
    }
}
```

The result looks pretty much the same:



If you now resize the application window, you will notice that its content stays centered, as we anchored the positioner to the center of `root`.

Qt Quick provides other Positioner[17] elements, such as Row[18], Grid[19] or Flow[20]. We will use some of them during the course of this guide.

4.4 Properties

In previous sections, we used item properties by assigning or binding various values in order to change an item. Properties play an essential role and have a much broader use. In this section we talk about other use cases and use the knowledge gained in our application. Lets take a look at this simple code segment:

```
Rectangle {
  id: myButton
  color: "white"
}
```

The element `myButton` automatically contains all default properties of Rectangle[21], including all default values. Implementing a button, you might need to add some other properties on top of default ones, for example, a border color or even a generic pen color which is used to paint both the label text and the border color. Implementing other items, you might need to keep some values from the application logic. How to add new properties?

You can define a new property using the following syntax:

[17] http://qt-project.org/doc/qt-4.8/qml-positioners.html
[18] http://qt-project.org/doc/qt-4.8/qml-row.html
[19] http://qt-project.org/doc/qt-4.8/qml-grid.html
[20] http://qt-project.org/doc/qt-4.8/qml-flow.html
[21] http://qt-project.org/doc/qt-4.8/qml-rectangle.html

```
[default] property <type> <name>[: defaultValue]
```

A new property for the border color in our Button would then look like this:

```
property string borderColor: "white"
```

We used the type "string" here. It is one of a range of types available, see this article[22] about property types in Qt documentation.

In some cases, it makes more sense to define an alias for an existing property instead of defining a new one. This can be done using the following syntax:

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

In fact, in the case of our button inheriting from Rectangle[23], we should define the new property as an alias since the Rectangle[24] element already has a property defining the the color of the border:

```
Rectangle {
  id: myButton
  property alias borderColor: myButton.border.color
  ...
}
```

The optional `default` keyword in the syntax outlined above are used to create *default properties*. They hold the child elements of an item. This is a more advanced use which is out of the scope of this guide. Refer to this[25] and this article[26] in Qt documentation.

As mentioned before, a property change generates notification signals that can have an handler to respond to the property change. Handlers are assigned to `on<property_name>Changed` properties which are automatically created for all items properties, including custom properties you have added. Lets do a small experiment with a rectangle to see this in action:

```
import QtQuick 1.1

Rectangle {
  width: 100; height: 100

  onWidthChanged: console.log("Rectangle width: ", width)
  onHeightChanged: console.log("Rectangle height: ", height)
}
```

The code above creates a window with an empty white rectangle. If you drag a corner to resize its desktop window, you will see something like this in the console:

```
Rectangle width:  640
Rectangle height:  460
Rectangle width:  100
```

[22]http://qt-project.org/doc/qt-4.8/qdeclarativebasictypes.html
[23]http://qt-project.org/doc/qt-4.8/qml-rectangle.html
[24]http://qt-project.org/doc/qt-4.8/qml-rectangle.html
[25]http://qt-project.org/doc/qt-4.8/qml-extending.html#default-property
[26]http://qt-project.org/doc/qt-4.8/propertybinding.html#default-properties

```
Rectangle height:  100
Rectangle height:  101
Rectangle height:  102
Rectangle width:  104
Rectangle height:  109
Rectangle width:  105
Rectangle height:  110
Rectangle height:  111
...
```

The code of handler reacting to changes can be much more sophisticated and can use JavaScript. We will talk about using JavaScript in Qt Quick in the section *Using JavaScript* (page 36)

Property binding is another fundamental aspect. It makes a property to follow changes of another property. This works across items, and is actually used across items most of the time. Binding occurs each time you use `:`, and have a value that can change on the right-hand side:

```
id1.text: id2.text  // text in the id1 follows changes of the text in id2
width: 16    height/9  // this keeps an item being the 16:9 aspect ratio :-)
```

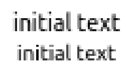
The latter is interesting as you can also do some calculations or even bind to a function returning a value.

The following example combines what we just learned about properties in one small application:

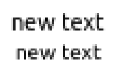
(`property_binding/property_binding.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Rectangle {
    id: root

    property int adaptiveSize: root.width/10

    width: 100
    height: 100

    Text {
        id: text1
        anchors.centerIn: parent
        text: "initial text"
        onTextChanged: {
            console.log ("text 1 is changed: " + text1.text)
        }
    }

    Text {
        id: text2
        anchors.top: text1.bottom
        anchors.horizontalCenter: text1.horizontalCenter
        text: text1.text
        font.pixelSize: adaptiveSize
        onTextChanged: {
            console.log ("text 2 is changed: " + text2.text)
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            text1.text = "new text"
            console.log ("text 1 is now: " + text1.text)
            console.log ("text 2 is now: " + text2.text)
        }
    }
}
```

The application starts like this:



As the `text2.text` is bound to `text1.text`, the property change in `text1` on a mouse click is automatically passed to `text2`:



The following appears in the console:

```
text 2 is changed: initial text
text 1 is changed: new text
text 2 is changed: new text
text 1 is now: new text
text 2 is now: new text
```

The mouse click was received by the MouseArea[27] element and processed by the JavaScript code in the `onClicked` handler. We will take a closer look at this in one of the following chapters.

If you resize the window the pixel font size of the text in `text2` is dynamically scaled to be 1/10 of the width of the `root`:

[27]http://qt-project.org/doc/qt-4.8/qml-mousearea.qml

Later on, we use properties to define custom parameters that define the appearance of our application as well as keep a few settings.

4.5 Other Visual Composition Elements

Qt Quick 1.x provides a broad selection of items which can be used for UI creation. Qt documentation covers them[28] in full details and we are not going to discuss all of them here. In general, there are four sets of elements which can be used for composing application UIs:

- Drawn elements such as text and rectangles
- Loaded content such as images and web pages as well as fonts
- Visualization elements which arrange the above in various ways and can use models with data if needed
- Effects such as animations and transitions while displaying the above

Qt5 adds two more:

- Vector-oriented graphics canvas
- 3D rendering with Open GL effects and shaders

These topics are covered by other guides available from here[29]

Even though you do not see high-level UI components on the list, Qt Quick gives you more than enough power to compose very rich UIs. These UIs can be like cartoons, where images receive user input and gets transformed smoothly into other images or elements, rectangles rounded to become circles, ellipses, and so on. You can also create classical UI components if required without significant effort. When you start working on a UI, try to first decompose it in basic elements, text and images and views. Think how elements interact with each other, what happens upon user input or changes in the application logic.

[28]http://qt-project.org/doc/qt-4.8/qdeclarativeelements.html
[29]http://qt-project.org/wiki/Developer-Guides/

What's Next?

In the next chapter we will learn how to load external content and create the required UI appearance.

CHAPTER 5

Loading and Displaying Content

As the next step in the development of our clock application, we are going to add a background image instead of a solid color and display time and date in a different font than the standard one. This type of content is usually provided in external files. We have to learn how to load and display this kind of content. We should also learn how to customize the standard appearance of elements to make an application look more attractive.

5.1 Accessing and loading content

Qt Quick 1.x allows the loading of images and fonts located on the network or in the local file system. The path to related files is defined relatively to the location of the QML file where the content is loaded. The same relative paths can be used when content is loaded over the network. This makes moving the content from the file system to the network very easy and does not require large changes in the code. See this article[1] for more details.

Loading external content can always end up with problems. Files get misplaced. The network connection can be too slow or the server is off-line. You don't want that. Image[2] and FontLoader[3] provide `status` and `progress` properties for handling such situations. `progress` changes its value from `0.0` to `1.0` in accordance to the actual progress of loading. If content is loaded from the file system, `progress` is set to `1.0` almost instantly if we do not use it in our application. The code segment for loading a font and tracing the status of loading looks like this:

```
FontLoader {
    id: ledFont
    source: "./content/resources/font/LED_REAL.TTF"
    onStatusChanged: if (ledFont.status == FontLoader.Error)
                         console.log ("Font \"" +
                                      source +
                                      "\" cannot be loaded")
}
```

[1] http://qt-project.org/doc/qt-4.8/qdeclarativenetwork.html
[2] http://qt-project.org/doc/qt-4.8/qml-image.html
[3] http://qt-project.org/doc/qt-4.8/qml-fontloader.html

Our new font, LED_REAL.TTF, is stored in the content folder, which is located in the same folder as our application. If for any reason it does not load, an error message is posted on the console and the application continues using a default font.

5.2 Basic Image Parameters

Loading images works almost the same way:

```
Image {
    id: background
    source: "./content/resources/light_background.png"
    fillMode: "Tile"
    anchors.fill: parent
    onStatusChanged: if (background.status == Image.Error)
                         console.log ("Background image \"" +
                                      source +
                                      "\" cannot be loaded")
}
```

We need to change a few image properties. The image should cover the entire surface of the top-level element (its parent). In many cases, filling a background is done with small images which just design a pattern. Scaling them to the size of background would change the pattern, and the background would not be the same as planned. In these cases, you set fillMode to Tile as shown in the code above.

Dealing with Image Sizes

Most of the visual properties of an image can be changed using its parent (Item[a]) properties. Other Image's[b] properties help handle critical image aspect - its size. If you deal with large images, you should set the sourceSize property to a size which you really need, otherwise the image loads in its full size. It is worth to notice the difference between paintedHeight and height, and between paintedWidht and width. The *painted* property pairs describe the size of the area taken by the image when it is painted on the screen, whereas the another pair describes the loaded size of the image. Both can be different if the image is scaled. If you change the scale property inherited from Item[c], be aware of the impact of the fillMode property on the actual scaling result. The smooth property smoothens the edges of scaled images, but also incurs a performance cost. See the Image[d] documentation for more details.

[a]http://qt-project.org/doc/qt-4.8/qml-item.html
[b]http://qt-project.org/doc/qt-4.8/qml-image.html
[c]http://qt-project.org/doc/qt-4.8/qml-item.html
[d]http://qt-project.org/doc/qt-4.8/qml-image.html

5.3 Basic Text Parameters

In the first section, we loaded a custom LED-like font. We now need to assign it to the appropriate text elements. Text[4] has a grouped `font` property. It has to be used to customize various font-related text properties. One of them, `font.family`, holds the name of a font to be used.

Finding the Fonts Installed

Sometimes it might be tricky to find out which fonts are installed on your target system and how to spell their names right. A small example provided in Qt Quick lists names of all available[a] rendered in their fonts.

[a] http://qt-project.org/doc/qt-4.8/declarative-text-fonts-fonts-qml-fonts-qml-availablefonts-qml.html

Other Text[5] properties allow a broad variation of the visual appearance of a text. In our application, we use the `color`, `style`, and size-related properties.

The customized text element for displaying time looks like this:

```
Text {
    id: timeText
    text: root.currentTime
    font.pixelSize: root.height    root.timeTextProportion
    font.family: ledFont.name
    font.bold: true
    color: root.textColor
    style: Text.Raised
    styleColor: "black"
}
```

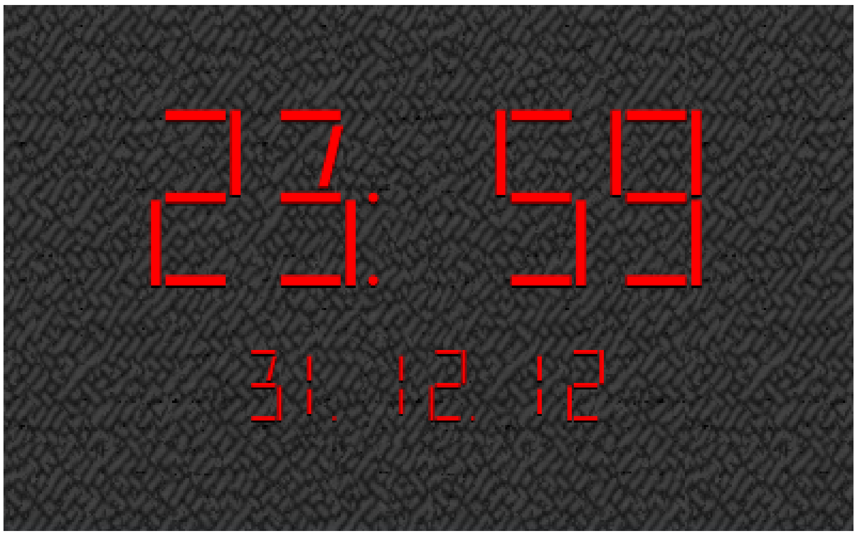
Our plan is to implement as much flexibility as possible in the size and layout of all elements as we want to run our application on different screen sizes. This is why the code above binds the `pixelSize` of `timeText` to the height of the `root` element. This will scale the text whenever the height of `root` changes. There are ways to make it better, but the current version of Qt Quick unfortunately does not provide font metrics data in the font properties. If it were available, we could make the text size adapt to the width changes as well.

In order to make the `timeText` element appear more attractive, we use some tricks for the visual appearance by setting `style` to `Text.Raised` and `styleColor` to `"black"`. This detaches the text from the background so it seems like it's hovering over it.

The Text[6] element also provides basic layout controls (for example, you can set how the text should be wrapped using `wrapMode`). We're going to use this property later. The most important thing to note about text formatting is that Text[7] supports rich text[8] markup. This makes it possible to display one block of text in various formatting, colors, sizes etc.

The appearance of our application is now less boring:

[4] http://qt-project.org/doc/qt-4.8/qml-text.html
[5] http://qt-project.org/doc/qt-4.8/qml-text.html
[6] http://qt-project.org/doc/qt-4.8/qml-text.html
[7] http://qt-project.org/doc/qt-4.8/qml-text.html
[8] http://qt-project.org/doc/qt-4.8/Qt's richtext-html-subset.html



The full source code is listed in the last section of this chapter.

5.4 Get ready for translation

When starting to develop an application, it makes a lot of sense to have some basic thoughts about international versions even if you do not plan for it now. Qt provides a set of tools which are used as the base by Qt Quick. If you know and use the essential part of the tools in the beginning of your development, it can save a lot of time later when your application becomes a complex system with a lot of code.

Qt Quick provides almost the same APIs for translation as in Qt Script. See the section, Produce Translations[9] for more details. The major API is the `qsTr()` function, which should be used to wrap all visual strings in an application. The *Hello World* greeting in our first application should then look like this:

```
import QtQuick 1.1

Rectangle {
    width: 360
    height: 360
    Text {
        anchors.centerIn: parent
        text: qsTr("Hello World")
    }
    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }
}
```

The application continues to run the same way as before using `qsTr()`. You need to use the Qt translation tool chain to see translation in action. See the QML Internationalization[10] article

[9] http://qt-project.org/doc/qt-4.8/scripting.html#produce-translations
[10] http://qt-project.org/doc/qt-4.8/qdeclarativei18n.html

for more details about how to generate and integrate translation files. When translation files are available, there are three ways to load them:

- by using the `qmlviewer` command-line option `-translate`
- by placing them in the `i18n` sub-folder (see this example[11])
- by modifying the default behavior of the `qmlviewer` and loading QML and translation files in your own way (this is beyond of scope for this guide)

If your plan is to make a version of an application for right-to-left languages, for example, Arabic, take a look at the QML Right-to-left User Interfaces[12] article.

Switching Languages at Runtime

Currently, there are no standard APIs to load a new language after a Qt Quick application has already started. It is still possible with some additional code. See this article on the Qt developer Wiki about possible workarounds: “How to do dynamic translation in QML”[a]

[a]http://qt-project.org/wiki/How_to_do_dynamic_translation_in_QML/

If you are interested in more details, check these further readings about internationalization in Qt:

- Qt documentation: Writing Source Code for Translation[13]
- Qt documentation: Internationalization with Qt[14]
- Qt developer Wiki: “Qt Internationalization” article[15]

5.5 Static Clock Application Code

This is the code of our application including all enhancements as discussed in this chapter:

(`static_clock2/static_clock2.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

// this will be a component later

Rectangle {
    id: root

    property bool showDate: true
    property bool showSeconds: true
    property string currentTime: "23:59"
    property string currentDate: "31.12.12"
```

[11]http://qt-project.org/doc/qt-4.8/declarative-i18n.html
[12]http://qt-project.org/doc/qt-4.8/qml-righttoleft.html
[13]http://qt-project.org/doc/qt-4.8/i18n-source-translation.html
[14]http://qt-project.org/doc/qt-4.8/internationalization.html
[15]http://qt-project.org/wiki/QtInternationalization

```
// the sizes are in proportion to the hight of the clock.
// There are three borders, text and date.
// 3*borderProportion+timeTextProportion+dateTextProportion has to be 1.0
property real borderProportion: 0.1
property real timeTextProportion: 0.5
property real dateTextProportion: 0.2
property string textColor: "red"

height:240
width:400

Image {
    id: background
    source: "../content/resources/light_background.png"
    fillMode: "Tile"
    anchors.fill: parent
    onStatusChanged: if (background.status == Image.Error)
                         console.log (qsTr("Background image \"") +
                                      source +
                                      qsTr("\" cannot be loaded"))
}

FontLoader {
    id: ledFont
    source: "../content/resources/font/LED_REAL.TTF"
    onStatusChanged: if (ledFont.status == FontLoader.Error)
                         console.log (qsTr("Font \"") +
                                      source +
                                      qsTr("\" cannot be loaded"))
}

Column {
    id: clockText
    anchors.centerIn: parent
    spacing: root.height*root.borderProportion

    Text {
        id: timeText
        text: root.currentTime
        font.pixelSize: root.height*root.timeTextProportion
        font.family: ledFont.name // use "Series 60 ZDigi" on Symbian instead
        font.bold: true
        color: root.textColor
        style: Text.Raised
        styleColor: "black"
    }

    Text {
        id: dateText
        text: currentDate
        color: textColor
        anchors.horizontalCenter: parent.horizontalCenter
        font.family: ledFont.name // use "Series 60 ZDigi" on Symbian instead
        font.pixelSize: root.height*root.dateTextProportion
        visible:  showDate
        style: Text.Raised
        styleColor: "black"
        }
    }
}
```

What's Next?

In the next chapter we will take a look at scripting in Qt Quick. A small script code can make our clock tick.

CHAPTER 6

Using JavaScript

We've already used JavaScript a lot in our code, but we have only scratched the surface. JavaScript can be used in many more sophisticated and powerful ways in a Qt Quick application. In fact, Qt Quick is implemented as a JavaScript extension. JavaScript can be used almost anywhere as long as the code returns the value of the expected type. Moreover, using JavaScript is the standard way of writing parts of the application code which deal with application logic and calculations.

There are two important topics we need to talk about before we continue developing our application.

6.1 JavaScript is not JavaScript

JavaScript has its origins in the web development. In the course of time, JavaScript has rapidly grown in its popularity and inspired development of many extensions and add-ons. In order to support a broad use, JavaScript was formalized as a programming language in *ECMAScript-262* standard. It is important to underline that *ECMAScript-262 standard* only covers the language aspects and leaves out any API providing additional functionality such as objects and libraries to access the web page content. Despite the standardization efforts, many JavaScript details in web development are still browser-specific - even though the situation has improved in the last few years. See the Wikipedia article about JavaScript[1] for more details.

JavaScript is also used outside of the web where its functionality is tailored to support a use case. Still, the use of JavaScript for client-side programming in web development is dominating. Due to this, all books and most web resources about JavaScript are actually dedicated to web development. Qt Quick belongs to one of the platforms which use JavaScript outside of the web. If you read books or other materials about JavaScript to understand and use it better with Qt Quick, be aware of this difference.

Qt development teams are doing their best to provide all the details the use of JavaScript in Qt Quick, and this guide is part of that effort.

[1] http://en.wikipedia.org/wiki/Javascript

6.2 More About JavaScript

This guide contains an annex about *basics of JavaScript* (page 97) tailored for Qt Quick. We strongly recommend reading it if you are not familiar with JavaScript, but have some general background in programming.

In addition to the references in the annex, consider reading the following articles on the Mozilla Developer Network:

- "About JavaScript"[2]
- "A re-introduction to JavaScript"[3]
- "JavaScript Guide"[4]

The following three articles in Qt Documentation explain essential details of JavaScript in Qt Quick:

- Integrating JavaScript[5] - key aspects to be aware of when using JavaScript in Qt Quick
- ECMAScript Reference[6] - a list of built-in objects, functions and properties supported by QtScript and so in Qt Quick
- QML Scope[7] - explains the visibility of JavaScript objects and Qt Quick items

Note that significant changes and large updates to Qt Documentation may come with the future Qt releases to provide a full coverage of the use of JavaScript in Qt Quick. Stay tuned and check Qt Documentation again and again.

6.3 Adding Logic to Make the Clock Tick

We've already used a bit of JavaScript in previous sections. Most of it was for catching error conditions while loading a custom font and an image. In this section, we use JavaScript to show the actual time and date.

We are going to use the `Date` from the global object to get the current time and date. The returned data has to be formatted so that we only keep parts of the date and time information that we need. We use Qt.formatDateTime[8] for this:

```
function getFormattedDateTime(format) {
    var date = new Date
    return Qt.formatDateTime(date, format)
}
```

[2] http://developer.mozilla.org/en/JavaScript/About_JavaScript
[3] http://developer.mozilla.org/en/A_re-introduction_to_JavaScript
[4] http://developer.mozilla.org/en/JavaScript/Guide
[5] http://qt-project.org/doc/qt-4.8/qdeclarativejavascript.html
[6] http://qt-project.org/doc/qt-4.8/ecmascript.html
[7] http://qt-project.org/doc/qt-4.8/qdeclarativescope.html
[8] http://qt-project.org/doc/qt-4.8/qml-qt.html#formatDateTime-method

The :Qt.formatDateTime[9] function is part of QML Global Object[10], which provides many other useful utilities in addition to the standard set defined by the ECMAScript Reference[11]. It is worth taking a closer look at its documentation.

The `getFormattedDateTime()` function is used in another function, which creates actual values for the Text[12] elements in our clock:

```
function updateTime() {
    root.currentTime = "<big>" +
            getFormattedDateTime(Style.timeFormat) +
            "</big>" +
            (showSeconds ? "<sup><small> " + getFormattedDateTime("ss") +
                "</small></sup>" : "");
    root.currentDate = getFormattedDateTime(Style.dateFormat);
}
```

Note: We use rich-text formatting in the text value of the time as discussed in the previous section.

We also use the conditional operator (also called the "ternary operator") on the value of `showSeconds`. `showSeconds` is a custom property that defines whether the seconds must be shown on the clock. Using the conditional operator in Qt Quick is a very convenient way to bind a property (or any other variable) to a value depending on a condition.

The `updateTime()` function needs a trigger so that the `currentTime` and `currentDate` properties are constantly updated. We use the Timer[13] item for this:

```
Timer {
    id: updateTimer
    running: Qt.application.active && visible == true
    repeat: true
    triggeredOnStart: true
    onTriggered: {
        updateTime()
        // refresh the interval to update time each second or minute.
        // consider the delta in order to update on a full minute
        interval = 1000    (showSeconds? 1 : (60 - getFormattedDateTime("ss")))
    }
}
```

Our timer implements some interesting aspects.

In order to optimize battery consumption, we bind the timer's `running` property to two other properties, which stops the timer, thereby reducing CPU activity. It stops if our clock element becomes invisible (when used as a component in another application) or if our application becomes inactive (running in the background or iconified).

We also assign a value to the `interval` property while not loading, but when the timer is triggered. This is needed to catch the delta time to the full minute when seconds are not used.

[9] http://qt-project.org/doc/qt-4.8/qml-qt.html#formatDateTime-method
[10] http://qt-project.org/doc/qt-4.8/qdeclarativeglobalobject.html
[11] http://qt-project.org/doc/qt-4.8/ecmascript.html
[12] http://qt-project.org/doc/qt-4.8/qml-text.html
[13] http://qt-project.org/doc/qt-4.8/qml-timer.html

This ensures that we update the clock exactly on the minute.

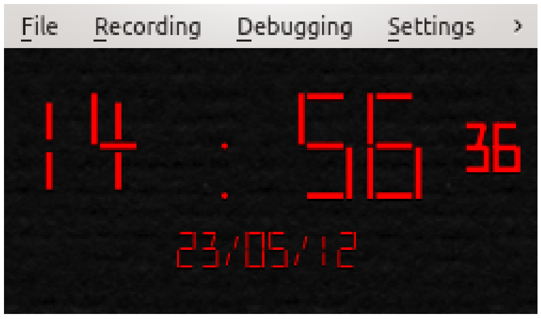
The full code of our application including all enhancements discussed above looks like this:

(`NightClock/NightClock.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Rectangle {
    id: root

    property bool showDate: true
    property bool showSeconds: true
    property string currentTime
    property string currentDate
    // the sizes are in proportion to the hight of the clock.
    // There are three borders, text and date.
    // 3*borderProportion+timeTextProportion+dateTextProportion has to be 1.0
    property real borderProportion: 0.1
    property real timeTextProportion: 0.5
    property real dateTextProportion: 0.2
    property string textColor: "red"
    property string timeFormat: "hh :mm"
    property string dateFormat: "dd/MM/yy"

    height:120
    width:250

    color: "transparent"

    // returns formatted time and date
    function getFormattedDateTime(format) {
        var date = new Date
        return Qt.formatDateTime(date, format)
    }

    function updateTime() {
        root.currentTime = "<big>" +
                getFormattedDateTime(timeFormat) +
                "</big>" +
                (showSeconds ? "<sup><small> " + getFormattedDateTime("ss") +
                               "</small></sup>" : "");
        root.currentDate = getFormattedDateTime(dateFormat);
    }

    Image {
        id: background
        source: "../content/resources/background.png"
        fillMode: "Tile"
        anchors.fill: parent
        onStatusChanged: if (background.status == Image.Error)
                             console.log (qsTr("Background image \"") +
                                          source +
                                          qsTr("\" cannot be loaded"))
    }

    FontLoader {
        id: ledFont
        // unfortunately, the font will not load on a Symbian device,
        // and the default font will be used:
        // http://bugreports.qt-project.org/browse/QTBUG-6611
        // The bug should be fixed in 4.8
        source: "../content/resources/font/LED_REAL.TTF"
        onStatusChanged: if (ledFont.status == FontLoader.Error)
                             console.log("Font \"" + source + "\" cannot be loaded")
    }

    Timer {
        id: updateTimer
        running: Qt.application.active && visible == true
        repeat: true
        triggeredOnStart: true
        onTriggered: {
            updateTime()
            // refresh the interval to update the time each second or minute.
            // consider the delta in order to update on a full minute
            interval = 1000*(showSeconds? 1 : (60 - getFormattedDateTime("ss")))
        }
    }

    // trigger an update if the showSeconds setting has changed
    onShowSecondsChanged: {
        updateTime()
    }

    Column {
        id: clockText
        anchors.centerIn: parent
        spacing: root.height*root.borderProportion

        Text {
            id: timeText
            textFormat: Text.RichText
            text: root.currentTime
            font.pixelSize: root.height*root.timeTextProportion
            font.family: ledFont.name // use "Series 60 ZDigi" on Symbian instead
            font.bold: true
            color: root.textColor
            style: Text.Raised
            styleColor: "black"
        }

        Text {
            id: dateText
            text: root.currentDate
            color: root.textColor
            anchors.horizontalCenter: parent.horizontalCenter
            font.family: ledFont.name // use "Series 60 ZDigi" on Symbian instead
            font.pixelSize: root.height*root.dateTextProportion
            visible: root.showDate
            style: Text.Raised
            styleColor: "black"
        }
    }
```

```
}
```

The appearance of the application has remained the same:



6.4 Importing JavaScript Files

If your application has a lot of JavaScript code, consider moving it to a separate file. You can import those files just like we imported the Qt Quick module. Due to a special role that JavaScript plays in Qt Quick, you must define the namespace for the content of the that file, for example, *Logic* in this example. Your source code would then use `Logic.foo()` instead of just `foo()`. The import statement looks like this:

```
import QtQuick 1.1
import "logic.js" as Logic
```

Note: If the application logic is complex, consider implementing it in C++ and importing it into Qt Quick. See the "Extending QML Functionalities using C++"[14] article for more details.

When you import a JavaScript file, it is used like a library and has the scope of the QML file importing it. In some cases you might need a stateless library or a set of global variables shared by multiple QML files. You can use the `.pragma library` declaration for this. See the "Integrating JavaScript"[15] article in Qt Documentation for more details.

We move the JavaScript functions of our clock into the `logic.js` file imported as `Logic`. We also move all style properties into the `style.js` file imported as `Style`. This considerably simplifies the code and allows sharing the style with other components that we're going to develop later.

The complete code of our application importing JavaScript files as discussed above looks like this:

(`components/NightClock.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1
import "../js/style.js" as Style
import "../js/logic.js" as Logic
```

[14]http://qt-project.org/doc/qt-4.8/qml-extending.html
[15]http://qt-project.org/doc/qt-4.8/qdeclarativejavascript.html

```
Rectangle {
    id: root

    property bool showDate: true
    property bool showSeconds: true
    property string currentTime
    property string currentDate
    property string textColor: "green"

    height:120
    width:300
    color: "transparent"

    function updateTime() {
        root.currentTime = "<big>" + Logic.getFormattedDateTime(Style.timeFormat) + "</b
                (showSeconds ? "<sup><small> " + Logic.getFormattedDateTime("ss") + "</s
        root.currentDate = Logic.getFormattedDateTime(Style.dateFormat);
    }

    FontLoader {
        id: ledFont
        // unfortunately, the font will not load on a Symbian device,
        // and the default font will be used:
        // http://bugreports.qt-project.org/browse/QTBUG-6611
        // The bug should be fixed in 4.8
        source: "../content/resources/font/LED_REAL.TTF"
        onStatusChanged: if (ledFont.status == FontLoader.Error)
                             console.log("Font \"" + source + "\" cannot be loaded")
    }

    Timer {
        id: updateTimer
        running: Qt.application.active && visible == true
        repeat: true
        triggeredOnStart: true
        onTriggered: {
            updateTime()
            // refresh the interval to update the time each second or minute.
            // consider the delta in order to update on a full minute
            interval = 1000*(showSeconds? 1 : (60 - Logic.getFormattedDateTime("ss")))
        }
    }

    // trigger an update if the showSeconds setting has changed
    onShowSecondsChanged: {
        updateTime()
    }

    Column {
        id: clockText
        anchors.centerIn: parent
        spacing: root.height*Style.borderProportion

        Text {
            id: timeText
            textFormat: Text.RichText
            text: root.currentTime
            font.pixelSize: root.height*Style.timeTextProportion
            font.family: ledFont.name // use "Series 60 ZDigi" on Symbian instead
            font.bold: true
            color: root.textColor
            style: Text.Raised
            styleColor: "black"
        }

        Text {
            id: dateText
            text: root.currentDate
            color: root.textColor
            anchors.horizontalCenter: parent.horizontalCenter
            font.family: ledFont.name // use "Series 60 ZDigi" on Symbian instead
            font.pixelSize: root.height*Style.dateTextProportion
            visible: root.showDate
            style: Text.Raised
            styleColor: "black"
        }
    }
}
```

More Advanced Use of JavaScript

If you are interested in a more advanced use of JavaScript with Qt Quick, consider reading “Qt Quick Application Developer Guide for Desktop” available under this link[a].

[a] http://qt-project.org/wiki/Developer-Guides/

What’s Next?

In the next chapter, we start developing the weather forecast application and learn how to retrieve and represent data in Qt Quick.

CHAPTER 7

Acquire and Visualize Data

In this section, we leave our clock application for a while and start another one: a weather forecast application. This section focuses on handling data. Our previous code kept data in properties and in JavaScript variables. This is only sufficient for small and simple applications. Sooner or later you will need to deal with larger sets of data.

Qt Quick implements the known model-view architecture and provides a handy set of APIs for this. There is a selection of *models* which keep and, if needed, acquire data. *View* elements read model items and render each of them with the help of a *delegate* in a specific way. For example, as a grid or as a list.

7.1 Models

Qt Quick models are very simple since they are based on the concept of lists. The three kinds of models that are used the most are:

- an `int` value (useful to display something multiple times)
- a JavaScript array of objects
- list models, for example, ListModel[1] and XmlListModel[2] elements

See the *Models and Data Handling* section in the "QML Elements"[3] article for a full list of model related items. There are also some advanced approaches which are discussed in the "QML Data Models"[4] article in Qt Documentation.

We are going to use XmlListModel[5] and take a look at a few examples where an `int` and an `array` are used as models.

Our weather forecast application uses Google weather APIs to get the data.

[1] http://qt-project.org/doc/qt-4.8/qml-listmodel.html
[2] http://qt-project.org/doc/qt-4.8/qml-xmllistmodel.html
[3] http://qt-project.org/doc/qt-4.8/qdeclarativeelements.html
[4] http://qt-project.org/doc/qt-4.8/qdeclarativemodels.html
[5] http://qt-project.org/doc/qt-4.8/qml-xmllistmodel.html

Note: Google weather APIs are not announced as a regular internet service yet.

With these APIs, you can make a query on the web and receive weather data in XML as a response. As this is a very common way of data provisioning, Qt Quick provides a dedicated model for it: XmlListModel[6].

XmlListModel[7] uses XPath and XQuery (see this article in Wikipedia[8]) to read the data delivered as XML. XmlListModel[9] uses XmlRole[10] to create model items for selected XML tree nodes. Let's see how this works.

The query URL is formatted like this:

```
http://www.google.com/ig/api?weather=[LOCATION]&hl=[LANGUAGE]
```

It returns the current weather conditions and a forecast for the next four days. If `LOCATION` is set as `Munich` and `LANGUAGE` is set as English, it looks like this:

```
http://www.google.com/ig/api?weather=Munich&hl=en
```

It returns the following XML output:

```
<?xml version="1.0" ?>
<xml_api_reply version="1">
  <weather module_id="0" tab_id="0" mobile_row="0" mobile_zipped="1" row="0" sec
    <forecast_information>
      <city data="Munich, Bavaria" />
      <postal_code data="Munich" />
      <latitude_e6 data="" />
      <longitude_e6 data="" />
      <forecast_date data="2012-02-22" />
      <current_date_time data="1970-01-01 00:00:00 +0000" />
      <unit_system data="US" />
      </forecast_information>
    <current_conditions>
      <condition data="Clear" />
      <temp_f data="39" />
      <temp_c data="4" />
      <humidity data="Humidity: 56%" />
      <icon data="/ig/images/weather/sunny.gif" />
      <wind_condition data="Wind: E at 8 mph" />
      </current_conditions>
    <forecast_conditions>
      <day_of_week data="Wed" />
      <low data="27" />
      <high data="43" />
      <icon data="/ig/images/weather/sunny.gif" />
      <condition data="Clear" />
```

[6] http://qt-project.org/doc/qt-4.8/qml-xmllistmodel.html
[7] http://qt-project.org/doc/qt-4.8/qml-xmllistmodel.html
[8] http://en.wikipedia.org/wiki/XPath
[9] http://qt-project.org/doc/qt-4.8/qml-xmllistmodel.html
[10] http://qt-project.org/doc/qt-4.8/qml-xmlrole.html

```
      </forecast_conditions>
    <forecast_conditions>
      <day_of_week data="Thu" />
      <low data="36" />
      <high data="43" />
      <icon data="/ig/images/weather/chance_of_rain.gif" />
      <condition data="Chance of Rain" />
      </forecast_conditions>
    <forecast_conditions>
      <day_of_week data="Fri" />
      <low data="36" />
      <high data="54" />
      <icon data="/ig/images/weather/sunny.gif" />
      <condition data="Clear" />
      </forecast_conditions>
    <forecast_conditions>
      <day_of_week data="Sat" />
      <low data="34" />
      <high data="48" />
      <icon data="/ig/images/weather/chance_of_rain.gif" />
      <condition data="Chance of Rain" />
      </forecast_conditions>
  </weather>
</xml_api_reply>
```

A model which queries and processes this data looks like this:

```
import QtQuick 1.1

Item {
    id: root
    property string location: "Munich"
    property string baseURL: "http://www.google.com"
    property string dataURL: "/ig/api?weather="
    // some other values: "de", "es", "fi", "fr", "it", "ru"
    property string language: "en"

    XmlListModel {
        id: weatherModelCurrent
        source: baseURL + dataURL + location + "&hl=" + language
        query: "/xml_api_reply/weather/current_conditions"

        XmlRole { name: "condition"; query: "condition/@data/string()" }
        XmlRole { name: "temp_f"; query: "temp_f/@data/string()" }
        XmlRole { name: "humidity"; query: "humidity/@data/string()" }
        XmlRole { name: "icon_url"; query: "icon/@data/string()" }
        XmlRole { name: "wind_condition"; query: "wind_condition/@data/string()" }
    }

    XmlListModel {
        id: weatherModelForecast
        source: baseURL + dataURL + location  + "&hl=" + language
        query: "/xml_api_reply/weather/forecast_conditions"
        XmlRole { name: "day_of_week"; query: "day_of_week/@data/string()" }
        XmlRole { name: "low"; query: "low/@data/string()" }
        XmlRole { name: "high"; query: "high/@data/string()" }
        XmlRole { name: "icon_url"; query: "icon/@data/string()" }
        XmlRole { name: "condition"; query: "condition/@data/string()" }
    }
}
```

If you take a closer look at the code inside the XmlRole[11] elements, you will notice that they basically create model items with property-value pairs by mapping them to the specified nodes in the XML tree starting at the node specified in `query`. Like Image[12] and Font[13], XmlListModel[14] provides `status` and `progress` properties, which can be used to track the loading progress and catch the errors. Additionally, there is a `reload()` method which forces the model to query the URL again and load updated data. We will use this later to make sure that the weather forecast is always up-to-date.

7.2 Repeater and Views

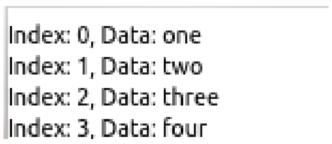
Now we need to visualize the weather data collected in our models. There are various ways to do this in Qt Quick. Most visualization elements are inherited from Flickable[15]:

- ListView[16]
- GridView[17]
- PathView[18]

These elements act like view ports and use delegate elements to draw each model item. Views expect a fixed size to be set via `height` and `width`. The content is shown inside that specified area and can be "flicked" (by default, up and down):

```
import QtQuick 1.1

ListView {
    width: 150; height: 50
    model: ["one", "two", "three", "four", "five"] // or just a number, e.g 10
    delegate: Text { text: "Index: " + model.index + ", Data: " + model.modelData }
}
```

This is how it looks on the screen:



```
Index: 0, Data: one
Index: 1, Data: two
Index: 2, Data: three
Index: 3, Data: four
```

[11]http://qt-project.org/doc/qt-4.8/qml-xmlrole.html
[12]http://qt-project.org/doc/qt-4.8/qml-image.html
[13]http://qt-project.org/doc/qt-4.8/qml-font.html
[14]http://qt-project.org/doc/qt-4.8/qml-xmllistmodel.html
[15]http://qt-project.org/doc/qt-4.8/qml-flickalbe.html
[16]http://qt-project.org/doc/qt-4.8/qml-listview.html
[17]http://qt-project.org/doc/qt-4.8/qml-gridview.html
[18]http://qt-project.org/doc/qt-4.8/qml-pathview.html

The best use case for views is when a large number of model items has to be displayed. Views provide a built-in scrolling or flicking functionality to support ergonomic representation of large data sets. There are also some performance reasons for this as views only load items that become visible and not the entire set of them.

Advanced Use of Views

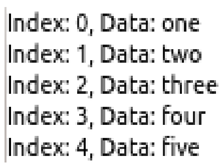
Views provide rich functionality and can be used to create pretty sophisticated UIs. If you are interested, consider reading the The Qt Quick Carousel Tutorial[a]

[a]http://qt-project.org/wiki/Qt_Quick_Carousel

If you have a small number of model items that has to be placed one after the other in a certain order, it makes more sense to use a Repeater[19]. A Repeater[20] creates specified elements for each item in the model. These elements must be placed on the screen by a positioner, for example, Column[21], Grid[22], and so on. The above example can be modified to use a Repeater[23]:

```
import QtQuick 1.1

Column {
    Repeater {
        model: ["one", "two", "three", "four", "five"] // or just a number, e.g 10
        Text { text: "Index: " + model.index + ", Data: " + model.modelData }
    }
}
```

This is how it looks on the screen:



```
Index: 0, Data: one
Index: 1, Data: two
Index: 2, Data: three
Index: 3, Data: four
Index: 4, Data: five
```

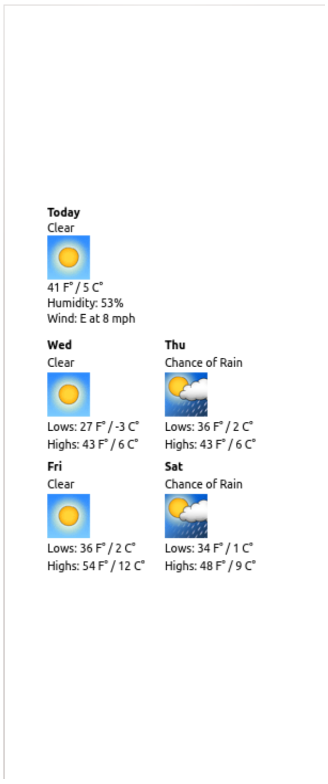
Note that all items are now visible even though the size of the containing element `Column` is not specified. Repeater[24] calculates the size of the elements and Column[25] resizes accordingly. Check the "Presenting Data with Views" article[26] article in Qt Documentation for more details.

We will finish our application by adding two visualization elements, each of which uses its own delegate. We need separate delegates as the current weather conditions data and the forecast data have different structures, which we would like to present in different ways.

But what should be used as visualization elements? It is possible with either a view a or with a Repeater[27]. The `weatherModelForecast` items are displayed by a GridView[28] and it

[19]http://qt-project.org/doc/qt-4.8/qml-repeater.html
[20]http://qt-project.org/doc/qt-4.8/qml-repeater.html
[21]http://qt-project.org/doc/qt-4.8/qml-column.html
[22]http://qt-project.org/doc/qt-4.8/qml-grid.hrml
[23]http://qt-project.org/doc/qt-4.8/qml-repeater.html
[24]http://qt-project.org/doc/qt-4.8/qml-repeater.html
[25]http://qt-project.org/doc/qt-4.8/qml-column.html
[26]http://qt-project.org/doc/qt-4.8/qml-views.html
[27]http://qt-project.org/doc/qt-4.8/qml-repeater.html
[28]http://qt-project.org/doc/qt-4.8/qml-gridview.html

looks like this:



Today
Clear
41 F° / 5 C°
Humidity: 53%
Wind: E at 8 mph

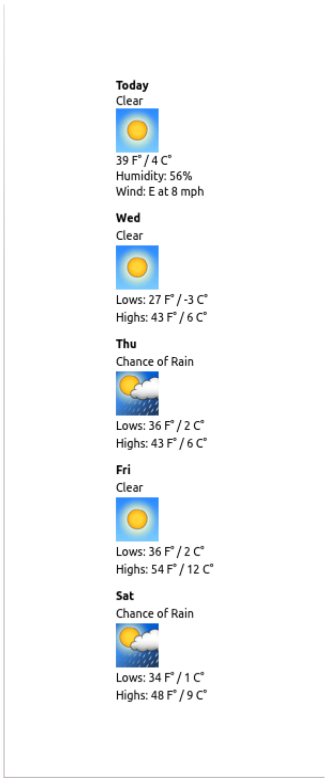
Wed
Clear
Lows: 27 F° / -3 C°
Highs: 43 F° / 6 C°

Thu
Chance of Rain
Lows: 36 F° / 2 C°
Highs: 43 F° / 6 C°

Fri
Clear
Lows: 36 F° / 2 C°
Highs: 54 F° / 12 C°

Sat
Chance of Rain
Lows: 34 F° / 1 C°
Highs: 48 F° / 9 C°

The same items displayed by a Repeater[29] looks like this:



Today
Clear
39 F° / 4 C°
Humidity: 56%
Wind: E at 8 mph

Wed
Clear
Lows: 27 F° / -3 C°
Highs: 43 F° / 6 C°

Thu
Chance of Rain
Lows: 36 F° / 2 C°
Highs: 43 F° / 6 C°

Fri
Clear
Lows: 36 F° / 2 C°
Highs: 54 F° / 12 C°

Sat
Chance of Rain
Lows: 34 F° / 1 C°
Highs: 48 F° / 9 C°

[29] http://qt-project.org/doc/qt-4.8/qml-repeater.html

The `weatherModelCurrent` contains just one item. Due to this, a Repeater[30] is sufficient for displaying it and we keep this approach. The complete source code of the application:

(`Weather/weather.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Item {
    id: root
    property string location: "Munich"
    property bool tempInC: true
    property string baseURL: "http://www.google.com"
    property string dataURL: "/ig/api?weather="
    // some other values: "de", "es", "fi", "fr", "it", "ru"
    property string language: "en"

    width: 300
    height: 700

    function f2C (tempInF) {
        return  (5/9*(tempInF - 32)).toFixed(0)
    }

    XmlListModel {
        id: weatherModelCurrent
        source: baseURL + dataURL + location + "&hl=" + language
        query: "/xml_api_reply/weather/current_conditions"

        XmlRole { name: "condition"; query: "condition/@data/string()" }
        XmlRole { name: "temp_f"; query: "temp_f/@data/string()" }
        XmlRole { name: "humidity"; query: "humidity/@data/string()" }
        XmlRole { name: "icon_url"; query: "icon/@data/string()" }
        XmlRole { name: "wind_condition"; query: "wind_condition/@data/string()" }
    }

    Component {
        id: currentConditionDelegate
        Column {
            Text { text: qsTr("Today"); font.bold: true }
            Text { text: model.condition }
            Image { source: baseURL + model.icon_url }
            Text { text: model.temp_f + " F° / " + f2C (model.temp_f) + " C°" }
            Text { text: model.humidity }
            Text { text: model.wind_condition }
        }
    }

    XmlListModel {
        id: weatherModelForecast
        source: baseURL + dataURL + location  + "&hl=" + language
        query: "/xml_api_reply/weather/forecast_conditions"

        XmlRole { name: "day_of_week"; query: "day_of_week/@data/string()" }
        XmlRole { name: "low"; query: "low/@data/string()" }
        XmlRole { name: "high"; query: "high/@data/string()" }
```

[30] http://qt-project.org/doc/qt-4.8/qml-repeater.html

```
        XmlRole { name: "icon_url"; query: "icon/@data/string()" }
        XmlRole { name: "condition"; query: "condition/@data/string()" }
    }

    Component {
        id: forecastConditionDelegate
        Column {
            spacing: 2
            Text { text: model.day_of_week; font.bold: true }
            Text { text: model.condition }
            Image { source: baseURL + model.icon_url }
            Text { text: qsTr("Lows: ") +
                         model.low +
                         " F° / "
                         + f2C (model.low) + " C°"}
            Text { text: qsTr("Highs: ") +
                         model.high +
                         " F° / " +
                         f2C (model.high) + " C°"}
        }
    }

    Column {
        id: allWeather
        anchors.centerIn: parent
        anchors.margins: 10
        spacing: 10

        Repeater {
            id: currentReportList
            model: weatherModelCurrent
            delegate: currentConditionDelegate
        }

        /* we can use a GridView...*/
        GridView {
            id: forecastReportList
            width: 220
            height: 220
            cellWidth: 110; cellHeight: 110
            model: weatherModelForecast
            delegate: forecastConditionDelegate
        }
        /**/

        /* ..a Repeater
        Repeater {
            id: forecastReportList
            model: weatherModelForecast
            delegate: forecastConditionDelegate
        }
        */
    }

    MouseArea {
        anchors.fill: parent
        onClicked: Qt.quit()
    }
}
```

We do not need this feature in our application, but it is important to mention. In the current version, Qt Quick does not provide direct access to the local file system unless you hard-code a name of the file you would like to load.

A `FolderListModel` C++ plug-in is provided as a lab project in Qt 4.7.4 and higher to provide access to the file system. See “FolderListModel - a C++ model plugin” article[31] in Qt Documentation for details. An earlier version of this plugin is used to develop a text editor in a getting started tutorial[32].

What’s Next?

In the next chapter, we start to combine the clock and the weather forecast features into one application. We make components based on the code we have developed so far and use these components to compose the final application.

[31]http://qt-project.org/doc/qt-4.8/src-imports-folderlistmodel.html

[32]http://qt-project.org/doc/qt-4.8/gettingstartedqml.html

CHAPTER 8

Components and Modules

The next step is to get a version that combines the features of the weather forecast application and the clock application we have already developed. Fortunately, we need not implement those features again or copy the code. We can slightly modify the available applications and re-use them as components. This will be the focus of this chapter.

In the next chapter, we will take another step forward and further enhance the application by adding more components while learning more about how they can be used.

8.1 Creating Components and Collecting Modules

The notion of components in Qt Quick is very simple: any item composed from other elements or components can become a component itself. Components are building blocks for any larger application. When using modules, it is possible to create component collections (libraries).

In order to create a component, you have to create a new file saved as `<NameOfComponent>.qml` with only one root element in it (the same as you would do with an ordinary Qt Quick application). It is important to underline that the name of the file has to start with a capital letter. From now on, the new component will be available under the name `<NameOfComponent>` to all other Qt Quick applications residing in the same directory. Generally, files with a `qml` extension are referred to as QML Documents[1].

When your work progresses, you will probably get many files in the application's folder. Later on, you might even have the need to host components in different versions. This is where Qt Quick *modules* come to the rescue. The first step is to move all components (basically, files) which belong to the same group of functionality in a new folder. Then you need to create a `qmldir` file containing meta-information about these components in that folder. Afterwards this folder becomes a module, which can be imported into your application the same way as you would import the standard Qt Quick elements:

```
import QtQuick 1.1
import "components" 1.0
```

[1]http://qt-project.org/doc/qt-4.8/qdeclarativedocuments.html

See Defining New Components[2] in Qt documentation for more details about components.

If you move the directory with modules, you have to change the path in all QML documents using them. It is also possible to provision modules for global use by any application. See the QML Modules[3] article for more details.

Note: Components can also be developed as C++ plug-ins, which is beyond the scope of this guide. See this article[4] if you are interested.

In some cases, you must define in-line components, for example, when you pass a reference to a component to another element in the same QML file. This is used more frequently for delegates in views. See the main item's code of the final application in the *Integrated Application* (page 58) section.

If you experience problems while importing modules or separate components, set the environment variable `QML_IMPORT_TRACE` to `1` (see "Debugging QML"[5]) for more debugging tips.

Let's take a look at how this is used in our application.

We move `NightClock.qml` to a new folder called `components`, which also contains two new components: `Weather` and `WeatherModelItem`. As mentioned above, we also add a `qmldir` file to describe a new module:

(`components/qmldir` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
Configure 1.0 Configure.qml
NightClock 1.0 NightClock.qml
WeatherModelItem 1.0 WeatherModelItem.qml
Weather 1.0 Weather.qml
```

`Weather` and `WeatherModelItem` include the code from the previous sections in a reworked and extended form. We will take a closer look at the changes in the next section.

8.2 Defining Interfaces and Default Behavior

Moving code into a separate file is just the first step in making a component. You should decide and define how a new component should be used, i.e. which interfaces are provided to change its appearance and behavior. If you have used Qt with C++, you should keep in mind that using components in Qt Quick is different from using classes and libraries in C++. Qt Quick follows JavaScript in this as well. If you use an external component in your item, it is loaded almost as if it had been defined in-line taking over all its properties, handlers, signals, and so on. You can bind that existing property to another value, and use existing signals and handlers. You can also extend that component by declaring additional properties, new signals, handlers, and JavaScript

[2]http://qt-project.org/doc/qt-4.8/qmlreusablecomponents.html#qml-define-components
[3]http://qt-project.org/doc/qt-4.8/qdeclarativemodules.html
[4]http://qt-project.org/doc/qt-4.8/qml-extending.html
[5]http://qt-project.org/doc/qt-4.8/qdeclarativedebugging.html

functions. As all these steps are optional, a component has to have a default appearance and behavior if loaded as is. For example:

```
import QtQuick 1.1
import "components" 1.0

Item {
    id: root
    NightClock {
        id: clock
    }
}
```

This is the same as executing `NightClock` as a stand-alone Qt Quick application.

Let's try this out and create a new application called *clock-n-weather*, which uses three components based on the code we developed earlier:

- `NightClock` - the digital clock
- `WeatherModelItem` - a new weather model that combines the forecast and current weather
- `Weather` - the delegate for drawing the weather data for one day

Most of the code for these components should not be new to you as we discussed them in the previous sections. `NightClock` remained unchanged. We are just binding a few of its properties (for example, `showDate`, `showSeconds`) to values from the `root` item and add new values to customize `NightClock`:

```
...
NightClock {
    id: clock
    height: 80
    width: 160
    showDate: root.showDate
    showSeconds: root.showSeconds
    textColor: Style.onlineClockTextColor
}
...
```

Properties `showDate` and `showSeconds` are configuration parameters, which we are used as property values of the root element. In a later section, we will add a `Configure` component to manage these as well as a few other values.

8.3 Handling Scope

As mentioned above, the `WeatherModelItem` component uses the code from the application in the previous section, but works very differently. The rationale for making this change is to unite the forecast model and the current condition model in one component so that we can use them as a universal weather model:

(`components/WeatherModelItem.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Item {
    id: root
    property alias forecastModel: forecast
    property alias currentModel: current
    property string location: "Munich"
    property bool forceOffline: false
    property string baseURL: "http://www.google.com"
    property string dataURL: "/ig/api?weather="
    property string source: baseURL + dataURL + location.split(' ').join('%20')
    property int interval: 5
    property bool modelDataError: false
    property string statusMessage: ""

    XmlListModel {
        id: forecast
        source: root.source
        query: "/xml_api_reply/weather/forecast_conditions"

        XmlRole { name: "day_of_week"; query: "day_of_week/@data/string()" }
        XmlRole { name: "low"; query: "low/@data/string()" }
        XmlRole { name: "high"; query: "high/@data/string()" }
        XmlRole { name: "condition"; query: "condition/@data/string()" }
        XmlRole { name: "temp_c"; query: "temp_c/@data/string()" }

        onStatusChanged: {
            root.modelDataError = false
            if (status == XmlListModel.Error) {
                root.state = "Offline"
                root.statusMessage = "Error occurred: " + errorString()
                root.modelDataError = true
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Ready) {
                // check if the loaded model is not empty, and post a message
                if (get(0) === undefined) {
                    root.state = "Offline"
                    root.statusMessage = "Invalid location \"" + root.location + "\""
                    root.modelDataError = true
                } else {
                    root.state = "Live Weather"
                    root.statusMessage = "Live current weather is available"
                }
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Loading) {
                root.state = "Loading"
                root.statusMessage = "Forecast data is loading..."
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Null) {
                root.state = "Loading"
                root.statusMessage = "Forecast data is empty..."
                //console.log("Weather Clock: " + root.statusMessage)
            } else {
                root.modelDataError = false
                console.log("Weather Clock: unknown XmlListModel status:" + status)
            }
        }
    }

    XmlListModel {
        id: current
        source: root.source
        query: "/xml_api_reply/weather/current_conditions"

        XmlRole { name: "condition"; query: "condition/@data/string()" }
        XmlRole { name: "temp_c"; query: "temp_c/@data/string()" }

        onStatusChanged: {
            root.modelDataError = false
            if (status == XmlListModel.Error) {
                root.state = "Offline"
                root.statusMessage = "Error occurred: " + errorString()
                root.modelDataError = true
                //console.log("Weather Clock: Error reading current: " + root.statusMess
            } else if (status == XmlListModel.Ready) {
                // check if the loaded model is not empty, and post a message
                if (get(0) === undefined) {
                    root.state = "Offline"
                    root.statusMessage = "Invalid location \"" + root.location + "\""
                    root.modelDataError = true
                } else {
                    root.state = "Live Weather"
                    root.statusMessage = "Live current weather is available"
                }
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Loading) {
                root.state = "Loading"
                root.statusMessage = "Current weather is loading..."
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Null) {
                root.state = "Loading"
                root.statusMessage = "Current weather is empty..."
                //console.log("Weather Clock: " + root.statusMessage)
            } else {
                root.modelDataError = true
                console.log("Weather Clock: unknown XmlListModel status:" + status)
            }
        }
    }

    Timer {
        // note that this interval is not accurate to a second on a full minute
        // since we omit adjustment on seconds like in the clock interval
        // to simplify the code
        interval: root.interval*60000
        running: Qt.application.active && !root.forceOffline
        repeat: true
        onTriggered: {
            current.reload()
            forecast.reload()
        }
    }
}
```

In the code block above, you can see the two models being hosted under one item. We need to access them separately later when using them in views. If the imported `WeatherModelItem` has the `weatherModelItem` id, you might suggest to access them as `weatherModelItem.forecast` and `weatherModelItem.current`. Unfortunately, this will not work. The problem is that the child items of an imported component are by default not visible. A way to solve this problem is to use alias properties to export their `id`:

```
property alias forecastModel: forecast
property alias currentModel: current
```

Our models can now be accessed as `weatherModelItem.forecastModel` and `weatherModelItem.forecastModel`.

Scope and visibility of items, their properties and JavaScript objects are a very important aspect in Qt Quick. We strongly advise reading the QML Scope[6] article in Qt Documentation.

The article referenced above also explains how Qt Quick scope mechanism resolves name conflicts. It is important to keep those rules in mind. A good practice in a daily work is to always qualify the properties you bind to. This makes your application's code easier for others to understand and avoid unexpected side effects. Doing this, you should write for example:

```
Item {
    id: myItem
    ...
    enable: otherItem.visible
}
```

instead of just:

```
Item {
    id: myItem
    ...
    enable: visible
}
```

8.4 Integrated Application

There are other enhancements made to the code from the previous sections which are worth noting.

The most important one is that we've added a timer that triggers the reloading of both models:

```
Timer {
    // note that this interval is not accurate to a second on a full minute
    // as we omit adjustment on seconds like in the clock interval
    // to simplify the code
    interval: root.interval    60000
    running: Qt.application.active && !root.forceOffline
    repeat: true
    onTriggered: {
```

[6] http://qt-project.org/doc/qt-4.8/qdeclarativescope.html

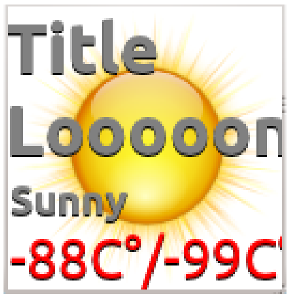
```
            current.reload()
            forecast.reload()
        }
}
```

This timer is similar to the timer in the clock application and updates the weather data in the modes each `root.interval` seconds. `root.interval` is a configuration parameter defined as a property and bound to according value in the parent item.

We also have an updated delegate component for drawing weather conditions. The major change is to use local weather icons instead of loading them from the Internet. This has many advantages such as saving the bandwidth (if the application is running on a mobile device) or just a different look-n-feel which better meets our expectations and is less dependent on external content. We use a very nice set of weather icons from the KDE[7] project. We rename them to match weather condition descriptions and add just a few statements in JavaScript to load them from the local file system:

```
Image {
    id: icon
    anchors.fill: parent
    smooth: true
    fillMode: Image.PreserveAspectCrop
    source: "../content/resources/weather_icons/" +
        conditionText.toLowerCase().split(' ').join('_') + ".png"
    onStatusChanged: if (status == Image.Error) {
                         // we set the icon to an empty image
                         // if we failed to find one
                         source = ""
                         console.log("no icon found for the weather condition: \""
                                     + conditionText + "\"")
    }
}
```

Notice that the `Weather` component can also be started as a stand alone Qt Quick application if needed. It uses the default property values in this case. This is useful for testing the component under various conditions. The weather component looks like this:



The main item of the complete application using components looks like this:

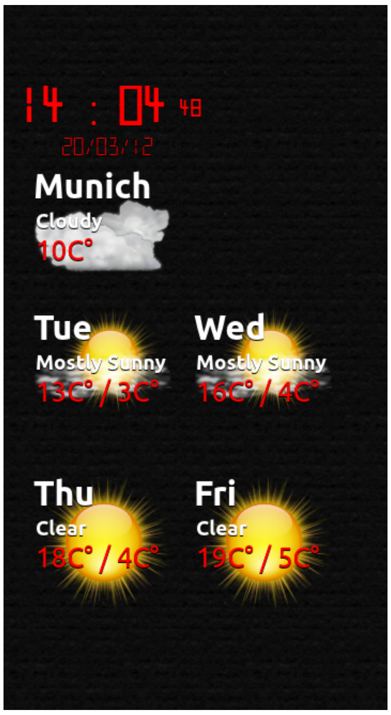
(`clock-n-weather/ClockAndWeather.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

[7] http://www.kde.org

```
import QtQuick 1.1

import "../components" 1.0
import "../js/logic.js" as Logic
import "../js/style.js" as Style

Rectangle {
    id: root
    property string defaultLocation: "Munich"
    property int defaultInterval: 60 // in seconds
    property bool showSeconds: true
    property bool showDate: true

    width: 360
    height: 640

    Image {
        id: background
        source: "../content/resources/background.png"
        fillMode: "Tile"
        anchors.fill: parent
        onStatusChanged: if (background.status == Image.Error)
                             console.log("Background image \"" +
                                         source +
                                         "\" cannot be loaded")
    }

    WeatherModelItem {
        id: weatherModelItem
        location: root.defaultLocation
        interval: root.defaultInterval
    }

    Component {
        id: weatherCurrentDelegate
        Weather {
            id: currentWeatherItem
            labelText: root.defaultLocation
            conditionText: model.condition
            tempText: model.temp_c + "C°"
        }
    }

    Component {
        id: weatherForecastDelegate
        Weather {
            id: forecastWeatherItem
            labelText: model.day_of_week
            conditionText: model.condition
            tempText: Logic.f2C (model.high) +
                      "C° / " +
                      Logic.f2C (model.low) +
                      "C°"
        }
    }

    Column {
        id: clockAndWeatherScreen
        anchors.centerIn: root

        NightClock {
            id: clock
            height: 80
            width: 160
            showDate: root.showDate
            showSeconds: root.showSeconds
            textColor: Style.onlineClockTextColor
        }

        Repeater {
            id: currentWeatherView
            model: weatherModelItem.currentModel
            delegate: weatherCurrentDelegate
        }

        GridView {
            id: forecastWeatherView
            width: 300
            height: 300
            cellWidth: 150; cellHeight: 150
            model: weatherModelItem.forecastModel
            delegate: weatherForecastDelegate
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: Qt.quit()
    }
}
```

We load `WeatherModelItem` as `weatherModelItem` and define two delegates based on the `Weather` component. Then we have a Column[8] with our `NightClock` component, a Repeater[9] with the current weather condition data, and a GridView[10] with the forecast. That's it! This is how it looks on the screen:

[8] http://qt-project.org/doc/qt-4.8/qml-column.html
[9] http://qt-project.org/doc/qt-4.8/qml-repeater.html
[10] http://qt-project.org/doc/qt-4.8/qml-gridview.html



8.5 Further Readings

Using components is a powerful way to extend Qt Quick functionality. Qt Quick components are used to create a full set of UI elements on Symbian[11]. There is a dedicated page about this topic[12] on the Qt Project wiki. The Qt Components project[13] on Gitorious hosts several other implementations, including Qt Components for desktop[14].

What's Next?

In the next chapter, we will focus on the interaction with the user. We will learn how Qt Quick supports this and how we can create simple UI components that suit our needs.

[11] http://doc.qt.digia.com/qtquick-components-symbian-1.1/index.html
[12] http://qt-project.org/wiki/Qt_Quick_Components
[13] http://qt.gitorious.org/qt-components
[14] http://qt.gitorious.org/qt-components/desktop

CHAPTER 9

Interactive UI with Multiple Top-Level Windows

Now that our application is becoming more complete, we need to add some functionality to make it usable on a daily basis. First of all, we need a real button to quit and stop using the entire application window for this. Additionally, we need a new top-level window where we can manage configuration settings. When the user changes settings, the application should verify changes and let the user know if something is wrong. To implement this, we need some basic UI elements.

9.1 A Button

The first thing is to create a button component, which will be used to quit the application, open the configuration window, close it, and so on. Our button should have basic visual parameters and send a signal when it is clicked. The button should also give some visual response that it has received user input. Certainly, a button can have many more features. There may be dozens of approaches for implementing a button, but we'll just describe one which suits our needs.

Our button can be a simple click-sensitive rectangle with rounded corners. In previous sections, we saw that an element can receive mouse events if we include a MouseArea[1] element and let it fill the entire surface of that element. We are going to use this approach for the button. Additionally, our button has to emit a signal notifying relevant parts of the application that it has been clicked. We need to use Qt Quick signals to implement this. Let's take a look at how they work first.

We already got in touch with a related Qt Quick functionality when we saw that it is possible to implement a handler which reacts to property changes, for example, the `status` property of Image[2]:

```
Image {
    id: background
```

[1]http://qt-project.org/doc/qt-4.8/qml-mousearea.html
[2]http://qt-project.org/doc/qt-4.8/qml-image.html

```
    source: "./content/resources/light_background.png"
    fillMode: "Tile"
    anchors.fill: parent
    onStatusChanged: if (background.status == Image.Error)
                         console.log (qsTr("Background image \"") +
                                      source +
                                      qsTr("\" cannot be loaded"))
}
```

Signals are very similar to the property notification changes. Signal handlers work the same, whereas they process a signal explicitly emitted in an item instead of a property change. Signal handlers can also receive signal parameters, which is not the case in property change handlers. Emitting a signal is a function call.

This is how it works for our `Button` component:

(`src/utils/Button.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1
import "../js/style.js" as Style

Rectangle {
    id: root

    property string text: "Button"

    color: "transparent"

    width: label.width + 15
    height: label.height + 10

    border.width: Style.borderWidth
    border.color: pressedColor(Style.penColor)
    radius: Style.borderRadius

    signal clicked (variant mouse)
    signal pressedAtXY (string coordinates)

    function pressedColor (color) {
        return mouseArea.pressed ? Qt.darker(color, 5.0) : color
    }

    function logPresses (mouse) {
        pressedAtXY (mouse.x + "," + mouse.y)
    }

    Component.onCompleted: {
        mouseArea.clicked.connect(root.clicked)
    }

    Text {
        id: label
        anchors.centerIn: parent
        color: pressedColor(Style.penColor)
        text: parent.text
        font.pixelSize: Style.textPixelSize
    }

    MouseArea {
        id: mouseArea
        anchors.fill: parent
        Connections {
            onPressed: logPresses(mouse)
        }
        // this works as well instead of using Connections
        // onPressed: logPresses(mouse)
    }
}
```

`Button` defines two signals: `clicked` and `pressedAtXY`. We only use `clicked` in our application, and `pressedAtXY` has been added for demonstration purposes. Both signals are emitted in different ways. `pressedAtXY` is called from a JavaScript function called as an `onPressed` handler. `clicked` is connected directly to the `clicked` signal of the `mouseArea` item. Both ways have their own use cases. A direct signal-to-signal connection allows simple signal forwarding. This is what is needed in our `Button`, which should behave like a `MouseArea` when processing mouse events. In some other cases, you might have the need to add some additional processing before emitting a signal, like in the `logPresses` function.

A very important point to note here is the naming of signal parameters. If you take a look at the code for `mouseArea` above, you might wonder where the `mouse` parameter comes from. We did not declare it in our application. It actually belongs to the definition of `clicked` signal of the MouseArea[3] element. The same happens with our `pressedAtXY` signal, which defines a `coordinates` parameter. All items using `Button` and processing the `pressedAtXY` signal has to access its parameter under the exact name. For example:

```
Button {
    id: toggleStatesButton
    ...
    onPressedAtXY: {
        console.log ("pressed at: " + coordinates)
    }
}
```

Note that we define the `clicked` signal as:

```
signal clicked (variant mouse)
```

We do this even though `mouse` is of the MouseEvent[4] type (according to the documentation for MouseArea[5]). In the current version of Qt Quick, signal parameters can only be of basic types[6]. This should not concern you as the type is converted to the appropriate type when it arrives.

For more details about using signals in Qt Quick, see the QML Signal and Handler Event

[3]http://qt-project.org/doc/qt-4.8/qml-mousearea.html
[4]http://qt-project.org/doc/qt-4.8/qml-mouseevent.html
[5]http://qt-project.org/doc/qt-4.8/qml-mousearea.html
[6]http://qt-project.org/doc/qt-4.8/qdeclarativebasictypes.html

System[7] article in Qt documentation. You should also check the documentation of MouseArea[8] as well as the QML Mouse Events[9] article to discover more possibilities such as getting other mouse events, tracing hovering, and implementing drag-drop.

As any proper button, our `Button` should provide some visual feedback when it is clicked. We do this by tweaking its colors a bit. We have a small JavaScript function which modifies the color value to a new, *pressed* value. It makes the color darker in our case:

```
function pressedColor (color) {
    return mouseArea.pressed ? Qt.darker(color, 5.0) : color
}
```

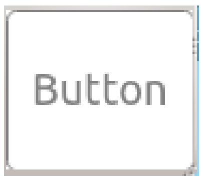
We are going to toggle the color of the button border and of its label text. We bind the return value of this function to the border of the button:

```
border.color: pressedColor(Style.penColor)
```

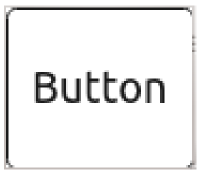
Then we bind it to a color property of its label text:

```
color: pressedColor(Style.penColor)
```

That's it! This is how our `Button` looks when unpressed:



and pressed:



9.2 A Simple Dialog

The `Dialog` is another utility component that we need. We use it to notify the user about critical situations. Our `Dialog` is very simple. It pops up on top of another element and just displays a text message that must be confirmed by clicking the *OK* button. This is the code for the new `Dialog` component:

(`src/utils/Dialog.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1
import "../js/style.js" as Style
```

[7]http://qt-project.org/doc/qt-4.8/qmlevents.html
[8]http://qt-project.org/doc/qt-4.8/qml-mousearea.html
[9]http://qt-project.org/doc/qt-4.8/mouseevents.html

```
Rectangle {
    id: root

    property string message: "Error! This is a long message with details"

    width: 100
    height: 40

    color: Style.backgroundColor
    border.color: Style.penColor
    border.width: Style.borderWidth
    radius: Style.borderRadius

    visible: true

    function show(text) {
        root.message = text;
        root.visible = true;
    }

    function hide() {
        root.visible = false;
    }

    Text {
        id: messageText
        anchors.top: parent.top
        anchors.topMargin: Style.baseMargin
        anchors.left: parent.left
        anchors.right: parent.right
        horizontalAlignment: Text.AlignHCenter
        wrapMode: "WordWrap"
        text: root.message
        font.pixelSize: Style.textPixelSize
        color: Style.penColor
        onPaintedHeightChanged: {
            root.height = messageText.paintedHeight + okButton.height + 3*Style.baseMarg
        }
    }

    Button {
        id: okButton
        text: qsTr("OK")
        anchors.top: messageText.bottom
        anchors.topMargin: Style.baseMargin
        anchors.horizontalCenter: parent.horizontalCenter
        onClicked: root.hide()
    }
}
```

The `Dialog` is used by adding it as a child item to another element where it will pop-up from:

```
Item {
  id: root
  ...

  Dialog {
      id: errorDialog
      width: root.width
      anchors.centerIn: parent
      z: root.z+1
      visible: false
  }
  ...

  Button {
    id: exitButton
    ...
    onClicked: {
      ...
      errorDialog.show (qsTr("The location cannot be empty"));
      ...
    }
  }
  ...
}
```

When loaded, the `Dialog` initially stays invisible. It goes on top of its parent (`root` in the code segment above). `z: root.z+1` does this trick. We bind its `z` property to a value which is always higher than the value of `root.z`. Later, we call `show` with a message to be displayed. `show` makes the `Dialog` visible and stores the message text to be displayed. When the user clicks the *OK* button, the `Dialog` hides itself again.

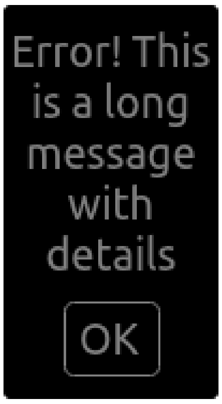
Note: The TabWidget Example[10] in Qt documentation shows another approach toward dynamically showing and hiding elements on top of others.

Our `Dialog` has a few other features which are useful to know. In order to use the screen space efficiently, it copies the width from its parent. We also set the `messageText` property, `wrapMode` to the `WordWrap` value. When the `Dialog` opens with a long message text, the message wraps it to the `Dialog` width. The `messageText` element changes the height of the root `Dialog` when its height has changed due to wrapping:

```
Rectangle {
  id: root
  ...
  Text {
    id: messageText
    ...
    onPaintedHeightChanged: {
      root.height = messageText.paintedHeight +
                    okButton.height +
                    3    Style.baseMargin
    }
  ...
}
```

This is how it looks on the screen:

[10] http://qt-project.org/doc/qt-4.8/declarative-ui-components-tabwidget.html



9.3 A Checkbox

We can use the Text Input[11] element to get text or digit based user input, but we need something else for on-off type of settings. Usually, this is done using the checkbox UI elements. There is no checkbox element in Qt Quick, and we are going to make it from scratch. It is not a problem at all as we can easily create one using Qt Quick. This is the complete code for our new `CheckBox` component:

(`src/utils/CheckBox.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

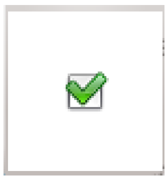
```
import QtQuick 1.1

Item {
    id: root
    property bool checked: true
    // we should pre-set the size to get it working perperly in a positioner
    width: checkBox.width
    height: checkBox.height

    Image {
        id: checkBox
        source: root.checked ?
                    "../content/resources/checkbox.png" :
                    "../content/resources/draw-rectangle.png"
        Keys.onPressed: {
            if (event.key == Qt.Key_Return ||
                    event.key == Qt.Key_Enter ||
                    event.key == Qt.Key_Space)
                root.checked = !root.checked;
        }
        MouseArea {
            anchors.fill: parent
            onClicked: {
                root.checked = !root.checked;
            }
        }
    }
```

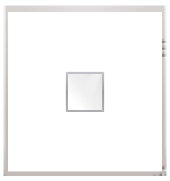
[11] http://qt-project.org/doc/qt-4.8/qml-textinput.html

```
    //onValueChanged: console.log ("value: " + root.value)
}
```

Our `CheckBox` is based on Item[12]. It extends it just by one boolean property called `checked`. If the box is checked, `checked` is `true`. Otherwise it is `false`. The entire visual implementation of the `CheckBox` consists of two images which are flipped back and forth. This is done by binding the `source` property of the `checkBox` Image[13] item to a checkbox image or to an image of a normal rectangle depending on the `checked` property.

This is how our `CheckBox` looks on the screen when checked and unchecked:





Further on, there is a section of code, which includes the keyboard navigation handling. This topic will be discussed in the next section.

9.4 Handling Keyboard Input and Navigation

Another important aspect of interaction with the user is handling the keyboard input and navigation. We will explore this while walking through the implementation of the `Configure` component based on the new UI components we introduced earlier.

Meanwhile, we have several hard-coded property values which actually should be changeable by the user:

- Location name for the weather forecast
- Time interval in which the weather data should be updated
- Turning off the seconds and date display to make the clock more compact

The name and interval properties require a text input field, whereas the last one can be implemented using the `Checkbox`.

Text input is straightforward: Qt Quick provides the Text Input[14] element for this. We use it to get a new value for the forecast location and new value for the forecast update interval. The Text Input[15] element binds the captured keyboard input to the `text` property. When we load this element, we preset according to `text` properties with the `locationTextInput` and

[12]http://qt-project.org/doc/qt-4.8/qml-item.html
[13]http://qt-project.org/doc/qt-4.8/qml-image.html
[14]http://qt-project.org/doc/qt-4.8/qml-textinput.html
[15]http://qt-project.org/doc/qt-4.8/qml-textinput.html

`forecastUpdateInterval` values to display the current settings to the user. Users can start editing and we do not need to take care of any details for text input handling:

```
...
TextInput {
    id: locationTextInput
    ...
    width: controlElements.width - locationTextInput.x - controlElements.spacing
    text: locationText
    focus: true
}
...
TextInput {
    id: updateTextInput
    ...
    text: forecastUpdateInterval
    maximumLength: 3
    // we use IntValidator just to filter the input. onAccepted is not used here
    validator: IntValidator{bottom: 1; top: 999;}
}
...
```

The code above has a few things on top.

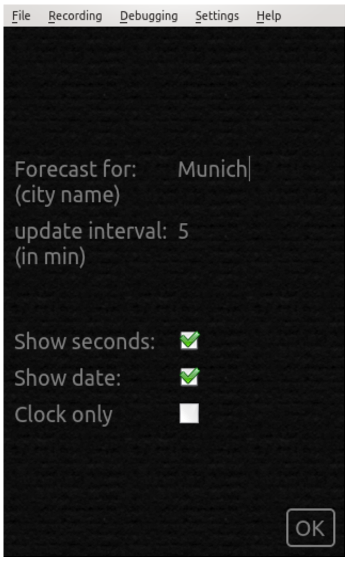
`updateTextInput` uses a validator to limit the length of the text and ensure that we get digits in a proper range.

Location names do not need a validator, but they need something to handle text input which is longer than just a few digits in `updateTextInput`. This can be achieved limiting the `width` to ensure that a long text does not leave the boundaries of the top-level item. If we do not do this and keep `width` undefined, Text Input[16] will expand, follow the entered text and at some time leave the visual boundaries.

Note: If you have a multi-line text that needs to be edited by the user, you can use the Text Edit[17] element.

`locationTextInput` receives the keyboard focus explicitly, because we set `focus` to `true`. When `Configure` is loaded, the user can start changing the location name:

[16] http://qt-project.org/doc/qt-4.8/qml-textinput.html
[17] http://qt-project.org/doc/qt-4.8/qml-textedit.html



Forecast for: Munich
(city name)
update interval: 5
(in min)
Show seconds: ✓
Show date: ✓
Clock only

The elements Text Input[18] and our new `CheckBoxes` react to mouse clicks. How can the user navigate from one input element to another if we would like to support navigation with keyboard in addition to the mouse? What should we do if we need to enable keyboard input in `CheckBoxes` as well? .

Qt Quick provides key navigation and raw key processing for these cases. Let's take a look at key navigation first.

These are the changes in the code for our two Text Input[19] elements to support key navigation:

```
TextInput {
    id: locationTextInput
    ...
    focus: true
    KeyNavigation.up: offlineCheckBox
    KeyNavigation.down: updateTextInput
}
...
TextInput {
    id: updateTextInput
    ...
    KeyNavigation.up: locationTextInput
    KeyNavigation.down: secondsCheckBox
}
```

`locationTextInput` explicitly pulls the focus by setting its `focus property to ``true`. The Key Navigation[20] items provide attached properties, which monitor key presses and move of the input focus from one element to another. Key Navigation[21] is a big help in our case where we have many elements and need to organize the movement of the input focus in a certain way.

In the code sample above, the input focus is moved from the `locationTextInput` item to the `updateTextInput` item if the down* arrow key is pressed. The focus goes back from

[18]http://qt-project.org/doc/qt-4.8/qml-textinput.html
[19]http://qt-project.org/doc/qt-4.8/qml-textinput.html
[20]http://qt-project.org/doc/qt-4.8/qml-keynavigation.html
[21]http://qt-project.org/doc/qt-4.8/qml-keynavigation.html

updateTextInput to locationTextInput if the user presses the *up* key and so on. We add such statements to all relevant elements in the Configure component.

While processing user input, you sometimes need to capture particular keys. This is the case with our Checkboxes. Working with desktop applications, users have learned that it is possible to toggle a check box with the space key*. We should implement this feature in our application.

This is where the Keys[22] items can be used. It is basically a kind of signal sender for almost every key on the keyboard. Its signals have KeyEvent[23] as a parameter, and it contains detailed information about the key pressed. We use Keys[24] in our checkboxes. The code segment in the previous section uses the attached Keys.onPressed property, which toggles the Checkbox state on *Return*, *Enter* and *Space* keys.

More details about keyboard input processing is available in the Keyboard Focus in QML[25] article in Qt Documentation.

By now we have got all input elements and can process user input. One step is still needed to finish our Configure component. This is a verification and storing of the new values.

When the user clicks the exitButton, we need to check the new setting values and pass them to the application if they are ok. This is also a place where we use our Dialog to inform the user that the new values are not OK if needed. In this case, the Configure does not close and stays open until the user provides the correct values. See the onClicked handler code for exitButton to learn how this is achieved:

(src/components/Configure.qml in qt_quick_app_dev_intro_src.zip, see *Downloads* (page 2) section)

```
import QtQuick 1.1
import "../utils" 1.0
import "../js/style.js" as Style

Rectangle {
    id: root
    property bool showSeconds: true
    property bool showDate: true
    property int forecastUpdateInterval: 5 // minutes
    property string locationText: "Munich"
    property bool forceOffline: false

    width: 320
    height: 480

    Image {
        id: background
        source: Style.backgroundImage
        fillMode: "Tile"
        anchors.fill: parent
    }
```

[22]http://qt-project.org/doc/qt-4.8/qml-keys.html
[23]http://qt-project.org/doc/qt-4.8/qml-keyevent.html
[24]http://qt-project.org/doc/qt-4.8/qml-keys.html
[25]http://qt-project.org/doc/qt-4.8/qdeclarativefocus.html

```
Grid {
    id: controlElements
    spacing: 10
    columns: 2
    anchors.left: root.left
    anchors.leftMargin: spacing
    anchors.verticalCenter: root.verticalCenter
    anchors.right: root.right

    Text {
        id: locationLabel
        text: qsTr("Forecast for:<br>(city name)")
        color: locationTextInput.focus?
                   Qt.lighter(Style.penColor) : Style.penColor
        font.pixelSize: Style.textPixelSize
    }

    TextInput {
        id: locationTextInput
        width: controlElements.width - locationTextInput.x - controlElements.spacing
        text: locationText
        font.pixelSize: Style.textPixelSize
        color: Style.penColor
        focus: true
        KeyNavigation.up: offlineCheckBox
        KeyNavigation.down: updateTextInput
    }

    Text {
        id: updateLabel
        height: 90
        text: qsTr("update interval: <br>(in min)")
        color: updateTextInput.focus?
                   Qt.lighter(Style.penColor) : Style.penColor
        font.pixelSize: Style.textPixelSize
    }

    TextInput {
        id: updateTextInput
        width: locationTextInput.width
        text: forecastUpdateInterval
        font.pixelSize: Style.textPixelSize
        color: Style.penColor
        maximumLength: 3
        // we use IntValidator just to filter the input
        // onAccepted is not used here
        validator: IntValidator{bottom: 1; top: 999;}
        KeyNavigation.up: locationTextInput
        KeyNavigation.down: secondsCheckBox
    }

    Text {
        id: secondsLabel
        text: qsTr("Show seconds:")
        color: secondsCheckBox.focus?
                   Qt.lighter(Style.penColor) : Style.penColor
        font.pixelSize: Style.textPixelSize
    }

    CheckBox {
        id: secondsCheckBox
        checked: showSeconds
        KeyNavigation.up: updateTextInput
        KeyNavigation.down: dateCheckBox
    }

    Text {
        id: dateLabel
        text: qsTr("Show date:")
        color: dateCheckBox.focus?
                   Qt.lighter(Style.penColor) : Style.penColor
        font.pixelSize: Style.textPixelSize
    }

    CheckBox {
        id: dateCheckBox
        checked: showDate
        KeyNavigation.up: secondsCheckBox
        KeyNavigation.down: offlineCheckBox
    }

    Text {
        id: offlineLabel
        text: qsTr("Clock only")
        color: offlineCheckBox.focus?
                   Qt.lighter(Style.penColor) : Style.penColor
        font.pixelSize: Style.textPixelSize
    }

    CheckBox {
        id: offlineCheckBox
        checked: forceOffline
        KeyNavigation.up: secondsCheckBox
        KeyNavigation.down: locationTextInput
    }
}

Dialog {
    id: errorDialog
    width: root.width
    anchors.centerIn: parent
    z: root.z+1
    visible: false
}

Button {
    id: exitButton
    text: qsTr("OK")
    anchors.right: parent.right
    anchors.bottom: parent.bottom
    anchors.margins: 10
    onClicked: {
        // update interval and location cannot be empty
        // update interval cannot be zero
```

```
                if (updateTextInput.text == "" || updateTextInput.text == 0)
                    errorDialog.show (qsTr("The update interval cannot be empty"))
                else if (locationTextInput.text == "")
                    errorDialog.show (qsTr("The location cannot be empty"))
                else {
                    forecastUpdateInterval = updateTextInput.text
                    root.locationText = locationTextInput.text
                    root.visible = false
                }
                // update check box relevant settings
                root.showSeconds = secondsCheckBox.checked
                root.showDate = dateCheckBox.checked
                root.forceOffline = offlineCheckBox.checked
            }
        }
}
```

What's Next?

You've probably noticed that the `offlineCheckBox` item with the *magical* `forceOffline` setting associated with it. This setting is new. It is used to toggle the states in the next version of our application which will be the topic for the next chapter. We also will take a look at animations in Qt Quick and use them to implement some nice effects in the final version of our application.

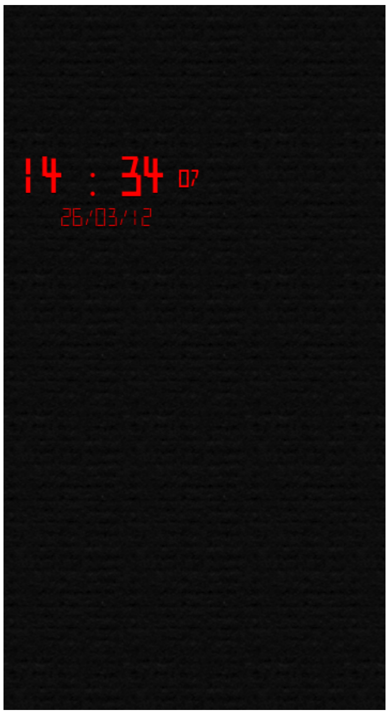
CHAPTER 10

UI Dynamics and Dynamic UI

In the previous sections, we learned how to add items while developing an application and make them invisible when needed. What should we do if we'd like our applications to look totally different depending on how the data and user input changes? These changes might be aimed toward modifying more then just visibility. This might be quite complex with Qt Quick as we need to change all related elements. How do we make dynamic changes in the application UI visually appealing or even make them a part of the user experience? We have not covered this at all.

10.1 Using States

Access to the network is essential for the weather related part of our application in the current version. It visualizes data received from the internet. If your computer is offline and you start the `clock-n-weather` application in `qt_quick_app_dev_intro_src.zip` (see *Downloads* (page 2) section), you will see just the clock and a lot of empty space around it:



This is because `WeatherModelItem` failed to get the weather data. Due to this, there are no model items to display. If you use this application on a notebook or on a mobile device, this situation might occur very frequently. It would be great if our application would be able to handle situations when the network is down. We can accomplish this by using the State[1] item provided by Qt Quick.

Each item in Qt Quick has a `state` property which holds the name of the current state. There is also a `states` property which is a list of States[2]. This property contains all states known for that item. Each of the States[3] in the list has a string name and defines a set of property values. If required, it can even contain some script code, which is executed when that state becomes the current one. An item can be set to a state just by assigning the name of a selected state to the `state` property. See the documentation for State[4] and QML States[5] for more details.

We will add three states to the main item of our application:

- Offline* - It is an initial state in the startup phase. It is also applied if there is no network connection or the if application should stay offline
- Loading* - A network connection is available, but `WeatherModelItem` is still loading weather data. This state is useful on slow network connections (on mobile devices for example).
- Live Weather* - Updated weather data is available and displayed.

In the Offline* and *Loading* states, the application should show just the clock in a larger size in the middle of the screen. When *Live Weather* is active, the application should show the weather data as well.

[1]http://qt-project.org/doc/qt-4.8/qml-state.html
[2]http://qt-project.org/doc/qt-4.8/qml-state.html
[3]http://qt-project.org/doc/qt-4.8/qml-state.html
[4]http://qt-project.org/doc/qt-4.8/qml-state.html
[5]http://qt-project.org/doc/qt-4.8/qdeclarativestates.html

As our new states are so closely related to the `status` of the `WeatherModelItem`, we just bind them directly. The `WeatherModelItem` does not define any real states. We hijack its `states` property to store *Offline*, *Loading* and *Live Weather* values depending on the `status` of the `current` or `forecast` models:

(`src/components/WeatherModelItem.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

Item {
    id: root
    property alias forecastModel: forecast
    property alias currentModel: current
    property string location: "Munich"
    property bool forceOffline: false
    property string baseURL: "http://www.google.com"
    property string dataURL: "/ig/api?weather="
    property string source: baseURL + dataURL + location.split(' ').join('%20')
    property int interval: 5
    property bool modelDataError: false
    property string statusMessage: ""

    XmlListModel {
        id: forecast
        source: root.source
        query: "/xml_api_reply/weather/forecast_conditions"

        XmlRole { name: "day_of_week"; query: "day_of_week/@data/string()" }
        XmlRole { name: "low"; query: "low/@data/string()" }
        XmlRole { name: "high"; query: "high/@data/string()" }
        XmlRole { name: "condition"; query: "condition/@data/string()" }
        XmlRole { name: "temp_c"; query: "temp_c/@data/string()" }

        onStatusChanged: {
            root.modelDataError = false
            if (status == XmlListModel.Error) {
                root.state = "Offline"
                root.statusMessage = "Error occurred: " + errorString()
                root.modelDataError = true
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Ready) {
                // check if the loaded model is not empty, and post a message
                if (get(0) === undefined) {
                    root.state = "Offline"
                    root.statusMessage = "Invalid location \"" + root.location + "\""
                    root.modelDataError = true
                } else {
                    root.state = "Live Weather"
                    root.statusMessage = "Live current weather is available"
                }
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Loading) {
                root.state = "Loading"
                root.statusMessage = "Forecast data is loading..."
                //console.log("Weather Clock: " + root.statusMessage)
            } else if (status == XmlListModel.Null) {
```

```
            root.state = "Loading"
            root.statusMessage = "Forecast data is empty..."
            //console.log("Weather Clock: " + root.statusMessage)
        } else {
            root.modelDataError = false
            console.log("Weather Clock: unknown XmlListModel status:" + status)
        }
    }
}

XmlListModel {
    id: current
    source: root.source
    query: "/xml_api_reply/weather/current_conditions"

    XmlRole { name: "condition"; query: "condition/@data/string()" }
    XmlRole { name: "temp_c"; query: "temp_c/@data/string()" }

    onStatusChanged: {
        root.modelDataError = false
        if (status == XmlListModel.Error) {
            root.state = "Offline"
            root.statusMessage = "Error occurred: " + errorString()
            root.modelDataError = true
            //console.log("Weather Clock: Error reading current: " + root.statusMess
        } else if (status == XmlListModel.Ready) {
            // check if the loaded model is not empty, and post a message
            if (get(0) === undefined) {
                root.state = "Offline"
                root.statusMessage = "Invalid location \"" + root.location + "\""
                root.modelDataError = true
            } else {
                root.state = "Live Weather"
                root.statusMessage = "Live current weather is available"
            }
            //console.log("Weather Clock: " + root.statusMessage)
        } else if (status == XmlListModel.Loading) {
            root.state = "Loading"
            root.statusMessage = "Current weather is loading..."
            //console.log("Weather Clock: " + root.statusMessage)
        } else if (status == XmlListModel.Null) {
            root.state = "Loading"
            root.statusMessage = "Current weather is empty..."
            //console.log("Weather Clock: " + root.statusMessage)
        } else {
            root.modelDataError = true
            console.log("Weather Clock: unknown XmlListModel status:" + status)
        }
    }
}

Timer {
    // note that this interval is not accurate to a second on a full minute
    // since we omit adjustment on seconds like in the clock interval
    // to simplify the code
    interval: root.interval*60000
    running: Qt.application.active && !root.forceOffline
```

```
        repeat: true
        onTriggered: {
            current.reload()
            forecast.reload()
        }
    }
}
```

The actual states are introduced in the main item, `WeatherClock`. This item gets two new child items holding all elements to be displayed in states with different visualization:

- `clockScreen` item - shows a bigger clock when the main item is in *Offline* or *Loading* state
- `weatherScreen` item - shows clock and the weather forecast during the *Live Weather* state, which is basically the same as we had in the `clock-n-weather` application.

As a final step, we just bind the states of `WeatherClock` to the values of the `WeatherModelItem` state:

```
...
Rectangle {
    id: root
    ...
    state: forceOffline ? "Offline" : weatherModelItem.state
    ...
    states: [
        State {
            name: "Offline"
            PropertyChanges {target: clockScreen; visible: true}
            PropertyChanges {target: weatherScreen; visible: false}
        },
        State {
            name: "Live Weather"
            PropertyChanges {target: clockScreen; visible: false}
            PropertyChanges {target: weatherScreen; visible: true}
        },
        State {
            name: "Loading"
            PropertyChanges {target: clockScreen; visible: true}
            PropertyChanges {target: weatherScreen; visible: false}
            PropertyChanges {target: busyIndicator; on: true}
        }
    ]
...
}
```

Our State[6] definitions contain PropertyChanges[7] items which change the visibility of our new screens and turn on the `busyIndicator` in the Loading* state.

The *Loading* state might be active for quite some time. If the clock does not show seconds, the whole application might appear as if it were hanging. We need a animated busy indicator to show the user that the application is still running. The Qt example RSS News Reader[8] provides

[6]http://qt-project.org/doc/qt-4.8/qml-state.html
[7]http://qt-project.org/doc/qt-4.8/qml-propertychanges.html
[8]http://qt-project.org/doc/qt-4.8/demos-declarative-rssnews-qml-rssnews-content-rssfeeds-qml.html

a very nice one. We can use that with minor modifications. Our `busyIndicator` becomes visible in the *Loading* state and informs the user that the application is processing data in the background.

You may have noticed that we use the new `forceOffline` setting here, which was first spotted in the last chapter. If `forceOffline` is set to `true`, the application stays in the *Offline* state regardless of changes in `weatherModelItem`.

If we now change states, changes occur instantly. The application would look more attractive if there were transitions and animation effects applied during state changes. We will take a look at this in the next section.

10.2 Adding Animations

Animations are not only useful for visual effects. They can also serve as a base for features that could be difficult to get done by other means (for example, our busy indicator mentioned in the last section). Qt Quick provides a very rich animation framework that is simple to use. Covering it in great detail is beyond the scope of this guide, but we can spend some time understanding what animations do and how to start using them.

Generally, all animations manipulate one or more properties of an element, thereby modifying its visual appearance. This modification can have various dynamics and run in various time spans. There can be numerous animations running in parallel or sequentially applied to the same or to different elements. You can start an animation explicitly or implicitly upon a property change. You can also permanently assign an animation to a property so that an animation starts as soon as a property changes. Although there is a generic Animation[9] element, most of the time, you will probably use one of the predefined animation elements[10] provided by Qt Quick. It's very easy to add animations to an application. The major challenge is to find out which animations to use and how to use them to compose the required visual effect.

Animations are very related to Transitions[11], which defines how an element is transformed from one State[12] to another. In most cases, a transition includes an animation.

Qt documentation provides an overview of all animations and transitions, and provides details about using them in the QML Animation and Transitions[13] article.

The code segment below shows two transitions between the *Offline* and *Live Weather* states in our application:

```
transitions: [
    Transition {
        from: "Offline"
        to: "Live Weather"
        PropertyAnimation {
            target: weatherScreen
            property: "opacity"
```

[9]http://qt-project.org/doc/qt-4.8/qml-animation.html
[10]http://qt-project.org/doc/qt-4.8/qml-animation-transition.html
[11]http://qt-project.org/doc/qt-4.8/qml-transition.html
[12]http://qt-project.org/doc/qt-4.8/qml-state.html
[13]http://qt-project.org/doc/qt-4.8/qdeclarativeanimation.html

```
            from: 0
            to: 1
            easing.type: Easing.Linear
            duration: 5000
        }
    },
    Transition {
        from: "Live Weather"
        to: "Offline"
        PropertyAnimation {
            target: clockScreen
            property: "opacity"
            from: 0
            to: 1
            easing.type: Easing.Linear
            duration: 5000
        }
    }
]
```

The state changes swap the visibility of the off-line view and the full view with weather data. On top of this, we add an animation which changes the `opacity` property. This fades the screen out letting it disappear fully in 5 seconds.

Note: Theoretically, a slight flickering might be visible on the screen in the beginning of transitions as the target element becomes fully visible first and immediately after this its opacity is turned to `0` in the beginning of the animation.

The functionality of our *busy* indicator is completely based on animations! There is almost no other code in its implementation:

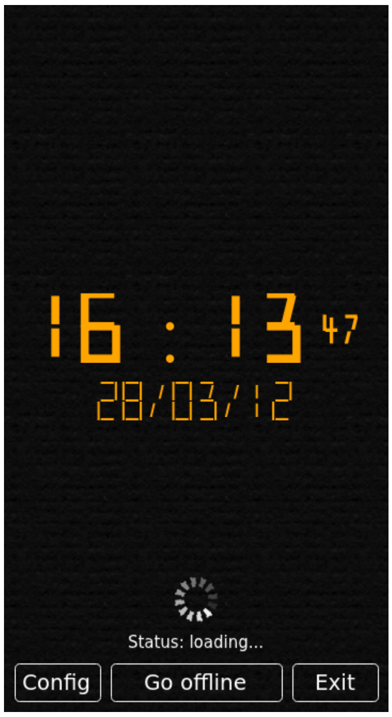
(`utils/BusyIndicator.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
// This is taken from the "RSS News" demo provided in Qt
// The original code has been modified to adapt to the application structure

import QtQuick 1.1

Image {
    id: root
    property bool on: false

    source: "../content/resources/busy.png"
    visible: root.on

    NumberAnimation on rotation {
        running: root.on; from: 0; to: 360; loops: Animation.Infinite; duration: 1200
    }
}
```

We load `BusyIndicator` as follows:

```
// it is off and invisible by default
BusyIndicator {
    id: busyIndicator
    anchors.horizontalCenter: root.horizontalCenter
    anchors.bottom: statusText.top
    anchors.margins: 10
}
```

And this is how it looks in our application when it starts up:



Another animation is used to implement a visual effect on the clock and weather items in the reworked main item of our application. This is discussed in the next section.

10.3 Supporting the Landscape Mode

If our application is to run on a mobile device, it should have a layout of the `clockScreen` and `weatherScreen` tailored to the landscape display orientation. We do not need many changes in `clockScreen` for this, as it contains only one item. Changes in `weatherScreen` might be larger...

An interesting approach toward simplifying the implementation is to use Flow[14] instead of the previously used Column[15]. Flow[16] arranges its children dynamically depending on its own size. If needed, it wraps children into the appropriate rows and columns.

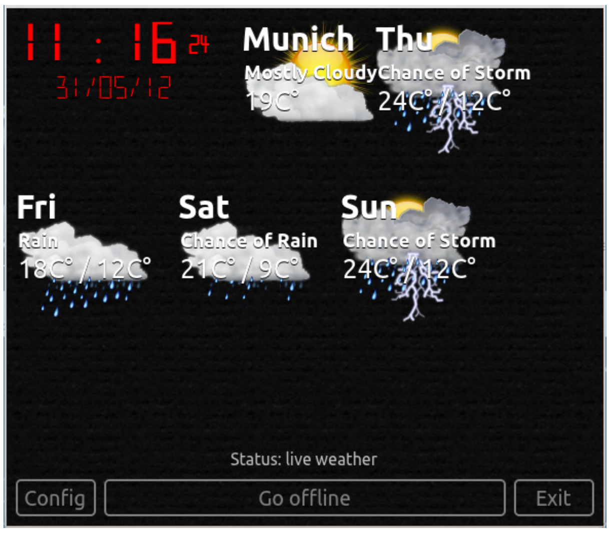
Flow[17] has one more cool feature. This is the `move` property where we can define a

[14]http://qt-project.org/doc/qt-4.8/qml-flow.html
[15]http://qt-project.org/doc/qt-4.8/qml-column.html
[16]http://qt-project.org/doc/qt-4.8/qml-flow.html
[17]http://qt-project.org/doc/qt-4.8/qml-flow.html

Transition[18], which is applied when the children in a Flow[19] start moving. We use a NumberAnimation[20] applied to the coordinates of the children and select a bounce effect (`Easing.OutBounce`) for `easing.type`:

```
...
move: Transition {
    NumberAnimation {
        properties: "x,y"
        duration: 500
        easing.type: Easing.OutBounce
    }
}
...
```

This is how our application looks on the screen if we resize the main window:



10.4 Finalizing the Main Item

We need to rework on the main item to add a few new features. You've seen parts of the related code in this and in the earlier sections. Let's put them all together and take a look at some other details.

First, we take the code from the main item, `ClockAndWeather.qml` (see `clock-n-weather/ClockAndWeather.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section) and add animations and transitions as discussed in this chapter.

Additionally, the reworked main item gets three buttons and a status text at the bottom of the screen.

Clicking this `exitButton` is now used to quit the application. Clicks inside the `root` items are not used for this anymore.

[18]http://qt-project.org/doc/qt-4.8/qml-transition.html
[19]http://qt-project.org/doc/qt-4.8/qml-flow.html
[20]http://qt-project.org/doc/qt-4.8/qml-numberanimation.html

The `toggleStatesButton` allows the user to force the *Offline* state. This is useful to use the screen space for a bigger clock by hiding the weather forecast. It prevents regular data transfer over Internet as well.

The `configureButton` displays the the `configure` element, which holds and manipulates the configuration parameters. The main item just binds them to the appropriate properties of other items. This implements a kind of global application state. We will discuss alternative solutions for this in the last chapter.

The status text is updated upon changes to the states.

The complete code of the new main item looks like this:

(`WeatherClock/WeatherClock.qml` in `qt_quick_app_dev_intro_src.zip`, see *Downloads* (page 2) section)

```
import QtQuick 1.1

import "../utils" 1.0
import "../components" 1.0
import "../js/style.js" as Style
import "../js/logic.js" as Logic

Rectangle {
    id: root
    property string defaultLocation: configure.locationText
    property int defaultInterval: configure.forecastUpdateInterval
    property bool showSeconds: configure.showSeconds
    property bool showDate: configure.showDate
    property bool forceOffline: configure.forceOffline
    state: forceOffline ? "Offline" : weatherModelItem.state

    width: 360
    height: 640

    onStateChanged: {
        if (state == "Offline")
            statusText.showStatus ("offline");
        else if (state == "Loading")
            statusText.showStatus ("loading...");
        else if (state == "Live Weather")
            statusText.showStatus ("live weather");
    }

    Image {
        id: background
        source: Style.backgroundImage
        fillMode: "Tile"
        anchors.fill: parent
        onStatusChanged: if (background.status == Image.Error)
                             console.log("Background image \"" +
                                         source +
                                         "\" cannot be loaded")
    }

    Dialog {
        id: errorDialog
        width: root.width
        anchors.centerIn: parent
        z: root.z+1
        visible: false
    }

    WeatherModelItem {
        id: weatherModelItem
        location: root.defaultLocation
        interval: root.defaultInterval
        forceOffline: root.forceOffline

        onModelDataErrorChanged: {
            if (weatherModelItem.modelDataError)
                errorDialog.show(weatherModelItem.statusMessage)
        }
    }

    Component {
        id: weatherCurrentDelegate
        Weather {
            id: currentWeatherItem
            labelText: root.defaultLocation
            conditionText: model.condition
            tempText: model.temp_c + "C°"
        }
    }

    Component {
        id: weatherForecastDelegate
        Weather {
            id: forecastWeatherItem
            labelText: model.day_of_week
            conditionText: model.condition
            tempText: Logic.f2C (model.high) +
                      "C° / " +
                      Logic.f2C (model.low) +
                      "C°"
        }
    }

    NightClock {
        id: clockScreen
        height: 130
        anchors.centerIn: root
        showDate: root.showDate
        showSeconds: root.showSeconds
        textColor: Style.offlineClockTextColor
    }

    Flow {
        id: weatherScreen
        width: root.width
        height: root.height
        anchors.fill: parent
        anchors.margins: Style.baseMargin
        spacing: 30
        NightClock {
            id: clock
            height: 80
            width: 190
            showDate: root.showDate
            showSeconds: root.showSeconds
            textColor: Style.onlineClockTextColor
        }

        ListView {
            id: currentWeatherView
            width: 100
            height: 100
            model: weatherModelItem.currentModel
            delegate: weatherCurrentDelegate
            interactive: false
        }

        Repeater {
            id: forecastWeatherView
            model: weatherModelItem.forecastModel
            delegate: weatherForecastDelegate
        }

        move: Transition {
            NumberAnimation {
                properties: "x,y"
                duration: 500
                easing.type: Easing.OutBounce
            }
        }
    }

    Text {
        id: statusText
        anchors.horizontalCenter: root.horizontalCenter
        anchors.bottom: exitButton.top
        anchors.margins: Style.baseMargin
        color: Qt.lighter(Style.penColor)
        font.pixelSize: Style.textPixelSize*0.8
        text: qsTr("Status: starting...")

        function showStatus (newStatusText) {
            text = qsTr("Status: " + newStatusText);
        }
    }

    // it is off and invisible by default
    BusyIndicator {
        id: busyIndicator
        anchors.horizontalCenter: root.horizontalCenter
        anchors.bottom: statusText.top
        anchors.margins: Style.baseMargin
    }

    Button {
        id: configureButton
        text: qsTr("Config")
        anchors.left: root.left
        anchors.bottom: root.bottom
        anchors.margins: Style.baseMargin
        onClicked: {
            configure.visible = true;
        }

    }

    Button {
        id: exitButton
        text: qsTr("Exit")
        width: configureButton.width
        anchors.right: root.right
        anchors.bottom: root.bottom
        anchors.margins: Style.baseMargin
        onClicked: Qt.quit()
    }

    Button {
        id: toggleStatesButton
        anchors.right: exitButton.left
        anchors.left: configureButton.right
        anchors.bottom: root.bottom
        anchors.margins: Style.baseMargin
        // simple binding like this "text: root.state" works here to, but it is more di
        // we use explicit strngs instead
        text: root.state == "Offline" ? qsTr("Get weather") : qsTr("Go offline")
        onClicked: {
            if (root.state == "Offline")
                configure.forceOffline = false;
            else
                configure.forceOffline = true;
        }
        // for experimental purposes...
        // onPressedAtXY: {
        //     console.log ("pressed at: " + coordinates)
        // }
    }

    Configure {
        id: configure
        anchors.fill: root
        z: root.z + 1
        visible: false
        showSeconds: true
        showDate: true
        forecastUpdateInterval: 5
        locationText: qsTr("Munich")
        forceOffline: false
    }

    states: [
        State {
            name: "Offline"
            PropertyChanges {target: clockScreen; visible: true}
            PropertyChanges {target: weatherScreen; visible: false}
        },
        State {
            name: "Live Weather"
            PropertyChanges {target: clockScreen; visible: false}
            PropertyChanges {target: weatherScreen; visible: true}
        },
        State {
            name: "Loading"
            PropertyChanges {target: clockScreen; visible: true}
            PropertyChanges {target: weatherScreen; visible: false}
            PropertyChanges {target: busyIndicator; on: true}
        }
    ]

    transitions: [
        Transition {
            from: "Offline"
            to: "Live Weather"
            PropertyAnimation {
                target: weatherScreen
                property: "opacity"
                from: 0
                to: 1
                easing.type: Easing.Linear
                duration: 5000
            }
        },
        Transition {
            from: "Live Weather"
            to: "Offline"
            PropertyAnimation {
                target: clockScreen
                property: "opacity"
                from: 0
                to: 1
                easing.type: Easing.Linear
                duration: 5000
            }
        }
    ]
}
```

What's Next?

Our application is now complete and you have learned major aspects of Qt Quick!

Certainly, our final application can be enhanced and extended with many features. We selected a minimal subset to cover the scope of this guide without going into too many details. The next and the last chapter discuss a few selected enhancements.

CHAPTER 11

Doing More, Learning More

11.1 Porting to Qt5

Qt5 contains a new version Qt Quick: 2.0. Additionally, due to modularization, there are a few changes in the location pre-installed components. We need to make two changes to get our application running on Qt5:

1. Replace `import QtQuick 1.x` with `import QtQuick 2.0`
2. When using `XmlListModel` we need to add `import QtQuick.XmlListModel 2.0`

11.2 Porting to a mobile device

It is very easy to get our application running on Symbian Anna or Belle devices as well as on N9.

You can use the template application in Qt Creator while creating a new project. Go the the menu, *File* → *New File or Project* and select *Qt Quick Application (Build-In Elements)* project type in the *Applications* project category.

Note: These steps apply to the project wizard in Qt Creator 2.6. The project wizard in older versions of Qt Creator has a slightly different layout

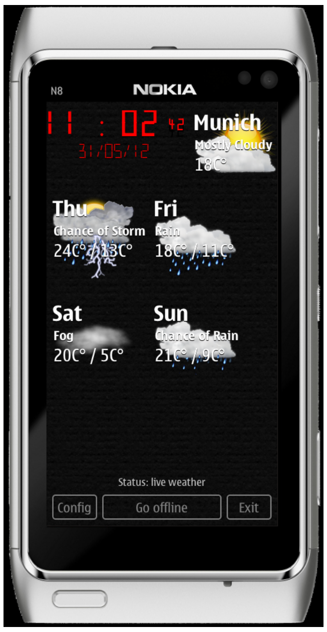
The wizard creates a simple application showing “Hello World”, similar to one we discussed at the beginning of this guide. This simple application also contains some C++ code and all other files required to compile and package the application.

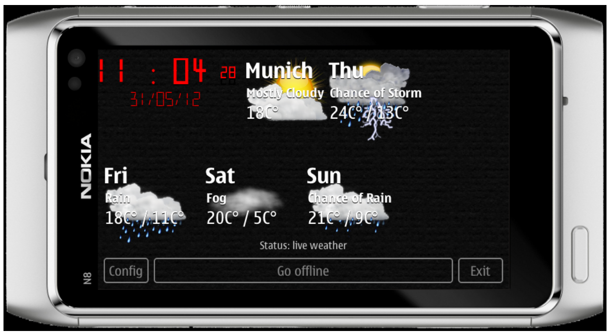
You can just replace the “Hello World” QML code with the the final application code. The major steps to do this are:

- Copy the QML files from the `WeatherClock` folder as well as from the `js`, `components`, `content` and `utils` folders (available in qt_quick_app_dev_intro_src.zip; see the *Downloads* (page 2) section) to the qml/<name_of_the_project> sub-folder in the project folder.

- Delete the main.qml file created by the wizard in that folder and rename the WeatherClock.qml into main.qml
- **Adapt paths to the new location of the QML component and resources:**
  - remove "../" in imports in main.qml
  - remove "../" in front of the value of backgroundImage in ./js/style.js
  - add "../ + " in the front of the source property value of the background item in Configure.qml in the components sub-folder
- The current layout and sizes are tailored for devices with 360x640 screen resultion. For example, Nokia N8. If your device has another screen resolution, you need to change all size-related properties accordingly.

That's it! You can now compile and run the application! The is how it looks in the Simulator in portrait and landscape modes:





11.3 Enhancements and New Features

*Better handling of configuration parameters**

We currently keep configuration parameters in the `Configure` component, which provides a UI as well. All configuration changes are lost when the user quits the application.

A much better implementation would be to split the `Configure` component in a UI element and a configuration item. The latter can be loaded in any other item that needs access to the configuration parameters. The user can change configuration parameters via the new UI element. Loading of default values and saving them before the application quits can be done by a dedicated setting item that uses the Offline Storage APIs[1]) provided by Qt Quick. The "Qt Quick Application Developer Guide for Desktop"[2] explains this in detail in the *4.2. Store and Load Data from a Database Using Javascript* section. When the application starts for the first time, a set of default values is stored in the database. During the next startup, the values from the database are read and assigned to the appropriate properties of the *configuration item*. All this can be done in the `onCompleted` handler in the main item. We can store current configuration parameters before we call `Qt.quit()` on click of `exitButton`.

*Internationalization**

A new version of the application could be available in multiple languages. We already use the `qsTr()` macro. Google weather data can be queried in multiple languages as well. This can save quite some effort. Unfortunately, there is a small issue in our application concerning this. Our weather icons are named after weather condition names in English. If the weather data is in another language, icons will not be found with the current implementation as the file names do not match the condition names. A possible solition would be to use file names in URLs for default icons referred in the weather data as file names for the local icons.

*Using Mobility APIs to get the current location automatically**

Instead of a predefined location, we could use Mobility API[3] and get the location automatically if the application is running on a mobile device.

*Using other weather feeds**

It might be a good idea to support at least one additional weather feed. Most of them require registration and in some cases a fee payment as well if the application is used for commercial purposes. You can consider adding other feeds in your version of the application. You can find more information about other weather feeds here:

- 5 Weather APIs – From WeatherBug to Weather Channel[4]
- Add Weather To Your Website With Autobrand® : Weather Underground[5]
- A Weather API Designed for Developers[6]

[1] http://qt-project.org/doc/qt-4.8/qdeclarativeglobalobject.html
[2] http://qt-project.org/wiki/Developer-Guides/
[3] http://doc.qt.digia.com/qtmobility/index.html
[4] http://blog.programmableweb.com/2009/04/15/5-weather-apis-from-weatherbug-to-weather-channel/
[5] http://www.wunderground.com/autobrand/info.asp
[6] http://www.wunderground.com/weather/api

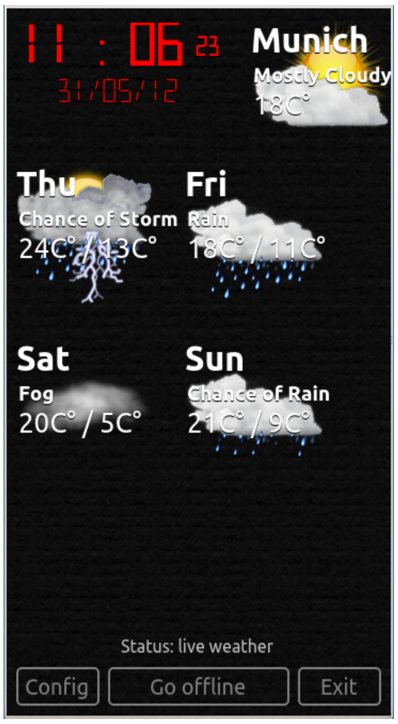
What's Next?

This is the end of the guide! The next chapter concludes it!

CHAPTER 12

Lesson Learned and Further Reading

This guide has given you an introduction to application programming in Qt Quick. We've touched all major aspects of Qt Quick by extending the code of a very simple "Hello World" application to become a real application, which can be used on a daily basis. This is it:



The main purpose of this guide was to help you get started and show where you can go if you need more details. The guide does not cover all details as that would overlap with Qt documentation[1] and other resources on the internet.

We did not touch one very import point in detail, but at least mentioned it a few times. This is about using C++ to extend Qt Quick and provide interfaces to existing software systems. The following articles in Qt Documentation are a good starting point to learn more about this:

- QML for Qt Programmers[2]

[1] http://qt-project.org/doc/qt-4.8/index.html
[2] http://qt-project.org/doc/qt-4.8/qtprogrammers.html

- Using QML Bindings in C++ Applications[3]
- Extending QML Functionality using C++[4]

In addition to the Qt Quick examples provided with Qt, many other interesting examples are included in the Qt training materials:

- "Qt Quick - Introduction to Qt Quick"[5]
- "Qt Quick - Rapid User Interface Prototyping"[6]

Good luck and have fun using Qt Quick!

[3]http://qt-project.org/doc/qt-4.8/qtbinding.html

[4]http://qt-project.org/doc/qt-4.8/qml-extending.html

[5]http://qt-project.org/videos/watch/qt_quick_introduction_to_qt_quick_part_1_4

[6]http://qt-project.org/videos/watch/qt_quick_rapid_user_interface_prototyping

CHAPTER 13

Annexure: JavaScript Language Overview

This article provides an overview of the JavaScript language. The idea is to provide a thorough overview of all of the language's features supported by Qt Quick. You may want to read through this article from start to finish to learn about all the basic features of this language - especially when you are getting started with a related technology such as QML.

This article is a slightly modified copy of the "JavaScript Language Overview"[1] article on the Qt Project Wiki. Its content has been tested on Qt 4.8 with Qt Quick 1.1. Additionally, this article provides a Qt Quick application which runs all code examples listed below. This application is available in the `js_basics` folder in `qt_quick_app_dev_intro_src.zip`, see the *Downloads* (page 2) section.

13.1 Introduction

JavaScript is a minimalistic dynamically typed scripting language. It is truly object-oriented, although it lacks support for classes. Frequently associated with client-side web development, JavaScript is a language of its own. Originally developed at Netscape and nowadays standardized as "ECMAScript-262 (3rd and 5th edition)"[2] , the language has found wide-spread use and circulates under various names. "JScript" is Microsoft's derivative of the language. "JavaScript" was the original name chosen by Netscape when the language was introduced with Netscape 2. Adobe's ActionScript was also based on ECMAScript-262 before version 3 was released.

Qt has been supporting a JavaScript engine compatible with ECMAScript-262 since Qt 4.3. This engine is called "QtScript" and was originally an independent implementation. Since Qt 4.5, QtScript has been based on JavaScriptCore from WebKit. Qt Quick makes intense use of QtScript.

[1]http://qt-project.org/wiki/JavaScript

[2]http://www.ecma-international.org/publications/standards/Ecma-262.htm

13.2 The Type System

JavaScript supports the following fundamental types:

> boolean number string object function

New variables are introduced into the current scope using the `var` statement:

```
var flag = false  // a boolean
var x = 1., y = 2 // numbers can be integers and reals
var s1 = 'abc'; // a string
```

To query the type of a variable, use the `typeof` keyword. `typeof` returns the name of the type as a string.

```
var x = 0; typeof x // 'number'
typeof { x: 1 } // 'object'
typeof typeof { x : 1 } // 'string'
typeof JSON.parse('{"a": 7}') // 'object'
```

Everything in JavaScript acts like an object.

```
1.3333333.toFixed(2) // '1.33'
7..toString() // '7'
```

Note that in JavaScript the expression `7.toString()` can't be interpreted correctly. `7.` is parsed into a number and thereafter results in a syntax error.

The primitive types `boolean`, `number`, and `string` are implicitly converted into objects when needed. For this purpose, the global object provides special constructor functions, which can also be invoked manually:

```
typeof 1. // 'number'
typeof new Number(1.) // 'object'
typeof new String('Hi!') // 'object'
```

Functions are special kinds of objects. They only differ from objects because they can be called and used as constructors. Properties can be added to functions dynamically:

```
function f(x) { return f.A      x * x }
f.A = 2.7

function Buffer() {}
Buffer.MAX_LINE_LENGTH = 4096
```

Usually those properties serve as global constants and therefore are written in uppercase.

Objects themselves can be expressed using an array or object literal. Arrays have no separate type, but are specialized objects which use array indexes as properties:

```
var o = { name: 'Werner', age: 84 } // allocate simple object
print(o.name, o[age])
// both notations are valid, but [] notation allows generated strings
var a = ['a', 'b', 7, 11.]
// an array, equivalent to {'0': 'a', '1': 'b', '2': 7, '3': 11.}
typeof o, a // 'object', 'object'
```

13.3 Expressions

The expression syntax follows mostly "C" syntax (as in C++ or Java). As a major difference, there is no sharp distinction between statements and expressions. Basically everything evaluates to something. Function declarations and compounds can be included on-the-fly:

```
function f() {} // evaluates 'undefined'
function f() {} + 1 // evaluates to 1, because 'undefined' is casted to 0
(function() {}) // evaluates to a function object
(function() { return 0; })() // evaluates to 0
```

Expressions are separated by semicolons or line breaks.

13.4 Branching

Conditional branches follow "C" syntax.

```
if (<expression>)
    <statement1>
else // optional
    <statement2> // optional
```

The switch statement follows the same fall through semantics as in "C":

```
switch(<expression>) {
    case <expression>:
        <statement-list-1>
        break;
    case <expression>:
        <statement-list-2>
        break;
    ...
    default:
        <statement-list-n>
}
```

13.5 Repetitions and Iterators

Repeated actions can be expressed using `do`, `while` and `for` loops:

```
...
do <statement> while (<expression>)
...
while (<expression>) <statement>
...
for (<init-expression>;<test-expression>;<step-expression>) <statement>
...
```

For iterating objects JavaScript provides a special `for-in` statement:

```
for (<expression>; in <object>;) <statement>
```

The given expression needs to be suitable for the left-hand side of an assignment. In the simplest case, it is just a variable declaration. Consider the following example:

```
var a = [1,2,3,4]
for (var i in a)
    print(i, a[i]    a[i])
// '0', 1
// '1', 4
// '2', 9
// '3', 16
```

Here the variable `i` is assigned to all keys of the array `a` consecutively. In the next example, the left-hand expression is dynamically generated:

```
var o = {a0: 11, a1: 7, a2: 5}
var k = []
for(k[k.length] in o);
```

The keys of `o` are copied to `k`. The loop statement itself is left empty. For each member in `o`, the name is assigned to another member of `k`.

13.6 Labeled Loops, Break and Continue

In JavaScript, loop statements can be given labels. The `break` and `continue` statements break or continue the current loop. It is possible to break an outer loop from the inner loop by using the label name as shown in the following example:

```
label_x:
for (var x = 0; x < 11; ++x) {
    for (var y = 0; y < 11; ++y) {
        if ((x + y) % 7 == 0) break label_x;
    }
}
```

13.7 Objects and Functions

Objects are created using an object literal or the `new` operator.

In the following example, a point coordinate is expressed as an object literal:

```
var p = { x: 0.1, y: 0.2 }
```

Objects are entirely dynamic sets of properties. New properties are introduced on first assignment. They can be deleted again by using the `delete` operator. To query if an object contains a certain property, use the `in` operator.

```
'z' in p // false
p.z = 0.3 // introduce new property 'z'
```

```
'z' in p // true
delete p.z // remove 'z' from p
p.z // undefined
```

Property values can be of any type - including the `function` type. Methods in JavaScript are just function properties. When a function is invoked in method notation, it gets a reference to the object as an implicit argument called, `this`.

```
p.move = function(x, y) {
    this.x = x
    this.y = y
}
p.move(1, 1) // invoke a method
```

JavaScript allows any function to be called as a method of any object by using the `call` method, however, there are only a few cases in which you would want to use the `call` method.

```
p2 = { x: 0, y: 0 }
p.move.call(p2, 1, 1)
```

13.8 Prototype-based Inheritance

The second way of creating objects is by using the `new` keyword together with a constructor function*:

```
var p = new Object
p.x = 0.1
p.y = 0.2
```

The `new` operator allocates a new object and calls the given constructor to initialize the object. In this case, the constructor is called `Object`, but it could be any other function as well. The constructor function gets passed the newly created object as the implicit `this` argument. In JavaScript there are no classes, but hierarchies of constructor functions which operate like object factories. Common constructor functions are written with a starting capital letter to distinguish them from average functions. The following example shows how to create point coordinates using a constructor function:

```
function Point(x, y) {
    this.x = x
    this.y = y
}
var p = new Point(1, 2)
```

Each function in JavaScript can be used as a constructor in combination with the `new` operator. To support inheritance, each function has a default property named `prototype`. Objects created from a constructor inherit all properties from the constructor's prototype. Consider the following example:

```
function Point(x, y) {
    this.x = x
    this.y = y
}
```

```
Point.prototype = new Object // can be omitted here
Point.prototype.translate = function(dx, dy) {
    this.x += dx
    this.y += dy
}
```

First we declared a new function called `Point`, which is meant to initialize a point. Thereafter we create our own prototype object, which in this case is redundant. The prototype of a function already defaults to an empty object. Properties which should be shared among all points are assigned to the prototype. In this case, we define the `translate` function which moves a point by a certain distance.

We can now instantiate points using the Point constructor:

```
var p0 = new Point(1, 1)
var p1 = new Point(0, 1)
p1.translate(1, -1)
p0 === p1 // false
p0.translate === p1.translate // true
```

The `p0` and `p1` objects carry their own `x` and `y` properties, but they share the `translate` method. Whenever an object's property value is requested by name, the underlying JavaScript engine first looks into the object itself and, if it doesn't contain that name, it falls back to the object's prototype. Each object carries a copy of its constructor's prototype for this purpose.

If an object actually contains a certain property, or if it is inherited, it can be inquired using the `Object.hasOwnProperty()` method.

```
p0.hasOwnProperty("x") // true
p0.hasOwnProperty("translate") // false
```

So far, we have only defined a single constructor with no real object hierarchy. We will now introduce two additional constructors to show how to chain prototypes and thereby build up more complex relationships between objects:

```
function Point(x, y) {
    this.x = x
    this.y = y
}
Point.prototype = {
    move: function(x, y) {
        this.x = x
        this.y = y
    },
    translate: function(dx, dy) {
        this.x += dx
        this.y += dy
    },
    area: function() { return 0; }
}

function Ellipsis(x, y, a, b) {
    Point.call(this, x, y)
    this.a = a
    this.b = b
}
```

```
Ellipsis.prototype = new Point
Ellipsis.prototype.area = function() { return Math.PI      this.a * this.b; }

function Circle(x, y, r) {
    Ellipsis.call(this, x, y, r, r)
}
Circle.prototype = new Ellipsis
```

Here we have three constructors which create points, ellipsis and circles. For each constructor, we have set up a prototype. When a new object is created using the `new` operator, the object is given an internal copy of the constructor's prototype. The internal reference to the prototype is used when resolving property names which are not directly stored in an object. Thereby properties of the prototypes are reused among the objects created from a certain constructor. For instance, let us create a circle and call its `move` method:

```
var circle = new Circle(0, 0, 1)
circle.move(1, 1)
```

The JavaScript engine first looks into the `circle` object to see if it has a `move` property. As it can't find one, it asks for the prototype of `circle`. The circle object's internal prototype reference was set to `Circle.prototype` during construction. It was created using the `Ellipsis` constructor, but that doesn't contain a `move` property either. Therefore, the name resolution continues with the prototype's prototype, which is created with the `Point` constructor. This time the name resolution succeeds as the `Point` constructor contains the `move` property. The internal prototype references are commonly referred to as the *prototype chain* of an object.

To query information about the prototype chain, JavaScript provides the `instanceof` operator.

```
circle instanceof Circle // true
circle instanceof Ellipsis // true
circle instanceof Point // true
circle instanceof Object // true
circle instanceof Array // false, is not an Array
```

As properties are introduced when they are first assigned, properties delivered by the prototype chain are overloaded when newly assigned. The `Object.hasOwnProperty` method and the `in` operator allow the place where a property is stored to be investigated.

```
circle.hasOwnProperty("x") // true, assigned by the Point constructor
circle.hasOwnProperty("area") // false
"area" in circle // true
```

As can be seen, the `in` operator resolves names using the prototype chain, while the `Object.hasOwnProperty` only looks into the current object.

In most JavaScript engines, the internal prototype reference is called `__proto__` and is accessible from the outside. In our next example, we will use the `__proto__` reference to explore the prototype chain. You should avoid using this property in all other contexts as it is a non-standard. First let us define a function to inspect an object by iterating its members:

```
function inspect(o) { for (var n in o) if (o.hasOwnProperty(n)) print(n, "=", o[n]); }
```

The `inspect` function prints all members stored in an object so if we now apply this function to the `circle` object as well as to its prototypes, we obtain the following output:

```
js> inspect(circle)
x = 1
y = 1
a = 1
b = 1
js> inspect(circle.__proto__)
x = undefined
y = undefined
a = undefined
b = undefined
js> inspect(circle.__proto__.__proto__)
x = undefined
y = undefined
area = function () { return Math.PI       this.a * this.b; }
js> inspect(circle.__proto__.__proto__.__proto__)
move = function (x, y) {
        this.x = x
        this.y = y;
    }
translate = function (dx, dy) {
        this.x += dx
        this.y += dy;
    }
area = function () { return 0; }
js> inspect(circle.__proto__.__proto__.__proto__.__proto__)
js>
```

As you can see, the `move` method is actually stored in `circle.__proto__.__proto__.__proto__`. You can also see a lot of redundant undefined members, but this shouldn't cause you any concern as prototype objects are shared among instances.

13.9 Scopes, Closures and Encapsulation

In JavaScript, execution starts in the global scope. Predefined global functions such as `Math` or `String` are properties of the global object. The global object serves as the root of the scope chain and is the first object created. In addition to the standard properties of the global object (see Qt Quick ECMAScript Reference[3]), Qt Quick provides a Qt global object[4] with some additional properties.

Usually, the global object can be referenced from the global scope by explicitly using the `this` keyword. The value of `this` is currently undefined in Qt Quick in the majority of contexts. See "QML JavaScript Restrictions" in Integrating JavaScript[5] in Qt documentation.

[3]http://qt-project.org/doc/qt-4.8/ecmascript.html
[4]http://qt-project.org/doc/qt-4.8/qdeclarativeglobalobject.html
[5]http://qt-project.org/doc/qt-4.8/qdeclarativejavascript.html

Further scopes are created on-demand whenever a function is called. Scopes are destroyed as any other object when they are no longer needed. When a function is defined, the enclosing scope is kept with the function definition and used as the parent scope for the function invocation scope. The new scope that is created upon function invocation is commonly referred to as the activation object*. The scope in which functions are defined is commonly referred to as the *lexical scope*.

The following example shows how to use lexical scopes to hide private members:

```
function Point(x, y) {
    this.getX = function() { return x; }
    this.setX = function(x2) { x = x2; }
    this.getY = function() { return y; }
    this.setY = function(y2) { y = y2; }
}
```

When the `Point` constructor is invoked, it creates get and set methods. The newly generated scope for the invocation of the `Point` constructor carries the `x` and `y` members. The getters and setters reference this scope and therefore it is retained for the lifetime of the newly created object. Interestingly there is no other way to access `x` and `y` other than via the set and get methods. This way JavaScript supports *data encapsulation*.

The concept of a function referencing the enclosing scope and retaining it for the lifetime of the function is commonly called a *closure*. Low-level programming languages such as “C” do not support closures because local scopes are created using stack frames and therefore need to be destroyed when the function returns.

13.10 Namespaces

Functions play a pivotal role in JavaScript. They serve as simple functions, methods, and constructors, and are used to encapsulate private properties. Additionally functions serve as anonymous namespaces:

```
(function() {
    // my code
    var smth = new Smth       // safe
    other = [1,2,3]           // bad, goes into global scope
    Array = function() {}     // forbidden
}) ()
var smthElse = {}             // bad, goes into global scope
```

An anonymous function is defined and executed on-the-fly. Global initialization code in particular is commonly wrapped in such a way to prevent polluting the global scope. As the global object can be modified as any other object in JavaScript, wrapping code in such a way reduces the risk of accidentally overwriting a global variable. To ensure that it actually works, all variables need to be duly introduced using the `var` statement.

Named namespaces can also be created with functions. If for instance we wanted to write a utility library for painting applications, we could write:

```
function PaintUtil() {
    PaintUtil.Point = function(x, y) {
        this.move = function(x2, y2) { x = x2; y = y2 }
        this.getX = function() { return x; }
        this.getY = function() { return y; }
    }
    // Ellipsis, Circle, other painting utility methods
}
PaintUtil()
```

Once this little library module is executed, it provides the single `PaintUtil` object, which makes the utility functions accessible. A point can be instantiated using the constructor provided by `PaintUtil` as follows:

```
var p = new PaintUtil.Point(0.1, 0.2)
```

Reusable JavaScript modules should only introduce a single global object with a distinguishable name.

13.11 Common Methods

JavaScript allows the default behavior of an object to be changed using the `valueOf()` and the `toString()` methods. `valueOf()` is expected to return a value of fundamental type. It is used to compare objects (when sorting them) and to evaluate expressions comprising of objects and fundamental types. `toString()` is invoked when an object is cast to a string. In JavaScript, objects are compared for equality differently than for being greater or lower. Comparison for equality always compares the object references. Comparison for being lower or greater, on the other hand, converts objects by first converting the objects to values of fundamental types. First `valueOf()` is invoked, and if it doesn't return a fundamental type, it calls `toString()` instead.

For our `Point` class, we could define the methods as follows:

```
Point.prototype.valueOf = function () {
    return Math.sqrt(this.getX()      this.getX() + this.getY() * this.getY());
}
Point.prototype.toString = function () {
    return this.getX().toString() + "," + this.getY().toString();
}
```

13.12 Exceptions

JavaScript provides an exception handling mechanism like most other high-level languages. Exceptions are thrown using the `throw` statement. Any value can be used as an exception object:

```
throw <expression>;
```

When an exception is thrown, JavaScript unwinds the current scope until it arrives at a try-catch scope:

```
try {
    <statement-list>
}
catch (<name for exception object>) {
    // handle exception
}
finally {
    // always go through here
}
```

The name of the exception object is only locally defined inside the catch scope. Exceptions can be re-thrown.

13.13 Resources

Useful web links:

- “The JavaScript Reference”[6] on the Mozilla Developer Network
- “JavaScript. The core.” by Dmitry A. Soshnikov”[7]
- “Changes to JavaScript: EcmaScript 5 by Mark Miller”[8] - a video from Google Tech Talk, May 18, 2009
- “Standard ECMA-262”[9] - PDF download of the official standard

Recommended Books:

*“JavaScript: The Good Parts” by Douglas Crockford[10] * “Part I - Core JavaScript” in “JavaScript: The Definitive Guide” by David Flanagan[11]

genindex

[6] https://developer.mozilla.org/en/JavaScript/Reference
[7] http://dmitrysoshnikov.com/ecmascript/javascript-the-core/
[8] http://www.youtube.com/watch?v=Kq4FpMe6cRs
[9] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[10] http://oreilly.com/catalog/9780596517748.do
[11] http://shop.oreilly.com/product/9780596805531.do