Foreword

Way back in 1991, I sat on a park bench in Trondheim, Norway, together with Haavard Nord. We were doing our non-military service together at the regional hospital there, and needed to develop software for the storage and analysis of ultrasound images. The hospital used all sorts of computers and wanted the system to work on Unix, Mac, and Windows. This was a huge challenge and we scanned the market for available class libraries that could help us. We were appalled by the quality of what we found. On that park bench we decided to come up with our own solution to the challenge.

We were young, ambitious, and naïve. Sick and tired of wasting our time finding out how to use non-intuitive tools and libraries, we set our sights on improving the situation. We wanted to change the world of software development ever so slightly. Our goal was to make life easier for software developers. To make it possible to focus on what we all know is the fun side of developing software: being creative and turning out well-written code. So, we created the first crude versions of Qt, and incorporated Trolltech a few years later.

I think we achieved at least part of our goal. Qt has had tremendous success since it was first released in 1995.

In 2008 Trolltech was acquired by Nokia and in April 2009 it was time for me to move on. After 15 years and 27 days in the company I was no longer on the inside.

The product is in good hands, and the passion and hard work of the team are the same as ever. The Trolls at Nokia are making sure that Qt continues to be the rock solid framework you expect. Lars Knoll (of kHTML—WebKit—fame) today leads almost 150 dedicated Qt engineers. Nokia has also added the LGPL as a licensing option, making Qt accessible to even more developers.

This fall I was invited by Nokia as a guest of honor at the Qt Developer Days in Munich, Germany. This user conference—which also takes place in the U.S.—is a fantastic venue for Qt enthusiasts and has been increasing in size year by year. It was great feeling the buzz and talking to Qt users from all over Europe. I spoke to many developers who told me that Qt makes a real difference in their software work. That makes an old hacker feel good.

Qt as a good tool and class library is only half the story behind its success. You also need good documentation, tutorials, and books. After all, the goal was to make life easier for developers.
That is why I was never in doubt, back in 2003. I was President of Trolltech and Mark Summerfield, the head of documentation, came into my office. He wanted to write a book about Qt together with Jasmin Blanchette. A really good book, written by someone with intimate knowledge of the product and with a passion for explaining things clearly and intuitively. Who was better fit for the task than the head of Qt documentation, together with one of the best Qt developers?

The end result was a great book about Qt, which has since been updated and expanded.

Mark has now completed another important project.

A good book on advanced Qt programming has been missing in the arsenal of Qt programmers. I'm very happy that Mark has written one. He is a fantastic technical writer with all the necessary background to write authoritatively about Qt programming. His focus on detail and ability to express himself clearly and intuitively have always impressed me. In other words: You are in for a treat!

You are holding in your hands (or reading on-screen) an excellent opportunity to expand on your knowledge of all the cool stuff you can do with Qt.

Happy programming!

Eirik Chambe-Eng
The southern Alps, France
December 24, 2009
Introduction

For some time I have wanted to write a Qt book that covered topics that were too advanced for C++ GUI Programming with Qt 4,* even though that book itself has proved quite challenging for some readers. There is also some specialized material—not all of it difficult—that I wanted to cover that simply does not belong in a first book on Qt programming. Furthermore, in view of the sheer size of Qt, no one book can possibly do justice to all that it offers, so there was clearly room for the presentation of new material.

What I’ve done in this book is to take a selection of modules and classes from a variety of areas and shown how to make good use of them. The topics chosen reflect both my own interests and also those that seem to result in the most discussion on the qt-interest mailing list. Some of the topics are not covered in any other book, while other topics cover more familiar ground—for example, model/view programming. In all cases, I have tried to provide more comprehensive coverage than is available elsewhere.

So the purposes of this book are to help Qt programmers deepen and broaden their Qt knowledge and to increase the repertoire of what they can achieve using Qt. The “advanced” aspect often refers more to what you will be able to achieve than to the means of achieving it. This is because—as always—Qt insulates us as far as possible from irrelevant detail and underlying complexity to provide easy-to-use APIs that we can use simply and directly to great effect. For example, we will see how to create a music player without having to know anything about how things work under the hood; we will need to know only the high-level API that Qt provides. On the other hand, even using the high-level QtConcurrent module, the coverage of threading is necessarily challenging.

This book assumes that readers have a basic competence in C++ programming, and at least know how to create basic Qt applications—for example, having read a good Qt 4 book, and having had some practical experience. Readers are also assumed to be familiar with Qt’s reference documentation, at least as far as being able to navigate it to look up the APIs of classes of interest. In addition, some chapters assume some basic topic-specific knowledge—for example, Chapter 1 assumes some knowledge of JavaScript and web programming, and the threading chapters assume a basic understanding of threading and Qt’s threading classes. All these assumptions mean that this book can avoid ex-

plaining many details and classes that are already familiar to Qt programmers, such as using layouts, creating actions, connecting signals and slots, and so on, leaving the book free to focus on the less familiar material.

Of course, no single volume book can realistically do justice to Qt’s more than 700 public classes—almost 800 in Qt 4.6—and its much more than one million words of documentation, so no attempt is made to do so here. Instead this book provides explanations and examples of how to use some of Qt’s most powerful features, complementing the reference documentation rather than duplicating it.

The book’s chapters have been designed to be as self-contained as possible, so it is not necessary to read the book from beginning to end in chapter order. To support this, where particular techniques are used in more than one chapter, the explanation is given in just one place and cross-references are given elsewhere. Nonetheless, if you plan to read odd chapters out of order, I recommend that you at least do an initial skim read of the entire book, since chapters and sections devoted to one particular topic may of necessity have material relating to other topics. Also, I have tried to include lots of small details from Qt’s API throughout, to make the book’s content richer, and to show as many features as possible in context, so useful information appears throughout.

As with all my previous books, the quoted code snippets are of “live code”, that is, the code was automatically extracted from the examples’ source files and directly embedded in the PDF that went to the publisher—so there are no cut and paste errors, and the code works. The examples are available from www.qtrac.eu/aqpbook.html and are licensed under the GPL (GNU General Public License version 3). The book presents more than twenty-five examples spread over more than 150 .hpp and .cpp files, and amounting to well over 20 000 lines of code. Although all of the most important pieces of code are quoted and explained in the book, there are numerous small details that there isn’t space to cover in the book itself, so I recommend downloading the examples and at least reading the source code of those examples that are in your areas of particular interest. In addition to the examples, some modules containing commonly used functionality are also provided. These all use the AQP namespace to make them easy to reuse, and they are all introduced in the first couple of chapters, and then used throughout the book.

All the examples—except for those in the last chapter which use Qt 4.6-specific features—have been tested with Qt 4.5 and Qt 4.6 on Linux, Mac OS X, and Windows. Applications built using Qt 4.5 will run unchanged with Qt 4.6, and later Qt 4.x versions, because Qt maintains backward compatibility between minor releases. However, where there are differences between the two Qt versions, the book shows and explains the Qt 4.6-specific approach, while the source code uses #if QT_VERSION so that the code compiles with either version with the best practices used for each. A few examples may work with earlier Qt 4.x versions, particularly Qt 4.4, and some examples could be backported to
Introduction

an earlier Qt version—however, the focus of this book is purely on Qt 4.5 and Qt 4.6, so there is no explicit coverage of backporting.

The book shows best Qt 4.6 practices, and despite Qt 4.6’s numerous new features compared with Qt 4.5, this makes few differences to the code. One trivial difference is that Qt 4.6 has a shortcut for the “quit” action and Qt 4.5 hasn’t; the source code uses the shortcut for Qt 4.6 and has equivalent code for Qt 4.5 by using #if QT_VERSION. A much more important difference is that Qt 4.6 introduced the QGraphicsObject class and also changed the behavior of graphics items when it comes to communicating geometry changes. We explain the differences in a sidebar and show the Qt 4.6 approach in the book’s code snippets, but in the source code, #if QT_VERSION is used to show how to do the same things using Qt 4.6 and Qt 4.5 or earlier, and using the best approach for both. In the book’s last chapter, Qt 4.6-specific features are shown, with two out of the three examples covered being conversions of examples presented earlier, and that make use of the Qt 4.6 animation and state machine frameworks. Modifying earlier examples makes it easier to see how to go from the traditional Qt approach to using the new frameworks.

The next version of Qt—Qt 4.7—will focus on stability, speed, and apart from the new Qt Quick technology (which provides a means of creating GUIs declaratively using a JavaScript-like language), will introduce fewer new features than in previous releases. Nonetheless, despite the huge ongoing development effort that is being put into Qt, and its ever increasing scope, this book should serve as a useful resource for learning about and using important Qt technologies in the Qt 4.x series, especially for Qt 4.5, Qt 4.6, and later versions, for some years to come.

Acknowledgements

My first acknowledgement is of my friend Trenton Schulz, an ex-senior software engineer at Nokia’s Qt Development Frameworks (formerly Trolltech) who is now a research scientist at the Norwegian Computing Center. Trenton has proved to be a reliable, insightful, and challenging reviewer, whose careful reading, high standards, and numerous suggestions have considerably helped to improve this book.

My next acknowledgement is of another friend, Jasmin Blanchette, also an ex-senior software engineer at Qt Development Frameworks, coauthor with me of the C++ GUI Programming with Qt 4 book, and now researching for a PhD at the Technische Universität München. We both came up with the idea for this book some time ago, and it is only due to pressure of work that he has been an excellent—and demanding—reviewer, rather than coauthor.

I would also like to thank many people who work for (or worked for) Qt Development Frameworks who read portions of the book and provided useful feedback,
or who answered technical questions, or both. These include Andreas Aardal Hanssen (who gave particularly excellent feedback and suggestions regarding the graphics/view chapters, and who drafted the off-screen rendering sidebar for me), Andy Shaw, Bjørn Erik Nilsen, David Boddie, Henrik Hartz, Kavindra Devi Palaraja, Rainer Schmid (now at froglogic), Simon Hausmann, Thierry Bastian, and Volker Hilsheimer.

The Italian software company www.develer.com was kind enough to provide me with free repository hosting to aid my peace of mind over the long process of writing the book. And several of their developers gave me useful feedback, particularly on some of the examples in the early chapters. I’m especially thankful to Gianni Valdambrini, Giovanni Bajo, Lorenzo Mancini (who set up the repository for me), and Tommaso Massimi.

A special thank you to rough-cut reader Alexey Smirnov who spotted some errors and encouraged me to add support for network proxies to some of the networking examples.

I also want to thank froglogic’s founders, Reginald Stadlbauer and Harri Porten—the part-time consultancy work I do for them has helped fund the time to write this book, as well as introducing me to some programming technologies and ideas that were new to me. They’ve also turned me into a big fan of their GUI application testing tool, Squish.

My friend Ben Thompson also deserves thanks—for reminding me of certain mathematical concepts that I’d forgotten, and especially for his patience in explaining them to me until I understood them again.

This book (and some of my others) would not have been possible without Qt. So I’m very grateful to Eirik Chambe-Eng and Haavard Nord for creating Qt—and especially to Eirik for allowing me to write my first book as part of my daily work at Trolltech, and for taking the time and care to write the foreword to this book.

Special thanks to my editor, Debra Williams Cauley, both for quite independently suggesting that I write this book in the first place, and for her support and practical help as the work progressed. Also thanks to Jennifer Lindner who gave useful input on the book’s structure as well as other feedback that I incorporated. Thanks also to Anna Popick, who managed the production process so well, and to the proofreader, Barbara Wood, who did such fine work.

I also want to thank my wife, Andrea, who experiences all the ups and downs of writing along with me, for her enduring love and support.
Model/View Views

This chapter covers model/view views, and is the last chapter covering Qt's model/view architecture. Just like the previous two chapters, this chapter assumes a basic familiarity with the model/view architecture, as described at the beginning of Chapter 3 (88 ➤).

Qt's standard model views, QListView, QTableView, QColumnView, and QTreeView, are sufficient for most purposes, most of the time. And like other Qt classes, they can be subclassed—or we can use custom delegates—to affect how they display the model's items. However, there are two situations where we need to create a custom view. One is where we want to present the data in a radically different way from how Qt's standard views present data, and the other is where we want to visualize two or more data items combined in some way.

Broadly speaking there are two approaches we can take to the creation of custom views. One approach is used when we want to create a view component—that is, a view that is potentially reusable with a number of different models and that must fit in with Qt's model/view architecture. In such cases we would normally subclass QAbstractItemView, and provide the standard view API so that any model could make use of our view. The other approach is useful when we want to visualize the data in a particular model in such a unique way that the visualization has little or no potential for reuse. In these cases we can simply create a custom model viewer that has exactly—and only—the functionality required. This usually involves subclassing QWidget and providing our own API, but including a setModel() method.

In this chapter we will look at two examples of custom views. The first is a generic QAbstractItemView subclass that provides the same API as Qt's built-in views, and that can be used with any model, although it is designed in particular for the presentation and editing of list models. The second is a visualizer view that is specific to a particular model and that provides its own API.
QAbstractItemView Subclasses

In this section we will show how to create a QAbstractItemView subclass that can be used as a drop-in replacement for Qt’s standard views. In practice, of course, just as there are list, table, and tree models, there are corresponding views, and so here we will develop a custom list view, although the principles are the same for all QAbstractItemView subclasses.

Figure 6.1 shows the central area of the Tiled List View application (tiled-listview). The area has two views that are using the same model: on the left a standard QListView, and on the right a TiledListView. Notice that although the widgets are the same size and use the same font, the TiledListView shows much more data. Also, as the figure illustrates, the TiledListView does not use multiple columns; rather, it shows as many items as it can fit in each row—for example, if it were resized to be a bit wider, it would fit four or more items on some rows.

One usability difference that makes keyboard navigation faster and easier—and more logical—in the TiledListView is that using the arrow keys does not simply go forward or backward in the list of items. When the user navigates through the items using the up (or down) arrow keys, the selected item is changed to the item visually above (or below) the current item. Similarly, when the user navigates using the left (or right) arrow keys, the selected item is changed to the item to the left (or right) as expected, unless the current item is at the left (or right) edge. For the edge cases, the selected item is changed to the item that is logically before (or after) the current item.

The QAbstractItemView API is large, and at the time of this writing, the Qt documentation does not explicitly specify which parts of the API must be reimplemented by subclasses and which base class implementations are sufficient. However, some of the methods are pure virtual and so must be reimplemented. Also, Qt comes with the examples/itemviews/chart example which serves as a useful guide for custom view implementations.

The API we have implemented for the TiledListView, and the one that we consider to be the minimum necessary for a custom QAbstractItemView subclass,
is shown in Table 6.1. Qt’s chart example reimplements all the methods listed in the table, and also the mousePressEvent() and mouseMoveEvent() event handlers (to provide rubber band support—something not needed for the TiledListView). The chart example also implements the edit() method to initiate editing—again, something we don’t need to do for the TiledListView even though it is editable, because the inherited base class’s behavior is sufficient.

Before we look at the TiledListView class, here is how an instance is created and initialized.

```cpp
TiledListView *tiledListView = new TiledListView;
tiledListView->setModel(model);
```

As these two lines make clear, the TiledListView is used in exactly the same way as any other view class.

Since the API that must be implemented is shown in Table 6.1, we won’t show the class’s definition in the header file, apart from the private data, all of which is specific to the TiledListView.

```cpp
private:
    mutable int idealWidth;
    mutable int idealHeight;
    mutable QHash<int, QRectF> rectForRow;
    mutable bool hashIsDirty;
```

The idealWidth and idealHeight are the width and height needed to show all the items without the need for scrollbars. The rectForRow hash returns a QRectF of the correct position and size for the given row. (Note that since the TiledListView is designed for showing lists, a row corresponds to an item.) All these variables are concerned with behind-the-scenes bookkeeping, and since they are used in const methods we have been forced to make them mutable.

Rather than updating the rectForRow hash whenever a change takes place, we do lazy updates—that is, we simply set hashIsDirty to true when changes occur. Then, whenever we actually need to access the rectForRow hash, we recalculate it only if it is dirty.

We are now almost ready to review the TiledListView implementation, and will do so, starting with the constructor, and including the private supporting methods as necessary. But first we must mention an important conceptual point about QAbstractItemView subclasses.

The QAbstractItemView base class provides a scroll area for the data it displays. The only part of the widget that is a QAbstractItemView subclass that is visible is its viewport, that is, the part that is shown by the scroll area. This visible area is accessible using the viewport() method. It doesn’t really matter what size the widget actually is; all that matters is what size the widget would need to be to show all of the model’s data (even if that is far larger than the screen). We will
### Table 6.1  The `QAbstractItemView` API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dataChanged(topLeft, bottomRight)</code></td>
<td>This slot is called when the items with model indexes in the rectangle from <code>topLeft</code> to <code>bottomRight</code> change</td>
</tr>
<tr>
<td><code>horizontalOffset()</code></td>
<td>Returns the view’s horizontal offset</td>
</tr>
<tr>
<td><code>indexAt(point)</code></td>
<td>Returns the model index of the item at position <code>point</code> in the view’s viewport</td>
</tr>
<tr>
<td><code>isIndexHidden(index)</code></td>
<td>Returns <code>true</code> if the item at <code>index</code> is a hidden item (and therefore should not be shown)</td>
</tr>
<tr>
<td><code>mousePressEvent(event)</code></td>
<td>Typically used to set the current model index to the index of the clicked item</td>
</tr>
<tr>
<td><code>moveCursor(how, modifiers)</code></td>
<td>Returns the model index of the item after navigating <code>how</code> (e.g., up, down, left, or right), and accounting for the keyboard <code>modifiers</code></td>
</tr>
<tr>
<td><code>paintEvent(event)</code></td>
<td>Paints the view’s contents on the viewport</td>
</tr>
<tr>
<td><code>resizeEvent(event)</code></td>
<td>Typically used to update the scrollbars</td>
</tr>
<tr>
<td><code>rowsAboutToBeRemoved(parent, start, end)</code></td>
<td>This slot is called when rows from <code>start</code> to <code>end</code> under <code>parent</code> are about to be removed</td>
</tr>
<tr>
<td><code>rowsInserted(parent, start, end)</code></td>
<td>This slot is called when rows from <code>start</code> to <code>end</code> are inserted under the <code>parent</code> model index</td>
</tr>
<tr>
<td><code>scrollContentsBy(dx, dy)</code></td>
<td>Scrolls the view’s viewport by <code>dx</code> and <code>dy</code> pixels</td>
</tr>
<tr>
<td><code>scrollTo(index, hint)</code></td>
<td>Scrolls the view to ensure that the item at the given model <code>index</code> is visible, and respecting the scroll <code>hint</code> as it scrolls</td>
</tr>
<tr>
<td><code>setModel(model)</code></td>
<td>Makes the view use the given <code>model</code></td>
</tr>
<tr>
<td><code>setSelection(rect, flags)</code></td>
<td>Applies the selection <code>flags</code> to all of the items in or touching the rectangle <code>rect</code></td>
</tr>
<tr>
<td><code>updateGeometries()</code></td>
<td>Typically used to update the geometries of the view’s child widgets, e.g., the scrollbars</td>
</tr>
<tr>
<td><code>verticalOffset()</code></td>
<td>Returns the view’s vertical offset</td>
</tr>
<tr>
<td><code>visualRect(index)</code></td>
<td>Returns the rectangle occupied by the item at the given model <code>index</code></td>
</tr>
<tr>
<td><code>visualRegionForSelection(selection)</code></td>
<td>Returns the viewport region for the items in the <code>selection</code></td>
</tr>
</tbody>
</table>

*This method is pure virtual, so it must be reimplemented in subclasses.*
see how this affects our code when we look at the `calculateRectsIfNecessary()` and `updateGeometries()` methods.

```cpp
tiledListView::TiledListView(QWidget *parent)
: QAbstractItemView(parent), idealWidth(0), idealHeight(0),
  hashIsDirty(false)
{
  setFocusPolicy(Qt::WheelFocus);
  setFont(QApplication::font("QListView"));
  horizontalScrollBar()->setRange(0, 0);
  verticalScrollBar()->setRange(0, 0);
}
```  

The constructor calls the base class and initializes the private data. Initially the view’s “ideal” size is 0 × 0 since it has no data to display.

Unusually, we call `setFont()` to set the widget’s font rather than do what we normally do in custom widgets and just use the inherited font. The font returned by the `QApplication::font()` method, when given a class name, is the platform-specific font that is used for that class. This makes the `TiledListView` use the correct font even on those platforms (such as Mac OS X) that use a slightly different-sized font from the default `QWidget` font for `QListView`.

Since there is no data we set the scrollbars’ ranges to (0, 0); this ensures that the scrollbars are hidden until they are needed, while leaving the responsibility for hiding and showing them to the base class.

```cpp
void TiledListView::setModel(QAbstractItemModel *model)
{
  QAbstractItemView::setModel(model);
  hashIsDirty = true;
}
```  

When a model is set we first call the base class implementation, and then set the private `hashIsDirty` flag to true to ensure that when the `calculateRectsIfNecessary()` method is called, it will update the `rectForRow` hash.

The `indexAt()`, `setSelection()`, and `viewportRectForRow()` methods all need to know the size and position of the items in the model. This is also true indirectly of the `mousePressEvent()`, `moveCursor()`, `paintEvent()`, and `visualRect()` methods, since all of them call the methods that need the sizes and positions. Rather than compute the rectangles dynamically every time they are needed, we have chosen to trade some memory for the sake of speed by caching them in the `rectForRow` hash. And rather than keeping the hash up to date by calling `calculateRectsIfNecessary()` whenever a change occurs, we simply keep track of whether

---

*For more about how Qt’s font and palette propagation works, see labs.qt.nokia.com/blogs/2008/11/16.*
the hash is dirty, and only recalculate the rectangles when we actually need to access the hash.

```cpp
const int ExtraHeight = 3;

void TiledListView::calculateRectsIfNecessary() const {
    if (!hashIsDirty)
        return;
    const int ExtraWidth = 10;
    QFontMetrics fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    const int MaxWidth = viewport()->width();
    int minimumWidth = 0;
    int x = 0;
    int y = 0;
    for (int row = 0; row < model()->rowCount(rootIndex()); ++row) {
        QModelIndex index = model()->index(row, 0, rootIndex());
        QString text = model()->data(index).toString();
        int textWidth = fm.width(text);
        if (!(x == 0 || x + textWidth + ExtraWidth < MaxWidth)) {
            y += RowHeight;
            x = 0;
        } else if (x != 0)
            x += ExtraWidth;
        rectForRow[row] = QRectF(x, y, textWidth + ExtraWidth,
                                  RowHeight);
        if (textWidth > minimumWidth)
            minimumWidth = textWidth;
        x += textWidth;
    }
    idealWidth = minimumWidth + ExtraWidth;
    idealHeight = y + RowHeight;
    hashIsDirty = false;
    viewport()->update();
}
```

This method is the heart of the TiledListView, at least as far as its appearance is concerned, since—as we will see shortly—all the painting is done using the rectangles created in this method.

We begin by seeing if the rectangles need to be recalculated at all. If they do we begin by calculating the height needed to display a row, and the maximum width that is available to the viewport, that is, the available visible width.
In the method's main loop we iterate over every row (i.e., every item) in the model, and retrieve the item's text. We then compute the width needed by the item, and compute the x- and y-coordinates where the item should be displayed —these depend on whether the item can fit on the same line (i.e., the same visual row) as the previous item, or if it must start a new line. Once we know the item's size and position, we create a rectangle based on that information and add it to the rectForRow hash, with the item's row as the key.

Notice that during the calculations in the loop, we use the actual visible width, but assume that the available height is whatever is needed to show all the items given this width. Also, to retrieve the model index we want, we pass a parent index of QAbstractItemView::rootIndex() rather than an invalid model index (QModelIndex()). Both work equally well for list models, but it is better style to use the more generic rootIndex() in QAbstractItemView subclasses.

At the end we recompute the ideal width (the width of the widest item plus some margin), and the ideal height (the height necessary to show all the items at the viewport's current width, no matter what the viewport's actual height is) —at this point the y variable holds the total height of all the rows. The ideal width may be greater than the available width, for example, if the viewport is narrower than the width needed to display the longest item—in which case the horizontal scrollbar will automatically be shown. Once the computations are complete, we call update() on the viewport (since all painting is done on the viewport, not on the QAbstractItemView custom widget itself), so that the data will be repainted.

At no point do we refer to or care about the actual size of the QAbstractItemView custom widget itself—all the calculations are done in terms of the viewport and of the ideal width and height.

```cpp
QRect TiledListView::visualRect(const QModelIndex &index) const
{
    QRect rect;
    if (index.isValid())
        rect = viewportRectForRow(index.row()).toRect();
    return rect;
}
```

This pure virtual method must return the rectangle occupied by the item with the given model index. Fortunately, its implementation is very easy because we pass the work on to our private viewportRectForRow() method that makes use of the rectForRow hash.

```cpp
QRectF TiledListView::viewportRectForRow(int row) const
{
    calculateRectsIfNecessary();
    QRectF rect = rectForRow.value(row).toRect();
```
if (!rect.isValid())
    return rect;
return QRectF(rect.x() - horizontalScrollBar()->value(),
             rect.y() - verticalScrollBar()->value(),
             rect.width(), rect.height());
}

This method is used by the `visualRect()` method and by the `moveCursor()` and `paintEvent()` methods. It returns a `QRectF` for maximum accuracy (e.g., for the `paintEvent()` method); other callers convert the returned value to a plain integer-based `QRect` using the `QRectF::toRect()` method.

The `calculateRectsIfNecessary()` method must be called by methods that access the `rectForRow` hash, before the access takes place. If the `rectForRow` hash is up to date, the `calculateRectsIfNecessary()` method will do nothing; otherwise it will recompute the rectangles in the hash ready for use.

![Figure 6.2 Widget vs. viewport coordinates](image)

The rectangles in the `rectForRow` hash have the x- and y-coordinates of their rows (items) based on the ideal width (usually the visible width) and the ideal height (the height needed to display all the items at the current width). This means that the rectangles are effectively using widget coordinates based on the ideal size of the widget (the actual size of the widget is irrelevant). The `viewportRectForRow()` method must return a rectangle that is in viewport coordinates, so we adjust the coordinates to account for any scrolling. Figure 6.2 illustrates the difference between widget and viewport coordinates.

    bool isIndexHidden(const QModelIndex& ) const { return false; }

We must reimplement this pure virtual method, and have done so in the header since it is so trivial. This method is designed for data that can have hidden items—for example, a table with hidden rows or columns. For this view, no
items are hidden because we don’t offer support for hiding them, so we always return false.

```cpp
void TiledListView::scrollTo(const QModelIndex &index, 
    QAbstractItemView::ScrollHint)
{
    QRect viewRect = viewport()->rect();
    QRect itemRect = visualRect(index);
    if (itemRect.left() < viewRect.left())
        horizontalScrollBar()->setValue(horizontalScrollBar()->value() 
            + itemRect.left() - viewRect.left());
    else if (itemRect.right() > viewRect.right())
        horizontalScrollBar()->setValue(horizontalScrollBar()->value() 
            + qMin(itemRect.right() - viewRect.right(), 
            itemRect.left() - viewRect.left()));
    if (itemRect.top() < viewRect.top())
        verticalScrollBar()->setValue(verticalScrollBar()->value() + 
            itemRect.top() - viewRect.top());
    else if (itemRect.bottom() > viewRect.bottom())
        verticalScrollBar()->setValue(verticalScrollBar()->value() + 
            qMin(itemRect.bottom() - viewRect.bottom(), 
            itemRect.top() - viewRect.top()));
    viewport()->update();
}
```

This is another pure virtual method that we are obliged to implement. Fortunately, the implementation is straightforward (and is almost the same as that used in Qt’s chart example).

If the item to be scrolled to has a rectangle that is left of the viewport’s left edge, then the viewport must be scrolled. The scrolling is done by changing the horizontal scrollbar’s value, adding to it the difference between the item rectangle’s left edge and the viewport’s left edge. All the other cases work in an analogous way.

Note that this method calls `visualRect()` which in turn calls `viewportRectForRow()` which in turn calls `calculateRectsIfNecessary()`—as already noted, this last method recalculates the rectangles in the `rectForRow` hash if the hash is dirty.

```cpp
QModelIndex TiledListView::indexAt(const QPoint &point_) const
{
    QPoint point(point_);
    point.rx() += horizontalScrollBar()->value();
    point.ry() += verticalScrollBar()->value();
    calculateRectsIfNecessary();
}
QHashIterator<int, QRectF> i(rectForRow);
while (i.hasNext()) {
    i.next();
    if (i.value().contains(point))
        return model()->index(i.key(), 0, rootIndex());
}
return QModelIndex();

This pure virtual method must return the model index of the item at the given point. The point is in viewport coordinates, but the rectangles in rectForRow are in widget coordinates. Rather than convert each rectangle that we check to see if it contains the point, we do a one-off conversion of the point into widget coordinates.

The QPoint::rx() and QPoint::ry() methods return non-const references to the point’s x- and y-coordinates, making it easy to change them. Without these methods we would have to do, for example, point.setX(horizontalScrollBar()->value() + point.x()).

We make sure that the rectForRow hash is up to date, and then we iterate over every row (item) in the hash—in an arbitrary order since hashes are unordered. If we find a value, that is, a rectangle, that contains the point, we immediately return the corresponding model index.

For models with large numbers of items (beyond the low thousands), this method might run slowly since in the worst case every item’s rectangle must be checked, and even in the average case, half of the items must be checked. For the TiledListView this is unlikely to be a problem, since putting thousands of items in a list model of any kind is probably unhelpful to users—a tree model that grouped the items and made the top-level list of items a more manageable size would almost certainly be better.

void TiledListView::dataChanged(const QModelIndex &topLeft,
                                const QModelIndex &bottomRight)
{
    hashIsDirty = true;
    Q_AbstractItemView::dataChanged(topLeft, bottomRight);
}

This method is called whenever model data changes. We set hashIsDirty to true to make sure that when calculateRectsIfNecessary() is called it will update the rectForRow hash when the hash is next needed, and then we call the base class implementation. Notice that we do not call viewport->update() to schedule a repaint. The changed data might not be visible so a repaint might be unnecessary, and if it were necessary, the dataChanged() base class implementation would schedule the repaint for us.
QAbstractItemView Subclasses

```
void TiledListView::rowsInserted(const QModelIndex &parent, int start,
                                 int end)
{
    hashIsDirty = true;
    QAbstractItemView::rowsInserted(parent, start, end);
}

void TiledListView::rowsAboutToBeRemoved(const QModelIndex &parent,
                                          int start, int end)
{
    hashIsDirty = true;
    QAbstractItemView::rowsAboutToBeRemoved(parent, start, end);
}
```

If new rows are inserted into the model, or if rows are going to be removed, we must make sure that the view responds appropriately. These cases are easily handled by passing the work on to the base class; all that we must do is ensure that the `rectForRow` hash is marked as dirty so that it will be recalculated if necessary—for example, if the base class methods schedule a repaint.

```
QModelIndex TiledListView::moveCursor(
    QAbstractItemView::CursorAction cursorAction,
    Qt::KeyboardModifiers)
{
    QModelIndex index = currentIndex();
    if (index.isValid()) {
        if (((cursorAction == MoveLeft && index.row() > 0) ||
             (cursorAction == MoveRight &&
              index.row() + 1 < model()->rowCount())) {
            const int offset = (cursorAction == MoveLeft ? -1 : 1);
            index = model()->index(index.row() + offset,
                                    index.column(), index.parent());
        }
        else if (((cursorAction == MoveUp && index.row() > 0) ||
                   (cursorAction == MoveDown &&
                    index.row() + 1 < model()->rowCount())) {
            QFontMetrics fm(font());
            const int RowHeight = (fm.height() + ExtraHeight) *
                                   (cursorAction == MoveUp ? -1 : 1);
            QRect rect = viewportRectForRow(index.row()).toRect();
            QPoint point(rect.center().x(),
                          rect.center().y() + RowHeight);
            while (point.x() >= 0) {
                index = indexAt(point);
                if (index.isValid())
                    break;
        }
    } else {
        // Handle invalid index cases...
    }
Just as the `calculateRectsIfNecessary()` method is at the heart of the Tiled-ListView’s appearance, this method is at the heart of its behavior. The method must return the model index of the item that the requested move action should navigate to—or an invalid model index if no move should occur.

If the user presses the left (or right) arrow key we must return the model index of the previous (or next) item in the list—or of the current item if the previous (or next) item is the list model’s first (or last) item. This is easily achieved by creating a new model index based on the current model index but using the previous (or next) row.

Handling the up and down arrow keys is slightly more subtle than handling the left and right arrow keys. In both cases we must compute a point above or below the current item. It doesn’t matter if the computed point is outside the viewport, so long as it is within an item’s rectangle.

If the user presses the up (or down) arrow key we must return the model index of the item that appears above (or below) the current item. We begin by getting the current item’s rectangle in the viewport. We then create a point that is exactly one row above (or below) the current item vertically, and at the item’s center horizontally. We then use the `indexAt()` method to retrieve the model index for the item at the given point. If we get a valid model index, there is an item above (or below) the current one, and we have its model index, so we are finished and can return that index.

But the model index might be invalid: this is possible because there may not be an item above (or below). Recall from the screenshot (208) that the items at the right-hand edge are ragged, because lines are of different lengths. If this is the case, we move the point left by the width of one “n” character and try again, repeatedly moving left until either we find an item (i.e., until we get a valid model index), or until we move beyond the left edge which means that there is no item above (or below). There will be no item above (or below) when the user presses the up (or down) arrow on an item that is in the first (or last) line.

If the `moveCursor()` method returns an invalid `QModelIndex`, the `QAbstractItemView` base class harmlessly does nothing.

We have not written any code for handling selections—and we don’t need to since we are using the `QAbstractItemView` API. If the user moves with the Shift key held down, the selection will be extended to create a selection of contiguous items. Similarly, while the user holds down the Ctrl key (⌘ key on Mac OS X),
they can click arbitrary items and each one will be selected to create a selection that may include non-contiguous items.

We have left the implementation of support for the Home, End, PageUp, and PageDown keys as an exercise—they just require that the moveCursor() method be extended to handle more CursorActions (such as QAbstractItemView::MoveHome and QAbstractItemView::MovePageUp).

```cpp
int TiledListView::horizontalOffset() const
{
    return horizontalScrollBar()->value();
}

int TiledListView::verticalOffset() const
{
    return verticalScrollBar()->value();
}
```

These pure virtual methods must be reimplemented. They must return the x- and y-offsets of the viewport within the (ideal-sized) widget. They are trivial to implement since the scrollbars' values are the offsets we need.

```cpp
void TiledListView::scrollContentsBy(int dx, int dy)
{
    scrollDirtyRegion(dx, dy);
    viewport()->scroll(dx, dy);
}
```

This method is called when the scrollbars are moved; its responsibility is to ensure that the viewport is scrolled by the amounts given, and to schedule an appropriate repaint. Here we set up the repaint by calling the QAbstractItemView::scrollDirtyRegion() method, before performing the scroll. Alternatively, instead of calling scrollDirtyRegion(), we could call viewport->update(), after performing the scroll.

The base class implementation simply calls viewport->update() and doesn’t actually scroll. Note that if we want to scroll programmatically we should do so by calling QScrollBar::setValue() on the scrollbars, not by calling this method.

```cpp
void TiledListView::setSelection(const QRect &rect, QFlags<QItemSelectionModel::SelectionFlag> flags)
{
    QRect rectangle = rect.translated(horizontalScrollBar()->value(), verticalScrollBar()->value()).normalized();
    calculateRectsIfNecessary();
    QHashIterator<int, QRectF> i(rectForRow);
    int firstRow = model()->rowCount();
```
int lastRow = -1;
while (i.hasNext()) {
    i.next();
    if (i.value().intersects(rectangle)) {
        firstRow = firstRow < i.key() ? firstRow : i.key();
        lastRow = lastRow > i.key() ? lastRow : i.key();
    }
}
if (firstRow != model()->rowCount() && lastRow != -1) {
    QItemSelection selection(
        model()->index(firstRow, 0, rootIndex()),
        model()->index(lastRow, 0, rootIndex()));
    selectionModel()->select(selection, flags);
} else {
    QModelIndex invalid;
    QItemSelection selection(invalid, invalid);
    selectionModel()->select(selection, flags);
}

This pure virtual method is used to apply the given selection flags to all the items that are in or touching the specified rectangle. The actual selection must be made by calling QAbstractItemView::selectionModel()->select(). The implementation shown here is very similar to the one used by Qt’s chart example.

The rectangle is passed using viewport coordinates, so we begin by creating a rectangle that uses widget coordinates since those are the ones used by the rectForRow hash. We then iterate over all the rows (items) in the hash—in arbitrary order—and if an item’s rectangle intersects the given rectangle, we expand the first and last rows that the selection spans to include the item if it isn’t already included.

If we have valid first and last selection rows, we create a QItemSelection that spans these rows (inclusively) and update the view’s selection model. But if one or both rows are invalid, we create an invalid QModelIndex and update the selection model using it.

QRegion TiledListView::visualRegionForSelection(
    const QItemSelection &selection) const
{
    QRegion region;
    foreach (const QItemSelectionRange &range, selection) {
        for (int row = range.top(); row <= range.bottom(); ++row) {
            for (int column = range.left(); column < range.right();
                ++column) {


```
QModelIndex index = model()->index(row, column, rootIndex());
    region += visualRect(index);
}
}
return region;

This pure virtual method must be reimplemented to return the QRegion that encompasses all the view's selected items as shown in the viewport and using viewport coordinates. The implementation we have used is very similar to that used by Qt's chart example.

We start by creating an empty region. Then we iterate over all the selections—if there are any. For each selection we retrieve a model index for every item in the selection, and add each item's visual rectangle to the region.

Our visualRect() implementation calls viewportRectForRow() which in turn retrieves the rectangle from the rectForRow hash and returns it transformed into viewport coordinates (since rectForRow's rectangles use widget coordinates). In this particular case we could have bypassed the visualRect() call and made direct use of the rectForRow hash, but we preferred to do a more generic implementation that is easy to adapt for other custom views.

void TiledListView::paintEvent(QPaintEvent*)
{
    QPainter painter(viewport());
    painter.setRenderHints(QPainter::Antialiasing |
                    QPainter::TextAntialiasing);
    for (int row = 0; row < model()->rowCount(rootIndex()); ++row) {
        QModelIndex index = model()->index(row, 0, rootIndex());
        QRectF rect = viewportRectForRow(row);
        if (!rect.isValid() || rect.bottom() < 0 ||
            rect.y() > viewport()->height())
            continue;
        QStyleOptionViewItem option = viewOptions();
        option.rect = rect.toRect();
        if (selectionModel()->isSelected(index))
            option.state |= QStyle::State_Selected;
        if (currentIndex() == index)
            option.state |= QStyle::State_HasFocus;
        itemDelegate()->paint(&painter, option, index);
        paintOutline(&painter, rect);
    }
Painting the view is surprisingly straightforward since every item’s rectangle has already been computed and is available in the rectForRow hash. But notice that we paint on the widget’s viewport, not on the widget itself. And as usual, we explicitly switch on antialiasing since we cannot assume what the default render hints are.

We iterate over every item and get each one’s model index and its rectangle in viewport coordinates. If the rectangle is invalid (it shouldn’t be), or if it is not visible in the viewport—that is, its bottom edge is above the viewport, or its y-coordinate is below the viewport—we don’t bother to paint it.

For those items we do paint, we begin by retrieving the QStyleOptionViewItem supplied by the base class. We then set the option’s rectangle to the item’s rectangle—converting from QRectF to QRect using QRectF::toRect()—and update the option’s state appropriately if the item is selected or is the current item.

Most importantly, we do not paint the item ourselves! Instead we ask the view’s delegate—which could be the base class’s built-in QStyledItemDelegate, or a custom delegate set by the class’s client—to paint the item for us. This ensures that the view supports custom delegates.

The items are painted in lines, packing them in to make as much use of the available space as possible. But because each item’s text could contain more than one word we need to help the user to be able to visually distinguish between different items. We do this by painting an outline around each item.

```cpp
void TiledListView::paintOutline(QPainter *painter,
                                 const QRectF &rectangle)
{
    const QRectF rect = rectangle.adjusted(0, 0, -1, -1);
    painter->save();
    painter->setPen(QPen(palette().dark().color(), 0.5));
    painter->drawRect(rect);
    painter->setPen(QPen(Qt::black, 0.5));
    painter->drawLine(rect.bottomLeft(), rect.bottomRight());
    painter->drawLine(rect.bottomRight(), rect.topRight());
    painter->restore();
}
```

The outline is drawn by painting a rectangle, and then painting a couple of lines—one just below the bottom of the rectangle, and one just to the right of the rectangle—to provide a very subtle shadow effect.

```cpp
void TiledListView::resizeEvent(QResizeEvent*)
{
    hashIsDirty = true;
    calculateRectsIfNecessary();
}
If the view is resized we must recalculate all the items’ rectangles and update the scrollbars. We have already seen the calculateRectsIfNecessary() method (212 ◁), so we just need to review updateGeometries().

```cpp
void TiledListView::updateGeometries()
{
    QFontMetrics fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    horizontalScrollBar()->setSingleStep(fm.width("n"));
    horizontalScrollBar()->setPageStep(viewport()->width());
    horizontalScrollBar()->setRange(0,
        qMax(0, idealWidth - viewport()->width()));
    verticalScrollBar()->setSingleStep(RowHeight);
    verticalScrollBar()->setPageStep(viewport()->height());
    verticalScrollBar()->setRange(0,
        qMax(0, idealHeight - viewport()->height()));
}
```

This protected slot was introduced with Qt 4.4 and is used to update the view’s child widgets—for example, the scrollbars.

The widget’s ideal width and height are calculated in calculateRectsIfNecessary(). The height is always sufficient to show all the model’s data, and so is the width, if the viewport is wide enough to show the widest item. As mentioned earlier, it does not really matter what the view widget’s actual size is, since the user only ever sees the viewport.

We make the horizontal scrollbar’s single step size (i.e., how far it moves when the user clicks one of its arrows) the width of an “n”, that is, one character. And we make its page step size (i.e., how far it moves when the user clicks left or right of the scrollbar’s slider) the width of the viewport. We also set the horizontal scrollbar’s range to span from 0 to the widget’s ideal width, not counting the viewport’s width (because that much can already be seen). The vertical scrollbar is set up in an analogous way.

```cpp
void TiledListView::mousePressEvent(QMouseEvent *event)
{
    QAbstractItemView::mousePressEvent(event);
    setCurrentIndex(indexAt(event->pos()));
}
```

This is the last event handler that we need to implement. We use it to make the item the user clicked the selected and current item. Because our view is a QAbstractItemView subclass, which itself is a QAbstractScrollArea subclass, the
mouse event’s position is in viewport coordinates. This isn’t a problem since the
indexAt() method expects the QPoint it is passed to be in viewport coordinates.

One final point to note about the TiledListView class is that it assumes that
the user is using a left to right language, such as English. Arabic and Hebrew
users will find the class confusing because they use right to left languages.
We leave modifying the class to work both left to right and right to left as an
exercise for the reader. (The widget’s left to right or right to left status is avail-
able from QWidget::layoutDirection(); this is normally the same as QApplica-
tion::layoutDirection() but it is best to use the QWidget variant to be strictly
correct.)

Like all of Qt’s standard view classes, TiledListView has a one to one correspon-
dence between data items and display items. But in some situations we might
want to visualize two or more items combined together in some way—but this
isn’t supported by the QAbstractItemView API, nor can it be achieved by using
custom delegates. Nonetheless, we can still produce a view that visualizes our
data exactly as we want—as we will see in the next section—but in doing so we
must eschew the QAbstractItemView API and provide our own API instead.

Model-Specific Visualizing Views

In this section we will create a view class from scratch as a QWidget subclass,
and will provide our own API that is different from the QAbstractItemView API.
It would have been possible to create a QAbstractItemView subclass, but since
the view we want to create is specific to one particular model and shows some
of its items combined, there seemed little point in making it comply with an
API that wasn’t needed or relevant.

The visualizer we will create is designed to present a table of census data. The
model that holds the data is a table model, where each row holds a year, a count
of the males, a count of the females, and the total of males and females. Fig-

\begin{figure}
\centering
\includegraphics[width=\textwidth]{census_visualizer.png}
\caption{A QTableView and a CensusVisualizer view}
\end{figure}
Figure 6.3 shows the central area of the Census Visualizer application (censusvisualizer). The area has two views of the data. On the left a standard QTableView presents the data in the conventional way. On the right a CensusVisualizer view is used to represent the data, and it does so by showing the males and females as colored bars proportional to their numbers and using gradient fills.

We could not use Qt’s QHeaderView to present the visualizer’s headers because we have combined two columns. Because of this we have created the CensusVisualizer view as a QWidget that aggregates three other widgets inside itself: a custom CensusVisualizerHeader to provide the horizontal header, a custom CensusVisualizerView to visualize the data, and a QScrollArea to contain the CensusVisualizerView and provide support for scrolling and resizing. The relationships between these classes are shown in Figure 6.4.

![Figure 6.4](image)

We will start by looking at the creation of the visualizer in the application’s main() function.

```cpp
CensusVisualizer *censusVisualizer = new CensusVisualizer;
censusVisualizer->setModel(model);
```

This looks and works exactly like we’d expect—the visualizer is created and we call CensusVisualizer::setModel() to give it the model. Later on in the program’s main() function, the QTableView is created, both views are laid out, and various signal-slot connections are made to give the program its behavior. We will ignore all of that and just focus our attention on the design and coding of the visualizer class and its aggregated header and view classes.

## The Visualizer Widget

The visualizer widget is the one that our clients will use directly, so we will start by reviewing the CensusVisualizer class. This will give us the context we need to then go on to look at the two custom classes that the visualizer aggregates to provide its appearance. Here’s the CensusVisualizer’s definition in the header file, but excluding its private data:
class CensusVisualizer : public QWidget
{
    Q_OBJECT

public:
    explicit CensusVisualizer(QWidget *parent=0);
    QAbstractItemModel *model() const { return m_model; }
    void setModel(QAbstractItemModel *model);
    QScrollArea *scrollArea() const { return m_scrollArea; }
    int maximumPopulation() const { return m_maximumPopulation; }
    int widthOfYearColumn() const { return m_widthOfYearColumn; }
    int widthOfMaleFemaleColumn() const;
    int widthOfTotalColumn() const { return m_widthOfTotalColumn; }
    int selectedRow() const { return m_selectedRow; }
    void setSelectedRow(int row);
    int selectedColumn() const { return m_selectedColumn; }
    void setSelectedColumn(int column);
    void paintItemBorder(QPainter *painter, const QPalette &palette,
                          const QRect &rect);
    QString maleFemaleHeaderText() const;
    int maleFemaleHeaderTextWidth() const;
    int xOffsetForMiddleOfColumn(int column) const;
    int yOffsetForRow(int row) const;

public slots:
    void setCurrentIndex(const QModelIndex &index);

signals:
    void clicked(const QModelIndex&);

private:
    ...}

Although the data isn’t shown, it is worth noting that the aggregated CensusVisualizerHeader is held in the header private member variable and the CensusVisualizerView is held in the view private member variable—both are pointers, of course. The class also holds a pointer to the model and to the QScrollArea that contains the CensusVisualizerView. The other private member data are all integers most of whose getters are implemented inline and shown here, and whose setters—for those that are writable—we will review shortly.

The maximum population is used by the view to compute the maximum widths of the male–female bars to make the best use of the available space, and is calculated whenever setModel() is called.

The width getters are used by both the header and the view when they are painting themselves. The selected row and column are kept track of and their
values are used by the header to highlight the selected column, and by the view
to highlight the selected item (or the selected male–female item pair).

The signal is included so that if the selected item is changed by the user clicking on the view, we emit a clicked() signal to notify any interested objects.

The non-inline parts of the CensusVisualizer class are the constructor and ten
methods. The paintItemBorder(), maleFemaleHeaderText(), and maleFemaleHeaderTextWidth() methods are used by the aggregated header and view, so we will defer our review of them until we see them used, but we will review all the others here.

```cpp
const int Invalid = -1;
CensusVisualizer::CensusVisualizer(QWidget *parent)
    : QWidget(parent), m_model(0), m_selectedRow(Invalid),
      m_selectedColumn(Invalid), m_maximumPopulation(Invalid)
{
    QFontMetrics fm(font());
    m_widthOfYearColumn = fm.width("W9999W");
    m_widthOfTotalColumn = fm.width("W9,999,999W");
    view = new CensusVisualizerView(this);
    header = new CensusVisualizerHeader(this);
    m_scrollArea = new QScrollArea;
    m_scrollArea->setBackgroundRole(QPalette::Light);
    m_scrollArea->setWidget(view);
    m_scrollArea->installEventFilter(view);
    QBoxLayout*layout = new QVBoxLayout;
    layout->addWidget(header);
    layout->addWidget(m_scrollArea);
    layout->setContentsMargins(0, 0, 0, 0);
    layout->setSpacing(0);
   setLayout(layout);
    connect(view, SIGNAL(clicked(const QModelIndex&)),
            this, SIGNAL(clicked(const QModelIndex&)));
}
```

We begin by setting fixed widths for the year and total columns based on the
largest numbers we expect them to handle, plus some margin.* The width of
the total column set here is just an initial default; the actual width is recalculat-
ed in the setModel() method and depends on the model's maximum population.

We then create the aggregated view and header widgets. Although we pass
this as their parent, because we use a QScrollArea to contain the view, the view
will be reparented to the QScrollArea.

*In this book the practice is to use “W”s when we want horizontal padding, and “n”s when we want
a single character’s width, for example, for horizontal scrolling.
The QScrollArea class is unusual for Qt in that it is not designed to be sub-classed. Instead the usage pattern is to aggregate the QScrollArea inside another widget as we have done here. Although this approach is by far the easiest to use, if we want to use inheritance, we can derive our subclass from QAbstractScrollArea as some of Qt’s built-in classes do.

We install the view as an event filter for the scroll area—this means that every event that goes to the scroll area will first be sent to the view’s eventFilter() method. We will see why this is necessary when we review the CensusVisualizerView class further on.

The layout is quite conventional except that we set the layout’s margins and spacing to 0; this makes the CensusVisualizer have the same look as other widgets, with no extraneous border area, and with no gap between the CensusVisualizerHeader and the CensusVisualizerView (contained in the QScrollArea).

The connection is slightly unusual since it is a signal–signal connection. These set up a relationship whereby when the first signal is emitted the second signal is emitted as a consequence. So in this case, when the user clicks the view (i.e., to select an item), the view’s clicked() signal goes to the CensusVisualizer, and this in turn emits a matching clicked() signal with the same QModelIndex parameter. This means that CensusVisualizer clients can connect to the CensusVisualizer’s clicked() signal without having to know or care about the internals. This makes the CensusVisualizer much more of a self-contained component than it would be if, for example, it exposed the widgets it aggregates.

```cpp
enum {Year, Males, Females, Total};

void CensusVisualizer::setModel(QAbstractItemModel *model) {
    if (model) {
        QLocale locale;
        for (int row = 0; row < model->rowCount(); ++row) {
            int total = locale.toInt(model->data(
                model->index(row, Total)).toString());
            if (total > m_maximumPopulation)
                m_maximumPopulation = total;
        }
        QString population = QString::number(m_maximumPopulation);
        population = QString("%1%2")
            .arg(population.left(1).toInt() + 1)
            .arg(QString(population.length() - 1, QChar('0')));
        m_maximumPopulation = population.toInt();
        QFontMetrics fm(font());
        m_widthOfTotalColumn = fm.width(QString("W%1%2W")
            .arg(population)
            .arg(QString(population.length() / 3, ',')));
    }
}```
m_model = model;
header->update();
view->update();
}

When a new model is set we must tell the header and view to update themselves. But first we must calculate a suitable maximum population. We do this by finding the biggest total population in the data, and then rounding it up to the smallest number with a most significant digit that is one larger. For example, if the biggest total is 8 392 174, the maximum becomes 9 000 000.

The algorithm used is very crude, but effective: we create a string that starts with the number's first digit plus one, followed by one less than as many zeros as there are digits in the number, and convert this string to an int. For the zeros we used one of QString's two-argument constructors that takes a count and a character and returns a string that contains exactly count occurrences of the character.

Notice that we cannot retrieve the totals using model->data(model->index(row, Total).toInt(), because the model happens to hold the values as localized strings (e.g., “8,392,174” in the U.S. and UK, and “8.392.174” in Germany), rather than as integers. The solution is to use toString() to extract the data and then to use QLocale::toInt()—which takes an integer in the form of a localized string and returns the integer value.

The QLocale class also has corresponding toFloat() and toDouble() methods, as well as methods for other integral types—such as toUInt()—and also methods for extracting dates and times from localized strings. When a QLocale is constructed it defaults to using the application's current locale, but this can be overridden by using the one-argument constructor and a locale name that has the ISO 639 language code and ISO 3166 country code, or the two-argument constructor using Qt's language and country enums.

In the constructor we set an initial width for the total column, but here we can set one that is appropriate for the actual total. The width is set to be the number of pixels needed to show the maximum number, plus space for a couple of “W”s for padding, plus space for a comma (or other grouping marker) for every three digits.

const int ExtraWidth = 5;
int CensusVisualizer::widthOfMaleFemaleColumn() const
{
    return width() - (m_widthOfYearColumn +
        m_widthOfTotalColumn + ExtraWidth +
        m_scrollArea->verticalScrollBar()->sizeHint().width());
}
This method returns a suitable width for the male–female column. It calculates the width as the maximum available width given the width of the CensusVisualizer itself, the widths of the other two columns, the width of the scroll area’s vertical scrollbar, and a little bit of margin. This ensures that when the CensusVisualizer is resized, any extra width is always given to the male–female column.

```cpp
void CensusVisualizer::setSelectedRow(int row)
{
    m_selectedRow = row;
    view->update();
}

void CensusVisualizer::setSelectedColumn(int column)
{
    m_selectedColumn = column;
    header->update();
}
```

If the selected row is changed programmatically, the view must update itself to show the correct highlighted item. Similarly, if the selected column is changed, the header must highlight the title of the selected column.

```cpp
void CensusVisualizer::setCurrentIndex(const QModelIndex &index)
{
    setSelectedRow(index.row());
    setSelectedColumn(index.column());
    int x = xOffsetForMiddleOfColumn(index.column());
    int y = yOffsetForRow(index.row());
    m_scrollArea->ensureVisible(x, y, 10, 20);
}
```

This slot is provided as a service to clients, so that they can change the CensusVisualizer’s selected item by using a signal–slot connection.

Once the row and column are set, we make sure that they are visible in the scroll area. The QScrollArea::ensureVisible() method takes x- and y-coordinates, and optionally some horizontal and vertical margin (which defaults to 50 pixels each). We’ve reduced the margins so as to avoid unwanted scrolling when the user clicks the top or bottom visible row.

There is actually a trade-off to be made here. If the vertical margin is too large, clicking the top or bottom item will cause unnecessary scrolling. And if the margin is too small, if the user Tabs to the widget and uses the down arrow to reach the bottom item, the item won’t be shown fully.
int CensusVisualizer::xOffsetForMiddleOfColumn(int column) const
{
    switch (column) {
    case Year: return widthOfYearColumn() / 2;
    case Males: return widthOfYearColumn() +
                 (widthOfMaleFemaleColumn() / 4);
    case Females: return widthOfYearColumn() +
                 ((widthOfMaleFemaleColumn() * 4) / 3);
    default: return widthOfYearColumn() +
                 widthOfMaleFemaleColumn() +
                 (widthOfTotalColumn() / 2);
    }
}

This method is used to get a suitable x-offset for the current column. It does this by computing the given column's horizontal midpoint based on the column widths.

const int ExtraHeight = 5;

int CensusVisualizer::yOffsetForRow(int row) const
{
    return static_cast<int>((QFontMetricsF(font()).height() + ExtraHeight) * row);
}

This method is used to get the y-offset for the given row, which it calculates by multiplying the height of one row by the given row index.

The x- and y-offsets returned by the xOffsetForMiddleOfColumn() and yOffsetForRow() methods assume that the CensusVisualizerView is exactly the size needed to show all the data. This assumption is valid because the CensusVisualizerView enforces it—as we will see when we look at the CensusVisualizerView::eventFilter() method. This means that even though only a portion of the view might be displayed, we don't have to do any scrolling-related computations because the QScrollArea that contains the CensusVisualizerView takes care of them for us.

We have now finished reviewing the CensusVisualizer class. Apart from the constructor and the setModel() method, it has very little code. This is because all of the widget's appearance, and most of its behavior, are handled by the instances of the CensusVisualizerHeader and CensusVisualizerView classes that the CensusVisualizer creates and lays out in its constructor. We will now review each of these aggregated classes in turn, starting with the header.
The Visualizer’s Aggregated Header Widget

The CensusVisualizerHeader widget provides the column headers for the CensusVisualizer, as Figure 6.3 illustrates (224 ➤). Since we are painting it ourselves we have taken the opportunity to give it a stronger three-dimensional look than QHeaderView normally provides by using a different gradient fill. (If we had wanted to exactly match QHeaderView, we could have done the painting using QStyle methods.)

The class’s definition in the header file is quite simple; here is its complete public API:

```cpp
class CensusVisualizerHeader : public QWidget
{
  Q_OBJECT

public:
  explicit CensusVisualizerHeader(QWidget *parent)
    : QWidget(parent) {}

  QSize minimumSizeHint() const;
  QSize sizeHint() const { return minimumSizeHint(); }

protected:
  void paintEvent(QPaintEvent *event);

private:
  QSize minimumSizeHint() const
  {
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(parent);
    Q_ASSERT(visualizer);
    return QSize(visualizer->widthOfYearColumn() +
                 visualizer->maleFemaleHeaderTextWidth() +
                 visualizer->widthOfTotalColumn(),
                 QFontMetrics(font()).height() + ExtraHeight);
  }
```

The constructor has an empty body. The only methods that are implemented are the `minimumSizeHint()`, the `sizeHint()`, the `paintEvent()`, and a couple of private helper methods (covered later) that `paintEvent()` calls.

```cpp```

The column widths are available from the parent CensusVisualizer, so we must cast—using `qobject_cast<>()` as here, or `dynamic_cast<>()`—to get a pointer to the parent that we can use to access the data we require. (If `dynamic_cast<>()` is used the compiler must have RTTI—Run Time Type Information—turned on, which most do by default nowadays.) The minimum width we need is the
sum of the widths of all the columns, and the minimum height is the height of a character in the widget’s font plus some margin.

The `maleFemaleHeaderTextWidth()` method, and the method it depends on, are provided by the `CensusVisualizer` class since they are used by both of the aggregated custom widgets. We show them here for completeness.

```cpp
int CensusVisualizer::maleFemaleHeaderTextWidth() const
{
    return QFontMetrics(font()).width(maleFemaleHeaderText());
}

QString CensusVisualizer::maleFemaleHeaderText() const
{
    if (!m_model)
        return " - ";
    return QString("%1 - %2")
        .arg(m_model->headerData(Males, Qt::Horizontal).toString())
        .arg(m_model->headerData(Females, Qt::Horizontal)
            .toString());
}
```

The `maleFemaleHeaderTextWidth()` method returns the width needed by the male–female column to show its title, and the `maleFemaleHeaderText()` method returns the title itself.

```cpp
void CensusVisualizerHeader::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHints(QPainter::Antialiasing|
        QPainter::TextAntialiasing);
    paintHeader(&painter, height());
    painter.setPen(QPen(palette().button().color().darker(), 0.5));
    painter.drawRect(0, 0, width(), height());
}
```

The `paintEvent()` method sets up the painter, passes most of the work on to the `paintHeader()` method, and finishes off by drawing a rectangle around the entire header.

```cpp
void CensusVisualizerHeader::paintHeader(QPainter *painter,
const int RowHeight)
{
    const int Padding = 2;
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());
    Q_ASSERT(visualizer);
```
paintHeaderItem(painter,
    QRect(0, 0, visualizer->widthOfYearColumn() + Padding,
          RowHeight),
    visualizer->model()->headerData(Year, Qt::Horizontal)
          .toString(),
    visualizer->selectedColumn() == Year);
paintHeaderItem(painter,
    QRect(visualizer->widthOfYearColumn() + Padding, 0,
          visualizer->widthOfMaleFemaleColumn(), RowHeight),
    visualizer->maleFemaleHeaderText(),
    visualizer->selectedColumn() == Males ||
    visualizer->selectedColumn() == Females);
}

This method paints each column header in turn. For each one it calls 
paintHeaderItem(), passing it the painter, the rectangle in which to do the painting, 
the text to paint, and whether this item (i.e., this column) is selected. We have 
 omitted the code for the total column since it is very similar to that used for the 
year column.

```cpp
void CensusVisualizerHeader::paintHeaderItem(QPainter *painter, 
    const QRect &rect, const QString &text, bool selected)
{
    CensusVisualizer *visualizer = qobject_cast<CensusVisualizer*>(
        parent());
    Q_ASSERT(visualizer);
    int x = rect.center().x();
    QLinearGradient gradient(x, rect.top(), x, rect.bottom());
    QColor color = selected ? palette().highlight().color()
        : palette().button().color();
    gradient.setColorAt(0, color.darker(125));
    gradient.setColorAt(0.5, color.lighter(125));
    gradient.setColorAt(1, color.darker(125));
    painter->fillRect(rect, gradient);
    visualizer->paintItemBorder(painter, palette(), rect);
    painter->setPen(selected ? palette().highlightedText().color() : palette().buttonText().color());
    painter->drawText(rect, text, QTextOption(Qt::AlignCenter));
}
```

This is the method that actually paints each header item. We begin by getting 
a pointer to the CensusVisualizer since we use one of its methods. Then we cre-
ate a linear gradient whose coloring depends on whether this item is selected. 
The gradient goes from a lighter color in the middle to a darker color at the top 
and bottom, using lighter and darker colors than the ones used by QHeaderView,
to produce a stronger three-dimensional effect. Once the gradient is set up we use it to paint the item’s background. Next we draw an outline around the item—actually we draw just two lines, one along the bottom, and the other on the right edge. And finally, we draw the text centered in the middle.

For completeness, here is the `paintItemBorder()` method:

```cpp
void CensusVisualizer::paintItemBorder(QPainter *painter,
    const QPalette &palette, const QRect &rect)
{
    painter->setPen(QPen(palette.button().color().darker(), 0.33));
    painter->drawLine(rect.bottomLeft(), rect.bottomRight());
    painter->drawLine(rect.bottomRight(), rect.topRight());
}
```

We chose to draw the “outline” using just two lines because in this example it produces a better effect than drawing a rectangle.

This completes our review of the `CensusVisualizerHeader` class. The class is surprisingly straightforward, with most of the work simply a matter of setting up the painter and gradient and doing some simple drawing. This is quite a contrast with the `CensusVisualizerView` class where we must implement both its appearance and its behavior, as we will see in the next subsection.

**The Visualizer’s Aggregated View Widget**

The custom `CensusVisualizerView` widget is used to display the model’s data. It doesn’t matter as such what size the widget is since it is embedded in a `QScrollArea` which provides scrollbars when necessary and generally takes care of all scrolling-related matters for us. This leaves us free to concentrate on the widget’s appearance and behavior. Here is the public part of the widget’s definition from the header file:

```cpp
class CensusVisualizerView : public QWidget
{
    Q_OBJECT

public:
    explicit CensusVisualizerView(QWidget *parent);
    QSize minimumSizeHint() const;
    QSize sizeHint() const;
    signals:
    void clicked(const QModelIndex&);
    protected:
    bool eventFilter(QObject *target, QEvent *event);
```
The class also has several private methods, all of which are used to support painting the data—and which we will review later—and one private data member, a pointer to the parent CensusVisualizer. We will briefly look at the public methods and the slot, and then work our way through the protected event handlers to see what they do and how they do it—but first we will look at the constructor.

```cpp
CensusVisualizerView::CensusVisualizerView(QWidget *parent)
    : QWidget(parent)
{
    visualizer = qobject_cast<CensusVisualizer*>(parent);
    Q_ASSERT(visualizer);
    setFocusPolicy(Qt::WheelFocus);
    setMinimumSize(minimumSizeHint());
}
```

The CensusVisualizerView is created inside the CensusVisualizer constructor and passed as parent the CensusVisualizer itself (227 ➤). Nonetheless we have chosen to keep a private CensusVisualizer pointer member (visualizer), to give us access to the CensusVisualizer, because after the view has been constructed it is given to a QScrollArea—and this takes ownership of the view and becomes the view's parent. (Alternatively we could avoid keeping a member variable and access the visualizer by calling qobject_cast<CensusVisualizer*>(parent()->parent()) instead.)

Qt provides several different focus policies: Qt::NoFocus (useful for labels and other read-only widgets), Qt::TabFocus (the widget accepts focus when tabbed to), Qt::ClickFocus (the widget accepts focus when clicked), Qt::StrongFocus (this combines tab and click focus), and Qt::WheelFocus (this is strong focus plus accepting the focus when the mouse wheel is used). Here we have used Qt::WheelFocus which is the usual choice for editable widgets.

We have omitted the minimumSizeHint() method's implementation since it is almost identical to CensusVisualizerHeader::minimumSizeHint() (232 ➤), the only difference being that here we have the visualizer member built into the class. (The CensusVisualizerHeader's parent is the CensusVisualizer and it isn't reparented, so it doesn't need a separate visualizer member variable.)

```cpp
QSize CensusVisualizerView::sizeHint() const
{
    int rows = visualizer->model()->rowCount() ? visualizer->model()->rowCount() : 1;
    ...
return QSize(visualizer->widthOfYearColumn() +
    qMax(100, visualizer->maleFemaleHeaderTextWidth()) +
    visualizer->widthOfTotalColumn(),
    visualizer->yOffsetForRow(rows));
}

If a model has been set we allow enough room for all its rows; otherwise we
allow room for a single row. The y-offset returned by the CensusVisualizer::
yOffsetForRow() method is the height we need since we pass it a row that is
equal to the number of rows in the model. For the columns we use the fixed
widths calculated when the CensusVisualizer was constructed, plus the comput-
ed width of the male–female column (or 100 pixels, whichever is greater).

bool CensusVisualizerView::eventFilter(QObject *target, QEvent *event)
{
    if (QScrollArea *scrollArea = visualizer->scrollArea()) {
        if (target == scrollArea && event->type() == QEvent::Resize) {
            if (QResizeEvent *resizeEvent =
                static_cast<QResizeEvent*>(event)) {
                QSize size = resizeEvent->size();
                size.setHeight(sizeHint().height());
                int width = size.width() - (ExtraWidth +
                scrollArea->verticalScrollBar()->sizeHint()
                .width());
                size.setWidth(width);
                resize(size);
            }
        }
    }
    return QWidget::eventFilter(target, event);
}

The CensusVisualizerView was made an event filter for the QScrollArea that
contains it (227 ➧). This means that every event that is sent to the QScrollArea
goes to this method first.

The only event we are interested in is QEvent::Resize. When this event occurs,
that is, when the scroll area is resized, we also resize the CensusVisualizerView
widget. We always make the view the height needed to show all of its data,
and we set its width to the available width while allowing for the width of the
vertical scrollbar. This means that if the user has scrolled the view and, for
example, clicks a row, we can work as if the entire widget is visible without
having to account for the scrolling to compute which row was clicked.

Inside an eventFilter() reimplementation we are free, at least in principle, to
do what we like with the event: we can change it, replace it, delete it, or ignore
it. To stop an event from going further (whether or not we do anything with
it), or if we delete an event, we must return true to indicate that it has been handled; otherwise we must return false. Here we make use of the event, but don’t want to interfere with its behavior, so we leave the arguments unchanged and call the base class implementation at the end.

```cpp
void CensusVisualizerView::mousePressEvent(QMouseEvent *event)
{
  int row = static_cast<int>(event->y() /
                             (QFontMetricsF(font()).height() + ExtraHeight));
  int column;
  if (event->x() < visualizer->widthOfYearColumn())
    column = Year;
  else if (event->x() < (visualizer->widthOfYearColumn() +
                          visualizer->widthOfMaleFemaleColumn() / 2))
    column = Males;
  else if (event->x() < (visualizer->widthOfYearColumn() +
                         visualizer->widthOfMaleFemaleColumn()))
    column = Females;
  else
    column = Total;
  visualizer->setSelectedRow(row);
  visualizer->setSelectedColumn(column);
  emit clicked(visualizer->model()->index(row, column));
}
```

The `QMouseEvent::y()` method returns the mouse click’s y-offset relative to the top of the widget. Thanks to the `CensusVisualizerView` being embedded in a `QScrollArea`, and thanks to it always being exactly high enough to hold all the data—something we ensure in the `eventFilter()`—we can work directly with the y-offset no matter whether the widget has been scrolled. So here, we determine the row by dividing the y-offset by the height of one row.

To work out the column, we compare the x-offset: if it is less than the width of the year column then the year column was clicked; if it is less than the width of the year column plus half the width of the male–female column then the male column was clicked; and so on.

Once the row and column are known we tell the `CensusVisualizer` to select them, safe in the knowledge that doing this will also result in `update()` being called both on this view and on the header so that the correct row and column are properly highlighted. And finally, we emit the `clicked()` signal with the model index—as computed by the model—of the selected item, which in turn will cause the `CensusVisualizer` to emit its own `clicked()` signal with the same model index for the benefit of any connected objects.

```cpp
void CensusVisualizerView::keyPressEvent(QKeyEvent *event)
{
```
if (visualizer->model()) {
    int row = Invalid;
    int column = Invalid;
    if (event->key() == Qt::Key_Left) {
        column = visualizer->selectedColumn();
        if (column == Males || column == Total)
            --column;
        else if (column == Females)
            column = Year;
    }
    ... 
    else if (event->key() == Qt::Key_Up)
        row = qMax(0, visualizer->selectedRow() - 1);
    else if (event->key() == Qt::Key_Down)
        row = qMin(visualizer->selectedRow() + 1,
                   visualizer->model()->rowCount() - 1);
    if (row != Invalid ? visualizer->selectedRow() : row;
        column = column == Invalid ? visualizer->selectedColumn()
                          : column;
    if (row != visualizer->selectedRow())
        column != visualizer->selectedColumn()) {
        QModelIndex index = visualizer->model()->index(row,
                                                       column);
        visualizer->setCurrentIndex(index);
        emit clicked(index);
        return;
    }
    QWidget::keyPressEvent(event);
}

This event handler is used to provide navigation inside the view by the use of the keyboard arrow keys.

Inside the CensusVisualizer we keep track of the selected row and column, but in the case of the male and female columns they are visually—and therefore from the user’s perspective—a single column. To account for this, if the user presses the left arrow and the current column is either the male or the female column, we set the column to be the year column. If the current column is the year column, we do nothing, and if the current column is the total column we set the column to be the female column. The handling of right arrow presses is very similar (so we have omitted the code): if the current column is either the male or the female column, we set the column to be the total column. And if the current column is the year column we set it to be the male column, and if the current column is the total column we do nothing.

If the user presses the up arrow, we set the current row to be one less than the current row—or do nothing if they are already on the first row. And similarly,
if the user presses the down arrow, we set the current row to be one more than
the current row—or do nothing if they are already on the last row.

If the new selected row or column or both are different from the currently se-
lected ones, we set the selected row and column. This will cause update() to be
called on the view and the header, and will also ensure that the selected item is
visible. We also emit a clicked() signal with the selected item’s model index.

At the end, if we selected a new item, we must not call the base class implemen-
tation, since we have handled the key press ourselves and don’t want it to go
to the scroll area. This is because the scroll area handles the arrow keys itself,
interpreting them as requests to scroll, which we don’t want—or need—since
we handle the scrolling ourselves. And conversely, if we didn’t handle the key
press, we call the base class implementation to handle it for us.

Compare this method with the mouse event handler where we set the row and
column without having to ensure that the selected item is visible—since the
user must have clicked it. But here, the user could be pressing, say, the down
arrow, on the last visible row, so we must call QScrollArea::ensureVisible()
(which is done by CensusVisualizer::setCurrentIndex(); 230 ◄) so that the view
is scrolled appropriately.

Adding support for the Home, End, PageUp, and PageDown keys follows the same
principles as the code used for the arrow keys, and is left as an exercise. (When
implementing PageUp and PageDown, it is conventional to move up or down by
the widget’s visible height minus one line or row so that the user has one line
of context by which they can orient themselves.)

The eventFilter(), mousePressEvent(), and keyPressEvent() methods that we
have just reviewed provide the view’s behavior. Now we will look at the paint-
Event() and the private helper methods it uses to see how the view’s appearance
is rendered.

```cpp
void CensusVisualizerView::paintEvent(QPaintEvent *event)
{
    if (!visualizer->model())
        return;
    QFontMetricsF fm(font());
    const int RowHeight = fm.height() + ExtraHeight;
    const int MinY = qMax(0, event->rect().y() - RowHeight);
    const int MaxY = MinY + event->rect().height() + RowHeight;
    QPainter painter(this);
    painter.setRenderHints(QPainter::Antialiasing|
        QPainter::TextAntialiasing);
    int row = MinY / RowHeight;
    int y = row * RowHeight;
    for (; row < visualizer->model()->rowCount(); ++row) {
```
This method begins by computing some constants, in particular, the height to allow for each row, and the paint event’s minimum and maximum \( y \)-coordinates, minus or plus one row’s height to ensure that even if only a portion of a row is visible, it is still painted.

Since the widget is inside a \QScrollArea and its height is always precisely that needed to show all the items, we do not \textit{need} to compute any offsets or work out for ourselves what is visible and what isn’t. However, for the sake of efficiency, we should paint only visible items.

The paint event that is passed in has a \QRect that specifies the rectangle that needs repainting. For small widgets we often ignore this rectangle and just repaint the whole thing, but for a model-visualizing widget that could have large amounts of data we want to be more efficient and only paint what needs painting. So with the constants in place, we set up the painter and calculate the first row that needs painting, and that row’s \( y \)-coordinate. (It may be tempting to initialize \( y \) with \( y = \text{MinY} \); but \text{MinY} is not usually the same as \( \text{row} \times \text{RowHeight} \) because of the—desired—integer truncation that occurs in the \( \text{MinY} / \text{RowHeight} \) expression.)

With everything in place, we iterate through the model’s rows, starting at the first one that is visible, and painting each one until the \( y \)-coordinate takes us beyond the rectangle that needs repainting, at which point we stop. This ensures that we retrieve and paint at most the rows that are visible plus two extra rows, which could be a considerable savings if the model has thousands or tens of thousands of rows or more.
This method is used simply to create a suitable rectangle and call a column-specific paint method for each column.

```cpp
void CensusVisualizerView::paintYear(QPainter *painter, int row,
const QRect &rect)
{
paintItemBackground(painter, rect,
    row == visualizer->selectedRow() &&
    visualizer->selectedColumn() == Year);
painter->drawText(rect,
    visualizer->model()->data(
        visualizer->model()->index(row, Year)).toString(),
    QTextOption(Qt::AlignCenter));
}
```

Once the background is painted, all that remains is for the item’s text to be drawn. The text is retrieved from the model and painted centered in its column.

The `CensusVisualizerView::paintTotal()` method is very similar to this one (so we don’t show it), with the only difference being that we right-align the total.

```cpp
void CensusVisualizerView::paintItemBackground(QPainter *painter,
    const QRect &rect, bool selected)
{
painter->fillRect(rect, selected ? palette().highlight() :
            palette().base());
visualizer->paintItemBorder(painter, palette(), rect);
painter->setPen(selected ? palette().highlightedText().color() :
            palette().windowText().color());
}
```

Which background and foreground colors to use depends on whether the item is selected. This method paints the background and the border and sets the pen color ready for the caller to paint its text.

The `paintMaleFemale()` method is slightly longer so we will review it in three parts.

```cpp
void CensusVisualizerView::paintMaleFemale(QPainter *painter,
    int row, const QRect &rect)
{
    QRect rectangle(rect);
    QLocale locale;
    int males = locale.toInt(visualizer->model()->data(
        visualizer->model()->index(row, Males)).toString());
    int females = locale.toInt(visualizer->model()->data(
        visualizer->model()->index(row, Females)).toString());
    ```
We begin by finding out how many males and females there are and the total they sum to. (We discussed the use of QLocale to get numbers from localized strings earlier; \(\text{229} \Leftarrow \).) Then we compute how much width the complete colored bar should occupy and use that to work out the offset by which the bar must be indented from the left and from the right to make the bar the right size within the available rectangle.

```cpp
painter->fillRect(rectangle,
    (row == visualizer->selectedRow() &&
     (visualizer->selectedColumn() == Females ||
      visualizer->selectedColumn() == Males))
    ? palette().highlight() : palette().base());
```

The first thing we paint is the background, with the color determined by whether the males or females column is selected.

```cpp
visualizer->paintItemBorder(painter, palette(), rectangle);
```

```cpp
toward the end, we paint the item's border and then resize the available rectangle—potentially making it smaller—so that it has the correct size and position to serve as the rectangle for drawing the colored bar. Finally, we draw the bar—with a tiny reduction in height—using a gradient fill which goes from dark green to light green (left to right) for the male part, and from light red to dark red (left to right) for the female part.

```cpp
QLinearGradient CensusVisualizerView::maleFemaleGradient(qreal x1, qreal y1, qreal x2, qreal y2, qreal crossOver)
{
    QLinearGradient gradient(x1, y1, x2, y2);
    QColor maleColor = Qt::green;
    QColor femaleColor = Qt::red;
    gradient.setColorAt(0, maleColor.darker());
    gradient.setColorAt(crossOver - 0.001, maleColor.lighter());
    gradient.setColorAt(crossOver + 0.001, femaleColor.lighter());
```
gradient.setColorAt(1, femaleColor.darker());
return gradient;
}

This method is shown for completeness. It creates a linear gradient that goes from dark to light in one color and then from light to dark in another color with a crossover between the colors at the specified position. The crossover point is computed by the caller as males / total; this ensures that the widths of the male and female parts are in correct proportion to their populations.

Qt also has QConicalGradient and QRadialGradient classes with similar APIs.

We have now finished the CensusVisualizer class and its aggregated CensusVisualizerHeader and CensusVisualizerView classes that do so much of the work. Creating custom classes like this is ideal when we have a model that we want to visualize in a unique way and where items are shown combined in some way so that using a custom delegate or a custom view based on the QAbstractItemView API is not sufficient.

We have now completed our review of the TiledListView class, and of the CensusVisualizer class. The TiledListView is much shorter because it didn't have to show any column captions and because it could rely on the base class for some of its functionality. If we want to present model data in unique ways, for example, graphically, or if we want to present some model items combined, then a custom delegate is insufficient and we must use a custom view. If we take the approach used for the CensusVisualizer class, we get complete control, and only have to implement the features that we actually need. However, if we choose to create a QAbstractItemView subclass, we still get complete control, we get some functionality for free, and we get much more potential for reuse—but we are obliged to reimplement all the pure virtual methods, and in general at least those methods listed in Table 6.1 (210 ➔).

This chapter is the last of the four chapters dedicated to Qt’s model/view architecture. In general, it is easiest to start by using a QStandardItemModel, subclassing it (or QStandardItem) to make the data serializable and deserializable. Later on, if the need arises, a custom model can always be used as a drop-in replacement. Similarly, using one of Qt’s standard views is the best way to start viewing model data, and if the need for customizing the appearance or editing of items is required, it is best—and easiest—to use custom delegates. However, if no combination of standard view and custom delegate can visualize the data in the desired way, then we must create a custom view using one of the approaches shown in this chapter.
All non-global functions and methods are listed under their class (or their class’s base class—which could be QWidget or Qobject), and as top-level terms in their own right. Where a method or function name is close enough to a concept, the concept is not usually listed. For example, there is no entry for “joining a string list”, but there are entries for the QString::join() method. Note also that many references are purely to quoted code (i.e., to show examples of use).

Symbols

key, 218, 345
() (call function or method operator); see operator()
*(multiplication and dereference operator); see operator*()
+= (augmented assignment operator); see operator+=()
- (negation and subtraction operator); see operator-()
< (less than operator); see operator<()
<< (append and put to operator); see operator<<()
== (equal operator); see operator==()
>>(get from operator); see operator>>()
[] (subscript operator); see operator[]()

actions() (QWidget), 29
actions, exclusive, 13
activeThreadCount() (QThreadPool), 251, 256, 257
addAction()
   QActionGroup, 12
   QMenu, 12, 347, 349, 430, 463
addActions() (QWidget), 345
addButton() (QMessageBox), 101, 137
addDockWidget() (QMainWindow), 414
addItem()
   QComboBox, 35, 442, 483
   QGraphicsScene, 397, 398, 435, 450
addItems() (QComboBox), 203
addSeparator() (QMenu), 12, 462
addToLeftJavaScriptWindowObject()
   QWebView, 40
addTransition() (QState), 479, 484, 487, 489
addWidget() (QGraphicsScene), 396, 397
adjust() (QRect/QRectF), 382
adjusted() (QRect/QRectF), 191, 222, 243, 457
advance() (QGraphicsItem), 403
aggregating widgets, 225
<algorithm>, 432
algorithms
   Kahan summation, 280
   Kuhn-Munkres, 15
alpha channel; see transparency
Alt key, 15
alternateBase() (QPalette), 336
anchor() (QTextCursor), 328, 332
animation, 133
  bouncing, 472
  framework, 469–474, 481–490
  of dialogs, 481–490
  of properties, 470
  policy, 53
QDockWidget, 414
see also video
antialiasing; see QGraphicsView::setRenderHints() and QPaint::setRenderHints()
antialiasing policy, 376, 379, 423
append()
  QList, 285
  QVector, 433
appendPlainText() (QPlainTextEdit), 256
appendRow()
  QStandardItem, 312
  QStandardItemModel, 99, 106, 312
application settings; see QSettings
applicationDirPath() (QApplication), 33
applicationName() (QApplication), 32, 56
applicationPathOf() (AQP), 33, 57
applications
  settings policy, 20
  taskbar tray, 7
  terminating, 8, 79, 257
web-based, 5
AQP
  accelerateActions(), 26, 345
  accelerateMenu(), 12, 15, 29, 56, 132, 347
  accelerateWidget(), 15, 93, 291
  accelerateWidgets(), 15, 395
  applicationPathOf(), 33, 57
  chunkSizes(), 252, 253, 258, 309
  Error, 104, 105, 126, 144
  filenameFilter(), 58, 84
  hoursMinutesSecondsForMsec(), 74
  information(), 61, 280
AQP (cont.)
  okToClearData(), 61, 136–138
  okToDelete(), 61, 100, 101, 140
  question(), 61, 285
  radiansFromDegrees(), 192
  suffixesForMimeTypes(), 71, 72
  warning(), 60, 61, 76, 77, 374
arg() (QString), 13, 18, 74, 100, 187, 202, 205, 295
aspect ratio, 382, 422, 440
assertion policy, 200
assignProperty() (QState), 478, 479, 483, 486, 487
at()
  QList, 120, 150, 183, 432, 471
  QString, 326
  QVector, 432
atBlockEnd() (QTextCursor), 328
atBlockStart() (QTextCursor), 328
atEnd() (QTextCursor), 328
atStart() (QTextCursor), 328
audio
  category, 66, 81
  codecs, 53
  meta-data, 71
  policy, 53, 55
  aural cues, 53
availableGeometry() (QDesktopWidget), 377

B
base() (QPalette), 243
baseName() (QFileInfo), 84
begin()
  QList, 327
  QTextDocument, 352, 385
beginEditBlock() (QTextCursor), 328, 329
beginInsertColumns() (QAbstractItemModel), 118
beginInsertRows() (QAbstractItemModel), 118, 125, 166, 171, 176
index 501

beginMoveColumns() (QAbstractItemModel), 159
beginMoveRows() (QAbstractItemModel), 159
beginRemoveColumns() (QAbstractItemModel), 118
beginRemoveRows() (QAbstractItemModel), 118, 125, 167, 175
binary file formats, 103, 104
bits() (QImage), 301
block() (QTextCursor), 328
blockCharFormat() (QTextCursor), 328
blockFormat() (QTextCursor), 328, 357
blocking calls, 102
.bmp (files), 364, 375, 388
bool, volatile, 247–248, 307
Boost library, 62
default library, 472
destroyItem() (QRect/QRectF), 215
bouncing, 472
boundedTo() (QSize/QSizeF), 299
boundingRect() (QGraphicsItem), 399, 400, 403, 433, 466
Browser Window example, 22–30
buddies, 444
busy indicator, 296
busy-waiting loop, 74
button; see QAbstractButton and
QIAbstractPushButton
button()
QMessageBox, 137
QPalette, 233, 234
buttonText() (QPalette), 234
cancel() (QFutureWatcher), 272
canceling edits, 200
cap() (QRegExp), 340
ceil() (std), 435, 436
Census Visualizer example, 224–244
center() (QRect/QRectF), 191, 218, 243, 401
changes, unsaved, 94, 99, 136, 137
changing binary file formats, 104
changing trees, policy, 154
character formatting; see QTextCharFormat
charFormat() (QTextCursor), 328
child()
QStandardItem, 141, 146
QTreeWidgetItem, 79
childItems() (QGraphicsItem), 403
chop() (QString), 254, 322
chunkSizes() (AQP), 252, 253, 258, 309
class, template, 304
clear()
QComboBox, 35
QGraphicsScene, 417
QList, 292, 294
QStandardItemModel, 103, 144, 308
QString, 110
clearSelection()
QAbstractItemView, 275
QGraphicsScene, 420, 421, 450
QTextCursor, 328, 334, 336
clickedButton() (QMessageBox), 101, 137
clipboard; see QClipboard
clipboard() (QApplication), 428, 429, 438
close() (QFile), 374
closeEvent() (QWidget), 79, 136, 257, 313
cloud computing, 5
codecs, audio, sound, and video, 53
codecs, text, 374
collidesWithItem() (QGraphicsItem), 403
collidesWithPath() (QGraphicsItem), 403
C++⇔JavaScript communication, 39
caching
graphics items, 464
icons, 11, 17, 19, 443–444
items, 273
pixmaps, 443–444
strings, 108
clear() (QFutureWatcher), 272
collidingItems() (QGraphicsItem), 403
collision detection, 391, 399
color gradient; see QConicalGradient, QLinearGradient, and QRadialGradient
color syntax highlighting; see QSyntaxHighlighter and syntax highlighting
colorNames() (QColor), 442
column() (QModelIndex), 120, 202, 203, 204
columnCount()
    QAbstractItemModel, 99, 118, 123, 163
    QStandardItemModel, 273
columnNumber() (QTextCursor), 328
combobox; see QComboBox
combobox model, 93
compare() (QString), 70, 331
complete() (QCompleter), 327
completion, 320–322, 322–323, 323–336
completionCount() (QCompleter), 327
completionPrefix() (QCompleter), 327, 331
components, 207, 228
computing cloud, 5
conditionals policy, 200
configuration (application); see QSettings
confirming edits, 199, 200
connect() (QObject), 12, 27, 39, 74, 94, 95, 227, 273, 344, 355, 484
connection, signal–signal, 228, 355, 356
connections; see QObject::connect()
constBegin()
    QHash, 306
    QVector, 431
constEnd()
    QHash, 306
    QVector, 431
constructors; using single shot timers, 12
contains()
    QGraphicsItem, 403
    QList, 116
    QRect/QRectF, 216, 427
    QSet, 29, 69, 108
    QString, 27
context menu, 349
    creating, 12, 344
    policy, 26, 29, 344
contextMenuEvent() (QGraphicsItem), 460
contiguous selections, 218
controversy, threading, 246
convenience functions, avoiding, 61
converting
    HTML to plain text, 17, 37
    long double to string, 281
coordinates, 392
    logical, 392
    physical, 392
    system, 392
    viewport, 392
    viewport vs. widget, 214, 216, 220, 221, 222, 224
    window, 392
copying to the clipboard, 428
cosmetic pens, 401, 435
count()
    QComboBox, 35
    QList, 116, 120, 157, 285, 293, 308
createEditor() (QStyledItemDelegate), 194, 199, 203
createIndex() (QAbstractItemModel), 164, 165, 174, 175, 176, 179
createList() (QTextCursor), 328, 367
createPlugin() (QWebPage), 47–48
Cross Fader example, 287–302
cross-thread signals, 300
CSS (Cascading Style Sheet), 317, 395
<stdio>, 281
<stdlib>, 46
Ctrl key, 15, 218, 345
cues, aural and visual, 53
current line highlighting, 320, 336–337

currentBlockState() (QSyntaxHighlighter), 341

currentCharFormat() (QTextEdit), 350

currentCompletion() (QCompleter), 327

currentDate() (QDate), 192

currentFrame() (QTextCursor), 328

currentIndex()
  QAbstractItemView, 136, 138, 140, 218, 222
  QComboBox, 38, 446

currentItem() (QTreeWidget), 75

currentList() (QTextCursor), 328

currentPath() (QDir), 189

currentTable() (QTextCursor), 328

currentText() (QComboBox), 96, 204
cursor, mouse; see QCursor
cursor, text; see QTextCursor
cursorRect() (QPlainTextEdit), 327
custom events, 255–256
custom events vs. slot invocation, 260
custom graphics items, 400, 447–466
custom items vs. QStandardItem, 156
custom models vs. QStandardItemModel, 112
custom property interpolation, 489–490
custom shapes, 400
cutting to the clipboard, 428

data stream version policy, 104
dataChanged()  
  QAbstractItemModel, 118, 124, 166, 174, 178, 179  
  QAbstractItemView, 210, 216
date/times, ISO 8601, 143, 188
deadlock, 268
delete, 69, 93, 160, 175, 181, 195, 293, 308
eliminating, 62
  see also qDeleteAll()
deleteChar() (QTextCursor), 328
deleteLater() (QObject), 292, 294
deletePreviousChar() (QTextCursor), 328
deleting
  dialogs, 101
    model rows, 99–100
desktop() (QApplication), 377
desktop applications vs. web applications, 5–6
devices, media, 63
dialogs
  animated, 481–490
  deleting, 101
  modal, 61
  modeless, 102
  window-modal, 356
  see also QDialog
directories, iterating; see QDir-Iterator
directories, standard; see QDir
DirectShow library, 60, 63
DirectX library, 60
disconnect() (QObject), 55
dock windows; see QDockWidget
document; see QTextDocument
document()
  QPlainTextEdit, 324, 335
  QTextCursor, 328
document-centric, 131
documentLayout() (QTextDocument), 384
documents
  inserting images into, 371
documents (cont.)
  inserting tables into, 369–371
  selecting in, 326, 330, 332, 336
DOM (Document Object Model),
  XML, 18
downloading, 13
downloads, synchronous, 43
drag and drop, 168–173
drawBackground()
  QGraphicsScene, 435
  QGraphicsView, 417
drawChord() (QPainter), 465
drawContents() (QTextDocument), 375, 376, 384
drawEllipse() (QPainter), 192, 465
drawImage() (QPainter), 382
drawing; see QPainter and the graphics/view classes
drawLine() (QPainter), 192, 222, 235, 386
drawPath() (QPainter), 401, 463, 466
drawPixmap() (QPainter), 197
drawRect() (QPainter), 222, 382
drawRoundedRect() (QPainter), 465
drawText() (QPainter), 193, 202, 380, 382, 386, 401
dropMimeData() (QAbstractItemModel), 159, 171
dynamic_cast<>(), 232, 349, 423, 424–425, 434, 439, 455, 471
  avoiding, 424–425
  vs. qobject_cast<>(), 349

E

easing curves; see QEasingCurve
edit() (QAbstractItemView), 99, 139, 153
edit, cancel, 200
edit confirmation, 199, 200
editing, in-place, 99, 121, 153, 194
editor, Scintilla, 323
editor widget; see QLineEdit, QPlainTextEdit, and QTextEdit

efficiency vs. accuracy, 310
elidedText() (QFontMetrics), 29, 30
eliminating delete, 62
emit, 238, 239, 300, 301, 302, 315, 349, 457, 485
encoding (files), 145, 147, 374
end()
  QList, 327
  QPainter, 421
endEditBlock() (QTextCursor), 329
endInsertColumns() (QAbstractItemModel), 118
endInsertRows() (QAbstractItemModel), 118, 125, 166, 171, 176
endMoveColumns() (QAbstractItemModel), 159
endMoveRows() (QAbstractItemModel), 159
endRemoveColumns() (QAbstractItemModel), 118
endRemoveRows() (QAbstractItemModel), 118, 125, 167, 175
endsWith() (QString), 13, 41, 322
ensureVisible()
  QGraphicsItem, 403
  QScrollArea, 230
Enter key, 153, 199, 200
Error (AQP), 104, 105, 126, 144
errostring() (QFile), 144, 374
Esc key, 30, 82, 200
escape() (Qt), 352, 366
escaping HTML and XML, 145, 147, 352, 353
evaluate() (QScriptEngine), 283
evaluateJavaScript() (QWebFrame), 41
event() (QWidget), 256
event filter, 82–83, 228, 237–238
event loop, 74, 83, 134, 259, 300
event processing, 102, 255
eventFilter() (QObject), 83, 237–238
events, custom, 255–256
events, custom vs. slot invocation, 260
events, posting, 255–256
examples
  Browser Window, 22–30
  Census Visualizer, 224–244
  Cross Fader, 287–302
  Find Dialog, 481–490
  Find Duplicates, 302–316, 320–322
  Folder View, 188–193
  Image2Image, 249–260, 320
  Matrix Quiz, 44–52
  Movie Jingle, 54–60
  New York Review of Books Viewer, 31–43
  Number Grid, 186–188, 263–286
  Output Sampler, 361–388
  Page Designer, 409–467, 470–474
  Petri Dish, 392–402, 475–480
  Play Music, 64–79
  Play Video, 80–85
  RssPanel, 20
  TextEdit, 353–358
  TiledListView, 208–224
  Timelog (custom model), 151–183, 193–200, 343–353
  Timelog (QStandardItemModel), 130–150, 193–200, 343–353
  Weather Tray Icon, 7–20
  XmlEdit, 322–342
  Zipcodes (custom model), 113–128, 201–205
  Zipcodes (QStandardItemModel), 90–107, 201–205
exception (std), 105
exclusive actions, 13
exec()
  QApplication, 8
  QDialog, 61
  QDialog, problems with, 102
executing JavaScript, 41
exists() (QFile), 33, 76
expand()
  QAbstractItemView, 189
  QTreeWidget, 312
expandedTo() (QSize/QSizeF), 299
exporting
  .html files, 374
  .odt files, 373
  .pdf and .ps files, 372
  pixmap files, 375, 423
  scenes, 420–423
  .svg files, 375, 422
extension; see files

F
F2 key, 153
file quality (QTextDocument), 361–364
fileName() (QFileInfo), 312
filenameFilter() (AQP), 58, 84
files
  binary formats, 103
  .bmp, 364, 375, 388
  changing binary formats, 104
downloading, 13
encoding, 145, 147, 374
exporting, 372, 373, 374, 375, 422, 423
formats, 58
  .gif, 53
  .html, 363, 374
  .jpg and .jpeg, 375, 388
locating, 33
  .mng, 53
  .ods, 363
  .odt, 362, 363, 373
  .oga and .ogg, 54
  .pdf and .ps, 362, 372, 387
  pixmap format, 364
  .png, 364, 375, 388
  .ppm, 375
  printing, 387, 388
  .pro, 114
  .qm, 276
suffixes, 58, 60, 69, 71, 72
  .svg, 362, 363, 375, 382, 387, 422
  .tiff, 375, 388
  .wav, 53
files (cont.)
   .xpm, 375
fill() (QPixmap), 347
fillRect() (QPainter), 191, 202, 376, 388
filterAcceptsRow() (QSortFilterProxyModel), 108, 111
filtered() (QtConcurrent), 270, 271
filteredReduced() (QtConcurrent), 277, 278–279
filtering models; see QSortFilterProxyModel
filters, 261–262, 270–281
findFirst() (QPixmapCache), 197, 443
Find Dialog example, 481–490
Find Duplicates example, 302–316, 320–322
findChildren() (QObject), 32
findData() (QComboBox), 445
findText() (QComboBox), 100, 204
firewalls, 9
first() (QList), 309, 312
flags() (QAbstractItemModel), 118, 120, 161
floating-point format, 104
floating-point numbers, summing, 280
flyweight items, 112
focus policy, 50, 236
focusInEvent() (QWidget), 24
Folder View example, 188–193
folders, iterating; see QDirIterator
font()
   QApplication, 211
   QTextCharFormat, 357
   QWidget, 218, 486
fontItalic()
   QTextCharFormat, 352, 357
   QTextEdit, 350
fontMetrics() (QPainter), 380, 382, 385, 386
fontPointSize() (QTextCharFormat), 357
fontStrikeOut() (QTextCharFormat), 348, 350, 352
fontWeight()
   QTextCharFormat, 352, 357
   QTextEdit, 344, 350
foreach, 94, 106, 252, 292, 294, 297, 309, 310, 347, 397, 442
foreground() (QTextCharFormat), 352
forever, 311
format properties, 336, 371
formats (files), 58
formatting, text; see QTextBlockFormat and QTextCharFormat
fromHtml() (QTextDocumentFragment), 17, 37
fromLocalFile() (QUrl), 45, 297
fromSet() (QList), 267
fromString() (QDateTime), 147, 173
fromUserInput() (QUrl), 297
fromUtf8() (QString), 41, 105
full-screen mode, 83
function, template, 138
functional programming, 261
functionctors, 252, 271, 274–275, 278
   see also operator()
future() (QFutureWatcher), 275, 285

G

geometry() (QWidget), 427
get() (QNetworkAccessManager), 13
get from operator; see operator<<(,)
getExistingDirectory() (QFileDialog), 68
getOpenFileName() (QFileDialog), 58, 83
getAddressName() (QFileDialog), 60
.gif (files), 53
globalInstance() (QThreadPool), 251, 258
globalObject() (QScriptEngine), 283
grabWidget() (QPixmap), 197
gradient; see QConicalGradient, QLinearGradient, and QRadialGradient
graphics item coordinates, 392
graphics items, caching, 464
graphicsEffect() (QWidget), 486, 487, 488
GraphicsItemFlag enum (QGraphicsItem), 406–407
graphics/view classes; see QGraphicsItem, QGraphicsScene, and QGraphicsView
group() (QGraphicsItem), 403, 416
GStreamer library, 60
guarded pointers, 62, 290
see also QPointer and QWeakPointer
GUI blocking, 102

H
hasComplexSelection() (QTextCursor), 329
hasFormat() (QMimeData), 429, 438
hash, thread-safe, 304
hasHtml() (QMimeData), 429, 438
hasSelection() (QTextCursor), 329
hasText() (QMimeData), 429, 438
hasUncaughtException()
(QScriptEngine), 283
headerData() (QAbstractItemModel), 118, 120, 121, 163
headerDataChanged() (QAbstractItemModel), 118, 125
height()
QFontMetrics, 121, 198, 212
QImage, 300, 301
QRect/QRectF, 214, 382
hide()
QGraphicsItem, 403
QWidget, 275, 280, 308, 313
hiding rows, 141
higher-order functions, 261
highlight() (QPalette), 234, 243
highlightBlock() (QSyntaxHighlighter), 337, 339
highlightedText() (QPalette), 234, 242
highlighting, current line, 320
highlighting, syntax; see current line highlighting, QSyntaxHighlighter, and syntax highlighting
home() (QDir), 189
homePath() (QDir), 189, 321
horizontal padding, 227
horizontalHeader() (QTableView), 95
horizontalHeaderItem() (QStandardItemModel), 144
horizontalOffset() (QAbstractItemView), 210, 219
horizontalScrollBar() (QAbstractScrollBar), 211, 223
hoursMinutesSecondsForMSec() (AQP), 74
HTML, 363
converting to plain text, 17, 37
escaping, 352, 353
exporting, 374
rendering, 30
text width, 198
see also .html, QPlainTextEdit, QTextDocument, and QTextEdit
.html (files), 363, 374
html() (QMimeData), 429
HTTP redirection, 19

I
icon() (QAction), 431
icons (caching), 11, 17, 19, 443–444
idealThreadCount() (QThread), 252, 253, 258, 309
idealWidth() (QTextDocument), 198, 350
Image2Image example, 249–260, 320
images
in documents, 319, 371
storing, 60
in-place editing, 99, 121, 153, 194
index()
QAbstractItemModel, 108, 111, 118, 153, 154, 164, 212
index() (cont.)
  QModelIndex, 189
  QStandardItem, 134, 139
  QStandardItemModel, 279
index, model; see QModelIndex
indexAt() (QAbstractItemView), 210, 216
indexOf() (QList), 116, 157,
  QStringList, 341
indexOfProperty() (QMetaObject), 419, 434, 439
information() (AQP), 61, 280
insert()
  QCache, 19
  QList, 125, 157
  QMultiHash, 305, 339
  QPixmapCache, 197, 443
insertBlock() (QTextCursor), 329, 368, 369, 371
insertColumns() (QAbstractItemModel), 116, 118, 123
insertFrame() (QTextCursor), 329, 369
insertHtml() (QTextCursor), 329
insertImage() (QTextCursor), 329, 371
insertList() (QTextCursor), 329, 367
insertRow() (QAbstractItemModel), 115, 125, 153, 167
insertRows() (QAbstractItemModel), 115, 116, 118, 123, 125, 166
insertTable() (QTextCursor), 329, 369
insertText() (QTextCursor), 329, 331, 368, 369, 371
installEventFilter() (QObject), 82, 227
internalPointer() (QModelIndex), 162
international numbers, 98
internationalization; see QLocale
Internet-aware widgets, 6
Internet downloading, 13
interpolation; see property animation
interpretText() (QAbstractSpinBox), 204
intersects() (QRect/QRectF), 220
intProperty() (QTextCharFormat), 371
invalidateFilter() (QSortFilterProxyModel), 110
invisibleRootItem()
  QStandardItemModel, 141, 144, 146, 147, 312
  QTreeWidget, 73
invokeMethod() (QMetaObject), 133–134, 259–260, 423
isCanceled() (QFutureWatcher), 275, 285
isCopyOf() (QTextCursor), 329
isDir() (QFileInfo), 309
isEmpty()
  QList, 157, 308, 428
  QString, 17, 326
  QStringList, 352
isFile() (QFileInfo), 69, 315
isFinished() (QThread), 297
isIndexHidden() (QAbstractItemView), 210, 214
isLetter() (QString), 326
isNull()
  QIcon, 17
  QPixMap, 59
  QPoint/QPointF, 427
  QTextCursor, 329
ISO 8601 date/times, 143, 188
isObscuredBy() (QGraphicsItem), 403
isRunning()
  QFutureWatcher, 272
  QThread, 310
isSelected() (QGraphicsItem), 403, 454, 457, 463
isSymLink() (QFileInfo), 309, 315
isValid()
  QModelIndex, 100, 120, 139, 154, 162
  QVariant, 434
isVisible() (QGraphicsItem), 403
long double, converting to string, 281

M

Mac OS X
- platform differences, 26, 61, 211, 378
- sheets, 61

magic numbers, 72, 106

main window; see QMainWindow

main window startup policy, 133

mainFrame() (QWebPage), 39, 41

mapFromGlobal() (QWidget), 427

mapped() (QtConcurrent), 281, 282

mappers, 262, 281-286

mapToGlobal() (QWidget), 349

mapToScene() (QGraphicsView), 427

matchedLength() (QRegExp), 340

Matrix Quiz example, 44-52

max_element() (std), 431, 432

MD5 (Message-Digest algorithm 5), 302

media data sources, devices, and nodes, 63
- see also audio and video

media path, 64

menus
- creating context, 12, 344, 349
- creating popup, 29

mergeBlockCharFormat() (QTextCursor), 329

mergeBlockFormat() (QTextCursor), 329, 368

mergeCharFormat() (QTextCursor), 330, 368

mergeCurrentCharFormat() (QTextEditor), 348

meta-data, audio, 71

metaObject() (QObject), 419, 423, 439

method invocation, 134

method pointer, 137

mid()
- QList, 150, 183, 252, 253, 258, 285, 309

MIME types, 71, 72, 159, 169, 428
- see alsoQMimeData

mimeData()
- QAbstractItemModel, 159, 169
- QClipboard, 429, 438

mimeTypes() (QAbstractItemModel), 159, 169

min_element() (std), 431, 432

minimumSizeHint() (QWidget), 232, 236, 350, 441, 486, 487

.mng (files), 53

modal (window-modal) dialogs, 356

modal dialogs, 61

model index; see QModelIndex

modeless dialogs, 102

models
- class hierarchy, 89
- creation policy, 112
- custom vs. QStandardItemModel, 112
- deleting rows, 99-100
- filtering, 99
- for combobox, 93
- for views, 95
- kinds of, 89

mapping selection model indexes, 100

non-string data, 99, 106, 186

proxy, 92, 93

proxy source model, 94

Resizable, 116, 123, 125, 158

selecting items in, 97

sorting, 111, 312

- see alsoQAbstractItemModel,
- QItemSelectionModel, and QSortFilterProxyModel

ModelTest, 114-115, 135

model/view classes; see QAbstractItemModel and QAbstractItemView

modules
- Phonon, 60, 72, 85
- QtCore, 20, 348
modules (cont.)

QtGui, 348
QtMultimedia, 86, 447
QtNetwork, 6
QtWebKit, 6, 21
QtXml, 16, 20

monitors, 268
mouse cursor; see QCursor
mouse wheel, 236
mouseButtons() (QApplication), 472
mouseDoubleClickEvent() (QGraphicsItem), 403, 449, 460
mouseMoveEvent() (QGraphicsItem), 404, 458
mousePressEvent() (QWidget), 238, 335
moveBy() (QGraphicsItem), 404, 418, 423
moveCursor() (QAbstractItemView), 210, 218
MoveOperation enum (QTextCursor), 334
movePosition() (QTextCursor), 330, 331, 332
moveTo() (QRect/QRectF), 382
Movie Jingle example, 54–60
movies, location of, 83
moving items in views, 158–159, 173–180
music files, location of, 68
music player; see Play Music example
mutex; see QMutex
mutexes vs. read-write locks, 268

N

name() (QColor), 443
network proxying, 9
New York Review of Books Viewer example, 31–43

nodes, media, 63
non-contiguous selections, 219
non-resizable models, 123
non-string model data, 99, 106, 186
normalized() (QRect/QRectF), 220
number() (QString), 50, 121, 401, 402
Number Grid example, 186–188, 263–286
numbers, localized, 98
numbers, rounding up, 229
see also qRound()
numeric_limits<>::max() (std), 186
numeric_limits<>::min() (std), 436

O

object() (QCache), 17
.ods (files), 363
.odt (files), 362, 363, 373
off-screen rendering, 464
.oga and .ogg (files), 54
okToClearData() (AQP), 61, 136–138
okToDelete() (AQP), 61, 100, 101, 140
opacity; see transparency
opaqueArea() (QGraphicsItem), 404
open()

QDialog, 102
QFile, 41, 104, 126, 315, 374
Open Document Format; see .odt
openUrl() (QDesktopServices), 297
operator()(), 271, 274–275, 283
operator*(), 432
operator++() (QPoint/QPointF), 427
operator--() (QPoint/QPointF), 427
operator<(), 330
for QStringList items, 116, 117
QTreeWidgetItem, 70
operator<() (QDataStream), 126, 416, 452
QList, 157, 203, 295, 309, 345, 395, 397, 442, 483, 486
QSet, 108, 268, 418
QTextStream, 374
QVector, 370
Index

operator==( )
  for QList items, 116, 117
operator>>( ) (QDataStream), 127, 418, 453
operator[]()  
  QHash, 212, 396, 397
  QList, 124
  QMap, 295, 296
ordering trees, 70
outline; see QPainterPath
Output Sampler example, 361–388

P

padding, horizontal, 227
page break, inserting, 368
Page Designer example, 409–467, 470–474
page numbers, 362
paint()  
  QGraphicsItem, 399, 401–402, 404, 463
  QStyledItemDelegate, 191, 194, 196, 202
paintEvent()  
  QAbstractScrollArea, 210, 222
  QWidget, 233, 241
painting
  only what’s needed, 241
  .pdf and .ps files, 387
  pixmaps, 388
  rich text, 383–386
  .svg files, 387
  widgets, 195–198
  see also QPainter and the graphics/view classes
palette; see QPalette
palette() (QWidget), 222, 233, 234, 242, 243, 336
paragraph; see QTextBlock
paragraph formatting; see QTextBlockFormat
parameterizing slot invocation, 13
parent() (cont.)  
  QModelIndex, 100, 140, 141
  QObject, 236
parentItem() (QGraphicsItem), 404
pasting from the clipboard, 429
path, media, 64
path, painter; see QPainterPath
path separators, 57
  .pdf (files), 362, 372, 387
pens, cosmetic, 401, 435
Petri Dish example, 392–402, 475–480
Phonon classes, 63
Phonon module, 60, 72, 85
physical coordinates, 392
pixel() (QImage), 300
pixmap files, 375, 423
pixmap formats, 364
pixmaps (caching), 443–444
platform differences, 359
  Mac OS X, 26, 61, 211, 378
  Windows, 254
Play Music example, 64–79
Play Video example, 80–85
playing a sound, 55
  .png (files), 364, 375, 388
pointer, to method, 137
pointers, guarded and smart, 62, 290
policies
  animation, 53
  antialiasing, 376, 379, 423
  application settings, 20
  assertion, 200
  audio, 53, 55
  conditionals, 200
  context menu, 26, 29, 344
  data stream version, 104
  focus, 50, 236
  model creation, 112
  QRegExp capturing, 339
  sound, 53, 55
  startup, 133
trees, changing, 154
polling, 256–257, 260
popup() (QCompleter), 327, 333
popup menu, creating, 29
Portable Document Format; see .pdf
pos()
   QCursot, 29, 427, 462
   QGraphicsItem, 404, 452
   QWidget, 427
position() (QTextCursor), 330, 331, 332
postEvent() (QApplication), 254, 255
posting events, 255–256
PostScript; see .ps
pow() (std), 439, 440
   .ppm (files), 375
predefined actions, QWebView, 26
preparingGeometryChange() (QGraphics-
   Item), 404, 461
prepend()
   QList, 150, 182
   QString, 27, 385
preview, print, 377–378
previousBlockState() (QSyntaxHigh-
   lighter), 337, 341
print() (QTextDocument), 373, 378
print preview, 377–378
printer setup, 377
printing
   .pdf and .ps files, 387
   pixmaps, 388
   scenes, 420–423
   .svg files, 387
   .pro (files), 114
progressMaximum() (QFutureWatcher), 273
progressMinimum() (QFutureWatcher), 273
properties, of formats, 336, 371
property() (QObject), 434, 455
property animation, 470
property interpolation, custom, 489–490
property system, 419, 449, 470, 474
proxy model; see QSortFilterProxy-
   Model
proxy server, 9
   .ps (files), 362, 372, 387
put to operator; see operator>>()

Q

Q_ARG, 259
Q_ASSERT, 76, 114, 121, 158, 161, 164, 174,
   175, 199, 200, 204
Q_PROPERTY, 449, 456, 472, 477
Q_WS_MAC, 26
Q_WS_X11, 17
qAbs(), 46
QAbstractButton, 101
   setChecked(), 196
   setText(), 291
text(), 251, 291
see also QCheckBox, QPushButton, and
   QRadioButton
QAbstractItemModel, 117, 158, 160
   beginInsertColumns(), 118
   beginInsertRows(), 118, 125, 166,
   171, 176
   beginMoveColumns(), 159
   beginMoveRows(), 159
   beginRemoveColumns(), 118
   beginRemoveRows(), 118, 125, 167,
   175
class hierarchy, 89
columnCount(), 99, 118, 123, 163
createIndex(), 164, 165, 174, 175,
   176, 178, 179
data(), 108, 111, 118, 120–122, 161,
   196, 322
dataChanged(), 118, 124, 166, 174,
   178, 179
dropMimeData(), 159, 171
endInsertColumns(), 118
endInsertRows(), 118, 125, 166, 171,
   176
endMoveColumns(), 159
endMoveRows(), 159
endRemoveColumns(), 118
endRemoveRows(), 118, 125, 167, 175
QAbstractItemModel (cont.)
- flags(), 118, 120, 161
- headerData(), 118, 120, 121, 163
- headerDataChanged(), 118, 125
- index(), 108, 111, 118, 153, 154, 164, 212
- insertColumns(), 116, 118, 123
- insertRow(), 115, 125, 153, 167
- insertRows(), 115, 116, 118, 123, 125, 166
- mimeData(), 159, 169
- mimeTypes(), 159, 169
- parent(), 118, 165
- removeColumns(), 116, 118, 123
- removeRow(), 100, 126, 140
- removeRows(), 116, 118, 123, 125, 167
- reset(), 181
- rowCount(), 115, 118, 123, 154, 163, 212
- setData(), 118, 124, 166, 204
- setHeaderData(), 118, 125, 165
- sort(), 442
- supportedDragActions(), 159, 168
- supportedDropActions(), 159, 168
- see also QAbstractListModel and QAbstractTableModel

QAbstractItemView, 207, 208, 209, 211
- clearSelection(), 275
- currentIndex(), 136, 138, 140, 218, 222
- dataChanged(), 210, 216
- edit(), 99, 139, 153
- expand(), 189
- horizontalOffset(), 210, 219
- indexAt(), 210, 216
- isIndexHidden(), 210, 214
- itemDelegate(), 222
- mousePressEvent(), 210, 223
- moveCursor(), 210, 218
- resizeEvent(), 210, 223
- rootIndex(), 212, 213, 216, 221
- rowsAboutToBeRemoved(), 210, 217
- rowsInserted(), 210, 217
- scrollContentsBy(), 210, 219
- scrollDirtyRegion(), 219

QAbstractItemView (cont.)
- scroll(), 97, 134, 189, 210, 215
- scrollTo(), 134, 189, 210, 215
- scrollToBottom(), 99
- selectionModel(), 95, 100, 136, 220, 222, 275, 279
- setColumnHidden(), 189
- setCurrentIndex(), 134, 189, 223
- setCurrentItem(), 99
- setDragDropMode(), 152
- setEditTriggers(), 271, 275
- setItemDelegateForColumn(), 135, 152, 189
- setModel(), 135, 152, 189, 209, 210, 211
- setSelection(), 210, 220
- updateGeometries(), 210, 223
- verticalOffset(), 210, 219
- viewOptions(), 222
- visualRect(), 210, 213, 221
- visualRegionForSelection(), 210, 221
- see also QColumnView, QHeaderView, QListView, QTableView, and QTreeView

QAbstractListModel, 123
- see also QAbstractItemModel

QAbstractScrollBarArea, 223, 228
- horizontalScrollBar(), 211, 223
- paintEvent(), 210, 222
- setHorizontalScrollBarPolicy(), 344
- setVerticalScrollBarPolicy(), 344
- verticalScrollBar(), 211, 223, 230, 312
- viewport(), 199, 209, 212, 215, 219, 222
- see also QGraphicsView, QPlainTextEdit, QPlainTextEdit, QScrollArea, and QTextEdit

QAbstractSocket, 9

QAbstractSpinBox
- interpretText(), 204
- setAlignment(), 186, 205
Index 515

QAbstractSpinBox (cont.)
   see also QDoubleSpinBox and QSpinBox
QAbstractTableModel, 116, 117
   see also QAbstractItemModel
QAction, 12, 30, 55, 346, 348, 354, 426
data(), 29, 348, 426, 431
icon(), 431
setCheckable(), 12, 346, 463
setChecked(), 12, 346, 350, 357, 463
setData(), 12, 29, 346, 347, 431, 463
setIcon(), 59, 357, 431
setMenu(), 345, 430
setShortcuts(), 26, 57
setText(), 59
QActionGroup, 12
   addAction(), 12
   triggered(), 13
qAlpha(), 300
QApplication, 8
   applicationDirPath(), 33
   applicationName(), 32, 56
   clipboard(), 428, 429, 438
desktop(), 377
exec(), 8
font(), 211
keyboardModifiers(), 471
layoutDirection(), 224
mouseButtons(), 472
postEvent(), 254, 255
restoreOverrideCursor(), 69
sendEvent(), 255
setApplicationName(), 8, 412
setAttribute(), 345
setOrganizationDomain(), 8
setOrganizationName(), 8
setOverrideCursor(), 69
setQuitOnLastWindowClosed(), 8
style(), 121
qBlue(), 300, 301
qBound(), 46, 490
QBrush, 441, 445, 466
QByteArray, 14, 41, 58, 169, 171, 307, 315, 428, 429
readAll(), 19, 41
QCache, 10–11, 17
   insert(), 19
   object(), 17
QChar, 229, 295
QCheckBox, 194, 481, 483
   setChecked(), 483
   see also QAbstractButton
QClipboard, 427–430, 438
   mimeData(), 429, 438
   setMimeData(), 428
QCloseEvent, 257, 313
QColor, 192, 234, 244, 347, 352, 435, 442, 470
   colorNames(), 442
   darker(), 233, 234, 244, 465
   lighter(), 234, 244, 466
   name(), 443
QColorDialog, 354, 356
QColumnView, 88, 185
   see also QAbstractItemView
QComboBox, 88, 185, 203, 322, 442, 481, 483
   addItem(), 35, 442, 483
   addItems(), 203
   clear(), 35
   count(), 35
   currentIndex(), 38, 446
   currentText(), 96, 204
   findData(), 445
   findText(), 100, 204
   itemData(), 36, 445, 446
   currentIndex(), 35, 100, 204, 445, 483
   setItemIcon(), 446
   setModel(), 93
   setModelColumn(), 93, 94
QCompleter, 320, 321, 322, 324, 331
   complete(), 327
   completionCount(), 327
   completionPrefix(), 327, 331
   currentCompletion(), 327
   popup(), 327, 333
   setCaseSensitivity(), 321, 324, 325
   setCompletionMode(), 324
QCompleter (cont.)
  setCompletionPrefix(), 327
  setModel(), 321, 324
  setModelSorting(), 324, 325, 327
  setWrapAround(), 324, 325
qCompress(), 169
QConicalGradient, 244
QCryptographicHash, 315
QCursor::pos(), 29, 427, 462
QDataStream, 104, 106, 126, 127, 416,
  418, 428, 453
  operator<((), 126, 416, 452
  operator>()(), 127, 418, 453
  setVersion(), 104, 106, 127
QDate
  currentDate(), 192
QDateTime, 142, 146, 156, 170
  fromString(), 147, 173
  toString(), 146, 170, 193
qDebug(), 43, 46
qDeleteAll(), 157
QDesktopServices
  openUrl(), 297
  storageLocation(), 68, 83
QDesktopWidget
  availableGeometry(), 377
  screenCount(), 377
  screenGeometry(), 377
QDialog
  exec(), 61
  exec(), problems with, 102
  open(), 102
  show(), 102
QDir
  currentPath(), 189
  home(), 189
  homePath(), 189, 321
  rootPath(), 189
  separator(), 254, 322
  tempPath(), 189
  toNativeSeparators(), 254, 312,
  321, 322
QDirIterator, 69, 251, 309, 315
QDirModel, 321, 322
QDockWidget, 414–415
QDomDocument, 14, 16
QDoubleSpinBox, 186, 354
  setDecimals(), 186
  setRange(), 186
  see also QAbstractSpinBox and
  QSpinBox
QEasingCurve, 470
QEvent, 83, 255, 256
QEventLoop, 74, 83
  exec(), 74, 83
QFile, 41, 104, 126, 315, 374
  close(), 374
  errorString(), 144, 374
  exists(), 33, 76
  open(), 41, 104, 126, 315, 374
  readAll(), 315
QFileDialog
  getExistingDirectory(), 68
  getOpenFileName(), 58, 83
  getSaveFileName(), 60
QFileInfo, 312
  absolutePath(), 58, 60
  baseName(), 84
  fileName(), 312
  isDir(), 309
  isFile(), 69, 315
  isSymLink(), 309, 315
  size(), 312, 315
  suffix(), 69, 251, 254
QFileSystemModel, 189, 321
QFileSystemWatcher, 190
QFinalState, 474, 477, 479
QFont, 380, 386, 487, 490
QFontComboBox, 354
QFontMetrics, 29, 121, 212, 227, 350
  elidedText(), 29, 30
  height(), 121, 198, 212
  lineSpacing(), 380, 385
  width(), 121, 198, 212, 227, 385, 386
QFormLayout, 444
QFrame, 24
  setFrameStyle(), 25
QFuture, 265–266, 269, 271, 272, 278,
  282
QFutureWatcher, 265–266, 269
QFutureWatcher (cont.)
cancel(), 272
future(), 275, 285
isCanceled(), 275, 285
isRunning(), 272
progressMaximum(), 273
progressMinimum(), 273
result(), 280
setFuture(), 271, 278, 282
waitForFinished(), 272
qFuzzyCompare(), 46, 274, 275, 454
QGraphicsAbstractShapeItem
setBrush(), 398
setPen(), 398
see also QGraphicsItem
QGraphicsAnchorLayout, 397
QGraphicsBlurEffect, 483
QGraphicsColorizeEffect, 483
QGraphicsDropShadowEffect, 483
QGraphicsEffect, 405
QGraphicsEllipseItem, 398
see also QGraphicsAbstractShapeItem
QGraphicsGridLayout, 397
QGraphicsItem, 390, 393, 399–407, 416, 418, 420, 436, 472, 483
advance(), 403
boundingRect(), 399, 400, 403, 433, 466
childItems(), 403
class hierarchy, 391
collidesWithItem(), 403
collidesWithPath(), 403
collidingItems(), 403
contains(), 403
closeMenuEvent(), 460
closeCurrentVisible(), 403
GraphicsItemFlag enum, 406–407
group(), 403, 416
hide(), 403
isObscuredBy(), 403
isSelected(), 403, 454, 457, 463
isVisible(), 403
itemChange(), 393, 407, 450, 451
keyPressEvent(), 403
QGraphicsItem (cont.)
mouseDoubleClickEvent(), 403, 449, 460
mouseMoveEvent(), 404, 458
mousePressEvent(), 404, 458
mouseReleaseEvent(), 459
moveBy(), 404, 418, 432
opaqueArea(), 404
paint(), 399, 401–402, 404, 463
parentItem(), 404
pos(), 404, 452
prepareGeometryChange(), 404, 461
resetTransform(), 404
rotation(), 404
scale(), 404
scene(), 404
sceneBoundingRect(), 404, 433
scenePos(), 404
setCacheMode(), 464
setCursor(), 458, 459
setFlag(), 391, 404
setFlags(), 398, 404, 450
setFont(), 450
setGraphicsEffect(), 405
setGroup(), 405
setParentItem(), 405
setPen(), 435
setPos(), 405, 450, 453
setRotation(), 393, 405, 451
setScale(), 393, 405, 451
setSelected(), 405, 420, 421, 450
setToolTip(), 405
setTransform(), 393, 405, 451, 454
setTransformOriginPoint(), 393, 451
setVisible(), 405, 435
setX(), 405
setY(), 405
setZValue(), 405, 435, 453
shape(), 400, 405, 466
show(), 405
toGraphicsObject(), 424, 425
toolTip(), 405
transform(), 405
type(), 400, 405, 416, 425, 449, 456
QGraphicsItem (cont.)
update(), 406, 461
vs. QGraphicsObject, 390
x(), 406
y(), 406
zValue(), 406, 452
see also QGraphicsAbstractShapeItem
qgraphicsitem_cast<>(), 401, 424, 449
QGraphicsItemGroup, 391, 403, 405, 417, 435
QGraphicsLayoutItem, 397
QGraphicsLinearLayout, 397
QGraphicsLineItem, 435
QGraphicsObject, 390, 424, 447, 459–466
vs. QGraphicsItem, 390
QGraphicsOpacityEffect, 483, 488
QGraphicsPathItem, 390
QGraphicsProxyWidget, 390, 393, 396, 447
QGraphicsRectItem, 455–459
QGraphicsScene, 390, 394, 413, 467, 477
addItem(), 397, 398, 435, 450
addWidget(), 396, 397
clear(), 417
clearSelection(), 420, 421, 450
drawBackground(), 435
items(), 415, 438
layouts, 396–398
removeItem(), 428, 429
render(), 421–422
selectedItems(), 421, 423, 428, 431, 434, 438, 439, 455
setBackgroundColorBrush(), 435
setItemIndexMethod(), 394, 477
setSceneRect(), 398, 413
views(), 450, 451
width(), 435
QGraphicsSceneMouseEvent, 458, 459
QGraphicsSimpleTextItem, 390
QGraphicsTextItem, 390, 447–455
QGraphicsVideoItem, 447
QGraphicsView, 390, 393, 398, 439–440, 467
QGraphicsView (cont.)
drawBackground(), 417
mapToScene(), 427
scale(), 439, 440
setBackgroundColorBrush(), 398, 417
setDragMode(), 439
setRenderHints(), 398, 439
setScene(), 413
see also QAbstractScrollArea
QGraphicsWebView, 390, 447
QGraphicsWidget, 390, 393, 397, 399, 447
qGray(), 300
qGreen(), 300, 301
QGridLayout, 295
QHash, 69, 209, 396
constBegin(), 306
constEnd(), 306
operator[](), 212, 396, 397
value(), 73, 214
values(), 305, 306
vs. QMultiHash, 304
see also QMultiHash
qHash(), 11
QHashIterator, 216, 220, 340
QHeaderView, 49
see also QAbstractItemView
QIcon, 17
isNull(), 17
QImage, 254, 291, 300, 376, 382, 388, 423
bits(), 301
height(), 300, 301
pixel(), 300
save(), 254, 294, 302, 376, 388, 423
scaled(), 299
scanLine(), 300
setPixel(), 300
size(), 299
width(), 300, 301
QImageReader, 376
QImageWriter, 375, 388
QInputDialog::getText(), 28
QIODevice, 14, 63
QItemDelegate, 185
QItemEditorCreatorBase, 186
QItemEditorFactory, 186
QItemSelection, 97, 220
QItemSelectionModel, 95, 96, 97, 100, 275, 279
QItemSelectionRange, 221
QKeyEvent, 239, 333, 334, 349, 457
QKeyEvent, 26, 57, 82, 325, 345
QLabel, 194, 290, 295, 395, 481, 483
setBuddy(), 483
setText(), 196
setTextFormat(), 194
setWordWrap(), 194
QLinearGradient, 234, 244
QLineEdit, 321, 322
setCompleter(), 321
QLine/QLineF, 470
QLinkedList, 116
append(), 285
at(), 120, 150, 183, 432, 471
begin(), 327
clear(), 292, 294
contains(), 116
count(), 116, 120, 157, 285, 293, 308
del(), 327
first(), 309, 312
fromSet(), 267
indexOf(), 116, 157
insert(), 125, 157
isEmpty(), 157, 308, 428
items needing operator<(), 116, 117
items needing operator==(), 116, 117
lastIndexOf(), 116
mid(), 150, 183, 252, 253, 258, 285, 309
operator<<(), 157, 203, 295, 309, 345, 395, 397, 442, 483, 486
operator[]( ), 124
prepend(), 150, 182
removeAll(), 116, 327
QList (cont.)
removeAt(), 125
removeOne(), 116
swap(), 157
takeAt(), 158
takeFirst(), 352, 486
value(), 157
QListIterator, 29, 126, 146, 170, 275, 285, 428, 433, 455
QListView, 88, 185
see also QAbstractItemView
QListWidget, 88
QLocale, 229, 243
toDouble(), 229
toFloat(), 229
toInt(), 229, 243
toUInt(), 229
.qm (files), 276
QMainWindow, 66, 81, 91, 93, 114, 132, 394, 413, 477
addDockWidget(), 414
setCentralWidget(), 135, 152, 398, 413
setDockOptions(), 414
setDockOptions(), 414
qmak, 114
qMakePair(), 315, 347, 442
QMap, 39, 39, 106, 290
operator[](), 295, 296
vs. QMultiMap, 304
see also QMultiMap
QMapIterator, 35, 106
qMax(), 46, 196, 223, 486, 490
QMenu, 10, 29, 347, 349, 430, 462
addAction(), 12, 347, 349, 430, 463
addSeparator(), 12, 462
exec(), 30, 349, 462
QMessageBox, 61, 101, 137
addButton(), 101, 137
button(), 137
clickedButton(), 101, 137
clickedButton(), 101, 137
deselect(), 101, 137
setIcon(), 101, 137
setStyleSheet(), 101, 137
QNetworkReply (cont.)
setWindowModality(), 61, 101, 137
QMetaObject, 419, 423, 439
indexOfProperty(), 419, 434, 439
invokeMethod(), 133–134, 259–260, 423
QMimeTypeData, 159, 169, 171, 428, 429–430, 438
data(), 429
hasFormat(), 429, 438
hasHtml(), 429, 438
hasText(), 429, 438
html(), 429
setData(), 428
text(), 429
qMin(), 46, 191, 215
QModelIndex, 97, 100, 108, 111, 118, 125, 138, 139, 153, 154, 162, 164
column(), 120, 202, 203, 204
index(), 189
internalPointer(), 162
isValid(), 100, 120, 139, 154, 162
parent(), 100, 140, 141
row(), 100, 120, 140
QMouseEvent, 223, 238, 335
QMovie, 56, 58
QMultiHash, 304, 306, 338
insert(), 305, 339
remove(), 306
vs. QHash, 304
see also QHash
QMultiMap
see also QMap
QMultiMap vs. QMap, 304
QMutableListIterator, 293, 308
QMutableMapIterator, 37
QMutex, 267
vs. QReadWriteLock, 268
QMutexLocker, 267, 268
QNetworkAccessManager, 6, 10, 11
get(), 13
QNetworkProxy, 9
QNetworkReply, 14, 19
QNetworkRequest, 13
QObject, 39, 470
connect(), 12, 27, 39, 74, 94, 95, 227,
273, 344, 355, 484
deleteLater(), 292, 294
disconnect(), 55
eventFilter(), 83, 237–238
findChildren(), 32
installEventFilter(), 82, 227
metaObject(), 419, 423, 439
parent(), 236
property(), 434, 455
sender(), 200, 348, 349, 426, 431
setProperty(), 455
signal–signal connection, 228, 355, 356
tr(), 33, 140, 256, 275–276, 280, 285
QObject_cast<>(), 199, 200, 204, 232,
236, 295, 348, 349, 424, 425, 426
vs. dynamic_cast<>(), 349
QPageSetupDialog, 377
QPainter, 233, 375, 376, 379, 387, 388, 421, 422, 423
drawChord(), 465
drawEllipse(), 192, 465
drawImage(), 382
drawLine(), 192, 222, 235, 386
drawPath(), 401, 463, 466
drawPixmap(), 197
drawRect(), 222, 382
drawRoundedRect(), 465
drawText(), 193, 202, 380, 382, 386, 401
drawPath(), 38
end(), 421
fillRect(), 191, 202, 376, 388
fontMetrics(), 380, 382, 385, 386
restore(), 191, 202, 222, 384
save(), 191, 202, 222, 384
setBrush(), 192, 401, 463, 466
setFont(), 380, 382, 383, 386
setPen(), 192, 193, 202, 222, 386, 401, 463
setRenderHints(), 191, 202, 376, 388,
423
setViewport(), 375, 376, 384, 387, 388
QPainter (cont.)
viewport(), 376, 380, 384, 385, 388
QPainterPath, 390, 401, 461, 466
QPainterPathStroker, 401
QPaintEvent, 241
QPair, 142, 146, 156, 170, 307, 315, 347, 442
QPalette, 191, 193, 196, 202
alternateBase(), 336
base(), 243
button(), 233, 234
buttonText(), 234
dark(), 222
highlight(), 234, 243
highlightedText(), 234, 242
windowText(), 242
QParallelAnimationGroup, 473–474
QPen, 222, 233, 235, 435, 466
cosmetic, 401, 435
QPixmap, 197, 347, 443
fill(), 347
grabWidget(), 197
isNull(), 59
loadFromData(), 19
save(), 60
QPixmapCache, 197–198, 443–444
find(), 197, 443
insert(), 197, 443
setCacheLimit(), 443
QPlainTextEdit, 322, 323, 324, 336
appendPlainText(), 256
cursorRect(), 327
document(), 324, 335
setExtraSelections(), 336
setTextCursor(), 331, 334, 335
textCursor(), 326, 331, 334, 335, 336
toListPlainText(), 327
vs. QPlainTextEdit, 323
see also QAbstractScrollArea and QPlainTextEdit
QPoint, 62, 290, 293, 307, 308, 309, 310
QPointF, 192, 216, 349, 427, 461, 470, 471
QPoint/QPointF (cont.)
isNull(), 427
operator+=(), 427
operator-(), 427
rx(), 216, 218
ry(), 216
setX(), 216, 401
setY(), 401
x(), 216, 218
QPolygon/QPolygonF, 461
qPrintable(), 43, 46
QPrintDialog, 421
QPrinter, 373, 411, 413
QPrintPreviewDialog, 377
QProcess, 248
QProgressBar, 290, 295
setRange(), 295, 296
QPropertyAnimation, 470, 471–474, 487, 488, 489
setDuration(), 471, 487, 488, 489
setEasingCurve(), 471, 488
setEndValue(), 474, 487, 488, 489
setKeyValueAt(), 471, 472
setStartValue(), 474
start(), 471, 472
QPushButton, 101, 395
see also QAbstractButton
QRadialGradient, 244
QRadioButton, 94
see also QAbstractButton
qrand(), 45, 46, 50
QReadLocker, 305
QReadWriteLock, 304, 305
vs. QMutex, 268
qreal, unsuitable for files, 104
QRect/QRectF, 191, 212, 382, 458, 470, 486
adjust(), 382
adjusted(), 191, 222, 243, 457
bottom(), 215
center(), 191, 218, 243, 401
contains(), 216, 427
height(), 214, 382
intersects(), 220
left(), 215, 243
QRect/QRectF (cont.)
- moveTo(), 382
- normalized(), 220
- right(), 215, 243
- setBottomLeft(), 458
- setBottomRight(), 458
- setHeight(), 191
- setSize(), 382
- setWidth(), 191, 327
- size(), 382
- top(), 215
- translated(), 220
- united(), 466
- width(), 214, 382
- x(), 382
- y(), 382
qRed(), 300, 301
QRegExp, 327, 337, 338, 339, 340, 345, 385
- cap(), 340
- capturing policy, 339
- indexIn(), 340
- matchedLength(), 340
- setMinimal(), 339
- setPatternSyntax(), 339
QRegion, 221
qRegisterAnimationInterpolator<>()
- 489, 490
QResizeEvent, 237
QRgb, 300, 301
qRgb(), 300, 301
QRgba, 300
qRound(), 46, 193, 243, 300, 301, 380, 398, 465
qRound64(), 46
QRunnable, 257-260
- run(), 258
QScintilla text editor, 323
QScopedPointer, 61, 62, 101, 137, 428
QScriptEngine, 283
- evaluate(), 283
- globalObject(), 283
- hasUncaughtException(), 283
- uncaughtException(), 283
QScriptValue, 283
QScrollBar
- setPageSize(), 223
- setRange(), 211, 223
- setSingleStep(), 223
QSet, 29, 71, 108, 267, 418, 420
- contains(), 29, 69, 108
- operator<<(), 108, 268, 418
QSettings, 8, 12, 133, 36, 413
QSharedPointer, 61, 62, 101
QShortcut, 82, 325, 345
QSignalTransition, 487, 488, 489
QSize/QSizeF, 198, 232, 237, 350, 357, 382, 413, 470, 486
- boundedTo(), 299
- expandedTo(), 299
- rheight(), 486
- rwidth(), 377
- scale(), 382
- setWidth(), 486
qSort(), 116, 127, 327, 330
QSortFilterProxyModel, 107–113
- filterAcceptsRow(), 108, 111
- invalidateFilter(), 110
- setFilterKeyColumn(), 111
- setFilterRegExp(), 111
- setSourceModel(), 93, 108, 114
- sort(), 93, 111
QSound, 55, 57
QSpinBox, 205
- setRange(), 205
- setValue(), 204
- textFromValue(), 205
- value(), 204, 291
see also QAbstractSpinBox and
QDoubleSpinBox
qsrand(), 45, 46
qStableSort(), 331
QStack, 147
QStandardItem, 99, 106, 112, 142, 147,
186–188, 279, 312
QStandardItem (cont.)
  appendRow(), 312
  child(), 141, 146
  clone(), 187
  data(), 187
  index(), 134, 139
  row(), 141
  rowCount(), 140, 141, 146
  setData(), 187, 285
  setTextAlignment(), 312
  text(), 146
  vs. custom item, 156
QStandardItemEditorCreator, 186
QStandardItemModel, 102, 112, 144, 186, 265
  appendRow(), 99, 106, 312
  clear(), 103, 144, 308
  columnCount(), 273
  horizontalHeaderItem(), 144
  index(), 279
  invisibleRootItem(), 141, 144, 146, 147, 312
  item(), 273, 279, 285
  itemChanged(), 94, 136
  rowCount(), 273, 312
  setColumnCount(), 308
  sort(), 312
  takeRow(), 129
  vs. custom models, 112
QState, 474, 477, 478, 479, 481, 483, 486, 487
  addTransition(), 479, 484, 487, 489
  assignProperty(), 478, 479, 483, 486, 487
QStateMachine, 474, 477, 481, 483
  setInitialState(), 477, 483
  start(), 475, 483
QStatusBar::showMessage(), 56, 93, 275, 285
QString, 13, 229
  arg(), 13, 18, 74, 100, 187, 202, 205, 295
  at(), 326
  chop(), 254, 322
  clear(), 110
QString (cont.)
  compare(), 70, 331
  contains(), 27
  endsWith(), 13, 41, 322
  from long double, 281
  fromUtf8(), 41, 105
  indexOf(), 341
  isEmpty(), 17, 326
  isLetter(), 326
  lastIndexOf(), 341
  left(), 229
  length(), 121, 229, 326
  localeAwareCompare(), 331
  mid(), 13
  number(), 50, 121, 401, 402
  prepend(), 27, 385
  replace(), 292
  right(), 331
  split(), 327, 385
  startsWith(), 13
  toLower(), 70, 254
  toUpper(), 251
  toUtf8(), 105
  trimmed(), 33, 71
QStringList, 12, 71, 106, 150, 169, 182, 203, 308, 309, 352, 397
  isEmpty(), 352
  join(), 71, 285
  removeDuplicates(), 268, 327
  sort(), 442
  see also QList
QStringListModel, 324
QStringRef, 147
QStyle, 194
  sizeFromContents(), 121
QStyleFactory, 185, 190, 194, 201
  createEditor(), 194, 199, 203
  paint(), 191, 194, 196, 202
  setEditorData(), 194, 199, 204
  setModelData(), 194, 200, 204
  sizeHint(), 194, 198
  updateEditorGeometry(), 194
QStyleOptionComboBox, 121
QStyleOptionGraphicsItem, 393, 401, 402
QStyleOptionViewItem, 191, 193, 202, 222
QSvgGenerator, 375, 387, 422
QSvgRenderer, 382
QSyntaxHighlighter, 337, 338
  currentBlockState(), 341
  highlightBlock(), 337, 339
  previousBlockState(), 337, 341
  setCurrentBlockState(), 337, 339, 341
  setFormat(), 340, 341
QSystemTrayIcon, 10
  setIcon(), 17
Qt
  escape(), 352, 366
  ItemDataRole enum, 119
  ItemFlag enum, 119
  ItemSelectionMode enum, 406
Qt⇔JavaScript communication, 39
Qt rich text; see QTextDocument
QTableView, 88, 91, 93, 185
  horizontalHeader(), 95
  setItemDelegate(), 93
  setModel(), 93
  sortByColumn(), 95, 111
  verticalHeader(), 93
  see also QAbstractItemView
QTableWidget, 49, 88
QTableWidgetItem, 49
QtConcurrent
  filtered(), 270, 271
  filteredReduced(), 277, 278–279
  mapped(), 281, 282
  run(), 252–257
QtCore module, 20, 348
<QtDebug>, 46
QTextBlock, 318, 328, 352, 385
QTextBlockFormat, 318, 329, 357, 358, 368, 371
  font(), 357
QTextCharFormat (cont.)
  fontItalic(), 352, 357
  fontPointSize(), 357
  fontStrikeOut(), 348, 350, 352
  fontWeight(), 352, 357
  foreground(), 352
  intProperty(), 371
  setBackground(), 336
 setFontItalic(), 338
  setFontStrikeOut(), 348
  setForeground(), 338
  setProperty(), 336, 371
  setVerticalAlignment(), 348
  verticalAlignment(), 350, 352
QTextCursor, 326, 331, 332, 334, 335, 357, 367–371
  anchor(), 328, 332
  atBlockEnd(), 328
  atBlockStart(), 328
  atEnd(), 328
  atStart(), 328
  beginEditBlock(), 328, 329
  block(), 328
  blockCharFormat(), 328
  blockFormat(), 328, 357
  charFormat(), 328
  clearSelection(), 328, 334, 336
  columnNumber(), 328
  createList(), 328, 367
  currentFrame(), 328
  currentList(), 328
  currentTable(), 328
  deleteChar(), 328
  deletePreviousChar(), 328
  document(), 328
  endEditBlock(), 329
  hasComplexSelection(), 329
  hasSelection(), 329
  insertBlock(), 329, 368, 369, 371
  insertFragment(), 329, 369
  insertFrame(), 329
  insertHtml(), 329, 368
  insertImage(), 329, 371
  insertList(), 329, 367
  insertTable(), 329, 369
QTextCursor (cont.)
insertText(), 329, 331, 368, 369, 371
isCopyOf(), 329
isNull(), 329
joinPreviousEditBlock(), 329
mergeBlockCharFormat(), 329
mergeBlockFormat(), 329, 368
mergeCharFormat(), 330, 368
MoveOperation enum, 334
movePosition(), 330, 331, 332
position(), 330, 331, 332
removeSelectedText(), 330, 334
select(), 326, 330
selectedTableCells(), 330
selectedText(), 326, 330
selection(), 319, 330
selectionEnd(), 330
selectionStart(), 330
setBlockCharFormat(), 330
setBlockFormat(), 330
setCharFormat(), 330
setPosition(), 330, 331, 332
setVisualNavigation(), 330
visualNavigation(), 330
begin(), 352, 354
beginLayout(), 384
drawContents(), 375, 376, 384
file quality, 361–364
idealWidth(), 198, 350
print(), 373, 378
setDefaultFont(), 198, 368, 384
setDocumentMargin(), 372
setHtml(), 37, 198, 365, 383
setPageSize(), 372
setTextWidth(), 384
setUseDesignMetrics(), 384
toHtml(), 374
toPlainText(), 37
QTextDocumentFragment, 319, 329, 330, 369
fromHtml(), 17, 37
QTextDocumentFragment (cont.)
toPlainText(), 17, 37
vs. QTextFragment, 319
QTextDocumentWriter, 373
QTextEdit, 322, 335, 336, 344, 354
currentCharFormat(), 350
fontItalic(), 350
fontWeight(), 344, 350
mergeCurrentCharFormat(), 348
setAcceptRichText(), 344
setAlignment(), 354
setFontFamily(), 354
setFontItalic(), 344
setFontPointSize(), 354
setFontWeight(), 344, 354
setHtml(), 354
setLineWrapMode(), 344
setTabChangesFocus(), 344
setTextColor(), 348, 356
setWordWrapMode(), 344
textColor(), 357
textCursor(), 357
toHtml(), 351, 354
vs. QPlainTextEdit, 323
see also QAbstractScrollView and QPlainTextEdit
QTextFormat, 336, 371
QTextFragment, 318, 319, 352, 385
vs. QTextDocumentFragment, 319
QTextFrame, 319, 328, 329
QTextFrameFormat, 319, 329
QTextImageFormat, 319, 329
QTextLength, 370
QTextList, 318, 328, 329
QTextListFormat, 329
QTextOption, 193, 202
QTextStream, 374
operator<<(), 374
setCodec(), 374
QTextTable, 319, 328, 329, 369
QTextTableCell, 319, 370, 371
QTextTableFormat, 319, 329, 370
QtGui module, 348
QThread, 298, 314
   idealThreadCount(), 252, 253, 258, 309
   isFinished(), 297
   isRunning(), 310
   start(), 296, 309
   wait(), 292, 293, 308
QThreadPool
   activeThreadCount(), 251, 256, 257
   globalInstance(), 251, 258
   start(), 258
   waitForDone(), 251, 257
QTimeLine, 133
QTimer, 74, 83, 133
   setInterval(), 74, 83
   setSingleShot(), 74, 83
   singleShot(), 11, 14, 32, 133, 413
QtMultimedia module, 86, 447
QtNetwork module, 6
QToolBar, 354
QTransform, 405, 454
QTreeView, 88, 132, 135, 163, 185, 188
   expand(), 312
   resizeColumnToContents(), 69
   setAllColumnsShowFocus(), 135, 152
   setColumnWidth(), 312
   setRowHidden(), 141
   setSortingEnabled(), 308, 313
see also QAbstractItemView
QTreeWidget, 88
   currentItem(), 75
   invisibleRootItem(), 73
   itemBelow(), 79
QTreeWidgetItem, 66, 70, 79
   child(), 79
   operator<(), 70
QtWebKit module, 6, 21
QtXml module, 16, 20
   question() (AQP), 61, 285
QuickTime library, 60
QUILoader, 48
qUncompress(), 171
QUrl, 13
   fromLocalFile(), 45, 297
   fromUserInput(), 297
   toString(), 11
QVariant, 12, 66, 118, 161, 186, 442, 470, 489
   invalid, 120, 122, 161, 163, 434
   isValid(), 434
   toBool(), 166
   toByteArray(), 133
   toDate(), 348
   toDouble(), 187, 273, 434
   toInt(), 97, 100, 124, 445
   toPoint(), 348
   toSize(), 445
   toString(), 70, 98, 348
   toStringList(), 133
   toUInt(), 104
   value<>(), 121, 348, 434, 445
QVariantAnimation, 489
QBoxLayout, 227
QVector, 116, 252, 258, 309, 370, 431, 433
   append(), 433
   at(), 432
   constBegin(), 431
   constEnd(), 431
   operator<(), 370
qVersion(), 46
QWeakPointer, 62, 290
QWebElement, 21
QWebFrame, 21, 40, 41
   addToJavaScriptWindowObject(), 40
   evaluateJavaScript(), 41
QWebHistory, 21
QWebHistoryItem, 21, 29
QWebInspector, 22–23
QWebPage, 21, 25, 32, 39, 47
   createPlugin(), 47–48
   mainFrame(), 39, 41
QWebSettings, 21, 23, 39, 45
QWebView, 21, 25, 48, 390
   load(), 27
   predefined actions, 26
QWheelEvent, 439
QWidget, 48, 226, 227, 236, 290, 396, 441, 450, 481, 483
actions(), 29
addActions(), 345
closeEvent(), 79, 136, 257, 313
event(), 256
focusInEvent(), 24
font(), 218, 486
graphicsEffect(), 486, 487, 488
hide(), 275, 280, 308, 313
isWindowModified(), 137, 438
keyPressEvent(), 239, 333, 349, 457
layout(), 295
layoutDirection(), 224
mapFromGlobal(), 427
mapToGlobal(), 349
minimumSizeHint(), 232, 236, 350, 441, 486, 487
mousePressEvent(), 238, 335
paintEvent(), 233, 241
palette(), 222, 233, 234, 242, 243, 336
pos(), 427
resize(), 237, 377
restoreGeometry(), 133
saveGeometry(), 136
setAttribute(), 101
setBackgroundColorRole(), 227
setContextMenu(), 12
setContextMenuPolicy(), 26, 29, 344
setFixedSize(), 194, 196, 441
setFocus(), 24, 99, 308, 313, 394
setFocusPolicy(), 211, 236
setFont(), 211
setGraphicsEffect(), 483
setLayout(), 444
setMinimumSize(), 236, 398
setPalette(), 196
setStyleSheet(), 395
setToolTip(), 345
setWindowFilePath(), 411, 413, 421
QWidget (cont.)
setWindowModified(), 91, 94, 132, 437
setWindowTitle(), 132, 137, 412
show(), 8, 273, 313
sizeHint(), 194, 232, 237, 327, 350, 486
wheelEvent(), 439
window(), 450
windowFilePath(), 412, 421
QWriteLocker, 305, 306
QXmlStreamAttributes, 147
QXmlStreamReader, 147, 171, 173, 181
QXmlStreamWriter, 144, 169, 170, 181

R
radiansFromDegrees() (AQP), 192
RAII (Resource Acquisition Is Initialization), 62
RAND_MAX, 46
read-write locks; see QMutex and QReadWriteLock
read-write locks vs. mutexes, 268
readAll()
    QByteArray, 19, 41
    QFile, 315
redirection, HTTP, 19
reducers, 262–263, 277–281
reinterpret_cast<>() , 301
remove() (QMultiHash), 306
removeAll() (QList), 116, 327
removeAt() (QList), 125
removeColumns() (QAbstractItemModel), 116, 118, 123
removeDuplicates() (QStringList), 268, 327
removeItem() (QGraphicsScene), 428, 429
removeOne() (QList), 116
removeRow() (QAbstractItemModel), 100, 126, 140
removeRows() (QAbstractItemModel), 116, 118, 123, 125, 167
Index

removeSelectedText() (QTextCursor), 330, 334, 335
render() (QGraphicsScene), 421–422
rendering, HTML, 30
rendering, off-screen, 464
replace() (QString), 292
reset() (QAbstractItemModel), 181
resetTransform() (QGraphicsItem), 404
resizable models, 116, 123, 125, 158
resizeColumnToContents() (QTreeView), 69
resizeEvent() (QAbstractItemView), 210, 223
resize() (QWidget), 237, 377
restore() (QPainter), 191, 202, 222, 384
restoreGeometry() (QWidget), 133
restoreOverrideCursor() (QApplication), 69
result() (QFutureWatcher), 280
Return key, 199, 200
RGB (Red, Green, Blue), 192, 300
rheight() (QSize/QSizeF), 486
rich text; see QTextDocument
rich text, painting, 383–386
right()
  QRect/QRectF, 215, 243
  QString, 331
root item, of trees, 147
rootIndex() (QAbstractItemView), 212, 213, 216, 221
rootPath() (QDir), 189
rotation() (QGraphicsItem), 404
rounding up numbers, 229
  see also qRound()
row()
  QModelIndex, 100, 120, 140
  QStandardItem, 141
rowCount()
  QAbstractItemModel, 115, 118, 123, 154, 163, 212
  QStandardItem, 140
  QStandardItemModel, 273, 312
rows, hiding, 141
rowsAboutToBeRemoved() (QAbstractItemView), 210, 217
rowsInserted() (QAbstractItemView), 210, 217
RssPanel example, 20
RTTI (Run Time Type Information), 232, 349, 424
rubber band selections, 439
run()
  QRunnable, 258
  QtConcurrent, 252–257
rwidth() (QSize/QSizeF), 377
rx() (QPoint/QPointF), 216, 218
ry() (QPoint/QPointF), 216

S
save()
  QImage, 254, 294, 302, 376, 388, 423
  QPainter, 191, 202, 222, 384
  QPixmap, 60
saveGeometry() (QWidget), 136
saving data, 103–105, 144–146
saving scenes, 415–417
Scalable Vector Graphics; see .svg
scale()
  QGraphicsItem, 404
  QGraphicsView, 439, 440
  QSize/QSizeF, 382
scaled() (QImage), 299
scaling, 382
scanLine() (QImage), 300
scene() (QGraphicsItem), 404
sceneBoundingRect() (QGraphicsItem), 404, 433
scenePos() (QGraphicsItem), 404
scenes
  coordinates, 433
  exporting, 420–423
  indexing, 395
  loading, 417–420
  printing, 420–423
  saving, 415–417
  see also QGraphicsScene
Scintilla text editor, 323
screenCount() (QDesktopWidget), 377
screenGeometry() (QDesktopWidget), 377
scroll area vs. viewport, 209, 213
scrollbar; see QScrollBar
scrollContentsBy() (QAbstractItemView), 210, 219
scrollDirtyRegion() (QAbstractItemView), 219
scrolling; see QAbstractScrollArea and QScrollArea
scrollTo() (QAbstractItemView), 97, 134, 189, 210, 215
scrollToBottom() (QAbstractItemView), 99
SCXML (State Chart XML), 470
select() (QTextCursor), 326, 330
selectedItems() (QGraphicsScene),
421, 423, 428, 431, 434, 438, 439, 455
selectedTableCells() (QTextCursor), 330
selectedText() (QTextCursor), 326, 330
selecting in documents, 326, 330, 332, 336
selecting items in views, 97
selection() (QTextCursor), 319, 330
selectionEnd() (QTextCursor), 330
selectionModel() (QAbstractItemView), 95, 100, 136, 220, 222, 275, 279
selections, 218–219, 330, 332, 336
rubber band, 439
selectionStart() (QTextCursor), 330
sender() (QObject), 200, 348, 349, 426, 431
sendEvent() (QApplication), 255
separator() (QDir), 254, 322
separator, path, 57
setAcceptRichText() (QTextEdit), 344
setAlignment()
QAbstractSpinBox, 186, 205
QTextEdit, 354
setAllColumnsShowFocus() (QAbstractItemView), 135, 152
setApplicationName() (QApplication), 8, 412
setAttribute()
QApplication, 345
QWidget, 101
setBackground() (QTextCharFormat), 336
setBackgroundBrush()
QGraphicsScene, 435
QGraphicsView, 398, 417
setBackgroundRole() (QWidget), 227
setBlockCharFormat() (QTextCursor), 330
setBlockFormat() (QTextEdit), 330
setBottomLeft() (QRect/QRectF), 458
setBottomRight() (QRect/QRectF), 458
setBrush()
QGraphicsAbstractShapeItem, 398
QPainter, 192, 401, 463, 466
setBuddy() (QLabel), 483
setCacheLimit() (QPixmapCache), 443
setCacheMode() (QGraphicsItem), 464
setCaseSensitivity() (QCompleter), 321, 324, 325
setCentralWidget() (QMainWindow), 135, 152, 398, 413
setCharFormat() (QTextCursor), 330
setCheckable() (QAction), 12, 346, 463
setChecked()
QAbstractButton, 196
QAction, 12, 346, 350, 357, 463
QCheckBox, 483
setCodec() (QTextStream), 374
setColumnCount() (QStandardItemModel), 308
setColumnHidden() (QAbstractItemView), 189
setColumnWidth() (QTreeView), 312
setCompleter() (QLineEdit), 321
setCompletionMode() (QCompleter), 324
setCompletionPrefix() (QCompleter), 327
setContextMenu(QWidget), 12
setContextMenuPolicy(QWidget), 26, 29, 344
setCurrentBlockState(QSyntaxHighlighter), 337, 339, 341
setCurrentIndex()
  QAbstractItemView, 134, 189, 223
  QComboBox, 35, 100, 204, 445, 483
setCurrentItem(QAbstractItemView), 99
setCursor(QGraphicsItem), 458, 459
setData()
  QAbstractItemModel, 118, 124, 166, 204
  QAction, 12, 29, 346, 347, 431, 463
  QMimeData, 428
  QStandardItem, 187, 285
setDecimals(QDoubleSpinBox), 186
setDefaultButton(QMessageBox), 101
setDefaultFont(QTextDocument), 198, 368, 384
setDockOptions(QMainWindow), 414
setDocumentMargin(QTextDocument), 372
setDragDropMode(QAbstractItemView), 152
setDragMode(QGraphicsView), 439
setDuration(QPropertyAnimation), 471, 487, 488, 489
setEasingCurve(QPropertyAnimation), 471, 488
setEditorData(QStyledItemDelegate), 194, 199, 204
setEditTriggers(QAbstractItemView), 271, 275
setEndValue(QPropertyAnimation), 474, 487, 488, 489
setExtraSelections(QPlainTextEdit), 336
setFilterKeyColumn(QSortFilterProxyModel), 111
setFilterRegExp(QSortFilterProxyModel), 111
setFixedSize(QWidget), 194, 196, 441
setFlag(QGraphicsItem), 391, 404
setFlags(QGraphicsItem), 398, 404, 450
setFocus(QWidget), 24, 99, 308, 313, 394
setFocusPolicy(QWidget), 211, 236
setFont()
  QGraphicsItem, 450
  QPainter, 380, 382, 383, 386
  QWidget, 211
setFontFamily(QTextEdit), 354
setFontItalic()
  QTextCharFormat, 338
setFont(QTextEdit), 344
setFontPointSize(QTextEdit), 354
setFontStrikeOut(QTextCharFormat), 348
setFontWeight(QTextEdit), 344, 354
setForeground(QTextCharFormat), 338
setFormat(QSyntaxHighlighter), 340, 341
setGraphicsEffect(QGraphicsItem), 405
setGroup(QGraphicsItem), 483
setHeaderData(QAbstractItemModel), 118, 125, 165
setHeight(QRect/QRectF), 191
setHorizontalScrollBarPolicy(QAbstractScrollArea), 344
setHtml()
  QTextDocument, 37, 198, 365, 383
setInitialState(QStateMachine), 477, 483
setInterval(QTimer), 34, 83
setItemDelegate(QTableView), 93
setItems(QGraphicsItem), 405
QWidget, 483
setGroup(QGraphicsItem), 405
setHeaderData(QAbstractItemModel), 118, 125, 165
setHeight(QRect/QRectF), 191
setHorizontalScrollBarPolicy(QAbstractScrollArea), 344
setHtml()
  QTextDocument, 37, 198, 365, 383
  QPlainTextEdit, 354
setIcon()
  QAction, 59, 357, 431
  QMessageBox, 101, 137
  QSystemTrayIcon, 17
setInitialState(QStateMachine), 477, 483
setInterval(QTimer), 74, 83
setItemDelegate(QTableView), 93
setItemDelegateForColumn() (QAbstractItemView), 135, 152, 189
setItemIcon() (QComboBox), 446
setItemIndexMethod() (QGraphicsScene), 394, 477
setKeyValueAt() (QPropertyAnimation), 471, 472
setLayout() (QWidget), 444
setLineWrapMode() (QTextEdit), 344
setMenu() (QAction), 345, 430
setMimeData() (QClipboard), 428
setMinimal() (QRegExp), 339
setMinimumSize() (QWidget), 236, 398
setModel() (QAbstractItemView, 135, 152, 189, 209, 210, 211 QComboBox, 93 QCompleter, 321, 324 QTableView, 93
setModelColumn() (QComboBox), 93, 94
setModelData() (QStyledItemDelegate), 324, 325, 327
setModelSorting() (QCompleter), 393, 405, 451
setOrganizationDomain() (QApplication), 8
setOrganizationName() (QApplication), 8
setOverrideCursor() (QApplication), 69
pageSize() (QTextDocument), 372
setPageStep() (QScrollBar), 223
setPalette() (QWidget), 196
setProperty() (QObject, 455 QTextCharFormat, 336, 371
setQuitOnLastWindowClosed() (QApplication), 8
setRotation() (QDoubleSpinBox), 186 QProgressBar, 295, 296 QScrollBar, 211, 223 QSpinBox, 205
setRenderHints() (QGraphicsView, 398, 439 QPainter, 191, 202, 376, 388, 423
setScale() (QGraphicsView), 393, 398, 405, 451
setScale() (QGraphicsScene), 398, 413
setSelected() (QGraphicsItem), 405, 420, 421, 450
setSelected() (QAbstractItemView), 210, 220
setShortcuts() (QAction), 26, 57
setSingleShot() (QTimer), 74, 83
setSingleStep() (QScrollBar), 223
setSize() (QRect/QRectF), 382
setSortingEnabled() (QTreeView), 308, 313
setSourceModel() (QSortFilterProxyModel), 93, 108, 114
setStartValue() (QPropertyAnimation), 474
setStyleSheet() (QWidget), 395
setTabChangesFocus() (QTextEdit), 344
setTabPosition() (QMainWindow), 414
setText() (QAbstractButton), 291 QAction, 59 QLabel, 196 QMessage, 101, 137, 139
setTextAlignment() (QStandardItem), 312
setTextColor() (QTextEdit), 348, 356
setTextCursor() (QPlainTextEdit), 331, 334, 335
setTextFormat() (QLabel), 194
setTextWidth() (QTextDocument), 384
settings; see QSettings
settings, policy, 20
setTextCursor() (QPlainTextEdit), 331, 334, 335
setTextFormat() (QLabel), 194
setTextWidth() (QTextDocument), 384
settings; see QSettings
settings, policy, 20
setToolTip() (QGraphicsItem), 405
QWidget, 345
setTransform() (QGraphicsItem), 393, 405, 451, 454
setTransformOriginPoint() (QGraphicsItem), 393, 405, 451
setup, printer, 377
setUseDesignMetrics() (QTextDocument), 384
setValue() (QSpinBox), 204
setVersion() (QDataStream), 104, 106, 127
setVerticalAlignment() (QTextCharFormat), 348
setVerticalScrollBarPolicy() (QAbstractScrollArea), 344
setViewport() (QPainter), 375, 376, 384, 387, 388
setVisible() (QGraphicsItem), 405, 435
setVerticalAlignment() (QTextCursor), 330
setWidth() (QScrollArea), 227
setWidth()
  QRect/QRectF, 191, 327
  QSize/QSizeF, 486
setWindowFilePath() (QWidget), 411, 413, 421
setWindowModality() (QMessageBox), 61, 101, 137
setWindowTitle() (QWidget), 91, 94, 132, 437
setWindowTitle() (QWidget), 132, 137, 412
setTextWrapMode() (QTextEdit), 344
setTextWrapMode() (QCompleter), 324, 325
setX()
  QGraphicsItem, 405
  QPoint/QPointF, 216, 401
setY()
  QGraphicsItem, 405
  QPoint/QPointF, 401
setZValue() (QGraphicsItem), 405, 435, 453
shape() (QGraphicsItem), 400, 405, 466
shapes, custom, 400
shared_ptr() (std), 62
shortcuts, keyboard, 15, 82
show()
  QDialog, 102
  QGraphicsItem, 405
  QWidget, 8, 273, 313
showMessage() (QStatusBar), 56, 93, 275, 285
signal—signal connection, 228, 355, 356
signals, across threads, 300
single shot timers in constructors, 12
singleShot() (QTimer), 11, 14, 32, 133, 413, 477
size()
  QFileInfo, 312, 315
  QImage, 299
  QRect/QRectF, 382
sizeFromContents() (QStyle), 121
sizeHint()
  QStyledItemDelegate, 194, 198
  QWidget, 194, 232, 237, 327, 350, 486
slot invocation, 134
  parameterizing, 13
slot invocation vs. custom events, 260
smart pointers, 62
  see also QScopedPointer, QSharedPointer, and QWeakPointer
snprintf(), 281
software patents, disadvantages of, 53
sort()
    QAbstractItemModel, 442
    QSortFilterProxyModel, 93, 111
    QStandardItemModel, 312
    QStringList, 442
sortByColumn() (QTableView), 95, 111
sorting models; see QSortFilterProxyModel
sorting trees, 70
sound
    codecs, 53
    playing, 55
    policy, 53, 55
Source-Highlight Qt library, 337
Space key, 65
spinbox; see QAbstractSpinBox, QDoubleSpinBox, and QSpinBox
split() (QString), 327, 385
standard directories; see QDir
start()
    QPropertyAnimation, 471, 472
    QStateMachine, 475, 483
    QThread, 296, 309
    QThreadPool, 258
startsWith() (QString), 13
startup policy, 133
state machine framework, 474–490
states; see QFinalState, QState, and QStateMachine
status bar; see QStatusBar
std
    ceil(), 435, 436
    exception, 105
    max_element(), 431, 432
    min_element(), 431, 432
    numeric_limits::max(), 186
    numeric_limits::min(), 436
    pow(), 439, 440
    shared_ptr(), 62
stopping threads, 292–294
storageLocation() (QDesktopServices), 68, 83
storing images, 60
strings
    caching, 108
    localized, 276
    see also QLocale and QString
style() (QApplication), 121
suffix; see files
suffix() (QFileInfo), 69, 251, 254
suffixesForMimeTypes() (AQP), 71, 72
summing floating-point numbers, 280
supportedDragActions() (QAbstractItemModel), 159, 168
supportedDropActions() (QAbstractItemModel), 159, 168
surrogate items, 264, 266
.svg (files), 362, 363, 375, 382, 387, 422
swap() (QList), 157
synchronous downloads, 43
syntax highlighting, 320, 337–342
    see also QSyntaxHighlighter
syntax highlighting, Source-Highlight Qt library, 337

T

Tab key, 50
table models vs. tree models, 89
tables
    in documents, 319
    inserting into documents, 369–371
    see also QTableView and QTableWidget
takeAt() (QList), 158
takeFirst() (QList), 352, 486
takeRow() (QStandardItemModel), 129
taskbar tray applications, 7
template class, 304
template function, 138
tempPath() (QDir), 189
terminating applications, 8, 79, 257
text()
    QAbstractButton, 251, 291
    QMimeData, 429
    QStandardItem, 146
text, selecting, 326

text cursor; see QTextCursor
text editor; see QLineEdit, QPlainTextEdit, and QTextEdit
text editor, Scintilla, 323
textColor() (QTextEdit), 357
textCursor()
    QPlainTextEdit, 326, 331, 334, 335, 336
    QTextEdit, 357
TextEdit example, 353–358
textFromValue() (QSpinBox), 205

thread-safe hash, 304

threading controversy, 246

threads
    signals across, 300
    stopping, 292–294

.tiff (files), 375, 388

TiledListView example, 208–224

Timelog (custom model) example, 151–183, 193–200, 343–353

Timelog (QStandardItemModel) example, 130–150, 193–200, 343–353
toBool() (QVariant), 166
toByteArray() (QVariant), 133
toDate() (QVariant), 348
toDouble()
    QLocale, 229
    QVariant, 187, 273, 434

toFloat() (QLocale), 229
toGraphicsObject() (QGraphicsItem), 424, 425
toHtml()
    QPlainTextDocument, 374
    QTextEdit, 351, 354

toInt()
    QLocale, 229, 243
    QVariant, 97, 100, 124, 445
toLower() (QString), 70, 254
toNativeSeparators() (QDir), 254, 312, 321, 322
toolTip() (QGraphicsItem), 405
top() (QRect/QRectF), 215
toPlainText()
    QPlainTextEdit, 327
toPlainText() (cont.)
    QPlainTextDocument, 37
    QPlainTextDocumentFragment, 17, 37
toPoint() (QVariant), 348
toSize() (QVariant), 445
toString()
    QDate, 146, 170, 193
    QUrl, 11
    QVariant, 70, 98, 348
toStringList() (QVariant), 133
toUInt()
    QLocale, 229
    QVariant, 104
toupper() (QString), 251
toUtf8() (QString), 105

tr() (QObject), 33, 140, 256, 275–276, 280, 285

transform() (QGraphicsItem), 405

transformations, graphics, 391

transitions; see state machine framework

translated() (QRect/QRectF), 220

transparency, 300

tree changing policy, 154

tree changing policy, 154
tree changing policy, 154
tree changing policy, 154


tree changing policy, 154


tree changing policy, 154

tree changing policy, 154


tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154


tree changing policy, 154

tree changing policy, 154


tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154


tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

tree changing policy, 154

UncaughtException() (QScriptEngine), 283

Unicode, 41, 374

united() (QRect/QRectF), 466

unsaved changes, 94, 99, 136, 137

update() (QGraphicsItem), 406, 461

updateEditorGeometry() (QStyledItemDelegate), 194
Index

updateGeometries() (QAbstractItemView), 210, 223
updates, lazy, 209
user data, 73
user interface blocking, 102
UTF-8 encoding, 374
UTI (Uniform Type Identifier), 72

V

value()
  QHash, 73, 214
  QList, 157
  QSpinBox, 204, 291
value>() (QVariant), 121, 348, 434, 445
values() (QHash), 305, 306
verticalAlignment() (QTextCharFormat), 350, 352
verticalHeader() (QTableView), 93
verticalOffset() (QAbstractItemView), 210, 219
verticalScrollBar() (QAbstractScrollArea), 211, 223, 230, 312
video
codecs, 53
  files, location of, 83
video player; see Play Video example
view components, 207
viewOptions() (QAbstractItemView), 222
viewport()
  QAbstractScrollArea, 199, 209, 212, 215, 219, 222
  QPainter, 376, 380, 384, 385, 388
viewport coordinates, 214, 216, 220, 221, 222, 224, 392
viewport vs. scroll area, 209, 213
views; see QAbstractItemView and QGraphicsView
views() (QGraphicsScene), 450, 451
views, moving items in, 158–159, 173–180
visual cues, 53
visualNavigation() (QTextCursor), 330
visualRect() (QAbstractItemView), 210, 213, 221
visualRegionForSelection() (QAbstractItemView), 210, 221
volatile bool, 247–248, 307

W

wait() (QThread), 292, 293, 308
waitForDone() (QThreadPool), 251, 257
waitForFinished() (QFutureWatcher), 272
warning() (AQP), 60, 61, 76, 77, 374
.wav (files), 53
Weather Tray Icon example, 7–20
web applications vs. desktop applications, 5–6
web-based applications, 5
Web Inspector; see QWebInspector
web pages, zooming, 23
wheel, mouse, 236
wheelEvent() (QWidget), 439
widgets
  aggregating, 225
  coordinates, 214, 216, 220
  Internet-aware, 6
  painting, 195–198
  when size doesn't matter, 209, 213, 235, 238
  see also QWidget
width()
  QFontMetrics, 121, 198, 212, 227, 385, 386
  QGraphicsScene, 435
  QImage, 300, 301
  QRect, 214, 382
width, of HTML text, 198
window() (QWidget), 450
window coordinates, 392
window-modal dialogs, 356
windowFilePath() (QWidget), 412, 421
Windows platform differences, 254
windowText() (QPalette), 242
write locks; see QMutex and QRead-WriteLock

X
x()
  QGraphicsItem, 406
  QPoint/QPointF, 216, 218
  QRect/QRectF, 382
XML; see QXmlStreamReader and QXmlStreamWriter
XML, DOM (Document Object Model), 18
XML escaping, 145, 147, 352, 353
XmlEdit example, 322–342
  .xpm (files), 375

Y
y()
  QGraphicsItem, 406
  QRect/QRectF, 382

Z
z value, 391
Zipcodes (custom model) example, 113–128, 201–205
Zipcodes (QStandardItemModel) example, 90–107, 201–205
zooming web pages, 23
zValue() (QGraphicsItem), 406, 452