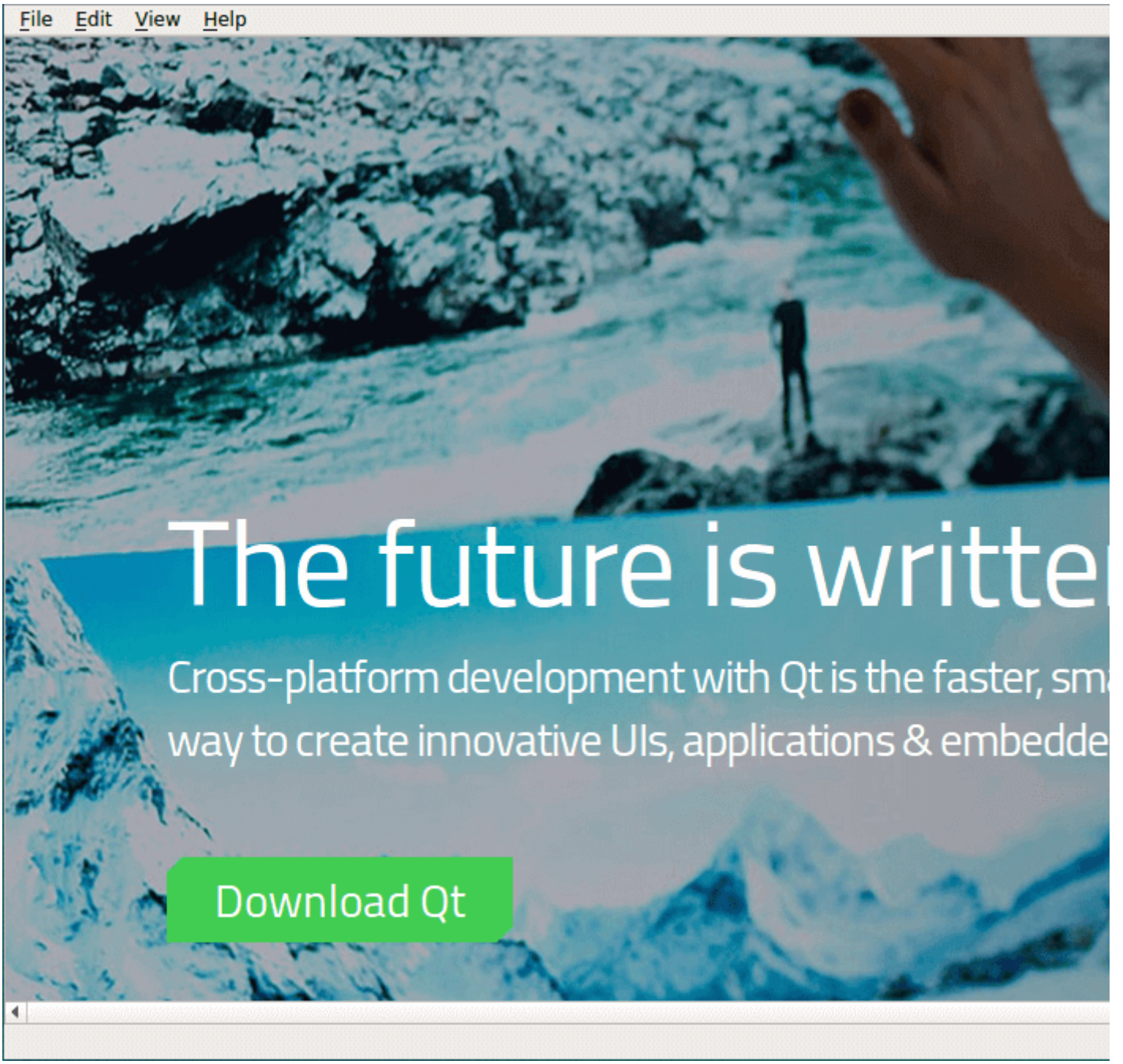
Contents

Image Viewer Example

QLabel is typically used for displaying text, but it can also display an image. QScrollArea provides a scrolling view around another widget. If the child widget exceeds the size of the frame, QScrollArea automatically provides scroll bars.

The example demonstrates how QLabel's ability to scale its contents (QLabel::scaledContents), and QScrollArea's ability to automatically resize its contents (QScrollArea::widgetResizable), can be used to implement zooming and scaling features. In addition the example shows how to use QPainter to print an image.



Screenshot of the Image Viewer example

With the Image Viewer application, the users can view an image of their choice. The **File** menu gives the user the possibility to:

- **Open...** - Open an image file
- **Print...** - Print an image
- **Exit** - Exit the application

Once an image is loaded, the **View** menu allows the users to:

- **Zoom In** - Scale the image up by 25%
- **Zoom Out** - Scale the image down by 25%

- **Normal Size** - Show the image at its original size
- **Fit to Window** - Stretch the image to occupy the entire window

In addition the **Help** menu provides the users with information about the Image Viewer example in particular, and about Qt in general.

ImageViewer Class Definition

```
class ImageViewer : public QMainWindow
{
    Q_OBJECT

public:
    ImageViewer();
    bool loadFile(const QString &);

private slots:
    void open();
    void saveAs();
    void print();
    void copy();
    void paste();
    void zoomIn();
    void zoomOut();
    void normalSize();
    void fitToWindow();
    void about();

private:
    void createActions();
    void createMenus();
    void updateActions();
    bool saveFile(const QString &fileName);
    void setImage(const QImage &newImage);
    void scaleImage(double factor);
    void adjustScrollBar(QScrollBar *scrollBar, double factor);

    QImage image;
    QLabel *imageLabel;
    QScrollArea *scrollArea;
    double scaleFactor;

#ifndef QT_NO_PRINTER
    QPrinter printer;
#endif

    QAction *saveAsAct;
    QAction *printAct;
    QAction *copyAct;
    QAction *zoomInAct;
    QAction *zoomOutAct;
    QAction *normalSizeAct;
    QAction *fitToWindowAct;
};
```

The ImageViewer class inherits from QMainWindow. We reimplement the constructor, and create several private slots to facilitate the menu entries. In addition we create four private functions.

We use createActions() and createMenus() when constructing the ImageViewer widget. We use the updateActions() function to update the menu entries when a new image is loaded, or when the **Fit to Window** option is toggled. The zoom slots use scaleImage() to perform the zooming. In turn, scaleImage() uses adjustScrollBar() to preserve the focal point after scaling an image.

ImageViewer Class Implementation

```
ImageViewer::ImageViewer()
    : imageLabel(new QLabel)
    , scrollArea(new QScrollArea)
    , scaleFactor(1)
{
    imageLabel->setBackgroundRole(QPalette::Base);
    imageLabel->setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);
    imageLabel->setScaledContents(true);

    scrollArea->setBackgroundRole(QPalette::Dark);
    scrollArea->setWidget(imageLabel);
    scrollArea->setVisible(false);
    setCentralWidget(scrollArea);

    createActions();

    resize(QGuiApplication::primaryScreen()->availableSize() * 3 / 5);
}
```

In the constructor we first create the label and the scroll area.

We set imageLabel's size policy to ignored, making the users able to scale the image to whatever size they want when the **Fit to Window** option is turned on. Otherwise, the default size polizy (preferred) will make scroll bars appear when the scroll area becomes smaller than the label's minimum size hint.

We ensure that the label will scale its contents to fill all available space, to enable the image to scale properly when zooming. If we omitted to set the imageLabel's scaledContents property, zooming in would enlarge the QLabel, but leave the pixmap at its original size, exposing the QLabel's background.

We make imageLabel the scroll area's child widget, and we make scrollArea the central widget of the QMainWindow. At the end we create the associated actions and menus, and customize the ImageViewer's appearance.

```
static void initializeImageFileDialog(QFileDialog &dialog, QFileDialog::AcceptMode acceptMode)
{
    static bool firstDialog = true;
```

```
    if (firstDialog) {
        firstDialog = false;
        const QStringList picturesLocations = QStandardPaths::standardLocations(QStandardPaths::PicturesLocation);
        dialog.setDirectory(picturesLocations.isEmpty() ? QDir::currentPath() : picturesLocations.last());
    }

    QStringList mimeTypeFilters;
    const QByteArrayList supportedMimeTypes = acceptMode == QFileDialog::AcceptOpen
        ? QImageReader::supportedMimeTypes() : QImageWriter::supportedMimeTypes();
    for (const QByteArray &mimeTypeName : supportedMimeTypes)
        mimeTypeFilters.append(mimeTypeName);
    mimeTypeFilters.sort();
    dialog.setMimeTypeFilters(mimeTypeFilters);
    dialog.selectMimeTypeFilter("image/jpeg");
    if (acceptMode == QFileDialog::AcceptSave)
        dialog.setDefaultSuffix("jpg");
}

void ImageViewer::open()
{
    QFileDialog dialog(this, tr("Open File"));
    initializeImageFileDialog(dialog, QFileDialog::AcceptOpen);

    while (dialog.exec() && !loadFile(dialog.selectedFiles().first())) {}
}
```

In the `open()` slot, we show a file dialog to the user. We compile a list of mime types for use as a filter by querying QImageReader for the available mime type names.

We show the file dialog until a valid file name is entered or the user cancels.

The function `loadFile()` is used to load the image.

```
bool ImageViewer::loadFile(const QString &fileName)
{
    QImageReader reader(fileName);
    reader.setAutoTransform(true);
    const QImage newImage = reader.read();
    if (newImage.isNull()) {
        QMessageBox::information(this, QGuiApplication::applicationDisplayName(),
                                 tr("Cannot load %1: %2")
                                 .arg(QDir::toNativeSeparators(fileName), reader.errorString()));
        return false;
    }
```

In the `loadFile()` function, we instantiate a QImageReader and enable automatic transformations by calling QImageReader::setAutoTransform(). For files in JPEG format, this ensures that portrait mode images of digital cameras are shown correctly by applying the appropriate orientation read from the EXIF meta data stored in the image file.

We then load the image using QImageReader::read(). If this returns a null image, indicating that the file is not an image file, we use a QMessageBox to alert the user.

The QMessageBox class provides a modal dialog with a short message, an icon, and some buttons. As with QFileDialog the easiest way to create a QMessageBox is to use its static convenience functions. QMessageBox provides a range of different messages arranged along two axes: severity (question, information, warning and critical) and complexity (the number of necessary response buttons). In this particular example an information message with an **OK** button (the default) is sufficient, since the message is part of a normal operation.

```
    scaleFactor = 1.0;

    scrollArea->setVisible(true);
    printAct->setEnabled(true);
    fitToWindowAct->setEnabled(true);
    updateActions();

    if (!fitToWindowAct->isChecked())
        imageLabel->adjustSize();
}
```

If the format is supported, we display the image in `imageLabel` by setting the label's pixmap. Then we enable the **Print** and **Fit to Window** menu entries and update the rest of the view menu entries. The **Open** and **Exit** entries are enabled by default.

If the **Fit to Window** option is turned off, the QScrollArea::widgetResizable property is `false` and it is our responsibility (not QScrollArea's) to give the QLabel a reasonable size based on its contents. We call {QWidget::adjustSize()}{adjustSize()} to achieve this, which is essentially the same as

```
imageLabel->resize(imageLabel->pixmap()->size());
```

In the print() slot, we first make sure that an image has been loaded into the application:

```
void ImageViewer::print()
{
    Q_ASSERT(imageLabel->pixmap());
#ifdef QT_CONFIG(printdialog)
```

If the application is built in debug mode, the Q_ASSERT() macro will expand to

```
if (!imageLabel->pixmap()) {
    qFatal("%s", "ASSERT: \"imageLabel->pixmap()\" in file ...");
```

In release mode, the macro simply disappear. The mode can be set in the application's .pro file. One way to do so is to add an option to **qmake** when building the application:

```
qmake "CONFIG += debug" foo.pro
```

or

```
qmake "CONFIG += release" foo.pro
```

Another approach is to add this line directly to the .pro file.

```
    QPrintDialog dialog(&printer, this);
    if (dialog.exec()) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = imageLabel->pixmap()->size();
        size.scale(rect.size(), Qt::KeepAspectRatio);
        painter.setViewport(rect.x(), rect.y(), size.width(), size.height());
        painter.setWindow(imageLabel->pixmap()->rect());
        painter.drawPixmap(0, 0, *imageLabel->pixmap());
    }
#endif
}
```

Then we present a print dialog allowing the user to choose a printer and to set a few options. We construct a painter with a QPrinter as the paint device. We set the painter's window and viewport in such a way that the image is as large as possible on the paper, but without altering its aspect ratio.

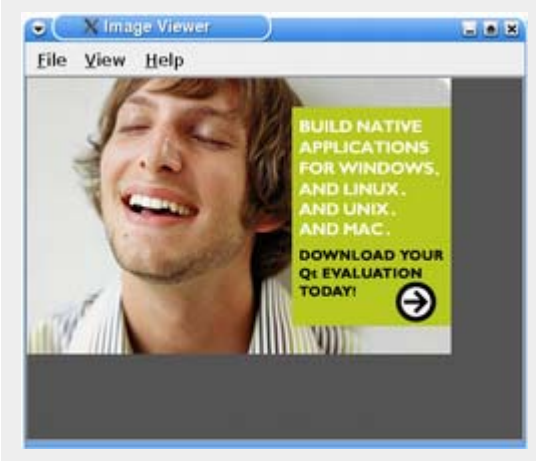
In the end we draw the pixmap at position (0, 0).

```
void ImageViewer::zoomIn()
{
    scaleImage(1.25);
}

void ImageViewer::zoomOut()
{
    scaleImage(0.8);
}
```

We implement the zooming slots using the private scaleImage() function. We set the scaling factors to 1.25 and 0.8, respectively. These factor values ensure that a **Zoom In** action and a **Zoom Out** action will cancel each other (since 1.25 * 0.8 == 1), and in that way the normal image size can be restored using the zooming features.

The screenshots below show an image in its normal size, and the same image after zooming in:




```
void ImageViewer::normalSize()
{
    imageLabel->adjustSize();
    scaleFactor = 1.0;
}
```

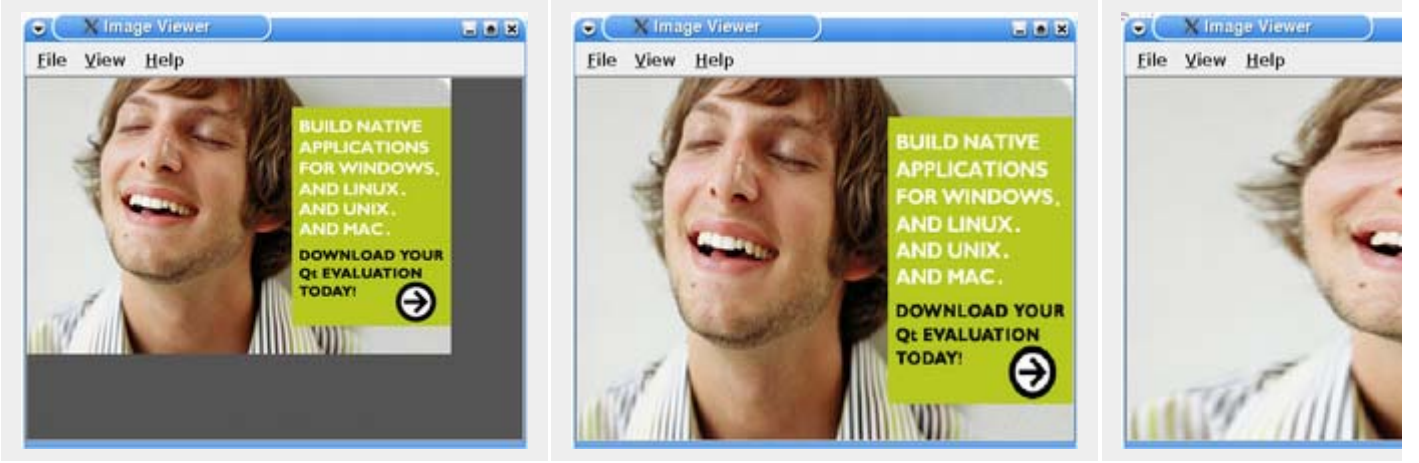
When zooming, we use the QLabel's ability to scale its contents. Such scaling doesn't change the actual size hint of the contents. And since the adjustSize() function use those size hint, the only thing we need to do to restore the normal size of the currently displayed image is to call adjustSize() and reset the scale factor to 1.0.

```
void ImageViewer::fitToWindow()
{
    bool fitToWindow = fitToWindowAct->isChecked();
    scrollArea->setWidgetResizable(fitToWindow);
    if (!fitToWindow)
        normalSize();
    updateActions();
}
```

The fitToWindow() slot is called each time the user toggled the **Fit to Window** option. If the slot is called to turn on the option, we tell the scroll area to resize its child widget with the QScrollArea::setWidgetResizable() function. Then we disable the **Zoom In**, **Zoom Out** and **Normal Size** menu entries using the private updateActions() function.

If the QScrollArea::widgetResizable property is set to false (the default), the scroll area honors the size of its child widget. If this property is set to true, the scroll area will automatically resize the widget in order to avoid scroll bars where they can be avoided, or to take advantage of extra space. But the scroll area will honor the minimum size hint of its child widget independent of the widget resizable property. So in this example we set imageLabel's size policy to ignored in the constructor, to avoid that scroll bars appear when the scroll area becomes smaller than the label's minimum size hint.

The screenshots below shows an image in its normal size, and the same image with the **Fit to window** option turned on. Enlarging the window will stretch the image further, as shown in the third screenshot.



If the slot is called to turn off the option, the {QScrollArea::setWidgetResizable} property is set to . We also restore the image pixmap to its normal size by adjusting the label's size to its content. And in the end we update the view menu entries.

```
void ImageViewer::about()
{
    QMessageBox::about(this, tr("About Image Viewer"),
            tr("<p>The <b>Image Viewer</b> example shows how to combine QLabel "
               "and QScrollArea to provide a custom image viewer. A QLabel may "
               "be displaying a text, but it can also display an image. "
               "QScrollArea provides a scrolling view around another widget. "
               "If the child widget exceeds the size of the frame, QScrollArea "
               "automatically provides scroll bars. </p><p>The example "
               "demonstrates how QLabel's ability to scale its contents "
               "(QLabel::scaledContents), and QScrollArea's ability to "
               "automatically resize its contents "
               "(QScrollArea::widgetResizable), can be used to implement "
               "zooming and scaling features. </p><p>In addition the example "
               "shows how to use QPainter to print an image.</p>"));
}
```

We implement the slot to create a message box describing what the example is designed to show.

```
void ImageViewer::createActions()
{
    QMenu *fileMenu = menuBar()->addMenu(tr("&File"));

    QAction *openAct = fileMenu->addAction(tr("&Open..."), this, &ImageViewer::open);
    openAct->setShortcut(QKeySequence::Open);

    saveAsAct = fileMenu->addAction(tr("Save &As..."), this, &ImageViewer::saveAs);
    saveAsAct->setEnabled(false);

    printAct = fileMenu->addAction(tr("&Print..."), this, &ImageViewer::print);
    printAct->setShortcut(QKeySequence::Print);
    printAct->setEnabled(false);

    fileMenu->addSeparator();

    QAction *exitAct = fileMenu->addAction(tr("E&xit"), this, &QWidget::close);
    exitAct->setShortcut(tr("Ctrl+Q"));

    QMenu *editMenu = menuBar()->addMenu(tr("&Edit"));

    copyAct = editMenu->addAction(tr("&Copy"), this, &ImageViewer::copy);
    copyAct->setShortcut(QKeySequence::Copy);
    copyAct->setEnabled(false);

    QAction *pasteAct = editMenu->addAction(tr("&Paste"), this, &ImageViewer::paste);
    pasteAct->setShortcut(QKeySequence::Paste);

    QMenu *viewMenu = menuBar()->addMenu(tr("&View"));

    zoomInAct = viewMenu->addAction(tr("Zoom &In (25%)"), this, &ImageViewer::zoomIn);
    zoomInAct->setShortcut(QKeySequence::ZoomIn);
    zoomInAct->setEnabled(false);

    zoomOutAct = viewMenu->addAction(tr("Zoom &Out (25%)"), this, &ImageViewer::zoomOut);
    zoomOutAct->setShortcut(QKeySequence::ZoomOut);
    zoomOutAct->setEnabled(false);

    normalSizeAct = viewMenu->addAction(tr("&Normal Size"), this, &ImageViewer::normalSize);
    normalSizeAct->setShortcut(tr("Ctrl+S"));
    normalSizeAct->setEnabled(false);

    viewMenu->addSeparator();

    fitToWindowAct = viewMenu->addAction(tr("&Fit to Window"), this, &ImageViewer::fitToWindow);
    fitToWindowAct->setEnabled(false);
    fitToWindowAct->setCheckable(true);
    fitToWindowAct->setShortcut(tr("Ctrl+F"));

    QMenu *helpMenu = menuBar()->addMenu(tr("&Help"));

    helpMenu->addAction(tr("&About"), this, &ImageViewer::about);
    helpMenu->addAction(tr("About &Qt"), &QApplication::aboutQt);
}
```

In the private function, we create the actions providing the application features and populate a menu with them.

We assign a short-cut key to each action and connect them to the appropriate slots. We only enable the and at the time of creation, the others are updated once an image has been loaded into the application. In addition we make the checkable.

The QMenu class provides a menu widget for use in menu bars, context menus, and other popup menus. The QMenuBar class provides a horizontal menu bar that consists of a list of pull-down menu items. So we put the menus in the 's menu bar which we retrieve with the QMainWindow::menuBar() function.

```
void ImageViewer::updateActions()
{
    saveAsAct->setEnabled(!image.isNull());
    copyAct->setEnabled(!image.isNull());
    zoomInAct->setEnabled(!fitToWindowAct->isChecked());
    zoomOutAct->setEnabled(!fitToWindowAct->isChecked());
    normalSizeAct->setEnabled(!fitToWindowAct->isChecked());
}
```

The private function enables or disables the **Zoom In**, **Zoom Out** and

Normal Size menu entries depending on whether the **Fit to Window** option is turned on or off.

```
void ImageViewer::scaleImage(double factor)
{
    Q_ASSERT(imageLabel->pixmap());
    scaleFactor *= factor;
    imageLabel->resize(scaleFactor * imageLabel->pixmap()->size());

    adjustScrollBar(scrollArea->horizontalScrollBar(), factor);
    adjustScrollBar(scrollArea->verticalScrollBar(), factor);

    zoomInAct->setEnabled(scaleFactor < 3.0);
    zoomOutAct->setEnabled(scaleFactor > 0.333);
}
```

In `scaleImage()`, we use the `factor` parameter to calculate the new scaling factor for the displayed image, and resize `imageLabel`. Since we set the scaledContents property to `true` in the constructor, the call to QWidget::resize() will scale the image displayed in the label. We also adjust the scroll bars to preserve the focal point of the image.

At the end, if the scale factor is less than 33.3% or greater than 300%, we disable the respective menu entry to prevent the image pixmap from becoming too large, consuming too much resources in the window system.

```
void ImageViewer::adjustScrollBar(QScrollBar *scrollBar, double factor)
{
    scrollBar->setValue(int(factor * scrollBar->value()
                            + ((factor - 1) * scrollBar->pageStep()/2)));
}
```

Whenever we zoom in or out, we need to adjust the scroll bars in consequence. It would have been tempting to simply call

```
scrollBar->setValue(int(factor * scrollBar->value()));
```

but this would make the top-left corner the focal point, not the center. Therefore we need to take into account the scroll bar handle's size (the page step).

Files:

- widgets/imageviewer/imageviewer.cpp
- widgets/imageviewer/imageviewer.h
- widgets/imageviewer/main.cpp
- widgets/imageviewer/imageviewer.pro