10. Item View Classes

Many applications let the user search, view, and edit individual items that belong to a data set. The data might be held in files or accessed from a database or a network server. The standard approach to dealing with data sets like this is to use Qt's item view classes.

In earlier versions of Qt, the item view widgets were populated with the entire contents of a data set; the users would perform all their searches and edits on the data held in the widget, and at some point the changes would be written back to the data source. Although simple to understand and use, this approach doesn’t scale well to very large data sets and doesn’t lend itself to situations where we want to display the same data set in two or more different widgets.

The Smalltalk language popularized a flexible approach to visualizing large data sets: model–view–controller (MVC). In the MVC approach, the model represents the data set and is responsible for fetching the data that is needed for viewing and for writing back any changes. Each type of data set has its own model, but the API that the models provide to the views is uniform no matter what the underlying data set. The view presents the data to the user. With any large data set only a limited amount of data will be visible at any one time, so that is the only data that the view asks for. The controller mediates between the user and the view, converting user actions into requests to navigate or edit data, which the view then transmits to the model as necessary.

Qt provides a model/view architecture inspired by the MVC approach. In Qt, the model behaves the same as it does for classic MVC. But instead of a controller, Qt uses a slightly different abstraction: the delegate. The delegate
is used to provide fine control over how items are rendered and edited. Qt provides a default delegate for every type of view. This is sufficient for most applications, so we usually don’t need to care about it.

Using Qt’s model/view architecture, we can use models that only fetch the data that is actually needed for display in the view. This makes handling very large data sets much faster and less memory hungry than reading all the data. And by registering a model with two or more views, we can give the user the opportunity of viewing and interacting with the data in different ways, with little overhead. Qt automatically keeps multiple views in sync, reflecting changes to one in all the others. An additional benefit of the model/view architecture is that if we decide to change how the underlying data set is stored, we just need to change the model; the views will continue to behave correctly.

In many situations, we only need to present relatively small numbers of items to the user. In these common cases, we can use Qt’s convenience item view classes (QListWidget, QTableWidget, and QTreeWidget) and populate them with items directly. These classes behave in a similar way to the item view classes provided by earlier versions of Qt. They store their data in “items” (for example, a QTableWidget contains QTableWidgetItem). Internally, the convenience classes use custom models that make the items visible to the views.

For large data sets, duplicating the data is often not an option. In these cases, we can use Qt’s views (QListView, QTableView, and QTreeView), in conjunction with a data model, which can be a custom model or one of Qt’s predefined models. For example, if the data set is held in a database, we can combine a QTableView with a QSqlTableModel.

**Using the Item View Convenience Classes**

Using Qt’s item view convenience subclasses is usually simpler than defining a custom model and is appropriate when we don’t need the benefits of separating the model and the view. We used this technique in Chapter 4 when we subclassed QTableWidget and QTableWidgetItem to implement spreadsheet functionality.
In this section, we will show how to use the convenience item view subclasses to display items. The first example shows a read-only QListWidget, the second example shows an editable QTableWidget, and the third example shows a read-only QTreeWidget.

We begin with a simple dialog that lets the user pick a flowchart symbol from a list. Each item consists of an icon, a text, and a unique ID.

![Figure 10.3. The Flowchart Symbol Picker application](image)

Let’s start with an extract from the dialog’s header file:

```cpp
class FlowChartSymbolPicker : public QDialog
{
    Q_OBJECT

public:
    FlowChartSymbolPicker(const QMap<int, QString> &symbolMap, QWidget *parent = 0);
    int selectedId() const { return id; }
    void done(int result);
    ...
};
```

When we construct the dialog, we must pass it a QMap<int, QString>, and after it has executed we can retrieve the chosen ID (or -1 if the user didn’t select any item) by calling selectedId().

```cpp
FlowChartSymbolPicker::FlowChartSymbolPicker(
    const QMap<int, QString> &symbolMap, QWidget *parent)
{
    id = -1;
    listWidget = new QListWidget;
    listWidget->setIconSize(QSize(60, 60));
    QMapIterator<int, QString> i(symbolMap);
```
while (i.hasNext()) {
    i.next();
    QListWidgetItem *item = new QListWidgetItem(i.value(),
        listWidget);
    item->setIcon(iconForSymbol(i.value()));
    item->setData(Qt::UserRole, i.key());
}

We initialize `id` (the last selected ID) to -1. Next we construct a `QListWidget`, a convenience item view widget. We iterate over each item in the flowchart symbol map and create a `QListWidgetItem` to represent each one. The `QListWidgetItem` constructor takes a `QString` that represents the text to display, followed by the parent `QListWidget`.

Then we set the item’s icon and we call `setData()` to store our arbitrary ID in the `QListWidgetItem`. The `iconForSymbol()` private function returns a `QIcon` for a given symbol name.

`QListWidgetItem`’s have several roles, each of which has an associated `QVariant`. The most common roles are `Qt::DisplayRole`, `Qt::EditRole`, and `Qt::IconRole`, and for these there are convenience setter and getter functions (`setText()`, `setIcon()`), but there are several other roles. We can also define custom roles by specifying a numeric value of `Qt::UserRole` or higher. In our example, we use `Qt::UserRole` to store each item’s ID.

The omitted part of the constructor is concerned with creating the buttons, laying out the widgets, and setting the window’s title.

```
void FlowChartSymbolPicker::done(int result)
{
    id = -1;
    if (result == QDialog::Accepted) {
        QListWidgetItem *item = listWidget->currentItem();
        if (item)
            id = item->data(Qt::UserRole).toInt();
    }
    QDialog::done(result);
}
```

The `done()` function is reimplemented from `QDialog`. It is called when the user presses OK or Cancel. If the user clicked OK, we retrieve the relevant item and extract the ID using the `data()` function. If we were interested in the item’s text, we could retrieve it by calling `item->data(Qt::DisplayRole).toString()` or more conveniently, `item->text()`.

By default, `QListWidget` is read-only. If we wanted the user to edit the items, we could set the view’s edit triggers using `QAbstractItemView::setEditTriggers()`, for example, a setting of `QAbstractItemView::AnyKeyPressed` means that the user can begin editing an item just by starting to type. Alternatively, we could provide an Edit button (and perhaps Add and Delete buttons) and connect them to slots so that we could handle the editing operations programmatically.
Now that we have seen how to use a convenience item view class for viewing and selecting data, we will look at an example where we can edit data. Again we are using a dialog, this time one that presents a set of \((x, y)\) coordinates that the user can edit.

![Coordinate Setter application](image)

**Figure 10.4.** The Coordinate Setter application

As with the previous example, we will focus on the item view relevant code, starting with the constructor.

```cpp
CoordinateSetter::CoordinateSetter(QList<QPointF> *coords,
                                 QWidget *parent)
                             : QDialog(parent)
                           {
    coordinates = coords;

    tableWidget = new QTableWidget(0, 2);
    tableWidget->setHorizontalHeaderLabels(
        QStringList() << tr("X") << tr("Y"));

    for (int row = 0; row < coordinates->count(); ++row) {
        QPointF point = coordinates->at(row);
        addRow();
        tableWidget->item(row, 0)->setText(QString::number(point.x()));
        tableWidget->item(row, 1)->setText(QString::number(point.y()));
    }
}
```

The QTableWidget constructor takes the initial number of table rows and columns to display. Every item in a QTableWidget is represented by a QTableWidgetItem, including horizontal and vertical header items. The setHorizontalHeaderLabels() function sets the text for each horizontal table widget item to the corresponding text in the string list it is passed. By default, QTableWidget provides a vertical header with rows labeled from 1, which is exactly what we want, so we don’t need to set the vertical header labels manually.
Once we have created and centered the column labels, we iterate through the coordinate data that was passed in. For every \((x, y)\) pair, we create two QTableWidgetItem corresponding to the \(x\) and \(y\) coordinates. The items are added to the table using QTableWidget::setItem(), which takes a row and a column in addition to the item.

By default, QTableWidget allows editing. The user can edit any cell in the table by navigating to it and then either pressing F2 or simply by typing. All changes made by the user in the view will be automatically reflected into the QTableWidgetItem. To prevent editing, we can call setEditTriggers(QAbstractItemView::NoEditTriggers).

```cpp
void CoordinateSetter::addRow()
{
    int row = tableWidget->rowCount();
    tableWidget->insertRow(row);
    QTableWidgetItem *item0 = new QTableWidgetItem;
    item0->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 0, item0);

    QTableWidgetItem *item1 = new QTableWidgetItem;
    item1->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 1, item1);
    tableWidget->setCurrentItem(item0);
}
```

The addRow() slot is invoked when the user clicks the Add Row button. We append a new row using insertRow(). If the user attempts to edit a cell in the new row, the QTableWidget will automatically create a new QTableWidgetItem.

```cpp
void CoordinateSetter::done(int result)
{
    if (result == QDialog::Accepted) {
        coordinates->clear();
        for (int row = 0; row < tableWidget->rowCount(); ++row) {
            double x = tableWidget->item(row, 0)->text().toDouble();
            double y = tableWidget->item(row, 1)->text().toDouble();
            coordinates->append(QPointF(x, y));
        }
    }
    QDialog::done(result);
}
```

Finally, when the user clicks OK, we clear the coordinates that were passed in to the dialog, and create a new set based on the coordinates in the QTableWidget’s items.

For our third and final example of Qt’s convenience item view widgets, we will look at some snippets from an application that shows Qt application settings using a QTreeWidget. Read-only is the default for QTreeWidget.
Here's an extract from the constructor:

```cpp
SettingsViewer::SettingsViewer(QWidget *parent)
    : QDialog(parent)
{
    organization = "Trolltech";
    application = "Designer";

    treeWidget = new QTreeWidget;
    treeWidget->setColumnCount(2);
    treeWidget->setHeaderLabels(
        QStringList() << tr("Key") << tr("Value"));
    treeWidget->header()->setResizeMode(0, QHeaderView::Stretch);
    treeWidget->header()->setResizeMode(1, QHeaderView::Stretch);
    ... 
    setWindowTitle(tr("Settings Viewer"));
    readSettings();
}
```

To access an application’s settings, a `QSettings` object must be created with the organization’s name and the application’s name as parameters. We set default names (“Designer” by “Trolltech”) and then construct a new `QTreeWidget`. At the end, we call the `readSettings()` function.

```cpp
void SettingsViewer::readSettings()
{
    QSettings settings(organization, application);
    treeWidget->clear();
    addChildSettings(settings, 0, "");
    treeWidget->sortByColumn(0);
    treeWidget->setFocus();
    setWindowTitle(tr("Settings Viewer - %1 by %2")
        .arg(application).arg(organization));
}
```
Application settings are stored in a hierarchy of keys and values. The addChildSettings() private function takes a settings object, a parent QTreeWidgetItem, and the current "group". A group is the QSettings equivalent of a file system directory. The addChildSettings() function can call itself recursively to traverse an arbitrary tree structure. The initial call from the readSettings() function passes 0 as the parent item to represent the root.

```cpp
void SettingsViewer::addChildSettings(QSettings &settings,
                                      QTreeWidgetItem *parent, const QString &group)
{
    QTreeWidgetItem *item;

    settings.beginGroup(group);

    foreach (QString key, settings.childKeys()) {
        if (parent) {
            item = new QTreeWidgetItem(parent);
        } else {
            item = new QTreeWidgetItem(treeWidget);
        }
        item->setText(0, key);
        item->setText(1, settings.value(key).toString());
    }

    foreach (QString group, settings.childGroups()) {
        if (parent) {
            item = new QTreeWidgetItem(parent);
        } else {
            item = new QTreeWidgetItem(treeWidget);
        }
        item->setText(0, group);
        addChildSettings(settings, item, group);
    }
    settings.endGroup();
}
```

The addChildSettings() function is used to create all the QTreeWidgetItem objects. It iterates over all the keys at the current level in the settings hierarchy and creates one QTableWidgetItem per key. If 0 was passed as the parent item, we create the item as a child of the QTreeWidgetItem itself (making it a top-level item); otherwise, we create the item as a child of parent. The first column is set to the name of the key and the second column to the corresponding value.

Next, the function iterates over every group at the current level. For each group, a new QTreeWidgetItem is created with its first column set to the group's name. The function then calls itself recursively with the group item as the parent to populate the QTreeWidgetItem with the group's child items.

The item view widgets shown in this section allow us to use a style of programming that is very similar to that used in earlier versions of Qt: reading an entire data set into an item view widget, using item objects to represent data elements, and (if the items are editable) writing back to the data source. In the following sections, we will go beyond this simple approach and take full advantage of Qt's model/view architecture.
Using Predefined Models

Qt provides several predefined models for use with the view classes:

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QStringListModel</td>
<td>Stores a list of strings</td>
</tr>
<tr>
<td>QStandardItemModel</td>
<td>Stores arbitrary hierarchical data</td>
</tr>
<tr>
<td>QDirModel</td>
<td>Encapsulates the local file system</td>
</tr>
<tr>
<td>QSqlQueryModel</td>
<td>Encapsulates an SQL result set</td>
</tr>
<tr>
<td>QSqlTableModel</td>
<td>Encapsulates an SQL table</td>
</tr>
<tr>
<td>QSqlRelationalTableModel</td>
<td>Encapsulates an SQL table with foreign keys</td>
</tr>
<tr>
<td>QSortFilterProxyModel</td>
<td>Sorts and/or filters another model</td>
</tr>
</tbody>
</table>

In this section, we will look at how to use the QStringListModel, the QDirModel, and the QSortFilterProxyModel. The SQL models are covered in Chapter 13.

Let’s begin with a simple dialog that users can use to add, delete, and edit a QStringList, where each string represents a team leader.

![Figure 10.6. The Team Leaders application](image)

Here’s the relevant extract from the constructor:

```cpp
TeamLeadersDialog::Team LeadersDialog(const QStringList &leaders,
                                        QWidget *parent)
    : QDialog(parent)
{   
    model = new QStringListModel(this);
    model->setStringList(leaders);

    listView = new QListView;
    listView->setModel(model);
    listView->setEditTriggers(QAbstractItemView::AnyKeyPressed |
                                QAbstractItemView::DoubleClicked);

    ...  
}
```
We begin by creating and populating a QStringListModel. Next we create a QListView and set its model to the one we have just created. We also set some editing triggers to allow the user to edit a string simply by starting to type on it or by double-clicking it. By default, no editing triggers are set on a QListView, making the view effectively read-only.

```cpp
void TeamLeadersDialog::insert()
{
    int row = listView->currentIndex().row();
    model->insertRows(row, 1);
    QModelIndex index = model->index(row);
    listView->setCurrentIndex(index);
    listView->edit(index);
}
```

When the user clicks the Insert button, the insert() slot is invoked. The slot begins by retrieving the row number for the list view's current item. Every data item in a model has a corresponding “model index”, which is represented by a QModelIndex object. We will look at model indexes in detail in the next section, but for now it is sufficient to know that an index has three main components: a row, a column, and a pointer to the model to which it belongs. For a one-dimensional list model, the column is always 0.

Once we have the row number, we insert one new row at that position. The insertion is performed on the model, and the model automatically updates the list view. We then set the list view's current index to the blank row we just inserted. Finally, we set the list view to editing mode on the new row, just as if the user had pressed a key or double-clicked to initiate editing.

```cpp
void TeamLeadersDialog::del()
{
    model->removeRows(listView->currentIndex().row(), 1);
}
```

In the constructor, the Delete button's clicked() signal is connected to the del() slot. Since we are just deleting the current row, we can call removeRows() with the current index position and a row count of 1. Just like with insertion, we rely on the model to update the view accordingly.

```cpp
QStringList TeamLeadersDialog::leaders() const
{
    return model->stringList();
}
```

Finally, the leaders() function provides a means of reading back the edited strings when the dialog is closed.

TeamLeadersDialog could be made into a generic string list editing dialog simply by parameterizing its window title. Another generic dialog that is often required is one that presents a list of files or directories to the user. The next example uses the QDirModel class, which encapsulates the computer's file system and is capable of showing (and hiding) various file attributes. This model
can apply a filter to restrict the kinds of file system entries that are shown and can order the entries in various ways.

**Figure 10.7.** The Directory Viewer application

We will begin by looking at the creation and setting up of the model and the view in the Directory Viewer dialog's constructor.

```cpp
DirectoryViewer::DirectoryViewer(QWidget *parent)
: QDialog(parent)
{
    model = new QDirModel;
    model->setReadOnly(false);
    model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);

    treeView = new QTreeView;
    treeView->setModel(model);
    treeView->header()->setStretchLastSection(true);
    treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
    treeView->header()->setSortIndicatorShown(true);
    treeView->header()->setClickable(true);

    QModelIndex index = model->index(QDir::currentPath());
    treeView->expand(index);
    treeView->scrollTo(index);
    treeView->resizeColumnToContents(0);

    model->sort(0, Qt::AscendingOrder);
    model->setFilter(QDir::AllEntries | QDir::Readable | QDir::Executable | QDir::Hidden | QDir::System | QDir::NoDotAndDotDot);
}
```

Once the model has been constructed, we make it editable and set various initial sort ordering attributes. We then create the QTreeView that will display the model's data. The QTreeView's header can be used to provide user-controlled sorting. By making the header clickable, the user can sort by whichever column header they click, with repeated clicks alternating between ascending and descending orders. Once the tree view's header has been set up, we get the model index of the current directory and make sure that this directory is visible by expanding its parents if necessary using expand(), and scrolling to
it using `scrollTo()`. Then we make sure that the first column is wide enough to show all its entries without using ellipses (...).

In the part of the constructor code that isn’t shown here, we connected the Create Directory and Remove buttons to slots to perform these actions. We do not need a Rename button since users can rename in-place by pressing F2 and typing.

```cpp
void DirectoryViewer::createDirectory()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;

    QString dirName = QInputDialog::getText(this,
        tr("Create Directory"),
        tr("Directory name"));
    if (!dirName.isEmpty()) {
        if (!model->mkdir(index, dirName).isValid())
            QMessageBox::information(this, tr("Create Directory"),
                tr("Failed to create the directory"));
    }
}
```

If the user enters a directory name in the input dialog, we attempt to create a directory with this name as a child of the current directory. The `QDirModel::mkdir()` function takes the parent directory’s index and the name of the new directory, and returns the model index of the directory it created. If the operation fails, it returns an invalid model index.

```cpp
void DirectoryViewer::remove()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;

    bool ok;
    if (model->fileInfo(index).isDir()) {
        ok = model->rmdir(index);
    } else {
        ok = model->remove(index);
    }
    if (!ok)
        QMessageBox::information(this, tr("Remove"),
            tr("Failed to remove %1").arg(model->fileName(index)));
}
```

If the user clicks Remove, we attempt to remove the file or directory associated with the current item. We could use `QDir` to accomplish that, but `QDirModel` offers convenience functions that work on `QModelIndex`es.

The last example in this section shows how to use `QSortFilterProxyModel`. Unlike the other predefined models, this model encapsulates an existing model and manipulates the data that passes between the underlying model and the
view. In our example, the underlying model is a `QStringListModel` initialized with the list of color names recognized by Qt (obtained through `QColor::colorNames()`). The user can type a filter string in a `QLineEdit` and specify how this string is to be interpreted (as a regular expression, a wildcard pattern, or a fixed string) using a combobox.

![Figure 10.8. The Color Names application](image)

Here’s an extract from the `ColorNamesDialog` constructor:

```cpp
ColorNamesDialog::ColorNamesDialog(QWidget *parent) :
    QDialog(parent)
{
    sourceModel = new QStringListModel(this);
    sourceModel->setStringList(QColor::colorNames());
    proxyModel = new QSortFilterProxyModel(this);
    proxyModel->setSourceModel(sourceModel);
    proxyModel->setFilterKeyColumn(0);
    listView = new QListView;
    listView->setModel(proxyModel);
    ...
    syntaxComboBox = new QComboBox;
    syntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);
    syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
    syntaxComboBox->addItem(tr("Fixed string"), QRegExp::FixedString);
    ...
}
```

The `QStringListModel` is created and populated in the usual way. This is followed by the construction of the `QSortFilterProxyModel`. We pass the underlying model using `setSourceModel()` and tell the proxy to filter based on column 0 of the original model. The `QComboBox::addItem()` function accepts an optional “data” argument of type `QVariant`; we use this to store the `QRegExp::PatternSyntax` value that corresponds to each item’s text.
void ColorNamesDialog::reapplyFilter()
{
    QRegExp::PatternSyntax syntax =
        QRegExp::PatternSyntax(syntaxComboBox->itemData(
            syntaxComboBox->currentIndex()).toInt());
    QRegExp regExp(filterLineEdit->text(), Qt::CaseInsensitive, syntax);
    proxyModel->setFilterRegExp(regExp);
}

The reapplyFilter() slot is invoked whenever the user changes the filter string or the pattern syntax combobox. We create a QRegExp using the text in the line edit. Then we set its pattern syntax to the one stored in the syntax combobox’s current item’s data. When we call setFilterRegExp(), the new filter becomes active and the view is automatically updated.

**Implementing Custom Models**

Qt’s predefined models offer a convenient means of handling and viewing data. However, some data sources cannot be used efficiently using the predefined models, and for these situations it is necessary to create custom models optimized for the underlying data source.

Before we embark on creating custom models, let’s first review the key concepts used in Qt’s model/view architecture. Every data element in a model has a model index and a set of attributes, called roles, that can take arbitrary values. We saw earlier in the chapter that the most commonly used roles are Qt::DisplayRole and Qt::EditRole. Other roles are used for supplementary data (for example, Qt::ToolTipRole, Qt::StatusTipRole, and Qt::WhatsThisRole), and yet others for controlling basic display attributes (such as Qt::FontRole, Qt::TextAlignmentRole, Qt::TextColorRole, and Qt::BackgroundColorRole).

![Schematic view of Qt’s models](image)

**Figure 10.9.** Schematic view of Qt’s models
For a list model, the only relevant index component is the row number, accessible from `QModelIndex::row()`. For a table model, the relevant index components are the row and column numbers, accessible from `QModelIndex::row()` and `QModelIndex::column()`. For both list and table models, every item’s parent is the root, which is represented by an invalid `QModelIndex`. The first two examples in this section show how to implement custom table models.

A tree model is similar to a table model, with the following differences. Like a table model, the parent of top-level items is the root (an invalid `QModelIndex`), but every other item’s parent is some other item in the hierarchy. Parents are accessible from `QModelIndex::parent()`. Every item has its role data, and zero or more children, each an item in its own right. Since items can have other items as children, it is possible to represent recursive (tree-like) data structures, as the final example in this section will show.

The first example in this section is a read-only table model that shows currency values in relation to each other.

![Figure 10.10. The Currencies application](image)

The application could be implemented using a simple table, but we want to use a custom model to take advantage of certain properties of the data to minimize storage. If we were to store the 162 currently traded currencies in a table, we would need to store 162 × 162 = 26,244 values; with the custom model presented below, we only need to store 162 values (the value of each currency in relation to the U.S. dollar).

The `CurrencyModel` class will be used with a standard `QTableView`. The `CurrencyModel` is populated with a `QMap<QString, double>`; each key is a currency code and each value is the value of the currency in U.S. dollars. Here’s a code snippet that shows how the map is populated and how the model is used:

```cpp
QMap<QString, double> currencyMap;
currencyMap.insert("AUD", 1.3259);
currencyMap.insert("CHF", 1.2970);
... 
currencyMap.insert("SGD", 1.6901);
currencyMap.insert("USD", 1.0000);
```
Now we can look at the implementation of the model, starting with its header:

```cpp
class CurrencyModel : public QAbstractTableModel {
public:
    CurrencyModel(QObject *parent = 0);
    void setCurrencyMap(const QMap<QString, double> &map);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role) const;

private:
    QString currencyAt(int offset) const;
    QMap<QString, double> currencyMap;
};
```

We have chosen to subclass `QAbstractTableModel` for our model since that most closely matches our data source. Qt provides several model base classes, including `QAbstractListModel`, `QAbstractTableModel`, and `QAbstractItemModel`. The `QAbstractItemModel` class is used to support a wide variety of models, including those that are based on recursive data structures, while the `QAbstractListModel` and `QAbstractTableModel` classes are provided for convenience when using one-dimensional or two-dimensional data sets.

```
QObject
    \|-- QAbstractItemModel
    \   \-- QAbstractListModel
         \-- QAbstractTableModel
```

**Figure 10.11.** Inheritance tree for the abstract model classes

For a read-only table model, we must reimplement three functions: `rowCount()`, `columnCount()`, and `data()`. In this case, we have also reimplemented `headerData()`, and we provide a function to initialize the data (`setCurrencyMap()`).
Implementing Custom Models

We do not need to do anything in the constructor, except pass the parent parameter to the base class.

```cpp
int CurrencyModel::rowCount(const QModelIndex & /* parent */) const
{
    return currencyMap.count();
}

int CurrencyModel::columnCount(const QModelIndex & /* parent */) const
{
    return currencyMap.count();
}
```

For this table model, the row and column counts are the number of currencies in the currency map. The parent parameter has no meaning for a table model; it is there because rowCount() and columnCount() are inherited from the more generic QAbstractItemModel base class, which supports hierarchies.

```cpp
QVariant CurrencyModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        QString rowCurrency = currencyAt(index.row());
        QString columnCurrency = currencyAt(index.column());

        if (currencyMap.value(rowCurrency) == 0.0)
            return "####";

        double amount = currencyMap.value(columnCurrency) / currencyMap.value(rowCurrency);

        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}
```

The data() function returns the value of any of an item’s roles. The item is specified as a QModelIndex. For a table model, the interesting components of a QModelIndex are its row and column number, available using row() and column().

If the role is Qt::TextAlignmentRole, we return an alignment suitable for numbers. If the display role is Qt::DisplayRole, we look up the value for each currency and calculate the exchange rate.

We could return the calculated value as a double, but then we would have no control over how many decimal places were shown (unless we use a custom delegate). Instead, we return the value as a string, formatted as we want.
10. Item View Classes

```cpp
QVariant CurrencyModel::headerData(int section,
                                    Qt::Orientation /* orientation */,
                                    int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    return currencyAt(section);
}
```

The `headerData()` function is called by the view to populate its horizontal and vertical headers. The `section` parameter is the row or column number (depending on the orientation). Since the rows and columns have the same currency codes, we do not care about the orientation and simply return the code of the currency for the given section number.

```cpp
void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
{
    currencyMap = map;
    reset();
}
```

The caller can change the currency map using `setCurrencyMap()`. The `QAbstractItemModel::reset()` call tells any views that are using the model that all their data is invalid; this forces them to request fresh data for the items that are visible.

```cpp
QString CurrencyModel::currencyAt(int offset) const
{
    return (currencyMap.begin() + offset).key();
}
```

The `currencyAt()` function returns the key (the currency code) at the given offset in the currency map. We use an STL-style iterator to find the item and call `key()` on it.

As we have just seen, it is not difficult to create read-only models, and depending on the nature of the underlying data, there are potential savings in memory and speed with a well-designed model. The next example, the Cities application, is also table-based, but this time all the data is entered by the user.

This application is used to store values indicating the distance between any two cities. Like the previous example, we could simply use a `QTableWidget` and store one item for every city pair. However, a custom model could be more efficient, because the distance from any city \( A \) to any different city \( B \) is the same whether traveling from \( A \) to \( B \) or from \( B \) to \( A \), so the items are mirrored along the main diagonal.

To see how a custom model compares with a simple table, let us assume that we have three cities, \( A \), \( B \), and \( C \). If we store a value for every combination, we would need to store nine values. A carefully designed model would require only the three items \((A, B), (A, C),\) and \((B, C)\).
Here’s how we set up and use the model:

```cpp
QStringList cities;
cities << "Arvika" << "Boden" << "Eskilstuna" << "Falun"
<< "Filipstad" << "Halmstad" << "Helsingborg" << "Karlstad"
<< "Kiruna" << "Kramfors" << "Motala" << "Sandviken"
<< "Skara" << "Stockholm" << "Sundsvall" << "Trelleborg";

CityModel cityModel;
cityModel.setCities(cities);

QTableView tableView;
tableView.setModel(&cityModel);
tableView.setAlternatingRowColors(true);
```

We must reimplement the same functions as we did for the previous example. In addition, we must also reimplement `setData()` and `flags()` to make the model editable. Here is the class definition:

```cpp
class CityModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    CityModel(QObject *parent = 0);

    void setCities(const QStringList &cityNames);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    bool setData(const QModelIndex &index, const QVariant &value, int role);
    QVariant headerData(int section, Qt::Orientation orientation, int role) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;

private:
    int offsetOf(int row, int column) const;
    QStringList cities;
    QVector<int> distances;
};
```
10. Item View Classes

For this model, we are using two data structures: cities of type QStringList to hold the city names, and distances of type QVector<int> to hold the distance between each unique pair of cities.

```c++
CityModel::CityModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

The constructor does nothing beyond pass on the parent parameter to the base class.

```c++
int CityModel::rowCount(const QModelIndex & /* parent */) const
{
    return cities.count();
}
```

```c++
int CityModel::columnCount(const QModelIndex & /* parent */) const
{
    return cities.count();
}
```

Since we have a square grid of cities, the number of rows and columns is the number of cities in our list.

```c++
QVariant CityModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        if (index.row() == index.column())
            return 0;
        int offset = offsetOf(index.row(), index.column());
        return distances[offset];
    }
    return QVariant();
}
```

The `data()` function is similar to what we did in `CurrencyModel`. It returns 0 if the row and column are the same, because that corresponds to the case where the two cities are the same; otherwise, it finds the entry for the given row and column in the distances vector and returns the distance for that particular pair of cities.

```c++
QVariant CityModel::headerData(int section,
    Qt::Orientation /* orientation */,
    int role) const
{
    if (role == Qt::DisplayRole)
        return cities[section];
    return QVariant();
}
```
The headerData() function is simple because we have a square table with every row having an identical column header. We simply return the name of the city at the given offset in the cities string list.

```cpp
bool CityModel::setData(const QModelIndex &index,
                        const QVariant &value, int role)
{
    if (index.isValid() && index.row() != index.column()
        && role == Qt::EditRole) {
        int offset = offsetOf(index.row(), index.column());
        distances[offset] = value.toInt();

        QModelIndex transposedIndex = createIndex(index.column(),
                                                 index.row());

        emit dataChanged(index, index);
        emit dataChanged(transposedIndex, transposedIndex);
        return true;
    }
    return false;
}
```

The setData() function is called when the user edits an item. Providing the model index is valid, the two cities are different, and the data element to modify is the Qt::EditRole, the function stores the value the user entered in the distances vector.

```
QModelIndex CityModel::fromItem(const QVariant & data)
{
    for (int i = 0; i < cities.size(); ++i)
        if (distances[i] == data.toInt())
            return createIndex(i, i);
    return QAbstractItemModel::createIndex(-1, -1);
}
```

The createIndex() function is used to generate a model index. We need it to get the model index of the item on the other side of the main diagonal that corresponds with the item being set, since both items must show the same data. The createIndex() function takes the row before the column; here we invert the parameters to get the model index of the diagonally opposite item to the one specified by index.

We emit the dataChanged() signal with the model index of the item that was changed. The reason this signal takes two model indexes is that it is possible for a change to affect a rectangular region of more than one row and column, so the indexes passed are the index of the top left and bottom right items of those that have changed. We also emit the dataChanged() signal for the transposed index to ensure that the view will refresh the item. Finally, we return true or false to indicate whether or not the edit succeeded.

```
Qt::ItemFlags CityModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags flags = QAbstractItemModel::flags(index);
    if (index.row() != index.column())
        flags |= Qt::ItemIsEditable;
    return flags;
}
```

The flags() function is used by the model to communicate what can be done with an item (for example, whether it is editable). The default implementation from QAbstractTableModel returns Qt::ItemIsSelectable | Qt::ItemIsEnabled. We
add the Qt::ItemIsEditable flag for all items except those lying on the diagonals
(which are always 0).

```cpp
void CityModel::setCities(const QStringList &cityNames)
{
    cities = cityNames;
    distances.resize(cities.count() * (cities.count() - 1) / 2);
    distances.fill(0);
    reset();
}
```

If a new list of cities is given, we set the private QStringList to the new list,
resize and clear the distances vector, and call QAbstractItemModel::reset() to
notify any views that their visible items must be refetched.

```cpp
int CityModel::offsetOf(int row, int column) const
{
    if (row < column)
        qSwap(row, column);
    return (row * (row - 1) / 2) + column;
}
```

The offsetOf() private function computes the index of a given city pair in the
distances vector. For example, if we had cities A, B, C, and D, and the user
updated row 3, column 1, B to D, the offset would be $3 \times (3 - 1)/2 + 1 = 4$. If
the user had instead updated row 1, column 3, D to B, thanks to the qSwap(),
exactly the same calculation would be performed and an identical offset would
be returned.

```
<table>
<thead>
<tr>
<th>Cities</th>
<th>Table Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>A→B</td>
</tr>
<tr>
<td>C</td>
<td>A→C</td>
</tr>
<tr>
<td>D</td>
<td>A→D</td>
</tr>
</tbody>
</table>
```

**Figure 10.13.** The cities and distances data structures and the table model

The last example in this section is a model that shows the parse tree for a given
regular expression. A regular expression consists of one or more terms, separated
by '|' characters. Thus, the regular expression “alpha|bravo|charlie”
contains three terms. Each term is a sequence of one or more factors; for example,
the term “bravo” consists of five factors (each letter is a factor). The factors
can be further decomposed into an atom and an optional quantifier, such as ‘∗’,
‘+’, and ‘?’. Since regular expressions can have parenthesized subexpressions,
they can have recursive parse trees.

The regular expression shown in Figure 10.14, “ab|(cd)?e”, matches an ‘a’
followed by a ‘b’, or alternatively either a ‘c’ followed by a ‘d’ followed by an ‘e’,
or just an ‘e’ on its own. So it will match “ab” and “cde”, but not “bc” or “cd”.
The Regexp Parser application consists of four classes:

- **RegExpWindow** is a window that lets the user enter a regular expression and shows the corresponding parse tree.
- **RegExpParser** generates a parse tree from a regular expression.
- **RegExpModel** is a tree model that encapsulates a parse tree.
- **Node** represents an item in a parse tree.

Let’s start with the Node class:

```cpp
class Node {
public:
    enum Type { RegExp, Expression, Term, Factor, Atom, Terminal }

    Node(Type type, const QString &str = "");
    ~Node();

    Type type;
    QString str;
    Node *parent;
    QList<Node *> children;
};
```

Every node has a type, a string (which may be empty), a parent (which may be 0), and a list of child nodes (which may be empty).

```cpp
Node::Node(Type type, const QString &str) {
    this->type = type;
    this->str = str;
    parent = 0;
}
```
The constructor simply initializes the node’s type and string. Because all the
data is public, code that uses Node can manipulate the type, string, parent, and
children directly.

Node::~Node()
{
    qDeleteAll(children);
}

The qDeleteAll() function iterates over a container of pointers and calls
delete on each one. It does not set the pointers to 0, so if it is used outside of a
destructor it is common to follow it with a call to clear() on the container that
holds the pointers.

Now that we have defined our data items (each represented by a Node), we are
ready to create a model:

```
class RegExpModel : public QAbstractItemModel
{
public:
    RegExpModel(QObject *parent = 0);
    ~RegExpModel();
    void setRootNode(Node *node);

    QModelIndex index(int row, int column,
        const QModelIndex &parent) const;
    QModelIndex parent(const QModelIndex &child) const;

    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
        int role) const;

private:
    Node *nodeFromIndex(const QModelIndex &index) const;

    Node *rootNode;
};
```

This time we have inherited from QAbstractItemModel rather than from its con-
venience subclass QAbstractTableModel, because we want to create a hierarchical
model. The essential functions that we must reimplement remain the same,
except that we must also implement index() and parent(). To set the model’s
data, we have a setRootNode() function that must be called with a parse tree’s
root node.

```
RegExpModel::RegExpModel(QObject *parent)
    : QAbstractItemModel(parent)
{
    rootNode = 0;
}```
In the model's constructor, we just need to set the root node to a safe null value and pass on the parent to the base class.

```cpp
class RegExpModel:~RegExpModel()
{
    delete rootNode;
}
```

In the destructor we delete the root node. If the root node has children, each of these is deleted, and so on recursively, by the Node destructor.

```cpp
void RegExpModel::setRootNode(Node *node)
{
    delete rootNode;
    rootNode = node;
    reset();
}
```

When a new root node is set, we begin by deleting any previous root node (and all of its children). Then we set the new root node and call `reset()` to notify any views that they must refetch the data for any visible items.

```cpp
QModelIndex RegExpModel::index(int row, int column, const QModelIndex &parent) const
{
    if (!rootNode)
        return QModelIndex();
    const QModelIndex &parentIndex = parent.isValid() ? parent : QModelIndex();
    Node *parentNode = nodeFromIndex(parentIndex);
    return createIndex(row, column, parentNode->children[row]);
}
```

The `index()` function is reimplemented from `QAbstractItemModel`. It is called whenever the model or the view needs to create a `QModelIndex` for a particular child item (or a top-level item if the parent is an invalid `QModelIndex`). For table and list models, we don't need to reimplement this function, because `QAbstractListModel` and `QAbstractTableModel`'s default implementations normally suffice.

In our `index()` implementation, if no parse tree is set, we return an invalid `QModelIndex`. Otherwise, we create a `QModelIndex` with the given row and column and with a `Node *` for the requested child. For hierarchical models, knowing the row and column of an item relative to its parent is not enough to uniquely identify it; we must also know who the parent is. To solve this, we can store a pointer to the internal node in the `QModelIndex`. `QModelIndex` gives us the option of storing a `void *` or an `int` in addition to the row and column numbers.

The `Node *` for the child is obtained through the parent node's `children` list. The parent node is extracted from the parent model index using the `nodeFromIndex()` private function:

```cpp
Node *RegExpModel::nodeFromIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        return static_cast<Node *>(index.internalPointer());
    } else {
        return nullptr;
    }
}
```
The `nodeFromIndex()` function casts the given index's `void *` to a `Node *`, or returns the root node if the index is invalid, since an invalid model index is used to represent the root in a model.

```cpp
int RegExpModel::rowCount(const QModelIndex &parent) const
{
    Node *parentNode = nodeFromIndex(parent);
    if (!parentNode)
        return 0;
    return parentNode->children.count();
}
```

The number of rows for a given item is simply how many children it has.

```cpp
int RegExpModel::columnCount(const QModelIndex & /* parent */) const
{
    return 2;
}
```

The number of columns is fixed at 2. The first column holds the node types; the second column holds the node values.

```cpp
QModelIndex RegExpModel::parent(const QModelIndex &child) const
{
    Node *node = nodeFromIndex(child);
    if (!node)
        return QModelIndex();
    Node *parentNode = node->parent;
    if (!parentNode)
        return QModelIndex();
    Node *grandparentNode = parentNode->parent;
    if (!grandparentNode)
        return QModelIndex();
    int row = grandparentNode->children.indexOf(parentNode);
    return createIndex(row, child.column(), parentNode);
}
```

Retrieving the parent `QModelIndex` from a child is a bit more work than finding a parent's child. We can easily retrieve the parent node using `nodeFromIndex()` and going up using the `Node`'s parent pointer, but to obtain the row number (the position of the parent among its siblings), we need to go back to the grandparent and find the parent's index position in its parent's (that is, the child's grandparent's) list of children.

```cpp
QVariant RegExpModel::data(const QModelIndex &index, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    Node *node = nodeFromIndex(index);
    if (!node)
        return QVariant();
}
In `data()`, we retrieve the `Node *` for the requested item and we use it to access the underlying data. If the caller wants a value for any role except `Qt::DisplayRole` or if we cannot retrieve a `Node` for the given model index, we return an invalid `QVariant`. If the column is 0, we return the name of the node's type; if the column is 1, we return the node's value (its string).

```cpp
QVariant RegExpModel::headerData(int section, Qt::Orientation orientation, int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == 0) {
            return tr("Node");
        } else if (section == 1) {
            return tr("Value");
        }
    }
    return QVariant();
}
```

In our `headerData()` reimplementation, we return appropriate horizontal header labels. The `QTreeView` class, which is used to visualize hierarchical models, has no vertical header, so we ignore that possibility.

Now that we have covered the `Node` and `RegExpModel` classes, let's see how the root node is created when the user changes the text in the line edit:

```cpp
void RegExpWindow::regExpChanged(const QString &regExp)
{
    RegExpParser parser;
    return QVariant();
    if (index.column() == 0) {
        switch (node->type) {
            case Node::RegExp:
                return tr("RegExp");
            case Node::Expression:
                return tr("Expression");
            case Node::Term:
                return tr("Term");
            case Node::Factor:
                return tr("Factor");
            case Node::Atom:
                return tr("Atom");
            case Node::Terminal:
                return tr("Terminal");
            default:
                return tr("Unknown");
        }
    } else if (index.column() == 1) {
        return node->str;
    }
    return QVariant();
}
```
Node *rootNode = parser.parse(regExp);
regExpModel->setRootNode(rootNode);
}

When the user changes the text in the application’s line edit, the main window’s regExpChanged() slot is called. In this slot, the user’s text is parsed and the parser returns a pointer to the root node of the parse tree.

We have not shown the RegExpParser class because it is not relevant for GUI or model/view programming. The full source for this example is on the CD.

In this section, we have seen how to create three different custom models. Many models are much simpler than those shown here, with one-to-one correspondences between items and model indexes. Further model/view examples are provided with Qt itself, along with extensive documentation.

**Implementing Custom Delegates**

Individual items in views are rendered and edited using delegates. In most cases, the default delegate supplied by a view is sufficient. If we want to have finer control over the rendering of items, we can often achieve what we want simply by using a custom model: In our data() reimplementation we can handle the Qt::FontRole, Qt::TextAlignmentRole, Qt::TextColorRole, and Qt::BackgroundColorRole, and these are used by the default delegate. For example, in the Cities and Currencies examples shown earlier, we handled the Qt::TextAlignmentRole to get right-aligned numbers.

If we want even greater control, we can create our own delegate class and set it on the views that we want to make use of it. The Track Editor dialog shown below makes use of a custom delegate. It shows the titles of music tracks and their durations. The data held by the model will be simply QStrings (titles) and ints (seconds), but the durations will be separated into minutes and seconds and will be editable using a QTimeEdit.

![Figure 10.15. The Track Editor dialog](image-url)
Implementing Custom Delegates

The Track Editor dialog uses a QTableWidget, a convenience item view subclass that operates on QTableWidgetItem. The data is provided as a list of Tracks:

```cpp
class Track
{
public:
    Track(const QString &title = "", int duration = 0);
    QString title;
    int duration;
};
```

Here is an extract from the constructor that shows the creation and population of the table widget:

```cpp
TrackEditor::TrackEditor(QList<Track> *tracks, QWidget *parent)
    : QDialog(parent)
{
    this->tracks = tracks;
    tableWidget = new QTableWidget(tracks->count(), 2);
    tableWidget->setItemDelegate(new TrackDelegate(1));
    tableWidget->setHorizontalHeaderLabels(
        QStringList() << tr("Track") << tr("Duration"));
    for (int row = 0; row < tracks->count(); ++row) {
        Track track = tracks->at(row);
        QTableWidgetItem *item0 = new QTableWidgetItem(track.title);
        tableWidget->setItem(row, 0, item0);
        QTableWidgetItem *item1
            = new QTableWidgetItem(QString::number(track.duration));
        item1->setTextAlignment(Qt::AlignRight);
        tableWidget->setItem(row, 1, item1);
    }
}
```

The constructor creates a table widget, and instead of simply using the default delegate, we set our custom TrackDelegate, passing it the column that holds time data. We begin by setting the column headings, and then iterate through the data, populating the rows with the name and duration of each track.

The rest of the constructor and the rest of the TrackEditor dialog holds no surprises, so we will now look at the TrackDelegate that handles the rendering and editing of track data.

```cpp
class TrackDelegate : public QItemDelegate
{
    Q_OBJECT

public:
    TrackDelegate(int durationColumn, QObject *parent = 0);
```
We use QItemDelegate as our base class, so that we benefit from the default delegate implementation. We could also have used QAbstractItemDelegate if we had wanted to start from scratch. To provide a delegate that can edit data, we must implement `createEditor()`, `setEditorData()`, and `setModelData()`. We also implement `paint()` to change the rendering of the duration column.

```cpp
class TrackDelegate : public QItemDelegate {
public:
    TrackDelegate(int durationColumn, QObject *parent) :
        QItemDelegate(parent)
    {
        this->durationColumn = durationColumn;
    }

private slots:
    void commitAndCloseEditor();

private:
    int durationColumn;
};
```

The `durationColumn` parameter to the constructor tells the delegate which column holds the track duration.

```cpp
void TrackDelegate::paint(QPainter *painter, const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QString text = QString("%1:%2")
            .arg(secs / 60, 2, 10, QChar('0'))
            .arg(secs % 60, 2, 10, QChar('0'));

        QStyleOptionViewItem myOption = option;
        myOption.displayAlignment = Qt::AlignRight | Qt::AlignVCenter;
        drawDisplay(painter, myOption, myOption.rect, text);
    } else{
        QItemDelegate::paint(painter, option, index);
    }
}
```

Since we want to render the duration in the form “minutes:seconds”, we have reimplemented the `paint()` function. The `arg()` calls take an integer to render as a string, how many characters the string should have, the base of the integer (10 for decimal), and the padding character.
Implementing Custom Delegates

To right-align the text, we copy the current style options and overwrite the default alignment. We then call `QItemDelegate::drawDisplay()` to draw the text, followed by `QItemDelegate::drawFocus()`, which will draw a focus rectangle if the item has focus and will do nothing otherwise. Using `drawDisplay()` is very convenient, especially when used with our own style options. We could also draw using the painter directly.

```cpp
QWidget *TrackDelegate::createEditor(QWidget *parent,
const QStyleOptionViewItem &option,
const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = new QTimeEdit(parent);
        timeEdit->setDisplayFormat("mm:ss");
        connect(timeEdit, SIGNAL(editingFinished()),
                this, SLOT(commitAndCloseEditor()));
        return timeEdit;
    } else {
        return QItemDelegate::createEditor(parent, option, index);
    }
}
```

We only want to control the editing of track durations, leaving the editing of track names to the default delegate. We achieve this by checking which column the delegate has been asked to provide an editor for. If it's the duration column, we create a `QTimeEdit`, set the display format appropriately, and connect its `editingFinished()` signal to our `commitAndCloseEditor()` slot. For any other column, we pass on the edit handling to the default delegate.

```cpp
void TrackDelegate::commitAndCloseEditor()
{
    QTimeEdit *editor = qobject_cast<QTimeEdit*>(sender());
    emit commitData(editor);
    emit closeEditor(editor);
}
```

If the user presses Enter or moves the focus out of the `QTimeEdit` (but not if they press Esc), the `editingFinished()` signal is emitted and the `commitAndCloseEditor()` slot is called. This slot emits the `commitData()` signal to inform the view that there is edited data to replace existing data. It also emits the `closeEditor()` signal to notify the view that this editor is no longer required, at which point the model will delete it. The editor is retrieved using `QObject::sender()`, which returns the object that emitted the signal that triggered the slot. If the user cancels (by pressing Esc), the view will simply delete the editor.

```cpp
void TrackDelegate::setEditorData(QWidget *editor,
const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit*>(editor);
        timeEdit->setTime(QTime(0, secs / 60, secs % 60));
    } else {
```
When the user initiates editing, the view calls createEditor() to create an editor, and then setEditorData() to initialize the editor with the item’s current data. If the editor is for the duration column, we extract the track’s duration in seconds and set the QTimeEdit’s time to the corresponding number of minutes and seconds; otherwise, we let the default delegate handle the initialization.

```cpp
void TrackDelegate::setModelData(QWidget *editor,
                                 QAbstractItemModel *model,
                                 const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        QTime time = timeEdit->time();
        int secs = (time.minute() * 60) + time.second();
        model->setData(index, secs);
    } else {
        QItemDelegate::setModelData(editor, model, index);
    }
}
```

If the user completes the edit (for example, by left-clicking outside the editor widget, or by pressing Enter or Tab) rather than canceling it, the model must be updated with the editor’s data. If the duration was edited, we extract the minutes and seconds from the QTimeEdit, and set the data to the corresponding number of seconds.

Although not necessary in this case, it is entirely possible to create a custom delegate that finely controls the editing and rendering of any item in a model. We have chosen to take control of a particular column, but since the QModelIndex is passed to all the QItemDelegate functions that we reimplement, we can take control by column, row, rectangular region, parent, or any combination of these, right down to individual items if required.

In this chapter, we have presented a broad overview of Qt’s model/view architecture. We have shown how to use the view convenience subclasses, how to use Qt’s predefined models, and how to create custom models and custom delegates. But the model/view architecture is so rich that we have not had the space to cover all the things it makes possible. For example, we could create a custom view that does not render its items as a list, table, or tree. This is done by the Chart example located in Qt’s examples/itemviews/chart directory, which shows a custom view that renders model data in the form of a pie chart.

It is also possible to use multiple views to view the same model without any formality. Any edits made through one view will be automatically and immediately reflected in the other views. This kind of functionality is particularly useful for viewing large data sets where the user may wish to see sections of data that are logically far apart. The architecture also supports selections: Where two
or more views are using the same model, each view can be set to have its own independent selections, or the selections can be shared across the views.

Qt's online documentation provides comprehensive coverage of item view programming and the classes that implement it. See [http://doc.trolltech.com/4.1/model-view.html](http://doc.trolltech.com/4.1/model-view.html) for a list of all the relevant classes, and [http://doc.trolltech.com/4.1/model-view-programming.html](http://doc.trolltech.com/4.1/model-view-programming.html) for additional information and links to the relevant examples included with Qt.