Pemrograman Model/View pada Qt Framework

Code less.
Create more.
Deploy everywhere.

Kompilasi oleh @baddwin

Pemrograman Model/View pada Qt Framework

Pengantar Model/View Programming

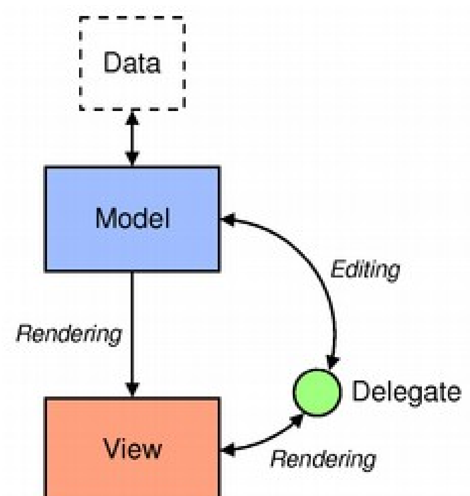
Qt berisi seperangkat kelas item view yang menggunakan arsitektur model/view untuk mengelola hubungan antara data dan caranya disajikan kepada pengguna. Pemisahan fungsionalitas yang diperkenalkan oleh arsitektur ini memberikan fleksibilitas yang lebih besar pada pengembang untuk menyesuaikan presentasi dari item, dan menyediakan antarmuka model standar untuk memungkinkan berbagai sumber data agar bisa digunakan dengan penggambaran item yang ada. Dalam dokumen ini, kami memberikan pengenalan singkat paradigma model / view, menguraikan konsep yang bersangkutan, dan menjelaskan arsitektur sistem penggambaran item. Masing-masing komponen dalam arsitektur dijelaskan, dan contoh diberikan untuk menunjukkan bagaimana menggunakan kelas yang tersedia.

Arsitektur model / view

Model-View-Controller (MVC) adalah pola desain yang berasal dari Smalltalk yang sering digunakan ketika membangun antarmuka pengguna. Dalam Design Patterns, Gamma dkk. menulis:

> MVC terdiri dari tiga macam objek. Model adalah objek aplikasinya, View adalah presentasi layarnya, dan Controller mendefinisikan cara user interface bereaksi terhadap input pengguna. Sebelum MVC, pengguna desain antarmuka cenderung mengumpulkan obyek ini bersama-sama. MVC memisahkan mereka untuk meningkatkan fleksibilitas dan penggunaannya kembali.

Jika objek penggambaran dan pengendali digabungkan, hasilnya adalah arsitektur model / view. Ini masih memisahkan cara data itu disimpan dari cara penyajian kepada pengguna, tetapi memberikan kerangka lebih sederhana berdasarkan prinsip-prinsip yang sama. Pemisahan ini memungkinkan untuk menampilkan data yang sama dalam beberapa penggambaran yang berbeda, dan menerapkan jenis penggambaran baru, tanpa mengubah struktur data yang mendasarinya. Untuk memungkinkan penanganan input pengguna yang fleksibel, kami memperkenalkan konsep delegasi. Keuntungan memiliki delegasi dalam kerangka ini adalah bahwa dia memungkinkan untuk kustomisasi cara item data digambarkan dan diedit.



Arsitektur Model / view

Model berkomunikasi dengan sumber data,

menyediakan sebuah antarmuka untuk komponen lain dalam arsitektur. Sifat komunikasi tergantung pada jenis sumber data, dan cara model diimplementasikan. View memperoleh indeks model dari model itu; ini adalah referensi untuk item-item dari data. Dengan menyediakan indeks model untuk model, view bisa mendapat detail item data dari sumber data. Dalam view yang standar, delegasi menggambar item data. Bila item diedit, delegasi berkomunikasi dengan model secara langsung dengan menggunakan indeks model.

Secara umum, kelas model / view dapat dipisahkan menjadi tiga kelompok yang dijelaskan di atas: model, view, dan delegasi. Masing-masing komponen ini didefinisikan oleh kelas abstrak yang menyediakan antarmuka umum dan, dalam beberapa kasus, implementasi default dari fitur-fitur. Kelas abstrak dimaksudkan agar bisa diturunkan dalam rangka memberikan kelengkapan fungsi yang diharapkan oleh komponen lainnya; ini juga memungkinkan komponen khusus bisa ditulis.

Model, view, dan delegasi berkomunikasi satu sama lain menggunakan sinyal dan slot:

1. Sinyal dari model memberi tahu view tentang perubahan pada data yang dimiliki oleh sumber data.
2. Sinyal dari view memberikan informasi tentang interaksi pengguna dengan item yang ditampilkan.
3. Sinyal dari delegasi digunakan selama mengedit untuk memberitahu model dan view tentang keadaan editor.

Model

Semua model-model item berdasarkan pada kelas QAbstractItemModel. Kelas ini mendefinisikan sebuah antarmuka yang digunakan oleh view dan delegasi untuk mengakses data. Data itu sendiri tidak harus disimpan dalam model; dapat disimpan dalam struktur data atau repositori yang disediakan oleh kelas terpisah, file, database, atau komponen aplikasi lainnya.

Konsep dasar mengenai model disajikan pada bagian Kelas-kelas Model.

QAbstractItemModel menyediakan antarmuka data yang cukup fleksibel untuk menangani view yang mewakili data dalam bentuk tabel, daftar, dan pohon. Namun, ketika menerapkan model baru untuk daftar dan struktur data sejenis tabel, kelas QAbstractListModel dan QAbstractTableModel merupakan titik awal yang lebih baik karena mereka menyediakan implementasi standar sesuai dengan fungsi umum. Masing-masing dari kelas-kelas ini dapat diturunkan untuk menyediakan model yang mendukung jenis khusus dari daftar dan tabel.

Proses menurunkan kelas model dibahas pada bagian Membuat Model Baru.

Qt menyediakan beberapa model siap pakai yang dapat digunakan untuk menangani item data:

1. QStringListModel digunakan untuk menyimpan daftar sederhana item

QString.

2. QStandardItemModel mengelola struktur pohon yang lebih kompleks dari item, masing-masing dapat berisi data yang berubah-ubah.

3. QFileSystemModel menyediakan informasi tentang file dan direktori dalam sistem berkas-berkas lokal.

4. QSqlQueryModel, QSqlTableModel, dan QSqlRelationalTableModel digunakan untuk mengakses database dengan menggunakan kaidah model / view.

Jika model-model standar tidak memenuhi kebutuhan Anda, Anda dapat menurunkan QAbstractItemModel, QAbstractListModel, atau QAbstractTableModel untuk mencipta-kan model kustom Anda sendiri.

Views

Implementasi lengkap disediakan untuk berbagai jenis tampilan: QListView menampilkan daftar item, QTableView menampilkan data dari model dalam sebuah tabel, dan QTreeView menunjukkan item-item model data dalam daftar hirarkis. Masing-masing kelas ini berdasarkan pada kelas dasar abstrak QAbstractItemView. Meskipun kelas-kelas ini adalah implementasi yang siap pakai, mereka juga dapat diturunkan untuk memberikan view yang disesuaikan.

View yang tersedia akan diusut pada bagian Kelas-kelas View.

Delegasi

QAbstractItemDelegate adalah kelas dasar abstrak untuk delegasi dalam kerangka model / view. Implementasi delegasi default disediakan oleh QStyledItemDelegate, dan ini digunakan sebagai delegasi default dengan view standar Qt. Namun, QStyledItemDelegate dan QItemDelegate alternatif independen untuk pelukisan dan menyediakan editor untuk item dalam view. Perbedaan antara mereka adalah bahwa QStyledItemDelegate menggunakan gaya yang sekarang untuk melukis itemnya. Oleh karena itu kami merekomendasikan menggunakan QStyledItemDelegate sebagai kelas dasar ketika mengimplementasikan delegasi kustom atau ketika bekerja dengan stylesheet Qt.

Delegasi dijelaskan pada bagian Kelas-kelas Delegasi.

Penyortiran

Ada dua cara untuk mendekati penyortiran dalam arsitektur Model / view; pendekatan mana yang dipilih bergantung pada model yang mendasarinya.

Jika model Anda bisa diurutkan, yaitu jika ia menggunakan kembali fungsi QAbstractItemModel::sort(), baik QTableView maupun QTreeView menyediakan API yang memungkinkan Anda untuk mengurutkan data model secara terprogram. Selain itu, Anda dapat mengaktifkan pernyortiran interaktif (yaitu memungkinkan pengguna

untuk menyortir data dengan mengklik header dari view), dengan menghubungkan sinyal QHeaderView::sortIndicatorChanged() ke slot QTableView::sortByColumn() atau slot QTreeView::sortByColumn(), masing-masing.

Pendekatan alternatif, jika model Anda tidak memiliki antarmuka yang diperlukan atau jika Anda ingin menggunakan tampilan daftar untuk menyajikan data Anda, adalah dengan menggunakan model proxy untuk mengubah struktur model Anda sebelum menyajikan data dalam view. Hal ini dibahas secara rinci pada bagian Model-model Proxy.

Kelas yang Memudahkan

Sejumlah kelas yang memudahkan berasal dari kelas standar view untuk kepentingan aplikasi yang mengandalkan kelas-kelas item view dan tabel Qt yang berbasis item. Mereka tidak dimaksudkan untuk diturunkan.

Contoh kelas tersebut termasuk QListWidget, QTreeWidget, dan QTableWidget.

Kelas-kelas ini kurang fleksibel dibandingkan dengan kelas-kelas view, dan tidak dapat digunakan dengan model yang berubah-ubah. Kami menyarankan agar Anda menggunakan pendekatan model / view untuk menangani data dalam item view kecuali jika Anda sangat membutuhkan seperangkat kelas berbasis item.

Jika Anda ingin memanfaatkan fitur yang disediakan oleh pendekatan model / view sementara masih menggunakan antarmuka berbasis item, pertimbangkan untuk menggunakan kelas view, seperti QListView, QTableView, dan QTreeView dengan QstandardItemModel.

Menggunakan model dan views

Bagian berikut ini menjelaskan cara menggunakan pola model / view di Qt. Setiap bagian mencakup contoh dan diikuti oleh bagian yang menunjukkan cara membuat komponen baru.

Dua model yang disertakan dalam Qt

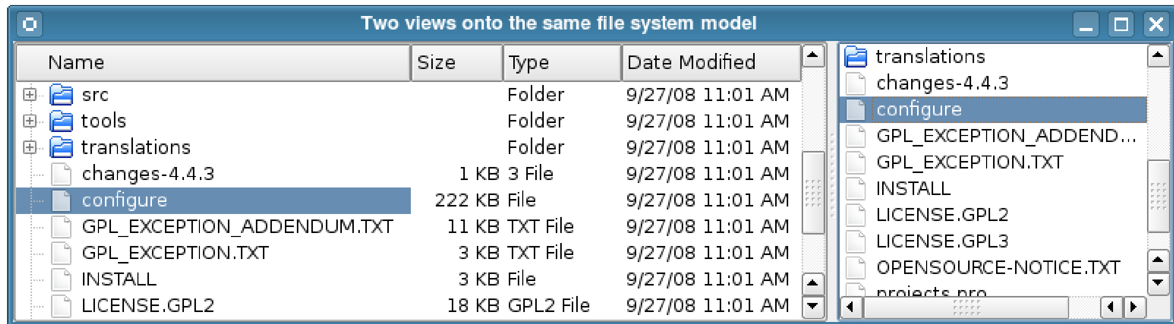
Dua model standar yang disediakan oleh Qt adalah QStandardItemModel dan QFileSystemModel. QStandardItemModel adalah model multi-purpose yang dapat digunakan untuk mewakili berbagai struktur data yang berbeda yang dibutuhkan oleh daftar, tabel, dan view pohon. Model ini juga memegang item-item dari data. QFileSystemModel adalah model yang menangani informasi mengenai isi dari direktori. Akibatnya, ia tidak menangani item dari data itu sendiri, tetapi hanya melambangkan file dan direktori pada sistem pengarsipan lokal.

QFileSystemModel menyediakan model siap pakai untuk bereksperimen dengannya, dan dapat dengan mudah dikonfigurasi untuk menggunakan data yang ada. Dengan menggunakan model ini, kita dapat menunjukkan cara membuat sebuah model untuk digunakan dengan view yang sudah jadi, dan mengeksplorasi cara

memanipulasi data dengan menggunakan indeks-indeks model.

Menggunakan view dengan model yang sudah ada

Kelas QListView dan QTreeView adalah view yang paling cocok untuk digunakan dengan QFileSystemModel. Contoh yang disajikan di bawah ini menampilkan isi dari sebuah direktori dalam tampilan pohon bersebelahan dengna informasi yang sama dalam tampilan daftar. View tersebut berbagi pilihan oleh pengguna sehingga item yang dipilih akan disorot dalam kedua view.



Kita membuat sebuah QFileSystemModel hingga siap digunakan, dan menciptakan beberapa view untuk menampilkan isi dari direktori. Ini menunjukkan cara paling sederhana untuk menggunakan model. Pembangunan dan penggunaan model tersebut dilakukan dari dalam fungsi main() tunggal:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplitter *splitter = new QSplitter;

    QFileSystemModel *model = new QFileSystemModel;
    model->setRootPath(QDir::currentPath());
```

Model ini dibentuk untuk menggunakan data dari sistem file tertentu. Panggilan ke setRootPath() memberitahu model drive mana pada sistem file untuk diperlihatkan ke view.

Kita menciptakan dua view sehingga kita bisa memeriksa item-item yang dimiliki model dalam dua cara yang berbeda:

```
QTreeView *tree = new QTreeView(splitter);
tree->setModel(model);
tree->setRootIndex(model->index(QDir::currentPath()));

QListView *list = new QListView(splitter);
list->setModel(model);
list->setRootIndex(model->index(QDir::currentPath()));
```

View tersebut dibangun dengan cara yang sama seperti widget lainnya. Menyiapkan view untuk menampilkan item dalam model ini hanya masalah memanggil fungsi setModel() nya dengan model direktori sebagai argumen. Kita menyaring data yang diberikan oleh model dengan memanggil fungsi setRootIndex() pada setiap view, memberikan indeks model yang cocok dari model sistem file untuk direktori saat ini.

Fungsi index() yang digunakan dalam hal ini adalah unik untuk QFileSystemModel; kita menyediakan direktori dan mengembalikan sebuah indeks Model. Indeks-indeks model dibahas dalam Kelas-kelas Model.

Fungsi-fungsi lainnya hanya menampilkan view dalam widget splitter, dan menjalankan event loop aplikasi:

```
    splitter->setWindowTitle("Two views onto the same file system model");
    splitter->show();
    return app.exec();
}
```

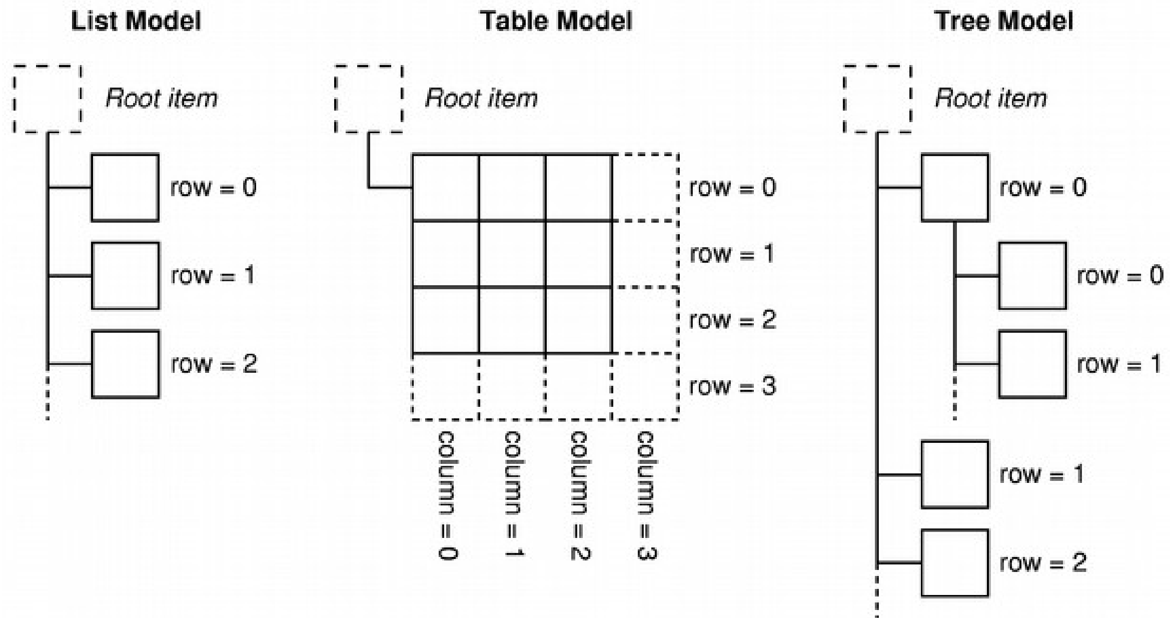
Dalam contoh di atas, kita tidak menyebutkan bagaimana menangani pilihan item. Pokok masalah ini dibahas secara lebih rinci dalam bagian tentang Penanganan Seleksi dalam Item Views.

Kelas-kelas Model

Sebelum meneliti bagaimana menangani seleksi, Anda mungkin perlu untuk memeriksa konsep yang digunakan dalam kerangka model / view.

Konsep-konsep dasar

Dalam arsitektur model / view, model memberikan antarmuka standar yang digunakan view dan delegasi untuk mengakses data. Dalam Qt, antarmuka standar didefinisikan oleh kelas QAbstractItemModel. Tidak peduli bagaimana item data disimpan dalam struktur data yang mendasarinya, semua kelas turunan dari QAbstractItemModel mewakili data sebagai struktur hirarkis yang mengandung tabel-tabel berisi item-item. Views menggunakan kaidah ini untuk mengakses item data dalam model, tetapi mereka tidak dibatasi dalam cara mereka menyajikan informasi ini kepada pengguna.



Model juga memberitahu setiap view yang tersemat tentang perubahan data melalui mekanisme sinyal dan slot.

Bagian ini menjelaskan beberapa konsep dasar yang penting bagi cara item data yang diakses oleh komponen lain melalui kelas model. Konsep yang lebih lanjut dibahas dalam bagian berikutnya.

Indeks-indeks Model

Untuk memastikan bahwa representasi dari data yang disimpan terpisah dari caranya diakses, konsep sebuah indeks model diperkenalkan. Setiap bagian dari informasi yang dapat diperoleh melalui model diwakili oleh indeks Model. View dan delegasi menggunakan indeks ini untuk meminta item data untuk ditampilkan.

Akibatnya, hanya model yang harus tahu cara untuk mendapatkan data, dan jenis data yang dikelola oleh model dapat didefinisikan cukup secara umum. Indeks-indeks Model berisi penunjuk ke model yang menciptakan mereka, dan ini mencegah kebingungan ketika bekerja dengan lebih dari satu model.

```
QAbstractItemModel *model = index.model();
```

Indeks-indeks model memberikan referensi sementara ke bagian-bagian informasi, dan dapat digunakan untuk mengambil atau mengubah data melalui model. Karena model dapat menata kembali struktur internal mereka dari waktu ke waktu, indeks-indeks model bisa menjadi tidak sah, dan tidak perlu disimpan. Jika diperlukan referensi jangka panjang untuk bagian informasi, indeks model persisten harus dibuat. Ini menyediakan referensi untuk informasi yang modelnya terus up-to-date. Indeks-indeks model temporer disediakan oleh kelas `QModelIndex`, dan indeks model persisten disediakan oleh kelas `QPersistentModelIndex`.

Untuk mendapatkan indeks model yang sesuai dengan item data, tiga properti harus ditetapkan untuk model: nomor baris, nomor kolom, dan indeks model dari item induk. Bagian berikut ini menggambarkan dan menjelaskan sifat-sifat ini secara rinci.

Baris dan kolom

Dalam bentuk yang paling dasar, model dapat diakses sebagai sebuah tabel sederhana di mana item-itemnya diletakkan berdasarkan nomor baris dan kolom mereka. Ini tidak berarti bahwa potongan-potongan data yang mendasarinya disimpan dalam struktur array; penggunaan nomor baris dan kolom hanya kaidah agar komponen-komponen bisa berkomunikasi satu sama lain. Kita dapat mengambil informasi tentang item yang ditentukan dengan menetapkan nomor baris dan kolom ke model, dan kita menerima indeks yang mewakili item:

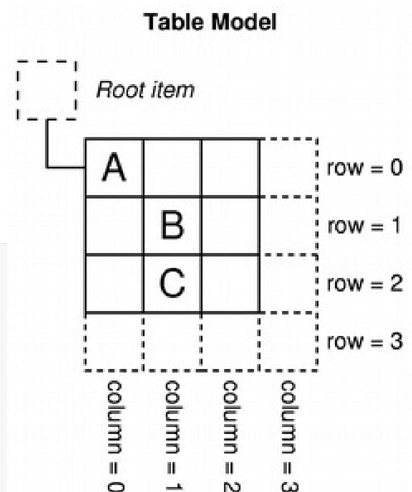
```
QModelIndex index = model->index(row, column, ...);
```

Model yang menyediakan interface untuk struktur data tingkat sederhana dan tunggal seperti daftar dan tabel tidak perlu diberi informasi lain, tetapi seperti yang telah diterangkan kode di atas, kita perlu memberikan informasi lebih ketika mengambil indeks Model.

Baris dan kolom

Diagram ini menunjukkan representasi dari model table dasar yang setiap itemnya diletakkan berdasarkan sepasang nomor baris dan kolom. Kita mendapatkan indeks model yang mengacu pada item data dengan memberikan nomor baris dan kolom yang relevan dengan model.

```
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexB = model->index(1, 1, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
```



Item tingkat atas dalam model selalu direferensikan dengan menetapkan `QModelIndex()` sebagai item induk mereka. Hal ini dibahas dalam bagian berikutnya.

Induk dari item-item

Antarmuka sejenis tabel untuk data item yang disediakan oleh model sangat ideal bila menggunakan data dalam view tabel atau daftar; sistem nomor baris dan kolom memetakan persis seperti cara view menampilkan item. Namun, struktur semacam

view pohon memerlukan model untuk menunjukkan antarmuka yang lebih fleksibel untuk item di dalamnya. Akibatnya, setiap item juga dapat menjadi induk dari tabel lain dari item-item, dalam banyak cara yang sama bahwa item tingkat puncak dalam tampilan pohon dapat berisi daftar item-item lain.

Ketika meminta indeks untuk item model, kita harus memberikan beberapa informasi tentang induk item. Di luar model, satu-satunya cara untuk merujuk ke salah satu item adalah melalui indeks Model, sehingga indeks Model induk juga harus diberikan:

```
QModelIndex index = model->index(row, column, parent);
```

Induk, baris, dan kolom

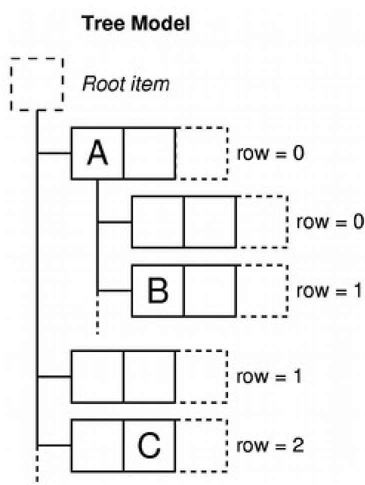


Diagram ini menunjukkan representasi dari model pohon yang setiap itemnya disebutkan oleh induk, nomor baris, dan nomor kolom. Item "A" dan "C" direpresentasikan sebagai saudara tingkat puncak dalam model:

```
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

Item "A" memiliki sejumlah anak. Sebuah indeks model untuk item "B" diperoleh dengan kode berikut:

```
QModelIndex indexB = model->index(1, 0, indexA);
```

Peranan item-item

Item dalam model dapat melakukan berbagai peranan untuk komponen lain, agar berbagai jenis data bisa diberikan untuk situasi yang berbeda. Sebagai contoh, Qt::DisplayRole digunakan untuk mengakses string yang dapat ditampilkan sebagai teks dalam view. Secara khusus, item-item berisi data untuk sejumlah peranan yang berbeda, dan peranan standar didefinisikan oleh Qt::ItemDataRole .

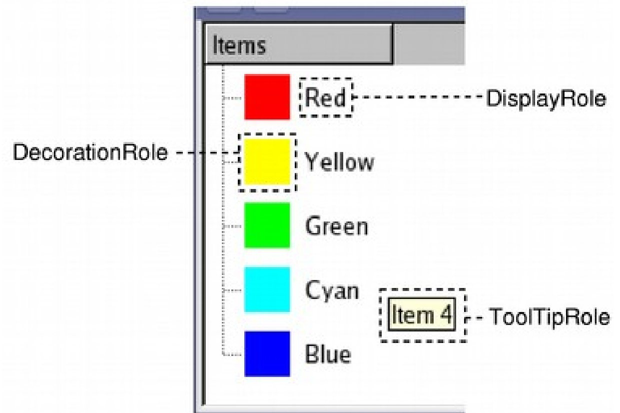
Kita bisa meminta data item dari model dengan memberikan indeks model yang sesuai dengan item, dan dengan menentukan peranan untuk mendapatkan jenis data yang kita inginkan :

```
QVariant value = model->data(index, role);
```

Peranan item

Peranan menunjukkan pada model jenis data mana yang dirujuk. Views dapat menampilkan peran dalam cara yang berbeda, sehingga sangat penting untuk memberikan informasi yang tepat untuk masing-masing peran.

Bagian Membuat Model Baru mencakup beberapa penggunaan spesifik dari peranan secara lebih rinci.



Penggunaan yang paling umum untuk data item yang tercakup dalam peranan standar ditetapkan dalam `Qt::ItemDataRole`. Dengan menyediakan data item yang sesuai untuk masing-masing peran, model dapat memberikan petunjuk untuk view dan delegasi tentang bagaimana item harus disajikan kepada pengguna. Berbagai jenis view memiliki kebebasan untuk menafsirkan atau mengabaikan informasi ini sesuai kebutuhan. Hal ini juga memungkinkan untuk mendefinisikan peran tambahan untuk tujuan khusus-aplikasi.

Ikhtisar

1. Indeks-indeks model memberikan informasi pada view dan delegasi tentang lokasi item-item yang disediakan oleh model dengan cara yang independen dari setiap struktur data yang mendasarinya.
2. Item disebutkan dengan nomor baris dan kolom mereka, dan dengan indeks model dari item induknya.
3. Indeks-indeks Model dibangun oleh model atas permintaan komponen lainnya, seperti view dan delegasi.
4. Jika indeks model yang valid ditentukan untuk item induk ketika indeks diminta menggunakan indeks ( ) , indeks kembali mengacu pada item di bawah item induk dalam model. Indeks yang diperoleh mengacu pada anak dari barang tersebut.
5. Jika indeks model yang valid ditentukan untuk item orangtua ketika indeks diminta dengan index(), indeks yang dikembalikan mengacu ke item tingkat puncak dalam model.
6. Peranan membedakan antara berbagai jenis data yang terkait dengan item.

Menggunakan indeks-indeks model

Untuk menunjukkan bagaimana data dapat diambil dari model, dengan menggunakan indeks-indeks model, kita membuat `QFileSystemModel` tanpa view dan menampilkan nama-nama file dan direktori dalam widget. Meskipun hal ini tidak menunjukkan cara normal menggunakan model, ini menunjukkan kaidah yang digunakan oleh model ketika berurusan dengan indeks-indeks model.

Kita membangun sebuah model sistem file dengan cara sebagai berikut :

```
QFileSystemModel *model = new QFileSystemModel;
QModelIndex parentIndex = model->index(QDir::currentPath());
int numRows = model->rowCount(parentIndex);
```

Dalam hal ini, kita membangun QFileSystemModel default, mendapatkan indeks induk menggunakan implementasi khusus dari index() yang disediakan oleh model itu, dan kita menghitung jumlah baris dalam model dengan menggunakan fungsi rowCount().

Demi kemudahan, kita hanya tertarik pada item dalam kolom pertama dari model. Kita memeriksa setiap baris pada gilirannya, memperoleh indeks model untuk item pertama dalam setiap baris, dan membaca data yang disimpan untuk item itu dalam model.

```
for (int row = 0; row < numRows; ++row) {
    QModelIndex index = model->index(row, 0, parentIndex);
```

Untuk mendapatkan indeks model, kita menentukan nomor baris, nomor kolom (nol untuk kolom pertama), dan indeks model yang sesuai untuk induk dari semua item yang kita inginkan. Teks yang disimpan dalam setiap item diambil menggunakan fungsi data() dari model. Kita menentukan indeks model dan DisplayRole untuk memperoleh data untuk item dalam bentuk string.

```
    QString text = model->data(index, Qt::DisplayRole).toString();
    // Display the text in a widget.

}
```

Contoh di atas menunjukkan prinsip-prinsip dasar yang digunakan untuk mengambil data dari model:

- Dimensi dari model dapat ditemukan dengan menggunakan rowCount() dan columnCount(). Fungsi-fungsi ini umumnya perlu menentukan indeks Model induk.
- Indeks-indeks model digunakan untuk mengakses item dalam model. Baris, kolom, dan indeks Model induk diperlukan untuk menentukan item.
- Untuk mengakses item tingkat puncak dalam model, tentukan indeks Model nol sebagai indeks induk dengan QModelIndex().
- Item berisi data untuk peranan yang berbeda. Untuk memperoleh data untuk peran tertentu, indeks Model dan peranan harus diberikan kepada model.

Bacaan lebih lanjut

Model-model baru dapat dibuat dengan menerapkan antarmuka standar yang

disediakan oleh QAbstractItemModel. Dalam bagian Membuat Model Baru, kami mendemonstrasikan ini dengan menciptakan sebuah model mudah yang siap digunakan untuk menangani daftar string.

Kelas-Kelas View

Konsep

Dalam arsitektur model/view, view memperoleh item data dari model dan menyajikannya kepada pengguna. Cara data itu disajikan tidak perlu menyerupai representasi dari data yang diberikan oleh model, dan mungkin benar-benar berbeda dari struktur data yang mendasarinya yang digunakan untuk menyimpan item data.

Pemisahan konten dan penyajian diperoleh dengan menggunakan model antarmuka standar yang disediakan oleh QAbstractItemModel, antarmuka view standar yang disediakan oleh QAbstractItemView, dan penggunaan indeks-indeks model yang mewakili item data secara umum. Views biasanya mengatur tata letak keseluruhan data yang diperoleh dari model. Mereka mungkin menggambar setiap item dari datanya sendiri, atau menggunakan delegasi untuk menangani baik penggambaran maupun fitur editing.

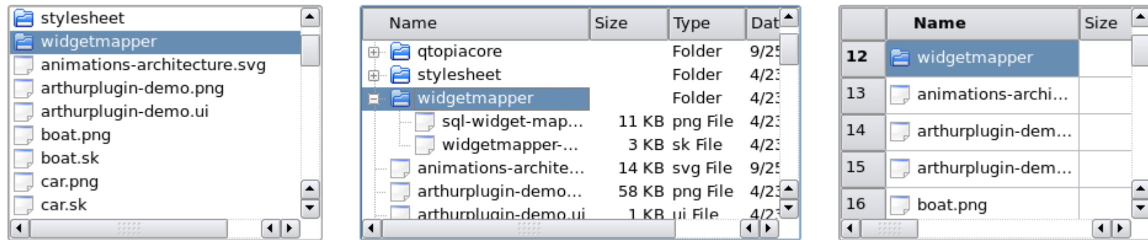
Selain penyajian data, view menangani navigasi antara item-item, dan beberapa aspek seleksi item. View juga mengimplementasikan fitur antarmuka pengguna dasar, seperti menu konteks dan drag-drop. view dapat menyediakan fasilitas editing default untuk item, atau mungkin bekerja dengan delegasi untuk memberikan editor kustom.

Sebuah view dapat dibangun tanpa model, tetapi model harus disediakan sebelum dapat menampilkan informasi yang berguna. Views melacak item yang telah dipilih oleh pengguna melalui penggunaan pilihan yang dapat dipertahankan secara terpisah untuk tiap-tiap view, atau dibagi antara banyak view.

Beberapa view, seperti QTableView dan QTreeView, menampilkan header serta item-item. Ini juga dilaksanakan oleh kelas view, QHeaderView. Header biasanya mengakses model yang sama dengan view yang berisi mereka. Mereka mengambil data dari model menggunakan fungsi QAbstractItemModel::headerData(), dan biasanya menampilkan informasi header dalam bentuk label. Header baru dapat diturunkan dari kelas QHeaderView untuk memberikan label yang lebih khusus untuk view.

Menggunakan pandangan yang ada

Qt menyediakan tiga kelas view siap pakai yang menyajikan data dari model dalam cara-cara yang familiar bagi sebagian besar pengguna. QListView dapat menampilkan item dari model sebagai daftar sederhana, atau dalam bentuk tampilan ikon klasik. QTreeView menampilkan item dari model sebagai hirarki daftar, yang memungkinkan struktur bersarang untuk digambarkan dengan cara yang kompak. QTableView menyajikan item dari model dalam bentuk tabel, seperti tata letak aplikasi spreadsheet.



Perilaku default dari pandangan standar yang ditunjukkan di atas seharusnya cukup untuk sebagian besar aplikasi. Mereka menyediakan fasilitas editing dasar, dan dapat disesuaikan untuk memenuhi kebutuhan user interface yang lebih khusus.

Menggunakan model

Kita mengambil model daftar string yang kita buat sebagai contoh model, mengaturnya dengan beberapa data, dan membangun view untuk menampilkan isi dari model. Semua ini dapat dilakukan dalam satu fungsi:

```
int main(int argc, char *argv[]) { QApplication app(argc, argv);
// Unindented for quoting purposes:
QStringList numbers; numbers << "One" << "Two" << "Three" << "Four" << "Five";
QAbstractItemModel *model = new StringListModel(numbers);
```

Perhatikan bahwa `StringListModel` tersebut dinyatakan sebagai `QAbstractItemModel`. Hal ini memungkinkan kita untuk menggunakan antarmuka abstrak ke model, dan memastikan bahwa kode masih bekerja, bahkan jika kita mengganti string daftar model dengan model yang berbeda.

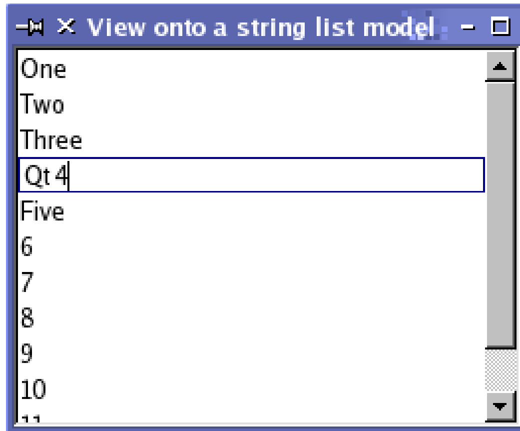
Tampilan daftar yang disediakan oleh `QListView` cukup untuk menyajikan item dalam model daftar string. Kita membangun view, dan mengatur model menggunakan baris kode berikut:

```
QListView *view = new QListView; view->setModel(model);
```

View ditampilkan dengan cara normal:

```
view->show();
return app.exec();
}
```

View menggambar isi model, mengakses data melalui antarmuka model. Ketika pengguna mencoba untuk mengedit item, view menggunakan delegasi standar untuk memberikan widget Editor.



Gambar di atas menunjukkan bagaimana QListView merepresentasikan data dalam model daftar string. Karena model ini dapat diedit, view secara otomatis memungkinkan setiap item dalam daftar untuk diedit menggunakan delegasi default.

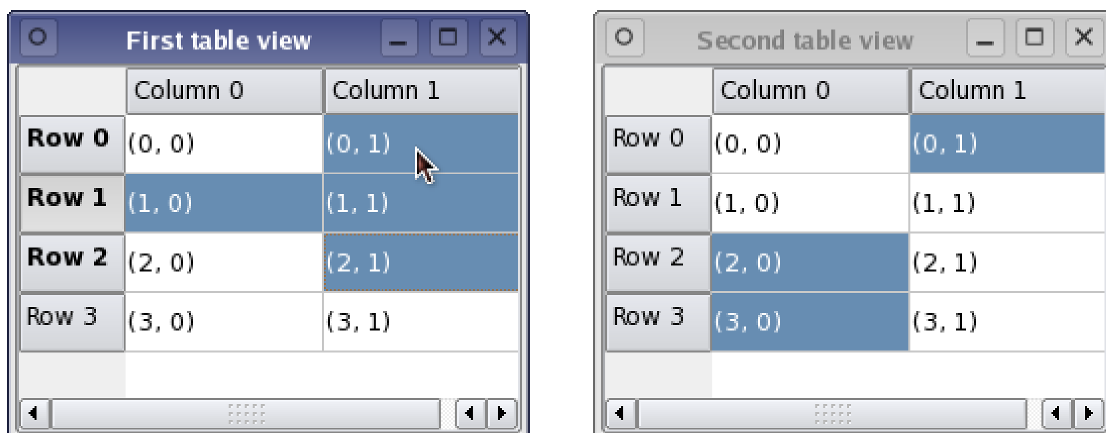
Menggunakan beberapa view dari model

Menyediakan beberapa view ke model yang sama hanya masalah pengaturan model yang sama untuk setiap tampilan. Dalam kode berikut kita membuat dua view tabel, masing-masing menggunakan model tabel sederhana yang sama yang telah kita buat untuk contoh ini:

```
QTableView *firstTableView = new QTableView;
QTableView *secondTableView = new QTableView;

firstTableView->setModel(model);
secondTableView->setModel(model);
```

Penggunaan sinyal dan slot dalam arsitektur Model / view berarti bahwa perubahan pada model tersebut dapat disebarkan ke semua view yang tersemat, memastikan bahwa kita selalu dapat mengakses data yang sama terlepas dari tampilan yang digunakan.



Gambar di atas menunjukkan dua view yang berbeda ke model yang sama, masing-

masing berisi jumlah item yang dipilih. Meskipun data dari model ditampilkan secara konsisten di seluruh view, masing-masing view mempertahankan model seleksi internalnya sendiri. Hal ini dapat berguna dalam situasi tertentu, tetapi untuk banyak aplikasi, model seleksi bersama lebih diinginkan.

Penanganan pilihan item

Mekanisme untuk menangani pilihan item dalam view disediakan oleh kelas `QItemSelectionModel`. Semua view standar membangun model pilihan mereka sendiri secara default, dan berinteraksi dengan mereka dengan cara biasa. Model seleksi yang digunakan oleh pandangan dapat diperoleh melalui fungsi `selectionModel()`, dan model seleksi pengganti dapat ditentukan dengan `setSelectionModel()`. Kemampuan untuk mengendalikan model seleksi yang digunakan oleh view berguna ketika kita ingin memberikan beberapa pandangan yang konsisten ke model data yang sama.

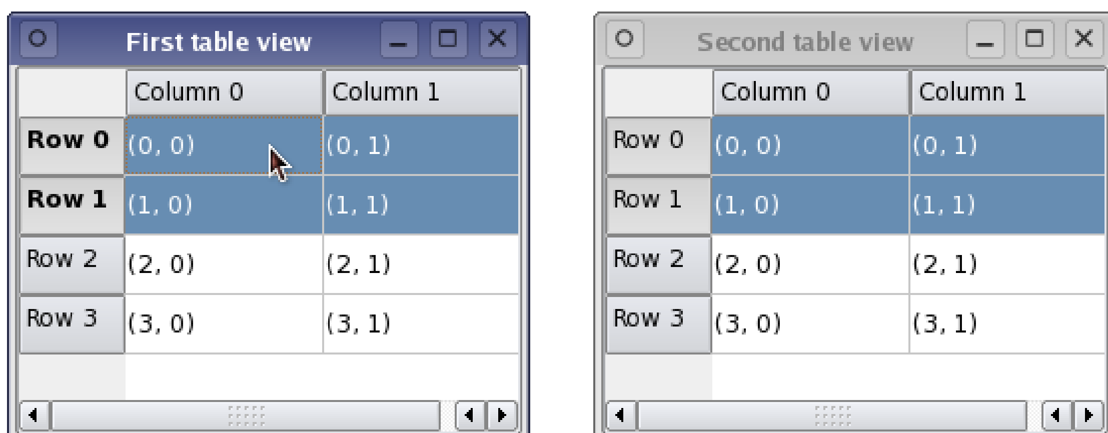
Secara umum, kecuali Anda menurunkan kelas model atau view, Anda tidak perlu memanipulasi isi pilihan secara langsung. Namun, antarmuka dengan model seleksi dapat diakses, jika diperlukan, dan ini dieksplorasi dalam Penanganan Seleksi dalam View Item.

Berbagi Pilihan antara views

Meskipun itu mudah bahwa kelas-kelas view menyediakan model pilihan mereka sendiri secara default, ketika kita menggunakan lebih dari satu tampilan ke model yang sama, sering diinginkan kedua data model dan seleksi pengguna ini ditunjukkan secara konsisten dalam semua view. Karena kelas tampilan memungkinkan model seleksi internal mereka diganti, kita dapat mencapai pilihan terpadu antara tampilan dengan baris berikut:

```
secondTableView->setSelectionModel(firstTableView->selectionModel());
```

View kedua diberikan model seleksi untuk pandangan pertama. Kedua view sekarang beroperasi pada model seleksi yang sama, menjaga kedua data dan item yang dipilih agar disinkronkan.



Dalam contoh di atas, dua view dari jenis yang sama digunakan untuk menampilkan data model yang sama itu. Namun, jika dua jenis view yang berbeda yang digunakan, item yang dipilih mungkin ditampilkan sangat berbeda dalam setiap tampilan; misalnya, pilihan yang berdekatan dalam tampilan tabel dapat direpresentasikan sebagai satu set terfragmentasi dari item yang disorot dalam tampilan pohon.

Kelas-kelas delegasi

Konsep

Berbeda dengan pola Model-View-Controller, desain model / view tidak memuat komponen yang benar-benar terpisah untuk mengelola interaksi dengan pengguna. Umumnya, view bertanggung jawab atas presentasi data dari model ke pengguna, dan untuk memproses input pengguna. Untuk memungkinkan beberapa fleksibilitas dalam cara masukan ini diperoleh, interaksi dilakukan oleh delegasi. Komponen ini memberikan kemampuan input dan juga bertanggung jawab untuk menggambar setiap item dalam beberapa view. Antarmuka standar untuk mengendalikan delegasi didefinisikan dalam kelas `QAbstractItemDelegate`.

Delegasi diharapkan mampu membuat isinya sendiri dengan menerapkan fungsi `paint()` dan `sizeHint()`. Namun, delegasi berbasis widget sederhana dapat menurunkan kelas `QItemDelegate` bukannya `QAbstractItemDelegate`, dan memanfaatkan implementasi default dari fungsi ini.

Editor untuk delegasi dapat diimplementasikan baik dengan menggunakan widget untuk mengelola proses editing ataupun dengan penanganan kejadian secara langsung. Pendekatan pertama dibahas nanti dalam bagian ini, dan juga ditunjukkan pada contoh Delegasi Spin Box.

Contoh Pixelator menunjukkan bagaimana untuk membuat delegasi kustom yang melakukan penggambaran khusus untuk tampilan tabel.

Menggunakan delegasi yang sudah ada

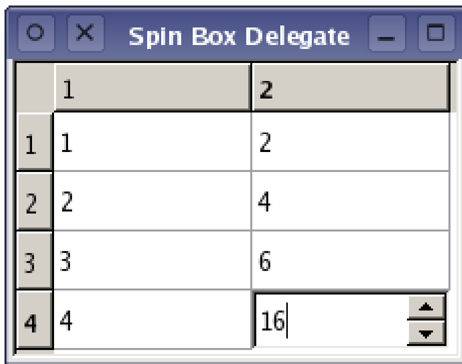
Views standar yang disediakan dengan Qt menggunakan contoh `QItemDelegate` untuk menyediakan fasilitas editing. Implementasi standar ini dari antarmuka delegasi menggambar item dalam gaya yang biasa untuk masing-masing view standar: `QListView`, `QTableView`, dan `QTreeView`.

Semua peranan standar akan ditangani oleh delegasi default yang digunakan oleh view standar. Cara penafsirannya dijelaskan dalam dokumentasi `QItemDelegate`.

Delegasi yang digunakan oleh view dikembalikan oleh fungsi `itemDelegate()`. Fungsi `setItemDelegate()` memungkinkan Anda untuk menginstal sebuah delegasi kustom untuk tampilan standar, dan itu perlu untuk menggunakan fungsi ini saat mengatur delegasi untuk tampilan kustom.

Sebuah delegasi yang sederhana

Delegasi yang diterapkan di sini menggunakan QSpinBox untuk menyediakan fasilitas editing, dan terutama ditujukan untuk penggunaan dengan model yang menampilkan bilangan bulat. Meskipun kita mendirikan model tabel berbasis bilangan bulat kustom untuk tujuan ini, kita bisa dengan mudah menggunakan QStandardItemModel sebagai gantinya, karena delegasi kustom mengontrol entri data.



Kita membangun view tabel untuk menampilkan isi dari model, dan ini akan menggunakan delegasi kustom untuk mengedit.

Kita menurunkan kelas delegasi dari QItemDelegate karena kita tidak ingin menulis fungsi tampilan kustom. Namun, kita masih harus menyediakan fungsi untuk mengelola widget Editor:

```
class SpinBoxDelegate : public QStyledItemDelegate { Q_OBJECT
public: SpinBoxDelegate(QObject *parent = 0);
QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem &option,
                      const QModelIndex &index) const;

void setEditorData(QWidget *editor, const QModelIndex &index) const;
void setModelData(QWidget *editor, QAbstractItemModel *model,
                  const QModelIndex &index) const;

void updateEditorGeometry(QWidget *editor,
    const QStyleOptionViewItem &option, const QModelIndex &index) const;
};
```

Perhatikan bahwa tidak ada widget Editor yang ditetapkan ketika delegasi dibangun. Kita hanya membangun sebuah widget Editor ketika dibutuhkan.

Menyediakan editor

Dalam contoh ini, ketika tampilan tabel harus menyediakan editor, ia meminta delegasi untuk memberikan widget editor yang sesuai untuk item yang dimodifikasi. Fungsi createEditor() dipenuhi dengan segala sesuatu yang delegasinya harus mampu untuk membuat sebuah widget yang sesuai:

```
QWidget SpinBoxDelegate::createEditor(QWidget *parent, const
QStyleOptionViewItem &/* option */, const QModelIndex &/* index */) const {
      QSpinBox *editor = new QSpinBox(parent); editor->setFrame(false);
      editor->setMinimum(0); editor->setMaximum(100);
      return editor;
}
```

Perhatikan bahwa kita tidak perlu menyimpan pointer ke widget Editor karena view bertanggung jawab untuk menghancurkannya ketika tidak lagi diperlukan.

Kita memasang standar event filternya delegasi pada editor untuk memastikan bahwa ia menyediakan pintasan editing standar yang pengguna harapkan. Pintasan tambahan dapat ditambahkan ke editor untuk memungkinkan perilaku yang lebih canggih; ini dibahas pada bagian Petunjuk Editing.

View memastikan bahwa editor dan geometrinya data diatur dengan benar dengan memanggil fungsi yang kita definisikan kemudian untuk tujuan ini. Kita bisa membuat editor yang berbeda tergantung pada indeks model yang disediakan oleh view. Sebagai contoh, jika kita memiliki satu kolom integer dan satu kolom string, kita bisa mengembalikan baik QSpinBox ataupun QLineEdit, tergantung pada kolom yang sedang diedit.

Delegasi harus menyediakan fungsi untuk menyalin data model ke editor. Dalam contoh ini, kita membaca data yang disimpan dalam peranan tampilan, dan menetapkan nilai dalam spinbox yang bersesuaian.

```
void SpinBoxDelegate::setEditorData(QWidget *editor, const QModelIndex &index)
const {
      int value = index.model()->data(index, Qt::EditRole).toInt();
      QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
      spinBox->setValue(value);
}
```

Dalam contoh ini, kita tahu bahwa widget editor adalah spinbox, tapi kita bisa memberikan editor yang berbeda untuk berbagai jenis data dalam model, dalam hal ini kita akan perlu untuk membuang widget untuk jenis yang sesuai sebelum mengakses fungsi anggotanya.

Mengirimkan data ke model

Ketika pengguna telah selesai mengedit nilai dalam spinbox, view meminta delegasi untuk menyimpan nilai yang diedit dalam model dengan memanggil fungsi setModelData().

```
void SpinBoxDelegate::setModelData(QWidget editor, QAbstractItemModel *model,
const QModelIndex &index) const {
      QSpinBox *spinBox = static_cast<QSpinBox>(editor);
      spinBox->interpretText(); int value = spinBox->value();
      model->setData(index, value, Qt::EditRole);
}
```

Karena view mengelola widget editor untuk delegasi, kita hanya perlu memperbarui model dengan isi yang disediakan editor. Dalam hal ini, kita memastikan bahwa spinbox up-to-date, dan memperbarui model dengan nilai yang dikandungnya menggunakan indeks tertentu.

Kelas QItemDelegate standar memberi tahu view ketika telah selesai mengedit

dengan memancarkan sinyal closeEditor(). View memastikan bahwa widget Editor ditutup dan dihancurkan. Dalam contoh ini, kita hanya menyediakan fasilitas editing sederhana, jadi kita tak perlu memancarkan sinyal ini.

Semua operasi data dilakukan melalui antarmuka yang disediakan oleh QAbstractItemModel. Hal ini membuat delegasi seringnya independen dari jenis data yang memanipulasi, tetapi beberapa perkiraan harus dibuat untuk menggunakan beberapa jenis widget Editor. Dalam contoh ini, kita telah mengasumsikan bahwa model selalu mengandung nilai-nilai integer, tapi kita masih bisa menggunakan delegasi ini dengan berbagai jenis model karena QVariant memberikan nilai default yang masuk akal untuk data tak terduga.

Memperbarui geometri editor

Ini adalah tanggung jawab delegasi untuk mengelola geometri editor. Geometri harus mengatur kapan editor dibuat, dan ketika ukuran item atau posisi dalam tampilan berubah. Untungnya, view memberikan semua informasi geometri yang diperlukan dalam objek pilihan view.

```
void SpinBoxDelegate::updateEditorGeometry(QWidget editor, const
QStyleOptionViewItem &option, const QModelIndex &/* index */) const {
      editor->setGeometry(option.rect);
}
```

Dalam hal ini, kita hanya menggunakan informasi geometri yang disediakan oleh pilihan view pada item persegi panjang. Sebuah delegasi yang menggambar item dengan beberapa elemen tidak akan menggunakan item persegi panjang secara langsung. Ini akan meletakkan editor dalam kaitannya dengan unsur-unsur lain dalam item.

Petunjuk editing

Setelah mengedit, delegasi harus memberikan petunjuk kepada komponen lain tentang hasil dari proses editing, dan memberikan petunjuk yang akan membantu setiap operasi pengeditan berikutnya. Hal ini dicapai dengan memancarkan sinyal closeEditor() dengan petunjuk yang cocok. Hal ini diurus oleh penyaring event QItemDelegate default yang kita instal pada spinbox ketika ia dibangun.

Perilaku kotak berputar dapat disesuaikan untuk membuatnya lebih user friendly. Dalam event filter default yang disediakan oleh QItemDelegate, jika pengguna menekan Enter untuk mengkonfirmasi pilihan mereka di spinbox, delegasi mengubah nilai ke model dan menutup spinbox. Kita dapat mengubah perilaku ini dengan menginstal penyaring event kita sendiri pada spinbox, dan memberikan petunjuk pengeditan yang sesuai dengan kebutuhan kita; misalnya, kita mungkin mengeluarkan closeEditor() dengan petunjuk EditNextItem untuk secara otomatis mulai mengedit item berikutnya dalam tampilan.

Pendekatan lain yang tidak memerlukan penggunaan filter event adalah dengan menyediakan widget editor kita sendiri, mungkin dengan menurunkan kelas QSpinBox

untuk kemudahan. Pendekatan alternatif ini akan memberi kita lebih banyak kontrol atas bagaimana widget Editor berperilaku pada harga menulis kode tambahan. Hal ini biasanya lebih mudah untuk menginstal filter event dalam delegasi jika Anda perlu untuk menyesuaikan perilaku Editor standar Qt widget.

Delegasi tidak perlu memancarkan petunjuk ini, tetapi yang seperti itu akan kurang terintegrasi ke dalam aplikasi, dan akan kurang bermanfaat daripada yang memancarkan petunjuk untuk mendukung tindakan editing umum.

Penanganan pilihan dalam item view

Konsep

Model seleksi yang digunakan dalam kelas item view memberikan gambaran umum tentang pilihan berdasarkan fasilitas dari arsitektur model / view. Meskipun kelas standar untuk memanipulasi pilihan cukup untuk item view yang disediakan, model seleksi memungkinkan Anda untuk membuat model seleksi khusus untuk memenuhi persyaratan untuk model item dan view Anda sendiri.

Informasi tentang item yang dipilih dalam view disimpan dalam sebuah instance dari kelas `QItemSelectionModel`. Ini mempertahankan Model indeks untuk item dalam model tunggal, dan independen dari setiap view. Karena akan ada banyak view ke model, adalah mungkin untuk berbagi pilihan antara view, memungkinkan aplikasi untuk menampilkan beberapa view dalam cara yang konsisten.

Pilihan-pilihan terdiri dari rentang seleksi. Ini secara efisien mempertahankan informasi tentang banyak pilihan item dengan merekam hanya indeks awal dan akhir model untuk setiap rentang item yang dipilih. Pilihan yang tidak berdekatan dari item-item dibangun dengan menggunakan lebih dari satu rentang pilihan untuk menggambarkan pilihan.

Seleksi diterapkan pada koleksi indeks-indeks model yang ditangani oleh model seleksi. Pilihan terakhir dari item yang diterapkan dikenal sebagai pilihan saat ini. Efek dari seleksi ini dapat dimodifikasi bahkan setelah penerapannya melalui penggunaan beberapa jenis perintah seleksi. Ini dibahas kemudian dalam bagian ini.

Item saat ini dan item yang terpilih

Dalam view, selalu ada item saat ini dan item yang terpilih - dua keadaan yang independen. Item dapat menjadi item saat ini dan yang terpilih pada waktu yang sama. View bertanggung jawab untuk memastikan bahwa selalu ada item saat ini sebagai navigasi keyboard, misalnya, membutuhkan item saat ini.

Tabel berikut ini menyoroti perbedaan antara item saat ini dan item yang dipilih.

Item saat ini	Item terpilih
Hanya ada satu item saat ini.	Bisa terdapat beberapa item yang dipilih.
Item saat ini akan berubah dengan tombol navigasi atau tombol klik mouse.	Keadaan item yang dipilih adalah diatur atau diset, tergantung pada beberapa modus yang telah ditetapkan – misalnya, pemilihan tunggal, beberapa seleksi, dll – ketika pengguna berinteraksi dengan item .
Item saat ini akan diedit jika tombol edit, F2, ditekan atau item diklik ganda (asalkan editing diaktifkan) .	Item saat ini dapat digunakan bersama-sama dengan anchor untuk menentukan rentang yang harus dipilih atau terpilih (atau kombinasi dari keduanya) .
Item saat ini ditunjukkan oleh persegi panjang fokus.	Item yang dipilih akan ditandai dengan persegi panjang seleksi.

Ketika memanipulasi pilihan, hal ini sering berguna jika kita memikirkan `QItemSelectionModel` sebagai catatan keadaan pemilihan dari semua item dalam model item. Setelah model seleksi diatur, koleksi dari item-item dapat dipilih, tidak dipilih, atau keadaan pilihan mereka dapat diubah tanpa perlu tahu mana item yang sudah dipilih. Indeks dari semua item yang dipilih dapat diambil sewaktu-waktu, dan komponen lainnya dapat mengetahui perubahan ke model pemilihan melalui mekanisme sinyal dan slot.

Menggunakan model seleksi

Kelas-kelas view standar menyediakan model seleksi standar yang dapat digunakan dalam sebagian besar aplikasi. Sebuah model seleksi milik satu view dapat diperoleh dengan menggunakan fungsi `selectionModel()` milik view, dan dibagi antara banyak view dengan `setSelectionModel()`, sehingga pembangunan model seleksi baru umumnya tidak diperlukan.

Pilihan dibuat dengan menentukan model, dan sepasang indeks model ke `QItemSelection`. Ini menggunakan indeks untuk mengacu pada item dalam model yang diberikan, dan menafsirkan mereka sebagai item kiri-atas dan kanan-bawah di blok item yang dipilih. Untuk menerapkan seleksi untuk item dalam model, pilihan itu perlu untuk disampaikan kepada model seleksi; ini dapat dicapai dalam beberapa cara, masing-masing memiliki efek yang berbeda pada pilihan yang sudah hadir dalam model seleksi.

Memilih item

Untuk menunjukkan beberapa fitur utama dari pemilihan, kita membangun sebuah contoh dari model tabel kustom dengan 32 item totalnya, dan membuka tampilan tabel ke data:

```
TableModel *model = new TableModel(8, 4, &app);
```

```
QTableView *table = new QTableView(0);
table->setModel(model);

QItemSelectionModel *selectionModel = table->selectionModel();
```

Model pemilihan default dari tampilan tabel diambil untuk digunakan nanti. Kita tidak memodifikasi item dalam model, melainkan memilih beberapa item yang akan ditampilkan view di bagian kiri atas tabel. Untuk melakukan ini, kita perlu mengambil indeks model yang sesuai dengan item kiri-atas dan kanan-bawah di daerah yang akan dipilih:

```
QModelIndex topLeft;
QModelIndex bottomRight;

topLeft = model->index(0, 0, QModelIndex());
bottomRight = model->index(5, 2, QModelIndex());
```

Untuk memilih item-item itu dalam model, dan melihat perubahan yang sesuai dalam tampilan tabel, kita perlu membangun objek seleksi kemudian menerapkannya ke model pemilihan:

```
QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

Pemilihan ini diterapkan pada model pemilihan menggunakan perintah yang didefinisikan oleh kombinasi isyarat (flag) seleksi. Dalam hal ini, isyarat yang digunakan menyebabkan item yang tercatat dalam obyek seleksi dimasukkan dalam model seleksi, terlepas dari kondisi sebelumnya. Seleksi yang dihasilkan ditunjukkan oleh view.

Pemilihan item dapat diubah dengan menggunakan berbagai operasi yang didefinisikan oleh isyarat seleksi. Pemilihan yang dihasilkan dari operasi ini mungkin

memiliki struktur yang kompleks, tetapi ditampilkan secara efisien oleh model seleksi. Penggunaan isyarat seleksi yang berbeda untuk memanipulasi item yang dipilih, dijelaskan ketika kita memeriksa cara memperbarui pilihan.

Membaca keadaan seleksi

Indeks-indeks model yang tersimpan dalam model seleksi dapat dibaca dengan menggunakan fungsi selectedIndexes(). Ini mengembalikan sebuah list tak berurut dari indeks model yang dapat kita jabarkan selama kita tahu mereka adalah untuk model yang mana:

```
QModelIndexList indexes = selectionModel->selectedIndexes();
QModelIndex index;

foreach(index, indexes) {
    QString text = QString("(%1,%2)").arg(index.row()).arg(index.column());
    model->setData(index, text);
}
```

Kode di atas menggunakan kata kunci foreach yang mudah milik Qt untuk menjabarkan, dan memodifikasi item-item yang sesuai dengan indeks yang dikembalikan oleh model seleksi.

Model pemilihan memancarkan sinyal untuk menunjukkan perubahan dalam pemilihan. Ini memberitahukan komponen lain tentang perubahan pada baik pemilihan secara keseluruhan maupun item yang difokuskan saat ini dalam model item. Kita dapat menghubungkan sinyal selectionChanged() ke sebuah slot, dan memeriksa item-item dalam model yang dipilih atau tidak terpilih ketika seleksi berubah. Slot ini dipanggil dengan dua objek QItemSelection : satu berisi daftar indeks yang sesuai dengan item yang baru dipilih; yang lain berisi indeks yang sesuai dengan item yang baru saja tidak dipilih.

Dalam kode berikut, kita menyediakan satu slot yang menerima sinyal selectionChanged(), mengisi item yang dipilih dengan string, dan membersihkan isi dari item yang tidak dipilih.

```
void MainWindow::updateSelection(const QItemSelection &selected, const
      QItemSelection &deselected) {
QModelIndex index;
QModelIndexList items = selected.indexes();
foreach (index, items) {
    QString text = QString("(%1,%2)").arg(index.row()).arg(index.column());
    model->setData(index, text);
}

items = deselected.indexes();

foreach (index, items)
```

```
    model->setData(index, "");
}
```

Kita bisa melacak item yang difokuskan saat ini dengan menghubungkan sinyal currentChanged() ke slot yang dipanggil dengan dua indeks-indeks Model. Ini bersesuaian dengan item yang difokuskan sebelumnya, dan item yang difokuskan saat ini.

Dalam kode berikut, kita menyediakan slot yang menerima sinyal currentChanged(), dan menggunakan informasi yang diberikan untuk memperbarui status bar dari QMainWindow :

```
void MainWindow::changeCurrent(const QModelIndex &current, const QModelIndex
      &previous) {
statusBar()->showMessage( tr("Moved from (%1,%2) to (%3,%4)")
      .arg(previous.row()).arg(previous.column())
      .arg(current.row()).arg(current.column()));
}
```

Pemantauan pilihan yang dibuat oleh pengguna adalah langsung dengan sinyal-sinyal ini, tetapi kita juga dapat memperbarui model pemilihan secara langsung.

Memperbarui seleksi

Perintah Seleksi disediakan oleh kombinasi isyarat seleksi, didefinisikan oleh QItemSelectionModel::SelectionFlag. Setiap isyarat seleksi memberi tahu model pemilihan cara memperbarui catatan internal dari item yang dipilih ketika salah satu dari fungsi select() dipanggil. Isyarat yang paling umum digunakan adalah isyarat Select yang menginstruksikan model seleksi untuk merekam item ditetapkan sebagai terpilih. Isyarat Toggle menyebabkan model seleksi membalikkan keadaan item tertentu, memilih item yang tidak dipilih yang diberikan, dan membatalkan pilihan item yang dipilih saat ini. Isyarat Deselect membatalkan pilihan semua item yang ditentukan.

Setiap item dalam model pemilihan diperbarui dengan menciptakan pilihan dari item, dan menerapkannya ke model seleksi. Dalam kode berikut, kita menerapkan pilihan kedua dari item untuk model tabel yang ditunjukkan di atas, dengan menggunakan perintah Toggle untuk membalikkan keadaan pemilihan item yang diberikan.

```
QItemSelection toggleSelection;

topLeft = model->index(2, 1, QModelIndex());
bottomRight = model->index(7, 3, QModelIndex());
toggleSelection.select(topLeft, bottomRight);

selectionModel->select(toggleSelection, QItemSelectionModel::Toggle);
```

Hasil operasi ini akan ditampilkan dalam tampilan tabel, menyediakan cara mudah

untuk memvisualkan apa yang telah kita capai:

Selected Items in a table model

	Column 0	Column 1	Column 2	Column 3
Row 0				
Row 1				
Row 2				
Row 3				
Row 4				
Row 5				
Row 6				
Row 7				

Secara default, perintah-perintah seleksi hanya beroperasi pada masing-masing item yang ditentukan oleh indeks Model. Namun, isyarat yang digunakan untuk menggambarkan perintah seleksi dapat dikombinasikan dengan isyarat tambahan untuk mengubah seluruh baris dan kolom. Sebagai contoh jika Anda memanggil select() dengan hanya satu indeks, tetapi dengan perintah yang merupakan kombinasi dari Select dan Rows, seluruh baris yang berisi item yang dimaksud akan dipilih. Kode berikut menunjukkan penggunaan isyarat Rows dan Columns :

```
QItemSelection columnSelection;

topLeft = model->index(0, 1, QModelIndex());
bottomRight = model->index(0, 2, QModelIndex());

columnSelection.select(topLeft, bottomRight);

selectionModel->select(columnSelection,
    QItemSelectionModel::Select | QItemSelectionModel::Columns);

QItemSelection rowSelection;

topLeft = model->index(0, 0, QModelIndex());
bottomRight = model->index(1, 0, QModelIndex());

rowSelection.select(topLeft, bottomRight);

selectionModel->select(rowSelection,
    QItemSelectionModel::Select | QItemSelectionModel::Rows);
```

Meskipun hanya empat indeks yang disediakan ke model pemilihan, penggunaan isyarat seleksi Rows dan Columns berarti bahwa dua kolom dan dua baris yang dipilih. Gambar berikut menunjukkan hasil dari dua pilihan ini :

Perintah-perintah yang dilakukan pada contoh model telah terlibat semua pengumpulan pilihan dari item dalam model. Hal ini juga mungkin untuk menghapus seleksi, atau untuk mengganti pilihan saat ini dengan yang baru.

Untuk mengganti pilihan saat ini dengan pilihan baru, ggabungkan isyarat seleksi lainnya dengan isyarat `Current`. Perintah yang menggunakan isyarat ini menginstruksikan model seleksi untuk menggantikan koleksi saat ini dari indeks model dengan yang ditetapkan dalam panggilan untuk `select()`. Untuk menghapus semua pilihan sebelum Anda mulai menambahkan yang baru, gabungkan isyarat seleksi lain dengan isyarat `Clear`. Ini memiliki efek me-reset koleksi model pemilihan dari indeks-indeks model.

Memilih semua item dalam model

Untuk memilih semua item dalam model, maka perlu membuat pilihan untuk setiap tingkat dari model yang mencakup semua item dalam tingkat itu. Kita melakukan ini dengan mengambil indeks yang sesuai dengan item kiri-atas dan kanan-bawah dengan indeks induk yang diberikan:

```
QModelIndex topLeft = model->index(0, 0, parent);
QModelIndex bottomRight = model->index(model->rowCount(parent)-1,
    model->columnCount(parent)-1, parent);
```

Pilihan dibangun dengan indeks ini dan modelnya. Item yang sesuai kemudian dipilih dalam model pemilihan :

```
QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

Hal ini perlu dilakukan untuk semua tingkatan dalam model. Untuk item tingkat puncak, kita akan mendefinisikan indeks induk dengan cara biasa :

```
QModelIndex parent = QModelIndex();
```

Untuk model hirarkis, fungsi `hasChildren()` digunakan untuk menentukan apakah

suatu item adalah induk dari tingkat lain dari item.

Membuat model baru

Pemisahan fungsionalitas antara komponen Model / view memungkinkan model bisa dibuat yang dapat memanfaatkan view yang ada. Pendekatan ini memungkinkan kita menyajikan data dari berbagai sumber dengan menggunakan komponen antarmuka pengguna grafis standar, seperti `QListView`, `QTableView`, dan `QTreeView`.

Kelas `QAbstractItemModel` menyediakan antarmuka yang cukup fleksibel untuk mendukung sumber data yang mengatur informasi dalam struktur hirarkis, memungkinkan untuk kemungkinan data itu akan dimasukkan, dihapus, dimodifikasi, atau diurutkan dalam beberapa cara. Hal ini juga menyediakan dukungan untuk operasi drag dan drop.

Kelas `QAbstractListModel` dan `QAbstractTableModel` memberikan dukungan untuk interface kepada struktur data non-hirarkis sederhana, dan lebih mudah untuk digunakan sebagai titik awal untuk model daftar dan tabel sederhana.

Pada bagian ini, kita membuat model *read-only* sederhana untuk mengeksplorasi prinsip-prinsip dasar dari arsitektur model / view. Kemudian dalam bagian ini, kita mengadaptasi model sederhana ini agar item dapat dimodifikasi oleh pengguna.

Untuk contoh model yang lebih kompleks, lihat contoh Model Pohon Sederhana.

Persyaratan kelas turunan `QAbstractItemModel` dijelaskan secara lebih rinci dalam bagian Referensi Subclassing Model.

Merancang model

Ketika membuat model baru untuk struktur data yang ada, penting untuk mempertimbangkan jenis model yang harus digunakan untuk menyediakan sebuah antarmuka ke data. Jika struktur data dapat direpresentasikan sebagai daftar atau tabel dari item-item, Anda dapat menurunkan kelas `QAbstractListModel` atau `QAbstractTableModel` karena kelas-kelas ini menyediakan implementasi standar yang cocok untuk banyak fungsi.

Namun, jika struktur data yang mendasarinya hanya dapat diwakili oleh struktur pohon hirarkis, perlu menurunkan kelas `QAbstractItemModel`. Pendekatan ini dicatat di contoh Model Pohon Sederhana.

Pada bagian ini, kita menerapkan model sederhana berdasarkan daftar string, sehingga `QAbstractListModel` menyediakan kelas dasar yang ideal untuk membangunnya.

Apapun bentuk yang diambil oleh struktur data yang mendasarinya, biasanya ini ide yang baik untuk melengkapi API standar `QAbstractItemModel` dalam model khusus dengan satu yang memungkinkan akses yang lebih alami untuk struktur data yang mendasarinya. Hal ini membuat lebih mudah untuk mengisi model dengan data,

namun masih memungkinkan komponen umum Model / view lain untuk berinteraksi dengan menggunakan API standar. Model yang digambarkan di bawah ini menyediakan konstruktor kustom untuk tujuan ini.

Contoh model read-only

Model yang diterapkan di sini adalah model data read-only sederhana, non-hirarkis, berdasarkan kelas `QStringListModel` standar. Ada kelas `QStringList` sebagai sumber data internal, dan menerapkan hanya apa yang dibutuhkan untuk membuat model yang berfungsi. Untuk membuat implementasinya lebih mudah, kita menurunkan kelas `QAbstractListModel` karena dia mendefinisikan perilaku default yang masuk akal untuk model daftar, dan dia menampilkan antarmuka yang lebih sederhana daripada kelas `QAbstractItemModel`.

Ketika menerapkan model, penting untuk diingat bahwa `QAbstractItemModel` tidak menyimpan data apapun sendiri, dia hanya menyajikan antarmuka yang digunakan view untuk mengakses data. Untuk model read-only minimal, hanya perlu menerapkan beberapa fungsi karena ada implementasi default untuk sebagian besar antarmuka. Deklarasi kelas adalah sebagai berikut:

```
class StringListModel : public QAbstractListModel
{
    Q_OBJECT

public:
    StringListModel(const QStringList &strings, QObject *parent = 0)
        : QAbstractListModel(parent), stringList(strings) {}

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

private:
    QStringList stringList;
};
```

Terlepas dari konstruktor model, kita hanya perlu mengimplementasikan dua fungsi: `rowCount()` mengembalikan jumlah baris dalam model dan `data()` mengembalikan sebuah item data yang sesuai dengan indeks model yang ditentukan.

Model yang bagus juga menerapkan `headerData()` untuk memberikan sesuatu untuk ditampilkan kepada view pohon dan tabel di header mereka.

Perhatikan bahwa ini adalah model non-hirarkis, sehingga kita tidak perlu khawatir tentang hubungan induk-anak. Jika model kita adalah hirarkis, kita juga harus menerapkan fungsi `index()` dan `parent()`.

Daftar string disimpan secara internal di variabel anggota pribadi stringList.

Dimensi model

Kita ingin jumlah baris dalam model agar menjadi sama dengan jumlah string dalam daftar string. Kita menerapkan fungsi rowCount() dengan seperti ini:

```
int StringListModel::rowCount(const QModelIndex &parent) const
{
    return stringList.count();
}
```

Karena model ini non-hirarkis, kita dapat mengabaikan indeks model yang sesuai dengan item induk. Secara default, model yang berasal dari QAbstractListModel hanya berisi satu kolom, jadi kita tidak perlu reimplement fungsi columnCount().

Model header dan data

Untuk item dalam view ini, kita ingin mengembalikan string dalam daftar string. Fungsi data() bertanggung jawab untuk mengembalikan item data yang sesuai dengan argumen indeks:

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

    if (index.row() >= stringList.size())
        return QVariant();

    if (role == Qt::DisplayRole)
        return stringList.at(index.row());
    else
        return QVariant();
}
```

Kita hanya mengembalikan QVariant yang valid jika indeks model yang disediakan valid, nomor baris adalah dalam kisaran item dalam daftar string, dan peran yang diminta adalah salah satu yang kita dukung.

Beberapa view, seperti QTreeView dan QTableView, mampu menampilkan header bersama dengan data item. Jika model kita ditampilkan dalam tampilan dengan header, kita ingin header untuk menunjukkan nomor baris dan kolom. Kita dapat memberikan informasi tentang header dengan menurunkan fungsi headerData():

```
QVariant StringListModel::headerData(int section, Qt::Orientation orientation,
                                     int role) const
{
```

```
    if (role != Qt::DisplayRole)
        return QVariant();

    if (orientation == Qt::Horizontal)
        return QString("Column %1").arg(section);
    else
        return QString("Row %1").arg(section);
}
```

Sekali lagi, kita mengembalikan QVariant valid hanya jika peran adalah salah satu yang kita dukung. Orientasi header juga diperhitungkan ketika memutuskan data kembalian yang tepat.

Tidak semua view menampilkan header dengan data item, dan yang bisa, mungkin dikonfigurasi untuk menyembunyikan mereka. Meskipun demikian, disarankan agar Anda menerapkan fungsi `headerData()` untuk memberikan informasi yang relevan tentang data yang diberikan oleh model.

Item dapat memiliki beberapa peran, memberikan data yang berbeda tergantung pada peran tertentu. Item dalam model kita hanya memiliki satu peran, DisplayRole, jadi kita mengembalikan data untuk item terlepas dari peran yang ditentukan. Namun, kita bisa menggunakan kembali data yang kita sediakan untuk DisplayRole dalam peran lain, seperti ToolTipRole yang dapat digunakan oleh view untuk menampilkan informasi tentang item dalam tooltip.

Model yang dapat diedit

Model read-only menunjukkan bagaimana pilihan sederhana dapat disajikan kepada pengguna tetapi untuk banyak aplikasi, model daftar yang dapat diedit jauh lebih berguna. Kita dapat memodifikasi model read-only untuk membuat item dapat diedit dengan mengubah fungsi `data()` yang ki terapkan untuk read-only, dan dengan menerapkan dua fungsi tambahan: `flags()` dan `setData()`. Deklarasi fungsi berikut ditambahkan ke definisi kelas:

```
    Qt::ItemFlags flags(const QModelIndex &index) const;
    bool setData(const QModelIndex &index, const QVariant &value,
                 int role = Qt::EditRole);
```

Membuat model dapat diedit

Sebuah delegasi memeriksa apakah item dapat diedit sebelum membuat editor. Model ini harus membiarkan delegasi tahu bahwa item-itemnya dapat diedit. Kita melakukan ini dengan mengembalikan isyarat (flag) yang benar untuk setiap item dalam model; dalam kasus ini, kita mengaktifkan semua item dan membuat mereka berduadapat dipilih dan diedit:

```
Qt::ItemFlags StringListModel::flags(const QModelIndex &index) const
```

```
{
    if (!index.isValid())
        return Qt::ItemIsEnabled;

    return QAbstractItemModel::flags(index) | Qt::ItemIsEditable;
}
```

Perhatikan bahwa kita tidak perlu tahu bagaimana delegasi melakukan proses editing yang sebenarnya. Kita hanya harus menyediakan cara bagi delegasi untuk mengatur data dalam model. Hal ini dicapai melalui fungsi setData() :

```
bool StringListModel::setData(const QModelIndex &index,
                              const QVariant &value, int role)
{
    if (index.isValid() && role == Qt::EditRole) {

        stringList.replace(index.row(), value.toString());
        emit dataChanged(index, index);
        return true;
    }
    return false;
}
```

Dalam model ini, item dalam daftar string yang sesuai dengan indeks model diganti dengan nilai yang diberikan. Namun, sebelum kita dapat memodifikasi daftar string, kita harus memastikan bahwa indeks tersebut valid, item tersebut dari jenis yang tepat, dan peran itu didukung. Dengan kaidah, kita bersikukuh bahwa peran itu adalah EditRole karena ini adalah peran yang digunakan oleh delegasi item standar. Tetapi untuk nilai-nilai boolean, Anda dapat menggunakan Qt::CheckStateRole dan atur isyarat Qt::ItemIsUserCheckable; kotak centang kemudian digunakan untuk mengedit nilai. Data dasar dalam model ini adalah sama untuk semua peran, sehingga detail ini membuat lebih mudah untuk mengintegrasikan model dengan komponen standar.

Ketika data telah ditetapkan, model harus memberi tahu view bahwa beberapa data telah berubah. Hal ini dilakukan dengan memancarkan sinyal dataChanged(). Karena hanya satu item data yang telah berubah, kisaran item yang ditentukan dalam sinyal terbatas hanya pada satu indeks Model.

Juga fungsi data() perlu diubah untuk menambahkan tes Qt::EditRole:

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

    if (index.row() >= stringList.size())
        return QVariant();
```

```
    if (role == Qt::DisplayRole || role == Qt::EditRole)
        return stringList.at(index.row());
    else
        return QVariant();
}
```

Menyisipkan dan menghapus baris

Ini memungkinkan untuk mengubah jumlah baris dan kolom dalam model. Dalam model daftar string hanya masuk akal untuk mengubah jumlah baris, jadi kita hanya reimplement fungsi untuk memasukkan dan mengeluarkan baris. Ini dinyatakan dalam definisi kelas:

```
  bool insertRows(int position, int rows, const QModelIndex &index =
QModelIndex());
  bool removeRows(int position, int rows, const QModelIndex &index =
QModelIndex());
```

Karena baris dalam model ini sesuai dengan string dalam daftar, fungsi `insertRows()` menyisipkan sejumlah string kosong ke dalam daftar string sebelum posisi yang ditentukan. Jumlah string yang dimasukkan setara dengan jumlah baris yang ditentukan.

Indeks induk biasanya digunakan untuk menentukan di mana baris harus ditambahkan dalam model. Dalam kasus ini, kita hanya memiliki daftar top-level tunggal dari string-string, jadi kita hanya memasukkan string kosong ke dalam daftar itu.

```
bool StringListModel::insertRows(int position, int rows, const QModelIndex
&parent)
{
    beginInsertRows(QModelIndex(), position, position+rows-1);

    for (int row = 0; row < rows; ++row) {
        stringList.insert(position, "");
    }

    endInsertRows();
    return true;
}
```

Model ini memanggil fungsi `beginInsertRows()` dulu untuk menginformasikan komponen lain bahwa jumlah baris akan berubah. Fungsi ini menentukan nomor baris dari baris baru pertama dan terakhir yang akan dimasukkan, dan indeks model untuk item induknya. Setelah mengubah daftar string, dia memanggil `endInsertRows()` untuk

menyelesaikan operasi dan menginformasikan komponen lain bahwa dimensi model telah berubah, mengembalikan true yang menunjukkan sukses.

Fungsi untuk menghapus baris dari model ini juga sederhana untuk ditulis. Baris yang dihapus dari model ditentukan oleh posisi dan jumlah baris yang diberikan. Kita mengabaikan indeks induk untuk menyederhanakan implementasi kita, dan hanya menghapus item yang sesuai dari daftar string.

```
bool StringListModel::removeRows(int position, int rows, const QModelIndex &parent)
{
    beginRemoveRows(QModelIndex(), position, position+rows-1);

    for (int row = 0; row < rows; ++row) {
        stringList.removeAt(position);
    }

    endRemoveRows();
    return true;
}
```

Fungsi `beginRemoveRows()` selalu dipanggil sebelum data yang mendasarinya dihapus, dan menentukan baris pertama dan terakhir yang akan dihapus. Hal ini memungkinkan komponen lain untuk mengakses data sebelum menghilang. Setelah baris dihapus, model memancarkan `endRemoveRows()` untuk menyelesaikan operasi dan memberi tahu komponen lain bahwa dimensi model telah berubah.

Langkah berikutnya

Kita dapat menampilkan data yang diberikan oleh model ini, atau model lain, menggunakan kelas `QListView` untuk menyajikan item-itemnya model dalam bentuk daftar vertikal. Untuk model daftar string, view ini juga menyediakan editor default sehingga item bisa dimanipulasi. Kita meneliti kemungkinan yang disediakan oleh kelas standar view dalam bagian Kelas-kelas View.

Dokumen Model Subclassing Reference membahas persyaratan turunan-turunan `QAbstractItemModel` secara lebih rinci, dan menyediakan panduan untuk fungsi virtual yang harus dilaksanakan untuk mengaktifkan berbagai fitur dalam berbagai jenis model.

Kelas Item View yang Mudah

Widget berbasis-item memiliki nama yang mencerminkan kegunaannya: `QListWidget` menyediakan daftar item, `QTreeWidget` menampilkan struktur pohon multi-level, dan `QTableWidget` menyediakan tabel dari item-item sel. Setiap kelas mewarisi perilaku dari kelas `QAbstractItemView` yang mengimplementasikan perilaku

umum untuk pemilihan item dan manajemen header.

Widget Daftar

Daftar tingkat tunggal dari item biasanya ditampilkan menggunakan QListWidget dan sejumlah QListWidgetItem. Sebuah widget Daftar dibangun dengan cara yang sama seperti widget lainnya :

```
QListWidget *listWidget = new QListWidget(this);
```

Daftar item dapat ditambahkan langsung ke widget daftar ketika mereka dibangun:

```
    new QListWidgetItem(tr("Sycamore"), listWidget);
    new QListWidgetItem(tr("Chestnut"), listWidget);
    new QListWidgetItem(tr("Mahogany"), listWidget);
```

Mereka juga dapat dibangun tanpa widget daftar induk dan ditambahkan ke daftar pada beberapa waktu kemudian:

```
    QListWidgetItem *newItem = new QListWidgetItem;
    newItem->setText(itemText);
    listWidget->insertItem(row, newItem);
```

Setiap item dalam daftar dapat menampilkan label teks dan ikon. Warna dan font yang digunakan untuk menggambar teks dapat diubah untuk memberikan penampilan yang disesuaikan untuk item. Tooltips, status tip, dan bantuan "Apa Ini?" semuanya mudah dikonfigurasi untuk memastikan bahwa daftar tersebut terintegrasi ke dalam aplikasi dengan tepat.

```
    newItem->setToolTip(toolTipText);
    newItem->setStatusTip(toolTipText);
    newItem->setWhatsThis(whatsThisText);
```

Secara default, item dalam daftar disajikan dalam urutan penciptaan mereka. Daftar item dapat diurutkan sesuai dengan kriteria yang diberikan dalam Qt::SortOrder untuk menghasilkan daftar item yang diurutkan dalam urutan abjad maju atau mundur:

```
    listWidget->sortItems(Qt::AscendingOrder);
    listWidget->sortItems(Qt::DescendingOrder);
```

Widget pohon

Pohon atau daftar hirarkis dari item disediakan oleh kelas QTreeWidget dan QTreeWidgetItem. Setiap item dalam widget pohon dapat memiliki item anak sendiri, dan dapat menampilkan jumlah kolom informasi. Widget pohon diciptakan sama seperti widget lainnya:

```
QTreeWidget *treeWidget = new QTreeWidget(this);
```

Sebelum item bisa ditambahkan ke widget pohon, jumlah kolom harus diset. Sebagai contoh, kita bisa mendefinisikan dua kolom, dan membuat header untuk memberikan label di bagian atas setiap kolom:

```
    treeWidget->setColumnCount(2);
    QStringList headers;
    headers << tr("Subject") << tr("Default");
    treeWidget->setHeaderLabels(headers);
```

Cara termudah untuk mengatur label untuk setiap bagian adalah dengan menyediakan daftar string. Untuk header yang lebih canggih, Anda dapat membuat item pohon, menghias sesuai keinginan, dan menggunakannya sebagai headernya widget pohon.

Item tingkat puncak dalam widget pohon dibangun dengan widget pohon sebagai widget induknya. Mereka dapat dimasukkan dalam urutan bebas, atau Anda dapat memastikan bahwa mereka terdaftar dalam urutan tertentu dengan menentukan item sebelumnya ketika membangun setiap item:

```
    QTreeWidgetItem *cities = new QTreeWidgetItem(treeWidget);
    cities->setText(0, tr("Cities"));
    QTreeWidgetItem *osloItem = new QTreeWidgetItem(cities);
    osloItem->setText(0, tr("Oslo"));
    osloItem->setText(1, tr("Yes"));

    QTreeWidgetItem *planets = new QTreeWidgetItem(treeWidget, cities);
```

Widget pohon yang berurusan dengan item tingkat atas sedikit berbeda untuk item-item lainnya dari yang lebih dalam di pohon. Item-item dapat dihapus dari tingkat puncak pohon dengan memanggil fungsi `takeTopLevelItem()` milik widget pohon, tapi item dari tingkat yang lebih rendah akan dihapus dengan memanggil fungsi `takeChild()` milik item induk mereka. Item-item dimasukkan ke dalam tingkat atas pohon dengan fungsi `insertTopLevelItem()`. Pada tingkat yang lebih rendah di pohon, fungsi `insertChild()` milik item induk digunakan.

Sangat mudah untuk memindahkan item antara tingkat puncak dan tingkat yang lebih rendah di pohon. Kita hanya perlu memeriksa apakah item itu item top-level atau tidak, dan informasi ini diberikan oleh fungsi `parent()` milik masing-masing item. Sebagai contoh, kita dapat menghapus item saat ini di widget pohon tidak peduli pada lokasinya:

```
     QTreeWidgetItem *parent = currentItem->parent();
    int index;

    if (parent) {
```

```
        index = parent->indexOfChild(treeWidget->currentItem());
        delete parent->takeChild(index);
    } else {
        index = treeWidget->indexOfTopLevelItem(treeWidget->currentItem());
        delete treeWidget->takeTopLevelItem(index);
    }
```

Memasukkan item di tempat lain di widget pohon mengikuti pola yang sama:

```
     QTreeWidgetItem *parent = currentItem->parent();
    QTreeWidgetItem *newItem;
    if (parent)
        newItem = new QTreeWidgetItem(parent, treeWidget->currentItem());
    else
        newItem = new QTreeWidgetItem(treeWidget, treeWidget->currentItem());
```

Widget tabel

Tabel item-item yang serupa dengan yang ditemukan dalam aplikasi spreadsheet, dibangun dengan QTableWidget dan QTableWidgetItem. Ini menyediakan widget tabel bergulir dengan header dan item untuk digunakan di dalamnya.

Tabel dapat dibuat dengan sejumlah set baris dan kolom, atau ini dapat ditambahkan ke tabel tak berukuran sebagaimana yang diperlukan.

```
    QTableWidget *tableWidget;
    tableWidget = new QTableWidget(12, 3, this);
```

Item dibuat di luar tabel sebelum ditambahkan ke tabel di lokasi yang dibutuhkan:

```
    QTableWidgetItem *newItem = new QTableWidgetItem(tr("%1").arg(
        pow(row, column+1)));
    tableWidget->setItem(row, column, newItem);
```

Header horisontal dan vertikal dapat ditambahkan ke tabel dengan membangun item luar tabel dan menggunakan mereka sebagai header:

```
    QTableWidgetItem *valuesHeaderItem = new QTableWidgetItem(tr("Values"));
    tableWidget->setHorizontalHeaderItem(0, valuesHeaderItem);
```

Perhatikan bahwa baris dan kolom dalam tabel dimulai dari nol.

Fitur-fitur umum

Ada sejumlah fitur berbasis-item yang umum untuk masing-masing kelas-kelas mudah yang tersedia melalui antarmuka yang sama di masing-masing kelas. Kami menyajikan ini pada bagian berikut ini dengan beberapa contoh untuk widget yang

berbeda. Lihatlah daftar *Kelas-kelas Model / View* untuk masing-masing widget untuk rincian lebih lanjut tentang penggunaan masing-masing fungsi yang digunakan.

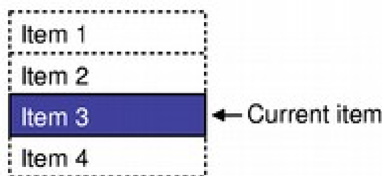
Item yang tersembunyi

Kadang-kadang berguna untuk dapat menyembunyikan item dalam tampilan widget item daripada menghapusnya. Item untuk semua widget di atas dapat disembunyikan dan kemudian ditampilkan lagi. Anda dapat menentukan apakah item itu tersembunyi dengan memanggil fungsi `isItemHidden()`, dan item dapat disembunyikan dengan `setItemHidden()`.

Karena operasi ini berbasis item, fungsi yang sama ini tersedia untuk ketiga kelas mudah.

Seleksi

Cara item yang dipilih dikendalikan oleh modus seleksi widget (`QAbstractItemView::SelectionMode`). Properti ini mengatur apakah pengguna dapat memilih satu atau banyak item dan dalam pilihan banyak item, apakah seleksi harus jajaran yang berkesinambungan dari item-item. Modus seleksi bekerja dengan cara yang sama untuk semua widget di atas.



Pilihan item tunggal: Dimana pengguna harus memilih satu item dari widget, modus `SingleSelection` default adalah yang paling sesuai. Dalam modus ini, item saat ini dan item yang dipilih adalah sama.

System	Monitor	OS
Blue	15"	Linux
Gray	17"	OS X
White	19"	Windows

Pilihan multi-item: Dalam modus ini, pengguna dapat mengaktifkan keadaan pemilihan item di widget tanpa mengubah pilihan yang ada, sangat mirip dengan cara checkbox non-eksklusif dapat diaktifkan secara independen.

Items	Prices	Tax
Parts	100	5%
Widgets	56	1%
Service	112	22%

Pilihan diperluas: Widget yang sering memerlukan banyak item yang berdekatan untuk dipilih, seperti yang ditemukan dalam spreadsheet, memerlukan modus `ExtendedSelection`. Dalam modus ini, rentang berurutan dari item dalam widget dapat

dipilih dengan mouse maupun keyboard. Pilihan yang kompleks, yang melibatkan banyak item yang tidak berdekatan dengan item-item lainnya yang dipilih dalam widget, juga bisa dibuat jika tombol pengubah digunakan. Jika pengguna memilih item tanpa menggunakan tombol pengubah, pemilihan yang ada akan dihapus.

Item yang dipilih di widget dibaca menggunakan fungsi selectedItems(), memberikan daftar item yang relevan yang dapat di-iterasi. Sebagai contoh, kita dapat menemukan jumlah dari semua nilai-nilai numerik dalam daftar item yang dipilih dengan kode berikut:

```
QList<QTableWidgetItem *> selected = tableWidget->selectedItems();
QTableWidgetItem *item;
int number = 0;
double total = 0;

foreach (item, selected) {
    bool ok;
    double value = item->text().toDouble(&ok);

    if (ok && !item->text().isEmpty()) {
        total += value;
        number++;
    }
}
```

Perhatikan bahwa untuk mode seleksi tunggal, item saat ini akan ada dalam seleksi. Dalam modus multi-seleksi dan seleksi diperluas, item saat ini mungkin tidak terletak dalam pemilihan, tergantung pada cara pengguna membentuk seleksi .

Mencari

Hal ini sering berguna untuk dapat menemukan item dalam widget item view, baik sebagai pengembang atau pun sebagai layanan untuk menyajikan kepada pengguna. Ketiga kelas-kelas mudah item view itu memberikan fungsi umum findItems() untuk membuat ini sekonsisten dan sesederhana mungkin.

Item-item dicari oleh teks yang mengandung sesuai dengan kriteria yang ditentukan oleh pemilihan nilai dari Qt::MatchFlags. Kita bisa mendapatkan daftar pencocokan item dengan fungsi findItems():

```
QTreeWidgetItem *item;
QList<QTreeWidgetItem *> found = treeWidget->findItems(
    itemText, Qt::MatchWildcard);

foreach (item, found) {
    treeWidget->setItemSelected(item, true);
    // Show the item->text(0) for each item.
```

```
    }
```

Kode di atas menyebabkan item dalam widget pohon dipilih jika mereka berisi teks yang diberikan dalam string pencarian. Pola ini juga dapat digunakan dalam widget daftar dan tabel.

Menggunakan Drag dan Drop dengan Item Views

Infrastruktur Drag dan drop Qt didukung penuh oleh kerangka model / view. Item dalam daftar, tabel, dan pohon dapat di-*drag* di dalam view, dan data dapat diimpor dan diekspor sebagai data MIME-encoded.

View standar otomatis mendukung drag dan drop internal, di mana item-item dipindahkan untuk mengubah urutan di mana mereka ditampilkan. Secara default, drag dan drop tidak diaktifkan untuk view ini karena mereka dikonfigurasi untuk kesederhanaan, penggunaan yang paling umum. Untuk memungkinkan item untuk didrag, properti tertentu dari view perlu diaktifkan, dan item sendiri juga harus memungkinkan terjadinya drag.

Persyaratan untuk sebuah model yang hanya memungkinkan item untuk diekspor dari view, dan yang tidak memungkinkan data di-drop ke dalamnya, adalah lebih sedikit dibandingkan untuk model yang sepenuhnya mengaktifkan drag dan drop.

Lihat juga *Referensi Subclassing Model* Untuk informasi lebih lanjut tentang mengaktifkan dukungan drag and drop dalam model-model baru.

Menggunakan views yang mudah

Masing-masing jenis item yang digunakan dengan `QListWidget`, `QTableWidget`, dan `QTreeWidget` dikonfigurasi untuk menggunakan satu set isyarat yang berbeda secara default. Misalnya, setiap `QListWidgetItem`atau `QTreeWidgetItem` awalnya diaktifkan, checkable, dipilih, dan dapat digunakan sebagai sumber operasi drag dan drop; setiap `QTableWidgetItem` juga dapat diedit dan digunakan sebagai target operasi drag dan drop.

Meskipun semua item standar memiliki salah satu isyarat atau dua-duanya yang ditetapkan untuk drag dan drop, biasanya Anda perlu mengatur berbagai properti dalam view itu sendiri untuk memanfaatkan dukungan built-in untuk drag dan drop:

1. Untuk mengaktifkan drag item, atur properti `dragEnabled` dari view menjadi `true`.
2. Untuk memungkinkan pengguna bisa drop item baik internal maupun eksternal dalam view, atur view di properti `acceptDrops` milik `viewport()` menjadi `true`.
3. Untuk menunjukkan pada pengguna di mana item saat ini sedang didrag akan ditempatkan jika didrop, atur properti `showDropIndicator` milik view. Ini menyediakan informasi yang terus diperbarui untuk pengguna mengenai

penempatan item dalam view.

Sebagai contoh, kita dapat mengaktifkan drag and drop dalam widget daftar dengan baris kode berikut:

```
QListWidget *listWidget = new QListWidget(this);
listWidget->setSelectionMode(QAbstractItemView::SingleSelection);
listWidget->setDragEnabled(true);
listWidget->viewport()->setAcceptDrops(true);
listWidget->setDropIndicatorShown(true);
```

Hasilnya adalah widget daftar yang memungkinkan item untuk disalin ke dalam view, dan bahkan memungkinkan pengguna mendrag item antara tampilan yang berisi jenis data yang sama. Dalam kedua situasi itu, item disalin bukannya dipindah.

Untuk memungkinkan pengguna untuk memindahkan item-item ke dalam view, kita harus mengatur `dragDropMode` widget daftar:

```
listWidget->setDragDropMode(QAbstractItemView::InternalMove);
```

Menggunakan kelas-kelas model / view

Menyiapkan tampilan untuk drag dan drop mengikuti pola yang sama yang digunakan dengan views yang mudah. Sebagai contoh, `QListView` dapat diatur dengan cara yang sama sebagaimana `QListWidget`:

```
QListView *listView = new QListView(this);
listView->setSelectionMode(QAbstractItemView::ExtendedSelection);
listView->setDragEnabled(true);
listView->setAcceptDrops(true);
listView->setDropIndicatorShown(true);
```

Karena akses ke data yang ditampilkan oleh view dikendalikan oleh model, model yang digunakan juga harus memberikan dukungan untuk operasi drag dan drop. Tindakan yang didukung oleh model dapat ditentukan dengan mengimplementasi ulang fungsi `QAbstractItemModel::supportedDropActions()`. Misalnya, operasi menyalin dan memindahkan diaktifkan dengan kode berikut:

```
Qt::DropActions DragDropListModel::supportedDropActions() const
{
    return Qt::CopyAction | Qt::MoveAction;
}
```

Meskipun setiap kombinasi nilai dari `Qt::DropActions` dapat diberikan, model harus ditulis untuk mendukung mereka. Misalnya, untuk memungkinkan `Qt::MoveAction` untuk digunakan dengan baik dengan model daftar, model harus menyediakan sebuah implementasi dari `QAbstractItemModel::removeRows()` , baik secara langsung maupun dengan mewarisi implementasi dari kelas dasar.

Mengaktifkan drag dan drop untuk item

Model menunjukkan pada view item mana yang dapat didrag, dan yang akan menerima drop, dengan mengimplementasi ulang fungsi QAbstractItemModel::flags() untuk memberikan isyarat yang cocok.

Sebagai contoh, sebuah model yang menyediakan daftar sederhana berdasarkan QAbstractListModel dapat mengaktifkan drag and drop untuk masing-masing item dengan memastikan bahwa isyarat yang dikembalikan berisi nilai Qt::ItemIsDragEnabled dan Qt::ItemIsDropEnabled:

```
Qt::ItemFlags DragDropListModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags defaultFlags = QStringListModel::flags(index);

    if (index.isValid())
        return Qt::ItemIsDragEnabled | Qt::ItemIsDropEnabled | defaultFlags;
    else
        return Qt::ItemIsDropEnabled | defaultFlags;
}
```

Perhatikan bahwa item bisa didrop ke tingkat puncak dari model, tapi drag hanya diaktifkan untuk item yang valid.

Dalam kode di atas, karena model ini berasal dari QStringListModel, kita mendapatkan satu set standar isyarat dengan memanggil implementasi dari fungsi flags().

Meng-encode data yang diekspor

Ketika item-item data diekspor dari model dalam operasi drag dan drop, mereka dikodekan ke dalam format yang tepat, sesuai dengan satu jenis MIME atau lebih. Model menyatakan jenis MIME yang dapat mereka gunakan untuk memasok item dengan reimplementasi fungsi QAbstractItemModel::mimetypes(), mengembalikan daftar jenis MIME standar.

Sebagai contoh, sebuah model yang hanya menyediakan teks biasa akan memberikan implementasi berikut:

```
QStringList DragDropListModel::mimeTypes() const
{
    QStringList types;
    types << "application/vnd.text.list";
    return types;
}
```

Model ini juga harus memberikan kode untuk menyandikan (encode) data dalam format yang diperlihatkan. Hal ini dicapai dengan implementasi ulang fungsi

QAbstractItemModel::mimeData() untuk menyediakan objek QMimeData, seperti dalam setiap operasi drag dan drop.

Kode berikut menunjukkan bagaimana setiap item data yang bersesuaian dengan daftar dari indeks yang diberikan, dikodekan sebagai teks biasa dan disimpan dalam sebuah objek QMimeData.

```
QMimeData *DragDropListModel::mimeData(const QModelIndexList &indexes) const
{
    QMimeData *mimeData = new QMimeData();
    QByteArray encodedData;

    QDataStream stream(&encodedData, QIODevice::WriteOnly);

    foreach (const QModelIndex &index, indexes) {
        if (index.isValid()) {
            QString text = data(index, Qt::DisplayRole).toString();
            stream << text;
        }
    }

    mimeData->setData("application/vnd.text.list", encodedData);
    return mimeData;
}
```

Karena daftar dari indeks-indeks model disediakan ke fungsi, pendekatan ini cukup umum untuk digunakan dalam model hirarkis maupun non-heirarchical.

Perhatikan bahwa tipe data kustom harus dinyatakan sebagai obyek meta dan operator stream itu harus diimplementasikan untuk mereka. Lihat deskripsi kelas QMetaObject untuk rinciannya.

Memasukkan data yang didrop ke dalam model

Cara setiap model yang diberikan menangani data yang didrop tergantung pada jenis keduanya (daftar, tabel, atau pohon) dan cara isinya mungkin akan disajikan kepada pengguna. Secara umum, pendekatan yang dilakukan untuk mengakomodasi data yang didrop seharusnya salah satu yang paling sesuai dengan penampungan data yang mendasari model.

Berbagai jenis model cenderung menangani data yang didrop dengan cara yang berbeda. Model daftar dan tabel hanya menyediakan struktur datar di mana item data disimpan. Akibatnya, mereka dapat menyisipkan baris baru (dan kolom) ketika data dijatuhkan pada item yang ada dalam view, atau mereka mungkin menimpa isi item dalam model menggunakan beberapa data yang diberikan. Model pohon sering dapat menambahkan item anak yang berisi data baru ke penampungan data yang mendasarinya, dan karena itu akan berperilaku lebih dapat diduga sejauh pengguna

mempedulikan.

Data yang didrop ditangani oleh implementasi ulang sebuah model dari QAbstractItemModel::dropMimeData(). Sebagai contoh, sebuah model yang menangani daftar sederhana dari string dapat menyediakan implementasi yang menangani data yang jatuh ke item yang ada secara terpisah untuk data yang jatuh ke tingkat atas dari model (yaitu, ke item yang tidak valid).

Model ini harus memastikan dahulu bahwa operasi harus dipatuhi, data yang disediakan dalam format yang dapat digunakan, dan bahwa tujuan dalam model ini valid:

```
bool DragDropListModel::dropMimeData(const QMimeData *data,
    Qt::DropAction action, int row, int column, const QModelIndex &parent)
{
    if (action == Qt::IgnoreAction)
        return true;

    if (!data->hasFormat("application/vnd.text.list"))
        return false;

    if (column > 0)
        return false;
```

Sebuah model daftar string satu kolom sederhana dapat mengindikasikan kegagalan jika data yang diberikan bukan teks biasa, atau jika jumlah kolom yang diberikan untuk drop tidak valid.

Data yang akan dimasukkan ke dalam model diperlakukan berbeda tergantung pada apakah itu dijatuhkan ke item yang sudah ada atau tidak. Dalam contoh sederhana ini, kita ingin mengizinkan drop antara item-item yang ada, sebelum item pertama dalam daftar, dan setelah item terakhir.

Bila didrop, indeks model yang sesuai dengan item induk juga akan valid, menunjukkan bahwa drop terjadi pada item, atau akan menjadi tidak valid, menunjukkan bahwa drop terjadi di suatu tempat dalam view yang sesuai dengan tingkat atas dari model.

```
    int beginRow;

    if (row != -1)
        beginRow = row;
```

Kita awalnya memeriksa nomor baris yang disediakan untuk melihat apakah kita dapat menggunakannya untuk memasukkan item-item ke dalam model, terlepas dari apakah indeks induk valid atau tidak .

```
    else if (parent.isValid())
```

```
        beginRow = parent.row();
```

Jika indeks Model induk valid, drop terjadi pada item. Dalam model daftar sederhana ini, kita mengetahui jumlah baris item dan menggunakan nilai tersebut untuk memasukkan item-itme yang didrop ke tingkat atas dari model.

```
    else
        beginRow = rowCount(QModelIndex());
```

Bila terjatuh di tempat lain di view, dan nomor baris tidak dapat digunakan, kita tambahkan item ke tingkat atas dari model.

Dalam model hirarkis, ketika drop terjadi pada item, akan lebih baik untuk memasukkan item baru ke dalam model sebagai anak-anak dari item tersebut. Pada contoh sederhana yang ditunjukkan di sini, model hanya memiliki satu tingkat, sehingga pendekatan ini tidak tepat .

Decoding Data yang diimpor

Setiap pelaksanaan `dropMimeData()` juga harus men-*decode* data dan masukkan ke dalam struktur data yang mendasari model.

Untuk model daftar string sederhana, item yang dikodekan dapat diterjemahkan dan di-*stream* ke `QStringList`:

```
    QByteArray encodedData = data->data("application/vnd.text.list");
    QDataStream stream(&encodedData, QIODevice::ReadOnly);
    QStringList newItems;
    int rows = 0;

    while (!stream.atEnd()) {
        QString text;
        stream >> text;
        newItems << text;
        ++rows;
    }
```

String kemudian dapat dimasukkan ke dalam penampungan data yang mendasarinya. Untuk konsistensi, hal ini dapat dilakukan melalui antarmuka model sendiri:

```
    insertRows(beginRow, rows, QModelIndex());
    foreach (const QString &text, newItems) {
        QModelIndex idx = index(beginRow, 0, QModelIndex());
        setData(idx, text);
        beginRow++;
    }
```

```
    return true;
}
```

Perhatikan bahwa model biasanya akan perlu menyediakan implementasi dari fungsi QAbstractItemModel::insertRows() dan QAbstractItemModel::setData().

Model-model Proxy

Dalam kerangka model / view, item data yang diberikan oleh model tunggal dapat dibagi oleh sejumlah view, dan masing-masing mungkin dapat mewakili informasi yang sama dengan cara yang sama sekali berbeda. View kustom dan delegasi adalah cara yang efektif untuk memberikan representasi yang sangat berbeda dari data yang sama. Namun, aplikasi sering harus memberikan view konvensional ke versi yang diolah dari data yang sama, seperti view yang diurutkan berbeda ke daftar dari item-item.

Meskipun tampaknya tepat untuk melakukan operasi pemilahan dan penyaringan sebagai fungsi internal view, pendekatan ini tidak memungkinkan beberapa pandangan untuk berbagi hasil dari operasi berpotensi mahal tersebut. Pendekatan alternatif, yang melibatkan penyortiran dalam model itu sendiri, menyebabkan masalah yang sama di mana masing-masing harus menampilkan item data yang diatur menurut operasi pengolahan terbaru.

Untuk mengatasi masalah ini, kerangka model / view menggunakan model proxy untuk mengelola informasi yang diberikan antara model individu dan view. Model proxy adalah komponen yang berperilaku seperti model biasa dari perspektif sebuah view, dan mengakses data dari model sumber atas nama view itu. Sinyal dan slot yang digunakan oleh kerangka model / view memastikan bahwa setiap view diperbarui secara tepat tidak peduli berapa banyak model proxy ditempatkan antara dirinya dan model sumber.

Menggunakan model Proxy

Model Proxy dapat disisipkan di antara model yang sudah ada dan sejumlah view. Qt dibekali dengan model proxy yang standar, QSortFilterProxyModel, yang biasanya dipakai dan digunakan secara langsung, tetapi juga dapat diturunkan untuk memberikan perilaku penyaringan dan penyortiran kustom. Kelas QSortFilterProxyModel dapat digunakan dengan cara berikut:

```
    QSortFilterProxyModel *filterModel = new QSortFilterProxyModel(parent);
    filterModel->setSourceModel(stringListModel);

    QListView *filteredView = new QListView;
    filteredView->setModel(filterModel);
```

Karena model proxy mewarisi dari QAbstractItemModel, mereka dapat dihubungkan ke jenis view apapun, dan dapat dibagi antara view. Mereka juga dapat digunakan

untuk memproses informasi yang diperoleh dari model proxy lain dalam susunan pipeline.

Kelas QSortFilterProxyModel dirancang untuk dipakai dan digunakan secara langsung dalam aplikasi. Model Proxy lebih khusus dapat dibuat dengan menurunkan kelas-kelas ini dan melaksanakan operasi perbandingan yang diperlukan.

Menyesuaikan model Proxy

Umumnya, jenis pengolahan yang digunakan dalam model proksi melibatkan pemetaan setiap item data dari lokasi aslinya dalam model sumber ke salah satu lokasi yang berbeda dalam model proxy. Pada beberapa model, beberapa item mungkin tidak memiliki lokasi yang sesuai dalam model proksi; model ini *menyaring* model proxy. Views mengakses item dengan menggunakan indeks-indeks model yang disediakan oleh model proxy, dan ini tidak mengandung informasi tentang model sumber atau lokasi dari item asli dalam model itu.

QSortFilterProxyModel mengaktifkan data dari model sumber untuk disaring sebelum dipasok ke view, dan juga memungkinkan isi dari model sumber yang akan dipasok ke view sebagai data pra-diurutkan.

Penyaringan kustom model-model

Kelas QSortFilterProxyModel menyediakan model penyaringan yang cukup serbaguna, yang mana dapat digunakan dalam berbagai situasi umum. Untuk pengguna tingkat lanjut, QSortFilterProxyModel dapat diturunkan, menyediakan mekanisme yang memungkinkan filter kustom untuk dilaksanakan.

Subclass dari QSortFilterProxyModel dapat reimplement dua fungsi virtual yang dipanggil setiap kali indeks Model dari model proxy diminta atau digunakan:

1. filterAcceptsColumn() digunakan untuk menyaring kolom tertentu dari bagian model sumber.
2. filterAcceptsRow() digunakan untuk menyaring baris tertentu dari bagian model sumber.

Implementasi default fungsi di atas di QSortFilterProxyModel mengembalikan true untuk memastikan bahwa semua item melewati view; reimplementation fungsi ini harus mengembalikan false untuk menyaring baris dan kolom individual.

Penyortiran kustom model-model

Contoh QSortFilterProxyModel menggunakan fungsi built-in qStableSort() Qt untuk mengatur pemetaan antara item dalam model sumber dan yang ada dalam model proxy, yang memungkinkan hirarki yang diurutkan dari item untuk ditampilkan ke view tanpa memodifikasi struktur model sumber. Untuk memberikan perilaku menyortir kustom, reimplement fungsi lessThan() untuk melakukan perbandingan kustom.

Referensi subclassing Model

Turunan kelas-kelas model harus menyediakan implementasi dari banyak fungsi virtual yang didefinisikan di kelas dasar QAbstractItemModel. Jumlah fungsi yang harus dilaksanakan tergantung pada jenis model -apakah itu menyediakan view daftar sederhana, tabel, atau hirarki kompleks dari item-item. Model yang mewarisi dari QAbstractListModel dan QAbstractTableModel dapat memanfaatkan implementasi default dari fungsi yang disediakan oleh kelas-kelas itu. Model yang menampilkan item data dalam struktur seperti pohon harus menyediakan implementasi untuk banyak fungsi virtual dalam QAbstractItemModel.

Fungsi-fungsi yang harus diimplementasikan dalam turunan kelas model dapat dibagi menjadi tiga kelompok:

1. **Penanganan data Item**: Semua model harus mengimplementasikan fungsi untuk mengaktifkan view dan delegasi untuk menanyakan dimensi model, memeriksa item-item, dan mengambil data.
2. **Navigasi dan penciptaan indeks**: model hirarkis harus menyediakan fungsi yang dapat dipanggil oleh view untuk menavigasi struktur seperti pohon yang mereka tampilkan, dan memperoleh indeks-indeks Model untuk item.
3. **Dukungan Drag dan drop dan penanganan tipe MIME**: Model mewarisi fungsi yang mengontrol cara operasi drag and drop internal dan eksternal dilakukan. Fungsi ini memungkinkan item data untuk dijelaskan dalam hal jenis MIME yang dapat mengerti komponen lain dan aplikasi.

Penanganan data item

Model dapat memberikan berbagai tingkat akses ke data yang mereka berikan: Mereka dapat menjadi komponen read-only sederhana, beberapa model dapat mendukung operasi mengubah ukuran, dan lainnya memungkinkan item untuk diedit.

Read-Only akses

Untuk memberikan akses read-only ke data yang disediakan oleh model, fungsi-fungsi berikut harus diimplementasikan dalam kelas turunan model:

flags()	Digunakan oleh komponen lain untuk mendapatkan informasi tentang setiap item yang disediakan oleh model. Dalam banyak model, kombinasi dari isyarat harus mencakup Qt::ItemIsEnabled dan Qt::ItemIsSelectable.
data()	Digunakan untuk memberikan data item ke view dan delegasi. Umumnya, model hanya perlu menyediakan data untuk Qt::DisplayRole dan peran pengguna aplikasi-spesifik apapun, tetapi ini juga praktik yang baik untuk menyediakan data untuk Qt::ToolTipRole, Qt::AccessibleTextRole, dan Qt::AccessibleDescriptionRole. Lihat dokumentasi enum Qt::ItemDataRole untuk informasi tentang jenis yang terkait

	dengan peran masing-masing.
headerData()	Menyediakan untuk view informasi yang ditampilkan di header mereka. Informasi ini hanya diambil oleh view yang dapat menampilkan informasi header.
rowCount()	Menyediakan jumlah baris data yang dipaparkan oleh model.

Keempat fungsi harus diimplementasikan di semua jenis model, termasuk model daftar (kelas turunan QAbstractListModel) dan model tabel (kelas turunan QAbstractTableModel).

Selain itu, fungsi-fungsi berikut ini harus diimplementasikan dalam kelas turunan langsung QAbstractTableModel dan QAbstractItemModel:

columnCount()	Menyediakan jumlah kolom data yang dipaparkan oleh model. Model daftar tidak menyediakan fungsi ini karena sudah diterapkan di QAbstractListModel.

Item yang Editable

Model Editable memungkinkan item data untuk diubah, dan mungkin juga menyediakan fungsi untuk memungkinkan baris dan kolom untuk dimasukkan dan dihapus. Untuk mengaktifkan editing, fungsi-fungsi berikut harus dilaksanakan dengan benar:

flags()	Harus mengembalikan kombinasi yang tepat dari isyarat untuk setiap item. Secara khusus, nilai yang dikembalikan oleh fungsi ini harus menyertakan Qt::ItemIsEditable di samping nilai-nilai yang diterapkan ke item dalam model read-only.
setData()	Digunakan untuk mengubah item data yang terkait dengan indeks model yang ditentukan. Untuk dapat menerima input pengguna, disediakan oleh elemen antarmuka pengguna, fungsi ini harus menangani data yang terkait dengan Qt::EditRole. Implementasi juga dapat menerima data yang terkait dengan berbagai macam peran yang ditentukan oleh Qt::ItemDataRole. Setelah mengubah item data, model harus memancarkan sinyal dataChanged() untuk menginformasikan komponen lain tentang perubahan itu.
setHeaderData()	Digunakan untuk memodifikasi informasi header horizontal dan vertikal. Setelah mengubah item data, model harus memancarkan sinyal headerDataChanged() untuk menginformasikan komponen lain tentang perubahan itu.

Model yang resizable

Semua jenis model yang dapat mendukung penyisipan dan penghapusan baris. Model tabel dan model hirarkis juga dapat mendukung penyisipan dan penghapusan

kolom. Hal ini penting untuk memberitahukan komponen lain tentang perubahan dimensi model baik sebelum maupun setelah mereka terjadi. Akibatnya, fungsi-fungsi berikut dapat diterapkan untuk memungkinkan model untuk diubah ukurannya, tetapi implementasi harus memastikan bahwa fungsi yang tepat dipanggil untuk memberitahu view dan delegasi yang terpasang:

insertRows()	Digunakan untuk menambahkan baris dan item baru dari data untuk semua jenis model. Implementasi harus memanggil `beginInsertRows()` sebelum memasukkan baris baru ke dalam struktur data yang mendasarinya, dan panggil `endInsertRows()` *segera setelahnya*.
removeRows()	Digunakan untuk menghapus baris dan item data yang dikandungnya dari semua jenis model. Implementasi harus memanggil `beginRemoveRows()` *sebelum* baris dihapus dari struktur data yang mendasarinya, dan panggil `endRemoveRows()` *segera setelah itu*.
insertColumns()	Digunakan untuk menambahkan kolom dan item baru dari data untuk model tabel dan model hirarkis. Implementasi harus memanggil `beginInsertColumns()` *sebelum* memasukkan kolom baru ke dalam struktur data yang mendasarinya, dan panggil `endInsertColumns()` *segera setelah itu*.
removeColumns()	Digunakan untuk menghapus kolom dan item data yang dikandungnya dari model tabel dan model hirarkis Implementasi harus memanggil `beginRemoveColumns()` *sebelum* kolom dihapus dari struktur data yang mendasarinya, dan panggil `endRemoveColumns()` *segera setelah itu*.

Secara umum, fungsi ini seharusnya mengembalikan true jika operasi itu berhasil. Namun, mungkin ada kasus di mana operasi hanya sebagian berhasil; misalnya, jika kurang dari jumlah tertentu baris dapat dimasukkan. Dalam kasus tersebut, model harus return false untuk menunjukkan kegagalan untuk mengaktifkan komponen yang melekat agar menangani situasi.

Sinyal yang dipancarkan oleh fungsi yang dipanggil dalam implementasi dari API mengubah ukuran memberikan kesempatan kepada komponen yang terpasang untuk mengambil tindakan sebelum data menjadi tidak tersedia. Enkapsulasi operasi insert dan menghapus dengan fungsi begin dan end juga memungkinkan model untuk mengelola indeks model persisten dengan benar.

Biasanya, fungsi begin dan end mampu menginformasikan komponen lain tentang perubahan struktur yang mendasari model. Untuk perubahan yang lebih kompleks untuk struktur model, mungkin melibatkan reorganisasi internal atau menyortir data, perlu untuk memancarkan sinyal `layoutChanged()` untuk menyebabkan view yang melekat untuk diperbarui.

Pengisian malas model data

Pengisian malas model data secara efektif memungkinkan permintaan informasi tentang model untuk ditangguhkan sampai benar-benar dibutuhkan oleh view.

Beberapa model harus mendapatkan data dari sumber yang jauh, atau harus melakukan operasi yang memakan waktu untuk mendapatkan informasi tentang cara data tersebut akan diatur. Karena view umumnya meminta informasi sebanyak mungkin untuk secara akurat menampilkan model data, dapat berguna untuk membatasi jumlah informasi yang dikembalikan kepada mereka untuk mengurangi permintaan kelanjutan untuk data yang tidak perlu.

Dalam model hirarkis di mana menemukan jumlah anak dari item yang diberikan adalah operasi yang mahal, hal ini berguna untuk memastikan bahwa implementasi `rowCount()` model hanya dipanggil ketika diperlukan. Dalam kasus tersebut, `fungsi` hasChildren() dapat diimplemen ulang untuk menyediakan cara murah untuk view agar memeriksa keberadaan anak-anak dan dalam kasus QTreeView, menggambar dekorasi yang sesuai untuk item induknya.

Apakah implementasi ulang dari `hasChildren()` mengembalikan benar atau salah, itu mungkin tidak diperlukan untuk view memanggil `rowCount()` untuk mengetahui berapa banyak anak-anak yang ada. Misalnya, QTreeView tidak perlu tahu berapa banyak anak-anak yang ada jika item induk belum diperluas untuk menunjukkan kepada mereka.

Jika diketahui bahwa banyak item akan memiliki anak-anak, implementasi ulang `hasChildren()` untuk mengembalikan *true* secara mutlak kadang-kadang merupakan pendekatan yang berguna untuk diambil. Hal ini memastikan bahwa setiap item dapat kemudian diperiksa untuk anak-anak sementara membuat populasi awal dari model data secepat mungkin. Satu-satunya kelemahan adalah bahwa item tanpa anak-anak dapat ditampilkan secara salah dalam beberapa view sampai pengguna mencoba untuk melihat item anak yang tidak ada.

Penciptaan Navigasi dan indeks Model

Model hirarkis harus menyediakan fungsi yang dapat dipanggil view untuk menavigasi struktur seperti pohon yang mereka ekspos, dan memperoleh Model indeks untuk item.

Induk dan anak-anak

Karena struktur yang diekspos ke view ditentukan oleh struktur data yang mendasarinya, terserah kepada masing-masing kelas turunan model untuk membuat indeks modelnya sendiri dengan menyediakan implementasi dari fungsi berikut:

index()	Memberi indeks model untuk item induk, fungsi ini memungkinkan view dan delegasi untuk mengakses anak-anak dari item tersebut. Jika tidak ada item anak yang valid dapat ditemukan --sesuai dengan baris tertentu, kolom, dan indeks Model induk-- fungsi harus mengembalikan QModelIndex(), yang merupakan indeks model yang valid.
parent()	Menyediakan indeks model yang sesuai dengan induk dari item anak yang diberikan. Jika indeks model yang ditentukan sesuai dengan item top-level dalam model, atau jika tidak ada item induk yang valid dalam model, fungsi harus mengembalikan indeks model yang valid, dibuat dengan konstruktor QModelIndex() kosong.

Kedua fungsi di atas menggunakan fungsi pabrik createIndex() untuk menghasilkan indeks untuk komponen lain untuk digunakan. Hal yang biasa bagi model untuk memasok beberapa identifier unik untuk fungsi ini untuk memastikan bahwa indeks model dapat kembali dihubungkan dengan item yang sesuai di lain waktu.

Dukungan Drag dan drop dan penanganan tipe MIME

kelas Model / view mendukung operasi drag and drop, memberikan perilaku default yang cukup untuk banyak aplikasi. Namun, juga memungkinkan untuk menyesuaikan cara item dikodekan selama operasi drag dan drop, apakah mereka akan disalin atau dipindahkan secara default, dan bagaimana mereka dimasukkan ke dalam model yang ada.

Selain itu, kelas view yang mudah menerapkan perilaku khusus yang harus mengikuti apa yang diharapkan oleh pengembang yang ada. Bagian *Views yang Mudah* memberikan gambaran tentang perilaku ini.

Data MIME

Secara default, built-in model dan view menggunakan tipe MIME intern (application/x-qabstractitemmodeldatalist) untuk mengedarkan informasi tentang indeks-indeks model. Ini menentukan data untuk daftar item, yang berisi nomor baris dan kolom dari setiap item, dan informasi tentang peran yang mendukung setiap item.

Data yang dikodekan menggunakan jenis MIME ini dapat diperoleh dengan memanggil QAbstractItemModel::mimeData() dengan QModelIndexList berisi item yang akan diserialisasi.

Ketika menerapkan dukungan drag and drop dalam model kustom, adalah mungkin untuk mengekspor item-item dari data dalam format khusus dengan implementasi ulang fungsi berikut:

mimeData()	Fungsi ini dapat di-reimplement untuk mengembalikan data dalam format selain tipe MIME internal default application/x-qabstractitemmodeldatalist. Sub-kelas dapat memperoleh objek QMimeData default dari kelas dasar dan menambahkan data ke dalam format tambahan.

Bagi banyak model, hal ini berguna untuk memberikan isi item dalam format umum yang diwakili oleh jenis MIME seperti text/plain dan image/png. Perhatikan bahwa gambar, warna dan dokumen HTML dengan mudah dapat ditambahkan ke objek QMimeData dengan fungsi QMimeData::setImageData(), QMimeData::setColorData(), dan QMimeData::setHtml().

Menerima data yang didrop

Ketika operasi drag and drop dilakukan atas view model yang mendasari ditanya untuk menentukan jenis operasi yang didukung dan jenis MIME yang dapat diterima. Informasi ini disediakan oleh fungsi QAbstractItemModel::supportedDropActions() dan QAbstractItemModel::mimetypes(). Model yang tidak menimpa implementasi yang disediakan oleh QAbstractItemModel, mendukung operasi copy dan jenis MIME internal default untuk item.

Ketika data item yang diserialisasi dijatuhkan ke view, data dimasukkan ke dalam model saat ini menggunakan implementasi dari QAbstractItemModel::dropMimeData(). Implementasi standar dari fungsi ini tidak akan menghapus semua data dalam model; sebaliknya, ia mencoba untuk memasukkan item data baik sebagai saudara kandung dari item, ataupun sebagai anak-anak dari item tersebut.

Untuk memenafaatkan implementasi standar QAbstractItemModel untuk built-in tipe MIME, model baru harus memberikan reimplementasi dari fungsi berikut:

insertRows() insertColumns()	Fungsi ini memungkinkan model untuk secara otomatis memasukkan data baru menggunakan implementasi yang disediakan oleh QAbstractItemModel::dropMimeData().
setData()	Memungkinkan baris dan kolom baru untuk diisi dengan item-item.
setItemData()	Fungsi ini memberikan dukungan yang lebih efisien untuk mengisi item baru.

Untuk menerima bentuk lain dari data, fungsi-fungsi ini harus diimplementasi ulang:

supportedDropActions()	Digunakan untuk mengembalikan kombinasi tindakan drop, yang menunjukkan jenis operasi drag and drop yang diterima model.
mimetypes()	Digunakan untuk mengembalikan daftar jenis MIME yang dapat diterjemahkan dan ditangani oleh model. Umumnya, jenis MIME yang didukung untuk input ke dalam model adalah sama dengan yang dapat digunakan ketika pengkodean data untuk digunakan oleh komponen eksternal.
dropMimeData()	Melakukan decoding sebenarnya dari data yang ditransfer dengan operasi drag dan drop, menentukan di mana dalam model itu akan ditetapkan, dan menyisipkan

	baris dan kolom di mana diperlukan. Bagaimana fungsi ini diimplementasikan dalam subclass tergantung pada kebutuhan data yang terpapar oleh masing-masing model.

Jika implementasi fungsi dropMimeData() mengubah dimensi model dengan memasukkan atau menghapus baris atau kolom, atau jika item data diubah, harus berhati-hati untuk memastikan bahwa semua sinyal yang relevan dipancarkan. Hal ini dapat berguna untuk hanya memanggil reimplementasi fungsi lainnya dalam subclass, seperti setData(), insertRows(), dan insertColumns(), untuk memastikan bahwa model tersebut berperilaku secara konsisten.

Dalam rangka untuk memastikan operasi drag bekerja dengan baik, penting untuk reimplement fungsi-fungsi berikut yang menghapus data dari model:

- removeRows()
- removeRow()
- removeColumns()
- removeColumn()

Untuk informasi lebih lanjut tentang drag dan drop dengan view item, lihat *Menggunakan drag dan drop dengan view item*.

Views yang mudah

View yang mudah (QListWidget, QTableWidget, dan QTreeWidget) menimpa fungsionalitas standar drag dan drop untuk menyediakan perilaku yang kurang fleksibel, tapi lebih alami yang sesuai untuk banyak aplikasi. Sebagai contoh, karena lebih umum untuk men-drop data ke dalam sel di QTableWidget, mengganti isi yang ada dengan data yang ditransfer, model yang mendasari akan mengatur data item target daripada menyisipkan baris baru dan kolom ke dalam model. Untuk informasi lebih lanjut tentang drag dan drop dalam view yang mudah, Anda dapat melihat *Menggunakan drag and drop dengan view item*.

Optimasi kinerja untuk data dalam jumlah besar

Fungsi canFetchMore() mengecek jika induk memiliki lebih banyak data tersedia dan mengembalikan true atau false secara bersesuaian. Fungsi fetchMore() mengambil data berdasarkan induk yang ditentukan. Kedua fungsi tersebut dapat dikombinasikan, misalnya, dalam sebuah query database yang melibatkan data tambahan untuk mengisi QAbstractItemModel. Kita reimplement canFetchMore() untuk menunjukkan jika ada lebih banyak data yang akan diambil dan fetchMore() untuk mengisi model seperti yang diperlukan.

Contoh lain misalnya mengisi model pohon secara dinamis, di mana kita reimplement fetchMore() ketika cabang di model pohon diperluas.

Jika implementasi ulang Anda atas fetchMore() menambahkan baris untuk model, Anda harus memanggil beginInsertRows() dan endInsertRows(). Juga, baik

`canFetchMore()` maupun `fetchMore()` harus direimplementasikan sebagaimana implementasi standar mereka mengembalikan *false* dan tidak melakukan apa-apa.

© 2014 Digia Plc and/or its subsidiaries. Documentation contributions included herein are the copyrights of their respective owners.

The documentation provided herein is licensed under the terms of the GNU Free Documentation License version 1.3 as published by the Free Software Foundation.

Digia, Qt and their respective logos are trademarks of Digia Plc in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.