# Exercises Lecture 9 – Custom Models

**Aim:** This exercise will take you through the process of creating your own model. You will explore data roles and item editing from the model's viewpoint.

**Duration:** 1h

## Implementing a list model

This exercise comes with a source code package. Extract the package in your working directory. The package contains a set of starting point projects. For this first step, use the *colornames* project as the starting point.

The projects provides a GUI that shows a list of color names from a model. In the implementation, the model class, `ListModel`, is incomplete. The goal for this step is to finish it.

The first stage of designing a model is to decide where the data is kept. In this case, the data is a list of color names, so a `QStringList` will is a good candidate. Add a private `m_colorNames` string list member to the `ListModel` class. It will be used to hold the source data.

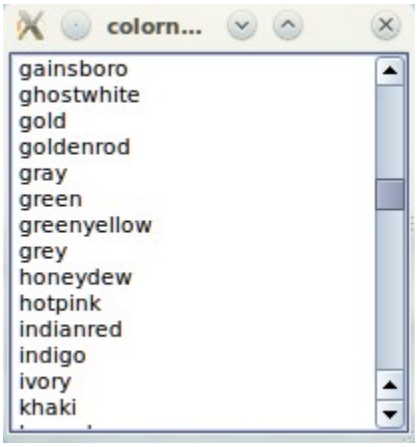
In the constructor of the `ListModel` class, you need to populate the model with data. Qt contains a special static call returning a list of colors it is able to name – `QColor::colorNames()`. In the constructor, fill the model's data storage with the return value of `QColor::colorNames()`.

Viewing the model still does not show any data. This is due to two reasons. First, the model does not return an size. Second, it does not return any data.

Start by implementing the `rowCount` method so that it returns the number of items sorted in the `m_colorNames` string list. This gives the models a size.

The data of the model is returned from the `data` method. It is easy to implement, but not straight forward. It is important to return the data for the correct role. In this case, return the color name for the requested row if the requested index is valid and the role is `DisplayRole`. In all other cases, return an uninitialized `QVariant`.

Running the application now should present you with a pretty list of color names.

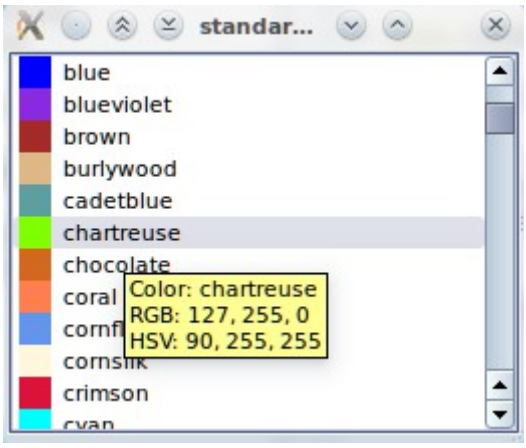



## Handling other roles

A nice feature to any color list is to show the actual color. You can either start from the *standardroles* project or continue your last project.

Although you are working with a single column model, that does not mean you are limited to one piece of information per role. Using different roles, you can pass multiple sets of data to the view.

In this step, you will extend the `ListModel` model to display a small icon of the color in question. The color is returned as a `QColor` instance when the `DecorationRole` is requested. In addition to the color, a tool tip with the RGB and HSV representations of the color will be returned when the `ToolTipRole` is requested.




Start in the data method and retrieve the `QColor` for the row in question. To convert the name of a color to the color itself you can use the constructor for the `QColor` class that takes a string containing the name as its argument.

```
    const QString &name = m_colorNames.at(row);
    QColor color = QColor(name);
```

Returning this color for the `DecorationRole` lets the view add the small color samples that we are looking for.

For each `QColor` it is possible to request the RGB values (through `red()`, `green()` and `blue()`) as well as the HSV values (using `hue()`, `saturation()` and `value()`). For the `ToolTipRole`, take these values and compose them into a string that you return. Try to format the result as the tool tip shown in the figure above.

## *Making the model editable*

So far the model only returned item values but did not allow the user to change them. The next step is to allow the user to change existing entries. Start from the *editablecolors* project. This project is basically the outcome of the previous step, but with a few additions.

To make a model editable, you need to take care of two things. First you need to program the model to inform that a particular item can be modified. This is done by reimplementing the `flags` method. Then you need to provide an interface to actually alter the data of the model. This is done by reimplementing the `setData` method.

The `QAbstractItemModel` provides a sane default implementation of `flags` method but it does not allow modification of items by default. When reimplementing this method it is important to remember to use the binary or operator (bar, "`|`") to add the values together. Using the plus operator ("`+`") can lead to unexpected behaviour.

In your reimplementation of the method, simply add the base class' return value to the ItemIsEnabled flag.

```
    return QAbstractListModel::flags(index) | Qt::ItemIsEnabled;
```

To complete the functionality, you need to add an implementation of the `setData()` method. If you look at its signature, you will notice that the default role value is `Qt::EditRole`. This is a special role that should be used when reading or writing editable data.

The implementation is simple – when you receive the `Qt::EditRole`, use the value given. If any other role is passed as the argument – immediately exit the method with `false` return value. Remember to return `true` when accepting a value.

Before returning true you have to emit the `dataChanged()` signal to notify all views that a range of indexes changed some of their data. Use the `value.toString()` method to get a string to update your `QStringList` with.

For a consistent behavior you should also modify your `data()` implementation to handle `EditRole` in exactly the same manner as `DisplayRole`, i.e to return exactly the same value.

# Solution Tips

### *Step 1*

Use a simple condition check to see if you should return a real value from `data()`:

```
if(index.isValid() && role == Qt::DisplayRole) {
    return m_items.at(index.row());
}
return QVariant();
```

### *Step 2*

For the tooltip use `QString`'s percent notation to specify the format and inject the integers using `QString::arg()`. You can use rich text for your tooltip for better effect:

```
QString toolTip = QString("Color: %1<br/>"
                          "RGB: %2, %3, %4<br/>"
                          "HSV: %5, %6, %7");
return toolTip.arg(name)
              .arg(color.red())
              .arg(color.green())
              .arg(color.blue())
              .arg(color.hue())
              .arg(color.saturation())
              .arg(color.value());
```

### *Step 3*

When emitting `dataChanged()` in this case the range you pass to it consists of only one index:

```
emit dataChanged(index, index);
```