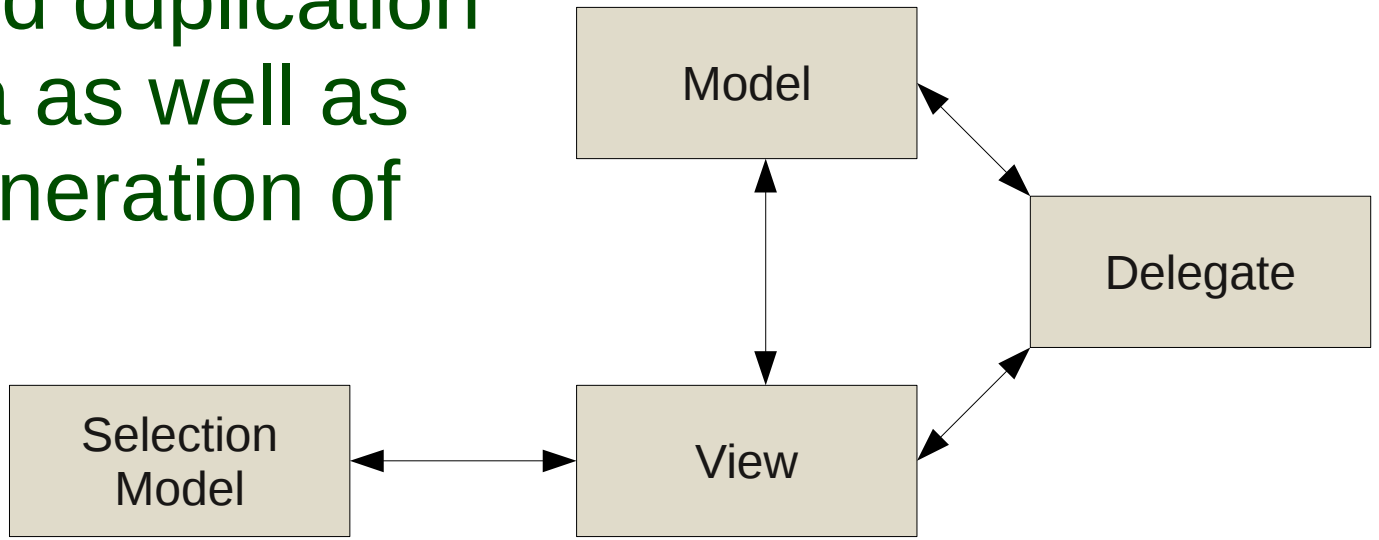
Qt in Education

Custom Models

NOKIA

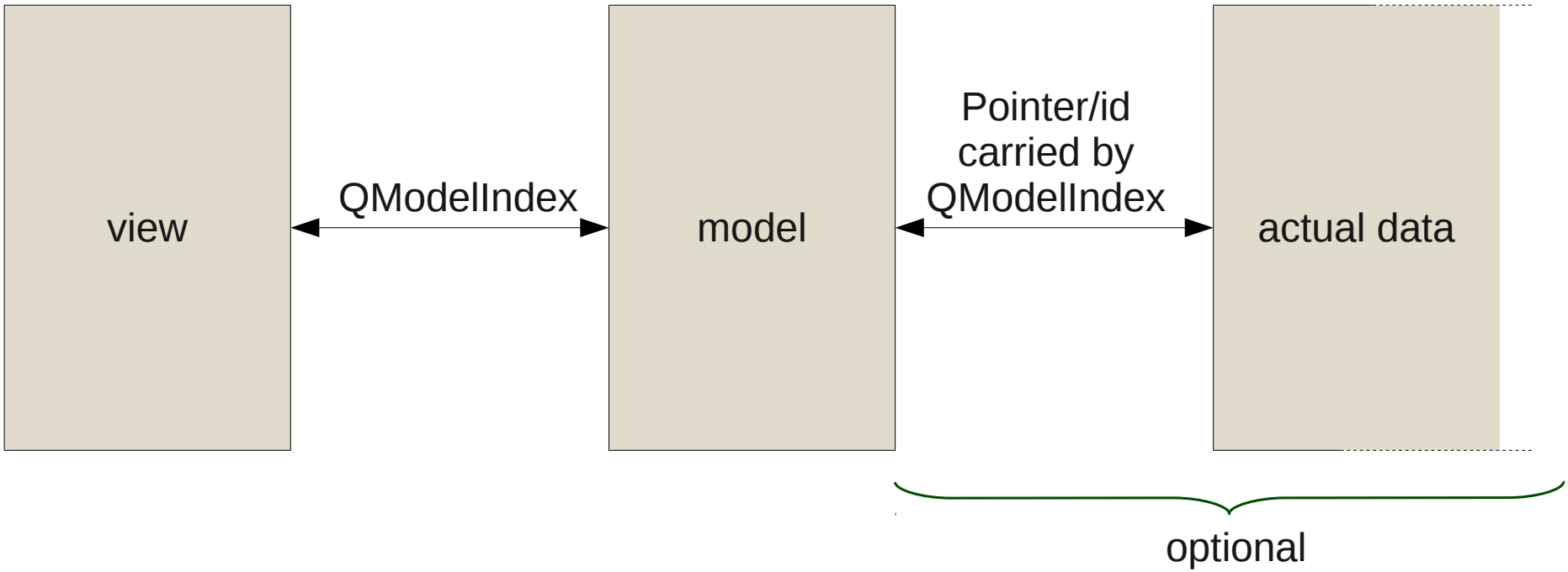
Model View

- The model view classes focus on separating data from visualization from modification
- The separation of data makes it possible to avoid duplication of data as well as live generation of data



Model Indexes

- The key to communication between the model and view is the QModelIndex class



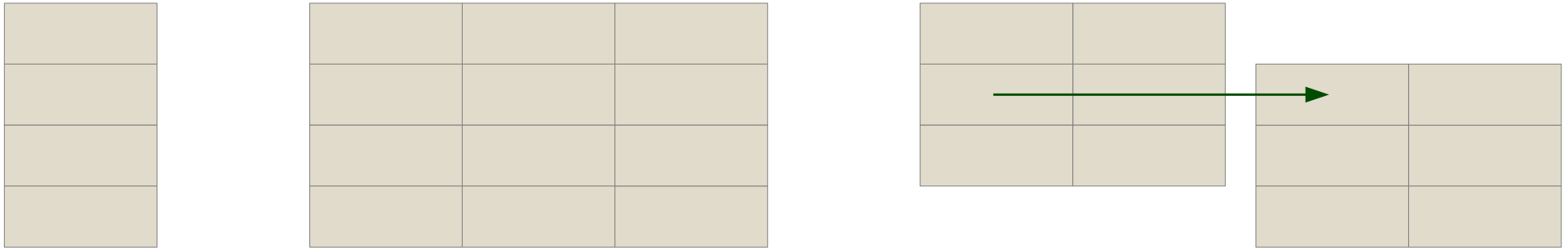
Model Indexes

- The QAbstractItemModel interface exposes model indexes to the view

```
QModelIndex index(int row, int column, const QModelIndex &parent) const
QModelIndex parent(const QModelIndex &index) const

bool hasChildren(const QModelIndex &parent) const
int columnCount(const QModelIndex &parent) const
int rowCount(const QModelIndex &parent) const

QVariant data(const QModelIndex &index, int role) const
bool setData(const QModelIndex &index, const QVariant &value, int role)
```

Creating Models

- All models are derived from the QAbstractItemModel class
- For non-tree models it is better to start from the simplified interfaces
  - QAbstractListModel – for single-column lists
  - QAbstractTableModel – for tables



Implementing a List Model

- A list model is best implemented from the QAbstractListModel, re-implementing:

mandatory

- rowCount – the number of rows in the model
- data – for providing data to the views

optional but recommended

- headerData – for providing a header

for editable models

- flags – for indicating whether items are editable, selectable, etc
- setData – for editable models

A List Model

- A non-editable, minimal list model
  - Headers are uncommon in single-column lists

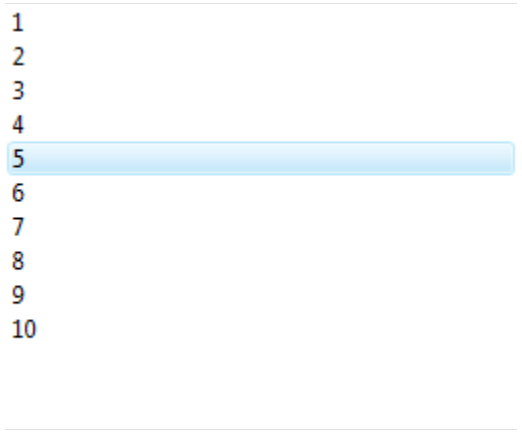
```
class ListModel : public QAbstractListModel
{
    Q_OBJECT
public:
    explicit ListModel(QObject *parent = 0);

    int rowCount(const QModelIndex &parent=QModelIndex()) const;
    QVariant data(const QModelIndex &index,
                  int role=Qt::DisplayRole) const;
};
```

A List Model

```
int ListModel::rowCount(const QModelIndex &parent) const
{
    return 10;
}

QVariant ListModel::data(const QModelIndex &index, int role) const
{
    if(!index.isValid() || role!=Qt::DisplayRole)
        return QVariant();

    return QVariant(index.row()+1);
}
```

Using the List Model

```
QListView *listView = new QListView(this);
listView->setModel(new ListModel(this));
```



Implementing a Table Model

- Table models are derived from QAbstractTableModel – which is a super-set of QAbstractListModel
- Re-implement the following methods
  - columnCount – the number of columns
  - rowCount, data, headerData, flags, setData

A Table Model

- An editable table model

```
class TableModel : public QAbstractTableModel
{
    Q_OBJECT
public:
    explicit TableModel(QObject *parent = 0);

    int rowCount(const QModelIndex &parent=QModelIndex()) const;
    int columnCount(const QModelIndex &parent=QModelIndex()) const;

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

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

private:
    int m_data[10][10];
};
```

A Table Model

```
TableModel::TableModel(QObject *parent) :
    QAbstractTableModel(parent)
{
    for(int i=0; i<10; ++i)
        for(int j=0; j<10; ++j)
            m_data[i][j] = (i+1)*(j+1);
}

int TableModel::rowCount(const QModelIndex &parent) const
{
    return 10;
}

int TableModel::columnCount(const QModelIndex &parent)
const
{
    return 10;
}
```

A Table Model

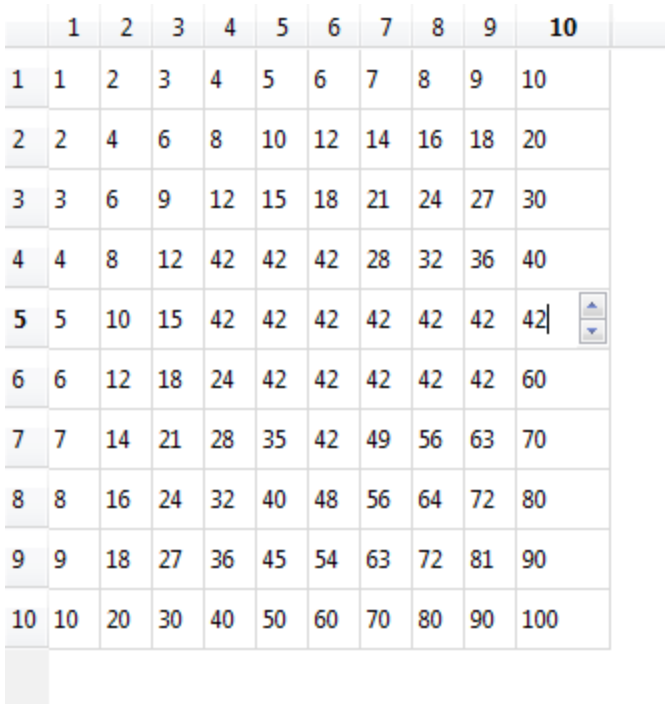
```
QVariant TableModel::data(const QModelIndex &index, int role) const
{
    if(!index.isValid())
        return QVariant();

    if(role == Qt::DisplayRole || role == Qt::EditRole)
        return QVariant(m_data[index.row()][index.column()]);

    return QVariant();
}

QVariant TableModel::headerData(int section,
    Qt::Orientation orientation, int role) const
{
    if(role != Qt::DisplayRole)
        return QVariant();

    return QVariant(section+1);
}
```

A Table Model

```
Qt::ItemFlags TableModel::flags(const QModelIndex &index) const
{
    return (Qt::ItemIsSelectable |
            Qt::ItemIsEditable   |
            Qt::ItemIsEnabled);
}

bool TableModel::setData(const QModelIndex &index,
    const QVariant &value, int role)
{
    if(!index.isValid())
        return false;

    if(role != Qt::EditRole)
        return false;

    if(!value.canConvert<int>())
        return false;

    m_data[index.row()][index.column()] = value.toInt();
    emit dataChanged(index, index);
    return true;
}
```

Using the Table Model

```
QTableView *tableView = new QTableView(this);
tableView->setModel(new TableModel(this));
```

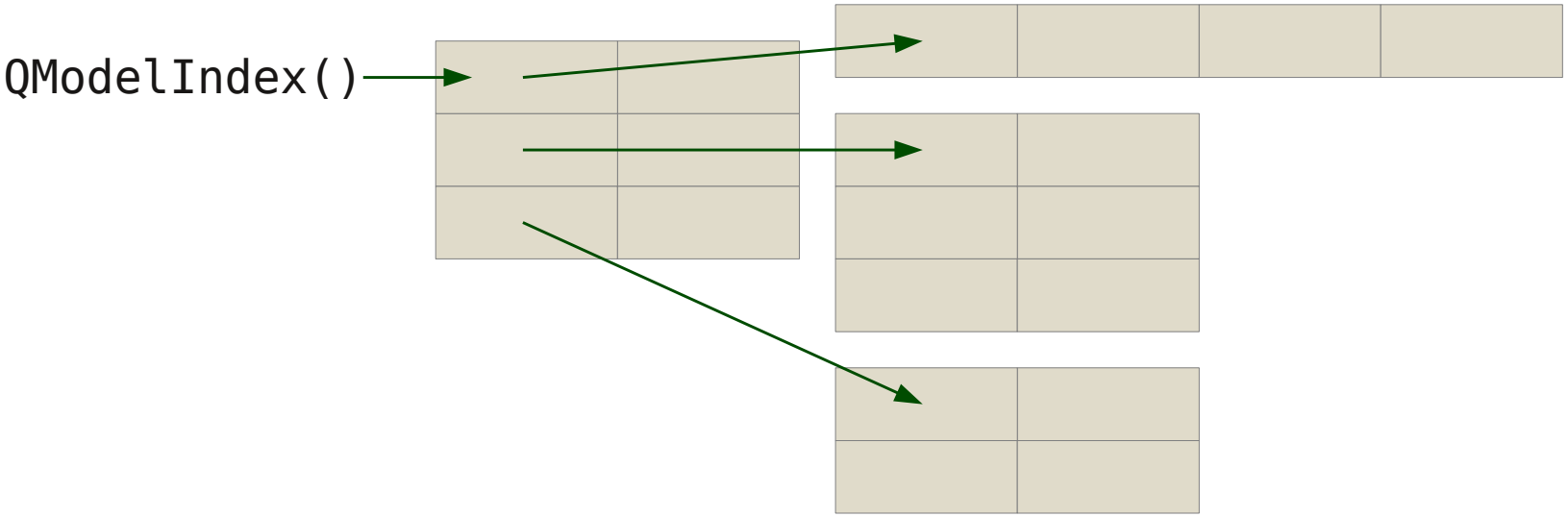


	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	42	42	42	28	32	36	40
5	5	10	15	42	42	42	42	42	42	42
6	6	12	18	24	42	42	42	42	42	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Implementing a Tree Model

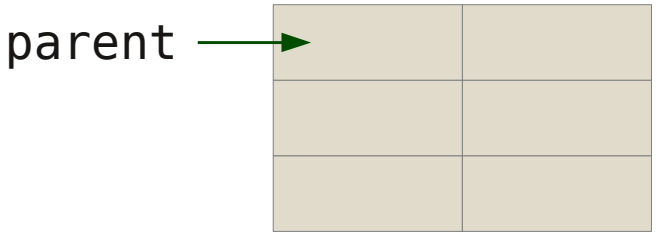
- Tree models need to implement the full QAbstractItemModel interface
  - All methods from lists and tables
  - parent – returns the model index of the parent of the item
  - index – returns a model index for a given item of a given parent

The Structure of Trees



- Tables containing other tables
- Only the first column is allowed to be the parent of a sub-table
- The root index has no parent, i.e. an invalid QModelIndex

The index Method




```
QModelIndex index(int row, int column, const QModelIndex &parent) const
```

- Given a parent (could be invalid, i.e. root) returns a QModelIndex for the requested row and column
- The model uses createIndex to create a valid QModelIndex instance, or QModelIndex() for invalid

```
QModelIndex createIndex(int row, int column, void *ptr) const
QModelIndex createIndex(int row, int column, quint32 id) const
```

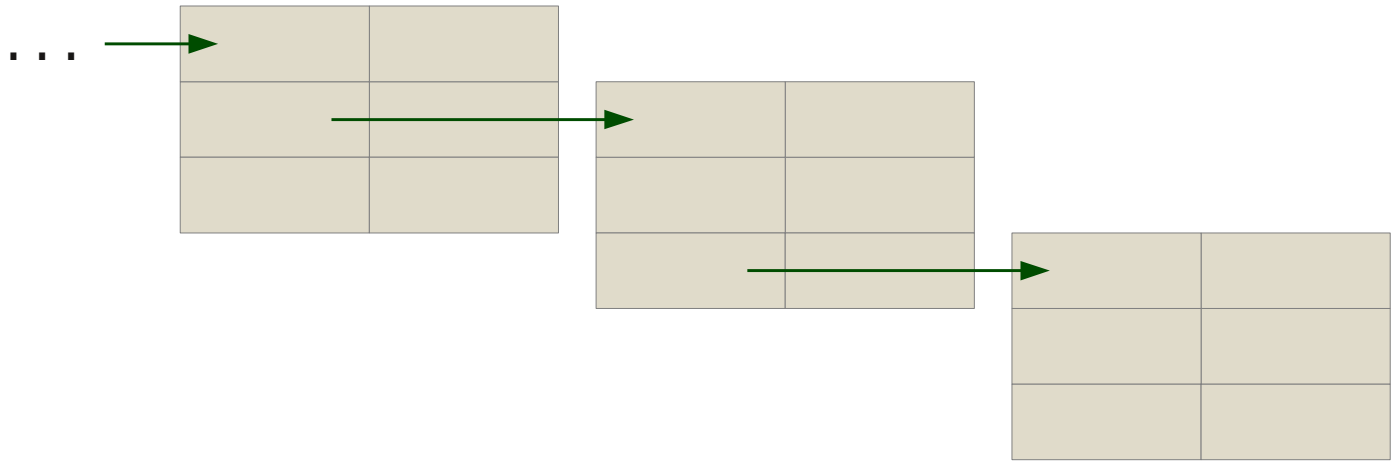
createIndex

- The create index method provides storage for an integer or a pointer
  - This is used as the link from the model to the underlaying data

```
QModelIndex createIndex(int row, int column, void *ptr) const
QModelIndex createIndex(int row, int column, quint32 id) const
```

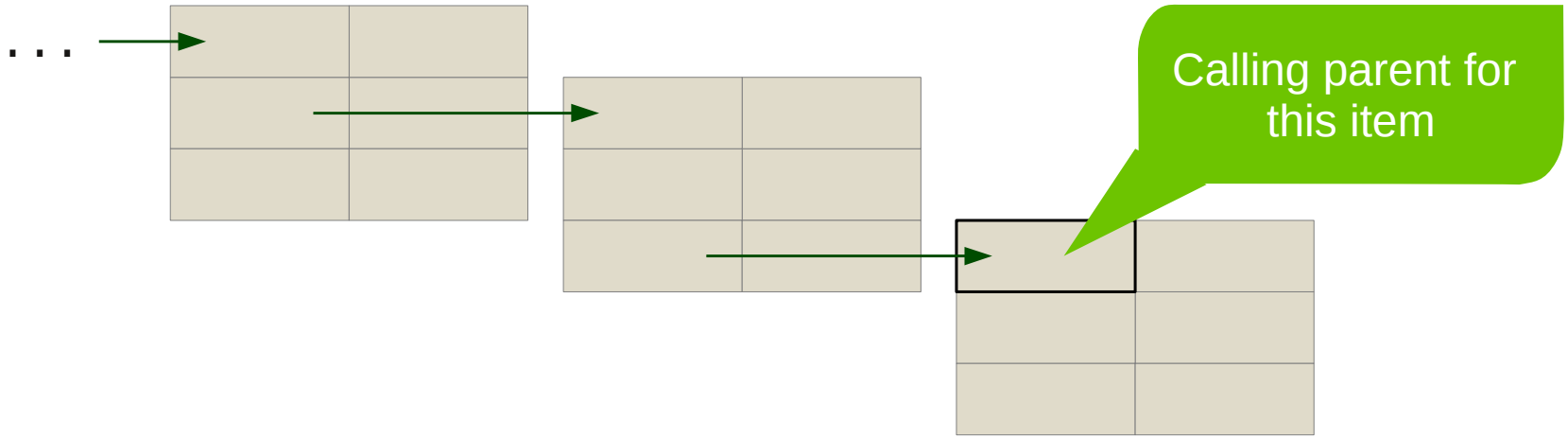
The parent Method

- The parent method returns a QModelIndex for the item's parent
  - Requires the row of the parent
    - Might require the parent's parent



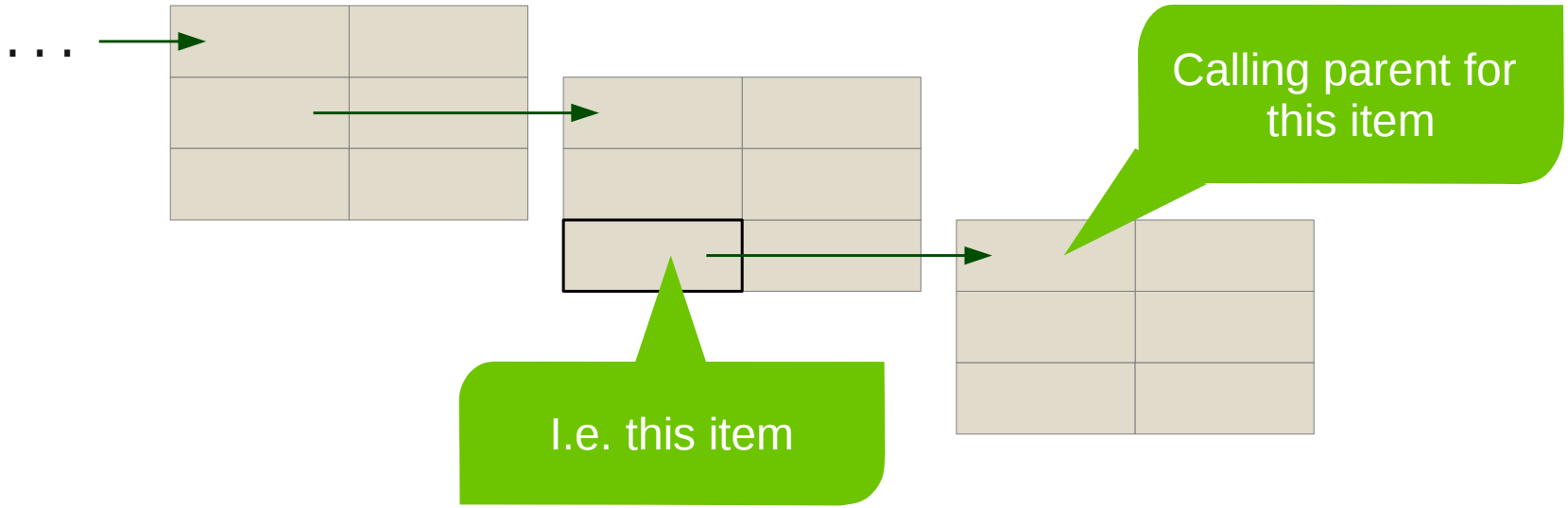
The parent Method

- The parent method returns a QModelIndex for the parent
  - Requires the row of the parent
    - Might require the parent's parent



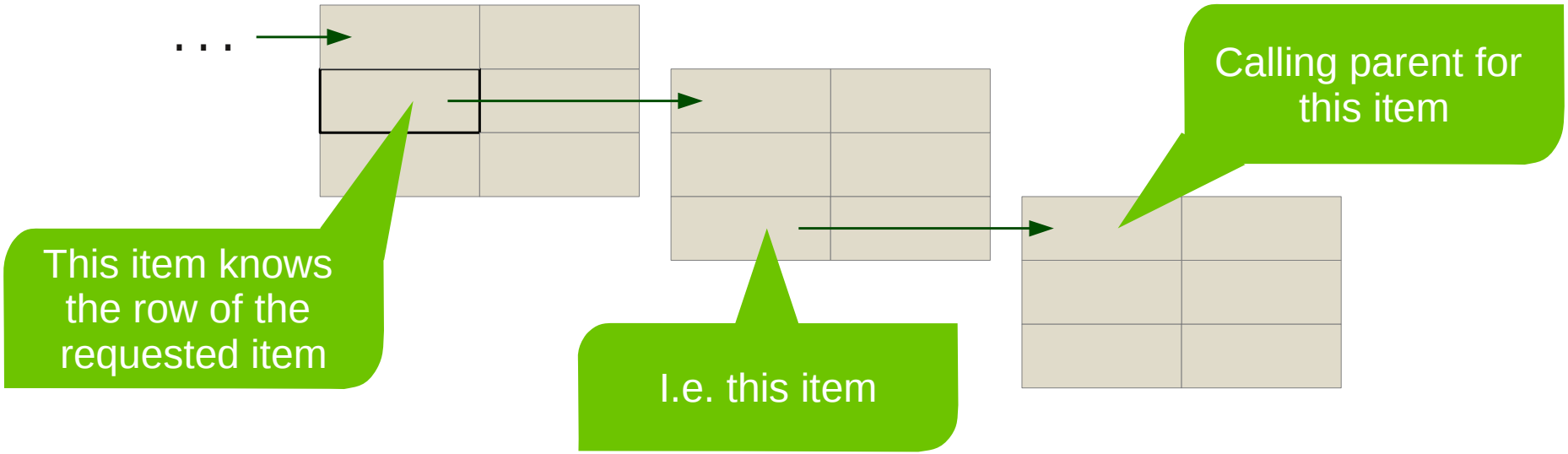
The parent Method

- The parent method returns a QModelIndex for the parent
  - Requires the row of the parent
    - Might require the parent's parent



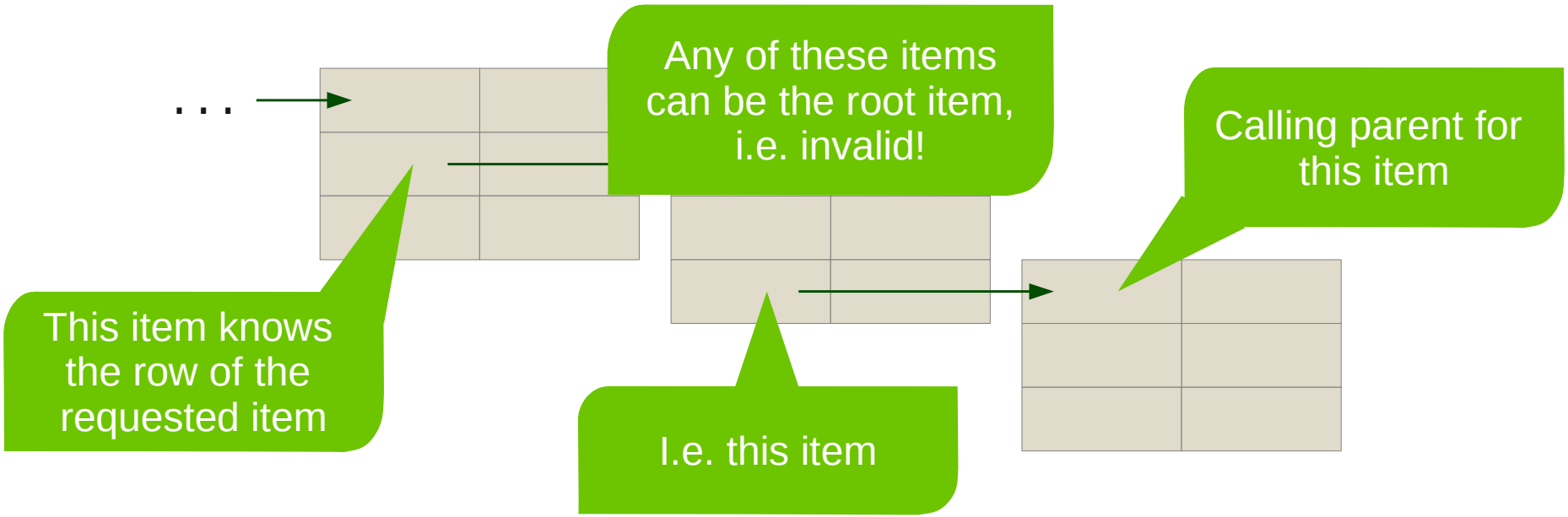
The parent Method

- The parent method returns a QModelIndex for the parent
  - Requires the row of the parent
    - Might require the parent's parent



The parent Method

- The parent method returns a QModelIndex for the parent
  - Requires the row of the parent
    - Might require the parent's parent



The parent Method

- The parent method can be easy to implement as long as the underlaying data contains the needed information
  - Each item needs to know its parent
  - Each item needs to know its row
- The alternative is to traverse the data to find the information needed

A Tree Model

- A basic tree model exposing a hierarchy of QObject instances
  - Does not take changing QObject hierarchies into account!

```
class TreeModel : public QAbstractItemModel
{
    Q_OBJECT
public:
    explicit TreeModel(QObject *rootObject, QObject *parent = 0);


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

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

    int rowCount(const QModelIndex &parent=QModelIndex()) const;
    int columnCount(const QModelIndex &parent=QModelIndex()) const;


private:
    QObject *m_rootObject;
};
```

A Tree Model

```
QVariant TreeModel::data(const QModelIndex &index, int role) const
{
    if(!index.isValid() || role != Qt::DisplayRole)
        return QVariant();


    QObject *object = static_cast<QObject*>(index.internalPointer());
    if(index.column() == 0)
        return object->objectName();
    else
        return object->metaObject()->className();
}


QVariant TreeModel::headerData(int section, Qt::Orientation orientation, int role) const
{
    if(role != Qt::DisplayRole || orientation != Qt::Horizontal)
        return QVariant();


    switch(section)
    {
    case 0:
        return QVariant("Object");
    case 1:
        return QVariant("Class");
    default:
        return QVariant();
    }
}
```

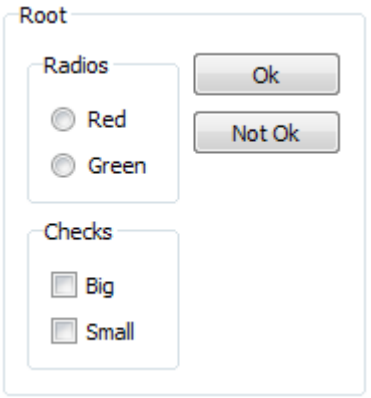
A Tree Model

```
QModelIndex TreeModel::index(int row, int column,
    const QModelIndex &parent) const
{
    if(column>1)
        return QModelIndex();

    if(!parent.isValid())
    {
        if(row == 0)
            return createIndex(0, column, (void*)m_rootObject);
        else
            return QModelIndex();
    }

    QObject *parentObject = static_cast<QObject*>(parent.internalPointer());
    if(row < parentObject->children().count())
        return createIndex(row, column,
            (void*)parentObject->children().at(row));
    else
        return QModelIndex();
}
```

A Tree Model

```
QModelIndex TreeModel::parent(const QModelIndex &index) const
{
    if(!index.isValid())
        return QModelIndex();

    QObject *object = static_cast<QObject*>(index.internalPointer());
    if(object == m_rootObject)
        return QModelIndex();

    QObject *parentObject = object->parent();
    if(parentObject == m_rootObject)
        return createIndex(0, 0, (void*)m_rootObject);

    QObject *parentParentObject = parentObject->parent();
    if(!parentParentObject)
        return QModelIndex();
    else
        return createIndex(
            parentParentObject->children().indexOf(parentObject),
            0, (void*)parentObject);
}
```

Using the Tree Model




```
QGroupBox *rootObject = setupGroup();
QTreeView *treeView = new QTreeView(this);
treeView->setModel(new TreeModel(rootObject, this));
```

Object	Class
rootFrame	QGroupBox
horizontalLayout	QHBoxLayout
verticalLayout_4	QVBoxLayout
verticalLayout	QVBoxLayout
radioFrame	QGroupBox
verticalLayout_2	QVBoxLayout
redRadio	QRadioButton
greenRadio	QRadioButton
checkFrame	QGroupBox
verticalLayout_3	QVBoxLayout
bigCheck	QCheckBox
smallCheck	QCheckBox
okButton	QPushButton
notOkButton	QPushButton

Break

Verifying Models

- Parts of the model interface is by convention, i.e. not checked at compile time
  - Can break at run-time
  - Needs testing!
- The ModelTest class from labs.qt.nokia.com monitors models and reports unwanted behavior

  http://labs.qt.nokia.com/page/Projects/Itemview/Modeltest

Modifying Models

- A model can change for a number of reasons
  - A delegate calls setData
  - The underlaying data is changed
- Changes can affect indexes, data or the entire model

Modifying Data

- When data is modified, even through setData, the dataChanged signal must be emitted

```
bool TableModel::setData(const QModelIndex &index,
    const QVariant &value, int role)
{
    if(!index.isValid()         ||
       role != Qt::EditRole     ||
       !value.canConvert<int>() )
        return false;

    m_data[index.row()][index.column()] = value.toInt();
    emit dataChanged(index, index);
    return true;
}
```

Modifying Structure

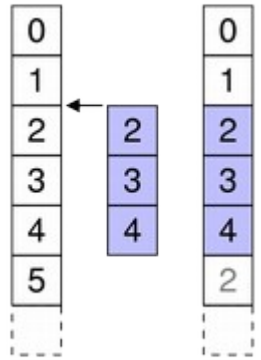
- Inserting and removing rows and columns must be communicated from the model
- The signals emitted for columns is analogous to the signals for rows

- When adding rows
  - Call beginInsertRows
  - insert rows
  - Call endInsertRows
- When removing rows
  - Call beginRemoveRows
  - remove rows
  - Call endRemoveRows

Inserting Rows

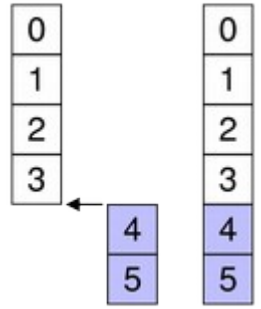
- Inserting

```
{
    beginInsertRows(parent, startRow,
        newRows.count()+startRow-1);
    m_data.insert(startRow, newRows);
    endInsertRows();
}
```



- Inserting

```
{
    beginInsertRows(parent, m_data.count(),
        newRows.count()+m_data.count()-1);
    m_data.append(newRows);
    endInsertRows();
}
```



Changing Underlaying Data

- If a model serves as an interface for underlaying data, changes in that data affect the model
  - The data must inform the model prior to inserting or removing rows or columns
  - The entire model must be reset
- Reset the entire model by calling reset
  - This invalidates selections, indexes, etc

Verifying Models

- The ModelTest class monitors models for common errors and mistakes
  - Can be downloaded from qt.gitorious.com/qt, from the tests/auto/modeltest subdirectory
  - Adding it to your code is trivial

```
QAbstractItemModel *model = new CustomModel(this);
ModelTest *tester = new ModelText(model);
```

  - Using the model, triggering changes in the model will cause ModelTest to assert the interface of the model

Ensuring Performance

- If traversing underlaying data is slow, there are API functions to re-implement
- If rowCount is slow, re-implementing hasChildren reduces the number of calls to rowCount
- If fetching data slow, re-implementing canFetchMore and fetchMore lets the view request information as it is needed

Implementing a Slow Model

- Implementing the API for slow models, but for a quick model

```
class LazyModel : public QAbstractItemModel
{
    Q_OBJECT
public:
...
    bool canFetchMore(const QModelIndex &parent) const;
    void fetchMore(const QModelIndex &parent);
    bool hasChildren(const QModelIndex &parent=QModelIndex()) const;
...
```

A Slow Model

```
bool LazyModel::canFetchMore(const QModelIndex &parent) const
{
    if(parent.isValid() && parent.internalId() == -1 && m_returnedRows < 10)
        return true;
    else
        return false;
}


void LazyModel::fetchMore(const QModelIndex &parent)
{
    if(parent.isValid() && parent.internalId() == -1 && m_returnedRows < 10)
    {
        beginInsertRows(parent, rowCount(), rowCount());
        m_returnedRows ++;
        endInsertRows();
    }
}


bool LazyModel::hasChildren(const QModelIndex &parent) const
{
    if(!parent.isValid() || parent.internalId() == -1)
        return true;
    else
        return false;
}
```

Using a Slow Model

```
QTreeView *treeView = new QTreeView(this);
treeView->setModel(new LazyModel(this));
```