Qt Quick on the Desktop

Presented by David Johnson

Overview

- Qt Quick Layouts
- Application Window
- Desktop Controls
- Views and Navigation
- Custom Styles
- Dialogs
- Migrating to Qt Quick Controls
- Migration Case Study

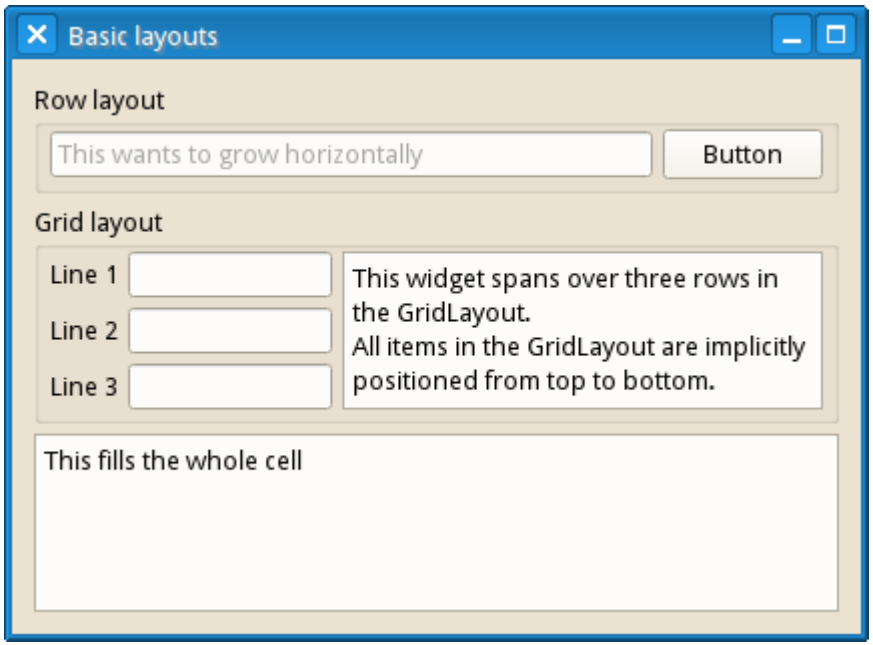
Introduction

- Qt Quick Layouts and controls are new in Qt 5.1
  - Requires an import of Qt Quick 2.1
- Suitable for mouse oriented desktop applications
- Touch screen oriented controls to be released later
- Qt 5.1 also provides new dialogs

Introduction

- Other attempts at QML components
  - Symbian and MeeGo components (both defunct)
  - Blackberry 10, Jolla, KDE Plasma
- Qt Quick Controls are different
  - Standardized
  - Cross-Platform

Qt Quick Layouts



Qt Quick Layouts

- Manage the position and size of child items
  - Similar to widget layout managers
  - More flexible than positioners
- Suitable for general QML use
  - Not limited to desktop controls
- `RowLayout`, `ColumnLayout`, `GridLayout`
- Can be nested
- Can be combined with anchors

Qt Quick Layouts

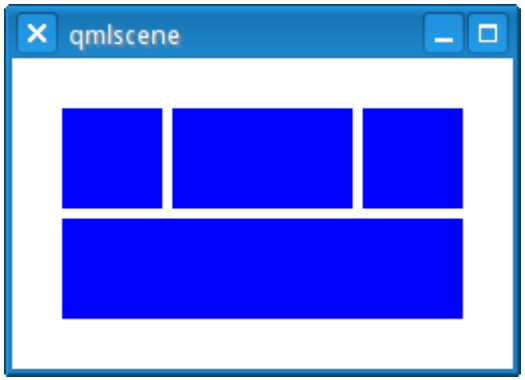
- Layout object provides attached properties
- Stretchable
  - `fillHeight`, `fillWidth`
- Size constraints
  - `maximumHeight`, `maximumWidth`
  - `minimumHeight`, `minimumWidth`
  - `preferredHeight`, `preferredWidth`
- Grid position and spans
  - `column`, `columnSpan`, `row`, `rowSpan`

Qt Quick Layouts

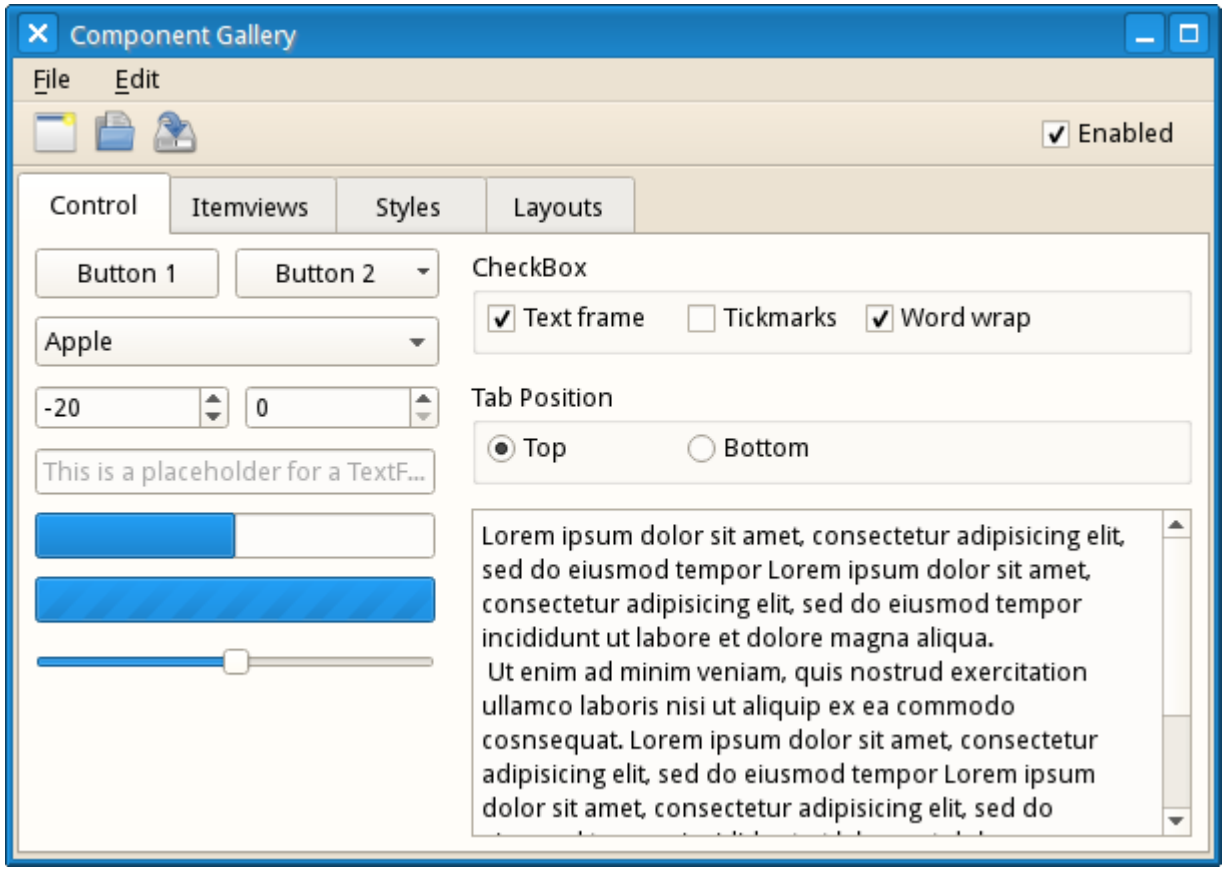
- Fill rules
  - If `fillHeight` and `fillWidth` are true, the item can grow or shrink between its minimum and maximum sizes
  - If `fillHeight` and `fillWidth` are false, the item is set to its preferred size
- Preferred size defaults to an item's implicit size
  - Roughly analogous to a size hint
- Note that there are no stretch factors

Qt Quick Layouts

```
...
ColumnLayout {
    RowLayout {
        Rectangle {
            Layout.preferredHeight: 50
            Layout.preferredWidth: 50
        }
        Rectangle {
            Layout.preferredHeight: 50
            Layout.fillWidth: true
        }
        Rectangle {
            Layout.preferredHeight: 50
            Layout.preferredWidth: 50
        }
    }
    Rectangle {
        Layout.preferredHeight: 50
        Layout.fillWidth: true
    }
}
```



Qt Quick Controls



Qt Quick Controls

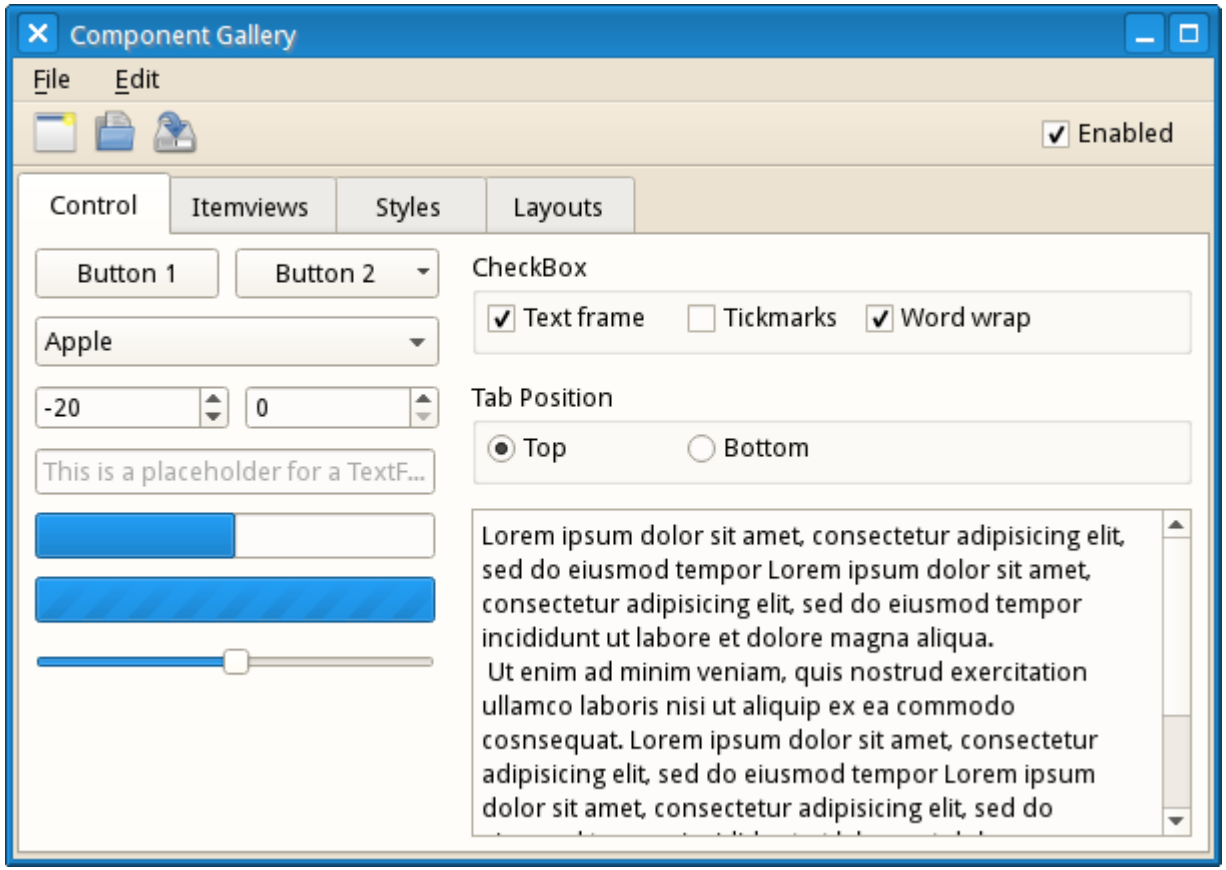
- Set of components to create user interfaces in QML
- Provide traditional mouse oriented controls
  - Touch oriented controls will be provided in the future
  - Not limited to the desktop
- If `QApplication` is used the controls will follow the desktop look and feel
  - Uses `QStyle` underneath
  - Otherwise a generic pixmap based style is used

Qt Quick Controls

- Can't cover all controls in detail here
- For more information see the page "Qt Quick Controls" in Assistant or Creator

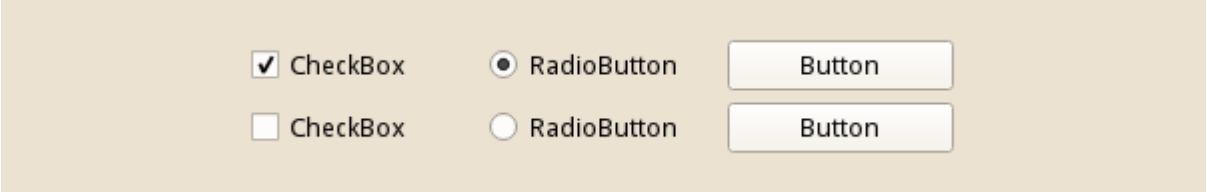
A quick whirlwind tour...

Desktop Controls



Button Controls

- All have similar properties and signals
  - `text`, `tooltip`, `pressed`, `checked`, `hovered`, etc.
  - `clicked()`
- `Button` - standard push button
- `ToolButton` - typically used in toolbars
- `CheckBox` - checkable indicator
- `RadioButton` - exclusive checkable indicator



Button Controls

- `ExclusiveGroup` is an item to control the exclusivity of checkable controls: Actions, buttons, menu items
- All button types can be checkable and exclusive

```
ExclusiveGroup { id: group }
RadioButton {
    exclusiveGroup: group
    checked: true
}
RadioButton {
    exclusiveGroup: group
}
```

Input Controls

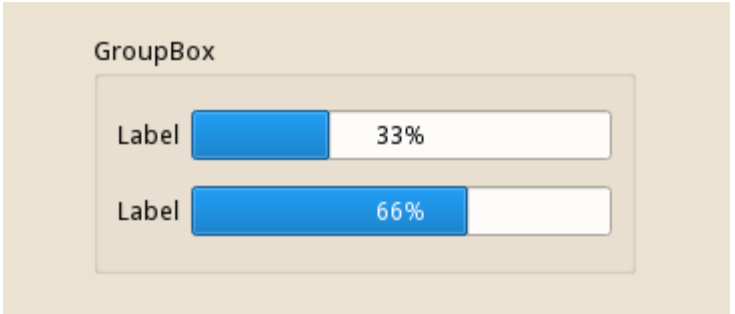
- `ComboBox` - drop down list of items
  - Items provided by a model or string list
- `Slider` - Sliding range control
  - Horizontal or vertical orientation
  - Default range is 0 to 1
- `SpinBox` - Classic spin box control
  - Value type is real
  - Zero decimals and 1.0 step size by default

Input Controls

- `TextArea` - multi-line rich text editor control
  - Embeds a `TextEdit` within a `ScrollView` (coming up)
  - Provides scrollbars and follows the system font and palette
  - Access to the underlying text document
- `TextField` - single line plain text editor control
  - Adapts a `TextInput` to the desktop
  - Provides a frame and follows the system font and palette

Display Controls

- `Label` - display text
  - Inherits `Text` type
  - Follows the system font and palette
- `ProgressBar` - displays progress of a operation
  - Horizontal or vertical orientation
  - Update `value` property as progress changes
- `GroupBox` - group box frame with title
  - Container for other controls



Qt 5.2 Controls

- Coming in Qt 5.2.0
- BusyIndicator – Spinner/throbber style indicator
- Switch – Touch style toggle switch
- Both are stylable
- Both are suitable for desktop or touch interfaces

Views and Navigation

- `ScrollView` - scrolling view
  - Can replace or decorate a `Flickable`

```
ScrollView {
    Image { imageSource: "largeImage.png" }
}
```

- `SplitView` - lays out items with a splitter handle
  - Horizontal or vertical orientation
  - Can be nested
  - Supports a subset of `Layout` properties on children

Views and Navigation

- `StackView` - provides for stack-based navigation
  - `push()` and `pop()` allow for navigation "history"
  - Could be used to create wizards and information flow
  - NOT the same as `QStackedLayout`!
- `TabView` – provides a tab style interface of pages
  - Uses Tab, which is inherited from Loader
  - `addTab()`, `getTab()`, `insertTab()` all return a `Tab` object

Views and Navigation

- `TableView` - scrollable list view with columns
  - Specialized `ListView`, displays each item as a row
  - Each field/role is a column
  - Data provided by `ListModel` or custom model
  - Column header visible by default, there is no row header
- `TableViewColumn`
  - Defines column titles and roles
  - `alignment`, `width`, `elide`, etc.

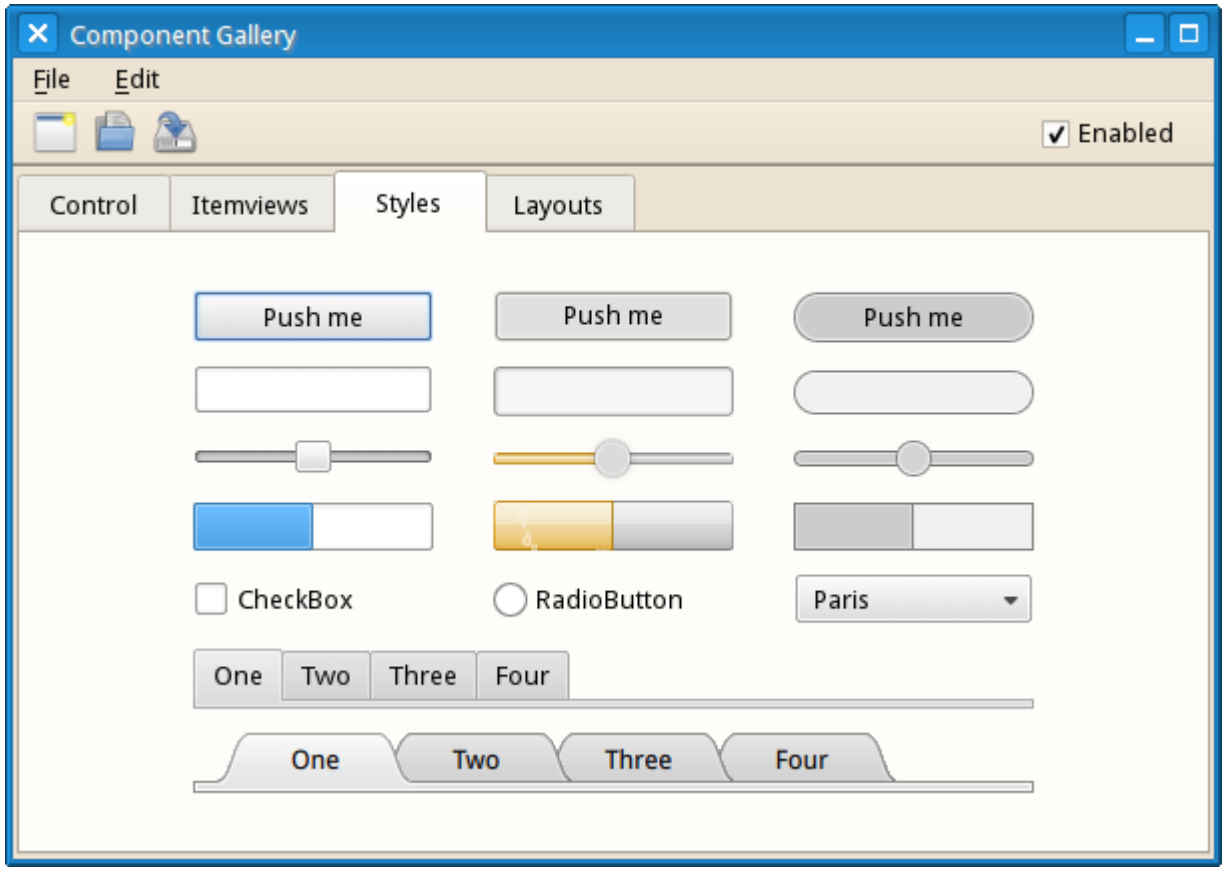
Views and Navigation

```
ListModel {
    id: phoneBook
    ListElement{ name: "Bill" ; phone: "555-3264" }
    ListElement{ name: "Bert" ; phone: "555-4267" }
    ListElement{ name: "Tom"  ; phone: "555-0473" }
}

TableView {
    model: phoneBook
    TableViewColumn { role: "name"; title: "Name" }
    TableViewColumn { role: "phone"; title: "Phone" }
}
```

What's Missing?

- Some widgets in the QtWidgets module do not have any corresponding item in the Qt Quick Controls module
  - `QDockWidget`, `QDial`, `QDateEdit`, `QtimeEdit`, `QCalendar`, `QLCDNumber`, `QTreeView`, etc.
- More controls are coming in future releases...

Styles


Component Gallery
File
Edit
Enabled
Control
Itemviews
Styles
Layouts
Push me
Push me
Push me
CheckBox
RadioButton
Paris
One
Two
Three
Four
One
Two
Three
Four

Custom Styles

- Controls will use the style provided by the application
  - `QApplication` apps use use `QStyle` to render themselves
  - `QGuiApplication` apps use a default generic style
- Can provide custom QML based styles for controls
- Custom styles are applied to individual controls
- Style types must be imported

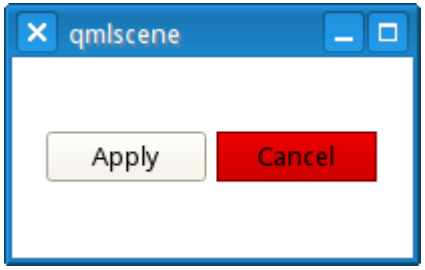
```
import QtQuick.Controls.Styles 1.0
```

Custom Styles

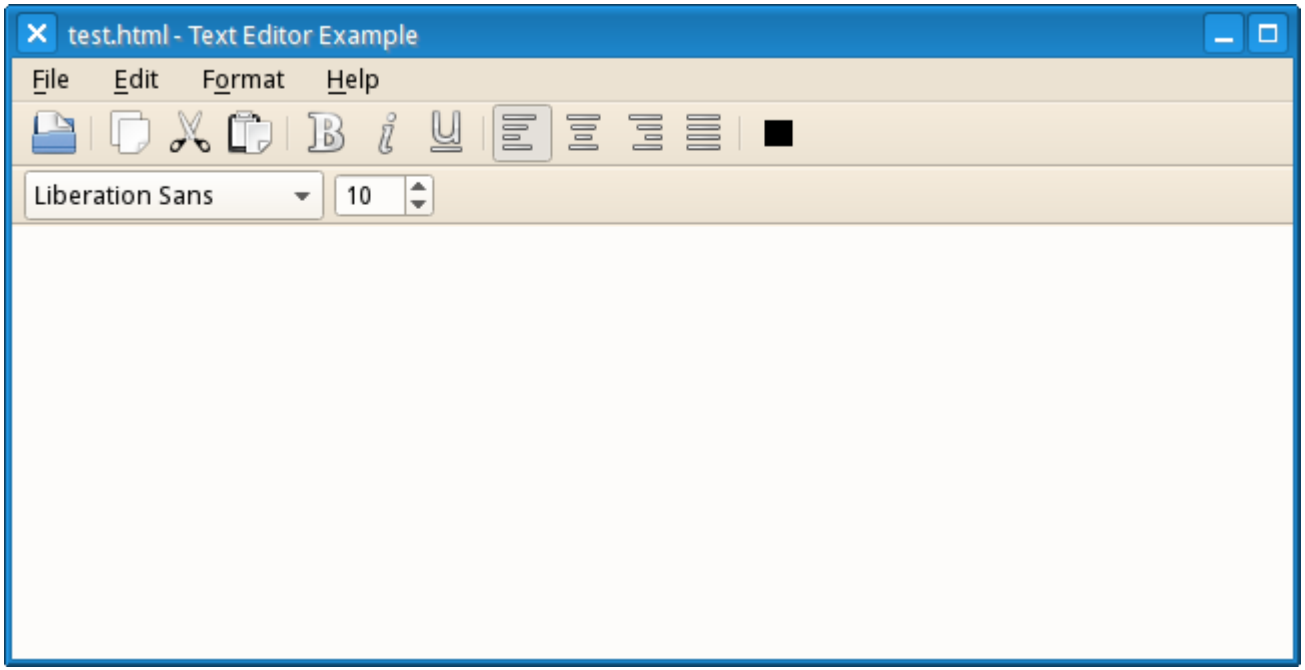
- Controls have a `style` property
- Can be set to the appropriate style component
  - `Button` → `ButtonStyle`
  - `Slider` → `SliderStyle`
  - etc.
- Some controls don't use custom styles
  - `Label`, `Menu`, `GroupBox`, etc.

Custom Styles

```
Button {
    id: cancel
    text: "Cancel"
    style: ButtonStyle {
        background: Rectangle {
            border.width: 1
            border.color: "#800"
            gradient: Gradient {
                GradientStop { position: 0;
                    color: control.pressed ? "#c00" : "#e00" }
                GradientStop { position: 1;
                    color: control.pressed ? "#a00" : "#c00" }
            }
        }
    }
}
```



Applications



Application Window

- `ApplicationWindow`
  - Top level window providing main window items
  - Properties: `menuBar`, `toolBar`, `statusBar`
- `MenuBar` can contain `Menus`, `MenuItems`, and `MenuSeparators`
- `ToolBar` is designed for `ToolButtons` and related controls, but can contain any item
- `StatusBar` and `ToolBar` do not provide layouts of their own, but are typically used with a `RowLayout`

Application Window

- Content is parented to an implicit content area
- Some limitations:
  - Toolbars are not drag-able or dock-able
  - No facilities for dock windows
  - Not a layout, does not provide layout constraints

Application Window

```
ApplicationWindow {
    id: window
    menuBar: MenuBar {
        Menu { MenuItem {...} ...}
        Menu { MenuItem {...} ...}
        ...
    }

    toolBar: ToolBar {
        RowLayout {
            anchors.fill: parent
            ToolButton {...}
            ToolButton {...}
            ...
        }
    }
   ...
```

Application Window

```
    ...
    statusBar: StatusBar {
        RowLayout {
            Label { text: window.filename }
            CheckBox { text: "Read Only" }
        }
    }

    Item {
        id: content
        anchors.fill: parent
        ColumnLayout {
            ...
        }
    }
}
```

Action

- `Action`: Interface object that can be bound to items
- Support for `Button`, `ToolButton`, and `MenuItem`
  - Bind an Action object to their action property
- Properties:
  - `text`, `iconSource`, `iconName`, `checkable`, `checked`, `enabled`, `shortcut`, `tooltip`, etc.
- Signals:
  - `toggled(checked)`
  - `triggered()`

Action

```
Action {
    id: openAction
    text: "&Open"
    shortcut: "Ctrl+O"
    iconName: "document-open"
    iconSource: "images/document-open.png"
    onTriggered: fileDialog.open()
}
...
MenuItem { action: openAction }
...
ToolButton { action: openAction }
...
```

QQmlApplicationEngine

- Traditional QML applications used `QQuickView`
  - Reparents the root item to an implicit window
  - This conflicts with a root `ApplicationWindow`
- Desktop apps can use `QQmlApplicationEngine`
  - Combines `QQmlEngine` and `QQmlComponent`
  - Many conveniences
    - Connects `Qt.quit()` signal, loads translations, etc.
  - But very different from `QQuickView`!

QQmlApplicationEngine

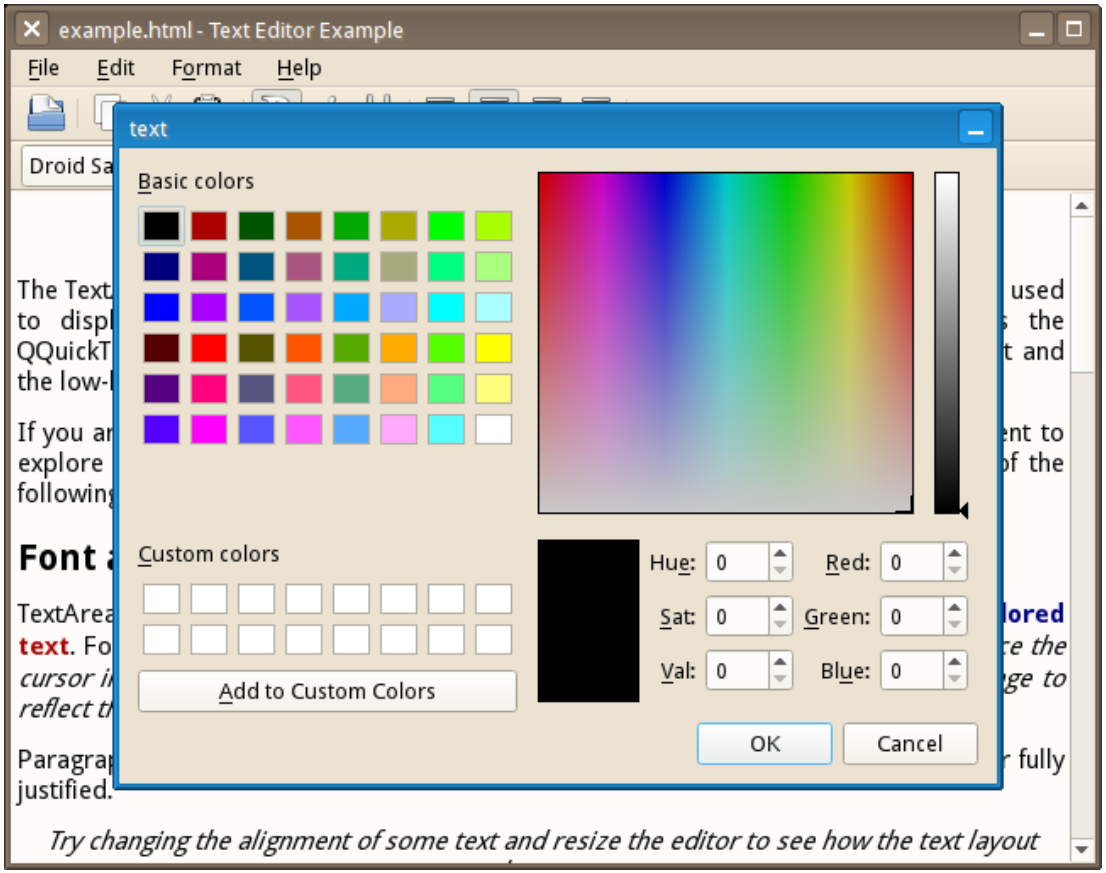
```
int main(int argc, char **argv)
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine("main.qml");

    QObject *root = engine.rootObjects().at(0);
    QQuickWindow *window =
                  qobject_cast<QQuickWindow*>(root);
    if (!window) {
        qFatal("Error: No window found!");
    }
    window->show();

    return app.exec();
}
```

QQmlApplicationEngine

- Can avoid casting the root object to `QQuickWindow`
  - Set `ApplicationWindow`'s visibility property
  - `AutomaticVisibility` will show window at startup
    - Windowed or fullscreen, depending on platform default

```
ApplicationWindow {
    id: mywindow
    title: "My Window"
    visibility: AutomaticVisibility
    ...
}
```

Dialogs


example.html - Text Editor Example
File
Edit
Format
Help
text
Basic colors
Custom colors
Add to Custom Colors
Hue:
Sat:
Val:
Red:
Green:
Blue:
OK
Cancel
Try changing the alignment of some text and resize the editor to see how the text layout

Qt Quick Dialogs

- Qt 5.1 also includes two standard dialogs
  - `ColorDialog` and `FileDialog`
  - Qt 5.2 will include `FontDialog`
- Will use the standard platform dialogs if possible
  - Will fall back to `QColorDialog` and `QFileDialog`
  - ...or QML implementations if not available

Qt Quick Dialogs

```
import QtQuick.Dialogs 1.0
...

FileDialog {
    id: fileOpenDialog
    title: "Please choose a text file"
    nameFilters: ["Text files (*.txt)"]
    selectExisting: true
    selectMultiple: false

    onAccepted: {
        document.file = fileUrl
    }
}
```

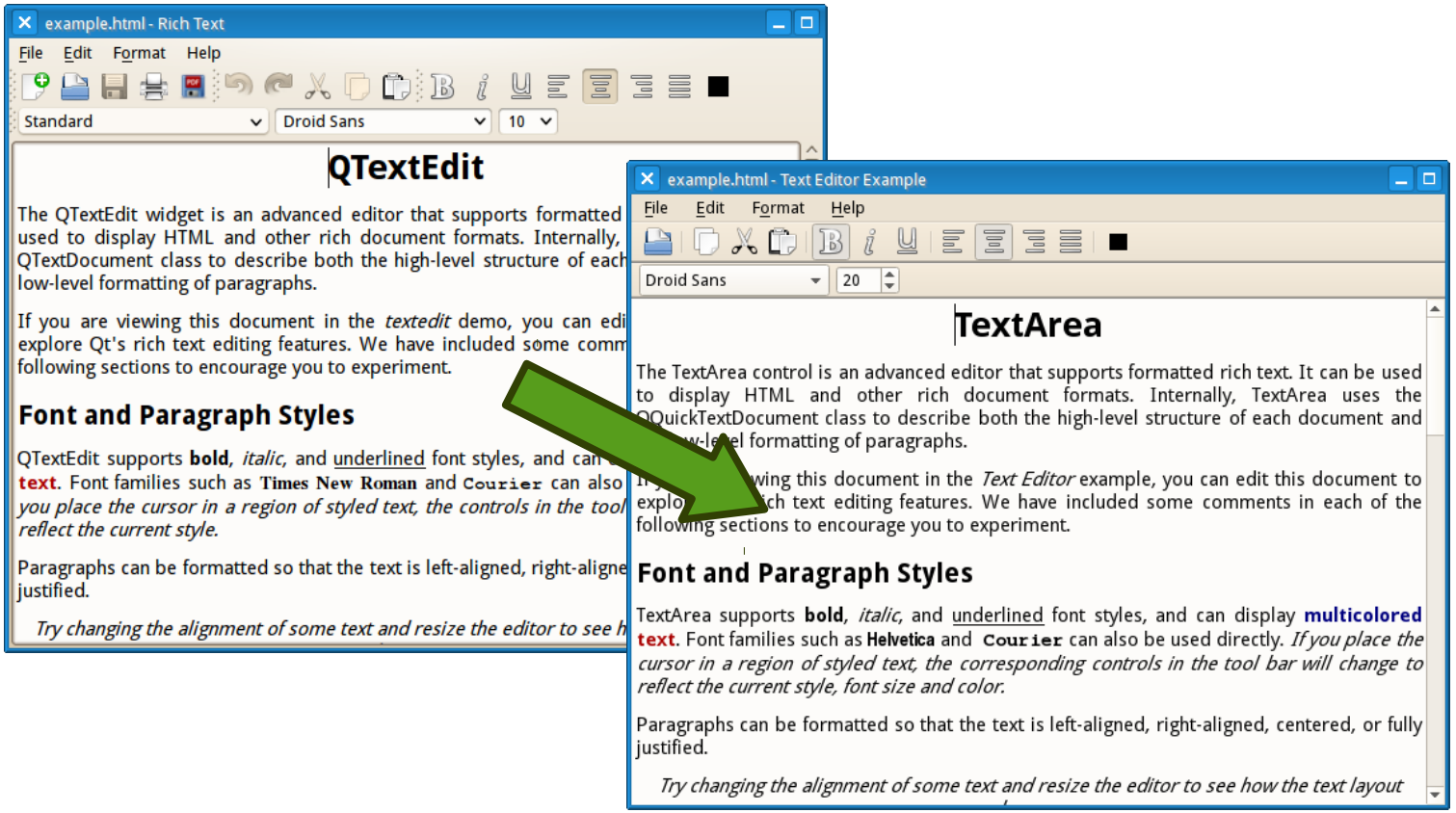
Qt Quick Dialogs

- What about other standard dialogs?
  - Widget based dialogs can still be used
  - Requires the use of `QApplication`
  - `QFontDialog`, `QMessageBox`, etc.
- Write your own dialogs using `ApplicationWindow`
  - flags property should be set to `Qt.Dialog`
  - modality property set to `Qt.WindowModal` for modal dialogs

Custom Dialogs

```
ApplicationWindow {
    id: messageBox
    Width: 280 ; height: 120
    title: "Status"
    flags: Qt.Dialog | Qt.WindowCloseButtonHint
    modality: Qt.WindowModal

    ColumnLayout {
        anchors.fill: parent

        Label {
            Layout.fillWidth: true ; Layout.fillHeight: true
            text: "Processing has finished"
        }
        Button {
            text: "Ok"
            onClicked: messageBox.close()
        }
    }
}
```

Migrating to Qt Quick Controls

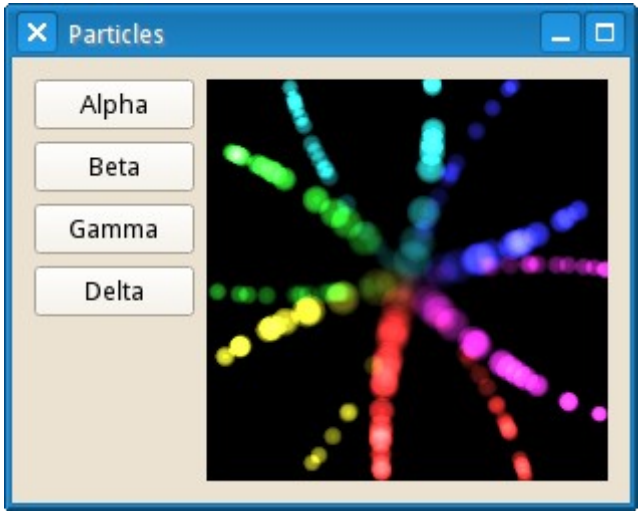


Widgets vs Controls

- When should you use Controls versus Widgets?
- Use Widgets if:
  - Application is a traditional desktop app
- Use Qt Quick Controls if:
  - Integrating QML into the desktop
  - Leveraging Qt Quick for fluid animations and transitions
  - Needing widget-like controls in a QML based application
- Largely subjective, personal preference
  - Use what you feel comfortable with

Integrating Controls

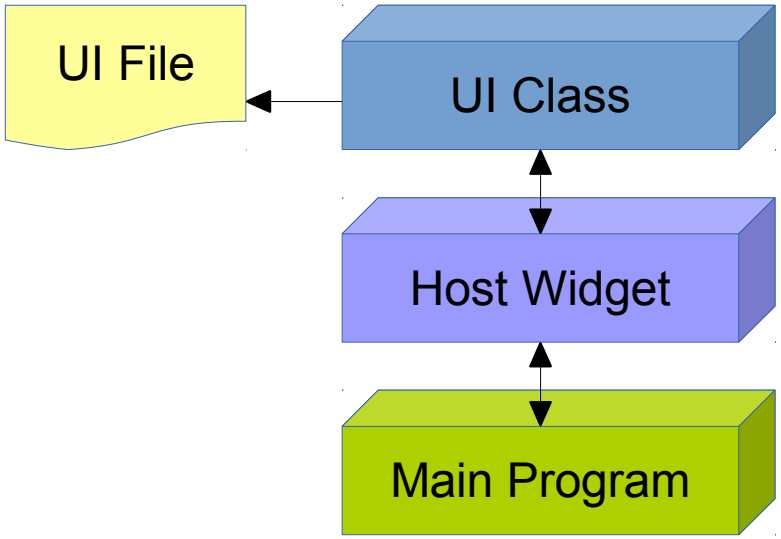
- Can embed `QWindow` instances into a `QWidget`
  - Allows embedding of QML items into a `QWidget` via `QQuickView`
  - New in Qt 5.1



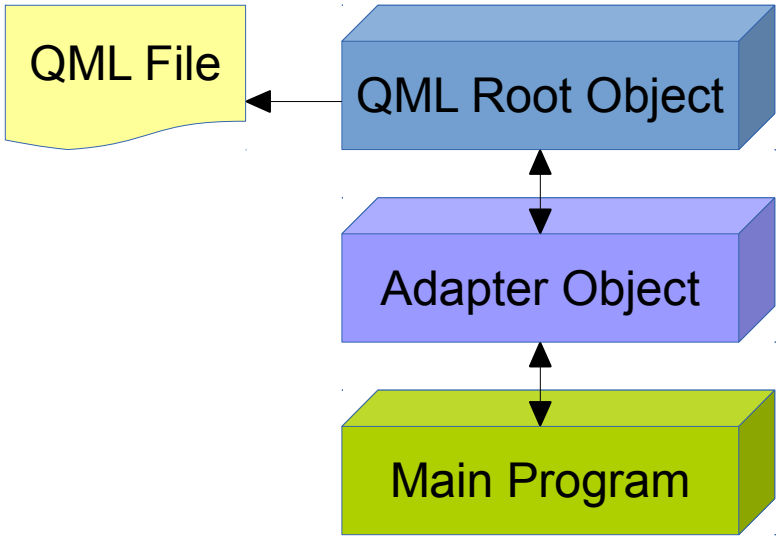

```
QQuickView *view = new QQuickView();
view->setSource(QUrl("Embedded.qml"));
QWidget *container = QWidget::createWindowContainer(view);
container->setMinimumSize(view->size());
mainlayout->addWidget(container);
```

Architecture

- C++ and UI File



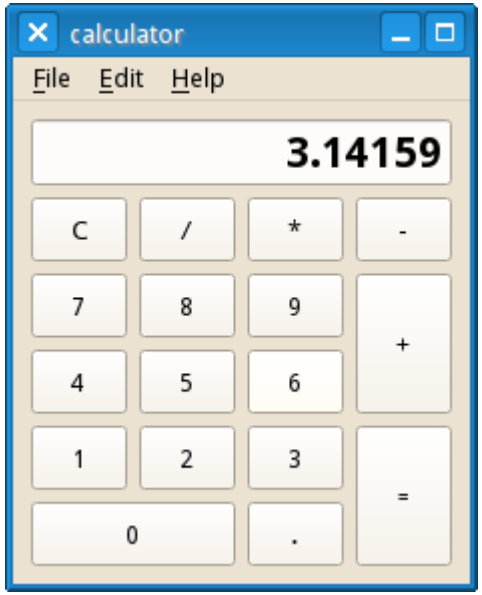
- C++ and QML



Architecture

- Clean separation between C++ and QML
- Keep imperative JavaScript to the barest minimum
- QML root object is the root of the QML object tree
  - `Window` or `ApplicationWindow`
- Adapter object communicates with the root object
  - Via properties, signals, and functions
  - Should be limited in scope to the user interface
- The backend application is ignorant of UI details

Migration Case Study



Migration Case Study

- Ported a `QWidget` based application to Qt Quick
  - Simple desktop calculator example written in C++
  - Only parts of the source code will be shown
- Porting is not straight forward
  - Good architecture makes it easier

Let's run through the code quickly...

calculator.pro

```
TARGET = calculator
TEMPLATE = app

QT += qml quick widgets

SOURCES += main.cpp calculator.cpp

HEADERS  += calculator.h

OTHER_FILES += qml/Calculator.qml

RESOURCES += icons/icons.qrc qml/qml.qrc
```

widgets for the desktop

Calculator.qml

```
import QtQuick 2.1
import QtQuick.Controls 1.0
import QtQuick.Layouts 1.0
import QtQuick.Dialogs 1.0

ApplicationWindow {
    id: calculator
    title: "Calculator"

    property alias displayText: display.text
    signal keyClicked(int key)
    signal cutTriggered()
    signal copyTriggered()
    signal pasteTriggered()
    signal aboutTriggered()
    ...
```

necessary imports

public interface

Calculator.qml

```
Action {
    id: quitAction
    text: "&Quit"
    shortcut: "Ctrl+Q"
    iconSource: "application-exit.png"
    iconName: "application-exit"
    onTriggered: Qt.quit()
}
Action {
    id: cutAction
    text: "Cu&t"
    shortcut: "Ctrl+X"
    iconSource: "edit-cut.png"
    iconName: "edit-cut"
    onTriggered: calculator.cutTriggered()
}
...
```

Calculator.qml

```
menuBar: MenuBar {
    id: mainMenu
    Menu {
        title: "&File"
        MenuItem { action: quitAction }
    }
    Menu {
        title: "&Edit"
        MenuItem { action: cutAction }
        MenuItem { action: copyAction }
        MenuItem { action: pasteAction }
    }
    Menu {
        title: "&Help"
        MenuItem { action: aboutAction }
    }
}
...
```

using actions

Calculator.qml

```
GridLayout {
    id: mainLayout
    columns: 4
    anchors.fill: parent

    TextField {
        id: display
        Layout.columnSpan: 4
        Layout.fillWidth: true
        font.pointSize: 16;
        font.bold: true
        horizontalAlignment: TextInput.AlignRight
        maximumLength: 15
        readOnly: true
    }
    ...
```

Layout properties

Calculator.qml

```
Button {
    text: "C"
    Layout.preferredWidth: 48
    Layout.preferredHeight: 40
    Layout.fillWidth: true
    Layout.fillHeight: true
    onClicked: calculator.keyClicked(Qt.Key_Delete)
}

Button {
    text: "/"
    Layout.preferredWidth: 48
    Layout.preferredHeight: 40
    Layout.fillWidth: true
    Layout.fillHeight: true
    onClicked: calculator.keyClicked(Qt.Key_Slash)
}
...
```

main.cpp

```
int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QQmlApplicationEngine engine(QUrl("qrc:/Calculator.qml"));

    QObject *root = engine.rootObjects().at(0);
    QQuickWindow *window = qobject_cast<QQuickWindow*>(root);
    if (!window) {
        qCritical("Error: No window found");
        return -1;
    }

    Calculator calculator;
    calculator.setWindow(window);

    return app.exec();
}
```

QQmlApplicationEngine

calculator.h

```
class Calculator : public QObject
{
    Q_OBJECT
public:
    Calculator(QObject *parent = 0);
    ~Calculator();
    void setWindow(QQuickWindow *window);

public slots:
    void editCut();
    void editCopy();
    void editPaste();
    void helpAbout();
    void keyClicked(int key);

private:
    QQuickWindow *mWindow;
};
```

calculator.cpp

```
void Calculator::setWindow(QQuickWindow *window)
{
    if (mWindow != 0) mWindow->disconnect(this);
    mWindow = window;

    if (mWindow) {
        mWindow->setIcon(":/accessories-calculator.png");

        connect(mWindow, SIGNAL(keyClicked(int)),
                this, SLOT(keyClicked(int)));
        connect(mWindow, SIGNAL(cutTriggered()),
                this, SLOT(editCut()));
        ...

        mWindow->setProperty("displayText", "0");
    }
}
```

accessing QML

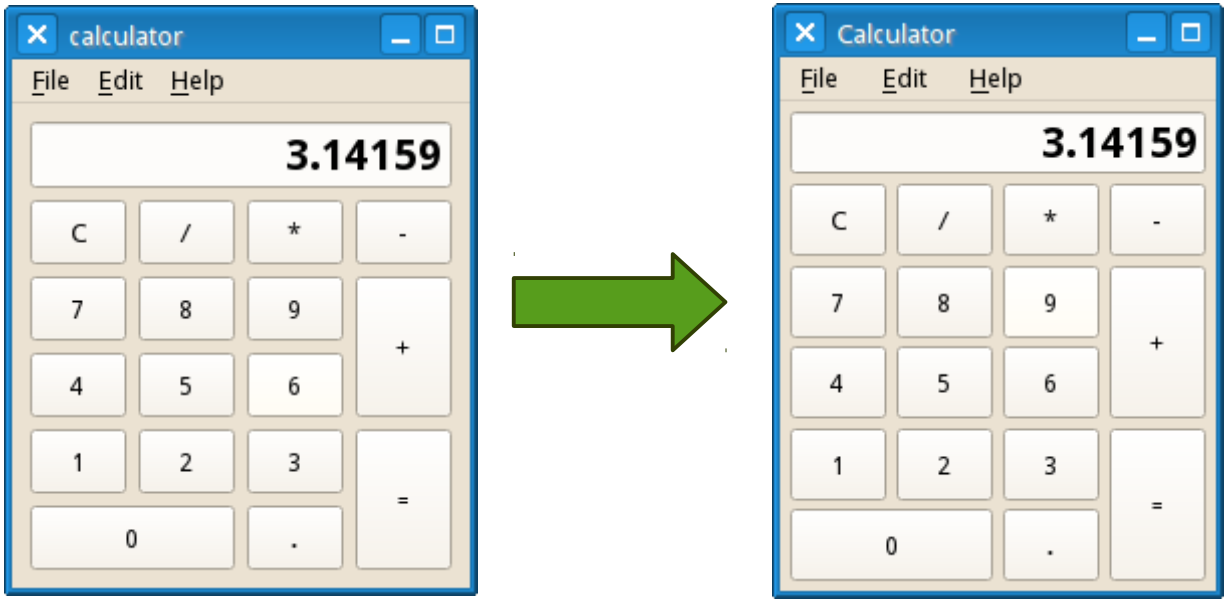
calculator.cpp

```
void Calculator::editCut()
{
    editCopy();
    mWindow->setProperty("displayText", "0");
}

void Calculator::editCopy()
{
    QString text = mWindow->property("displayText").toString();
    qApp->clipboard()->setText(text);
}

void Calculator::editPaste()
{
    mWindow->setProperty("displayText",
                         qApp->clipboard()->text());
}
```

accessing QML

Final Result

ftp://ftp.ics.com/pub/pickup/calculator-controls.zip



Thank You!