Qt Tips & Tricks

Presented by:

Integrated Computer Solutions Inc.

Agenda

- Qt Tips and Tricks
- Effective QML

Qt Tips and Tricks

Agenda

- MetaObject Tips
  - Qt Properties, Dynamic Function Calling
- Implicit Sharing Tips
  - Using Your Data Types
- Model – View Tips
  - QAbstractItemModel As Interface To Data
- Performance Tips
  - Looping, QPixmap/QImage, QNetworkAccessManager
- Threading Tips
  - Cross Thread Signals/Slots, Event Loops
- Miscellaneous Tips
  - ComboBox itemData(), i18n, casts

Meta Object Tips

QObject

- Heart and Soul of Qt Object
  - Signals and Slots are implemented here
  - QObjects can have “child objects”
    - Parents have some control over children
      - Deleting them, laying them out, etc
  - Also Qt Properties!

Introspection

- QObjects can report at runtime
  - Class name, Super class
  - Lists of signals and list their arguments
  - Lists of functions and list their arguments
  - Invoke methods by name
    - `QMetaObject::invokeMethod(objPtr, "function");`

Meta Object Compiler

- Introspection info is generated by moc
  - Reads header files. Writes source code
    - `moc -o moc_class.cpp class.h`
  - MetaObject is static
    - One instance per QObject subclass

Print Enum and Flag Values

- Qt’s meta object system records enum names
  - Information usually thrown out by the compiler
  - Makes it trivial to save/print string values for enums

Q_ENUM Example

```
class MyClass : public ... {
   Q_OBJECT

public:
   enum Priority {
      HIGH = 0,
      MED,
      LOW
   }; Q_ENUM(Priority)

};
```

Q_FLAG Example

```
class MyClass : public ... {
   Q_OBJECT

public:
   enum StatusField {
      HeaterOn = 1,
      PumpOn   = 2,
      AcPower   = 4
  };
  Q_DECLARE_FLAGS(StatusFlags, StatusField)
  Q_FLAG(StatusFlags)
};
Q_DECLARE_OPERATORS_FOR_FLAGS(MyClass::StatusFlags)
```

Accessing Enum Strings

- Via QtMeta Object
  - `obj->metaObject()->metaEnumAt(index)`
- Via qDebug()
  - `qDebug() << (HeaterOn | AcPower)`
  - (HeaterOn, AcPower) appears on stdout

Dynamic Function Calling

- `QMetaObject::invokeMethod(...)`
  - Can invoke any slot
  - Sync or Async
  - With parameters and return type
    - No access to return type for async invocations
- Useful for delayed initialization
  - Like `QTimer::singleShot()`, but with arguments!
- Useful for IPC mechanisms

QMetaObject::invokeMethod()

```
bool QMetaObject::invokeMethod(
         QObject* obj,
         const char* member,
         Qt::ConnectionType type,
         QGenericReturnArgument ret,
         QGenericArgument val0,
         ...
         QGenericArgument val9 );
```

invokeMethod Helper Macros

- `QGenericReturnArgument` and `QGenericArgument` are internal helper classes for argument marshalling
- Do not use these classes directly, use the convenience macros instead
  - Create a `QGenericArgument`
    `Q_ARG( Type, const Type & value )`
  - Create a `QGenericReturnArgument`
    `Q_RETURN_ARG( Type, Type& value )`

invokeMethod Connection Type

- Invoke method immediately (synchronous)

```
QMetaObject::invokeMethod(
          obj, "doStuff",
          Qt::DirectConnection);
```

- Place in event queue and invoke method when event is processed (asynchronous)

```
QMetaObject::invokeMethod(
          obj, "doStuff",
          Qt::QueuedConnection);
```

When to use invokeMethod

- Use when calling methods “by name”
  - Method name does not have to be known at compile time
    - Think IPC
- Use for delayed invocation
  - Method calls will be posted to the event loop
  - …potentially the event loop of other threads

    This is how cross-thread signals/slots work

Implicit Sharing Tips

Implicitly Shared Classes

- Most of Qt’s Data Classes are implicitly shared
  - Copies of classes point to the same internal data
    - Very fast copies. Saves memory
    - Reference counted
  - Data is actually copied on modification
    - Copy-On-Write semantics
  - `QString`, `QPixmap`, `QImage`, `QByteArray`, etc
- You can roll your own implicitly shared classes
  - Using the same classes Qt Engineers use!

[ContainerClass]::detach()

- Copy of data is performed in the `detach()` function
  - If this appears in profiling data you may be accidentally copying data

Custom Implicitly Shared Data

- Inherit from `QSharedData`
  - Provides required **`ref()`** and **`deref()`** impls
    - These are atomic thread-safe impls
- Create a flyweight object with `QSharedDataPointer<>`as a private member
  - Hides sharing implementation from client code

Custom Implicitly Shared Data

```
class EmployeeData : public QSharedData
{
public:
   EmployeeData() : id(-1) { name.clear(); }
   EmployeeData(const EmployeeData &other)
       : QSharedData(other), id(other.id),name(other.name){}
   ~EmployeeData() { }

   int id;
   QString name;
};
```

Custom Implicitly Shared Data

```
class Employee
{
public:
   Employee() { d = new EmployeeData; }
   Employee(int id, QString name) {
        d = new EmployeeData;
        setId(id);
        setName(name);
   }
  Employee(const Employee &other)
       : d (other.d) {}
private:
   QSharedDataPointer<EmployeeData> d;
};
```

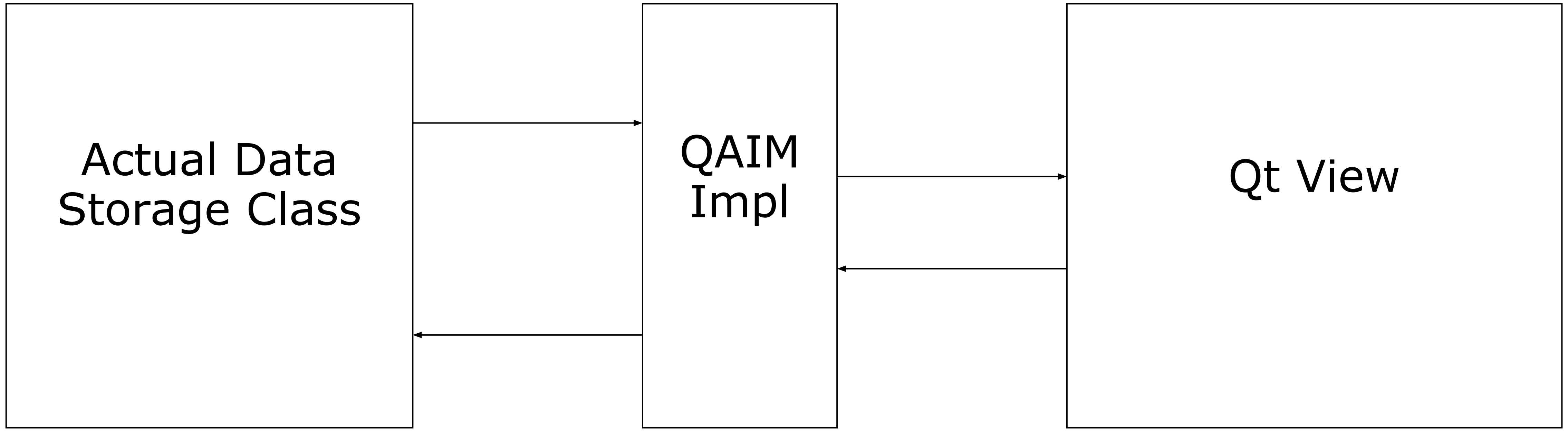
Model – View Tips

Model – View Tips

- Avoid using all-in-one Model/View widgets
  - `QListWidget`, `QTableWidget`, `QTreeWidget`
    - Needs to copy data!
    - Syncing issues WILL arise
    - Only really useful for simple lists, small amount of data, that does not change.
- Avoid using `QStandardItem` Models
  - Same reasons as above.

Model – View Tips

- Use `QAbstractItemModel` (QAIM) as an Interface
  - Wrap your data with QAIM for use with Qt's Model-View Classes



QModelIndex

- Representation of a cell
  - Row, Column, Parent
  - `QAIM::index(int row, int col, QModelIndex parent)`
  - Used through the QAIM API
- Internal implementation is
  - Row, Column, QAIM*, Identifier (void* or int)
  - QAIM::`createIndex(int row, int col, void* ptr)`
  - Very Small. Very Fast.
- Transient objects DO NOT STORE!
  - Could be instantly invalidated by inserts/removes

QPersistentModelIndex

- Storable QModelIndex
  - Implicit conversion to/from QModelIndex
- Model Index that is maintained by the Model
  - Row incremented on another row inserted
  - Row decremented on another row removed
  - Index set to QModelIndex() when row is removed
- Watch out for performance issues
  - Updating these indexes does take time

QAIM API

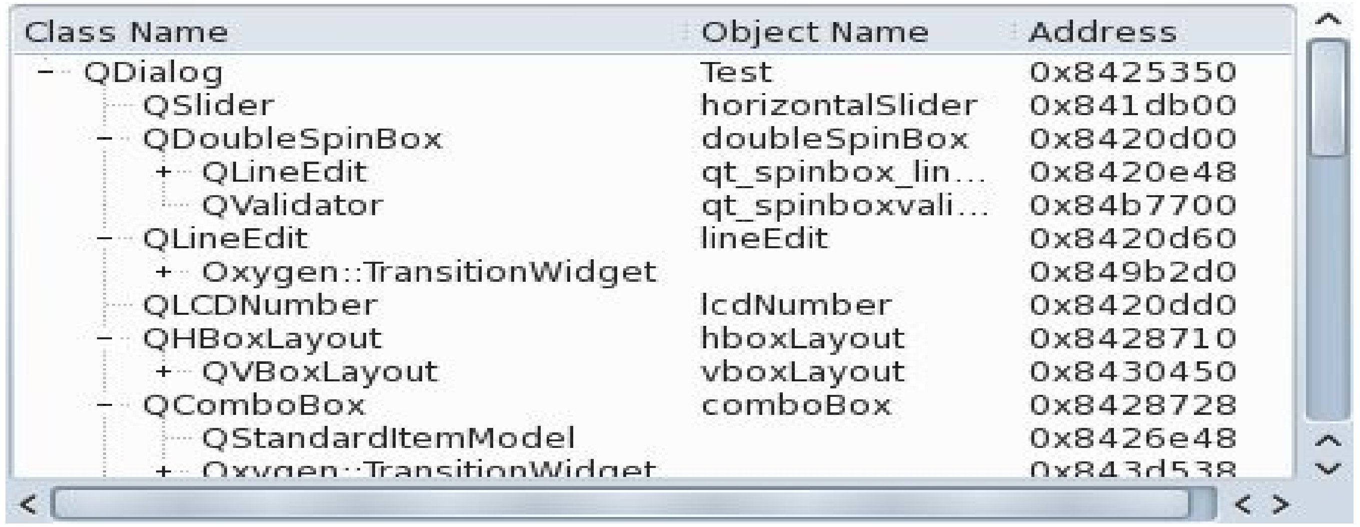
- Read Only Tables (Use `QAbstractTableModel`)
  - `int rowCount(const QModelIndex &parent = QModelIndex()) const`
  - `int columnCount(const QModelIndex &parent = QModelIndex()) const`
  - `QVariant data(const QModelIndex &index, int role) const`
    - Different roles for display, editing, pixmap, etc.
    - It’s like a 3rd Dimension. Cells have role depth.
- Editable Tables (Use `QAbstractTableModel`)
  - `bool setData(const QModelIndex &index, const QVariant &value, int role)`
  - `void insertRows(int row, int count, QModelIndex parent)`
  - `void removeRows(int row, int count, QModelIndex parent)`

QAIM API

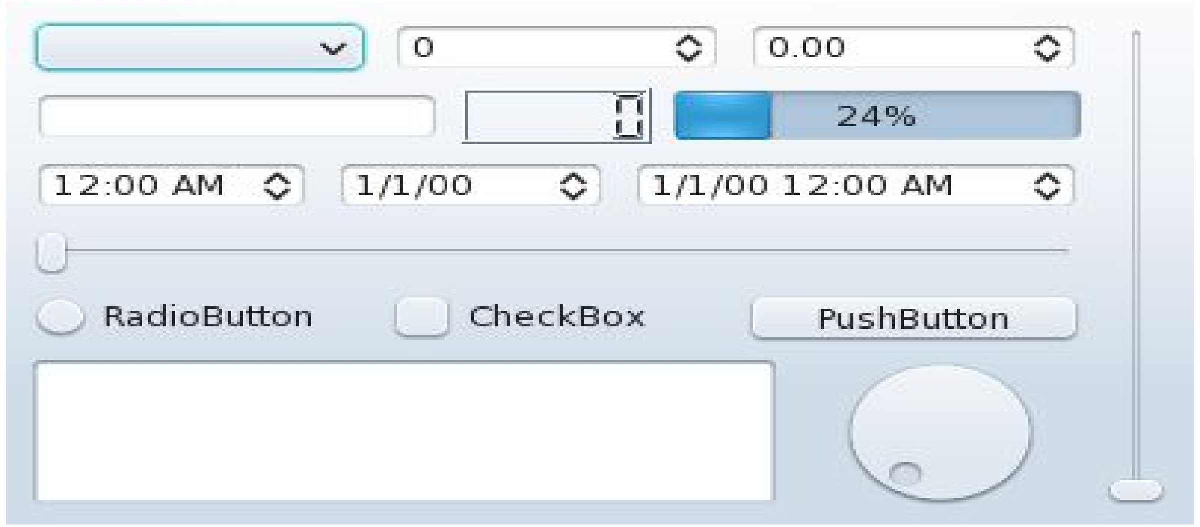
- Trees (Use `QAbstractItemModel`)
  - `QModelIndex index(int row, int column, const QModelIndex &parent) const`
  - `QModelIndex parent(const QModelIndex &child) const`
- Implementations of the above can be a little mind bending
  - But well worth the effort

Model – View Example

- Example of QAIM as a wrapper.
  - To Qt Application Widget Hierarchy!
    - It’s a doubly linked tree! parent() and children()
  - Extremely short code wrapper code.
  - Check out the ObjectBrowser Example!



```
Class Name                          Object Name          Address
− QDialog                           Test                 0x8425350
    QSlider                         horizontalSlider     0x841db00
  − QDoubleSpinBox                  doubleSpinBox        0x8420d00
    + QLineEdit                     qt_spinbox_lin...    0x8420e48
      QValidator                    qt_spinboxvali...    0x84b7700
  − QLineEdit                       lineEdit             0x8420d60
    + Oxygen::TransitionWidget                           0x849b2d0
    QLCDNumber                      lcdNumber            0x8420dd0
  − QHBoxLayout                     hboxLayout           0x8428710
    + QVBoxLayout                   vboxLayout           0x8430450
  − QComboBox                       comboBox             0x8428728
      QStandardItemModel                                 0x8426e48
    + Oxygen::TransitionWidget                           0x843d538
```



Performance Tips

Looping Performance Tips

- Use Iterators!
  - Maps, Hashes, Linked List Iterators are much faster than [i] index lookups.
  - Code is more complex, but worth it.
  - Most data classes in Qt are implicitly shared
    - Don’t be afraid to copy dereferenced iterator values
      - Const reference is still better.

STL Iterators

- Compatible with STL Algorithms
  - Const and non-const versions
  - Always use const version when appropriate
- Forwards

```
QList<int>::iterator i;
for (i = list.begin(); i != list.
end(); ++i)
   *i += 2;
```

- Reverse

```
QList<QString>::iterator i = list.
end();
while (i != list.begin()) {
   --i;
   *i += w; }
```

Java-like Iterators

- Iterators with a Java Style API
  - Roughly symmetrical forward and reverse APIs
  - Mutable iterator classes allow list modification
- Forwards

```
QListIterator<QString> i(list);
while (i.hasNext())
   qDebug() << i.next();
```

- Reverse (and Mutable)

```
QMutableListIterator<int> i(list);
i.toBack();
while (i.hasPrevious()) {
   if (i.previous() % 2 != 0)
      i.remove(); }
```

Looping Performance Tips

- Use const references for foreach()
  - Yes, Qt has it’s own foreach macro. Use it!
    - Avoids typos/fence posting when iterating a whole container
  - Using a const ref variable avoids a copy

```
foreach(const BigData& data, bigList)
{
   doSomething(data);
}
```

QImage vs. QPixmap

- `QImage`
  - Platform independent array bitmap
  - Lives in application memory space
  - Easy to manipulate pixels (Query/set colors)
  - Needs to be copied to graphics memory to draw

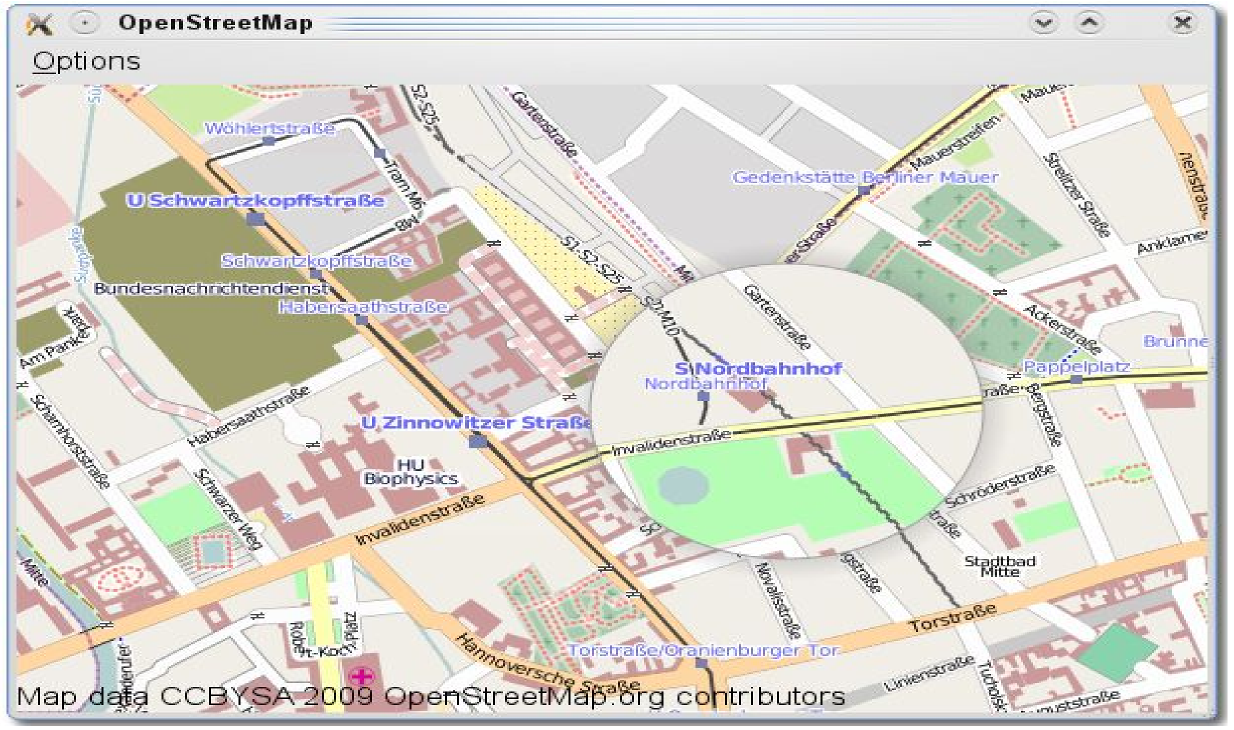
- `QPixmap`
  - Native representation of a bitmap (YUV, etc)
  - Lives in System (X Server) or even GPU Memory
  - No ability to set individual pixels
  - Very fast to draw. Bitmap is closer to hardware.

QNetworkAccessManager

- `QNetworkAccessManager` (QNAM) is awesome
  - Multiple protocols (HTTP/FTP/HTTPS)
  - SSL integrated
  - Provides caching of data
    - Can be persistent across runtimes
- LightMaps from Qt Labs is a perfect example

QNetworkAccessManager Cache

- LightMaps example Uses OpenStreetMap tiles
  - QNAM automatically caches tiles as they are loaded
    - Makes panning much faster
    - And code is very clean



QNetworkAccessManager Cache

- In constructor

```
m_manager = new QNetworkAccessManager(this);

QNetworkDiskCache *cache = new QNetworkDiskCache;
cache->setCacheDirectory(cachePath);
m_manager->setCache(cache);

connect(m_manager, SIGNAL(finished(QNetworkReply*)),
        this, SLOT(handleNetworkData(QNetworkReply*)));
```

QNetworkAccessManager Cache

- In download() (simplified to fit)

```
QString path = "http://domain.org/%1/%2/%3.png";
m_url = QUrl(path.arg(zoom).arg(x).arg(y));

QNetworkRequest request;
request.setUrl(m_url);
request.setAttribute(QNetworkRequest::User, grab);
m_pendingReplies << m_manager->get(request);
```

Miscellaneous Tips

Safer Casts with qobject_cast

- **`qobject_cast<>()`** is a library safe dynamic cast
  - Behaves much like dynamic_cast
    - Returns NULL pointer on error
- Uses Qt Meta-Object System (Introspection)
  - moc records all signals, slots, properties
    - Also inheritance hierarchy and string class names
- Actual impl compares static QMetaObject*s
  - Fast! Faster than gcc's **`dynamic_cast<>`**

Finding Children

- `T findChildren<T>(QString name=QString())`
  - Returns descendants from any level of parenting tree
  - Built-in `qobject_cast<>()`

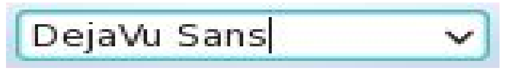
```
QList<MyWidget*> children = findChildren<MyWidget*>();
//Children has all instance of MyWidget in dialog.

QList<MyWidget*> children = findChildren<MyWidget*>(“Hi”);
//Children has one instance of MyWidget with
//objectName() == Hi
```

Use QComboBox itemData()

- Use user data when inserting items
    - `addItem(const QString & text, const QVariant userData)`
    - QVariant is a wrapper class for many Qt data classes
        - Can be extended to support custom classes
- Easy way to store mappings
    - User selectable string/Icon to Enum/Color/Font

Use QComboBox itemData()

- Font Combo Box Example:



```
MyDialog::MyDialog() {
   QStringList fonts = QFontDatabase::families();
   foreach(QString family, fontList)
      m_combo->addItem(family, QFont(family));
}

MyDialog::indexChanged(int index) {
   setFont(m_combo->itemData(i).toFont());
}
```

i18n Tips

- Wrap all user visible strings in tr()
  - lupdate, lrelease and linguist take care of the rest
- Use static QObject::translate() outside of QObject scope
- Be careful when combining strings
  - `"File " + fileName + " saved."`
    - Can’t easily be translated
  - `QString("File %1 saved.").arg(fileName);`
    - % identifiers can be moved by the translation
    - %1 - %99 can be used in any string

Threading Tips

Threading Tips

- Use classes that use background processing
  - `QNetworkAccessManager`, `QHostInfo`
  - Sockets, etc
- Use Qt event loops for producer/consumer
  - You don’t have to write synchronization code

Event Loop Work Queues

- Use per-thread Qt event loops as work queues
  - Use cross thread signals and slots to assign work
  - Use cross thread signals and slots to return results
  - Avoids locking the work queue
    - `QEventLoop` has built-in locks

Threading Tips

- Create a worker thread with **`run() {exec();}`**
  - This is the default impl of run()
- Connect signals to thread slots to dispatch work
- Connect to thread signals to get results
- Watch out for QThread’s Thread Affinity
  - It belongs to the thread that CREATED it
  - Not a big deal, just use a helper class created in the spawned thread.

Event Loop Work Queues

```
connect(this, SIGNAL(workAvailable(WorkType)),
         thread->worker(), SLOT(doWork(WorkType)));

connect(thread->worker(), SIGNAL(workComplete(WorkType)),
         this, SLOT(processWorkDone(WorkType))
…
//Auto Connection will cause events to be dispatched to the other thread
emit workAvailable(work);
…
void processWorkDone(WorkType) { //Work is received
};
```

Event Loop Work Queues

```
class MyThread : public QThread
{
public:
   void run() {exec();}
   Worker* worker();
};
class Worker : public QObject
{
signals:
   workComplete(WorkType work);
public slots:
   doWork(WorkType work);
};
```

General Threading Tips

- Use QMutex with QMutexLocker
  - Constructor Locks; Destructor Unlocks

```
void exclusiveFunction() {
    QMutexLocker mutexLocker(m_mutex); //Constructor locks
    …
} //Destructor unlocks
```

- Be careful not to hold the lock too long with a scoped lock!

Effective QML

Agenda

- Building Blocks of QML
- Declarative Code
- Creating New Item Types
- Dynamic Item Creation
- States
- Using C++ and QML

Building Blocks of QML

QQuickItem

- Most Qt Objects inherit `QObject`
  - `QQuickItem` is no exception
    - Gets many of it’s features directly from `QObject`
  - We will be leveraging these capabilities throughout class

Qt Properties

- Combination of Get/Set/Notify
  - Allows introspection system to use these functions as one concept
  - Properties have been in Qt for a very long time
    - Qt Designer is based on properties
    - QML is also based on properties

Declaration of a Qt Property

```
#include <QObject>

class Car : public QObject
{
   Q_OBJECT
   Q_PROPERTY(int value READ value WRITE setValue NOTIFY valueChanged)

public:
   int getValue() const;
   void setValue(int newValue);

signals:
   void valueChanged(int value);
};
```

Declarative Code

Basic QML Syntax

- QML is declarative language
  - With hooks for procedural JavaScript
    - Use as little JavaScript as possible
- QML files a read at runtime
  - The declarative parts create C++ instances
  - JavaScript is JIT interpreted

QtQuick Hello World

```
import QtQuick 2.2

Rectangle{
   id: toplevel
   color: "blue"
   Text {
      text: "Hello World"
   }

   MouseArea {
      anchors.fill: parent
      onClicked: Qt.quit()
   }
}
```

Qt Quick Items

- Rectangle, Text and MouseArea
  - Are implemented in C++
  - Instances of `QQuickRectangle`, `QQuickText`, Etc
  - Loading QML is slower than compiled code
    - At runtime performance is great

QML Bindings

- ":" is the binding operator
  - Right of the binding operator is JavaScript

  -
    ```
    Text {
       text: "Hello World " + Math.rand()
    }
    ```

Bindings are Declarative

- When any property used in a binding changes the expression is recalculated
  -
    ```
    Gauge {
        value: Math.min(gaugeMax, Math.max(gaugeMin, oilPressure.value))
    }
    ```
  - Value is updated whenever properties change
    - gaugeMax, gaugeMin or oilPressure.value
  - Inline binding are anonymous functions. Auto-recalculation rules also apply when assigning a named function to a binding
    - `value: calculateValue()`

JavaScript is Procedural

- Avoid this!

```
Gauge {

       Component.onCompleted: {
         setGaugeValue(oilPressure.value)
         oilPressure.valueChanged.connect(setGaugeValue)
       }

       onGaugeMinChanged: setGaugeValue(value)
       onGaugeMaxChanged: setGaugeValue(value)

       function setGaugeValue(oilValue) {
         value = Math.min(gaugeMax, Math.max(gaugeMin, oilValue))
       }
     }
```

Broken Bindings

- Assignment operator breaks bindings
  - Binding works for awhile. Then doesn’t.

```
Gauge {
   id: gauge
   visible: Dashboard.isOilPressureVisible
}

Button {
   onClicked: { // Tries to temporarily hide gauge
      if (gauge.visible)
         gauge.visible = false
      else
         gauge.visible = Dashboard.isOilPressureVisible
   }
}
```

Creating New Items

Dividing Code Into Components

- Often a desire to put too much code in one QML file
  - Common issue for all programming languages
  - QML makes it easy to componentize your code
- Component refers to an item that can be instanced multiple times

Creating New Items

- Simply create a new .qml file
  - Type is named after the filename
    - Must begin with a capital letter
  - Implement
    - Properties
    - Signals
    - Functions

Using Custom Component

```
Rectangle{ // Main.qml
   id: toplevel
   color: "black"

   Button {
      text: "Click Me"
      onClicked: toplevel.color = "white"
   }
}
```

Custom Button Component

```
Rectangle{ // Button.qml
    id: button
    property alias text: label.text
    signal clicked()

    color: "blue"
    width: 100; height: 50

    Text {
        id: label
        anchors.centerIn: parent
    }

    MouseArea{
    id: ma
        anchors.fill: parent
        onClicked: button.clicked()
    }
}
```

Alias Properties

- Proxies properties to child items
  - Allows hiding of implementation details
  - Saves memory and binding recalculations

Property Scope

- Public Scope
  - All public properties of the root item
    - Custom properties defined on the root item
- Private Scope
  - All child items and their properties

Public Members

```
Rectangle{ // Button.qml
    id: button
    property alias text: label.text
    signal clicked()

    color: "blue"

    Text {
        id: label
        anchors.centerIn: parent
    }

    MouseArea{
        id: ma
        anchors.fill: parent
        onClicked: button.clicked()
    }
}
```

Private Members

```
Rectangle{ // Button.qml
    id: button
    property alias text: label.text
    signal clicked()

    color: "blue"

    Text {
        id: label
        anchors.centerIn: parent
    }

    MouseArea{
        id: ma
        anchors.fill: parent
        onClicked: button.clicked()
    }
}
```

Private Properties

```
Rectangle { // Button.qml
   id: button
   property alias text: label.text
   signal clicked()

   QtObject {
      id: internal
      property int centerX: button.width()/2
   }

   Text {
      x: internal.centerX
   }
}
```

Dynamic Creation of Items

Creating Items Dynamically

- Procedural Way
  - Component createObject(parent, bindings) function
- Declarative Way
  - Loader Item
  - Repeater Item
  - ListView / GridView Items

Procedural Creation

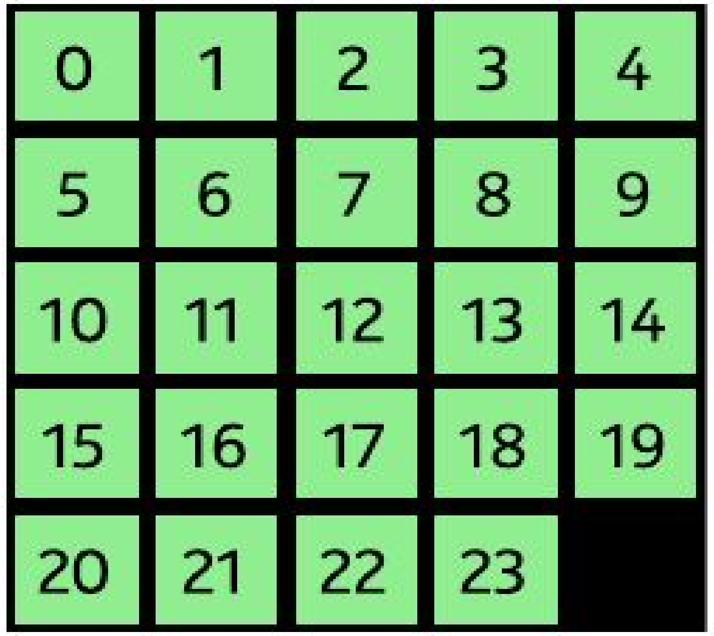
```
Item {
   id: screen
   property SettingDialog dialog: undefined

   Button {
      text: "Settings..."
      onClicked: {
         var component = Qt.createComponent("SettingsDialog.qml")
         screen.dialog = component.createObject(screen, { "anchors.centerIn": screen })
         screen.dialog.close.connect(screen.destroySettingsDialog)
      }
      function destroySettingsDialog()
       {
         screen.dialog.destroy()
         screen.dialog = undefined
      }
   }
}
```

Declarative Creation

```
Item {
      Button {
        text: "Settings..."
        onClicked: loader.sourceComponent = dialogComponent

     Loader {
        id: loader
        anchors.fill: parent
     }

     Component {
       id: dialogComponent
       SettingsDialog {
           anchors.centerIn: parent
           onClose: loader.sourceComponent = undefined
       }
     }
   }
```

Creating Multiple Items

```
Rectangle {
    width: 400; height: 400
    color: "black"
    Grid {
        x: 5; y:5
        rows: 5; columns: 5
        Repeater {
            model: 24
            Rectangle {
                width: 70; height: 70
                color: "lightgreen"
                Text {
                    anchors.centerIn: parent
                    text: index
                }
            }
        }
    }
}
```



Repeater

- Repeaters can use all types of data models
  - JavaScript Array
  - ListModel
  - JSON
  - `QList<QObject*>`
  - `QQmlListProperty`
  - `QAbstractItemModel`
- Model data is accessed via attached properties

States and Transitions

States

- State Machines can make your code “more declarative”
  - A basic state machine is built into every Item
    - No sub states or state history

States

- Every Item has a states property
  - States contain
    - Name
    - When Clause
    - List of PropertyChanges{} objects

Setting States

- Item can be set to a give state two ways
  - 1) “state” property is set to the name of the State
    - item.state = “Pressed”
  - 2) The when clause of the State is true
    - When clauses must be mutually exclusive
      - They are evaluated in creation order

Button States

```
Item {
    Rectangle { id: bkg; color: "blue" }
    MouseArea { id: ma }

    states: [
        State {
            name: "Pressed"
            when: ma.pressed
            PropertyChanges { target: bkg; color: "red" }
        },
        State {
            name: "Disabled"
            when: !(ma.enabled)
            PropertyChanges { target: bkg; color: "grey" }
        }
    ]
}
```

Default State

- The initial bindings are the “Default State”
  - The name of the default state is “”
  - Default state is in effect when
    - No when clauses are satisfied
    - “state” property is set to “”

Properties When in a State

- The bindings of a QML document are defined as
  - The default state bindings
  - Overlaid with PropertyChanges from the current state
  - This will save you a ton of typing
    - States do not need to be unwound
    - Set common properties in the default state
      - Avoids writing duplicate PropertyChanges

Transitions

- Run animations on a state change
  - Control how properties will change
    - Qt will automatically interpolate values
  - Control in which order properties change

Transitions

```
[ ... ]
transitions: [
           Transition {
               from: ""; to: "Pressed"
               PropertyAnimation { target: bkg
                   properties: "color"
                   duration: 500
               }
           },
           Transition {
               from: "*"; to: "Disabled"
               PropertyAnimation { target: bkg
                   properties: "color"
                   duration: 250
               }
           }
       ]
[ ... ]
```

Transition Defaults

- `Transition{}` defaults to
  - `from: "*"; to: "*"`
  - That Transition will apply to all state changes

- `PropertyAnimation`
  - When a target is not specified
    - That animation will apply to all items

Button Transition

```
Item {
    Rectangle { id: bkg; color: "blue" }
    MouseArea { id: ma }

    states: [
        State { name: "Pressed"; when: ma.pressed
            PropertyChanges { target: bkg; color: "red" }
        },
        State { name: "Disabled"; when: !(ma.enabled)
            PropertyChanges { target: bkg; color: "grey" }
        }
    ]
    transitions: [
        Transition {
            PropertyAnimation { properties: "color"; duration: 500 }
        }
    ]
}
```

The Behavior type

- Behavior allows you to set up an animation whenever a property changes.

```
Behavior on x { SpringAnimation {
        spring: 1
        damping: 0.2
    }

}
```

Demo qml-animations/ex-animations/spring-animation.qml

Using C++ and QML

Integrated Computer Solutions

Drive QML with C++

Model – View Pattern

- C++ code can know nothing about the UI
  - Properties, Slots and Signals are the interface in QML
    - QML Items connect or bind to C++ Objects
- Good design is enforced
  - C++ cannot depend on UI
    - Avoids "accidental" storage of data inside UI components
  - C++ is more portable to other UI frameworks

C++ Integration Techniques

- Expose object instances from C++ to QML
  - Objects appear as global variables to QML
    - Effectively singletons
- Expose C++ types to QML
  - New types are available for QML programmers to use
    - Remember how Rectangle and Text are actually C++?

Creating Properties in C++

- Properties are the combination of
  - Read function
  - Write function
  - Notify signal
    - Signals/slots is Qt’s object communication system

C++ Property Header

```
class CoffeeMaker : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int temp READ getTemp WRITE setTemp NOTIFY tempChanged)

public:
    int getTemp() const;
    void setTemp(int temp);

signals:
    void tempChanged(); //Using a parameter is not required by QtQuick

private:
    int m_temp;
};
```

Source is as usual

```
int CoffeeMaker::getTemp() const
{
    return m_temp;
}
void CoffeeMaker::setTemp(int temp)
{
    if (m_temp != temp)
    {
        m_temp = temp;
        emit tempChanged();
    }
}
```

Complex Proeprties

- `QObject*` can be used as a property
  - Used for encapsulation and creating trees of properties
    - Properties can have properties!

Invokable C++ Methods

- Methods can be called from QML
  - Any slot can be called
  - Any `Q_INVOKABLE` can be called

Invokable C++ Return Types

- Any basic Qt or C++ type
  - int, double, QString, etc
- Any returned **`QObject`*** belongs to QML
  - Will be deleted by QML during GC
  - NOTE: **`QObject`*** returned from a Q_PROPERTY
    - Belongs to C++

Invokable C++ Functions

```
class CoffeeMaker : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int temp READ getTemp WRITE setTemp NOTIFY tempChanged)

public:
    int getTemp() const;
    void setTemp(int temp);
    Q_INVOKABLE void startBrew();

public slots:
    void stopBrew();

signals:
    void tempChanged(); //Using a parameter is not required by QtQuick
};
```

Exposing Instances

```
int main(int argc, char** argv)
{
    QGuiApplication app(argc, argv);

    CoffeeMaker maker;

    QQuickView view;
    view.rootContext()->setContextProperty("maker", &maker);
    view.setSource(Qurl("qrc:/main.qml"));
    view.show();

    return app.exec();
}
```

Exposing Instances QML

```
import QtQuick 2.2

Rectangle {
    width: 1024
    height: 768

    Text {
        anchors.centerIn: parent
        text: "Coffee Temp" + maker.temp
    }

    MouseArea {
        anchors.fill: parent
        onClicked: maker.startBrew();
    }
}
```

Exposing C++ Types to QML

- Rather than making one CoffeeMaker in main
  - Allow QML Programmer to create N CoffeeMaker items
  - All of the above applies to exposed types
    - Instead of using setContextProperty
    - Use **`qmlRegisterType<>()`**

Expose C++ Types

```
int main(int argc, char** argv)
{
   QGuiApplication app(argc, argv);

   qmlRegisterType<CoffeeMaker>("MrCoffee", 1, 0,"CoffeeMaker");

   QQuickView view;
   view.setSource(QUrl("qrc:/main.qml"));
   view.show();

   return app.exec();
}
```

Expose C++ Types QML

```
import QtQuick 2.2
import MrCoffee 1.0

Rectangle {

   CoffeeMaker { id: maker }

   Text {
     anchors.centerIn: parent
     text: "Coffee Temp" + maker.temp
   }

   MouseArea {
     anchors.fill: parent
     onClicked: maker.startBrew();
   }
}
```

Integrated Computer Solutions

Thank You!

Prepared by the Engineers of
Integrated Computer Solutions, Inc.

www.ics.com