Qt Quick Best Practices Part 3

Justin Noel
Senior Consulting Engineer
ICS, Inc.

Agenda

- C++ / QML Integration
- Reusing Existing C++ Code

Using C++ and QML

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

Creating Properties in C++

- Inherit from QObject
- Use the Q_OBJECT macro
- Use the Q_PROPERTY macro

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


};
```

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();
   }
}
```

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();
}
```

Basic C++ Integration 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();
    }
}
```

Complex Proeprties

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

Complex Properties Header

```
class CoffeeMaker : public QObject
{
   Q_OBJECT
   Q_PROPERTY(QObject* options READ getOptions CONSTANT)

public:
   QObject* getOptions() const { return &m_options; };
   Q_INVOKABLE void startBrew();

private:
   Options m_options;
};
```

Complex Properties Header

```
class Options: 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();
};
```

Complex Properties QML

```
import QtQuick 2.2

Rectangle {
    width: 1024
    height: 768

    Text {
      anchors.centerIn: parent
      text: “Coffee temp” + maker.options.temp
    }

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

Enum Properties

- C++ enums can be used as QML types
  - Use Q_ENUMS macro
  - Use qmlRegisterUncreatableType<>(…)
    - Package Name
    - Major Version
    - Minor Version
    - Type Name
    - Error Message

Enum Properties Header

```
class CoffeeMaker: public QObject
{
   Q_OBJECT
   Q_ENUMS(Strength)
   Q_PROPERTY(Strength strength READ getStrength
                               WRITE setStrength
                               NOTIFY strengthChanged)

public:
   CoffeeMaker();
   enum Strength { Light, Medium, Dark };
   Strength getStrength() const;
   void setStrength(Strength newStrength);

signals:
   void strengthChanged();
};
```

Enum Properties Source

```
CoffeeMaker::CoffeeMaker()
{
  qmlRegisterUncreatableType<CoffeeMaker>(“MrCoffee”,
                                          1, 0,
                                          “CoffeeMaker”,
                                          “Do not instance.”);
}
```

Enum Properties QML

```
import QtQuick 2.2
import MrCoffee 1.0 //Needed to get TypeInfo for CoffeeMaker
Rectangle {
    width: 1024
    height: 768

    Text {
      anchors.centerIn: parent
      text:textFromStrength(maker.strength)//Evaluated as int
    }

    function textFromStrength(strength) { … }

    MouseArea {
      anchors.fill: parent
      onClicked: maker.startBrew(CoffeeMaker.Strong); //Used by name
    }
}
```

Reusing Existing Code

- QML requires that properties
  - READ functions returns correct value
    - Could be called anytime
  - NOTIFY signal emitted when prop changes
    - Immediate call to READ returns new changed value

Reuse Techniques

- Direct – Add Q_PROPERTY
  - Model is already QObject based
  - Stores its own data
- Wrapper – Write a new QObject class
  - Model is not (or cannot be) QObject based
  - Model does not store data

Direct Reuse Technique

- Use Q_PROPERTY with existing
  - READ function
  - WRITE function (optional)
  - NOTIFY signal
- Use Q_INVOKABLE on existing methods
  - Functions you want callable from the UI

Existing Header

```
class CoffeeMaker : public QObject
{
    Q_OBJECT
public:
   float targetTemp() const;
   void setTargetTemp(float taregetTemp);

   float temp() const;

signals:
   void targetTempChanged(float targetTemp);
   void tempChanged(float temp);

private:
   … //Members
};
```

Direct Reuse Header

```
class CoffeeMaker : public QObject
{
    Q_OBJECT
    Q_PROPERTY(float targetTemp READ targetTemp NOTIFY targetTempChanged)
    Q_PROPERTY(float temp READ temp NOTIFY tempChanged

public:
   float targetTemp() const;
   Q_INVOKABLE void setTargetTemp(float taregetTemp);

   float temp() const;

signals:
   void targetTempChanged(float targetTemp);
   void tempChanged(float temp);
   ...
};
```

Read Only Properties

- Properties can be read-only
  - Slots or Q_INVOKABLE functions
    - Can change state and emit signals
- Sometimes it’s cleaner to have
  - Read only properties
  - Q_INVOKABLE setter functions

Direct Reuse Issues

- Inherited NOTIFY signals compile error
  - NOTIFY signal needs be in the same class as Q_PROPERTY declaration
- Workaround:
  - Specify new signal in subclass
  - Use SIGNAL – SIGNAL connection in ctor

Wrapper Reuse Technique

- Class that provides the QObject interface
  - Inheritance
  - Composition – Easier to test
- Less chance of “rocking the boat”
  - More typing. More code

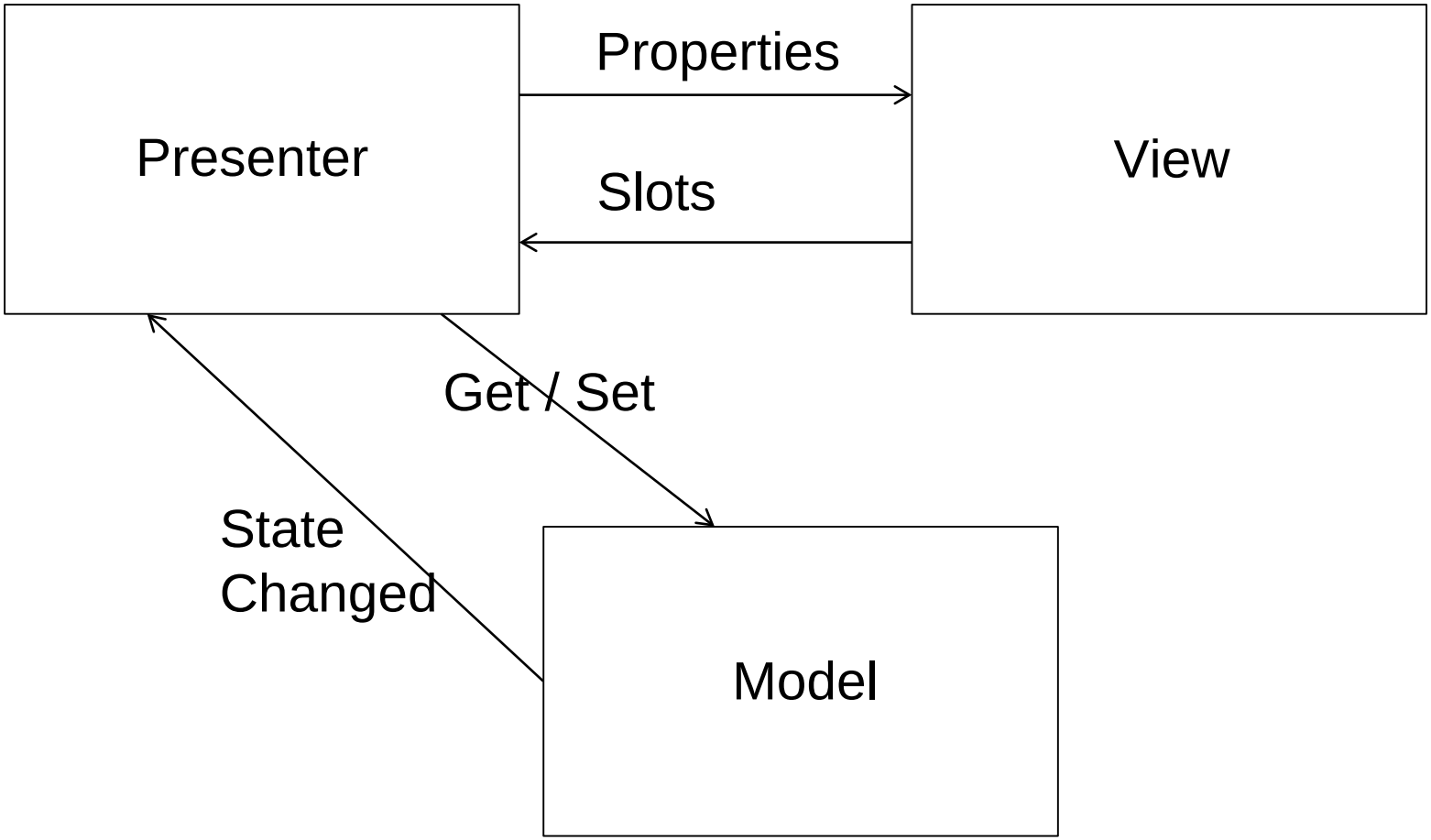
Wrappers fix a lot of issues

- Wrappers can work around limitations
  - Class is template based
    - Can’t directly inherit QObject
  - Class does not use signals
    - Uses some other callback mechanism
  - Class does not follow get/set/notify pattern
    - Wrapper can implement local data cache

Model-View-Presenter Pattern

- Wrappers can be an implementation of MVP
  - Also called Supervising Controller Pattern
  - Provides flexibility between the Model and U
    - Presentation Layer can “reformat” data
      - Create strings from multiple model values
      - Convert QList<Foo> into an QAbstractItemModel

Model – View Presenter



Existing Header

```
template<class T> class Option
{
public:
   T getSetting() const;
   void setSetting(T newSetting);

   void registerFooCallback(Callback<T>&);

private:
   T m_setting;
   CallbackCollection<T> m_settingCallbacks;
};
```

Wrapper Header

```
class DoubleOptionWrapper : public QObject, public Option<double>
{
   Q_OBJECT
   Q_PROPERTY(double setting READ getSetting WRITE setSetting
                             NOTIFY settingChanged)
public:
   DoubleOptionWrapper();

signals:
   void settingChanged();

private:
   void handleSettingCallback(double newSetting);
   Callback<double> m_settingCallback;
};
```

Wrapper Source

```
DoubleOptionWrapper::DoubleOptionWrapper() :
   m_settingCallback(this, DoubleOptionWrapper::handleSettingCallback)
{
   registerSettingCallback(m_settingCallback);
}

void DoubleOptionWrapper::handleSettingCallback(double newSetting)
{
   Q_UNUSED(newSetting)
   emit settingChanged();
}
```

Another Wrapper Example

- Issues
  - No storage of values in model
    - Does not support get function
- Does use QObject and signals

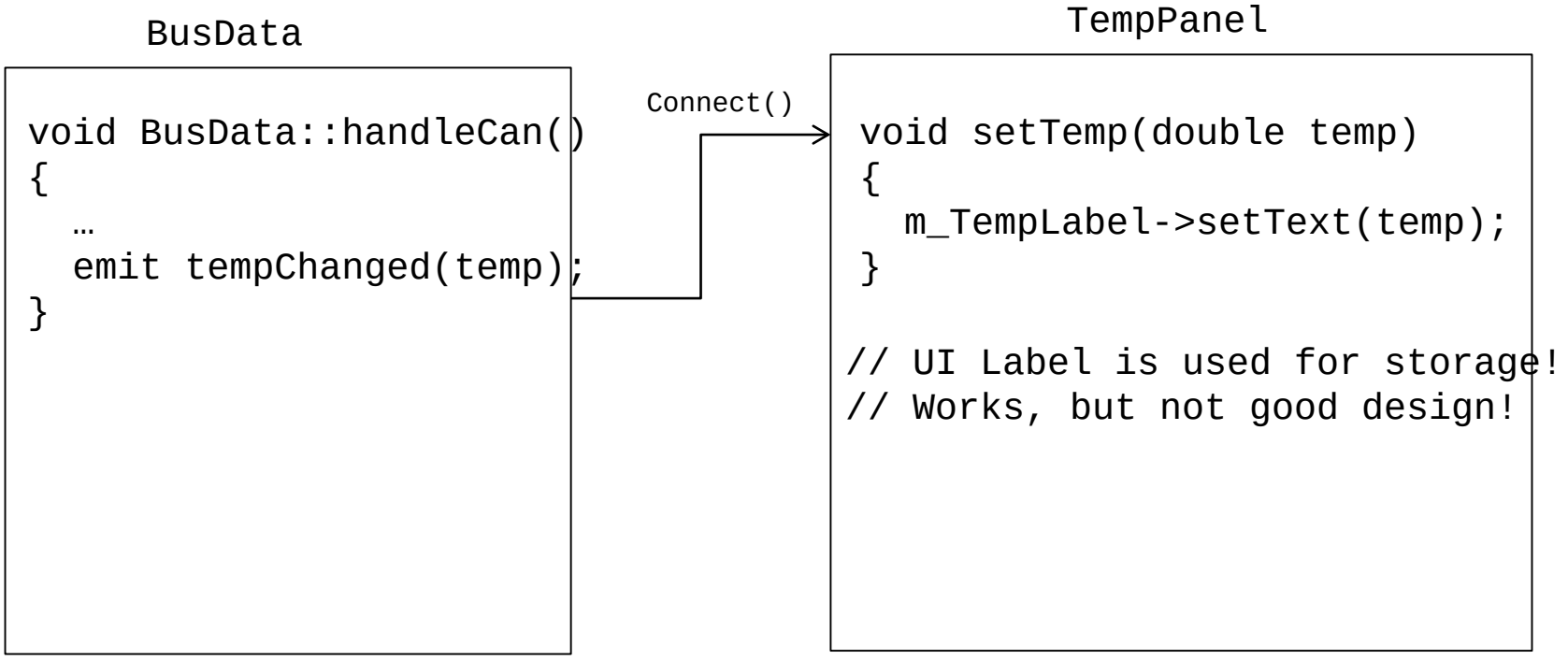
Existing Header

```
class BusData : public QObject
{
   Q_OBJECT
public:
   void requestSetTargetTemp(double temp);

signals:
   void targetTempChanged(double temp);
   void error(const QString& errorMessage);

private:
   CanBusComm m_canBus;
};
```

Existing Code Structure



Wrapper Header

```
class BusDataBridge : public QObject
{
   Q_OBJECT
   Q_PROPERTY(double targetTemp READ getTargetTemp NOTIFY targetTempChanged)

public:
   BusDataBridge(BusData& busData);
   double getTargetTemp() const;
   Q_INVOKABLE void requestSetTargetTemp(double temp);

signals:
   void targetTempChanged();

private slots:
   void handleTempTargetChanged(double temp);

private:
   BusData& m_busData;
   double m_targetTemp;
};
```

Wrapper Source

```
BusDataBridge::BusDataBridge(BusData& busData) :
   m_busData(busData)
{
   connect(m_busData, SIGNAL(targetTempChanged(double),
           this, SLOT(handleTargetTempChanged(double));

   connect(m_busData, SIGNAL(error(QString)),
           this, SIGNAL(error(QString)));
}

void BusDataBridge::handleTargetTemperatureChanged(double temp)
{
   if(m_temp != temp)
   {
      m_temp = temp;
      emit targetTemperatureChanged();
   }
}
```

Wrapper Source

```
double BusDataBridge::getTargetTemp() const
{
   return m_targetTemp;
}

void BusDataBridge::requestSetTargetTemperature(double temp)
{
   m_busData.requestSetTargetTemperature(temp);
}
```

Threading Considerations

- BusData example can be useful pattern
  - If BusData reads data on another thread
  - Sig/Slot connections work across threads
    - Qt will dispatch an async event automatically
      - Automatic Copy/Lock of data across threads
    - Storing data in Bridge object (on GUI thread).
      - Good! Avoids locking on read

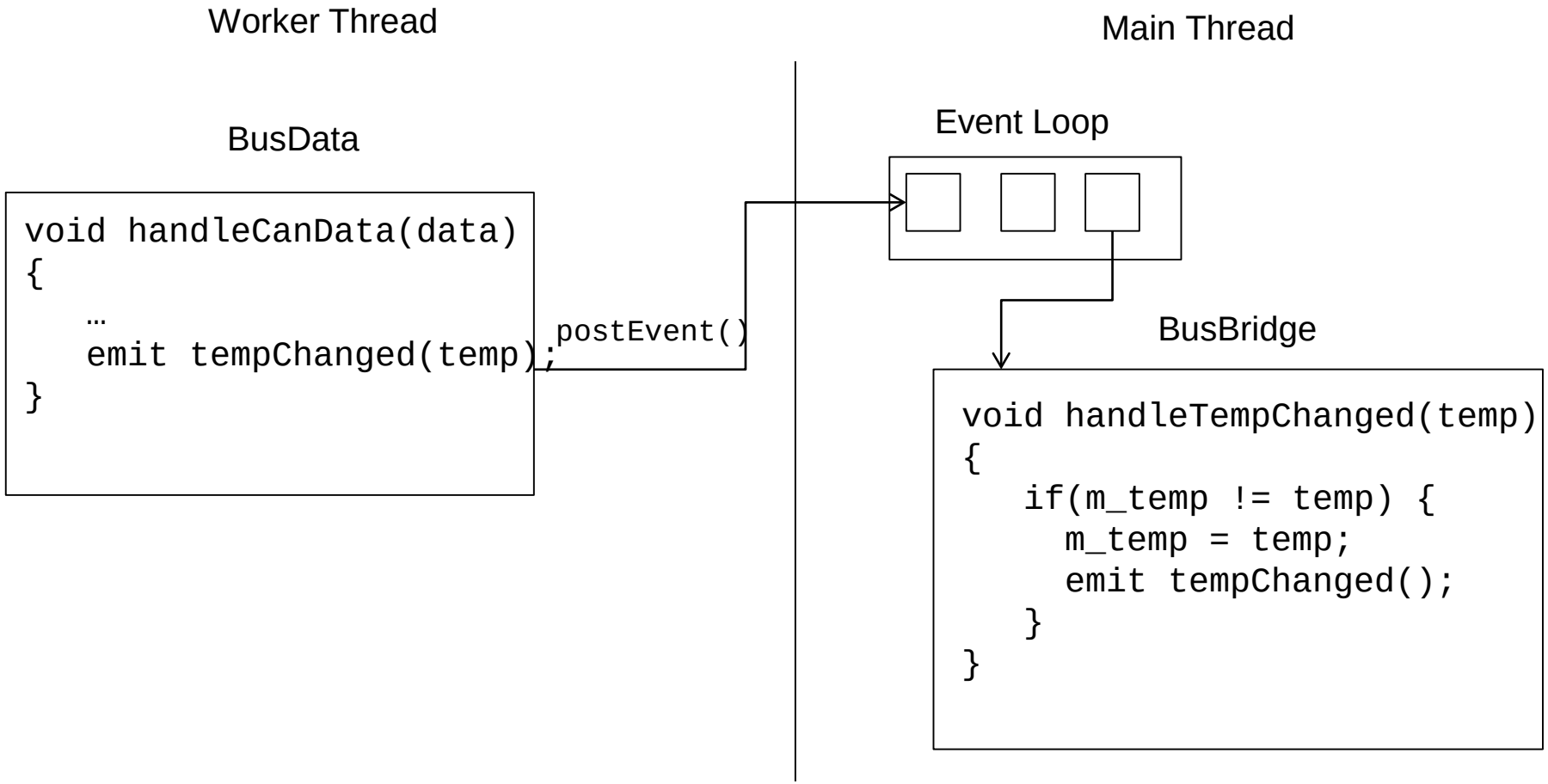
QObject Thread Affinity

- QObjects “belong” to a thread
  - Default is the thread that created the QObject
    - QObjects can be assigned another thread via
    - obj->moveToThread(otherThread)

Cross Thread Signals and Slots

- At emit time Qt compares thread ids
  - The id of the current thread calling emit signal
  - The id the receiver belongs to via obj->thread()
- If the threads are the same slots are called
- If different an event is packaged/posted

Cross Thread Signal and Slot



Passing Data Across Threads

- Signal parameters across threads
  - Need to be QVariant Compatible
    - Default constructor
    - Assignment Operator
    - Copy Constructor
    - Q_DECLARE_METATYPE(Type) at end of Header
    - qRegisterMetaType<Type>(“Type”); in Source

Implicit Sharing

- When copies are not actually copies
- Most data objects in Qt are implicitly shared
  - QString, QList, QMap, QVector, etc
- Implemented w/ thread safe ref counting
  - Copy constructor and assignment operator
    - Copies an internal pointer and increments the ref count

Exposing C++ Types to QML

- Rather than making 1 CoffeeMaker in main
  - Allow QML Programmer to create N CoffeMaker 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();
    }
}
```

Thank You!

Justin Noel
Senior Consulting Engineer
ICS, Inc.