Quick Introduction to Qt Programming

CMPS160

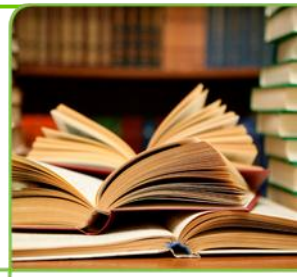
NOKIA

© 2010 Nokia Corporation and its Subsidiary(-ies).

The enclosed Qt Educational Training Materials are provided under the Creative Commons Attribution-Non-Commercial-Share Alike 2.5 License Agreement.

The full license text is available here: http://creativecommons.org/licenses/by-nc-sa/2.5/legalcode.

Nokia, Qt and the Nokia and Qt logos are the registered trademarks of Nokia Corporation in Finland and other countries worldwide.



What is Qt?

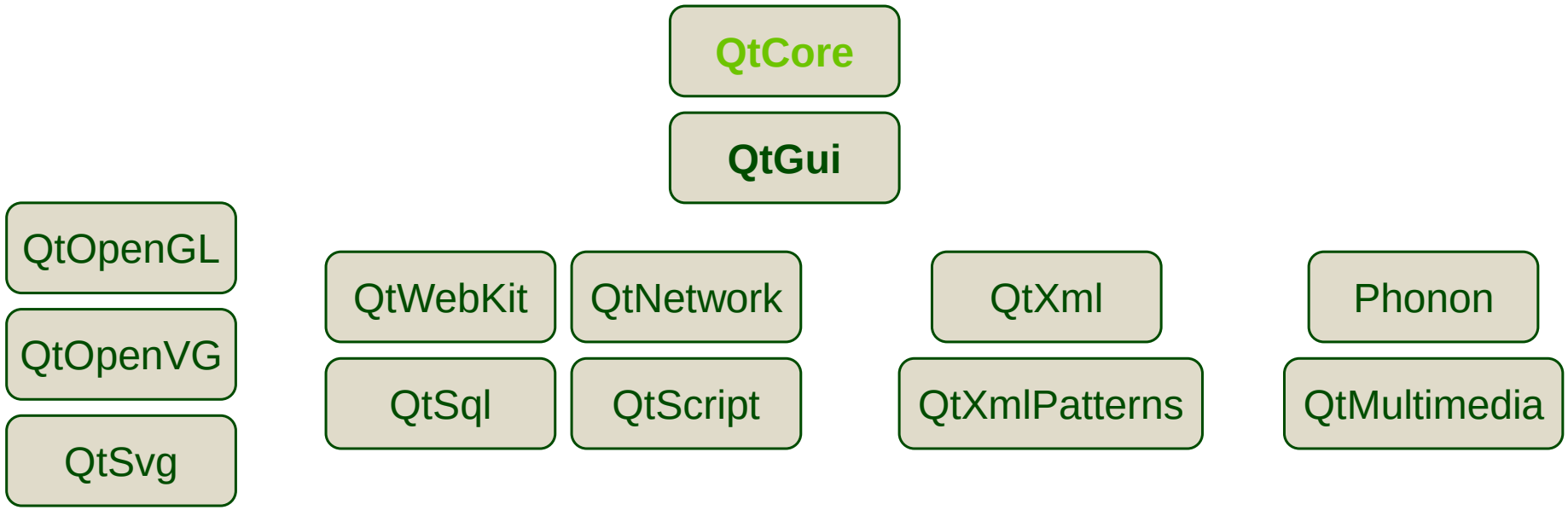
”Qt is a cross platform development framework written in C++.”

- C++ framework – bindings for other languages
  - Python, Ruby, C#, etcetera
- Originally for user interfaces – now for everything

  Databases, XML, WebKit, multimedia, networking, OpenGL, scripting, non-GUI...

What is Qt?

- Qt is built from modules
  - All modules have a common scheme and are built from the same API design ideas



What is Qt?

- Qt extends C++ with macros and introspection

```
foreach (int value, intList) { … }
```

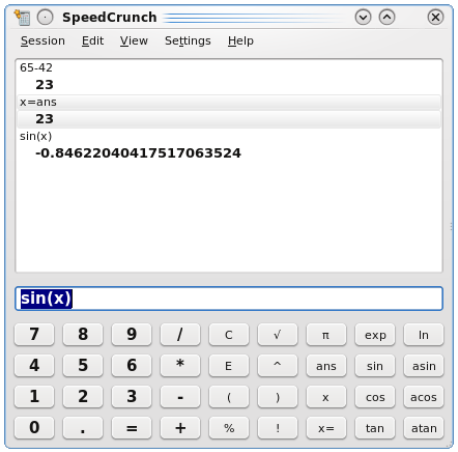
```
QObject *o = new QPustButton;
o->metaObject()->className(); // returns ”QPushButton”
```

```
connect(button, SIGNAL(clicked()), window, SLOT(close()));
```

- All code is still plain C++

The Purpose of Qt

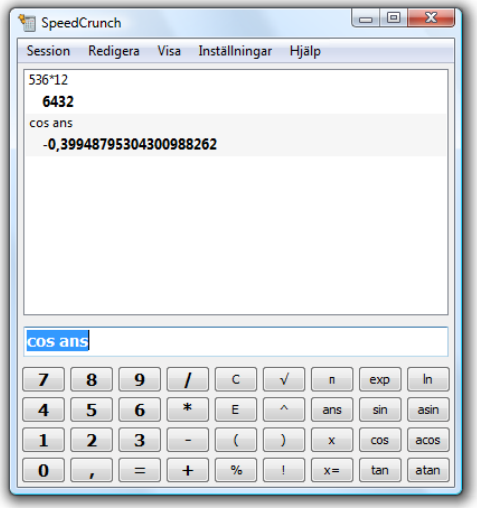
- Cross platform applications built from one source
- Builds native applications with native look and feel



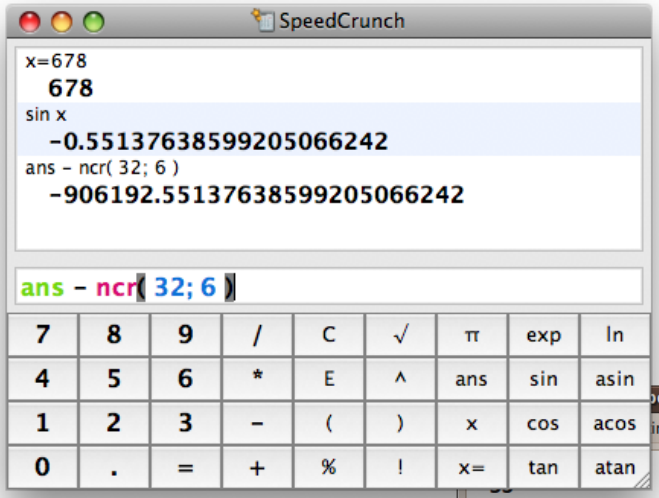
- Easy to (re)use API, high developer productivity, openess, fun to use

Desktop target platforms

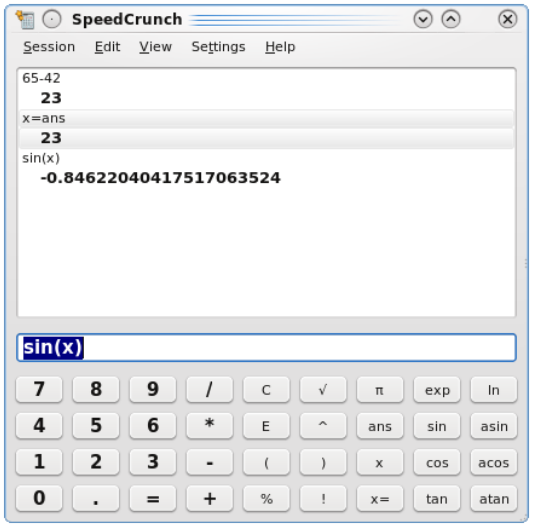
- Windows



- Mac OS X



- Linux/Unix X11



NOKIA

Embedded target platforms

- Windows CE

- Symbian

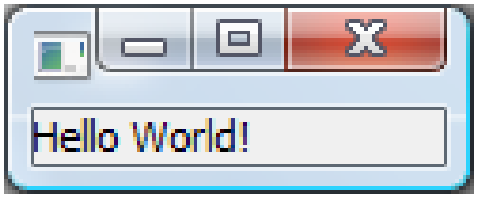
- Maemo

beagleboard.org

- Embedded Linux
  - Direct framebuffer access

NOKIA

Hello World



Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

Hello World

```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

Hello World

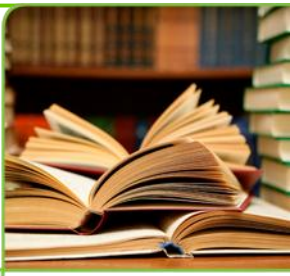
```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```

Hello World

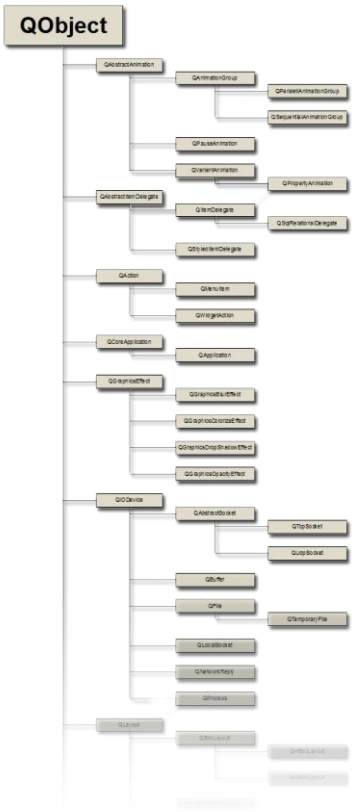
```
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```



The QObject

- `QObject` is the base class of almost all Qt classes and all widgets



- It contains many of the mechanisms that make up Qt
  - events
  - signals and slots
  - properties
  - memory management

The QObject

- `QObject` is the base class to most Qt classes. Examples of exceptions are:
  - Classes that need to be lightweight such as graphical primitives
  - Data containers (`QString`, `QList`, `QChar`, etc)
  - Classes that needs to be copyable, as `QObject`s cannot be copied

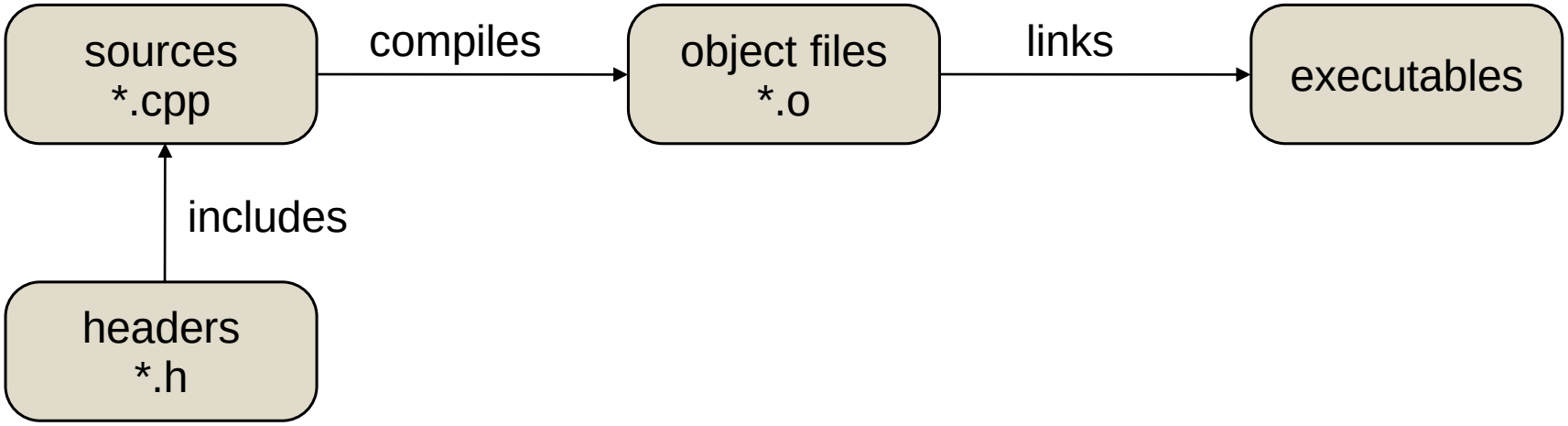
Meta data

- Qt implements introspection in C++
- Every `QObject` has a *meta object*
- The meta object knows about
  - class name (`QObject::className`)
  - inheritance (`QObject::inherits`)
  - properties
  - signals and slots
  - general information (`QObject::classInfo`)

Meta data

- The meta data is gathered at compile time by the meta object compiler, *moc*.

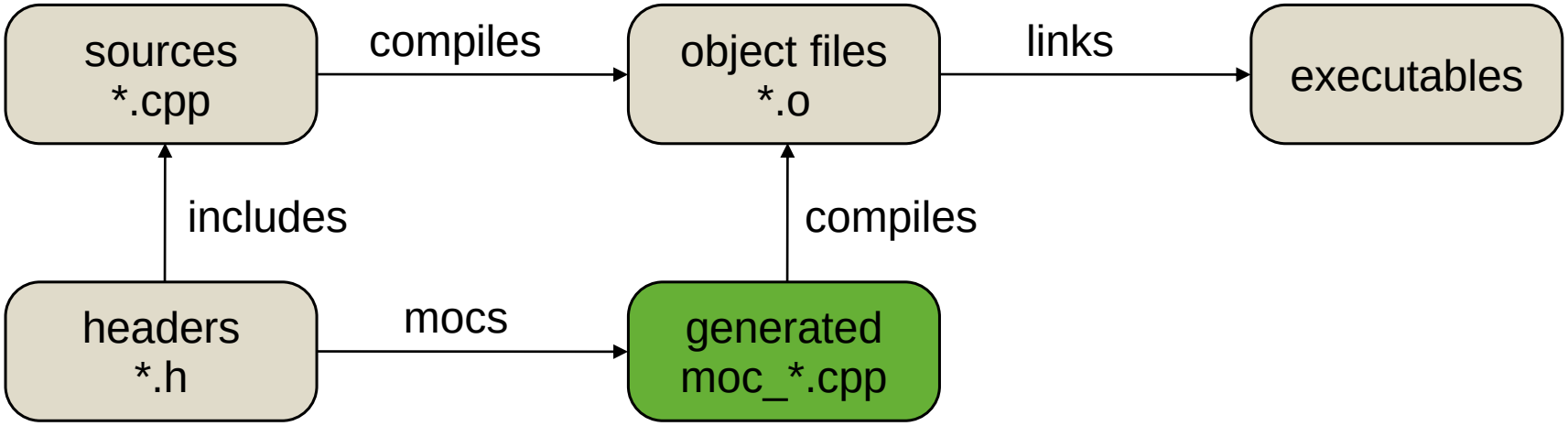
Ordinary C++ Build Process



Meta data

- The meta data is gathered at compile time by the meta object compiler, *moc*.

Qt C++ Build Process



- The moc harvests data from your headers.

Meta data

- What does moc look for?

Make sure that you inherit QObject first (could be indirect)

The Q_OBJECT macro, usually first

General info about the class

Qt keywords

```
class MyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO("author", "John Doe")

public:
    MyClass(const Foo &foo, QObject *parent=0);

    Foo foo() const;

public slots:
    void setFoo( const Foo &foo );

signals:
    void fooChanged( Foo );

private:
    Foo m_foo;
};
```

Signals and Slots

- Dynamically and loosely tie together events and state changes with reactions
- What makes Qt tick

Signals and Slots vs Callbacks

- A callback is a pointer to a function that is called when an event occurs, any function can be assigned to a callback
  - No type-safety
  - Always works as a direct call
- Signals and Slots are more dynamic
  - A more generic mechanism
  - Easier to interconnect two existing classes
  - Less knowledge shared between involved classes

What is a slot?

- A slot is defined in one of the slots sections

```
public slots:
    void aPublicSlot();
protected slots:
    void aProtectedSlot();
private slots:
    void aPrivateSlot();
```

- A slot can return values, but not through connections
- Any number of signals can be connected to a slot

```
connect(src, SIGNAL(sig()), dest, SLOT(slt()));
```

- It is implemented as an ordinary method
- It can be called as an ordinary method

What is a signal?

- A signal is defined in the signals section

```
signals:
   void aSignal();
```

- A signal always returns void
- A signal must not be implemented
  - The moc provides an implementation
- A signal can be connected to any number of slots
- Usually results in a direct call, but can be passed as events between threads, or even over sockets (using 3rd party classes)
- The slots are activated in arbitrary order
- A signal is emitted using the emit keyword

```
emit aSignal();
```

Making the connection

QObject*

```
QObject::connect( src, SIGNAL( signature ), dest, SLOT( signature ) );
```

<function name> (<arg type>...)

A signature consists of the function name and argument types. No variable names, nor values are allowed.

```
setTitle(QString text)
setValue(42)
```

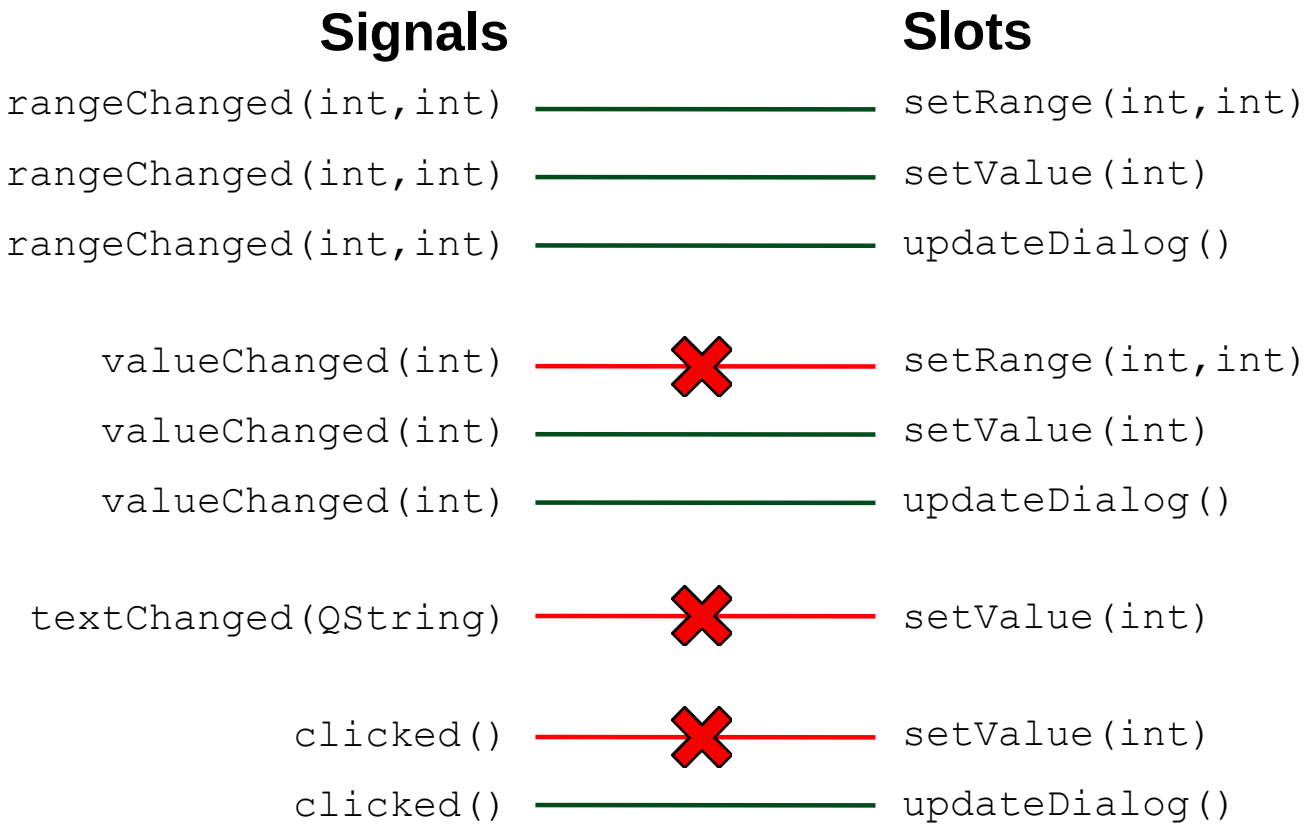
```
setItem(ItemClass)
```

Custom types reduces reusability.

```
clicked()
toggled(bool)
setText(QString)
textChanged(QString)
rangeChanged(int,int)
```

Making the connection

- Qt can ignore arguments, but not create values from nothing



Automatic Connections

- When using Designer it is convenient to have automatic connections between the interface and your code

on_ *object name* _ *signal name* (*signal parameters*)

```
on_addButton_clicked();
on_deleteButton_clicked();
on_listWidget_currentItemChanged(QListWidgetItem*,QListWidgetItem*)
```

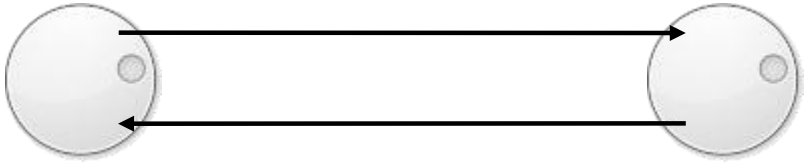
- Triggered by calling `QMetaObject::connectSlotsByName`
- Think about reuse when naming
  - Compare `on_widget_signal` to `updatePageMargins`

updatePageMargins can be connected to a number of signals or called directly.

Synchronizing Values

- Connect both ways

```
connect(dial1, SIGNAL(valueChanged(int)), dial2, SLOT(setValue(int)));
```

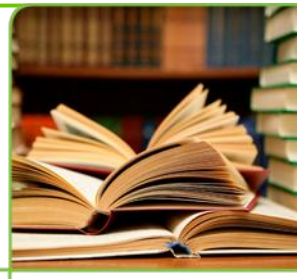


```
connect(dial2, SIGNAL(valueChanged(int)), dial1, SLOT(setValue(int)));
```

- An infinite loop must be stopped – no signal is emitted unless an actual change takes place

```
void QDial::setValue(int v)
{
    if(v==m_value)
        return;
    ...
```

This is the responsibility of all code that can emit signals – do not forget it in your own classes



Custom signals and slots

Add a notify signal here.

```
class AngleObject : public QObject
{
    Q_OBJECT
    Q_PROPERTY(qreal angle READ angle WRITE setAngle NOTIFY angleChanged)

public:
    AngleObject(qreal angle, QObject *parent = 0);
    qreal angle() const;

public slots:
    void setAngle(qreal);

signals:
    void angleChanged(qreal);

private:
    qreal m_angle;
};
```

Setters make natural slots.

Signals match the setters

Setter implementation details

```
void AngleObject::setAngle(qreal angle)
{
    if(m_angle == angle)
        return;

    m_angle = angle;
    emit angleChanged(m_angle);
}
```

Protection against infinite loops.
Do not forget this!

Update the internal state, then emit the signal.

Signals are “protected” so you can emit them from derived classes.

Lab 1 Notes

- Use QImage or QPixMap for rendering the image
- Override QMainWindow::keyPressEvent for keyboard event handling
- Keep track of old mouse position to determine paint stroke direction
- For 2D image patch transformations, use Qimage::copy and Qimage::transformed