

Qt in Education

The Qt object model and the signal slot concept













© 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.







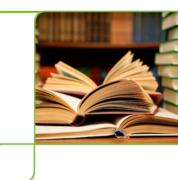
The full license text is available here: http://creativecommons.org/licenses/by-nc-sa/2.5/legalcode.

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

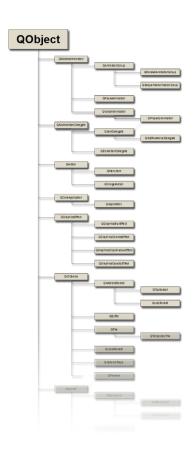




The QObject



- Q0bject 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

- Q0bject 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 Q0bjects cannot be copied



The QObject

"QObject instances are individuals!"

- They can have a name (Q0bject::objectName)
- They are placed in a hierarchy of Q0bject instances
- They can have connections to other Q0bject instances

Example: does it make sense to copy a widget at run-time?



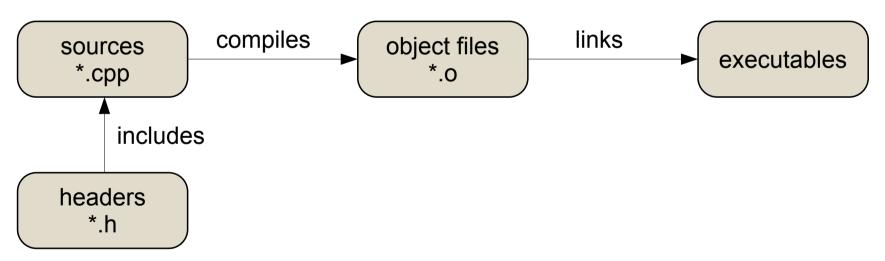


- Qt implements introspection in C++
- Every Q0bject has a meta object
- The meta object knows about
 - class name (Q0bject::className)
 - inheritance (QObject::inherits)
 - properties
 - signals and slots
 - general information (Q0bject::classInfo)



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

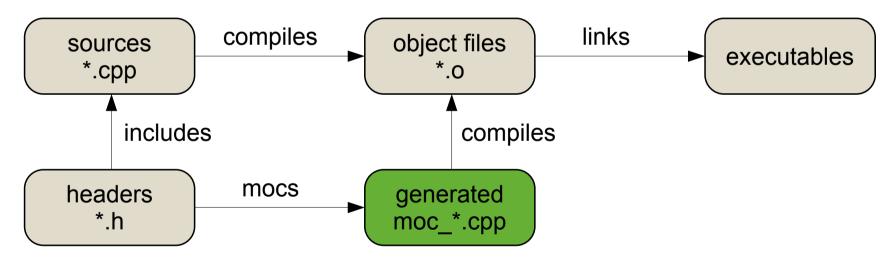
Ordinary C++ Build Process





 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.



 What does moc look for? Make sure that you inherit QObject first (could be indirect) class MyClass : public QObject The Q OBJECT **Q** OBJECT General info macro, usually first Q CLASSINFO("author", "John Doe") about the class public: MyClass(const Foo &foo, QObject *parent=0); Foo foo() const; public slots: Qt keywords void setFoo(const Foo &foo); signals: void fooChanged(Foo); private: Foo m foo;



Introspection



The classes know about themselves at run-time

```
if (object->inherits("QAbstractItemView"))
{
   QAbstractItemView *view = static_cast<QAbstractItemView*>(widget);
   view->...
Enables dynamic
casting without RTTI
```

```
enum CapitalsEnum { Oslo, Helsinki, Stockholm, Copenhagen };
int index = object->metaObject()->indexOfEnumerator("CapitalsEnum");
object->metaObject()->enumerator(index)->key(object->capital());
```

The meta object knows about the details

 Great for implementing scripting and dynamic language bindings Example:It is possible to convert enumeration values to strings for easier reading and storing







Q0bject have properties with getter and setter

methods

```
Setter, returns void, takes value as only argument
```

```
class QLabel : public QFrame
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)
public:
    QString text() const;
public slots:
    Void setText(const QString &);
};
Getter, const, returns value,
takes no arguments
};
```

- Naming policy: color, setColor
- For booleans: isEnabled, setEnabled





- Why setter methods?
 - Possible to validate settings

```
void setMin( int newMin )
{
   if( newMin > m_max )
   {
      qWarning("Ignoring setMin(%d) as min > max.", newMin);
      return;
   }
   ...
```

Possible to react to changes

```
void setMin( int newMin )
{
    ...
    m_min = newMin;
    updateMinimum();
}
```





- Why getter method?
 - Indirect properties

```
QSize size() const
{
    return m_size;
}
int width() const
{
    return m_size.width();
}
```









Using properties

Direct access

```
QString text = label->text();
label->setText("Hello World!");
```

Through the meta info and property system

```
QString text = object->property("text").toString();
object->setProperty("text", "Hello World");
```

Discover properties at run-time

```
int QMetaObject::propertyCount();
QMetaProperty QMetaObject::property(i);
QMetaProperty::name/isConstant/isDesignable/read/write/...
```





Dynamic properties

Lets you add properties to objects at run-time

```
bool ret = object->setProperty(name,
value)
```

true if the property has been defined using Q_PROPERTY

false if it is dynamically added

QObject::dynamicPropertyNames() const

returns a list of the dynamic properties

Can be used to "tag" objects, etc





Creating custom properties



Macro describing the property

```
class AngleObject : public QObject
    Q OBJECT
    Q PROPERTY(qreal angle READ angle WRITE setAngle)
public:
    AngleObject(qreal angle, QObject *parent = 0);
    greal angle() const;
                                                      Initial value
    void setAngle(qreal);
                                       Getter
                       Setter
private:
    qreal m_angle;
};
             Private state
```



Creating custom properties





Ordinary enum

declaration.

Custom properties - enumerations

Macro informing Qt that AngleMode is an enum type.

```
class AngleObject : public QObject
{
    Q_OBJECT
    Q_ENUMS(AngleMode)
    Q_PROPERTY(AngleMode angleMode READ ...)

public:
    enum AngleMode {Radians, Degrees};
    enum as type.
    ...
};
```

NOKIA



Memory Management

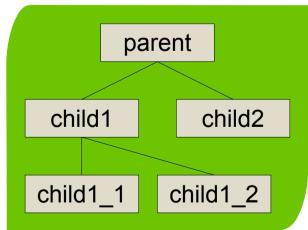


- Q0bject can have parent and children
- When a parent object is deleted, it deletes its children

```
Q0bject *parent = new Q0bject();
Q0bject *child1 = new Q0bject(parent);
Q0bject *child2 = new Q0bject(parent);
Q0bject *child1_1 = new Q0bject(child1);
Q0bject *child1_2 = new Q0bject(child1);
```

delete parent;

parent deletes child1 and child2 child1 deletes child1 1 and child1 2







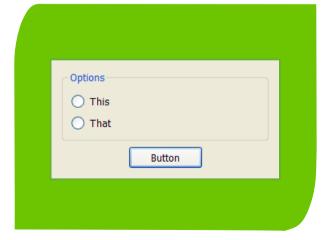
Memory Management

 This is used when implementing visual hierarchies.

```
QDialog *parent = new QDialog();
QGroupBox *box = new QGroupBox(parent);
QPushButton *button = new QPushButton(parent);
QRadioButton *option1 = new QRadioButton(box);
QRadioButton *option2 = new QRadioButton(box);
```

delete parent;

parent deletes box and button box deletes option1 and option2







Usage Patterns

Use the this-pointer as top level parent

```
Dialog::Dialog(QWidget *parent) : QDialog(parent)
{
    QGroupBox *box = QGroupBox(this);
    QPushButton *button = QPushButton(this);
    QRadioButton *option1 = QRadioButton(box);
    QRadioButton *option2 = QRadioButton(box);
    ...
```

Allocate parent on the stack

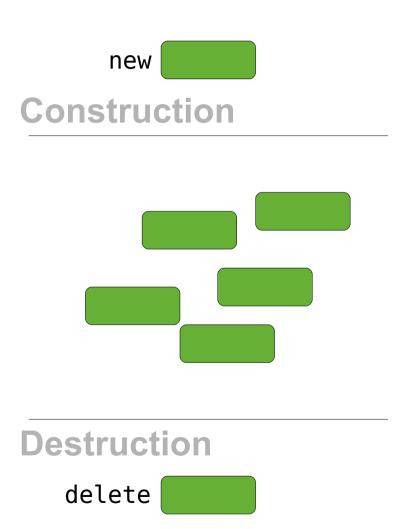
```
void Widget::showDialog()
{
    Dialog dialog;

    if (dialog.exec() == QDialog::Accepted)
    {
        ...
        dialog is deleted when
        the scope ends
}
```



Heap

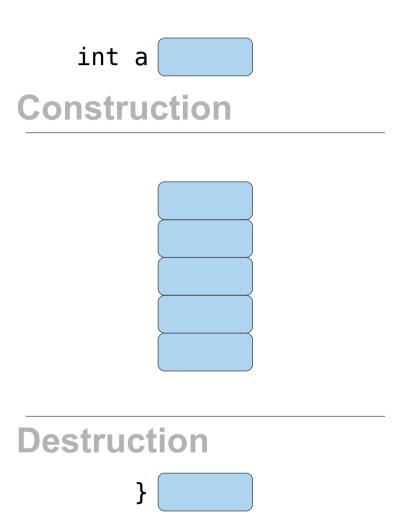
- When using new and delete, memory is allocated on the heap.
- Heap memory must be explicitly freed using delete to avoid memory leaks.
- Objects allocated on the heap can live for as long as they are needed.





Stack

- Local variables are allocated on the stack.
- Stack variables are automatically destructed when they go out of scope.
- Objects allocated on the stack are always destructed when they go out of scope.





Stack and Heap

 To get automatic memory management, only the parent needs to be allocated on the stack.

```
MyMainWindow QApplication
```

```
int main(int argc, char **argv)
{
     QApplication a(argc, argv);
     MyMainWindow w;
     w.show();
     return a.exec();
}
```

```
MyMainWindow::MyMainWindow(...
{
    new QLabel(this);
    new ...
}
```



Changing Ownership

Q0bjects can be moved between parents

```
obj->setParent(newParent);
```

The parents know when children are deleted

```
delete listWidget->item(0); // Removes the first item (unsafe)
```

 Methods that return pointers and "take" releases data from its owner and leaves it in the takers care

```
QLayoutItem *QLayout::takeAt(int);
QListWidgetItem *QListWidget::takeItem(int);

// Safe alternative
QListWidgetItem *item = listWidget->takeItem(0);
if (item) { delete item; }
```

List items are not children per se, but owned.

The example demonstrates the nomenclature.



Constructor Etiquette



 Almost all Q0bjects take a parent object with a default value of 0 (null)

```
Q0bject(Q0bject *parent=0);
```

- The parent of QWidgets are other QWidgets
- Classes have a tendency to provide many constructors for convenience (including one taking only parent)

```
QPushButton(QWidget *parent=0);
QPushButton(const QString &text, QWidget *parent=0);
QPushButton(const QIcon &icon, const QString &text, QWidget *parent=0);
```

The parent is usually the first argument with a default value

```
QLabel(const QString &text, QWidget *parent=0, Qt::WindowFlags
f=0);
```



Constructor Etiquette

- When creating your own Q0bjects, consider
 - Always allowing parent be 0 (null)
 - Having one constructor only accepting parent
 - parent is the first argument with a default value
 - Provide several constructors to avoid having to pass 0 (null) and invalid (e.g. QString()) values as arguments





Break





Signals and Slots

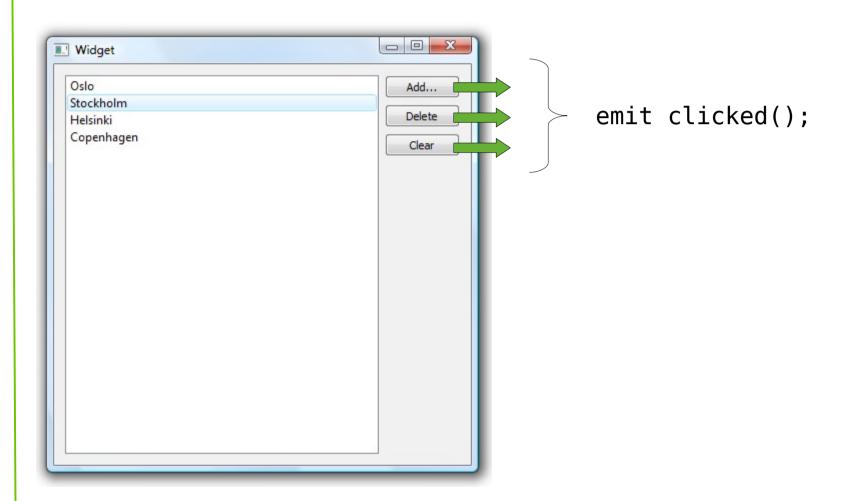


 Dynamically and loosely tie together events and state changes with reactions

What makes Qt tick



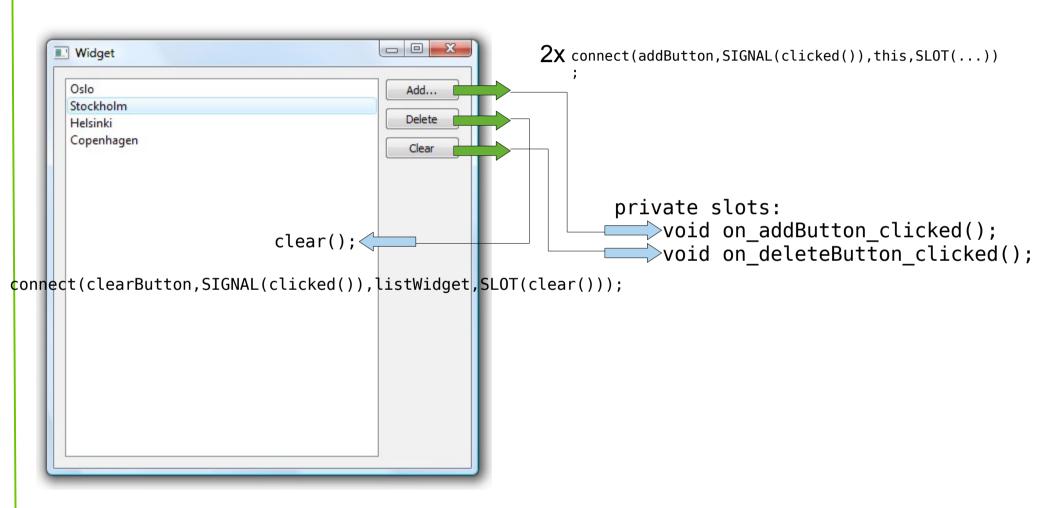
Signals and Slots in Action







Signals and Slots in Action







Signals and Slots in Action

```
{
                  emit clicked();
Add...
                                                OString newText =
                                                    QInputDialog::getText(this,
                                                                             "Enter text", "Text:");
                                                if( !newText.isEmpty() )
                                                    ui->listWidget->addItem(newText);
                                            }
              {
                  emit clicked(); [
                                                foreach (QListWidgetItem *item,
                                                          ui->listWidget->selectedItems())
                                                    delete item;
              {
                                                                  Oslo
                                                                  Stockholm
                                            clear();
                  emit clicked();
Clear
                                                                  Helsinki
                                                                  Copenhagen
```



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::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

Signals		Slots
<pre>rangeChanged(int,int)</pre>		<pre>setRange(int,int)</pre>
<pre>rangeChanged(int,int)</pre>		setValue(int)
<pre>rangeChanged(int,int)</pre>		updateDialog()
<pre>valueChanged(int) valueChanged(int) valueChanged(int)</pre>		
textChanged(QString)	*	setValue(int)
<pre>clicked() clicked()</pre>		<pre>setValue(int) updateDialog()</pre>





Automatic Connections

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

- 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
    0 OBJECT
    Q_PROPERTY(qreal angle READ angle WRITE setAngle NOTIFY angleChanged)
public:
    AngleObject(greal angle, QObject *parent = 0);
    greal angle() const;
                                 Setters make
public slots:
    void setAngle(greal);
                                 natural slots.
signals:
                                           Signals match
    void angleChanged(greal);
                                             the setters
private:
    qreal m angle;
};
```



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!

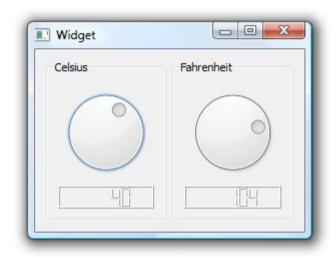
Update the internal state, then emit the signal.
```

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









- Uses the TempConverter class to convert between Celsius and Fahrenheit
- Emits signals when temperature changes





- The dialog window contains the following objects
 - A TempConverter instance
 - Two QGroupBox widgets, each containing
 - A QDial widget
 - A QLCDNumber widget





```
class TempConverter: public OObject
                                                    QObject as parent
    Q OBJECT
                      Q OBJECT macro first
                                                        parent pointer
public:
    TempConverter(int tempCelsius, Q0bject *parent = 0);
    int tempCelsius() const;
    int tempFahrenheit() const;
                                                 Read and write methods
public slots:
    void setTempCelsius(int);
    void setTempFahrenheit(int);
signals:
                                                 Emitted on changes
    void tempCelsiusChanged(int);
                                                 of the temperature
    void tempFahrenheitChanged(int);
private:
    int m tempCelsius;
                                                 Internal representation
};
                                                   in integer Celsius.
```



• The setTempCelsius slot:

```
void TempConverter::setTempCelsius(int tempCelsius)
{
   if(m_tempCelsius == tempCelsius)
        return;

m_tempCelsius = tempCelsius;
        Update object state

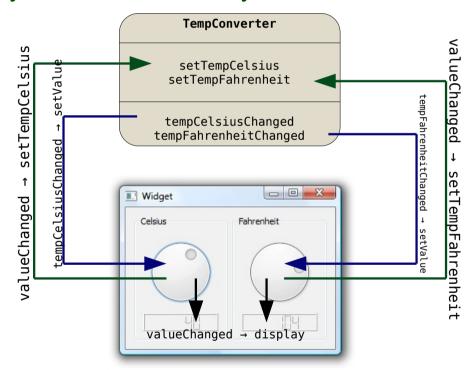
emit tempCelsiusChanged(m_tempCelsius);
   emit tempFahrenheitChanged(tempFahrenheit());
}
Emit signal(s)
reflecting changes
```

The setTempFahrenheit slot:

```
void TempConverter::setTempFahrenheit(int tempFahrenheit)
{
    int tempCelsius = (5.0/9.0)*(tempFahrenheit-32);
    setTempCelsius(tempCelsius);
}
Convert and pass on as Celsius is the internal representation
```

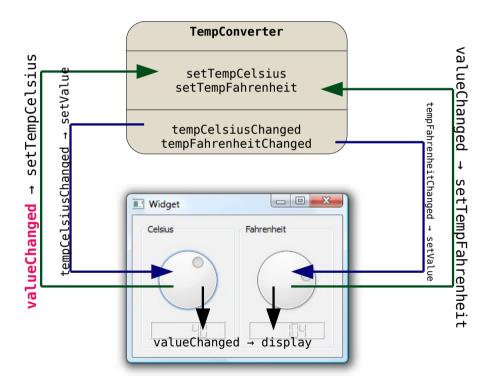


- The dials are interconnected through the TempConverter
- The LCD displays are driven directly from the dials



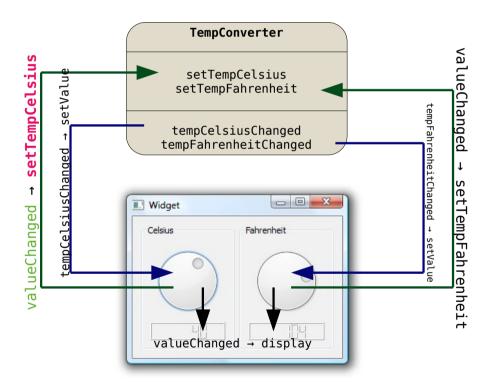
```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```





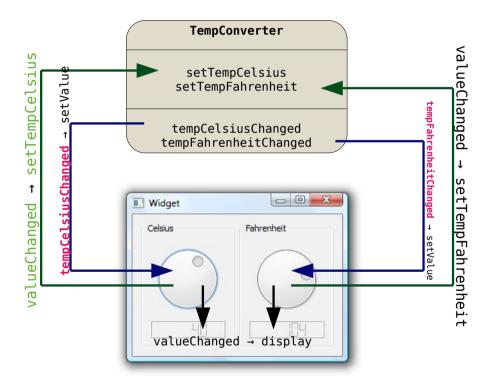
```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```





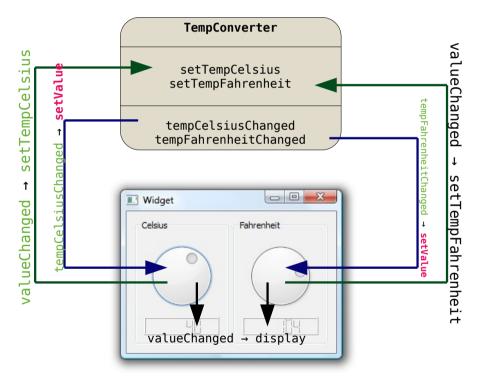
```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```





```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```

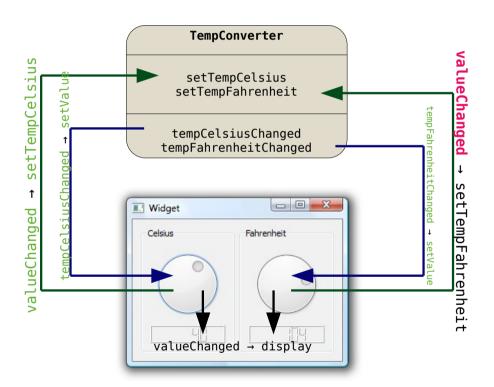




```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));

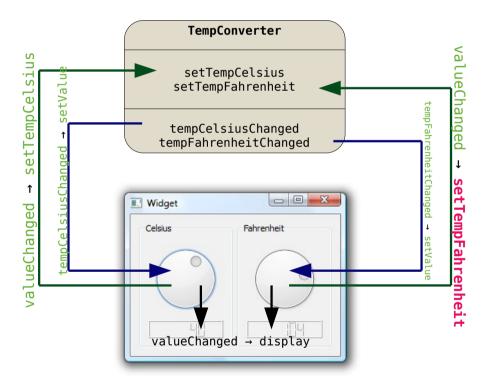
connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```





```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```

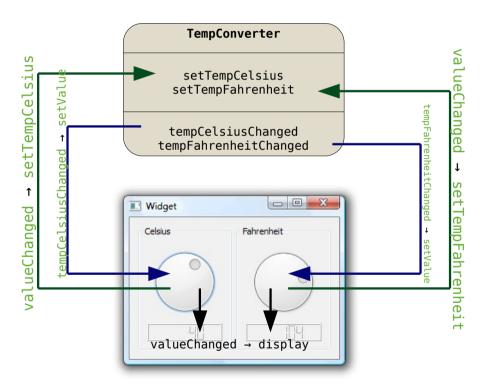




```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));

connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```





```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempFahrenheit(int)));
connect(fahrenheitDial, SIGNAL(valueChanged(int)), fahrenheitLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempFahrenheitChanged(int)), fahrenheitDial, SLOT(setValue(int)));
```



Connect with a value?



 A common scenario is that you want to pass a value in the connect statement

```
connect(key, SIGNAL(clicked()), this, SLOT(keyPressed(1)));
```

For instance, the keyboard example



This is not valid – it will not connect





Connect with a value?

Solution #1: multiple slots

```
. . .
                              public slots:
                                   void key1Pressed();
                                   void key2Pressed();
                                   void key3Pressed();
7
            9
                                   void key4Pressed();
                    connections
                                   void key5Pressed();
                                   void key6Pressed();
            3
                                   void key7Pressed();
                                   void key8Pressed();
      0
                                   void key9Pressed();
                                   void key0Pressed();
                                   . . .
```



Connect with a value?

Solution #2: sub-class emitter and add signal

```
QPushButton
QIntPushButton
```

```
signals:
   void clicked(int);
}
```

```
OIntPushButton *b:
b=new OIntPushButton(1);
connect(b, SIGNAL(clicked(int)),
    this, SLOT(kevPressed(int)));
b=new OIntPushButton(2);
connect(b, SIGNAL(clicked(int)),
    this, SLOT(keyPressed(int)));
b=new QIntPushButton(3);
connect(b, SIGNAL(clicked(int)),
    this, SLOT(keyPressed(int)));
```



Solution evaluation

- #1: multiple slots
 - Many slots containing almost the same code
 - Hard to maintain (one small change affects all slots)
 - Hard to extend (new slot each time)

- #2: sub-class emitter and add signal
 - Extra class that is specialized (hard to reuse)
 - Hard to extend (new sub-class for each special case)



The signal mapper

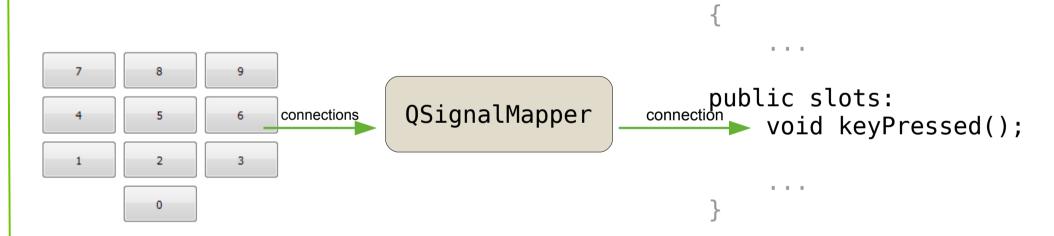
- The QSignalMapper class solves this problem
 - Maps a value to each emitter
 - Sits between reusable classes

Create a signal mapper



The signal mapper

 The signal mapper associates each button with a value. These values are mapped



 When a value is mapped, the signal mapper emits the mapped(int) signal, carrying the associated value