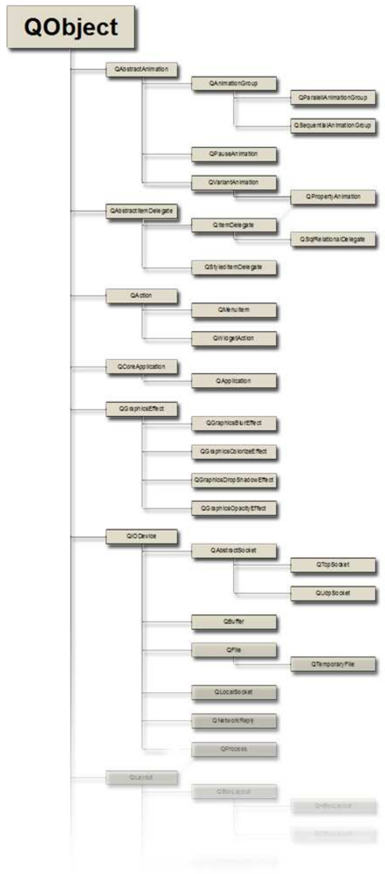
Qt in Education

The Qt object model and the signal slot concept

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

The QObject

“QObject instances are individuals!”

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

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

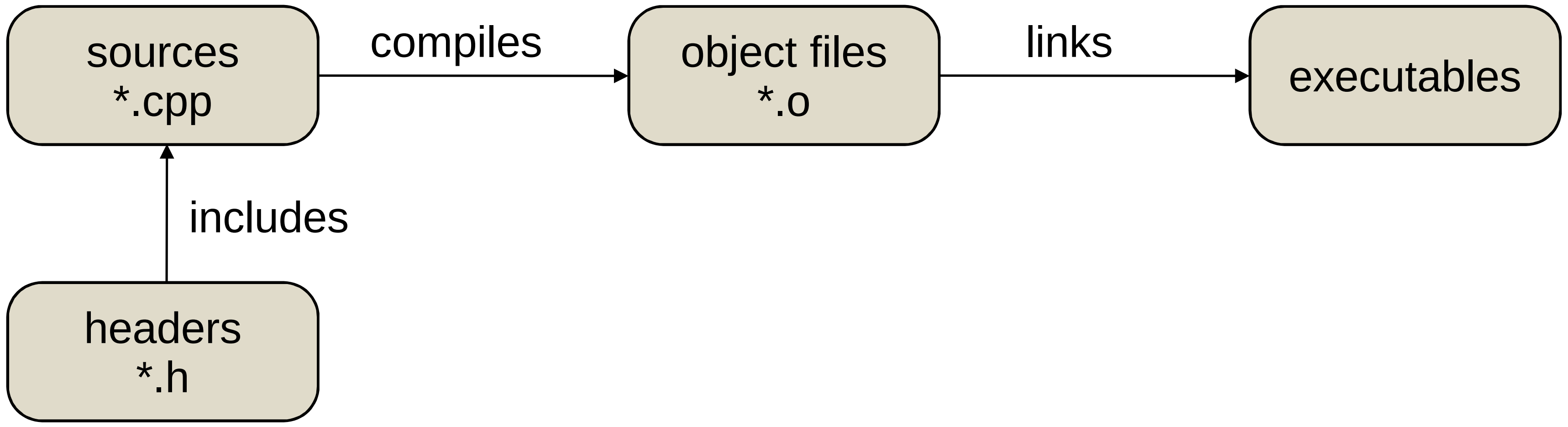
Meta data

- 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*.

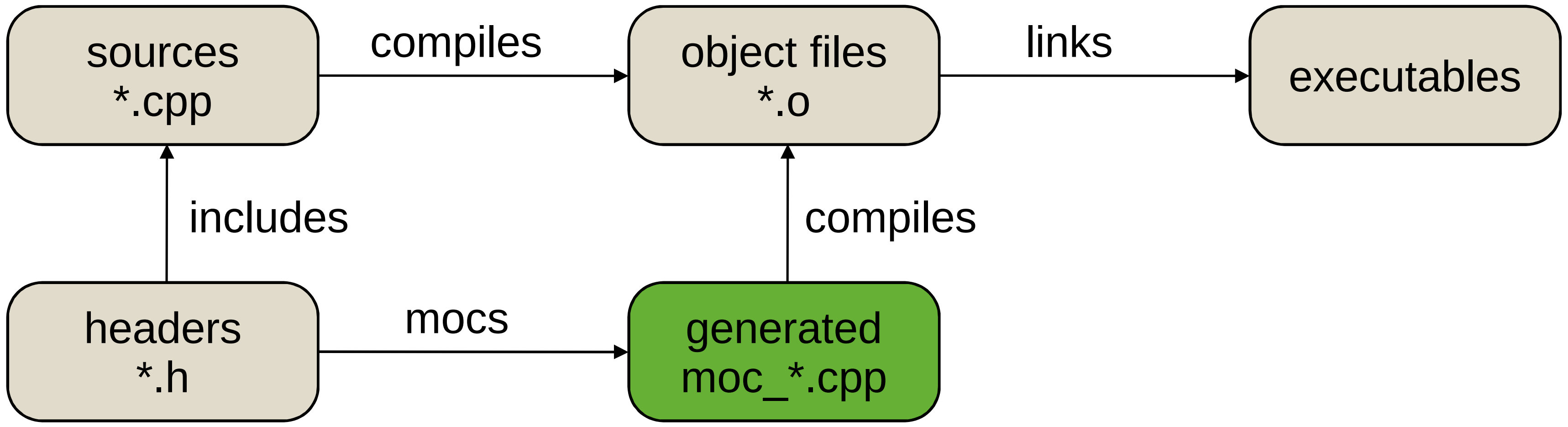
Ordinary C++ Build Process



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?

```
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;
};
```

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

The Q_OBJECT macro, usually first

General info about the class

Qt keywords

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

Example:It is possible to convert enumeration values to strings for easier reading and storing

- Great for implementing scripting and dynamic language bindings

Properties

- `QObject` have properties with getter and setter methods

```
class QLabel : public QFrame
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)
public:
    QString text() const;
public slots:
    void setText(const QString &);
};
```

Setter, returns void, takes value as only argument

Getter, const, returns value, takes no arguments

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

Properties

- 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();
}
```

Properties

- Why getter method?
  - Indirect properties

```
QSize size() const
{
    return m_size;
}

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

Properties

```
Q_PROPERTY(type name
           READ getFunction
           [WRITE setFunction]
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL])
```

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);

    qreal angle() const;

    void setAngle(qreal);

private:
    qreal m_angle;
};
```

Initial value

Getter

Setter

Private state

Creating custom properties

```
AngleObject::AngleObject(qreal angle, QObject *parent) :
        QObject(parent), m_angle(angle)
{
}

qreal AngleObject::angle() const
{
    return m_angle;
}

void AngleObject::setAngle(qreal angle)
{
    m_angle = angle;
    doSomething();
}
```

Initial value

Getter simply returns the value. Here you can calculate complex values.

Update internal state, then react to the change.

Custom properties - enumerations

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

public:
    enum AngleMode {Radians, Degrees};

    ...
};
```

Macro informing Qt that AngleMode is an enum type.

Property using enum as type.

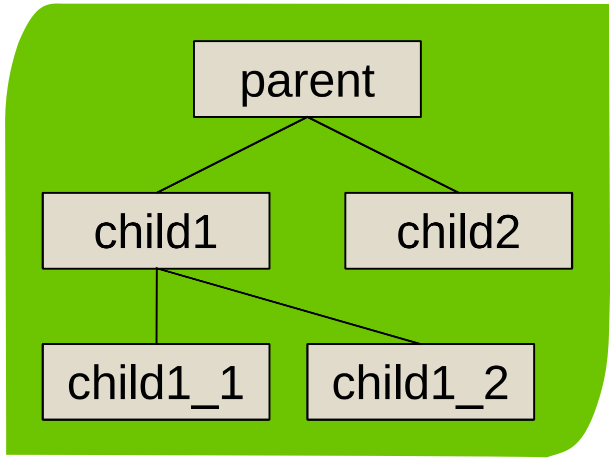
Ordinary enum declaration.

Memory Management

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

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

delete parent;
```

parent deletes child1 and child2
child1 deletes child1_1 and child1_2

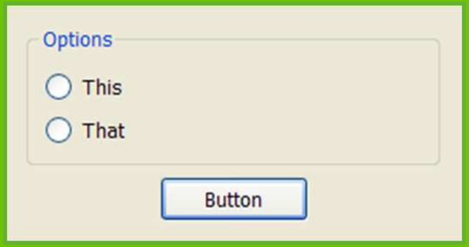


Memory Management

- 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);
    ...
```

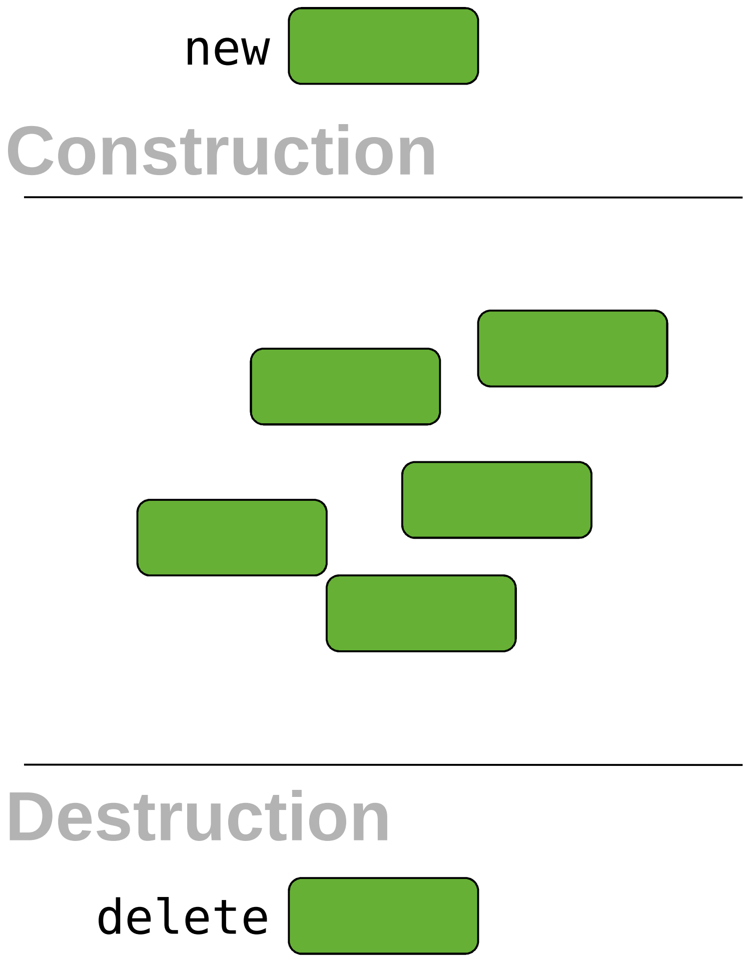
- Allocate parent on the stack

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

    if (dialog.exec() == QDialog::Accepted)
    {
        ...
    }
}
```

`dialog` is deleted when the scope ends

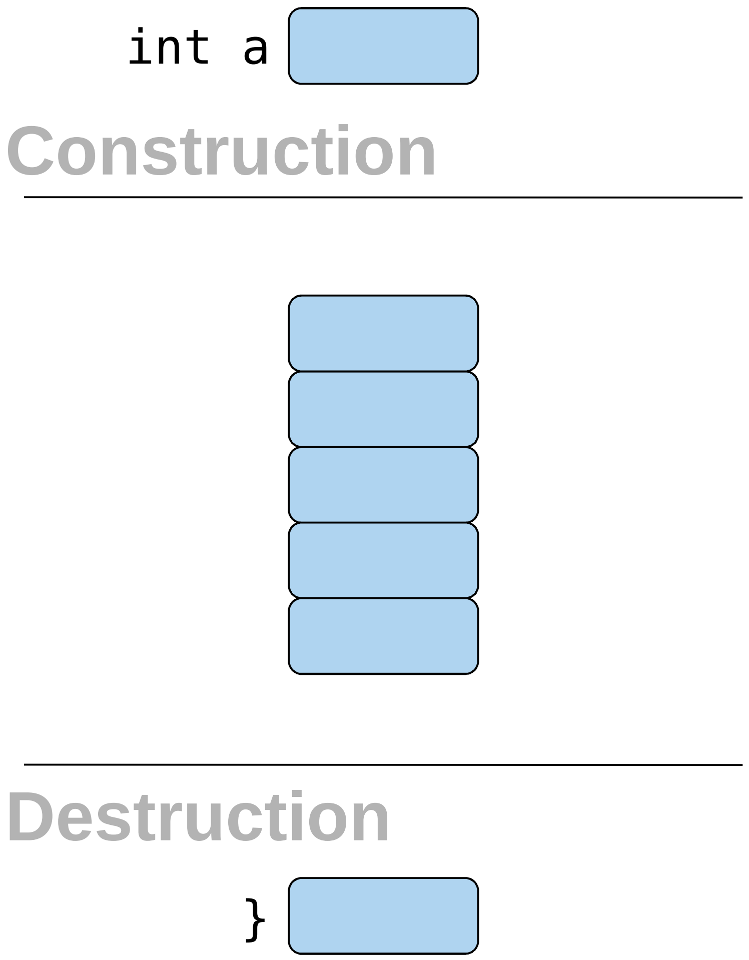
Heap

- 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.



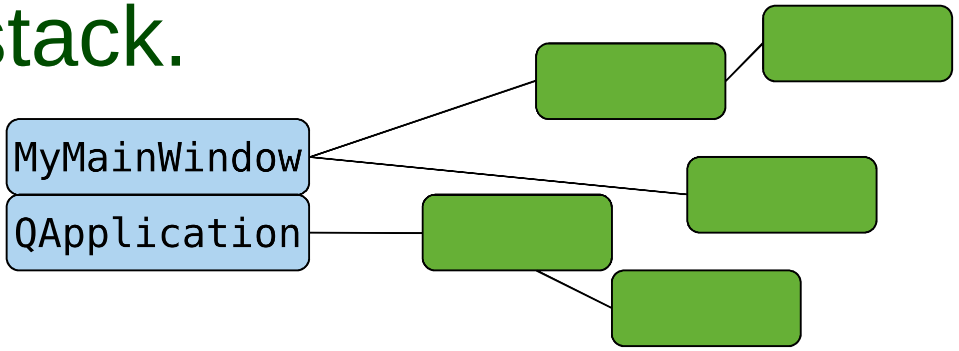
Stack

- 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.



```
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

- `QObjects` 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 `QObjects` take a parent object with a default value of 0 (null)

```
QObject(QObject *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

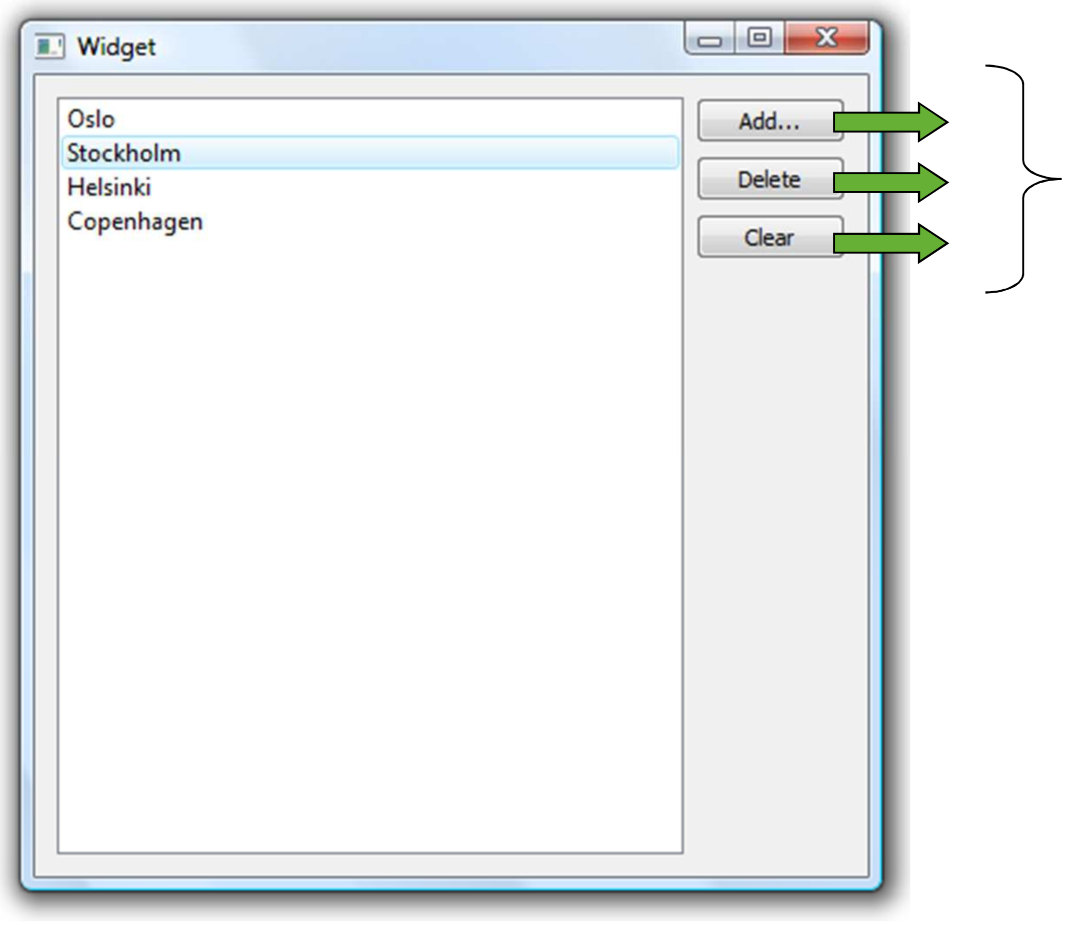
- When creating your own `QObjects`, consider
  - Always allowing `parent` be 0 (null)
  - Having one constructor only accepting `parent`
  - `parent` is the first argument with a default value
  - Provide several constructors to avoid having to pass 0 (null) and invalid (e.g. `QString()`) values as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

Break

Signals and Slots

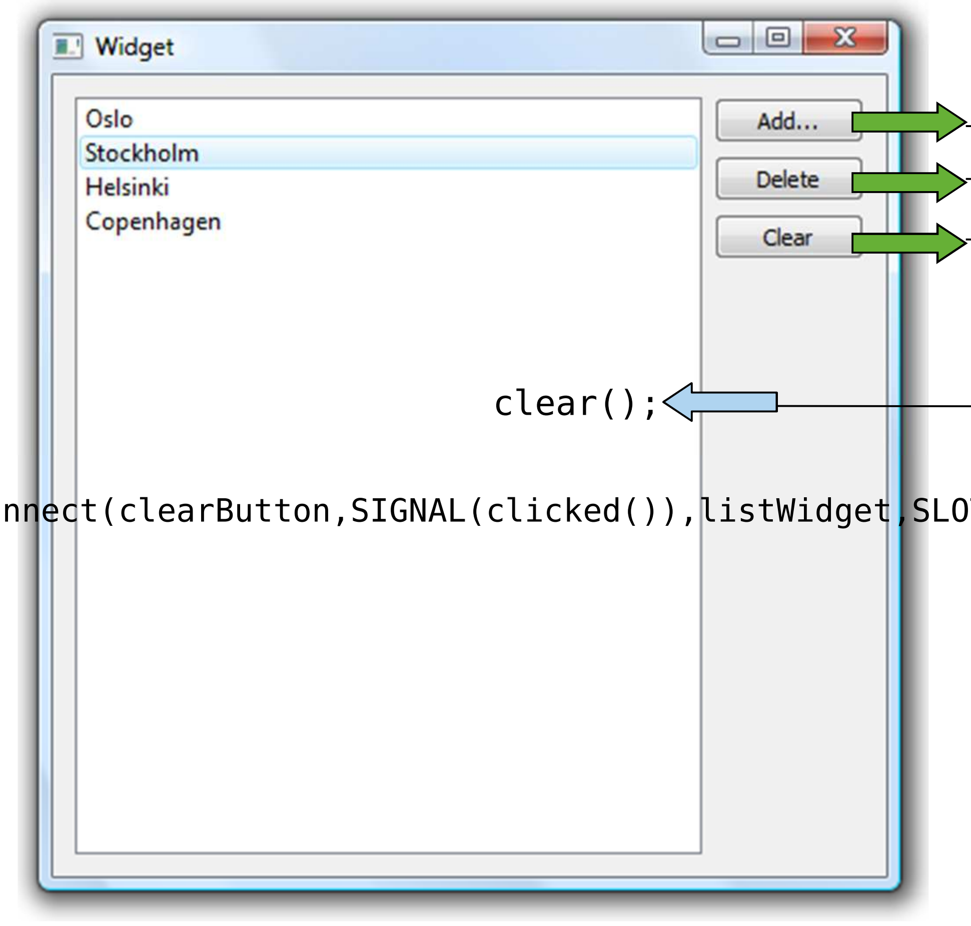
- Dynamically and loosely tie together events and state changes with reactions
- What makes Qt tick

Signals and Slots in Action



```
emit clicked();
```

Signals and Slots in Action



2x `connect(addButton,SIGNAL(clicked()),this,SLOT(...));`

```
private slots:
void on_addButton_clicked();
void on_deleteButton_clicked();
```

```
clear();
```

```
connect(clearButton,SIGNAL(clicked()),listWidget,SLOT(clear()));
```

Signals and Slots in Action

```
{
    ...
    emit clicked();
    ...
}
```

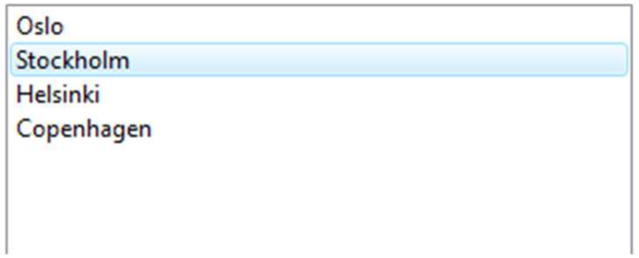
```
{
    QString newText =
        QInputDialog::getText(this,
                              "Enter text", "Text:");

    if( !newText.isEmpty() )
        ui->listWidget->addItem(newText);
}
```

```
{
    ...
    emit clicked();
    ...
}
```

```
{
    foreach (QListWidgetItem *item,
             ui->listWidget->selectedItems())
    {
        delete item;
    }
}
```

```
{
    ...
    emit clicked();
    ...
}
```

```
clear();
```


Oslo
Stockholm
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*

```
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
`rangeChanged(int,int)`	———	`setRange(int,int)`
`rangeChanged(int,int)`	———	`setValue(int)`
`rangeChanged(int,int)`	———	`updateDialog()`
`valueChanged(int)`	—✗—	`setRange(int,int)`
`valueChanged(int)`	———	`setValue(int)`
`valueChanged(int)`	———	`updateDialog()`
`textChanged(QString)`	—✗—	`setValue(int)`
`clicked()`	—✗—	`setValue(int)`
`clicked()`	———	`updateDialog()`

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*)
```

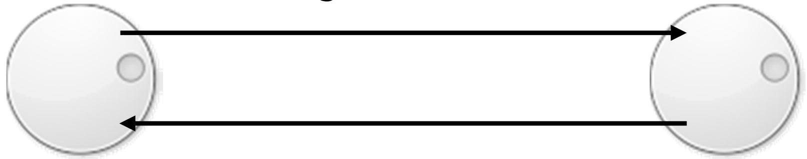
- Triggered by calling `QMetaObject::connectSlotsByName`
- Think about reuse when naming
  - Compare `on_widget_signal` to `updatePageMargins`

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)));
```



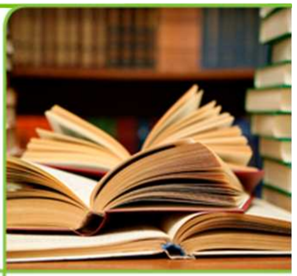
```
connect(dial2, SIGNAL(valueChanged(int)), dial1, SLOT(setValue(int)));
```

- 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

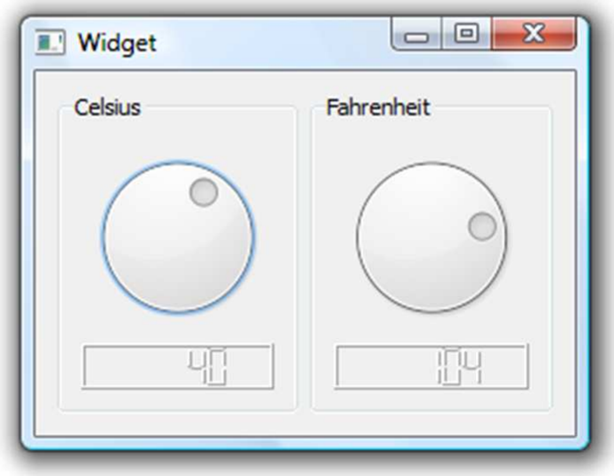
```
void AngleObject::setAngle(qreal angle)
{
    if(m_angle == angle)
        return;

    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.

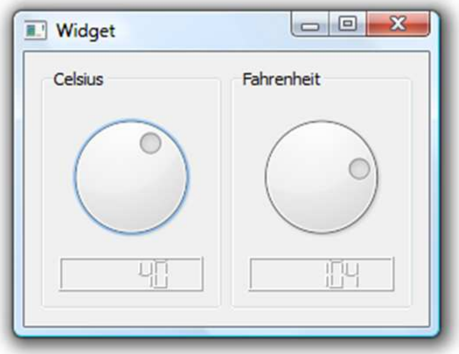
Temperature Converter



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

Temperature Converter

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



Temperature Converter

```
class TempConverter : public QObject
{
    Q_OBJECT

public:
    TempConverter(int tempCelsius, QObject *parent = 0);

    int tempCelsius() const;
    int tempFahrenheit() const;

public slots:
    void setTempCelsius(int);
    void setTempFahrenheit(int);

signals:
    void tempCelsiusChanged(int);
    void tempFahrenheitChanged(int);

private:
    int m_tempCelsius;
};
```

QObject as parent

Q_OBJECT macro first

parent pointer

Read and write methods

Emitted on changes of the temperature

Internal representation in integer Celsius.

Temperature Converter

- The `setTempCelsius` slot:

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

    m_tempCelsius = tempCelsius;

    emit tempCelsiusChanged(m_tempCelsius);
    emit tempFahrenheitChanged(tempFahrenheit());
}
```

Test for change to break recursion

Update object state

Emit signal(s) reflecting changes

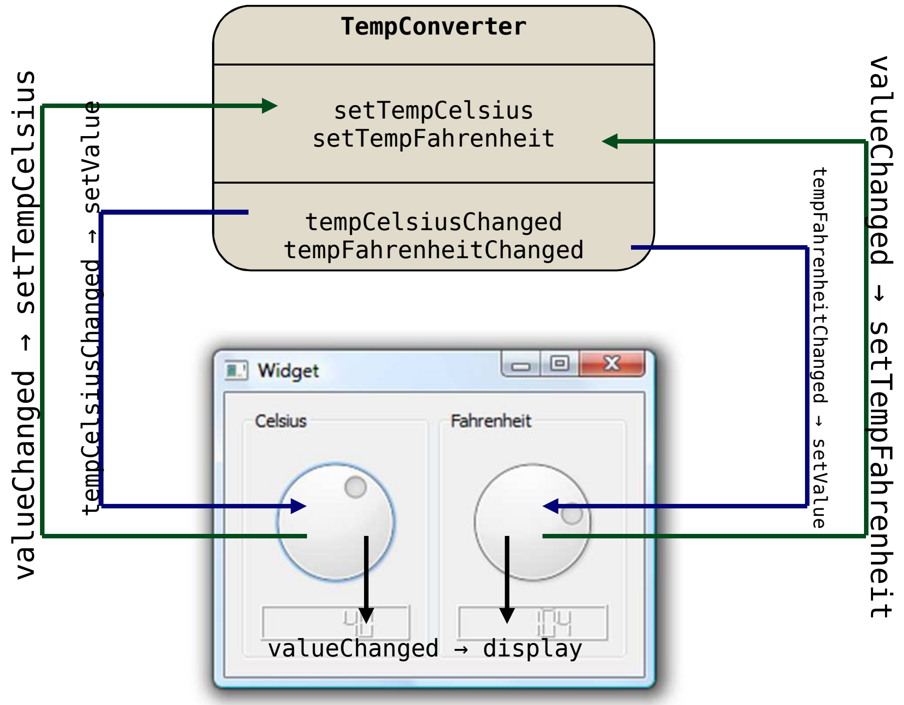
- The `setTempFahrenheit` slot:

```
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

Temperature Converter

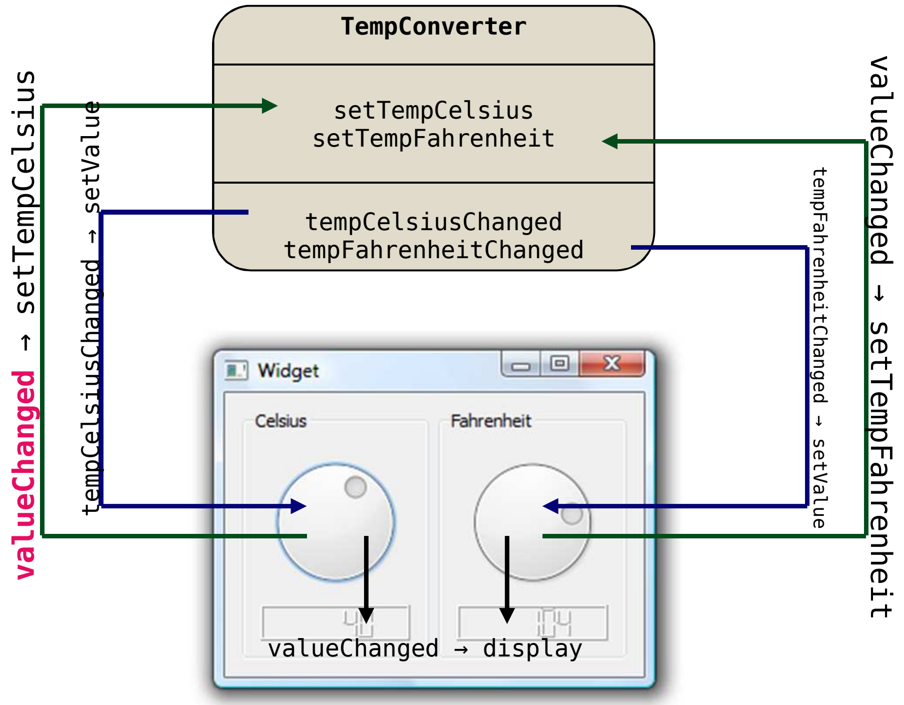
- The dials are interconnected through the `TempConverter`
- The LCD displays are driven directly from the dials



```
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)));
```

Temperature Converter

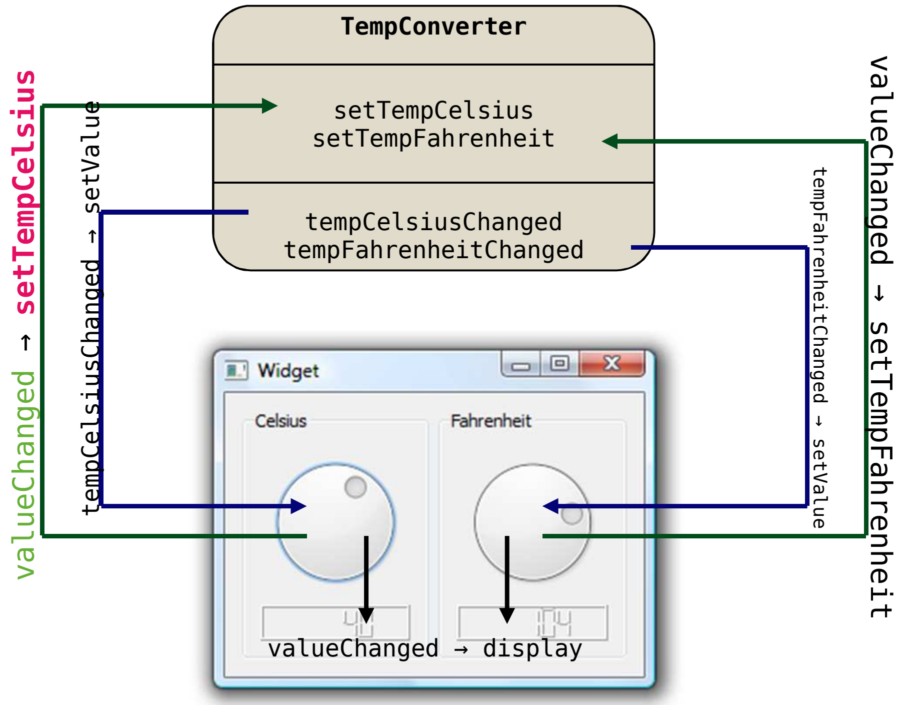
- The user moves the celsiusDial



```
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)));
```

Temperature Converter

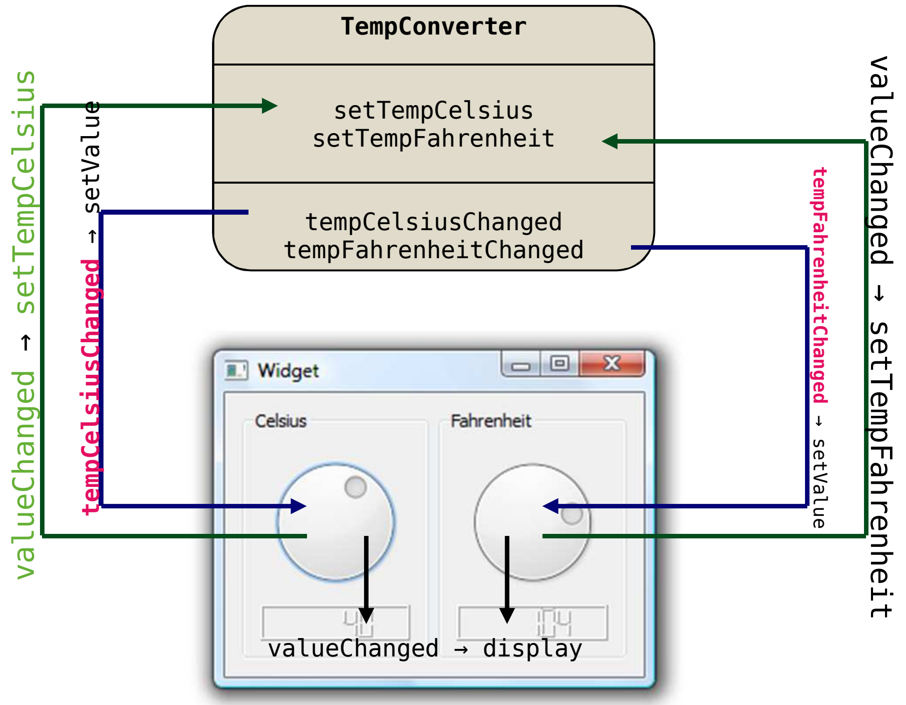
- The user moves the celsiusDial



```
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)));
```

Temperature Converter

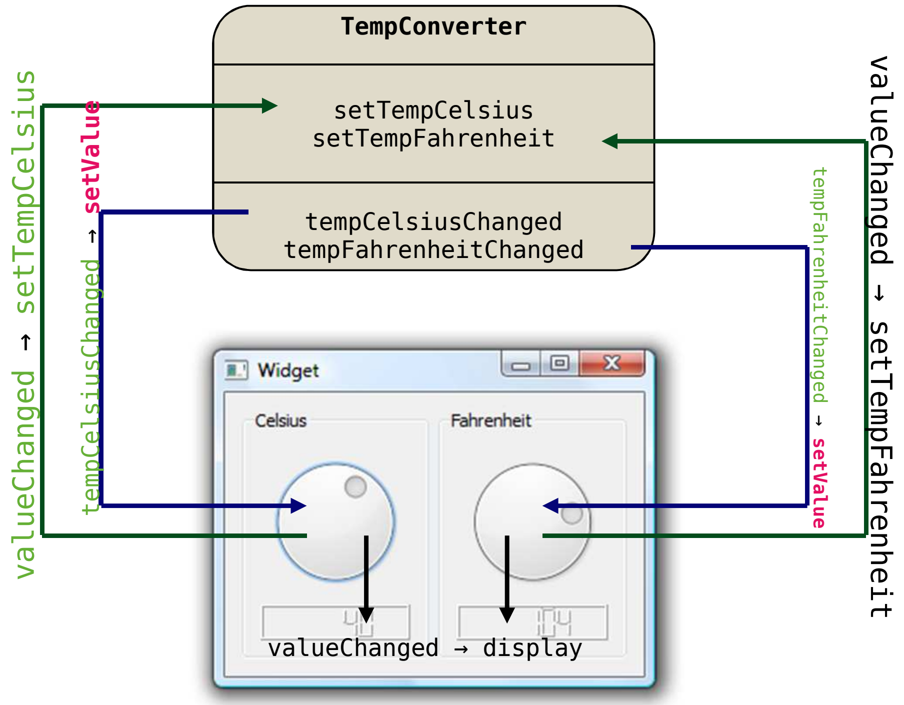
- The user moves the celsiusDial



```
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)));
```

Temperature Converter

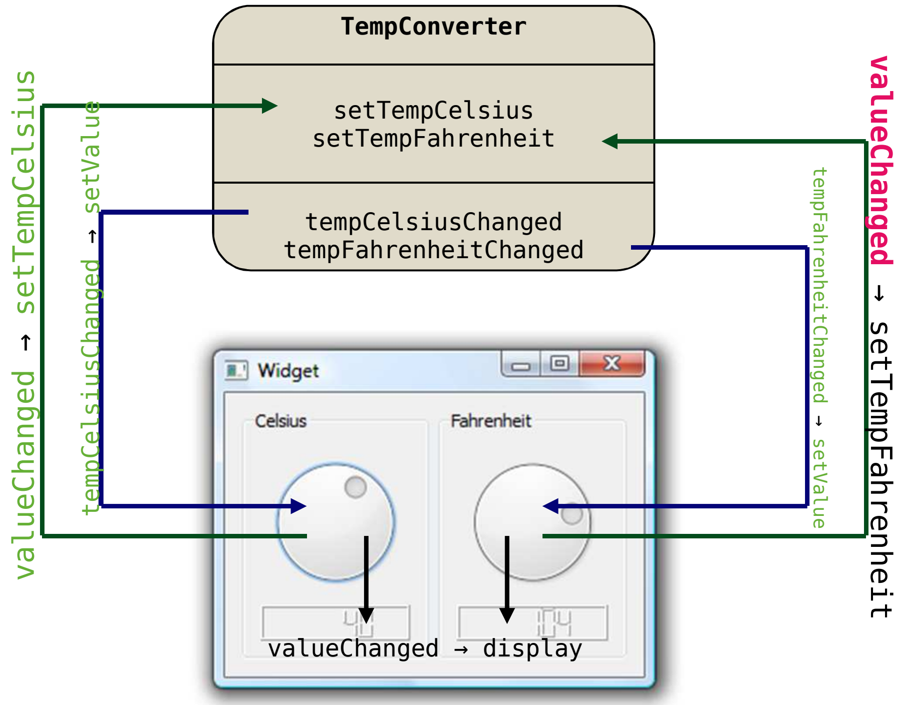
- The user moves the celsiusDial



```
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)));
```

Temperature Converter

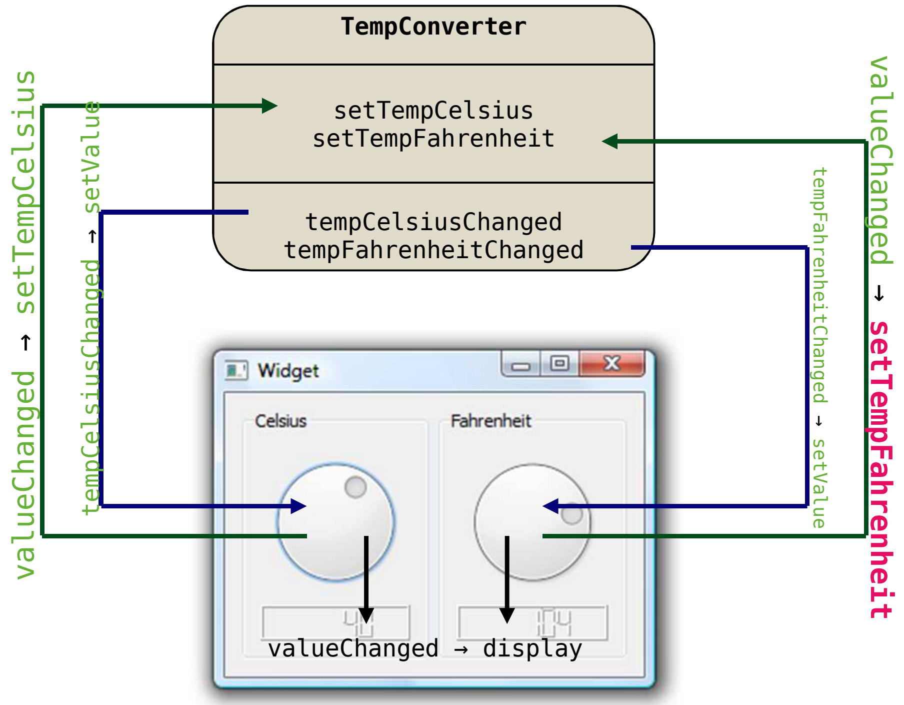
- The user moves the celsiusDial



```
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)));
```

Temperature Converter

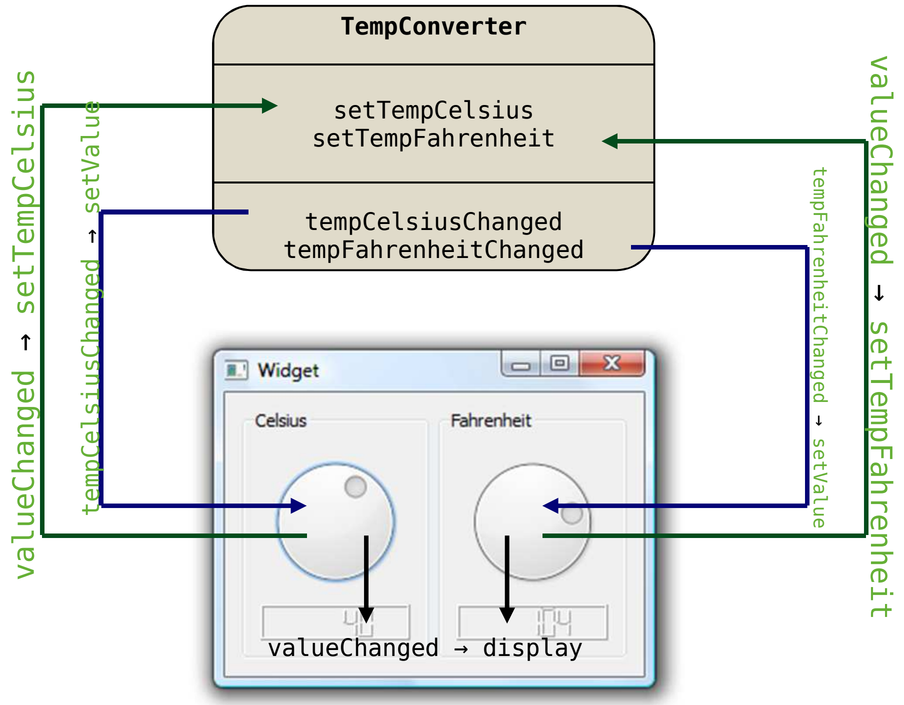
- The user moves the celsiusDial



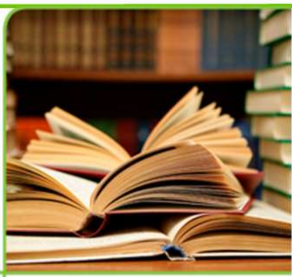
```
connect(celsiusDial, SIGNAL(valueChanged(int)), tempConverter, SLOT(setTempCelsius(int)));
connect(celsiusDial, SIGNAL(valueChanged(int)), celsiusLcd, SLOT(display(int)));
connect(tempConverter, SIGNAL(tempCelsiusChanged(int)), celsiusDial, SLOT(setValue(int)));

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)));
```

Temperature Converter

- The user moves the celsiusDial



```
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)));
```

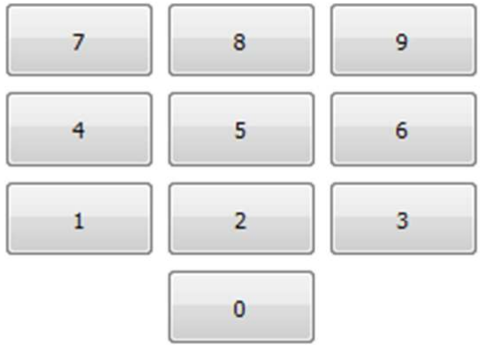


Connect with a value?

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

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

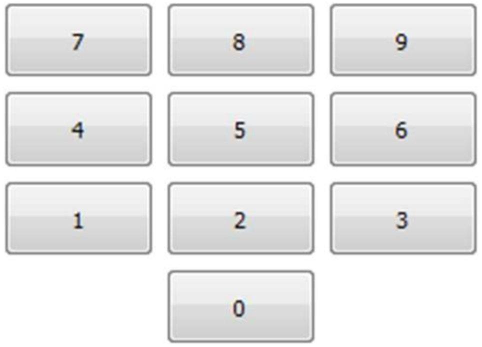
- For instance, the keyboard example



- This is not valid – it will not connect

Connect with a value?

- Solution #1: multiple slots

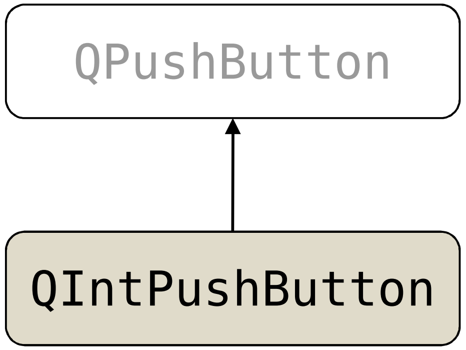


connections →

```
{
    ...

public slots:
    void key1Pressed();
    void key2Pressed();
    void key3Pressed();
    void key4Pressed();
    void key5Pressed();
    void key6Pressed();
    void key7Pressed();
    void key8Pressed();
    void key9Pressed();
    void key0Pressed();

    ...
}
```

Connect with a value?

- Solution #2: sub-class emitter and add signal



```
{
    ...

signals:
    void clicked(int);

    ...
}
```

```
{
    QIntPushButton *b;

    b=new QIntPushButton(1);
    connect(b, SIGNAL(clicked(int)),
        this, SLOT(keyPressed(int)));

    b=new QIntPushButton(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

```
{
    QSignalMapper *m = QSignalMapper(this);
    QPushButton *b;

    b=new QPushButton("1");
    connect(b, SIGNAL(clicked()),
            m, SLOT(map()));
    m->setMapping(b, 1);

    ...

    connect(m, SIGNAL(mapped(int)), this, SLOT(keyPressed(int)));
}
```

Create a signal mapper

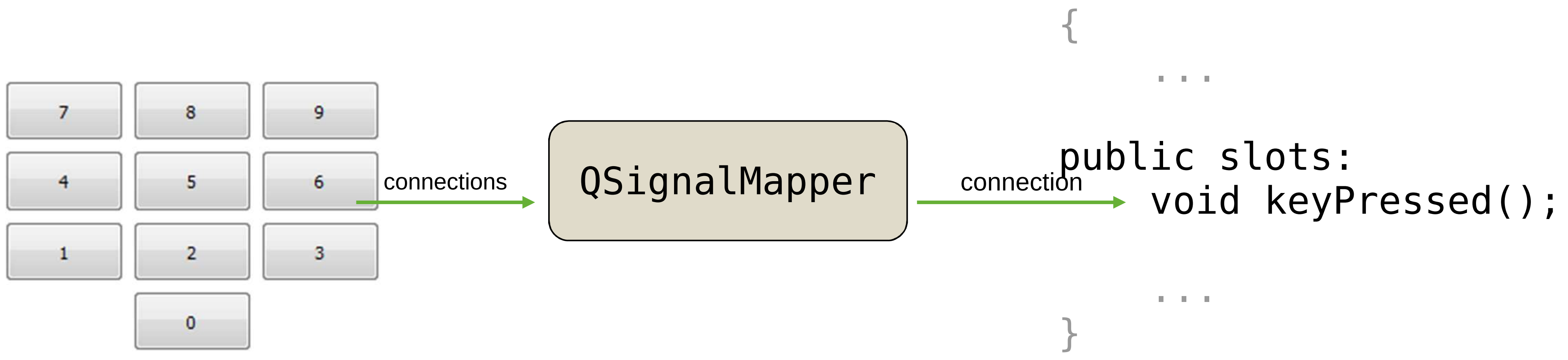
Connect the buttons to the mapper

Associate an emitter with a value

Connect the mapper to the slot

The signal mapper

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



```
{
      ...
public slots:
    void keyPressed();
      ...
}
```

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