App@phone —> QML
App@Desktop -> Qt Widget

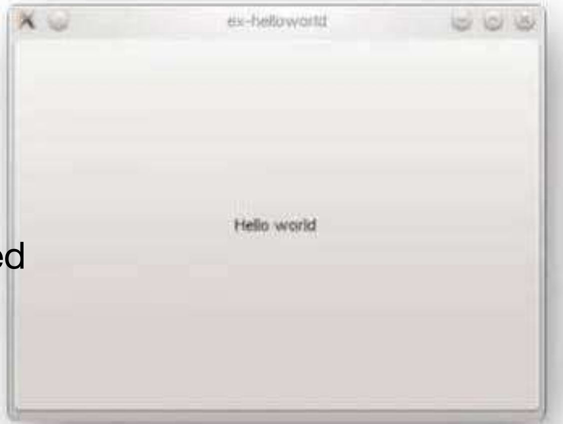
Qt Widgets Basics

- **Episode 1 - Hello World**
- Episode 2 - Compiling Hello World
- Episode 3 - The Qt Help text
- Episode 4 - Qt's Object Model
- Episode 5 - Qt's Object Model and QWidget Basics
- Episode 6 - Signals & Slots
- Episode 7 - Qt4's Signals & Slots
- Episode 8 - Signals & Slots with Lambdas
- Episode 9 - Adding your own Signals and Slots
- Episode 10 - The Backstage Tour - Part 1
- Episode 11 - The Backstage Tour - Part 2
- Episode 12 - Event Handling

"Hello World" with QtWidgets

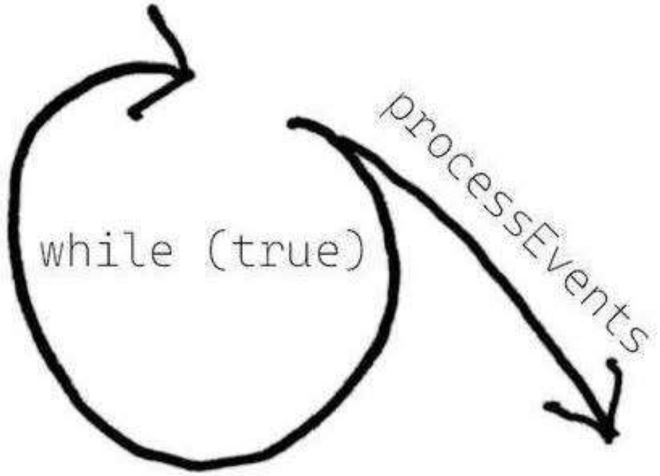
// main.cpp

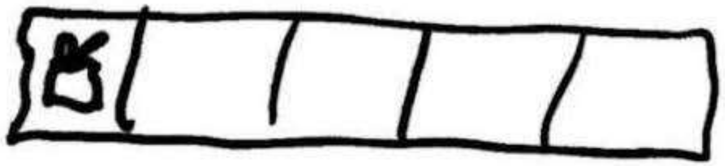
```
#include <QtWidgets>

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);     //QApplication(int argc,char *argv[]) is called
  QPushButton button("Hello world");
  button.show();
  return app.exec(); //start the event loop
}
```



Demo: fundamentals/ex-helloworld

Event loop —> Event-driven App

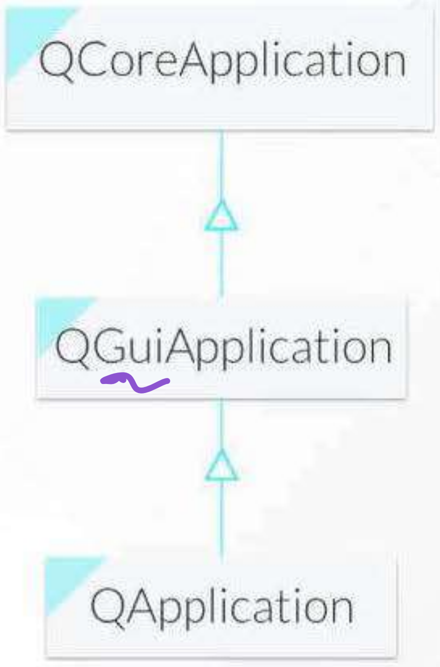




QCoreApplication, QGuiApplication & QApplication

- QCoreApplication
  - Pass command line arguments
  - Provides an event-loop to process and dispatch events
  - Internationalization - QObject::tr()
  - Access to application path and pid
- Non-Gui applications

Qt Docs: QCoreApplication



window management
Mouse cursor handling
Clipboard interaction
Keeping track of sys properties
    palatte
    fonts
    style hints
Non-Widget(Qt Quick) App

Session management
Widget specific initialization and finalization
Style management Qtsyle
Widget management

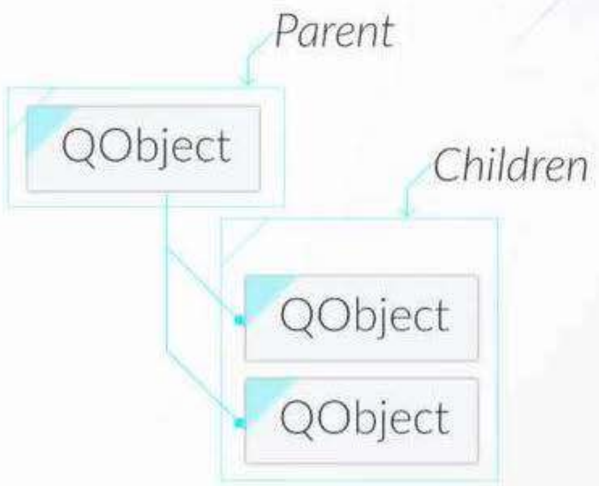
Qt's C++ Object Model - QObject

- `QObject` is the heart of Qt's object model.
- Adds features to C++, like
  - Signals and slots
  - Properties
  - Event handling
  - Memory management
  - ...
- Some features are standard C++.
  - Some use Qt's meta-object system.
- `QObject` has no visual representation.

Object Tree

- `QObjects` organize themselves in object trees.
  - Based on parent-child relationship
- `QObject(QObject *parent = nullptr)`
  - Parent adds object to list of children.
  - Parent owns children.
- Construction/Destruction
  - Tree can be constructed in any order.
  - Tree can be destroyed in any order.
    - If object has a parent, object is first removed from parent.
    - If object has children, each child is deleted first.
    - No object is deleted twice.

Note: Parent-child relationship is NOT inheritance.



Creating Objects

- **On Heap** - QObject with parent:

New will allocate the memory and return the first address of the memory

```
QLabel *label = new QLabel("Some Text", parent);
```

  - It is forbidden to copy QObject instances.

Create obj on stack will be deleted when out of scope

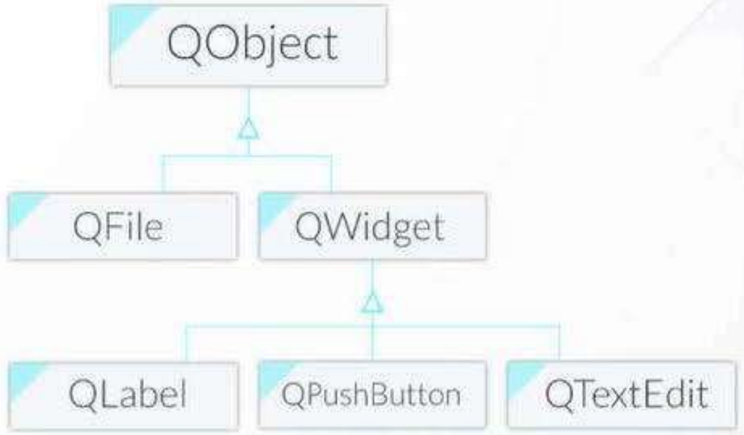
- **On Stack** - QObject without parent:
  - QFile, usually local to a function → This will be out of scope after reading it
  - QGuiApplication (local to main())
  - Top level QWidgets: QMainWindow
- **On Stack** - "value" types

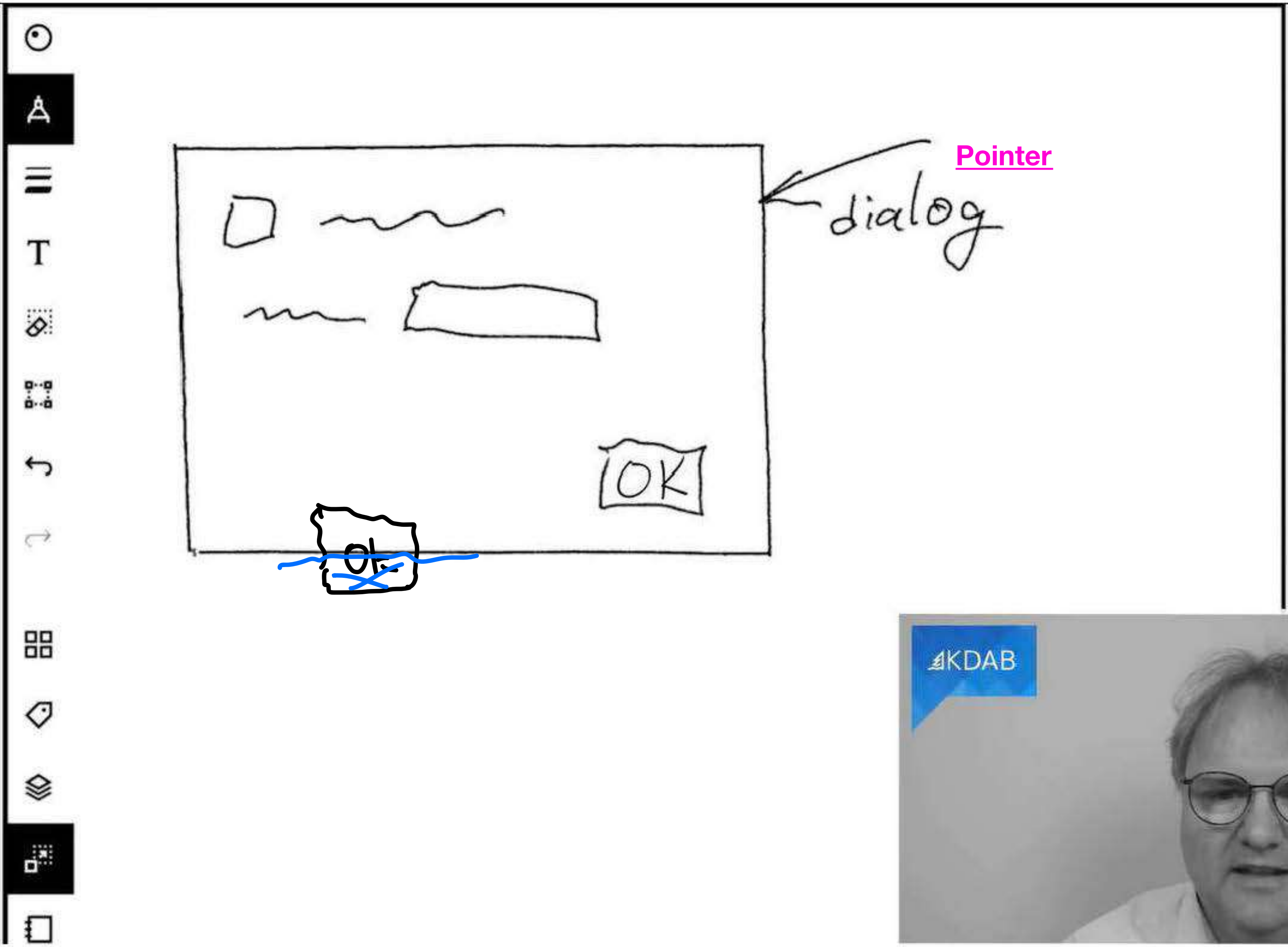
```
QString name;
QStringList list;
QColor color;
```

  - *Implicitly shared - Cheap to copy*
  - "Value" types are not QObject subclasses.

Qt's Object Model - QWidget

- **Derived from `QObject`**
  - Adds visual representation
- **Base of widget-based user interface objects**
- **Receives events**
  - e.g. mouse, keyboard events
- **Paints itself on screen**
  - Using styles





Pointer

Object Tree and QWidget

- **`new QWidget(/* nullptr */)`**
  - Widget with no parent = "window"

→top-level window

- **QWidget's children**
  - Positioned in parent's coordinate system
  - Clipped by parent's boundaries

Depend on parent coordinate sys

- **QWidget parent**
  - Propagates state changes
    - Hides/shows children when it is hidden/shown itself
    - Enables/disables children when it is enabled/disabled itself

Callbacks

General Problem

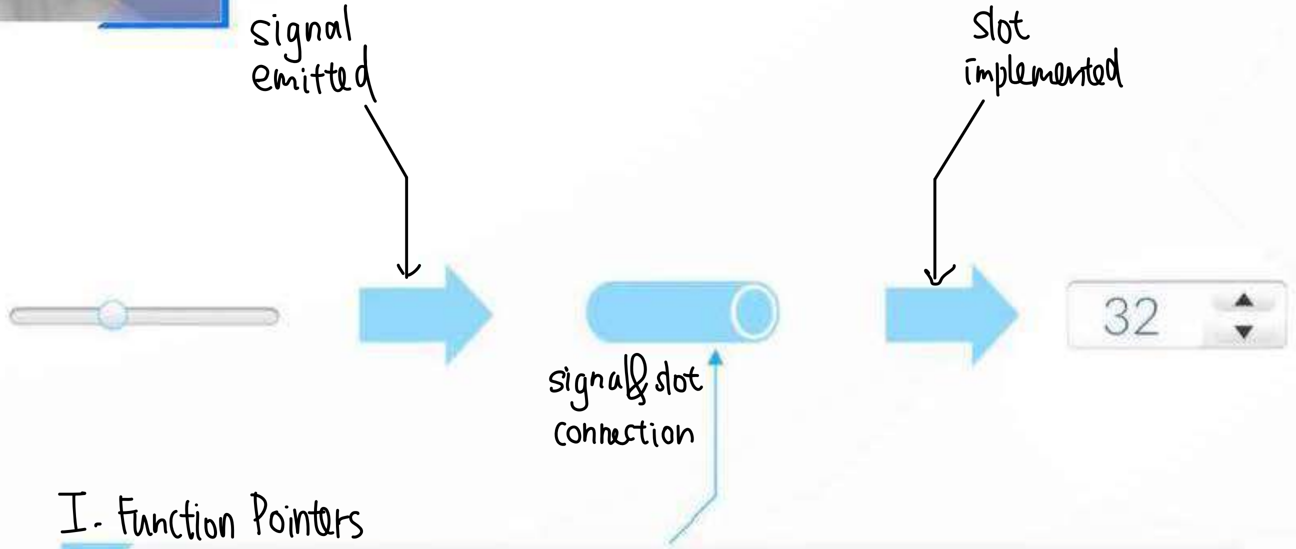
intension

How do you get from "the user clicks a button" to your business logic?

state changes

- Possible solutions
  - Callbacks
    - Based on function pointers
    - Not type-safe
  - Observer Pattern (Listener)
    - Based on interface classes
    - Needs listener registration
    - Many interface classes
- Qt uses
  - Signals and slots for high-level (semantic) callbacks.
  - Virtual methods for low-level (syntactic) events.

Connecting Signals to Slots



I. Function Pointers

```
QObject::connect ( slider,     &QSlider::valueChanged,
                   spinBox,  &QSpinBox::setValue );
```

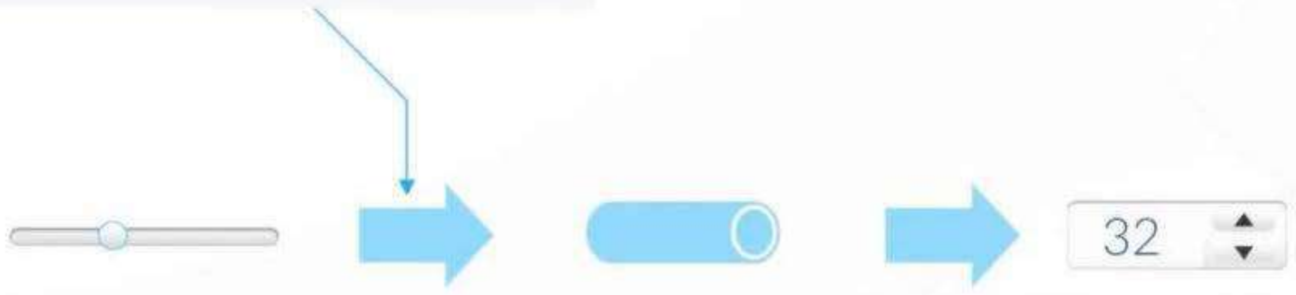
signal emission
⇓
slot invocation

II. Signal/slot macros

III. Function Obj

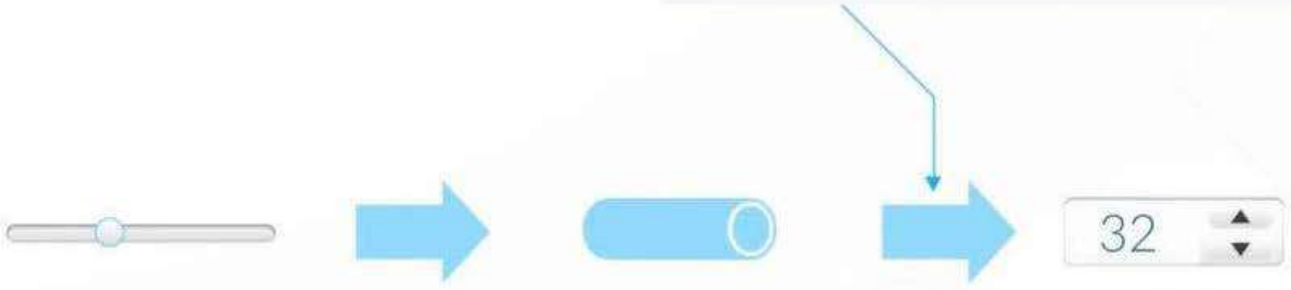
Connecting Signals to Slots

```
void QSlider::mouseMoveEvent(...)
{
    ...

    emit valueChanged(newValue);

    ...
}
```



Connecting Signals to Slots

```
void QSpinBox::setValue(int value)
{
   ...

   m_value = value;

   ...
}
```



Connect - Function Pointers

- Function pointers

```
connect( const QObject * sender,   PointerToMemberFunction signal,
         const QObject * receiver, PointerToMemberFunction method )
```

- Example:

```
QSlider *slider = new QSlider( Qt::Horizontal );
QSpinBox *spin = new QSpinBox;
QObject::connect( slider, &QSlider::valueChanged,
                  spin, &QSpinBox::setValue );
```

Demo: objects/ex-connect-function-pointers

Function Objects: Lambda Functions

- Lambda functions

```
connect( const QObject * sender, PointerToMemberFunction signal,
         Functor functor )
```

- Example:

```
QPushButton *button = new QPushButton( "Press Me!" );
QObject::connect( button, &QPushButton::pressed,
                  [button] { button->setText("Release Me!"); } );
```

Demo: objects/ex-connect-function-pointers

Connect

Rule for Signal/Slot Connection
Can ignore arguments, but cannot create values from nothing

Signal		Slot
rangeChanged(int,int)	✓	setRange(int,int)
	✓	setValue(int)
	✓	update()
valueChanged(int)	✓	setValue(int)
	✓	update()
	✗	setRange(int,int)
textChanged(QString)	✗	setValue(int)
clicked()	✓	update()
	✗	setValue(int)

Variations of Signal/Slot Connections

Signal(s)	Connect to	Slot(s)
one	✓	many
many	✓	one
one	✓	another signal

- Signal to signal connection

```
connect(button, &QPushButton::clicked, this, &MyClass::okSignal);
connect(button, SIGNAL(clicked()),     this, SIGNAL(okSignal()));
```

Disconnecting a Connection

- Function pointers

```cpp
QObject::connect( sender,  &Sender::valueChanged,
                  receiver, &Receiver::updateValue );
...
QObject::disconnect( sender, &Sender::valueChanged,
                     receiver, &Receiver::updateValue );
```

- SIGNAL/SLOT macros

```cpp
QObject::connect( sender,  SIGNAL(valueChanged(int)),
                  receiver, SLOT(updateValue(int)) );
...
QObject::disconnect( sender, SIGNAL(valueChanged(int)),
                     receiver, SLOT(updateValue(int)) );
```

Disconnecting a Connection (cont'd)

- Connection handle

```
QMetaObject::Connection m_connection;
m_connection = QObject::connect(...);
...
QObject::disconnect( m_connection );
```

Custom Slots

- File: **myclass.h**

```cpp
class MyClass : public QObject
{
  Q_OBJECT // marker for moc
  // ...
public slots:
  void setValue(int value); // a custom slot
};
```

- File: **myclass.cpp**

```cpp
void MyClass::setValue(int value) {
  // slot implementation
}
```

$$\begin{tikzcd}\boxed{x.h} \arrow[r, bend left, "moc"] & \boxed{moc-x.cpp}\end{tikzcd}$$

Custom Signals

- File: **myclass.h**

```
class MyClass : public QObject
{
  Q_OBJECT // marker for moc
  // ...
signals:
  void valueChanged(int value); // a custom signal
};
```

- File: **myclass.cpp**

```
// No implementation for a signal
```

- Sending a signal

```
emit valueChanged(value);
```

- **Note:** The Q_OBJECT macro is always required when defining custom signals.

Demo: objects/ex-window-watcher

Event Handling

- `QObject::event(QEvent *event)`
  - Handles all events for this object
- `QWindow` and `QWidget` have specialized event handlers.
  - `mousePressEvent()` for mouse clicks
  - `keyPressEvent()` for key presses

drag&drop

- Accepting an event
  - *event->accept()/event->ignore()*
    - Accepts or ignores the event
    - Accepted is the default.
- Event propagation
  - Events might be propagated to parent widget if the event is ignored.
  - Events on non-QWidgets are never automatically propagated.

Demo: objects/ex-allevents

Interlude: Painting

- Painting is done using the class `QPainter`.
  - `drawLine()`
  - `drawText()`
  - `drawPixmap()`
  - ...
- Painting is done in `paintEvent()`.
- Request a paint event from code using `update()`.
- `QPixmap` - off-screen pixel storage

Demo: objects/ex-paint-program

Qt Designer

- **Episode 13 - First Steps in Qt Designer**
- Episode 14 - Hooking your Qt Designer UI up to C++ Code
- Episode 15 - Layout in Qt Designer
- Episode 16 - Signals/Slots from UI Take II
- Episode 17 - Buddies and Tab Order
- Episode 18 - Custom Widgets
- Episode 19 - Pointers to Qt Widgets and More