Code less.
Create more.
Deploy everywhere.

# Qt event loop, networking and I/O API

Thiago Macieira, Qt Core Maintainer

**Aspen, May 2013**

# Who am I?

- **Open Source developer for 15 years**
- **C++ developer for 13 years**
- **Software Architect at Intel's Open Source Technology Center (OTC)**
- **Maintainer of two modules in the Qt Project**
  - QtCore and QtDBus
- **MBA and double degree in Engineering**
- **Previously, led the "Qt Open Governance" project**

# Qt 5

## First major version in 7 years

### Goals:

- New graphics stack
- Declarative UI design with QML
- More modular for quicker releases
- New, modern features

### Release status:

- Qt 5.0.2 released in April
- Qt 5.1.0 beta 1 released in May

# Agenda

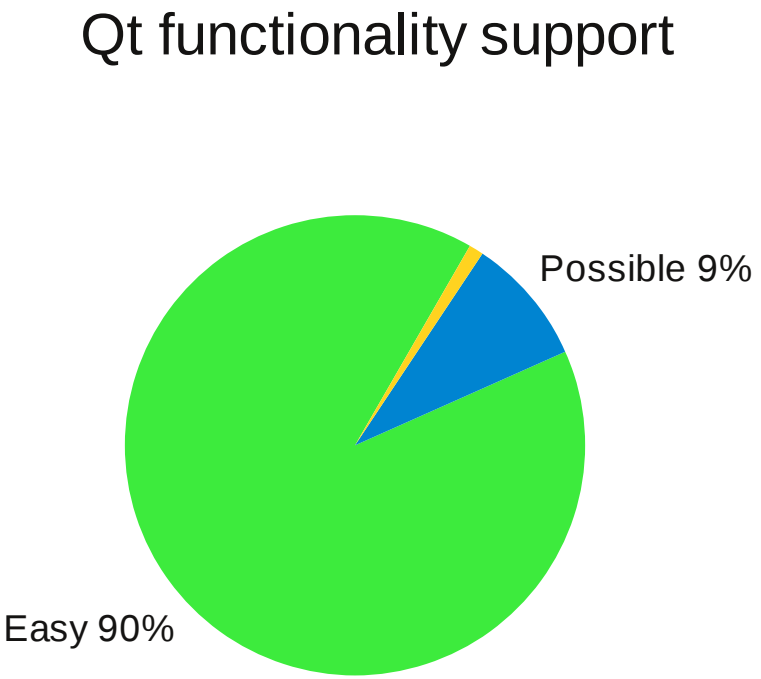
- Qt API Basics
- The event loop
- Event loops and threads
- Networking and I/O

# Qt API Basics

# API principles

## Qt strives to be:

- Easy to use (intuitive)
- Powerful
- Cross-platform
- Backwards compatible for years



# Naming convention

- camelCase with first letter lowercase
- Properties: nouns or sometimes adjectives
  - state, value, duration, size; visible, enabled, checkable
- Mutators and actions (including slots): verbs in the imperative
  - lock, set, create, load, append, replace, compare
- Signals: verbs in the past tense (sometimes implicit)
  - started, stopped, connected; bytes(Were)Written, undo(Became)Available
- Exceptions to the rules exist:
  - Compatibility with Standard Library, like begin() instead of beginning()

# Events vs signals

- Events derive from QEvent
- Carry **information**
- Directed to one destination
- Usually from the outside world (spontaneous event)
- Require overriding virtuals to be handled

- Signals are member functions[1]
- Indicate **state changes**
- Are not directed
- Usually indicate processing done by an object
- Does not require a new class, just a receiver member (slot)





[1] There's a reason for that: http://www.macieira.org/blog/2011/09/the-future-of-moc/

# Signals and slots

- The main communication mechanism in Qt
- M:N connections
- Does not require a new class for handling a specific purpose

You can connect one signal to another

```
QObject::connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));

// convenience signal forwards
QObject::connect(doc, SIGNAL(undoAvailable(bool)), q, SIGNAL(undoAvailable(bool)));
QObject::connect(doc, SIGNAL(redoAvailable(bool)), q, SIGNAL(redoAvailable(bool)));
```

New syntax:

```
QObject::connect(thread, &QThread::finished, thread, &QObject::deleteLater);
QObject::connect(qApp, &QCoreApplication::aboutToQuit, [=](){});
```

# Emitting a signal

- Signals are member functions whose body is implemented by **moc**
- To emit a signal, simply call the signalfunction

```
emit readBufferSizeChanged(size);
```

- `emit` and `Q_EMIT` are just for readability
  - They are `#define`'d to nothing

# Connection types

1. DirectConnection:
   - Slot is called immediately, by simple function call
   - Synchronous, same thread
2. QueuedConnection:
   - An event is posted so the slot is called later
   - Asynchronous, potentially run in another thread
3. BlockingQueuedConnection:
   - Same as QueuedConnection, but includes a semaphore to wait
   - Synchronous, always run in another thread

- AutoConnection: choose at emission time

# Agenda

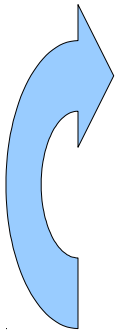
- Qt API Basics
- The event loop
- Event loops and threads
- Networking and I/O

# The Event Loop

# Classes relating to the event loop

- QAbstractEventDispatcher
- QEventLoop
- QCoreApplication
- QTimer & QBasicTimer
- QSocketNotifier & QWinEventNotifer
- QThread
-

# What an event loop does



- While !interrupted:
- If there are new events, dispatch them
- Wait for more events

- Event loops are the inner core of modern applications
  - Headless servers and daemons, GUI applications, etc.
  - Everything but shell tools and file-processing tools

# Qt event loop feature overview

- Receives and translates GUI events
  - NSEvents (on Mac)
  - MSG (on Windows)
  - X11 events (using XCB)
  - Wayland events
  - Many others
- Manages the event queue (priority queue)
- Allows for event filtering
- Integrates with:
  - Generic Unix `select(2)`
  - Glib
  - Mac's CFRunLoop
  - Windows's WaitForMultipleObjects
- Services timers
  - With coarse timer support
- Polls sockets
- Polls Windows events
- Thread-safe waking up

# QAbstractEventDispatcher

- Each thread has one instance
- Only of interest if you're porting to a new OS...

```
    virtual bool processEvents(QEventLoop::ProcessEventsFlags flags) = 0;
    virtual bool hasPendingEvents() = 0;

    virtual void registerSocketNotifier(QSocketNotifier *notifier) = 0;
    virtual void unregisterSocketNotifier(QSocketNotifier *notifier) = 0;

    virtual void registerTimer(int timerId, int interval,
                               Qt::TimerType timerType, QObject *object) = 0;
    virtual bool unregisterTimer(int timerId) = 0;

#ifdef Q_OS_WIN
    virtual bool registerEventNotifier(QWinEventNotifier *notifier) = 0;
    virtual void unregisterEventNotifier(QWinEventNotifier *notifier) = 0;
#endif

    virtual void wakeUp() = 0;
    virtual void interrupt() = 0;
```

# Data sources: QTimer

- Millisecond-based timing
- One signal: `timeout()`
- Convenience function for single-shot timers
- Three levels of coarseness:
  - Precise: accurate to the millisecond
  - Coarse (default): up to 5% error introduced so timers can be coalesced
  - VeryCoarse: rounded to the nearest second

```
QTimer timer;
timer.setInterval(2000);
QObject::connect(&timer, &QTimer::timeout, []() {
    qDebug() << "Timed out";
    qApp->exit(1);
});
```

# Data sources: QSocketNotifier

- Polling for read, write or exceptional activity on sockets
- On Unix, can poll anything that has a file descriptor
- One signal: `activated(int)`

```
PipeReader::PipeReader(int fd, QObject *parent)
    : QObject(parent), notifier(nullptr), m_fd(fd)
{
    notifier = new QSocketNotifier(m_fd, QSocketNotifier::Read, this);
    connect(notifier, &QSocketNotifier::activated, this, &PipeReader::readFromFd);
}
```

# On Unix, it really means...

Atomic variable

```
while (!interrupted.load()) {
    struct timeval tm = maximumWaitTime();
    fd_set readFds = enabledReadFds;
    fd_set writeFds = enabledWriteFds;
    fd_set exceptFds = enabledExceptFds;

    emit aboutToBlock();
    ::select(maxFd + 1, &readFds, &writeFds, &exceptFds, &tm);
    emit awake();

    sendPostedEvents();
    dispatchFds(&readFds, &writeFds, &exceptFds);
    dispatchExpiredTimers();
}
```

# Data sources: QWinEventNotifier

- Windows is different...
- Similar to QSocketNotifier
- One signal: `activated(HANDLE)`

# Data sinks: objects (slots and event handlers)

- Objects receive signals and event
- Starting with Qt 5, signals can be connected to lambdas, non-member functions, functors and some types of member functions

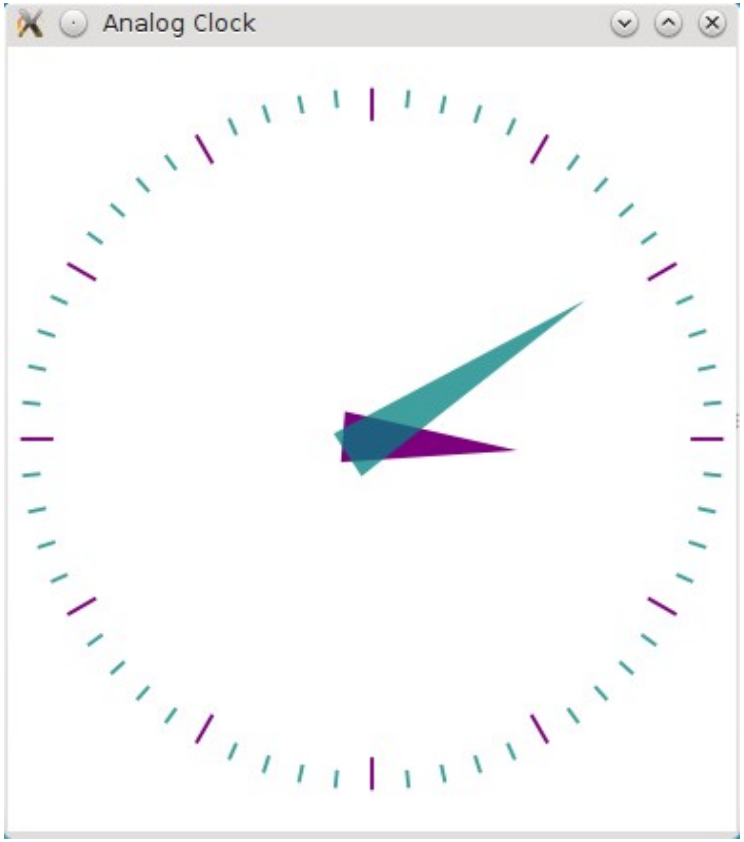
```
public slots:
    void start();

private slots:
    void closeChannel();
    void readFromFd();
```

```
void PipeReader::closeChannel()
{
    ::close(m_fd);
    emit channelClosed();
}
```

# Q{Core,Gui,}Application

- One instance of QCoreApplication per application
  - QGuiApplication if you want to use QWindow
  - QApplication if you want to use QWidget
- Defines the “GUI thread” and connects to the UI server
- Starts the main event loop

# A typical main(): exec()




```
int main(int argc, char **argv)
{
    QGuiApplication app(argc, argv);

    AnalogClockWindow clock;
    clock.show();

    app.exec();
}
```

# Simple example: timed read from a pipe (Unix)

```
class PipeReader : public QObject
{
    Q_OBJECT
public:
    PipeReader(int fd, QObject *parent = 0);

signals:
    void dataReceived(const QByteArray &data);
    void channelClosed();
    void timeout();

private slots:
    void closeChannel();
    void readFromFd();
};
```

# Simple example: timed read from a pipe (Unix)

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    PipeReader reader(fileno(stdin));
    QObject::connect(&reader, &PipeReader::channelClosed, &QCoreApplication::quit);
    QObject::connect(&reader, &PipeReader::dataReceived, [](const QByteArray &data) {
        qDebug() << "Received" << data.length() << "bytes";
    });

    QTimer timer;
    timer.setInterval(2000);
    QObject::connect(&timer, &QTimer::timeout, []() {
        qDebug() << "Timed out";
        qApp->exit(1);
    });

    timer.start();
    return a.exec();
}
```

# Nesting event loops: QEventLoop

- Make non-blocking operations into “blocking” ones
- Without freezing the UI

- Avoid if you can!

# Using the pipe reader with QEventLoop

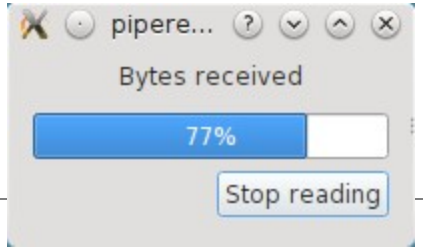
```
QByteArray nestedLoop()
{
    QEventLoop loop;
    QByteArray data;
    PipeReader reader(fileno(stdin));
    QObject::connect(&reader, &PipeReader::dataReceived,
                     [&](const QByteArray &newData) { data += newData; });
    QObject::connect(&reader, &PipeReader::channelClosed, &loop, &QEventLoop::quit);

    loop.exec();
    return data;
}
```

# exec() in other classes

- It appears in modal GUI classes
  - QDialog
  - QProgressDialog
  - QFileDialog

  - QMenu
- Like QEventLoop's exec(), avoid if you can!
  - Use show() and return to the main loop

# Using the pipe reader with QProgressDialog




```
int progressDialog()
{
    QProgressDialog dialog("Bytes received", "Stop reading", 0, 100); // expect 100 bytes
    PipeReader reader(fileno(stdin));
    QObject::connect(&reader, &PipeReader::channelClosed,
                     &dialog, &QProgressDialog::accept);
    QObject::connect(&reader, &PipeReader::dataReceived, [&](const QByteArray &data) {
        qDebug() << "Received" << data.length() << "bytes";
        dialog.setValue(qMin(dialog.maximum(), dialog.value() + data.length()));
    });

    dialog.exec();
    return dialog.result() == QDialog::Accepted ? 0 : 1;
}
```

# Threading with Qt

# QThread

- **Manages** a new thread in the application
  - Starting, waiting, requesting or forcing exit, notifying of exit, etc.
- Also provides some methods for the **current** thread
  - Sleeping, yielding

# Why threads?

## Good reasons

- Calling blocking functions
- CPU-intensive work
- Real-time work
- Scalability

## Bad reasons

- To use `sleep()`
- Networking and I/O
  - Except for scalability

# A typical thread (without event loops)

```
class MyThread : public QThread
{
public:
    void run()
    {
        // code that takes a long time to run goes here
    }
};
```

# Thread example: blocking read from a pipe

```
void run()
{
    forever {
        QByteArray buffer(4096, Qt::Uninitialized);
        ssize_t r = ::read(m_fd, buffer.data(), buffer.size());
        if (r <= 0)
            return;
        buffer.resize(r);
        emit dataReceived(buffer);
    }
}
```

# Threads and event loops: thread affinity

- Each QObject is associated with one thread
- Where its event handlers and slots will be called
- Where the object must be deleted

# Moving objects to threads

```
ThreadedPipeReader2::ThreadedPipeReader2(int fd, QObject *parent)
    : QThread(parent), reader(new PipeReader(fd))
{
    reader->moveToThread(this);
    connect(reader, &PipeReader::dataReceived, this,
            &ThreadedPipeReader2::dataReceived, Qt::DirectConnection);
    connect(reader, &PipeReader::channelClosed,
            [=]() { delete reader; quit(); });
}
```

Automatically moves children

```
PipeReader::PipeReader(int fd, QObject *parent)
    : QObject(parent), notifier(nullptr), m_fd(fd)
{
    notifier = new QSocketNotifier(m_fd, QSocketNotifier::Read, this);
    connect(notifier, &QSocketNotifier::activated, this, &PipeReader::readFromFd);
}
```

# Connection types *redux*

- DirectConnection:
  - Slot is called immediately, by simple function call
  - Synchronous, same thread
- QueuedConnection:
  - An event is posted so the slot is called later
  - Asynchronous, potentially run in another thread
- BlockingQueuedConnection:
  - Same as QueuedConnection, but includes a semaphore to wait
  - Synchronous, always run in another thread
- AutoConnection: choose at emission time

# A typical thread with event loop

```
void ThreadedPipeReader2::run()
{
    // preparation goes here
    exec();
    // clean-up goes here
}
```

# Qt I/O and Networking

# The I/O classes

## Random access

- QFile
  - QTemporaryFile
  - QSaveFile
- QBuffer

## Sequential access

- QNetworkReply
- QProcess
- QLocalSocket
- QTcpSocket
- QUdpSocket

# QIODevice

- Base class of all Qt I/O classes
- Provides:
  - CRLF translation
  - `read()`, `readAll()`, `readLine()`, `write()`, `peek()`
  - Signals: `readyRead()`, `bytesWritten()`
  - Buffering support, `bytesAvailable()`, `bytesToWrite()`
  - `atEnd()`, `size()`, `pos()`, `seek()`

# Random-access I/O characteristics

- **Defining feature**: seek() and size()
- Device can be put in unbuffered mode
- All I/O is synchronous
  - I/O latency is not considered as *blocking*
- No notification support (signals not emitted)

# Random-access example: QFile

Can open a file name, a FILE* or a file descriptor

```
QFile f;
f.open(stdin, QIODevice::ReadOnly | QIODevice::Text);
while (!f.atEnd()) {
    QByteArray line = f.readLine().trimmed();
    qDebug() << "Line is" << line.length() << "bytes long";
}
```

# Sequential-access I/O

- **Defining feature**: you must read sequentially
- I/O functions are **non-blocking** (asynchronous)
- All operations are **buffered**
  - Unlimited size by default
  - Actual I/O happens in the event loop
- Signals notify of incoming or outgoing activity

# The waitFor functions

- Exception to the non-blocking rule
- Provided for synchronous I/O
- Operate on the buffers
- Matched 1:1 with an I/O signal
  - `readyRead` → `waitForReadyRead`
  - `bytesWritten` → `waitForBytesWritten`
  - `connected / started` → `waitForConnected / waitForStarted`
  - `disconnected / finished` → `waitForDisconnected / waitForFinished`

# Synchronous sequential access

- Remember: read() and write() operate on **buffers**
- Must call waitForReadyRead() and/or waitForBytesWritten()
- Both functions execute **both** input and output
  - No buffer deadlock

# Asynchronous sequential access

- read() when readyRead() is emitted
- write() when necessary
- Event loop takes care of the rest!
- To limit buffer size:
  - Input buffer: setReadBufferSize()
  - Output buffer: manual control using bytesToWrite()

# When to use synchronous or asynchronous

- Networking in the GUI thread? **Always** asynchronous
- Multiple I/O in the same thread? Asynchronous
- **Short** child process? Synchronous
- Writing a blocking function? Synchronous

# QProcess example

```
AsyncProcess()
{
    auto proc = new QProcess;
    connect(proc,
            (void (QProcess::*)(int))&QProcess::finished,
            [=]() {
        proc->readLine(); // skip first line
        proc->deleteLater();
        emit dataReceived(proc->readLine().trimmed());
    });
    proc->start("qmake", QStringList() << "-v");
}
```

```
QByteArray syncProcess()
{
    QProcess proc;
    proc.start("qmake", QStringList() << "-v");
    proc.waitForFinished();

    proc.readLine(); // skip first line
    return proc.readLine().trimmed();
}
```

# HTTP/0.9 downloader example

```
QByteArray http09Downloader()
{
    QTcpSocket socket;
    socket.connectToHost("qt-project.org", 80);
    socket.write("GET /\r\n");
    socket.waitForDisconnected();
    return socket.readAll();
}
```

# Gopher example

```
QByteArray gopher()
{
    QTcpSocket socket;
    socket.connectToHost("gopher.floodgap.com", 70);
    socket.write("/feeds/latest\r\n");
    socket.waitForDisconnected();
    return socket.readAll();
}
```

TURE INTEL LINUX WIRELESS GUPNP KVM POKY OFONO LINUX KERNEL
OP YOCTO CONNMAN XEN
CS SYNCEVOLUTION SIMPLE FIRMWARE INTERFACE (SFI) ENTERPRISE SECURITY INFRASTRUCTURE

INTEL OPEN SOURCE
TECHNOLOGY CENTER