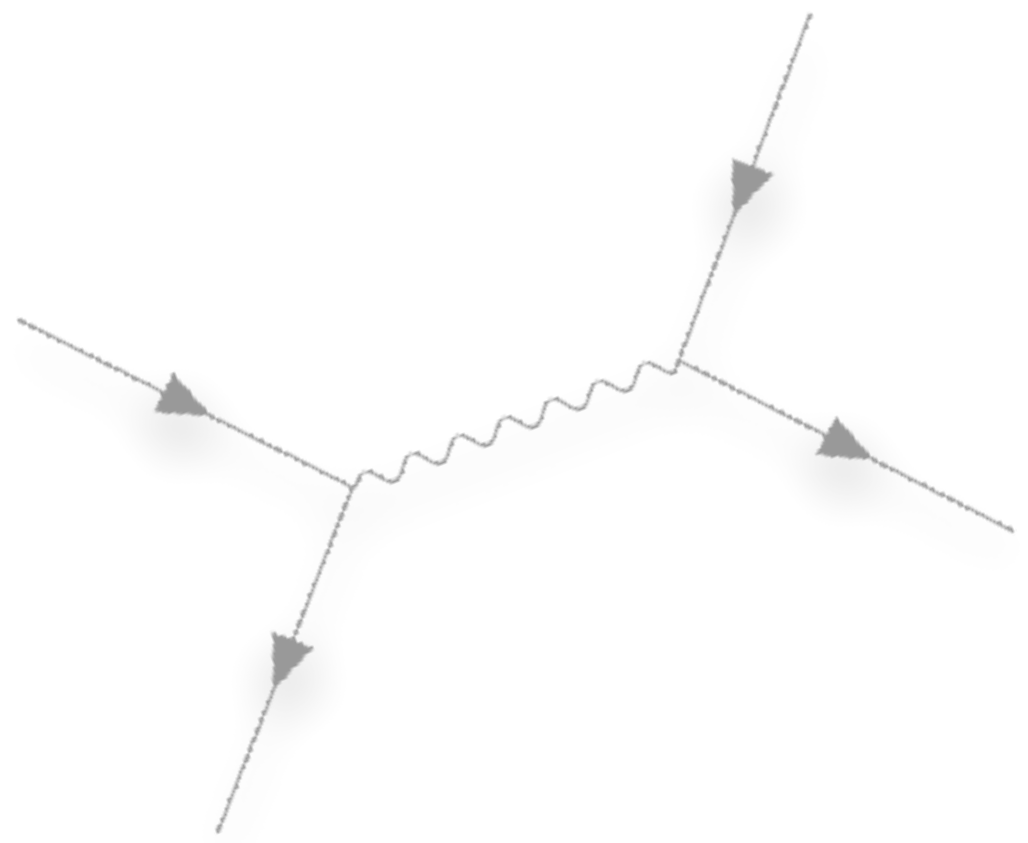
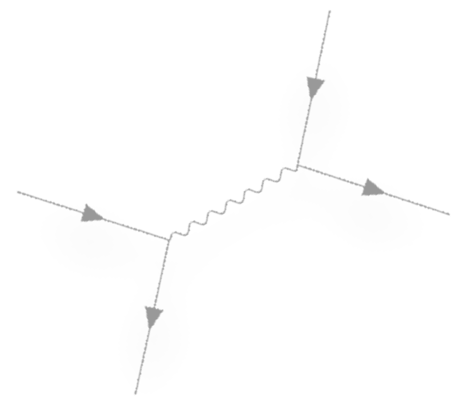
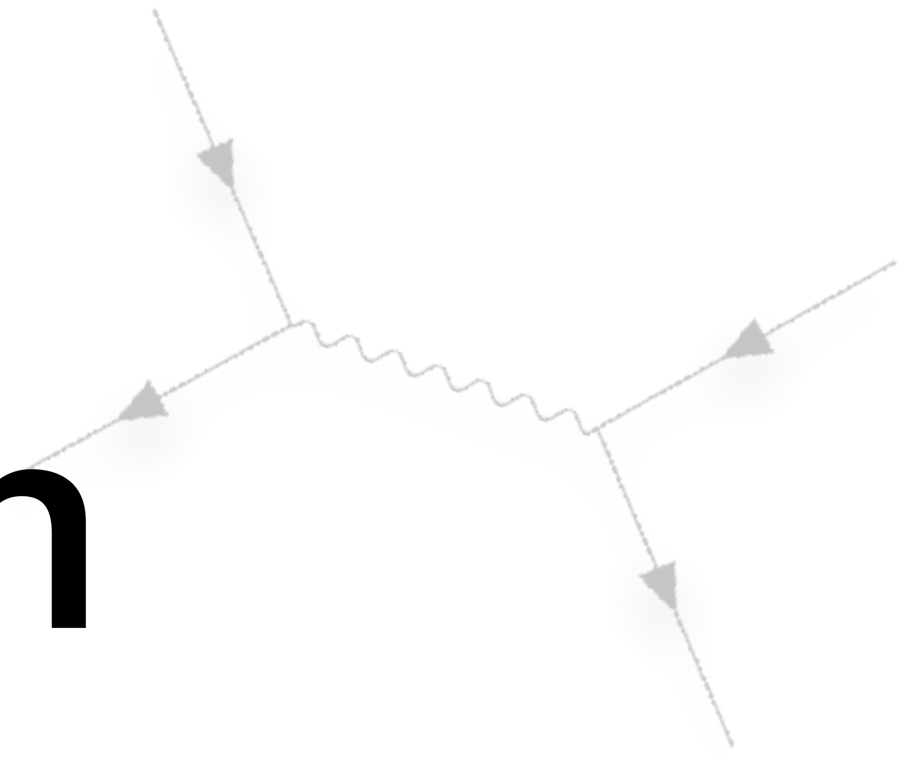
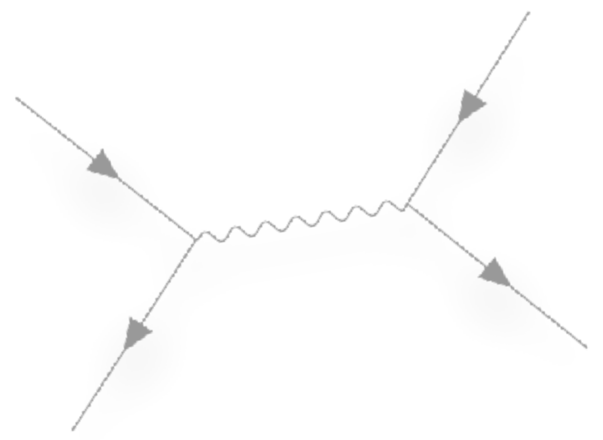
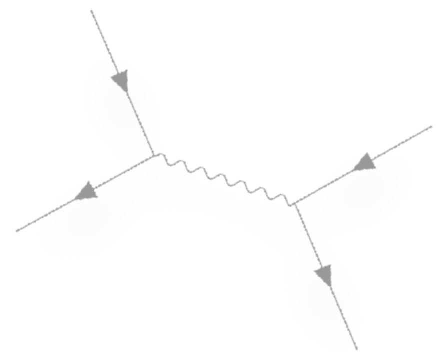



A Backgrounder on Background Tasks



David Rowland
Tracktion Corporation

Notes

- Slides are numbered
- Slides/examples will be on GitHub (as a PIP!)
- Code style tweaked for slide presentation

Contents

- Types of tasks and their runners
- Motivation for building `TaskRunner` framework
- Common pitfalls
- Expanding with composition

What is a Background Task?

- Background tasks are low-priority sections of code that run on background threads
- They may or may not block the UI until completion
  - E.g. rendering a DAW session vs. checking for updates
- Are used to avoid blocking the message thread and stalling the UI
- Some can be cancelled, some not
  - E.g. audio export vs. WAV proxy creation
- Can be dozens going on at the same time you don't even know about

- At startup:
  - Checking for software updates
  - Checking for translation updates
  - Checking for news/video content to display
  - Loading loop/preset libraries
  - Reading previous analytics from disk
  - Sending analytics to web servers
- Silently during operation:
  - Loading DAW sessions
  - Creating WAV proxy files
  - Rendering clip content (Edit clips, Clip FX, warp time, reverse etc.)
  - Scanning for plugins
  - Scanning for loops
  - Capturing plugin thumbnails
  - Audio file tempo detection
- From user direction:
  - Exporting/rendering audio
  - Archiving/unpacking projects
  - Downloading demo songs
  - Audio operations (normalise/trim silence/mono/sample rate conversion)

Type of Background Tasks

	Blocking*	Time-sliced**
Single thread	• `TaskRunner` • `juce::Thread` • `std::thread` • *Maybe* `std::async`	• `juce::TimeSliceThread`
Multiple threads	• `juce::Thread::launch` • *Maybe* `std::async`	• `juce::ThreadPool`

- *Blocking: should check yourself if the calling thread needs to exit
  - Not always possible - e.g. with `std::thread, std::future`
- **Time-sliced: return a job status
  - e.g. `juce::ThreadPool::jobNeedsRunningAgain/juce::TimeSliceClient::useTimeSlice`

Building a TaskRunner

Motivation

- Common task to run a time consuming operation on a background thread
- Used to avoid blocking the UI and keep it responsive
- Low-priority so often isn’t time critical (unlike the audio thread)
- E.g. drawing thumbnails, analysing tempo/beats, image processing, web uploads/downloads etc.
- Need to communicate back to the message thread at some point, often to update UI or continue an operation
- *Predates `juce::Thread::launch and juce::ThreadPool::addJob`

Tip

```
#define ASSERT_MESSAGE_THREAD \
jassert (MessageManager::getInstance()->isThisTheMessageThread());
```

- When working with multi-threaded code, add assertions before you have specifically dealt with thread-safety
- This helps pinpoint where you need to add synchronisation and avoids misuse from calling code

Requirements

- Simple, quick and easy to use
- Essentially one function without external dependancies
- Can run any number of tasks
- Runner can be used in a multitude of ways including as a member or singleton-like
- Possibility to have multiple TaskRunners for different types of tasks (so no inherent singleton!)

```
/** Adds a task to be run on a background thread. */
void addTask (std::function<void()>);
```

Interface

- Use `juce::Thread` as the thread source
  - `run()` override
- Optional name for debugging if required
- Destructor and `addTask` method
- `std::vector` to hold tasks
- Guarded with a `juce::CriticalSection`

```cpp
/**
    Runs a number of tasks sequentially on a background thread.
*/
struct TaskRunner  : private Thread
{
    /** Creates a TaskRunner with an optional thread name. */
    TaskRunner (const String& threadName = String());

    /** Destructor. */
    ~TaskRunner();

    /** Adds a task to be run on a background thread. */
    void addTask (std::function<void()>);

private:
    CriticalSection tasksLock;
    std::vector<std::unique_ptr<std::function<void()>>> tasks;

    std::unique_ptr<std::function<void()>> getNextTask();
    void run() override;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(TaskRunner)
};
```

Implementation

- Call `stopThread` in the destructor
- Give the tasks some time to complete

```
TaskRunner::TaskRunner (const String& threadName)
    : Thread (threadName)
{
}

TaskRunner::~TaskRunner()
{
   stopThread (5000);
}
```

Implementation

- Take the tasks lock using a scoped lock
- Move the task to run in to the back of the tasks vector
- Start the thread (if it’s not already running)
- Call `notify` to wake it up if it’s asleep

```
void TaskRunner::addTask (std::function<void()> task)
{
    {
        const ScopedLock sl (tasksLock);
        tasks.push_back (
            std::make_unique<std::function<void()>> (
                std::move (task))
        );
    }

    startThread (1);
    notify();
}
```

Implementation

- Continue whilst the thread hasn’t been asked to exit
- Grab the next task and run it
- Otherwise go to sleep to avoid using resources

```
void TaskRunner::run()
{
    while (! threadShouldExit())
    {
        if (auto task = getNextTask())
            (*task)();
        else
            wait (1000);
    }
}
```

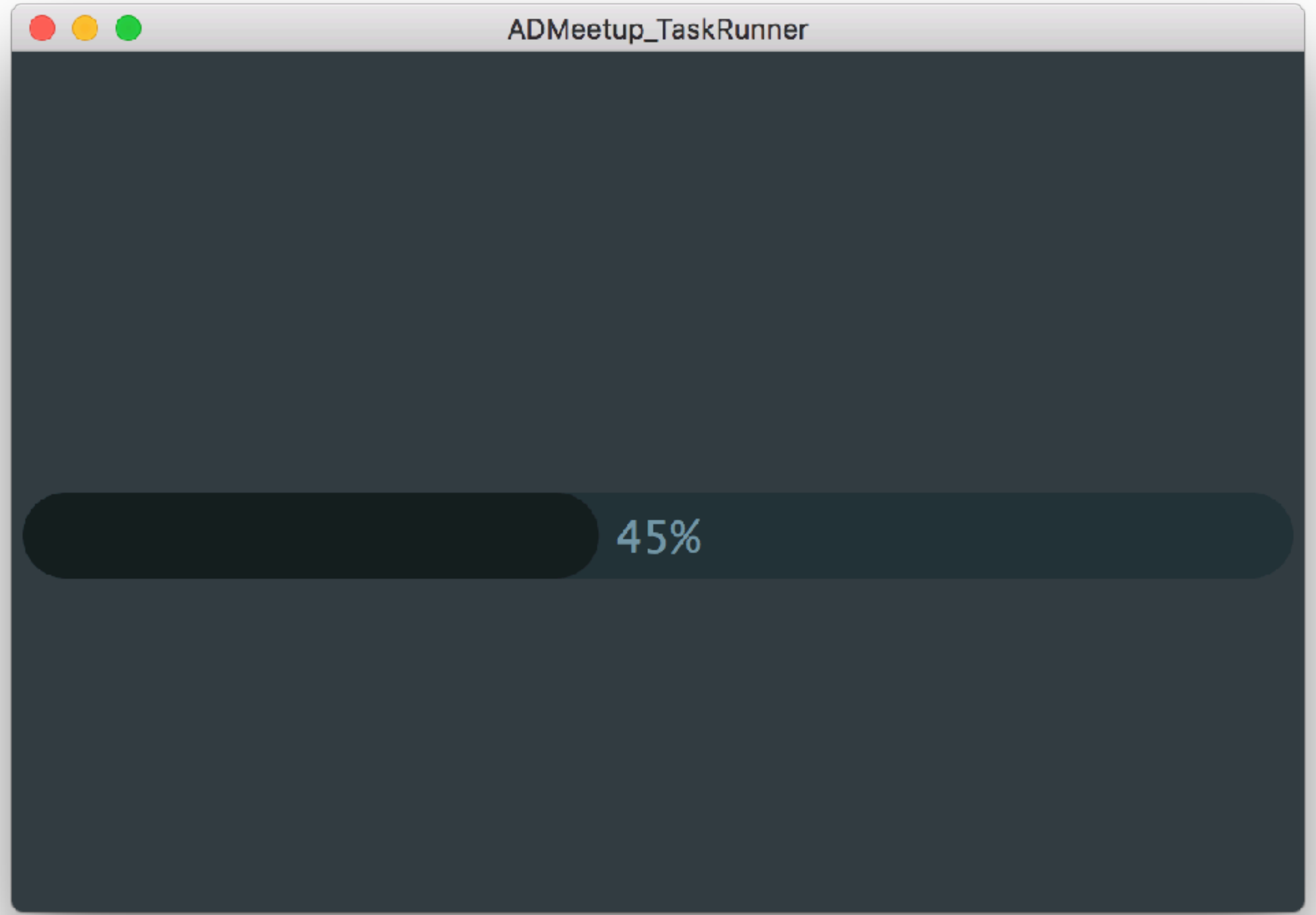
Implementation

- Return a `std::unique_ptr` to the task to run
- Take the `tasksLock` as we're interacting with the tasks vector
- If there are no tasks, return a `nullptr`
- Otherwise, move the first task out
- Erase the now empty shell of a `std::unique_ptr`
- Return the pointer to the task to run

```cpp
std::unique_ptr<std::function<void()>>
TaskRunner::getNextTask()
{
    const ScopedLock sl (tasksLock);

    if (tasks.empty())
        return nullptr;

    auto task = std::move (tasks.front());
    tasks.erase (tasks.begin());

    return task;
}
```

Demo

- Run a long task and send the time of completion back to the UI to show in a label



Usage - Simulate a long task

```cpp
void runTenSecondTask (double& progress)
{
    const double durationInSeconds = 10.0;
    const auto startTime = Time::getCurrentTime();
    const auto timeToEndAt = startTime + RelativeTime::seconds (durationInSeconds);

    for (;;)
    {
        const auto currentTime = Time::getCurrentTime();
        progress = jlimit (0.0, 1.0, (currentTime - startTime).inSeconds() / durationInSeconds);

        if (Thread::currentThreadShouldExit())
            break;

        if (currentTime > timeToEndAt)
            break;

        Thread::sleep (100);
    }
}
```

Usage

- How (not) to do it
- Simple component with a progress bar
- (Ignore the fact that writing to the `progress` member is data-race)
- A task is started using a lambda
  - Capture the `this` pointer to gain access to `progress`
  - Pass this on to the task
- Usually we need to get data back to the message thread, even if it's just to notify of completion (could be a UI update etc. though)
  - Here we simulate this by displaying the time of the task completion

```
class MyComponent   : public Component
{
public:
    //==============================================================================
    MyComponent()
    {
        statusLabel.setJustificationType (Justification::centred);
        addAndMakeVisible (statusLabel);
        addAndMakeVisible (progressBar);
        setSize (600, 400);

        taskRunner.addTask ([this]
                            {
                                runTenSecondTask (progress);
                                displayTime (Time::getCurrentTime());
                            });
    }

    //==============================================================================
    void paint (Graphics&) override
    void resized() override

private:
    //==============================================================================
    TaskRunner taskRunner;
    double progress = 1.0;
    ProgressBar progressBar { progress };

    Label statusLabel;

    void displayTime (Time time)
    {
        statusLabel.setText (time.toString (true, true), dontSendNotification);
    }

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MyComponent)
};
```

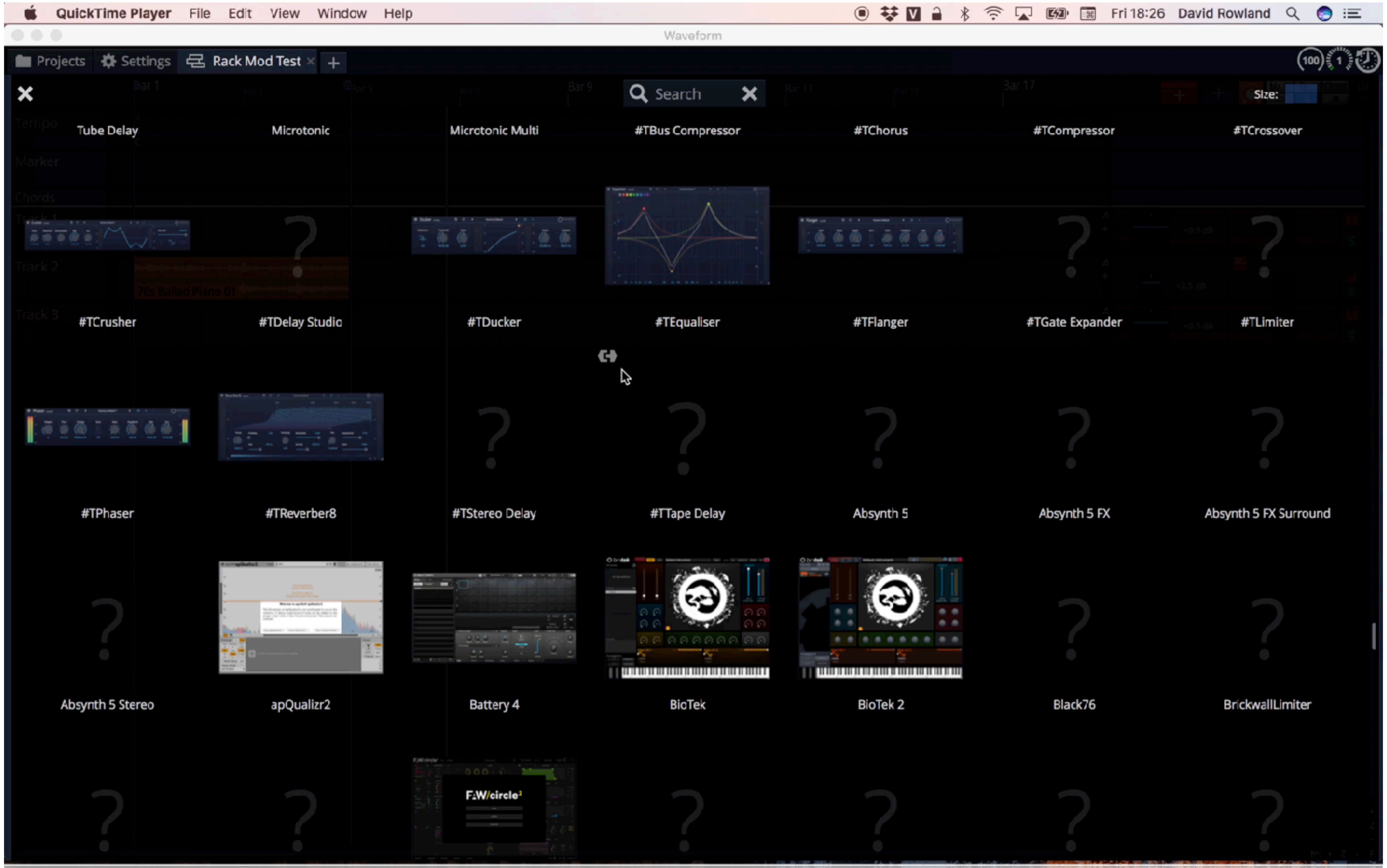
Dealing with Pitfalls

Usage - Notifying of completion

- **1) `this` may have been deleted**
- **2) Can't call in to `displayTime` from a background thread**
- `taskRunner` may outlive the `this` if it is shared
- Not safe to call JUCE (and every other UI framework) UI methods
  - Xcode now has the "Main Thread Checker" diagnostic precisely for this purpose

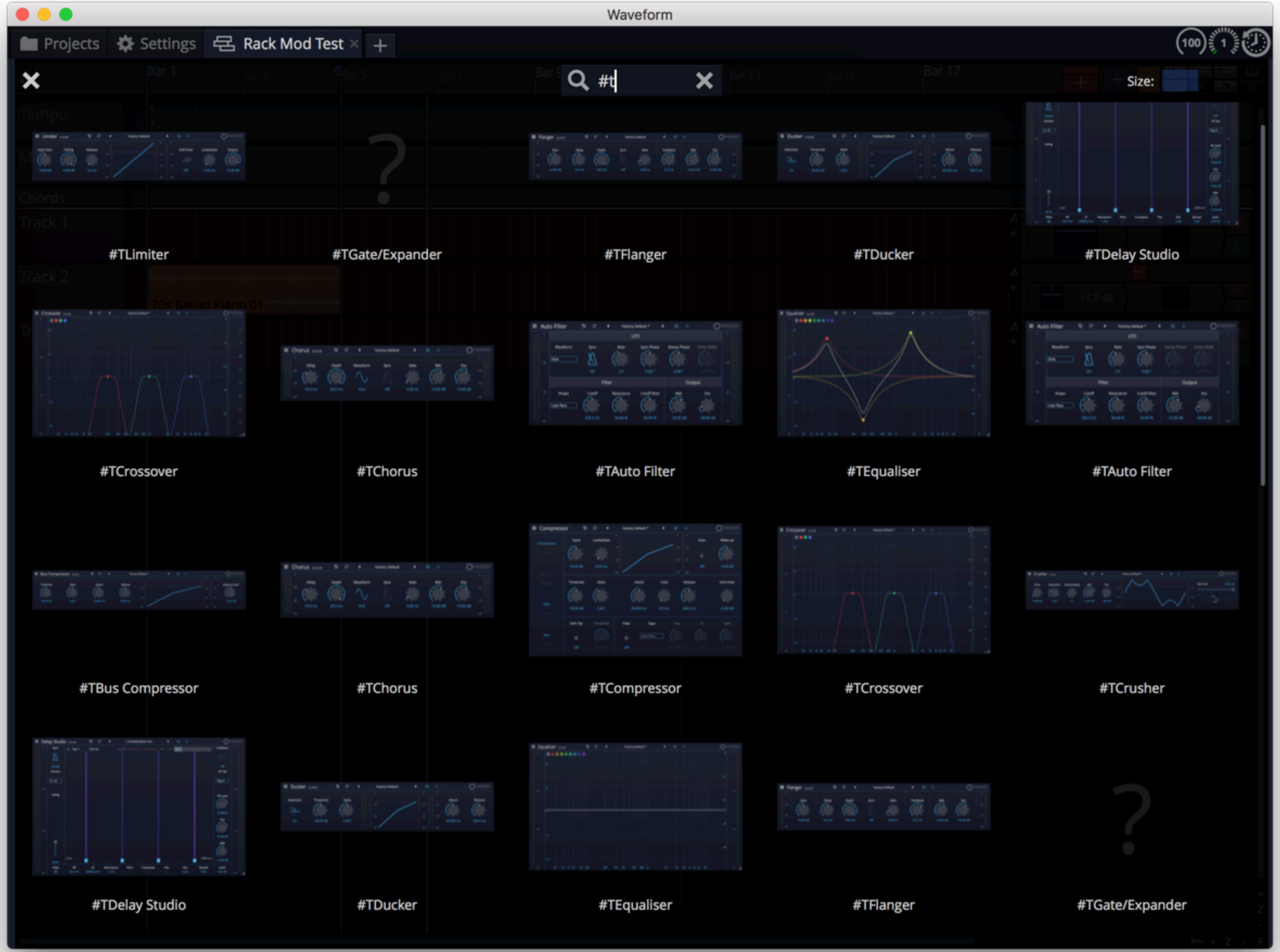
```
taskRunner.addTask ([this]
                    {
                        runTenSecondTask (progress);
                        displayTime (Time::getCurrentTime());
                    });
```

Typical Use of Task Runner



Typical Use of Task Runner

- Create a library of thumbnails/ images/beat displays etc.
- Scrolling through list quickly
- Don't want `TaskRunner` or thread per image so share one
- Display components may be reused or outlive tasks used to create their final content
- This is a technique used to keep scrolling responsive



Usage - Notifying of completion

- **1) `this` may have been deleted**
- Capture a `SafePointer<MyComponent>`
    - `SafePointer<MyComponent>` is a `WeakReference<MyComponent>`, akin to `std::weak_ptr`
    - *N.B. if we call a non-const method of MyComponent via the SafePointer we need to make the lambda mutable*
    - `SafePointer` is implemented using an atomic reference count so it's ok to check it from a background thread
    - However, it's a *weak pointer*, it doesn't keep `MyComponent` alive so it could be deleted after the check!

```
taskRunner.addTask (
[sp = SafePointer<MyComponent> (this)] () mutable
{
    // Run long task here...

    if (sp != nullptr)
        sp->displayTime (Time::getCurrentTime());
});
```

Usage - Notifying of completion

- **2) Can't call in to `displayTime` from a background thread**
- Due to message thread only call: `Label::setText`
- Calls `repaint` etc. and asserts/crashes
- Use `MessageManager::callAsync` to defer the call to the message thread
  - Not ok as the next call on could delete the component and `taskRunner`
  - `taskRunner` could be shared so could outlive the `this` pointer
  - Good chance `this` will be dangling!

```
taskRunner.addTask ([this]
{
    // Run long task here...

    MessageManager::getInstance()->callAsync ([this]
    {
        displayTime (Time::getCurrentTime());
    });
});
```

Usage - Notifying of completion

- Combine the techniques
  - Capture a `SafePointer`
  - Pass it on the to `callAsync` lambda *by value* to make a copy of it
  - Now we're on the message thread, `MyComponent` won't be deleted after we've checked the `SafePointer`
  - `displayTime` is safe to call from the message thread

```
taskRunner.addTask (
[sp = SafePointer<MyComponent> (this)] () mutable
{
    // Run long task here...

    MessageManager::getInstance()->callAsync (
    [sp] () mutable
    {
        if (sp != nullptr)
            sp->displayTime (Time::getCurrentTime());
    });
});
```

Midpoint Summary

- Looked at how to succinctly create a vector of tasks
- How to run those sequentially on a background thread
- How to check to see if those jobs should exit early
- How to safely pass data back to the message thread
- How to ensure weakly referenced objects are still valid
- Haven't looked at
  - Progress
  - How to cancel jobs

Expanding Capabilities

Expanding the Example

- Written task runners based around `std::future`/`juce::Thread`/`ThreadPool`/`TimeSliceClient`
- Every time I’ve created a new enclosing class to contain the number of jobs, their progress, methods to cancel them etc.
- Ideally, this should be separate and de-coupled from the task running implementation and the UI used to display it
- Using this approach, it doesn’t matter what threading model is used

Progress

- `Progress` simply represents a *named* progress
- `setProgress` is a mutator method to ensure a valid range

```
struct Progress
{
    Progress (String nameToUse)
        : name (std::move (nameToUse))
    {
    }

    String getName() const
    {
        return name;
    }

    void setProgress (double newProgress) noexcept
    {
        jassert (isPositiveAndNotGreaterThan (newProgress, 1.0));
        progress.store (newProgress);
    }

    double getProgress() const noexcept
    {
        return progress.load();
    }

private:
    const String name;
    std::atomic<double> progress { 0.0 };
};
```

Progress

- Atomic bool to represent if the task has been cancelled
- Provide a method to signal cancel and check state
- Once cancelled, this can't be un-cancelled

```
struct Progress
{
    Progress (String nameToUse) : name (std::move (nameToUse)) {}

    String getName() const      { return name; }

    void setProgress (double newProgress) noexcept
    {
        jassert (isPositiveAndNotGreaterThan (newProgress, 1.0));
        progress.store (newProgress);
    }

    double getProgress() const noexcept
    {
        return progress;
    }

    void cancel() noexcept
    {
        cancelled = true;
    }

    bool hasBeenCancelled() const noexcept
    {
        return cancelled;
    }

private:
    const String name;
    std::atomic<double> progress { 0.0 };
    std::atomic<bool> cancelled { false };
};
```

ProgressList

- Factory method to create a progress
  - Adds it to the internal list
  - Starts a timer which will clean up any complete progresses
  - Broadcasts a change message
- Broadcasts a change message when a progress is added or removed
- Method to return total progress
- Method to return all the progresses
  - For use in a list box etc.
- Timer callback updates internal list
- Use erase/remove idiom to stop monitoring a progress if it has
  - Completed (progress >= 1)
  - Been cancelled
- Notice `Progress` has no back-pointer to `ProgressList`

```cpp
struct ProgressList : public ChangeBroadcaster,
                      private Timer
{
    ProgressList() = default;

    std::shared_ptr<Progress> createProgress (const String& name)
    {
        ASSERT_MESSAGE_THREAD
        auto progress = std::make_shared<Progress> (name);
        progresses.push_back (progress);
        startTimer (100);
        sendChangeMessage();

        return progress;
    }

    double getTotalProgress() const;

    std::vector<std::shared_ptr<Progress>> getProgresses() const
    {
        ASSERT_MESSAGE_THREAD
        return progresses;
    }

private:
    std::vector<std::shared_ptr<Progress>> progresses;

    void timerCallback() override
    {
        removeExpiredProgresses();

        if (progresses.size() == 0)
            stopTimer();
    }

    void removeExpiredProgresses()
    {
        ASSERT_MESSAGE_THREAD
        const auto sizeBefore = progresses.size();
        progresses.erase (remove_if (begin (progresses), end (progresses),
                                     [] (auto p)
                                     {
                                         return p->hasBeenCancelled() || p->getProgress() >= 1.0;
                                     }),
                          end (progresses));

        if (sizeBefore > progresses.size())
            sendChangeMessage();
    }

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(ProgressList)
};
```

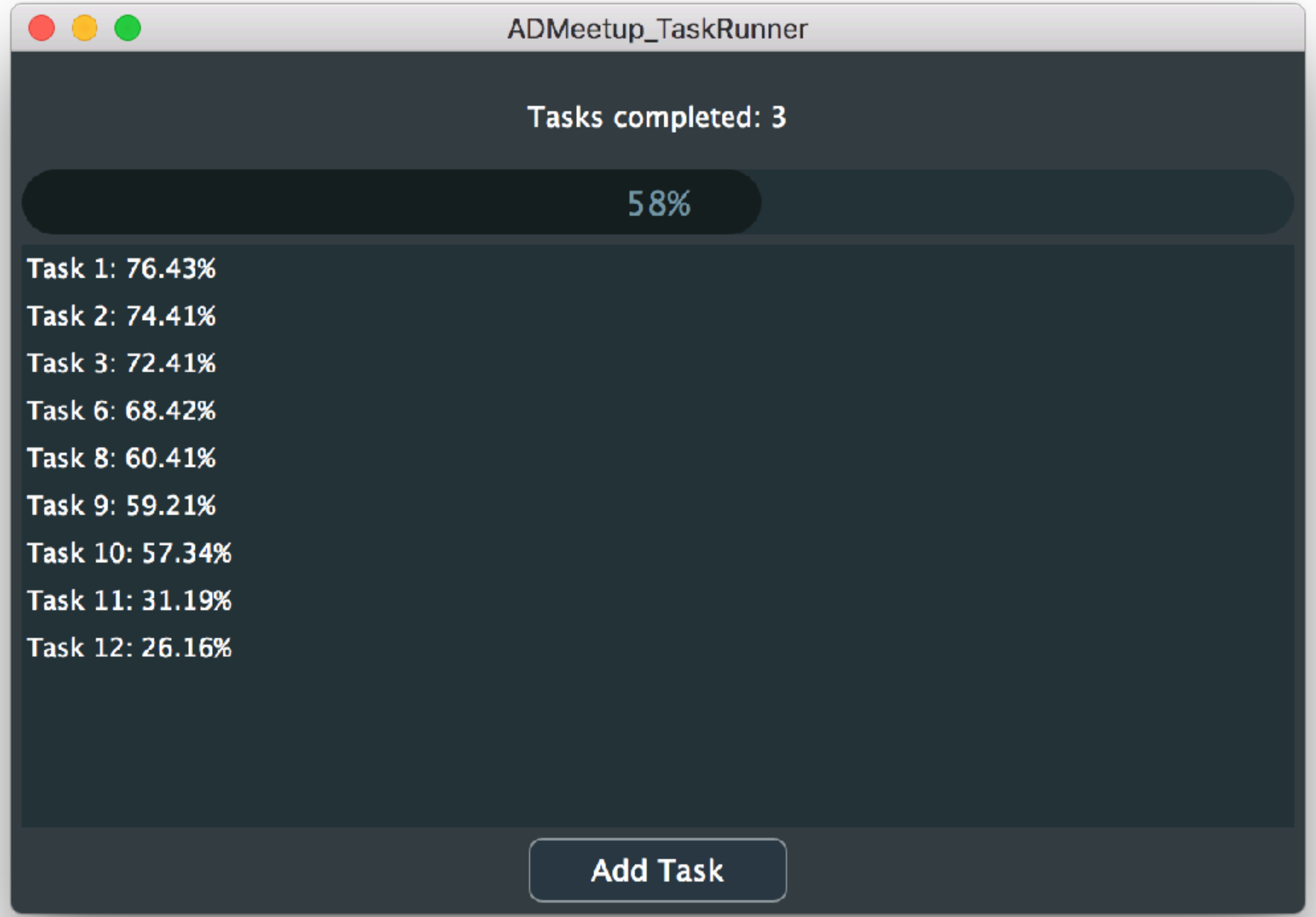
ProgressList

- Iterates all objects and returns the total progress
- Not the most efficient method but isn't being called that often (~25 Hz)

```
double getTotalProgress() const
{
    ASSERT_MESSAGE_THREAD
    if (progresses.empty())
        return 1.0;

    double total = 0.0;
    int count = 0;

    for (auto p : progresses)
    {
        const double progress = p->getProgress();

        if (progress < 0.0)
            return -1.0;

        total += jlimit (0.0, 1.0, progress);
        ++count;
    }

    return total / count;
}
```

Demo

- Demo - ProgressList ListBox



ADMeetup_TaskRunner

Tasks completed: 3

58%

```
Task 1: 76.43%
Task 2: 74.41%
Task 3: 72.41%
Task 6: 68.42%
Task 8: 60.41%
Task 9: 59.21%
Task 10: 57.34%
Task 11: 31.19%
Task 12: 26.16%
```

Update Long Task Simulation

```
void runTenSecondTask (std::function<bool (double)> updateProgressFn)
{
    const double durationInSeconds = 10.0;
    const auto startTime = Time::getCurrentTime();
    const auto timeToEndAt = startTime + RelativeTime::seconds (durationInSeconds);

    for (;;)
    {
        const auto currentTime = Time::getCurrentTime();

        if (! updateProgressFn (jlimit (0.0, 1.0, (currentTime - startTime).inSeconds() / durationInSeconds)))
            break;

        if (Thread::currentThreadShouldExit())
            break;

        if (currentTime > timeToEndAt)
            break;

        Thread::sleep (100);
    }
}
```

ProgressListComponent

- Creates a named `Progress` managed by the `progressList`
- Capture this in to the task
- The task then updates this `progress`
- Return `true` if job hasn't been cancelled
- Completion notification is done as before
- Note how we're using `juce::Thread::launch` now

```
auto progress = progressList.createProgress ("Task " + String (++numTasksStarted));
Thread::launch ([sp = SafePointer<MyComponent> (this), progress] () mutable
                {
                    runTenSecondTask ([progress] (double p)
                                      {
                                          progress->setProgress (p);
                                          return ! progress->hasBeenCancelled();
                                      });

                    MessageManager::getInstance()->callAsync ([sp] () mutable
                                                              {
                                                                  if (sp != nullptr)
                                                                      sp->taskCompleted();
                                                              });
                });
```

ProgressListComponent

- Listen to the `progressList` in order to update the `ListBox` content
- The number of rows is the size of the number of progresses
- Each row is then drawn by getting the appropriate `Progress` from the list

```cpp
void updateProgressAndList()
{
    progress = progressList.getTotalProgress();
    box.repaint();
}

void changeListenerCallback (ChangeBroadcaster*) override
{
    if (progressList.getProgresses().empty())
        stopTimer();
    else
        startTimerHz (25);

    updateProgressAndList();
    box.updateContent();
}

int getNumRows() override
{
    return (int) progressList.getProgresses().size();
}

void paintListBoxItem (int row, Graphics& g, int w, int h, bool isSelected) override
{
    auto progresses = progressList.getProgresses();

    if (row >= progresses.size())
        return;

    if (auto progress = progresses[row])
    {
        g.setColour (progress->hasBeenCancelled() ? Colours::red : Colours::white);
        g.setFont (14.0f);
        g.drawText (progress->getName() + ": " + String (progress->getProgress() * 100.0) + "%",
                    Rectangle<int> (w, h).reduced (2), Justification::centredLeft);
    }
}
```

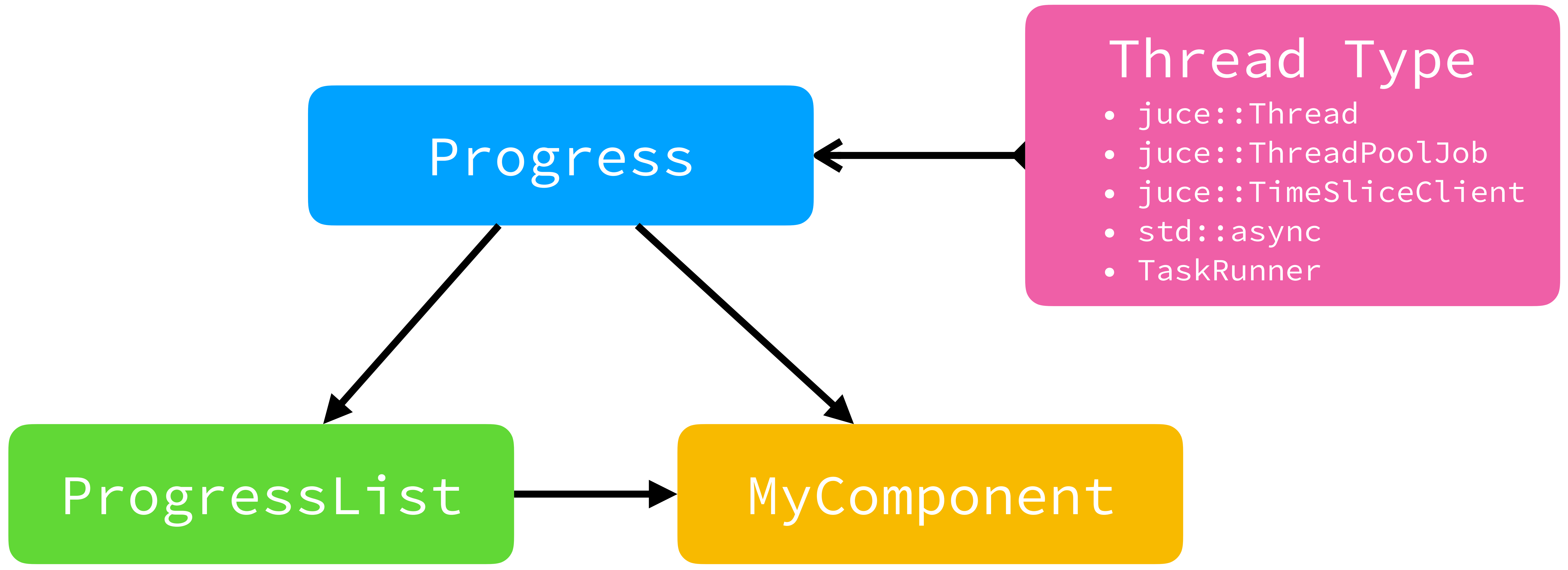
ProgressListComponent - Cancelling

- If the row is clicked we simply call `Progress::cancel`

```
void listBoxItemClicked (int row, const MouseEvent&) override
{
    auto progresses = progressList.getProgresses();

    if (row >= progresses.size())
        return;

    if (auto progress = progresses[row])
        progress->cancel();
}
```

Modularity

- Heavy use of composition to reduce coupling
- Use lambdas to construct our task *classes* on the fly
- `Progress` is agnostic to what creates/references it
- `ProgressList` is agnostic to what `Component` creates/references it
- Easy to change the thread type running the tasks



Changing the Threading

- Using `juce::Thread::launch` causes shutdown problems
- Could solve by waiting for jobs to finish and threads to exit
- Easier to use a `juce::ThreadPool`

```
MyComponent()
{
    //...
    Thread::launch ([sp, progress]
                    {
    //...
}

private:
//==========================================
ProgressList progressList;
int numTasksStarted = 0, numTasksCompleted = 0;
```

```
MyComponent()
{
    //...
    threadPool.addJob ([sp, progress]
                       {
    //...
}

private:
    //==========================================
    ThreadPool threadPool;
    ProgressList progressList;
    int numTasksStarted = 0, numTasksCompleted = 0;
```

Summary

- Created a custom `TaskRunner`
  - Ensured we exit task early if thread should exit
  - Avoided dangling `this` pointers
  - Safely passed data back to the message thread
- Created compose-able classes to manage task state
  - Progress
  - Cancel
- Swapped the `TaskRunner` for a `juce::Thread`
  - Then swapped for a `juce::ThreadPool` etc.
  - Could be swapped to any thread running mechanism

Questions?

Presentation/code available on GitHub:
https://github.com/drowaudio/presentations

Twitter:
@drowaudio