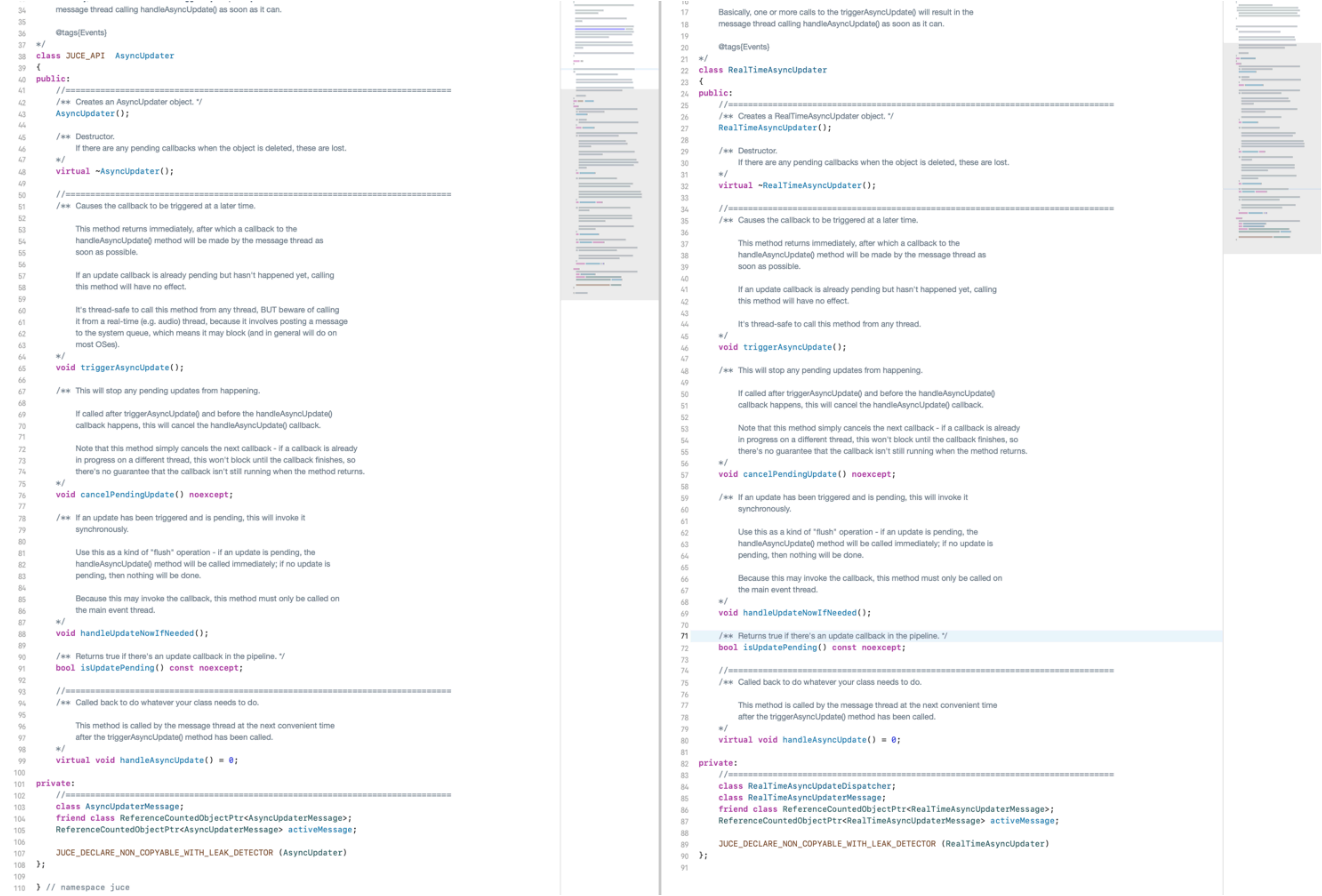

```
    message thread calling handleAsyncUpdate() as soon as it can.

    @tags{Events}
*/
class JUCE_API  AsyncUpdater
{
public:
    //==============================================================================
    /** Creates an AsyncUpdater object. */
    AsyncUpdater();

    /** Destructor.
        If there are any pending callbacks when the object is deleted, these are lost.
    */
    virtual ~AsyncUpdater();

    //==============================================================================
    /** Causes the callback to be triggered at a later time.

        This method returns immediately, after which a callback to the
        handleAsyncUpdate() method will be made by the message thread as
        soon as possible.

        If an update callback is already pending but hasn't happened yet, calling
        this method will have no effect.

        It's thread-safe to call this method from any thread, BUT beware of calling
        it from a real-time (e.g. audio) thread, because it involves posting a message
        to the system queue, which means it may block (and in general will do on
        most OSes).
    */
    void triggerAsyncUpdate();

    /** This will stop any pending updates from happening.

        If called after triggerAsyncUpdate() and before the handleAsyncUpdate()
        callback happens, this will cancel the handleAsyncUpdate() callback.

        Note that this method simply cancels the next callback - if a callback is already
        in progress on a different thread, this won't block until the callback finishes, so
        there's no guarantee that the callback isn't still running when the method returns.
    */
    void cancelPendingUpdate() noexcept;

    /** If an update has been triggered and is pending, this will invoke it
        synchronously.

        Use this as a kind of "flush" operation - if an update is pending, the
        handleAsyncUpdate() method will be called immediately; if no update is
        pending, then nothing will be done.

        Because this may invoke the callback, this method must only be called on
        the main event thread.
    */
    void handleUpdateNowIfNeeded();

    /** Returns true if there's an update callback in the pipeline. */
    bool isUpdatePending() const noexcept;

    //==============================================================================
    /** Called back to do whatever your class needs to do.

        This method is called by the message thread at the next convenient time
        after the triggerAsyncUpdate() method has been called.
    */
    virtual void handleAsyncUpdate() = 0;

private:
    //==============================================================================
    class AsyncUpdaterMessage;
    friend class ReferenceCountedObjectPtr<AsyncUpdaterMessage>;
    ReferenceCountedObjectPtr<AsyncUpdaterMessage> activeMessage;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (AsyncUpdater)
};

} // namespace juce
```

```
    Basically, one or more calls to the triggerAsyncUpdate() will result in the
    message thread calling handleAsyncUpdate() as soon as it can.

    @tags{Events}
*/
class RealTimeAsyncUpdater
{
public:
    //==============================================================================
    /** Creates a RealTimeAsyncUpdater object. */
    RealTimeAsyncUpdater();

    /** Destructor.
        If there are any pending callbacks when the object is deleted, these are lost.
    */
    virtual ~RealTimeAsyncUpdater();

    //==============================================================================
    /** Causes the callback to be triggered at a later time.

        This method returns immediately, after which a callback to the
        handleAsyncUpdate() method will be made by the message thread as
        soon as possible.

        If an update callback is already pending but hasn't happened yet, calling
        this method will have no effect.

        It's thread-safe to call this method from any thread.
    */
    void triggerAsyncUpdate();

    /** This will stop any pending updates from happening.

        If called after triggerAsyncUpdate() and before the handleAsyncUpdate()
        callback happens, this will cancel the handleAsyncUpdate() callback.

        Note that this method simply cancels the next callback - if a callback is already
        in progress on a different thread, this won't block until the callback finishes, so
        there's no guarantee that the callback isn't still running when the method returns.
    */
    void cancelPendingUpdate() noexcept;

    /** If an update has been triggered and is pending, this will invoke it
        synchronously.

        Use this as a kind of "flush" operation - if an update is pending, the
        handleAsyncUpdate() method will be called immediately; if no update is
        pending, then nothing will be done.

        Because this may invoke the callback, this method must only be called on
        the main event thread.
    */
    void handleUpdateNowIfNeeded();

    /** Returns true if there's an update callback in the pipeline. */
    bool isUpdatePending() const noexcept;

    //==============================================================================
    /** Called back to do whatever your class needs to do.

        This method is called by the message thread at the next convenient time
        after the triggerAsyncUpdate() method has been called.
    */
    virtual void handleAsyncUpdate() = 0;

private:
    //==============================================================================
    class RealTimeAsyncUpdateDispatcher;
    class RealTimeAsyncUpdaterMessage;
    friend class ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage>;
    ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage> activeMessage;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RealTimeAsyncUpdater)
};
```

```
    message thread calling handleAsyncUpdate() as soon as it can.

    @tags{Events}
*/
class JUCE_API  AsyncUpdater
{
public:
    //==============================================================================
    /** Creates an AsyncUpdater object. */
    AsyncUpdater();

    /** Destructor.
        If there are any pending callbacks when the object is deleted, these are lost.
    */
    virtual ~AsyncUpdater();

    //==============================================================================
    /** Causes the callback to be triggered at a later time.

        This method returns immediately, after which a callback to the
        handleAsyncUpdate() method will be made by the message thread as
        soon as possible.

        If an update callback is already pending but hasn't happened yet, calling
        this method will have no effect.

        It's thread-safe to call this method from any thread, BUT beware of calling
        it from a real-time (e.g. audio) thread, because it involves posting a message
        to the system queue, which means it may block (and in general will do on
        most OSes).
    */
    void triggerAsyncUpdate();

    /** This will stop any pending updates from happening.

        If called after triggerAsyncUpdate() and before the handleAsyncUpdate()
        callback happens, this will cancel the handleAsyncUpdate() callback.

        Note that this method simply cancels the next callback - if a callback is already
        in progress on a different thread, this won't block until the callback finishes, so
        there's no guarantee that the callback isn't still running when the method returns.
    */
    void cancelPendingUpdate() noexcept;

    /** If an update has been triggered and is pending, this will invoke it
        synchronously.

        Use this as a kind of "flush" operation - if an update is pending, the
        handleAsyncUpdate() method will be called immediately; if no update is
        pending, then nothing will be done.

        Because this may invoke the callback, this method must only be called on
        the main event thread.
    */
    void handleUpdateNowIfNeeded();

    /** Returns true if there's an update callback in the pipeline. */
    bool isUpdatePending() const noexcept;

    //==============================================================================
    /** Called back to do whatever your class needs to do.

        This method is called by the message thread at the next convenient time
        after the triggerAsyncUpdate() method has been called.
    */
    virtual void handleAsyncUpdate() = 0;

private:
    //
    class AsyncUpdaterMessage;
    friend class ReferenceCountedObjectPtr<AsyncUpdaterMessage>;
    ReferenceCountedObjectPtr<AsyncUpdaterMessage> activeMessage;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (AsyncUpdater)
};

} // namespace juce
```

```
    Basically, one or more calls to the triggerAsyncUpdate() will result in the
    message thread calling handleAsyncUpdate() as soon as it can.

    @tags{Events}
*/
class RealTimeAsyncUpdater
{
public:
    //==============================================================================
    /** Creates a RealTimeAsyncUpdater object. */
    RealTimeAsyncUpdater();

    /** Destructor.
        If there are any pending callbacks when the object is deleted, these are lost.
    */
    virtual ~RealTimeAsyncUpdater();

    //==============================================================================
    /** Causes the callback to be triggered at a later time.

        This method returns immediately, after which a callback to the
        handleAsyncUpdate() method will be made by the message thread as
        soon as possible.

        If an update callback is already pending but hasn't happened yet, calling
        this method will have no effect.

        It's thread-safe to call this method from any thread.
    */
    void triggerAsyncUpdate();

    /** This will stop any pending updates from happening.

        If called after triggerAsyncUpdate() and before the handleAsyncUpdate()
        callback happens, this will cancel the handleAsyncUpdate() callback.

        Note that this method simply cancels the next callback - if a callback is already
        in progress on a different thread, this won't block until the callback finishes, so
        there's no guarantee that the callback isn't still running when the method returns.
    */
    void cancelPendingUpdate() noexcept;

    /** If an update has been triggered and is pending, this will invoke it
        synchronously.

        Use this as a kind of "flush" operation - if an update is pending, the
        handleAsyncUpdate() method will be called immediately; if no update is
        pending, then nothing will be done.

        Because this may invoke the callback, this method must only be called on
        the main event thread.
    */
    void handleUpdateNowIfNeeded();

    /** Returns true if there's an update callback in the pipeline. */
    bool isUpdatePending() const noexcept;

    //==============================================================================
    /** Called back to do whatever your class needs to do.

        This method is called by the message thread at the next convenient time
        after the triggerAsyncUpdate() method has been called.
    */
    virtual void handleAsyncUpdate() = 0;

private:
    //
    class RealTimeAsyncUpdateDispatcher;
    class RealTimeAsyncUpdaterMessage;
    friend class ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage>;
    ReferenceCountedObjectPtr<RealTimeAsyncUpdaterMessage> activeMessage;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RealTimeAsyncUpdater)
};
```

juce::AsyncUpdater

```cpp
private:
    //==============================================================================
    class AsyncUpdaterMessage;
    friend class ReferenceCountedObjectPtr<AsyncUpdaterMessage>;
    ReferenceCountedObjectPtr<AsyncUpdaterMessage> activeMessage;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (AsyncUpdater)
};
```