```
//==============================================================================
/**
    An individual Benchmark.
    To measure a benchmark, simply create one of these with a valid description
    then before the code you are measuring call start and stop afterwards.
    Once you've done that, call getResult() to return the duration the benchmark took to run.

    To collect a set of BenchmarkResults see @BenchmarkList
*/
class Benchmark
{
public:
    /** Creates a Benchmark for a given BenchmarkDescription. */
    Benchmark (BenchmarkDescription desc)
        : description (std::move (desc))
    {
    }

    /** Starts timing the benchmark. */
    void start()
    {
        measurement.start();
    }

    /** Stops timing the benchmark. */
    void stop()
    {
        measurement.stop();
    }

    /** Returns the timing results. */
    BenchmarkResult getResult() const
    {
        return createBenchmarkResult (description, measurement.getStatistics());
    }

private:
    BenchmarkDescription description;
    tracktion::graph::PerformanceMeasurement measurement { {}, -1, false };
};
```

```
/** Describes a benchmark.
    These fields will be used to sort and group your benchmarks for comparison over time.
*/
struct BenchmarkDescription
{
    size_t hash = 0;            /**< A hash uniquely identifying this benchmark. */
    std::string category;       /**< A category for grouping. */
    std::string name;           /**< A human-readable name for the benchmark. */
    std::string description;    /**< An optional description that might include configs etc. */
    std::string platform { juce::SystemStats::getOperatingSystemName().toStdString() };
};
```

```
/** Holds the duration a benchmark took to run. */
struct BenchmarkResult
{
    BenchmarkDescription description;
    MeasurementResults metrics;
    juce::Time date { juce::Time::getCurrentTime() };
};
```

26

```cpp
/** Describes a benchmark.
    These fields will be used to sort and group your benchmarks for comparison over time.
*/
struct BenchmarkDescription
{
    size_t hash = 0;            /**< A hash uniquely identifying this benchmark. */
    std::string category;       /**< A category for grouping. */
    std::string name;           /**< A human-readable name for the benchmark. */
    std::string description;    /**< An optional description that might include configs etc. */
    std::string platform { juce::SystemStats::getOperatingSystemName().toStdString() };
};
```

```cpp
/** Holds the duration a benchmark took to run. */
struct BenchmarkResult
{
    BenchmarkDescription description;
    MeasurementResults metrics;
    juce::Time date { juce::Time::getCurrentTime() };
};
```

```cpp
//==============================================================================
/**
    An individual Benchmark.
    To measure a benchmark, simply create one of these with a valid description
    then before the code you are measuring call start and stop afterwards.
    Once you've done that, call getResult() to return the duration the benchmark took to run.

    To collect a set of BenchmarkResults see @BenchmarkList
*/
class Benchmark
{
public:
    /** Creates a Benchmark for a given BenchmarkDescription. */
    Benchmark (BenchmarkDescription desc)
        : description (std::move (desc))
    {
    }

    /** Starts timing the benchmark. */
    void start()
    {
        measurement.start();
    }

    /** Stops timing the benchmark. */
    void stop()
    {
        measurement.stop();
    }

    /** Returns the timing results. */
    BenchmarkResult getResult() const
    {
        return createBenchmarkResult (description, measurement.getStatistics());
    }

private:
    BenchmarkDescription description;
    tracktion::graph::PerformanceMeasurement measurement { {}, -1, false };
};
```

```
class ResamplingBenchmarks : public juce::UnitTest
{
public:
    ResamplingBenchmarks()
        : juce::UnitTest ("Resampling Benchmarks", "tracktion_benchmarks")
    {
    }

    void runTest() override
    {
        runResamplingRendering ("lagrange",     ResamplingQuality::lagrange);
        runResamplingRendering ("sincFast",     ResamplingQuality::sincFast);
        runResamplingRendering ("sincMedium",   ResamplingQuality::sincMedium);
        runResamplingRendering ("sincBest",     ResamplingQuality::sincBest);
    }

private:
    //==============================================================================
    //==============================================================================
    void runResamplingRendering (juce::String qualityName,
                                 ResamplingQuality quality)
    {
        auto& engine = *Engine::getEngines()[0];
        auto edit = Edit::createSingleTrackEdit (engine);
        edit->ensureNumberOfAudioTracks (1);
        auto t = getAudioTracks (*edit)[0];

        const auto durationOfFile = 30s;
        auto sinFile = getSinFile<juce::WavAudioFormat> (fileSampleRate, durationOfFile, 2, 220.0f);
        const auto timeRange = TimeRange (0s, TimePosition (durationOfFile));
        auto waveClip = t->insertWaveClip (sinFile->getFile().getFileName(), sinFile->getFile(),
                                           {{ timeRange }}, false);
        waveClip->setUsesProxy (false);
        waveClip->setResamplingQuality (quality);

        {
            ScopedBenchmark sb (createBenchmarkDescription ("Resampling", "WaveNode quality", "30s sin wave, 96KHz to 44.1Khz, " + qualityName.toStdString()));
            Renderer::measureStatistics ("Rendering resampling",
                                         *edit, timeRange,
                                         toBitSet ({ t }),
                                         256, playbackSampleRate);
        }
    }
};

static ResamplingBenchmarks resamplingBenchmarks;
```