Design Document for `openJuice`

`mikomikotaishi`

General guidelines

Use the latest stable release of C++ (at the time of writing, that is C++23).

Use external libraries sparingly - and only when absolutely necessary (i.e. `<regex>` uses an inferior regular expression implementation, so opt to use `Boost.regex`, or `Boost.asio` for networking capabilities).

Build scripts

There are scripts that are used to compile the project simply:

- `quick-cmake-build.py` (Python; for simplicity and cross-compatibility)
- `QuickCMakeBuild.java` (Java with JBang; experimental)

Coding style

When contributing, make sure that you have `git pull`ed the most recent revision.

Import statements and include directives

Generally we will avoid writing `#include` as much as possible, however it is certainly impossible to completely forgo the use of this directive. Use `#include` sparingly, only when absolutely necessary. Some cases in which it is necessary include writing `#include <filesystem>` to allow aliases from `stdlib.os` to work, or using `#include <toml++/toml.hpp>` or `#include <SDL3/SDL.h>` In all other cases, simply invoke `import` instead.

With all import statements, we aim to match the Java style for organising the use of `import`. This means:

- `import` all (homemade) standard library modules (under `stdlib` first), alphabetically.
  - The exception to this rule is the module `stdlib.core`, which is always `import`ed first.
- `import` all module partitions (if any)
- `import` all internal modules necessary, alphabetically
- `import` all external modules necessary, alphabetically (if any)

For standard library modules, while you are free to `import` only the needed header within the module (i.e. `import stdlib.os.Filesystem;` instead of `import stdlib.os;`), please do so only sparingly and only if you have a very compelling reason.

Namespaces

If using all symbols from an imported namespace, simply use `using namespace` to reduce cluttering the codebase. For instance, whenever anything of the `collections` namespace is invoked, we write `using namespace collections;`.

However, as mentioned with regards to importing symbols from namespaces, there are exceptions to when to use `using namespace` and when to only write `using`. For instance, whenever invoking a smart pointer from the namespace `mem`, we prefer to write, for instance, `using mem::SharedPtr, mem::UniquePtr;` (for whichever smart pointers are needed), in an alphabetised list.

Warnings

If at any point outdated or deprecated parts of the standard library must be used, mark the area using the preprocessor directive `#warning`, followed by a descriptive and consistent warning of the issue. The `#warning` directive will display a compiler warning during compilation so that this issue can be found again and handled some time in the future.

Doxygen comments

Use Javadoc-style comments. Comment all classes, methods, fields, enums, namespaces, namespace members, and important global variables/constants. Obviously, this does not mean to write copius amounts of documentation where it is unnecessary - so long as comments something is not obvious, it should be commented.

Classes and class relations (Inheritance)

Indicate any important details about a class (such as its purpose, whether it is singleton, etc.) Specify whether a class is concrete, or if it is abstract. See below on guidelines:

Use `@class` to denote a concrete class, a utility class (in the sense of Java), or an abstract class that has fields. Use `@interface` to denote an abstract class without fields (an "interface"). (Even though an interface, strictly speaking, does not have constructors, we will continue to count an abstract class with only methods as an interface.)

Use `@extends` to indicate that a concrete class inherits from another concrete class, or to indicate that an abstract class (an "interface") inherits from another abstract class. Use `@implements` to indicate that a concrete class inherits from an abstract class.

Refer to any class that is abstract (or should be abstract) as an "interface" in Doxygen comments (although C++ does not offer "interfaces" in the strictest sense, this is done for clarity). Only refer to a concrete class as a "class" in Doxygen comments.

When applicable, create a "marker interface" with no methods (besides a `protected` constructor and `public virtual` destructor) that is used solely to denote that a class implements some behaviour. Such marker interfaces belong in `engine.utility.Interfaces:*`.

Namespaces

Indicate the purpose of a namespace.

Avoid using utility classes (in the sense that they exist in Java - a static class with static methods with a private constructor) in favour of using namespaces, unless there is a truly compelling reason to use a utility class.

Standard library modules

Indicate the header that the standard library module is associated with, and indicate the purpose of the grouping of standard library header modules.

Module naming scheme

This project follows a Java-style convention of naming modules, with the primary difference of omitting the top level domain and project name. In other words, the module should be named by its path relative to the project root delimited by `.` (to denote hierarchy), with the final word in the module name being the name of the file, in PascalCase. The name of the file itself should be in PascalCase.

Standard library modules

This project has decided not to use the C++ standard library modules, simply because at the current time of writing they are not widely available, and have some questionable design choices (namely only allowing the option to be imported in their entirety). As such, we have decided to write standard library modules of our own that wrap the existing standard library headers into logical groupings and namespaces.

The naming convention continues the module naming convention - a header `X` part of collection `x` should be named `stdlib.x.Y` (where `Y` should be in PascalCase), and the module that exports all headers belonging to `x` will be named `stdlib.x` (in lowercase).

The option to override the default name of the standard library modules is offered by defining the macro `NO_RESERVED_STD`, however the codebase does not use this for future compatibility.

The standard library modules provided are divided into their own directories, and each exported by another module that imports them into a collection of header modules, such as `stdlib.core`, `stdlib.math`, `stdlib.os`, etc. This is to simplify the number of code packages needed to be imported into a file. At the end of these, we include `export using namespace stdlib;` for convenience.

There are instances in which both the header and the module must be imported due to some issues with the existing code. If so, then do as necessary.

If a header that is necessary is does not have an associated standard library module, please write it yourself in a logical directory and namespace, and import it to whatever module should export it.