Object lifetime and ownership

- Absolutely do NOT use `delete`, `deleteAndZero`, etc. There are very very few situations where you can't use a smart pointer or some other automatic lifetime management class.
- Do not use `new` unless there's no alternative. Whenever you type `new`, always treat it as a failure to find a better solution. If a local variable can be allocated on the stack rather than the heap, then always do so.
- Never use `new` or malloc to allocate a C++ array. Always prefer a juce::HeapBlock or some other container class.
- ..and just to make it doubly clear: You should (almost) never need to use malloc or calloc at all!
- If a parent class needs to create and own some kind of child object, always use composition as your first choice. If that's not possible (e.g. if the child needs a pointer to the parent for its constructor), then use a ScopedPointer or std::unique_ptr. Whenever possible, pass an object as a reference rather than a pointer. If possible, make it a const reference.
- Obviously avoid static and global variables. Sometimes there's no alternative, but if there is an alternative, then use it, no matter how much effort it involves.
- If allocating a local POD structure (e.g. an operating-system structure in native code), and you need to initialise it with zeros, use the `= {};` syntax as your first choice for doing this. If for some reason that's not appropriate, use the zerostruct() function, or in case that isn't suitable, use zeromem(). Avoid memset().
- Treat Component::deleteAllChildren() as a last resort – never use it if there's a cost-free alternative.
- The juce::ScopedPointer class was written to be compatible with pre-C++11 compilers, so although it does offer C++11 move functionality for supported compilers, it's not as versatile as std::unique_ptr. So if you can use std::unique_ptr in your own code, that's probably a better bet. We may eventually migrate the JUCE codebase to std::unique_ptr.
- When returning heap objects from functions, most of the JUCE codebase predates the modern C++ style of returning a std::unique_ptr to indicate that the caller takes ownership. We will at some point be moving to that style, but in the meantime, functions are annotated to make clear how ownership is passed around.