Intro to value lifecycle

So far, we've explained how Mojo allows you to build high-performance code that is memory safe *without* manually managing memory, using Mojo's ownership model. However, Mojo is designed for systems programming, which often requires manual memory management for custom data types. So, Mojo lets you do that as you see fit. To be clear, Mojo has no reference counter and no garbage collector.

Mojo also has no built-in data types with special privileges. All data types in the standard library (such as `Bool`, `Int`, and `String`) are implemented as structs. You can actually write your own replacements for these types by using low-level primitives provided by MLIR dialects.

What's great about the Mojo language is that it provides you these low-level tools for systems programming, but within a framework that helps you build things that are safe and easy to use from higher-level programs. That is, you can get under the hood and write all the "unsafe" code you want, but as long as you do so in accordance with Mojo's value semantics, the programmer instantiating your type/object doesn't need to think about memory management at all, and the behavior will be safe and predictable, thanks to value ownership.

In summary, it's the responsibility of the type author to manage the memory and resources for each value type, by implementing specific lifecycle methods, such as the constructor, copy constructor, move constructor, and destructor, as necessary. Mojo doesn't create any constructors by default, although it does add a trivial, no-op destructor for types that don't define their own.

In the following pages, we'll explain exactly how to define these lifecycle methods in accordance with value semantics so your types play nicely with value ownership.

Lifecycles and lifetimes

First, let's clarify some terminology:

- The "lifecycle" of a value is defined by various dunder methods in a struct. Each lifecycle event is handled by a different method, such as the constructor (`__init__()`), the destructor (`__del__()`), the copy constructor (`__copyinit__()`), and the move constructor (`__moveinit__()`). All values that are declared with the same type have the same lifecycle.
- The "lifetime" of a value is defined by the span of time during program execution in which each value is considered valid. The life of a value begins when it is initialized (via `__init__()`, `__copyinit__()` or `__moveinit__()`) and ends when it is destroyed (`__del__()`), or consumed in some other way (for example, as part of a `__moveinit__()` call).

No two values have the exact same life span, because every value is created and destroyed at a different point in time (even if the difference is imperceptible).

> **Lifetime type**
>
> The concept of lifetimes is related to the `lifetime` type, a Mojo primitive used to track ownership. For most Mojo programming, you won't need to work with `lifetime` values directly. For information, see Lifetimes and references.

The life of a value in Mojo begins when a variable is initialized and continues up until the value is last used, at which point Mojo destroys it. Mojo destroys every value/object as soon as it's no longer used, using an "as soon as possible" (ASAP) destruction policy that runs after every sub-expression. The Mojo compiler takes care of releasing resources after last use when needed.

As you might imagine, keeping track of a value's life can be difficult if a value is shared across functions many times during the life of a program. However, Mojo makes this predictable partly through its value semantics and value ownership (both prerequisite readings for the following sections). The final piece of the puzzle for lifetime management is the value lifecycle: every value (defined in a struct) needs to implement key lifecycle methods that define how a value is created and destroyed.