# Death of a value

As soon as a value/object is no longer used, Mojo destroys it. Mojo does *not* wait until the end of a code block—or even until the end of an expression—to destroy an unused value. It destroys values using an “as soon as possible” (ASAP) destruction policy that runs after every sub-expression. Even within an expression like `a+b+c+d`, Mojo destroys the intermediate values as soon as they're no longer needed.

Mojo uses static compiler analysis to find the point where a value is last used. Then, Mojo immediately ends the value's lifetime and calls the `__del__()` destructor to perform any necessary cleanup for the type.

For example, notice when the `__del__()` destructor is called for each instance of `MyPet`:

```
@value
struct MyPet:
    var name: String
    var age: Int

    fn __del__(owned self):
        print("Destruct", self.name)

fn pets():
    var a = MyPet("Loki", 4)
    var b = MyPet("Sylvie", 2)
    print(a.name)
    # a.__del__() runs here for "Loki"

    a = MyPet("Charlie", 8)
    # a.__del__() runs immediately because "Charlie" is never used

    print(b.name)
    # b.__del__() runs here

pets()
```

```
Loki
Destruct Loki
Destruct Charlie
Sylvie
Destruct Sylvie
```

Notice that each initialization of a value is matched with a call to the destructor, and `a` is actually destroyed multiple times—once for each time it receives a new value.

Also notice that this `__del__()` implementation doesn't actually do anything. Most structs don't require a custom destructor, and Mojo automatically adds a no-op destructor if you don't define one.

## Default destruction behavior

You may be wondering how Mojo can destroy a type without a custom destructor, or why a no-op destructor is useful. If a type is simply a collection of fields, like the `MyPet` example, Mojo only needs to destroy the fields: `MyPet` doesn't dynamically allocate memory or use any long-lived resources (like file handles). There's no special action to take when a `MyPet` value is destroyed.

Looking at the individual fields, `MyPet` includes an `Int` and a `String`. The `Int` is what Mojo calls a *trivial type*. It's a statically-sized bundle of bits. Mojo knows exactly how big it is, so those bits can be reused to store something else.

The `String` value is a little more complicated. Mojo strings are mutable. The `String` object has an internal buffer—a `List` field, which holds the characters that make up the string. A `List` stores its contents in dynamically allocated memory on the heap, so the string can grow or shrink. The string itself doesn't have any special destructor logic, but when Mojo destroys a string, it calls the destructor for the `List` field, which de-allocates the memory.

Since `String` and `Int` don't require any custom destructor logic, they both have no-op destructors: literally, `__del__()` methods that don't do anything. This may seem pointless, but it means that Mojo can call the destructor on any value when its lifetime ends. This makes it easier to write generic containers and algorithms.

## Benefits of ASAP destruction

Similar to other languages, Mojo follows the principle that objects/values acquire resources in a constructor (`__init__()`) and release resources in a destructor (`__del__()`). However, Mojo's ASAP destruction has some advantages over scope-based destruction (such as the C++ RAII pattern, which waits until the end of the code scope to destroy values):

- Destroying values immediately at last-use composes nicely with the "move" optimization, which transforms a "copy+del" pair into a "move" operation.
- Destroying values at end-of-scope in C++ is problematic for some common patterns like tail recursion, because the destructor call happens after the tail call. This can be a significant performance and memory problem for certain functional programming patterns, which is not a problem in Mojo, because the destructor call always happens before the tail call.

Additionally, Mojo's ASAP destruction works great within Python-style `def` functions. That's because Python doesn't really provide scopes beyond a function scope, so the Python garbage collector cleans up resources more often than a scope-based destruction policy would. However, Mojo does not use a garbage collector, so the ASAP destruction policy provides destruction guarantees that are even more fine-grained than in Python.

The Mojo destruction policy is more similar to how Rust and Swift work, because they both have strong value ownership tracking and provide memory safety. One difference is that Rust and Swift require the use of a dynamic "drop flag"—they maintain hidden shadow variables to keep track of the state of your values to provide safety. These are often optimized away, but the Mojo approach eliminates this overhead entirely, making the generated code faster and avoiding ambiguity.

## Destructor

Mojo calls a value's destructor (`__del__()` method) when the value's lifetime ends (typically the point at which the value is last used). As we mentioned earlier, Mojo provides a default, no-op destructor for all types, so in most cases you don't need to define the `__del__()` method.

You should define the `__del__()` method to perform any kind of cleanup the type requires. Usually, that includes freeing memory for any fields where you dynamically allocated memory (for example, via `UnsafePointer`) and closing any long-lived resources such as file handles.

However, any struct that is just a simple collection of other types does not need to implement the destructor.

For example, consider this simple struct:

```
struct MyPet:
    var name: String
    var age: Int

    fn __init__(inout self, name: String, age: Int):
        self.name = name
        self.age = age
```

There's no need to define the `__del__()` destructor for this, because it's a simple collection of other types (`String` and `Int`), and it doesn't dynamically allocate memory.

Whereas, the following struct must define the `__del__()` method to free the memory allocated by its `UnsafePointer`:

```
from memory.unsafe_pointer import UnsafePointer

struct HeapArray:
    var data: UnsafePointer[Int]
    var size: Int

    fn __init__(inout self, size: Int, val: Int):
        self.size = size
        self.data = UnsafePointer[Int].alloc(self.size)
        for i in range(self.size):
            (self.data + i).init_pointee_copy(val)

    fn __del__(owned self):
        for i in range(self.size):
            (self.data + i).destroy_pointee()
        self.data.free()
```

Note that a pointer doesn't *own* any values in the memory it points to, so when a pointer is destroyed, Mojo doesn't call the destructors on those values.

So in the `HeapArray` example above, calling `free()` on the pointer releases the memory, but doesn't call the destructors on the stored values. To invoke the destructors, use the `destroy_pointee()` method provided by the `UnsafePointer` type.

**Note**

You can't just call the destructor explicitly. Because `__del__()` takes `self` as an `owned` value, and owned arguments are copied by default, `foo.__del__()` actually creates and destroys a *copy* of `foo`. When Mojo destroys a value, however, it passes in the original value as `self`, not a copy.

It's important to notice that the `__del__()` method is an "extra" cleanup event, and your implementation does not override any default destruction behaviors. For example, Mojo still destroys all the fields in `MyPet` even if you implement `__del__()` to do nothing:

```
struct MyPet:
    var name: String
    var age: Int

    fn __init__(inout self, name: String, age: Int):
        self.name = name
        self.age = age

    fn __del__(owned self):
        # Mojo destroys all the fields when they're last used
        pass
```

However, the `self` value inside the `__del__()` destructor is still whole (so all fields are still usable) until the destructor returns, as we'll discuss more in the following section.

## Field lifetimes

In addition to tracking the lifetime of all objects in a program, Mojo also tracks each field of a structure independently. That is, Mojo keeps track of whether a "whole object" is fully or partially initialized/destroyed, and it destroys each field independently with its ASAP destruction policy.

For example, consider this code that changes the value of a field:

```
@value
struct MyPet:
    var name: String
    var age: Int

fn use_two_strings():
    var pet = MyPet("Po", 8)
    print(pet.name)
    # pet.name.__del__() runs here, because this instance is
    # no longer used; it's replaced below

    pet.name = String("Lola") # Overwrite pet.name
    print(pet.name)
    # pet.__del__() runs here
```

The `pet.name` field is destroyed after the first `print()`, because Mojo knows that it will be overwritten below. You can also see this behavior when using the transfer sigil:

```
fn consume(owned arg: String):
    pass

fn use(arg: MyPet):
    print(arg.name)

fn consume_and_use():
    var pet = MyPet("Selma", 5)
    consume(pet.name^)
    # pet.name.__moveinit__() runs here, which destroys pet.name
    # Now pet is only partially initialized

    # use(pet)  # This fails because pet.name is uninitialized

    pet.name = String("Jasper")  # All together now
    use(pet)                     # This is ok
    # pet.__del__() runs here (and only if the object is whole)
```

Notice that the code transfers ownership of the `name` field to `consume()`. For a period of time after that, the `name` field is uninitialized. Then `name` is reinitialized before it is passed to the `use()` function. If you try calling `use()` before `name` is re-initialized, Mojo rejects the code with an uninitialized field error.

Also, if you don't re-initialize the name by the end of the `pet` lifetime, the compiler complains because it's unable to destroy a partially initialized object.

Mojo's policy here is powerful and intentionally straight-forward: fields can be temporarily transferred, but the "whole object" must be constructed with the aggregate type’s initializer and destroyed with the aggregate destructor. This means it's impossible to create an object by initializing only its fields, and it's likewise impossible to destroy an object by destroying only its fields.

## Field lifetimes during destruct and move

The consuming-move constructor and destructor face an interesting situation with field lifetimes, because, unlike other lifecycle methods, they both take an instance of their own type as an `owned` argument, which is about to be destroyed. You don't really need to worry about this detail when implementing these methods, but it might help you better understand field lifetimes.

Just to recap, the move constructor and destructor method signatures look like this:

```
struct TwoStrings:
    fn __moveinit__(inout self, owned existing: Self):
        # Initializes a new `self` by consuming the contents of `existing`
    fn __del__(owned self):
        # Destroys all resources in `self`
```

**Note**

There are two kinds of "self" here: capitalized `Self` is an alias for the current type name (used as a type specifier for the `existing` argument), whereas lowercase `self` is the argument name for the implicitly-passed reference to the current instance (also called "this" in other languages, and also implicitly a `Self` type).

Both of these methods face an interesting but obscure problem: they both must dismantle the `existing`/`self` value that's `owned`. That is, `__moveinit__()` implicitly destroys sub-elements of `existing` in order to transfer ownership to a new instance (read more about the move constructor), while `__del__()` implements the deletion logic for its `self`. As such, they both need to own and transform elements of the `owned` value, and they definitely don’t want the original `owned` value's destructor to also run—that could result in a double-free error, and in the case of the `__del__()` method, it would become an infinite loop.

To solve this problem, Mojo handles these two methods specially by assuming that their whole values are destroyed upon reaching any return from the method. This means that the whole object may be used as usual, up until the field values are transferred or the method returns.

For example, the following code works as you would expect (within the destructor, we can still pass ownership of a field value to another function, and there's no infinite loop to destroy `self`):

```
fn consume(owned str: String):
    print('Consumed', str)

struct TwoStrings:
    var str1: String
    var str2: String

    fn __init__(inout self, one: String):
        self.str1 = one
        self.str2 = String("bar")

    fn __moveinit__(inout self, owned existing: Self):
        self.str1 = existing.str1
        self.str2 = existing.str2

    fn __del__(owned self):
        self.dump() # Self is still whole here
        # Mojo calls self.str2.__del__() since str2 isn't used anymore

        consume(self.str1^)
        # self.str1 has been transferred so it is also destroyed now;
        # `self.__del__()` is not called (avoiding an infinite loop).

    fn dump(inout self):
        print('str1:', self.str1)
        print('str2:', self.str2)

fn use_two_strings():
    var two_strings = TwoStrings("foo")
```

## Explicit lifetimes

So far, we've described how Mojo destroys a value at the point it's last used, and this works great in almost all situations. However, there are very rare situations in which Mojo simply cannot predict this correctly and will destroy a value that is still referenced through some other means.

For instance, perhaps you're building a type with a field that carries a pointer to another field. The Mojo compiler won't be able to reason about the pointer, so it might destroy a field ( `obj1` ) when that field is technically no longer used, even though another field ( `obj2` ) still holds a pointer to part of it. So, you might need to keep `obj1` alive until you can execute some special logic in the destructor or move initializer.

You can force Mojo to keep a value alive up to a certain point by assigning the value to the `_` discard pattern at the point where it's okay to destroy it. For example:

```
fn __del__(owned self):
    self.dump() # Self is still whole here

    consume(self.obj2^)
    _ = self.obj1
    # Mojo keeps `obj1` alive until here, after its "last use"
```

In this case, if `consume()` refers to some value in `obj1` somehow, this ensures that Mojo does not destroy `obj1` until after the call to `consume()`, because assignment to the discard variable `_` is actually the last use.

For other situations, you can also scope the lifetime of a value using the Python-style `with` statement. That is, for any value defined at the entrance to a `with` statement, Mojo will keep that value alive until the end of the `with` statement. For example:

```
with open("my_file.txt", "r") as file:
    print(file.read())

    # Other stuff happens here (whether using `file` or not)...
    foo()
    # `file` is alive up to the end of the `with` statement.

# `file` is destroyed when the statement ends.
bar()
```