Mojo Ownership

Who owns a value, and how it passes between functions

Mojo language reference
v1.0.0b2

keyword type built-in "string" number operator @decorator True/False/None # comment

THE MODEL

Value ownership is fundamental to Mojo. Every value has exactly one owner, and how values move between owners runs through the whole language.

You encounter ownership in two situations: **variables** and **function calls**.

Variables can own or reference a value. **Argument conventions** describe how a function uses a value: read-only, reference, mutable, owned, produced, or consumed.

VAR OWNS, REF REFERS

```
var data = [1, 2, 3] # owns the list
ref view = data[0] # a 2nd name, no copy
view = 9 # writes through to data
print(data) # [9, 2, 3]
```

var always means "I own this." **ref** means "this is a view into someone else's value." A struct's **var** field owns its value and a struct type owns its fields.

VAR ASSIGNMENT

A **var** assignment uses the right-hand side's policy: it determines whether the value is materialized, constructed, copied, or transferred. A call or expression returning a value constructs; one returning a reference copies out the referenced value.

YOU WRITE	THE VAR TAKES OWNERSHIP OF
`5.0 / "Hello" / [1, 2, 3] (a literal)`	a materialized literal
`SomeType()`	a freshly constructed value
`some_value (ImplicitlyCopyable)`	an implicit copy
`some_value.copy() (Copyable)`	an explicit copy
`some_value^ (Movable)`	the source's value, transferred
`some_ref`	a copy of the referenced value

A copy doesn't change a value's ownership. Only **^** moves the value to a new owner.

ARGUMENT CONVENTIONS ON SELF

`self`	read (immutable)
`mut self`	modify the instance
`out self`	build it (in __init__())
`deinit self`	destroy the instance
`ref self`	parametric mutability

TRANSFER OWNERSHIP WITH ^

```
def exclaim(var s: String):
    s += "!"
    print(s)

var g = "Hello"
exclaim(g) # copy: g still usable
exclaim(g^) # transfer: g uninitialized
# print(g) # error: used after transfer
```

The **var** argument takes ownership of the original only with **^**; a plain call implicitly copies (**String** is **ImplicitlyCopyable**), so **g** stays usable. Either value, the copy or the transferred original, ends its lifetime after the **print** (its last use). The same **^** drains a collection in a loop: **for var x in items^** moves each element out.

CALL SITES: PASSING VALUES

YOU WRITE	INTO A VAR ARG
`f(x)`	implicit copy (**ImplicitlyCopyable** only)
`f(x.copy())`	explicit copy
`f(x^)`	transfer; x uninitialized after

A borrowing argument (**read**, **mut**, **ref**) has no **^** lever: you write **f(x)**, and it views the value in place.

ORIGINS ON REFERENCES

```
def first[T: Movable](ref xs: List[T]) -> ref[origin_of(xs)] T:
    return xs[0]

ref x = first(xs) # len(xs) known to be > 0
```

A **ref** return carries an **origin** so the compiler tracks where it points, whether it stays valid, and whether access is mutable. Values are destroyed at last use; a live **ref** keeps the value it refers to alive.

ARGUMENT CONVENTIONS: THE DECISION TABLE

CONVENTION	OWNS IT?	MUTABLE?	CALLER KEEPS IT?	REACH FOR IT WHEN
`(read)`	no	no	yes	reading a value without changing it (the default)
`mut`	no	yes	yes	changing the caller's value in place
`var`	yes (own copy)	yes	yes, unless ^	you need a local, mutable copy
`out`	becomes the value	yes	it is the result	returning by name instead of →
`deinit`	yes (consumes)	yes	no	destructors and the source of a move
`ref`	no (refers)	parametric	yes	returning or holding a reference with an origin

A convention sits before the argument name: **def f(mut x: Int)**. With no convention, an argument is a read-only borrow: a view into a value you don't own. **mut** makes it a writable view.

LITERALS

```
var i = 5 # Int, machine width (default)
var i32: Int32 = 5 # SIMD[DType.int32, 1]
```

Literals are produced by the lexer, not built by a constructor. Each compile-time type (**IntLiteral**, **FloatLiteral**, **StringLiteral**) materializes into a runtime value. By default, integer literals are **Int**, floats are **Float64**, and strings are **String**. Use type annotations for specific types like **Byte** (**UInt8**), **Int16**, or **BFloat16**.

TRIVIAL VALUES

Trivial register types (**Int**, **Float64**, **SIMD**) are **ImplicitlyCopyable** with no destructor. A copy is a register copy; **^** is a no-op (the compiler warns transfer has no effect); there's nothing to destroy. The rules still apply, they just compile to register moves or nothing.

MUTABILITY

Values aren't mutable or immutable. Access is.

`var`	always mutable
`ref`	inherits the mutability of what it refers to

LIFECYCLE METHODS

`__init__(out self, …)`	construct
`__init__(out self, *, copy: Self)`	copy
`__init__(out self, *, deinit move: Self)`	move
`__del__(deinit self)`	destroy

Copy, move, and destructors can't raise. The var assignment table shows which one each assignment runs.

NON-OWNING VIEWS

```
var data = ["a", "b", "c", "d", "e"]
var s = data[1:3] # a Span view, no copy: [b, c]
s[0] = "X" # writes through: data is [a, X, c, d, e]

var text = "Hello, World!"
var hi = text[codepoint=0:5] # a StringSlice view: "Hello"
```

A view is a non-owning window into a buffer someone else owns. **Span** views contiguous elements; **StringSlice** views UTF-8 text. Like **ref**, a view carries an origin, so the compiler keeps the source alive and tracks whether the view stays valid.

RETURNS: HANDING A VALUE OUT

YOU WRITE	WHAT IT DOES
`return x`	copy out (when **ImplicitlyCopyable**)
`return x.copy()`	copy out
`return x^`	transfer out
`-> T`	return a value
`-> ref[origin] T`	return a reference

Like a var assignment, **return x** copies; when **x** is at its last use the compiler moves it instead (you own it, so it can be moved).

No **-> T^**: the **^** goes on the returned value in **return x^**, not on the return type.