

# Structs

A Mojo struct is a data structure that allows you to encapsulate fields and methods that operate on an abstraction, such as a data type or an object. **Fields** are variables that hold data relevant to the struct, and **methods** are functions inside a struct that generally act upon the field data.

For example, if you're building a graphics program, you can use a struct to define an `Image` that has fields to store information about each image (such as the pixels) and methods that perform actions on it (such as rotate it).

For the most part, Mojo's struct format is designed to provide a static, memory-safe data structure for high-level data types used in programs. For example, all the data types in Mojo's standard library (such as `Int`, `Bool`, `String`, and `Tuple`) are defined as structs.

If you understand how [functions](#) and [variables](#) work in Mojo, you probably noticed that Mojo is designed to provide dynamic programming features in a `def` function while enforcing stronger code safety in `fn` functions. When it comes to structs, Mojo leans toward the safe side: You can still choose whether to use either `def` or `fn` declarations for methods, but all fields must be declared with `var`.

## Struct definition

You can define a simple struct called `MyPair` with two fields like this:

```
struct MyPair:
  var first: Int
  var second: Int
```

However, you can't instantiate this struct because it has no constructor method. So here it is with a constructor to initialize the two fields:

```
struct MyPair:
  var first: Int
  var second: Int

  fn __init__(inout self, first: Int, second: Int):
    self.first = first
    self.second = second
```

Notice that the first argument in the `__init__()` method is `inout self`. For now, ignore `inout` (it's an [argument convention](#) that declares `self` as a mutable reference); all you need to know right now is that `self` must be the first argument. It references the current struct instance (it allows code in the method to refer to "itself"). *When you call the constructor, you never pass a value for `self`—Mojo passes it in automatically.*

The `__init__()` method is one of many [special methods](#) (also known as "dunder methods" because they have double *underscores*) with pre-determined names.

### Note

You can't assign values when you declare fields. You must initialize all of the struct's fields in the constructor. (If you try to leave a field uninitialized, the code won't compile.)

Once you have a constructor, you can create an instance of `MyPair` and set the fields:

```
var mine = MyPair(2,4)
print(mine.first)
```

2

## Methods

In addition to special methods like `__init__()`, you can add any other method you want to your struct. For example:

```
struct MyPair:
  var first: Int
  var second: Int

  fn __init__(inout self, first: Int, second: Int):
    self.first = first
    self.second = second

  fn get_sum(self) -> Int:
    return self.first + self.second
```

```
var mine = MyPair(6, 8)
print(mine.get_sum())
```

Notice that `get_sum()` also uses the `self` argument, because this is the only way you can access the struct's fields in a method. The name `self` is just a convention, and you can use any name you want to refer to the struct instance that is always passed as the first argument.

Methods that take the implicit `self` argument are called *instance methods* because they act on an instance of the struct.

### Note

The `self` argument in a struct method is the only argument in an `fn` function that does not require a type. You can include it if you want, but you can elide it because Mojo already knows its type (`MyPair` in this case).

## Static methods

A struct can also have *static methods*. A static method can be called without creating an instance of the struct. Unlike instance methods, a static method doesn't receive the implicit `self` argument, so it can't access any fields on the struct.

To declare a static method, use the `@staticmethod` decorator and don't include a `self` argument:

```
struct Logger:

    fn __init__(inout self):
        pass

    @staticmethod
    fn log_info(message: String):
        print("Info: ", message)
```

You can invoke a static method by calling it on the type (in this case, `Logger`). You can also call it on an instance of the type. Both forms are shown below:

```
Logger.log_info("Static method called.")
var l = Logger()
l.log_info("Static method called from instance.")
```

```
Info: Static method called.
Info: Static method called from instance.
```

# Structs compared to classes

If you're familiar with other object-oriented languages, then structs might sound a lot like classes, and there are some similarities, but also some important differences. Eventually, Mojo will also support classes to match the behavior of Python classes.

So, let's compare Mojo structs to Python classes. They both support methods, fields, operator overloading, decorators for metaprogramming, and more, but their key differences are as follows:

- Python classes are dynamic: they allow for dynamic dispatch, monkey-patching (or “swizzling”), and dynamically binding instance fields at runtime.
- Mojo structs are static: they are bound at compile-time (you cannot add methods at runtime). Structs allow you to trade flexibility for performance while being safe and easy to use.
- Mojo structs do not support inheritance (“sub-classing”), but a struct can implement [traits](#).
- Python classes support class attributes—values that are shared by all instances of the class, equivalent to class variables or static data members in other languages.
- Mojo structs don't support static data members.

Syntactically, the biggest difference compared to a Python class is that all fields in a struct must be explicitly declared with `var`.

In Mojo, the structure and contents of a struct are set at compile time and can't be changed while the program is running. Unlike in Python, where you can add, remove, or change attributes of an object on the fly, Mojo doesn't allow that for structs.

However, the static nature of structs helps Mojo run your code faster. The program knows exactly where to find the struct's information and how to use it without any extra steps or delays at runtime.

Mojo's structs also work really well with features you might already know from Python, like operator overloading (which lets you change how math symbols like `+` and `-` work with your own data, using [special methods](#)).

As mentioned above, all Mojo's standard types (`Int`, `String`, etc.) are made using structs, rather than being hardwired into the language itself. This gives you more flexibility and control when writing your code, and it means you can define your own types with all the same capabilities (there's no special treatment for the standard library types).

## Special methods

Special methods (or “dunder methods”) such as `__init__()` are pre-determined method names that you can define in a struct to perform a special task.

Although it's possible to call special methods with their method names, the point is that you never should, because Mojo automatically invokes them in circumstances where they're needed (which is why they're also called "magic methods"). For example, Mojo calls the `__init__()` method when you create an instance of the struct; and when Mojo destroys the instance, it calls the `__del__()` method (if it exists).

Even operator behaviors that appear built-in (`+`, `<`, `==`, `|`, and so on) are implemented as special methods that Mojo implicitly calls upon to perform operations or comparisons on the type that the operator is applied to.

Mojo supports a long list of special methods; far too many to discuss here, but they generally match all of [Python's special methods](#) and they usually accomplish one of two types of tasks:

- Operator overloading: A lot of special methods are designed to overload operators such as `<` (less-than), `+` (add), and `|` (or) so they work appropriately with each type. For example, look at the methods listed for Mojo's [Int type](#). One such method is `__lt__()`, which Mojo calls to perform a less-than comparison between two integers (for example, `num1 < num2`).
- Lifecycle event handling: These special methods deal with the lifecycle and value ownership of an instance. For example, `__init__()` and `__del__()` demarcate the beginning and end of an instance lifetime, and other special methods define the behavior for other lifecycle events such as how to copy or move a value.

You can learn all about the lifecycle special methods in the [Value lifecycle](#) section. However, most structs are simple aggregations of other types, so unless your type requires custom behaviors when an instance is created, copied, moved, or destroyed, you can synthesize the essential lifecycle methods you need (and save yourself some time) by adding the `@value` decorator.

## @value decorator

When you add the [@value decorator](#) to a struct, Mojo will synthesize the essential lifecycle methods so your object provides full value semantics. Specifically, it generates the `__init__()`, `__copyinit__()`, and `__moveinit__()` methods, which allow you to construct, copy, and move your struct type in a manner that's value semantic and compatible with Mojo's ownership model.

For example:

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

Mojo will notice that you don't have a member-wise initializer, a move constructor, or a copy constructor, and it will synthesize these for you as if you had written:

```
struct MyPet:
    var name: String
```

```
var age: Int
```

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

```
fn __copyinit__(inout self, existing: Self):  
    self.name = existing.name  
    self.age = existing.age
```

```
fn __moveinit__(inout self, owned existing: Self):  
    self.name = existing.name^  
    self.age = existing.age
```

Without the copy and move constructors, the following code would not work because Mojo would not know how to copy an instance of `MyPet`:


```
var dog = MyPet("Charlie", 5)  
var poodle = dog  
print(poodle.name)
```

Charlie

When you add the `@value` decorator, Mojo synthesizes each special method above only if it doesn't exist already. That is, you can still implement a custom version of each method.

In addition to the `inout` argument convention you already saw with `__init__()`, this code also introduces `owned`, which is another argument convention that ensures the argument has unique ownership of the value. For more detail, see the section about [value ownership](#).

Was this page helpful?  

 Edit this page