

Traits

A *trait* is a set of requirements that a type must implement. You can think of it as a contract: a type that *conforms* to a trait guarantees that it implements all of the features of the trait.

Traits are similar to Java *interfaces*, C++ *concepts*, Swift *protocols*, and Rust *traits*. If you're familiar with any of those features, Mojo traits solve the same basic problem.

Background

In dynamically-typed languages like Python, you don't need to explicitly declare that two classes are similar. This is easiest to show by example:

```

%%python
class Duck:
    def quack(self):
        print("Quack.")

class StealthCow:
    def quack(self):
        print("Moo!")

def make_it_quack_python(maybe_a_duck):
    try:
        maybe_a_duck.quack()
    except:
        print("Not a duck.")

make_it_quack_python(Duck())
make_it_quack_python(StealthCow())

```

The `Duck` and `StealthCow` classes aren't related in any way, but they both define a `quack()` method, so they work the same in the `make_it_quack()` function. This works because Python uses dynamic dispatch—it identifies the methods to call at runtime. So `make_it_quack_python()` doesn't care what types you're passing it, only the fact that they implement the `quack()` method.

In a statically-typed environment, this approach doesn't work: [fn functions](#) require you to specify the type of each argument. If you wanted to write this example in Mojo *without* traits, you'd need to write a function overload for each

input type. All of the examples from here on are in Mojo, so we'll just call the function `make_it_quack()` going forward.

```
@value
struct Duck:
    fn quack(self):
        print("Quack")

@value
struct StealthCow:
    fn quack(self):
        print("Moo!")

fn make_it_quack(definitely_a_duck: Duck):
    definitely_a_duck.quack()

fn make_it_quack(not_a_duck: StealthCow):
    not_a_duck.quack()

make_it_quack(Duck())
make_it_quack(StealthCow())

Quack
Moo!
```

This isn't too bad with only two classes. But the more classes you want to support, the less practical this approach is.

You might notice that the Mojo versions of `make_it_quack()` don't include the `try/except` statement. We don't need it because Mojo's static type checking ensures that you can only pass instances of `Duck` or `StealthCow` into the `make_it_quack()` function.

Using traits

Traits solve this problem by letting you define a shared set of *behaviors* that types can implement. Then you can write a function that depends on the trait, rather than individual types. As an example, let's update the `make_it_quack()` example using traits. The first step is defining a trait:

```
trait Quackable:
    fn quack(self):
        ...
```

A trait looks a lot like a struct, except it's introduced by the `trait` keyword. Right now, a trait can only contain method signatures, and cannot include method implementations. Each method signature must be followed by three

dots (. . .) to indicate that the method is unimplemented.

TODO

In the future, we plan to support defining fields and default method implementations inside a trait. Right now, though, a trait can only declare method signatures.

Next we create some structs that conform to the `Quackable` trait. To indicate that a struct conforms to a trait, include the trait name in parenthesis after the struct name. You can also include multiple traits, separated by commas. (If you're familiar with Python, this looks just like Python's inheritance syntax.)

```
@value
struct Duck(Quackable):
    fn quack(self):
        print("Quack")

@value
struct StealthCow(Quackable):
    fn quack(self):
        print("Moo!")
```

The struct needs to implement any methods that are declared in the trait. The compiler enforces conformance: if a struct says it conforms to a trait, it must implement everything required by the trait or the code won't compile.

Finally, you can define a function that takes a `Quackable` like this:

```
fn make_it_quack[T: Quackable](maybe_a_duck: T):
    maybe_a_duck.quack()
```

This syntax may look a little unfamiliar if you haven't dealt with Mojo [parameters](#) before. What this signature means is that `maybe_a_duck` is an argument of type `T`, where `T` is a type that must conform to the `Quackable` trait. TODO: This syntax is a little verbose, and we hope to make it more ergonomic in a future release.

Using the method is simple enough:

```
make_it_quack(Duck())
make_it_quack(StealthCow())
```

```
Quack
Moo!
```

Note that you don't need the square brackets when you call `make_it_quack()`: the compiler infers the type of the argument, and ensures the type has the required trait.

One limitation of traits is that you can't add traits to existing types. For example, if you define a new `Numeric` trait, you can't add it to the standard library `Float64` and `Int` types. However, the standard library already includes a few traits, and we'll be adding more over time.

Traits can require static methods

In addition to regular instance methods, traits can specify required static methods.

```
trait HasStaticMethod:
  @staticmethod
  fn do_stuff(): ...

fn fun_with_traits[T: HasStaticMethod]():
  T.do_stuff()
```

Implicit trait conformance

Mojo also supports *implicit* trait conformance. That is, if a type implements all of the methods required for a trait, it's treated as conforming to the trait, even if it doesn't explicitly include the trait in its declaration:

```
struct RubberDucky:
  fn quack(self):
    print("Squeak!")

make_it_quack(RubberDucky())
```

Implicit conformance can be handy if you're defining a trait and you want it to work with types that you don't control—such as types from the standard library, or a third-party library.

However, we still strongly recommend explicit trait conformance wherever possible. This has two advantages:

- Documentation. It makes it clear that the type conforms to the trait, without having to scan all of its methods.
- Future feature support. When default method implementations are added to traits, they'll only work for types that explicitly conform to traits.

Trait inheritance

Traits can inherit from other traits. A trait that inherits from another trait includes all of the requirements declared by the parent trait. For example:

```

trait Animal:
    fn make_sound(self):
        ...

# Bird inherits from Animal
trait Bird(Animal):
    fn fly(self):
        ...

```

Since `Bird` inherits from `Animal`, a struct that conforms to the `Bird` trait needs to implement **both** `make_sound()` and `fly()`. And since every `Bird` conforms to `Animal`, a struct that conforms to `Bird` can be passed to any function that requires an `Animal`.

To inherit from multiple traits, add a comma-separated list of traits inside the parenthesis. For example, you could define a `NamedAnimal` trait that combines the requirements of the `Animal` trait and a new `Named` trait:

```

trait Named:
    fn get_name(self) -> String:
        ...

trait NamedAnimal(Animal, Named):
    pass

```

Traits and lifecycle methods

Traits can specify required [lifecycle methods](#), including constructors, copy constructors and move constructors.

For example, the following code creates a `MassProducible` trait. A `MassProducible` type has a default (no-argument) constructor and can be moved. It uses the built-in [Movable](#) trait, which requires the type to have a [move constructor](#).

The `factory[]()` function returns a newly-constructed instance of a `MassProducible` type.

```

trait DefaultConstructible:
    fn __init__(inout self): ...

trait MassProducible(DefaultConstructible, Movable):
    pass

fn factory[T: MassProducible]() -> T:
    return T()

struct Thing(MassProducible):
    var id: Int

```

```
fn __init__(inout self):  
    self.id = 0  
  
fn __moveinit__(inout self, owned existing: Self):  
    self.id = existing.id  
  
var thing = factory[Thing]()
```

Note that [@register_passable\("trivial"\)](#) types have restrictions on their lifecycle methods: they can't define copy or move constructors, because they don't require any custom logic.

For the purpose of trait conformance, the compiler treats trivial types as copyable and movable.

Built-in traits

The Mojo standard library currently includes a few traits. They're implemented by a number of standard library types, and you can also implement these on your own types:

- [Absable](#)
- [AnyType](#)
- [Boolable](#)
- [BoolableCollectionElement](#)
- [BoolableKeyElement](#)
- [CollectionElement](#)
- [Comparable](#)
- [ComparableCollectionElement](#)
- [Copyable](#)
- [Defaultable](#)
- [Formattable](#)
- [Hashable](#)
- [Indexer](#)
- [Intable](#)
- [IntableRaising](#)
- [KeyElement](#)
- [Movable](#)
- [PathLike](#)
- [Powable](#)
- [Representable](#)

- [RepresentableCollectionElement](#)
- [RepresentableKeyElement](#)
- [Sized](#)
- [Stringable](#)
- [StringableCollectionElement](#)
- [StringableRaising](#)
- [StringRepresentable](#)
- [Roundable](#)
- [ToFormatter](#)
- [Truncable](#)

The API reference docs linked above include usage examples for each trait. The following sections discuss a few of these traits.

The Sized trait

The [Sized](#) trait identifies types that have a measurable length, like strings and arrays.

Specifically, `Sized` requires a type to implement the `__len__()` method. This trait is used by the built-in [len\(\)](#) function. For example, if you're writing a custom list type, you could implement this trait so your type works with `len()`:

```
struct MyList(Sized):
    var size: Int
    # ...

    fn __init__(inout self):
        self.size = 0

    fn __len__(self) -> Int:
        return self.size

print(len(MyList()))
```

0

The Intable and IntableRaising traits

The [Intable](#) trait identifies a type that can be implicitly converted to `Int`. The [IntableRaising](#) trait describes a type can be converted to an `Int`, but the conversion might raise an error.

Both of these traits require the type to implement the `__int__()` method. For example:

```
@value
struct Foo(Intable):
    var i: Int

    fn __int__(self) -> Int:
        return self.i

var foo = Foo(42)
print(int(foo) == 42)
```

True

The Stringable, Representable, and Formattable traits

The [Stringable](#) trait identifies a type that can be implicitly converted to [String](#). The [StringableRaising](#) trait describes a type that can be converted to a [String](#), but the conversion might raise an error. Any type that conforms to [Stringable](#) or [StringableRaising](#) also works with the built-in [str\(\)](#) function to explicitly return a [String](#). These traits also mean that the type can support both the `{!s}` and `{}` format specifiers of the [String](#) class' [format\(\)](#) method. These traits require the type to define the [__str__\(\)](#) method.

In contrast, the [Representable](#) trait that defines a type that can be used with the built-in [repr\(\)](#) function, as well as the `{!r}` format specifier of the [format\(\)](#) method. This trait requires the type to define the [__repr__\(\)](#) method, which should compute the "official" string representation of a type. If at all possible, this should look like a valid Mojo expression that could be used to recreate a struct instance with the same value.

The [StringRepresentable](#) trait denotes a trait composition of the [Stringable](#) and [Representable](#) traits. It requires a type to implement both a [__str__\(\)](#) and a [__repr__\(\)](#) method.

The [Formattable](#) trait describes a type that can be converted to a stream of UTF-8 encoded data by writing to a formatter object. The [print\(\)](#) function requires that its arguments conform to the [Formattable](#) trait. This enables efficient stream-based writing by default, avoiding unnecessary intermediate [String](#) heap allocations.

The [Formattable](#) trait requires a type to implement a [format_to\(\)](#) method, which is provided with an instance of [Formatter](#) as an argument. You then invoke the [Formatter](#) instance's [write\(\)](#) method to write a sequence of [Formattable](#) arguments constituting the [String](#) representation of your type.

While this might sound complex at first, in practice you can minimize boilerplate and duplicated code by using the [String.format_sequence\(\)](#) static function to implement the type's [Stringable](#) implementation in terms of its [Formattable](#) implementation. Here is a simple example of a type that implements all of the [Stringable](#), [Representable](#), and [Formattable](#) traits:


```

@value
struct Dog(Stringable, Representable, Formattable):
    var name: String
    var age: Int

    fn __repr__(self) -> String:
        return "Dog(name=" + repr(self.name) + ", age=" + repr(self.age) + ")"

    fn __str__(self) -> String:
        return String.format_sequence(self)

    fn format_to(self, inout writer: Formatter) -> None:
        writer.write("Dog(", self.name, ", ", self.age, ")")

var dog = Dog("Rex", 5)
print(repr(dog))
print(dog)

var dog_info = String("String: {!s}\nRepresentation: {!r}").format(dog, dog)
print(dog_info)

```

```

Dog(name='Rex', age=5)
Dog(Rex, 5)
String: Dog(Rex, 5)
Representation: Dog(name='Rex', age=5)

```

The AnyType trait

When building a generic container type, one challenge is knowing how to dispose of the contained items when the container is destroyed. Any type that dynamically allocates memory needs to supply a [destructor](#) (`__del__()` method) that must be called to free the allocated memory. But not all types have a destructor, and your Mojo code has no way to determine which is which.

The [AnyType](#) trait solves this issue: every trait implicitly inherits from `AnyType`, and all structs conform to `AnyType`, which guarantees that the type has a destructor. For types that don't have one, Mojo adds a no-op destructor. This means you can call the destructor on any type.

This makes it possible to build generic collections without leaking memory. When the collection's destructor is called, it can safely call the destructors on every item it contains.

Generic structs with traits

You can also use traits when defining a generic container. A generic container is a container (for example, an array or hashmap) that can hold different data types. In a dynamic language like Python it's easy to add different types of items to a container. But in a statically-typed environment the compiler needs to be able to identify the types at compile time. For example, if the container needs to copy a value, the compiler needs to verify that the type can be copied.

The [List](#) type is an example of a generic container. A single `List` can only hold a single type of data. For example, you can create a list of integer values like this:

```
from collections import List


var list = List[Int](1, 2, 3)
for i in range(len(list)):
    print(list[i], sep=" ", end="")
```

1 2 3

You can use traits to define requirements for elements that are stored in a container. For example, `List` requires elements that can be moved and copied. To store a struct in a `List`, the struct needs to conform to the `CollectionElement` trait, which requires a [copy constructor](#) and a [move constructor](#).

Building generic containers is an advanced topic. For an introduction, see the section on [parameterized structs](#).

Was this page helpful?  

 Edit this page