The Move Book

Move Compiler supports receiver syntax e.f() , which allows defining methods which can be called on instances of a struct. The term "receiver" specifically refers to the instance that receives the method call. This is like the method syntax in other programming languages. It is a convenient way to define functions that operate on the fields of a struct, providing direct access to the struct's fields and creating cleaner, more intuitive code than passing the struct as a parameter.

If the first argument of a function is a struct internal to the module that defines the function, then the function can be called using the . operator. However, if the type of the first argument is defined in another module, then method won't be associated with the struct by default. In this case, the . operator syntax is not available, and the function must be called using standard function call syntax.

When a module is imported, its methods are automatically associated with the struct.

Method aliases help avoid name conflicts when modules define multiple structs and their methods. They can also provide more descriptive method names for structs.

Here's the syntax:

Public aliases are only allowed for structs defined in the same module. For structs defined in other modules, aliases can still be created but cannot be made public.

In the example below, we changed the hero module and added another type - Villain . Both Hero and Villain have similar field names and methods. To avoid name conflicts, we prefixed methods with hero_ and villain_ respectively. However, using aliases allows these methods to be called on struct instances without the prefix:

In the test function, the health method is called directly on the Hero and Villain instances without the prefix, as the compiler automatically associates the methods with their respective structs.

Note: In the test function, hero.health() is calling the aliased method, not directly accessing the private health field. While the Hero and Villain structs are public, their fields remain private to the module. The method call hero.health() uses the public alias defined by public use fun hero_health as Hero.health , which provides controlled access to the private field.

It is also possible to associate a function defined in another module with a struct from the current module. Following the same approach, we can create an alias for the method defined in another module. Let's use the bcs::to_bytes method from the Standard Library and associate it with the Hero struct. It will allow serializing the Hero struct to a vector of bytes.

Method syntax

If the first argument of a function is a struct internal to the module that defines the function, then the function can be called using the . operator. However, if the type of the first argument is defined in another module, then method won't be associated with the struct by default. In this case, the . operator syntax is not available, and the function must be called using standard function call syntax.

When a module is imported, its methods are automatically associated with the struct.

```bash module book::hero;

/// A struct representing a hero. public struct Hero has drop { health: u8, mana: u8, }

/// Create a new Hero. public fun new(): Hero { Hero { health: 100, mana: 100 } }

/// A method which casts a spell, consuming mana. public fun heal_spell(hero: &mut Hero) { hero.health = hero.health + 10; hero.mana = hero.mana - 10; }

/// A method which returns the health of the hero. public fun health(hero: &Hero): u8 { hero.health }

/// A method which returns the mana of the hero. public fun mana(hero: &Hero): u8 { hero.mana }

# [test]

// Test the methods of the `Hero` struct. fun test_methods() { let mut hero = new(); hero.heal_spell();

assert!(hero.health() == 110);
```

```
assert!(hero.mana() == 90);

} ```
```

Method aliases help avoid name conflicts when modules define multiple structs and their methods. They can also provide more descriptive method names for structs.

Here's the syntax:

```bash // for local method association use fun function_path as Type.method_name;

// exported alias public use fun function_path as Type.method_name; ```

Public aliases are only allowed for structs defined in the same module. For structs defined in other modules, aliases can still be created but cannot be made public.

In the example below, we changed the hero module and added another type - Villain . Both Hero and Villain have similar field names and methods. To avoid name conflicts, we prefixed methods with hero_ and villain_ respectively. However, using aliases allows these methods to be called on struct instances without the prefix:

```bash module book::hero_and_villain;

/// A struct representing a hero. public struct Hero has drop { health: u8, }

/// A struct representing a villain. public struct Villain has drop { health: u8, }

/// Create a new Hero. public fun new_hero(): Hero { Hero { health: 100 } }

/// Create a new Villain. public fun new_villain(): Villain { Villain { health: 100 } }

// Alias for the `hero_health` method. Will be imported automatically when // the module is imported. public use fun hero_health as Hero.health;

public fun hero_health(hero: &Hero): u8 { hero.health }

// Alias for the `villain_health` method. Will be imported automatically // when the module is imported. public use fun villain_health as Villain.health;

public fun villain_health(villain: &Villain): u8 { villain.health }

# [test]

// Test the methods of the `Hero` and `Villain` structs. fun test_associated_methods() { let hero = new_hero(); assert!(hero.health() == 100);

```
let villain = new_villain();
assert!(villain.health() == 100);
```

} ```

In the test function, the health method is called directly on the Hero and Villain instances without the prefix, as the compiler automatically associates the methods with their respective structs.

Note: In the test function, hero.health() is calling the aliased method, not directly accessing the private health field. While the Hero and Villain structs are public, their fields remain private to the module. The method call hero.health() uses the public alias defined by public use fun hero_health as Hero.health , which provides controlled access to the private field.

It is also possible to associate a function defined in another module with a struct from the current module. Following the same approach, we can create an alias for the method defined in another module. Let's use the bcs::to_bytes method from the Standard Library and associate it with the Hero struct. It will allow serializing the Hero struct to a vector of bytes.

```bash // TODO: better example (external module...) module book::hero_to_bytes;

// Alias for the `bcs::to_bytes` method. Imported aliases should be defined // in the top of the module. // public use fun bcs::to_bytes as Hero.to_bytes;
```

/// A struct representing a hero. public struct Hero has drop { health: u8, mana: u8, }

/// Create a new Hero. public fun new(): Hero { Hero { health: 100, mana: 100 } }

[test]

// Test the methods of the `Hero` struct. fun test_hero_serialize() { // let mut hero = new(); // let serialized = hero.to_bytes(); // assert!(serialized.length() == 3, 1); } ```

Method Aliases

Method aliases help avoid name conflicts when modules define multiple structs and their methods. They can also provide more descriptive method names for structs.

Here's the syntax:

```bash // for local method association use fun function_path as Type.method_name;

// exported alias public use fun function_path as Type.method_name; ```

Public aliases are only allowed for structs defined in the same module. For structs defined in other modules, aliases can still be created but cannot be made public.

In the example below, we changed the hero module and added another type - Villain . Both Hero and Villain have similar field names and methods. To avoid name conflicts, we prefixed methods with hero_ and villain_ respectively. However, using aliases allows these methods to be called on struct instances without the prefix:

```bash module book::hero_and_villain;

/// A struct representing a hero. public struct Hero has drop { health: u8, }

/// A struct representing a villain. public struct Villain has drop { health: u8, }

/// Create a new Hero. public fun new_hero(): Hero { Hero { health: 100 } }

/// Create a new Villain. public fun new_villain(): Villain { Villain { health: 100 } }

// Alias for the `hero_health` method. Will be imported automatically when // the module is imported. public use fun hero_health as Hero.health;

public fun hero_health(hero: &Hero): u8 { hero.health }

// Alias for the `villain_health` method. Will be imported automatically // when the module is imported. public use fun villain_health as Villain.health;

public fun villain_health(villain: &Villain): u8 { villain.health }

# [test]

// Test the methods of the `Hero` and `Villain` structs. fun test_associated_methods() { let hero = new_hero(); assert!(hero.health() == 100);

```
let villain = new_villain();
assert!(villain.health() == 100);
```

} ```

In the test function, the health method is called directly on the Hero and Villain instances without the prefix, as the compiler automatically associates the methods with their respective structs.

Note: In the test function, hero.health() is calling the aliased method, not directly accessing the private health field. While the Hero and Villain structs are public, their fields remain private to the module. The method call hero.health() uses the public alias defined by public use fun hero_health as Hero.health , which provides controlled access to the private field.

It is also possible to associate a function defined in another module with a struct from the current module. Following the same approach, we can create an alias for the method defined in another module. Let's use the bcs::to_bytes method from the Standard Library and associate it with the Hero struct. It will allow serializing the Hero struct to a vector of bytes.

```bash // TODO: better example (external module...) module book::hero_to_bytes;

// Alias for the `bcs::to_bytes` method. Imported aliases should be defined // in the top of the module. // public use fun bcs::to_bytes as Hero.to_bytes;

/// A struct representing a hero. public struct Hero has drop { health: u8, mana: u8, }

/// Create a new Hero. public fun new(): Hero { Hero { health: 100, mana: 100 } }

# [test]

// Test the methods of the `Hero` struct. fun test_hero_serialize() { // let mut hero = new(); // let serialized = hero.to_bytes(); // assert!(serialized.length() == 3, 1); } ```

## Aliasing an external module's method

It is also possible to associate a function defined in another module with a struct from the current module. Following the same approach, we can create an alias for the method defined in another module. Let's use the bcs::to_bytes method from the Standard Library and associate it with the Hero struct. It will allow serializing the Hero struct to a vector of bytes.

```bash // TODO: better example (external module...) module book::hero_to_bytes;

// Alias for the `bcs::to_bytes` method. Imported aliases should be defined // in the top of the module. // public use fun bcs::to_bytes as Hero.to_bytes;

/// A struct representing a hero. public struct Hero has drop { health: u8, mana: u8, }

/// Create a new Hero. public fun new(): Hero { Hero { health: 100, mana: 100 } }

# [test]

// Test the methods of the `Hero` struct. fun test_hero_serialize() { // let mut hero = new(); // let serialized = hero.to_bytes(); // assert!(serialized.length() == 3, 1); } ```

## Further reading