The Move Book

Every variable in Move has a scope and an owner. The scope is the range of code where the variable is valid, and the owner is the scope that this variable belongs to. Once the owner scope ends, the variable is dropped. This is a fundamental concept in Move, and it is important to understand how it works.

A variable defined in a function scope is owned by this scope. The runtime goes through the function scope and executes every expression and statement. After the function scope ends, the variables defined in it are dropped or deallocated.

In the example above, the variable a is owned by the owner function, and the variable b is owned by the other function. When each of these functions are called, the variables are defined, and when the function ends, the variables are discarded.

If we changed the owner function to return the variable a , then the ownership of a would be transferred to the caller of the function.

Additionally, if we passed the variable a to another function, the ownership of a would be transferred to this function. When performing this operation, we move the value from one scope to another. This is also called move semantics .

Each function has a main scope, and it can also have sub-scopes via the use of blocks. A block is a sequence of statements and expressions, and it has its own scope. Variables defined in a block are owned by this block, and when the block ends, the variables are dropped.

However, if we return a value from a block, the ownership of the variable is transferred to the caller of the block.

Some types in Move are copyable , which means that they can be copied without transferring ownership. This is useful for types that are small and cheap to copy, such as integers and booleans. The Move compiler will automatically copy these types when they are passed to or returned from a function, or when they're moved to another scope and then accessed in their original scope.

Ownership

A variable defined in a function scope is owned by this scope. The runtime goes through the function scope and executes every expression and statement. After the function scope ends, the variables defined in it are dropped or deallocated.

```bash module book::ownership;

public fun owner() { let a = 1; // a is owned by the `owner` function } // a is dropped here

public fun other() { let b = 2; // b is owned by the `other` function } // b is dropped here

# [test]

fun test_owner() { owner(); other(); // a & b are not valid here } ```

In the example above, the variable a is owned by the owner function, and the variable b is owned by the other function. When each of these functions are called, the variables are defined, and when the function ends, the variables are discarded.

If we changed the owner function to return the variable a , then the ownership of a would be transferred to the caller of the function.

```bash module book::ownership;

public fun owner(): u8 { let a = 1; // a defined here a // scope ends, a is returned }

# [test]

fun test_owner() { let a = owner(); // a is valid here } // a is dropped here ```

Additionally, if we passed the variable a to another function, the ownership of a would be transferred to this function. When performing this operation, we move the value from one scope to another. This is also called move semantics .

```bash module book::ownership;

public fun owner(): u8 { let a = 10; a } // a is returned
```

public fun take_ownership(v: u8) { // v is owned by `take_ownership` } // v is dropped here

[test]

fun test_owner() { let a = owner(); take_ownership(a); // a is not valid here } ```

Each function has a main scope, and it can also have sub-scopes via the use of blocks. A block is a sequence of statements and expressions, and it has its own scope. Variables defined in a block are owned by this block, and when the block ends, the variables are dropped.

```bash module book::ownership;

public fun owner() { let a = 1; // a is owned by the `owner` function's scope { let b = 2; // the block that declares b owns it { let c = 3; // the block that declares c owns it }; // c is dropped here }; // b is dropped here // a = b; // error: b is not valid here // a = c; // error: c is not valid here } // a is dropped here ```

However, if we return a value from a block, the ownership of the variable is transferred to the caller of the block.

```bash module book::ownership;

public fun owner(): u8 { let a = 1; // a is owned by the `owner` function's scope let b = { let c = 2; // the block that declares c owns it c // c is returned from the block and transferred to b }; a + b // both a and b are valid here } ```

Some types in Move are copyable , which means that they can be copied without transferring ownership. This is useful for types that are small and cheap to copy, such as integers and booleans. The Move compiler will automatically copy these types when they are passed to or returned from a function, or when they're moved to another scope and then accessed in their original scope.

## Returning a Value

If we changed the owner function to return the variable a , then the ownership of a would be transferred to the caller of the function.

```bash module book::ownership;

public fun owner(): u8 { let a = 1; // a defined here a // scope ends, a is returned }

# [test]

fun test_owner() { let a = owner(); // a is valid here } // a is dropped here ```

Additionally, if we passed the variable a to another function, the ownership of a would be transferred to this function. When performing this operation, we move the value from one scope to another. This is also called move semantics .

```bash module book::ownership;

public fun owner(): u8 { let a = 10; a } // a is returned

public fun take_ownership(v: u8) { // v is owned by `take_ownership` } // v is dropped here

# [test]

fun test_owner() { let a = owner(); take_ownership(a); // a is not valid here } ```

Each function has a main scope, and it can also have sub-scopes via the use of blocks. A block is a sequence of statements and expressions, and it has its own scope. Variables defined in a block are owned by this block, and when the block ends, the variables are dropped.

```bash module book::ownership;

public fun owner() { let a = 1; // a is owned by the `owner` function's scope { let b = 2; // the block that declares b owns it { let c = 3; // the block that declares c owns it }; // c is dropped here }; // b is dropped here // a = b; // error: b is not valid here // a = c; // error: c is not valid here } // a is dropped here ```
```

However, if we return a value from a block, the ownership of the variable is transferred to the caller of the block.

```bash module book::ownership;

public fun owner(): u8 { let a = 1; // a is owned by the `owner` function's scope let b = { let c = 2; // the block that declares c owns it c // c is returned from the block and transferred to b }; a + b // both a and b are valid here } ```

Some types in Move are copyable , which means that they can be copied without transferring ownership. This is useful for types that are small and cheap to copy, such as integers and booleans. The Move compiler will automatically copy these types when they are passed to or returned from a function, or when they're moved to another scope and then accessed in their original scope.

## Passing by Value

Additionally, if we passed the variable a to another function, the ownership of a would be transferred to this function. When performing this operation, we move the value from one scope to another. This is also called move semantics .

```bash module book::ownership;

public fun owner(): u8 { let a = 10; a } // a is returned

public fun take_ownership(v: u8) { // v is owned by `take_ownership` } // v is dropped here

# [test]

fun test_owner() { let a = owner(); take_ownership(a); // a is not valid here } ```

Each function has a main scope, and it can also have sub-scopes via the use of blocks. A block is a sequence of statements and expressions, and it has its own scope. Variables defined in a block are owned by this block, and when the block ends, the variables are dropped.

```bash module book::ownership;

public fun owner() { let a = 1; // a is owned by the `owner` function's scope { let b = 2; // the block that declares b owns it { let c = 3; // the block that declares c owns it }; // c is dropped here }; // b is dropped here // a = b; // error: b is not valid here // a = c; // error: c is not valid here } // a is dropped here ```

However, if we return a value from a block, the ownership of the variable is transferred to the caller of the block.

```bash module book::ownership;

public fun owner(): u8 { let a = 1; // a is owned by the `owner` function's scope let b = { let c = 2; // the block that declares c owns it c // c is returned from the block and transferred to b }; a + b // both a and b are valid here } ```

Some types in Move are copyable , which means that they can be copied without transferring ownership. This is useful for types that are small and cheap to copy, such as integers and booleans. The Move compiler will automatically copy these types when they are passed to or returned from a function, or when they're moved to another scope and then accessed in their original scope.

## Scopes with Blocks

Each function has a main scope, and it can also have sub-scopes via the use of blocks. A block is a sequence of statements and expressions, and it has its own scope. Variables defined in a block are owned by this block, and when the block ends, the variables are dropped.

```bash module book::ownership;

public fun owner() { let a = 1; // a is owned by the `owner` function's scope { let b = 2; // the block that declares b owns it { let c = 3; // the block that declares c owns it }; // c is dropped here }; // b is dropped here // a = b; // error: b is not valid here // a = c; // error: c is not valid here } // a is dropped here ```

However, if we return a value from a block, the ownership of the variable is transferred to the caller of the block.

```bash module book::ownership;

public fun owner(): u8 { let a = 1; // a is owned by the `owner` function's scope let b = { let c = 2; // the block that declares c owns it c // c is returned from the block and transferred to b }; a + b // both a and b are valid here } ```

Some types in Move are copyable , which means that they can be copied without transferring ownership. This is useful for types that are small and cheap to copy, such as integers and booleans. The Move compiler will automatically copy these types when they are passed to or returned from a function, or when they're moved to another scope and then accessed in their original scope.

## Copyable Types

Some types in Move are copyable , which means that they can be copied without transferring ownership. This is useful for types that are small and cheap to copy, such as integers and booleans. The Move compiler will automatically copy these types when they are passed to or returned from a function, or when they're moved to another scope and then accessed in their original scope.

## Further reading