The Move Book

In programming, a capability is a token that gives the owner the right to perform a specific action. It is a pattern that is used to control access to resources and operations. A simple example of a capability is a key to a door. If you have the key, you can open the door. If you don't have the key, you can't open the door. A more practical example is an Admin Capability which allows the owner to perform administrative operations, which regular users cannot.

In the Sui Object Model , capabilities are represented as objects. An owner of an object can pass this object to a function to prove that they have the right to perform a specific action. Due to strict typing, the function taking a capability as an argument can only be called with the correct capability.

There's a convention to name capabilities with the Cap suffix, for example, AdminCap or KioskOwnerCap .

A very common practice is to create a single AdminCap object on package publish. This way, the application can have a setup phase where the admin account prepares the state of the application.

Utilizing objects as capabilities is a relatively new concept in blockchain programming. And in other smart-contract languages, authorization is often performed by checking the address of the sender. This pattern is still viable on Sui, however, overall recommendation is to use capabilities for better security, discoverability, and code organization.

Let's look at how the new function that creates a user would look like if it was using the address check:

And now, let's see how the same function would look like with the capability:

Using capabilities has several advantages over the address check:

However, the address approach has its own advantages. For example, if an address is multisig, and transaction building gets more complex, it might be easier to check the address. Also, if there's a central object of the application that is used in every function, it can store the admin address, and this would simplify migration. The central object approach is also valuable for revokable capabilities, where the admin can revoke the capability from the user.

Capability is an Object

In the Sui Object Model , capabilities are represented as objects. An owner of an object can pass this object to a function to prove that they have the right to perform a specific action. Due to strict typing, the function taking a capability as an argument can only be called with the correct capability.

There's a convention to name capabilities with the Cap suffix, for example, AdminCap or KioskOwnerCap .

```bash module book::capability;

use std::string::String; use sui::event;

/// The capability granting the application admin the right to create new /// accounts in the system. public struct AdminCap has key, store { id: UID }

/// The user account in the system. public struct Account has key, store { id: UID, name: String }

/// A simple `Ping` event with no data. public struct Ping has copy, drop { by: ID }

/// Creates a new account in the system. Requires the `AdminCap` capability /// to be passed as the first argument. public fun new(_: &AdminCap, name: String, ctx: &mut TxContext): Account { Account { id: object::new(ctx), name, } }

/// Account, and any other objects, can also be used as a Capability in the /// application. For example, to emit an event. public fun send_ping(acc: &Account) { event::emit(Ping { by: acc.id.to_inner() }) }

/// Updates the account name. Can only be called by the `Account` owner. public fun update(account: &mut Account, name: String) { account.name = name; } ```

A very common practice is to create a single AdminCap object on package publish. This way, the application can have a setup phase where the admin account prepares the state of the application.

```bash module book::admin_cap;

/// The capability granting the admin privileges in the system. /// Created only once in the `init` function. public struct AdminCap has key { id: UID }

/// Create the AdminCap object on package publish and transfer it to the /// package owner. fun init(ctx: &mut TxContext) { transfer::transfer( AdminCap { id: object::new(ctx) }, ctx.sender() ) } ```

Utilizing objects as capabilities is a relatively new concept in blockchain programming. And in other smart-contract languages, authorization is often performed by checking the address of the sender. This pattern is still viable on Sui, however, overall recommendation is to use capabilities for better security, discoverability, and code organization.

Let's look at how the new function that creates a user would look like if it was using the address check:

```bash /// Error code for unauthorized access. const ENotAuthorized: u64 = 0;

/// The application admin address. const APPLICATION_ADMIN: address = @0xa11ce;

/// Creates a new user in the system. Requires the sender to be the application /// admin. public fun new(ctx: &mut TxContext): User { assert!(ctx.sender() == APPLICATION_ADMIN, ENotAuthorized); User { id: object::new(ctx) } } ```

And now, let's see how the same function would look like with the capability:

```bash /// Grants the owner the right to create new users in the system. public struct AdminCap {}

/// Creates a new user in the system. Requires the `AdminCap` capability to be /// passed as the first argument. public fun new(_: &AdminCap, ctx: &mut TxContext): User { User { id: object::new(ctx) } } ```

Using capabilities has several advantages over the address check:

However, the address approach has its own advantages. For example, if an address is multisig, and transaction building gets more complex, it might be easier to check the address. Also, if there's a central object of the application that is used in every function, it can store the admin address, and this would simplify migration. The central object approach is also valuable for revokable capabilities, where the admin can revoke the capability from the user.

## Using

A very common practice is to create a single AdminCap object on package publish. This way, the application can have a setup phase where the admin account prepares the state of the application.

```bash module book::admin_cap;

/// The capability granting the admin privileges in the system. /// Created only once in the `init` function. public struct AdminCap has key { id: UID }

/// Create the AdminCap object on package publish and transfer it to the /// package owner. fun init(ctx: &mut TxContext) { transfer::transfer( AdminCap { id: object::new(ctx) }, ctx.sender() ) } ```

Utilizing objects as capabilities is a relatively new concept in blockchain programming. And in other smart-contract languages, authorization is often performed by checking the address of the sender. This pattern is still viable on Sui, however, overall recommendation is to use capabilities for better security, discoverability, and code organization.

Let's look at how the new function that creates a user would look like if it was using the address check:

```bash /// Error code for unauthorized access. const ENotAuthorized: u64 = 0;

/// The application admin address. const APPLICATION_ADMIN: address = @0xa11ce;

/// Creates a new user in the system. Requires the sender to be the application /// admin. public fun new(ctx: &mut TxContext): User { assert!(ctx.sender() == APPLICATION_ADMIN, ENotAuthorized); User { id: object::new(ctx) } } ```

And now, let's see how the same function would look like with the capability:

```bash /// Grants the owner the right to create new users in the system. public struct AdminCap {}

/// Creates a new user in the system. Requires the `AdminCap` capability to be /// passed as the first argument. public fun new(_: &AdminCap, ctx: &mut TxContext): User { User { id: object::new(ctx) } } ```
```

Using capabilities has several advantages over the address check:

However, the address approach has its own advantages. For example, if an address is multisig, and transaction building gets more complex, it might be easier to check the address. Also, if there's a central object of the application that is used in every function, it can store the admin address, and this would simplify migration. The central object approach is also valuable for revokable capabilities, where the admin can revoke the capability from the user.

Address check vs Capability

Utilizing objects as capabilities is a relatively new concept in blockchain programming. And in other smart-contract languages, authorization is often performed by checking the address of the sender. This pattern is still viable on Sui, however, overall recommendation is to use capabilities for better security, discoverability, and code organization.

Let's look at how the new function that creates a user would look like if it was using the address check:

```bash /// Error code for unauthorized access. const ENotAuthorized: u64 = 0;

/// The application admin address. const APPLICATION_ADMIN: address = @0xa11ce;

/// Creates a new user in the system. Requires the sender to be the application /// admin. public fun new(ctx: &mut TxContext): User { assert!(ctx.sender() == APPLICATION_ADMIN, ENotAuthorized); User { id: object::new(ctx) } } ```

And now, let's see how the same function would look like with the capability:

```bash /// Grants the owner the right to create new users in the system. public struct AdminCap {}

/// Creates a new user in the system. Requires the `AdminCap` capability to be /// passed as the first argument. public fun new(_: &AdminCap, ctx: &mut TxContext): User { User { id: object::new(ctx) } } ```

Using capabilities has several advantages over the address check:

However, the address approach has its own advantages. For example, if an address is multisig, and transaction building gets more complex, it might be easier to check the address. Also, if there's a central object of the application that is used in every function, it can store the admin address, and this would simplify migration. The central object approach is also valuable for revokable capabilities, where the admin can revoke the capability from the user.