The Move Book

The module that defines main storage operations is sui::transfer . It is implicitly imported in all packages that depend on the Sui Framework , so, like other implicitly imported modules (e.g. std::option or std::vector), it does not require adding a use statement.

The transfer module provides functions to perform all three storage operations matching ownership types which we explained:

On this page we will only talk about so-called restricted storage operations, later we will cover public ones, after the store ability is introduced.

The transfer module is a go-to for most of the storage operations, except a special case with Dynamic Fields that awaits us in the next chapter.

In the Ownership and Scope and References chapters, we covered the basics of ownership and references in Move. It is important that you understand these concepts when using storage functions. Here is a quick recap of the most important points:

The transfer::transfer function is a public function used to transfer an object to another address. Its signature is as follows, only accepts a type with the key ability and an address of the recipient. Please, note that the object is passed into the function by value , therefore it is moved to the function scope and then moved to the recipient address:

In the next example, you can see how it can be used in a module that defines and sends an object to the transaction sender.

When the module is published, the init function will get called, and the AdminCap object which we created there will be transferred to the transaction sender. The ctx.sender() function returns the sender address for the current transaction.

Once the AdminCap has been transferred to the sender, for example, to 0xa11ce , the sender, and only the sender, will be able to access the object. The object is now account owned .

Account owned objects are a subject to true ownership - only the account owner can access them. This is a fundamental concept in the Sui storage model.

Let's extend the example with a function that uses AdminCap to authorize a mint of a new object and its transfer to another address:

The mint_and_transfer function is a public function that "could" be called by anyone, but it requires an AdminCap object to be passed as the first argument by reference. Without it, the function will not be callable. This is a simple way to restrict access to privileged functions called Capability . Because the AdminCap object is account owned , only 0xa11ce will be able to call the mint_and_transfer function.

The Gift s sent to recipients will also be account owned , each gift being unique and owned exclusively by the recipient.

A quick recap:

The transfer::freeze_object function is a public function that is used to put an object into an immutable state. Once an object is frozen , it can never be changed, and it can be accessed by anyone by immutable reference.

The function signature is as follows, only accepts a type with the key ability . Just like all other storage functions, it takes the object by value :

Let's expand on the previous example and add a function that allows the admin to create a Config object and freeze it:

Config is an object that has a message field, and the create_and_freeze function creates a new Config and freezes it. Once the object is frozen, it can be accessed by anyone by immutable reference. The message function is a public function that returns the message from the Config object. Config is now publicly available by its ID, and the message can be read by anyone.

Function definitions are not connected to the object's state. It is possible to define a function that takes a mutable reference to an object that is used as frozen. However, it won't be callable on a frozen object.

The message function can be called on an immutable Config object, however, two functions below are not callable on a frozen object:

To summarize:

Since the transfer::freeze_object signature accepts any type with the key ability, it can take an object that was created in the same scope, but it can also take an object that was owned by an account. This means that the freeze_object function can be used to

freeze an object that was transferred to the sender. For security concerns, we would not want to freeze the AdminCap object - it would be a security risk to allow access to it to anyone. However, we can freeze the Gift object that was minted and transferred to the recipient:

Single Owner -> Immutable conversion is possible!

The transfer::share_object function is a public function used to put an object into a shared state. Once an object is shared , it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the key ability :

Once an object is shared , it is publicly available as a mutable reference.

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Overview

The transfer module provides functions to perform all three storage operations matching ownership types which we explained:

On this page we will only talk about so-called restricted storage operations, later we will cover public ones, after the store ability is introduced.

The transfer module is a go-to for most of the storage operations, except a special case with Dynamic Fields that awaits us in the next chapter.

In the Ownership and Scope and References chapters, we covered the basics of ownership and references in Move. It is important that you understand these concepts when using storage functions. Here is a quick recap of the most important points:

```bash /// Moved by value public fun take(value: T) { / *value is moved here!* / abort 0 }

/// For immutable reference public fun borrow(value: &T) { / *value is borrowed here! can be read* / abort 0 }

/// For mutable reference public fun borrow_mut(value: &mut T) { / *value is mutably borrowed here!* / abort 0 } ```

The transfer::transfer function is a public function used to transfer an object to another address. Its signature is as follows, only accepts a type with the key ability and an address of the recipient. Please, note that the object is passed into the function by value , therefore it is moved to the function scope and then moved to the recipient address:

```
bash // File: sui-framework/sources/transfer.move public fun transfer<T: key>(obj: T, recipient: address);
```

In the next example, you can see how it can be used in a module that defines and sends an object to the transaction sender.

```bash module book::transfer_to_sender {

/// A struct with `key` is an object. The first field is `id: UID`!
public struct AdminCap has key { id: UID }

/// `init` function is a special function that is called when the module
/// is published. It is a good place to create application objects.
fun init(ctx: &mut TxContext) {
    // Create a new `AdminCap` object, in this scope.
```
```

```
    let admin_cap = AdminCap { id: object::new(ctx) };

    // Transfer the object to the transaction sender.
    transfer::transfer(admin_cap, ctx.sender());

    // admin_cap is gone! Can't be accessed anymore.
}

/// Transfers the `AdminCap` object to the `recipient`. Thus, the recipient
/// becomes the owner of the object, and only they can access it.
public fun transfer_admin_cap(cap: AdminCap, recipient: address) {
    transfer::transfer(cap, recipient);
}
```

} ```

When the module is published, the init function will get called, and the AdminCap object which we created there will be transferred to the transaction sender. The ctx.sender() function returns the sender address for the current transaction.

Once the AdminCap has been transferred to the sender, for example, to 0xa11ce , the sender, and only the sender, will be able to access the object. The object is now account owned .

Account owned objects are a subject to true ownership - only the account owner can access them. This is a fundamental concept in the Sui storage model.

Let's extend the example with a function that uses AdminCap to authorize a mint of a new object and its transfer to another address:

``bash /// SomeGiftobject that the admin canmint_and_transfer`. public struct Gift has key { id: UID }

/// Creates a new `Gift` object and transfers it to the `recipient`. public fun mint_and_transfer(_: &AdminCap, recipient: address, ctx: &mut TxContext) { let gift = Gift { id: object::new(ctx) }; transfer::transfer(gift, recipient); } ```

The mint_and_transfer function is a public function that "could" be called by anyone, but it requires an AdminCap object to be passed as the first argument by reference. Without it, the function will not be callable. This is a simple way to restrict access to privileged functions called Capability . Because the AdminCap object is account owned , only 0xa11ce will be able to call the mint_and_transfer function.

The Gift s sent to recipients will also be account owned , each gift being unique and owned exclusively by the recipient.

A quick recap:

The transfer::freeze_object function is a public function that is used to put an object into an immutable state. Once an object is frozen , it can never be changed, and it can be accessed by anyone by immutable reference.

The function signature is as follows, only accepts a type with the key ability . Just like all other storage functions, it takes the object by value :

```
bash // File: sui-framework/sources/transfer.move public fun freeze_object<T: key>(obj: T);
```

Let's expand on the previous example and add a function that allows the admin to create a Config object and freeze it:

``bash /// SomeConfigobject that the admin cancreate_and_freeze`. public struct Config has key { id: UID, message: String }

/// Creates a new `Config` object and freezes it. public fun create_and_freeze(_: &AdminCap, message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

```
// Freeze the object so it becomes immutable.
transfer::freeze_object(config);
```

}

/// Returns the message from the `Config` object. /// Can access the object by immutable reference! public fun message(c: &Config): String { c.message } ```

Config is an object that has a message field, and the create_and_freeze function creates a new Config and freezes it. Once the object is frozen, it can be accessed by anyone by immutable reference. The message function is a public function that returns the message from the Config object. Config is now publicly available by its ID, and the message can be read by anyone.

Function definitions are not connected to the object's state. It is possible to define a function that takes a mutable reference to an object that is used as frozen. However, it won't be callable on a frozen object.

The message function can be called on an immutable Config object, however, two functions below are not callable on a frozen object:

```bash // ═══ Functions below can't be called on a frozen object! ═══

/// The function can be defined, but it won't be callable on a frozen object. /// Only immutable references are allowed. public fun message_mut(c: &mut Config): &mut String { &mut c.message }

/// Deletes the `Config` object, takes it by value. /// Can't be called on a frozen object! public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() } ```

To summarize:

Since the transfer::freeze_object signature accepts any type with the key ability, it can take an object that was created in the same scope, but it can also take an object that was owned by an account. This means that the freeze_object function can be used to freeze an object that was transferred to the sender. For security concerns, we would not want to freeze the AdminCap object - it would be a security risk to allow access to it to anyone. However, we can freeze the Gift object that was minted and transferred to the recipient:

Single Owner -> Immutable conversion is possible!

```
bash /// Freezes the `Gift` object so it becomes immutable. public fun freeze_gift(gift: Gift) { transfer::freeze_object(gift); }
```

The transfer::share_object function is a public function used to put an object into a shared state. Once an object is shared , it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the key ability :

```
bash // File: sui-framework/sources/transfer.move public fun share_object<T: key>(obj: T);
```

Once an object is shared , it is publicly available as a mutable reference.

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

```bash /// Creates a newConfig` object and shares it. public fun create_and_share(message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

// Share the object so it becomes shared.
transfer::share_object(config);

} ```
```

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
bash /// Deletes the `Config` object, takes it by value. /// Can be called on a shared object! public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() }
```

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.

```
bash // Won't work! public fun transfer_shared(c: Config, to: address) { transfer::transfer(c, to); }
```

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.
```

Ownership and References: A Quick Recap

In the Ownership and Scope and References chapters, we covered the basics of ownership and references in Move. It is important that you understand these concepts when using storage functions. Here is a quick recap of the most important points:

```bash /// Moved by value public fun take(value: T) { / *value is moved here!* / abort 0 }

/// For immutable reference public fun borrow(value: &T) { / *value is borrowed here! can be read* / abort 0 }

/// For mutable reference public fun borrow_mut(value: &mut T) { / *value is mutably borrowed here!* / abort 0 } ```

The transfer::transfer function is a public function used to transfer an object to another address. Its signature is as follows, only accepts a type with the key ability and an address of the recipient. Please, note that the object is passed into the function by value , therefore it is moved to the function scope and then moved to the recipient address:

```
bash // File: sui-framework/sources/transfer.move public fun transfer<T: key>(obj: T, recipient: address);
```

In the next example, you can see how it can be used in a module that defines and sends an object to the transaction sender.

```bash module book::transfer_to_sender {

```
/// A struct with `key` is an object. The first field is `id: UID`!
public struct AdminCap has key { id: UID }

/// `init` function is a special function that is called when the module
/// is published. It is a good place to create application objects.
fun init(ctx: &mut TxContext) {
    // Create a new `AdminCap` object, in this scope.
    let admin_cap = AdminCap { id: object::new(ctx) };

    // Transfer the object to the transaction sender.
    transfer::transfer(admin_cap, ctx.sender());

    // admin_cap is gone! Can't be accessed anymore.
}

/// Transfers the `AdminCap` object to the `recipient`. Thus, the recipient
/// becomes the owner of the object, and only they can access it.
public fun transfer_admin_cap(cap: AdminCap, recipient: address) {
    transfer::transfer(cap, recipient);
}
```

} ```

When the module is published, the init function will get called, and the AdminCap object which we created there will be transferred to the transaction sender. The ctx.sender() function returns the sender address for the current transaction.

Once the AdminCap has been transferred to the sender, for example, to 0xa11ce , the sender, and only the sender, will be able to access the object. The object is now account owned .

Account owned objects are a subject to true ownership - only the account owner can access them. This is a fundamental concept in the Sui storage model.

Let's extend the example with a function that uses AdminCap to authorize a mint of a new object and its transfer to another address:

``bash /// SomeGiftobject that the admin canmint_and_transfer`. public struct Gift has key { id: UID }

/// Creates a new `Gift` object and transfers it to the `recipient`. public fun mint_and_transfer(_: &AdminCap, recipient: address, ctx: &mut TxContext) { let gift = Gift { id: object::new(ctx) }; transfer::transfer(gift, recipient); } ```

The mint_and_transfer function is a public function that "could" be called by anyone, but it requires an AdminCap object to be passed as the first argument by reference. Without it, the function will not be callable. This is a simple way to restrict access to privileged functions called Capability . Because the AdminCap object is account owned , only 0xa11ce will be able to call the mint_and_transfer function.

The Gift s sent to recipients will also be account owned , each gift being unique and owned exclusively by the recipient.

A quick recap:

The transfer::freeze_object function is a public function that is used to put an object into an immutable state. Once an object is frozen , it can never be changed, and it can be accessed by anyone by immutable reference.

The function signature is as follows, only accepts a type with the key ability . Just like all other storage functions, it takes the object by value :

```
bash // File: sui-framework/sources/transfer.move public fun freeze_object<T: key>(obj: T);
```

Let's expand on the previous example and add a function that allows the admin to create a Config object and freeze it:

```bash /// SomeConfigobject that the admin cancreate_and_freeze`. public struct Config has key { id: UID, message: String }

/// Creates a new Config object and freezes it. public fun create_and_freeze(_: &AdminCap, message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

// Freeze the object so it becomes immutable.
transfer::freeze_object(config);

}

/// Returns the message from the Config object. /// Can access the object by immutable reference! public fun message(c: &Config): String { c.message } ```
```

Config is an object that has a message field, and the create_and_freeze function creates a new Config and freezes it. Once the object is frozen, it can be accessed by anyone by immutable reference. The message function is a public function that returns the message from the Config object. Config is now publicly available by its ID, and the message can be read by anyone.

Function definitions are not connected to the object's state. It is possible to define a function that takes a mutable reference to an object that is used as frozen. However, it won't be callable on a frozen object.

The message function can be called on an immutable Config object, however, two functions below are not callable on a frozen object:

```bash // ═══ Functions below can't be called on a frozen object! ═══

/// The function can be defined, but it won't be callable on a frozen object. /// Only immutable references are allowed. public fun message_mut(c: &mut Config): &mut String { &mut c.message }

/// Deletes the Config object, takes it by value. /// Can't be called on a frozen object! public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() } ```
```

To summarize:

Since the transfer::freeze_object signature accepts any type with the key ability, it can take an object that was created in the same scope, but it can also take an object that was owned by an account. This means that the freeze_object function can be used to freeze an object that was transferred to the sender. For security concerns, we would not want to freeze the AdminCap object - it would be a security risk to allow access to it to anyone. However, we can freeze the Gift object that was minted and transferred to the recipient:

Single Owner -> Immutable conversion is possible!

```
bash /// Freezes the `Gift` object so it becomes immutable. public fun freeze_gift(gift: Gift) { transfer::freeze_object(gift); }
```

The transfer::share_object function is a public function used to put an object into a shared state. Once an object is shared , it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the key ability :

```
bash // File: sui-framework/sources/transfer.move public fun share_object<T: key>(obj: T);
```

Once an object is shared , it is publicly available as a mutable reference.

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it

works, we will create a function that creates and shares a Config object and then another one that deletes it:

```bash /// Creates a newConfig` object and shares it. public fun create_and_share(message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

// Share the object so it becomes shared.
transfer::share_object(config);

} ```
```

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
bash /// Deletes the `Config` object, takes it by value. /// Can be called on a shared object!
public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() }
```

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.

```
bash // Won't work! public fun transfer_shared(c: Config, to: address) { transfer::transfer(c, to);
}
```

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Transfer

The transfer::transfer function is a public function used to transfer an object to another address. Its signature is as follows, only accepts a type with the key ability and an address of the recipient. Please, note that the object is passed into the function by value , therefore it is moved to the function scope and then moved to the recipient address:

```
bash // File: sui-framework/sources/transfer.move public fun transfer<T: key>(obj: T, recipient:
address);
```

In the next example, you can see how it can be used in a module that defines and sends an object to the transaction sender.

```bash module book::transfer_to_sender {

/// A struct with `key` is an object. The first field is `id: UID`!
public struct AdminCap has key { id: UID }

/// `init` function is a special function that is called when the module
/// is published. It is a good place to create application objects.
fun init(ctx: &mut TxContext) {
    // Create a new `AdminCap` object, in this scope.
    let admin_cap = AdminCap { id: object::new(ctx) };

    // Transfer the object to the transaction sender.
    transfer::transfer(admin_cap, ctx.sender());

    // admin_cap is gone! Can't be accessed anymore.
}

/// Transfers the `AdminCap` object to the `recipient`. Thus, the recipient
/// becomes the owner of the object, and only they can access it.
public fun transfer_admin_cap(cap: AdminCap, recipient: address) {
    transfer::transfer(cap, recipient);
}

} ```
```

When the module is published, the init function will get called, and the AdminCap object which we created there will be transferred to the transaction sender. The ctx.sender() function returns the sender address for the current transaction.

Once the AdminCap has been transferred to the sender, for example, to 0xa11ce , the sender, and only the sender, will be able to

access the object. The object is now account owned .

Account owned objects are a subject to true ownership - only the account owner can access them. This is a fundamental concept in the Sui storage model.

Let's extend the example with a function that uses AdminCap to authorize a mint of a new object and its transfer to another address:

```bash /// SomeGiftobject that the admin canmint_and_transfer`. public struct Gift has key { id: UID }

/// Creates a new `Gift` object and transfers it to the `recipient`. public fun mint_and_transfer(_: &AdminCap, recipient: address, ctx: &mut TxContext) { let gift = Gift { id: object::new(ctx) }; transfer::transfer(gift, recipient); } ```

The mint_and_transfer function is a public function that "could" be called by anyone, but it requires an AdminCap object to be passed as the first argument by reference. Without it, the function will not be callable. This is a simple way to restrict access to privileged functions called Capability . Because the AdminCap object is account owned , only 0xa11ce will be able to call the mint_and_transfer function.

The Gift s sent to recipients will also be account owned , each gift being unique and owned exclusively by the recipient.

A quick recap:

The transfer::freeze_object function is a public function that is used to put an object into an immutable state. Once an object is frozen , it can never be changed, and it can be accessed by anyone by immutable reference.

The function signature is as follows, only accepts a type with the key ability . Just like all other storage functions, it takes the object by value :

```bash // File: sui-framework/sources/transfer.move public fun freeze_object<T: key>(obj: T);
```

Let's expand on the previous example and add a function that allows the admin to create a Config object and freeze it:

```bash /// SomeConfigobject that the admin cancreate_and_freeze`. public struct Config has key { id: UID, message: String }

/// Creates a new `Config` object and freezes it. public fun create_and_freeze(_: &AdminCap, message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

// Freeze the object so it becomes immutable.
transfer::freeze_object(config);

}

/// Returns the message from the `Config` object. /// Can access the object by immutable reference! public fun message(c: &Config): String { c.message } ```

Config is an object that has a message field, and the create_and_freeze function creates a new Config and freezes it. Once the object is frozen, it can be accessed by anyone by immutable reference. The message function is a public function that returns the message from the Config object. Config is now publicly available by its ID, and the message can be read by anyone.

Function definitions are not connected to the object's state. It is possible to define a function that takes a mutable reference to an object that is used as frozen. However, it won't be callable on a frozen object.

The message function can be called on an immutable Config object, however, two functions below are not callable on a frozen object:

```bash // ═══ Functions below can't be called on a frozen object! ═══

/// The function can be defined, but it won't be callable on a frozen object. /// Only immutable references are allowed. public fun message_mut(c: &mut Config): &mut String { &mut c.message }

/// Deletes the `Config` object, takes it by value. /// Can't be called on a frozen object! public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() } ```

To summarize:

Since the transfer::freeze_object signature accepts any type with the key ability, it can take an object that was created in the same
```

scope, but it can also take an object that was owned by an account. This means that the freeze_object function can be used to freeze an object that was transferred to the sender. For security concerns, we would not want to freeze the AdminCap object - it would be a security risk to allow access to it to anyone. However, we can freeze the Gift object that was minted and transferred to the recipient:

Single Owner -> Immutable conversion is possible!

```
bash /// Freezes the `Gift` object so it becomes immutable. public fun freeze_gift(gift: Gift) {
transfer::freeze_object(gift); }
```

The transfer::share_object function is a public function used to put an object into a shared state. Once an object is shared , it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the key ability :

```
bash // File: sui-framework/sources/transfer.move public fun share_object<T: key>(obj: T);
```

Once an object is shared , it is publicly available as a mutable reference.

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

``bash /// Creates a newConfig` object and shares it. public fun create_and_share(message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

```
// Share the object so it becomes shared.
transfer::share_object(config);
```

} ```

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
bash /// Deletes the `Config` object, takes it by value. /// Can be called on a shared object!
public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() }
```

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.

```
bash // Won't work! public fun transfer_shared(c: Config, to: address) { transfer::transfer(c, to);
}
```

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Freeze

The transfer::freeze_object function is a public function that is used to put an object into an immutable state. Once an object is frozen , it can never be changed, and it can be accessed by anyone by immutable reference.

The function signature is as follows, only accepts a type with the key ability . Just like all other storage functions, it takes the object by value :

```
bash // File: sui-framework/sources/transfer.move public fun freeze_object<T: key>(obj: T);
```

Let's expand on the previous example and add a function that allows the admin to create a Config object and freeze it:

``bash /// SomeConfigobject that the admin cancreate_and_freeze`. public struct Config has key { id: UID, message: String }

/// Creates a new `Config` object and freezes it. public fun create_and_freeze(_: &AdminCap, message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

```
// Freeze the object so it becomes immutable.
transfer::freeze_object(config);

}
```

/// Returns the message from the `Config` object. /// Can access the object by immutable reference! public fun message(c: &Config): String { c.message } ```

Config is an object that has a message field, and the create_and_freeze function creates a new Config and freezes it. Once the object is frozen, it can be accessed by anyone by immutable reference. The message function is a public function that returns the message from the Config object. Config is now publicly available by its ID, and the message can be read by anyone.

Function definitions are not connected to the object's state. It is possible to define a function that takes a mutable reference to an object that is used as frozen. However, it won't be callable on a frozen object.

The message function can be called on an immutable Config object, however, two functions below are not callable on a frozen object:

```bash // ==== Functions below can't be called on a frozen object! ====

/// The function can be defined, but it won't be callable on a frozen object. /// Only immutable references are allowed. public fun message_mut(c: &mut Config): &mut String { &mut c.message }

/// Deletes the `Config` object, takes it by value. /// Can't be called on a frozen object! public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() } ```

To summarize:

Since the transfer::freeze_object signature accepts any type with the key ability, it can take an object that was created in the same scope, but it can also take an object that was owned by an account. This means that the freeze_object function can be used to freeze an object that was transferred to the sender. For security concerns, we would not want to freeze the AdminCap object - it would be a security risk to allow access to it to anyone. However, we can freeze the Gift object that was minted and transferred to the recipient:

Single Owner -> Immutable conversion is possible!

```
bash /// Freezes the `Gift` object so it becomes immutable. public fun freeze_gift(gift: Gift) {
transfer::freeze_object(gift); }
```

The transfer::share_object function is a public function used to put an object into a shared state. Once an object is shared , it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the key ability :

```
bash // File: sui-framework/sources/transfer.move public fun share_object<T: key>(obj: T);
```

Once an object is shared , it is publicly available as a mutable reference.

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

```bash /// Creates a newConfig` object and shares it. public fun create_and_share(message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

// Share the object so it becomes shared.
transfer::share_object(config);

} ```
```

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
bash /// Deletes the `Config` object, takes it by value. /// Can be called on a shared object!
public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() }
```

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.
```

```
bash // Won't work! public fun transfer_shared(c: Config, to: address) { transfer::transfer(c, to);
}
```

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Owned -> Frozen

Since the transfer::freeze_object signature accepts any type with the key ability, it can take an object that was created in the same scope, but it can also take an object that was owned by an account. This means that the freeze_object function can be used to freeze an object that was transferred to the sender. For security concerns, we would not want to freeze the AdminCap object - it would be a security risk to allow access to it to anyone. However, we can freeze the Gift object that was minted and transferred to the recipient:

Single Owner -> Immutable conversion is possible!

```
bash /// Freezes the `Gift` object so it becomes immutable. public fun freeze_gift(gift: Gift) {
transfer::freeze_object(gift); }
```

The transfer::share_object function is a public function used to put an object into a shared state. Once an object is shared , it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the key ability :

```
bash // File: sui-framework/sources/transfer.move public fun share_object<T: key>(obj: T);
```

Once an object is shared , it is publicly available as a mutable reference.

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

``bash /// Creates a newConfig` object and shares it. public fun create_and_share(message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

```
// Share the object so it becomes shared.
transfer::share_object(config);
```

} ```

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
bash /// Deletes the `Config` object, takes it by value. /// Can be called on a shared object!
public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() }
```

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.

```
bash // Won't work! public fun transfer_shared(c: Config, to: address) { transfer::transfer(c, to);
}
```

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Share

The transfer::share_object function is a public function used to put an object into a shared state. Once an object is shared , it can be accessed by anyone by a mutable reference (hence, immutable too). The function signature is as follows, only accepts a type with the

key ability :

```
bash // File: sui-framework/sources/transfer.move public fun share_object<T: key>(obj: T);
```

Once an object is shared , it is publicly available as a mutable reference.

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

``bash /// Creates a newConfig` object and shares it. public fun create_and_share(message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

```
// Share the object so it becomes shared.
transfer::share_object(config);
```

} ```

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
bash /// Deletes the `Config` object, takes it by value. /// Can be called on a shared object!
public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() }
```

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.

```
bash // Won't work! public fun transfer_shared(c: Config, to: address) { transfer::transfer(c, to);
}
```

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Special Case: Shared Object Deletion

While the shared object can't normally be taken by value, there is one special case where it can - if the function that takes it deletes the object. This is a special case in the Sui storage model, and it is used to allow the deletion of shared objects. To show how it works, we will create a function that creates and shares a Config object and then another one that deletes it:

``bash /// Creates a newConfig` object and shares it. public fun create_and_share(message: String, ctx: &mut TxContext) { let config = Config { id: object::new(ctx), message };

```
// Share the object so it becomes shared.
transfer::share_object(config);
```

} ```

The create_and_share function creates a new Config object and shares it. The object is now publicly available as a mutable reference. Let's create a function that deletes the shared object:

```
bash /// Deletes the `Config` object, takes it by value. /// Can be called on a shared object!
public fun delete_config(c: Config) { let Config { id, message: _ } = c; id.delete() }
```

The delete_config function takes the Config object by value and deletes it, and the Sui Verifier would allow this call. However, if the function returned the Config object back or attempted to freeze or transfer it, the Sui Verifier would reject the transaction.

```
bash // Won't work! public fun transfer_shared(c: Config, to: address) { transfer::transfer(c, to);
}
```

To summarize:

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer

restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.

Next Steps

Now that you know main features of the transfer module, you can start building more complex applications on Sui that involve storage operations. In the next chapter, we will cover the Store Ability which allows storing data inside objects and relaxes transfer restrictions which we barely touched on here. And after that we will cover the UID and ID types which are the most important types in the Sui storage model.