The Move Book

Storage Operations that we described in the previous sections are restricted by default - they can only be called in the module defining the object. In other terms, the type must be internal to the module to be used in storage operations. This restriction is implemented in the Sui Verifier and is enforced at the bytecode level.

However, to allow objects to be transferred and stored in other modules, these restrictions can be relaxed. The sui::transfer module offers a set of public_* functions that allow calling storage operations in other modules. The functions are prefixed with public_ and are available to all modules and transactions.

The sui::transfer module provides the following public functions. They are almost identical to the ones we already covered, but can be called from any module.

To illustrate the usage of these functions, consider the following example: module A defines an ObjectK with key and ObjectKS with key + store abilities, and module B tries to implement a transfer function for these objects.

In this example we use transfer::transfer , but the behavior is identical for share_object and freeze_object functions.

To expand on the example above:

The decision on whether to add the store ability to a type should be made carefully. On one hand, it is de-facto a requirement for the type to be usable by other applications. On the other hand, it allows wrapping and changing the intended storage model. For example, a character may be intended to be owned by accounts, but with the store ability it can be frozen (cannot be shared - this transition is restricted).

Public Storage Operations

The sui::transfer module provides the following public functions. They are almost identical to the ones we already covered, but can be called from any module.

``bash // File: sui-framework/sources/transfer.move /// Public version of thetransfer` function. public fun public_transfer(object: T, to: address) {}

/// Public version of the `share_object` function. public fun public_share_object(object: T) {}

/// Public version of the `freeze_object` function. public fun public_freeze_object(object: T) {} ```

To illustrate the usage of these functions, consider the following example: module A defines an ObjectK with key and ObjectKS with key + store abilities, and module B tries to implement a transfer function for these objects.

In this example we use transfer::transfer , but the behavior is identical for share_object and freeze_object functions.

``bash /// DefinesObjectKandObjectKSwithkeyandkey + store` /// abilities respectively module book::transfer_a;

public struct ObjectK has key { id: UID } public struct ObjectKS has key, store { id: UID } ```

``bash /// Imports theObjectKandObjectKStypes fromtransfer_aand attempts /// to implement differenttransfer` functions for them module book::transfer_b;

// types are not internal to this module use book::transfer_a::{ObjectK, ObjectKS};

// Fails! ObjectK is not `store`, and ObjectK is not internal to this module public fun transfer_k(k: ObjectK, to: address) { sui::transfer::transfer(k, to); }

// Fails! ObjectKS has `store` but the function is not public public fun transfer_ks(ks: ObjectKS, to: address) { sui::transfer::transfer(ks, to); }

// Fails! ObjectK is not `store`, `public_transfer` requires `store` public fun public_transfer_k(k: ObjectK) { sui::transfer::public_transfer(k); }

// Works! ObjectKS has `store` and the function is public public fun public_transfer_ks(y: ObjectKS, to: address) { sui::transfer::public_transfer(y, to); } ```

To expand on the example above:

The decision on whether to add the store ability to a type should be made carefully. On one hand, it is de-facto a requirement for the type to be usable by other applications. On the other hand, it allows wrapping and changing the intended storage model. For example, a character may be intended to be owned by accounts, but with the store ability it can be frozen (cannot be shared - this transition is restricted).

Implications of

The decision on whether to add the store ability to a type should be made carefully. On one hand, it is de-facto a requirement for the type to be usable by other applications. On the other hand, it allows wrapping and changing the intended storage model. For example, a character may be intended to be owned by accounts, but with the store ability it can be frozen (cannot be shared - this transition is restricted).