Wrapped Objects

In many programming languages, you organize data structures in layers by nesting complex data structures in another data structure. In Move, you can organize data structures by putting a field of struct type in another, like the following:

To embed a struct type in a Sui object struct (with a key ability), the struct type must have the store ability.

In the preceding example, Bar is a normal struct, but it is not a Sui object since it doesn't have the key ability.

The following code turns Bar into an object, which you can still wrap in Foo :

Now Bar is also a Sui object type. If you put a Sui object of type Bar into a Sui object of type Foo , the object type Foo wraps the object type Bar . The object type Foo is the wrapper or wrapping object.

There are some interesting consequences of wrapping a Sui object into another. When an object is wrapped, the object no longer exists independently on-chain. You can no longer look up the object by its ID. The object becomes part of the data of the object that wraps it. Most importantly, you can no longer pass the wrapped object as an argument in any way in Sui Move calls. The only access point is through the wrapping object.

It is not possible to create circular wrapping behavior, where A wraps B, B wraps C, and C also wraps A.

At some point, you can then take out the wrapped object and transfer it to an address, modify it, delete it, or freeze it. This is called unwrapping . When an object is unwrapped , it becomes an independent object again, and can be accessed directly on-chain. There is also an important property about wrapping and unwrapping: the object's ID stays the same across wrapping and unwrapping.

There are a few ways to wrap a Sui object into another Sui object, and their use cases are typically different. This section describes three different ways to wrap a Sui object with typical use cases.

If you put a Sui object type directly as a field in another Sui object type (as in the preceding example), it is called direct wrapping . The most important property achieved through direct wrapping is that the wrapped object cannot be unwrapped unless the wrapping object is destroyed. In the preceding example, to make Bar a standalone object again, unpack (and hence delete) the Foo object. Direct wrapping is the best way to implement object locking, which is to lock an object with constrained access. You can unlock it only through specific contract calls.

The following example implementation of a trusted swap demonstrates how to use direct wrapping. Assume there is an NFT-style Object type that has scarcity and style . In this example, scarcity determines how rare the object is (presumably the more scarce the higher its market value), and style determines the object content/type or how it's rendered. If you own some of these objects and want to trade your objects with others, you want to make sure it's a fair trade. You are willing to trade an object only with another one that has identical scarcity , but want a different style (so that you can collect more styles).

First, define such an object type:

In a real application, you might make sure that there is a limited supply of the objects, and there is a mechanism to mint them to a list of owners. For simplicity and demonstration purposes, this example simplifies creation:

You can also enable a swap/trade between your object and others' objects. For example, define a function that takes two objects from two addresses and swaps their ownership. But this doesn't work in Sui. Only object owners can send a transaction to mutate the object. So one person cannot send a transaction that would swap their own object with someone else's object.

Another common solution is to send your object to a pool - such as an NFT marketplace or a staking pool - and perform the swap in the pool (either right away, or later when there is demand). Other chapters explore the concept of shared objects that can be mutated by anyone, and show how it enables anyone to operate in a shared object pool. This chapter focuses on how to achieve the same effect using owned objects. Transactions using only owned objects are faster and less expensive (in terms of gas) than using shared objects, since they do not require consensus in Sui.

To swap objects, the same address must own both objects. Anyone who wants to swap their object can send their objects to the third party, such as a site that offers swapping services, and the third party helps perform the swap and send the objects to the appropriate owner. To ensure that you retain custody of your objects (such as coins and NFTs) and not give full custody to the third party, use direct wrapping. To define a wrapper object type:

SwapRequest defines a Sui object type, wraps the object to swap, and tracks the original owner of the object. You might need to also pay the third party some fee for this swap. To define an interface to request a swap by someone who owns an Object :

In the preceding function, you must pass the object by value so that it's fully consumed and wrapped into SwapRequest to request swapping an object . The example also provides a fee (in the type of Coin) and checks that the fee is sufficient. The example turns Coin into Balance when it's put into the wrapper object. This is because Coin is a Sui object type and used only to pass around as Sui objects (such as transaction inputs or objects sent to addresses). For coin balances that need to be embedded in other structs, use Balance instead because to avoid the overhead of carrying around an unnecessary UID field.

The wrapper object is then sent to the service operator, with the address specified in the call as service .

The function interface for the function that the service operator can call to perform a swap between two objects sent from two addresses resembles:

Where s1 and s2 are two wrapped objects that were sent from different object owners to the service operator. Both wrapped objects are passed by value because they eventually need to be unpacked .

First, unpack the two object to obtain the inner fields:

Then, check that the swap is legitimate (the two objects have identical scarcity and different styles):

To perform the actual swap:

The preceding code sends o1 to the original owner of o2 , and sends o2 to the original owner of o1 . The service can then delete the wrapping SwapRequest objects:

Finally, the service merges together the fee1 and fee2 , and returns it. The service provider can then turn it into a coin, or merge it into some larger pool where it collects all fees:

After this call, the two objects are swapped and the service provider takes the service fee.

Since the contract defined only one way to deal with SwapRequest - execute_swap - there is no other way the service operator can interact with SwapRequest despite its ownership.

Find the full source code in the trusted_swap example.

To view a more complex example of how to use direct wrapping, see the escrow example.

When Sui object type Bar is directly wrapped into Foo , there is not much flexibility: a Foo object must have a Bar object in it, and to take out the Bar object you must destroy the Foo object. However, for more flexibility, the wrapping type might not always have the wrapped object in it, and the wrapped object might be replaced with a different object at some point.

To demonstrate this use case, design a simple game character: A warrior with a sword and shield. A warrior might have a sword and shield, or might not have either. The warrior should be able to add a sword and shield, and replace the current ones at any time. To design this, define a SimpleWarrior type:

Each SimpleWarrior type has an optional sword and shield wrapped in it, defined as:

When you create a new warrior, set the sword and shield to none to indicate there is no equipment yet:

You can then define functions to equip new swords or new shields:

The function in the preceding example passes a warrior as a mutable reference of SimpleWarrior , and passes a sword by value to wrap it into the warrior .

Note that because Sword is a Sui object type without drop ability, if the warrior already has a sword equipped, the warrior can't drop that sword. If you call option::fill without first checking and taking out the existing sword, an error occurs. In equip_sword , first check whether there is already a sword equipped. If so, remove it out and send it back to the sender. To a player, this returns an equipped sword to their inventory when they equip the new sword.

Find the source code in the simple_warrior example.

To view a more complex example, see hero .

The concept of wrapping objects in a vector field of another Sui object is very similar to wrapping through Option : an object can contain 0, 1, or many of the wrapped objects of the same type.

Wrapping through vector resembles:

The preceding example wraps a vector of Pet in Farm , and can be accessed only through the Farm object.