Transfer to Object

You can transfer objects to an object ID in the same way you transfer objects to an address, using the same functions. This is because Sui does not distinguish between the 32-byte ID of an address and the 32-byte ID of an object (which are guaranteed not to overlap). The transfer to object operation takes advantage of this feature, allowing you to provide an object ID as the address input of a transfer operation.

Because of the identical ID structure, you can use an object ID for the address field when transferring an object. In fact, all functionality around address-owned objects works the same for objects owned by other objects, you just replace the address with the object ID.

When you transfer an object to another object, you're basically establishing a form of parent-child authentication relationship. Objects that you have transferred to another object can be received by the (possibly transitive) owner of the parent object. The module that defines the type of the parent (receiving) object also defines the access control for receiving a child object.

These restrictions for accessing sent child objects are enforced dynamically by providing mutable access to the parent object's UID during the execution of the transaction. Because of this, you can transfer objects to and receive them from owned objects, dynamic field objects, wrapped objects, and shared objects.

One of the benefits of the transfer to object operation is the ability to have a stable ID for an on-chain wallet or account, for example. The transfer of the object doesn't affect its ID, regardless of the state of the object that you send it to. When you transfer an object, all of that object's child objects move with it, and the object's address remains the same whether you transfer it, wrap it, or hold it as a dynamic field.

Just like with normal object transfers, you must make sure that the object ID exists that you are transferring the object to. Additionally, make sure that the object that you are transferring to is not immutable. You can't access an object transferred to an immutable object.

Be aware of both the type of the object you are transferring to and the object that is being transferred. The object that is transferred to (parent) can always :

If the object being transferred has the key ability only, then:

Transferring an object to an object ID results in the same result as if you transferred the object to an address - the object's owner is the 32-byte address or object ID provided. Additionally, because there is no difference in the result of the object transfer, you can use existing RPC methods such as getOwnedObjects on the 32-byte ID. If the ID represents an address, then the method returns the objects owned by that address. If the ID is an object ID, then the method returns the objects the object ID owns (transferred objects).

After an object c has been sent to another object p , p must then receive c to do anything with it. To receive the object c , a Receiving(o: ObjectRef) argument type for programmable transaction blocks (PTBs) is used that takes an object reference containing the to-be-received object's ObjectID , Version , and Digest (just as owned object arguments for PTBs do). However, Receiving PTB arguments are not passed as an owned value or mutable reference within the transaction.

To explain further, look at the core of the receiving interface in Move, which is defined in the transfer module in the Sui framework:

Each Receiving argument referring to a sent object of type T in a PTB results in exactly one argument with a Move type of sui::transfer::Receiving . You can then use this argument to receive the sent object of type T with the transfer::receive function.

When you call the transfer::receive function, you must pass a mutable reference to the parent object's UID . You can't get a mutable reference to the UID of an object, though, unless the defining module of the object exposes it. Consequently, the module that defines the type of the parent object that is receiving the child object defines access control policies and other restrictions on receiving objects that are sent to it. See the authorization example for a demonstration of this pattern. The fact that the passed-in UID actually owns the object referenced by the Receiving parameter is dynamically checked and enforced. This allows access to objects that have been sent to, for example, dynamic fields where the ownership chain can only be established dynamically.

Because sui::transfer::Receiving has only the drop ability, the existence of a Receiving argument represents the ability, but not the obligation to receive the object of type T specified by the object reference in the PTB Receiving argument during that transaction. You can use some, none, or all Receiving arguments in a PTB without issue. Any object that corresponds to a Receiving argument remains untouched (in particular, its object reference remain the same) unless it is received.

Just like with custom transfer policies , Sui allows for the definition of custom receivership rules for key -only objects. In particular, you can use the transfer::receive function only on objects defined in the same module as the call to transfer::receive --just like you can

use the transfer::transfer function only on objects defined in the module where it's being used.

Similarly for objects that also have the store ability, anyone can use the transfer::public_receive function to receive them--just like transfer::public_transfer can transfer any objects that have the store ability on them.

This coupled with the fact that the parent object can always define custom rules around receivership means that you must consider the following matrix of permissions around receiving objects and the abilities of the object being sent based on the child object's abilities:

Just like with custom transfer policies, you can use and couple these restrictions to create powerful expressions. For example, you can implement soul-bound objects using both custom transfer and receivership rules.

When creating transactions, you interact with Receiving transaction inputs almost exactly as you would with other object arguments in the Sui TypeScript SDK. For example, if in the Simple Account example that follows you want to send a transaction that receives a coin object with ID 0xc0ffee that was sent to your account at 0xcafe , you can do the following using either the Sui TypeScript SDK or Sui Rust SDK:

Additionally, just as with object arguments that also have an ObjectRef constructor where you can provide an explicit object ID, version, and digest, there is also a ReceivingRef constructor that takes the same arguments corresponding to a receiving argument.

The following examples demonstrate receiving previously sent objects.

Generally, if you want to allow receiving sent objects from shared objects that are defined in the module, add dynamic authorization checks; otherwise, anyone could receive sent objects. In this example, a shared object (SharedObject) holds a counter that anyone can increment, but only the address 0xB0B can receive objects from the shared object.

Because the receive_object function is generic over the object being received, it can only receive objects that are both key and store . receive_object must also use the transfer::public_receive function to receive the object and not transfer::receive because you can only use receive on objects defined in the current module.

This example defines a basic account-type model where an Account object holds its coin balances in different dynamic fields. This Account is also transferable to a different address or object.

Importantly, the address that coins are to be sent with this Account object remains the same regardless of whether the Account object is transferred, wrapped (for example, in an escrow account), or moved into a dynamic field. In particular, there is a stable ID for a given Account object across the object's lifecycle, regardless of any ownership changes.

The ability to control the rules about how and when an object can be received, and how and when it can be transferred allows us to define a type of "soul-bound" object that can be used by value in a transaction, but it must always stay in the same place, or be returned to the same object.

You can implement a simple version of this with the following module where the get_object function receives the soul-bound object and creates a receipt that must be destroyed in the transaction in order for it to execute successfully. However, in order to destroy the receipt, the object that was received must be transferred back to the object it was received from in the transaction using the return_object function.

Transferring to object

Just like with normal object transfers, you must make sure that the object ID exists that you are transferring the object to. Additionally, make sure that the object that you are transferring to is not immutable. You can't access an object transferred to an immutable object.

Be aware of both the type of the object you are transferring to and the object that is being transferred. The object that is transferred to (parent) can always :

If the object being transferred has the key ability only, then:

Transferring an object to an object ID results in the same result as if you transferred the object to an address - the object's owner is the 32-byte address or object ID provided. Additionally, because there is no difference in the result of the object transfer, you can use existing RPC methods such as getOwnedObjects on the 32-byte ID. If the ID represents an address, then the method returns the objects owned by that address. If the ID is an object ID, then the method returns the objects the object ID owns (transferred objects).

After an object c has been sent to another object p , p must then receive c to do anything with it. To receive the object c , a

Receiving(o: ObjectRef) argument type for programmable transaction blocks (PTBs) is used that takes an object reference containing the to-be-received object's ObjectID , Version , and Digest (just as owned object arguments for PTBs do). However, Receiving PTB arguments are not passed as an owned value or mutable reference within the transaction.

To explain further, look at the core of the receiving interface in Move, which is defined in the transfer module in the Sui framework:

Each Receiving argument referring to a sent object of type T in a PTB results in exactly one argument with a Move type of sui::transfer::Receiving . You can then use this argument to receive the sent object of type T with the transfer::receive function.

When you call the transfer::receive function, you must pass a mutable reference to the parent object's UID . You can't get a mutable reference to the UID of an object, though, unless the defining module of the object exposes it. Consequently, the module that defines the type of the parent object that is receiving the child object defines access control policies and other restrictions on receiving objects that are sent to it. See the authorization example for a demonstration of this pattern. The fact that the passed-in UID actually owns the object referenced by the Receiving parameter is dynamically checked and enforced. This allows access to objects that have been sent to, for example, dynamic fields where the ownership chain can only be established dynamically.

Because sui::transfer::Receiving has only the drop ability, the existence of a Receiving argument represents the ability, but not the obligation to receive the object of type T specified by the object reference in the PTB Receiving argument during that transaction. You can use some, none, or all Receiving arguments in a PTB without issue. Any object that corresponds to a Receiving argument remains untouched (in particular, its object reference remain the same) unless it is received.

Just like with custom transfer policies , Sui allows for the definition of custom receivership rules for key -only objects. In particular, you can use the transfer::receive function only on objects defined in the same module as the call to transfer::receive --just like you can use the transfer::transfer function only on objects defined in the module where it's being used.

Similarly for objects that also have the store ability, anyone can use the transfer::public_receive function to receive them--just like transfer::public_transfer can transfer any objects that have the store ability on them.

This coupled with the fact that the parent object can always define custom rules around receivership means that you must consider the following matrix of permissions around receiving objects and the abilities of the object being sent based on the child object's abilities:

Just like with custom transfer policies, you can use and couple these restrictions to create powerful expressions. For example, you can implement soul-bound objects using both custom transfer and receivership rules.

When creating transactions, you interact with Receiving transaction inputs almost exactly as you would with other object arguments in the Sui TypeScript SDK. For example, if in the Simple Account example that follows you want to send a transaction that receives a coin object with ID 0xc0ffee that was sent to your account at 0xcafe , you can do the following using either the Sui TypeScript SDK or Sui Rust SDK:

Additionally, just as with object arguments that also have an ObjectRef constructor where you can provide an explicit object ID, version, and digest, there is also a ReceivingRef constructor that takes the same arguments corresponding to a receiving argument.

The following examples demonstrate receiving previously sent objects.

Generally, if you want to allow receiving sent objects from shared objects that are defined in the module, add dynamic authorization checks; otherwise, anyone could receive sent objects. In this example, a shared object (SharedObject) holds a counter that anyone can increment, but only the address 0xB0B can receive objects from the shared object.

Because the receive_object function is generic over the object being received, it can only receive objects that are both key and store . receive_object must also use the transfer::public_receive function to receive the object and not transfer::receive because you can only use receive on objects defined in the current module.

This example defines a basic account-type model where an Account object holds its coin balances in different dynamic fields. This Account is also transferable to a different address or object.

Importantly, the address that coins are to be sent with this Account object remains the same regardless of whether the Account object is transferred, wrapped (for example, in an escrow account), or moved into a dynamic field. In particular, there is a stable ID for a given Account object across the object's lifecycle, regardless of any ownership changes.

The ability to control the rules about how and when an object can be received, and how and when it can be transferred allows us to define a type of "soul-bound" object that can be used by value in a transaction, but it must always stay in the same place, or be returned to the same object.

You can implement a simple version of this with the following module where the get_object function receives the soul-bound object and creates a receipt that must be destroyed in the transaction in order for it to execute successfully. However, in order to destroy the receipt, the object that was received must be transferred back to the object it was received from in the transaction using the return_object function.

Receiving objects

After an object c has been sent to another object p , p must then receive c to do anything with it. To receive the object c , a Receiving(o: ObjectRef) argument type for programmable transaction blocks (PTBs) is used that takes an object reference containing the to-be-received object's ObjectID , Version , and Digest (just as owned object arguments for PTBs do). However, Receiving PTB arguments are not passed as an owned value or mutable reference within the transaction.

To explain further, look at the core of the receiving interface in Move, which is defined in the transfer module in the Sui framework:

Each Receiving argument referring to a sent object of type T in a PTB results in exactly one argument with a Move type of sui::transfer::Receiving . You can then use this argument to receive the sent object of type T with the transfer::receive function.

When you call the transfer::receive function, you must pass a mutable reference to the parent object's UID . You can't get a mutable reference to the UID of an object, though, unless the defining module of the object exposes it. Consequently, the module that defines the type of the parent object that is receiving the child object defines access control policies and other restrictions on receiving objects that are sent to it. See the authorization example for a demonstration of this pattern. The fact that the passed-in UID actually owns the object referenced by the Receiving parameter is dynamically checked and enforced. This allows access to objects that have been sent to, for example, dynamic fields where the ownership chain can only be established dynamically.

Because sui::transfer::Receiving has only the drop ability, the existence of a Receiving argument represents the ability, but not the obligation to receive the object of type T specified by the object reference in the PTB Receiving argument during that transaction. You can use some, none, or all Receiving arguments in a PTB without issue. Any object that corresponds to a Receiving argument remains untouched (in particular, its object reference remain the same) unless it is received.

Just like with custom transfer policies , Sui allows for the definition of custom receivership rules for key -only objects. In particular, you can use the transfer::receive function only on objects defined in the same module as the call to transfer::receive --just like you can use the transfer::transfer function only on objects defined in the module where it's being used.

Similarly for objects that also have the store ability, anyone can use the transfer::public_receive function to receive them--just like transfer::public_transfer can transfer any objects that have the store ability on them.

This coupled with the fact that the parent object can always define custom rules around receivership means that you must consider the following matrix of permissions around receiving objects and the abilities of the object being sent based on the child object's abilities:

Just like with custom transfer policies, you can use and couple these restrictions to create powerful expressions. For example, you can implement soul-bound objects using both custom transfer and receivership rules.

When creating transactions, you interact with Receiving transaction inputs almost exactly as you would with other object arguments in the Sui TypeScript SDK. For example, if in the Simple Account example that follows you want to send a transaction that receives a coin object with ID 0xc0ffee that was sent to your account at 0xcafe , you can do the following using either the Sui TypeScript SDK or Sui Rust SDK:

Additionally, just as with object arguments that also have an ObjectRef constructor where you can provide an explicit object ID, version, and digest, there is also a ReceivingRef constructor that takes the same arguments corresponding to a receiving argument.

The following examples demonstrate receiving previously sent objects.

Generally, if you want to allow receiving sent objects from shared objects that are defined in the module, add dynamic authorization checks; otherwise, anyone could receive sent objects. In this example, a shared object (SharedObject) holds a counter that anyone can increment, but only the address 0xB0B can receive objects from the shared object.

Because the receive_object function is generic over the object being received, it can only receive objects that are both key and store . receive_object must also use the transfer::public_receive function to receive the object and not transfer::receive because you can only use receive on objects defined in the current module.

This example defines a basic account-type model where an Account object holds its coin balances in different dynamic fields. This Account is also transferable to a different address or object.

Importantly, the address that coins are to be sent with this Account object remains the same regardless of whether the Account object is transferred, wrapped (for example, in an escrow account), or moved into a dynamic field. In particular, there is a stable ID for a given Account object across the object's lifecycle, regardless of any ownership changes.

The ability to control the rules about how and when an object can be received, and how and when it can be transferred allows us to define a type of "soul-bound" object that can be used by value in a transaction, but it must always stay in the same place, or be returned to the same object.

You can implement a simple version of this with the following module where the get_object function receives the soul-bound object and creates a receipt that must be destroyed in the transaction in order for it to execute successfully. However, in order to destroy the receipt, the object that was received must be transferred back to the object it was received from in the transaction using the return_object function.

Custom receiving rules

Just like with custom transfer policies , Sui allows for the definition of custom receivership rules for key -only objects. In particular, you can use the transfer::receive function only on objects defined in the same module as the call to transfer::receive --just like you can use the transfer::transfer function only on objects defined in the module where it's being used.

Similarly for objects that also have the store ability, anyone can use the transfer::public_receive function to receive them--just like transfer::public_transfer can transfer any objects that have the store ability on them.

This coupled with the fact that the parent object can always define custom rules around receivership means that you must consider the following matrix of permissions around receiving objects and the abilities of the object being sent based on the child object's abilities:

Just like with custom transfer policies, you can use and couple these restrictions to create powerful expressions. For example, you can implement soul-bound objects using both custom transfer and receivership rules.

When creating transactions, you interact with Receiving transaction inputs almost exactly as you would with other object arguments in the Sui TypeScript SDK. For example, if in the Simple Account example that follows you want to send a transaction that receives a coin object with ID 0xc0ffee that was sent to your account at 0xcafe , you can do the following using either the Sui TypeScript SDK or Sui Rust SDK:

Additionally, just as with object arguments that also have an ObjectRef constructor where you can provide an explicit object ID, version, and digest, there is also a ReceivingRef constructor that takes the same arguments corresponding to a receiving argument.

The following examples demonstrate receiving previously sent objects.

Generally, if you want to allow receiving sent objects from shared objects that are defined in the module, add dynamic authorization checks; otherwise, anyone could receive sent objects. In this example, a shared object (SharedObject) holds a counter that anyone can increment, but only the address 0xB0B can receive objects from the shared object.

Because the receive_object function is generic over the object being received, it can only receive objects that are both key and store . receive_object must also use the transfer::public_receive function to receive the object and not transfer::receive because you can only use receive on objects defined in the current module.

This example defines a basic account-type model where an Account object holds its coin balances in different dynamic fields. This Account is also transferable to a different address or object.

Importantly, the address that coins are to be sent with this Account object remains the same regardless of whether the Account object is transferred, wrapped (for example, in an escrow account), or moved into a dynamic field. In particular, there is a stable ID for a given Account object across the object's lifecycle, regardless of any ownership changes.

The ability to control the rules about how and when an object can be received, and how and when it can be transferred allows us to define a type of "soul-bound" object that can be used by value in a transaction, but it must always stay in the same place, or be returned to the same object.

You can implement a simple version of this with the following module where the get_object function receives the soul-bound object and creates a receipt that must be destroyed in the transaction in order for it to execute successfully. However, in order to destroy the receipt, the object that was received must be transferred back to the object it was received from in the transaction using the return_object function.

Using SDKs

When creating transactions, you interact with Receiving transaction inputs almost exactly as you would with other object arguments in the Sui TypeScript SDK. For example, if in the Simple Account example that follows you want to send a transaction that receives a coin object with ID 0xc0ffee that was sent to your account at 0xcafe , you can do the following using either the Sui TypeScript SDK or Sui Rust SDK:

Additionally, just as with object arguments that also have an ObjectRef constructor where you can provide an explicit object ID, version, and digest, there is also a ReceivingRef constructor that takes the same arguments corresponding to a receiving argument.

The following examples demonstrate receiving previously sent objects.

Generally, if you want to allow receiving sent objects from shared objects that are defined in the module, add dynamic authorization checks; otherwise, anyone could receive sent objects. In this example, a shared object (SharedObject) holds a counter that anyone can increment, but only the address 0xB0B can receive objects from the shared object.

Because the receive_object function is generic over the object being received, it can only receive objects that are both key and store . receive_object must also use the transfer::public_receive function to receive the object and not transfer::receive because you can only use receive on objects defined in the current module.

This example defines a basic account-type model where an Account object holds its coin balances in different dynamic fields. This Account is also transferable to a different address or object.

Importantly, the address that coins are to be sent with this Account object remains the same regardless of whether the Account object is transferred, wrapped (for example, in an escrow account), or moved into a dynamic field. In particular, there is a stable ID for a given Account object across the object's lifecycle, regardless of any ownership changes.

The ability to control the rules about how and when an object can be received, and how and when it can be transferred allows us to define a type of "soul-bound" object that can be used by value in a transaction, but it must always stay in the same place, or be returned to the same object.

You can implement a simple version of this with the following module where the get_object function receives the soul-bound object and creates a receipt that must be destroyed in the transaction in order for it to execute successfully. However, in order to destroy the receipt, the object that was received must be transferred back to the object it was received from in the transaction using the return_object function.

Examples

The following examples demonstrate receiving previously sent objects.

Generally, if you want to allow receiving sent objects from shared objects that are defined in the module, add dynamic authorization checks; otherwise, anyone could receive sent objects. In this example, a shared object (SharedObject) holds a counter that anyone can increment, but only the address 0xB0B can receive objects from the shared object.

Because the receive_object function is generic over the object being received, it can only receive objects that are both key and store . receive_object must also use the transfer::public_receive function to receive the object and not transfer::receive because you can only use receive on objects defined in the current module.

This example defines a basic account-type model where an Account object holds its coin balances in different dynamic fields. This Account is also transferable to a different address or object.

Importantly, the address that coins are to be sent with this Account object remains the same regardless of whether the Account object is transferred, wrapped (for example, in an escrow account), or moved into a dynamic field. In particular, there is a stable ID for a given Account object across the object's lifecycle, regardless of any ownership changes.

The ability to control the rules about how and when an object can be received, and how and when it can be transferred allows us to define a type of "soul-bound" object that can be used by value in a transaction, but it must always stay in the same place, or be returned to the same object.

You can implement a simple version of this with the following module where the get_object function receives the soul-bound object and creates a receipt that must be destroyed in the transaction in order for it to execute successfully. However, in order to destroy the receipt, the object that was received must be transferred back to the object it was received from in the transaction using the return_object function.