The Move Book

The UID type is defined in the sui::object module and is a wrapper around an ID which, in turn, wraps the address type. The UIDs on Sui are guaranteed to be unique, and can't be reused after the object was deleted.

New UID is created with the object::new(ctx) function. It takes a mutable reference to TxContext, and returns a new UID.

On Sui, UID acts as a representation of an object, and allows defining behaviors and features of an object. One of the key features - Dynamic Fields - is possible because of the UID type being explicit. Additionally, it allows the Transfer to Object (TTO) , which we will explain later in this chapter.

The UID type is created with the object::new(ctx) function, and it is destroyed with the object::delete(uid) function. The object::delete consumes the UID by value , and it is impossible to delete it unless the value was unpacked from an Object.

The UID does not need to be deleted immediately after the object struct is unpacked. Sometimes it may carry Dynamic Fields or objects transferred to it via Transfer To Object . In such cases, the UID may be kept and stored in a separate object.

The ability to return the UID of an object may be utilized in pattern called proof of deletion . It is a rarely used technique, but it may be useful in some cases, for example, the creator or an application may incentivize the deletion of an object by exchanging the deleted IDs for some reward.

In framework development this method could be used to ignore / bypass certain restrictions on "taking" the object. If there's a container that enforces certain logic on transfers, like Kiosk does, there could be a special scenario of skipping the checks by providing a proof of deletion.

This is one of the open topics for exploration and research, and it may be used in various ways.

When talking about UID we should also mention the ID type. It is a wrapper around the address type, and is used to represent an address-pointer. Usually, ID is used to point at an object, however, there's no restriction, and no guarantee that the ID points to an existing object.

ID can be received as a transaction argument in a Transaction Block . Alternatively, ID can be created from an address value using to_id() function.

TxContext provides the fresh_object_address function which can be utilized to create unique addresses and ID - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.

Fresh UID generation:

New UID is created with the object::new(ctx) function. It takes a mutable reference to TxContext, and returns a new UID.

```
bash let ctx = &mut tx_context::dummy(); let uid = object::new(ctx);
```

On Sui, UID acts as a representation of an object, and allows defining behaviors and features of an object. One of the key features - Dynamic Fields - is possible because of the UID type being explicit. Additionally, it allows the Transfer to Object (TTO) , which we will explain later in this chapter.

The UID type is created with the object::new(ctx) function, and it is destroyed with the object::delete(uid) function. The object::delete consumes the UID by value , and it is impossible to delete it unless the value was unpacked from an Object.

```bash let ctx = &mut tx_context::dummy();

let char = Character { id: object::new(ctx) };

let Character { id } = char; id.delete(); ```

The UID does not need to be deleted immediately after the object struct is unpacked. Sometimes it may carry Dynamic Fields or objects transferred to it via Transfer To Object . In such cases, the UID may be kept and stored in a separate object.

The ability to return the UID of an object may be utilized in pattern called proof of deletion . It is a rarely used technique, but it may be useful in some cases, for example, the creator or an application may incentivize the deletion of an object by exchanging the deleted IDs for some reward.

In framework development this method could be used to ignore / bypass certain restrictions on "taking" the object. If there's a
```

container that enforces certain logic on transfers, like Kiosk does, there could be a special scenario of skipping the checks by providing a proof of deletion.

This is one of the open topics for exploration and research, and it may be used in various ways.

When talking about UID we should also mention the ID type. It is a wrapper around the address type, and is used to represent an address-pointer. Usually, ID is used to point at an object, however, there's no restriction, and no guarantee that the ID points to an existing object.

ID can be received as a transaction argument in a Transaction Block . Alternatively, ID can be created from an address value using to_id() function.

TxContext provides the fresh_object_address function which can be utilized to create unique addresses and ID - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.

UID lifecycle

The UID type is created with the object::new(ctx) function, and it is destroyed with the object::delete(uid) function. The object::delete consumes the UID by value , and it is impossible to delete it unless the value was unpacked from an Object.

\`\`\`bash let ctx = &mut tx_context::dummy();

let char = Character { id: object::new(ctx) };

let Character { id } = char; id.delete(); \`\`\`

The UID does not need to be deleted immediately after the object struct is unpacked. Sometimes it may carry Dynamic Fields or objects transferred to it via Transfer To Object . In such cases, the UID may be kept and stored in a separate object.

The ability to return the UID of an object may be utilized in pattern called proof of deletion . It is a rarely used technique, but it may be useful in some cases, for example, the creator or an application may incentivize the deletion of an object by exchanging the deleted IDs for some reward.

In framework development this method could be used to ignore / bypass certain restrictions on "taking" the object. If there's a container that enforces certain logic on transfers, like Kiosk does, there could be a special scenario of skipping the checks by providing a proof of deletion.

This is one of the open topics for exploration and research, and it may be used in various ways.

When talking about UID we should also mention the ID type. It is a wrapper around the address type, and is used to represent an address-pointer. Usually, ID is used to point at an object, however, there's no restriction, and no guarantee that the ID points to an existing object.

ID can be received as a transaction argument in a Transaction Block . Alternatively, ID can be created from an address value using to_id() function.

TxContext provides the fresh_object_address function which can be utilized to create unique addresses and ID - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.

Keeping the UID

The UID does not need to be deleted immediately after the object struct is unpacked. Sometimes it may carry Dynamic Fields or objects transferred to it via Transfer To Object . In such cases, the UID may be kept and stored in a separate object.

The ability to return the UID of an object may be utilized in pattern called proof of deletion . It is a rarely used technique, but it may be useful in some cases, for example, the creator or an application may incentivize the deletion of an object by exchanging the deleted IDs for some reward.

In framework development this method could be used to ignore / bypass certain restrictions on "taking" the object. If there's a container that enforces certain logic on transfers, like Kiosk does, there could be a special scenario of skipping the checks by providing a proof of deletion.

This is one of the open topics for exploration and research, and it may be used in various ways.

When talking about UID we should also mention the ID type. It is a wrapper around the address type, and is used to represent an address-pointer. Usually, ID is used to point at an object, however, there's no restriction, and no guarantee that the ID points to an existing object.

ID can be received as a transaction argument in a Transaction Block . Alternatively, ID can be created from an address value using to_id() function.

TxContext provides the fresh_object_address function which can be utilized to create unique addresses and ID - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.

Proof of Deletion

The ability to return the UID of an object may be utilized in pattern called proof of deletion . It is a rarely used technique, but it may be useful in some cases, for example, the creator or an application may incentivize the deletion of an object by exchanging the deleted IDs for some reward.

In framework development this method could be used to ignore / bypass certain restrictions on "taking" the object. If there's a container that enforces certain logic on transfers, like Kiosk does, there could be a special scenario of skipping the checks by providing a proof of deletion.

This is one of the open topics for exploration and research, and it may be used in various ways.

When talking about UID we should also mention the ID type. It is a wrapper around the address type, and is used to represent an address-pointer. Usually, ID is used to point at an object, however, there's no restriction, and no guarantee that the ID points to an existing object.

ID can be received as a transaction argument in a Transaction Block . Alternatively, ID can be created from an address value using to_id() function.

TxContext provides the fresh_object_address function which can be utilized to create unique addresses and ID - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.

ID

When talking about UID we should also mention the ID type. It is a wrapper around the address type, and is used to represent an address-pointer. Usually, ID is used to point at an object, however, there's no restriction, and no guarantee that the ID points to an existing object.

ID can be received as a transaction argument in a Transaction Block . Alternatively, ID can be created from an address value using to_id() function.

TxContext provides the fresh_object_address function which can be utilized to create unique addresses and ID - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.

fresh_object_address

TxContext provides the fresh_object_address function which can be utilized to create unique addresses and ID - it may be useful in some application that assign unique identifiers to user actions - for example, an order_id in a marketplace.