# The Move Book

Sui Framework offers a variety of collection types that build on the dynamic fields and dynamic object fields concepts. These collections are designed to be a safer and more understandable way to store and manage dynamic fields and objects.

For each collection type we will specify the primitive they use, and the specific features they offer.

Unlike dynamic (object) fields which operate on UID, collection types have their own type and allow calling associated functions .

All of the collection types share the same set of methods, which are:

All collection types support index syntax for borrow and borrow_mut methods. If you see square brackets in the examples, they are translated into borrow and borrow_mut calls.

In the examples we won't focus on these functions, but rather on the differences between the collection types.

Bag, as the name suggests, acts as a "bag" of heterogeneous values. It is a simple, non-generic type that can store any data. Bag will never allow orphaned fields, as it tracks the number of fields and can't be destroyed if it's not empty.

Due to Bag storing any types, the extra methods it offers is:

Used as a struct field:

Using the Bag:

Defined in the sui::object_bag module. Identical to Bag , but uses dynamic object fields internally. Can only store objects as values.

Table is a typed dynamic collection that has a fixed type for keys and values. It is defined in the sui::table module.

Used as a struct field:

Using the Table:

Defined in the sui::object_table module. Identical to Table , but uses dynamic object fields internally. Can only store objects as values.

This section is coming soon!

## Common Concepts

All of the collection types share the same set of methods, which are:

All collection types support index syntax for borrow and borrow_mut methods. If you see square brackets in the examples, they are translated into borrow and borrow_mut calls.

```bash let hat: &Hat = &bag[b"key"]; let hat_mut: &mut Hat = &mut bag[b"key"];

// is equivalent to let hat: &Hat = bag.borrow(b"key"); let hat_mut: &mut Hat = bag.borrow_mut(b"key"); ```

In the examples we won't focus on these functions, but rather on the differences between the collection types.

Bag, as the name suggests, acts as a "bag" of heterogeneous values. It is a simple, non-generic type that can store any data. Bag will never allow orphaned fields, as it tracks the number of fields and can't be destroyed if it's not empty.

```
bash // File: sui-framework/sources/bag.move public struct Bag has key, store { /// the ID of this bag id: UID, /// the number of key-value pairs in the bag size: u64, }
```

Due to Bag storing any types, the extra methods it offers is:

Used as a struct field:

``bash /// Imported from thesui::bag` module. use sui::bag::{Self, Bag};

/// An example of a `Bag` as a struct field. public struct Carrier has key { id: UID, bag: Bag } ```
```

Using the Bag:

```bash let mut bag = bag::new(ctx);

// bag has the `length` function to get the number of elements assert!(bag.length() == 0, 0);

bag.add(b"my_key", b"my_value".to_string());

// length has changed to 1 assert!(bag.length() == 1, 1);

// in order: `borrow`, `borrow_mut` and `remove` // the value type must be specified let field_ref: &String = &bag[b"my_key"]; let field_mut: &mut String = &mut bag[b"my_key"]; let field: String = bag.remove(b"my_key");

// length is back to 0 - we can unpack bag.destroy_empty(); ```

Defined in the sui::object_bag module. Identical to Bag , but uses dynamic object fields internally. Can only store objects as values.

Table is a typed dynamic collection that has a fixed type for keys and values. It is defined in the sui::table module.

```
bash // File: sui-framework/sources/table.move public struct Table<phantom K: copy + drop + store,
phantom V: store> has key, store { /// the ID of this table id: UID, /// the number of key-value
pairs in the table size: u64, }
```

Used as a struct field:

```bash /// Imported from thesui::table` module. use sui::table::{Self, Table};

/// Some record type with `store` public struct Record has store { /... / }

/// An example of a `Table` as a struct field. public struct UserRegistry has key { id: UID, table: Table } ```

Using the Table:

```bash

# [test] fun test_table() {

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

// Table requires explicit type parameters for the key and value // ...but does it only once in initialization. let mut table = table::new(ctx);

// table has the `length` function to get the number of elements assert!(table.length() == 0, 0);

table.add(@0xa11ce, b"my_value".to_string()); table.add(@0xb0b, b"another_value".to_string());

// length has changed to 2 assert!(table.length() == 2, 2);

// in order: `borrow`, `borrow_mut` and `remove` let addr_ref = &table[@0xa11ce]; let addr_mut = &mut table[@0xa11ce];

// removing both values let _addr = table.remove(@0xa11ce); let _addr = table.remove(@0xb0b);

// length is back to 0 - we can unpack table.destroy_empty(); ```

Defined in the sui::object_table module. Identical to Table , but uses dynamic object fields internally. Can only store objects as values.

This section is coming soon!

## Bag

Bag, as the name suggests, acts as a "bag" of heterogeneous values. It is a simple, non-generic type that can store any data. Bag will never allow orphaned fields, as it tracks the number of fields and can't be destroyed if it's not empty.

```
bash // File: sui-framework/sources/bag.move public struct Bag has key, store { /// the ID of this
```
```

`bag id: UID, /// the number of key-value pairs in the bag size: u64, }`

Due to Bag storing any types, the extra methods it offers is:

Used as a struct field:

``bash /// Imported from thesui::bag` module. use sui::bag::{Self, Bag};

/// An example of a `Bag` as a struct field. public struct Carrier has key { id: UID, bag: Bag } ```

Using the Bag:

```bash let mut bag = bag::new(ctx);

// bag has the `length` function to get the number of elements assert!(bag.length() == 0, 0);

bag.add(b"my_key", b"my_value".to_string());

// length has changed to 1 assert!(bag.length() == 1, 1);

// in order: `borrow`, `borrow_mut` and `remove` // the value type must be specified let field_ref: &String = &bag[b"my_key"]; let field_mut: &mut String = &mut bag[b"my_key"]; let field: String = bag.remove(b"my_key");

// length is back to 0 - we can unpack bag.destroy_empty(); ```

Defined in the sui::object_bag module. Identical to Bag , but uses dynamic object fields internally. Can only store objects as values.

Table is a typed dynamic collection that has a fixed type for keys and values. It is defined in the sui::table module.

`bash // File: sui-framework/sources/table.move public struct Table<phantom K: copy + drop + store, phantom V: store> has key, store { /// the ID of this table id: UID, /// the number of key-value pairs in the table size: u64, }`

Used as a struct field:

``bash /// Imported from thesui::table` module. use sui::table::{Self, Table};

/// Some record type with `store` public struct Record has store { /* ... */ }

/// An example of a `Table` as a struct field. public struct UserRegistry has key { id: UID, table: Table } ```

Using the Table:

```bash

# [test] fun test_table() {

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

// Table requires explicit type parameters for the key and value // ...but does it only once in initialization. let mut table = table::new(ctx);

// table has the `length` function to get the number of elements assert!(table.length() == 0, 0);

table.add(@0xa11ce, b"my_value".to_string()); table.add(@0xb0b, b"another_value".to_string());

// length has changed to 2 assert!(table.length() == 2, 2);

// in order: `borrow`, `borrow_mut` and `remove` let addr_ref = &table[@0xa11ce]; let addr_mut = &mut table[@0xa11ce];

// removing both values let _addr = table.remove(@0xa11ce); let _addr = table.remove(@0xb0b);

// length is back to 0 - we can unpack table.destroy_empty(); ```

Defined in the sui::object_table module. Identical to Table , but uses dynamic object fields internally. Can only store objects as values.

This section is coming soon!

## ObjectBag

Defined in the sui::object_bag module. Identical to Bag , but uses dynamic object fields internally. Can only store objects as values.

Table is a typed dynamic collection that has a fixed type for keys and values. It is defined in the sui::table module.

```
bash // File: sui-framework/sources/table.move public struct Table<phantom K: copy + drop + store,
phantom V: store> has key, store { /// the ID of this table id: UID, /// the number of key-value
pairs in the table size: u64, }
```

Used as a struct field:

``bash /// Imported from thesui::table` module. use sui::table::{Self, Table};

/// Some record type with `store` public struct Record has store { / ... / }

/// An example of a `Table` as a struct field. public struct UserRegistry has key { id: UID, table: Table } ```

Using the Table:

```bash

# [test] fun test_table() {

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

// Table requires explicit type parameters for the key and value // ...but does it only once in initialization. let mut table = table::new(ctx);

// table has the `length` function to get the number of elements assert!(table.length() == 0, 0);

table.add(@0xa11ce, b"my_value".to_string()); table.add(@0xb0b, b"another_value".to_string());

// length has changed to 2 assert!(table.length() == 2, 2);

// in order: `borrow`, `borrow_mut` and `remove` let addr_ref = &table[@0xa11ce]; let addr_mut = &mut table[@0xa11ce];

// removing both values let _addr = table.remove(@0xa11ce); let _addr = table.remove(@0xb0b);

// length is back to 0 - we can unpack table.destroy_empty(); ```
```

Defined in the sui::object_table module. Identical to Table , but uses dynamic object fields internally. Can only store objects as values.

This section is coming soon!

## Table

Table is a typed dynamic collection that has a fixed type for keys and values. It is defined in the sui::table module.

```
bash // File: sui-framework/sources/table.move public struct Table<phantom K: copy + drop + store,
phantom V: store> has key, store { /// the ID of this table id: UID, /// the number of key-value
pairs in the table size: u64, }
```

Used as a struct field:

``bash /// Imported from thesui::table` module. use sui::table::{Self, Table};

/// Some record type with `store` public struct Record has store { / ... / }

/// An example of a `Table` as a struct field. public struct UserRegistry has key { id: UID, table: Table } ```

Using the Table:

```bash

# [test] fun test_table() {

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

// Table requires explicit type parameters for the key and value // ...but does it only once in initialization. let mut table = table::new(ctx);

// table has the `length` function to get the number of elements assert!(table.length() == 0, 0);

table.add(@0xa11ce, b"my_value".to_string()); table.add(@0xb0b, b"another_value".to_string());

// length has changed to 2 assert!(table.length() == 2, 2);

// in order: `borrow`, `borrow_mut` and `remove` let addr_ref = &table[@0xa11ce]; let addr_mut = &mut table[@0xa11ce];

// removing both values let _addr = table.remove(@0xa11ce); let _addr = table.remove(@0xb0b);

// length is back to 0 - we can unpack table.destroy_empty(); ```

Defined in the sui::object_table module. Identical to Table , but uses dynamic object fields internally. Can only store objects as values.

This section is coming soon!

## ObjectTable

Defined in the sui::object_table module. Identical to Table , but uses dynamic object fields internally. Can only store objects as values.

This section is coming soon!

## Summary

This section is coming soon!

## LinkedTable

This section is coming soon!
```