Table and Bag

You can extend existing objects using dynamic fields . Note that it's possible to delete an object that still has (potentially non-drop) dynamic fields. This might not be a concern when adding a small number of statically known additional fields to an object, but is particularly undesirable for on-chain collection types that could be holding unboundedly many key-value pairs as dynamic fields.

This topic describes two such collections -- Table and Bag -- built using dynamic fields, but with additional support to count the number of entries they contain, and protect against accidental deletion when non-empty.

The types and function discussed in this section are built into the Sui framework in modules table and bag . As with dynamic fields, there is also an object_ variant of both: ObjectTable in object_table and ObjectBag in object_bag . The relationship between Table and ObjectTable , and Bag and ObjectBag is the same as between a field and an object field: The former can hold any store type as a value, but objects stored as values are hidden when viewed from external storage. The latter can only store objects as values, but keeps those objects visible at their ID in external storage.

Table is a homogeneous map, meaning that all its keys have the same type as each other (K), and all its values have the same type as each other as well (V). It is created with sui::table::new , which requires access to a &mut TxContext because Table s are objects themselves, which can be transferred, shared, wrapped, or unwrapped, just like any other object.

See sui::object_table::ObjectTable for the object-preserving version of Table .

Bag is a heterogeneous map, so it can hold key-value pairs of arbitrary types (they don't need to match each other). Note that the Bag type does not have any type parameters for this reason. Like Table , Bag is also an object, so creating one with sui::bag::new requires supplying a &mut TxContext to generate an ID.

See sui::bag::ObjectBag for the object-preserving version of Bag .

The following sections explain the collection APIs. They use sui::table as the basis for code examples, with explanations where other modules differ.

All collection types come with the following functions, defined in their respective modules:

These functions add, read, write, and remove entries from the collection, respectively, and all accept keys by value. Table has type parameters for K and V so it is not possible to call these functions with different instantiations of K and V on the same instance of Table , however Bag does not have these type parameters, and so does permit calls with different instantiations on the same instance.

Like with dynamic fields, it is an error to attempt to overwrite an existing key, or access or remove a non-existent key.

The extra flexibility of Bag 's heterogeneity means the type system doesn't statically prevent attempts to add a value with one type, and then borrow or remove it at another type. This pattern fails at runtime, similar to the behavior for dynamic fields.

It is possible to query all collection types for their length and check whether they are empty using the following family of functions:

Bag has these functions, but they are not generic on K and V because Bag does not have these type parameters.

Tables can be queried for key containment with:

The equivalent functions for Bag are:

The first function tests whether bag contains a key-value pair with key k: K , and the second function additionally tests whether its value has type V .

Collection types protect against accidental deletion when they might not be empty. This protection comes from the fact that they do not have drop , so must be explicitly deleted, using this API:

This function takes the collection by value. If it contains no entries, it is deleted, otherwise the call fails. sui::table::Table also has a convenience function:

You can call the convenience function only for tables where the value type also has drop ability, which allows it to delete tables whether they are empty or not.

Note that drop is not called implicitly on eligible tables before they go out of scope. It must be called explicitly, but it is guaranteed to succeed at runtime.

Bag and ObjectBag cannot support drop because they could be holding a variety of types, some of which may have drop and some which may not.

ObjectTable does not support drop because its values must be objects, which cannot be dropped (because they must contain an id: UID field and UID does not have drop).

Equality on collections is based on identity, for example, an instance of a collection type is only considered equal to itself and not to all collections that hold the same entries:

This is unlikely to be the definition of equality that you want.

Related links