The Move Book

This section expands on the Dynamic Fields . Please, read it first to understand the basics of dynamic fields.

Another variation of dynamic fields is dynamic object fields , which have certain differences from regular dynamic fields. In this section, we will cover the specifics of dynamic object fields and explain how they differ from regular dynamic fields.

General recommendation is to avoid using dynamic object fields in favor of (just) dynamic fields, especially if there's no need for direct discovery through the ID. The extra costs of dynamic object fields may not be justified by the benefits they provide.

Dynamic Object Fields are defined in the sui::dynamic_object_fields module in the Sui Framework . They are similar to dynamic fields in many ways, but unlike them, dynamic object fields have an extra constraint on the Value type. The Value must have a combination of key and store , not just store as in the case of dynamic fields.

They're less explicit in their framework definition, as the concept itself is more abstract:

File: sui-framework/sources/dynamic_object_fields.move

Unlike Field type in the Dynamic Fields section, the Wrapper type only stores the name of the field. The value is the object itself, and is not wrapped .

The constraints on the Value type become visible in the methods available for dynamic object fields. Here's the signature for the add function:

The rest of the methods which are identical to the ones in the Dynamic Fields section have the same constraints on the Value type. Let's list them for reference:

Additionally, there is an id method which returns the ID of the Value object without specifying its type.

The main difference between dynamic fields and dynamic object fields is that the latter allows storing only objects as values. This means that you can't store primitive types like u64 or bool . It may be considered a limitation, if not for the fact that dynamic object fields are not wrapped into a separate object.

The relaxed requirement for wrapping keeps the object available for off-chain discovery via its ID. However, this property may not be outstanding if wrapped object indexing is implemented, making the dynamic object fields a redundant feature.

Dynamic Object Fields come a little more expensive than dynamic fields. Because of their internal structure, they require 2 objects: the Wrapper for Name and the Value. Because of this, the cost of adding and accessing object fields (loading 2 objects compared to 1 for dynamic fields) is higher.

Both dynamic field and dynamic object fields are powerful features which allow for innovative solutions in applications. However, they are relatively low-level and require careful handling to avoid orphaned fields. In the next section, we will introduce a higher-level abstraction - Dynamic Collections - which can help with managing dynamic fields and objects more effectively.

Definition

Dynamic Object Fields are defined in the sui::dynamic_object_fields module in the Sui Framework . They are similar to dynamic fields in many ways, but unlike them, dynamic object fields have an extra constraint on the Value type. The Value must have a combination of key and store , not just store as in the case of dynamic fields.

They're less explicit in their framework definition, as the concept itself is more abstract:

File: sui-framework/sources/dynamic_object_fields.move

```
bash /// Internal object used for storing the field and the name associated with the /// value. The separate type is necessary to prevent key collision with direct /// usage of dynamic_field public struct Wrapper<Name> has copy, drop, store { name: Name, }
```

Unlike Field type in the Dynamic Fields section, the Wrapper type only stores the name of the field. The value is the object itself, and is not wrapped .

The constraints on the Value type become visible in the methods available for dynamic object fields. Here's the signature for the add function:

```
bash /// Adds a dynamic object field to the object `object: &mut UID` at field /// specified by
`name: Name`. Aborts with `EFieldAlreadyExists` if the object /// already has that field with that
name. public fun add<Name: copy + drop + store, Value: key + store>( // we use &mut UID in several
spots for access control object: &mut UID, name: Name, value: Value, ) { /* implementation omitted
*/ }
```

The rest of the methods which are identical to the ones in the Dynamic Fields section have the same constraints on the Value type. Let's list them for reference:

Additionally, there is an id method which returns the ID of the Value object without specifying its type.

The main difference between dynamic fields and dynamic object fields is that the latter allows storing only objects as values. This means that you can't store primitive types like u64 or bool . It may be considered a limitation, if not for the fact that dynamic object fields are not wrapped into a separate object.

The relaxed requirement for wrapping keeps the object available for off-chain discovery via its ID. However, this property may not be outstanding if wrapped object indexing is implemented, making the dynamic object fields a redundant feature.

```bash module book::dynamic_object_field;

use std::string::String;

// there are two common aliases for the long module name: `dof` and // `ofield`. Both are commonly used and met in different projects. use sui::dynamic_object_field as dof; use sui::dynamic_field as df;

/// The `Character` that we will use for the example public struct Character has key { id: UID }

/// Metadata that doesn't have the `key` ability public struct Metadata has store, drop { name: String }

/// Accessory that has the `key` and `store` abilities. public struct Accessory has key, store { id: UID }

# [test]

fun equip_accessory() { let ctx = &mut tx_context::dummy(); let mut character = Character { id: object::new(ctx) };

```
// Create an accessory and attach it to the character
let hat = Accessory { id: object::new(ctx) };

// Add the hat to the character. Just like with `dynamic_fields`
dof::add(&mut character.id, b"hat_key", hat);

// However for non-key structs we can only use `dynamic_field`
df::add(&mut character.id, b"metadata_key", Metadata {
    name: b"John".to_string()
});

// Borrow the hat from the character
let hat_id = dof::id(&character.id, b"hat_key").extract(); // Option<ID>
let hat_ref: &Accessory = dof::borrow(&character.id, b"hat_key");
let hat_mut: &mut Accessory = dof::borrow_mut(&mut character.id, b"hat_key");
let hat: Accessory = dof::remove(&mut character.id, b"hat_key");

// Clean up, Metadata is an orphan now.
sui::test_utils::destroy(hat);
sui::test_utils::destroy(character);

} ```
```

Dynamic Object Fields come a little more expensive than dynamic fields. Because of their internal structure, they require 2 objects: the Wrapper for Name and the Value. Because of this, the cost of adding and accessing object fields (loading 2 objects compared to 1 for dynamic fields) is higher.

Both dynamic field and dynamic object fields are powerful features which allow for innovative solutions in applications. However, they are relatively low-level and require careful handling to avoid orphaned fields. In the next section, we will introduce a higher-level abstraction - Dynamic Collections - which can help with managing dynamic fields and objects more effectively.

## Usage & Differences with Dynamic Fields
```

The main difference between dynamic fields and dynamic object fields is that the latter allows storing only objects as values. This means that you can't store primitive types like u64 or bool . It may be considered a limitation, if not for the fact that dynamic object fields are not wrapped into a separate object.

The relaxed requirement for wrapping keeps the object available for off-chain discovery via its ID. However, this property may not be outstanding if wrapped object indexing is implemented, making the dynamic object fields a redundant feature.

```bash module book::dynamic_object_field;

use std::string::String;

// there are two common aliases for the long module name: `dof` and // `ofield`. Both are commonly used and met in different projects. use sui::dynamic_object_field as dof; use sui::dynamic_field as df;

/// The `Character` that we will use for the example public struct Character has key { id: UID }

/// Metadata that doesn't have the `key` ability public struct Metadata has store, drop { name: String }

/// Accessory that has the `key` and `store` abilities. public struct Accessory has key, store { id: UID }

# [test]

fun equip_accessory() { let ctx = &mut tx_context::dummy(); let mut character = Character { id: object::new(ctx) };

```
// Create an accessory and attach it to the character
let hat = Accessory { id: object::new(ctx) };

// Add the hat to the character. Just like with `dynamic_fields`
dof::add(&mut character.id, b"hat_key", hat);

// However for non-key structs we can only use `dynamic_field`
df::add(&mut character.id, b"metadata_key", Metadata {
    name: b"John".to_string()
});

// Borrow the hat from the character
let hat_id = dof::id(&character.id, b"hat_key").extract(); // Option<ID>
let hat_ref: &Accessory = dof::borrow(&character.id, b"hat_key");
let hat_mut: &mut Accessory = dof::borrow_mut(&mut character.id, b"hat_key");
let hat: Accessory = dof::remove(&mut character.id, b"hat_key");

// Clean up, Metadata is an orphan now.
sui::test_utils::destroy(hat);
sui::test_utils::destroy(character);
```

} ```

Dynamic Object Fields come a little more expensive than dynamic fields. Because of their internal structure, they require 2 objects: the Wrapper for Name and the Value. Because of this, the cost of adding and accessing object fields (loading 2 objects compared to 1 for dynamic fields) is higher.

Both dynamic field and dynamic object fields are powerful features which allow for innovative solutions in applications. However, they are relatively low-level and require careful handling to avoid orphaned fields. In the next section, we will introduce a higher-level abstraction - Dynamic Collections - which can help with managing dynamic fields and objects more effectively.

## Pricing Differences

Dynamic Object Fields come a little more expensive than dynamic fields. Because of their internal structure, they require 2 objects: the Wrapper for Name and the Value. Because of this, the cost of adding and accessing object fields (loading 2 objects compared to 1 for dynamic fields) is higher.

Both dynamic field and dynamic object fields are powerful features which allow for innovative solutions in applications. However, they are relatively low-level and require careful handling to avoid orphaned fields. In the next section, we will introduce a higher-level abstraction - Dynamic Collections - which can help with managing dynamic fields and objects more effectively.

## Next Steps
```

Both dynamic field and dynamic object fields are powerful features which allow for innovative solutions in applications. However, they are relatively low-level and require careful handling to avoid orphaned fields. In the next section, we will introduce a higher-level abstraction - Dynamic Collections - which can help with managing dynamic fields and objects more effectively.