The Move Book

Sui Object model allows objects to be attached to other objects as dynamic fields . The behavior is similar to how a Map works in other programming languages. However, unlike a Map which in Move would be strictly typed (we have covered it in the Collections section), dynamic fields allow attaching objects of any type. A similar approach from the world of frontend development would be a JavaScript Object type which allows storing any type of data dynamically.

There's no limit to the number of dynamic fields that can be attached to an object. Thus, dynamic fields can be used to store large amounts of data that don't fit into the object limit size.

Dynamic Fields allow for a wide range of applications, from splitting data into smaller parts to avoid object size limit to attaching objects as a part of application logic.

Dynamic Fields are defined in the sui::dynamic_field module of the Sui Framework . They are attached to object's UID via a name , and can be accessed using that name. There can be only one field with a given name attached to an object.

File: sui-framework/sources/dynamic_field.move

As the definition shows, dynamic fields are stored in an internal Field object, which has the UID generated in a deterministic way based on the object ID, the field name, and the field type. The Field object contains the field name and the value bound to it. The constraints on the Name and Value type parameters define the abilities that the key and value must have.

The methods available for dynamic fields are straightforward: a field can be added with add , removed with remove , and read with borrow and borrow_mut . Additionally, the exists_ method can be used to check if a field exists (for stricter checks with type, there is an exists_with_type method).

In the example above, we define a Character object and two different types of accessories that could never be put together in a vector. However, dynamic fields allow us to store them together in a single object. Both objects are attached to the Character via a vector (bytestring literal), and can be accessed using their respective names.

As you can see, when we attached the accessories to the Character, we passed them by value . In other words, both values were moved to a new scope, and their ownership was transferred to the Character object. If we changed the ownership of Character object, the accessories would have been moved with it.

And the last important property of dynamic fields we should highlight is that they are accessed through their parent . This means that the Hat and Mustache objects are not directly accessible and follow the same rules as the parent object.

Dynamic fields allow objects to carry data of any type, including those defined in other modules. This is possible due to their generic nature and relatively weak constraints on the type parameters. Let's illustrate this by attaching a few different values to a Character object.

In this example we showed how different types can be used for both name and the value of a dynamic field. The String is attached via a vector name, the u64 is attached via a u32 name, and the bool is attached via a bool name. Anything is possible with dynamic fields!

To prevent orphaned dynamic fields, please, use Dynamic Collection Types such as Bag as they track the dynamic fields and won't allow unpacking if there are attached fields.

The object::delete() function, which is used to delete a UID, does not track the dynamic fields, and cannot prevent dynamic fields from becoming orphaned. Once the parent UID is deleted, the dynamic fields are not automatically deleted, and they become orphaned. This means that the dynamic fields are still stored in the blockchain, but they will never become accessible again.

Orphaned objects are not a subject to storage rebate, and the storage fees will remain unclaimed. One way to avoid orphaned dynamic fields during unpacking of an object is to return the UID and store it somewhere temporarily until the dynamic fields are removed and handled properly.

In the examples above, we used primitive types as field names since they have the required set of abilities. But dynamic fields get even more interesting when we use custom types as field names. This allows for a more structured way of storing data, and also allows for protecting the field names from being accessed by other modules.

Two field names that we defined above are AccessoryKey and MetadataKey . The AccessoryKey has a String field in it, hence it can be used multiple times with different name values. The MetadataKey is an empty key, and can be attached only once.

As you can see, custom types do work as field names but as long as they can be constructed by the module, in other words - if they are internal to the module and defined in it. This limitation on struct packing can open up new ways in the design of the application.

This approach is used in the Object Capability pattern, where an application can authorize a foreign object to perform operations in it while not exposing the capabilities to other modules.

Mutable access to UID is a security risk. Exposing UID of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields . Make sure to understand the implications before exposing the UID as a mutable reference.

Because dynamic fields are attached to UID s, their usage in other modules depends on whether the UID can be accessed. By default struct visibility protects the id field and won't let other modules access it directly. However, if there's a public accessor method that returns a reference to UID , dynamic fields can be read in other modules.

In the example above, we show how to expose the UID of a Character object. This solution may work for some applications, however, it is important to remember that exposed UID allows reading any dynamic field attached to the object.

If you need to expose the UID only within the package, use a restrictive visibility, like public(package) , or even better - use more specific accessor methods that would allow only reading specific fields.

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

Definition

Dynamic Fields are defined in the sui::dynamic_field module of the Sui Framework . They are attached to object's UID via a name , and can be accessed using that name. There can be only one field with a given name attached to an object.

File: sui-framework/sources/dynamic_field.move

```
bash /// Internal object used for storing the field and value public struct Field<Name: copy + drop + store, Value: store> has key { /// Determined by the hash of the object ID, the field name /// value and it's type, i.e. hash(parent.id || name || Name) id: UID, /// The value for the name of this field name: Name, /// The value bound to this field value: Value, }
```

As the definition shows, dynamic fields are stored in an internal Field object, which has the UID generated in a deterministic way based on the object ID, the field name, and the field type. The Field object contains the field name and the value bound to it. The constraints on the Name and Value type parameters define the abilities that the key and value must have.

The methods available for dynamic fields are straightforward: a field can be added with add , removed with remove , and read with borrow and borrow_mut . Additionally, the exists_ method can be used to check if a field exists (for stricter checks with type, there is an exists_with_type method).

```bash module book::dynamic_fields;

// a very common alias for `dynamic_field` is `df` since the // module name is quite long use sui::dynamic_field as df; use std::string::String;

/// The object that we will attach dynamic fields to. public struct Character has key { id: UID }

// List of different accessories that can be attached to a character. // They must have the `store` ability. public struct Hat has key, store { id: UID, color: u32 } public struct Mustache has key, store { id: UID }
```

[test]

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

```
// Attach a hat to the character's UID
df::add(
    &mut character.id,
    b"hat_key",
    Hat { id: object::new(ctx), color: 0xFF0000 }
);

// Similarly, attach a mustache to the character's UID
df::add(
    &mut character.id,
    b"mustache_key",
    Mustache { id: object::new(ctx) }
);

// Check that the hat and mustache are attached to the character
//
assert!(df::exists_(&character.id, b"hat_key"), 0);
assert!(df::exists_(&character.id, b"mustache_key"), 1);

// Modify the color of the hat
let hat: &mut Hat = df::borrow_mut(&mut character.id, b"hat_key");
hat.color = 0x00FF00;

// Remove the hat and mustache from the character
let hat: Hat = df::remove(&mut character.id, b"hat_key");
let mustache: Mustache = df::remove(&mut character.id, b"mustache_key");

// Check that the hat and mustache are no longer attached to the character
assert!(!df::exists_(&character.id, b"hat_key"), 0);
assert!(!df::exists_(&character.id, b"mustache_key"), 1);

sui::test_utils::destroy(character);
sui::test_utils::destroy(mustache);
sui::test_utils::destroy(hat);

} ```
```

In the example above, we define a Character object and two different types of accessories that could never be put together in a vector. However, dynamic fields allow us to store them together in a single object. Both objects are attached to the Character via a vector (bytestring literal), and can be accessed using their respective names.

As you can see, when we attached the accessories to the Character, we passed them by value . In other words, both values were moved to a new scope, and their ownership was transferred to the Character object. If we changed the ownership of Character object, the accessories would have been moved with it.

And the last important property of dynamic fields we should highlight is that they are accessed through their parent . This means that the Hat and Mustache objects are not directly accessible and follow the same rules as the parent object.

Dynamic fields allow objects to carry data of any type, including those defined in other modules. This is possible due to their generic nature and relatively weak constraints on the type parameters. Let's illustrate this by attaching a few different values to a Character object.

```bash let mut character = Character { id: object::new(ctx) };

// Attach a `String` via a `vector<u8>` name df::add(&mut character.id, b"string_key", b"Hello, World!".to_string());

// Attach a `u64` via a `u32` name df::add(&mut character.id, 1000u32, 1_000_000_000u64);

// Attach a `bool` via a `bool` name df::add(&mut character.id, true, false); ```

In this example we showed how different types can be used for both name and the value of a dynamic field. The String is attached via a vector name, the u64 is attached via a u32 name, and the bool is attached via a bool name. Anything is possible with dynamic fields!

To prevent orphaned dynamic fields, please, use Dynamic Collection Types such as Bag as they track the dynamic fields and won't
```

allow unpacking if there are attached fields.

The object::delete() function, which is used to delete a UID, does not track the dynamic fields, and cannot prevent dynamic fields from becoming orphaned. Once the parent UID is deleted, the dynamic fields are not automatically deleted, and they become orphaned. This means that the dynamic fields are still stored in the blockchain, but they will never become accessible again.

```bash let hat = Hat { id: object::new(ctx), color: 0xFF0000 }; let mut character = Character { id: object::new(ctx) };

// Attach a `Hat` via a `vector<u8>` name df::add(&mut character.id, b"hat_key", hat);

// ! DO NOT do this in your code // ! Danger - deleting the parent object let Character { id } = character; id.delete();

// ...`Hat` is now stuck in a limbo, it will never be accessible again ```

Orphaned objects are not a subject to storage rebate, and the storage fees will remain unclaimed. One way to avoid orphaned dynamic fields during unpacking of an object is to return the UID and store it somewhere temporarily until the dynamic fields are removed and handled properly.

In the examples above, we used primitive types as field names since they have the required set of abilities. But dynamic fields get even more interesting when we use custom types as field names. This allows for a more structured way of storing data, and also allows for protecting the field names from being accessed by other modules.

```bash /// A custom type with fields in it. public struct AccessoryKey has copy, drop, store { name: String }

/// An empty key, can be attached only once. public struct MetadataKey has copy, drop, store {} ```

Two field names that we defined above are AccessoryKey and MetadataKey . The AccessoryKey has a String field in it, hence it can be used multiple times with different name values. The MetadataKey is an empty key, and can be attached only once.

```bash let mut character = Character { id: object::new(ctx) };

// Attaching via an `AccessoryKey { name: b"hat" }` df::add( &mut character.id, AccessoryKey { name: b"hat".to_string() }, Hat { id: object::new(ctx), color: 0xFF0000 } ); // Attaching via an `AccessoryKey { name: b"mustache" }` df::add( &mut character.id, AccessoryKey { name: b"mustache".to_string() }, Mustache { id: object::new(ctx) } );

// Attaching via a `MetadataKey` df::add(&mut character.id, MetadataKey {}, 42); ```

As you can see, custom types do work as field names but as long as they can be constructed by the module, in other words - if they are internal to the module and defined in it. This limitation on struct packing can open up new ways in the design of the application.

This approach is used in the Object Capability pattern, where an application can authorize a foreign object to perform operations in it while not exposing the capabilities to other modules.

Mutable access to UID is a security risk. Exposing UID of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields . Make sure to understand the implications before exposing the UID as a mutable reference.

Because dynamic fields are attached to UID s, their usage in other modules depends on whether the UID can be accessed. By default struct visibility protects the id field and won't let other modules access it directly. However, if there's a public accessor method that returns a reference to UID , dynamic fields can be read in other modules.

```
bash /// Exposes the UID of the character, so that other modules can read /// dynamic fields.
public fun uid(c: &Character): &UID { &c.id }
```

In the example above, we show how to expose the UID of a Character object. This solution may work for some applications, however, it is important to remember that exposed UID allows reading any dynamic field attached to the object.

If you need to expose the UID only within the package, use a restrictive visibility, like public(package) , or even better - use more specific accessor methods that would allow only reading specific fields.

```bash /// Only allow modules in the same package to access the UID. public(package) fun uid_package(c: &Character): &UID { &c.id }

/// Allow borrowing dynamic fields from the character. public fun borrow( c: &Character, n: Name ): &Value { df::borrow(&c.id, n) } ```

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Usage

The methods available for dynamic fields are straightforward: a field can be added with add , removed with remove , and read with borrow and borrow_mut . Additionally, the exists_ method can be used to check if a field exists (for stricter checks with type, there is an exists_with_type method).

```bash module book::dynamic_fields;

// a very common alias for `dynamic_field` is `df` since the // module name is quite long use sui::dynamic_field as df; use std::string::String;

/// The object that we will attach dynamic fields to. public struct Character has key { id: UID }

// List of different accessories that can be attached to a character. // They must have the `store` ability. public struct Hat has key, store { id: UID, color: u32 } public struct Mustache has key, store { id: UID }

# [test]

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

```
// Attach a hat to the character's UID
df::add(
    &mut character.id,
    b"hat_key",
    Hat { id: object::new(ctx), color: 0xFF0000 }
);

// Similarly, attach a mustache to the character's UID
df::add(
    &mut character.id,
    b"mustache_key",
    Mustache { id: object::new(ctx) }
);

// Check that the hat and mustache are attached to the character
//
assert!(df::exists_(&character.id, b"hat_key"), 0);
assert!(df::exists_(&character.id, b"mustache_key"), 1);

// Modify the color of the hat
let hat: &mut Hat = df::borrow_mut(&mut character.id, b"hat_key");
hat.color = 0x00FF00;

// Remove the hat and mustache from the character
let hat: Hat = df::remove(&mut character.id, b"hat_key");
let mustache: Mustache = df::remove(&mut character.id, b"mustache_key");

// Check that the hat and mustache are no longer attached to the character
assert!(!df::exists_(&character.id, b"hat_key"), 0);
assert!(!df::exists_(&character.id, b"mustache_key"), 1);

sui::test_utils::destroy(character);
sui::test_utils::destroy(mustache);
sui::test_utils::destroy(hat);
```
```

} ```

In the example above, we define a Character object and two different types of accessories that could never be put together in a vector. However, dynamic fields allow us to store them together in a single object. Both objects are attached to the Character via a vector (bytestring literal), and can be accessed using their respective names.

As you can see, when we attached the accessories to the Character, we passed them by value . In other words, both values were moved to a new scope, and their ownership was transferred to the Character object. If we changed the ownership of Character object, the accessories would have been moved with it.

And the last important property of dynamic fields we should highlight is that they are accessed through their parent . This means that the Hat and Mustache objects are not directly accessible and follow the same rules as the parent object.

Dynamic fields allow objects to carry data of any type, including those defined in other modules. This is possible due to their generic nature and relatively weak constraints on the type parameters. Let's illustrate this by attaching a few different values to a Character object.

```bash let mut character = Character { id: object::new(ctx) };

// Attach a `String` via a `vector<u8>` name df::add(&mut character.id, b"string_key", b"Hello, World!".to_string());

// Attach a `u64` via a `u32` name df::add(&mut character.id, 1000u32, 1_000_000_000u64);

// Attach a `bool` via a `bool` name df::add(&mut character.id, true, false); ```

In this example we showed how different types can be used for both name and the value of a dynamic field. The String is attached via a vector name, the u64 is attached via a u32 name, and the bool is attached via a bool name. Anything is possible with dynamic fields!

To prevent orphaned dynamic fields, please, use Dynamic Collection Types such as Bag as they track the dynamic fields and won't allow unpacking if there are attached fields.

The object::delete() function, which is used to delete a UID, does not track the dynamic fields, and cannot prevent dynamic fields from becoming orphaned. Once the parent UID is deleted, the dynamic fields are not automatically deleted, and they become orphaned. This means that the dynamic fields are still stored in the blockchain, but they will never become accessible again.

```bash let hat = Hat { id: object::new(ctx), color: 0xFF0000 }; let mut character = Character { id: object::new(ctx) };

// Attach a `Hat` via a `vector<u8>` name df::add(&mut character.id, b"hat_key", hat);

// ! DO NOT do this in your code // ! Danger - deleting the parent object let Character { id } = character; id.delete();

// ...`Hat` is now stuck in a limbo, it will never be accessible again ```

Orphaned objects are not a subject to storage rebate, and the storage fees will remain unclaimed. One way to avoid orphaned dynamic fields during unpacking of an object is to return the UID and store it somewhere temporarily until the dynamic fields are removed and handled properly.

In the examples above, we used primitive types as field names since they have the required set of abilities. But dynamic fields get even more interesting when we use custom types as field names. This allows for a more structured way of storing data, and also allows for protecting the field names from being accessed by other modules.

```bash /// A custom type with fields in it. public struct AccessoryKey has copy, drop, store { name: String }

/// An empty key, can be attached only once. public struct MetadataKey has copy, drop, store {} ```

Two field names that we defined above are AccessoryKey and MetadataKey . The AccessoryKey has a String field in it, hence it can be used multiple times with different name values. The MetadataKey is an empty key, and can be attached only once.

```bash let mut character = Character { id: object::new(ctx) };

// Attaching via an `AccessoryKey { name: b"hat" }` df::add( &mut character.id, AccessoryKey { name: b"hat".to_string() }, Hat { id: object::new(ctx), color: 0xFF0000 } ); // Attaching via an `AccessoryKey { name: b"mustache" }` df::add( &mut character.id, AccessoryKey { name: b"mustache".to_string() }, Mustache { id: object::new(ctx) } );

// Attaching via a `MetadataKey` df::add(&mut character.id, MetadataKey {}, 42); ```

As you can see, custom types do work as field names but as long as they can be constructed by the module, in other words - if they are internal to the module and defined in it. This limitation on struct packing can open up new ways in the design of the application.

This approach is used in the Object Capability pattern, where an application can authorize a foreign object to perform operations in it while not exposing the capabilities to other modules.

Mutable access to UID is a security risk. Exposing UID of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields . Make sure to understand the implications before exposing the UID as a mutable reference.

Because dynamic fields are attached to UID s, their usage in other modules depends on whether the UID can be accessed. By default struct visibility protects the id field and won't let other modules access it directly. However, if there's a public accessor method that returns a reference to UID , dynamic fields can be read in other modules.

```
bash /// Exposes the UID of the character, so that other modules can read /// dynamic fields.
public fun uid(c: &Character): &UID { &c.id }
```

In the example above, we show how to expose the UID of a Character object. This solution may work for some applications, however, it is important to remember that exposed UID allows reading any dynamic field attached to the object.

If you need to expose the UID only within the package, use a restrictive visibility, like public(package) , or even better - use more specific accessor methods that would allow only reading specific fields.

```bash /// Only allow modules in the same package to access the UID. public(package) fun uid_package(c: &Character): &UID { &c.id }

/// Allow borrowing dynamic fields from the character. public fun borrow( c: &Character, n: Name ): &Value { df::borrow(&c.id, n) } ```

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Foreign Types as Dynamic Fields

Dynamic fields allow objects to carry data of any type, including those defined in other modules. This is possible due to their generic nature and relatively weak constraints on the type parameters. Let's illustrate this by attaching a few different values to a Character object.

```bash let mut character = Character { id: object::new(ctx) };

// Attach a `String` via a `vector<u8>` name df::add(&mut character.id, b"string_key", b"Hello, World!".to_string());

// Attach a `u64` via a `u32` name df::add(&mut character.id, 1000u32, 1_000_000_000u64);

// Attach a `bool` via a `bool` name df::add(&mut character.id, true, false); ```

In this example we showed how different types can be used for both name and the value of a dynamic field. The String is attached via a vector name, the u64 is attached via a u32 name, and the bool is attached via a bool name. Anything is possible with dynamic fields!

To prevent orphaned dynamic fields, please, use Dynamic Collection Types such as Bag as they track the dynamic fields and won't allow unpacking if there are attached fields.

The object::delete() function, which is used to delete a UID, does not track the dynamic fields, and cannot prevent dynamic fields from becoming orphaned. Once the parent UID is deleted, the dynamic fields are not automatically deleted, and they become orphaned. This means that the dynamic fields are still stored in the blockchain, but they will never become accessible again.

```bash let hat = Hat { id: object::new(ctx), color: 0xFF0000 }; let mut character = Character { id: object::new(ctx) };

// Attach a `Hat` via a `vector<u8>` name df::add(&mut character.id, b"hat_key", hat);

// ! DO NOT do this in your code // ! Danger - deleting the parent object let Character { id } = character; id.delete();

// ...`Hat` is now stuck in a limbo, it will never be accessible again ```

Orphaned objects are not a subject to storage rebate, and the storage fees will remain unclaimed. One way to avoid orphaned dynamic fields during unpacking of an object is to return the UID and store it somewhere temporarily until the dynamic fields are removed and handled properly.

In the examples above, we used primitive types as field names since they have the required set of abilities. But dynamic fields get even more interesting when we use custom types as field names. This allows for a more structured way of storing data, and also allows for protecting the field names from being accessed by other modules.

```bash /// A custom type with fields in it. public struct AccessoryKey has copy, drop, store { name: String }

/// An empty key, can be attached only once. public struct MetadataKey has copy, drop, store {} ```

Two field names that we defined above are AccessoryKey and MetadataKey . The AccessoryKey has a String field in it, hence it can be used multiple times with different name values. The MetadataKey is an empty key, and can be attached only once.

```bash let mut character = Character { id: object::new(ctx) };

// Attaching via an `AccessoryKey { name: b"hat" }` df::add( &mut character.id, AccessoryKey { name: b"hat".to_string() }, Hat { id: object::new(ctx), color: 0xFF0000 } ); // Attaching via an `AccessoryKey { name: b"mustache" }` df::add( &mut character.id, AccessoryKey { name: b"mustache".to_string() }, Mustache { id: object::new(ctx) } );

// Attaching via a `MetadataKey` df::add(&mut character.id, MetadataKey {}, 42); ```

As you can see, custom types do work as field names but as long as they can be constructed by the module, in other words - if they are internal to the module and defined in it. This limitation on struct packing can open up new ways in the design of the application.

This approach is used in the Object Capability pattern, where an application can authorize a foreign object to perform operations in it while not exposing the capabilities to other modules.

Mutable access to UID is a security risk. Exposing UID of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields . Make sure to understand the implications before exposing the UID as a mutable reference.

Because dynamic fields are attached to UID s, their usage in other modules depends on whether the UID can be accessed. By default struct visibility protects the id field and won't let other modules access it directly. However, if there's a public accessor method that returns a reference to UID , dynamic fields can be read in other modules.

```
bash /// Exposes the UID of the character, so that other modules can read /// dynamic fields.
public fun uid(c: &Character): &UID { &c.id }
```

In the example above, we show how to expose the UID of a Character object. This solution may work for some applications, however, it is important to remember that exposed UID allows reading any dynamic field attached to the object.

If you need to expose the UID only within the package, use a restrictive visibility, like public(package) , or even better - use more specific accessor methods that would allow only reading specific fields.

```bash /// Only allow modules in the same package to access the UID. public(package) fun uid_package(c: &Character): &UID { &c.id }

/// Allow borrowing dynamic fields from the character. public fun borrow( c: &Character, n: Name ): &Value { df::borrow(&c.id, n) } ```

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields

and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Orphaned Dynamic Fields

To prevent orphaned dynamic fields, please, use Dynamic Collection Types such as Bag as they track the dynamic fields and won't allow unpacking if there are attached fields.

The object::delete() function, which is used to delete a UID, does not track the dynamic fields, and cannot prevent dynamic fields from becoming orphaned. Once the parent UID is deleted, the dynamic fields are not automatically deleted, and they become orphaned. This means that the dynamic fields are still stored in the blockchain, but they will never become accessible again.

```bash let hat = Hat { id: object::new(ctx), color: 0xFF0000 }; let mut character = Character { id: object::new(ctx) };

// Attach a `Hat` via a `vector<u8>` name df::add(&mut character.id, b"hat_key", hat);

// ! DO NOT do this in your code // ! Danger - deleting the parent object let Character { id } = character; id.delete();

// ...`Hat` is now stuck in a limbo, it will never be accessible again ```

Orphaned objects are not a subject to storage rebate, and the storage fees will remain unclaimed. One way to avoid orphaned dynamic fields during unpacking of an object is to return the UID and store it somewhere temporarily until the dynamic fields are removed and handled properly.

In the examples above, we used primitive types as field names since they have the required set of abilities. But dynamic fields get even more interesting when we use custom types as field names. This allows for a more structured way of storing data, and also allows for protecting the field names from being accessed by other modules.

```bash /// A custom type with fields in it. public struct AccessoryKey has copy, drop, store { name: String }

/// An empty key, can be attached only once. public struct MetadataKey has copy, drop, store {} ```

Two field names that we defined above are AccessoryKey and MetadataKey . The AccessoryKey has a String field in it, hence it can be used multiple times with different name values. The MetadataKey is an empty key, and can be attached only once.

```bash let mut character = Character { id: object::new(ctx) };

// Attaching via an `AccessoryKey { name: b"hat" }` df::add( &mut character.id, AccessoryKey { name: b"hat".to_string() }, Hat { id: object::new(ctx), color: 0xFF0000 } ); // Attaching via an `AccessoryKey { name: b"mustache" }` df::add( &mut character.id, AccessoryKey { name: b"mustache".to_string() }, Mustache { id: object::new(ctx) } );

// Attaching via a `MetadataKey` df::add(&mut character.id, MetadataKey {}, 42); ```

As you can see, custom types do work as field names but as long as they can be constructed by the module, in other words - if they are internal to the module and defined in it. This limitation on struct packing can open up new ways in the design of the application.

This approach is used in the Object Capability pattern, where an application can authorize a foreign object to perform operations in it while not exposing the capabilities to other modules.

Mutable access to UID is a security risk. Exposing UID of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields . Make sure to understand the implications before exposing the UID as a mutable reference.

Because dynamic fields are attached to UID s, their usage in other modules depends on whether the UID can be accessed. By default struct visibility protects the id field and won't let other modules access it directly. However, if there's a public accessor
```

method that returns a reference to UID , dynamic fields can be read in other modules.

```
bash /// Exposes the UID of the character, so that other modules can read /// dynamic fields.
public fun uid(c: &Character): &UID { &c.id }
```

In the example above, we show how to expose the UID of a Character object. This solution may work for some applications, however, it is important to remember that exposed UID allows reading any dynamic field attached to the object.

If you need to expose the UID only within the package, use a restrictive visibility, like public(package) , or even better - use more specific accessor methods that would allow only reading specific fields.

```bash /// Only allow modules in the same package to access the UID. public(package) fun uid_package(c: &Character): &UID { &c.id }

/// Allow borrowing dynamic fields from the character. public fun borrow( c: &Character, n: Name ): &Value { df::borrow(&c.id, n) } ```

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Custom Type as a Field Name

In the examples above, we used primitive types as field names since they have the required set of abilities. But dynamic fields get even more interesting when we use custom types as field names. This allows for a more structured way of storing data, and also allows for protecting the field names from being accessed by other modules.

```bash /// A custom type with fields in it. public struct AccessoryKey has copy, drop, store { name: String }

/// An empty key, can be attached only once. public struct MetadataKey has copy, drop, store {} ```

Two field names that we defined above are AccessoryKey and MetadataKey . The AccessoryKey has a String field in it, hence it can be used multiple times with different name values. The MetadataKey is an empty key, and can be attached only once.

```bash let mut character = Character { id: object::new(ctx) };

// Attaching via an `AccessoryKey { name: b"hat" }` df::add( &mut character.id, AccessoryKey { name: b"hat".to_string() }, Hat { id: object::new(ctx), color: 0xFF0000 } ); // Attaching via an `AccessoryKey { name: b"mustache" }` df::add( &mut character.id, AccessoryKey { name: b"mustache".to_string() }, Mustache { id: object::new(ctx) } );

// Attaching via a `MetadataKey` df::add(&mut character.id, MetadataKey {}, 42); ```

As you can see, custom types do work as field names but as long as they can be constructed by the module, in other words - if they are internal to the module and defined in it. This limitation on struct packing can open up new ways in the design of the application.

This approach is used in the Object Capability pattern, where an application can authorize a foreign object to perform operations in it while not exposing the capabilities to other modules.

Mutable access to UID is a security risk. Exposing UID of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields . Make sure to understand the implications before exposing the UID as a mutable reference.

Because dynamic fields are attached to UID s, their usage in other modules depends on whether the UID can be accessed. By default struct visibility protects the id field and won't let other modules access it directly. However, if there's a public accessor method that returns a reference to UID , dynamic fields can be read in other modules.
```

```
bash /// Exposes the UID of the character, so that other modules can read /// dynamic fields.
public fun uid(c: &Character): &UID { &c.id }
```

In the example above, we show how to expose the UID of a Character object. This solution may work for some applications, however, it is important to remember that exposed UID allows reading any dynamic field attached to the object.

If you need to expose the UID only within the package, use a restrictive visibility, like public(package) , or even better - use more specific accessor methods that would allow only reading specific fields.

```bash /// Only allow modules in the same package to access the UID. public(package) fun uid_package(c: &Character): &UID { &c.id }

/// Allow borrowing dynamic fields from the character. public fun borrow( c: &Character, n: Name ): &Value { df::borrow(&c.id, n) } ```

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Exposing UID

Mutable access to UID is a security risk. Exposing UID of your type as a mutable reference can lead to unwanted modifications or removal of the object's dynamic fields. Additionally, it affects the Transfer to Object and Dynamic Object Fields . Make sure to understand the implications before exposing the UID as a mutable reference.

Because dynamic fields are attached to UID s, their usage in other modules depends on whether the UID can be accessed. By default struct visibility protects the id field and won't let other modules access it directly. However, if there's a public accessor method that returns a reference to UID , dynamic fields can be read in other modules.

```
bash /// Exposes the UID of the character, so that other modules can read /// dynamic fields.
public fun uid(c: &Character): &UID { &c.id }
```

In the example above, we show how to expose the UID of a Character object. This solution may work for some applications, however, it is important to remember that exposed UID allows reading any dynamic field attached to the object.

If you need to expose the UID only within the package, use a restrictive visibility, like public(package) , or even better - use more specific accessor methods that would allow only reading specific fields.

```bash /// Only allow modules in the same package to access the UID. public(package) fun uid_package(c: &Character): &UID { &c.id }

/// Allow borrowing dynamic fields from the character. public fun borrow( c: &Character, n: Name ): &Value { df::borrow(&c.id, n) } ```

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Dynamic Fields vs Fields

Dynamic Fields are more expensive than regular fields, as they require additional storage and costs for accessing them. Their flexibility comes at a price, and it is important to understand the implications when making a decision between using dynamic fields and regular fields.

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Limits

Dynamic Fields are not subject to the object size limit , and can be used to store large amounts of data. However, they are still subject to the dynamic fields created limit , which is set to 1000 fields per transaction.

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Applications

Dynamic Fields can play a crucial role in applications of any complexity. They open up a variety of different use cases, from storing heterogeneous data to attaching objects as part of the application logic. They allow for certain upgradeability practices based on the ability to define them later and change the type of the field.

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.

## Next Steps

In the next section we will cover Dynamic Object Fields and explain how they differ from dynamic fields, and what are the implications of using them.