The Move Book

A case in the abilities system - a struct without any abilities - is called hot potato . It cannot be stored (not as an object nor as a field in another struct), it cannot be copied or discarded . Hence, once constructed, it must be gracefully unpacked by its module , or the transaction will abort due to unused value without drop.

If you're familiar with languages that support callbacks , you can think of a hot potato as an obligation to call a callback function. If you don't call it, the transaction will abort.

The name comes from the children's game where a ball is passed quickly between players, and none of the players want to be the last one holding it when the music stops, or they are out of the game. This is the best illustration of the pattern - the instance of a hot-potato struct is passed between calls, and none of the modules can keep it.

A hot potato can be any struct with no abilities. For example, the following struct is a hot potato:

Because the Request has no abilities and cannot be stored or ignored, the module must provide a function to unpack it. For example:

In the following example, the Promise hot potato is used to ensure that the borrowed value, when taken from the container, is returned back to it. The Promise struct contains the ID of the borrowed object, and the ID of the container, ensuring that the borrowed value was not swapped for another and is returned to the correct container.

Below we list some of the common use cases for the hot potato pattern.

As shown in the example above , the hot potato is very effective for borrowing with a guarantee that the borrowed value is returned to the correct container. While the example focuses on a value stored inside an Option , the same pattern can be applied to any other storage type, say a dynamic field .

Canonical example of the hot potato pattern is flash loans. A flash loan is a loan that is borrowed and repaid in the same transaction. The borrowed funds are used to perform some operations, and the repaid funds are returned to the lender. The hot potato pattern ensures that the borrowed funds are returned to the lender.

An example usage of this pattern may look like this:

The hot potato pattern can be used to introduce variation in the execution path. For example, if there is a module which allows purchasing a Phone for some "Bonus Points" or for USD, the hot potato can be used to decouple the purchase from the payment. The approach is very similar to how some shops work - you take the item from the shelf, and then you go to the cashier to pay for it.

This decoupling technique allows separating the purchase logic from the payment logic, making the code more modular and easier to maintain. The Ticket could be split into its own module, providing a basic interface for the payment, and the shop implementation could be extended to support other goods without changing the payment logic.

Hot potato can be used to link together different modules in a compositional way. Its module may define ways to interact with the hot potato, for example, stamp it with a type signature, or to extract some information from it. This way, the hot potato can be passed between different modules, and even different packages within the same transaction.

The most important compositional pattern is the Request Pattern , which we will cover in the next section.

The pattern is used in various forms in the Sui Framework. Here are some examples:

Defining a Hot Potato

A hot potato can be any struct with no abilities. For example, the following struct is a hot potato:

```
bash public struct Request {}
```

Because the Request has no abilities and cannot be stored or ignored, the module must provide a function to unpack it. For example:

``bash /// Constructs a newRequest` public fun new_request(): Request { Request {} }

/// Unpacks the `Request`. Due to the nature of the hot potato, this function /// must be called to avoid aborting the transaction. public fun confirm_request(request: Request) { let Request {} = request; } ```

In the following example, the Promise hot potato is used to ensure that the borrowed value, when taken from the container, is returned back to it. The Promise struct contains the ID of the borrowed object, and the ID of the container, ensuring that the

borrowed value was not swapped for another and is returned to the correct container.

``bash /// A generic container for any Object withkey + store`. The Option type /// is used to allow taking and putting the value back. public struct Container has key { id: UID, value: Option, }

/// A Hot Potato struct that is used to ensure the borrowed value is returned. public struct Promise { /// The ID of the borrowed object. Ensures that there wasn't a value swap. id: ID, /// The ID of the container. Ensures that the borrowed value is returned to /// the correct container. container_id: ID, }

/// A function that allows borrowing the value from the container. public fun borrow_val(container: &mut Container): (T, Promise) { assert!(container.value.is_some()); let value = container.value.extract(); let id = object::id(&value); (value, Promise { id, container_id: object::id(container) }) }

/// Put the taken item back into the container. public fun return_val(container: &mut Container, value: T, promise: Promise) { let Promise { id, container_id } = promise; assert!(object::id(container) == container_id); assert!(object::id(&value) == id); container.value.fill(value); } ```

Below we list some of the common use cases for the hot potato pattern.

As shown in the example above , the hot potato is very effective for borrowing with a guarantee that the borrowed value is returned to the correct container. While the example focuses on a value stored inside an Option , the same pattern can be applied to any other storage type, say a dynamic field .

Canonical example of the hot potato pattern is flash loans. A flash loan is a loan that is borrowed and repaid in the same transaction. The borrowed funds are used to perform some operations, and the repaid funds are returned to the lender. The hot potato pattern ensures that the borrowed funds are returned to the lender.

An example usage of this pattern may look like this:

```bash // Borrow the funds from the lender. let (asset_a, potato) = lender.borrow(amount);

// Perform some operations with the borrowed funds. let asset_b = dex.trade(loan); let proceeds = another_contract::do_something(asset_b);

// Keep the commission and return the rest to the lender. let pay_back = proceeds.split(amount, ctx); lender.repay(pay_back, potato); transfer::public_transfer(proceeds, ctx.sender()); ```

The hot potato pattern can be used to introduce variation in the execution path. For example, if there is a module which allows purchasing a Phone for some "Bonus Points" or for USD, the hot potato can be used to decouple the purchase from the payment. The approach is very similar to how some shops work - you take the item from the shelf, and then you go to the cashier to pay for it.

``bash /// APhone`. Can be purchased in a store. public struct Phone has key, store { id: UID }

/// A ticket that must be paid to purchase the `Phone`. public struct Ticket { amount: u64 }

/// Return the `Phone` and the `Ticket` that must be paid to purchase it. public fun purchase_phone(ctx: &mut TxContext): (Phone, Ticket) { ( Phone { id: object::new(ctx) }, Ticket { amount: 100 } ) }

/// The customer may pay for the `Phone` with `BonusPoints` or `SUI`. public fun pay_in_bonus_points(ticket: Ticket, payment: Coin) { let Ticket { amount } = ticket; assert!(payment.value() == amount); abort 0 // omitting the rest of the function }

/// The customer may pay for the `Phone` with `USD`. public fun pay_in_usd(ticket: Ticket, payment: Coin) { let Ticket { amount } = ticket; assert!(payment.value() == amount); abort 0 // omitting the rest of the function } ```

This decoupling technique allows separating the purchase logic from the payment logic, making the code more modular and easier to maintain. The Ticket could be split into its own module, providing a basic interface for the payment, and the shop implementation could be extended to support other goods without changing the payment logic.

Hot potato can be used to link together different modules in a compositional way. Its module may define ways to interact with the hot potato, for example, stamp it with a type signature, or to extract some information from it. This way, the hot potato can be passed between different modules, and even different packages within the same transaction.

The most important compositional pattern is the Request Pattern , which we will cover in the next section.

The pattern is used in various forms in the Sui Framework. Here are some examples:
```

Example Usage

In the following example, the Promise hot potato is used to ensure that the borrowed value, when taken from the container, is returned back to it. The Promise struct contains the ID of the borrowed object, and the ID of the container, ensuring that the borrowed value was not swapped for another and is returned to the correct container.

``bash /// A generic container for any Object withkey + store`. The Option type /// is used to allow taking and putting the value back. public struct Container has key { id: UID, value: Option, }

/// A Hot Potato struct that is used to ensure the borrowed value is returned. public struct Promise { /// The ID of the borrowed object. Ensures that there wasn't a value swap. id: ID, /// The ID of the container. Ensures that the borrowed value is returned to /// the correct container. container_id: ID, }

/// A function that allows borrowing the value from the container. public fun borrow_val(container: &mut Container): (T, Promise) { assert!(container.value.is_some()); let value = container.value.extract(); let id = object::id(&value); (value, Promise { id, container_id: object::id(container) }) }

/// Put the taken item back into the container. public fun return_val(container: &mut Container, value: T, promise: Promise) { let Promise { id, container_id } = promise; assert!(object::id(container) == container_id); assert!(object::id(&value) == id); container.value.fill(value); } ```

Below we list some of the common use cases for the hot potato pattern.

As shown in the example above , the hot potato is very effective for borrowing with a guarantee that the borrowed value is returned to the correct container. While the example focuses on a value stored inside an Option , the same pattern can be applied to any other storage type, say a dynamic field .

Canonical example of the hot potato pattern is flash loans. A flash loan is a loan that is borrowed and repaid in the same transaction. The borrowed funds are used to perform some operations, and the repaid funds are returned to the lender. The hot potato pattern ensures that the borrowed funds are returned to the lender.

An example usage of this pattern may look like this:

```bash // Borrow the funds from the lender. let (asset_a, potato) = lender.borrow(amount);

// Perform some operations with the borrowed funds. let asset_b = dex.trade(loan); let proceeds = another_contract::do_something(asset_b);

// Keep the commission and return the rest to the lender. let pay_back = proceeds.split(amount, ctx); lender.repay(pay_back, potato); transfer::public_transfer(proceeds, ctx.sender()); ```

The hot potato pattern can be used to introduce variation in the execution path. For example, if there is a module which allows purchasing a Phone for some "Bonus Points" or for USD, the hot potato can be used to decouple the purchase from the payment. The approach is very similar to how some shops work - you take the item from the shelf, and then you go to the cashier to pay for it.

``bash /// APhone`. Can be purchased in a store. public struct Phone has key, store { id: UID }

/// A ticket that must be paid to purchase the `Phone`. public struct Ticket { amount: u64 }

/// Return the `Phone` and the `Ticket` that must be paid to purchase it. public fun purchase_phone(ctx: &mut TxContext): (Phone, Ticket) { ( Phone { id: object::new(ctx) }, Ticket { amount: 100 } ) }

/// The customer may pay for the `Phone` with `BonusPoints` or `SUI`. public fun pay_in_bonus_points(ticket: Ticket, payment: Coin) { let Ticket { amount } = ticket; assert!(payment.value() == amount); abort 0 // omitting the rest of the function }

/// The customer may pay for the `Phone` with `USD`. public fun pay_in_usd(ticket: Ticket, payment: Coin) { let Ticket { amount } = ticket; assert!(payment.value() == amount); abort 0 // omitting the rest of the function } ```

This decoupling technique allows separating the purchase logic from the payment logic, making the code more modular and easier to maintain. The Ticket could be split into its own module, providing a basic interface for the payment, and the shop implementation could be extended to support other goods without changing the payment logic.

Hot potato can be used to link together different modules in a compositional way. Its module may define ways to interact with the hot potato, for example, stamp it with a type signature, or to extract some information from it. This way, the hot potato can be
```

passed between different modules, and even different packages within the same transaction.

The most important compositional pattern is the Request Pattern , which we will cover in the next section.

The pattern is used in various forms in the Sui Framework. Here are some examples:

Applications

Below we list some of the common use cases for the hot potato pattern.

As shown in the example above , the hot potato is very effective for borrowing with a guarantee that the borrowed value is returned to the correct container. While the example focuses on a value stored inside an Option , the same pattern can be applied to any other storage type, say a dynamic field .

Canonical example of the hot potato pattern is flash loans. A flash loan is a loan that is borrowed and repaid in the same transaction. The borrowed funds are used to perform some operations, and the repaid funds are returned to the lender. The hot potato pattern ensures that the borrowed funds are returned to the lender.

An example usage of this pattern may look like this:

```bash // Borrow the funds from the lender. let (asset_a, potato) = lender.borrow(amount);

// Perform some operations with the borrowed funds. let asset_b = dex.trade(loan); let proceeds = another_contract::do_something(asset_b);

// Keep the commission and return the rest to the lender. let pay_back = proceeds.split(amount, ctx); lender.repay(pay_back, potato); transfer::public_transfer(proceeds, ctx.sender()); ```

The hot potato pattern can be used to introduce variation in the execution path. For example, if there is a module which allows purchasing a Phone for some "Bonus Points" or for USD, the hot potato can be used to decouple the purchase from the payment. The approach is very similar to how some shops work - you take the item from the shelf, and then you go to the cashier to pay for it.

``bash /// APhone`. Can be purchased in a store. public struct Phone has key, store { id: UID }

/// A ticket that must be paid to purchase the `Phone`. public struct Ticket { amount: u64 }

/// Return the `Phone` and the `Ticket` that must be paid to purchase it. public fun purchase_phone(ctx: &mut TxContext): (Phone, Ticket) { ( Phone { id: object::new(ctx) }, Ticket { amount: 100 } ) }

/// The customer may pay for the `Phone` with `BonusPoints` or `SUI`. public fun pay_in_bonus_points(ticket: Ticket, payment: Coin) { let Ticket { amount } = ticket; assert!(payment.value() == amount); abort 0 // omitting the rest of the function }

/// The customer may pay for the `Phone` with `USD`. public fun pay_in_usd(ticket: Ticket, payment: Coin) { let Ticket { amount } = ticket; assert!(payment.value() == amount); abort 0 // omitting the rest of the function } ```

This decoupling technique allows separating the purchase logic from the payment logic, making the code more modular and easier to maintain. The Ticket could be split into its own module, providing a basic interface for the payment, and the shop implementation could be extended to support other goods without changing the payment logic.

Hot potato can be used to link together different modules in a compositional way. Its module may define ways to interact with the hot potato, for example, stamp it with a type signature, or to extract some information from it. This way, the hot potato can be passed between different modules, and even different packages within the same transaction.

The most important compositional pattern is the Request Pattern , which we will cover in the next section.

The pattern is used in various forms in the Sui Framework. Here are some examples:

## Summary
```