# The Move Book

In the Ownership and Scope section, we explained that when a value is passed to a function, it is moved to the function's scope. This means that the function becomes the owner of the value, and the original scope (owner) can no longer use it. This is an important concept in Move, as it ensures that the value is not used in multiple places at the same time. However, there are use cases when we want to pass a value to a function but retain ownership. This is where references come into play.

To illustrate this, let's consider a simple example - an application for a metro (subway) pass. We will look at 4 different scenarios where a card can be:

The initial layout of the metro pass application is simple. We define the Card type and the USES constant that represents the number of rides on a single card. We also add an error constant for the case when the card is empty.

References are a way to show a value to a function without giving up ownership. In our case, when we show the Card to the inspector, we don't want to give up ownership of it, and we don't allow the inspector to use up any of our rides. We just want to allow the reading of the value of our Card and to prove its ownership.

To do so, in the function signature, we use the & symbol to indicate that we are passing a reference to the value, not the value itself.

Because the function does not take ownership of the Card, it can read its data but cannot write to it, meaning it cannot modify the number of rides. Additionally, the function signature ensures that it cannot be called without a Card instance. This is an important property that allows the Capability Pattern , which we will cover in the next chapters.

Creating a reference to a value is often referred to as "borrowing" the value. For example, the method to get a reference to the value wrapped by an Option is called borrow .

In some cases, we want to allow the function to modify the Card. For example, when using the Card at a turnstile, we need to deduct a ride. To achieve this, we use the &mut keyword in the function signature.

As you can see in the function body, the &mut reference allows mutating the value, and the function can spend rides.

Lastly, let's illustrate what happens when we pass the value itself to the function. In this case, the function takes the ownership of the value, making it inaccessible in the original scope. The owner of the Card can recycle it and thereby relinquish ownership to the function.

In the recycle function, the Card is passed by value, transferring ownership to the function. This allows it to be unpacked and destroyed.

Note: In Move, _ is a wildcard pattern used in destructuring to ignore a field while still consuming the value. Destructuring must match all fields in a struct type. If a struct has fields, you must list all of them explicitly or use _ to ignore unwanted fields.

To illustrate the full flow of the application, let's put all the pieces together in a test.

## Layout

The initial layout of the metro pass application is simple. We define the Card type and the USES constant that represents the number of rides on a single card. We also add an error constant for the case when the card is empty.

```bash module book::metro_pass; /// Error code for when the card is empty. const ENoUses: u64 = 0;

/// Number of uses for a metro pass card. const USES: u8 = 3;

/// A metro pass card public struct Card { uses: u8 }

/// Purchase a metro pass card. public fun purchase(/ *pass a Coin* /): Card { Card { uses: USES } } ```

References are a way to show a value to a function without giving up ownership. In our case, when we show the Card to the inspector, we don't want to give up ownership of it, and we don't allow the inspector to use up any of our rides. We just want to allow the reading of the value of our Card and to prove its ownership.

To do so, in the function signature, we use the & symbol to indicate that we are passing a reference to the value, not the value itself.

```
bash /// Show the metro pass card to the inspector. public fun is_valid(card: &Card): bool {
card.uses > 0 }
```
```

Because the function does not take ownership of the Card, it can read its data but cannot write to it, meaning it cannot modify the number of rides. Additionally, the function signature ensures that it cannot be called without a Card instance. This is an important property that allows the Capability Pattern , which we will cover in the next chapters.

Creating a reference to a value is often referred to as "borrowing" the value. For example, the method to get a reference to the value wrapped by an Option is called borrow .

In some cases, we want to allow the function to modify the Card. For example, when using the Card at a turnstile, we need to deduct a ride. To achieve this, we use the &mut keyword in the function signature.

```
bash /// Use the metro pass card at the turnstile to enter the metro. public fun enter_metro(card: &mut Card) { assert!(card.uses > 0, ENoUses); card.uses = card.uses - 1; }
```

As you can see in the function body, the &mut reference allows mutating the value, and the function can spend rides.

Lastly, let's illustrate what happens when we pass the value itself to the function. In this case, the function takes the ownership of the value, making it inaccessible in the original scope. The owner of the Card can recycle it and thereby relinquish ownership to the function.

```
bash /// Recycle the metro pass card. public fun recycle(card: Card) { assert!(card.uses == 0, ENoUses); let Card { uses: _ } = card; }
```

In the recycle function, the Card is passed by value, transferring ownership to the function. This allows it to be unpacked and destroyed.

Note: In Move, _ is a wildcard pattern used in destructuring to ignore a field while still consuming the value. Destructuring must match all fields in a struct type. If a struct has fields, you must list all of them explicitly or use _ to ignore unwanted fields.

To illustrate the full flow of the application, let's put all the pieces together in a test.

```bash

# [test]

fun test_card_2024() { // declaring variable as mutable because we modify it let mut card = purchase();

card.enter_metro(); // modify the card but don't move it
assert!(card.is_valid()); // read the card!

card.enter_metro(); // modify the card but don't move it
card.enter_metro(); // modify the card but don't move it

card.recycle(); // move the card out of the scope

} ```
```

## References

References are a way to show a value to a function without giving up ownership. In our case, when we show the Card to the inspector, we don't want to give up ownership of it, and we don't allow the inspector to use up any of our rides. We just want to allow the reading of the value of our Card and to prove its ownership.

To do so, in the function signature, we use the & symbol to indicate that we are passing a reference to the value, not the value itself.

```
bash /// Show the metro pass card to the inspector. public fun is_valid(card: &Card): bool { card.uses > 0 }
```

Because the function does not take ownership of the Card, it can read its data but cannot write to it, meaning it cannot modify the number of rides. Additionally, the function signature ensures that it cannot be called without a Card instance. This is an important property that allows the Capability Pattern , which we will cover in the next chapters.

Creating a reference to a value is often referred to as "borrowing" the value. For example, the method to get a reference to the value wrapped by an Option is called borrow .

In some cases, we want to allow the function to modify the Card. For example, when using the Card at a turnstile, we need to deduct a ride. To achieve this, we use the &mut keyword in the function signature.

```
bash /// Use the metro pass card at the turnstile to enter the metro. public fun enter_metro(card:
&mut Card) { assert!(card.uses > 0, ENoUses); card.uses = card.uses - 1; }
```

As you can see in the function body, the &mut reference allows mutating the value, and the function can spend rides.

Lastly, let's illustrate what happens when we pass the value itself to the function. In this case, the function takes the ownership of the value, making it inaccessible in the original scope. The owner of the Card can recycle it and thereby relinquish ownership to the function.

```
bash /// Recycle the metro pass card. public fun recycle(card: Card) { assert!(card.uses == 0,
ENoUses); let Card { uses: _ } = card; }
```

In the recycle function, the Card is passed by value, transferring ownership to the function. This allows it to be unpacked and destroyed.

Note: In Move, _ is a wildcard pattern used in destructuring to ignore a field while still consuming the value. Destructuring must match all fields in a struct type. If a struct has fields, you must list all of them explicitly or use _ to ignore unwanted fields.

To illustrate the full flow of the application, let's put all the pieces together in a test.

```bash

# [test]

fun test_card_2024() { // declaring variable as mutable because we modify it let mut card = purchase();

card.enter_metro(); // modify the card but don't move it
assert!(card.is_valid()); // read the card!

card.enter_metro(); // modify the card but don't move it
card.enter_metro(); // modify the card but don't move it

card.recycle(); // move the card out of the scope

} ```
```

## Mutable Reference

In some cases, we want to allow the function to modify the Card. For example, when using the Card at a turnstile, we need to deduct a ride. To achieve this, we use the &mut keyword in the function signature.

```
bash /// Use the metro pass card at the turnstile to enter the metro. public fun enter_metro(card:
&mut Card) { assert!(card.uses > 0, ENoUses); card.uses = card.uses - 1; }
```

As you can see in the function body, the &mut reference allows mutating the value, and the function can spend rides.

Lastly, let's illustrate what happens when we pass the value itself to the function. In this case, the function takes the ownership of the value, making it inaccessible in the original scope. The owner of the Card can recycle it and thereby relinquish ownership to the function.

```
bash /// Recycle the metro pass card. public fun recycle(card: Card) { assert!(card.uses == 0,
ENoUses); let Card { uses: _ } = card; }
```

In the recycle function, the Card is passed by value, transferring ownership to the function. This allows it to be unpacked and destroyed.

Note: In Move, _ is a wildcard pattern used in destructuring to ignore a field while still consuming the value. Destructuring must match all fields in a struct type. If a struct has fields, you must list all of them explicitly or use _ to ignore unwanted fields.

To illustrate the full flow of the application, let's put all the pieces together in a test.

```bash

# [test]
```

fun test_card_2024() { // declaring variable as mutable because we modify it let mut card = purchase();

```
card.enter_metro(); // modify the card but don't move it
assert!(card.is_valid()); // read the card!

card.enter_metro(); // modify the card but don't move it
card.enter_metro(); // modify the card but don't move it

card.recycle(); // move the card out of the scope
```

} ```

## Passing by Value

Lastly, let's illustrate what happens when we pass the value itself to the function. In this case, the function takes the ownership of the value, making it inaccessible in the original scope. The owner of the Card can recycle it and thereby relinquish ownership to the function.

```
bash /// Recycle the metro pass card. public fun recycle(card: Card) { assert!(card.uses == 0,
ENoUses); let Card { uses: _ } = card; }
```

In the recycle function, the Card is passed by value, transferring ownership to the function. This allows it to be unpacked and destroyed.

Note: In Move, _ is a wildcard pattern used in destructuring to ignore a field while still consuming the value. Destructuring must match all fields in a struct type. If a struct has fields, you must list all of them explicitly or use _ to ignore unwanted fields.

To illustrate the full flow of the application, let's put all the pieces together in a test.

```bash

# [test]

fun test_card_2024() { // declaring variable as mutable because we modify it let mut card = purchase();

```
card.enter_metro(); // modify the card but don't move it
assert!(card.is_valid()); // read the card!

card.enter_metro(); // modify the card but don't move it
card.enter_metro(); // modify the card but don't move it

card.recycle(); // move the card out of the scope
```

} ```

## Full Example

To illustrate the full flow of the application, let's put all the pieces together in a test.

```bash

# [test]

fun test_card_2024() { // declaring variable as mutable because we modify it let mut card = purchase();

```
card.enter_metro(); // modify the card but don't move it
assert!(card.is_valid()); // read the card!

card.enter_metro(); // modify the card but don't move it
card.enter_metro(); // modify the card but don't move it

card.recycle(); // move the card out of the scope
```

} ```