Move Topics Explained

Absolutely! Here's a full list of **Move topics** I'll explain in **detail with examples**—just pick the ones you want to start with, or I can go one by one.

🔰 Beginner Topics

1. **What is Move? Why use Move for Blockchain?**
2. **Setting up Move CLI & Aptos CLI**
3. **Using Petra Wallet**
4. **Move Modules and Scripts**
5. **Move Basic Data Types (u8, u64, bool, address, vector)**
6. **Declaring & Using Structs in Move**
7. **Vector Usage (Add, Remove, Map)**
8. **Control Flow: If, While, Loops**
9. **Functions in Move**
10. **Generics in Move**
11. **Comments and Coding Conventions**

⚙️ Intermediate Topics

12. **Move Abilities (copy, drop, store, key)**
13. **Resource Management & Move Ownership Model**
14. **Modules, Imports, and `use` Statement**
15. **Visibility: `public`, `friend`, `entry`**
16. **Custom Errors & `abort` Handling**
17. **Assertions in Move**
18. **Working with Signers & Account Addresses**

19. **Unit Testing in Move (`#[test]` macros)**
20. **Package Structure and Compilation**

🚀 Advanced Topics

21. **Global Storage & Persistent Resources (`move_to`, `move_from`)**
22. **Event Emission in Move (`emit_event`)**
23. **Using `Table` and On-Chain Maps**
24. **View Functions (`view` keyword)**
25. **Upgradable Packages in Move**
26. **Resource Accounts and Named Addresses**
27. **Multi-Signature Support in Move**
28. **Move Bytecode and Move VM**
29. **Security & Formal Verification with Move Prover**

📚 Bonus

30. **Move Official Docs & Tutorials**
31. **Useful Aptos Dev Tools & Explorer**
32. **Deploying a Smart Contract on Aptos using CLI**

Would you like me to start from the **beginner**, or you want to pick a specific number/topic from above?

Just say something like:

> Explain Topic 6 - Structs in Move
> or

> Start from Topic 1 and go step-by-step.

Let's go! 🚀

Great! Let's dive into **Move Modules and Scripts**, one of the most fundamental topics in the Move language. I'll explain:

✅ 1. What is a Module in Move?

A **module** is like a smart contract in other blockchain languages. It defines:

- **Structs** (data types)
- **Functions** (logic)
- **Resources** (on-chain data that can't be copied)

It is deployed to an account and can be called by other accounts or scripts.

✅ 2. What is a Script in Move?

A **script** is like a one-time transaction. It:

- Executes a function (usually from a module)
- Can't define new structs or functions
- Is used by users to **interact** with modules

🔁 Analogy

Move Concept	Comparable In Solidity
Module	Smart Contract
Script	Transaction Function

📦 Example of a Module

Let's build a simple **Coin Bank Module**:

move

```
module MyCoin::Bank {
    use std::signer;

    // Define a resource (data owned by a user)
    struct Coin has key {
        value: u64,
    }

    // Initialize a Coin resource for the user
    public entry fun init_account(account: &signer) {
        move_to(account, Coin { value: 100 });
    }

    // Add coins to an account
    public entry fun deposit(account: &signer, amount: u64) {
        let coin = borrow_global_mut<Coin>(signer::address_of(account));
        coin.value = coin.value + amount;
    }

    // View coin balance
    public fun get_balance(addr: address): u64 {
        borrow_global<Coin>(addr).value
    }
}
```

🗒️ Example of a Script

Now let's write a script to call that module:

move

```
script {
    use MyCoin::Bank;

    fun main(account: &signer) {
        Bank::init_account(account);
        Bank::deposit(account, 50);
    }
}
```

This script:

- Initializes the account
- Deposits 50 coins

💡 Notes

- You **compile and publish** the module once.
- You **run scripts** for each transaction or interaction.
- Scripts **cannot declare structs or modules** – they can only call existing ones.

🛠 CLI Commands Example (Aptos)

To publish the module:

bash

```
aptos move publish --package-dir .
```

To run a script:

bash

```
aptos move run --function-id 'MyCoin::Bank::deposit' --args u64:100
```

🔚 Summary

Feature	Module	Script
Defines	Structs, Functions, Resources	Calls functions from modules
Usage	Deployed to the blockchain	One-time execution (tx)
State	Can store data	Cannot store data
Example	Coin Bank definition	Deposit/Withdraw transaction

Want me to walk you through **how to create, compile, and deploy a module + run a script** step-by-step with CLI? Or want the next topic like **Structs and Vectors**?

Perfect! Let's go over the **Primary Data Types in Move** with simple **examples and explanations**. These are the foundational building blocks for all logic in Move.

🧠 Move Primary Data Types

1. **Integers** (`u8`, `u64`, `u128`)

2. **Boolean** (`bool`)

3. **Address** (`address`)

1️⃣ Integers

Move supports three unsigned integer types:

- `u8` → 8-bit (0 to 255)
- `u64` → 64-bit (default for amounts, balances, etc.)

- `u128` → 128-bit (used for large numbers like token supply)

> **All are unsigned** – no negative values.

✅ Example

move

```
module Demo::IntExample {
    public fun add_numbers(a: u64, b: u64): u64 {
        a + b
    }

    public fun multiply(a: u128, b: u128): u128 {
        a * b
    }

    public fun small_number(): u8 {
        200
    }
}
```

🧮 Valid operations on integers:

- `+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`

2️⃣ Boolean (`bool`)

This is just like in most languages:

- `true` or `false`

✅ Example

move

```
module Demo::BoolExample {
    public fun is_positive(n: u64): bool {
        n > 0
    }
```

```
    public fun both_true(a: bool, b: bool): bool {
        a && b
    }

    public fun either_true(a: bool, b: bool): bool {
        a || b
    }

    public fun not_true(x: bool): bool {
        !x
    }
}
```

🔁 Boolean operations:

- `&&` → AND
- `||` → OR
- `!` → NOT
- `==`, `!=` → Comparison

3️⃣ Address (`address`)

The `address` type is a **20-byte** value that represents **on-chain accounts**.

You usually use this:

- For identifying users
- For storing who owns what

✅ Example

move

```
module Demo::AddressExample {
    use std::signer;

    public fun get_address(s: &signer): address {
```

```
        signer::address_of(s)
    }

    public fun is_owner(s: &signer, expected: address): bool {
        signer::address_of(s) == expected
    }
}
```

🧪 Type Check

Move is a **statically typed** language:

- You **must declare the type**
- Type mismatches will fail at **compile time**

move

```
let a: u64 = 100;
let b: bool = true;
let addr: address = @0x123;
```

❌ Common Mistakes

Mistake	Why it fails
`let x: u8 = 300;`	Exceeds `u8` limit (max 255)
`let a: u64 = true;`	Type mismatch
`@0x123 + 1`	You can't do math on addresses

🔚 Summary Table

Type	Use for	Example
`u8`	Small numbers (0–255)	`let x: u8 = 10;`
`u64`	Regular balances, amounts	`let amount: u64 = 1000;`
`u128`	Big values (like supply)	`let supply: u128 = 1_000_000;`
`bool`	True/False logic	`let active: bool = true;`
`address`	Blockchain account IDs	`let addr: address = @0x123;`

Want to go over **Vectors**, **Structs**, or **Control Flow** next? Let me know!

Awesome — let's dive into three essential types in Move: **Vector**, **Signer**, and **String**. These are used heavily in real-world smart contracts, especially in things like token balances, lists of data, and identity verification.

📦 1. Vector (`vector<T>`)

A **vector** is a growable list of elements of type `T`. It's the only native collection type in Move.

✅ Declaring & Using Vectors

move

```
module Demo::VectorExample {
    use std::vector;

    public fun sum_vector(v: vector<u64>): u64 {
        let mut sum = 0;
        let len = vector::length(&v);
        let mut i = 0;
```

```
        while (i < len) {
            sum = sum + *vector::borrow(&v, i);
            i = i + 1;
        }

        sum
    }

    public fun create_vec(): vector<u8> {
        vector::empty<u8>() // returns []
    }

    public fun add_item(): vector<u64> {
        let mut v = vector::empty<u64>();
        vector::push_back(&mut v, 42);
        vector::push_back(&mut v, 100);
        v
    }
}
```

🔧 Common Vector Functions

Function	Description
`vector::empty<T>()`	Create an empty vector
`vector::push_back(&mut vec, val)`	Add item
`vector::length(&vec)`	Get size
`vector::borrow(&vec, i)`	Get reference to element
`vector::remove(&mut vec, i)`	Remove and return element

🙋‍♂️ 2. Signer (`signer`)

`signer` is a **special type** used in entry functions to represent the **account signing the transaction**.

✅ Example

move

```
module Demo::SignerExample {
    use std::signer;

    // Only the signer can store something in their own storage
    struct MyData has key {
        value: u64,
    }

    public entry fun save_data(account: &signer, value: u64) {
        move_to(account, MyData { value });
    }

    public fun get_addr(account: &signer): address {
        signer::address_of(account)
    }
}
```

💡 Key Points:

- Only `entry` functions can receive `&signer`.
- Use `signer::address_of(&signer)` to get the user's address.
- Resources must be stored under the address of the signer.

🔤 3. String (Move doesn't have native strings!)

Move **does not support strings** directly. But you can simulate string-like behavior using `vector<u8>`.

✅ Simulated String (vector<u8>)

move

```
module Demo::StringLike {
    use std::vector;

    public fun to_ascii(): vector<u8> {
        b"hello" // Byte string literal
    }

    public fun greet(name: vector<u8>): vector<u8> {
        // Concatenate b"Hi, " + name
        let mut greeting = b"Hi, ";
        vector::append(&mut greeting, name);
        greeting
    }
}
```

🧠 Tips:

- `b"hello"` gives you a `vector<u8>`
- Useful for messages, logs, keys, etc.
- If you need real strings, do that off-chain or in frontend (e.g., DApp).

🧪 Summary Table

Type	Description	Example
`vector<T>`	Growable list of items	`vector<u64>`
`signer`	Special type for signer account	`&signer` in entry functions
`vector<u8>` (string-like)	Byte array	`b"hello"`

Want me to show how to **store a vector in a struct**, or maybe how to **loop over vector<u8> as a string**? Or move on to **Structs**, **Functions**, or **Control Flow**? Let me know!

Let's go through the next important set of **Move concepts**:

✅ Reference, Tuples & Unit, Local Variables & Scope

These help you write clean, readable, and memory-safe Move programs.

🔁 1. Reference (`&T`, `&mut T`)

References let you **borrow a value** without moving or copying it.

✳️ Types:

- `&T` → **Immutable** reference (read-only)
- `&mut T` → **Mutable** reference (can modify the value)

> Move uses *borrow checking* just like Rust to ensure safety.

✅ Example

move

```
module Demo::RefExample {
    public fun add_ref(a: &u64, b: &u64): u64 {
        *a + *b // Dereference with *
    }

    public fun modify(val: &mut u64) {
        *val = *val + 10;
    }
}
```

🧠 Key Rules:

- You can have **multiple immutable refs** OR **one mutable ref**, but not both at the same time.
- You must **dereference** using `*ref` to access the value.

🪢 2. Tuples & Unit Type

✨ Tuples

Tuples are **fixed-size collections** of values of different types.

✅ Example

move

```
module Demo::TupleExample {
    public fun make_tuple(): (u64, bool, address) {
        (100, true, @0x123)
    }

    public fun get_parts(): u64 {
        let (x, _, _) = make_tuple(); // pattern matching
        x
    }
}
```

🧩 Tuple Access

You can **pattern match** to extract elements.

🟤 Unit Type: `()`

The unit type is like `void` in other languages — it means **no meaningful value** is returned.

✅ Example

move

```
public fun say_hello(): () {
    // do something but return nothing
}
```

📦 3. Local Variables & Scope

🧠 Local Variables

- Declared with `let`
- Must be **initialized immediately**
- Immutable by default (no `mut` yet in Move)

✅ Example

move

```
module Demo::LocalVar {
    public fun calculate(): u64 {
        let a = 10;
        let b = 20;
        let sum = a + b;
        sum
    }
}
```

📏 Scope

- A variable only lives within the block it's declared.
- You can **reuse variable names** in different scopes.

move

```
public fun scoped(): u64 {
    let x = 10;
    {
        let x = 20; // this x shadows outer x
        x // returns 20
```

```
    };
    x // returns 10
}
```

✅ Summary Table

Concept	Purpose	Example
`&T`, `&mut T`	Borrowing values (read-only or modifiable)	`add_ref(&a, &b)`
Tuples	Return multiple values	`(u64, bool)`
Unit `()`	Return nothing	`public fun do_something(): () {}`
Local Vars	Temporary values in a function	`let x = 10;`
Scope	Controls variable lifetime	`{ let x = 20; }`

Would you like to continue with **Structs**, **Control Flow (if/else, loops)**, or **Functions & Generics** next?

Perfect! Let's break down the next essential topics in Move:

✅ Equality, Abort & Assert, Conditionals, While Loops

These are core for writing logical, safe, and efficient smart contracts in Move.

🔁 1. Equality

Move supports `==` and `!=` for comparing **primitive types** and **tuples**.

✅ Example:

move

```
module Demo::EqualityExample {
    public fun is_equal(a: u64, b: u64): bool {
        a == b
    }

    public fun not_equal(x: bool, y: bool): bool {
        x != y
    }

    public fun tuple_compare(): bool {
        let t1 = (1, true);
        let t2 = (1, true);
        t1 == t2 // true
    }
}
```

> ❗ Note: Custom `struct`s can't be compared using `==` unless you manually implement logic.

❌ 2. Abort & Assert

Move has built-in mechanisms for **error handling**:

🚨 `abort`

Stops execution with a given error code.

move

```
public fun fail_if_zero(x: u64) {
    if (x == 0) {
        abort 100; // Custom error code
```

```
    }
}
```

✅ `assert`

Shorthand to check conditions — aborts automatically with an error if condition fails.

move

```
use std::error;

public fun assert_example(x: u64) {
    assert(x > 0, error::invalid_argument(101));
}
```

🧠 When to Use:

Use Case	Tool
Stop on logic failure	`abort <code>`
Validate preconditions	`assert(condition, error_code)`

🔀 3. Conditionals: `if`, `else`

Conditionals work like in most languages.

✅ Example:

move

```
public fun max(a: u64, b: u64): u64 {
    if (a > b) {
        a
    } else {
        b
    }
}
```

- Every `if` expression **must return the same type** in all branches.
- No `else if` → just nest another `if`.

🔁 4. While Loops

Use `while` to **repeat logic** as long as a condition holds.

✅ Example:

move

```
public fun sum_upto(n: u64): u64 {
    let mut i = 0;
    let mut sum = 0;
    while (i <= n) {
        sum = sum + i;
        i = i + 1;
    }
    sum
}
```

❗ Move Notes:

- Move doesn't support `break` or `continue`, so you must manage conditions manually.
- Infinite loops can cause gas exhaustion — be careful in smart contracts!

✅ Summary Table

Concept	Syntax/Function	Purpose
Equality	`==`, `!=`	Compare values
Abort	`abort <code>`	Manually stop execution

Concept	Syntax/Function	Purpose
Assert	`assert(cond, code)`	Fail if condition not met
Conditionals	`if { } else { }`	Conditional branching
While Loops	`while (cond) {}`	Repeating logic

Want to continue with **Functions & Generics**, **Structs**, or jump into **Global Storage, Events, or Resource Management** next?

Great! Let's now dive into the heart of Move development: **Functions, Visibility, Structs & Resources, and Constants**.

✅ 1. Functions

Functions are blocks of reusable logic in Move. They can:

- Take parameters
- Return values
- Mutate state (if allowed)

✅ Basic Syntax:

```move
public fun add(a: u64, b: u64): u64 {
    a + b
}
```

🧠 Key Points:

- Parameters must include **types**

- Must always return a value or `()` (unit type)
- Use `return` for early exits or just place the result as the last expression

move

```
fun early_exit(x: u64): u64 {
    if (x > 100) {
        return 100;
    };
    x
}
```

🔓 2. Function Visibilities

Control where and how functions can be accessed.

Modifier	Access Level
`public`	Accessible from any module
`public(friend)`	Accessible from listed friend modules
`public(script)`	Callable from a transaction/script
no modifier	Private (within the module only)

✅ Examples:

move

```
fun private_fn(): u64 {
    1
}

public fun public_fn(): u64 {
    2
}
```

```
public(script) fun script_entry(): () {
    let val = public_fn();
}
```

> 🔐 Use `public(script)` only for entry points exposed to users.

🧱 3. Struct and Resources

✅ Struct

`struct` is used to define **custom data types**.

move

```
struct User has copy, drop {
    id: u64,
    name: vector<u8>,
}
```

➕ Abilities:

Ability	Meaning
`copy`	Can be duplicated
`drop`	Can be discarded
`store`	Can be stored in global storage
`key`	Acts like a resource (unique)

> A **resource** is a `struct` with the `key` ability – it can't be copied or dropped, only moved.

✅ Example (Resource):

move

```
struct Coin has key {
    value: u64,
}
```

You can now publish this `Coin` to an address using `move_to()` and retrieve it using `borrow_global()`.

📥 Example: Creating & Saving a Resource

move

```
module Demo::Wallet {
    struct Balance has key {
        amount: u64,
    }

    public fun create_balance(account: &signer) {
        let b = Balance { amount: 100 };
        move_to(account, b);
    }

    public fun read_balance(addr: address): u64 {
        borrow_global<Balance>(addr).amount
    }
}
```

🔢 4. Constants

Constants are **immutable values** defined at module level.

✅ Syntax:

move

```
const MAX_SUPPLY: u64 = 1_000_000;
```

🧠 Notes:

- Can only be **primitive types**
- Initialized at **compile time**
- Cannot be changed

✅ Example:

move

```
module Demo::Token {
    const DECIMALS: u8 = 6;

    public fun get_decimals(): u8 {
        DECIMALS
    }
}
```

✅ Summary

Concept	Usage Example	Notes
Function	`fun add(a: u64): u64 { a + 1 }`	Logic blocks, always return values
Visibility	`public`, `public(script)`, `fun`	Controls access level
Struct	`struct T {}`	Custom types with or without abilities
Resource (key)	`struct T has key {}`	Can be stored globally, not copied/dropped
Constant	`const X: u64 = 10;`	Immutable module-level values

Would you like to move on to:

- 📦 Global Storage (`move_to`, `borrow_global`, etc.)
- 📚 Generics
- 🔁 Vectors & Loops in-depth
- 🧪 Unit Testing in Move

Let me know what's next!

Awesome choice! These three topics — **Generics**, **Type Arguments**, and **Phantom Type Parameters** — are *core* to writing **reusable, flexible, and safe** smart contracts in Move.

🧠 1. Generics in Move

Generics allow you to write code that works with **any type**, not just one specific type.

✅ Example: Generic Struct

move

```
module Demo::GenericExample {
    struct Box<T> has copy, drop {
        value: T,
    }

    public fun wrap<T>(val: T): Box<T> {
        Box { value: val }
    }

    public fun unwrap<T>(b: Box<T>): T {
        b.value
    }
}
```

🔍 Explanation:

- `Box<T>` : a generic struct that can store any type `T`
- `wrap<T>` : a generic function that returns a `Box<T>`
- `unwrap<T>` : extracts the value of type `T` from the box

📌 2. Type Arguments

Type Arguments are how you pass specific types to generic structs/functions during **usage**.

✅ Example:

```
move
```

```
module Demo::MyStorage {
    struct Wrapper<T> has key {
        inner: T,
    }

    public fun store_u64(account: &signer, value: u64) {
        let item = Wrapper<u64> { inner: value };
        move_to<Wrapper<u64>>(account, item);
    }

    public fun get_u64(addr: address): u64 {
        borrow_global<Wrapper<u64>>(addr).inner
    }
}
```

🔍 How It Works:

- We defined a generic `Wrapper<T>`
- At **call time**, we instantiate it with `u64` : `Wrapper<u64>`

- The same `Wrapper<T>` can be reused for `bool`, `vector<u8>`, or any `struct`

> 🧠 **Type arguments** are Move's way to bring **type safety** and **reuse**.

👻 3. Phantom Type Parameters

A **Phantom Type** is a **type parameter that is not actually used** inside a struct or function — it's just used to **label or differentiate** resources by type.

> Great for tagging resources or making types unique *without storing extra data*.

✅ Syntax:

```move
struct Token<phantom T> has store {
    total_supply: u64,
}
```

✅ Real Example: Token Type Branding

```move
module Demo::PhantomExample {
    struct USD {}
    struct INR {}

    struct Coin<phantom T> has store, key {
        value: u64,
    }
```

```
    public fun mint_usd(account: &signer) {
        let c = Coin<USD> { value: 1000 };
        move_to<Coin<USD>>(account, c);
    }

    public fun mint_inr(account: &signer) {
        let c = Coin<INR> { value: 80000 };
        move_to<Coin<INR>>(account, c);
    }

    // This will only accept Coin<USD>, not Coin<INR>
    public fun check_balance_usd(addr: address): u64 {
        borrow_global<Coin<USD>>(addr).value
    }
}
```

🔍 What's Happening:

- `Coin<phantom T>` has no field that uses `T`, but Move still tracks the type
- Ensures that `Coin<USD>` and `Coin<INR>` are treated as **completely separate** types
- Prevents users from accidentally mixing them

✅ Summary Table

Feature	Description	Example
Generic Struct	Works with any type	`struct Box<T> { val: T }`
Generic Function	Parameterized by type	`fun wrap<T>(x: T): Box<T>`
Type Arguments	Provide actual types when using generics	`Box<u64>`, `Coin<USD>`
Phantom Type	Type that is tracked but unused in memory layout	`struct Token<phantom T>`

🧠 When to Use What?

You Want To...	Use
Store or operate on any type	**Generics**
Enforce type restrictions at runtime	**Type Arguments**
Tag or distinguish types without storing them	**Phantom Types**

Would you like next:

- 🔒 `Global Storage` & `move_to`, `borrow_global`
- 🧪 `Unit Tests` & test syntax
- 📦 `Module Publishing` and deployment

Let me know!

Awesome — this section is about **Move language design essentials**: how types behave (`abilities`), how to organize and share code (`uses`, `friends`), and how packages work via `Move.toml`.

Let's dive into:

✅ 1. Type Abilities

Abilities in Move define what you can *do* with a struct.

🔑 The 4 Abilities:

Ability	What it allows
`copy`	Duplicate the value (e.g., `let b = a`)
`drop`	Discard the value without using it
`store`	Save in global storage (e.g., `move_to`)
`key`	Resource semantics (unique, globally stored)

✅ Example:

```move
struct Coin has key, store {
    amount: u64,
}
```

🧠 Meaning:

- `key`: this can be stored with `move_to` / `borrow_global`
- `store`: can live in global storage
- No `copy` or `drop`: must be **moved**, not duplicated or discarded

---

🚫 Invalid Examples:

```move
// ❌ Missing store
move_to(account, some_struct); // Error if `store` not declared

// ❌ Can't drop a value
let _ = coin; // Error if no `drop`
```

Use `drop` and `copy` for memory-managed structs, and `store + key` for resources.

📦 2. Uses and Aliases

Use `use` statements to **import** other modules or structs for convenience.

✅ Syntax:

```move
use std::string;
use std::vector;
use 0x1::Token::Coin;
use 0x1::Math as M;
```

Syntax	Meaning
`use module::Name`	Import a module or struct
`use module::Item as X`	Import with an alias

✅ Example:

```move
use std::vector;

public fun create_vector(): vector<u8> {
    vector::empty<u8>()
}
```

> 🧠 Keeps code clean & avoids full path usage.

🤝 3. Friends

The `friend` keyword **grants access** to **private functions and structs** of another module.

✅ Use Case:

You want `ModuleB` to access `ModuleA`'s internal types or private logic.

✅ Example:

move

```
// Module A
module A {
    friend B;

    struct Secret has copy { val: u64 }

    fun get_secret(): Secret {
        Secret { val: 42 }
    }
}
```

move

```
// Module B
module B {
    use A;
```

```
    public fun peek(): u64 {
        let s = A::get_secret(); // allowed because A declared B as friend
        s.val
    }
}
```

> 🔒 Great for safe APIs where only certain modules get full access.

🧰 4. Packages (`Move.toml`)

Move packages are like Rust's `Cargo.toml` system.

The `Move.toml` file tells the Move build system about:

- Name
- Address mappings
- Dependencies
- Modules

✅ Basic Example `Move.toml`

toml

```toml
[package]
name = "MyToken"
version = "0.1.0"
authors = ["you"]

[addresses]
MyToken = "0x123"
Std = "0x1"

[dependencies]
MoveStdlib = { local = "../move-stdlib" }
```

🧠 Breakdown:

Section	Description
`[package]`	Info about your package (name, version)
`[addresses]`	Named addresses used in your code
`[dependencies]`	External packages you import

✅ Folder Structure

pgsql

```
MyToken/
│
├── Move.toml
└── sources/
    └── token.move
```

Use `move build`, `move test`, `move publish`, etc.

✅ Summary

Feature	Description
Abilities	Define behavior (copy, drop, store, key) for structs
Uses/Aliases	Shorten imports and reduce path clutter
Friends	Share private items between specific modules

Feature	Description
Packages	`Move.toml` defines build system, dependencies, and layout

Want to continue with:

- 📦 `Global Storage & Resource Management` (`move_to`, `borrow_global_mut`)
- 🧪 `Testing in Move` (unit testing & `#[test]`)
- 🔁 `Events, View Functions, Package Upgrade`
- 🧾 `Resource Accounts and Multisig`

Let me know what you'd like next!

Awesome — let's get into **Move Testing**!

Testing in Move is powerful and built into the Move Prover runtime. You can write simple tests, check behavior, and even **test expected failures** like aborted transactions.

✅ 1. `#[test]` – Basic Unit Test

Move allows you to annotate functions with `#[test]` to make them executable tests.

🧪 Example: Basic Test

move

```
module MyModule {
    public fun add(x: u64, y: u64): u64 {
        x + y
    }
```

```
    #[test]
    public fun test_add() {
        let result = add(2, 3);
        assert!(result == 5, 100); // aborts if not true
    }
}
```

🧠 Key Points:

- `#[test]` marks a function as a unit test.
- `assert!(condition, error_code)` checks expected behavior.
- If the condition is false → test **fails** with abort code.

✅ 2. `#[test_only]` – Helpers for Tests

Use `#[test_only]` when you want to define **helper functions** that should only be accessible **inside tests**, **not in production**.

🧪 Example: Test-only Function

move

```
module MyModule {
    #[test_only]
    public fun square(x: u64): u64 {
        x * x
    }

    #[test]
    public fun test_square() {
        let result = square(4);
        assert!(result == 16, 101);
    }
}
```

🔒 Why use `#[test_only]`?

- These functions aren't published to the blockchain.
- They can be used to test internal logic more flexibly.

❌ 3. Expected Failures (`expected_failure`)

Sometimes you want to **test that something fails** (like `assert!`, `abort`, or `borrow_global` failure).

Use `#[test(expected_failure)]` to **assert failure is expected**.

🧪 Example: Test Abort

move

```
module MyModule {
    public fun will_abort() {
        assert!(false, 42); // always aborts
    }

    #[test(expected_failure)]
    public fun test_expected_abort() {
        will_abort(); // Test will pass because it aborts
    }
}
```

🔍 Without `expected_failure`:

- This test would **fail** because it aborts.

🔍 With `expected_failure`:

- The test **passes** *only if it aborts*.

> 🧠 Useful for testing invalid input or edge cases that should fail!

🧪 Running Tests

Use the CLI:

```bash
move test
```

It will:

- Compile your modules
- Run all functions marked `#[test]`
- Report results

✅ Summary

Feature	Description
`#[test]`	Run as a unit test
`#[test_only]`	Helper code for testing only
`#[test(expected_failure)]`	Confirms the test fails (e.g., via `assert!` or `abort`)

✅ Bonus Tips

- Use `assert!(x == y, ERR_CODE)` for checking values
- Combine with `signer` and `move_to` to test full flows
- Modularize test logic using `#[test_only]`

Want me to show:

- A **complete Move test file** for something like `Coin` transfer?
- How to test **resources**, **event emission**, or **access control**?
- How to do **property-based testing** with the Move Prover?

Let me know!

Global Storage in Move

Global Storage in Move is used to store resources and data **globally**, accessible by address, across the blockchain. It allows resources (such as tokens or accounts) to be **persisted** and **accessed** by different transactions.

✅ Key Concepts of Global Storage

1. **Resources**: Types that can live in global storage. These are types that are tracked by the Move VM and have ownership.
2. `move_to`: Used to store a resource at a specific address.
3. `borrow_global` **and** `borrow_global_mut`: Used to borrow a reference to a resource stored at a specific address.
4. `remove_from`: Deletes a resource from storage.

🧑‍💻 1. Storing Resources with `move_to`

`move_to` is used to move a resource from local scope to global storage.

✅ Example:

```move
module GlobalStorageExample {
    struct Token has store {
```

```
        value: u64,
    }

    public fun store_token(account: &signer, value: u64) {
        let token = Token { value };
        move_to<Token>(account, token); // move the token to the account's address
    }
}
```

🧠 Explanation:

- `move_to<Token>(account, token)` moves the resource `token` to the **global storage** at the address of `account`.

> 🧠 **Important:** You must have `has store` for resources that are stored globally.

👩‍💻 2. Accessing Resources with `borrow_global` and `borrow_global_mut`

These functions allow you to access stored resources.

- `borrow_global<T>(addr: address)`: Borrows an **immutable** reference to the resource at the specified address.
- `borrow_global_mut<T>(addr: address)`: Borrows a **mutable** reference to the resource at the specified address (i.e., allows modifying it).

✅ Example: Borrowing Resources

move

```
module GlobalStorageExample {
    struct Token has store {
        value: u64,
    }

    public fun store_token(account: &signer, value: u64) {
        let token = Token { value };
        move_to<Token>(account, token);
```

```
    }

    public fun get_token_value(account: address): u64 {
        let token_ref = borrow_global<Token>(account); // immutable borrow
        token_ref.value
    }

    public fun increment_token_value(account: address) {
        let token_ref = borrow_global_mut<Token>(account); // mutable borrow
        token_ref.value = token_ref.value + 1;
    }
}
```

🧠 Explanation:

- `borrow_global<Token>(account)` gives an **immutable reference** to the `Token` at `account`.
- `borrow_global_mut<Token>(account)` allows **modifying** the resource.

🧑‍💻 3. Removing Resources with `remove_from`

`remove_from` is used to delete a resource from global storage.

✅ Example: Removing a Resource

move

```
module GlobalStorageExample {
    struct Token has store {
        value: u64,
    }

    public fun store_token(account: &signer, value: u64) {
        let token = Token { value };
        move_to<Token>(account, token);
    }

    public fun remove_token(account: address) {
        remove_from<Token>(account); // remove the token from the given address
```

```
    }
}
```

🧠 Explanation:

- `remove_from<Token>(account)` deletes the `Token` resource from the address's storage.

> 🧠 **Note**: After a resource is removed, it's no longer accessible.

👨‍💻 4. Updating Resources

When you borrow a resource mutably using `borrow_global_mut`, you can update its fields directly.

✅ Example: Updating a Resource

move

```
module GlobalStorageExample {
    struct Token has store {
        value: u64,
    }

    public fun store_token(account: &signer, value: u64) {
        let token = Token { value };
        move_to<Token>(account, token);
    }

    public fun update_token(account: address, new_value: u64) {
        let token_ref = borrow_global_mut<Token>(account); // mutable borrow
        token_ref.value = new_value;
    }
}
```

🧠 Explanation:

- `borrow_global_mut<Token>(account)` allows you to modify the `Token` resource by changing its `value`.

🤦 5. Checking if a Resource Exists

You can use `exists` to check if a resource exists at a particular address in global storage.

✅ Example: Checking for Existence

move

```
module GlobalStorageExample {
    struct Token has store {
        value: u64,
    }

    public fun store_token(account: &signer, value: u64) {
        let token = Token { value };
        move_to<Token>(account, token);
    }

    public fun token_exists(account: address): bool {
        exists<Token>(account) // returns true if resource exists, false otherwise
    }
}
```

🧠 Explanation:

- `exists<Token>(account)` checks if a `Token` resource exists at the given address.

✅ Key Points Summary:

Operation	Description	Example Usage
`move_to`	Moves a resource to global storage	`move_to<Token>(account, token)`
`borrow_global`	Immutable borrow of a resource from global storage	`borrow_global<Token>(account)`

Operation	Description	Example Usage
`borrow_global_mut`	Mutable borrow of a resource from global storage	`borrow_global_mut<Token>(account)`
`remove_from`	Removes a resource from global storage	`remove_from<Token>(account)`
`exists`	Checks if a resource exists in global storage	`exists<Token>(account)`

🔧 Advanced Storage Operations

- **Global Struct Initialization**: Initialize resources in storage with complex initialization logic.
- `move_from`: Used to move a resource out of storage and into a local variable.

Would you like to explore more on:

- Advanced **resource management**?
- Implementing **state machines** with global storage?
- **Event emission** or **resource access control**?

Let me know your next steps!

Move Standard Library & Coding Conventions

1. **Move Standard Library**

The **Move Standard Library** provides core functionality and predefined modules to perform essential operations. It helps developers build applications efficiently by offering common utility functions, data structures, and operations for manipulating resources and managing global storage.

Key Modules in the Move Standard Library

Here are some of the most commonly used modules from the Move Standard Library:

1. `Vector` **Module**:
   - **Purpose**: Provides functions for working with vectors (dynamic arrays).
   - **Functions**:
     - `Vector::push_back`: Add an element to the end of a vector.
     - `Vector::pop_back`: Remove an element from the end of a vector.
     - `Vector::len`: Get the length of the vector.

   Example:

   ```
   move

   import 0x1::Vector;

   public fun example() {
       let v = Vector::empty<u64>();
       Vector::push_back(&mut v, 10);
       let length = Vector::len(&v);
   }
   ```

2. `Option` **Module**:
   - **Purpose**: Provides functions for working with optional values (`Option<T>`).
   - **Functions**:
     - `Option::is_some`: Check if an option has a value.
     - `Option::is_none`: Check if an option is `None`.
     - `Option::unwrap`: Extract the value from an `Option`, panics if `None`.

   Example:

   ```
   move

   import 0x1::Option;

   public fun example() {
       let opt = Option::some<u64>(42);
   ```

```
    let value = Option::unwrap(opt);
}
```

3. `Signer` **Module**:
   - **Purpose**: Provides functionality for handling the signing of transactions and managing accounts.
   - **Functions**:
     - `Signer::address_of`: Get the address of a signer.
     - `Signer::borrow`: Borrow a resource from the signer.

   Example:

```move
import 0x1::Signer;

public fun get_address(signer: &signer): address {
    Signer::address_of(signer)
}
```

4. `Math` **Module**:
   - **Purpose**: Provides basic mathematical operations (e.g., addition, subtraction).
   - **Functions**:
     - `Math::add`: Adds two values.
     - `Math::mul`: Multiplies two values.

   Example:

```move
import 0x1::Math;

public fun add_two_numbers(x: u64, y: u64): u64 {
    Math::add(x, y)
}
```

5. `Resource` **Module**:

- **Purpose**: Provides functions for working with resources (used for managing global storage).
- **Functions**:
  - `Resource::move_to`: Move a resource to storage.
  - `Resource::borrow`: Borrow a resource from storage.

Example:

```move
import 0x1::Resource;

public fun store_resource(account: &signer, value: u64) {
    let res = Resource::new(value);
    move_to(account, res);
}
```

---

2. Coding Conventions in Move

Move has several best practices and coding conventions that developers follow to write clean, maintainable, and efficient code. Here are some of the key coding conventions:

General Coding Guidelines:

1. **Consistency in Naming**:
   - Use **snake_case** for function names, variables, and parameters.
   - Use **CamelCase** for struct and module names.

   Example:

```move
struct MyToken {
    value: u64,
```

```
}

public fun transfer_tokens(amount: u64): bool {
    // function implementation
}
```

2. **Indentation and Spacing**:
   - Use 4 spaces for indentation.
   - Ensure there is a space after `,` in function arguments for better readability.
   - Always leave a blank line between function definitions or after the end of a module.

   Example:

   ```
   move

   public fun create_token(account: &signer, amount: u64) {
       let token = Token { value: amount };
       move_to<Token>(account, token);
   }
   ```

3. **Avoid Long Functions**:
   - Keep functions short and focused on a single task. If a function is getting long, consider refactoring it into smaller, more manageable functions.
   - Aim for a **single responsibility** for each function.
4. **Error Handling**:
   - Use `assert!` statements to validate conditions and abort transactions in case of errors.
   - Return values should use `Option<T>` or `Result<T, E>` when a function might fail.

   Example:

   ```
   move

   public fun safe_add(x: u64, y: u64): u64 {
       assert!(x + y <= u64::MAX, 101); // Check for overflow
       x + y
   }
   ```

5. **Use of Resources**:

- Resources must be used with care. They are **owned** and **move** in and out of storage. Always ensure that you are not trying to accidentally move a resource twice.
- Avoid unnecessary global storage operations when testing or developing, as they can increase transaction costs.

6. **Documentation**:
   - **Comment your code** generously. Each function should have a clear description of what it does, what arguments it takes, and what it returns.
   - Document **complex logic**, especially around resource management, and when interacting with external contracts or modules.

   Example:

```move
// This function transfers tokens from one account to another
public fun transfer_tokens(sender: &signer, receiver: address, amount: u64) {
    let sender_balance = borrow_global_mut<Token>(sender);
    let receiver_balance = borrow_global_mut<Token>(receiver);

    // Deduct from sender and add to receiver
    sender_balance.value = sender_balance.value - amount;
    receiver_balance.value = receiver_balance.value + amount;
}
```

Move-specific Conventions:

1. **Resource Handling**:
   - When working with resources, always ensure that you are either **moving** a resource into storage or **borrowing** it with `borrow_global` or `borrow_global_mut`.
2. **Module Organization**:
   - Group related functions logically in **modules** to maintain clarity and reusability.
   - Ensure functions in modules have **clear visibility** (public vs. private) based on the required access levels.
3. **Use of Abilities**:

- When defining a struct that is intended to be stored, use the appropriate `has store` ability to indicate that it can be stored in global storage.

Example: Full Code Snippet

move

```
module TokenModule {
    struct Token has store {
        value: u64,
    }

    // Transfer tokens from one account to another
    public fun transfer_tokens(sender: &signer, receiver: address, amount: u64) {
        let sender_balance = borrow_global_mut<Token>(sender);
        let receiver_balance = borrow_global_mut<Token>(receiver);

        // Ensure sender has enough balance
        assert!(sender_balance.value >= amount, 101);

        // Transfer amount
        sender_balance.value = sender_balance.value - amount;
        receiver_balance.value = receiver_balance.value + amount;
    }

    // Create a new token and store it at the given account address
    public fun create_token(account: &signer, value: u64) {
        let token = Token { value };
        move_to<Token>(account, token);
    }
}
```

Summary of Key Guidelines:

- **Naming**: `snake_case` for functions and variables, `CamelCase` for structs and modules.

- **Indentation**: Use 4 spaces for indentation.
- **Function Length**: Keep functions short and focused on a single task.
- **Error Handling**: Use assertions and appropriate return types like `Option<T>` or `Result<T, E>`.
- **Resource Handling**: Be cautious of the ownership and movement of resources.
- **Documentation**: Add comments to clarify complex logic or functionality.

Let me know if you want more examples or if you want to dive deeper into any of these topics!

Advanced Move Topics Explained in Detail

1. View Functions

View functions are special types of functions in Move that allow reading from the blockchain without modifying any global state. These functions are often used for querying the current state of resources, checking balances, or accessing metadata. View functions don't perform any transaction costs themselves, as they do not alter the blockchain state.

Key Features:

- They **do not mutate global state**.
- They can **read** global resources.
- They are **free of cost** in terms of gas, as they don't modify state.

Example:

move

```
module 0x1::Token {
    resource struct Token {
        balance: u64,
    }

    public fun get_balance(account: address): u64 {
        let token = borrow_global<Token>(account);
        token.balance
```

```
    }
}
```

In this example:

- `get_balance` is a view function that fetches the `balance` of a given account without modifying the state.

Notes:

- View functions are essential for reading data from the blockchain to display to users, without incurring transaction fees.

---

2. Events

Events in Move are used to log significant occurrences during contract execution. They allow smart contracts to record information on-chain and can be used by external entities (such as front-end applications or other smart contracts) to track specific actions.

Key Features:

- Events are **immutable** and recorded in the transaction logs.
- Events can be used to **notify external systems** about significant actions (e.g., a transfer of tokens).
- They are not stored in global state; rather, they are logs that are appended to a blockchain.

Example:

move

```
module 0x1::Token {
    public struct TransferEvent has store {
        from: address,
        to: address,
        amount: u64,
    }

    public fun emit_transfer_event(sender: address, recipient: address, amount: u64)
{
```

```
        let event = TransferEvent {
            from: sender,
            to: recipient,
            amount: amount,
        };
        emit event;
    }
}
```

In this example:

- The `emit_transfer_event` function emits an event each time a transfer occurs.
- The `TransferEvent` struct contains information about the transfer, such as the sender, recipient, and the amount transferred.
- The `emit event` keyword logs this information to the blockchain.

Notes:

- Events are useful for external listeners or frontend apps that need to track changes without querying the blockchain repeatedly.
- They help in **decoupling** the logic of state changes and external systems.

---

3. **Package Upgrade**

In Move, upgrading packages (smart contracts) is possible. This allows developers to modify the functionality of their contracts deployed on-chain by uploading a new version of the contract. However, upgrading a package is a delicate process, and Move's design encourages making contracts backward-compatible to ensure smooth upgrades.

Key Concepts:

- Move packages are typically immutable, but they can be **upgraded** through controlled processes.
- Package upgrades are necessary to deploy **bug fixes**, add new features, or patch vulnerabilities.

Steps for upgrading:

1. **Create New Version**: Modify your existing contract and add new features or fix bugs.

2. **Test**: Make sure the new version works with existing state and is compatible.
3. **Deploy New Package**: Deploy the new version as a new contract package.
4. **Upgrade**: Some contracts may have mechanisms that allow you to point to the new version of the package.

Example: While there is no direct upgrade function in Move, a common approach is to deploy a new version of the contract and use an upgradeable resource pattern (e.g., through proxies).

---

4. **Resource Account (Create, Deploy Contract, Upgrade Contract)**

A **resource account** in Move is a special account designed to hold **resources** like tokens, stateful contract objects, or assets. It's essential for managing resources securely and is often used in the context of **contract deployment** and **upgrades**.

Key Features:

- A resource account holds **global resources**.
- Used to **deploy contracts**, **create objects**, and **upgrade contracts**.

Creating Resource Accounts:

- Resource accounts can be created using `Signer` capabilities.

Deploying a Contract:

- When deploying a contract, a resource account may hold the contract's code (e.g., a smart contract).

Upgrading Contracts:

- To upgrade a contract, you deploy a **new contract package** and point any **resource accounts** or **objects** to the new version.

Example:

move

```
module 0x1::ResourceAccount {
    public fun create_resource_account(owner: address): address {
        let account = Signer::address_of(&owner);
```

```
        move_to(account, ResourceAccount::new());
        account
    }
}
```

In this example:

- `create_resource_account` deploys a new resource account to hold resources.
- The contract deployer (owner) can then interact with this account.

---

5. Object (Create, Deploy Contract)

An **object** in Move is a storage unit that is owned by an address. It can represent things like user wallets, contract state, or other user-specific data. Objects are deployed on-chain, and they can hold mutable state.

Key Features:

- Objects are **created** via Move contracts and deployed in storage.
- Objects can have **mutable state** and are accessed by their owner.

Creating and Deploying Objects:

- You define objects using resources and store them in a global storage location.

Example:

move

```
module 0x1::Wallet {
    resource struct Wallet {
        balance: u64,
    }

    public fun create_wallet(owner: address): address {
        let wallet = Wallet { balance: 0 };
        move_to(account, wallet);
        account
    }
}
```

In this example:

- `create_wallet` creates a `Wallet` object for an address (`owner`) and stores it on-chain.
- The `Wallet` object has a `balance` field, which is mutable and can be modified later.

Notes:

- Objects can be **accessed** and **modified** by their creator, allowing for dynamic contract states.

---

6. Multisig Onchain

Multisig is a concept where multiple parties must sign off on a transaction before it can be executed. In Move, you can implement **on-chain multisignature wallets** that require multiple signers to approve a transaction before it takes place.

Key Features:

- Multiple **signers** must agree before a transaction is executed.
- Can be useful for **governance**, **joint control** of resources, or **enhanced security**.

Example:

move

```
module 0x1::Multisig {
    resource struct MultisigWallet {
        required_signers: vector<address>,
        approved_signers: vector<address>,
    }

    public fun create_wallet(signers: vector<address>): address {
        let wallet = MultisigWallet {
            required_signers: signers,
            approved_signers: Vector::empty<address>(),
        };
        move_to(account, wallet);
        account
    }

    public fun approve_transaction(wallet: &mut MultisigWallet, signer: address) {
```

```
        if (!Vector::contains(&wallet.approved_signers, signer)) {
            Vector::push_back(&mut wallet.approved_signers, signer);
        }
    }

    public fun execute_transaction(wallet: &MultisigWallet): bool {
        if (Vector::len(&wallet.approved_signers) >= 2) {
            // Execute the transaction
            true
        } else {
            false
        }
    }
}
```

In this example:

- `MultisigWallet` is a resource that holds the list of required and approved signers.
- `approve_transaction` allows a signer to approve a transaction.
- `execute_transaction` checks if enough signers have approved before executing.

Summary of Key Concepts:

- **View Functions**: Functions that read data without modifying state and have no gas costs.
- **Events**: Logs to track significant actions (like transfers) on the blockchain.
- **Package Upgrade**: The process of deploying a new version of a smart contract and potentially upgrading resources.
- **Resource Account**: Special accounts that manage resources and deploy contracts. Used for contract creation and upgrades.
- **Objects**: Entities that store mutable state on-chain and can represent things like user wallets.
- **Multisig Onchain**: A contract that requires multiple signers to approve a transaction before it is executed.

These advanced topics in Move help developers create more flexible, secure, and upgradable smart contracts. Let me know if you'd like further clarification or examples on any of these topics!