Move

the Language for Secure Next Gen Smart Contracts

6th Scientific School on Blockchain & DLTs

Mirko Zichichi

mirko.zichichi@iota.org

IOTA

Resource

Resource

Flexibility

Resource

Flexibility

Security


Resource

Libra → Diem →

https://www.theverge.com/2019/6/26/18716326/facebook-libra-cryptocurrency-blockchain-irs-starbucks

"The scarcest resource in the world is not time or money, but man's brain power.

When these are used to develop software, if you can amplify brain capacity, i.e. do more per unit of time, this is one of the most impactful things you can achieve."

Blackshear, Sam, et al. "Move: A Language With Programmable Resources"
https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-programmable-resources/2020-05-26.pdf (2020).

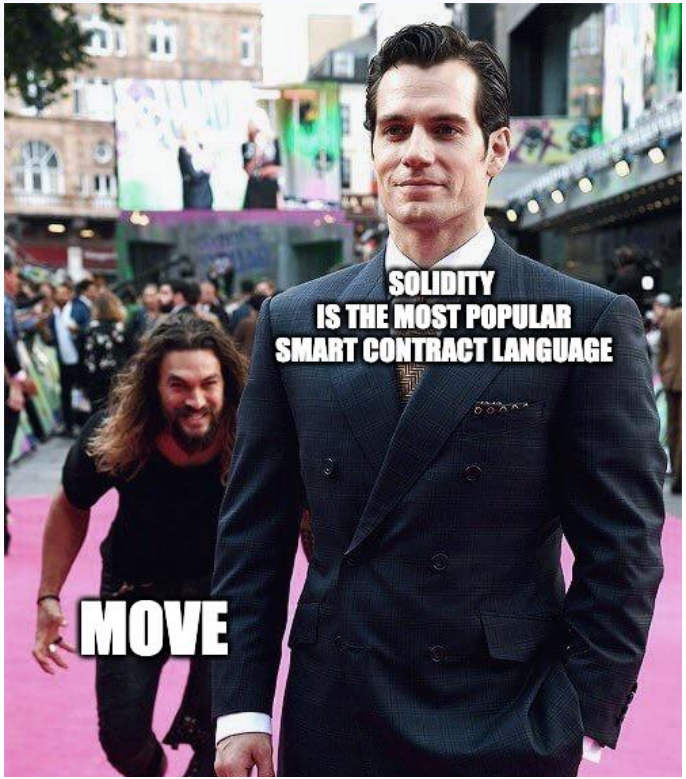
Move → Resource-oriented programming

- **Tangible** programming experience

- Linked to the physical intuitions of
  - **Exchange** → movement, transfer
  - **Ownership** → access control, possession

Criticism to existing blockchain languages

Criticism to existing blockchain languages
→ Ethereum Virtual Machine/Solidity



The Rise of Alternative Virtual Machines (AltVMs)

Criticism to existing blockchain languages

1. Indirect **asset representation**

Encoding assets using an integer number

→ but an integer is **not equivalent to an asset**.

```
mapping(address => uint) private balance;
```

Criticism to existing blockchain languages

2. Scarcity control of an asset is not built into the language

Criticism to existing blockchain languages

3. Access control not flexible

Criticism to existing blockchain languages

1. Indirect **asset representation**

2. Scarcity control of an asset is not built into the language

3. Access control not flexible

Move

*Representation of **state transitions** enabling ownership of **digital resources** to be encoded in an open source system*

First-class Resources

First-class Resources

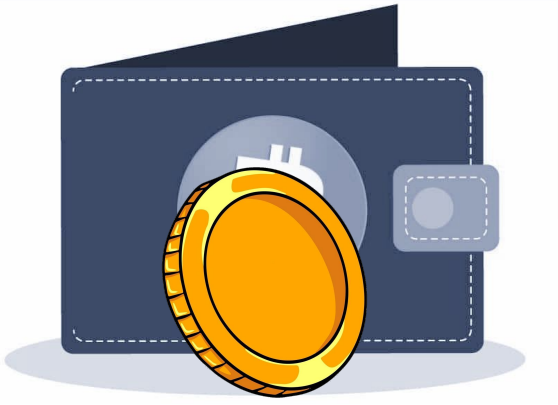
It provides the possibility of defining customized resource types with a semantics inspired by **linear logic**:

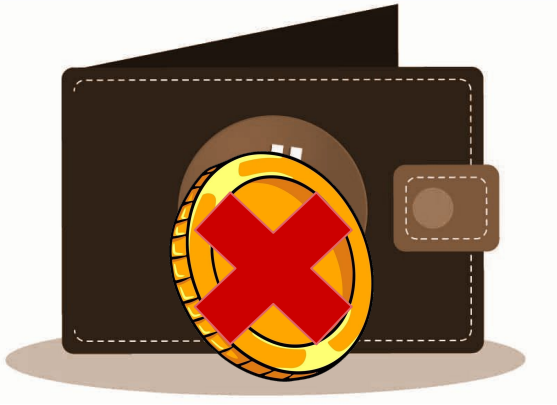
Move

First-class Resources

It provides the possibility of defining customized resource types with a semantics inspired by **linear logic**:

- a resource can never be implicitly copied or discarded



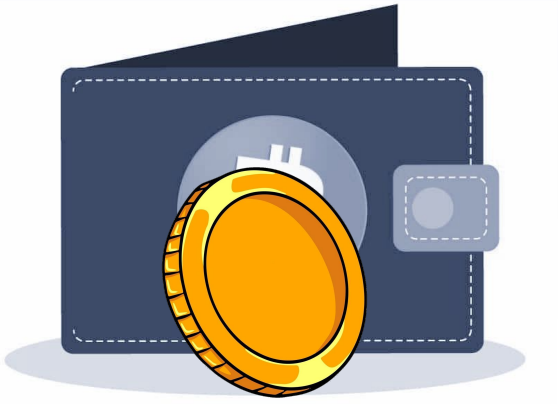


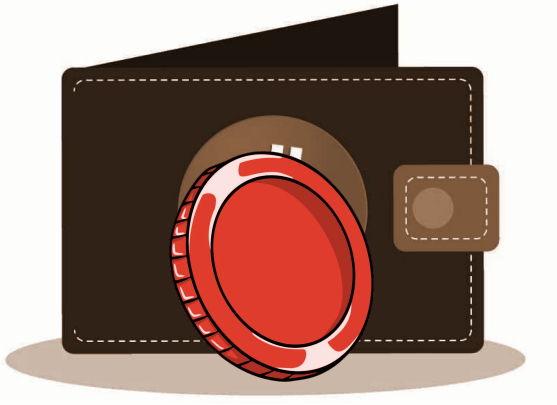
Move

First-class Resources

It provides the possibility of defining customized resource types with a semantics inspired by **linear logic**:

- a resource can never be implicitly copied or discarded



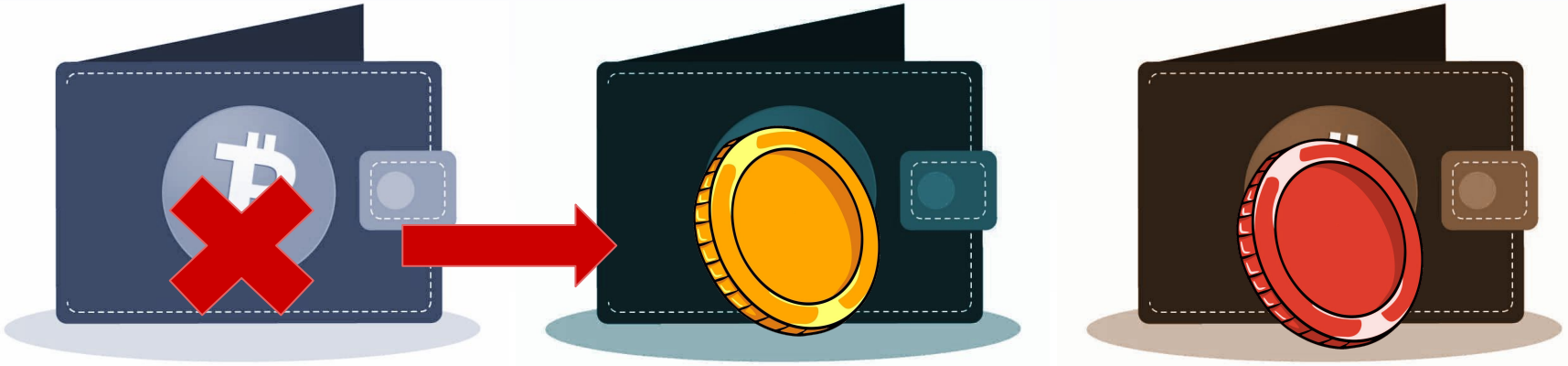


Move

First-class Resources

It provides the possibility of defining customized resource types with a semantics inspired by **linear logic**:

- a resource can never be implicitly copied or discarded
- only moved between the memory locations of the programme.



Move

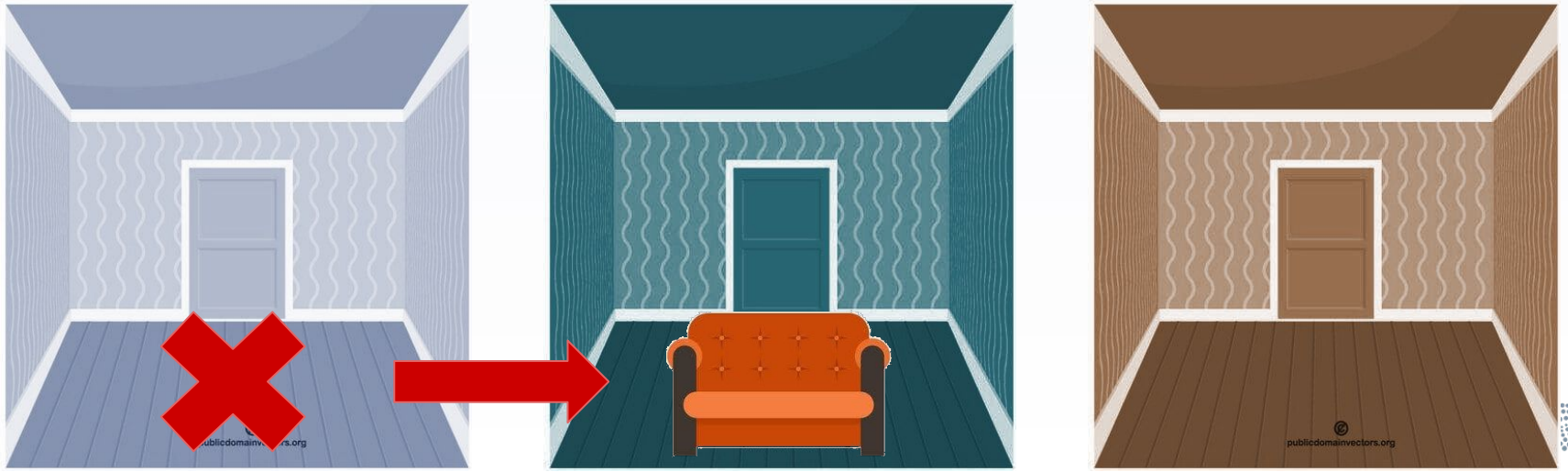
First-class Resources

Software that behaves in line with
your physical intuition.

Move

First-class Resources

Software that behaves in line with
your physical intuition.



Move

First-class Resources

Software that behaves in line with
your physical intuition.

Move

First-class Resources

Move programmers can **protect access to critical operations** on resources through the

- **Modules**: contain resource types and procedures that encode rules for resources.

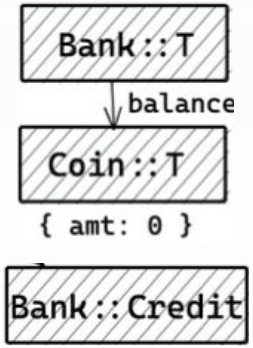
Move

First-class Resources

Solidity

Address	Ether Balance	Data
0x2	3.4	`contract Bank` `mapping (address => uint) credit;`

Move




```
module Bank
use 0x0::Coin;
resource T { balance: Coin::T }
resource Credit { amt: u64, bank: address }
```

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).


Resource


Flexibility

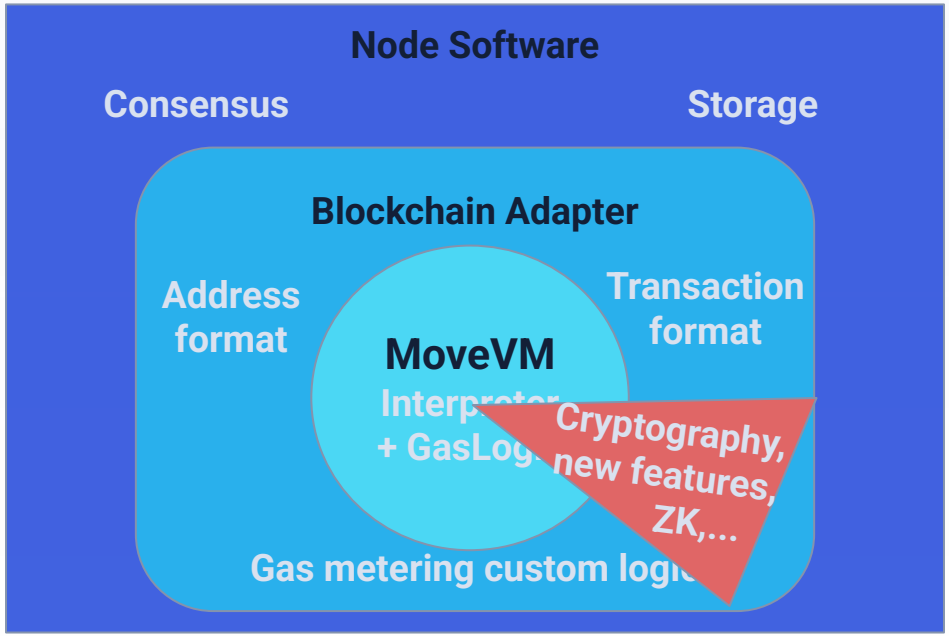
Move Virtual Machine

- **Blockchain agnostic:** we define how accounts and transactions work

- Core VM is **easily extensible** with:
  - Cryptography, signature schemes, ZKP verifiers
  - Blockchain specific features (mana generation, system transactions, account concept, etc.)

- Built-in **gas metering and safe math**: no undefined behavior is possible

Move Modularity



Move on Account vs Object Ledger

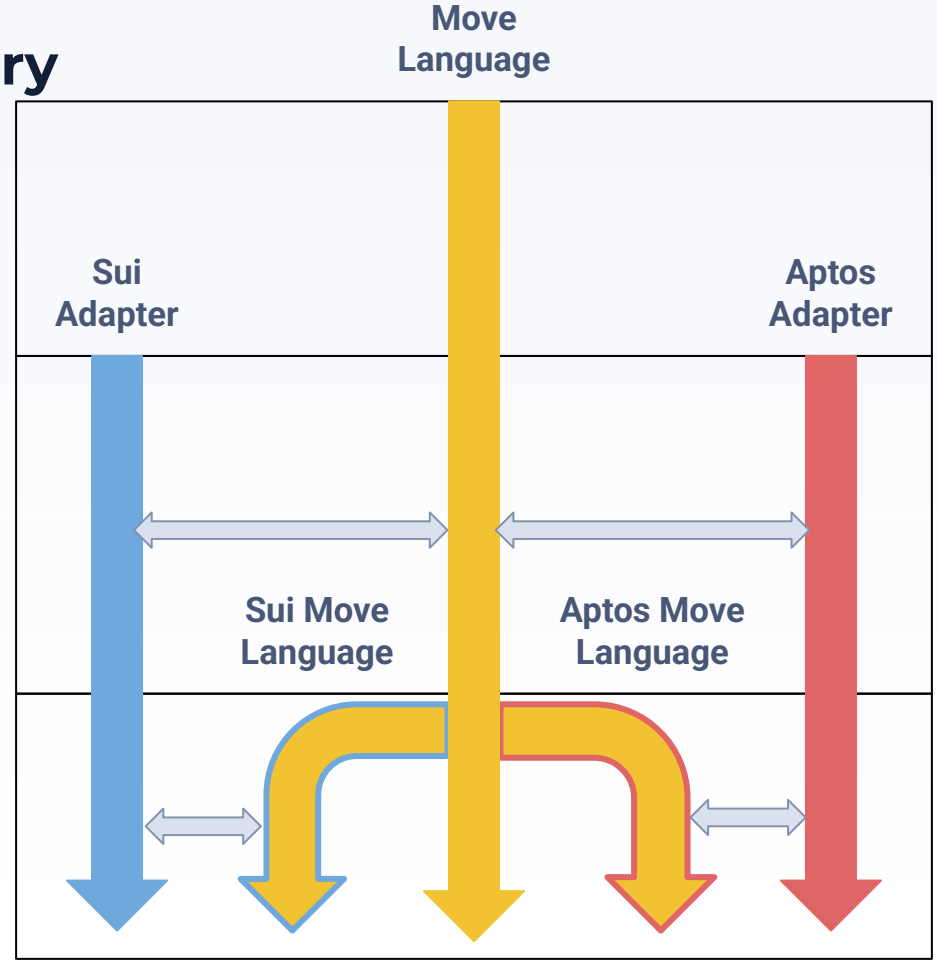
- **Unified Memory - Account Based Ledger**: EVM, WASM, ISC, Aptos, Core Move
  - Only sequential* execution
  - Convenient as you can access any memory location without prior request
- **Partitioned Memory - Object Based Ledger**: Sui Move, Cardano, Radix, Stardust, etc.
  - Parallel execution is possible, as **each SC names which objects it will touch**
  - Heavy usage of a particular SC doesn't degrade others
  - Execution needs only a fraction of the memory
  - UTXO is a special case of the object ledger

Move History

Early Move
Libra/Diem
2018-2021

Move Adapters
Sui & Aptos
2022-2023

Move 2024
Sui & Aptos Forks

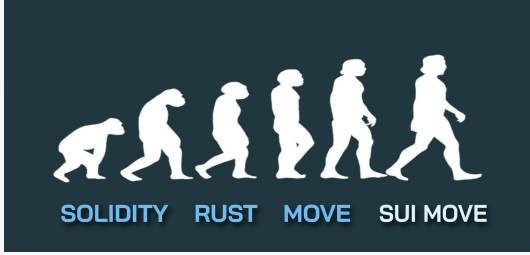



Flexibility


Security

Move

Security



- Inherits **memory and type safety** concepts from Rust
  - The compiler catches errors that would not normally be detected in other compilers (e.g. Solidity)

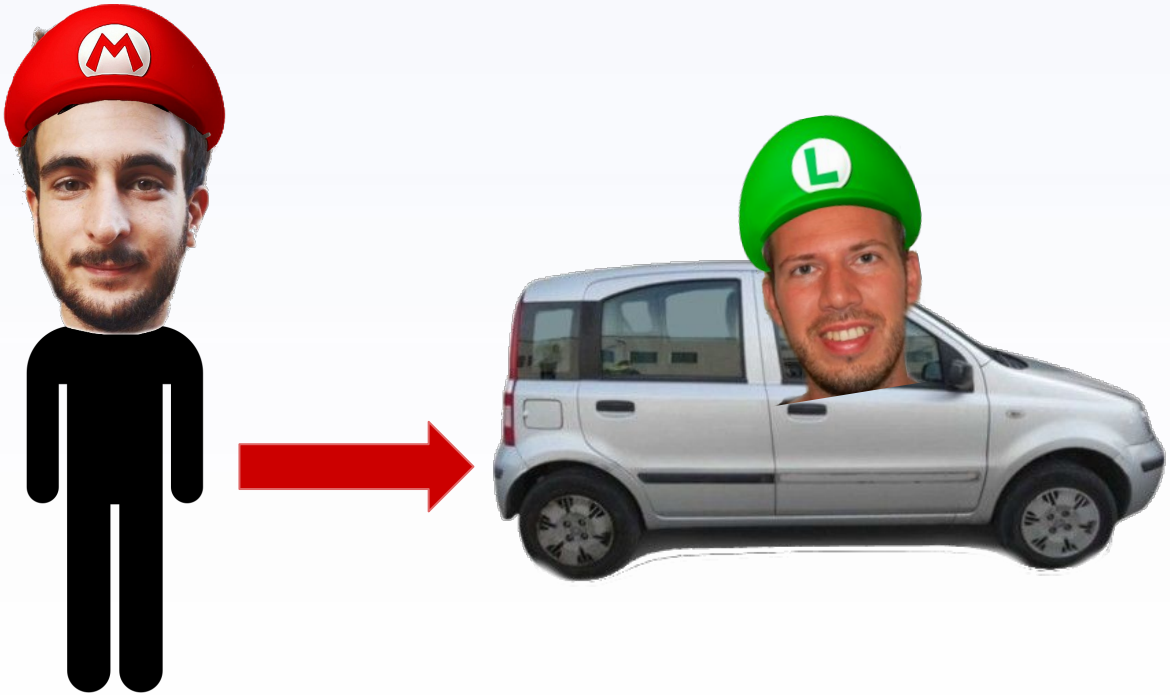
- **Resource safety**
  - Simple types like integers and addresses → can be copied
  - resources → can only be moved.
  - use of **linear logic prevents 'double spending'** (moving a resource twice).

Security

- **Access Control by default**
  - Forced by the language even though the programmer may forget to implement it.
- **Limited mutability**
  - Any mutation of a value in Move occurs via a ***'reference'*** as in Rust.
    - **by-value** → value
    - **mutable** → &mut value
    - **read-only** → &value

Pass a value to a function *by-value*

Pass a value to a function *by-value*



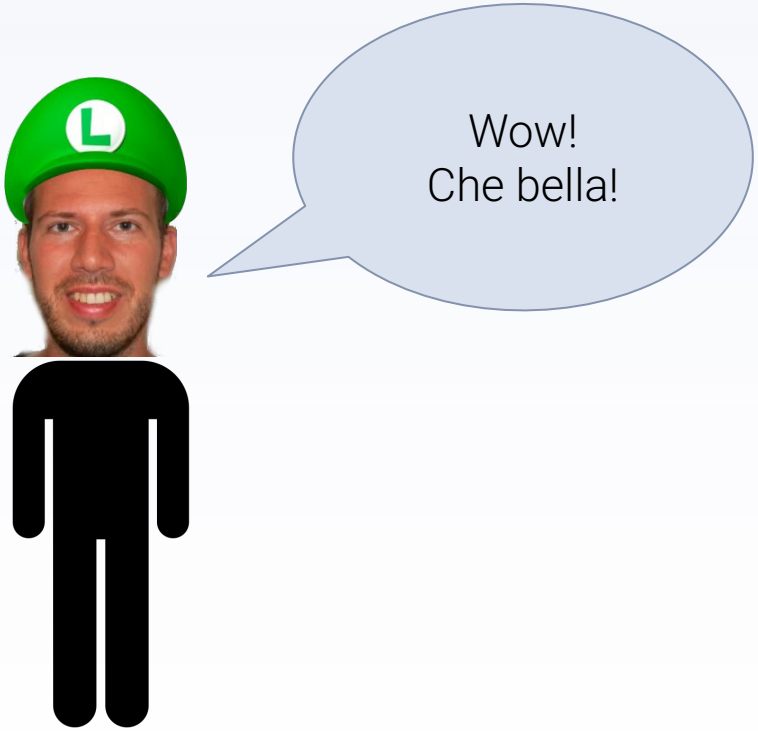
Pass a value to a function *by-value*

“Borrow” a value with *mutable ref (&mut)*

“Borrow” a value with *mutable ref (&mut)*

“Borrow” a value with *mutable ref (&mut)*

“Borrow” a value with *read-only ref (&)*



Security

Double check:

- the **high-level** programming language
  - is compiled using a **compiler that verifies security properties**

- the untyped **low-level** programming language
  - performs **security checks at runtime**

Move Security

Move smart contracts can be easily **Formally Verified**

asymptotic-code/**sui-prover**

Formal Verification tool for Move on Sui

4 Contributors

37 Issues

6 Stars

1 Fork

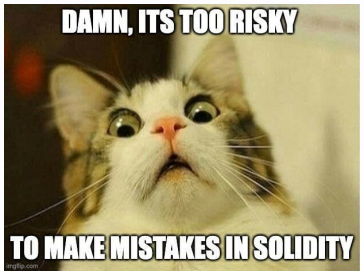
Move

Security

NO reentrancy.

Solidity

```
function withdraw() {
  uint amt = credit[msg.sender];
  msg.sender.transfer(amt);
  credit[msg.sender] = 0;
}
```



Move

```
fun withdraw(credit: Credit): Coin::T {
  Credit { amt, bank } = move credit;
  let t = borrow_global<T>(move bank);
  return Coin::withdraw(
    &mut t.balance, move amt
  );
}
```

Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

Security

NO reentrancy.

Main cause of reentrancy

→ **dynamic dispatch**:
within a smart contract you have a function whose definition is not known in advance to the developer.

In Move each time a function is called, the code that is called is statically known (static dispatch).


Security

IOTA flavored Move

Key differences between (Diem/Aptos) Move and IOTA/Sui Move (1/2)

- **Object-Centric Global Storage**
  - In (Diem) Move, transactions can **freely access resources,** *move_to* and *move_from*.
  - In IOTA Move transaction inputs are *explicitly specified using unique identifiers* for **objects** (as opposed to resources) and **packages** (sets of modules).
- **Addresses Represent Object IDs**
  - IOTA repurposes the address type as a **32-byte identifier** used for both *objects (object id)* and *accounts (address)*.
- **Objects with Key Ability and Globally Unique IDs**
  - In (Diem) Move, the *key ability* indicates that a type is a **resource**, which, along with an account address, can serve as a key in global storage.
  - In IOTA Move, the *key ability* denotes an **object type** and requires the struct's first field to be ***id: UID*** (which becomes the object id).

https://docs.iota.org/developer/iota-101/move-overview/

0. Basics - Custom Types

A **structure** in IOTA Move is a *custom type* that contains *key-value pairs*, where the key is the name of a property, and the value is what's stored.

Struct

```
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

0. Basics - Abilities

- Abilities are keywords in IOTA Move that define **how types behave at the compiler level**
  - ***copy***: the value of this type can be copied
    - usually basic types: Coin is an asset type that should not be duplicated, so it should not have copy ability
  - ***drop***: the value of this type can be automatically destroyed at the end of the scope
    - for types without drop ability, not destroying them manually will cause a compilation error.
  - ***key***: a type that can appear as a key in global storage
  - ***store***: the value of this type can be stored (for example, in another struct)

- Custom types that have the abilities *key* and *store* are considered to be **assets** in IOTA Move.
  - Assets are stored in global storage and can be transferred between accounts.

https://docs.iota.org/developer/getting-started/create-a-module

1. Object Basics

- The first field of the **struct** must be the id of the object with type **UID**

Struct

```
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}
```

Object

```
struct ColorObject has key {
    id: UID,
    red: u8,
    green: u8,
    blue: u8,
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

1. Object Basics - Key

- In Move the **key** ability denotes a type that can appear as a key in global storage
- Diem Move uses a **(type, address)-indexed map**
- IOTA Move uses a **map keyed** by **object IDs**.

```
use iota::object::UID;

struct ColorObject has key {
    id: UID,
```

https://docs.iota.org/developer/iota-101/objects/object-model

1. Object Basics - Create an Object

- The only way to create a new UID for a IOTA object is to call **object::new**.

object enjoyer

contract enjoyer

```
use iota::object;
// tx_context::TxContext creates an alias to the TxContext struct in the tx_context module.
use iota::tx_context::TxContext;


fun new(red: u8, green: u8, blue: u8, ctx: &mut TxContext): ColorObject {
    ColorObject {
        id: object::new(ctx),
        red,
        green,
        blue,
    }
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

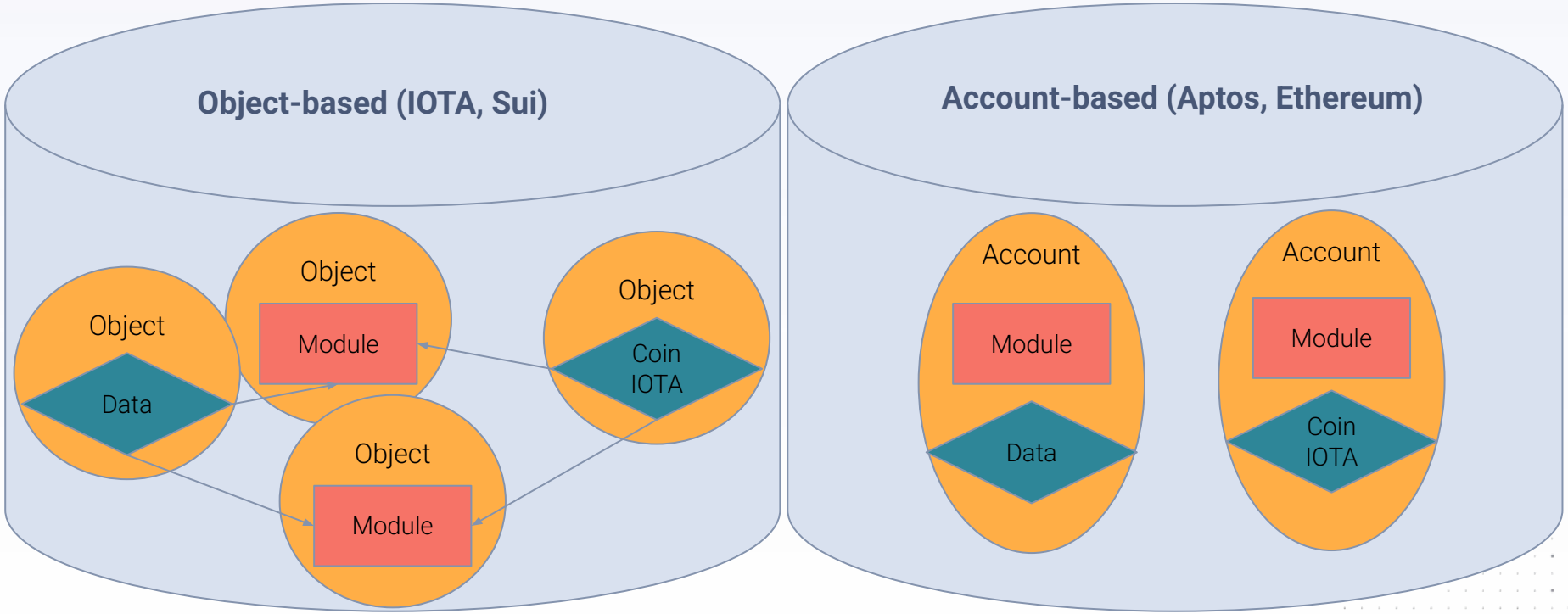
1. Object Basics - Store an Object

- The constructor puts the object value in a local variable.
- The object can then be placed in persistent global storage.

```
public entry fun create(red: u8, green: u8, blue: u8, ctx: &mut TxContext) {
    let color_object = new(red, green, blue, ctx);
    transfer::transfer(color_object, tx_context::sender(ctx))
}
```

https://docs.iota.org/developer/iota-101/objects/object-model

1. Object → Ledger → Storage



https://docs.iota.org/developer/iota-101/objects/object-model

2. Owned, Shared and Immutable Objects

- Objects in IOTA can have different types of **ownership**, with three categories:
  - **Owned mutable** object -> is owned by an address/object
  - **Shared mutable** object -> anyone can use it in a transaction
  - **Immutable** object -> an object that can't be mutated, transferred or deleted.

- In other blockchains, ***every object is shared***
  - In IOTA Move programmers have the choice to implement a particular use-case using **shared objects, owned objects, or a combination**.

- In IOTA, a transaction that touches a shared object needs to pass through the consensus mechanism. Whilst, a transaction that touches only owned objects does not need it.

https://docs.iota.org/developer/iota-101/objects/shared-owned

2. *Owned*, Shared and Immutable Objects

- **Address Owned object:** *exclusively accessible to their owner*
  - The owner is a 32-byte user address or object ID
  - Does not require consensus to be modified

```
module examples::custom_transfer {
  // Error code for trying to transfer a locked object
  const EObjectLocked: u64 = 0;

  public struct O has key {
    id: UID,
    // An `O` object can only be transferred if this field is `true`
    unlocked: bool
  }

  // Check that `O` is unlocked before transferring it
  public fun transfer_unlocked(object: O, to: address) {
     assert!(object.unlocked, EObjectLocked);
     iota::transfer::transfer(object, to)
  }
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

2. Owned, *Shared* and Immutable Objects

- **Shared object:** *anyone can read or write this object.*
  - mutable owned objects are single-writer
  - shared objects require to sequence reads and writes

```
/// Init function is often ideal place for initializing
/// a shared object as it is called only once.
fun init(ctx: &mut TxContext) {
    transfer::transfer(ShopOwnerCap {
        id: object::new(ctx)
    }, tx_context::sender(ctx));

    // Share the object to make it accessible to everyone!
    transfer::share_object(DonutShop {
        id: object::new(ctx),
        price: 1000,
        balance: balance::zero()
    })
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/shared

2. Owned, Shared and *Immutable* Objects

- Immutable objects have no owner, so anyone can use them without the need for ordering
  - packages are immutable objects
  - you can freeze an initially mutable object

```
public entry fun freeze_object(object: ColorObject) {
    transfer::freeze_object(object)
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/immutable

3. Using Objects

- IOTA Move **authentication mechanisms** ensure ***only you can use objects owned by you*** or ***shared*** in function calls.
- The object can be passed as a parameter to a function in two ways (core Move):
  - Pass by reference
    - *&ColorObject*
    - *&mut ColorObject*
  - Pass by value
    - *ColorObject*

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

3. Using Objects - Pass by Reference

- **Read-only references** (&) allow you to read data from the object
- **Mutable references** (&mut) allow you to mutate the data in the object.

```
/// Copies the values of `from_object` into `into_object`.
public entry fun copy_into(from_object: &ColorObject, into_object: &mut ColorObject) {
    into_object.red = from_object.red;
    into_object.green = from_object.green;
    into_object.blue = from_object.blue;
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

3. Using Objects - Pass by Value

- Pass objects by value into an entry function means the **object is moved out of storage**.
- Objects **cannot** be arbitrarily **dropped** and must be either consumed (e.g., transferred) or deleted

```
    public entry fun delete(object: ColorObject) {
        let ColorObject { id, red: _, green: _, blue: _ } = object;
        object::delete(id);
    }


public entry fun transfer(object: ColorObject, recipient: address) {
    transfer::transfer(object, recipient)
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

4. Object Wrapping

- In IOTA Move, you can organize data structs by putting a field of **struct** type in another
- To embed a struct type in an object struct (with a key ability), the struct type must have the **store ability**.

```
struct Wrapping has key {
    id: UID,
    obj: Wrapped,
}

struct Wrapped has key, store {
    value: u64,
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/wrapped

4. Object Wrapping

- When an object is **wrapped** into another object:
  - it **no longer exists independently** on the ledger; it becomes part of the data of the object that wraps it;
  - is no longer **findable** by its *objectID*;
  - is no longer passable as an argument in transactions procedures calls; the only access point is through the wrapping object (you need to pass this as argument).
- **Unwrapping**
  - you can then take out the wrapped object and transfer it to an address;
  - when an object is unwrapped, it becomes an independent object again;
  - **wrapped objects cannot be unwrapped unless the wrapping object is destroyed**

https://docs.iota.org/developer/iota-101/objects/object-ownership/wrapped

4. Object Wrapping

```
struct ObjectWrapper has key {
    id: UID,
    original_owner: address,
    to_swap: Object,
}
public entry fun request_swap(object: Object, service_address: address, ctx:
    let wrapper = ObjectWrapper {
        id: object::new(ctx),
        original_owner: tx_context::sender(ctx),
        to_swap: object,
    };
    transfer::transfer(wrapper, service_address);
}
 public entry fun execute_swap(wrapper1: ObjectWrapper, wrapper2: ObjectWrap
    // Unpack both wrappers, cross send them to the other owner.
    let ObjectWrapper {
        id: id1,
        original_owner: original_owner1,
        to_swap: object1,
    } = wrapper1;

    let ObjectWrapper {
        id: id2,
        original_owner: original_owner2,
        to_swap: object2,
    } = wrapper2;

    // Perform the swap.
    transfer::transfer(object1, original_owner2);
    transfer::transfer(object2, original_owner1);
}
```

https://docs.iota.org/developer/iota-101/objects/object-ownership/wrapped

5. Dynamic Fields

- IOTA Move provides **dynamic fields** with arbitrary *names*, added and removed on-the-fly (not fixed at publish), which can store heterogeneous values.

- This approach overcomes the following limitations:
  - Object's have a finite set of fields, fixed when its module is declared.
  - Objects can become very large if they wrap several other objects (high gas fees).
  - It is not possible to store a collection of objects (e.g., vector) of heterogeneous types.

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

5. Dynamic Fields - Add field

- This function takes the **Child object** *by value* and makes it a *dynamic field* of the **Parent object** with **name *b"child"***;
  - sender address owns the Parent object;
  - the Parent object owns the Child object, and can refer to it by the name *b"child"*.

```
use iota::dynamic_object_field as ofield;

public fun add_child(parent: &mut Parent, child: Child) {
    ofield::add(&mut parent.id, b"child", child);
}
```

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

5. Dynamic Fields - Access field

```
use iota::dynamic_object_field as ofield;

public fun mutate_child(child: &mut Child) {
    child.count = child.count + 1;
}

public fun mutate_child_via_parent(parent: &mut Parent) {
    mutate_child(ofield::borrow_mut(
        &mut parent.id,
        b"child",
    ));
}
```

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

5. Dynamic Fields - Remove field

```
use iota::dynamic_object_field as ofield;

public fun delete_child(parent: &mut Parent) {
    let Child { id, count: _ } = reclaim_child(parent);

    object::delete(id);
}

public fun reclaim_child(parent: &mut Parent, ctx: &mut TxContext): Child {
    ofield::remove(
        &mut parent.id,
        b"child",
    );

}
```

https://docs.iota.org/developer/iota-101/objects/dynamic-fields/

6. Transfer to Object

- *Transfer objects to an object ID* works in the **same way as an object transfer to an address** (using the same functions)

- Transfering an object to another object means establishing a form of **parent-child** authentication relationship.
  - Objects transferred to another object can be **received** by the owner of the parent object.
  - The **parent** (receiving) object **module defines the access control** for receiving a child obj.

```
// Transfers the object `b` to the address 0xADD
iota::transfer::public_transfer(b, @0xADD);

// Transfers the object `c` to the object with object ID 0x0B
iota::transfer::public_transfer(c, @0x0B);
```

https://docs.iota.org/developer/iota-101/objects/transfers/transfer-to-object

6. Transfer to Object - Receive

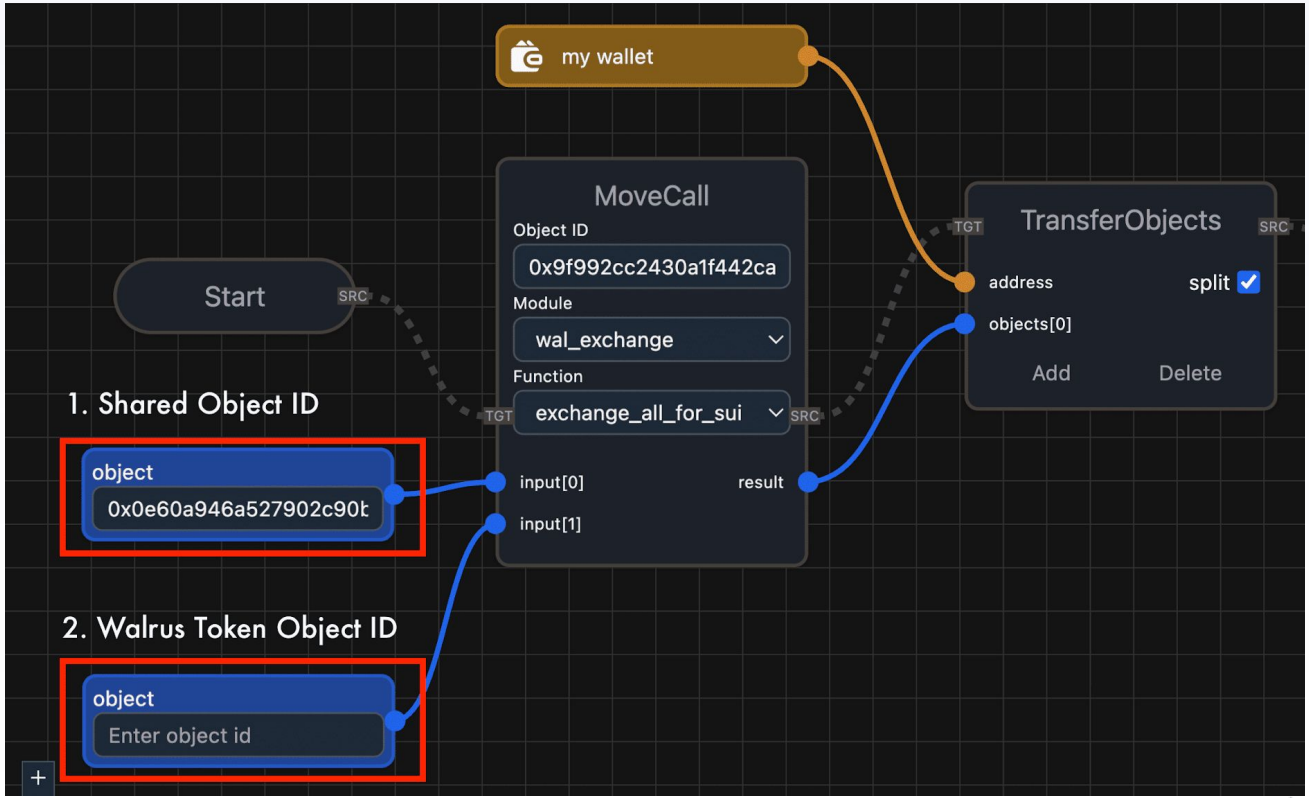
- After an object *c* has been sent to another object *p*, *p* must then receive *c* to do anything with it.
- The module of the type of *p* defines access control policies and other restrictions on *c*

```
/// This function will receive a coin sent to the `Account` object and then
/// join it to the balance for each coin type.
/// Dynamic fields are used to index the balances by their coin type.
public fun accept_payment<T>(account: &mut Account, sent: Receiving<Coin<T>>) {
    // Receive the coin that was sent to the `account` object
    // Since `Coin` is not defined in this module, and since it has the `store`
    // ability we receive the coin object using the `transfer::public_receive` function.
    let coin = transfer::public_receive(&mut account.id, sent);
    let account_balance_type = AccountBalance<T>{};
    let account_uid = &mut account.id;

    // Check if a balance of that coin type already exists.
    // If it does then merge the coin we just received into it,
    // otherwise create new balance.
    if (df::exists_(account_uid, account_balance_type)) {
        let balance: &mut Coin<T> = df::borrow_mut(account_uid, account_balance_type);
        coin::join(balance, coin);
    } else {
        df::add(account_uid, account_balance_type, coin);
    }
}
```

https://docs.iota.org/developer

7. Programmable Transaction Blocks



https://x.com/zktx_io/status/1883158121100816745

7. Programmable Transaction Blocks

- The **inputs value** of a PTB is value is a vector of arguments, either *objects* or *pure values*
- The **commands value** of a PTB is a vector of commands using *inputs* or *results* to execute code
    - *TransferObjects* sends (one or more) objects to a specified address
    - *SplitCoins* splits off (one or more) coins from a single coin. It can be any iota::coin::Coin<_>
    - *MergeCoins* merges (one or more) coins into a single coin
    - *MakeMoveVec* creates a vector of Move values
    - ***MoveCall*** invokes either an *entry* or a *public* Move function in a published package.
    - *Publish* creates a new package and calls the init function of each module in the package.
    - *Upgrade* upgrades an existing package.
- The **result values** is a vector of values that can be produced by each command; the type of the value can be any arbitrary Move type, not limited to objects or pure values.
- A PTB can perform up to 1,024 unique operations in a single execution.

https://docs.iota.org/developer/iota-101/transactions/ptb/prog-txn-blocks

7. Programmable Transaction Blocks
public **vs** *entry* **functions**

- The **public** modifier allows a function to be *called from a PTB* and also *from other modules*
  - NO restrictions on parameters
- The ***entry*** modifier allows a function to be called directly from a PTB as a module "entrypoint".
  - entry functions **parameters must be inputs** to the PTB (not results of previous command)
  - only allowed to return types that have drop
- Use the *entry* modifier when:
  - You want strong guarantees that your function is not being combined with third-party module functions (e.g., swap protocol that does not want a flash loan)
  - *public* function signatures must be maintained by upgrades (entry function not).
  - It is also possible to create a *public entry* function, can be called by other modules

https://docs.iota.org/developer/iota-101/move-overview/entry-functions

8. Hot Potato Pattern

1. This pattern requires that **function B** must be called ***immediately after*** **function A**, when **function A** returns a **hot potato** and **function B** consumes it.
2. Flash loan:
   a. create a **`Receipt` struct** that
      - cannot be discarded because it does not have `drop`,
      - cannot be put in persistent storage because it does not have `key`,
      - cannot be transferred or wrapped because it does not have `store`.
   b. Have a **`loan`** function that requests a loan of `amount` from `lender` and returns the **`Receipt`**
   c. the only way to get rid of it is to call **`repay`** at some point forcing to pay back the debt.

https://docs.iota.org/developer/iota-101/move-overview/patterns/hot-potato

9. One-Time Witness (OTW)

- Special type guaranteed to have **at most one instance**: useful for limiting certain actions to only happen once (e.g., creating a coin). The only instance is passed to its module's init function when its package is published. In Move, a type is considered a OTW if:
  - Its name is the **same as its module's names**, all **uppercased**.
  - It has **ONLY** the **drop ability**
  - It has **no fields**, or a single bool field.

```
module examples::mycoin {

    /// Name matches the module name
    struct MYCOIN has drop {}

    /// The instance is received as the first argument
    fun init(witness: MYCOIN, ctx: &mut TxContext) {
        /* ... */
    }
}
```

https://docs.iota.org/developer/iota-101/move-overview/one-time-witness

10. Generics

- Generics are **abstract stand-ins *for concrete types*** or other properties.

```
struct Box<T> {
    value: T
}
```

- **Conditions** to enforce that the type passed into the generic *must have certain abilities*.

```
// T must be copyable and droppable
struct Box<T: store + drop> has key, store {
    value: T
}
```

- Using generics in functions

```
public fun create_box<T>(value: T): Box<T> {
        Box<T> { value }
    }
```

```
// value will be of type storage::Box<bool>
    let bool_box = storage::create_box<bool>(true);
// value will be of the type storage::Box<u64>
    let u64_box = storage::create_box<u64>(1000000);
```

https://intro.sui-book.com/unit-three/lessons/2_intro_to_generics.html#intro-to-generics

11. Capability Pattern

- This pattern enables the **authorization of specific actions with an object**.
  - e.g., the UpgradeCap is used to authorize the upgrading of packages.
  - e.g. the TreasuryCap grants the authority to manage a Coin treasury functions.

```
/// Type representing the capability to create new `Item`s.
public struct AdminCap has key { id: UID }

/// Custom NFT-like type representing an item.
public struct Item has key, store { id: UID, name: String }

/// Module initializer, called once during the module's deployment.
/// This function creates a single instance of `AdminCap` and assigns it to the publisher.
fun init(ctx: &mut TxContext) {
    transfer::transfer(AdminCap {
        id: object::new(ctx)
    }, tx_context::sender(ctx))
}

/// Function to create a new `Item`. It requires `AdminCap` to authorize the action.
public fun create_item(_: &AdminCap, name: String, ctx: &mut TxContext): Item {
    let item = Item {
        id: object::new(ctx),
        name,
    };
    item
```

https://docs.iota.org/developer/iota-101/move-overview/patterns/capabilities

Interacting with a IOTA Move Module

Install IOTA

https://docs.iota.org/developer/getting-started/install-iota

Explorer:
https://explorer.iota.org/?network=testnet

Move Capture the Flag

https://docs.iota.org/developer/iota-move-ctf/introduction

Provide transaction hash digests where you got each flag for **challenges 1 to 7 by the end of the month**

Bonus for the ones that get the flag for challenge 8

Receive funds

```
$ iota client faucet
$ iota client gas
```

0. Write a IOTA Move Package

```
module my_first_package::my_module {

    // Imports
    use iota::object::{Self, UID};
    use iota::transfer;
    use iota::tx_context::{Self, TxContext};

    // Struct definitions
    struct Sword has key, store {
        id: UID,
        magic: u64,
        strength: u64,
    }

    struct Forge has key, store {
        id: UID,
        swords_created: u64,
    }

    // Module initializer to be executed when this module is published
    fun init(ctx: &mut TxContext) {
        let admin = Forge {
            id: object::new(ctx),
            swords_created: 0,
        };
        // Transfer the forge object to the module/package publisher
        transfer::public_transfer(admin, tx_context::sender(ctx));
    }

    // Accessors required to read the struct attributes
    public fun magic(self: &Sword): u64 {
        self.magic
    }

    public fun strength(self: &Sword): u64 {
        self.strength
    }

    public fun swords_created(self: &Forge): u64 {
        self.swords_created
    }

    // Public/entry functions

    // Private functions
}
```

https://docs.iota.org/developer/getting-started/create-a-module

1. Build and Publish a IOTA Move Package

```
$ iota move build
$ iota move test
$
$
$ iota client publish --gas-budget 5000000
```

```
#[test]
public fun test_sword() {
    // Create a dummy TxContext for testing.
    let mut ctx = tx_context::dummy();

    // Create a sword.
    let sword = Sword {
        id: object::new(&mut ctx),
        magic: 42,
        strength: 7,
    };

    // Check if accessor functions return correct values.
    assert!(magic(&sword) == 42 && strength(&sword) == 7, 1);
}
```

https://docs.iota.org/developer/getting-started/build-test
https://docs.iota.org/developer/getting-started/publish

2. Interact with a Package - PTB

- You can construct more advanced blocks of transactions using the `$ iota client ptb` command.

- In general, transactions on IOTA are composed of:
  - a number of **commands**
  - that execute on **inputs**
  - to define some **results**

https://docs.iota.org/references/cli/ptb

3. Programmable Transaction Blocks

```
$ iota client ptb \
--move-call 0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg::func
"<0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg1::TYPE1,0xd95b451
0206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg2::TYPE2>"
@0x0b72fb4d8106699c773bf58fd0a49ffe3a08bdd58f245946d160ed5463f7ba47 99 true \
--assign result_variable \
--move-call iota::tx_context::sender \
--assign sender \
--transfer-objects "[result_variable.2]" sender \
--move-call 0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg::func2
"<0xd95b4510206e13fbe9413bc61183ac3b8375c8971adc54c81eeb9c96d61b5ff1::pkg1::TYPE1"
@0x0b72fb4d8106699c773bf58fd0a49ffe3a08bdd58f245946d160ed5463f7ba47 result_variable.0 \
--gas-budget 50000000
```

https://docs.iota.org/references/cli/ptb

4. Binary Canonical Serialization (BCS)

- BCS is a **serialization format** developed in the context of the Diem blockchain
  - now extensively used in most of the blockchains based on Move (IOTA, Sui, Aptos, 0L).
- BCS is *not only used in the Move VM*, but also used in **transaction and event coding**.

```
var { bcs, fromHEX } = require('@mysten/bcs');
const Calzone = bcs.struct('Calzone', {
    flour: bcs.u16(),
    tomato_sauce: bcs.u16(),
    cheese: bcs.u16(),
});
const hex = "0a00030062027201120 0c800b4000000"
const calzone = Calzone.parse(fromHEX(hex));
```

https://docs.iota.org/references/ts-sdk/bcs
https://github.com/zefchain/bcs
https://www.npmjs.com/package/@mysten/bcs

What's left?

- Collections
- Events
- Package upgrades
- Proper Testing
- Clock and Random objects
- …

- https://docs.iota.org/developer/iota-101/move-overview/
- https://docs.iota.org/references/cli/client
- https://intro.sui-book.com/unit-one/lessons/1_set_up_environment.html

Thank you!

Mirko Zichichi

Engineering Team Lead, IOTA Foundation
mirko.zichichi@iota.org