Move

the Language for Secure Next Gen Smart Contracts

7th Scientific School on Blockchain & DLTs

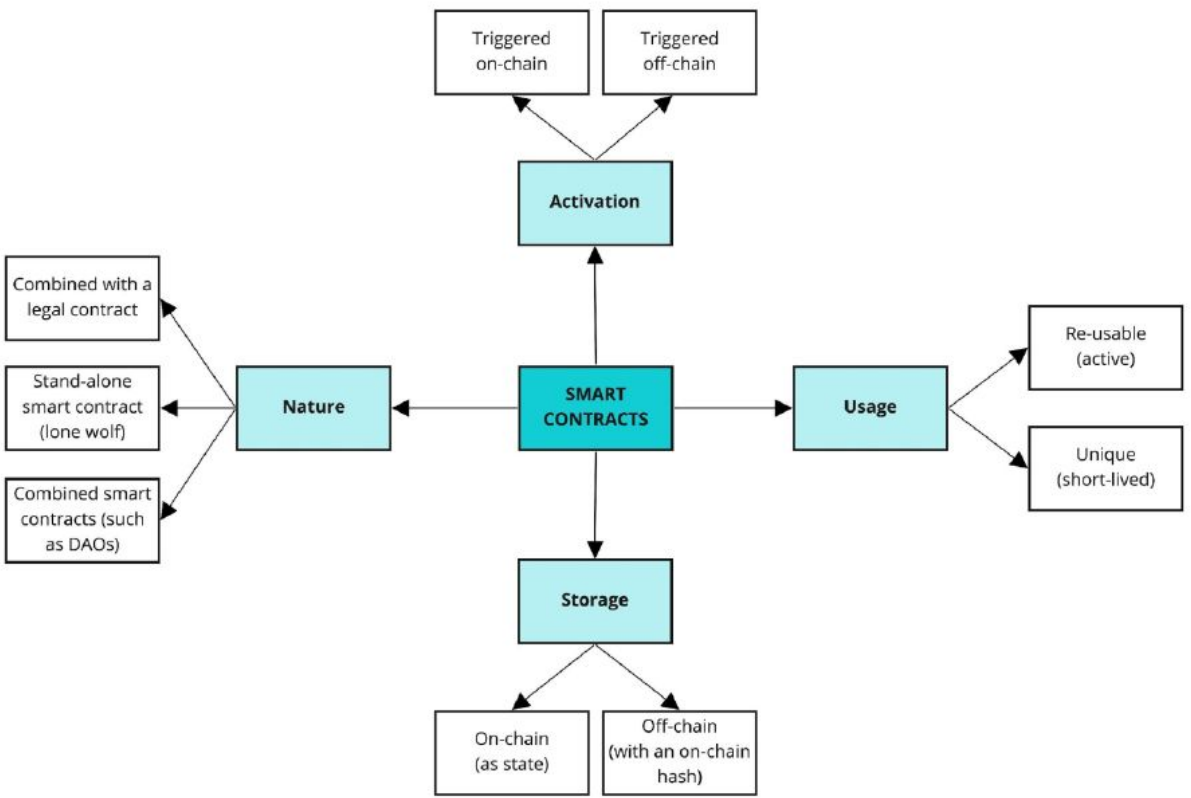
Mirko Zichichi
mirko.zichichi@iota.org
IOTA

```
/// 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);
    }
}
```

Smart Contract

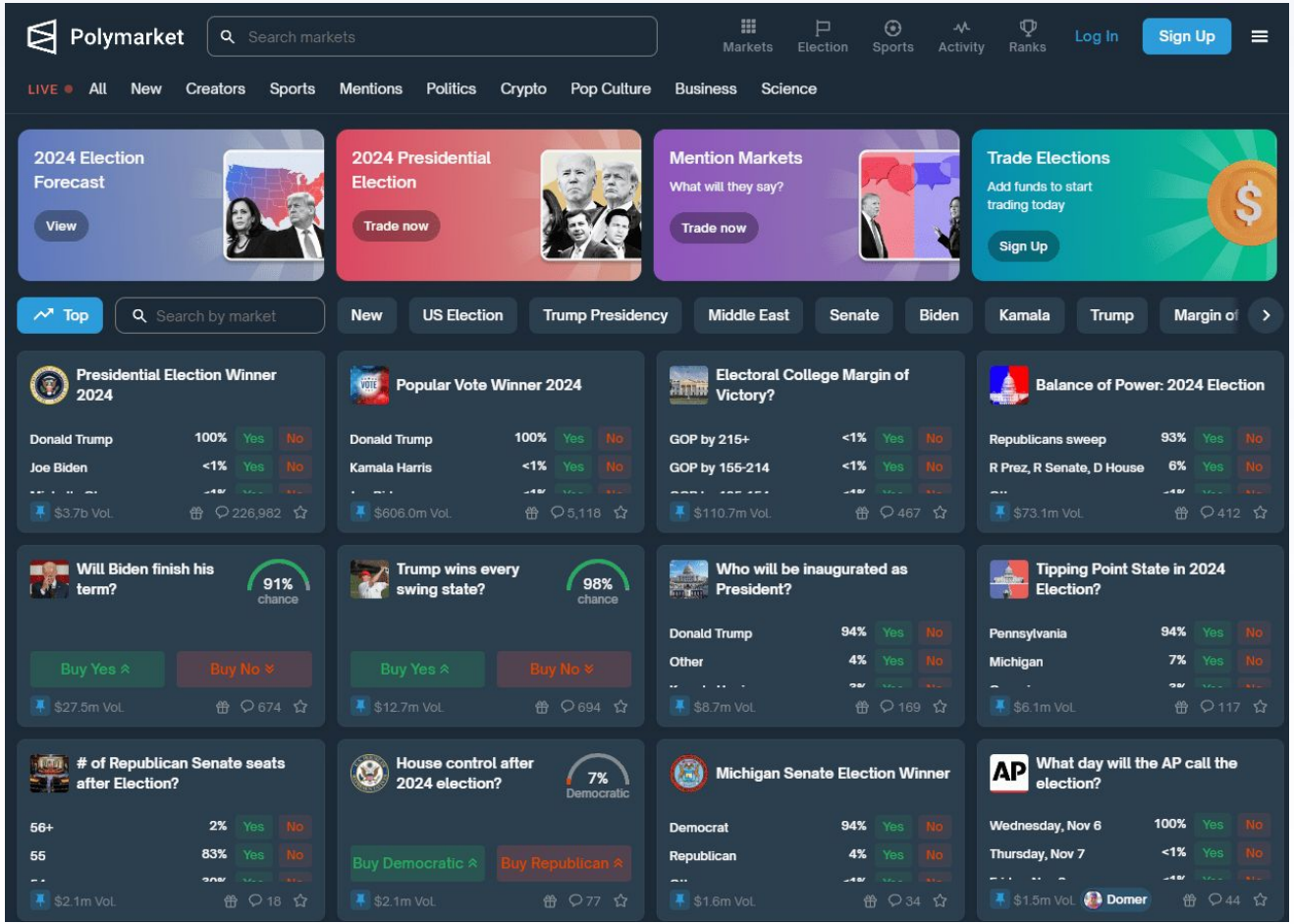
- "computerized transaction protocol that executes the terms of a contract."
- "The general objectives of smart contract design:
  - are to satisfy common **contractual conditions** (such as payment terms, liens, confidentiality, and even enforcement),
  - **minimize exceptions** both malicious and accidental,
  - and **minimize** the need for **trusted intermediaries**."
- The word **"smart"** comes from the Latin **"intelligere,"** which means *"to choose between."*
  - Smart contracts automate the choice according to pre-defined conditions

Nick Szabo, "Smart Contracts"
https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (1994).



Title: An overview of blockchain smart contracts
© Thibault Schrepel (2021)

Thibault Schrepel, "Smart Contracts" Smart Contracts and the Digital Single Market Through the Lens of a "Law + Technology" Approach, Study for the European Commission, https://digitalstrategy.ec.europa.eu/en/library/smart-contracts-and-digital-single-market-through-lens-law-plustechnology-approach (2021).



https://www.nftculture.com/nft-news/polymarket-and-prediction-markets-the-role-of-betting-in-election-forecasting/

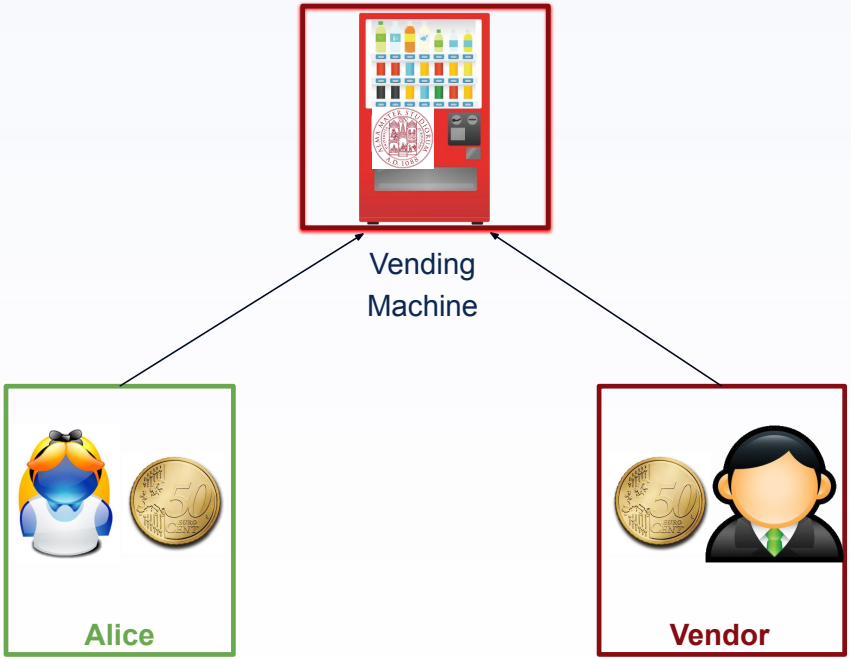
Code as Law + Law as Code

	Law	Code	Law + Code
Observation	**Law is Code:** Law is a "system of rules (law "programs" society)	**Code is Law:** Code regulates like the law ("lex informatica")	**Law needs Code:** Law and code regulate better together
Method	**Code of Law:** Codification of legal rules and standards	**Law of Code:** What the code says equals the law	**Code as Law:** Code embodies the law-of-the-land **Law as Code:** Translate law into a machine-consumable version

Thibault Schrepel, "Smart Contracts" Smart Contracts and the Digital Single Market Through the Lens of a "Law + Technology" Approach, Study for the European Commission, https://digitalstrategy.ec.europa.eu/en/library/smart-contracts-and-digital-single-market-through-lens-law-plustechnology-approach (2021).

Vending machine -> ancestor of Smart Contracts

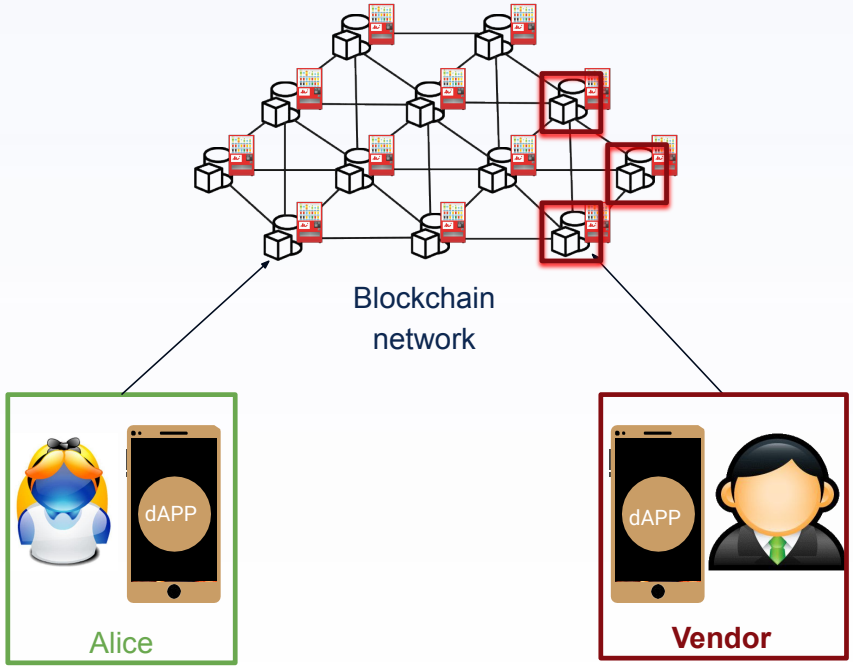
"anybody with coins can participate in an exchange with the vendor"



Nick Szabo, "The idea Smart Contracts"
https://nakamotoinstitute.org/library/the-idea-of-smart-contracts/ (1997).

Vending machine -> ancestor of Smart Contracts

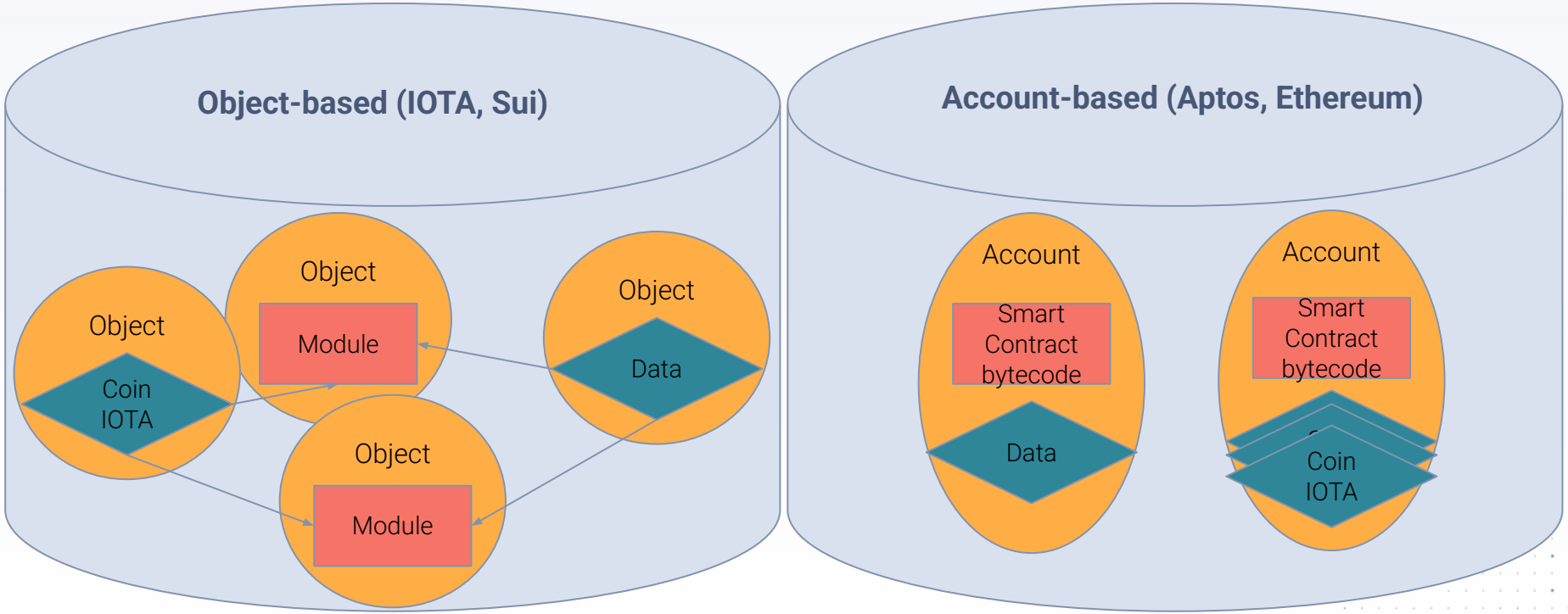
"anybody with coins can participate in an exchange with the vendor”



Nick Szabo, "The idea Smart Contracts"
https://nakamotoinstitute.org/library/the-idea-of-smart-contracts/ (1997).


Move

1. Object → Ledger → Storage



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

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

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

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 - 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

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 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

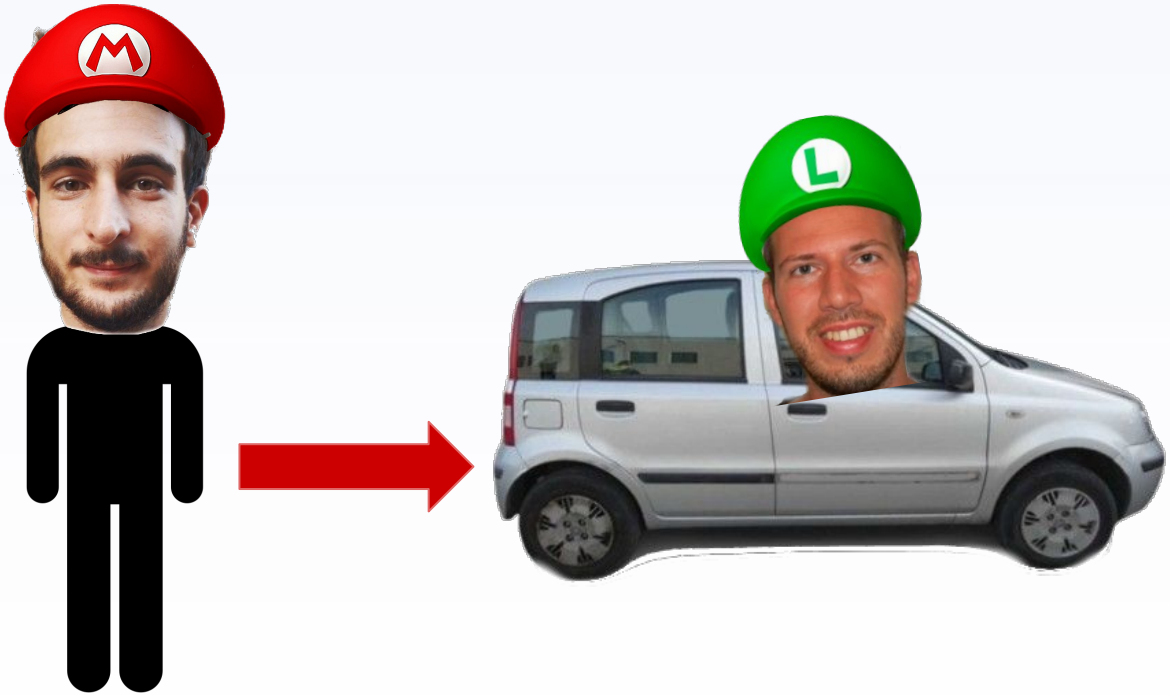
```
    public entry fun delete(object: ColorObject) {
        let ColorObject { id, red: _, green: _, blue: _ } = object;
        object::delete(id);
    }


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

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

Pass a value to a function *by-value*

the Object

Pass a value to a function *by-value*



Pass a value to a function *by-value*

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;
}
```

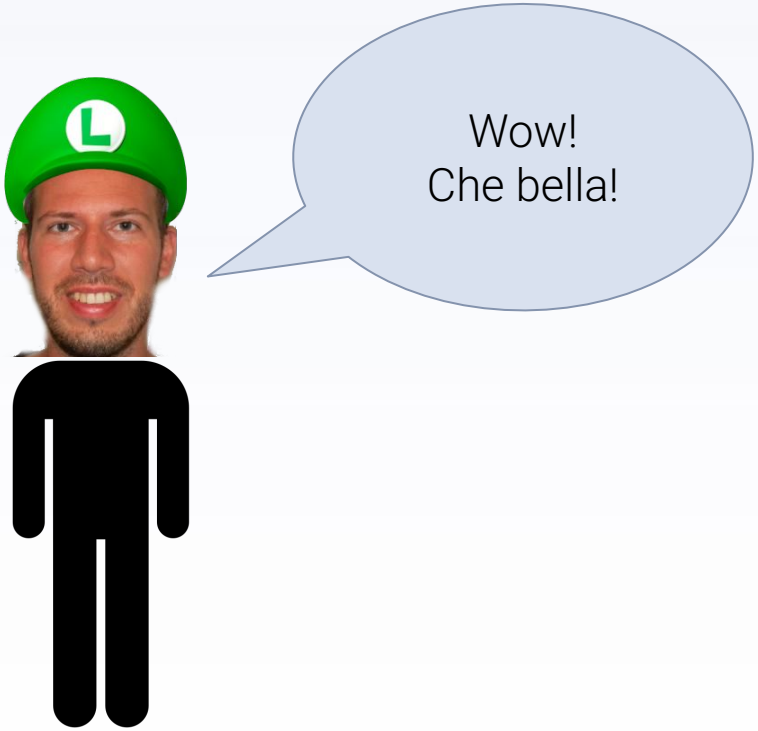
https://docs.iota.org/developer/iota-101/objects/object-ownership/address-owned

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

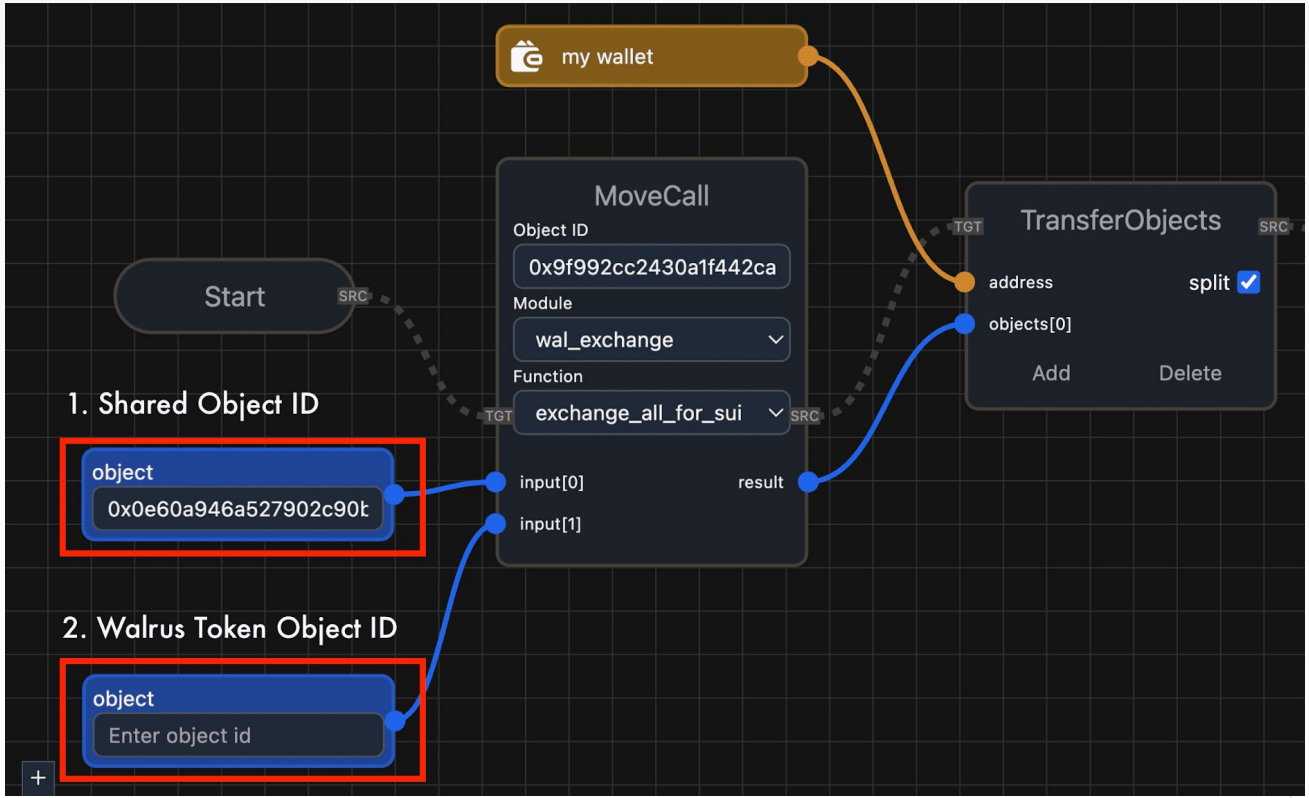
“Borrow” a value with *mutable ref (&mut)*

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

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



4. Programmable Transaction Blocks

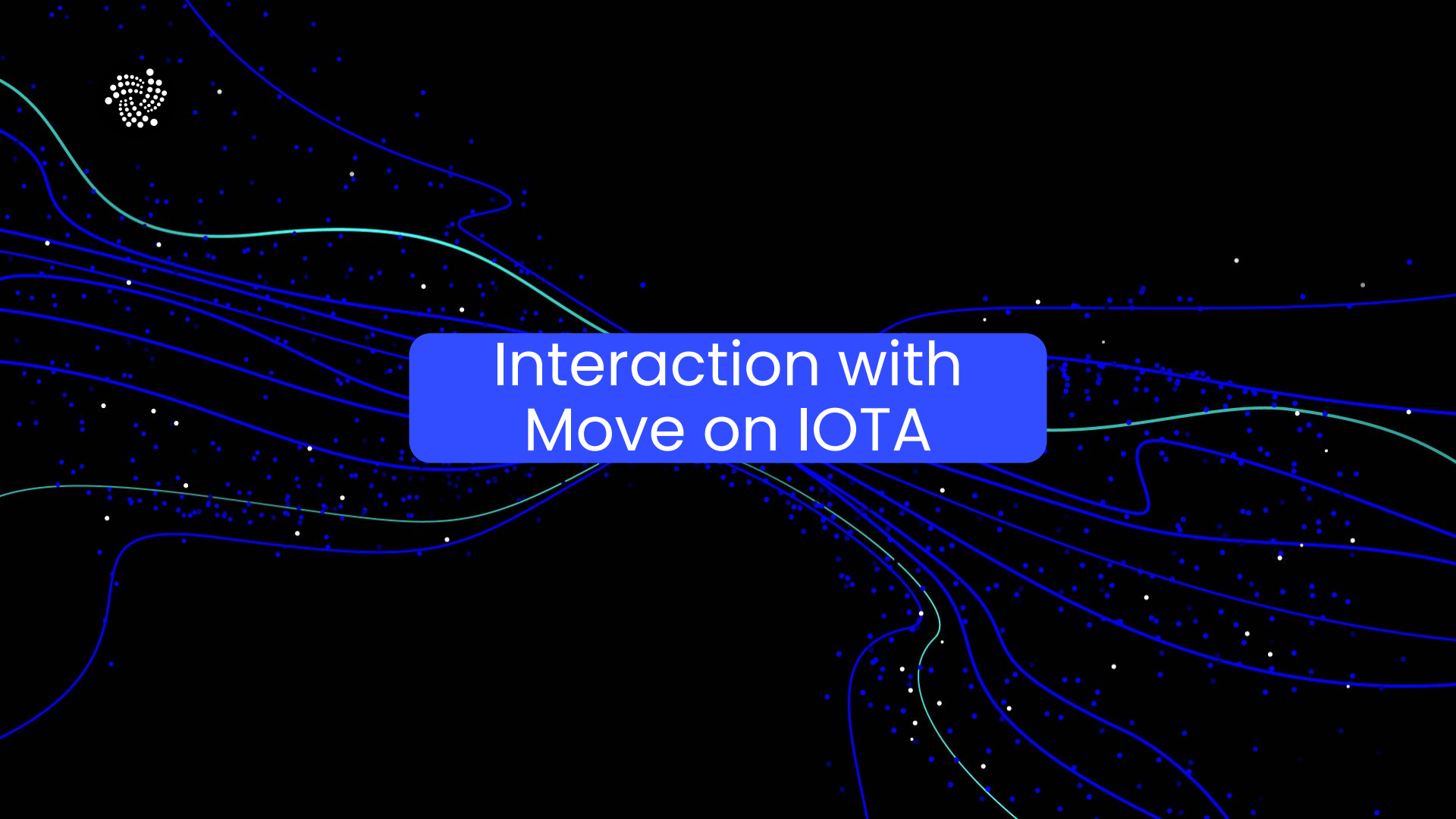


https://x.com/zktx_io/status/1883158121100816745

4. 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


Interaction with
Move on IOTA

Move Capture the Flag

https://github.com/miker83z/move-ctf-2026

Provide transaction hash digests where you got each flag
for **challenges 1 to 4 by the end of the school**

Challenges 1 to 3 are mandatory
Challenge 4 is strongly recommended
Challenge 5 is a bonus

0. Interacting with IOTA devnet using the CLI

From an empty terminal to Programmable Transaction Blocks (PTBs)

The **`iota`** CLI is the single tool you'll use to build, publish, call, and inspect Move packages on a network.

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

1. From zero to a funded account

1. Create your unique key
2. Point CLI at the Devnet RPC
3. Request and verify test coins

💡 **UNIT CONVERSION**

Amounts in CLI are shown in **NANOs**.

`1 IOTA = 1,000,000,000 NANOs`

```
$ iota client  # first run: creates your key (Ed25519)
$ iota client active-address  # show your address
$ iota client switch --env devnet
$ iota client faucet  # request test IOTA (10 per request)
$ iota client gas  # confirm the coins arrived
```

IOTA

2. Call a function and inspect objects

Invoke an entry function, then look at what you own

```
$ iota client call \
  --package $PKG --module checkpoint --function get_flag \
  --gas-budget 100000000
$ iota client objects  # objects you own (your new Flag appears here)
$ iota client object <OBJECT_ID>  # inspect a single object
```

💡 **USEFUL FLAGS**

`--args <values>`, `--type-args <T>` (for generics), `--json` (machine-readable output).

IOTA

3. Working with coins

Coins are objects; you split and merge them directly from the CLI

Used in the "mint" challenge — make an exact amount to satisfy function requirements.

```
$ iota client split-coin --coin-id <COIN> --amounts 200   # split 200 NANOs

$ iota client merge-coin --primary-coin <C1> --coin-to-merge <C2> --gas-budget
100000000
```

💡 **PRO TIP**
Splitting creates a new object ID for the result. Use `iota client objects` to find it.

IOTA

4. Programmable Transaction Blocks (PTBs)

A PTB chains several commands into **one atomic transaction**, using **--assign** to flow results between steps.

```
$ iota client ptb \
  --move-call $PKG::ingredient_heist::open_order --assign o0 \
  --move-call $PKG::ingredient_heist::set_flour o0 1234u16 --assign o1
  \
  --move-call $PKG::ingredient_heist::set_water o1 567u16 --assign o2
  \
  --move-call $PKG::ingredient_heist::set_yeast o2 9810u16 --assign o3
  \
  --move-call $PKG::ingredient_heist::set_salt o3 1112u16 --assign o4
  \
  --move-call $PKG::ingredient_heist::get_flag o4 \
  --gas-budget 100000000
```

PTB VOCABULARY

--assign o0: Names previous result (o0, o1...); reuse name as later argument.

gas: Transaction's gas coin; **coins.0** indexes first element.

@0x...: Address literal; **1234u16**: Type suffix for integers.

OTHER BUILDERS:

`--split-coins, --merge-coins, --make-move-vec, --transfer-objects`

Essential for values that cannot survive between transactions (e.g., "hot potatoes" that must be consumed immediately).

IOTA

4. Programmable Transaction Blocks (PTBs)

Another working PTB example.

```
$ iota client ptb \
  --move-call 0xd95b4510...5ff1::pkg::func "<0xd95b4510...5ff1::pkg1::TYPE1,0xd95b4510...5ff1::pkg2::TYPE2>"
  @0x0b72fb...ba47 99 true \
  --assign result_variable \
  --move-call iota::tx_context::sender \
  --assign sender \
  --transfer-objects "[result_variable.2]" sender \
  --move-call 0xd95b4510...5ff1::pkg::func2 "<0xd95b4510...5ff1::pkg1::TYPE1" @0x0b72fb...ba47
  result_variable.0 \
  --gas-budget 50000000
```

💡 **PRO TIP**

In the first move call it is used such a pattern to pass concrete instances of generic types as input to the function:
"<0xd95[...]5ff1::pkg1::TYPE1,0xd95[...]5ff1::pkg2::TYPE2>"

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

5. Flags you'll reuse across the challenges

A compact reference for advanced CLI operations

GAS MANAGEMENT

`--gas-budget <NANOs>` (Required for all txs)

`--gas <coinId>` (Specify object for payment)

`--gas-price` (Override default price)

OUTPUT & VISIBILITY

`--json` (Format for script processing)

`--display effects,events,object_changes`
(Verbose summaries of transaction results)

SIMULATION

`--dry-run` (Pre-flight local validation)

`--dev-inspect` (Execute without state change)

OFFLINE / ACCOUNT ABSTRACTION

`--serialize-unsigned-transaction`

`--serialize-signed-transaction`

`--auth-call-args <signature>`

Powers Account Abstraction (AA) logic

IOTA

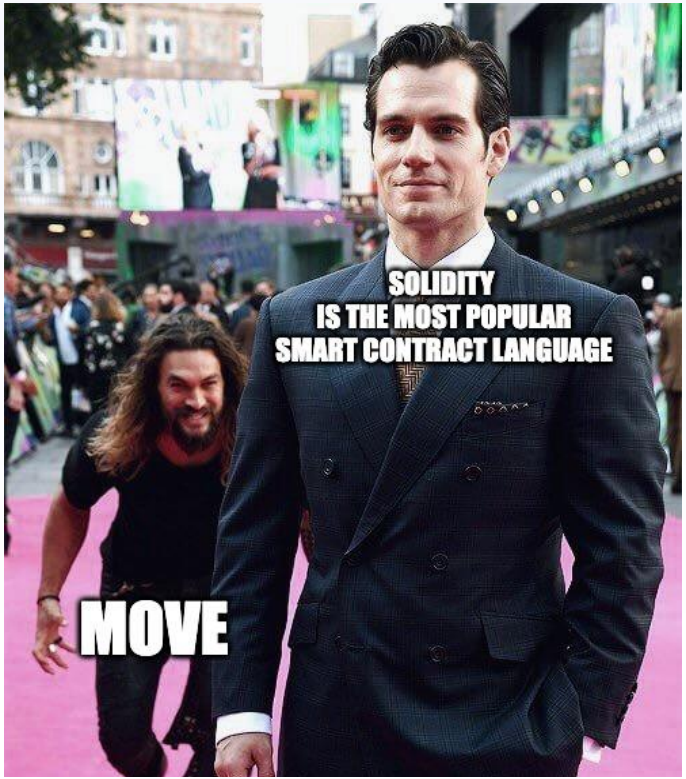
the Language for Secure Next Gen Smart Contracts

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
→ Ethereum Virtual Machine/Solidity



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
Resource

Resource

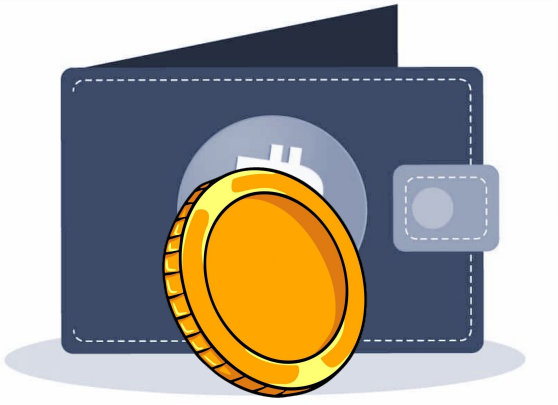
It provides the possibility of defining customized resource types with a semantics inspired by **linear logic**:

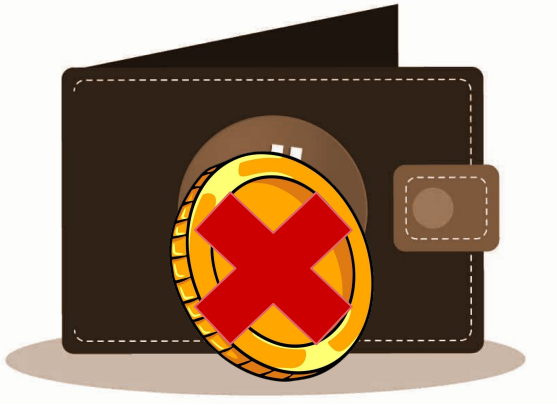
Move

Resource

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



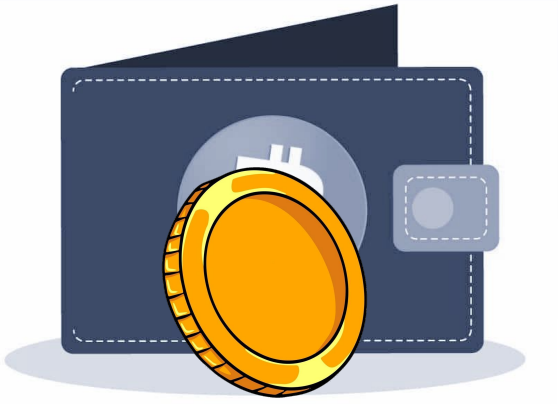


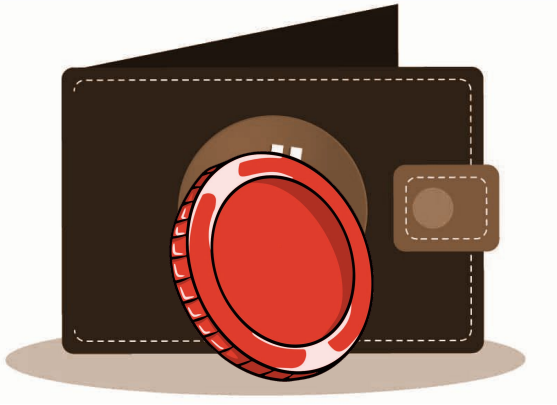
Move

Resource

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



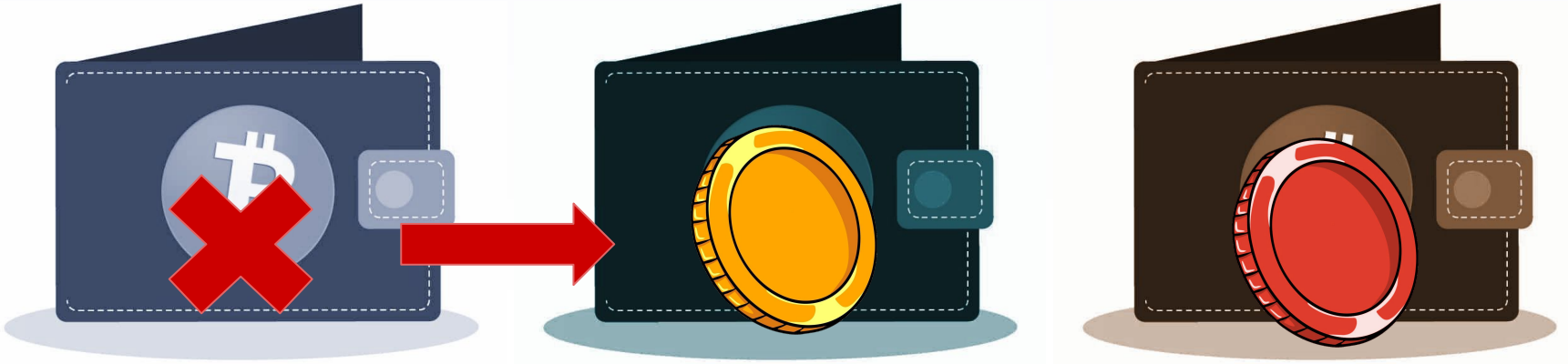


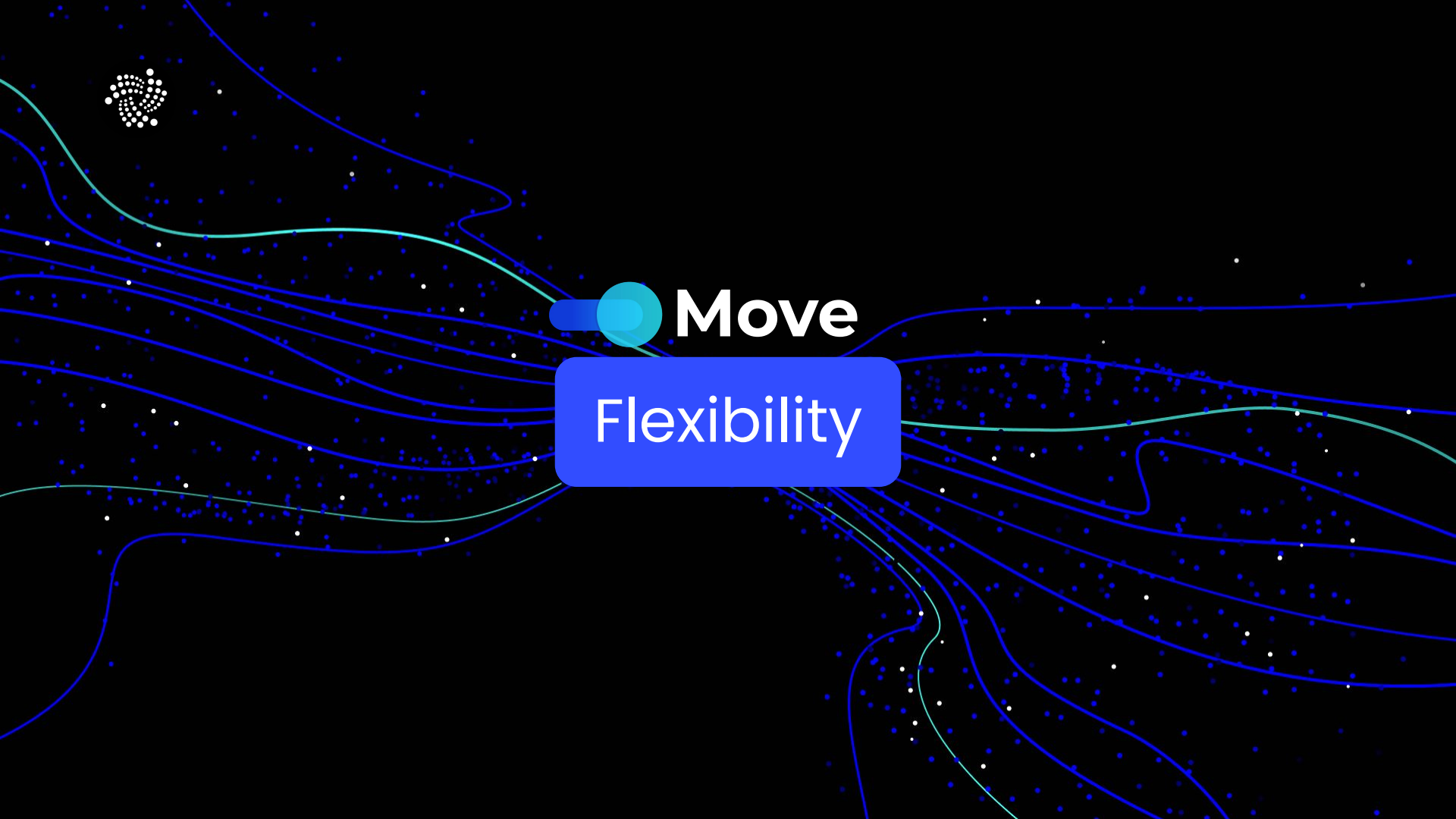
Move

Resource

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
Flexibility

Move Virtual Machine

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

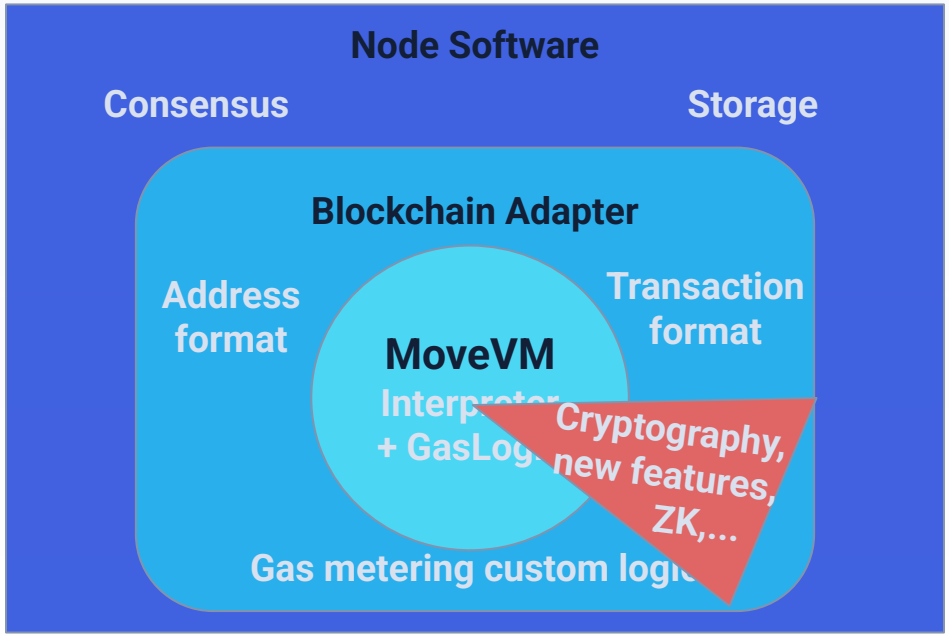
- Core VM is **easily extensible** with:
  - Cryptography, signature schemes, ZKP verifiers
  - Blockchain specific features (mana generation, system transactions, account concept, etc.)

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

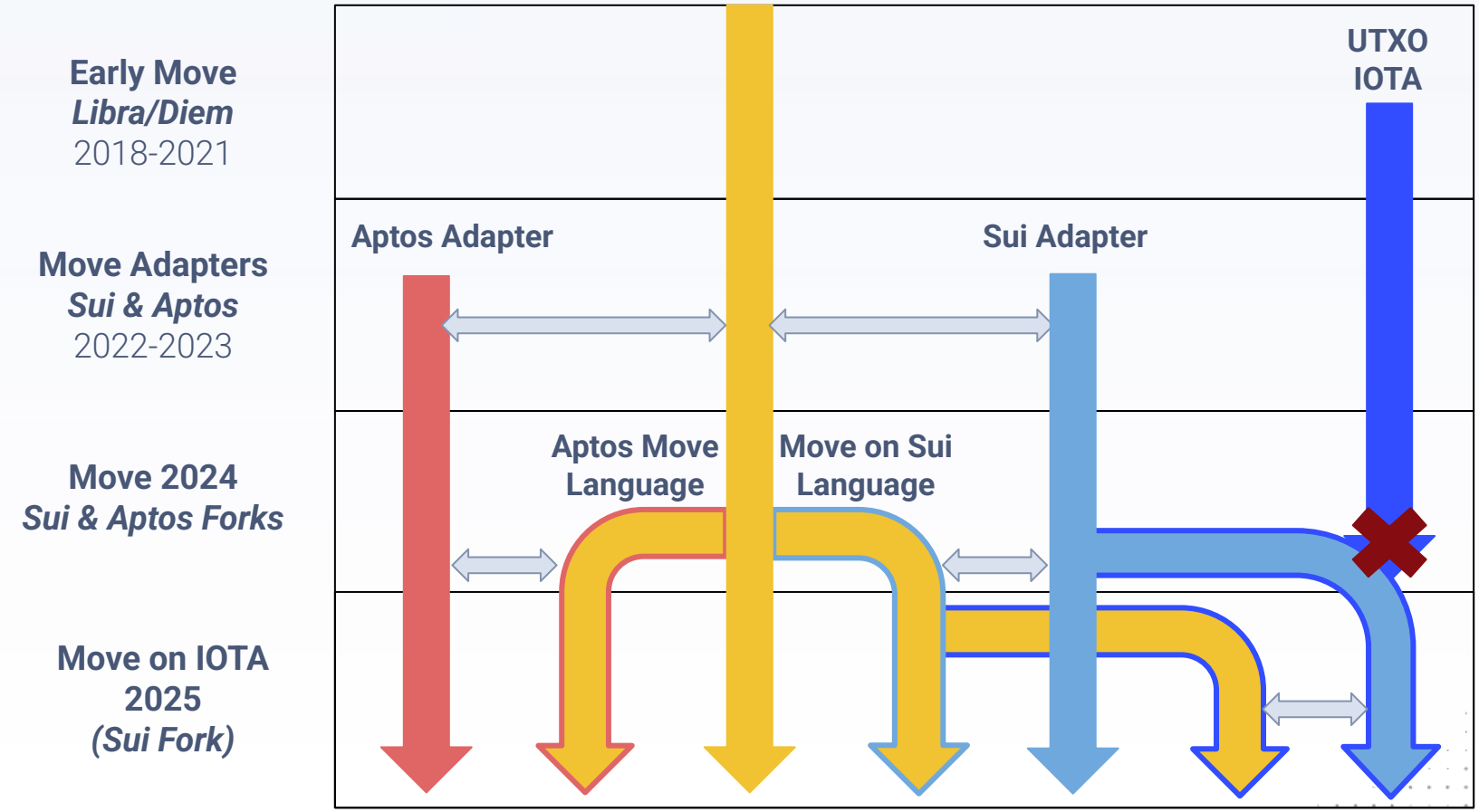
Move on Account vs Object Ledger

- **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 Modularity

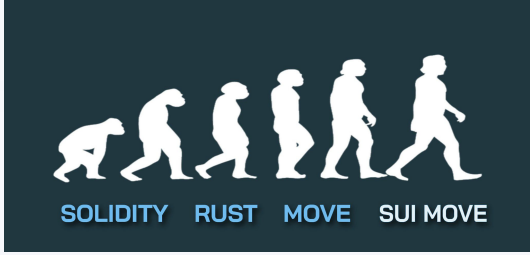


History of Move Language




Move
Security

Move



- 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).

Move

- **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

Move

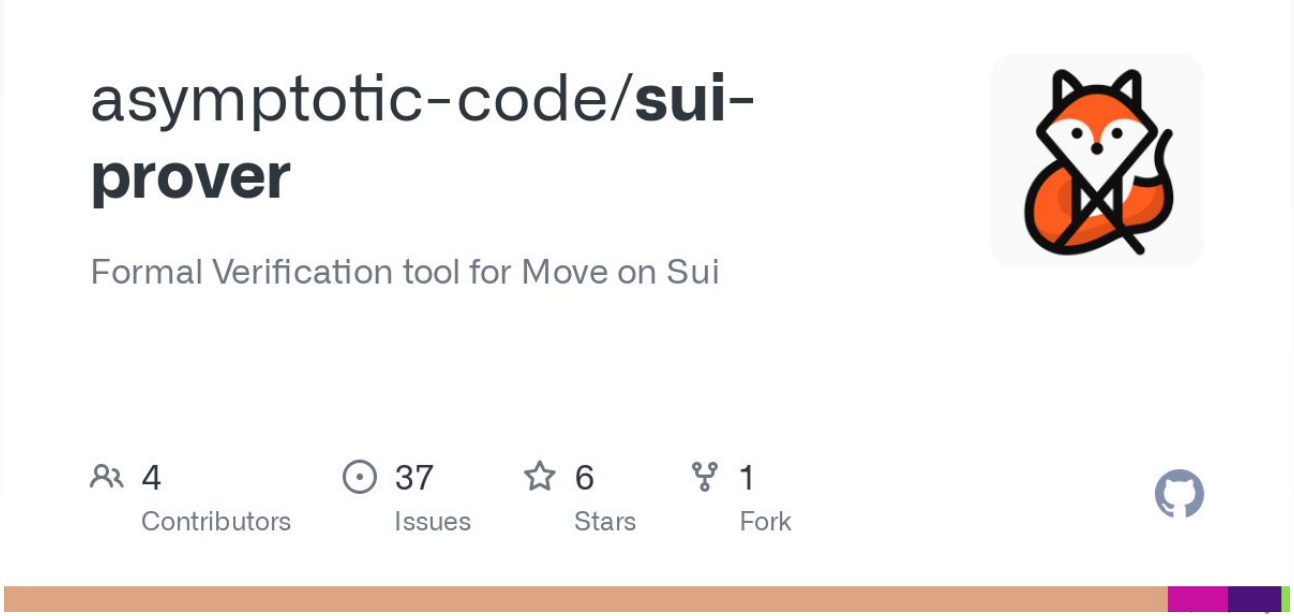
Double check:

- 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

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

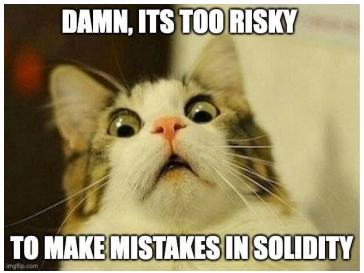


Move

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
  );
}
```

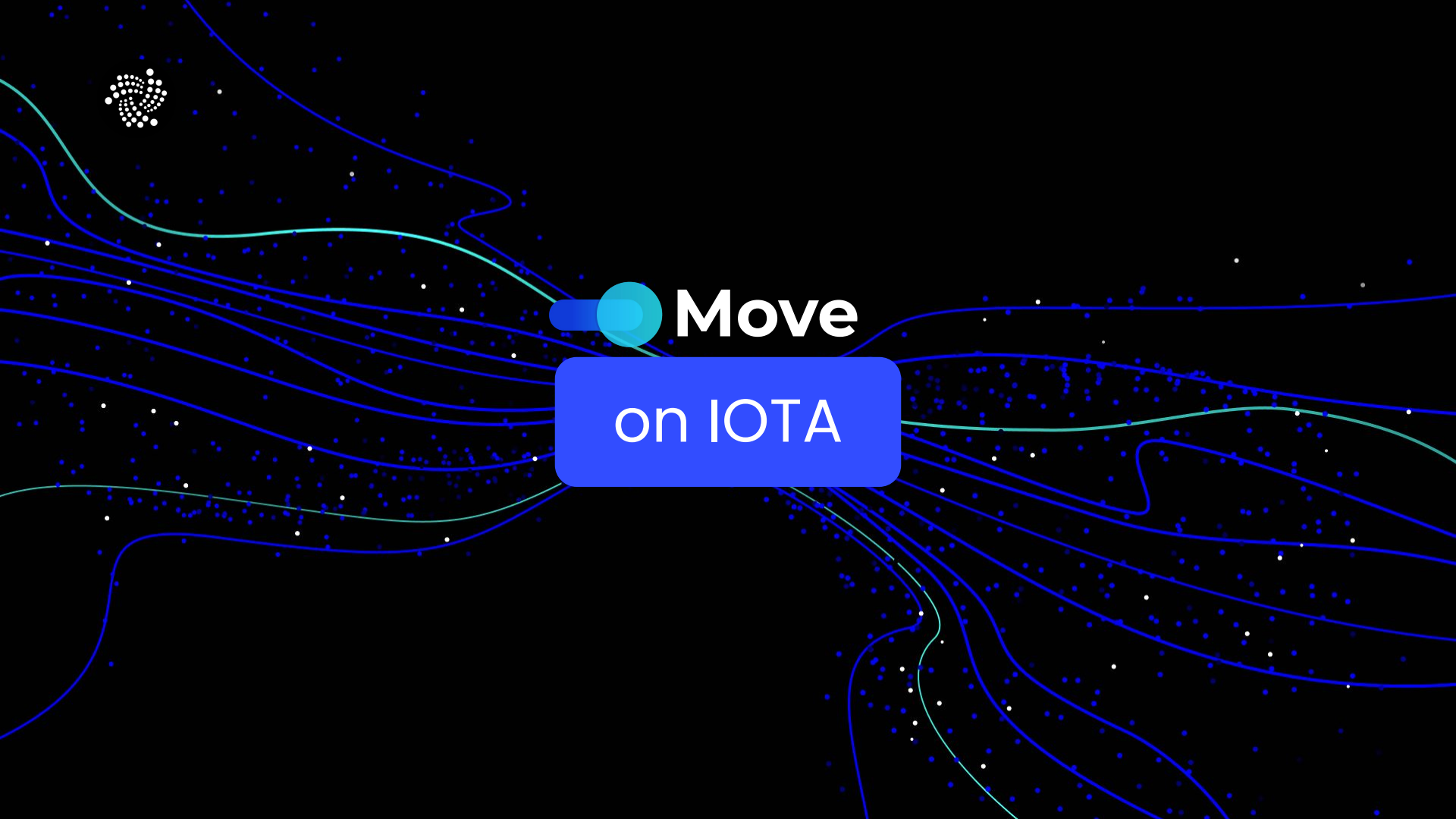
Blackshear, Sam, et al. "Resources: A safe language abstraction for money." arXiv preprint https://arxiv.org/abs/2004.05106 (2020).

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).


Move
on IOTA

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/

5. 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

5. 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

5. 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

6. 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/

6. 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/

6. 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/

6. 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/

7. 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

7. 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

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);
```

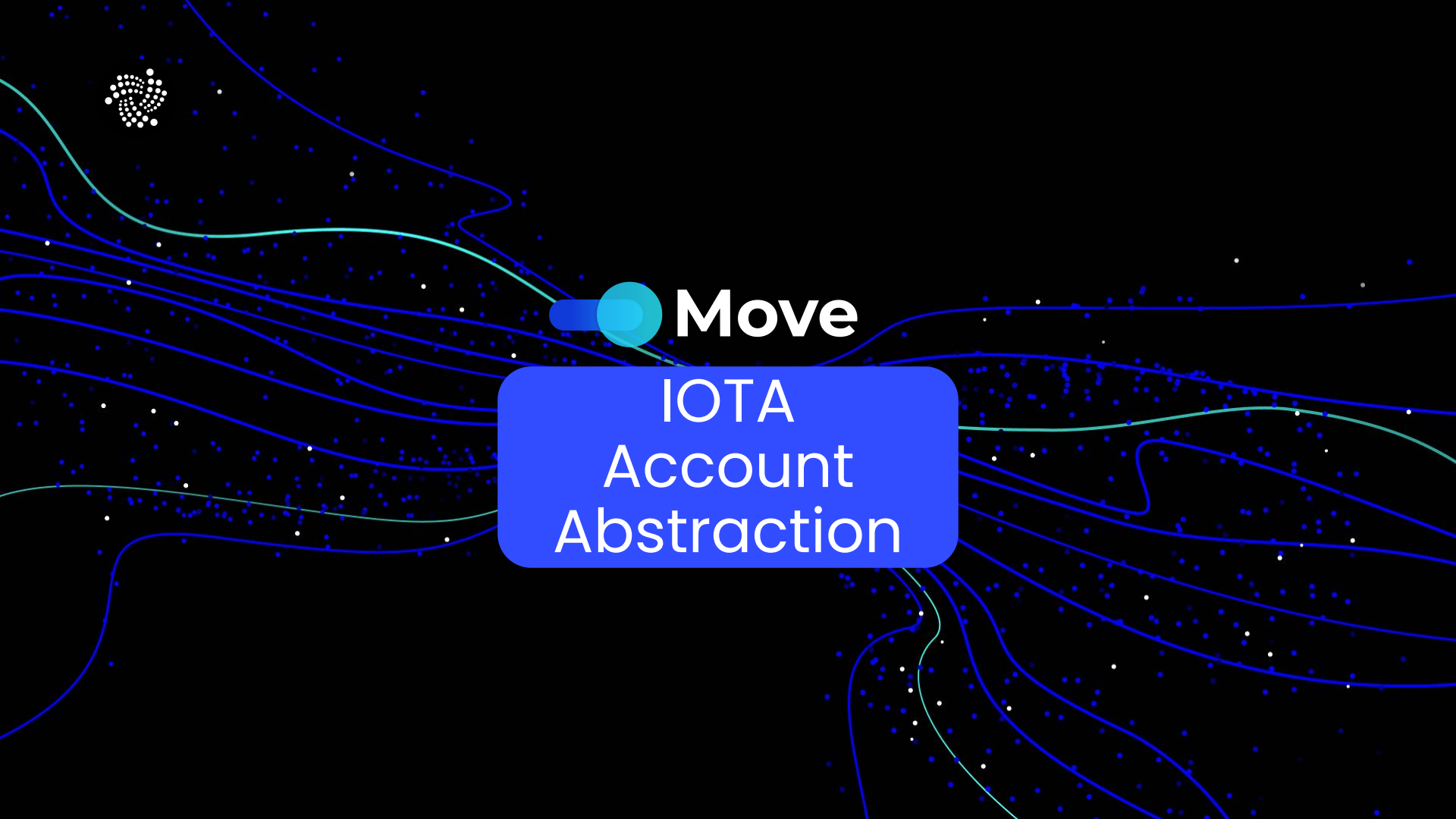
https://intro.sui-book.com/unit-three/lessons/2_intro_to_generics.html#intro-to-generics

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


Move
IOTA
Account
Abstraction

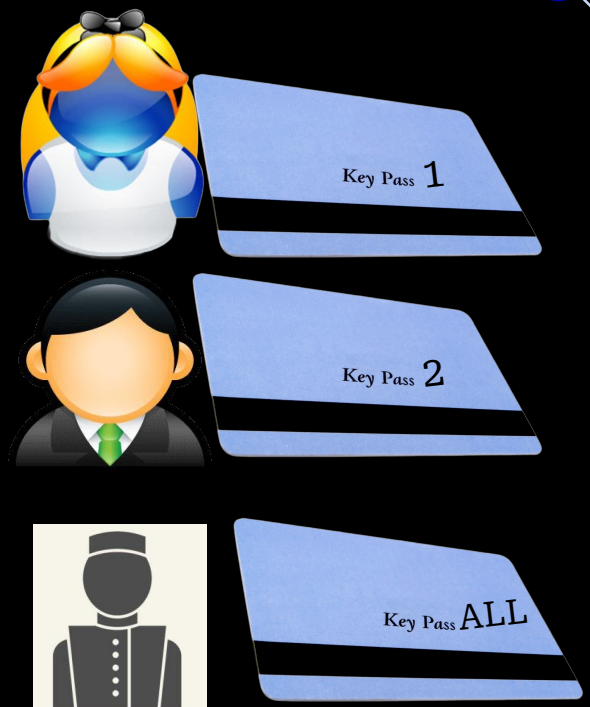
Externally Owned Account (EOA)

Externally Owned Account (EOA)

Externally Owned Account (EOA)

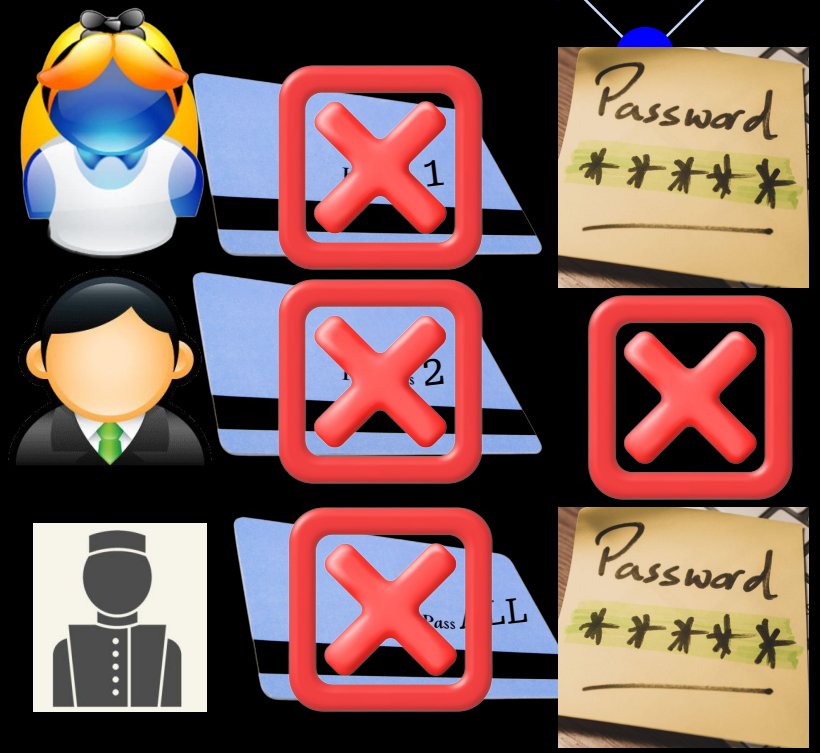
Externally Owned Account (EOA)

Abstract Account (AA)



IOTA

Abstract Account (AA)



12.1. What is Account Abstraction?

Programmable authentication for the next generation of smart accounts

Traditionally, accounts are rigid: **Signature = Ownership**. Rules are hard-coded into the protocol.

Account Abstraction moves this logic into a smart contract. The account itself defines what validates a transaction.

WHAT THIS UNLOCKS

- Custom signature schemes (e.g. Passkeys)
- Multi-signature & Social recovery
- Spending limits & Session keys
- Native Gas Sponsorship

"IOTA implements smart accounts natively through Move, providing the same power as Ethereum's ERC-4337 but with deeper protocol integration."

12.2. How IOTA implements Account Abstraction

On IOTA, an account is a **Move object**, and authentication is a Move function you write, marked `#[authenticator]`.

Before executing any transaction sent **as** that account, the protocol runs the account's logic:

✅ **Returns:** Transaction proceeds

❌ **Aborts:** Transaction is rejected

FRAMEWORK WIRING

```
iota::authenticator_function
  ↳ create_auth_function_ref_v1

iota::account
  ↳ create_account_v1
```

Account Abstraction is a **release candidate**, currently on **devnet/testnet**.
Docs: https://docs.iota.org/developer/account-abstraction

12.3. The key Move building blocks

Core components for implementing Account Abstraction

```
// 1. The account is just an object.
public struct AaAccount has key { id: UID }


// 2. The authenticator the protocol runs before any tx sent AS this account.
// Return = accept, abort = reject.
#[authenticator]
public fun authenticate(
   account: &AaAccount,
   signature: vector<u8>,
   auth_ctx: &AuthContext,
   ctx: &TxContext,
) {
   // verify the owner's signature over the transaction digest:
      assert!(
           ed25519::ed25519_verify(&signature, borrow_public_key(account_id), ctx.digest()),
           EEd25519VerificationFailed,
       );
}
```

The owner's public key is stored on the account as a **dynamic field**, and authenticate_ed25519 checks signature against it over ctx.digest().

12.4. Running a transaction as the account

PART A: GATED ACCESS LOGIC

```
// Gated so only the account can claim.
public entry fun do_something_with_aa(
   account: &AaAccount,
   ctx: &mut TxContext
) {
   assert!(
      account.id.to_address() == ctx.sender(),
      ENotTheAccount
   );
   // ... continue ...
}
```

PART B: CLIENT FLOW

01 Build **unsigned** `do_something_with_aa` tx using `--serialize-unsigned-transaction`

02 Compute the proofs you need for your custom auth logic, e.g., **sign** with owner key (`sign-raw`)

03 Attach the proofs and other attributes via the `--auth-call-args` flag

04 Submit with `iota client execute-combined-signed-tx`

Teaching Caution: *An authenticator is only as strong as what it **enforces**. If it ignores the boolean result of the verify function (missing `assert!`), anyone can act as the account.*

Docs: https://docs.iota.org/developer/account-abstraction | Challenge: **challenges/05-account-abstraction/**

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

Thanks!

IOTA

mirko.zichichi@iota.org