IOTA

4th Scientific School on Blockchain & DLTs

IOTA Smart Contracts and Move

Mirko Zichichi
Research Scientist, IOTA Foundation
mirko.zichichi@iota.org

slides in collaboration with Levente Pap (IOTA Foundation)

Mirko Zichichi

Research Scientist, IOTA Foundation

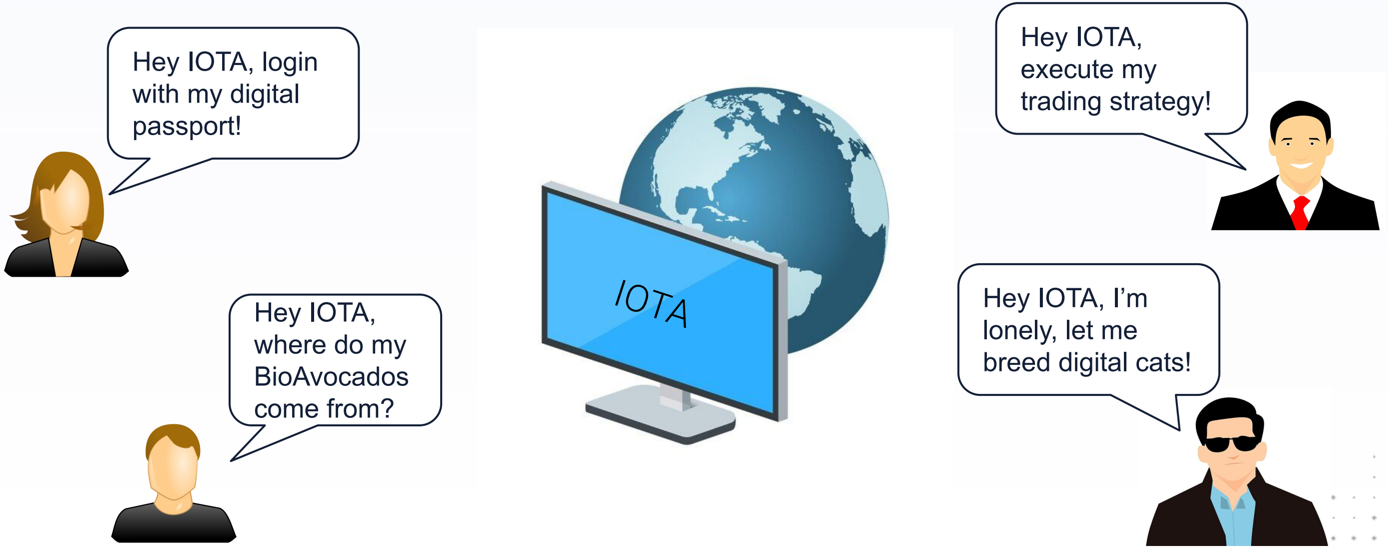
- PhD in **Law, Science and Technology** (MSCA grant)
- Universidad Politécnica de Madrid,
  University of Bologna,
  University of Turin
- Thesis: *"Decentralized Systems for the Protection and Portability of Personal Data"*

What are smart contracts?

Smart Contracts

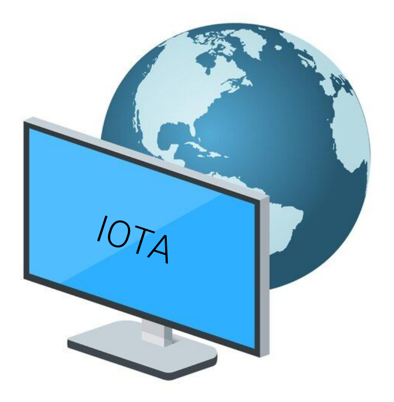
*“A smart contract is **code** deployed in a **blockchain environment**, **OR** the **source code** from which such code was compiled.”*

De Filippi, P. & Wray, C. & Sileno, G. (2021). Smart contracts. Internet Policy Review, 10(2). https://policyreview.info/glossary/smart-contracts



Blockchain Environment -> Shared Global Computer

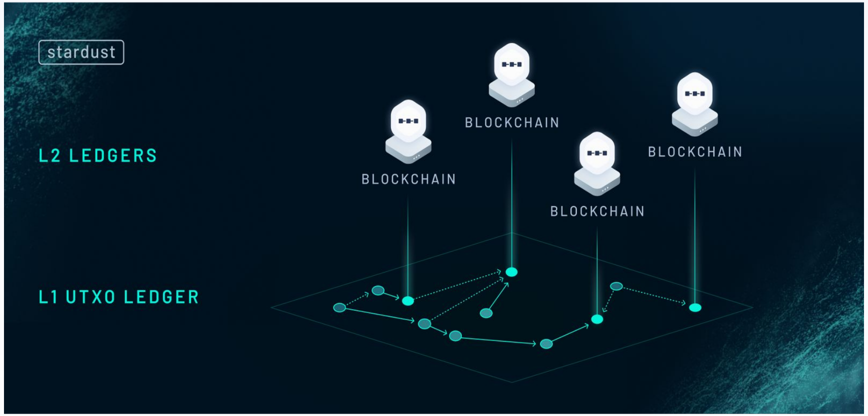
- *“Blockchain-based technologies can be understood as a distributed network of computers, ideally organised in a decentralised way,* ***mutually agreeing on a common state while tolerating failures*** *(incl. malicious behaviour)* ***to some extent****.”* https://policyreview.info/glossary/blockchain-based-technologies
- Notable blockchain operating systems:
  - Ethereum Virtual Machine: Ethereum, BSC, Avalanche, **ISC**, etc.
  - WebAssembly: Polkadot, Cosmos, Near, etc.
  - Bitcoin Script, Cardano Plutus, Radix Engine v2, etc.



IOTA Smart Contracts

IOTA Smart Contracts Today

- It’s a **Layer 2 (L2) solution** where smart contracts are handled off-tangle in their dedicated blockchain
- The blockchain is run by a permissioned committee of nodes.
- Uses Ethereum technology (EVM)
- Periodically commits the state to the **L1**



- **Layer 1** -> **Stardust VM**
- limited in its capabilities: you can’t write your own apps, but you can:
  - Create fungible tokens
  - Create NFTs
  - Store data and/or commitments on-tangle.
- Enhancing L1 with a better operating system -> **increases network’s utility**

https://wiki.iota.org/learn/smart-contracts/core_concepts/smart-contracts/

IOTA Smart Contracts

- In IOTA Smart Contracts, each ISC chain has a **L1 address** (also known as the Chain ID)

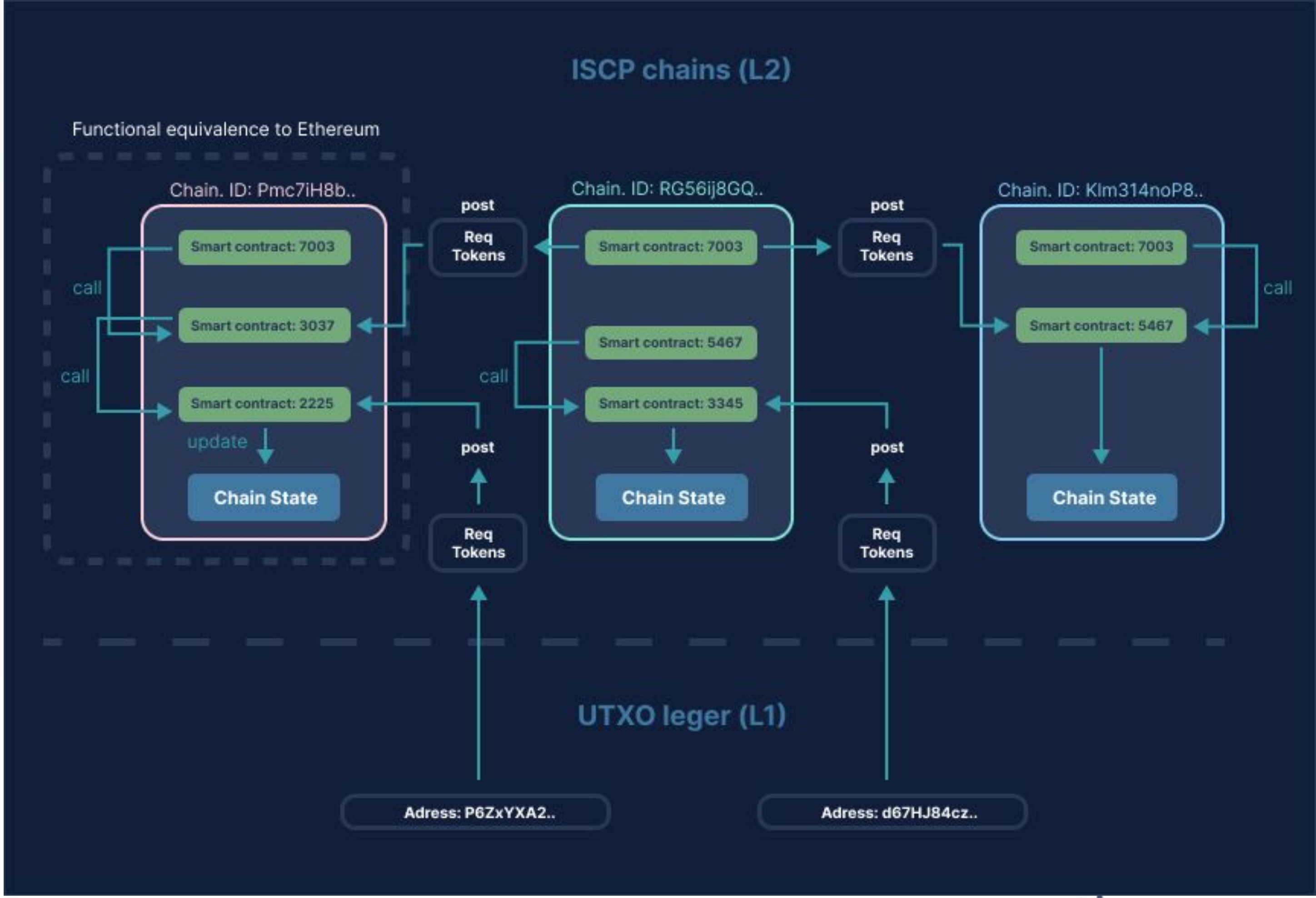
- This address enables an **ISC chain to control L1 assets** (base tokens, native tokens and NFTs)

- The ISC chain, then, acts as a **custodian** of the L1 assets on **behalf of different entities**, that can use them on the L2.

- The **L2 ledger** is a collection of on-chain accounts owned by different agents, i.e., a mapping: Agent (account) ID => balances of L2 assets

https://wiki.iota.org/learn/smart-contracts/core_concepts/smart-contracts/

IOTA Smart Contracts



https://wiki.iota.org/learn/smart-contracts/core_concepts/smart-contracts/

Why have not we built ISC on L1?

Blockchain w/ EVM

Relies on ordered transactions grouped in sequential blocks. The state of the shared computer is updated with each new block.

Global Accounts

User balances are in one giant “excel sheet”, which can only be modified by one transaction at a time.

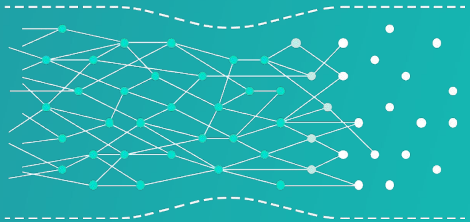
Fees in Base Currency

Execution and storage fees denominated in the native currency of the blockchain.

Heavy Block Execution

With every transaction the whole global state is updated and needs to be recalculated. Demands CPU and memory, so beefier machines.

Tangle



BlockDAG with parallel blocks and causal transaction ordering. Shared computer state is updated concurrently.

UTXO Accounts

User balances are represented as cash notes which can be exchanged any time without waiting for others.

No (explicit) Fees Before IOTA 2.0

In current IOTA fees are “paid” in PoW, in IOTA 2.0 with MANA.

Light Block Execution

Each block updates only part of the global state, so no need to recalculate everything. Execution could be distributed among several machines for scaling.

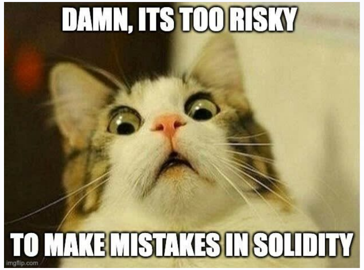
Levels of Programmability

Levels of Programmability

- Pure “Unspent Transaction Output” - UTXO (Chrysalis - IOTA 1.0)
  - Track money balances, **no programs**
- UTXOs with hard coded scripts (Stardust - IOTA 1.5)
  - Execute **predefined programs**
- UTXOs with limited scripting (Bitcoin)
  - Write **some programs**
- UTXOs with Turing-complete scripts (Cardano Extended UTXO)
  - Write **any program**, but **composability is cumbersome** *(no atomic combined operations)*
- Account based ledger with virtual machine (ETH, Near, Polkadot, etc.)
  - Write **any program** and **composability is easy**

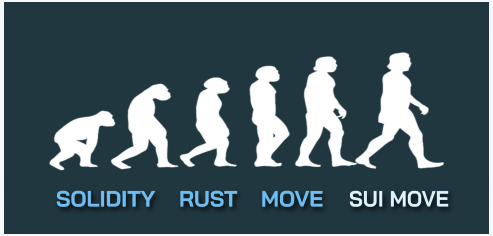
In ETH, Near, etc. full programmability and composability, but what about…

- **Safety?**
    - Multimillion dollar exploits happen on a weekly basis. SCs are not formally verifiable. The language in which they are coded has several “gotcha”s. Poor platform for programming money.
    - Composability through opaque interfaces and dynamic callbacks: hijacking contract calls is a feature, not a bug.
- **User experience?**
    - Each SC is a walled garden. Need to add assets manually to a wallet.
    - You never know what you actually approve via MetaMask.
- **Scalability?**
    - A popular NFT mint clogs the chain and increases gas fees for everyone. Transactions have to wait on each other, even if they are unrelated.



Move

Move Language



- Domain Specific Language for programming with assets

- Inherits **memory and type safety** concepts from Rust
  - **Compiler catches errors** that would normally go undetected in Solidity

- Treats **assets as first class citizens** that can travel between SC boundaries

- Programs are **formally verifiable**

- Built-in language level **permission controls**
  - Transparent what an SC can do with your assets (**read only, mutate, transfer**)

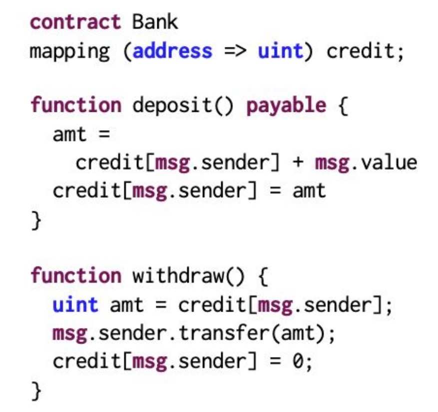
Move “Resource”

- Move represents assets using **user-defined linear resource types**.
- Move has ordinary types like *integers and addresses that can be copied*, but **resources can only be moved**.
- Move **resource safety** -> analogous to conservation of mass in the physical world
- ***Linearity***:
    - prevents “double spending” by moving a resource twice
    - forces a procedure to move all of its resources, avoiding accidental loss.

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

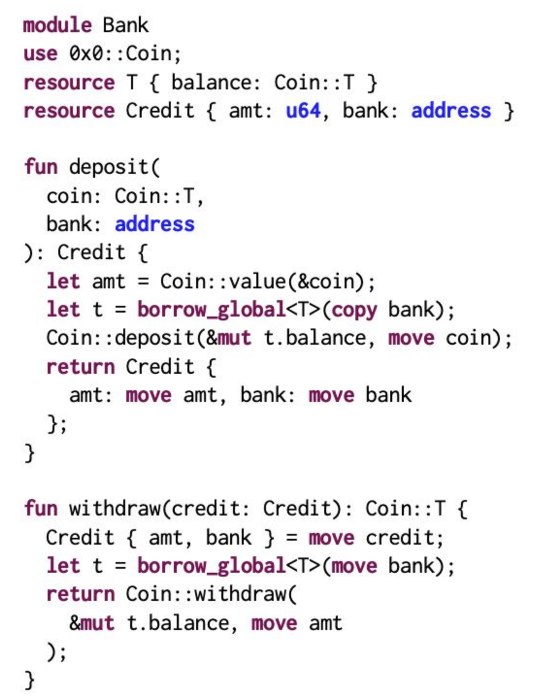
Move “Resource”

Solidity



```
contract Bank
mapping (address => uint) credit;

function deposit() payable {
  amt =
    credit[msg.sender] + msg.value
  credit[msg.sender] = amt
}

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

Move

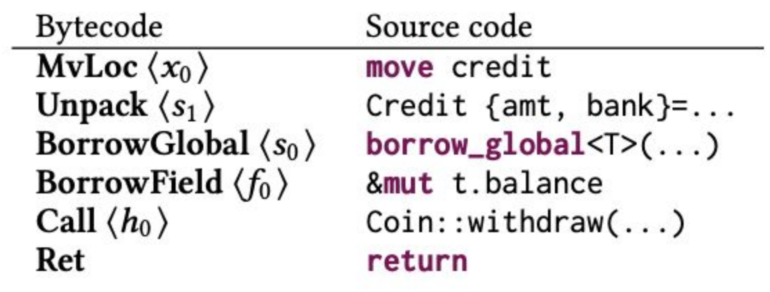


```
module Bank
use 0x0::Coin;
resource T { balance: Coin::T }
resource Credit { amt: u64, bank: address }

fun deposit(
  coin: Coin::T,
  bank: address
): Credit {
  let amt = Coin::value(&coin);
  let t = borrow_global<T>(copy bank);
  Coin::deposit(&mut t.balance, move coin);
  return Credit {
    amt: move amt, bank: move bank
  };
}

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).

Move Executable Bytecode

- A Move execution platform relies on a **compiler** to transform ***source language*** programs into programs in the Move **bytecode language**.



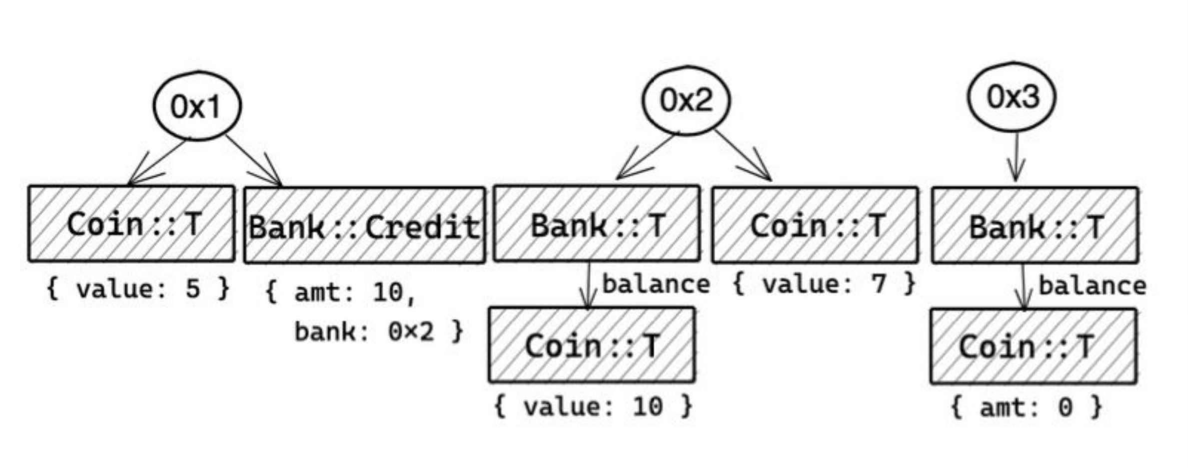
Bytecode	Source code
MvLoc $\langle x_0 \rangle$	`move credit`
Unpack $\langle s_1 \rangle$	`Credit {amt, bank}=...`
BorrowGlobal $\langle s_0 \rangle$	`borrow_global<T>(...)`
BorrowField $\langle f_0 \rangle$	`&mut t.balance`
Call $\langle h_0 \rangle$	`Coin::withdraw(...)`
Ret	`return`

- The Move execution platform relies on a *load-time* **bytecode verifier**, that enforces *type, memory, and resource safety*.
  - If the safety guarantees were only enforced by the compiler, an adversary could subvert them by writing malicious bytecode directly and deploying it

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

Move Persistent Global State

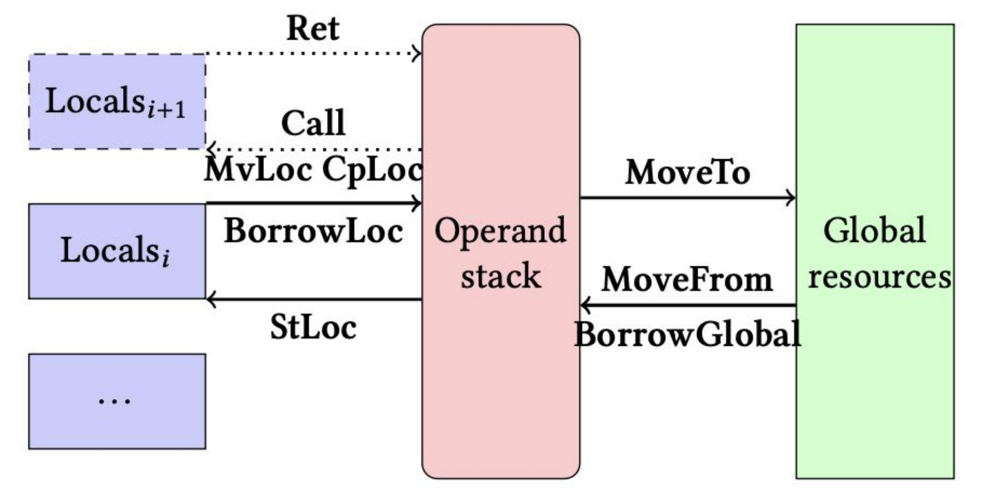
- Move execution occurs in the context of a **persistent global state** organized as a partial **map** from ***account addresses*** -> ***resource data values***



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

Move Execution

- Begins by executing the **main procedure of the transaction script**
- A **procedure** is defined by a ***type signature*** and an ***executable body*** (Move bytecode commands).
- Procedure calls are implemented using a standard **call stack** containing frames with a ***procedure name***, a set of ***local variables***, and a ***return address***.



Blackshear, Sam, et al. "Resources: A

Move Module, i.e., the Smart Contract

- A Move module can declare both **record types** and **procedures**.
- **Records** can store primitive data values (booleans, addresses, …) as well as other record values:
    - each record is declared as a resource or non-resource;
    - non-resource records cannot store resource records;
    - only resources can be stored in the global state.
- **Module's strong encapsulation:**
  privileged ***operations*** on the module's declared ***types*** can **only** be ***performed by procedures in the module***

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

Move References

- Move supports **references to** *records* and *primitive values*:

  all reads and writes of record fields occur through a reference.
- ,References are either:
  - exclusive/mutable -> ***&mut***
  - read-only -> **&**
- References are different from other Move values because they are **transient**
  - each reference must be created during the execution of a transaction script

    and **released before the end** of that transaction script.

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

Move Resource Safety

- At the beginning and end of a transaction script, all of the resources in the system reside in the **global state** *GS*.
- **Resource safety** is a conservation property that relates the set of resources present in state GS_{pre} before the script to the set of resources present in state GS_{post} after the script.
- In general terms, must **guarantee** that:
    - A **resource** *M::T* that is **present** in **post-state** GS_{post} was also **present** in **pre-state** GS_{pre} unless it is introduced by a *Pack* (Move bytecode for resource creation) inside *M* during script execution
    - A **resource** *M::T* that was **present** in **pre-state** GS_{pre} is also **present** in **post-state** GS_{post} unless it is eliminated by an *Unpack* (Move bytecode for resource deletion) inside *M* during script execution

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

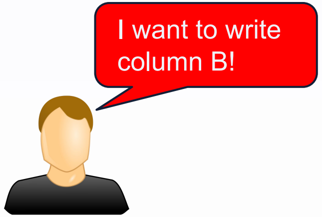
Move Flavors

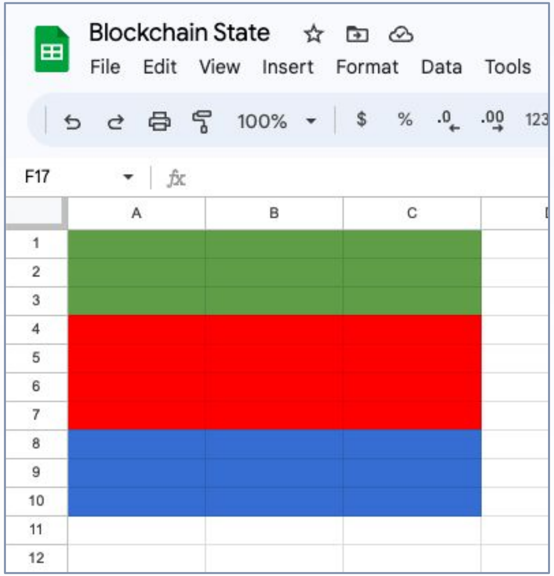
Move Virtual Machine as the Blockchain OS

- **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

How do you access the shared computer’s memory?

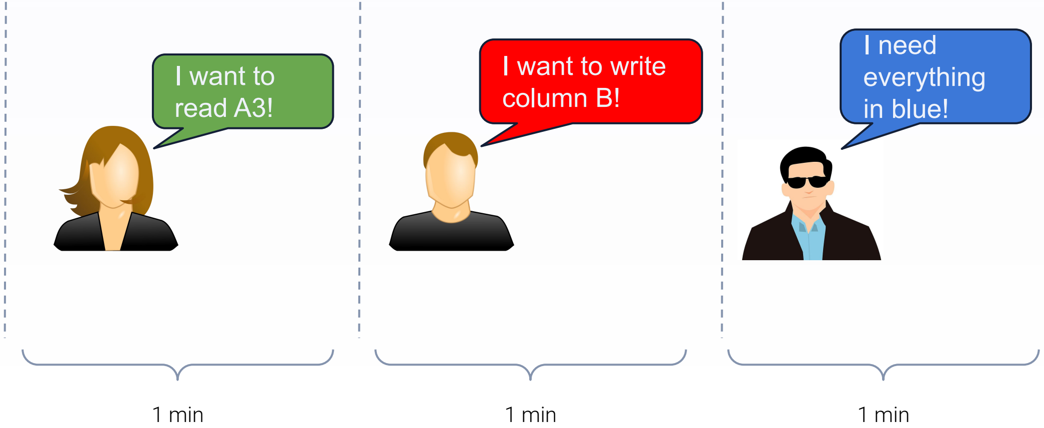
- Everybody wants to edit the same sheet
- One person needs 1 minute to update a cell
- Determine the order and time it takes to edit the sheet with below requests

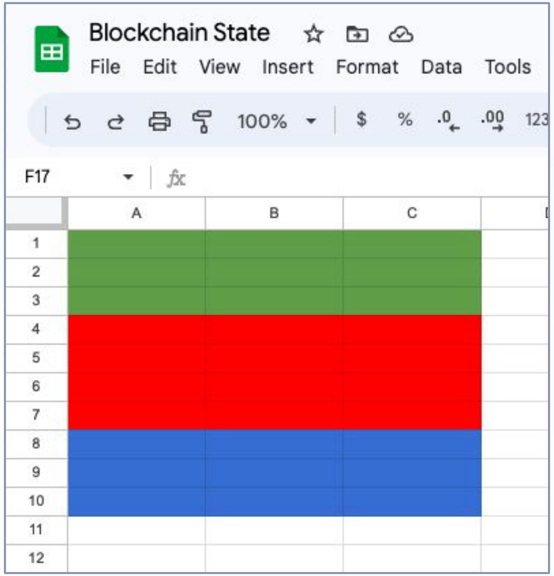




How do you access the shared computer’s memory?

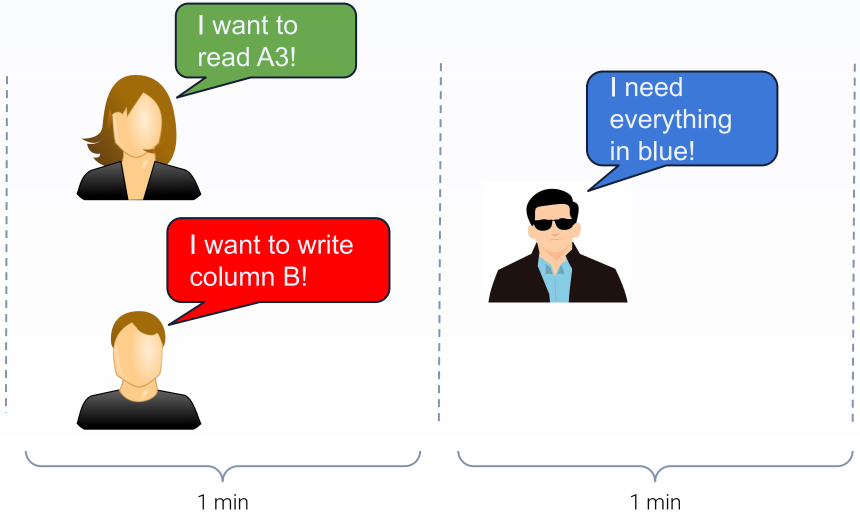
- The Aptos(EVM) way: **Unified Global Memory**
- Rule: one person at a time can open the sheet, make the changes and then save it.
- Takes 1 + 1 + 1 = 3 minutes until everyone finishes.

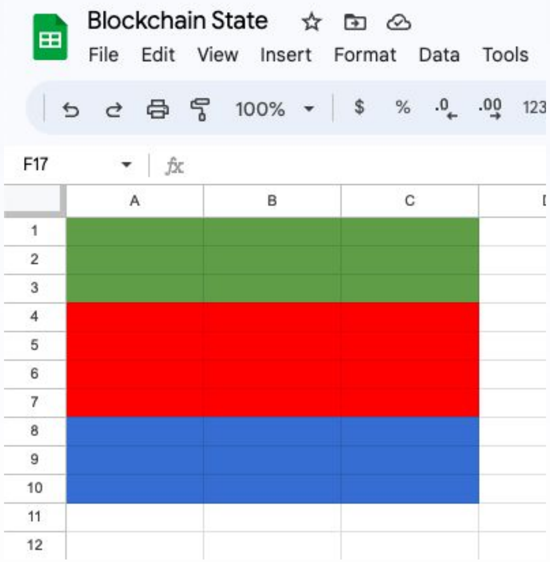




How do you access the shared computer’s memory?

- The Sui way: **Partitioned Global Memory**
- Rule: declare which cells you’ll edit. If they are not in use, go ahead and edit them!
- Takes 1 + 1 = 2 minutes until everyone finishes.





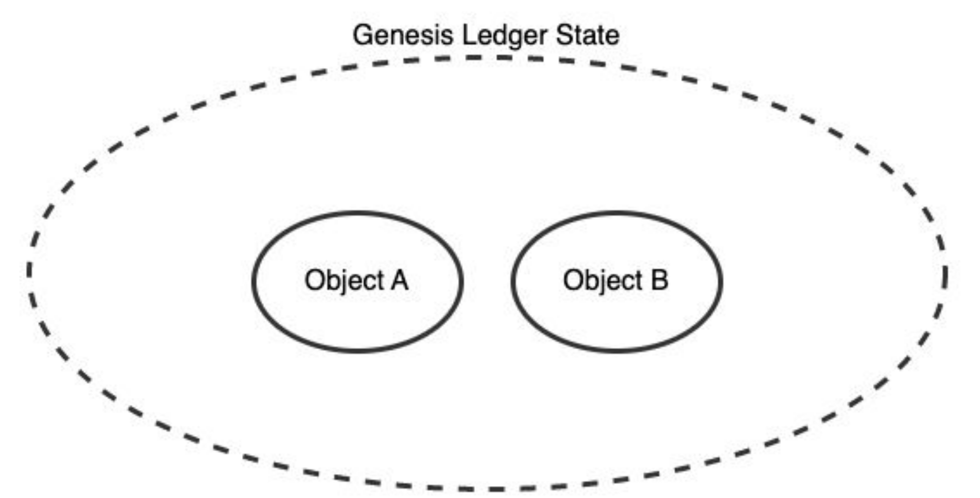
Move on Account vs Object Ledger

- Modelling a transaction's access to blockchain state with smart contracts is analogous to modelling memory access in a computer by different threads.
- The Blockchain OS determines the access strategy
- **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

IOTA flavored Move

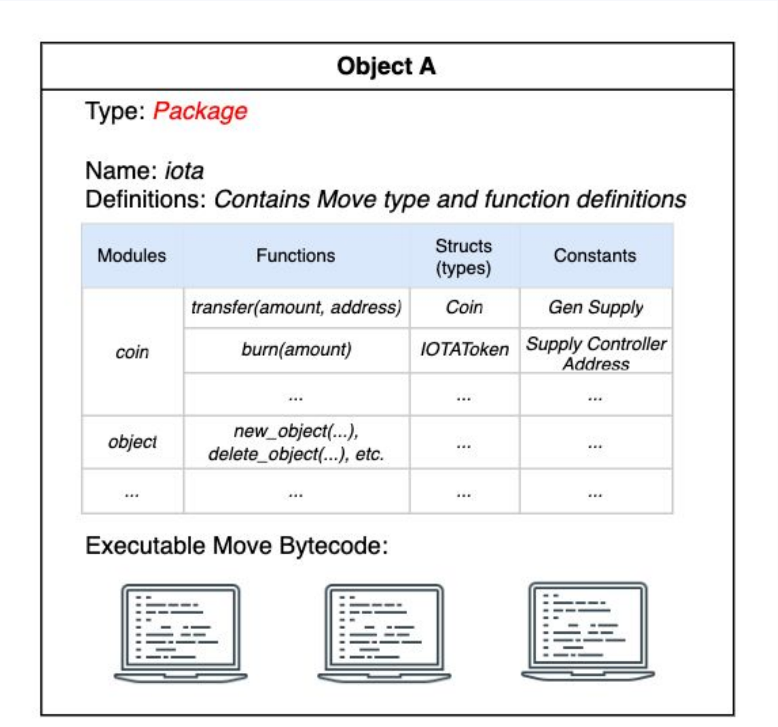
0. Ledger Basics - The State

- Every **entry in the ledger state is an Object**
- Object = Move Resource



0. Ledger Basics

- A **Package Object** is an immutable read-only object that contains one or several ***Move modules***
- During Genesis, a Object A is created containing the **framework package** -> a set of modules defining the main operations performed in the IOTA Tangle
- E.g., ***Coin* module** defines resource types, such as how an IOTA *Coin* looks like.
    - It exposes the *transfer* function that defines how to transfer the *Coin*.



Object A

Type: *Package*

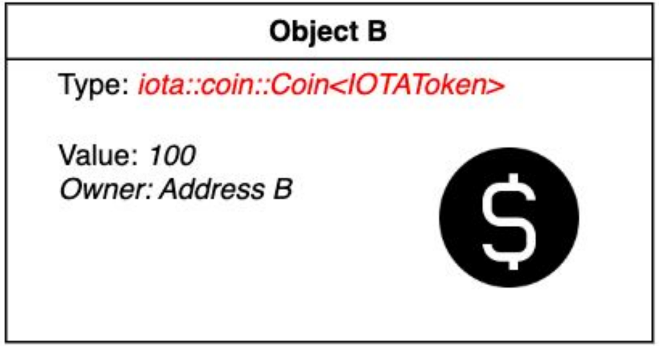
Name: *iota*
Definitions: *Contains Move type and function definitions*

Modules	Functions	Structs (types)	Constants
coin	*transfer(amount, address)*	*Coin*	*Gen Supply*
	burn(amount)	*IOTAToken*	*Supply Controller Address*
	...	...	...
object	*new_object(...),* *delete_object(...), etc.*	...	...
...	...	...	...

Executable Move Bytecode:

0. Ledger Basics

- A **Move Object** is an instantiation of a resource type previously defined in a module.
- **Object B** is a *Coin* that holds IOTA.
- Object B has two fields:
    - The amount of coins (100), and
    - The owner of the object (Address B)



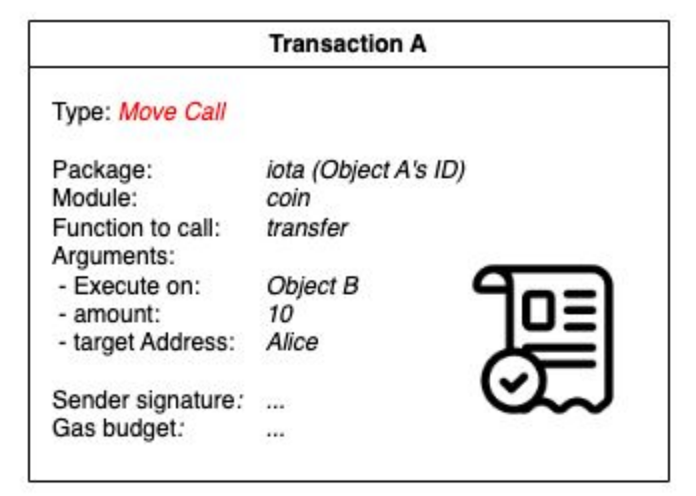
Object B

Type: *iota::coin::Coin<IOTAToken>*

Value: *100*
Owner: Address B

0. Ledger Basics

- **A transaction calls a function in a module**. The arguments to the function could be:
    - Move Objects,
    - Pure arguments (addresses, numbers, strings, bytes)



Transaction A

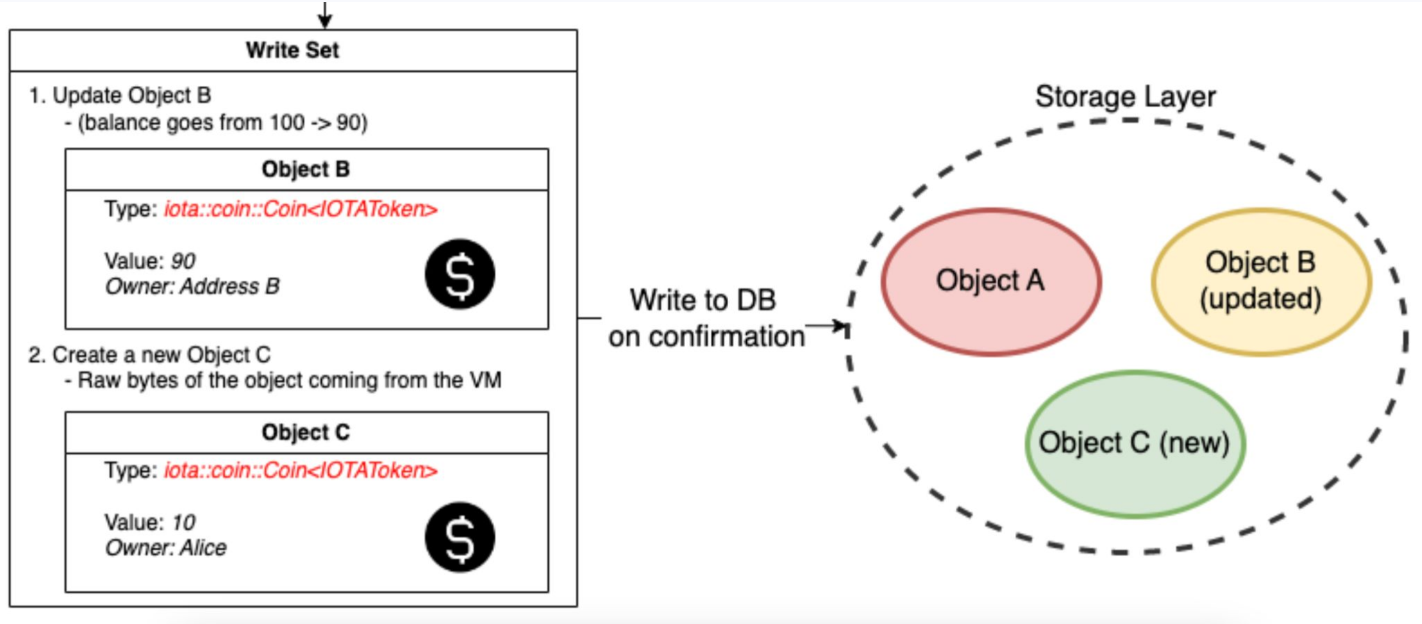
Type: *Move Call*

Package: *iota (Object A's ID)*
Module: *coin*
Function to call: *transfer*
Arguments:
- Execute on: *Object B*
- amount: *10*
- target Address: *Alice*

Sender signature: ...
Gas budget: ...

0. Ledger Basics - Transaction Execution

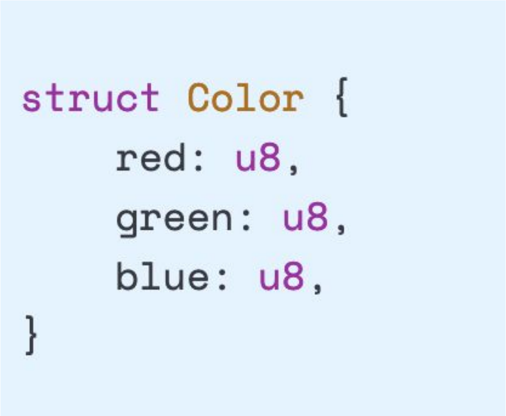
- The outcome of **transaction execution** is what to update in the ledger state. (write set)



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

```
use sui::object::UID;

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

https://docs.sui.io/build/programming-with-objects/ch1-object-basics

1. Object Basics - Key

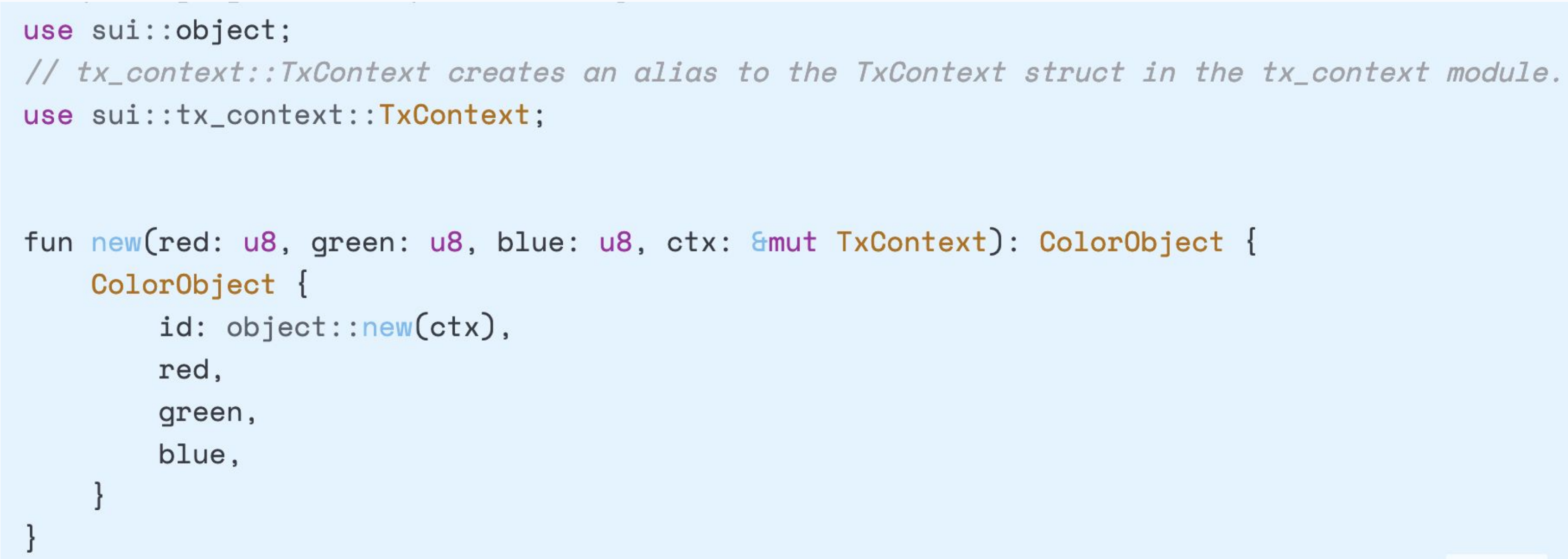
- In Move the **key** ability denotes a type that can appear as a key in global storage
- Core Move uses a **(type, address)-indexed map**
- Sui/IOTA Move uses a **map keyed** by **object IDs**.

```
use sui::object::UID;

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

https://docs.sui.io/build/programming-with-objects/ch1-object-basics

1. Object Basics - Create an Object

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

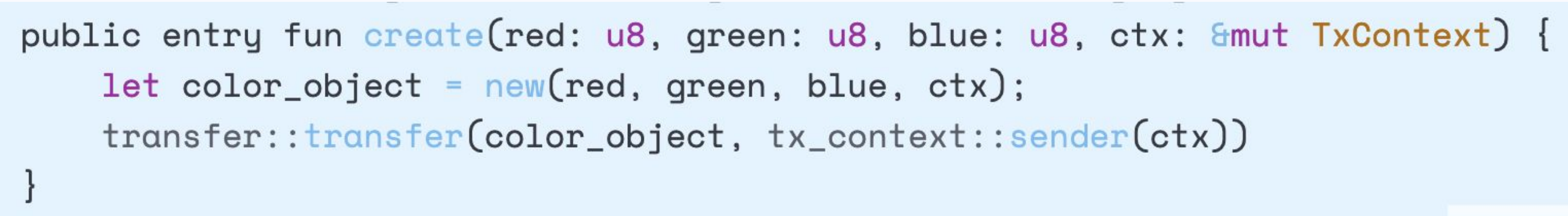


```
use sui::object;
// tx_context::TxContext creates an alias to the TxContext struct in the tx_context module.
use sui::tx_context::TxContext;


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

https://docs.sui.io/build/programming-with-objects/ch1-object-basics

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.sui.io/build/programming-with-objects/ch1-object-basics

2. Using Objects

- Sui/IOTA Move **authentication mechanisms** ensure ***only you can use objects owned by you*** 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.sui.io/build/programming-with-objects/ch2-using-objects

2. 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.sui.io/build/programming-with-objects/ch2-using-objects

2. 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.sui.io/build/programming-with-objects/ch2-using-objects

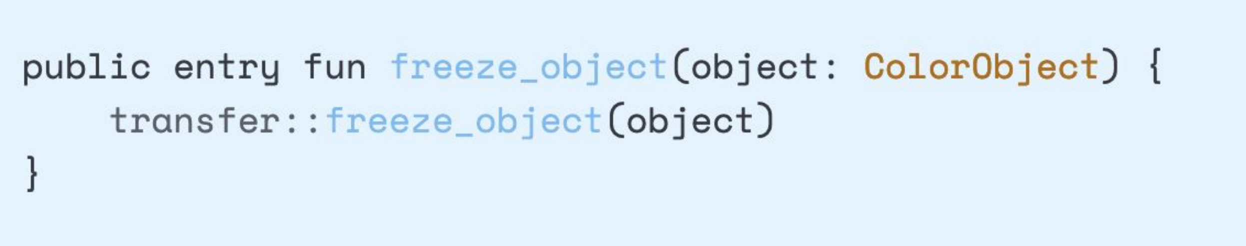
3. Shared and Immutable Objects

- Objects in IOTA can have different types of **ownership**, with two broad categories:
    - **mutable objects** -> **can be owned by an address/object or can be shared**
    - immutable objects -> an object that can't be mutated, transferred or deleted.
- **Shared object:** *anyone can read or write this object*.
    - mutable owned objects are single-writer
    - shared objects require to sequence reads and writes
- In other blockchains, ***every object is shared***
- In Sui/IOTA Move programmers have the choice to implement a particular use-case using **shared objects, owned objects, or a combination**.
- In Sui, 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.sui.io/learn/objects

3. Shared and Immutable Objects

- Objects in IOTA can have different types of **ownership**, with two broad categories:
  - mutable objects -> can be owned by an address/object or can be shared
  - **immutable objects** -> **an object that can't be mutated, transferred or deleted.**
- Immutable objects have no owner, so anyone can use them
  - packages are immutable objects
  - you can freeze an initially mutable object



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

https://docs.sui.io/build/programming-with-objects/ch3-immutable-objects

4. Object Wrapping

- In Sui/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.sui.io/build/programming-with-objects/ch4-object-wrapping

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.sui.io/build/programming-with-objects/ch4-object-wrapping

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.sui.io/build/programming-with-objects/ch4-object-wrapping

5. Dynamic Fields

- Sui/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.sui.io/build/programming-with-objects/ch5-dynamic-fields

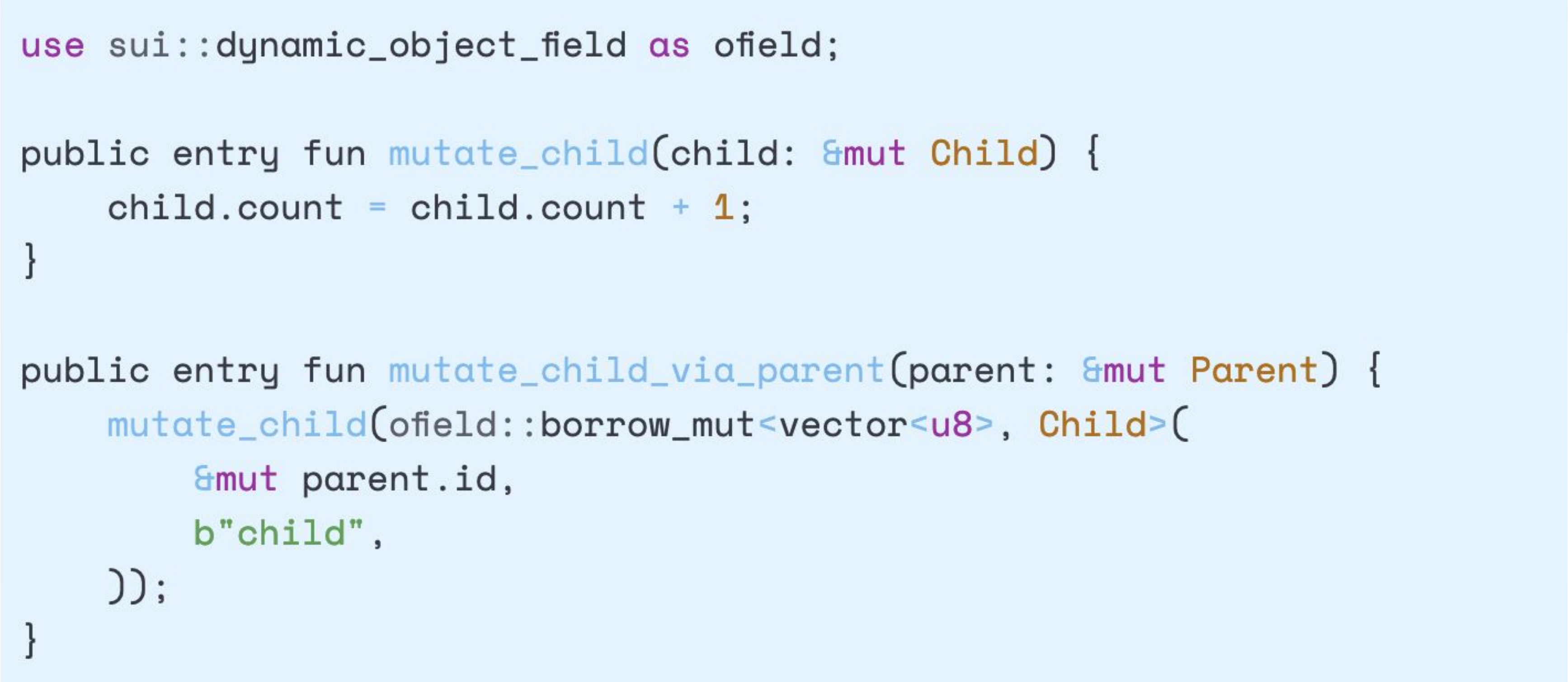
5. Dynamic Fields - Add field

- 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 sui::dynamic_object_field as ofield;

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

https://docs.sui.io/build/programming-with-objects/ch5-dynamic-fields

5. Dynamic Fields - Access field



```
use sui::dynamic_object_field as ofield;

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

public entry fun mutate_child_via_parent(parent: &mut Parent) {
    mutate_child(ofield::borrow_mut<vector<u8>, Child>(
        &mut parent.id,
        b"child",
    ));
}
```

https://docs.sui.io/build/programming-with-objects/ch5-dynamic-fields

5. Dynamic Fields - Remove field

```
use sui::dynamic_object_field as ofield;
use sui::{object, transfer, tx_context};
use sui::tx_context::TxContext;

public entry fun delete_child(parent: &mut Parent) {
    let Child { id, count: _ } = ofield::remove<vector<u8>, Child>(
        &mut parent.id,
        b"child",
    );

    object::delete(id);
}

public entry fun reclaim_child(parent: &mut Parent, ctx: &mut TxContext) {
    let child = ofield::remove<vector<u8>, Child>(
        &mut parent.id,
        b"child",
    );

    transfer::transfer(child, tx_context::sender(ctx));
}
```

https://docs.sui.io/build/progran

6. Write a IOTA Move Package - Modules file

```
cat my_first_package/Move.toml
[package]
name = "my_first_package"
version = "0.0.1"

[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir = "crates/sui-frame

[addresses]
my_first_package = "0x0"
sui = "0000000000000000000000000000000000000000000000000000000000000002"
```

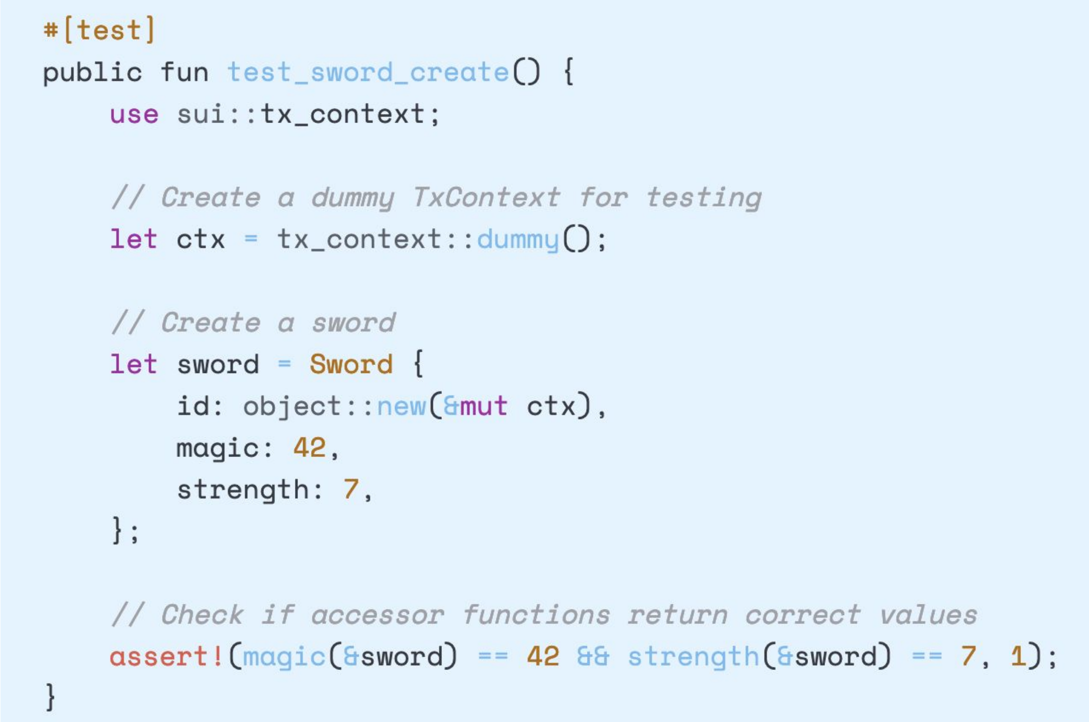
https://docs.sui.io/build/programming-with-objects/ch5-dynamic-fields

6. Write an IOTA Move Package

```
module my_first_package::my_module {

    // Part 1: Imports
    use sui::object::{Self, UID};
    use sui::transfer;
    use sui::tx_context::{Self, TxContext};

    // Part 2: Struct definitions
    struct Sword has key, store {
        id: UID,
        magic: u64,
        strength: u64,
    }

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

    // Part 3: 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::transfer(admin, tx_context::sender(ctx));
    }

    // Part 4: Accessors required to read the struct attributes
    public fun magic(self: &Sword): u64 {
        self.magic
    }
```

https://docs.sui.io/build/programming-with-objects/ch5-dynamic-fields

6. Write an IOTA Move Package - Testing



```
#[test]
public fun test_sword_create() {
    use sui::tx_context;

    // Create a dummy TxContext for testing
    let 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.sui.io/build/programming-with-objects/ch5-dynamic-fields

Open Research Questions

Research Questions

- **Shared objects:** they need causal total of transaction
  - Unlike most existing DLTs, **Sui** does **NOT** impose a **total order** on many TXs.
  - TXs touching *ONLY owned objects* are **causally ordered:**
    - if a transaction T1 produces output objects O1 that are used as input objects in a transaction T2, a validator must execute T1 before it executes T2.
  - IOTA 2.0 also uses casual order for UTXO TXs. **How can we integrate shared objects in it?** Is it needed an additional consensus mechanism (for total ordering)?
- Objects Ledger vs. IOTA's 2.0 Augmented UTXO. What are the Advantages and Disadvantages?
  - Considering we continue with the UTXO ledger. Will handling objects as UTXOs any adverse impact on the performance, security, or scalability of the L1?

https://docs.sui.io/build/programming-with-objects/ch5-dynamic-fields

IOTA

Questions (from you)?

Mirko Zichichi

Research Scientist, IOTA Foundation
mirko.zichichi@iota.org

IOTA

Thank you!

Mirko Zichichi

Research Scientist, IOTA Foundation
mirko.zichichi@iota.org