sui

Project Showcase: **On-chain RPG** (Part 1)

March 13, 2023

Agenda

sui

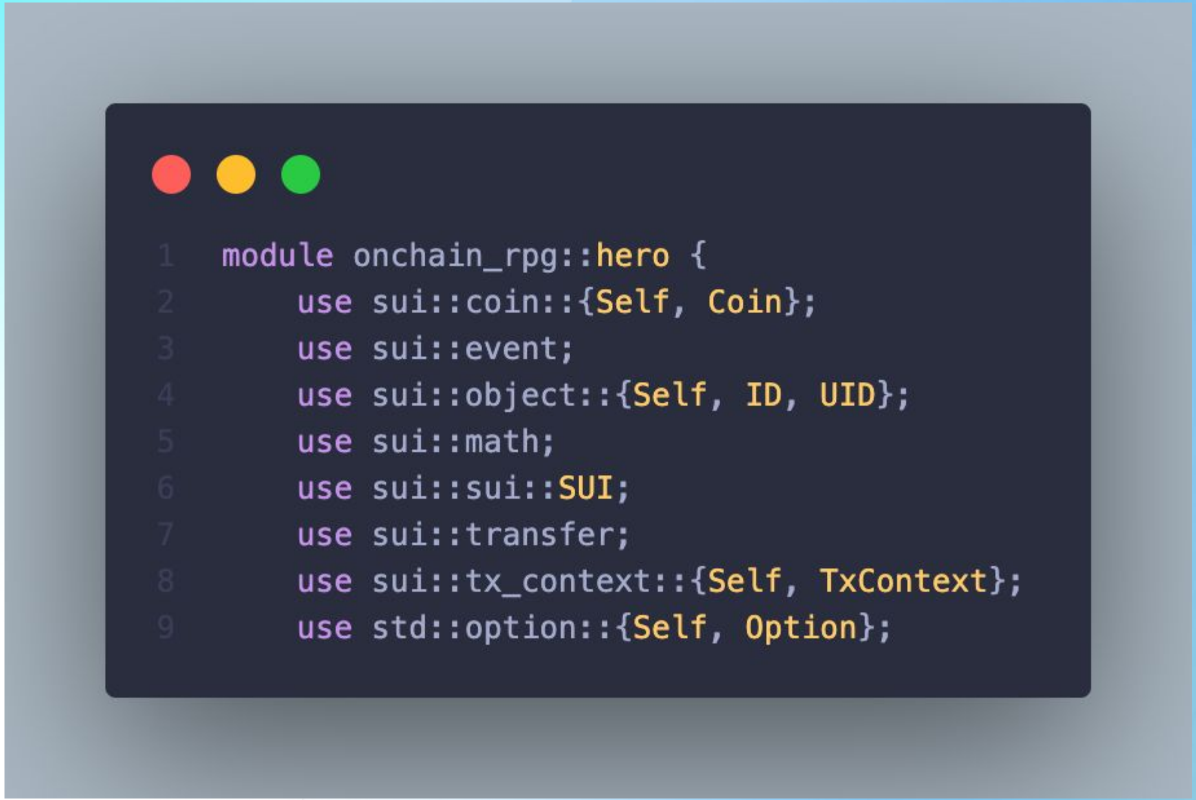
Game Time! ⚔️

In a new terminal enter: `sui move new onchain_rpg`

Adapted from:

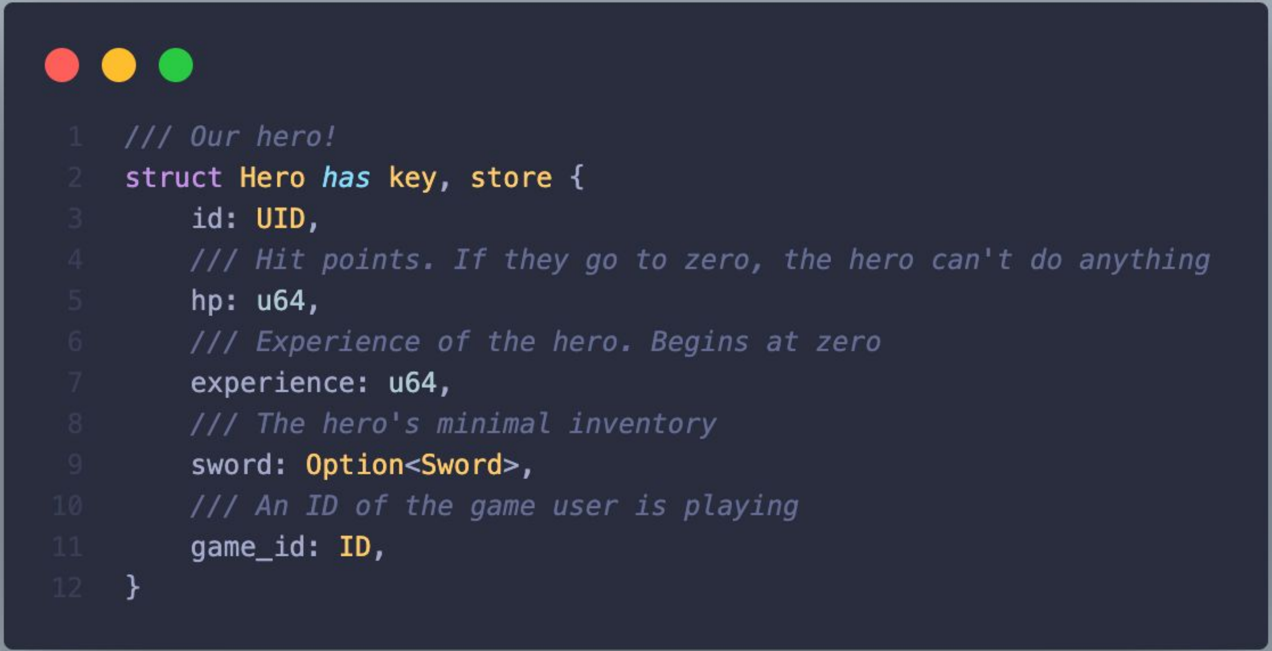
github.com/MystenLabs/sui/blob/main/sui_programmability/examples/games/sources/hero.move

sui



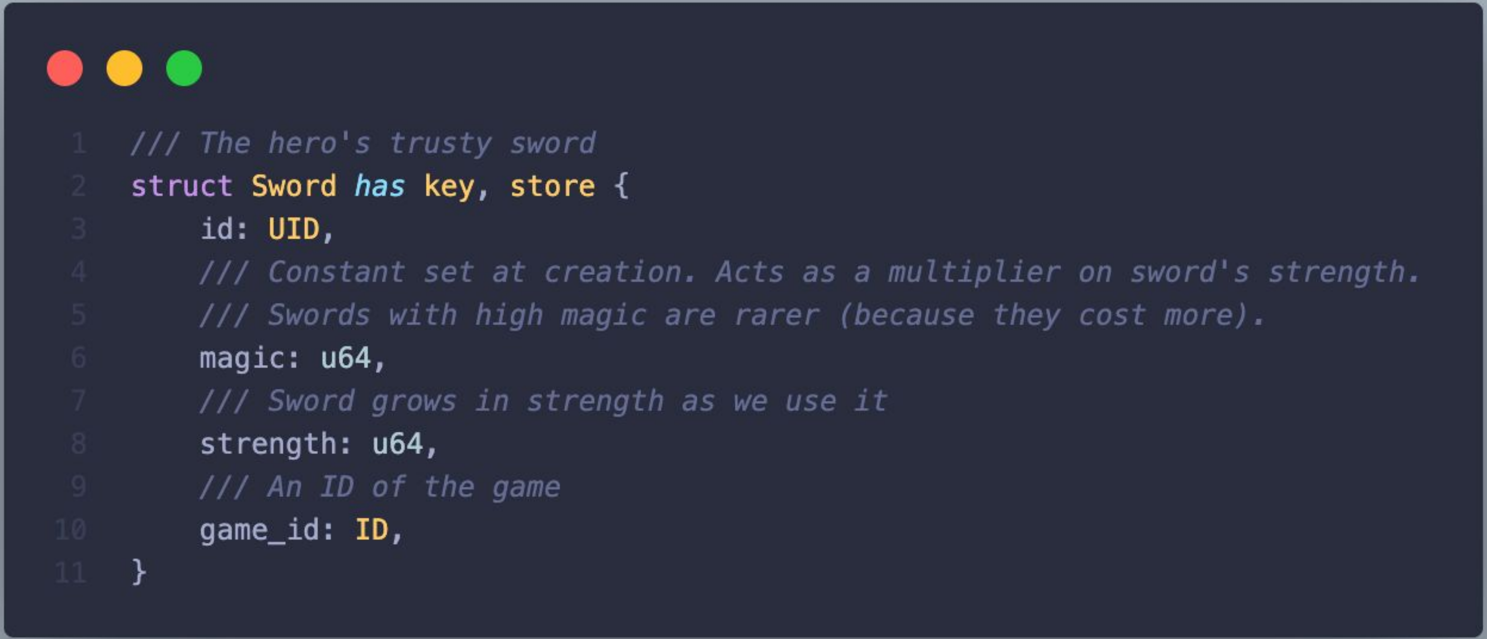
```
module onchain_rpg::hero {
    use sui::coin::{Self, Coin};
    use sui::event;
    use sui::object::{Self, ID, UID};
    use sui::math;
    use sui::sui::SUI;
    use sui::transfer;
    use sui::tx_context::{Self, TxContext};
    use std::option::{Self, Option};
```

Objects

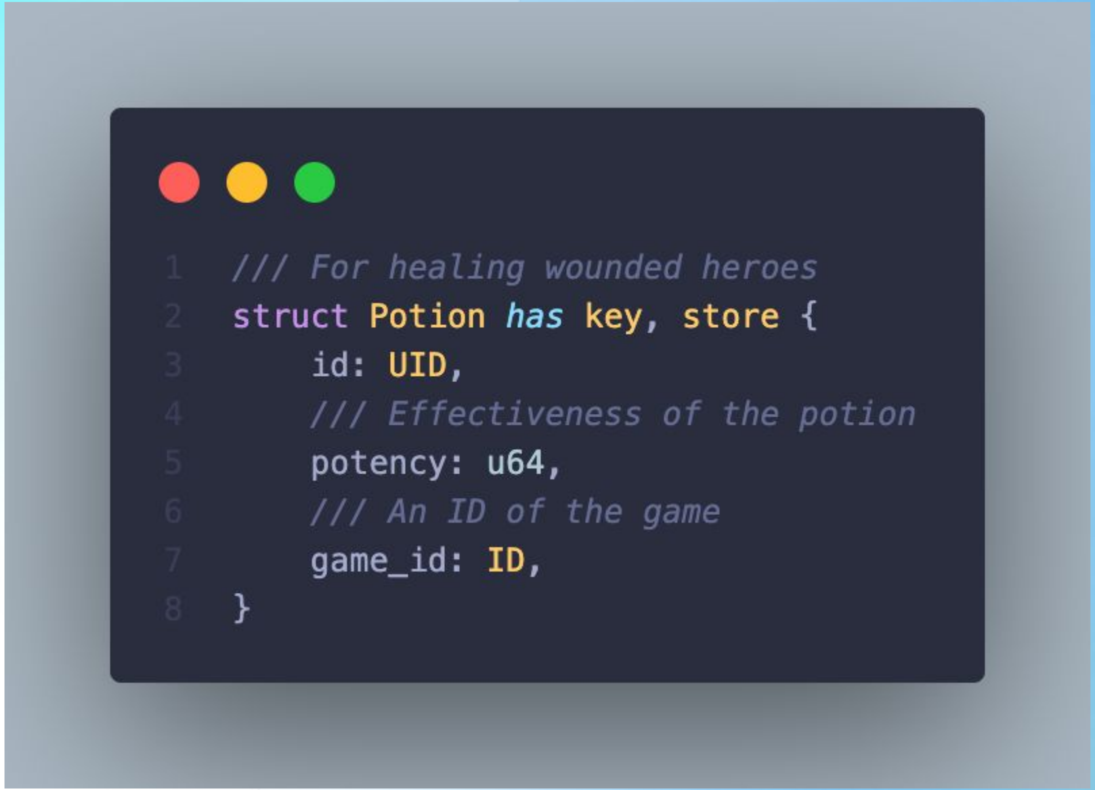


```
/// Our hero!
struct Hero has key, store {
    id: UID,
    /// Hit points. If they go to zero, the hero can't do anything
    hp: u64,
    /// Experience of the hero. Begins at zero
    experience: u64,
    /// The hero's minimal inventory
    sword: Option<Sword>,
    /// An ID of the game user is playing
    game_id: ID,
}
```

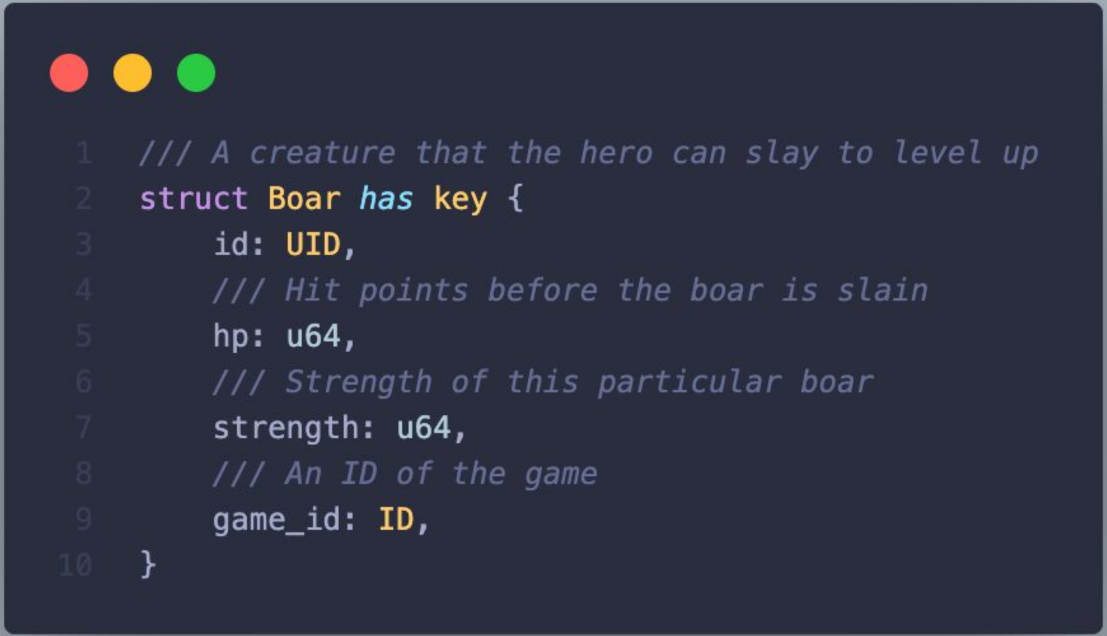
sui



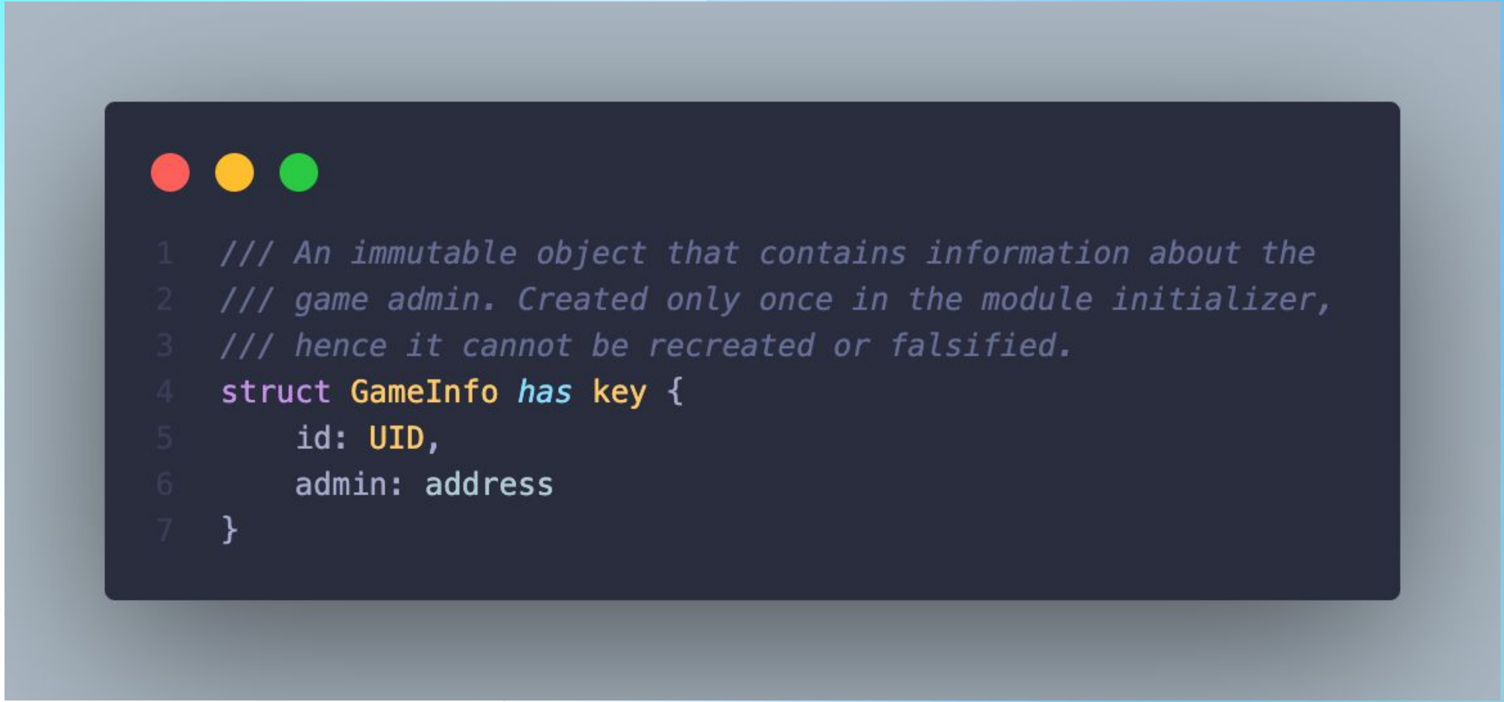
```
/// The hero's trusty sword
struct Sword has key, store {
    id: UID,
    /// Constant set at creation. Acts as a multiplier on sword's strength.
    /// Swords with high magic are rarer (because they cost more).
    magic: u64,
    /// Sword grows in strength as we use it
    strength: u64,
    /// An ID of the game
    game_id: ID,
}
```



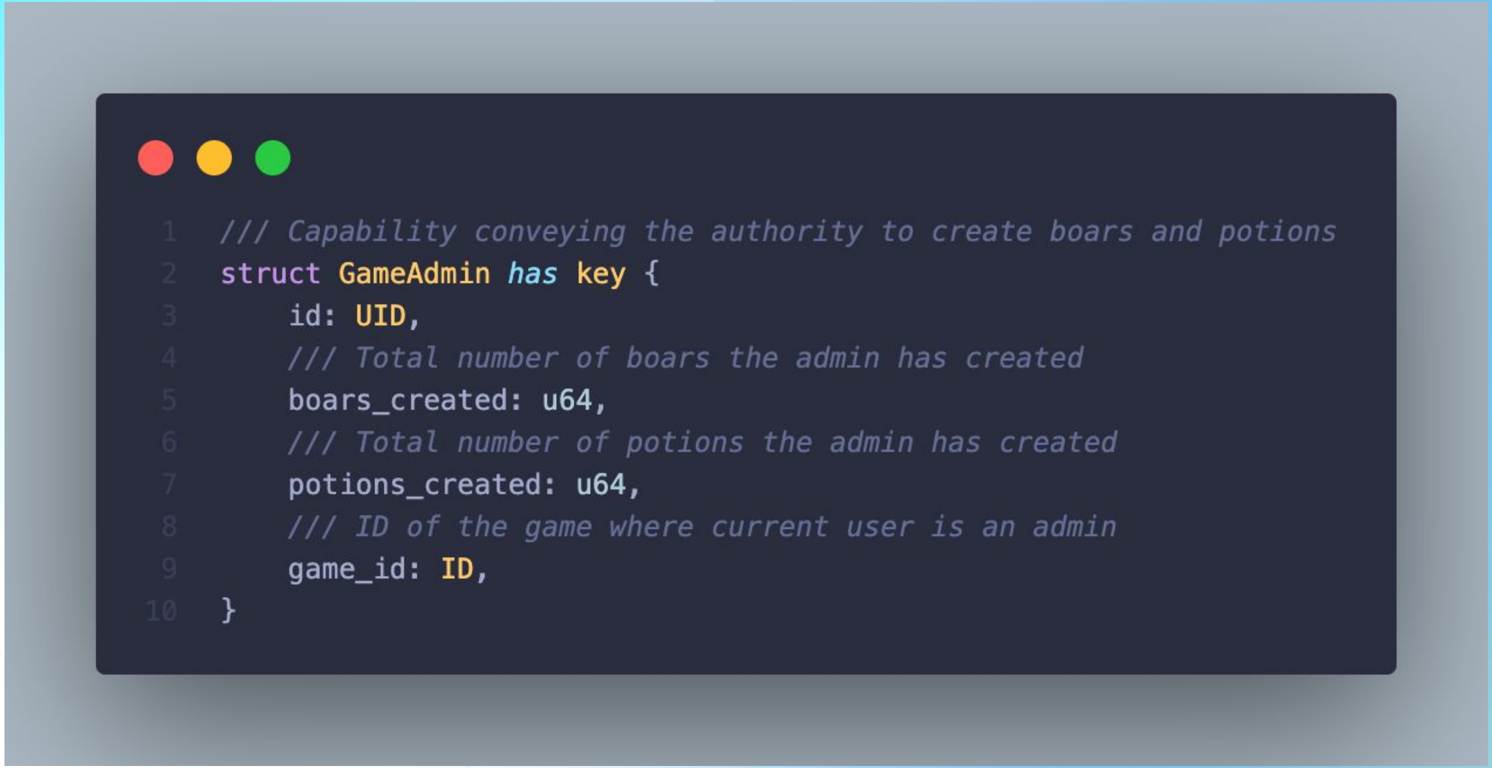
```
/// For healing wounded heroes
struct Potion has key, store {
    id: UID,
    /// Effectiveness of the potion
    potency: u64,
    /// An ID of the game
    game_id: ID,
}
```



```
/// A creature that the hero can slay to level up
struct Boar has key {
    id: UID,
    /// Hit points before the boar is slain
    hp: u64,
    /// Strength of this particular boar
    strength: u64,
    /// An ID of the game
    game_id: ID,
}
```



```
/// An immutable object that contains information about the
/// game admin. Created only once in the module initializer,
/// hence it cannot be recreated or falsified.
struct GameInfo has key {
    id: UID,
    admin: address
}
```



```
/// Capability conveying the authority to create boars and potions
struct GameAdmin has key {
    id: UID,
    /// Total number of boars the admin has created
    boars_created: u64,
    /// Total number of potions the admin has created
    potions_created: u64,
    /// ID of the game where current user is an admin
    game_id: ID,
}
```

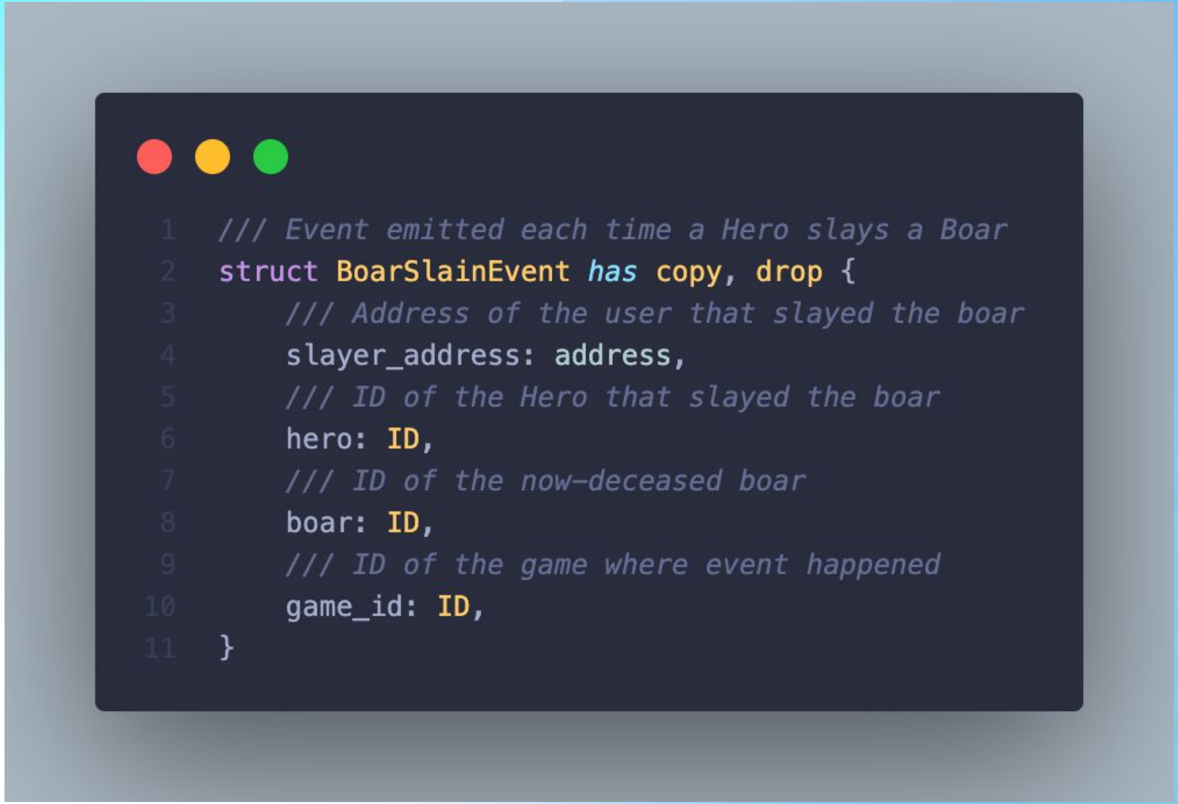
Events

sui

Wait... what are events?

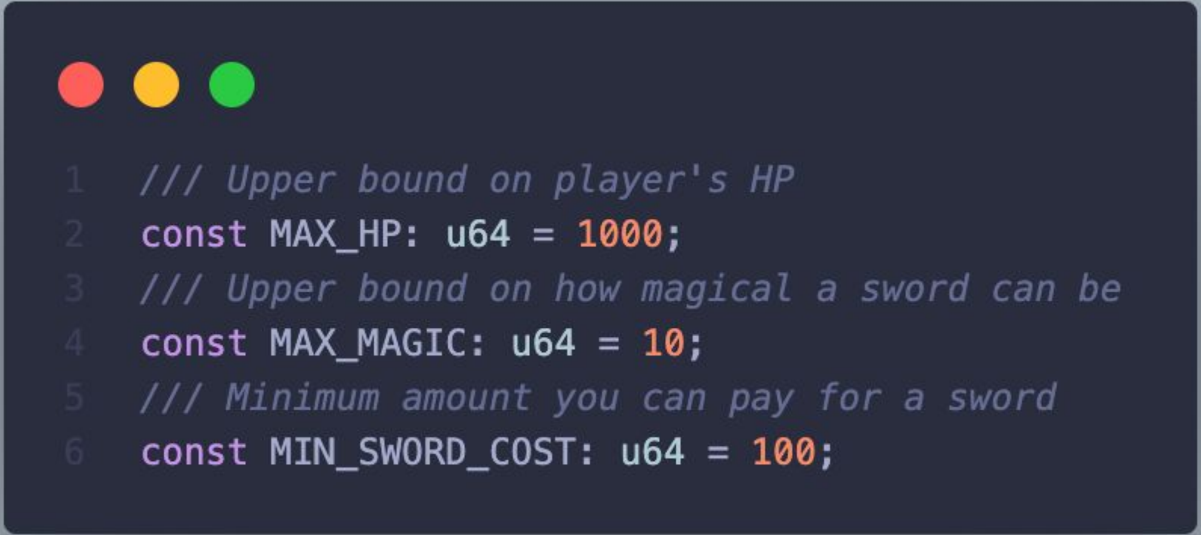
sui

🗼 🔉 ⛓️

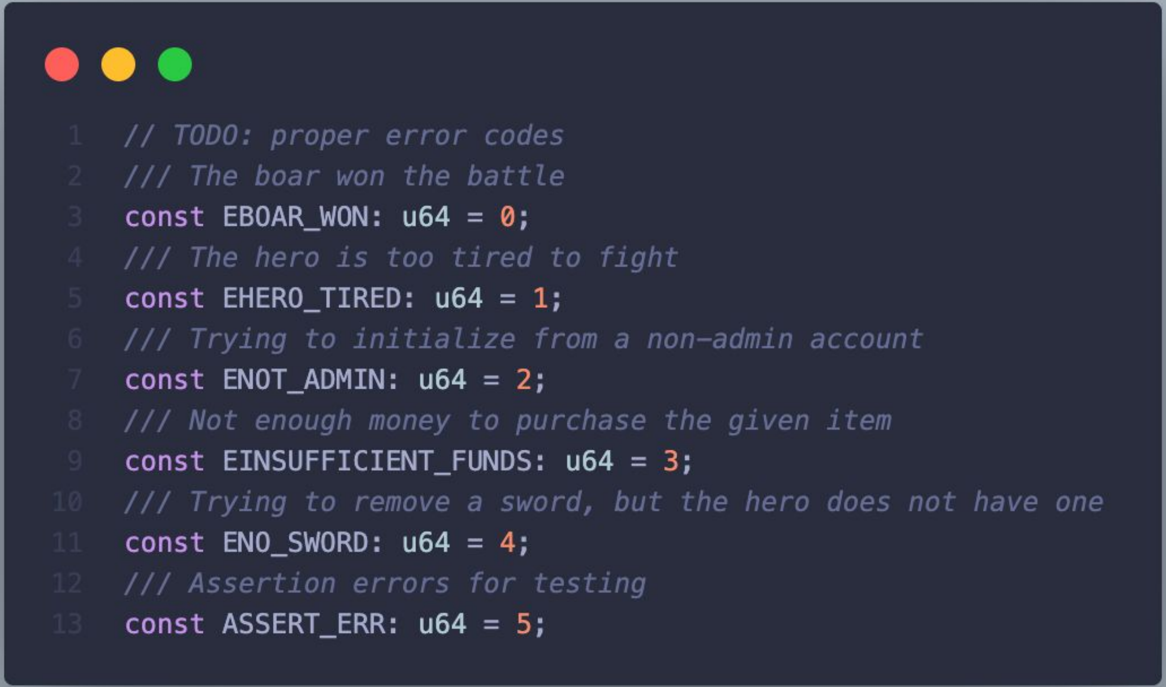


```
/// Event emitted each time a Hero slays a Boar
struct BoarSlainEvent has copy, drop {
    /// Address of the user that slayed the boar
    slayer_address: address,
    /// ID of the Hero that slayed the boar
    hero: ID,
    /// ID of the now-deceased boar
    boar: ID,
    /// ID of the game where event happened
    game_id: ID,
}
```

sui



```
/// Upper bound on player's HP
const MAX_HP: u64 = 1000;
/// Upper bound on how magical a sword can be
const MAX_MAGIC: u64 = 10;
/// Minimum amount you can pay for a sword
const MIN_SWORD_COST: u64 = 100;
```

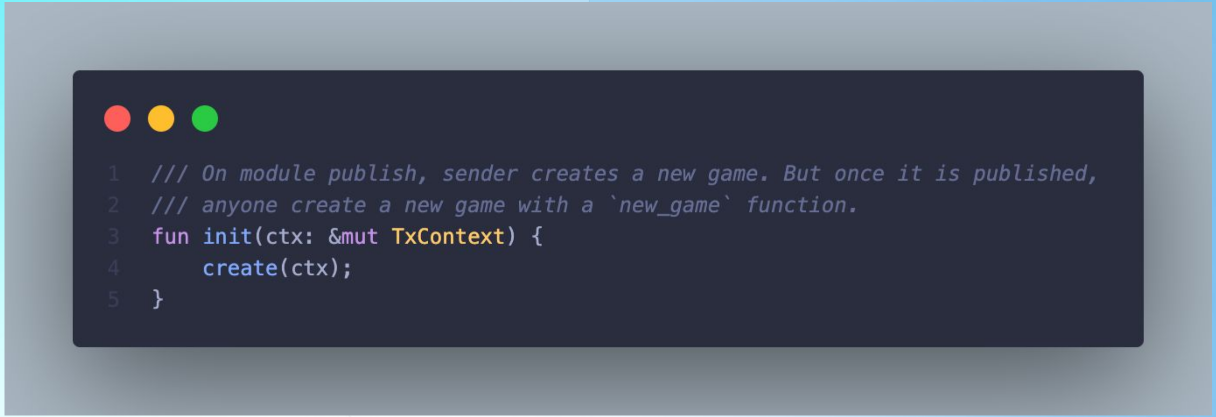


```
// TODO: proper error codes
/// The boar won the battle
const EBOAR_WON: u64 = 0;
/// The hero is too tired to fight
const EHERO_TIRED: u64 = 1;
/// Trying to initialize from a non-admin account
const ENOT_ADMIN: u64 = 2;
/// Not enough money to purchase the given item
const EINSUFFICIENT_FUNDS: u64 = 3;
/// Trying to remove a sword, but the hero does not have one
const ENO_SWORD: u64 = 4;
/// Assertion errors for testing
const ASSERT_ERR: u64 = 5;
```

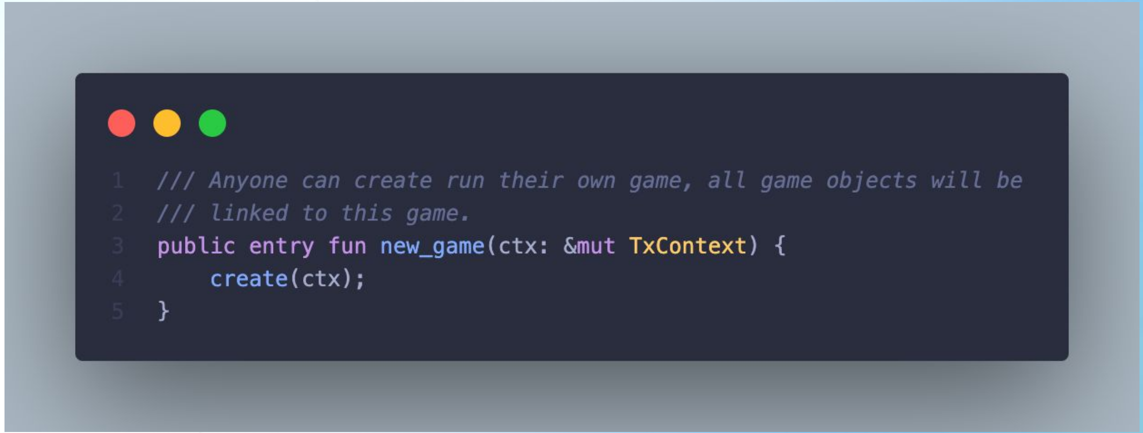
sui

Initialization

sui

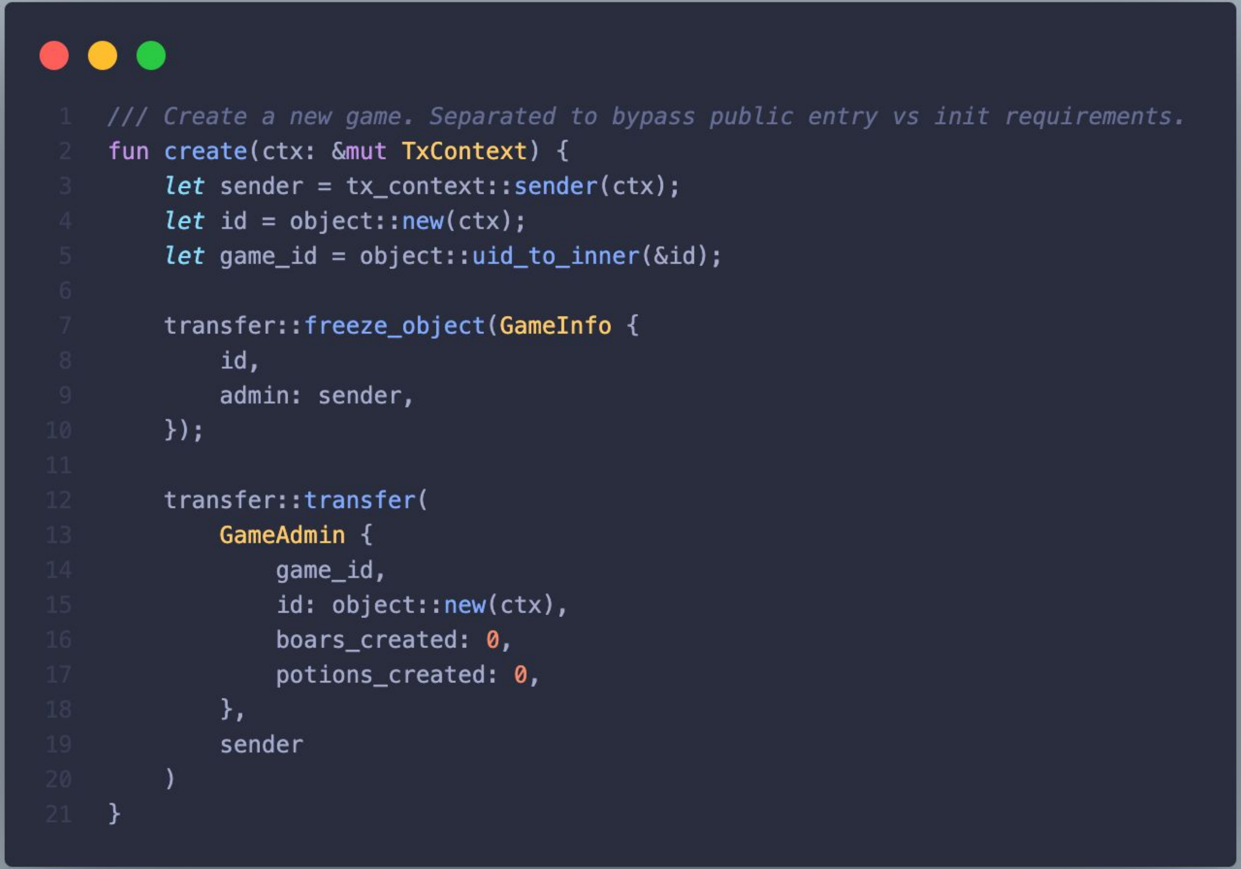


```
/// On module publish, sender creates a new game. But once it is published,
/// anyone create a new game with a `new_game` function.
fun init(ctx: &mut TxContext) {
    create(ctx);
}
```



```
/// Anyone can create run their own game, all game objects will be
/// linked to this game.
public entry fun new_game(ctx: &mut TxContext) {
    create(ctx);
}
```

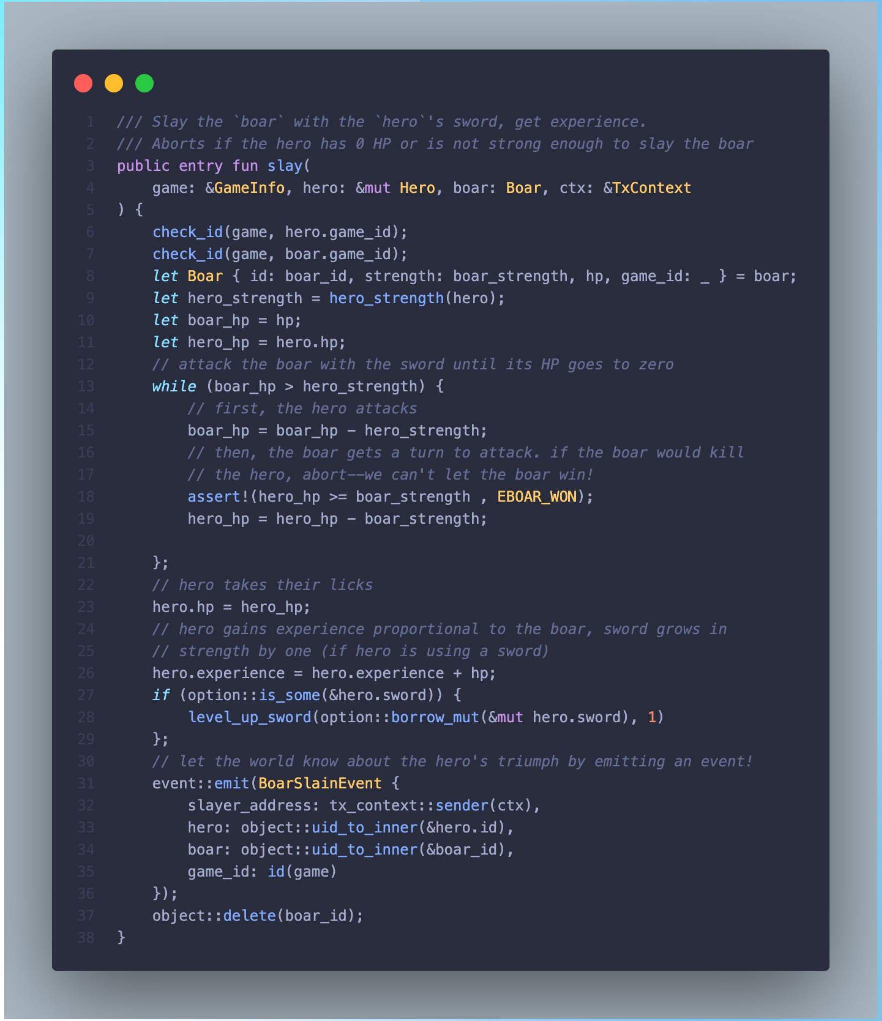
sui



```
/// Create a new game. Separated to bypass public entry vs init requirements.
fun create(ctx: &mut TxContext) {
    let sender = tx_context::sender(ctx);
    let id = object::new(ctx);
    let game_id = object::uid_to_inner(&id);

    transfer::freeze_object(GameInfo {
        id,
        admin: sender,
    });

    transfer::transfer(
        GameAdmin {
            game_id,
            id: object::new(ctx),
            boars_created: 0,
            potions_created: 0,
        },
        sender
    )
}
```

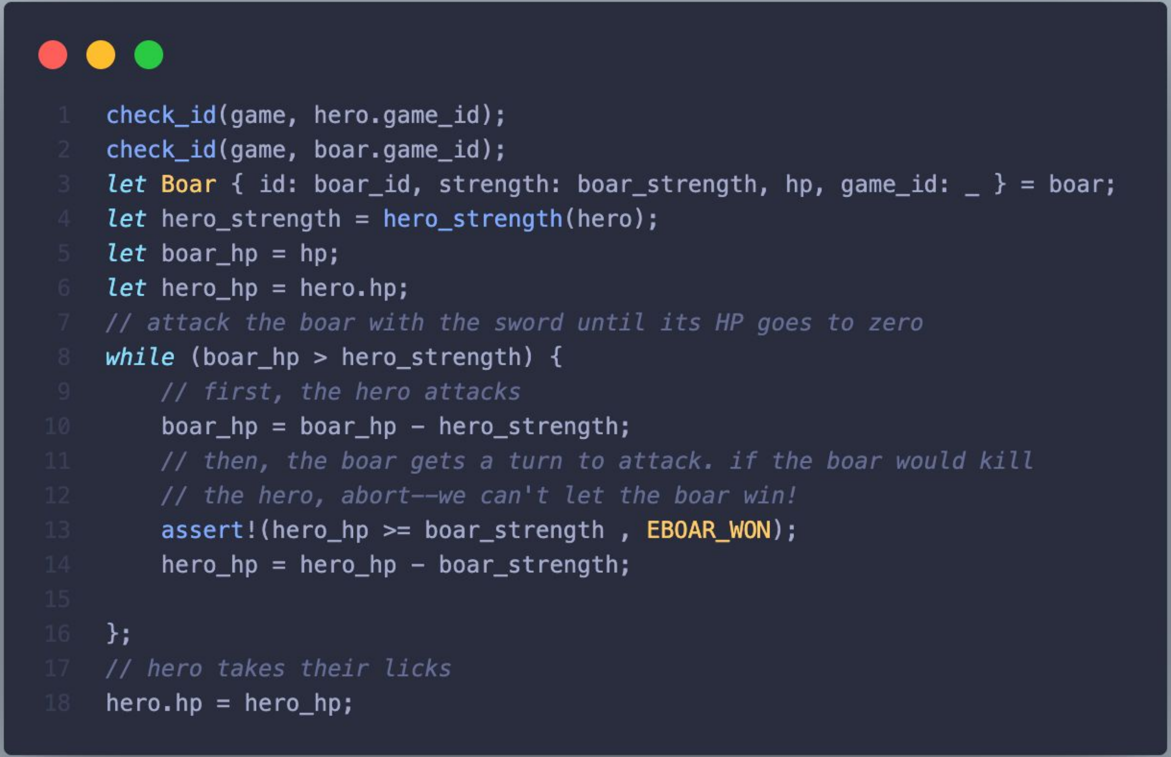
Gameplay

sui



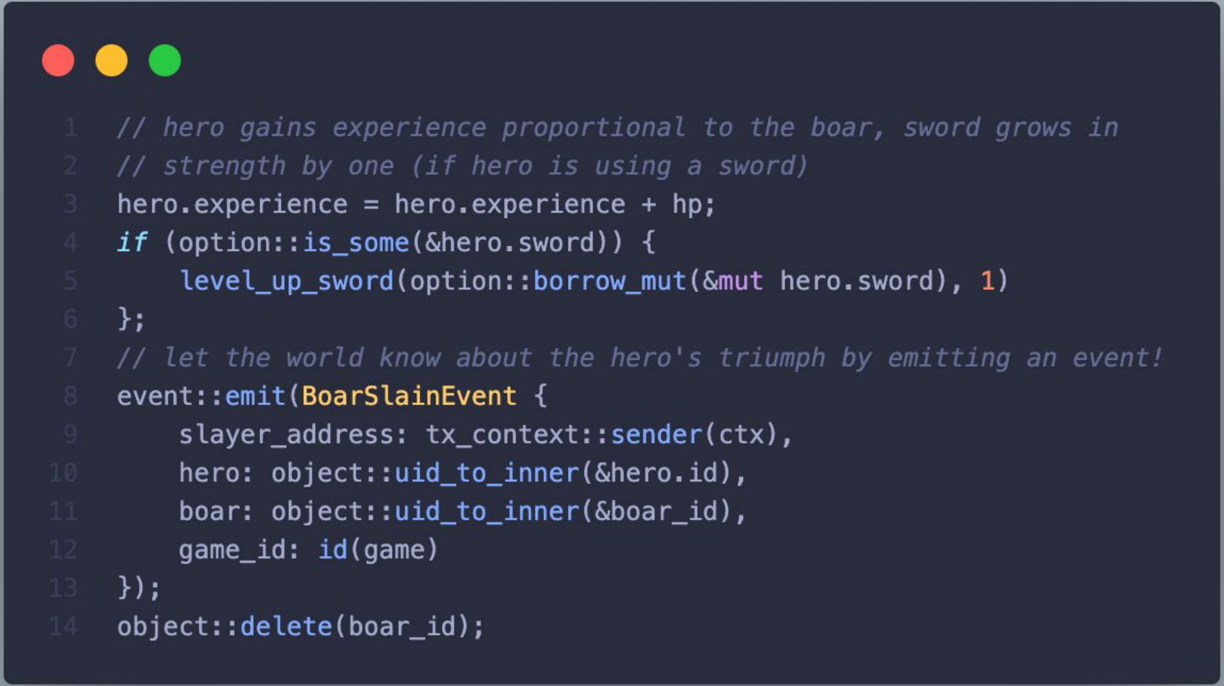
```
/// Slay the `boar` with the `hero`'s sword, get experience.
/// Aborts if the hero has 0 HP or is not strong enough to slay the boar
public entry fun slay(
    game: &GameInfo, hero: &mut Hero, boar: Boar, ctx: &TxContext
) {
    check_id(game, hero.game_id);
    check_id(game, boar.game_id);
    let Boar { id: boar_id, strength: boar_strength, hp, game_id: _ } = boar;
    let hero_strength = hero_strength(hero);
    let boar_hp = hp;
    let hero_hp = hero.hp;
    // attack the boar with the sword until its HP goes to zero
    while (boar_hp > hero_strength) {
        // first, the hero attacks
        boar_hp = boar_hp - hero_strength;
        // then, the boar gets a turn to attack. if the boar would kill
        // the hero, abort--we can't let the boar win!
        assert!(hero_hp >= boar_strength , EBOAR_WON);
        hero_hp = hero_hp - boar_strength;

    };
    // hero takes their licks
    hero.hp = hero_hp;
    // hero gains experience proportional to the boar, sword grows in
    // strength by one (if hero is using a sword)
    hero.experience = hero.experience + hp;
    if (option::is_some(&hero.sword)) {
        level_up_sword(option::borrow_mut(&mut hero.sword), 1)
    };
    // let the world know about the hero's triumph by emitting an event!
    event::emit(BoarSlainEvent {
        slayer_address: tx_context::sender(ctx),
        hero: object::uid_to_inner(&hero.id),
        boar: object::uid_to_inner(&boar_id),
        game_id: id(game)
    });
    object::delete(boar_id);
}
```

sui



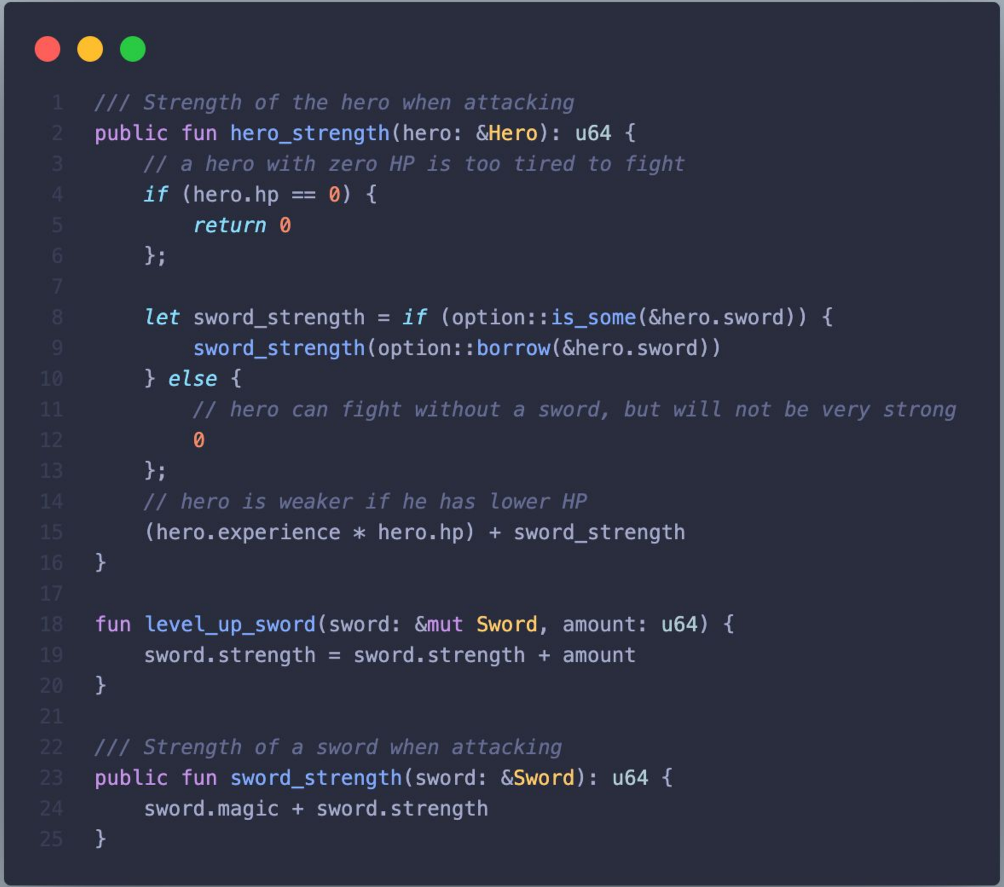
```
check_id(game, hero.game_id);
check_id(game, boar.game_id);
let Boar { id: boar_id, strength: boar_strength, hp, game_id: _ } = boar;
let hero_strength = hero_strength(hero);
let boar_hp = hp;
let hero_hp = hero.hp;
// attack the boar with the sword until its HP goes to zero
while (boar_hp > hero_strength) {
    // first, the hero attacks
    boar_hp = boar_hp - hero_strength;
    // then, the boar gets a turn to attack. if the boar would kill
    // the hero, abort--we can't let the boar win!
    assert!(hero_hp >= boar_strength , EBOAR_WON);
    hero_hp = hero_hp - boar_strength;

};
// hero takes their licks
hero.hp = hero_hp;
```

sui



```
// hero gains experience proportional to the boar, sword grows in
// strength by one (if hero is using a sword)
hero.experience = hero.experience + hp;
if (option::is_some(&hero.sword)) {
    level_up_sword(option::borrow_mut(&mut hero.sword), 1)
};
// let the world know about the hero's triumph by emitting an event!
event::emit(BoarSlainEvent {
    slayer_address: tx_context::sender(ctx),
    hero: object::uid_to_inner(&hero.id),
    boar: object::uid_to_inner(&boar_id),
    game_id: id(game)
});
object::delete(boar_id);
```

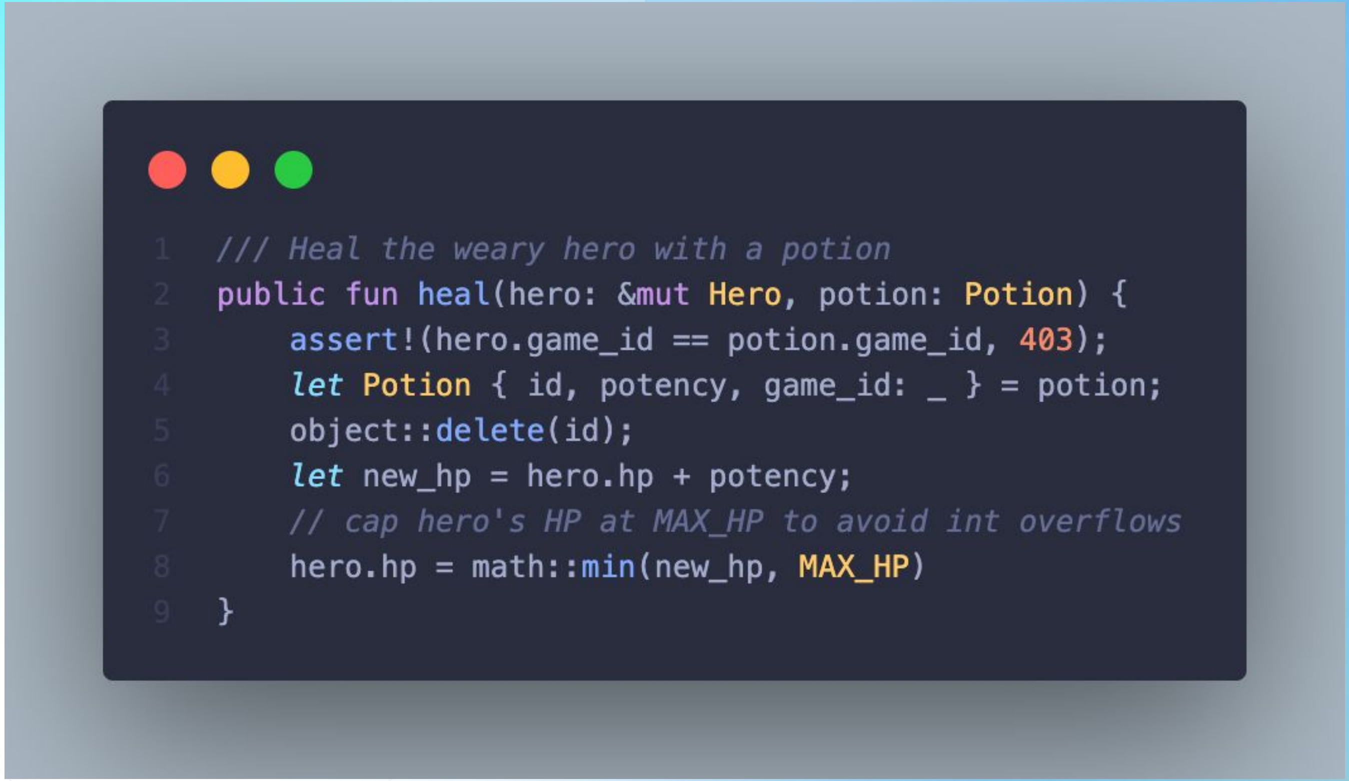
sui



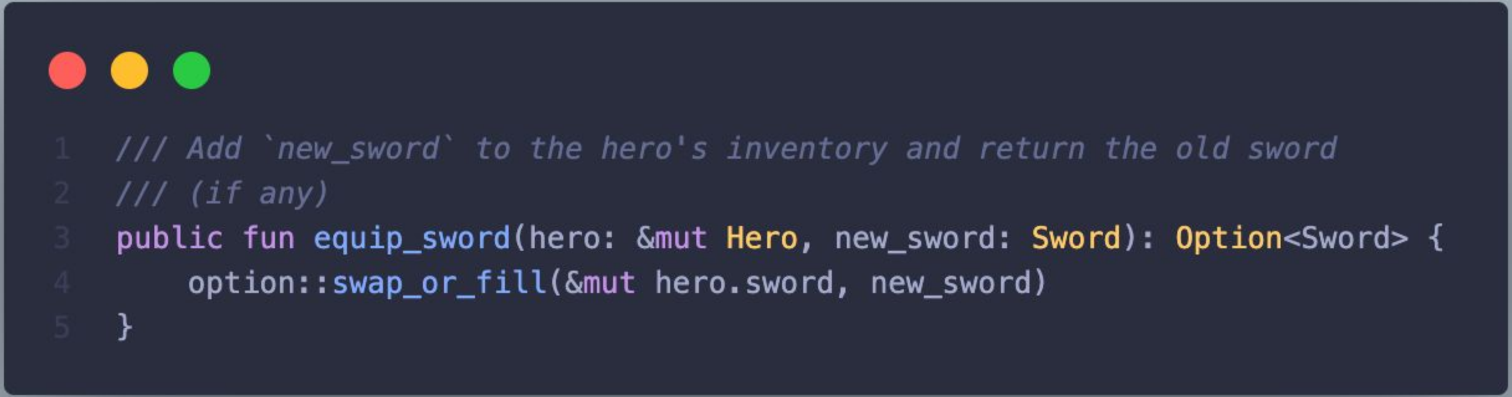
```
/// Strength of the hero when attacking
public fun hero_strength(hero: &Hero): u64 {
    // a hero with zero HP is too tired to fight
    if (hero.hp == 0) {
        return 0
    };

    let sword_strength = if (option::is_some(&hero.sword)) {
        sword_strength(option::borrow(&hero.sword))
    } else {
        // hero can fight without a sword, but will not be very strong
        0
    };
    // hero is weaker if he has lower HP
    (hero.experience * hero.hp) + sword_strength
}

fun level_up_sword(sword: &mut Sword, amount: u64) {
    sword.strength = sword.strength + amount
}

/// Strength of a sword when attacking
public fun sword_strength(sword: &Sword): u64 {
    sword.magic + sword.strength
}
```

sui

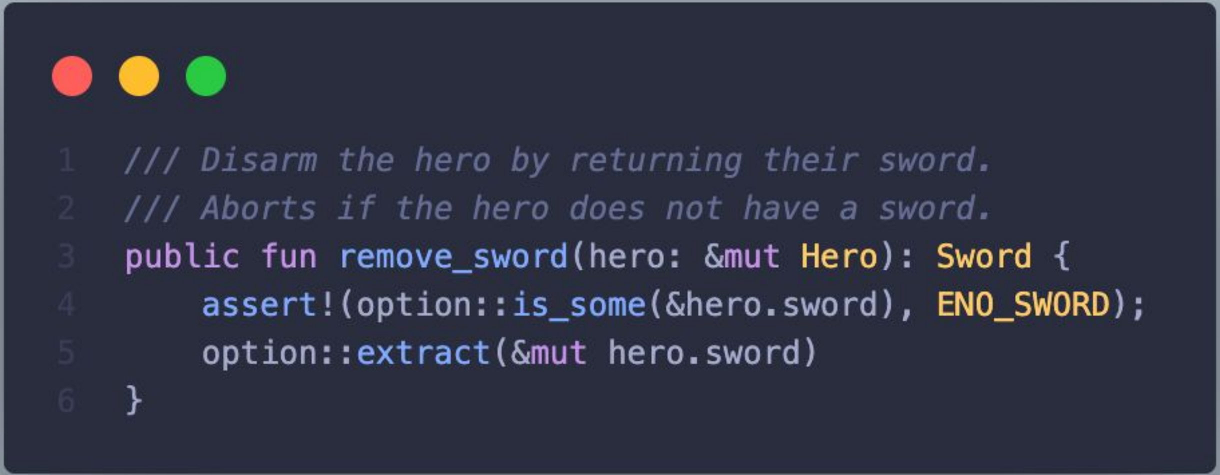
Inventory



```
/// Heal the weary hero with a potion
public fun heal(hero: &mut Hero, potion: Potion) {
    assert!(hero.game_id == potion.game_id, 403);
    let Potion { id, potency, game_id: _ } = potion;
    object::delete(id);
    let new_hp = hero.hp + potency;
    // cap hero's HP at MAX_HP to avoid int overflows
    hero.hp = math::min(new_hp, MAX_HP)
}
```



```
/// Add `new_sword` to the hero's inventory and return the old sword
/// (if any)
public fun equip_sword(hero: &mut Hero, new_sword: Sword): Option<Sword> {
    option::swap_or_fill(&mut hero.sword, new_sword)
}
```

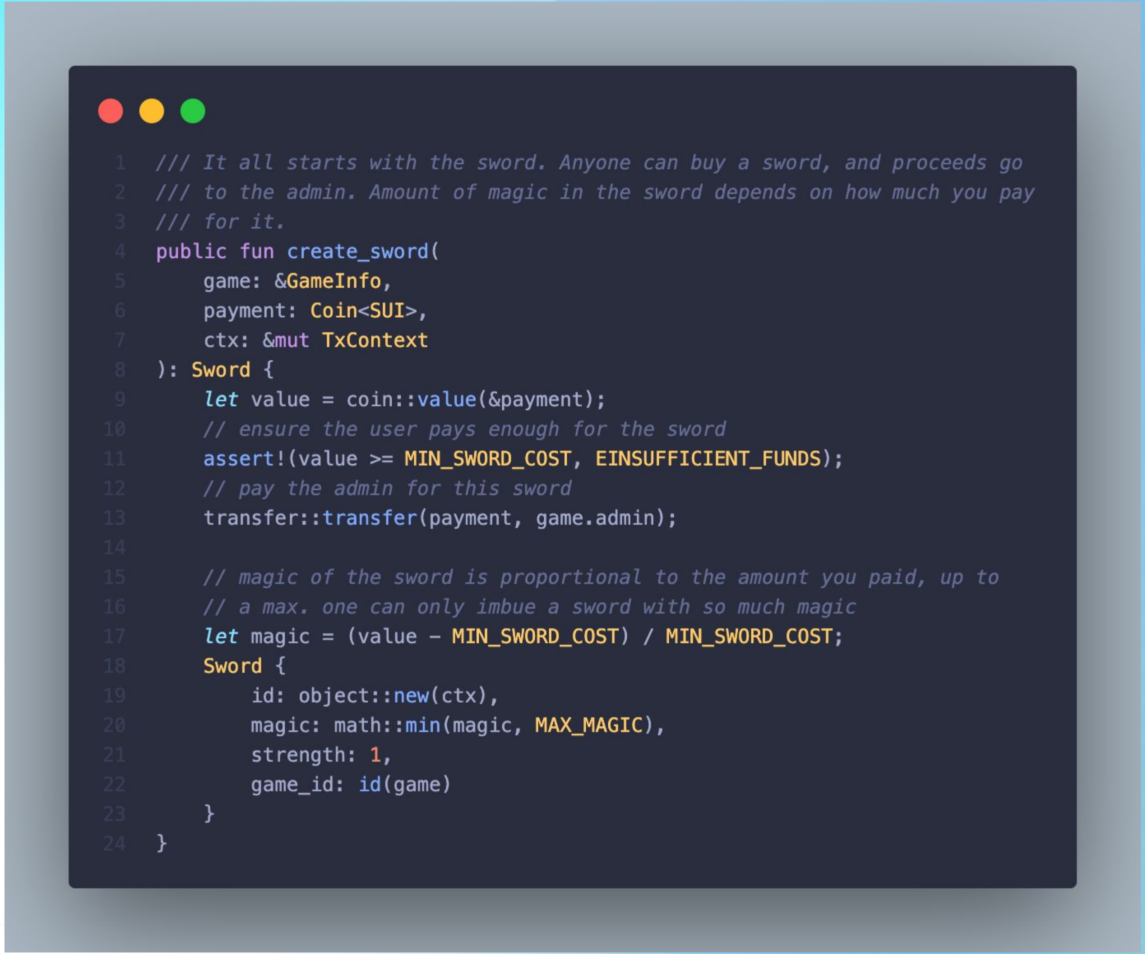


```
/// Disarm the hero by returning their sword.
/// Aborts if the hero does not have a sword.
public fun remove_sword(hero: &mut Hero): Sword {
    assert!(option::is_some(&hero.sword), ENO_SWORD);
    option::extract(&mut hero.sword)
}
```

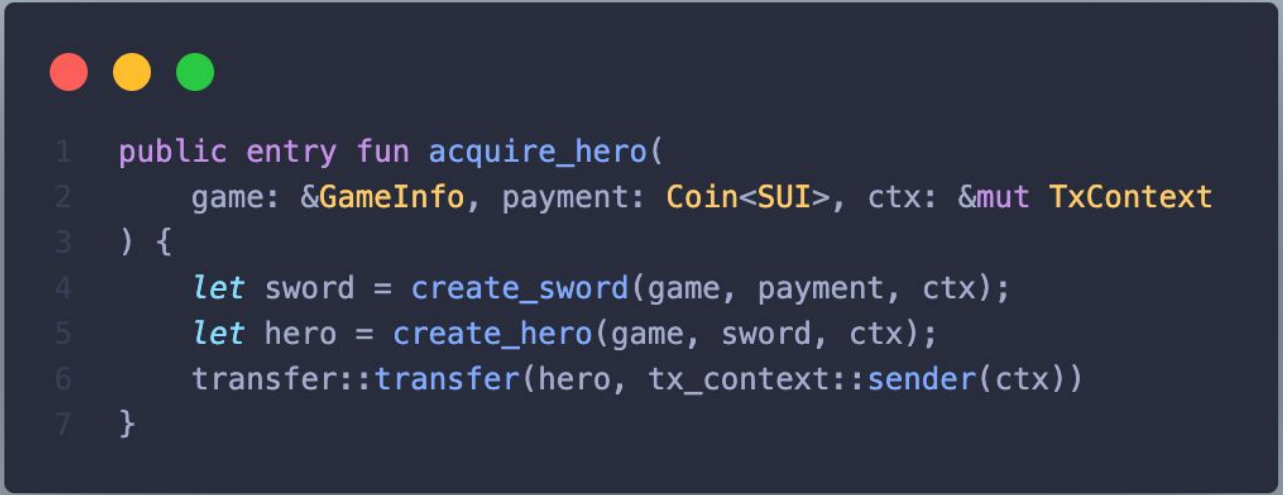
sui

Object Creation

sui

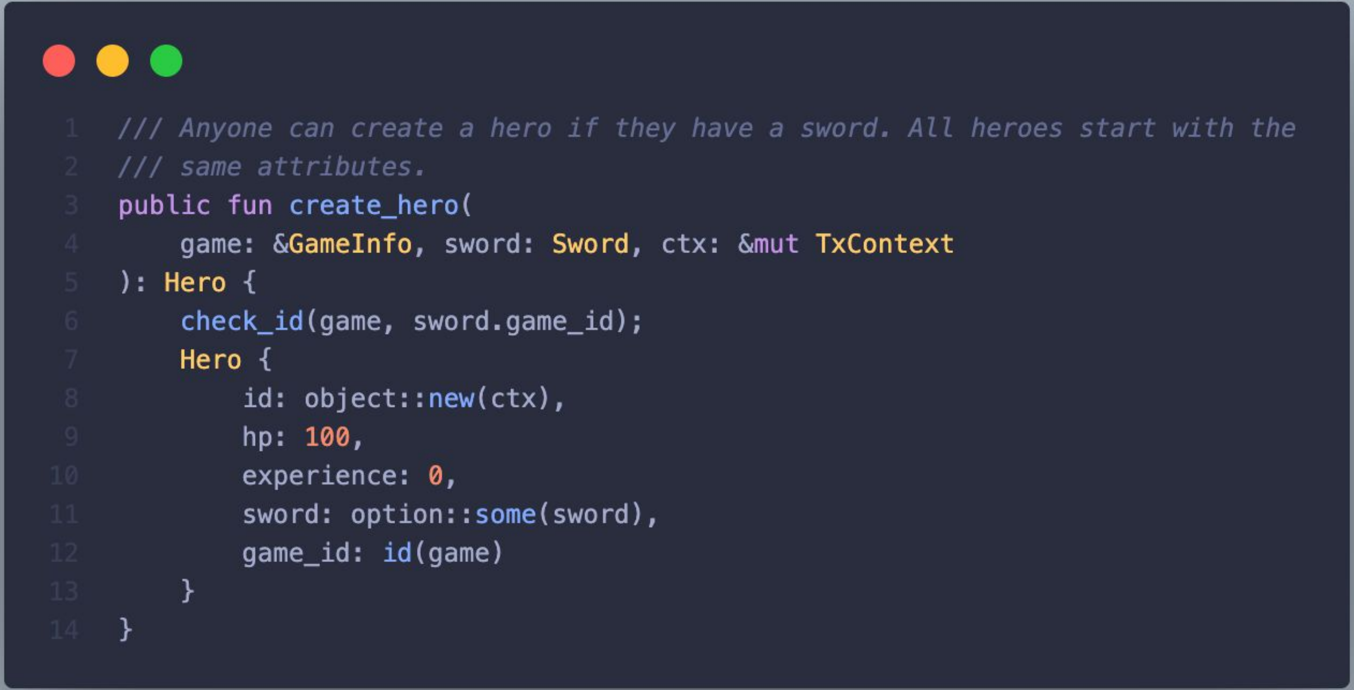


```
/// It all starts with the sword. Anyone can buy a sword, and proceeds go
/// to the admin. Amount of magic in the sword depends on how much you pay
/// for it.
public fun create_sword(
    game: &GameInfo,
    payment: Coin<SUI>,
    ctx: &mut TxContext
): Sword {
    let value = coin::value(&payment);
    // ensure the user pays enough for the sword
    assert!(value >= MIN_SWORD_COST, EINSUFFICIENT_FUNDS);
    // pay the admin for this sword
    transfer::transfer(payment, game.admin);

    // magic of the sword is proportional to the amount you paid, up to
    // a max. one can only imbue a sword with so much magic
    let magic = (value - MIN_SWORD_COST) / MIN_SWORD_COST;
    Sword {
        id: object::new(ctx),
        magic: math::min(magic, MAX_MAGIC),
        strength: 1,
        game_id: id(game)
    }
}
```



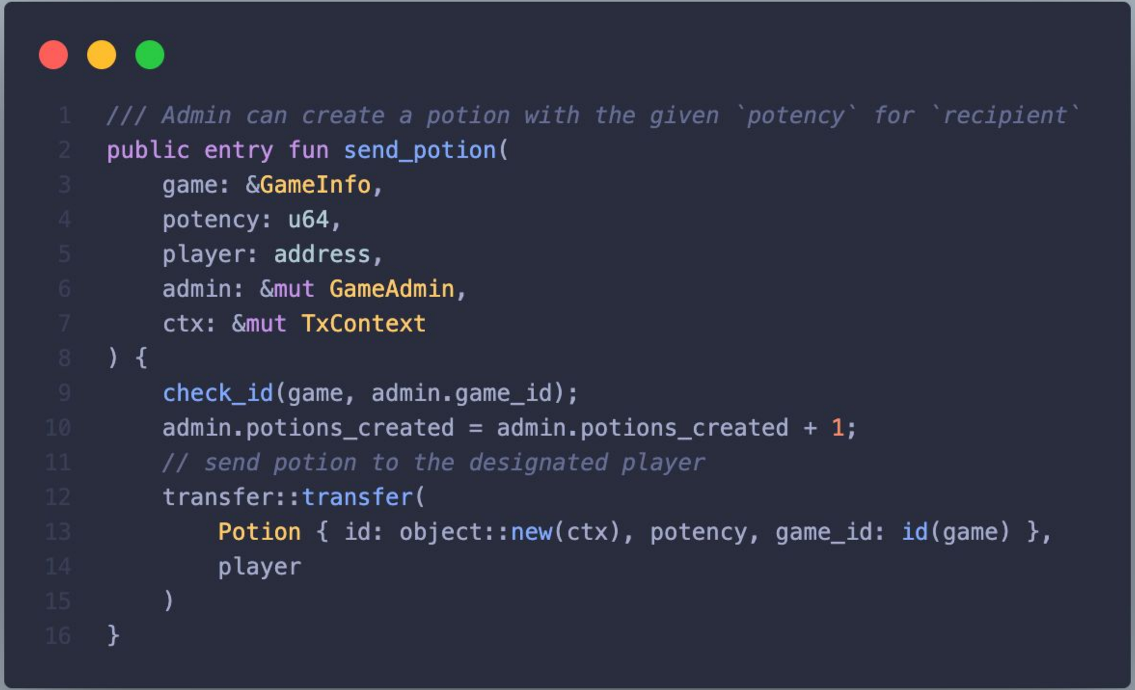
```
public entry fun acquire_hero(
    game: &GameInfo, payment: Coin<SUI>, ctx: &mut TxContext
) {
    let sword = create_sword(game, payment, ctx);
    let hero = create_hero(game, sword, ctx);
    transfer::transfer(hero, tx_context::sender(ctx))
}
```

sui



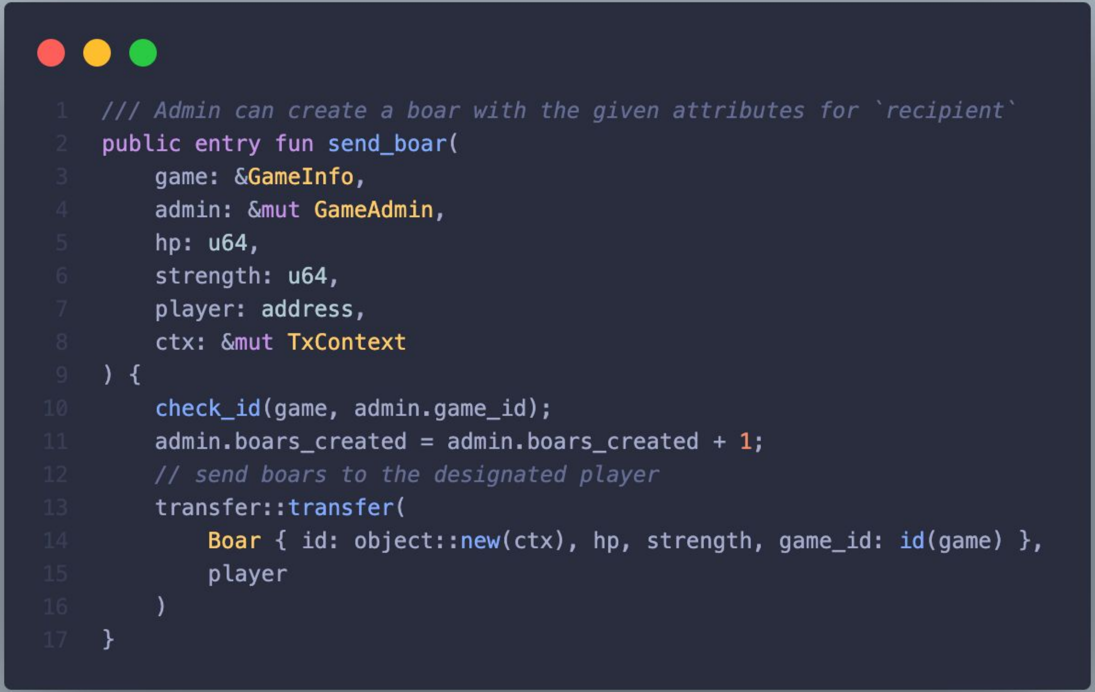
```
/// Anyone can create a hero if they have a sword. All heroes start with the
/// same attributes.
public fun create_hero(
    game: &GameInfo, sword: Sword, ctx: &mut TxContext
): Hero {
    check_id(game, sword.game_id);
    Hero {
        id: object::new(ctx),
        hp: 100,
        experience: 0,
        sword: option::some(sword),
        game_id: id(game)
    }
}
```

sui



```
/// Admin can create a potion with the given `potency` for `recipient`
public entry fun send_potion(
    game: &GameInfo,
    potency: u64,
    player: address,
    admin: &mut GameAdmin,
    ctx: &mut TxContext
) {
    check_id(game, admin.game_id);
    admin.potions_created = admin.potions_created + 1;
    // send potion to the designated player
    transfer::transfer(
        Potion { id: object::new(ctx), potency, game_id: id(game) },
        player
    )
}
```

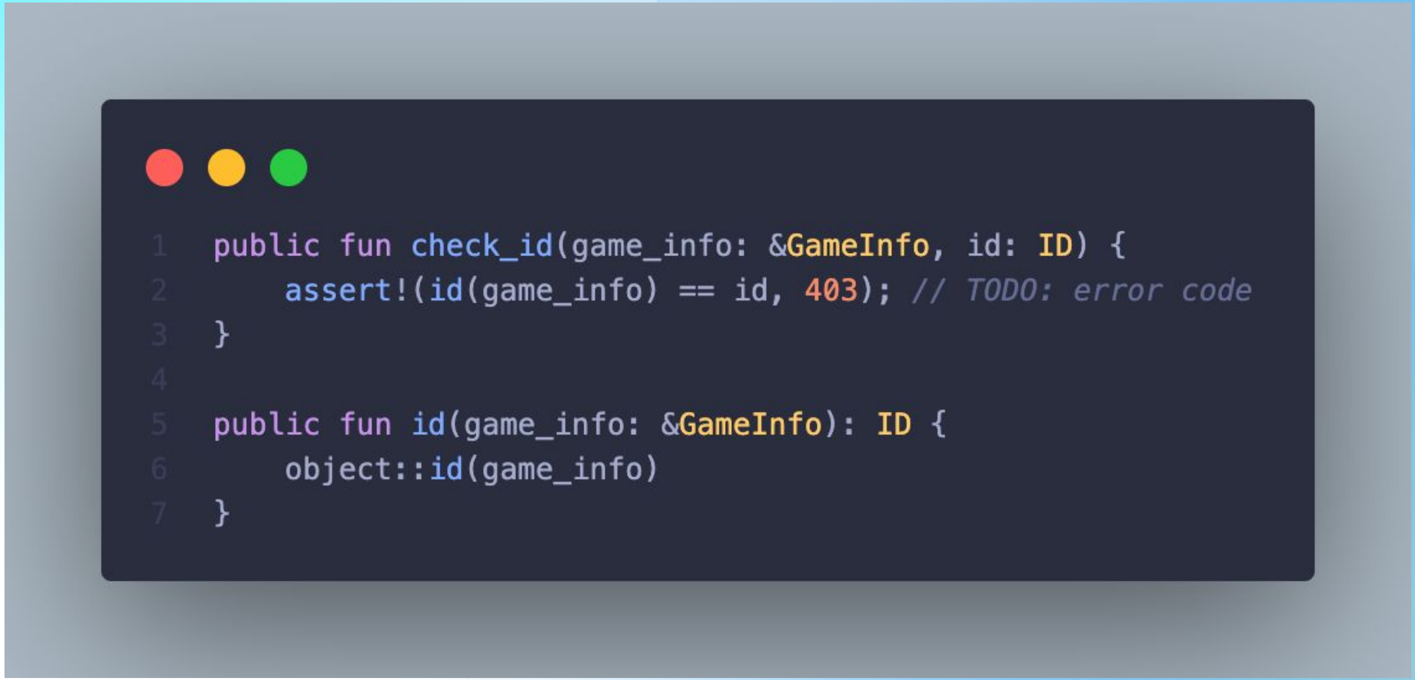
sui



```
/// Admin can create a boar with the given attributes for `recipient`
public entry fun send_boar(
    game: &GameInfo,
    admin: &mut GameAdmin,
    hp: u64,
    strength: u64,
    player: address,
    ctx: &mut TxContext
) {
    check_id(game, admin.game_id);
    admin.boars_created = admin.boars_created + 1;
    // send boars to the designated player
    transfer::transfer(
        Boar { id: object::new(ctx), hp, strength, game_id: id(game) },
        player
    )
}
```

sui

Game Integrity / Links Checks



```
public fun check_id(game_info: &GameInfo, id: ID) {
    assert!(id(game_info) == id, 403); // TODO: error code
}

public fun id(game_info: &GameInfo): ID {
    object::id(game_info)
}
```

Bibliography/ Further Reading

github.com/MystenLabs/sui/blob/main/sui_programmability/examples/games/

sources/hero.move

What's Next!

Next Workshop:
Project Showcase: **On-chain RPG**
(Part 2)

March 24, 2023

sui

Sui Vietnam Builder House!

lu.ma/sui.vietnam

sui

Survey + Questions?

Twitter: @0xShayan

sui