# Sui Move Hackathon

## Week 4 Puzzles

---

## Challenge 1: Event Ticketing System

### Problem Description

Build an event ticketing system where organizers can create events with limited tickets, users can purchase tickets, and tickets can be transferred. This teaches you about supply management, ownership transfers, and tracking sales in Sui Move.

### What You Need to Build

**Core Functionality:**

1. **Event Creation**
    - Organizer creates an event with details
    - Set ticket price and total supply
    - Track tickets sold
    - Event has unique ID, name, date, and venue
2. **Ticket Structure**
    - Each ticket is a unique object
    - Contains event details (name, date, venue)
    - Has ticket number (1 to total_supply)
    - Tracks original buyer
    - Can be transferred to others
3. **Buy Ticket Function**
    - User pays with SUI tokens
    - Check tickets still available
    - Mint new ticket with sequential number
    - Transfer ticket to buyer
    - Track revenue
    - Emit purchase event
4. **Transfer Ticket Function**
    - Owner can transfer ticket to another address
    - Update ownership
    - Emit transfer event

5. **View Functions**
    - Get event details
    - Check tickets remaining
    - Get total revenue
    - Verify ticket authenticity

## Expected Behavior

**Day 1:**

- Alice creates event "Sui Conference 2025"
    - Date: June 1, 2025
    - Venue: "San Francisco Convention Center"
    - Price: 100 SUI per ticket
    - Total supply: 1000 tickets
    - Tickets sold: 0

**Day 2:**

- Bob buys ticket → Receives Ticket #1
- Charlie buys ticket → Receives Ticket #2
- Tickets remaining: 998
- Event revenue: 200 SUI

**Day 3:**

- Bob transfers his ticket to David
- David now owns Ticket #1
- Original buyer still shows as Bob

**Day 4:**

- Event sells out → 1000 tickets sold
- Eve tries to buy → FAILS (sold out)

## Code Template

```
None
module events::ticket_system {
    use std::string::{Self, String};
    use sui::coin::{Self, Coin};
    use sui::balance::{Self, Balance};
    use sui::sui::SUI;
    use sui::event;
    use sui::transfer;
```

```
// ================== Errors ==================
const E_SOLD_OUT: u64 = 1;
const E_INSUFFICIENT_PAYMENT: u64 = 2;
const E_INVALID_SUPPLY: u64 = 3;
const E_INVALID_PRICE: u64 = 4;

// ================== Structs ==================

/// Event information (shared object)
public struct Event has key {
    id: UID,
    organizer: address,
    name: String,
    date: String,
    venue: String,
    ticket_price: u64,
    total_supply: u64,
    tickets_sold: u64,
    revenue: Balance<SUI>,
}

/// Individual ticket (owned object)
public struct Ticket has key, store {
    id: UID,
    event_id: ID,
    event_name: String,
    event_date: String,
    event_venue: String,
    ticket_number: u64,
    original_buyer: address,
}

// ================== Events ==================

public struct EventCreated has copy, drop {
    event_id: ID,
    organizer: address,
    name: String,
```

```
    total_supply: u64,
    ticket_price: u64,
}

public struct TicketPurchased has copy, drop {
    event_id: ID,
    ticket_id: ID,
    buyer: address,
    ticket_number: u64,
    price: u64,
}

public struct TicketTransferred has copy, drop {
    ticket_id: ID,
    from: address,
    to: address,
}

// ==================== Functions ====================

/// Create a new event
public entry fun create_event(
    name: vector<u8>,
    date: vector<u8>,
    venue: vector<u8>,
    ticket_price: u64,
    total_supply: u64,
    ctx: &mut TxContext
) {
    // TODO: Validate price > 0
    // TODO: Validate supply > 0
    // TODO: Convert vectors to Strings
    // TODO: Create Event object with empty balance
    // TODO: Set tickets_sold to 0
    // TODO: Share event object
    // TODO: Emit EventCreated event
}

/// Buy a ticket
```

```
    public entry fun buy_ticket(
        event: &mut Event,
        payment: Coin<SUI>,
        ctx: &mut TxContext
    ) {
        // TODO: Check tickets still available (tickets_sold <
total_supply)
        // TODO: Verify payment amount >= ticket_price
        // TODO: Get buyer address
        // TODO: Increment tickets_sold
        // TODO: Calculate ticket_number (tickets_sold)
        // TODO: Add payment to event revenue
        // TODO: Create Ticket object with event details
        // TODO: Transfer ticket to buyer
        // TODO: Handle excess payment (refund)
        // TODO: Emit TicketPurchased event
    }

    /// Transfer ticket to another address
    public entry fun transfer_ticket(
        ticket: Ticket,
        recipient: address,
        ctx: &TxContext
    ) {
        // TODO: Get ticket ID
        // TODO: Get sender address
        // TODO: Transfer ticket to recipient
        // TODO: Emit TicketTransferred event
    }

    /// Withdraw revenue (organizer only)
    public entry fun withdraw_revenue(
        event: &mut Event,
        ctx: &mut TxContext
    ) {
        // TODO: Verify caller is organizer
        // TODO: Get total revenue amount
        // TODO: Take all revenue from balance
        // TODO: Convert to coin
```

```
        // TODO: Transfer to organizer
    }

    // ================== View Functions ==================

    /// Get event name
    public fun get_event_name(event: &Event): &String {
        &event.name
    }

    /// Get tickets remaining
    public fun get_tickets_remaining(event: &Event): u64 {
        event.total_supply - event.tickets_sold
    }

    /// Get tickets sold
    public fun get_tickets_sold(event: &Event): u64 {
        event.tickets_sold
    }

    /// Check if sold out
    public fun is_sold_out(event: &Event): bool {
        event.tickets_sold >= event.total_supply
    }

    /// Get total revenue
    public fun get_revenue(event: &Event): u64 {
        balance::value(&event.revenue)
    }

    /// Get ticket details
    public fun get_ticket_number(ticket: &Ticket): u64 {
        ticket.ticket_number
    }

    public fun get_ticket_event_name(ticket: &Ticket): &String
{
        &ticket.event_name
    }
```

```
    public fun get_original_buyer(ticket: &Ticket): address {
        ticket.original_buyer
    }

    // ================== Tests ==================

    #[test]
    fun test_create_event_and_buy_ticket() {
        use sui::test_scenario;

        let organizer = @0xA;
        let buyer = @0xB;
        let mut scenario = test_scenario::begin(organizer);

        // Create event
        {
            create_event(
                b"Sui Conference",
                b"2025-06-01",
                b"San Francisco",
                100,
                1000,
                test_scenario::ctx(&mut scenario)
            );
        };

        // Buy ticket
        test_scenario::next_tx(&mut scenario, buyer);
        {
            let mut event =
test_scenario::take_shared<Event>(&scenario);
            let payment = coin::mint_for_testing<SUI>(100,
test_scenario::ctx(&mut scenario));

            buy_ticket(&mut event, payment,
test_scenario::ctx(&mut scenario));

            assert!(get_tickets_sold(&event) == 1, 0);
```

```
            assert!(get_tickets_remaining(&event) == 999, 1);

            test_scenario::return_shared(event);
        };

        // Check buyer received ticket
        test_scenario::next_tx(&mut scenario, buyer);
        {
            let ticket =
test_scenario::take_from_sender<Ticket>(&scenario);
            assert!(get_ticket_number(&ticket) == 1, 0);
            assert!(get_original_buyer(&ticket) == buyer, 1);
            test_scenario::return_to_sender(&scenario,
ticket);
        };

        test_scenario::end(scenario);
    }

    #[test]
    fun test_transfer_ticket() {
        // TODO: Test buying and transferring ticket
    }

    #[test]
    #[expected_failure(abort_code = E_SOLD_OUT)]
    fun test_cannot_buy_when_sold_out() {
        // TODO: Test buying when all tickets sold
    }

    #[test]
    #[expected_failure(abort_code = E_INSUFFICIENT_PAYMENT)]
    fun test_insufficient_payment_fails() {
        // TODO: Test buying with insufficient payment
    }

    #[test]
    fun test_withdraw_revenue() {
        // TODO: Test organizer withdrawing revenue
```

```
    }
}
```

### Move.toml Template

```
None
[package]
name = "ticket_system"
version = "0.0.1"
edition = "2024.beta"

[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir
= "crates/sui-framework/packages/sui-framework", rev =
"framework/testnet" }

[addresses]
events = "0x0"
```

### Success Criteria

- ✅ Create event with ticket details
- ✅ Buy tickets with SUI payment
- ✅ Track tickets sold and remaining
- ✅ Prevent overselling (sold out check)
- ✅ Transfer tickets between users
- ✅ Organizer can withdraw revenue
- ✅ Handle excess payment (refunds)
- ✅ Events emitted for all actions
- ✅ All tests pass

### Technical Hints

- Use shared Event object so anyone can buy tickets
- Each Ticket is owned object with `key + store`
- Sequential ticket numbers: use `tickets_sold + 1`
- Store revenue in `Balance<SUI>` within Event
- Use `coin::value()` to check payment amount
- Split coin for refunds if payment > ticket_price
- Transfer Ticket with `transfer::public_transfer()`

---

# Challenge 2: Simple Marketplace

## Problem Description

Create a basic marketplace where users can list items for sale, buy items with SUI tokens, and delist their items. This teaches you about escrow patterns, coin handling, and ownership transfer.

## What You Need to Build

**Core Functionality:**

1. **Marketplace Structure**
    - Shared marketplace object
    - Track all active listings
    - Store marketplace fee percentage
2. **List Item Function**
    - User lists an item for sale with price
    - Item transferred to marketplace (escrow)
    - Listing includes seller address, price, description
    - Emit listing event
3. **Buy Item Function**
    - Buyer pays the asking price in SUI
    - Marketplace takes small fee (e.g., 2%)
    - Seller receives payment minus fee
    - Item transferred to buyer
    - Listing removed
    - Emit purchase event
4. **Delist Item Function**
    - Seller can cancel listing
    - Item returned to seller
    - No payment exchanged
    - Emit delist event
5. **View Functions**
    - Get all active listings
    - Get listing count
    - Check if item is listed
    - Get listing details

## Expected Behavior

**Day 1:**

- Admin creates marketplace with 2% fee
- Alice lists "Cool NFT" for 1000 SUI
- Item held in escrow

**Day 2:**

- Bob buys "Cool NFT" for 1000 SUI
- Marketplace takes 20 SUI fee
- Alice receives 980 SUI
- Bob receives the NFT
- Listing removed

**Day 3:**

- Charlie lists "Rare Sword" for 500 SUI
- Charlie changes mind and delists
- Charlie gets item back

**Day 4:**

- Bob tries to delist Alice's old listing → FAILS (not the seller)

## Code Template

```
None
module marketplace::simple_marketplace {
    use sui::coin::{Self, Coin};
    use sui::balance::{Self, Balance};
    use sui::sui::SUI;
    use sui::table::{Self, Table};
    use std::string::{Self, String};
    use sui::event;
    use sui::transfer;

    // ================== Errors ==================
    const E_NOT_SELLER: u64 = 1;
    const E_LISTING_NOT_FOUND: u64 = 2;
    const E_INSUFFICIENT_PAYMENT: u64 = 3;
    const E_INVALID_PRICE: u64 = 4;
    const E_ITEM_ALREADY_LISTED: u64 = 5;

    // ================== Constants ==================
    const MARKETPLACE_FEE_PERCENT: u64 = 2; // 2% fee
```

```
    // ==================== Structs ====================

    /// Generic item that can be sold (for this example,
simple item)
    public struct Item has key, store {
        id: UID,
        name: String,
        description: String,
    }

    /// Listing information
    public struct Listing has store {
        item_id: ID,
        seller: address,
        price: u64,
        item: Item, // Item held in escrow
    }

    /// Shared marketplace
    public struct Marketplace has key {
        id: UID,
        listings: Table<ID, Listing>, // item_id -> Listing
        fee_balance: Balance<SUI>,
    }

    // ==================== Events ====================

    public struct MarketplaceCreated has copy, drop {
        marketplace_id: ID,
    }

    public struct ItemListed has copy, drop {
        item_id: ID,
        seller: address,
        price: u64,
    }

    public struct ItemSold has copy, drop {
```

```
    item_id: ID,
    seller: address,
    buyer: address,
    price: u64,
    fee: u64,
}

public struct ItemDelisted has copy, drop {
    item_id: ID,
    seller: address,
}

// ==================== Functions ====================

/// Initialize marketplace
fun init(ctx: &mut TxContext) {
    // TODO: Create Marketplace object
    // TODO: Share marketplace
    // TODO: Emit MarketplaceCreated event
}

/// List an item for sale
public entry fun list_item(
    marketplace: &mut Marketplace,
    item: Item,
    price: u64,
    ctx: &TxContext
) {
    // TODO: Validate price > 0
    // TODO: Get item ID
    // TODO: Check item not already listed
    // TODO: Get seller address
    // TODO: Create Listing with item in escrow
    // TODO: Add to listings table
    // TODO: Emit ItemListed event
}

/// Buy a listed item
public entry fun buy_item(
```

```
        marketplace: &mut Marketplace,
        item_id: ID,
        payment: Coin<SUI>,
        ctx: &mut TxContext
    ) {
        // TODO: Check listing exists
        // TODO: Remove listing from table
        // TODO: Verify payment amount >= price
        // TODO: Calculate fee (price * 2 / 100)
        // TODO: Calculate seller payment (price - fee)
        // TODO: Split payment coin into fee and seller
portions
        // TODO: Add fee to marketplace balance
        // TODO: Create coin for seller payment
        // TODO: Transfer seller payment
        // TODO: Transfer item to buyer
        // TODO: Handle excess payment (refund to buyer)
        // TODO: Emit ItemSold event
    }

    /// Delist an item (seller only)
    public entry fun delist_item(
        marketplace: &mut Marketplace,
        item_id: ID,
        ctx: &TxContext
    ) {
        // TODO: Check listing exists
        // TODO: Remove listing from table
        // TODO: Verify caller is seller
        // TODO: Extract item from listing
        // TODO: Return item to seller
        // TODO: Emit ItemDelisted event
    }

    // ==================== Helper Functions
====================

    /// Create an item (for testing/demo purposes)
    public entry fun create_item(
```

```
        name: vector<u8>,
        description: vector<u8>,
        ctx: &mut TxContext
    ) {
        let item = Item {
            id: object::new(ctx),
            name: string::utf8(name),
            description: string::utf8(description),
        };
        transfer::public_transfer(item,
tx_context::sender(ctx));
    }

    // ================== View Functions ==================

    /// Check if item is listed
    public fun is_listed(marketplace: &Marketplace, item_id:
ID): bool {
        table::contains(&marketplace.listings, item_id)
    }

    /// Get listing price (returns 0 if not found)
    public fun get_listing_price(marketplace: &Marketplace,
item_id: ID): u64 {
        if (!table::contains(&marketplace.listings, item_id))
{
            return 0
        };
        let listing = table::borrow(&marketplace.listings,
item_id);
        listing.price
    }

    /// Get listing seller (aborts if not found)
    public fun get_listing_seller(marketplace: &Marketplace,
item_id: ID): address {
        assert!(table::contains(&marketplace.listings,
item_id), E_LISTING_NOT_FOUND);
```

```
        let listing = table::borrow(&marketplace.listings,
item_id);
        listing.seller
    }

    /// Get total listings count
    public fun get_listing_count(marketplace: &Marketplace):
u64 {
        table::length(&marketplace.listings)
    }

    /// Get marketplace fee balance
    public fun get_fee_balance(marketplace: &Marketplace): u64
{
        balance::value(&marketplace.fee_balance)
    }

    // ==================== Admin Functions ====================

    /// Withdraw collected fees (in real scenario, add admin
cap)
    public entry fun withdraw_fees(
        marketplace: &mut Marketplace,
        ctx: &mut TxContext
    ) {
        let amount = balance::value(&marketplace.fee_balance);
        let fees = balance::split(&mut
marketplace.fee_balance, amount);
        let fees_coin = coin::from_balance(fees, ctx);
        transfer::public_transfer(fees_coin,
tx_context::sender(ctx));
    }

    // ==================== Tests ====================

    #[test_only]
    public fun init_for_testing(ctx: &mut TxContext) {
        init(ctx);
    }
```

```
    #[test]
    fun test_list_and_buy_item() {
        use sui::test_scenario;

        let admin = @0xAD;
        let seller = @0xA;
        let buyer = @0xB;
        let mut scenario = test_scenario::begin(admin);

        // Create marketplace
        {
            init_for_testing(test_scenario::ctx(&mut
scenario));
        };

        // Seller creates item
        test_scenario::next_tx(&mut scenario, seller);
        let item_id = {
            create_item(b"Cool NFT", b"A very cool NFT",
test_scenario::ctx(&mut scenario));
            test_scenario::next_tx(&mut scenario, seller);
            let item =
test_scenario::take_from_sender<Item>(&scenario);
            let id = object::id(&item);
            test_scenario::return_to_sender(&scenario, item);
            id
        };

        // Seller lists item
        test_scenario::next_tx(&mut scenario, seller);
        {
            let mut marketplace =
test_scenario::take_shared<Marketplace>(&scenario);
            let item =
test_scenario::take_from_sender<Item>(&scenario);

            list_item(&mut marketplace, item, 1000,
test_scenario::ctx(&mut scenario));
```

```
            assert!(get_listing_count(&marketplace) == 1, 0);
            assert!(is_listed(&marketplace, item_id), 1);

            test_scenario::return_shared(marketplace);
        };

        // Buyer purchases item
        test_scenario::next_tx(&mut scenario, buyer);
        {
            let mut marketplace =
test_scenario::take_shared<Marketplace>(&scenario);
            let payment = coin::mint_for_testing<SUI>(1000,
test_scenario::ctx(&mut scenario));

            buy_item(&mut marketplace, item_id, payment,
test_scenario::ctx(&mut scenario));
            assert!(get_listing_count(&marketplace) == 0, 0);
            assert!(!is_listed(&marketplace, item_id), 1);

            test_scenario::return_shared(marketplace);
        };

        // Check buyer received item
        test_scenario::next_tx(&mut scenario, buyer);
        {
            let item =
test_scenario::take_from_sender<Item>(&scenario);
            test_scenario::return_to_sender(&scenario, item);
        };

        test_scenario::end(scenario);
    }

    #[test]
    fun test_delist_item() {
        // TODO: Test listing and delisting
    }

    #[test]
```

```
    #[expected_failure(abort_code = E_NOT_SELLER)]
    fun test_only_seller_can_delist() {
        // TODO: Test non-seller cannot delist
    }

    #[test]
    #[expected_failure(abort_code = E_INSUFFICIENT_PAYMENT)]
    fun test_insufficient_payment_fails() {
        // TODO: Test buying with insufficient payment
    }

    #[test]
    fun test_marketplace_fee_collection() {
        // TODO: Test that marketplace collects 2% fee
    }
}
```

### Move.toml Template

```
None
[package]
name = "simple_marketplace"
version = "0.0.1"
edition = "2024.beta"

[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir
= "crates/sui-framework/packages/sui-framework", rev =
"framework/testnet" }

[addresses]
marketplace = "0x0"
```

### Success Criteria

- ✅ Create shared marketplace
- ✅ List items for sale (held in escrow)
- ✅ Buy items with SUI payment
- ✅ Marketplace collects fee (2%)

- ✅ Seller receives payment minus fee
- ✅ Delist items (seller only)
- ✅ Handle excess payment (refund buyer)
- ✅ Events emitted for all actions
- ✅ All tests pass

### Technical Hints

- Use `Table<ID, Listing>` to store active listings
- Store item in Listing struct (escrow pattern)
- Use `coin::split()` to divide payment into fee and seller portions
- Calculate fee: `(price * MARKETPLACE_FEE_PERCENT) / 100`
- Use `balance::join()` to add fees to marketplace balance
- Remove listing from table after purchase or delist
- Test with `coin::mint_for_testing()` to create test SUI

---

# Challenge 3: Lottery System

## Problem Description

Build a simple lottery system where users can buy tickets for a chance to win a prize pool. When the lottery ends, a winner is selected and receives all collected funds. This teaches you about collecting entries, randomness considerations, and prize distribution.

## What You Need to Build

### Core Functionality:

1. **Lottery Structure**
    - Shared lottery object
    - Set ticket price
    - Track all participants (entries)
    - Collect prize pool from ticket sales
    - Set start and end times
    - Track lottery status (active/ended)
2. **Create Lottery Function**
    - Organizer creates lottery with ticket price
    - Set lottery duration
    - Initialize empty prize pool
    - Set status to active
3. **Buy Ticket Function**
    - User pays ticket price in SUI

    - Check lottery is active
    - Add user to participants list
    - User can buy multiple tickets (increases chances)
    - Add payment to prize pool
    - Emit ticket purchase event
4. **Draw Winner Function**
    - Only callable after lottery ends
    - Select winner from participants
    - Transfer entire prize pool to winner
    - Mark lottery as ended
    - Emit winner announcement event
5. **View Functions**
    - Get ticket price
    - Get prize pool amount
    - Get total participants
    - Check if lottery is active
    - Get participant entries count
    - Get time remaining

## Expected Behavior

**Day 1:**

- Alice creates lottery
    - Ticket price: 1 SUI
    - Duration: 7 days
    - Prize pool: 0 SUI
    - Status: Active

**Day 2-6:**

- Bob buys 2 tickets → Pays 2 SUI
- Charlie buys 1 ticket → Pays 1 SUI
- David buys 3 tickets → Pays 3 SUI
- Prize pool: 6 SUI
- Total entries: 6 (Bob=2, Charlie=1, David=3)

**Day 7:**

- Eve tries to buy ticket → FAILS (lottery not ended yet, still accepting)
- Current participants: Bob, Charlie, David

**Day 8:**

- Lottery ends (7 days passed)
- Alice draws winner
- Winner selected: David (had 3 entries, 50% chance)

- David receives 60 SUI
- Status: Ended

**Day 9:**

- Frank tries to buy ticket → FAILS (lottery ended)

## Code Template

```
None
module lottery::simple_lottery {
    use sui::coin::{Self, Coin};
    use sui::balance::{Self, Balance};
    use sui::sui::SUI;
    use sui::clock::{Self, Clock};
    use sui::event;
    use sui::transfer;

    // ================== Errors ==================
    const E_LOTTERY_ENDED: u64 = 1;
    const E_LOTTERY_NOT_ENDED: u64 = 2;
    const E_INSUFFICIENT_PAYMENT: u64 = 3;
    const E_NO_PARTICIPANTS: u64 = 4;
    const E_ALREADY_DRAWN: u64 = 5;
    const E_INVALID_TICKET_PRICE: u64 = 6;

    // ================== Structs ==================

    /// Lottery object
    public struct Lottery has key {
        id: UID,
        organizer: address,
        ticket_price: u64,
        prize_pool: Balance<SUI>,
        participants: vector<address>, // Can contain
duplicates (multiple tickets)
        start_time: u64,
        end_time: u64,
        winner: Option<address>,
        status: bool, // true = active, false = ended
    }
```

```
    // ==================== Events ====================

    public struct LotteryCreated has copy, drop {
        lottery_id: ID,
        organizer: address,
        ticket_price: u64,
        end_time: u64,
    }

    public struct TicketPurchased has copy, drop {
        lottery_id: ID,
        buyer: address,
        tickets_bought: u64,
        amount_paid: u64,
    }

    public struct WinnerDrawn has copy, drop {
        lottery_id: ID,
        winner: address,
        prize_amount: u64,
    }

    // ==================== Functions ====================

    /// Create a new lottery
    public entry fun create_lottery(
        ticket_price: u64,
        duration_ms: u64,
        clock: &Clock,
        ctx: &mut TxContext
    ) {
        // TODO: Validate ticket_price > 0
        // TODO: Get current time
        // TODO: Calculate end_time = current_time +
duration_ms
        // TODO: Create Lottery object
        // TODO: Initialize empty participants vector
        // TODO: Set status to true (active)
        // TODO: Set winner to none
```

```
        // TODO: Share lottery object
        // TODO: Emit LotteryCreated event
    }

    /// Buy lottery ticket(s)
    public entry fun buy_ticket(
        lottery: &mut Lottery,
        payment: Coin<SUI>,
        clock: &Clock,
        ctx: &mut TxContext
    ) {
        let current_time = clock::timestamp_ms(clock);
        let buyer = tx_context::sender(ctx);

        // TODO: Check lottery is still active (status ==
true)
        // TODO: Check current time < end_time
        // TODO: Get payment amount
        // TODO: Calculate number of tickets = payment /
ticket_price
        // TODO: Verify payment is sufficient for at least 1
ticket
        // TODO: Add buyer to participants vector (once for
each ticket)
        // TODO: Add payment to prize pool
        // TODO: Handle excess payment (refund)
        // TODO: Emit TicketPurchased event
    }

    /// Draw winner (callable by anyone after lottery ends)
    public entry fun draw_winner(
        lottery: &mut Lottery,
        clock: &Clock,
        ctx: &mut TxContext
    ) {
        let current_time = clock::timestamp_ms(clock);

        // TODO: Check lottery has ended (current_time >=
end_time)
```

```
        // TODO: Check lottery is still active (not already
drawn)
        // TODO: Check there are participants
        // TODO: Select winner using deterministic method
        //       Use: lottery ID + current time to create
pseudo-random index
        //       Formula: index = (id_bytes + timestamp) %
participants.length
        // TODO: Get winner address from participants
        // TODO: Get prize pool amount
        // TODO: Transfer prize pool to winner
        // TODO: Set lottery status to false (ended)
        // TODO: Set winner in lottery
        // TODO: Emit WinnerDrawn event
    }

    // ================== Helper Functions
==================

    /// Simple deterministic winner selection
    /// In production, use a VRF or commit-reveal scheme
    fun select_winner(lottery: &Lottery, clock: &Clock): u64 {
        let participants_count =
vector::length(&lottery.participants);
        if (participants_count == 0) {
            abort E_NO_PARTICIPANTS
        };

        // Create pseudo-random number from lottery ID and
timestamp
        let lottery_id_bytes = object::id_bytes(lottery);
        let timestamp = clock::timestamp_ms(clock);

        // Simple deterministic selection (not
cryptographically secure)
        // In production, use Sui's randomness API or VRF
        let random_value = *vector::borrow(lottery_id_bytes,
0) as u64 + timestamp;
        random_value % participants_count
```

```
    }

    // ==================== View Functions ====================

    /// Get ticket price
    public fun get_ticket_price(lottery: &Lottery): u64 {
        lottery.ticket_price
    }

    /// Get prize pool amount
    public fun get_prize_pool(lottery: &Lottery): u64 {
        balance::value(&lottery.prize_pool)
    }

    /// Get total number of entries (tickets sold)
    public fun get_total_entries(lottery: &Lottery): u64 {
        vector::length(&lottery.participants)
    }

    /// Check if lottery is active
    public fun is_active(lottery: &Lottery, clock: &Clock):
bool {
        let current_time = clock::timestamp_ms(clock);
        lottery.status && current_time < lottery.end_time
    }

    /// Check if lottery has ended
    public fun has_ended(lottery: &Lottery, clock: &Clock):
bool {
        let current_time = clock::timestamp_ms(clock);
        current_time >= lottery.end_time
    }

    /// Get participant entry count
    public fun get_participant_entries(lottery: &Lottery,
participant: address): u64 {
        let mut count = 0;
        let mut i = 0;
        let len = vector::length(&lottery.participants);
```

```
        while (i < len) {
            if (*vector::borrow(&lottery.participants, i) ==
participant) {
                count = count + 1;
            };
            i = i + 1;
        };

        count
    }

    /// Get winner (returns none if not drawn yet)
    public fun get_winner(lottery: &Lottery): &Option<address>
{
        &lottery.winner
    }

    /// Get time remaining until lottery ends
    public fun get_time_remaining(lottery: &Lottery, clock:
&Clock): u64 {
        let current_time = clock::timestamp_ms(clock);
        if (current_time >= lottery.end_time) {
            0
        } else {
            lottery.end_time - current_time
        }
    }

    // ==================== Tests ====================

    #[test]
    fun test_create_and_buy_ticket() {
        use sui::test_scenario;

        let organizer = @0xA;
        let buyer = @0xB;
        let mut scenario = test_scenario::begin(organizer);
```

```
        // Create lottery
        {
            let clock =
clock::create_for_testing(test_scenario::ctx(&mut scenario));
            create_lottery(
                1, // 1 SUI per ticket
                7 * 24 * 60 * 60 * 1000, // 7 days
                &clock,
                test_scenario::ctx(&mut scenario)
            );
            clock::destroy_for_testing(clock);
        };

        // Buy ticket
        test_scenario::next_tx(&mut scenario, buyer);
        {
            let mut lottery =
test_scenario::take_shared<Lottery>(&scenario);
            let clock =
clock::create_for_testing(test_scenario::ctx(&mut scenario));
            let payment = coin::mint_for_testing<SUI>(20,
test_scenario::ctx(&mut scenario));

            buy_ticket(&mut lottery, payment, &clock,
test_scenario::ctx(&mut scenario));

            assert!(get_total_entries(&lottery) == 2, 0); // 2
tickets
            assert!(get_prize_pool(&lottery) == 20, 1);
            assert!(get_participant_entries(&lottery, buyer)
== 2, 2);

            clock::destroy_for_testing(clock);
            test_scenario::return_shared(lottery);
        };

        test_scenario::end(scenario);
    }
```

```
    #[test]
    fun test_draw_winner() {
        // TODO: Test complete lottery flow with winner
selection
    }

    #[test]
    #[expected_failure(abort_code = E_LOTTERY_ENDED)]
    fun test_cannot_buy_after_end() {
        // TODO: Test buying ticket after lottery ends
    }

    #[test]
    #[expected_failure(abort_code = E_LOTTERY_NOT_ENDED)]
    fun test_cannot_draw_before_end() {
        // TODO: Test drawing winner before lottery ends
    }

    #[test]
    #[expected_failure(abort_code = E_ALREADY_DRAWN)]
    fun test_cannot_draw_twice() {
        // TODO: Test that winner can only be drawn once
    }
}
```

### Move.toml Template

None

```
[package]
name = "simple_lottery"
version = "0.0.1"
edition = "2024.beta"

[dependencies]
Sui = { git = "https://github.com/MystenLabs/sui.git", subdir
= "crates/sui-framework/packages/sui-framework", rev =
"framework/testnet" }
```

```
[addresses]
lottery = "0x0"
```

### Success Criteria

- ✅ Create lottery with ticket price and duration
- ✅ Users can buy multiple tickets
- ✅ Track all participants with entry counts
- ✅ Collect prize pool from ticket sales
- ✅ Prevent ticket purchases after lottery ends
- ✅ Draw winner after lottery ends
- ✅ Transfer entire prize pool to winner
- ✅ Prevent drawing winner twice
- ✅ Events emitted for all actions
- ✅ All tests pass

### Technical Hints

- Use `vector<address>` to store participants (duplicates allowed for multiple tickets)
- Calculate tickets bought: `payment_amount / ticket_price`
- Add buyer multiple times to vector (once per ticket)
- Use deterministic selection for winner (in production, use VRF)
- Simple randomness: combine object ID bytes with timestamp
- Use `vector::length()` to get total entries
- Loop through vector to count specific participant's entries
- Check lottery ended: `current_time >= end_time`

### Important Note on Randomness

This implementation uses a simple deterministic method for winner selection (combining object ID and timestamp). This is **not secure for production** as it can be predicted. For real lottery applications, you should use:

- Sui's native randomness API (when available)
- Verifiable Random Functions (VRF)
- Commit-reveal schemes
- Off-chain randomness oracles

For this educational challenge, the simple method is acceptable.

---

## Common Testing Patterns for Week 4

### Testing with Vectors

```
None
let mut participants = vector::empty<address>();
vector::push_back(&mut participants, @0xA);
vector::push_back(&mut participants, @0xA); // Can add
duplicates
assert!(vector::length(&participants) == 2, 0);

// Count occurrences
let mut count = 0;
let mut i = 0;
while (i < vector::length(&participants)) {
    if (*vector::borrow(&participants, i) == @0xA) {
        count = count + 1;
    };
    i = i + 1;
};
assert!(count == 2, 1);
```

### Testing with Time

```
None
let mut clock = clock::create_for_testing(ctx);

// Start at time 0
let start = clock::timestamp_ms(&clock);

// Fast forward 7 days
clock::increment_for_testing(&mut clock, 7 * 24 * 60 * 60 *
1000);

let end = clock::timestamp_ms(&clock);
assert!(end == start + 7 * 24 * 60 * 60 * 1000, 0);

clock::destroy_for_testing(clock);
```

### Testing Coin Calculations

```
None
let payment = coin::mint_for_testing<SUI>(25, ctx);
let ticket_price = 10;
let tickets = coin::value(&payment) / ticket_price; // = 2
tickets
let excess = coin::value(&payment) % ticket_price; // = 5 SUI
excess
```

---

## Evaluation Criteria

- **Completeness**: All required functions implemented
- **Correctness**: Code compiles and works as expected
- **Code Quality**: Clean, readable code with comments
- **Git History**: At least 4 meaningful commits
- **Testing**: All tests pass with good coverage
- **Deployment**: Code deployed to testnet
- **Documentation**: Package ID shared in Discord

## Support & Resources

- **Sui Discord**: https://discord.gg/sui (https://discord.com/invite/sQEkBSZbd7)
- **Sui Forum**: https://forums.sui.io
- **Documentation**: https://docs.sui.io
- **Move Examples**: https://examples.sui.io

Good luck with your Week 4 challenges! 🚀