**Smart Contract Exercise**

**Introduction**

In the previous exercises we’ve taken a look at the fundamentals of how blockchain networks work. While understanding what’s going on behind the scenes of these network is essential for working with them, this knowledge is quite theoretical, as developing a new blockchain from scratch which will be used on a wide scale isn’t extremely common.

In this exercise we’ll dive into the more commonly seen side of blockchain development, which is developing programs called smart contracts which are capable of being run on the Ethereum blockchain.

We’ll do this by first walking through the syntax of a pre-written smart contract for where players can place bets on a simple dice game called Knockout. Then, you’ll be challenged to use said syntax as well as the rules of Blackjack to implement a smart contract where players can bet on games.

**Preliminary Setup**

Smart contracts which can run on the Ethereum network are written in the Solidity programming language. Solidity is similar to JavaScript syntactically, however, it’s statically typed and compiled. One of the most commonly used IDEs for solidity is called [Remix](https://remix.ethereum.org/), and it can be run in your internet browser of choice. To set up Remix for the first part of this exercise, follow these steps:

* In the File Explorers tab, create a new workspace named ‘Exercise 4 Workspace’
  + Graphical user interface, application

    Description automatically generated
* In the ‘contracts’ folder of this workspace, create a new file called ‘KnockOutToken.sol’
* In the Solidity Compilers tab, note the default compiler version, in this case, 0.8.7
  + Website

    Description automatically generated with medium confidence
* Open up the ‘KnockOutToken.sol’ file, paste the following text into it, and change its version to match the compiler version

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

* Finally, paste the provided code from the ‘KnockOutToken.txt’ file into the .sol file

**Part 1 - Syntax Explanation**

Now that we’ve set up an environment for in which solidity code can be viewed and developed, we can explore the syntax of the Knockout contract that’s been given.

import 'https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol';

contract KnockOutToken is ERC20 {

After defining the version of solidity which is being used in the first line of the smart contract, any imports which are being used must be defined. In this case we’ll be using the ERC20 standard for fungible tokens which was mentioned in lecture. It’s hosted on the OpenZepplin GitHub where it has been battle tested and improved throughout the course of its existence. We’ll be using the standard as an interface for our contract; the documentation of its functions can be found [here](https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20).

The remaining syntax is explained in the comments in the Knockout contract, read through them and ensure that you fully comprehend what they’re doing before moving to part two. The comments and code have been pasted below for your convenience.

// Declaring an address variable for the address who deployed the contract

address public opponent;

// Structure of a game of Knockout, only the bet data persists here for the game of Knockout, however more data could be stored here for more complex games

struct game { uint bet; }

// Creating a mapping between a user's address and a game of knockout (similar to a hash map)

mapping (address => game) games;

address[] public total\_games;

// The constructor of the smart contract, only called once on deployment of the contract, notice how it also calls the ERC20 constructor

constructor() ERC20('KnockoutToken', 'KOT') {

    // Assigning the account which deployed the smart contract to be the opponent for the game

    opponent = msg.sender;

    // Inital supply of 1000 tokens minted for the account which deployed the smart contract, with 18 decimal points (ERC20 default)

    \_mint(opponent, 1000 \* 10 \*\* 18);

}

// Creating a wrapper function around the the mint functionality in the ERC20 contract.

// The external keyword means it can be called outside of the smart contract

function mint(address to, uint amount) external {

    // The require function is used for validating conditions before executing the code in the rest of the function.

    // If the condition (first parameter) fails, the transaction will fail and the message (second parameter) will be returned

    require(msg.sender == opponent, "Only the opponent can mint tokens.");

    // Calling the \_mint function from the ERC20 contract (https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#ERC20-\_mint-address-uint256-)

    \_mint(to, amount);

}

// Creating a function which uses pseudo-randomness to simulate rolling a dice

// The private key word means it can only be called by other functions within the contract

// The view key word means it is view only and does not modify the contract's state

// The seed parameter is used in case multiple random numbers need to be generated in one block (using multiple seeds)

function roll\_dice(uint \_seed) private view returns (uint){

    // Generating pseudo-random value between 0 and 5 using the keccak256 hashing function

    uint roll = uint(keccak256(abi.encodePacked(block.difficulty, block.timestamp, \_seed))) % 6;

    // Adding one to the dice value so it's range is between 1 and 6, and then returning it

    return ++roll;

}

// An external function which a player can use to place a bet and start a new game

function new\_game(uint \_bet) external {

    // Require functions which ensure that the address calling the contract to start a new

    // game isn't already playing a game and has placed a non-zero bet for this new game

    require(games[msg.sender].bet == 0, "A current game is already in play.");

    require(\_bet > 0, "Bet must be greater than zero.");

    // Require function which ensures the address calling the contract has a balance can cover their bet

    uint balance = balanceOf(msg.sender);

    require((balance > 0) && (\_bet < balance), "Not enough tokens to play.");

        // Using a function from the ERC20 contract to send the bet to the smart contract deployer, in this case, the opponent of the game

    transfer(opponent, \_bet);

    // Create game object for player and appending it to the address-game mapping

    games[msg.sender] = game({ bet: \_bet });

    total\_games.push(msg.sender);

}

// A private function that resets the bet of a player in the address-game mapping once their game has ended

function end\_game(address player) private {

    games[player].bet = 0;

}

// An event must be created and emited in order to return information to a user who calls an external function outside of the smart contract

event return\_game\_state(uint player\_value, uint opponent\_value);

// An external function which a player can use to play a round of a game once they've created one by calling the new\_game function

function play\_round() external {

    // Require function which ensure that a game is in play for the player address which called the smart contract

    require(games[msg.sender].bet != 0, "Must start a new game to play.");

    // Using the roll\_dice function described earlier to roll a dice for the player and the opponent

    uint player\_roll = roll\_dice(0);

    uint opponent\_roll = roll\_dice(1);

    // Emitting the previously created event to inform the player of their dice value and the dice value of the opponent

    emit return\_game\_state(player\_roll, opponent\_roll);

    // When the player wins, pay out double their taken bet using functions from the ERC20 contract and end the game

    if (player\_roll > opponent\_roll) {

        uint win\_amount = games[msg.sender].bet \* 2;

        // Before a smart contract can transact tokens on behalf of another account, they must ask for approval from the other account

        \_approve(opponent, msg.sender, win\_amount);

       // Once approval is gained, they can transact the approved amount of tokens

        transferFrom(opponent, msg.sender, win\_amount);

        end\_game(msg.sender);

    } else if (opponent\_roll > player\_roll) { // When the oponent wins, don't pay out anything and end the game

        end\_game(msg.sender);

    }

    // If player and opponent tie, another round can be played

}

Next, we’ll test the functionality of this smart contract, to do this head to the ‘Deploy & Run Transactions’ tab in Remix. After making sure the Knockout smart contract is compiled, it should show up in the contract dropdown. Select it and click the deploy button. This will deploy the smart contract on a local instance of the Ethereum network running in your browser.

Graphical user interface, application, website

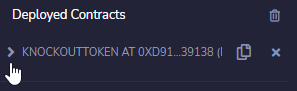
Description automatically generated

Once it’s deployed, notice that the address which was selected in the account dropdown has been charged a small amount of Ethereum for deploying the smart contract.

Graphical user interface, text, application, chat or text message

Description automatically generated

Next, in the deployed contract menu you should see the smart contract which was just deployed, click the arrow next to it to expand it and see the functions which an external source is able to call.



To test smart contract functions, start by using either the mint or transfer function to send ‘KnockOutTokens’ to the second address in the account dropdown. To do this, first copy it’s address to your clipboard by clicking the following button.

Graphical user interface, text, application, chat or text message

Description automatically generated

Now switch back to the first address in the account dropdown and paste the address you just copied in either the mint or transfer function, followed by a comma and the number of tokens you want to transfer to the secondary address and then transact.

Graphical user interface, application

Description automatically generated

Switch back to the secondary address once again. You should be able to verify that the tokens were minted for the address by using the balanceOf() function.

Graphical user interface, application

Description automatically generated

The secondary account is now able to start a new game by placing a bet using the new\_game() function. Once this is done, the play\_round() function is now usable for the second account, call that method and then open up the terminal to see the event which the function returned, within it you should see the value of the players die as well as the opponents die (along with other data).

Text

Description automatically generated

Unless the result is a tie, the balances of the accounts playing the game will have changed, you can use the balanceOf() function once again to verify this.

**Part 2 - Knowledge Application**

Now that you’ve seen how to create a smart contract which involves an ERC20 token that can be used for a simple game, you should have enough information to be able to apply what you’ve learned. We’ll do this through creating a new smart contract named ‘BlackJackToken.sol’ and pasting in the provided template code.

In Blackjack, your goal is to get as close to 21 as possible without going over. Initially, you are dealt two cards. In each turn, you can decide to hit (ask for a card to be added to your set) or to stand (do nothing). If you stand, you may not hit any more – you just wait for the game to end. The card point values are as follows: 2 – 10 have their face values, Jack is 11 points, Queen is 12 points, King is 13 points, and Aces are 1 or 11 points. To simplify the scoring in this version of Blackjack, Aces will only count for 1 point by default.

For extra credit, logic can be implemented such that Aces will count for 11 points by default. if you go over, the aces are converted to 1 point until your score is no longer over 21. Said another way, the aces are set to 1 or 11 so as to maximize your score without going over.

At the start of each new game, both players will draw two cards, and an event with the value of both of these cards will be emitted to the player. Once the player either busts or is done hitting and decides to stand, the dealer will then autonomously draw until they either bust or outscore the player, with the stipulation that they must stand if their cards total value equals 17 or more.

The implemented betting logic should be very similar to that which was created in the Knockout contract. If the player hits blackjack (scores 21 points), they’ll instantly be paid back 2.5 times their original bet, regardless of the dealer’s point value. If they stand and their points are at 20 points or less, when the dealer either busts or stands on 17 points or above, the player will either be paid back 2 times their original bet if they have more points than the dealer, paid back their original bet if they tie with the dealer, or not be paid back anything if the dealer outscores them.

Don’t forget to test the functionality of the smart contract before turning it in.