FT5003 Lab 2

Preparing the environment:

1. Open the sample Dice.sol using remix: https://bit.ly/Dice_sol

2. Open 639….32e > Dice.sol

3. Enable the “Solidity” environment

Alternatively, check out links to the code in the appendix.

Create a file called ‘Dice.sol’ and copy over the other Dice contract’s skeleton. We’ll be using this for the problems in the lab.

Problem 1: (Revise Coding)

- Extend the ***Dice*** contract to add a new property: ***luckyTimes*** which is incremented *every time the max number is rolled.*

- Add a getter function to read this property (Basically return luckyTimes value)

- Add an event called **luckytimesEvent** for the times max number is rolled

- Add a function called **destroyDice** to destroy dice and return ether

Problem 2: (OOP with Smart Contracts)

- Implement a ***DiceMarket*** contract. It receives ownership of the dice, and enable the functions. A commission fee is set by the owner during creation of ***DiceMarket*** Contract. Implement the following methods:

- ***list(uint256 id, uint256 price)*** – list a dice for sale. Price needs to be >= value + comission fee
    - o *First, transfer the dice to the DiceMarket contract’s address.*
    - o *Then, you should be able to list the dice in this market*

- ***unlist(uint256 id)*** – unlist dice from the market
    - o *Upon unlisting do not transfer the dice back to their owners.*
    - o *Simply delist them from the market, ie nobody should be able to buy the die.*

- ***checkPrice(uint256 id)*** – get price of dice

- ***buy(uint256 id)*** – Buy the dice at the requested price

    - If you want to implement an airtight solution, you should return any extra money to the msg.sender.

- Note: please set appropriate modifier to check for condition before allowing the execution of certain functions.

Problem 3: (ERC20 Standard)

- Issue a ERC-20 token, **DT** (DiceToken), such that
- It complies with ERC-20 Interface
- The total supply is 10,000 token
- Anyone can top up DT, with the price of 0.01 Eth per DT
- When the supply is not enough (e.g., someone wants to top up 200DT, but there is only 100DT left in supply), return with error message "DT supply is not enough".
- Hint: We'll be using the ERC20 contract accessible in the appendix

Lab 2 exercises:

Exercise 1:

- Extend the **Dice smart contract** and implement another contract called **DiceBattle**. **DiceBattle** allows the uses to roll 2 dice by supplying the diceId. The ownership of the **Dice** is transferred to the winner of the **DiceBattle.**
    - **See the skeleton for a detailed breakdown of the contract's purpose.**

Exercise 2:

- Modify problem 2 to use DT instead.
    - Perform the same functionalities as problem 2 but instead of using ether, use DT as payment method (for both commission and trade).
    - HINT: We created DT in the lab

Submission: Please submit a zip that contains 5 subfolders, corresponding to the 5 questions. For each folder, put all the necessary .sol files for that question in it.

Appendix

ERC20.sol

```
pragma solidity ^0.8.28;


//first need to approve the address of spender
// Check the allowance
//Finally able to call transferFrom to transfer tokens

/**
 * @title SafeMath
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {

  /**
  * @dev Multiplies two numbers, throws on overflow.
  */
  function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
      if (a == 0) {
      return 0;
      }
      c = a * b;
      assert(c / a == b);
      return c;
  }

  /**
  * @dev Integer division of two numbers, truncating the quotient.
  */
  function div(uint256 a, uint256 b) internal pure returns (uint256) {
      // assert(b > 0); // Solidity automatically throws when dividing by 0
      // uint256 c = a / b;
      // assert(a == b * c + a % b); // There is no case in which this
doesn't hold
      return a / b;
  }

  /**
  * @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is
greater than minuend).
  */
  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
      assert(b <= a);
      return a - b;
  }

  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal pure returns (uint256 c) {
      c = a + b;
      assert(c >= a);
      return c;
  }
}
```

```
contract ERC20 {
      using SafeMath for uint256;

      bool public mintingFinished = false;

      address public owner = msg.sender;

      mapping (address => mapping (address => uint256)) internal allowed;
      mapping(address => uint256) balances;


      string public constant name = "DiceToken";
      string public constant symbol = "DT";
      uint8 public constant decimals = 18;
      uint256 totalSupply_;

      event Transfer(address indexed from, address indexed to, uint256
value);
      event Approval(address indexed owner, address indexed spender,
uint256 value);
      event Mint(address indexed to, uint256 amount);
      event MintFinished();





  /**
  * @dev total number of tokens in existence
  */
  function totalSupply() public view returns (uint256) {
      return totalSupply_;
  }

      /**
  * @dev Gets the balance of the specified address.
  * @param _owner The address to query the the balance of.
  * @return An uint256 representing the amount owned by the passed address.
  */
  function balanceOf(address _owner) public view returns (uint256) {
      return balances[_owner];
  }


  /**
  * @dev transfer token for a specified address
  * @param _to The address to transfer to.
  * @param _value The amount to be transferred.
  */
  function transfer(address _to, uint256 _value) public returns (bool) {
      require(_to != address(0));
      require(_value <= balances[tx.origin], "msg.sender doesn't have
enough balance");

      balances[tx.origin] = balances[tx.origin].sub(_value);
      balances[_to] = balances[_to].add(_value);
      emit Transfer(tx.origin, _to, _value);
      return true;
  }
  /**
   * @dev Transfer tokens from one address to another
   * @param _from address The address which you want to send tokens from
   * @param _to address The address which you want to transfer to
   * @param _value uint256 the amount of tokens to be transferred
   */
  function transferFrom(address _from, address _to, uint256 _value) public
returns (bool) {
      require(_to != address(0));
      require(_value <= balances[_from], "From doesn't have enough
balance");
      require(_value <= allowed[_from][tx.origin], "Not allowed to spend
this much");

      balances[_from] = balances[_from].sub(_value);
      balances[_to] = balances[_to].add(_value);
      allowed[_from][tx.origin] = allowed[_from][tx.origin].sub(_value);
      emit Transfer(_from, _to, _value);
      return true;
  }

  /**
   * @dev Approve the passed address to spend the specified amount of
tokens on behalf of msg.sender.
   *
   * Beware that changing an allowance with this method brings the risk
that someone may use both the old
   * and the new allowance by unfortunate transaction ordering. One
possible solution to mitigate this
   * race condition is to first reduce the spender's allowance to 0 and set
the desired value afterwards:
   * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
   * @param _spender The address which will spend the funds.
   * @param _value The amount of tokens to be spent.
   */
  function approve(address _spender, uint256 _value) public returns (bool)
{
      allowed[msg.sender][_spender] = _value;
      emit Approval(msg.sender, _spender, _value);
      return true;
  }

  /**
   * @dev Function to check the amount of tokens that an owner allowed to a
spender.
   * @param _owner address The address which owns the funds.
   * @param _spender address The address which will spend the funds.
   * @return A uint256 specifying the amount of tokens still available for
the spender.
   */
  function allowance(address _owner, address _spender) public view returns
(uint256) {
      return allowed[_owner][_spender];
  }


      /**
   * @dev Function to mint tokens
   * @param _to The address that will receive the minted tokens.
   * @param _amount The amount of tokens to mint.
   * @return A boolean that indicates if the operation was successful.
   */
  function mint(address _to, uint256 _amount) onlyOwner canMint public
returns (bool) {
      totalSupply_ = totalSupply_.add(_amount);
      balances[_to] = balances[_to].add(_amount);
      emit Mint(_to, _amount);
      emit Transfer(address(0), _to, _amount);
      return true;
  }

  /**
   * @dev Function to stop minting new tokens.
   * @return True if the operation was successful.
   */
  function finishMinting() onlyOwner canMint public returns (bool) {
      mintingFinished = true;
      emit MintFinished();
      return true;
  }

  function getOwner() public view returns (address){
      return owner;
  }


   modifier onlyOwner() {
      require(msg.sender == owner);
      _;
  }


  modifier canMint() {
      require(!mintingFinished);
      _;
  }




}
```

DiceBattleSkeleton.sol

```
pragma solidity ^0.8.28;
import "./Dice.sol";

/*
1. First create dice using the Dice contract
2. Transfer both die to this contract using the contract's address
3. Use setBattlePair from each player's account to decide enemy
```

```
4. Use the battle function to roll, stop rolling and then compare the
numbers
5. The player with the higher number gets BOTH dice
6. If there is a tie, return the dice to their previous owner
*/


contract DiceBattle {
      Dice diceContract;
      mapping(address => address) battle_pair;

      constructor(Dice diceAddress) public {
      diceContract = diceAddress;
      }

      function setBattlePair(address enemy) public {
      // Require that only prev owner can allow an enemy
      // Each player can only select one enemy

      }

      function battle(uint256 myDice, uint256 enemyDice) public {
      // Require that battle_pairs align, ie each player has accepted a
battle with the other

      // Run battle

      }

      //Add relevant getters and setters
}
```