Peer To Peer Systems and Blockchains
Final Term
TRY: a nfT lotteRY

Francesco Carli - 559565

Academic Year 2021/2022

The goal of the project is to implement TRY, a lottery offering users collectibles as winning prizes. The rules of the lottery are inspired by Powerball, a popular type of lottery played in the United States. The lottery logic is implemented with a Solidity smart contract on the Ethereum blockchain. Before operating the lottery, the lottery manager buys a batch of collectibles and mints a Non Fungible Token (NFT) for each of them. Each lottery's winner may receive as a prize one of these collectibles.

1 NFT Contract

A non-fungible token (NFT) is used to define something or someone in a unique way. This type of token is perfect on applications that manage collectibles, access keys, lottery tickets, numbered seats for concerts or sporting events etc. Due to these particular features, each NFT is unique and has a different value with respect to an other NFT of the same smart contract.
ERC721 is the de-facto standard to represent the ownership of a NFT, and provides several interfaces, implementations and extensions. There are 4 main interfaces and OpenZeppelin Contracts implements all these interfaces and provides some additional extensions.
In the final term, it has been used the implementation of ERC721 Non-Fungible Token Standard, including the Metadata extension, to create the Smart Contract representing the set of collectibles of the lottery. This implementation contains several events and methods, and some of them have been override to meet the specific requirements of this assignment. The NFT Smart Contract, namely *NFT.sol*, extends *ERC721.sol* and so it inherits all its methods and data structures.
First of all, a **mapping from a *tokenId* to a *string*** has been declared, in order to maintain an association between a token and its content, which is a simple description of the NFT itself (just for simplicity). Then, the constructor is the same of ERC721 and it initializes the contract setting a *name* and a *symbol* to the collection. The first overridden method is ***mint***, which takes as parameters a *tokenId*, an address and a content. This method initially checks that the destination address that will own the NFT is the address of the minter (the lottery operator) and, if it is true, mints *tokenId* with the given content and transfers it to the address of the lottery operator.
Another overridden method is ***safeTransferFrom*** which takes as parameters the sender's address, the destination address and the *tokenId* to transfer. It requires that the address passed as *from* owns the *tokenId* and, if it is true, calls a method of ERC721.
Then, two additional methods have been added: they are ***getTokenContent*** and ***setTokenContent*** and are used respectively to provide a token content given its *tokenId* and to set the content of a *tokenId*. Both these methods require that the provided *tokenId* must exist and deal with the previously described mapping.
Finally, also a new event, called ***TokenContent***, has been added in order to show the content of the token after the invocation of ***setTokenContent***.

2 Lottery Contract

The lottery contract has several required methods and also some auxiliary ones. First of all, several data structures have been declared in order to represent the main actors and situations that may occur: in this way it is possible to represent a single lottery round, a collectible, a lottery player and a lottery winner. Several events have been declared in order to show useful information and outputs of the methods.

Required Methods

- ***startNewRound***: method called by the lottery operator to start a new lottery round. Before to start a new round, it checks if the new round is the first or if the previous round is finished, namely if it lasted at least M blocks. It uses the round's data structure to keep trace of the number of the round, the state and the block at which the round starts.
- ***buy***: payable method called by users to buy lottery tickets. It takes as input the numbers of the lottery (5 standard numbers between 1 and 69 and the powerball between 1 and 26). It creates a new player which it is added to the list of players and check also if the user which has bought the ticket need the change. In order to buy a ticket, a user has to own at least an amount of funds which is equal or greater than the price of a ticket.
- ***drawNumbers***: method called by the lottery operator after at least M blocks to draw the winning numbers. It uses an internal method, called ***generateRandomNumber***, to draw the winning numbers. At the end of this method the round state is modified in order to perform the check on the tickets of the players and see if there are winners.
- ***givePrizes***: method called by the lottery operator to check if there are winners and assign them the appropriate prizes, according to their ranks. It has been used a list of winners to keep trace of the number of winners and the appropriate prizes. This method manages also the case in which it could be possible to have a number of winners of a certain rank greater than the number of prizes of that rank: in this case, after the appropriate check, it is used a modified version of mint, called ***mintOnDemand***.
- ***mint***: method called by the lottery operator to mint several NFTs. In particular, this method takes as input a parameter n, so the lottery operator is able to mint n collectibles for each rank, from 1 to 8.
- ***closeLottery***: method called by the lottery operator to close the current lottery round. This method check the state of the lottery and, basing on the current state, it does all the operations to close the lottery in a proper manner. For example, if this method is called and the lottery is still active, users are able to buy tickets, so after the invocation, the method has to refund all the players.

Auxiliary Methods

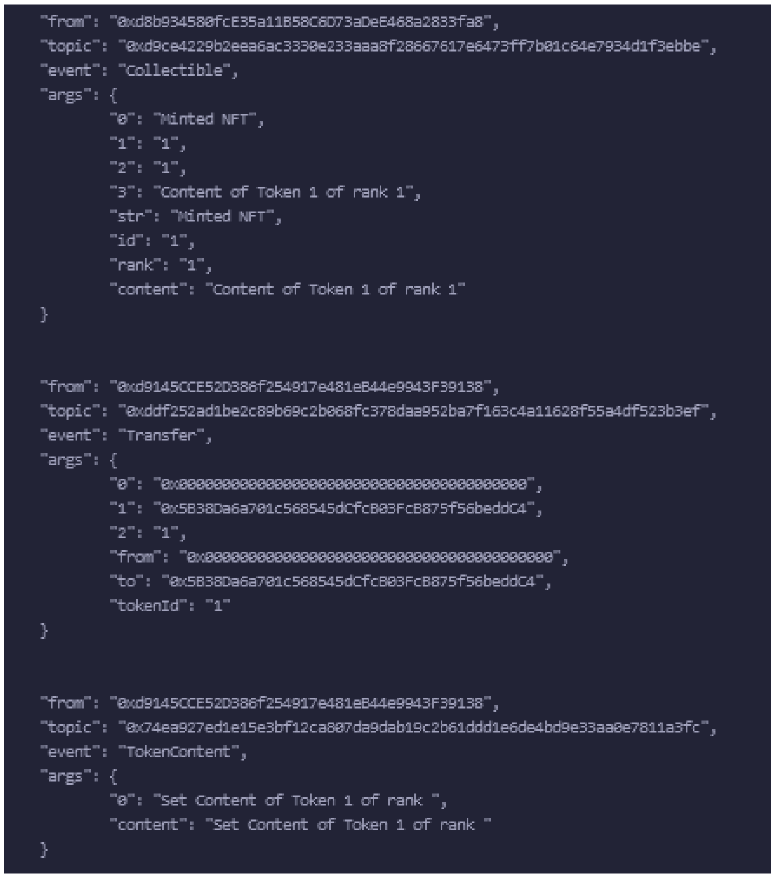
- ***resetDraws***: method which reset the data structure used to store the draw of a lottery round.
- ***convertDraws***: method used to check and correct the range of the standard numbers and the powerball.
- ***assignRanks***: method used to assign the rank to a winner ticket, based on the amount of correct numbers.
- ***generateRandomNumber***: method used to generate a random number in a deterministic way. This method is used to perform the draw of the lottery and to select a prize of a given rank in a random way. It takes as input a seed and uses a source of randomness from the blockchain. Seed, K and the height of the block where the lottery ends are passed to *keccak256 hash function* which compute a digest.
- ***delete*** : method used to reset all the data structures when the lottery round is closed.

- ***mintOnDemand***: method used to mint a batch of NFTs of a given rank and it is called when there is a number of winner of a certain rank greater than the number of prizes of a certain rank. To manage this situation, this method takes as parameters the amount of NFTs to mint and the rank. Thanks to this method, in the lottery there will be always present a batch of collectibles and it can't never happen that there are no collectibles in the lottery.

3 Event Logs generated by contracts

In order to show useful information and outputs, in the Lottery Smart Contract have been declared the following events:

- ***Collectible***: event used to show the output of a mint operation, such as *tokenId*, rank and content. There are also two more events, namely *Transfer* and *TokenContent*, which are declared in *NFT.sol* and represent respectively the transfer of the ownership and the initialization of the token content.



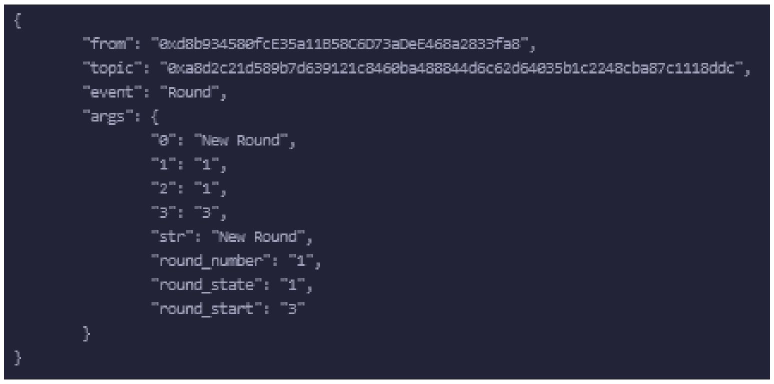
```
"from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
"topic": "0xd9ce4229b2eea6ac3330e233aaa8f28667617e6473ff7b01c64e7934d1f3ebbe",
"event": "Collectible",
"args": {
        "0": "Minted NFT",
        "1": "1",
        "2": "1",
        "3": "Content of Token 1 of rank 1",
        "str": "Minted NFT",
        "id": "1",
        "rank": "1",
        "content": "Content of Token 1 of rank 1"
}


"from": "0xd9145CCE52D386f254917e481eB44e9943F39138",
"topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
"event": "Transfer",
"args": {
        "0": "0x0000000000000000000000000000000000000000",
        "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
        "2": "1",
        "from": "0x0000000000000000000000000000000000000000",
        "to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
        "tokenId": "1"
}


"from": "0xd9145CCE52D386f254917e481eB44e9943F39138",
"topic": "0x74ea927ed1e15e3bf12ca807da9dab19c2b61ddd1e6de4bd9e33aa0e7811a3fc",
"event": "TokenContent",
"args": {
        "0": "Set Content of Token 1 of rank ",
        "content": "Set Content of Token 1 of rank "
}
```

Figure 1: Output of *mint* method.

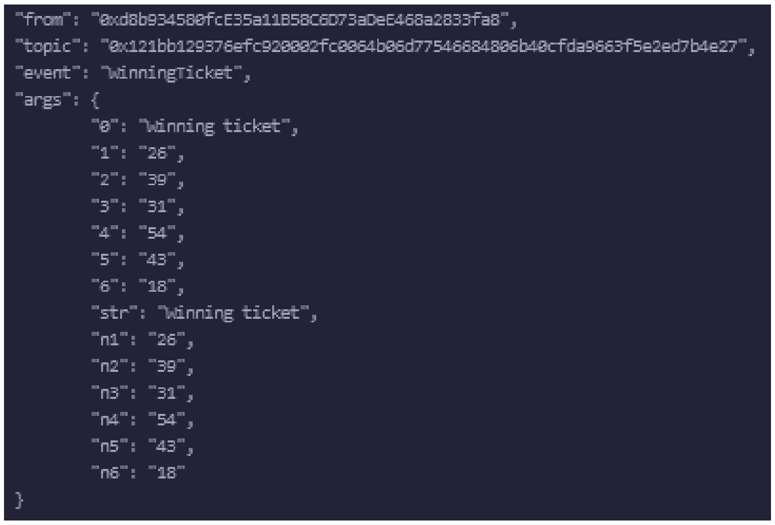
- ***Round***: event used to show the output of ***startNewRound***, such as the number of the round, its state and its starting block.



```
{
        "from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
        "topic": "0xa8d2c21d589b7d639121c8460ba488844d6c62d64035b1c2248cba87c1118ddc",
        "event": "Round",
        "args": {
                "0": "New Round",
                "1": "1",
                "2": "1",
                "3": "3",
                "str": "New Round",
                "round_number": "1",
                "round_state": "1",
                "round_start": "3"
        }
}
```

Figure 2: Output of *startNewRound* method.

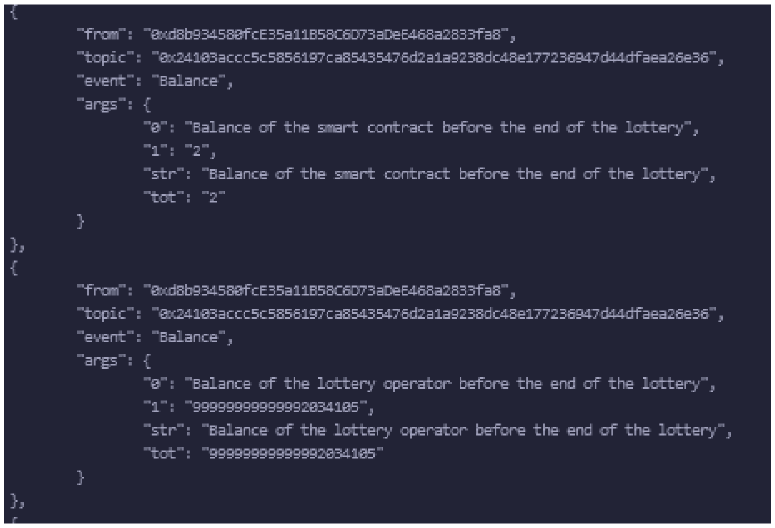
- ***Player***: event used to show the output of ***buy***, namely the address of a player and the numbers he has selected.
- ***WinningTicket***: event used to show the output of ***drawNumbers***, namely the winning numbers.



```
"from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
"topic": "0x121bb129376efc920002fc0064b06d77546684806b40cfda9663f5e2ed7b4e27",
"event": "WinningTicket",
"args": {
        "0": "Winning ticket",
        "1": "26",
        "2": "39",
        "3": "31",
        "4": "54",
        "5": "43",
        "6": "18",
        "str": "Winning ticket",
        "n1": "26",
        "n2": "39",
        "n3": "31",
        "n4": "54",
        "n5": "43",
        "n6": "18"
}
```

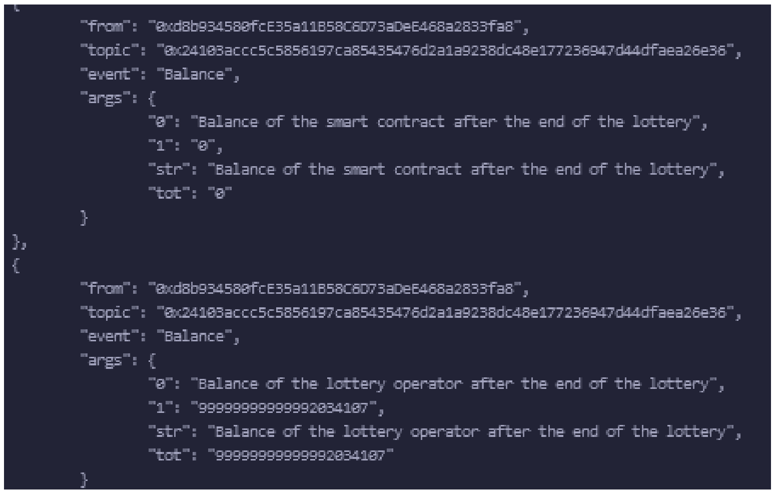
Figure 3: Output of *drawNumbers* method.

- ***Prize***: event used to show the output of ***givePrizes***, namely which prize of a given rank is associated with one of the appropriate winners.
- ***Balance***: event used to show the balance of the smart contract, of the lottery operator and the players before and after the call of ***closeLottery***.



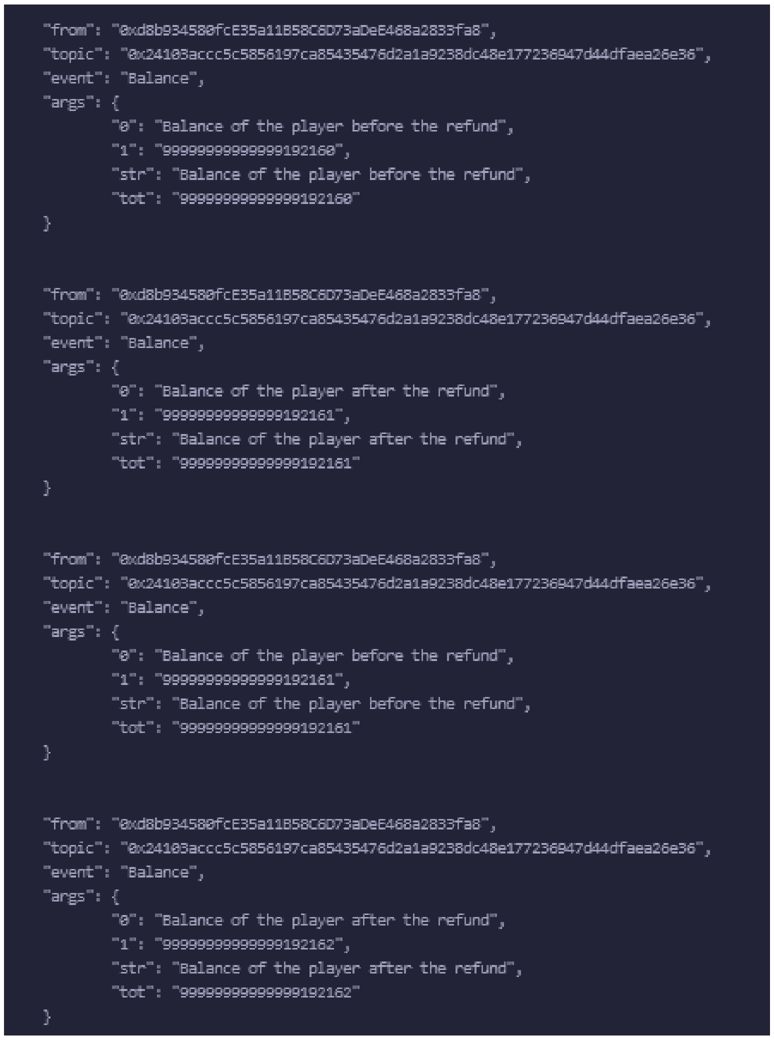
```
{
        "from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
        "topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
        "event": "Balance",
        "args": {
                "0": "Balance of the smart contract before the end of the lottery",
                "1": "2",
                "str": "Balance of the smart contract before the end of the lottery",
                "tot": "2"
        }
},
{
        "from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
        "topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
        "event": "Balance",
        "args": {
                "0": "Balance of the lottery operator before the end of the lottery",
                "1": "99999999999992034105",
                "str": "Balance of the lottery operator before the end of the lottery",
                "tot": "99999999999992034105"
        }
},
```

Figure 4: Output of *closeLottery* method.



```
        "from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
        "topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
        "event": "Balance",
        "args": {
                "0": "Balance of the smart contract after the end of the lottery",
                "1": "0",
                "str": "Balance of the smart contract after the end of the lottery",
                "tot": "0"
        }
},
{
        "from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
        "topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
        "event": "Balance",
        "args": {
                "0": "Balance of the lottery operator after the end of the lottery",
                "1": "99999999999992034107",
                "str": "Balance of the lottery operator after the end of the lottery",
                "tot": "99999999999992034107"
        }
```

Figure 5: Output of *closeLottery* method.



```
"from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
"topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
"event": "Balance",
"args": {
        "0": "Balance of the player before the refund",
        "1": "99999999999999192160",
        "str": "Balance of the player before the refund",
        "tot": "99999999999999192160"
}


"from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
"topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
"event": "Balance",
"args": {
        "0": "Balance of the player after the refund",
        "1": "99999999999999192161",
        "str": "Balance of the player after the refund",
        "tot": "99999999999999192161"
}


"from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
"topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
"event": "Balance",
"args": {
        "0": "Balance of the player before the refund",
        "1": "99999999999999192161",
        "str": "Balance of the player before the refund",
        "tot": "99999999999999192161"
}


"from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
"topic": "0x24103accc5c5856197ca85435476d2a1a9238dc48e177236947d44dfaea26e36",
"event": "Balance",
"args": {
        "0": "Balance of the player after the refund",
        "1": "99999999999999192162",
        "str": "Balance of the player after the refund",
        "tot": "99999999999999192162"
}
```

Figure 6: Output of *closeLottery* method in case lottery round was active.

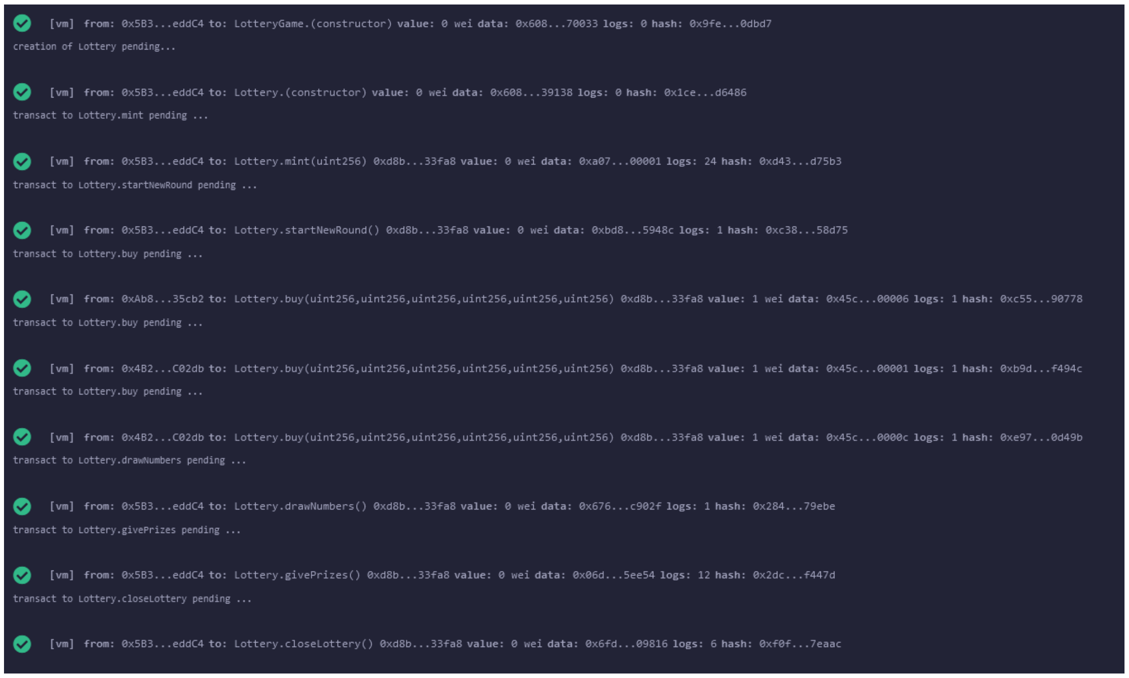
4 List of Operations

In order to run meaningful simulations of the system, in the following subsections there are lists of temporally ordered operations.

Lottery Simulation

The following steps simulate the standard execution of a lottery round.

1. Deploy of **NFT.sol**, namely the NFT Smart Contract.
2. Deploy of **Lottery.sol**, namely the Lottery Smart Contract with the appropriate parameters (ticket price, M, K, address of NFT Smart Contract).
3. Call ***mint*** passing a parameter n to mint n NFTs for each rank. The lottery will work also if this method won't be called, thanks to *mintOnDemand*.
4. Call ***startNewRound*** to start a new lottery round.
5. Change address and call ***buy*** to buy several tickets with numbers selected by the users. Check also to pass a value at least equal to the ticket price in order to buy a ticket. It is possible to call this method several times, from the same player's address or frome other addresses to have several players of the lottery.
6. After at least M blocks, select the address of the lottery operator and call ***drawNumbers*** to draw the winning numbers.
7. Call ***givePrizes*** to check if there are winners and assign appropriate prizes to them, according to the ranks.
8. Call ***closeLottery*** to close the lottery round and transfer the total balance of the lottery to the address of the lottery operator.



```
✓  [vm]  from: 0x5B3...eddC4 to: LotteryGame.(constructor) value: 0 wei data: 0x608...70033 logs: 0 hash: 0x9fe...0dbd7
creation of Lottery pending...

✓  [vm]  from: 0x5B3...eddC4 to: Lottery.(constructor) value: 0 wei data: 0x608...39138 logs: 0 hash: 0x1ce...d6486
transact to Lottery.mint pending ...

✓  [vm]  from: 0x5B3...eddC4 to: Lottery.mint(uint256) 0xd8b...33fa8 value: 0 wei data: 0xa07...00001 logs: 24 hash: 0xd43...d75b3
transact to Lottery.startNewRound pending ...

✓  [vm]  from: 0x5B3...eddC4 to: Lottery.startNewRound() 0xd8b...33fa8 value: 0 wei data: 0xbd8...5948c logs: 1 hash: 0xc38...58d75
transact to Lottery.buy pending ...

✓  [vm]  from: 0xAb8...35cb2 to: Lottery.buy(uint256,uint256,uint256,uint256,uint256,uint256) 0xd8b...33fa8 value: 1 wei data: 0x45c...00006 logs: 1 hash: 0xc55...90778
transact to Lottery.buy pending ...

✓  [vm]  from: 0x4B2...C02db to: Lottery.buy(uint256,uint256,uint256,uint256,uint256,uint256) 0xd8b...33fa8 value: 1 wei data: 0x45c...00001 logs: 1 hash: 0xb9d...f494c
transact to Lottery.buy pending ...

✓  [vm]  from: 0x4B2...C02db to: Lottery.buy(uint256,uint256,uint256,uint256,uint256,uint256) 0xd8b...33fa8 value: 1 wei data: 0x45c...0000c logs: 1 hash: 0xe97...0d49b
transact to Lottery.drawNumbers pending ...

✓  [vm]  from: 0x5B3...eddC4 to: Lottery.drawNumbers() 0xd8b...33fa8 value: 0 wei data: 0x676...c902f logs: 1 hash: 0x284...79ebe
transact to Lottery.givePrizes pending ...

✓  [vm]  from: 0x5B3...eddC4 to: Lottery.givePrizes() 0xd8b...33fa8 value: 0 wei data: 0x06d...5ee54 logs: 12 hash: 0x2dc...f447d
transact to Lottery.closeLottery pending ...

✓  [vm]  from: 0x5B3...eddC4 to: Lottery.closeLottery() 0xd8b...33fa8 value: 0 wei data: 0x6fd...09816 logs: 6 hash: 0xf0f...7eaac
```

Figure 7: Log about the previous sequence of steps.

Lottery Simulation with refund to users

The following steps simulate the execution of a lottery round and the close of the lottery before M blocks.

1. Deploy of **NFT.sol**, namely the NFT Smart Contract.
2. Deploy of **Lottery.sol**, namely the Lottery Smart Contract with the appropriate parameters (ticket price, M, K, address of NFT Smart Contract).
3. Call ***mint*** passing a parameter n to mint n NFTs for each rank.
4. Call ***startNewRound*** to start a new lottery round.
5. Change address and call ***buy*** to buy several tickets with numbers selected by the users.
6. Call ***closeLottery*** to close the lottery round and refund users which bought tickets during the active lottery round.

Lottery Simulation with distribution of prizes

The following steps simulate the execution of a lottery round and the close of the lottery before calling ***givePrizes***.

1. Deploy of **NFT.sol**, namely the NFT Smart Contract.
2. Deploy of **Lottery.sol**, namely the Lottery Smart Contract with the appropriate parameters (ticket price, M, K, address of NFT Smart Contract).
3. Call ***mint*** passing a parameter n to mint n NFTs for each rank.
4. Call ***startNewRound*** to start a new lottery round.
5. Change address and call ***buy*** to buy several tickets with numbers selected by the users.
6. After at least M blocks, select the address of the lottery operator and call ***drawNumbers*** to draw the winning numbers.
7. Call ***closeLottery*** to close the lottery round, check if there are winners, assign the appropriate prizes and transfer the total balance of the lottery to the address of the lottery operator.

5 Gas Estimation of Non-Trivial Functions

Ethereum has introduced the concept of gas: the main idea behind it is to pay for contract execution giving gas to smart contracts. Each computational step has a fixed gas cost and the EVM (Ethereum Virtual Machine) stops the execution of a smart contract if it goes out of gas.
"Buying" gas is like buying some computational power: gas price is in Ether, is variable and so it is up to the caller to decide the amount of Ether to spend, because executing a function in two different moments could be more or less expensive.
In each transaction there are gas price and a gas limit, with which is possible to compute the transaction fee. When a sender set the gas limit, he/she has to keep in mind that different operations have different gas costs and if the transaction goes out of gas, its execution will be halted and $gas\_price * gas\_limit$ will be deducted from the sender's balance in any case.
Checking the main transactions on Remix, it is possible to see some fields related to gas, in particular *transaction cost* and *execution cost*. Transaction cost is a sort of mix of the actual transaction cost plus the execution cost, is based on the cost of sending data to the blockchain and there are 4 items which make up the full transaction cost:

1. The base cost of a transaction (21000 gas).
2. The cost of a contract deployment (32000 gas).
3. The cost for every zero byte of data or code for a transaction (4 gas).
4. The cost of every non-zero byte of data or code for a transaction (68 gas).

Execution costs are based on the cost of computational operations which are executed as a result of the transaction.
The following list shows the amount of gas consumed by the main methods in *Lottery.sol*:

- ***startNewRound***: 108079 gas (first iteration), 59265 (second iteration).
- ***buy***: 210841 gas.
- ***drawNumbers***: 183568 gas.
- ***givePrizes***: 687137 gas with 2 winners.
- ***mint***: 1150365 gas for minting 8 NFTs.
- ***closeLottery***: 180785 gas.

In the previous list is not reported the deploy cost, which is a one-time cost. From the values above, it is possible to see that the three methods with less consumed gas are *startNewRound*, *drawNumbers* and *closeLottery*. Looking at the code, these behaviors seem to be appropriate because these three methods perform only simple operations like modify a struct which represent the lottery round, draw the winning numbers using a RNG and check again the struct of the round to close the lottery in a fairly manner.
Instead, if we look at *mint* and *givePrizes*, we see that their costs are higher than the others. Again, these aspects seem to be right because *mint* is the function used by the lottery operator to create new NFTs, so it requires to create a new collectible, initialize it, set its content and then call the *mint* method of *ERC721.sol*. Nevertheless, this is not a problem in a real scenario, because *mint* is called by the lottery operator only one time before the start of the lottery. Moreover, *givePrizes* is the most complex method of *Lottery.sol*: it contains several loops because it has to check if there are winners among all the lottery players and it has also to manage the prizes and their ranks, assigning the appropriate ones to the winners according to the amount of winner numbers. Then, its cost changes basing on if there are winners and how many are.

6 Smart Contract Security

The main security vulnerability about the Lottery Smart Contract is that Solidity is not capable of creating true random numbers and today every algorithm for creating random numbers is pseudo-random. The problem with Solidity is that complex algorithms cost too much, so more basic solutions are used. Besides that, Solidity code should be deterministic, as it will run on multiple nodes. We need an algorithm that is able to generate a random number once, and use it on multiple nodes. Developers should be aware of this problem because an attacker can be able to predict the result of a lottery or a voting in some specific cases.
The RNG used inside the lottery takes a input a seed, which is composed by *block.number* and the index of a loop. Then, inside the method is computed the height of the block where the current round lottery ends and seed, height of block and K are passed to *keccak256*, which computes the final digest.

$$\text{lottery\_number}_i = \text{hash}(\text{round\_start\_block} + \text{M}, \text{K}, \text{current\_block\_number} + i)$$

Many common and trivial solutions use data from the blockchain itself as inputs for RNGs, such as *block.timestamp*, *block.number*, *blockhash* and so on. Even if RNGs work, they don't provide true random numbers, because it is important to recall that everything on the blockchain is visible and publicly available, so a skilled attacker (e.g. a malicious miner) could be able to create a malicious contract capable of generate the winning numbers of the lottery.
A good solution includes a combination of several pseudo-random data inputs and the use of oracles or smart contracts to make it more reliable. So, it doesn't exist a solution to safely generate true random numbers inside a smart contract. The real solution is to get these true random numbers from the outside world using oracles. An oracle is another contract, which is able to interact with the outside world, gathers data, and so on and injects those data into the blockchain, allowing other contracts to use it.