Lesson 7 - The NFT Shop Challenge

Challenge explanation

- Application Features
  - Buy a ERC20 with ETH for a fixed ratio
  - Withdraw ETH by burning the ERC20 tokens
  - Buy (Mint) a new ERC721 for a configured price
  - Update owner account whenever a NFT is sold
  - Allow owner to withdraw from account
    - Only half of sales value is available for withdraw
  - Allow users to burn their NFTs to recover half of the purchase price
  - Update the NFT Mint price
    - Release from pool when decreasing price
    - Require allowance to top up pool reserves for liquidity when increasing price
- Architecture overview
- Contract external calls

Tests layout

- (Review) TDD methodology
- Best practices on external calls
- Dealing with decimals and divisions
  - Shifting decimal points
  - Underflow
  - Overflow
- (Review) Test syntax
- (Review) Positive and negative tests
- Integration tests

References

https://consensys.github.io/smart-contract-best-practices/development-recommendations/general/external-calls/

https://docs.soliditylang.org/en/latest/types.html#division

https://github.com/wissalHaji/solidity-coding-advices/blob/master/best-practices/rounding-errors-with-division.md

Test code reference

```
import { expect } from "chai";
import { ethers } from "hardhat";
```

```
describe("NFT Shop", () => {
  beforeEach(() => {});

  describe("When the Shop contract is deployed", () => {
    it("defines the ratio as provided in parameters", () => {
      throw new Error("Not implemented");
    });

    it("uses a valid ERC20 as payment token", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When a user purchase an ERC20 from the Token contract", () =>
{
    it("charges the correct amount of ETH", () => {
      throw new Error("Not implemented");
    });

    it("gives the correct amount of tokens", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When a user burns an ERC20 at the Token contract", () => {
    it("gives the correct amount of ETH", () => {
      throw new Error("Not implemented");
    });

    it("burns the correct amount of tokens", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When a user purchase a NFT from the Shop contract", () => {
    it("charges the correct amount of ETH", () => {
      throw new Error("Not implemented");
    });

    it("updates the owner account correctly", () => {
      throw new Error("Not implemented");
    });

    it("update the pool account correctly", () => {
      throw new Error("Not implemented");
    });

    it("favors the pool with the rounding", () => {
      throw new Error("Not implemented");
    });
  });
```

```
  describe("When a user burns their NFT at the Shop contract", () => {
    it("gives the correct amount of ERC20 tokens", () => {
      throw new Error("Not implemented");
    });
    it("updates the pool correctly", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When the owner withdraw from the Shop contract", () => {
    it("recovers the right amount of ERC20 tokens", () => {
      throw new Error("Not implemented");
    });

    it("updates the owner account correctly", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When the owner decreases the Mint price from the Shop
contract", () => {
    it("updates the pool and the owner account after increasing the
price", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When the owner increases the Mint price from the Shop
contract", () => {});

  describe("When there is enough tokens in the pool to cover the costs",
() => {
    it("updates the pool and the owner account after increasing the
price", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When there is not enough tokens in the pool to cover the
costs", () => {
    it("charges the correct amount and updates the pool and the owner
account after decreasing the price", () => {
      throw new Error("Not implemented");
    });
  });

  describe("When an attacker tries to exploit the contract", () => {});

  describe("When transferring ownership", () => {
    it("fails", () => {
```

```
        throw new Error("Not implemented");
      });
    });

    describe("When withdrawing funds", () => {
      it("fails ", () => {
        throw new Error("Not implemented");
      });
    });

    describe("When changing the fee", () => {
      it("fails", () => {
        throw new Error("Not implemented");
      });
    });

    describe("When trying to burn a NFT without giving allowance to it", ()
  => {
      it("fails ", () => {
        throw new Error("Not implemented");
      });
    });

    describe("When trying to buy ERC20 tokens without sufficient ETH
  balance", () => {
      it("fails", () => {
        throw new Error("Not implemented");
      });
    });
  });
```

Completing tests

- Coding patterns and design patterns
- Avoiding Front Running advantages
- Looking out for other attack vectors
- (Bonus) Implementing pausable to prevent front runners to take advantage of privileged information over “normal” users
- (Bonus) Demonstrate the money flow from buying the ERC20, then the NFT, then withdrawing by the owner

References

https://fravoll.github.io/solidity-patterns/

https://dev.to/jamiescript/design-patterns-in-solidity-1i28

https://halborn.com/what-is-a-front-running-attack/

https://quantstamp.com/blog/what-is-a-re-entrancy-attack

https://github.com/wissalHaji/solidity-coding-advices/tree/master/known-attack-patterns

https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/known_attacks/

https://github.com/wissalHaji/solidity-coding-advices/tree/master/design-patterns/security

Homework

- Create Github Issues with your questions about this lesson
- Read the references
- Get to know the Coding Patterns and Design Patterns in the reference
- (Optional) Experiment with dynamic pricing models for buying/selling the ERC20 and NFTs