Smart Contract Challenge Overview

Today's challenge is to build a smart contract that facilitates a token sale. This exercise aims to help you understand the workflow of developing smart contracts, from understanding requirements to coding unit tests and implementing the contract.

Learning Objectives

- Understand project requirements.
- Write unit tests.
- Code a smart contract.
- Collaborate effectively.
- Manage workflow.

Important Considerations

Remember, smart contracts are immutable. Once deployed, bugs cannot be easily fixed. Therefore, caution is paramount during development.

Review of Essential Concepts

We will review:

- Using Hardhat VM.
- Writing smart contracts with it.
- Writing tests.
- Deploying smart contracts.
- Getting wallet information.
- Solidity syntax.
- Building smart contracts.
- Making external calls.
- Performing operations.

Token Sale Contract Details

We will create a contract called `TokenSale.sol`, which allows users to:

- Buy ERC20 tokens by sending ETH to the contract.
- Mint ERC20 tokens on the fly.
- Withdraw ETH by burning tokens at a fixed ratio.
- Purchase NFTs (ERC721) with tokens at a configured price.
- Burn NFTs to recover half the purchase price in tokens, with the other half going to the contract owner.

Token and ETH Interaction

The contract enables interaction with ERC20 tokens. For example, like Uniswap, the contract can manipulate tokens by swapping.

Buying Tokens

Users can buy ERC20 tokens by sending ETH. The conversion is based on a fixed ratio (R), where:

```
1 ETH = R tokens
```

Burning Tokens

Users can withdraw ETH by burning tokens.

Purchasing NFTs

Tokens can be used to purchase NFTs at a fixed price (P).

```
1 NFT = P tokens
```

Working with prices is better than using ratios because of division issues.

NFT Burning and Revenue Sharing

When NFTs are burned:

- The user recovers half of the purchase price in tokens.
- The other half is kept in the contract for the owner to withdraw.

For example, if the price is 5 tokens:

- The owner can withdraw 2.5 tokens.
- The user recovers 2.5 tokens.

Money Flow Summary

1. ETH is used to buy ERC20 tokens.
2. ERC20 tokens are used to buy NFTs.
3. NFTs can be burned to recover half the purchase price in tokens.
4. ERC20 tokens can be burned to recover ETH.

Questions and Clarifications

The smart contract interacts with both ERC20 and ERC721 tokens, minting and burning them based on predefined logic.

User Burning NFTs

If a user burns an NFT, they receive half of the purchase price, while the contract owner receives the other half. The owner can withdraw their portion at any time, not necessarily immediately after the sale.

Contract Architecture

The decision to implement all functionalities within a single smart contract or to divide them into separate contracts using external calls will be discussed.

NFTs and Token Sales

Selling NFTs

- Normal artists or IP brands like Marvel can sell NFTs to sell unique pictures that users can prove they own.
- They can sell them using:
  - Dollars
  - ETH
  - Their own tokens
- Wrapping ETH into tokens simplifies the process, creating a wrapped ETH to buy NFTs.

NFT Floor Price Rebates

- The floor price for an NFT can act as a rebate.
- For example:
  - If an NFT costs $1million, abuyermightget$500,000 back as a guarantee.
- Sometimes NFT floor prices can drop to zero, so this feature can be helpful.
- This smart contract is more for explaining the underlying techniques rather than being financially useful.

Burning Tokens

- Burning tokens: Removing tokens from circulation so that no one can use them anymore.
- Example:
  - Start with 1 ETH to get 10 ERC20 tokens.
  - The price is five tokens for one NFT.
  - After buying one NFT, you have one NFT and five tokens.
  - Burning the NFT results in zero NFTs and 7.5 tokens, because 2.5 tokens are stored for the owner to withdraw later.
- Burning is the opposite of minting.

Project Setup: Token Sale

Creating a New Project

1. Copy everything from Project 5, except `node_modules`, `artifacts`, and `cache` folders.
2. Create a new folder named `token_sale`.
3. Paste the copied files into the new folder.
4. Install dependencies using `npm install`.
5. Compile to ensure everything is working: `npx hardhat compile`.

Verifying Smart Contracts

- Ensure the `20` and `721` smart contracts are present.
- Check the `contracts` folder for `MyNFT.sol` and `MyToken.sol`.
- Run `NPX hardhat compile` to verify.

Package Configuration

- When copying from another project, the `package.json` file might have the wrong name.
- You can manually correct this in the `package.json` file.
- Example:
    - Change the name from `ballot` to `token-sale`.
- This metadata configuration isn't crucial unless you're publishing the project as an npm package.
- Attention to detail and proper naming is appreciated for code organization.

Technical Debt

- Disorganization can lead to **technical debt**.
- Small issues accumulate and consume attention.
- Taking the time to organize and properly name things is valuable.

Creating a Token Sale Contract

New Smart Contract

1. Create a new smart contract file named `TokenSale.sol`.

2. Start with the pragma solidity identifier.

```
pragma solidity ^0.8.20;
```

3. Add the license identifier.

```
// SPDX-License-Identifier: MIT
```

4. Begin the contract definition.

```
contract TokenSale {
}
```

Editor Shortcuts

- Use editor shortcuts for efficiency.

- Examples:

    - Typing `pragma` and hitting tab to autocomplete.
    - Typing `contract` and hitting tab to create a code block.

- Intellisense and using TypeScript can help with accuracy and speed.

- Get used to autocomplete and auto-fixing.

Learning from Errors

Mistakes are valuable for learning. It's important to try things, even if they might be wrong.

Contract Management

Question

Can a contract simultaneously manage an ERC20 and ERC721 token?

Inheriting ERC20 and ERC721

- Inheriting both ERC20 and ERC721 in a single contract can lead to conflicts.
- ERC20 and ERC721 are not designed to be compatible.

Cat and Dog Metaphor

Analogy to explain incompatibility:

> A contract cannot simultaneously be a cat and a dog because they have conflicting characteristics.

Key Takeaway

> Avoid inheriting from a token contract unless you're creating a token.

External Calls

- Working with tokens involves external calls, which can be complex and pose security risks.
- Contracts should be composed, with one calling another.
- This approach is common in blockchain development, such as in DEXes, lending protocols, and games.

Architecture

- Structuring smart contracts with external calls can be more organized and safer.
- Determining where to draw the lines between contracts can be tricky but can be guided by architectural considerations.

Question

Is it possible to put ERC721 before ERC20 in the inheritance?

Answer

The token sale is not a token itself but a contract that sells tokens. Therefore, inheriting a token contract would be problematic.

Testing Smart Contracts

Avoid Direct Coding

Directly coding a smart contract can be confusing and lead to errors, especially regarding input parameters.

Suggestion

Write tests first to define scenarios and guide development.

Human-Readable Tests

Tests should be human-readable and understandable, even by non-technical people.

Sense of Progress

Tests provide a sense of progress as more tests pass, indicating completion percentage.

Testing Scenarios Examples

- When the shop contract is deployed:
    - Define the ratio as provided in the parameters.
    - Define the prices provided in the parameters.
    - Use a valid ERC20 as a payment token.
    - Use a valid ERC721 as an NFT collection.
- When the user burns an ERC20 from the token contract, the correct amount of ETH should be charged.
- The ETH should be sent to the smart contract.
- The correct amount of tokens should be returned.

Testing Smart Contracts

Importance of Testing Methodology

- A structured methodology is suggested for academic purposes due to its ease of understanding.
    - Enables pausing, breakpoint setting, and console logging.
    - Beneficial for constructing smart contracts.
- It is uncommon to use test-driven development methodology.
- In conventional software development, there's a practice of "moving fast and breaking things," which is fine for front ends when testing with end users.
- However, for smart contracts, it is very dangerous and can lead to:
    - Loss of money for users
    - Trouble for the project

  Vibe coding for smart contracts can be dangerous.

- It is advised to use a rigorous testing approach for blockchain software development.
- People using this method in hackathons gain a significant advantage.
- The token sale.ts file is a test file designed to fail from the start and then slowly coded to pass the tests.
- This process involves writing directly for the test and refactoring for better code.

First Test Scenario

- Goal: When deploying the smart contract, the ratio should be defined in the parameters.
    - Example:

```
const ratio = 10;
const price = 5;
```

- Task: Pass these parameters to the smart contracts.

Initial Steps

1. Create a to-do item:

```
// TODO: We would have ratio
const ratio = await tokenSaleContract.read.ratio();
expect(ratio).to.equal(ratio);
```

2. Use a load fixture to deploy the contract.
    - Copy the load fixture from a past project.

Load Fixture

- The load fixture can be copied from past projects like "Hello World" or "Ballot 2."
- It is recommended to keep these projects separate on your machine.
- The load fixture is used to deploy the contract.
- Update the deployContract function to deploy the tokenSale contract instead of the ballot contract.
- Pass the ratio and price as parameters for the token sale.

```
// Instead of ballot contract
const tokenSale = await deployContract("TokenSale", [ratio, price]);
```

- Use public client, deployer, order, account, and token sale contract.

Setting up the Test

- The initial test scenario involves comparing the ratio from the contract with a defined variable.
- The smart contract needs to be deployed first to read the ratio from it.
- A fixture is created to deploy the contract.
- The function deployContract is called to get the token sale contract.

```
const tokenSaleContract = await loadFixture(deployContract);
```

- The fixture can be used to load the token sale contract and compare the ratios.

Compilation and Error Handling

- Compile smart contracts using TypeScript with VM or type chain types with ethers.
- Compiling helps identify problems in the smart contract.
- If the constructor of the token sale does not set the ratio and price, it will result in an error.
- The solution is to implement a constructor with two `uint256` arguments.

  ```
  constructor(uint256 _ratio, uint256 _price) {
      ratio = _ratio;
      price = _price;
  }
  ```

Big Numbers and Contract Visibility

Big Numbers

- Parameters must be large numbers, represented as **big numbers** (e.g., `10n` instead of `10`).
- JavaScript may not handle large numbers natively, requiring **big number notation**.

Contract

```
contract TokenSale {
    // ...
}
```

- `10n` is a **big number**.
- `10` is a standard number.
- Making `uint256 ratio` and `price` **public** allows reading their states.

Floats in the EVM

- There are **no floating-point numbers** in the EVM (Ethereum Virtual Machine).
- The EVM requires deterministic behavior, which is challenging to achieve with floating-point arithmetic.
- Results of divisions or other operations are always **integers**, rounded towards zero.
- Using **prices** may be better than **ratios** due to these limitations.

Reading Contract State

- Marking a `uint256 ratio` as **public** creates a **getter function** automatically.
- As seen in lesson four, **public** variables generate functions for reading their values.

Debugging with Breakpoints

- Use breakpoints to inspect the contract's state during tests.
- The debug console allows for performing operations and examining values.
  - Example: Evaluating `1 + 2` results in `3`.
  - Example: Evaluating `0.1 + 0.2` in JavaScript may not equal `0.3` due to **binary representation issues**.
- **Binary bases (base 2)** can cause rounding issues, especially with numbers like `0.3`, which are not precisely defined in base 2.

Common Computer Science Problems

> Half of the problems in computer science are invented by the computers themselves.

- Separating blockchain problems from general computer science issues is beneficial.

Variable Shadowing

- Variable shadowing can occur when a variable name is reused in a nested scope, potentially leading to unexpected behavior.
- In Solidity, all variables are defined in the same scope, making shadowing a common issue.
- To avoid shadowing, a common practice is to prefix constructor arguments with an underscore (e.g., `_ratio`).

Contract Initialization

- Constructors are used for initial contract configuration.
- Contracts are designed to be flexible, allowing different parameters (e.g., ratios and prices) to be set upon deployment.
- Using a constructor enables the deployment of different contract instances with varying configurations without altering the code.

Immutability After Deployment

- Once a contract is deployed, the parameters set during construction cannot be changed.
- The constructor function is only triggered during deployment.
- There are no functions to modify the ratio and price after deployment.

Reusability of Contracts

- The way the contract is written allows for the deployment of multiple token sales with different parameters, similar to how different ballots with different proposals were deployed.
- Each deployed contract instance operates independently.

Debugging Workflow

- The ability to pause, debug, and log values during testing is crucial for understanding smart contract behavior.
- This structured approach helps in identifying and resolving issues efficiently.

Compilation

- After modifying a smart contract, it needs to be **recompiled** for the changes to take effect in the tests.
- Tests do not automatically compile the contract.

Contract Readability

- If you mark a variable as **public**, Solidity automatically creates a **getter function** for it.
- If you encounter an error related to reading a property (e.g., `cannot read properties of undefined (reading 'ratio')`), ensure the variable is declared as **public**.

Token Interaction in Smart Contracts

Using ERC20 and ERC721 Tokens

The goal is to integrate **ERC20** tokens for payment and **ERC721** tokens for a collection into the smart contract without using inheritance.

Payable Functions

Token Sale contracts should have **payable functions** to receive payments.

Interacting with External Contracts

Instead of making the token sale contract inherit from ERC20 and ERC721, we can interact with these tokens by making external calls.

Role Configuration

We can configure roles within the contract to manage interactions with the tokens, such as minting and burning.

Target Variable Analogy

Recalling Lesson 2 or 3, we can call another contract similarly to the `greetings invoker` example. In that example, we used a variable called `target` to allow for this relationship from one contract to another.

Simplified External Calls

We can simplify external calls using addresses as targets.

Contract Addresses

We can use public addresses for the payment token and NFT contract:

- `paymentToken` (ERC20)
- `NFTContract` (ERC721)

Implementing External Calls

Token Interaction

When we want to perform an action, such as interacting with our token, we would use the token's contract address.

Example: Hello World Analogy

Similar to the Hello World example from the first week, we can use an interface to interact with external contracts.

```
interface IHelloWorld {
    function helloWorld() external returns (string memory);
}

contract MyContract {
    address public helloWorldAddress;

    function invokeGreetings() public returns (string memory) {
        IHelloWorld helloWorldInterface = IHelloWorld(helloWorldAddress);
        return helloWorldInterface.helloWorld();
    }
}
```

ERC20 and ERC721 Interaction

```
interface IToken {
    function something() external returns (something);
}

contract TokenSale {
    address public tokenContractAddress;
    address public nftContractAddress;

    constructor(address _tokenContract, address _nftContract) {
        tokenContractAddress = _tokenContract;
        nftContractAddress = _nftContract;
    }

    function doSomethingWithToken() public returns (something) {
        IToken tokenInterface = IToken(tokenContractAddress);
        return tokenInterface.something();
    }
}
```

Passing Addresses in Constructor

We can pass the addresses of the ERC20 and ERC721 contracts during the token sale contract's deployment. These addresses are then saved within the smart contract.

Future Flexibility

Later, the address of the token contract or NFT contract could be changed if needed.

Deploying Contracts

First, deploy the ERC20 and ERC721 contracts. Then, pass their addresses to the token sale contract during deployment.

Passing Parameters

Passing parameters in the construction is similar to what we did in the ballot. The only complexity here is that we are passing addresses.

Testing Smart Contracts

Passing Token Sale Parameters

To test a **token sale smart contract**, you need to pass specific parameters, including:

- The **address** of a **token contract**.
- The **address** of an **NFT contract**.

Testing with Incorrect Addresses

When testing, you might intentionally pass **incorrect addresses** (e.g., addresses of regular accounts instead of smart contracts) to see how the contract behaves under such conditions.

Verifying ERC20 Token Validity

To verify if a token contract is a valid **ERC20 token**, you can check for the presence of specific functions:

- `balanceOf`
- `totalSupply`
- `symbol`

A basic test involves checking if the contract implements the `totalSupply` function.

Using Hardhat VM to Get Contracts

Instead of deploying a new smart contract every time, you can use Hardhat VM to get a contract at an address. This is useful for interacting with contracts that have already been deployed.

> Hardhat VM: A feature that allows you to read information from smart contracts by attaching to them, without needing to redeploy them.

The function to achieve this is `getContractAt`. It simplifies the process by only requiring the contract name (artifact name) and the address.

```
const tokenContract = await VM.getContractAt("MyToken", tokenContractAddre
```

Testing Total Supply

After obtaining a contract instance, you can test its functions, such as `totalSupply`. For example, you can check if the total supply at deployment is zero.

```
const totalSupply = await tokenContract.read.totalSupply();
expect(totalSupply).to.be.greaterThanOrEqual(0);
```

This test might fail if the provided address does not correspond to a valid smart contract, causing the function to return unexpected data (e.g., `0x`).

Identifying the Contract Name (Artifact)

The contract name used in `getContractAt` is the name of the artifact (the compiled contract), not necessarily the name of the token itself. This name can be found in the contract's artifact file.

Code Sharing and Exercises

While the complete code will be shared in future lessons, it's important to practice typing out the code to better remember and understand it.