Lesson 10/11 Study Guide

This study guide covers the continuation of the token sales challenge, focusing on divisions, patterns, and the use of big integers in JavaScript.

Challenge Continuation

We're picking up where we left off in Lesson 10, specifically addressing the issues with the constructor and how it interacts with token contracts.

Addressing the Constructor Issue

- Previously, the constructor was using a **wrong address** for the token contract.
- This caused issues when trying to call functions like `totalSupply` because the address didn't correspond to a valid smart contract.
- The error manifested as a failure to call the `totalSupply` function, among others.

Solution: Deploying a New Token Contract Instance

- To resolve this, we deploy a **new instance** of the token contract (`MyToken`).
- This ensures we have a valid contract address to pass to the token sale contract.
- The deployed contract's address is then passed as a parameter to the constructor function of the token sale contract.
- Similarly, a new instance of the NFT contract (`MyNFT`) is deployed and its address is used.

```
paymentTokenContract = await vm.deploy(MyToken);
nftContract = await vm.deploy(MyNFT);
```

Clarifying Contract Interactions

- For now, we are only **storing the addresses** of these contracts inside the token sale contract.
- The actual function calls will happen later.
- Storing the contract address involves passing the deployed contract as a parameter for the construction function of the token sale.

Pedagogical Note on Typing Code

- Actively typing code can significantly enhance retention and understanding.
- While it may slow down the immediate pace of learning, it reinforces the concepts in memory.
- It's recommended to focus on understanding during the live session and then type the code later while reviewing the recording.

Unit Testing and Production Considerations

Unit Testing Environment

- Unit tests create a **controlled environment** to quickly test contracts.
- Contracts like `MyToken` and `MyNFT` are deployed specifically for the test.

Production Environment

- In production, some contracts may already exist.
- It's the developer's responsibility to **track the proper addresses** of these existing contracts.
- Tools like `scaffold-eth` can help manage deployments and track addresses.

Big Number Comparison

- Ensure that when comparing numbers, especially `totalSupply`, comparisons are valid.
- This involves making sure the `totalSupply` (a big number) is greater than or equal to zero.

ERC20 Payment Token Verification

Initial State Validation

- To ensure the smart contract functions correctly, verify the initial `totalSupply` is zero.
- Remove initial minting on line 11 in the smart contract.
- Recompile the smart contract.

Basic Validation Test

- Check if the smart contract is a valid ERC20 payment token by confirming that `totalSupply` returns zero.
- Remember, this is a basic test; a comprehensive test would involve more checks.

Debugging Tips

- If you encounter issues reading the token contract's `totalSupply`, ensure:
  - The smart contract has been compiled.
  - The compiled contract is added to the fixture in the test suite.
- Use breakpoints in your testing environment to inspect contract addresses and deployed contracts.

Interface Compliance

- It was asked if there is a way to verify a contract type like ERC20 721 when getting the address.
- There is no native on-chain type check to confirm that a contract adheres to a specific interface.

> The **interface problem** is a known issue, where contracts may not behave as expected, leading to potential losses if tokens are sent to incompatible contracts.

- CLI tools can be used to check compliance with ERC standards, such as the tool used in lesson 24.
- Programmatic on-the-fly type checking via addresses is difficult.

Call Expectations

- Test contract calls using `expect` to verify if they are rejected or not.

  ```
  expect(tokenContract.name).not.to.be.reverted;
  ```

  - This tests whether specific signatures are present in the contract.
- Similar tests can be performed for ERC-721 contracts, checking for NFT collections, token owners, and token URIs.
- Forgetting `await` when calling contract functions is a common mistake.

Smart Contract Structure

Contract Interaction

- The token sale contract interacts with other contracts via addresses and interfaces.
- Later, the type will be changed from **address** to **interface** to simplify interactions.

Test Structure

- Test structure involves one contract token sale interfacing with all the contracts by address here by interface.

Payment Functionality

Function Outline

- Implement tests for:
  - Charging the correct ETH amount when a user buys ERC20 tokens.
  - Giving the correct amount of tokens.
- Discussed coding a `buyTokens` function and calling it from test scenarios.
- The plan is to code a function to purchase buy tokens and call it from those test scenarios.

Fixtures

- It is possible to use a `beforeEach` hook as a fixture to call before each test function for better test structure.
- For expediency, the example code is implemented directly within the test scenario.

Function Definition

- Asking for contributions and suggestions on how to code the function.
- Need to implement a function in the token sale to purchase tokens.
    - How would you code this function?

Considerations for `buyTokens` Function

- How to charge ETH from the buyer?

Suggestions for Function Definition

- Here are some suggestions on how to define the buyTokens function:

Suggestion	Details
Visibility Modifier	`public` for external access
Payable	`payable` modifier to accept ETH
Parameters	`uint256 amount` (quantity of tokens to buy)
External	`external` the contract should be called from outside
Address	Address of the buyer

Payable Function

- When a function is payable, the person sending the transaction specifies how much they want to pay.
- Amount is available via `msg.value`.

Buying Tokens with Native Tokens

Using `msg.sender` and `msg.value`

- When handling native tokens, the `msg.sender` contains the address of the buyer.
- The `msg.value` contains the amount of native tokens (e.g., ETH) being sent.
- The contract needs to:
    - Determine how many tokens to give to the buyer.
    - Ensure the correct amount of tokens is given.

Buying Tokens for Others

- In the default implementation, the buyer receives the tokens themselves.
- It is possible to implement functionality where someone pays for another address to receive the tokens.
    - Example: Uniswap allows specifying a different receiver.

Storing ETH for Burning Tokens

- The ETH received during the token sale is stored within the contract.
- This ETH is used later when users burn their ERC20 tokens to receive ETH back.
- A `burnTokens` function will be implemented later for this purpose.

Removing Quantity from Input

- The quantity of tokens to be minted is determined by the amount of ETH sent (`msg.value`) and a predefined ratio.
- The amount to be minted is calculated as:

$$amount_to_be_minted = msg.value * ratio$$

    - The **ratio** (r) means that each ETH buys r tokens. For example, if the ratio is 10, one ETH buys 10 tokens.

Relation to Gas Fee

- The `msg.value` is **not related** to the gas fee.
- It represents the amount being paid for the function call.

Testing the Smart Contract

Setting Up the Test

1. Pick the token sale contract from the fixture.
2. Use an account other than the deployer account for testing.
3. Create a transaction to call the `buyTokens` function.

```
transaction = await tokenSaleContract.write.buyTokens({ value: amoun
```

Specifying Ether Amounts

- Use `parseEther` to convert a string representation of ether to a big number.

```
import { parseEther } from "viem";
const amount = parseEther("1"); // represents 1 ether
```

Testing Token Purchase

- Test buying tokens by sending one ether to the contract.

```
const testBuyTokens = parseEther("1");
```

- Include the value in the transaction options.

```
transaction = await tokenSaleContract.write.buyTokens({
  value: testBuyTokens,
});
```

- Specify the account to use for the transaction.

```
transaction = await tokenSaleContract.write.buyTokens({
  value: testBuyTokens,
  account: allAccounts[1],
});
```

Verifying Token Balance

1. Get the token balance before the transaction.

```
const tokenBalanceBefore = await paymentTokenContract.read.balanceOf(
  accountAddress,
]);
```

2. Get the token balance after the transaction.

```
const tokenBalanceAfter = await paymentTokenContract.read.balanceOf([
  accountAddress,
]);
```

3. Calculate the difference in token balance.

```
const difference = tokenBalanceAfter - tokenBalanceBefore;
```

4. Expect the difference to be the amount of tokens that should have been minted based on the message value and ratio.

Important Notes for Public Testnets

- When testing on public testnets, it is important to wait for the transaction to be confirmed before proceeding.

```
await publicClient.waitForTransactionReceipt({ hash: transactionHash
```

- This ensures that the transaction is included in the blockchain before checking balances or performing other operations.

Reading Token Balance

- Use the `balanceOf` function of the token contract to get the token balance for an address.

~~~
await paymentTokenContract.read.balanceOf([account.address]);
```## Token Purchase and Minting
~~~

Calculation of Tokens

When a buyer sends **1 ETH** to the contract, the contract is expected to mint tokens for the buyer based on a predefined **ratio**.

- The initial ratio is **10**.
- The formula for calculating the number of tokens to be minted is:

```
Tokens Minted = Amount Paid * Ratio
```

  Where:

  - `Amount Paid` is the amount of ETH sent by the buyer.
  - `Ratio` is the predefined ratio for token minting.

For example, if someone sends 1 ETH with a ratio of 10, they should receive 10 tokens.

Testing Token Purchases

- **Test Structure**: The test involves checking the difference in tokens before and after the purchase to ensure the correct amount of tokens is minted.
- **Atomic Testing**: It is recommended to keep tests atomic, focusing on one specific aspect to make debugging easier.

Testing Considerations

Gas

- Gas considerations are not included in the initial testing phase.
- The focus is primarily on verifying the correct amount of tokens received.
- Gas usage will be addressed in later stages of testing.

Ether Balance

- There was a suggestion to include a test case that verifies if the contract stores the correct amount of ETH after a transaction.
- It was decided that such a test could be created separately to maintain atomic testing principles.

Before Each Hook

The `beforeEach` hook is a function that triggers before each test.

- It is used to set up the initial state for each test.
- It can be used for:
  - Deploying smart contracts.
  - Initializing variables.
  - Performing setup transactions.

Fixtures

- Fixtures are commonly used in conjunction with `beforeEach` hooks.
- Due to the architecture of fixtures, they often involve rewinding snapshots.
- Code in `beforeEach` may be repeated for each test to ensure a clean state.

Smart Contract Implementation

Minting Tokens

To create new tokens, the `_mint` function is typically used.

> _mint: A function that creates new tokens. It is private, meaning that it can be only be used internally within the contract or contracts that inherit from it.

- Since `_mint` is private, it cannot be called directly in the token sale contract.
- A public mint function in the token contract is used to call the internal `_mint` function.

Calling Smart Contracts

- Instead of using low-level calls, an interface can be used to interact with smart contracts.
- Importing the token contract interface (e.g., `myToken`) allows calling functions by name.

Example

```
// Importing the token contract interface
import { MyToken } from './MyToken.sol';

contract TokenSale {
    MyToken public tokenContract;

    constructor(MyToken _tokenContract) {
        tokenContract = _tokenContract;
    }

    function buyTokens() public payable {
        uint256 amountToMint = msg.value * ratio;
        tokenContract.mint(msg.sender, amountToMint);
    }
}
```

Parameters for Minting

The `mint` function requires two parameters:

1. Address: The address to mint the tokens to (`msg.sender`).
2. Amount: The amount of tokens to mint.

Debugging Minting Errors

Identifying the Missing Role

When a smart contract function is restricted to a specific role, such as a minter role, only accounts with that role can execute the function. An error occurs if an unauthorized account attempts to call it.

Debugging Process

1. Error Observation: The smart contract reverts with a custom error indicating access control issues.

   > AccessControl: unauthorized account `account address` is missing role `minter role hash`.

2. Debugging: Use debugging tools to identify which account is missing the required role.

   - In this case, the token sale contract was missing the minter role, preventing it from minting tokens.

Granting the Minter Role to the Token Sale Contract

To resolve the minting error, the minter role must be granted to the token sale contract.

1. Location: Grant the minter role as part of the setup in the deploy contracts fixture.
2. Implementation:
   - Retrieve the minter role hash.
   - Use the `grantRole` function to grant the minter role to the token sale contract.
   - Wait for the transaction to be confirmed.

```
// Retrieve the minter role hash
const MINTER_ROLE = ethers.keccak256(ethers.toUtf8Bytes("MINTER_ROLE"));

// Grant the minter role to the token sale contract
const grantMinterRoleTx = await paymentTokenContract.grantRole(MINTER_ROLE
await publicClient.waitForTransactionReceipt({ hash: grantMinterRoleTx.has
```

Casting Contract Types

When working with different contract types in a constructor, you may encounter compilation errors due to type mismatches. In such cases, you can use casting to convert the address type to the expected contract type. This involves updating the Application Binary Interface (ABI) to ensure proper interaction with the contract functions.

```
// Correct way to cast address types to contract types
constructor(MyToken _tokenContract, MyNFT _nftContract) {
    tokenContract = MyToken(_tokenContract);
    NFTContract = MyNFT(_nftContract);
}
```

Gas Cost Accounting

Calculating ETH Balance Differences

To ensure accurate accounting of ETH transactions, especially when dealing with payable functions, it's essential to consider gas costs.

1. Record ETH Balance: Get the ETH balance before and after a transaction.

```
// Get ETH balance before the transaction
const ethBalanceBefore = await publicClient.getBalance({ address: account.

// Execute the transaction

// Get ETH balance after the transaction
const ethBalanceAfter = await publicClient.getBalance({ address: account.a
```

2. Calculate ETH Difference: Subtract the ETH balance after the transaction from the ETH balance before the transaction.

$$ETH_{difference} = ETH_{before} - ETH_{after}$$

3. Account for Gas Costs: Deduct the gas costs from the ETH difference to get the actual value transferred.

$$ETH_{transferred} = ETH_{before} - ETH_{after} - Gas_{costs}$$

Determining Gas Costs

Gas costs are incurred due to the computational resources used during transaction execution. To accurately account for gas costs:

1. Retrieve Gas Usage: Obtain the gas used and the effective gas price from the transaction receipt.

```
// Get transaction receipt
const receipt = await publicClient.getTransactionReceipt({ hash: transacti

// Get gas used
const gasUsed = receipt.gasUsed;

// Get effective gas price
const gasPrice = receipt.effectiveGasPrice;
```

2. Calculate Total Gas Costs: Multiply the gas used by the effective gas price to determine the total gas costs.

$$Gas_{costs} = Gas_{used} * Gas_{price}$$

Gas Accounting and Testing

When testing smart contracts, it's crucial to account for the gas used by transactions. The cost is determined by multiplying the gas amount by the gas price.

Testing Gas Usage

To verify that a contract is charging the correct amount of ETH, ensure the test checks for the value to buy tokens plus the gas cost for that transaction.

Cumulative Gas vs. Gas Used

- Cumulative Gas Used: Gas used for all transactions up to a specific transaction in a block.
- Gas Used: Gas used by a specific transaction.
  - Use 'gas used' rather than 'cumulative gas used' to check gas usage.

Token Sale Contract

The token sale contract is similar to the one from the previous lecture but includes updated interfaces.

Buy Token Function

The `buyToken` function contains a single line of code.

Potential Errors

- TypeError: cannot convert undefined to a BigInt: This error might occur if the contracts are not compiled or if there's an issue with the contract's code.
- Ensure the contract is correctly importing the `MyToken` contract used earlier.
- Verify that the `Minter` role is properly handled, either hardcoded or obtained dynamically.

Learning Process

The focus of the lesson is not just to complete the token sale, but to understand the process of deploying a smart contract.

Consider these steps when deploying a smart contract:

1. Think about the cases.
2. Write the tests.
3. Discover requirements such as needing a `Minter` role, external calls, or inheritance.

Implementing Burn Tokens Function

Let's consider how to implement a `burnTokens` function, similar to the `buyTokens` function.

Function Signature

```
function burnTokens(uint256 amount) public external {

}
```

- `public external`: Makes the function accessible.
- `amount`: Specifies the number of tokens to be burned.
- `payable`: Functions in Solidity can only receive ETH directly. There is no generic "payable address" for tokens.

Calculating ETH to Return

Consider a scenario where users can withdraw ETH by burning ERC20 tokens. If the ratio is 10:1 (1 ETH buys 10 tokens), burning one token should return 0.1 ETH. The amount of ETH to return can be calculated as `amount / ratio`.

Decimals Issue

Using ratios can lead to issues with decimals. For example, if 1 ETH buys 3 tokens, returning one token results in fractions of ETH (e.g., 0.333...). Repeated transactions can cause small discrepancies due to these fractional amounts.

```
Example:
Buy 3 tokens with 1 ETH.
Return 1 token: Get back 0.333... ETH.
Return another token: Get back 0.333... ETH.
Return the last token: Get back 0.333... ETH.
```

The small, remaining amounts can accumulate and cause problems, such as in liquidity pools.

Avoiding Divisions

Divisions are dangerous in smart contracts.

Consider using **price per token** instead of ratios. This allows calculations to be done using addition, subtraction, and multiplication, which are safer.

Burning ERC20 Tokens

When working with ERC20 tokens, particularly when implementing functionalities like burning, there are several important considerations:

Ratios and Precision

- When dealing with ratios in smart contracts, especially in burning mechanisms, be aware of potential issues with **truncation**.
- If a ratio results in a fractional number, Solidity might truncate the decimal part, leading to a loss of precision.
- Example: If the ratio is 10 and you attempt to burn 9 wei (a small fraction of a token), the contract might return 0 due to truncation, effectively losing the burned tokens.
- This can open vectors for attacks or errors.

ERC20 Burnable Extension

- To enable token burning, you can import the `ERC20Burnable` extension from the OpenZeppelin library.

  ```
  import "@openzeppelin/contracts/extensions/ERC20Burnable.sol";
  ```

- By importing and implementing the `ERC20Burnable` extension, your contract inherits the `burn` and `burnFrom` functions.
  - `burn`: Removes tokens from the caller's balance and the total supply.
  - `burnFrom`: Allows a spender to burn tokens from a specified account, subject to allowance.

Burning Implementation Example

- To implement a token burning mechanism, you'll need a function in your contract that leverages the `burnFrom` function.

- This function will take an amount as input, burn tokens from the message sender (`msg.sender`), and potentially transfer a corresponding amount based on a predefined ratio.

  ```
  function burnTokens(uint256 amount) external {
      require(token.allowance(msg.sender, address(this)) >= amount, "In
      token.burnFrom(msg.sender, amount);
      payable(msg.sender).transfer(amount / ratio);
  }
  ```

Testing Burn Functionality

- To ensure the burn functionality works correctly, you need to write comprehensive tests.
- Before burning tokens on behalf of a user, the contract needs to be approved to spend tokens on their behalf. This is done using the `approve` function from the ERC20 standard.

```
const approveTransaction = await paymentToken.connect(account).approv
await approveTransaction.wait();
```

- After approving the contract, you can call the burn tokens function.
- The test should verify that the correct amount of tokens is burned and that the user's balance is updated accordingly.

```
const burnTransaction = await tokenSale.connect(account).burnTokens(a
await burnTransaction.wait();
expect(await paymentToken.balanceOf(account.address)).to.equal(0);
```

Key Points for ERC20 Transactions

- With ERC20 tokens, you don't send tokens directly to a smart contract using payable transactions.
- Instead, you must first approve the smart contract to spend tokens on your behalf.
- This is done using the `approve` function, which sets an allowance for the contract to spend your tokens.
- Once the contract has the necessary allowance, it can transfer or burn tokens on your behalf.

ERC721 Considerations

- When burning ERC721 tokens (NFTs), a similar approval process is required.
- You need to approve the contract to burn the specific NFT before initiating the burn transaction.
- This ensures proper accounting and authorization for financial operations involving ERC20 tokens and NFTs.