Introduction to Voting Ballot Smart Contract

In this lesson, we will dive into the **voting ballot smart contract** (`ballot.sol`) using the tools and concepts we learned in previous lessons. We'll explore Solidity syntax, including **mappings**, **arrays**, and **structs**. This smart contract serves as a practical demonstration for understanding how voting can be implemented on the **EVM** (Ethereum Virtual Machine).

Contract Overview

The `ballot.sol` smart contract facilitates a voting process where:

- A contract is created for each voting period.
- Each **proposal** has a short name.
- The contract creator grants voting rights to specific **addresses**.
- Individuals can **vote** or **delegate** their vote to someone else.
- Anyone can retrieve information about the **winning proposal**.

Setting Up the Project

Instead of repeating the entire Hardhat setup, we'll use a "cookie cutter" approach by copying necessary files from a previous project (e.g., `project2`).

Steps:

1. Create a new folder for the project (e.g., `project3`).
2. Copy all files from `project2` to `project3`, **except** for the `node_modules` folder and `package-lock.json` file.
    - **Note**: Avoid copying `node_modules` to save time.
3. Ensure hidden files (e.g., `.eslintrc.json`, `.gitignore`) are visible and copied.
4. Navigate to the new project directory in the terminal.
5. Run `npm install` to recreate the `node_modules` folder and `package-lock.json`.

Cleaning Up

You can clean up unnecessary files from the copied project, such as the old contract (`hello.sol`) and its test file.

Running Tests

To ensure everything is set up correctly, run the tests using:

```
npx hardhat test
```

This command compiles the smart contracts and executes the tests.

Testing with Hardhat CLI vs. Mocha Test Explorer

When testing, it's important to understand the differences between using the Hardhat CLI and Mocha Test Explorer:

- **Hardhat CLI**: Automatically compiles smart contracts before running tests.
- **Mocha Test Explorer**: Does not automatically compile smart contracts, potentially leading to errors if contracts are not compiled.## Completing an Incomplete Assignment
- For the incomplete assignment, you need to **change the owner**, **transfer ownership**, and **set text**.
- You don't need the last block of code to complete it.
- If you want feedback, open a **GitHub issue** and share your code there.

Setting Up the Environment

- Create a **new folder** for the new contract.
- Delete the other contracts and test files to avoid confusion.
- Create a new contract named `Ballot.sol`.
- Clean up with `npx hardhat clean`.
- Copy the code from Solidity by Example and paste it into your `Ballot.sol` file.

Ballot Contract Overview

- The `Ballot` contract is for implementing voting with delegation.
- The lecture goes over the contract's syntax using tests.

`struct` Keyword

- The smart contract makes use of the `struct` keyword, which is introduced for the first time in this lecture.

Chairperson

- `address public chairperson` is defined on line 21 and set in the constructor on line 32 as the deployer, similar to the owner.
- The **chairperson** has privileges, such as giving the right to vote.
- The **MSG.sender** of the constructor is the person deploying the smart contract, which makes them the chairperson.
- The person signing the transaction for the deployment of the smart contract is the chairperson.

MSG Global Variable

> MSG is a global variable available in all EVM (Ethereum Virtual Machine) blockchains.

- A list of all global variables can be found in **Lesson Three** under **Blockchain Global Variable References**.
- When listening, you capture only 10-20% of things, but when you write, build, or explain something, you learn 70-90%.

Structs, Mappings, Arrays, and ABI

- This lesson goes over **structs**, **mappings**, **arrays**, and **ABI study** using the exercise.
- The tests explain how the `giveRightToVote`, `delegate`, `vote`, and `winningProposal` functions work.

Test File: ballot.ts

- Create a test file named `ballot.ts`.
- The test file can have any name.
- It's a good idea to name smart contract files the same as their artifacts.
- Here is the starting coding block:

```
// imports, todos, etc.
```

Proposals Array

- An array of proposals is defined with short names for each option.

  Short Name: Bob, Lucas, Matos

Testing Smart Contracts

Test Isolation

- Tests are run separately to prevent **interference**.
- This is achieved through **encapsulation**.
- Encapsulation allows for testing specific scenarios in isolation.
- Allows you to go back in time to rerun tests.

Visual Test Execution

- The lecturer uses a test explorer to visualize test execution.
- Initially, all tests fail because no implementation exists.

Connecting Solidity and TypeScript

- Solidity files (.sol) cannot be directly imported into TypeScript (.ts) files.
- They operate in different environments and require a bridge.

Bridging the Gap

- Hardhat is used to create a local Ethereum environment.
- It communicates via JSON RPC.

Deploying Contracts

- The `deployContract` function from Hardhat is utilized within the VM.
- This method simplifies the deployment process, requiring only the artifact name.

Hidden Files

- Hidden files (starting with a dot) may not be copied during file transfers.
- Common hidden files: `.mocharc.json`, `.env`, `.gitignore`

Cleaning Up

- Clean up the cache (deleting the solidity file cache JSON) after making changes to solidity files.

Deploying Contracts with Hardhat VM

- To deploy a smart contract with Hardhat VM, you can use:

```
vm.deployContract
```

- This requires specifying the artifact name.

Deploying and Interacting with Smart Contracts Using Hardhat

Deploying Contracts

- To deploy a smart contract using Hardhat, the `VM.deploy` function is used, passing the contract name and parameters.

```
ballotContract = vm.deployContract("Ballot", parameters);
```

- The contract must be compiled first, which generates an **artifacts** folder containing necessary files.
- The contract name passed to `vm.deployContract` must match the name of the compiled contract. Otherwise, Hardhat will throw an error.

Interacting with Deployed Contracts

- The `read` function is used to access the state of the blockchain within the Hardhat environment.
- Calling `read` on a contract instance allows access to the contract's members, such as **chairperson**, **proposals**, **voters**, **winnerName**, and **winningProposal**.

```
ballotContract.read.chairperson();
```

- `read` retrieves data directly from Hardhat's blockchain state, simulating a view function. If the blockchain is offline or the value hasn't been updated, `read` will not reflect the changes.

Understanding Promises and `await`

- Smart contract interactions in JavaScript/TypeScript are **asynchronous** and return **promises**.
- A promise represents a value that might not be available immediately.
- To work with the result of a promise, the `await` keyword is used to pause execution until the promise resolves.
- Without `await`, TypeScript will continue executing without waiting for the promise to resolve, leading to unexpected behavior.

```
const chairperson = await ballotContract.read.chairperson();
console.log(chairperson);
```

Getting Wallet Clients

- `getWalletClients` retrieves Ethereum accounts with the ability to sign and pay for transactions within the Hardhat Runtime Environment (HRE).
- These accounts are pre-funded in the HRE.

```
const [deployer, otherAccount] = await hre.getWalletClients();
```

Testing Contract Interactions

- Testing involves comparing expected values with actual values retrieved from the contract.
- In the example, the chairperson's address is compared with the deployer's account address.
- Casing matters when comparing addresses; checksums may result in uppercase characters.

```
expect(deployer.account.address).to.equal(chairperson);
```

Debugging with Breakpoints

- Breakpoints can be used to pause execution and inspect variables.
- Debugging tools like the Moes explorer help in examining the state of promises and variables.
- Console logs can also be used for debugging, especially when the Moes test explorer is not working.

Key Concepts Recap

Concept	Description
`vm.deployContract`	Deploys a smart contract to the Hardhat network.
`read`	Reads data from the blockchain state.
`await`	Waits for a promise to resolve before continuing execution.
`getWalletClients`	Retrieves pre-funded Ethereum accounts for testing.
Asynchronous	Operations that do not block the execution of other operations.
Promises	Represents the eventual result of an asynchronous operation.

Debugging Tips and Troubleshooting

- When debugging, set a **breakpoint** after the line of code you want to inspect to see the calculated value.
- To set a breakpoint in VS Code:
  - Move your mouse slightly to the left of the line number.
  - Click to create a red dot, indicating the breakpoint.
  - Right-click and select "Debug Test" to start debugging.
- If you encounter an issue where the test appears to be running indefinitely, look for a panel at the top of VS Code to disconnect or stop the test.
- Ensure the Test Explorer extension is properly loaded in VS Code by reloading it if necessary.

Organizing Tests with Fixtures

The Problem of Repetitive Code

When writing tests, you might find yourself repeating the same setup steps, such as deploying contracts, over and over again. This is not efficient.

Introducing Fixtures

> **Fixtures**: Functions that set the desired state of the network before each test. They allow you to avoid repeating code and make your tests more maintainable.

How Fixtures Work

1. You define a **fixture function** that contains the code for deploying contracts and setting up the test environment.
2. The fixture function returns all the necessary objects and variables.
3. You use the `loadFixture` function to call the fixture at the beginning of each test.
4. `loadFixture` takes a **snapshot** of the blockchain state after the fixture is executed.
5. For subsequent calls to the fixture, the blockchain is reset to the snapshot state, avoiding the need to redeploy contracts.

Benefits of Using Fixtures

- **Code Reusability**: Avoid repeating the same setup code in every test.
- **Faster Tests**: Resetting to a snapshot is faster than redeploying contracts for each test.
- **Maintainability**: Easier to update the setup logic in one place (the fixture) instead of in every test.

Example

Instead of deploying the contract and picking the deployer in each test:

```
// Without fixture
const ballotContract = await deployBallot();
const deployer = await getDeployer();
```

You can use a fixture:

```
// With fixture
const { ballotContract, deployer } = await loadFixture(deployContract);
```##  Testing Environment and Debugging

-   The sandbox testing environment can be used as a teacher to discover a
-   It's essential to test claims and promises in web3 due to marketing hy
-   You can explore different token standards, protocols, and SDKs within

Understanding Fixtures

-   A **fixture** returns an object, which can be inherited in subsequent
-   The plContract fixture returns an object that can be inherited in othe
-   When returning something in a fixture in TypeScript, it returns a set
-   TypeScript provides syntaxes for objects (using curly brackets) and ar

Object Deconstruction

-   You can deconstruct objects to pick specific members.

    ```typescript
    const { ballotContract, deployer } = fixture();
    ```

    This ignores other members and picks only `ballotContract` and `deploy

Array Handling

-   Arrays can be destructured similarly.

    ```typescript
    const [deployer, secondAccount, thirdAccount, ...otherAccounts] = acco
    ```

    -   `deployer` is the first member.
    -   `secondAccount` is the second member.
    -   `thirdAccount` is the third member.
    -   `otherAccounts` is an array containing the remaining members.

Determining Array Length

-   To find the length of an array in real-time (not during debugging), yo
-   Use `getWalletClients` to get the array length of available accounts.

Limitations with Arrays in Solidity and EVM

-   Arrays in Solidity cannot be retrieved directly from the blockchain du
-   Arrays could have thousands of members, making it impractical to retur
-   Instead, you need to pick elements proposal by proposal using their in
-   To check the whole array, you would need to iterate through each posit

Complexities of Retrieving Proposals

-   Arrays in Solidity can't be directly accessed, you must access array e
-   The returned proposal is not just a string, but an object structured a
-   The `proposal` type is defined within the smart contract, similar to a

````
## Structs and Data in the Blockchain

### Structs as Bundled Data
- A **struct** is a structure of data in the blockchain, which carries mul
- Example:
  - Two positions are defined in a struct as 17 and 18.

### Comparing Objects
- Comparing objects requires accessing specific members within the struct.

### Complexities in Proposal Comparisons
1.  Cannot directly retrieve all proposals from an array.
2.  Proposals are not directly strings but objects.
3.  The name of a proposal is in **bytes32** format and must be converted

### Bytes32 and Hex Conversion

-   Data is stored in the blockchain as hexadecimal.
-   Proposals need to be converted to **hex** format before being passed.
-   Use a utility function to convert from **hex to string** for compariso

    ```js
    hexToString(proposal[0]);
    ```

### Example: Hex to String Conversion

-   Converting a hex string of "Proposal One" with appended zeros:

    ```js
    hexToString(proposal); // Output: "Proposal One\u0000\u0000\u0000..."
    ```

-   Fixing the size to 32 bytes ensures correct conversion without appende

### Code Example: Testing Proposal Strings

```js
test('should test all proposal strings', async () => {
    for (let i = 0; i < 3; i++) {
        const proposal = await ballotContract.proposals(i);
        expect(hexToString(proposal[0], 32)).to.equal(`Proposal ${i + 1}`)
    }
});
````

Limitations of Byte Size
````

- The size is limited to **32 bytes** because it is the maximum size that can be held in a certain string in a blockchain without breaking.
- Storage layout in blockchain:

    In each storage location, up to 32 bytes can be stored.
- Smart contracts use **bytes32** to store short names up to 32 bytes.

Debugging with TypeScript

- When debugging, the **hex string** is a JavaScript library.
- TypeScript configurations use **ECMAScript**.
- TypeScript provides illusions and mirages because the output is transpiled to JavaScript.
- Debugging may not have the same support as TypeScript due to the transpilation to JavaScript.
- TypeScript is a super set of JavaScript.

Mappings

Introduction to Mappings

- Mappings associate keys to values.
- Differ from arrays, which go from a number to an object.
- Mappings can go from anything to anything, a key-value relationship.

Voter Mapping Example

- Define a **voter** object with properties like:
    - `votingWeight`
    - `hasVoted`
    - `delegatedTo`
    - `proposal`
- Create a mapping for every address in the blockchain to a voter.

Key-Value Structure

> Unlike arrays which use numbers to objects, mappings can use any data type as a key (string, number, Boolean, or even another mapping).

Code Example: Setting Voting Weight

```
it('should set the voting weight for the chairperson as 1', async () => {
    const chairpersonVoter = await ballotContract.voters(chairpersonAddres
    expect(chairpersonVoter.votingWeight).to.equal(1);
});
```

Key points about mappings

- Mappings can answer even if you didn't define the initial value.
- The voters are read at the address of the chairperson, not at position zero.

Ballot Contract Address

The **ballot contract address** can be any address, including the zero address, your address, or a random address. When querying a **mapping** with an address, the mapping will always return a value, even if the address hasn't been explicitly defined in the contract.

Mapping Behavior

Mappings differ from arrays in how they handle undefined keys:

- **Arrays**: Accessing an out-of-bounds index will result in an error.
- **Mappings**: Accessing any address in a mapping will always return a value.

> If the address hasn't been explicitly defined, the mapping returns the **default values** for each data type: `false` for Booleans, empty for strings, and `0` for numbers.

Example: Voter Mapping

Consider a mapping of voters in a ballot contract:

```
mapping(address => Voter) public voters;
```

Querying this mapping with an arbitrary address will always return a `Voter` struct. If the address hasn't been assigned a voter, it will return a `Voter` struct with the default values for each field (e.g., `weight = 0`, `voted = false`, etc.).

Chairperson Voter Example

Even if the chairperson voter hasn't been explicitly defined in the smart contract, querying the mapping for the chairperson's address will still return a voter object, populated with default values.

Key Differences Between Arrays and Mappings

Feature	Arrays	Mappings
Accessing elements	Requires a valid index within bounds	Accepts any address
Error handling	Out-of-bounds access throws an error	Always returns a value (default if not explicitly set)

Homework Assignment

Complete as many tests as possible to prepare for tomorrow's lesson. Ensure your environment is set up with:

- A **wallet** with funds.
- An **RPC connection**.
- Access to a **testnet** for deploying and interacting with smart contracts.

Tomorrow's lesson will involve deploying and interacting with smart contracts on a public testnet.