Lesson 3: Solidity Smart Contract Syntax and Structure

We will continue exploring the syntax and structure of Solidity smart contracts, building upon the `hello.sol` exercise from the previous lesson. This will provide a practical understanding of key concepts, including Solidity types, memory architecture, storage architecture, accounts, and control structures.

Review of Key Concepts

State Variables

- A **state variable** is a variable that:
  - Exists outside of a function.
  - Persists in the blockchain.
  - Impacts the way a function has to be set up to interact with it.
  - Costs gas to change.

State Mutability

- To modify a state variable, a function needs to have a specific **state mutability**.
  - `non-payable` is needed to modify a variable stored on the blockchain.

    > Even though you pay for the transaction, you are not paying the smart contract. You are paying for updating that single text, which is stored in the blockchain.

  - `view` and `pure` are not sufficient to change a state variable.

Interacting with Smart Contracts

- You can interact with smart contracts without the source code using the **Application Binary Interface (ABI)**.
- The easiest way to interact with the ABI is through an **interface**.
    - You can use an interface for:
        - Conformity
        - Connecting to deployed smart contracts.

Questions and Answers

Pure vs. View

- State mutability (pure and view) states do not allow changing the state of the blockchain.
- `non-payable` and `payable` allow changing the state of the blockchain.
    - If a function accesses a state variable, it cannot be marked as `pure`.

Contract Interactions

The four options for function types (**non-payable**, **payable**, **view**, and **pure**) dictate how calls to the smart contract are handled.

Interacting with Contracts via Interface

You can interact with any smart contract as long as you have the **API**.

Gas Cost Considerations

Calling a smart contract using an interface does not change the **gas cost**. The **gas cost** is determined by the amount of operations that the contract performs on the blockchain.

State Changes in Blockchain

When a transaction is sent to the blockchain, it takes time to be included. Once included, it changes the **state of the blockchain**. These state changes are determined by the **bytecode** stored on the blockchain. To change a value, you must go through a function, which ensures the security of the blockchain.

Smart contracts have multiple storage locations, similar to a hotel having multiple rooms.

Persistent Key-Value Storage

Inside each account, there's a collection of objects:

- Code of the smart contract.
- A persistent **key-value store** mapping of 256-bit words to 256-bit words, called **storage**.

    > This is a state variable. Under the hood, Solidity manages the creation of a place in the persistent key store mapping, including accessors and boundary management. Solidity manages the complexity of managing storage addresses.

Storage Addresses

Each variable has a storage address. Solidity automatically manages these storage addresses. The number of state variables in a contract is limited by $2^{256} - 1$. This is because the storage is a mapping of 256-bit words to the power of 256.

Strings as Arrays in Solidity

In Solidity, **strings** are treated as **arrays** under the hood due to how the **EVM (Ethereum Virtual Machine)** handles storage. When a string exceeds 31 bytes, it is split and stored across multiple storage locations, with pointers linking the segments.

Example of String Storage

Consider a test where a smart contract stores a string. If the string is longer than 31 bytes, the storage changes as follows:

1. The first 31 bytes are stored in the initial storage location.
2. A pointer to the next storage location is created.
3. The next 31 bytes are stored in the new location, followed by another pointer, and so on.

When retrieving the string, the EVM must fetch all segments and concatenate them. A simple line of code in Solidity can translate to hundreds or thousands of operations under the hood to manage this storage and retrieval process.

Pointers and Storage Addresses

When a string is sent to a smart contract, it includes the length, a pointer, and the content. The pointer indicates the location of the next segment of the string if it exceeds 31 bytes.

Here's how the data is structured when sending a string:

1. **Length**: Indicates the size of the string.
2. **Pointer**: The address of the next chunk of the string if it's larger than 31 bytes.
3. **Content**: The actual string data.

Smart Contract Account Storage

Smart contracts have both **code** and **database** (storage) linked to them within the same account. This is different from traditional systems where code and databases are hosted separately. In the EVM, the smart contract manages both its code and the data it stores.

Mutating State via Valid Transactions Only

Reading information from a smart contract is free and public, even for private variables. However, changing the storage on the blockchain requires a **valid transaction**, which can only be initiated through:

- A direct transfer of Ether from one account to another.
- A call to a smart contract function.

The EVM is the only entity that can modify the storage. Direct manipulation of the blockchain data, even with a local node, is not possible because the cryptographic hashes would not match, invalidating the changes.

Immutability of the Blockchain

While you can update the value of a variable on the blockchain, you cannot remove the previous value from the records. All past transactions are permanently recorded.

> Once data is stored on the blockchain, it remains there forever. Updating a value doesn't erase the old one; it simply records a new transaction with the updated value.

Therefore, it is essential to be cautious about what data is stored on the blockchain, as it will be permanently accessible.

Cryptography on Chain Considerations

When using cryptography on-chain, avoid posting **personal data**, as it could be revealed in the future. Blockchains are best suited for:

- **Balance values**
- **Voting rights**
- **Ownership of assets**
- **Game scores**
- **Lottery outcomes**

These values should be small yet impactful (e.g., 32 bytes). Larger data, like the URI identifier for an NFT picture, should be stored in **decentralized storage**.

Web2 vs. Blockchain Data Storage

For Web2 applications, manual databases are still needed. Alternatively, systems like IPFS can be used for storage separate from the blockchain itself.

Technical Deep Dive: EVM Storage

Storage Costs in EVM

Writing to storage within the EVM (Ethereum Virtual Machine) is costly. Actions like reading, writing, initializing, and modifying storage variables consume significant gas.

> To optimize gas costs, it's advisable to use more transient memory options when possible.

Memory operations involve 256-bit widths, with the bytecode handling the back-and-forth swaps. Understanding the correlation between memory and stack is crucial for gas optimization.

Memory vs. Call Data

Consider the implications when storing text:

```
pragma solidity ^0.8.0;

contract Example {
    string public text;

    function setText(string memory newText) public {
        text = newText;
    }
}
```

Strings are arrays and are reference types that can be handled in **storage**, **call data**, or **memory**.

- **Storage**: Data stored on the blockchain
- **Call Data**: Read-only location where function call arguments are stored
- **Memory**: Temporary storage for use while a function is executing

When setting a text from memory, the value is copied from outside the blockchain into memory and then to storage. An example of this can be seen in the code snippet above in the `setText` function.

Gas Cost Comparison

Deploying a contract and setting the text to "ABC" using memory incurs a gas cost of 32,203 gas. The information passed (ABC) is already included in the input, stored in call data. Using call data directly can save gas.

Deploying a second version of the contract using call data results in a gas cost of 31,075 gas for the same transaction. This is because call data accesses the data directly, avoiding the memory copy.

Importance of Understanding EVM

Deep understanding of the EVM is essential because a single wrong definition or inefficient practice can lead to significant financial costs, potential hacks, and loss of funds for end-users.

Call Data Limitations

Call data has limitations:

- Cannot change the information
- Cannot modify or append data directly

If modification is needed, memory must be used.

```
pragma solidity ^0.8.0;

contract Example {
    string public text;

    // Using memory to allow modification
    function setText(string memory newText) public {
        string memory modifiedText = string(abi.encodePacked(newText, " ad
        text = modifiedText;
    }
}
```

Global Variables

Visibility Options for Functions

Consider the visibility options for functions.

```
pragma solidity ^0.8.0;

contract Example {
    string public text;

    constructor() {
        initialText();
    }

    function initialText() private {
        text = "Hello";
    }
}
```

What are the possible visibility options for `initialText` for this code to work?

- **External**: Cannot be called inside the contract
- **Internal**: Works
- **Private**: Works

In summary, the visibilities are:

Visibility	Description	Usable?
External	Can only be called externally	No
Internal	Can only be called from within the contract	Yes
Private	Can only be called from within the contract	Yes
Public	Can be called both internally and externally	Yes
Pure	Does not read or write to the blockchain state	Yes
View	Reads but does not write to the blockchain state	Yes

Solidity Inheritance and Overriding

Inheritance Example

- Creating a new smart contract **InheritedHello** that inherits from **HelloWorld**.
- Deploying **InheritedHello** gives it all the functionalities of **HelloWorld** without writing new code.

Constructor Behavior in Inherited Contracts

- A constructor in InheritedHello is defined to set an initial text.

```
constructor(string memory initialText) {
    setText(initialText);
}
```

- Due to privacy settings, direct access to state variables like `text` and `initialText` might be restricted.
- The `setText` function, inherited from HelloWorld, can be used within the constructor to set the initial text.
- Deploying HelloWorld and InheritedHello will result in two different contracts with potentially different initial texts, stored at different addresses on the blockchain.

Overriding Functions in Inherited Contracts

- The `initialText` function in HelloWorld can be overridden in InheritedHello to change its behavior.
- To allow a function to be overridden, it must be marked as `virtual` in the base contract.
- In the inherited contract, the `override` keyword is used to redefine the function.

```
pragma solidity ^0.8.0;

contract HelloWorld {
    string public text;

    constructor() {
        text = "Hello World";
    }

    function initialText() public virtual view returns (string memory
        return "Hello World";
    }

    function setText(string memory newText) public {
        text = newText;
    }
}

contract InheritedHello is HelloWorld {
    constructor(string memory initialText) {
        setText(initialText);
    }

    function initialText() public override view returns (string memor
        return "Hello There, General Kenobi";
    }
}
```

- When InheritedHello is deployed, it inherits all code from HelloWorld but uses the overridden `initialText` function.
- Changes in the inherited contract do not affect the original contract's state.

Key Concepts

- **Virtual Functions**:

  Functions in the base contract that are allowed to be overridden in derived contracts.

- **Override Keyword**:

  Used in the derived contract to specify that a function is replacing a virtual function from the base contract.

Solidity Design Philosophy

- Solidity is designed for **composability**, allowing developers to restrict or enable access to code functionalities.
- This design primarily focuses on **developer access** and doesn't directly manage information on the blockchain, such as crypto balances.

Contract Deployment and Metamask

- Contract deployment might require signing with Metamask, depending on the environment configuration.
- Using a virtual machine environment with pre-funded accounts removes the need for manual signing, streamlining deployment during development.
- Each deployment incurs a cost, paid by the account used for the deployment.

Demonstrating Value Changes

- Values can be changed by calling functions on the deployed contracts.
- Inheriting a contract doesn't affect the storage of the original contract; they exist at different addresses.

Constructor and Function Execution Order

- When a contract is inherited, the constructor of the base contract is called implicitly.
- However, if a function called within the constructor is overridden in the inherited contract, the overridden version is executed.

  Analogy: "Every time that you arrive home you put your quote in the hanger okay the quote hanger every time you arrive home you put your code there but someone changed the Cod hanger from the left to the right so you arrive home and you code falls in the ground because you are so used to putting the the quod there when you arrive at home"
- This can lead to unexpected behavior if the developer is not aware that the overridden function is being called during construction.

Function Overriding

For overriding a function, the base function needs to be **virtual**, and the overriding function needs the **override** keyword.

Reviewing Smart Contract Functions

The lecture goes over smart contract functions, covering aspects from previous lessons.

Initial Text Function

The initial text function example demonstrates how changing the text in one function doesn't affect another, setting the stage for more complex interactions.

`textHasChanged` Function

This function checks if the text has been changed from its original value.

- It returns a **boolean** value, unlike the string return from the initial function.
- Because it returns a value type, there's no need to specify where the return value has to be charged to memory.
- The function uses a **named return value**, which allows for implicit returns in Solidity.
    - When a return value is named, the `return` statement doesn't need to explicitly specify the value.
    - Solidity manages the return variable behind the scenes.
    - It can be especially useful when returning multiple values.

```
function textHasChanged() public returns (bool _){
    _; // implicit return
}
```

Naming Conventions

- **Underscore after:** Indicates things that are being returned (e.g., `return bool _;`).
- **Underscore before:** Denotes private or internal members.

String Comparison

- Solidity doesn't have built-in string manipulation functions.
- Strings can be compared by using the **hashes** of the strings.
    - Comparing hashes is an efficient way to check for equality.
    - Comparing hashes is preferred because Solidity lacks built-in length or index access for strings.
- If the hash of string A equals the hash of string B, they are almost certainly the same string.
    - The probability of hash collision is very low, especially in these situations.

`restore` Function

The `restore` function resets the text to its initial value.

- If the text has been changed, calling `restore` sets it back to the original text and updates the `textHasChanged` status.
- This demonstrates different return types and operations within a smart contract.

Encapsulation Pattern

Encapsulation: Organizing code so that public functions call internal or private functions.

The presenter explains that this pattern is not necessarily for gas optimization but is a common design pattern.

Benefits

- It is common in implementations like ERC20.
- Supports building composable and inheritable smart contracts.
- Helps control what derived contracts can or cannot change, preventing breakage.

Access Control

- `isChanged` function marked as **private**.
- `textHasChanged` function marked as **public virtual**.

This configuration allows `textHasChanged` to be overridden in derived contracts while keeping `isChanged` inaccessible externally.

Gas Considerations

Even if a function doesn't change the state due to conditional logic, gas is still consumed if the function is marked as `non-payable`.

Scenario

- The `textHasChanged` is false.
- The smart contract is called as `non-payable`.

Explanation

- Smart contracts are designed to recognize payable, non-payable, or view at call time.
- It's impossible to convert `non-payable` to `view` during execution.
- Gas is paid for someone to mine the transaction and return false.

Error Handling and State Reversion

Revert Function

The **revert** function is used to stop the execution of a transaction and revert the blockchain to its initial state.

- Even if the **PID** is used, there's a way to reverse execution using error handling.
- Conditions can be asserted or required, and the contract can be asked to revert.
- Gas is paid up to the point of reversion.

Preventing Overuse of Revert

- Revert can be used even if computation has changed.
- **Revert** undoes all changes back to before the transaction.
- Attack Scenario:
  - An attacker asks to do a lot of computation to waste resources and then reverts.
  - You pay gas up to the point of usage.
  - No changes are made to the blockchain.
- Transactions can be simulated in a local node to estimate gas costs.

Metamask Gas Estimation

When a revert is included in the constructor:

- Metamask estimates gas.
- If a transaction is likely to fail, Metamask warns the user.
- End-users can refrain from sending the transaction to save gas.

Revert vs. Require vs. Assert

There are three different ways to handle errors: `revert`, `require`, and `assert`.

Semantic Differences

- `revert` and `require` do the same thing but are positioned differently.
- `assert` is used for checking invariants.

Use Cases

- **Assert**: For things that are invariant constants, which are dramatic if not present.
  - Example: Asserting that a person has a head.
- **Require**: For things that are normal to happen.
  - Example: Requiring that a client has money to pay or a membership card.
- **Revert**: For cases where you want to stop the execution of a transaction and revert the blockchain to its initial state.

Example Scenario: Restore Function

Consider a scenario where a person tries to restore text that hasn't changed.

- Require the text to have changed; otherwise, revert to prevent wasting money.
- The function `restore` should only work when the text has changed so it can be restored to the initial text.
- The function should not work when the text is already the initial text.

Modifiers

Definition

> A **modifier** is a language technique that doesn't get deployed to the smart contract. It appends or prepends code inside another function.

How They Work

- A modifier puts a code block before a function.
- The amended semantic can be before, after, or in between, surrounding the function.
- A symbol jumps back to the function, applying the modifier to all functions where it's used.

Example: `onlyWhenTextChanged` Modifier

```
modifier onlyWhenTextChanged() {
    require(textHasChanged, "Text has not changed");
    _; // This underscore means "run the rest of the function body."
}
```

Usage

Instead of putting the `require` directly in the function, a modifier can be created:

```
modifier onlyWhenTextChanged {
    require(textHasChanged);
    _;
}

function restore() public onlyWhenTextChanged {
    // Function body
}
```

Key Points

- Modifiers do not get deployed to the blockchain.
- For the EVM, functions with modifiers are the same as functions without them.
- Modifiers improve readability.

Global Variables in Solidity

Solidity provides access to global variables within smart contracts. These variables provide information about the blockchain, transaction, and block, and can be accessed without explicit declaration.

Transaction and Message Properties

These variables provide information about the current transaction.

- `msg.sender`: Address of the account or contract that sent the transaction.
- `msg.value`: Amount of ether sent with the transaction, in wei.
- `gas`: Amount of gas available for the transaction.

Block Information

These variables provide information about the current block.

- `block.hash`: Hash of the current block.
- `block.number`: Current block number.
- `block.timestamp`: Timestamp of the current block.
- `block.difficulty`: Current block difficulty.

Assert vs. Require

`require` and `assert` are used to handle errors and conditions in Solidity, but they serve different purposes:

> `require` is used to validate conditions such as inputs, or state variables prior to execution.
>
> `assert` is used to validate conditions that should never be false. If an assert fails, it indicates a critical error in the contract's logic.

Differences between `require` and `assert`

Feature	`require`	`assert`
Purpose	Input validation, checking conditions before execution.	Validating conditions that should never be false, internal errors.
Error Type	Normal error that is expected to sometimes happen.	Critical error, indicates a bug in the contract.
Behavior	Reverts the transaction, refunds remaining gas.	Reverts the transaction, consumes all remaining gas (panics).
Use Case	Validating user input, checking preconditions.	Ensuring internal state consistency.
Documentation	Solidity Documentation	Solidity Documentation

Example

```
uint balance = 100;

// Using assert to check a condition that must always be true
assert(balance >= 0); // If balance goes below 0, it indicates a critical

if (balance < 0) {
    // Panic: this should never happen
}
```