Q&A Recap

- Transactions in the blockchain can transfer ETH, or send messages to smart contracts.
- These messages invoke functions within the smart contracts.
- Yesterday's experiment involved interacting with a **Uniswap** smart contract for currency exchange.
- Smart contracts have no backend or centralized services as they operate on-chain.

Coding Smart Contracts

Let's learn how to build smart contracts.

- Yesterday, we briefly experimented with Remix as an IDE.
- Today we will go line by line and understand the code.

Examining Blockchain Transactions

- Using **Etherscan** to inspect transactions.
- The input data in a transaction is not the program itself, but parameters for calling functions within the smart contract.
- The actual program (smart contract) is hosted at a specific address.
- Etherscan can be used to view the **bytecode** of the contract.

Bytecode and Opcodes

Bytecode

- The bytecode is the actual code of the smart contract, represented in hexadecimal format.
- Each two hexadecimal characters represents one byte.

Opcodes

- Opcodes are composed of bytes.
- The bytecode is a collection of opcodes in hexadecimal.

High Level Languages

- Instead of coding directly with opcodes (like assembly), high-level languages like Solidity are used.
- Solidity allows developers to write code in human-readable terms.
- A compiler then translates the Solidity code into bytecode.
- Remix will be used as a compiler.

Solidity Compiler

Solidity Code

- Solidity code can be written in a file explorer within Remix.
- The Solidity compiler is an off-chain program that converts Solidity code into two outputs:
    - An Application Binary Interface (ABI), which defines the functions and how to call them.
    - The bytecode itself.

Compilation

- Compilation can be done manually or automatically (auto-compile) in Remix.

Outputs

- The compiler outputs two files:
    - ABI: A JSON file that guides on how to call functions.
    - Bytecode: The compiled code that is deployed to the Ethereum Virtual Machine (EVM).

Understanding Solidity

Solidity's Role and the EVM

- The Ethereum Virtual Machine (EVM) cannot directly understand Solidity.
- Solidity serves as the input for a compiler.
- The compiler's output is bytecode, which the EVM can understand and use.

API and Smart Contract Interaction

- When a smart contract is published to the blockchain, the API (Application Binary Interface) isn't stored on the blockchain itself but is made available separately.
- The API informs others how to interact with the bytecode.
- The API can be published on platforms like Etherscan.

API Explained

- The API in the context of Solidity is the **Application Binary Interface**.
- The Solidity API is different from a traditional API.

  The API is structured more like a **JSON file**, detailing the expected inputs and outputs for the smart contract.

Compilation Process

- Solidity file $\rightarrow$ Compiler $\rightarrow$ API + Bytecode

Solidity's Design

Solidity is a language designed to target the Ethereum Virtual Machine.

- To ensure the compiler understands Solidity code, you need to adhere to the **language grammar**.
- Solidity's rules may not be as straightforward as those in other programming languages.
- Resistance to Solidity is common due to its differences from other languages.
- Solidity's design decisions are related to constraints of the **blockchain architecture**.

Structure of a Smart Contract

Every Solidity file needs to have a similar structure, including:

- Object declaration
- Formations to target
- Compiler
- State variables
- Functions
- Function modifiers
- Events
- Errors
- Strict types

Solidity File Requirements

For a Solidity source file to be valid for the compiler, it needs a few key definitions:

1. **License Identifier**:
    - It's not strictly required, but it's important for protecting your code from unintended usage.
    - A software license can provide:
        - Limitation of warranty
        - Limitations of liabilities
        - Trademark protection
    - Examples of open-source licenses:
        - MIT
        - GPL
        - Apache
2. **Version Pragma**:
    - Mandatory for compiling a smart contract.
    - It defines the range of compiler versions that can be used to compile the code.
    - Example: `pragma solidity ^0.8.0;`
    - You can specify a range (e.g., from version 0.7 including to version 0.9 excluding) or a specific version.
    - Using a specific version provides maximum control but less compatibility.
    - Breaking changes can occur even in minor version updates, such as changing an opcode.

Important Note

> Neither the license identifier nor the version pragma get deployed to the blockchain. They are used to communicate with the compiler program.

Opcode Changes and Immutability

- Changes to opcodes (lower-level operation identifiers) can occur between compiler versions.
- Publishing a smart contract means it's on the blockchain forever, and its execution flow will never change.
- The same smart contract compiled with different compiler versions can result in different bytecode.
- Different versions of compilers can produce different bytecodes.

Translation Analogy

> Picking a phrase in English and translating it using different dictionaries can result in different translations, even though they're based on the same original phrase.## Solidity Versions and Trade-offs

- Older versions of Solidity are more **battle-tested**, having been extensively tested for bugs.
- Newer versions may have bugs or compatibility issues, such as problems integrating or inheriting from other smart contracts.

Smart Contract Structure Grammar

The structure of a smart contract can be defined using a specific grammar.

```
Contract Definition -> [abstract] contract Identifier { Contract Body Elements
```

- **Abstract**: The `abstract` keyword is optional and indicates that the contract can be overridden (similar to `virtual` in other languages).
- **Contract**: The `contract` keyword is mandatory and must be in lowercase.
- **Identifier**: This is the name of the smart contract. It can be any word that is not a reserved keyword. Style guides recommend using capitalized words for contracts, but it is not a compiler requirement.

After defining the contract, you can optionally define:

- Layouts of memory of variables.
- Inheritance from other smart contracts.
- Bytecode.

A smart contract must:

- Open with a curly bracket `{`.
- Contain contract body elements such as constructor definitions.

Constructor Definition

```
Constructor Definition -> constructor ( Parameter List ) [Modifier] { Construc
```

- Parameter List: A list of parameters for the constructor (can be empty).
- Modifier: Options like `payable`, `internal`, or `public`.

Function Definition

```
Function Definition -> function Identifier ( Parameter List ) Visibility State
```

- Function: Must start with the `function` keyword in lowercase.
- Identifier: The name of the function.
- Parameter List: Includes the data location for each parameter.
- Visibility: Defines who can call the function (`external`, `public`, `internal`, or `private`).
- State Mutability: Defines how the function interacts with the contract's state (`pure`, `view`, `payable`, or none).
- Modifier: Additional modifiers for the function.
- Returns: Specifies the return type(s) of the function, along with the data location for each return parameter.

The function definition in Solidity is extensive because it needs to prepare the Ethereum Virtual Machine (EVM) to manage state access, invoke capabilities, and control the flow of the function call. This is a low-level process handled in the function call.

Function Visibility

When creating a contract function, you have four visibility options:

- External:

  Allows anyone from outside the smart contract to call the function.

- Public:

  Allows both external and internal function calls.

- Internal:

  Only allows calls from within the contract.

- Private:

  Like internal, but not visible to derived contracts (contracts that inherit from it).

State variables do not have external visibility. They can only be `public`, `internal`, and `private`.

State Mutability in Solidity

Solidity offers several options to define the mutability of functions, affecting their behavior and gas costs. There are four options for state mutability in Solidity, but the first two we're going to explore are `view` and `pure`.

View Functions

> View functions promise not to modify the state of the blockchain but can read it.

- They are similar to constants and can only work with:
    - Message data
    - Code that is static
- View functions can:
    - Get balances of accounts
    - Access block number and transaction variables
- View functions cannot:
    - Write to state variables
    - Emit events
    - Create new contracts
    - Use `selfdestruct`
    - Send Ether
    - Call functions not marked as `view` or `pure`
- View functions are free to call directly (no gas cost).

Pure Functions

Pure functions promise not to read or modify the blockchain state.

- They only depend on input parameters.
- They always return the same result for the same input (idempotent) when called directly.
- They cannot access any blockchain data.
- Pure functions might charge the user for computation if called from another context.

Example Scenario: NFT Transfer

To transfer an NFT from one address to another, you cannot use `view` or `pure` functions. Writing to the blockchain requires either `payable` or `non-payable` functions.

Remix IDE and Virtual Environment

- Remix is an online IDE used for Solidity development.
- It offers a virtual machine blockchain environment for testing.
- This virtual environment provides accounts with a lot of fake ETH for experimentation.
- You can deploy smart contracts to the virtual blockchain and interact with them.

Interacting with Deployed Contracts

1. In Remix, navigate to the "Deploy & Run Transactions" tab.
2. Deploy your smart contract using the orange "Deploy" button.
3. Expand the deployed contract in the bottom panel.
4. Click on the function name to invoke it.

State Variables

A state variable is a place in the smart contract where information can be stored.

```
string public myString;
constructor(string memory _myString) {
    myString = _myString;
}
```

The above code creates a state variable called `myString`, which allows you to store a string that persists within the smart contract at the same address where it's deployed.## Storing Variables in Smart Contracts

When you have code and the value of a variable, a space is set aside in the account of the smart contract to store collections of bytes. These collections of bytes can be set to "hello world" in the constructor function.

Constructor Function

The constructor function is executed during deployment, not when the smart contract is called later.

State Variables and Blockchain

If you use the variable `text` inside all the functions to dynamically update a return value, that **state variable** becomes part of the blockchain. If it's part of the blockchain, it **cannot** be declared as `pure`.

If you try to declare a state variable as `pure` when it's reading from the state, it requires the `view` keyword.

View Context

Anything that requires state needs to be in a **view context**, or a `non-payable` or `payable` context.

Setters and State Mutability

Setting Variable Values

To change a variable, you need a **setter** function that can set the value of the variable. An example is the `setText` function that receives a `newText` parameter.

Function: `setText`

- Receives one parameter: `newText`
- `newText` is a `string memory`
- This function is `public` and assigns the variable `text` to `newText`
- State mutability: `non-payable` (default mutability)

Documentation

It's useful to bookmark documentation pages for reference. Documentation changes often, so it's good to form the habit of going to the documentation and reading things directly.

Non-Payable vs. View

> Both `non-payable` and `payable` can modify the state, unlike `view` and `pure` functions.

Non-Payable Functions

Let's consider a `non-payable` function.

Orange Function

Interesting fact: The function is named `non-payable`, but you still pay for the transaction. You don't pay for the function itself, but you pay for the transaction.

Even though the function was declared before `hello world`, it is declared on line 11, and `setText` is declared on line 15. Because you're referring to the pointer in the memory, or in this case storage, it was set to something here, stored in that variable.

When you read this variable, it returns the new thing that you have put there. The `non-payable` function can change the state of a variable in a smart contract.

Gas and State Changes

For `pure` and `view` functions, you don't have to pay gas. But in this case, since we are changing the state of the blockchain, you have to pay. You pay for the transaction.

On-Chain vs. Off-Chain Execution

`pure` and `view` functions run on-chain, but you can evaluate them directly with a node. You don't need to run the whole consensus. You just go for a node and you evaluate the execution of that function against the state.

You can run a node yourself for free or get access to a free node.

Interacting with Smart Contracts

When you connect to the environment, you can deploy the smart contract and start interacting with the functions:

1. Pick the state of "hello world."
2. Set the text to something new.
3. Every time you set the text, you need to pay.

If you select another account and call `hello world` multiple times, you don't pay anything. But if you call `setText`, even though it's `non-payable`, it changes the text and you pay for the transaction. You don't pay for the function or the smart contract itself.

Gas Amount

You can set an amount of gas for certain function calls.

Private Strings and Public Functions

- When a **string is private** and a **function is public**, the function can be called from another account, even though the string itself cannot be directly accessed from outside the smart contract.
- Analogy:
    - Imagine your house, where someone can't just walk in and grab a beer from your fridge. However, they can knock on the door and ask you to bring them a beer.
    - In this scenario, the front door represents the public interface, while the contents of your fridge represent private data.

Interacting with State Change Calls in Public Testnet

- The second part of the lecture involves interacting with state change calls in a public testnet.
- To proceed, you need to have completed the previous day's lesson and have some testnet gas in your wallet.
- Select the **injected provider** option to connect to your MetaMask.
- Unlike a virtual machine blockchain, interacting with the public testnet requires real testnet tokens.
- The process involves deploying a smart contract and observing the interactions.
- The lecture highlights the importance of the API and introduces data locations.
- Similar to the previous day, interacting with the blockchain requires approval via MetaMask and involves paying a small fee (Cepulia in this case).
- Once the transaction is complete, the smart contract is deployed to the blockchain.
- The deployed smart contract can be viewed in the blockchain explorer.
- The bytecode is generated by the compiler, not written directly by the user.
- It's not possible to directly access the Solidity code from the deployed contract.
- The compiler can generate bytecode from various languages like Viper, Python, TypeScript, or Rust.
- Analogy:
    - You can go from orange to orange juice, but reversing the process is difficult.
    - The compiler creates the bytecode, but there isn't a direct link back to the original code.
- Attaching the Solidity code to the smart contract is not standard.

View Functions and State Changes

- When interacting with a deployed smart contract, **view functions** do not require payment.
- View functions simply retrieve existing information from the blockchain.
- When a smart contract is deployed, the transaction includes storing the bytecode and setting initial variables.
- State changes involve the deployer paying gas and the producer receiving rewards for mining or publishing the block.
- Initial variables can be set during the deployment transaction.
- Reading existing information from the blockchain only requires evaluating the current state and does not incur any fees.
- Changing the state of the blockchain requires payment.
- **Non-payable** functions do not mean you don't pay, but rather that you don't pay the contract directly. Instead, you pay for the transaction on the blockchain.
- The transaction involves a call to the `setText` method with a string parameter.
- State changes track the payment details (producer, receiver, gas) and the changes within the contract (e.g., changing "hello world" to "potato").

- Interactions with smart contracts can occur even without the original Solidity code.
- The opcodes are stored on the blockchain, enabling interaction through interfaces.
- An interface is defined as:

  > A contract that lacks its own definition and serves to shape the Application Binary Interface (ABI) of contracts.

- Key characteristics of interfaces:
  - Cannot have code or inherit from contracts.
  - All declared functions must be external.
  - Cannot have state variables or modifiers.
  - Represent the ABI of contracts.
- Interfaces can be attached to contracts or used directly to interact with the blockchain.
- Even if the original smart contract code is deleted, interactions are still possible through the interface, as long as the names and parameters match the ABI.
- This allows invoking the execution and changing the state of the smart contract, even without knowing how the functions work internally.

Smart Contract Interaction: Beyond the Code

Once a smart contract is stored on the blockchain, it can be interacted with using its Application Binary Interface (ABI), even without the original code.

Application Binary Interface (ABI)

> The ABI specifies how to interact with the smart contract, including what data to send, what to expect in return, and any constraints of the functions (e.g., `view-only`, `pure`, `payable`).

Example: Changing the state of a smart contract from "potato" to "tomato" using the ABI without needing to know the underlying code.

Accessing Smart Contract Functions

Challenges with Just the Address

It's difficult to determine the available functions from just the smart contract's address.

- Decompilation: Bytecode can be decompiled to infer functions, but it's not ideal.
- API Importance: The best approach is to use the ABI, which provides function names, parameters, and other necessary details.

Publishing and Sharing Interfaces

When deploying a smart contract, it's important to make the interface accessible to others.

- Verifying on Etherscan:
  - Submit the bytecode to Etherscan to verify the contract.
  - This allows others to easily understand the functions available in the contract.
- Open Source Practices:
  - Publish the contract code on platforms like GitHub.
  - Share the GitHub links on social media to promote transparency and accessibility.
  - Publish on IPFS and Swarm

Bytecode and Source Code

There isn't a direct link from bytecode back to the original source code.

- Decompilation Limitations: While decompilation is possible and can be deterministic, it doesn't guarantee the reconstructed code is the exact source used to create the bytecode.
- Analogy: Like going from orange juice to an orange; you can infer the source, but it's not a perfect reconstruction.

Interfaces for Conformity

Interfaces ensure contracts adhere to specific standards or include necessary functions.

- **Standard Compliance:** When creating a token, using an interface like **ERC-20** ensures the contract includes required functions like `transfer` and `balanceOf`.
- **Analogy:** Similar to using power adapters with different pin configurations when traveling.

Payable Functions Explained

Payable Functionality

Marking a function as `payable` enables the smart contract to receive **Ether (ETH)**.

- **Enabling Payments:** `payable` doesn't require a payment, but it allows the function to accept one.
- **Future Logic:** You can implement logic to require a minimum payment (e.g., `msg.value > 1 ETH`), but this is a concept for later lessons.

Demonstration in a Virtual Machine

Using a virtual machine to demonstrate `payable` functions without spending real **Testnet tokens**.

- **Deploying Payable Contracts:** When a contract with a `payable` function is deployed, the function is highlighted (e.g., in red) in the interface.
- **Setting Text and Paying:** You can set the text and send ETH to the contract simultaneously.

```
setText("AABVCCC") and send 10 ETH
```

- **Money Flow:**
    - One payment covers the transaction fee.
    - The other payment is stored within the smart contract.
- **Potential Use Case:** Implementing a system where setting text requires buying it from someone else.

Non-Payable Functions

Attempting to send ETH to a non-payable function will result in an error.

- **Error Message:** The transaction will fail, indicating that the function should be `payable` if you're sending value.
- **Inability to Force Value:** You can't force a contract to receive value if the function isn't set up to accept it.

Key Differences

Feature	Payable Function	Non-Payable Function
Payment	Accepts ETH payments	Does not accept ETH payments
Transaction Fee	Requires payment for the transaction	Requires payment for the transaction
Sending ETH	Succeeds if the function logic allows it	Fails with an error if attempting to send ETH
Contract Balance	Can store received ETH if designed to do so	Cannot store ETH sent during the transaction

Payable Functions and Function Annotations

Understanding Payable Functions

When a smart contract function is set as **non-payable**, it cannot receive Ether (ETH) payments. Attempting to send ETH to a non-payable function will cause the transaction to fail.

> A payable function can receive Ether (ETH) payments, while a non-payable function cannot.

In MetaMask, a failed transaction might be indicated in red, signifying that sending payment to a non-payable function is not allowed. Conversely, a successful (but not necessarily payable) transaction might be shown in orange.

It is normal to have zero Sepolia ETH in your MetaMask. If you need more, you can obtain it from a faucet.

Style Guides and Naming Conventions

Understanding **style guides** is crucial for maintaining readable and consistent code. For example, contracts and libraries should be written with capitalized words. Adhering to these guidelines is optional but helps in maintaining and improving code readability.

By familiarizing yourself with style guides and naming conventions, you can address questions about code layout, order, and naming conventions.

NAT Spec for Documentation

The **NAT Spec** is a documentation standard that allows you to embed documentation within your smart contracts. This documentation can be reused later for explaining various aspects of the contract, even to end-users.

You can include documentation in your smart contracts that will be ignored by the compiler, meaning they don't increase the contract's size or cost. This is achieved through special notations like triple forward slashes (`///`) or code blocks.

Documenting your smart contracts is highly recommended. Annotate what a function should return, what values it should retrieve, and what values it should store.

Benefits of Annotations

- Helps you remember the function of aspects of the smart contract in the future
- Allows others to read and understand the code
- Allows you to compare and understand the code between other developers

How to Annotate

Use triple forward slashes (`///`) or code blocks to insert annotations. Ensure you understand the specific annotations applicable to contracts, functions, and interfaces.

Annotations include:

- Explanations of what the function does
- Parameters that should be passed
- Return values