# Introduction to Web3 and Block Chain

Prepared By: Abdalrhman Mostafa

## Objective

The objective of this introduction is to introduce the fundamentals of smart contracts and blockchain technology.

## Prerequisites

- crypto wallet like MetaMask.
- Remix IDE.

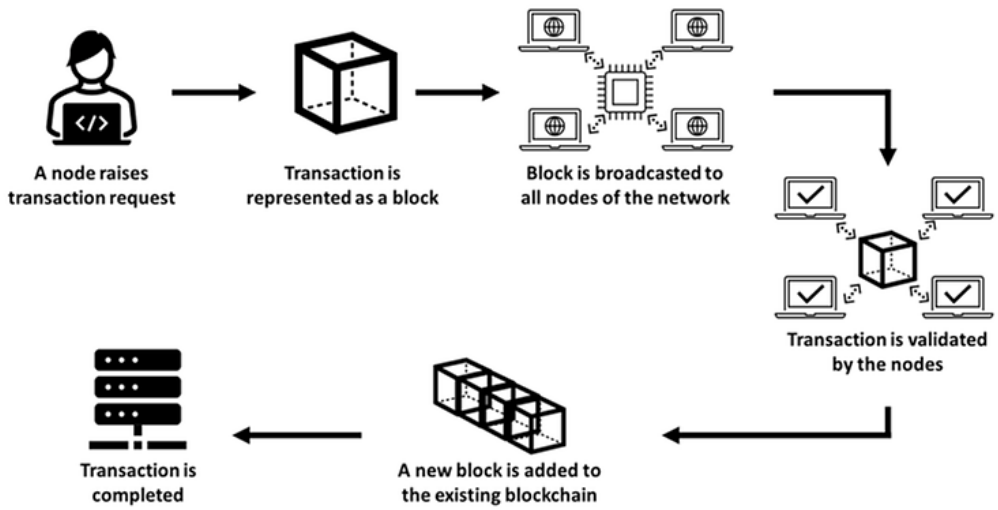
## Introduction

Blockchain: Blockchain is a type of distributed database or ledger shared among a computer network's nodes, that uses cryptography to secure its data. It is made up of a chain of blocks, each of which contains a set of transactions. The blocks are linked together using cryptography, which makes it very difficult to tamper with the data.

Nodes: Nodes are computers that participate in the blockchain network. They store blockchain data and verify new transactions.

Blocks: Blocks are the basic unit of data in a blockchain. They contain a set of transactions and a cryptographic hash of the previous block.

Transactions: Transactions are the basic unit of interaction in a blockchain. They can be used to transfer value, create contracts, or record other events.



## How blockchain technology works

Blockchain technology works by using a distributed network of computers to verify and record transactions. Each transaction is added to a block, which is then linked to the previous block using cryptography. This creates a chain of blocks, each of which is tamper-proof.

To add a new block to the blockchain, a miner must solve a complex mathematical problem. The first miner to solve the problem gets to add the block to the blockchain and is rewarded with cryptocurrency.

- Blockchain Demo and useful links.

# Some Important Principles:

## DApps

A short for Decentralized Applications, are applications that run on a decentralized network, typically utilizing blockchain technology. Unlike traditional applications that are hosted on centralized servers, DApps operate on a peer-to-peer network, where data and computations are distributed across multiple nodes.

## Tokenization

the process of representing real-world or digital assets as tokens on a blockchain network. It involves converting the ownership or rights of an asset into a digital representation, which can be traded, transferred, and recorded securely on the blockchain.

Tokenization enables the fractional ownership and transfer of assets that were traditionally indivisible or illiquid, such as real estate, artwork, securities, or even loyalty points. By representing these assets as tokens on a blockchain, it becomes easier to manage, trade, and track ownership of these assets in a transparent and decentralized manner.

## DeFi

A short for Decentralized Finance, refers to a category of financial applications and protocols built on blockchain technology, primarily leveraging smart contracts. DeFi aims to recreate traditional financial systems and services in a decentralized, open, and permissionless manner, without the need for intermediaries like banks or financial institutions.

DeFi platforms enable users to access a wide range of financial services, including lending, borrowing, trading, investing, and asset management, directly through decentralized applications (DApps) or protocols. These services are typically powered by cryptocurrencies or digital assets and operate on blockchain platforms like Ethereum.

## DAOs

A short for Decentralized Autonomous Organizations, are organizations or entities that operate on a blockchain network using smart contracts and decentralized governance mechanisms. DAOs aim to automate decision-making, governance, and operations through code, removing the need for traditional hierarchical structures and centralized control.

## Smart Contract:

A smart contract is a self-executing contract with the terms of the agreement directly written into code. It is deployed on a blockchain network and automatically executes predefined actions when specific conditions are met. Smart contracts eliminate the need for intermediaries, as they enforce the rules and facilitate transactions directly between parties.

## characteristics of smart contracts:

*Automation*: Smart contracts automate the execution of agreements, eliminating the need for manual intervention. Once deployed, they execute actions based on predetermined conditions, such as a specific date, a trigger event, or the fulfillment of certain criteria.

*Transparency*: Smart contracts operate on a blockchain, which provides transparency and visibility to all participants. The code and the contract's state are stored on the blockchain, allowing anyone to verify the contract's functions and outcomes.

*Security*: Smart contracts use blockchain's cryptographic features to ensure security. The code is tamper-resistant and immutable once deployed, reducing the risk of fraud or unauthorized changes. Transactions executed within smart contracts are also recorded on the blockchain, enhancing security and accountability.

*Efficiency*: Smart contracts automate processes, reducing the need for manual paperwork, intermediaries, and time-consuming administrative tasks. This efficiency can lead to cost savings and faster settlement times.

*Multi-party Agreements*: Smart contracts facilitate agreements between multiple parties, ensuring that all participants adhere to the agreed-upon terms. The code enforces the rules and ensures that all parties fulfill their obligations.

Smart contracts find application in various industries, including finance, supply chain management, insurance, voting systems, and more. They enable decentralized applications (DApps) and decentralized finance (DeFi) platforms to operate securely and transparently.

It's important to note that while smart contracts provide numerous benefits, they are only as good as the code written. Vulnerabilities or bugs in the code can lead to unexpected outcomes or security risks. Therefore, thorough testing and code review are essential to ensure the reliability and security of smart contracts.

## Smart Contract Applications:

Smart contracts are already being used in a variety of applications, including:

- **Decentralized finance (DeFi):** DeFi is a financial system that is built on blockchain technology. Smart contracts are used in DeFi to automate the lending, borrowing, and trading of cryptocurrencies.

- **Supply chain management:** Smart contracts can be used to track the movement of goods and materials through a supply chain. This can help to ensure the authenticity and traceability of products.
- **Insurance:** Smart contracts can be used to automate the issuance and settlement of insurance claims. This can help to reduce costs and improve efficiency.
- **Voting:** Smart contracts can be used to create a secure and transparent voting system. This can help to increase voter turnout and reduce fraud.
- **Real estate:** Smart contracts can be used to automate the buying, selling, and renting of real estate. This can help to make the process more efficient and secure.

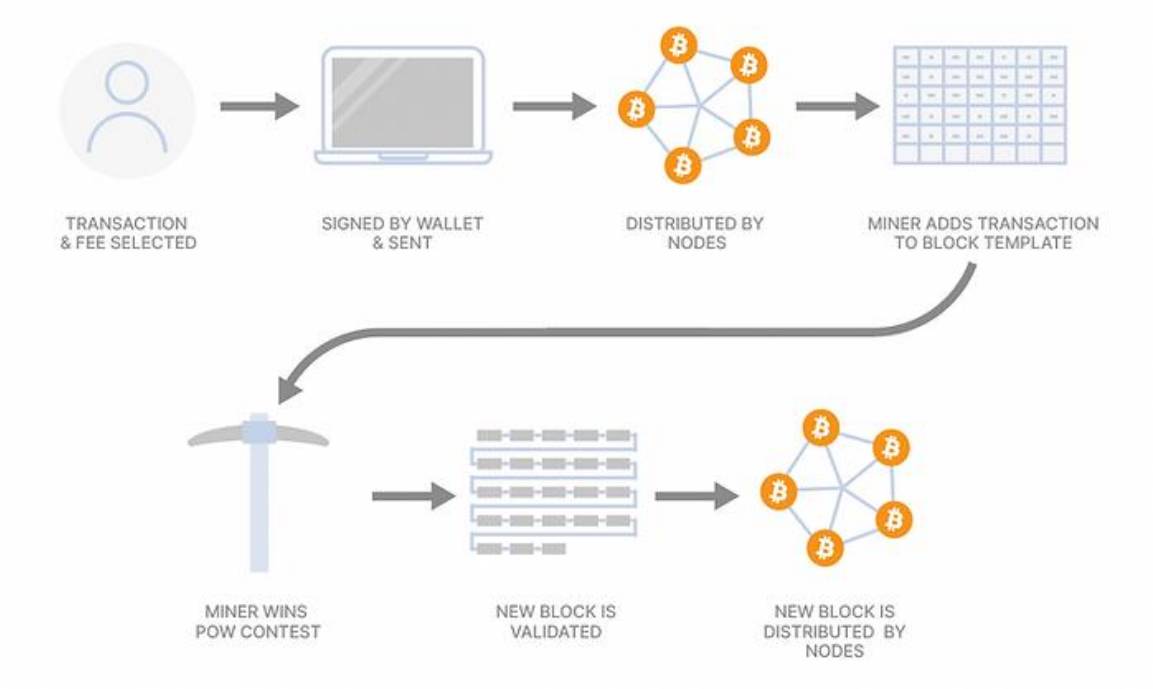
## Consensus Mechanism

The process of validating a new block: difference between proof of work and proof of stake
There are two main consensus mechanisms used to validate new blocks in a blockchain: proof of work and proof of stake.

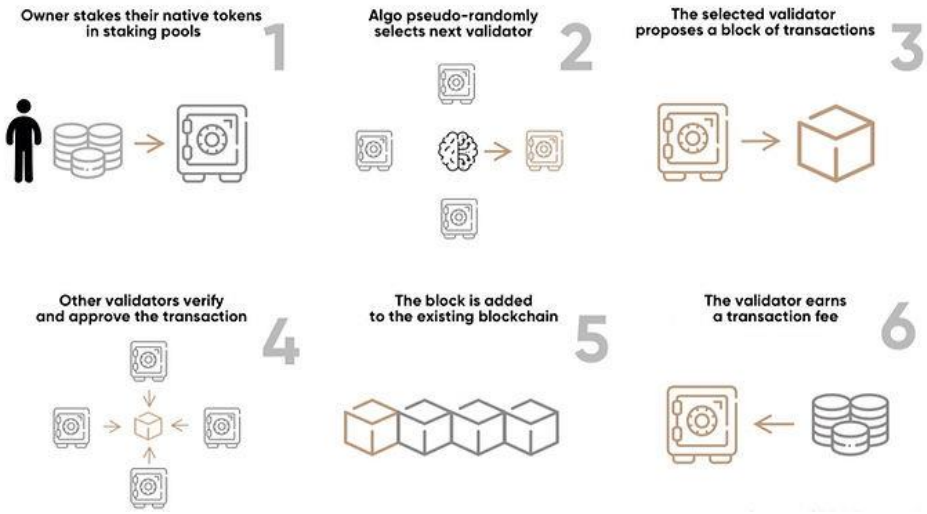
*Proof of work*: Proof of work is the most common consensus mechanism used in blockchains. It is a computationally intensive process that requires miners to solve complex mathematical problems to add new blocks to the blockchain.

*Proof of stake*: Proof of stake is a newer consensus mechanism that is less energy-intensive than proof of work. It is based on the idea that nodes with more stake in the network (i.e., more cryptocurrency) are more likely to be honest and participate in the validation process.

Introducing bitcoin, Bitcoin is the first and most well-known cryptocurrency. It was created in 2009 by an anonymous person or group of people under the pseudonym Satoshi Nakamoto. Bitcoin is based on the proof of work consensus mechanism.




Ethereum is a blockchain platform that allows for the creation of decentralized applications (DApps). It was created in 2015 by Vitalik Buterin. Ethereum is based on the proof of work consensus mechanism, but it is transitioning to proof of stake.




## Web 3:

### Definition of the web

The web is a global information system that connects computer networks and allows users to access and share information. It is made up of billions of web pages and other resources, which are linked together by hyperlinks.

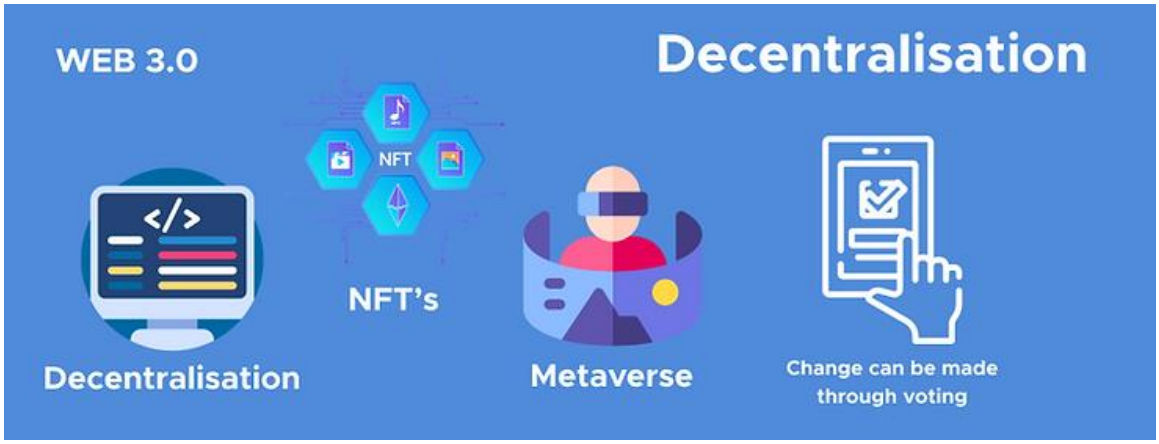
## The history of the web

The web has gone through three major generations:

*Web 1.0 (1991-2004)*: The first generation of the web was characterized by static websites that were mostly read-only. Users could view and download information, but they could not interact with it in a meaningful way.


*Web 2.0 (2004-present)*: The second generation of the web introduced more interactivity and user-generated content. Users could now create and share their own content, and they could also interact


*Web 3.0 (present-day)*: The third generation of the web is still under development, but it is envisioned as a more decentralized and intelligent web that is powered by blockchain technology. Web 3.0 will make it possible for users to own their own data and control how it is used. It will also make it possible for machines to understand and process information more effectively, leading to more personalized and relevant experiences for users.





Web 3.0 has the potential to revolutionize the way we interact with the internet. Some of the benefits and opportunities of Web 3.0 include:

Decentralization: Web 3.0 will make it possible to decentralize the internet, which means that it will be less controlled by a small number of big tech companies. This could lead to a more democratic and equitable internet.

Ownership of data: Web 3.0 will make it possible for users to own their own data. This means that users will have more control over how their data is used and who has access to it.

Personalization: Web 3.0 will make it possible for machines to understand and process information more effectively. This could lead to more personalized and relevant experiences for users.

Security: Web 3.0 will be more secure than Web 2.0 because it will be powered by blockchain technology. Blockchain is a secure and tamper-proof way of storing data.

Transparency: Web 3.0 will be more transparent than Web 2.0 because all transactions will be recorded on the blockchain. This will make it easier for users to track their data and ensure that it is being used in a way that they approve of.

In conclusion Web3 refers to the next generation of the internet that aims to create a decentralized and user-centric online experience. It leverages blockchain technology and peer-to-peer networks to enable direct interactions between users, removing the need for intermediaries and giving individuals more control over their data and digital assets.

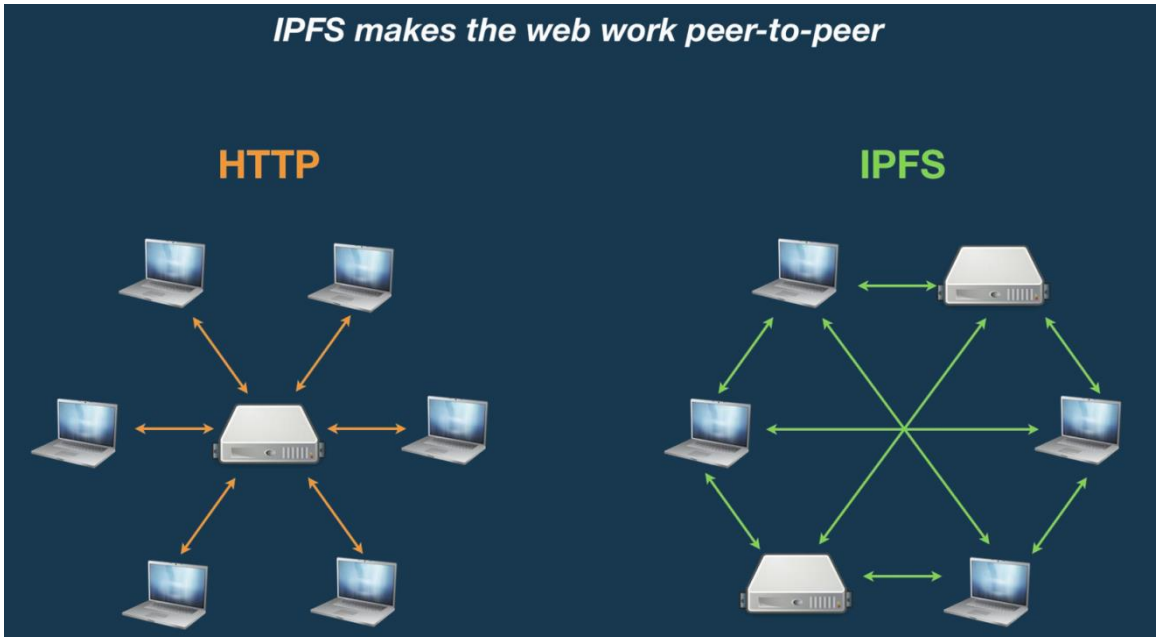
## IPFS

IPFS: stands for InterPlanetary File System, which is a protocol and peer-to-peer network designed to create a decentralized and distributed file storage system. Unlike traditional file systems that rely on centralized servers, IPFS aims to enable a more resilient and efficient way of storing and sharing files across the internet.

Content-addressed and decentralized: In IPFS, files are identified and retrieved based on their content, using a unique cryptographic hash derived from the file's content. This content-addressing approach ensures that files are globally identified, making them resistant to tampering and enabling efficient distribution.

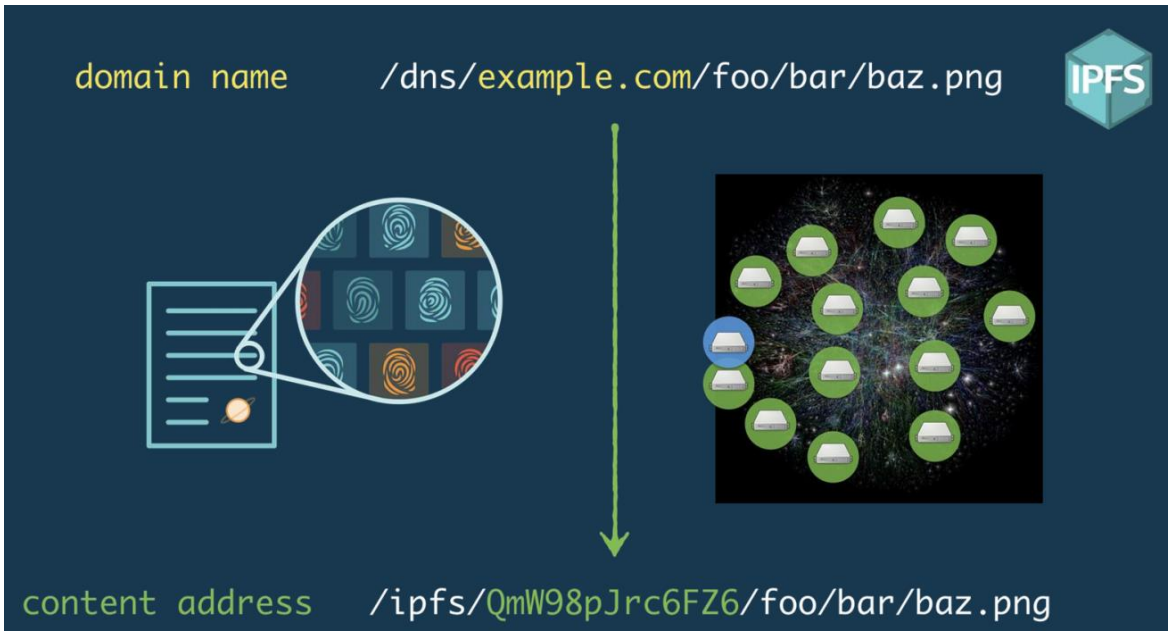
Distributed network: IPFS operates as a peer-to-peer network, where files are stored and shared by multiple nodes (computers) participating in the network. Each node hosts a portion of the files, and content is distributed across the network based on demand and availability.

Data redundancy and caching: IPFS promotes data redundancy by storing files across multiple nodes. This redundancy enhances fault tolerance and ensures that files remain accessible even if certain nodes go offline. Additionally, IPFS utilizes a distributed caching mechanism, in which frequently accessed files are cached closer to the requesting nodes, improving retrieval speed.




Versioning and deduplication: IPFS allows for versioning of files, meaning that multiple versions of a file can coexist and be accessed independently. Furthermore, IPFS utilizes deduplication techniques to reduce storage space by storing identical files only once, regardless of the number of copies in the network.

Collaboration and offline access: IPFS enables collaboration by allowing users to share files directly with each other, bypassing the need for centralized servers. Additionally, IPFS supports offline access, as files can be retrieved from nearby nodes even when internet connectivity is limited.



IPFS has various use cases, including distributed file storage, decentralized websites, decentralized applications (DApps), content delivery networks (CDNs), and more. It offers a promising alternative to traditional centralized file systems, promoting censorship-resistant and scalable file sharing.

Simply:

- IPFS is a decentralized storage and delivery network which builds on fundamental principles of P2P networking and content-based addressing.
- All content authenticated.
- No central server - all peers are the same.
- Content is never pushed to a different peer when adding it, only downloaded upon request.
- Content can be anything, from scientific datasets to blockchains. You could even host websites on it!

## Gas & Gas Price

### What is Gas?

1. **Definition**: In the context of Ethereum and similar blockchains, "gas" refers to the computational effort required to execute operations, like transactions or smart contract executions.
2. **Purpose**: Gas measures how much work an action or set of actions takes to perform. Every operation that can be performed by the Ethereum Virtual Machine (EVM) is assigned a fixed amount of gas.
3. **Limiting Resource Abuse**: Gas helps to prevent spam on the network and allocates resources on the blockchain efficiently.

### What is Gas Price?

1. **Definition**: Gas price is the amount of Ether (ETH) a user is willing to pay for each unit of gas. It's usually measured in "Gwei" (1 Gwei = $10^{-9}$ ETH).
2. **User-Set Prices**: Users can set the gas price they are willing to pay when sending a transaction. Miners usually prioritize transactions with higher gas prices.
3. **Dynamic Pricing**: The gas price fluctuates based on network demand. Higher demand leads to higher gas prices.

### Gas Limit

1. **Definition**: The gas limit is the maximum amount of gas the user is willing to spend on a transaction or smart contract execution.
2. **Transaction Cost**: The total cost of a transaction is calculated as **Gas Units * Gas Price**. For example, if a transaction uses 21,000 gas units and the gas price is 50 Gwei, the cost will be 0.00105 ETH.

3. **Refunds and Excess Gas**: If less gas is used than the limit, the excess is refunded to the user. However, if the gas limit is too low and the transaction requires more gas, it will fail, and the spent gas is not refunded.

### Estimating Gas Costs

- **Estimation Tools**: Many Ethereum wallets and interfaces automatically suggest an appropriate gas price and limit. There are also websites that provide real-time data on the average gas prices on the network.
- **Smart Contract Complexity**: The more complex a smart contract, the more gas it will require. Developers need to optimize smart contract code to minimize gas costs.

### Other Blockchains

- **Ethereum Alternatives**: Other blockchains like Binance Smart Chain or Polygon also use a similar concept of gas, but the cost structures can be different.
- **Layer 2 Solutions**: Solutions like Optimism or Arbitrum on Ethereum aim to reduce gas costs by handling transactions off the main Ethereum chain.

## ABI

ABI, or Application Binary Interface, is a critical concept in the context of Ethereum and other blockchain platforms that support smart contracts. It plays a vital role in the interaction between smart contracts and external applications or clients, including those written in languages like JavaScript, Python, or others.

### Understanding ABI in Blockchain

1. **Definition**: ABI is the standard way to interact with smart contracts in the Ethereum ecosystem, defining how data is formatted and encoded/decoded for transactions and calls between smart contracts and external clients.
2. **Functionality**: It acts like an interface between two binary program modules, often between smart contracts and the external world. It's akin to an API in the web development domain.
3. **Components**: An ABI specifies various details about the smart contract, including:
    - Available functions and their signatures.
    - How to call these functions.
    - The correct way to format data to interact with these functions.
    - Event types and how they are logged/emitted.

### Why is ABI Important?

1. **Smart Contract Interaction**: To interact with a smart contract (for instance, to execute its functions), you need to know the contract's ABI. It allows applications and clients to encode function calls in a way that the contract understands.
2. **Decoding Data**: The ABI is also used to decode data coming from the smart contract, such as return values from functions.

3. **Interoperability**: It ensures that different applications, possibly written in different programming languages, can interact with the smart contract in a standard and predictable way.

### How ABI Works

1. **Generation**: When a smart contract is compiled, the ABI is generated alongside the bytecode. The bytecode is what gets deployed to the blockchain, while the ABI is used by client applications.
2. **Usage in Applications**: Developers typically use the ABI in their client-side applications by importing it. This allows the application to construct and decode calls and transactions to interact with the smart contract.
3. **Encoding and Decoding**: The ABI details how arguments to functions should be encoded before being sent to the blockchain, and how return values should be decoded when they are received.

**Practical Application**

- **Smart Contract Development**: When developing a DApp (decentralized application), the front-end part of the application will use the ABI to interact with smart contracts deployed on the blockchain.
- **Tools and Frameworks**: Development frameworks like Truffle, Hardhat, and web3.js or ethers.js libraries in JavaScript use the ABI to facilitate interaction with smart contracts.
- **Updating Contracts**: If a smart contract is updated and redeployed, its ABI may change, necessitating updates in all applications that interact with it.

## Solidity

Solidity: is a programming language specifically designed for developing smart contracts on blockchain platforms, primarily Ethereum. It is a statically-typed, contract-oriented language that allows developers to write code for decentralized applications (DApps), decentralized autonomous organizations (DAOs), and other blockchain-based projects.

There is another programming language for smart contracts like Vyper, but we will talk about solidity.

Each Smart Contract has its own address.

### Solidity Syntax

First Thing introduce the compiler version of the smart contract to let the compiler of solidity knows what the version is used.

```
1. pragma solidity 0.8.8; // a fixed compile Version
```

```
1. pragma solidity ^0.8.0; // that version or any later version
```

```
1. pragma solidity >=0.8.0 <0.9.0; // any Version Between this Range
```

Define The License Of the smart Contract in the first line of code (Optional)

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

## Data types

- *Boolean*: Represents true or false values. It's declared using bool.
    - Could use the same logical and Comparisons operators as any language.

```
1. bool isTrue = true;
```

- *Integers*: Solidity supports both signed and unsigned integers of different sizes, including int8, int16, int256, uint8, uint16, uint256, and more.
    - Could use arithmetic operators and bitwise , shift operators as other languages.

```
1. int8 myInt = -42; // signed with the size of 8 bits
2. uint256 myUInt = 12345; // unsigned with the size of 256 bits (max)
3. int exponentiation = 0**0; // defined as 1
4. uint exp2 = 1e18; // is equal to 1 * 10**18
```

- *Address*: Represents Ethereum addresses. It can store the 20-byte address of an external account or contract. It's declared using address.

```
1. address payable recipient = 0x1234567890123456789012345678901234567890;
2.  // payable if u wanna pay the address , it's optional.
```

- *Bytes*: Solidity supports fixed-size and dynamic-size byte arrays. For example:
    - bytes: Fixed-size byte array.
    - bytes1, bytes2, ..., bytes32: Fixed-size byte arrays with specific sizes.
    - bytes[]: Dynamic-size byte array.

```
1. bytes32 myBytes = 0x1234567890abcdef; // max size is 32 for bytes
```

- *Strings*: Represents variable-length strings. It's declared using string.

```
1. string greeting = "Hello, World!"; // as an array of bytes
```

- *Enum*: Allows you to create custom enumerations with a finite set of values.

```
1. enum Status { Pending, Approved, Rejected }
2. Status myStatus = Status.Pending; // value of 0 for the first one.
```

- *Structs*: Custom user-defined composite data types, which allow you to group variables of different types together.

```
1. struct Person {
2.     string name;
3.     uint256 age;
4. }
5. Person alice;
6. alice.name = "Alice";
7. alice.age = 30;
```

- *Arrays*: Solidity supports both fixed-size and dynamic arrays for all data types, including user-defined types (struct).

```
1. uint256[] dynamicArray;
2. dynamicArray.push(7); // Add an element to the dynamic array
3. dynamicArray.pop();  // Remove the last element from the dynamic array
4. uint256 length = dynamicArray.length;  // Get the length of the dynamic array
5. uint256 x = dynamicArray[length - 1]; // store last element of array in a variable
6. Person[5] fixedArray; // define a fixed struct array
7. fixedArray[index] = _value;  // Set the value at a specific index
8. uint256[3] fixedArray = [10, 20, 30];  // Initialize a fixed-size array with values
```

- *Mappings*: Key-value data structures that are like hash maps or dictionaries in other languages.
- Mappings in Solidity have the following characteristics:
    - Efficient Look-Up: Mappings are very efficient for looking up and retrieving values associated with a specific key (address in the example). The look-up operation is constant time (O(1)).
    - No Iteration: Unlike arrays and structs, mappings do not support iteration. You cannot loop through all the keys or values in a mapping directly.
    - Undefined Default Value: If you try to access a key that has not been set in the mapping, it returns the default value for the value type. In the example, if an address is not in the balances mapping, accessing it will return 0.
    - State Variables: Mappings are often used to manage state variables within smart contracts, allowing you to track various data for different addresses or entities.
    - Cannot be Returned: You cannot return the entire mapping from a function. If you need to access the mapping from external code, you typically provide individual access functions to return key's value by another.

```
1.  mapping(address => uint256) balances;
2.  balances[“0x123”] = 1; // assign a value to map by accessing it with the key
3.  uint256 balance = balances[_address]; // store the value of map key in variable
```

- Fixed Point Numbers : Fixed point numbers are not fully supported by Solidity yet. They can be declared but cannot be assigned to or from.

## Casting:

- *Implicit Casting (Widening)*: This is a type of casting where a smaller data type is automatically converted to a larger data type without the need for explicit casting.

```
1. uint256 a = 42;
2. int256 b = a; // Implicit casting from uint256 to int256
```

- *Explicit Casting (Narrowing)*: Explicit casting is required when you want to convert a larger data type to a smaller one. This may result in data loss or truncation if the value cannot be accurately represented in the smaller data type.

```
1. int256 x = 1000;
2. int8 y = int8(x); // Explicit casting from int256 to int8 (data may be truncated)
```

- *Address Casting*: You can cast between different address types in Solidity. For example, you can cast from address to address payable and vice versa.

```
address payable recipient = address(someAddress); // Cast from address to address payable
```

- *Bytes to BytesN*: You can cast from bytes to a fixed-size bytesN type, where N is the desired size. Be cautious when doing this as it may lead to data truncation.

```
bytes memory data = new bytes(10);
bytes4 dataAsBytes4 = bytes4(data); // Explicit casting to bytes4
```

- *Enum to Integer*: You can cast from an enum to an integer and vice versa. Enum values are implicitly convertible to their underlying integer representation.

```
enum Status { Pending, Approved, Rejected }
Status currentStatus = Status.Approved;
uint8 statusValue = uint8(currentStatus); // Cast from enum to integer
```

- *Integer to Enum*: You can cast from an integer to an enum, but you need to be careful to ensure the integer corresponds to a valid enum value.

```
enum Status { Pending, Approved, Rejected }
uint8 statusValue = 1;
Status currentStatus = Status(statusValue); // Cast from integer to enum
```

- *Bytes to String* (Not Recommended): Solidity doesn't provide a direct casting method to convert bytes to string. Converting between these types can be complex and is generally not recommended in most cases. It's better to use libraries or external functions to handle such conversions.

Immutable and constant: In Solidity, immutable and constant are two different concepts that pertain to state variables and functions in smart contracts. They serve different purposes and have distinct characteristics:

*Immutable State Variables*:

- immutable is a state variable keyword introduced in Solidity 0.6.0. It is used to declare a state variable whose value can be set only once, and it is fixed thereafter. It's a more efficient and safer alternative to declaring constant values because it is computed and stored during contract deployment.
- immutable variables are suitable for values that are known and determined at deployment and don't change during the contract's lifetime. They can be used to save gas costs compared to constant variables, which are recalculated at each access.
- Immutable state variables can be of elementary types (e.g., uint, address, string) or reference types like structs.

```
pragma solidity ^0.8.0;

contract MyContract {
    immutable uint256 public fixedValue = 42;

    // ...
```

```
7. }
```

*Constant Functions (View and Pure)*:

- Constant functions are functions that promise not to modify the state of the contract. There are two types of constant functions: view and pure.
- view functions are read-only and do not modify the state. They can read state variables and are free to call other view functions.
- pure functions are even more restrictive and do not access state variables. They are used for mathematical calculations and do not consume gas when called externally.
- Constant functions are used for querying the state of the contract, and they are often called by external actors or other contracts.

```
 1. pragma solidity ^0.8.0;
 2. contract MyContract {
 3.     uint256 public stateValue = 100;
 4.
 5.     function getValue() public view returns (uint256) {
 6.         return stateValue; // This function is 'view'
 7.     }
 8.
 9.     function multiply(uint256 a, uint256 b) public pure returns (uint256) {
10.         return a * b; // This function is 'pure'
11.     }
12. }
```

Contract: you can think as if it is a class in other languages like java. But once it executed, we can't stop or changing it's functionality Unless we have prepared functions that update the state of the Smart contract.

```
1. Contract contractName {
2.}
```

Interface: is a way to define a contract's structure without providing the implementation details. Interfaces are like blueprints that specify the function signatures that a contract must adhere to. They are typically used for creating a standard set of functions that other contracts can implement, ensuring interoperability and compatibility.

```
1. interface MyInterface {
2.     function myFunction(uint256 input) external view returns (uint256);
3. }
```

Use the Interface:

You can use an interface to interact with contracts that adhere to it. When you use the interface, you can call the functions declared in the interface without knowing the implementation details of the actual contract.

```
 1. contract ConsumerContract {
 2.     MyInterface public myContract;
 3.
 4.     constructor(address _myContractAddress) {
 5.         myContract = MyInterface(_myContractAddress);
 6.     }
```

```
 7.
 8.     function useMyFunction(uint256 input) public view returns (uint256) {
 9.         return myContract.myFunction(input);
10.     }
11. }
```

**We have for, while loop and if/else conditioning in solidity like in c and java.**

## Functions:

```
1. Function functionName (optionalParameters) accessModifier optkeyword returns(Parameters)
2. {
3.        return values;
4. }
```

**accessModifier (Visibility Specifiers) are one of those:**

1. public → visible for all
2. private → visible only for current contact
3. internal → visible only internally (between contracts) (default)
4. external → visible only for functions and open for any outside call.

**They could be used with variables and arrays and maps as well.**

**Optkeyword like view and pure (read only function):**

*View Functions* **(formerly Constant Functions):** View functions promise not to modify the state of the contract. They are used for querying the state and do not consume gas when called externally. They can be called by external and internal actors.

```
1. function myFunction() public view returns (uint256) {}
```

*Pure Functions*: Pure functions are like view functions, but they don't even read the state. They are used for mathematical calculations and are even more gas efficient. They can be called by external and internal actors.

**Preferred to use to reading (get) constants to safe gas than view.**

```
1. function myFunction(uint256 a, uint256 b) public pure returns (uint256) {}
```

*Fallback Functions (deprecated) and Receive Functions*: A receive function is triggered when Ether is sent to a contract without any data, and a fallback function is triggered when a contract receives a call without a matching function signature. The receive function is used to handle incoming Ether transactions, while the fallback function can be used to implement custom behavior.

```
1. receive() external payable {}
2. fallback() external {}
```

*Constructor*: The constructor function is executed only once when the contract is deployed. It's used to initialize the contract's state variables.

Key points about constructors in Solidity:

- Name: The constructor has the same name as the contract and does not have a return type (not even void). It's recognized by the constructor keyword.

- Initialization: You can use the constructor to initialize state variables and perform other setup operations for the contract. These initializations occur only during contract deployment.
- Visibility: The visibility of the constructor (e.g., public, internal, or private) affects who can deploy the contract. By default, the constructor is public, meaning anyone can deploy the contract. If you make it internal or private, only other contracts can deploy it.
- Parameters: Constructors can accept parameters just like regular functions. This allows you to customize the initialization of the contract based on input values.
- Multiple Constructors: Starting from Solidity version 0.6.0, you can have multiple constructors in the same contract with different parameter lists. This feature allows for constructor overloading based on the number and types of parameters.
- Inheritance: If your contract inherits from other contracts, constructors of the base contracts are also executed in a specific order. The constructor of the derived contract is executed last.
- When you deploy a smart contract, the constructor is executed to set up the initial state of the contract. Once the contract is deployed, the constructor cannot be called again, and its behavior is not visible to external users of the contract.

```
1. constructor() public {
2.     // Initialization logic
3. }
```

**Revert Vs require Vs assert:** revert and require are two mechanisms used for error handling and control flow within smart contracts. They are used to check conditions and revert the execution of a transaction or function if those conditions are not met. Both mechanisms prevent further execution and revert any state changes that occurred before the error condition was encountered.

Revert:

- revert is used to revert the entire transaction, including any state changes and gas consumption, back to the previous state.
- It is typically used when a critical error condition is encountered, and you want to stop the execution of the transaction and revert all changes.
- You can provide an optional error message to explain the reason for the revert.

Require:

- require is used to check a condition and revert the transaction if the condition evaluates to false. It is used to validate inputs and conditions within a function.
- Unlike revert, require does not consume all the gas; it refunds any remaining gas to the sender. It is often used for input validation.
- You can provide an optional error message to explain why the condition was not met.

```
 1. function someFunction(uint256 value) public {
 2.     require(value > 0, "Value must be greater than 0");
 3.     // Continue execution if the condition is met; refund unused gas
 4. // Perform other operations}
 5.
 6. function anotherFunction() public {
 7.     if (someCondition) {
 8.         revert("Critical error: Something went wrong");
 9.     }
10.     // Continue execution if no revert occurred
11. }
12.
```

The key differences between revert and require are the extent of gas consumption and whether they refund unused gas:

- revert consumes all the gas and reverts all state changes and operations, making it suitable for critical errors.
- require refunds any unused gas and is often used for input validation and non-critical conditions.
- You can choose between revert and require based on the context and severity of the error condition. Use revert for critical errors that should completely halt the transaction, and use require for input validation and conditions that are not critical to the functioning of the contract.

**Assert:**

- Use assert for internal consistency checks within your contract. assert should not be used for input validation.
- It consumes all the gas, even if the condition is true, and is primarily used for checks that should never fail.

```
1. function someInternalFunction(uint256 value) internal {
2.     assert(value > 0);
3.     // Continue execution if the condition is true
4. }
```

- When an error occurs, the transaction is reverted, and any changes to the contract's state are undone. The user who initiated the transaction will receive an error message if a custom error message was provided.

Define a Custom Error: it saves more gas than storing message and pass it with revert.

```
1. error ValueMustBeGreaterThanZero(string message);// before define a contract , could have
2. no parameters
```

```
1. function checkValue(uint256 value) public pure {
2.     if (value <= 0) {
3.         revert ValueMustBeGreaterThanZero("Value must be greater than zero");
4.     }
5.     // Continue execution if the condition is met
6. }
```

**You could handle the reverted error at the client side or DApp with try-catch.**
**See the try-catch example here.**

*Modifier*: modifier name written next to access modifier of a function.

```
1. address public owner;
2. modifier onlyOwner() { // could pass parameters to it from the function calling it.
3.    require(msg.sender == owner, "Only the owner can call this function");
4.    _; // for executing the remaining function that use the modifier if not reverted.
5. }
```

**Each function type serves a specific purpose and provides different access and gas consumption characteristics. Solidity's flexibility in defining function types allows developers to design contracts that meet their specific requirements.**

**Functions can be overloaded as in other programming languages.**

**It is possible to return multiple objects in solidity function.**

```
1. function getMultipleValues() public onlyOwner() pure returns (uint256, string memory) {
2.         uint256 number = 42;
3.         string memory text = "Hello, World!";
4.         return (number, text);
```

```
5.     }
```

## Inheritance:

Inheritance is a fundamental feature in Solidity that allows you to create new smart contracts based on existing ones. With inheritance, you can reuse and extend the functionality of one or more base contracts. This promotes code reusability, modularity, and the implementation of the "is-a" relationship between contracts. Inheritance in Solidity is like class inheritance in object-oriented programming languages.

```
0. import "./Base3.sol"; // importing the parent contracts
1. contract MyContract is Base1, Base2, Base3 {
2.   // ... could override the functions of bases and use also by calling them and if a base
3. constructor has parameters we must pass it from the child constructor by calling
4. parentName(parametersPass) next to child constructor initialization, base could be also
5. an interface (Implement).
6.}
```

Start by defining a base contract with a function that is marked as virtual. This means that the function can be overridden in derived contracts.

```
1. pragma solidity ^0.8.0;
2. contract BaseContract {
3.     function myFunction() public virtual pure returns (string memory) {
4.         return "BaseContract";
5.     }
6. }
```

```
1. contract DerivedContract is BaseContract {
2.     function myFunction() public pure override returns (string memory) {
3.         return "DerivedContract";
4.     }
5. }
```

## Storage Vs Memory:

memory is a keyword used to store data for the execution of a contract. It holds functions argument data and is wiped after execution.

storage can be seen as the default solidity data storage. It holds data persistently and consumes more gas.

| storage | memory |
|---|---|
| Stores data in between function calls | Stores data temporarily |
| The data previously placed on the storage area is accessible to each execution of the smart contract | Memory is wiped completely once code is executed |
| Consumes more gas | Has less gas consumption, and better for intermediate calculations |
| Holds array, state and local variables of struct | Holds Functions argument |

Use **storage** for data that needs to be permanently stored and shared among functions and external actors.

Use memory for temporary data needed during the execution of a function, which does not need to be stored on the blockchain. It is more gas-efficient and has a local scope.

## New Keyword:

- the new keyword is used to create a new instance of a contract, effectively deploying a new contract on the Ethereum blockchain. It is primarily used for creating contract instances from contract templates or to interact with existing contracts. Here's how the new keyword is used:
- Creating New Contract Instances:
    - You can use the new keyword to create new instances of a contract. This is often used to deploy a new contract on the blockchain.

```
1. pragma solidity ^0.8.0;
2. contract MyContract {
3.     // Contract state and functions
4. }
```

To create a new instance of the MyContract contract, you can use the new keyword as follows:

```
1. MyContract myContractInstance = new MyContract(); // This deploys a new instance of
MyContract, and you can interact with it just like any other Solidity contract.
```

- Contract Interaction:
    - You can use the new keyword to interact with existing contracts and create new contract instances within your smart contract code.

```
 1. pragma solidity ^0.8.0;
 2. contract CallerContract {
 3.     MyContract public myContractInstance;
 4.     constructor() {
 5.         myContractInstance = new MyContract(); // CallerContract deploys a new instance
of MyContract when it is deployed.
 6.     }
 7. }
 8. contract MyContract {
 9.     // Contract state and functions
10. }
```

- Contract Upgrades:
    - The new keyword can also be used for upgrading smart contracts by deploying a new version of the contract and migrating data and state from the old contract to the new one. This is commonly used in the context of upgradable contracts.

    - It's important to note that the new keyword is not typically used for deploying complex, upgradable contracts, as it can lead to data migration challenges. More sophisticated approaches, such as proxy patterns, are often used for upgradability in Solidity.

## ways to transfer Ether (the cryptocurrency on the Ethereum network) between addresses or contracts:

- *transfer Function*: The transfer function is a method available on all address types in Solidity. It's a simple and safe way to send Ether and is often used for basic fund transfers.

- The transfer function is available on address payable types, and it sends the specified amount of Ether to the recipient. It's a simple way to avoid reentrancy issues.

```
address payable recipient = 0x1234567890123456789012345678901234567890;
function transferEther() public {
    recipient.transfer(msg.value);
}
```

- *send Function*: Similar to transfer, the send function is also used to send Ether, but it returns a boolean value indicating whether the transfer was successful or not.
- While send returns a boolean, it's important to note that relying solely on the return value for control flow can be dangerous due to the gas stipend and reentrancy issues. Therefore, it's often recommended to use transfer or the more advanced call function.

```
address payable recipient = 0x1234567890123456789012345678901234567890;
function sendEther() public {
    bool success = recipient.send(msg.value);

    if (!success) {
        // Handle failure
    }
}
```

- *call Function*: The call function is a more versatile way to send Ether and interact with other contracts. It returns a boolean value, and it can be used to call functions on other contracts, including handling the return values.
- The call function allows you to specify the amount of Ether to send along with additional data (function signature and parameters) if you are calling another contract.

```
address payable recipient = 0x1234567890123456789012345678901234567890;

function callTransfer() public {
    (bool success, ) = recipient.call{value: msg.value}(""); // you could store any of
the returned parameter from the function that returns more than 1 value.
    if (!success) {
        // Handle failure
    }
}
```

**Always be cautious when dealing with Ether transfers in Solidity to avoid security vulnerabilities, such as reentrancy attacks. Also, consider using more advanced patterns, such as the Withdrawal Pattern or checks-effects-interactions pattern, for more complex contract interactions involving Ether transfers.**

Events: events are a way for a contract to communicate that something has happened on the blockchain to the external world. They are a mechanism for logging and listening to specific occurrences within the contract. Events are defined using the event keyword, and they are emitted using the emit keyword.

```
pragma solidity ^0.8.0;

contract EventExample {
    event SomethingHappened(address indexed user, uint256 value);

    function doSomething(uint256 value) public {
        // Perform some actions
```

```
        emit SomethingHappened(msg.sender, value);
    }
}
```

- The SomethingHappened event is defined with two parameters: user and value.
- The doSomething function performs some actions and then emits the SomethingHappened event with relevant data.
- types of parameters of events:
    - Indexed Parameters:
        - You can declare up to three parameters as indexed in an event. These parameters allow for more efficient filtering when querying the event logs.
        - In the example above, user is declared as indexed. This allows clients to efficiently filter events based on the user parameter when searching through the blockchain's event logs.
    - Non-Indexed Parameters:
        - Parameters that are not explicitly declared as indexed are non-indexed parameters.
        - Non-indexed parameters provide additional data about the event but do not offer the same level of efficient filtering when querying event logs.
- Events are typically used in decentralized applications (DApps) to notify the frontend about state changes, log significant transactions, or trigger other processes. They provide a way for off-chain applications to listen for events emitted by on-chain smart contracts.
- When interacting with smart contracts from a frontend or another contract, you can subscribe to events to receive notifications about specific actions or changes in the state of the contract. This allows for a more responsive and interactive user experience in decentralized applications.

## Some Built-in functions and attributes:

- *block.chainid* : returns the current chain ID.
    - The chain ID is a unique identifier for a specific Ethereum network (e.g., Mainnet, Ropsten, Rinkeby, etc.).
- *block.coinbase* : returns the current block's miner address.
    - It represents the address that will receive the block reward for successfully mining the block.
- *block.difficulty* : returns the current block's difficulty level.
    - Difficulty is a measure of how difficult it is to find a new block and is adjusted dynamically to maintain a consistent block time.
- *block.gaslimit* : returns the gas limit of the current block.
    - It represents the maximum amount of gas that can be consumed by all transactions in the block.
- *block.number* : returns the current block number.
    - It represents the position of the block in the blockchain.
- *block.timestamp* : returns the current block's timestamp (Unix timestamp).
    - It represents the time when the block was mined.
- *msg.sender* : is a global variable representing the address that sent the current transaction.
    - It can be used to identify the sender of the transaction.
- *msg.sig* : is a global variable representing the first four bytes of the calldata, i.e., the function signature of the called function.
    - It can be used to identify the function being called.

- *msg.value* : is a global variable representing the amount of Ether (in wei) sent with the message (transaction).
    - It can be used to access the value sent in a transaction.
- *msg.data*: is the complete calldata of the transaction which is a non-modifiable, non-persistent area where function arguments are stored and behave mostly like memory.
    - It includes the function signature and parameters of the function being called.
- *msg.gas*: is the amount of gas remaining for the execution of the current call.
    - It provides information about the gas limit for the current call.
- *tx.origin*: is a global variable representing the address of the sender of the transaction that originated this execution.
    - It should generally be avoided due to security concerns related to the potential for unexpected contract interactions.
- *block.basefee*: returns the base fee of the current block in Wei.
    - It represents the minimum gas price required for transactions to be included in the block.
- *abi.encode* : is used to encode multiple values into a single packed byte array.
    - It is commonly used when constructing function call data for external function calls.

```
1. function encodeValues(uint256 a, uint256 b) public pure returns (bytes memory) {
2.     return abi.encode(a, b);
3. }
```

- *abi.encodePacked*: is similar to abi.encode but does not pad the values to match the storage layout.
    - It packs the values tightly without adding any padding.

```
1. function encodePackedValues(uint256 a, uint256 b) public pure returns (bytes memory) {
2.     return abi.encodePacked(a, b);
3. }
```

- *abi.encodeWithSignature*: is used to encode the function selector and arguments for a function with a known signature.
    - It simplifies the process of creating calldata for function calls.

```
1. function encodeFunctionCall(uint256 a, uint256 b) public pure returns (bytes memory) {
2.     return abi.encodeWithSignature("myFunction(uint256,uint256)", a, b);
3. }
```

- *abi.encodeWithSelector*: is similar to abi.encodeWithSignature but only includes the function selector without the argument names.
    - It's commonly used when the argument names are not needed.

```
1. function encodeFunctionCall(uint256 a, uint256 b) public pure returns (bytes memory) {
2.     return abi.encodeWithSelector(bytes4(keccak256("myFunction(uint256,uint256)")), a,
b);
3. }
```

- *abi.decode*: is used to decode a byte array back into individual values.
    - It is commonly used when parsing returned data from external calls.

```
1. function decodeValues(bytes memory data) public pure returns (uint256, uint256) {
2.     (uint256 a, uint256 b) = abi.decode(data, (uint256, uint256));
3.     return (a, b);
4. }
```

- *keccak256*: also known as SHA3-256 in Ethereum, is the most used hashing function in Solidity.
    - It takes arbitrary-length input and produces a 256-bit (32-byte) hash.

```
function hashData(string memory data) public pure returns (bytes32) {
    return keccak256(abi.encodePacked(data));
}
```

- *sha256*: is similar to keccak256 but produces a SHA2-256 hash.
    - Note that sha256 is less commonly used in Ethereum smart contracts than keccak256.

```
function sha256HashData(string memory data) public pure returns (bytes32) {
    return sha256(abi.encodePacked(data));
}
```

- *ripemd160*: produces a RIPEMD-160 hash, which is often used in Ethereum for address generation.
    - It produces a 160-bit (20-byte) hash.

```
function ripemd160HashData(string memory data) public pure returns (bytes20) {
    return ripemd160(abi.encodePacked(data));
}
```

- *ecrecover*: is used in conjunction with elliptic curve cryptography to recover an Ethereum address from a message, a signature, and the hash of the message.
    - It returns the address corresponding to the public key used to sign the message.

```
function recoverSigner(bytes32 hash, uint8 v, bytes32 r, bytes32 s) public pure returns (address) {
    return ecrecover(hash, v, r, s);
}
```

- *Strings*:

```
import "@openzeppelin/contracts/utils/Strings.sol";

Strings.toString(uint256)// to convert an int to a string.
string.concat(a, b); // where a , b are strings
```

- *Print in console for debugging*:

```
import "hardhat/console.sol";
 // in the contracts function, when we call a function it will print in console.
console.logBytes32(bytes32);
console.logBytes(bytes);
console.log(String/uint256);
```

## library

the **library** keyword is used to define a special type of contract. A library in Solidity is a collection of reusable functions and is somewhat similar to a class in object-oriented programming, but with some distinct characteristics and purposes.

**Characteristics of Solidity Libraries**

1. **No State**: Libraries cannot hold their state. They cannot have state variables, and they cannot send Ether.

2. **Reusable Code**: The primary purpose of a library is to include reusable code that can be shared across multiple contracts. Functions within a library can be called without creating an instance of that library.
3. **Gas Efficiency**: Using libraries can help in saving gas costs. If a function from a library is called, it's executed in the context of the calling contract. This is more gas-efficient than deploying the same function in multiple contracts.
4. **Static Functions**: Functions inside a library are often marked as **internal** and are thereby only callable within the library itself or from contracts that explicitly use the library.
5. **Using For**: The **using for** directive is used in Solidity to attach library functions to types. This allows types to be treated as objects with methods, in a manner somewhat analogous to object-oriented programming.

### *Best Practices for Using Libraries*

- **Optimization**: Use libraries for functions that are used across multiple contracts to optimize code reusability and gas costs.
- **Security**: Ensure that the library code is secure and tested, as vulnerabilities in libraries can affect all contracts that use them.
- **Versioning**: Keep track of library versions, especially when updating library code, as changes can have widespread implications on dependent contracts.

```
1. // SPDX-License-Identifier: MIT
2. pragma solidity ^0.8.0;
3.
4. // Define a library
5. library Math {
6.     function add(uint x, uint y) internal pure returns (uint) {
7.         return x + y;
8.     }
9. }
10.
11. // Use the library in a contract
12. contract Test {
13.     using Math for uint;
14.
15.     uint public result;
16.
17.     function addNumbers(uint x, uint y) public {
18.         result = x.add(y); // Using the 'add' function from the Math library
19.     }
20. }
21.
```

## Fund Me Project

We Try Here To Create a Smart Contract that Initialize the Deployer as the owner and then anyone could Fund that Smart Contract but with minimum 50 USD Value Which we Use an Off-Chain to get The Funded Value Amount in USD instead of ETH, So if the Funded Value Amount is less than 50 USD Then Revert that Transaction.

The Owner can withdraw the Balance of the smart contract to his account.

Fund.sol

```
1. // SPDX-License-Identifier: GPL-3.0
```

```
 2.
 3. pragma solidity >=0.8.2 <0.9.0;
 4.
 5. import "./price.sol";
 6.
 7. contract Fund {
 8.
 9.     using PriceConv for uint256;
10.     uint256 constant usd = 50 * 1e18; // 18 zeros
11.     mapping (address => uint256) public funders;
12.     address[] public afunders;
13.     address public immutable owner;
14.
15.     constructor() {
16.          owner = msg.sender;
17.     }
18.
19.     function fundMe() public payable {
20.         require(msg.value.getConversionRate() >= usd , "pay more");
21.         afunders.push(msg.sender);
22.         funders[msg.sender] += msg.value; // in wei
23.     }
24.
25.    function withdraw() public  {
26.
27.        require(msg.sender == owner , "not the owner");
28.         for (uint256 funderIndex=0; funderIndex < afunders.length; funderIndex++){
29.             address funder = afunders[funderIndex];
30.             funders[funder] = 0;
31.         }
32.         afunders = new address[](0);
33.      // transfer
34.         // payable(msg.sender).transfer(address(this).balance);
35.      // send
36.         // bool sendSuccess = payable(msg.sender).send(address(this).balance);
37.         // require(sendSuccess, "Send failed");
38.      // call
39.         (bool callSuccess, ) = payable(msg.sender).call{value:
address(this).balance}("");
40.         require(callSuccess, "Call failed");
41.     }
42.
43.     receive() external payable {
44.         fundMe();
45.     }
46.     fallback() external payable {
47.         fundMe();
48.     }
49. }
50.
```

Price.sol

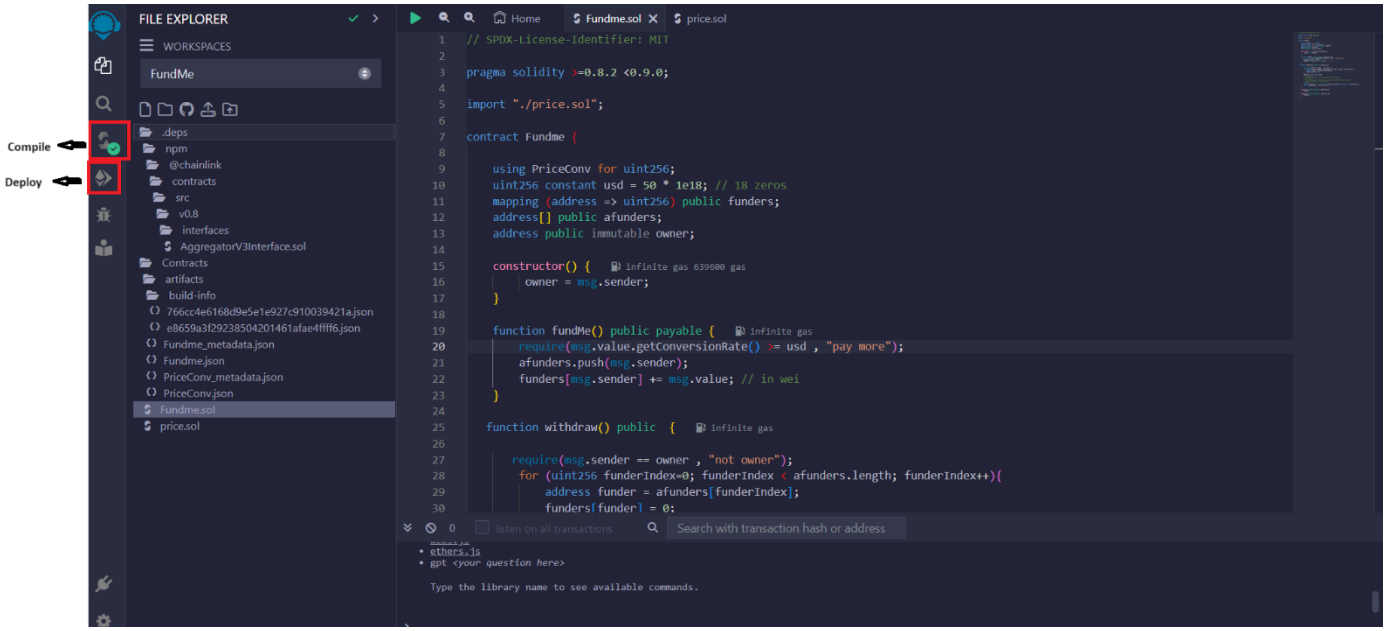
```
 1. // SPDX-License-Identifier: GPL-3.0
 2.
 3. pragma solidity >=0.8.2 <0.9.0;
 4.
 5. import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";
 6.
 7. library PriceConv {
 8.
 9.
10.     function getPrice() internal view returns (uint256){
```

```
11.         AggregatorV3Interface priceFeed =
AggregatorV3Interface(0x694AA1769357215DE4FAC081bf1f309aDC325306);
12.         (,int256 price , ,  ,) = priceFeed.latestRoundData();
13.         return  uint256(price * 1e10); // 10 y?
14.     }
15.
16.     function getConversionRate(uint256 etAmount) internal view returns(uint256) {
17.         uint256 etPrice = getPrice();
18.         uint256 et2usd = (etPrice * etAmount) / 1e18;
19.         return et2usd;
20.     }
21.
22.
23. }
24.
```

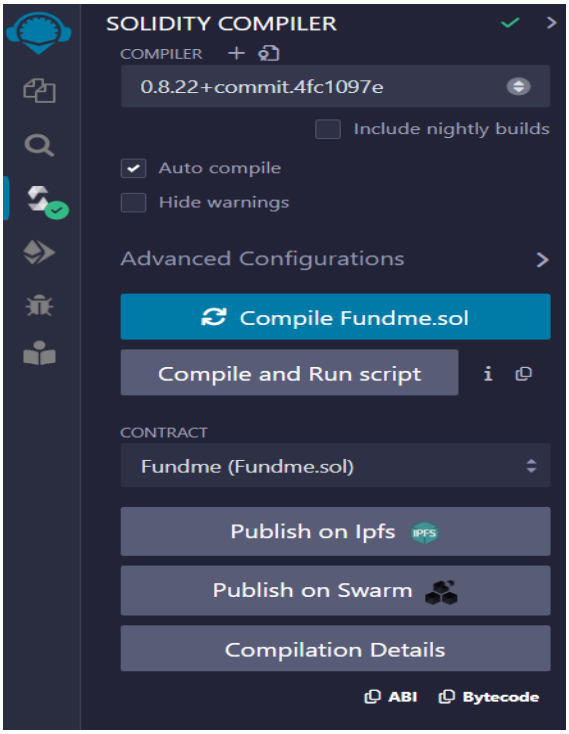
The Use address `0x694AA1769357215DE4FAC081bf1f309aDC325306` comes from the Chain-link Documentation where it is considered as a Blockchain Oracle That Retrieve Off-chain Data to My Smart Contract Which in our case we get the current ETH/USD Rate For Sepolia Network and for Other Networks we Could get from the Documentation.

## Deploy and Test

Go to Remix IDE and paste The above code in contract and keep the file name as the contract

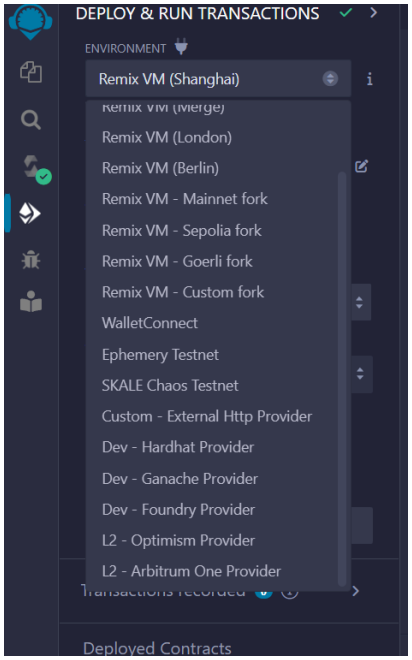



name after that choose the compile version that satisfies the requirements of compile version of pragma.

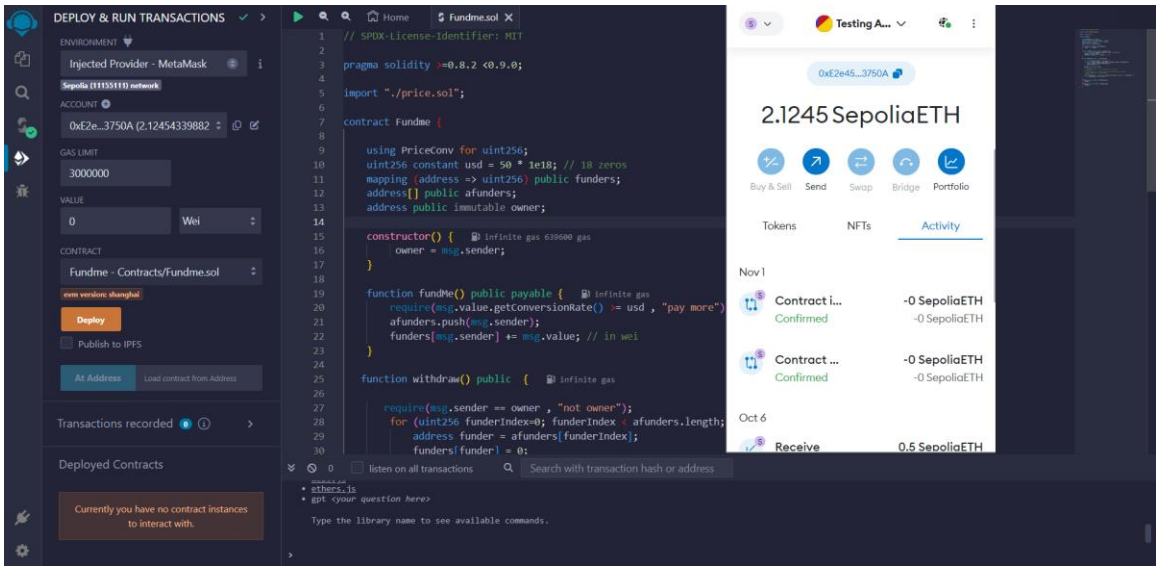



Once you compile successfully you will be able to deploy your smart contract into the network and then interact with it and you could use ABI and Bytecode also for Verifying the contract on [etherscan](etherscan).

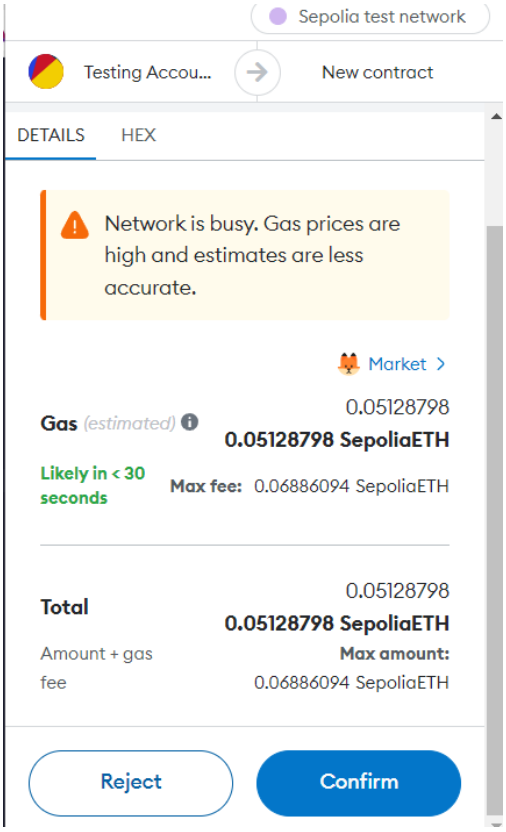
You can Deploy your smart contract on a local test of Remix as follows:



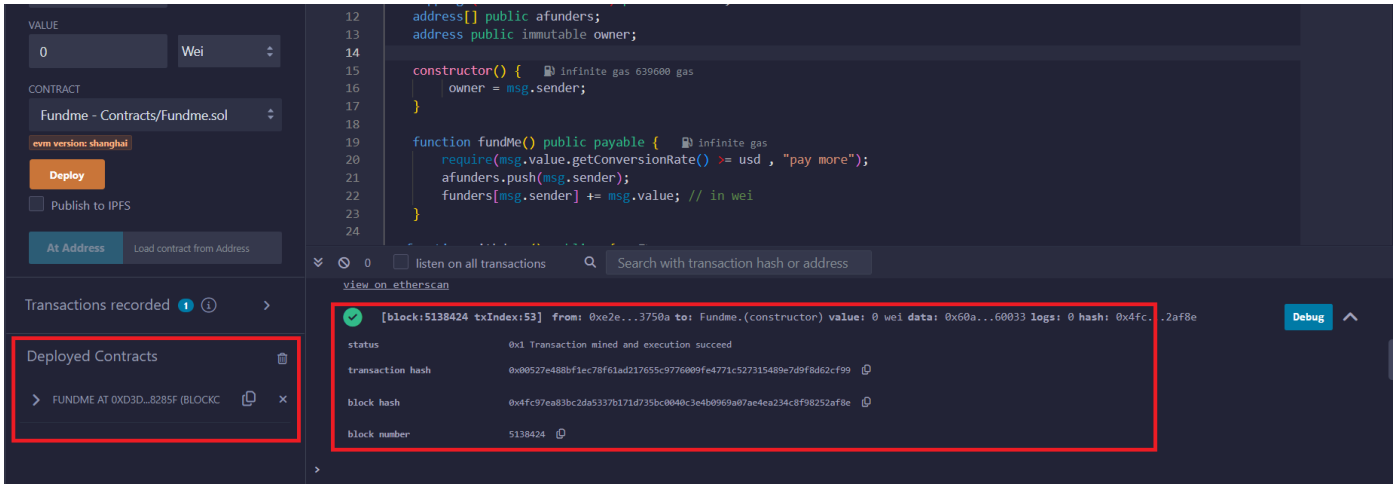
But in our case we will deploy on Spelophia test net to be able to Interact with it outside Remix, Choose Injected Provider after Establishing Your Wallet and Accounts on Crypto Wallets like MetaMask.




Choose the address to deploy and sign this transaction (the address is the public key of the account) and you will sign this transaction with your private key and asked to pay the gas fees.

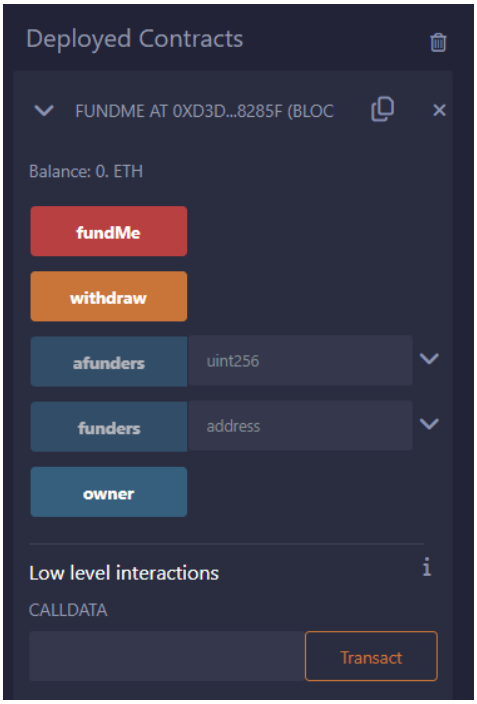


Now we successfully deployed our first smart contract.

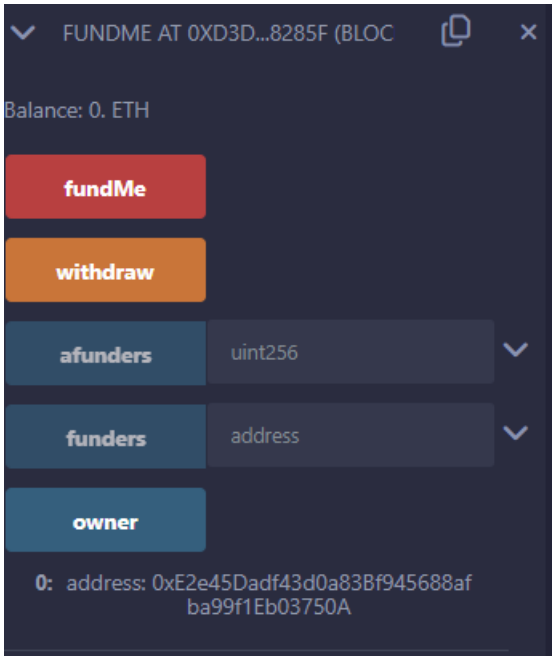



We are able to see block and transaction data also the deployed contract address and for more details we could find on etherscan, and on the left-side we (Deployed Contracts Section) it contains the Contacts that we have deployed each time we deploy a smart contract we are completely generate new contract so there will be different version when we update or do some edits and to keep the data to the new contract there are many methods like Proxy Contract Method.

Now let's interact with our contract by Remix UI, we will find for each function and each variable we declare a button that we could use.

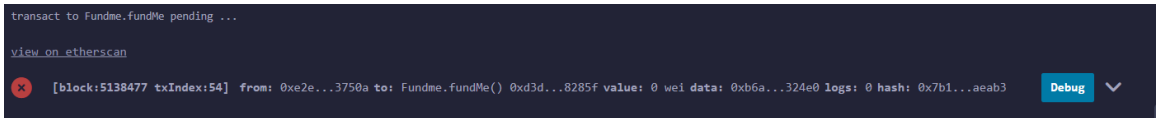


At first, we see that the contract balance is 0 because it hasn't funded yet, and if we press the owner, we will get the owner address which deployed the smart contract.

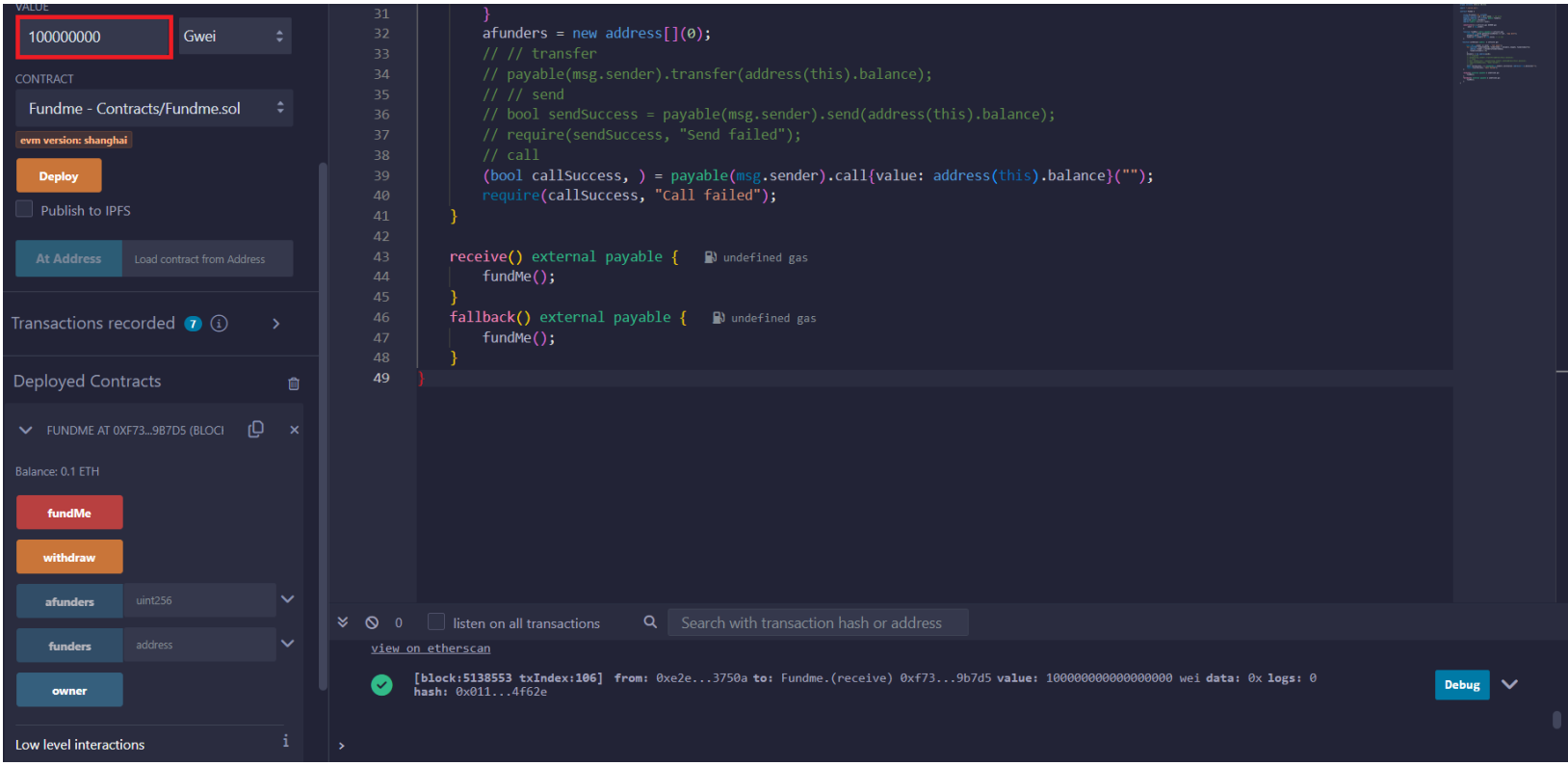



For retrieving data it hasn't gas fees because it doesn't modifying the blockchain status by making new transactions.

If we pressed fundMe the transaction reverted because we haven't funded with enough amount

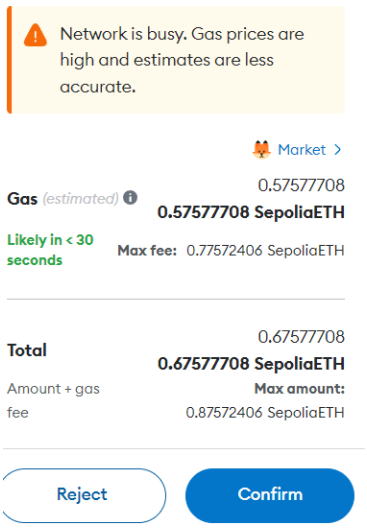



Enter the value to be sent and then hit the transact button and confirm the transaction then we notice that the balance of the smart contract is updated with the funded value.




finally, we are able to withdraw the balance of the smart contract to the owner account and we can see the transactions on MetaMask or etherscan.

In the last, I want to mention that when the network is busy the gas price become high and it is changed Periodically.




## Resources

- [Learn Blockchain, Solidity, and Full Stack Web3 Development with JavaScript – 32-Hour Course]

- Demystifying Blockchain — Part 1 (Anatomy of a Blockchain)
- Demystifying Blockchain — Part-2 (How a Blockchain Transaction works)
- The Evolution of the Internet: Web1, Web2, and the Web3 Revolution
- Web3 | PendHub
- Blockchain Facts: What Is It, How It Works, and How It Can Be Used
- Solidity Documentation
- What is IPFS