

# **Getting started with Solidity**

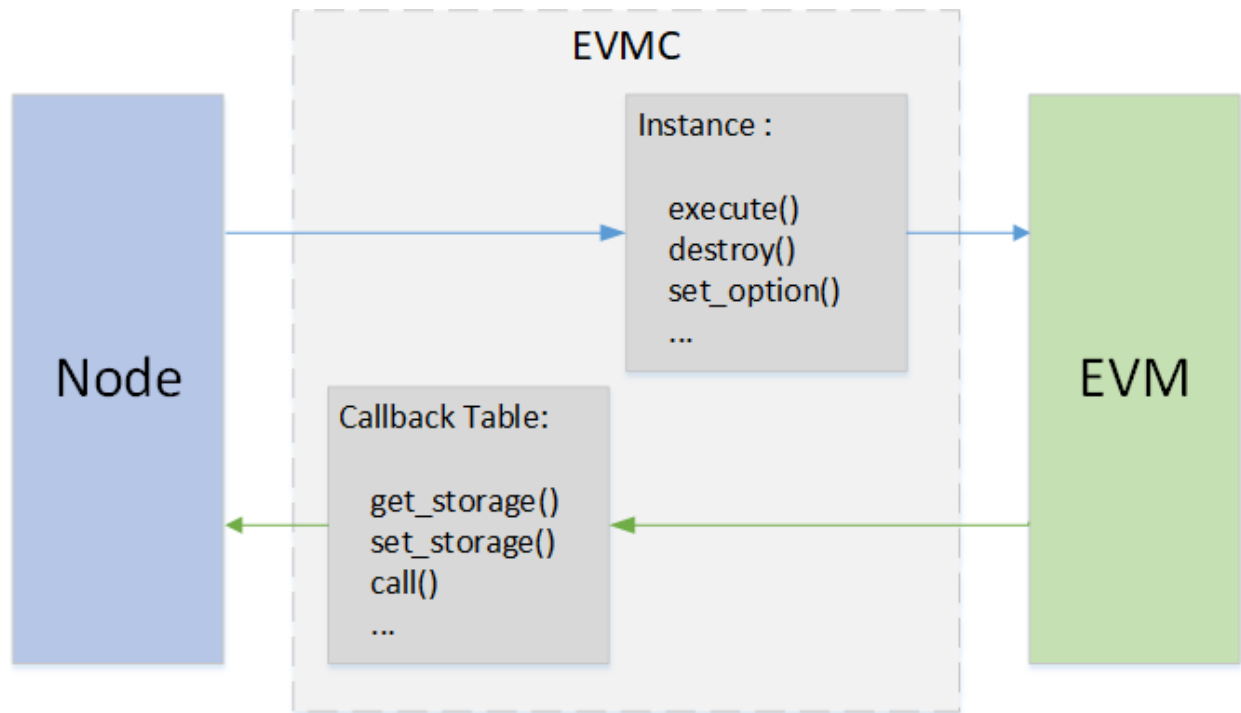
---

## **Introduction to solidity:-**

*Solidity is a programming language specifically designed for developing smart contracts on blockchain platforms, most notably the Ethereum network. It enables developers to write code that can be executed on the blockchain, allowing for the creation of decentralized applications (dApps) and self-executing contracts.*



## **What is EVM?**



*EVM stands for Ethereum Virtual Machine. It is a runtime environment that executes bytecode instructions, allowing the execution of smart contracts on the Ethereum blockchain. The EVM is a crucial component of the Ethereum network, as it provides the necessary infrastructure for running decentralized applications (dApps) and executing smart contracts in a secure and deterministic manner.*

### **Why Solidity ?**

*Solidity is specifically designed for developing smart contracts on the Ethereum platform, and it offers several functionalities that make it a powerful language for this purpose. Here are some key features and functionalities of Solidity:*

1. **Contract-oriented:** Solidity is a contract-oriented language, meaning it provides specific syntax and features for defining and implementing smart contracts. It allows developers to create reusable and modular contracts with properties like state variables, functions, events, and modifiers.
2. **Data Types:** Solidity supports a wide range of data types, including integers, booleans, strings, arrays, structs, mappings, and addresses. It also allows developers to define custom data types and structures, giving flexibility in storing and manipulating data within contracts.
3. **Modularity and Inheritance:** Solidity supports modularity through the use of libraries and inheritance. Developers can create libraries to encapsulate reusable code and inherit from other contracts to leverage their functionality. This promotes code reusability and makes contract development more efficient.
4. **Control Structures:** Solidity includes familiar control structures such as if-else statements, loops (for, while), and switch statements, allowing developers to implement complex decision-making and control flow within smart contracts.
5. **Events:** Solidity allows contracts to define and emit events. Events are a way to communicate and notify external applications or

*other contracts about specific occurrences within the contract. They enable off-chain systems to react to on-chain events.*

6. **Function Modifiers:** *Solidity provides function modifiers that allow developers to apply pre and post-conditions or implement access control to functions within a contract. Modifiers enhance code readability, maintainability, and security.*
7. **Ethereum Integration:** *Solidity is closely integrated with the Ethereum platform. It provides built-in functions and syntax for interacting with other contracts, accessing blockchain-specific functionalities (such as block information and timestamps), and handling Ether (Ethereum's native cryptocurrency) transfers.*
8. **Security Considerations:** *Solidity emphasizes security-conscious programming practices. It includes features like visibility modifiers (public, private, internal, external) to control access to functions and state variables. It also provides mechanisms to handle exception and error handling, preventing vulnerabilities like reentrancy attacks and integer overflow/underflow.*
9. **Tooling and Ecosystem:** *Solidity has a mature ecosystem with various development tools, libraries, frameworks, and testing frameworks. Tools like Remix, Truffle, and Hardhat provide*

*environments for writing, deploying, and testing Solidity contracts, making the development process more efficient and streamlined.*

## ***What are smart contracts and how it works ?***



*Smart contracts are self-executing contracts with the terms of the agreement written directly into lines of code. They are designed to automatically enforce the terms of the agreement and facilitate the exchange of assets or information between parties without the need for intermediaries.*

```

1  pragma solidity ^0.4.15;
2
3  contract Hello {
4
5      string public message;
6
7      function Hello(string initialMessage) public {
8          message = initialMessage;
9      }
10
11     function setMessage(string newMessage) public {
12         message = newMessage;
13     }
14 }

```

*Here's a basic overview of how smart contracts work:*

- **Programming Language:** Smart contracts are typically written using programming languages that are suitable for blockchain platforms, such as Solidity for Ethereum.
- **Agreement Definition:** The parties involved in a contract agree on the terms and conditions, which are then translated into code. This includes specifying the conditions that trigger the contract's execution and the actions to be taken.

- **Deployment:** *The smart contract is deployed to a blockchain network, such as Ethereum, where it becomes publicly available.*
- **Contract Execution:** *Once deployed, the smart contract is now active and can receive inputs or interact with other contracts or users on the blockchain.*
- **Triggering Conditions:** *The smart contract contains predefined conditions that, when met, initiate the contract's execution. These conditions can be time-based (e.g., a specific date), event-based (e.g., receiving a specific amount of cryptocurrency), or a combination of various factors.*
- **Automated Execution:** *When the triggering conditions are met, the smart contract automatically executes the actions programmed into it. These actions may include transferring funds, updating data, or triggering other smart contracts.*
- **Immutability:** *Once a smart contract is deployed and executed, its code and state are recorded on the blockchain, making it tamper-proof and transparent. The contract's execution and outcomes are verifiable by all participants.*

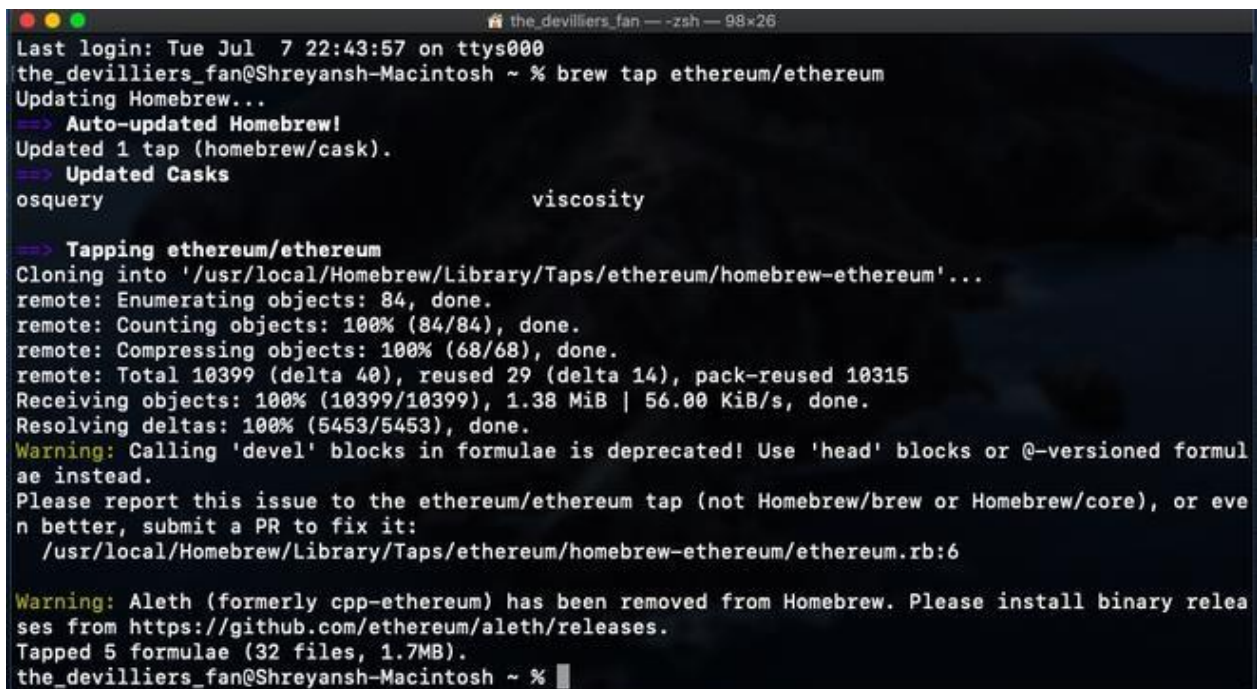


- **Intermediary Elimination:** Smart contracts remove the need for intermediaries, such as lawyers or brokers, as the code itself enforces the contract terms. This reduces costs, enhances efficiency, and increases trust among participants.

It's important to note that smart contracts operate within the constraints of the blockchain network they are deployed on. For example, on Ethereum, the execution of smart contracts requires gas fees to be paid in Ether (ETH), the native cryptocurrency of the Ethereum network.

## **Installation of solidity:-**

### **To install Solidity, you can follow these steps:**



```

the_devilliers_fan — zsh — 98x26
Last login: Tue Jul  7 22:43:57 on ttys000
the_devilliers_fan@Shreyansh-Macintosh ~ % brew tap ethereum/ethereum
Updating Homebrew...
=> Auto-updated Homebrew!
Updated 1 tap (homebrew/cask).
=> Updated Casks
osquery                                viscosity

=> Tapping ethereum/ethereum
Cloning into '/usr/local/Homebrew/Library/Taps/ethereum/homebrew-ethereum'...
remote: Enumerating objects: 84, done.
remote: Counting objects: 100% (84/84), done.
remote: Compressing objects: 100% (68/68), done.
remote: Total 10399 (delta 40), reused 29 (delta 14), pack-reused 10315
Receiving objects: 100% (10399/10399), 1.38 MiB | 56.00 KiB/s, done.
Resolving deltas: 100% (5453/5453), done.
Warning: Calling 'devel' blocks in formulae is deprecated! Use 'head' blocks or @-versioned formulae instead.
Please report this issue to the ethereum/ethereum tap (not Homebrew/brew or Homebrew/core), or even better, submit a PR to fix it:
  /usr/local/Homebrew/Library/Taps/ethereum/homebrew-ethereum/ethereum.rb:6

Warning: Aleth (formerly cpp-ethereum) has been removed from Homebrew. Please install binary releases from https://github.com/ethereum/aleth/releases.
Tapped 5 formulae (32 files, 1.7MB).
the_devilliers_fan@Shreyansh-Macintosh ~ %

```

**Install Node.js:** Solidity is primarily used with Node.js, so you need to have Node.js installed on your system. You can



download the installer for your operating system from the Node.js website (<https://nodejs.org>) and follow the installation instructions.

**Verify Node.js** installation: After installing Node.js, open a command prompt or terminal and run the following command to verify that Node.js is installed correctly:

```
node -v
```

This command should print the version of Node.js installed on your system.

**Install Solidity compiler (solc):** Solidity provides a command-line compiler called solc that you can use to compile Solidity smart contracts. Install the compiler globally on your system by running the following command:

```
npm install -g solc
```

This command installs the solc package from the npm registry and makes it available as a global command.

**Verify Solidity installation:** After installing solc, run the following command to verify that the installation was successful:

```
solc --version
```

*This command should print the version of the Solidity compiler installed on your system.*

*With Solidity and its compiler installed, you are now ready to write, compile, and deploy smart contracts using Solidity. You can use any text editor or integrated development environment (IDE) of your choice to write Solidity code.*

*Additionally, if you prefer using a development framework, you can explore tools like Truffle (<https://www.trufflesuite.com/truffle>) or Hardhat (<https://hardhat.org/>) that provide additional features and utilities for Solidity development. These frameworks include their own Solidity compiler, so you don't necessarily need to install solc separately if you choose to use them.*

*Remember to regularly update your Solidity compiler and development tools to access the latest features, bug fixes, and security patches.*

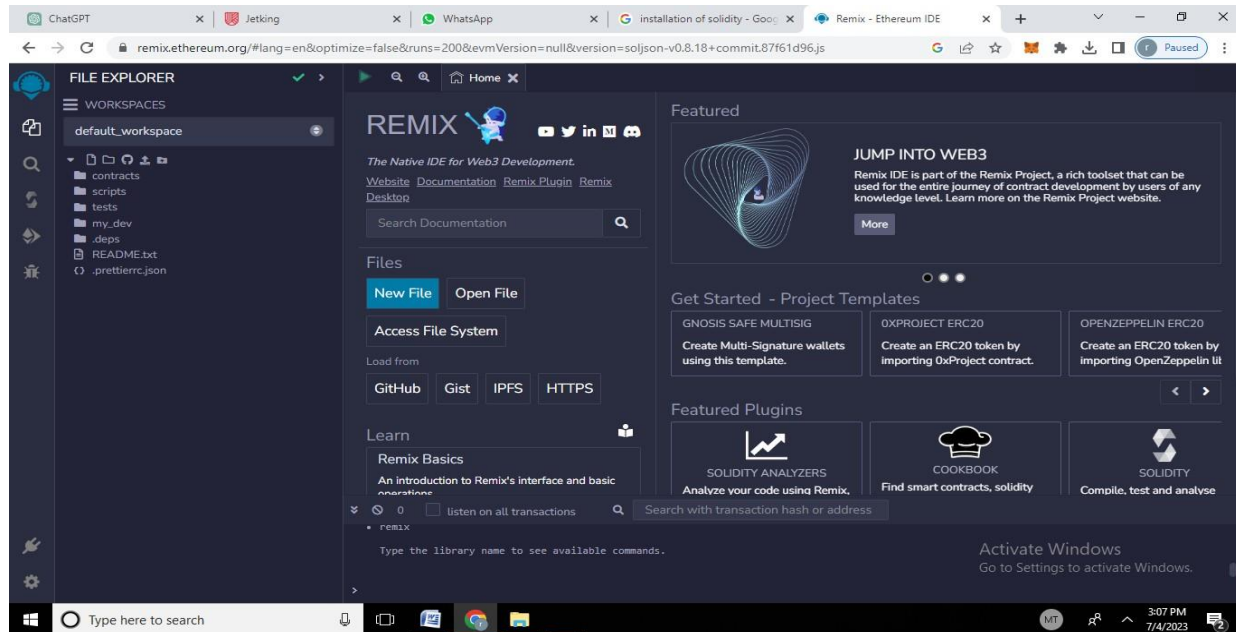
## **Introduction to Remix IDE:-**



- *Remix IDE is a powerful and popular web-based integrated development environment (IDE) specifically designed for smart contract development on the Ethereum blockchain. It provides developers with a comprehensive set of tools and features to write, compile, debug, and deploy their Solidity smart contracts.*
- *Remix IDE offers a user-friendly interface that allows developers to seamlessly interact with their contracts and test them before deploying to the Ethereum network. It provides a convenient way to write and edit Solidity code with features like syntax highlighting, auto-completion, and error checking, which helps catch common mistakes and improve coding efficiency.*

- *One of the key features of Remix IDE is its built-in Solidity compiler. It allows developers to compile their smart contracts directly within the IDE, ensuring that the code is syntactically correct and ready for deployment. The compiler also provides helpful error messages, making it easier to identify and fix any issues in the code.*
- *Remix IDE includes a powerful debugging feature that enables developers to step through their smart contracts line by line and inspect variables and function calls at each step. This helps in identifying and resolving any logical or runtime errors in the contract code.*
- *Another notable feature of Remix IDE is its integration with various Ethereum test networks and the ability to deploy contracts to both public and private networks. This allows developers to test their contracts in different environments and ensure their functionality before going live on the Ethereum mainnet.*
- *Additionally, Remix IDE provides a range of other tools and plugins, such as a built-in code analyzer, security analysis tools, and a unit testing framework, which further enhance the development experience and help in building secure and robust smart contracts.*

- *In summary, Remix IDE is a feature-rich web-based IDE that simplifies the development, testing, and deployment of Ethereum smart contracts. With its intuitive interface and powerful tools, it is widely used by developers to create and manage decentralized applications on the Ethereum blockchain.*



## **Writing your first smart contract :-**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleMessage {
    string public message;

    function setMessage(string memory newMessage) public {
        message = newMessage;
    }
}
```

## **Let's go through the code:**

- The `pragma solidity ^0.8.0;` statement specifies the version of Solidity being used (0.8.0 or higher).
- The contract `SimpleMessage { ... }` declaration defines the smartcontract named "SimpleMessage."
- The `string public message;` creates a public state variable called `message` to store the string message.
- The function `setMessage(string memory newMessage) public { ... }` is a public function that allows users to set the message by passing a string parameter.
- Inside the `setMessage` function, the message variable is updated with the provided message.
- To deploy and interact with this contract, you can use a development environment like Remix or Truffle. Once deployed, you can call the `setMessage` function, passing a new message as a parameter, to update the message variable. You can also access the message variable's value by reading it directly since it's declared as public.
- Remember to compile and test your contract thoroughly before deployment. Additionally, consider adding additional functionalities and error handling as you gain more familiarity with Solidity and smart contract development.

### **Value type in solidity:-**

**Bool:** Represents a boolean value (true or false).



**Integers:** Solidity provides various integer types with different ranges. For example:

**uint:** Unsigned integers (positive integers including zero).

**int:** Signed integers (positive, negative, or zero).

You can specify the number of bits for integers, such as `uint8`, `uint256`, `int32`, etc.

**Address:** Represents an Ethereum address. It holds a 20-byte value (size of an Ethereum address) and provides functions for interacting with addresses.

**Bytes:** Solidity provides two types of byte arrays:

**bytes:** A dynamic array of bytes.

**byte:** A single byte.

These types are useful for handling binary data or interacting with other contracts.

**Strings:** Represents a string of UTF-8 encoded characters. Solidity provides built-in functions for manipulating strings.

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

pragma solidity ^0.8.13;
contract ValueTypes
{
    bool public Myboolean_Variable=true;
    int public MyInteger_Variable=-123;
    uint public MyUint_Variable=123;
    address public MyPublic_Address=0xfC9Dfbb7A367cdd20e8F064c857d18138Eb9f9FC7;
    bytes32 public Myhash = 0xcedb8aa19dcf2ba82a4bbbbbb946792d9804ac19196c7c3cab1ab110a51084aaf;
}
```

## **Functions in solidity:-**

*A Solidity function definition encompasses the complete implementation of a function, including its name, parameters, return types, and the executable code within its body, enabling the execution of specific operations within a smart contract.*

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

pragma solidity ^0.8.13;

contract Functions
{
    function add(uint num1,uint num2) external pure returns (uint)
    {
        return num1+num2;
    }
    function sub(uint num1,uint num2) external pure returns (uint)
    {
        return num1-num2;
    }
    function mul(uint num1,uint num2) external pure returns (uint)
    {
        return num1*num2;
    }
    function div(uint num1,uint num2) external pure returns (uint)
    {
        return num1/num2;
    }
    function mod(uint num1,uint num2) external pure returns (uint)
    {
        return num1%num2;
    }
}
```

## **Variables in solidity:-**

## **State Variables:-**

*State variables are declared within a contract outside of any function. They have persistent storage on the blockchain and retain their values between function calls. State variables are visible to all functions within the contract, and their values can be accessed and modified by these functions. State variables are typically used to store contract data that needs to be persistent.*

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

pragma solidity ^0.8.13;

contract StateVariables
{
    uint public mynumber;

    function setMynumber(uint _mynumber) external
    {
        mynumber = _mynumber;
    }
    function resetMynumber() external
    {
        mynumber=0;
    }
    function getMynumber() external view returns (uint)
    {
        return mynumber;
    }
    function getMynumberplusone() external view returns (uint)
    {
        return mynumber + 1;
    }
}
```

*Local variables:- are declared within functions or code blocks and have a limited scope within that function or block. They are used to store temporary data and are not persisted between function calls. Local variables are typically used for calculations or intermediate storage within a function.*

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

```
pragma solidity ^0.8.13;
```

```
contract LocalVariables
```

```
{
```

```
    uint public firstvariable;
```

```
    bool public secondvariable;
```

```
    address public myAddress;
```

```
    function localvariables() external
```

```
    {
```

```
        uint Mynumber=123;
```

```
        bool Myboolean=false;
```

```
        Mynumber=Mynumber+123;
```

```
        Myboolean=true;
```

```
        firstvariable=123;
```

```
        secondvariable=true;
```

```
        myAddress=address(1);
```

```
    }
```

```
}
```

*Global variable:- State variables declared within a contract are accessible to all functions within that contract and can be considered as global variables within the contract's scope. These variables have a persistent storage location on the blockchain and retain their values between function calls.*

```

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.13;

contract GlobalVariables
{
    function MyglobalVariables() external view returns (address, uint, uint, bytes32)
    {
        address sender = msg.sender;
        // address that called this function
        uint timeStamp = block.timestamp;
        // timestamp (in seconds) of current block
        uint blockNum = block.number;
        // current block number
        bytes32 blockHash = blockhash(block.number);
        // hash of given block
        // here we get the hash of the current block
        // WARNING: only works for 256 recent blocks
        return (sender, timeStamp, blockNum, blockHash);
    }
}

```

## **View and pure function in**

### **solidity:- View Functions:**

*A view function is a function that promises not to modify the state of the contract. It can read the state variables and other contract data but cannot modify them. These functions are free to execute as they don't require any gas. View functions are denoted by the view keyword in their function signature.*

### **Pure Functions:**

*A pure function is a function that doesn't read or modify the state of the contract. It operates solely on the input parameters and local variables. Pure functions are also free to execute and don't require any gas. Pure functions are denoted by the pure keyword in their function signature.*

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

pragma solidity ^0.8.13;

contract ViewandPureFunctions
{
    uint public number1; // This is a State Variable

    function Viewfunction() external view returns(uint) // This is a view function so it can
    read state variable
    {
        return number1;
    }
    function Purefunction() external pure returns(uint) // This is a pure function so it
    cannot read state variable
    {
        return 100;
    }
    function addtoNumber1(uint number2) external view returns(uint)
    {
        return number1 + number2;
    }
    function add(uint num1,uint num2) external pure returns(uint)
    {
        return num1+num2;
    }
}
```

## **Constructor in solidity:-**

*In Solidity, a constructor is a special function that is executed only once during the contract deployment. It is used to initialize the state variables of a contract when it is first created. The constructor has the same name as the contract itself and does not have a return type.*

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

pragma solidity ^0.8.13;

contract ConstructorExample
{
    address public MyownersAddress;
    uint public number;
    string public Myname;

    constructor(string memory _Myname,uint _number)
    {
        MyownersAddress = msg.sender;
        Myname = _Myname;
        number = _number;
    }
}
```



## **Ownable in solidity:-**

*In Solidity, the "ownable" pattern is commonly used to implement ownership functionality in smart contracts. The "ownable" pattern allows for a contract to have an owner who has special privileges or permissions within the contract.*

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

pragma solidity ^0.8.13;

contract Ownable {
    address public owner;

    //Initialize the owner to the account address who deploys this Smart Contract.
    constructor() {
        owner = msg.sender;
    }

    //Restrict the function calls only to the current owner.
    modifier onlyOwner() {
        require(msg.sender == owner, "You are not the Owner of this Smart Contract");
        _;
    }

    //This function can only be called by the current owner only.
    function setOwner(address _newOwner) external onlyOwner {
        //New owner cannot be assigned to the Zero address.
        require(_newOwner != address(0), "The new Owner cannot be assigned to a Zero address");
        owner = _newOwner;
    }
}

//0x0000000000000000000000000000000000000000
```

## **Mapping in Solidity:-**

*In Solidity, the mapping is a data structure used to store key-value pairs. It is similar to a dictionary or associative array in other programming languages. The mapping is defined using the mapping keyword followed by the key and value types.*

```
{
key1[immutable]:value1[mutable],
key2:value2,
key3:value3
}
```

## Smart Contract for understanding Mappings:

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

pragma solidity ^0.8.13;

contract MappingsExample
{
    // Mapping from address to uint used to store balance of addresses
    mapping(address => uint) public balances;

    // Nested mapping from address to address to bool
    // used to store if first address is a friend of second address
    mapping(address => mapping(address => bool)) public isFriend;

    function MappingAddress() external
    {
        // Insert
        balances[msg.sender] = 10000;
        // Read
        uint mybal = balances[msg.sender];
        // Update
        balances[msg.sender] += 20000;
        // Delete
        delete balances[msg.sender];

        // msg.sender is a friend of this contract
        isFriend[msg.sender][address(this)] = true;
    }
}
```

## Enum in solidity:-

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is visible, showing the environment set to 'Remix VM (London)', the account '0x5B3...eddC4 (100 ether)', a gas limit of '3000000', and a value of '0 Wei'. The contract selected is 'StructsAndEnums - contracts/StructsAndEnums.sol'. The main editor displays the following Solidity code:

```
1 //SPDX-License-Identifier: Unlicense
2 pragma solidity 0.8.0;
3 contract StructsAndEnums{
4
5     struct Shipment {
6         string package;
7         string state;
8         address sender;
9         address receiver;
10        uint price;
11        bool delivered;
12    }
13
14    Shipment public purchase = Shipment("Flash Drive", "Texas",
15        0x5B38Da6a701c568545dCfcB03FcB875f56beddC4,
16        0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB, 20, true);
17 }
```

*An enum (short for enumeration) is a data type in programming that represents a set of named values or options. It allows you to define a*

*fixed number of distinct values that a variable can hold. Each value within an enum is assigned an underlying integer value, starting from 0 by default, and incremented by 1 for each subsequent value.*

*Here are some differentiating features and explanations of enum:*

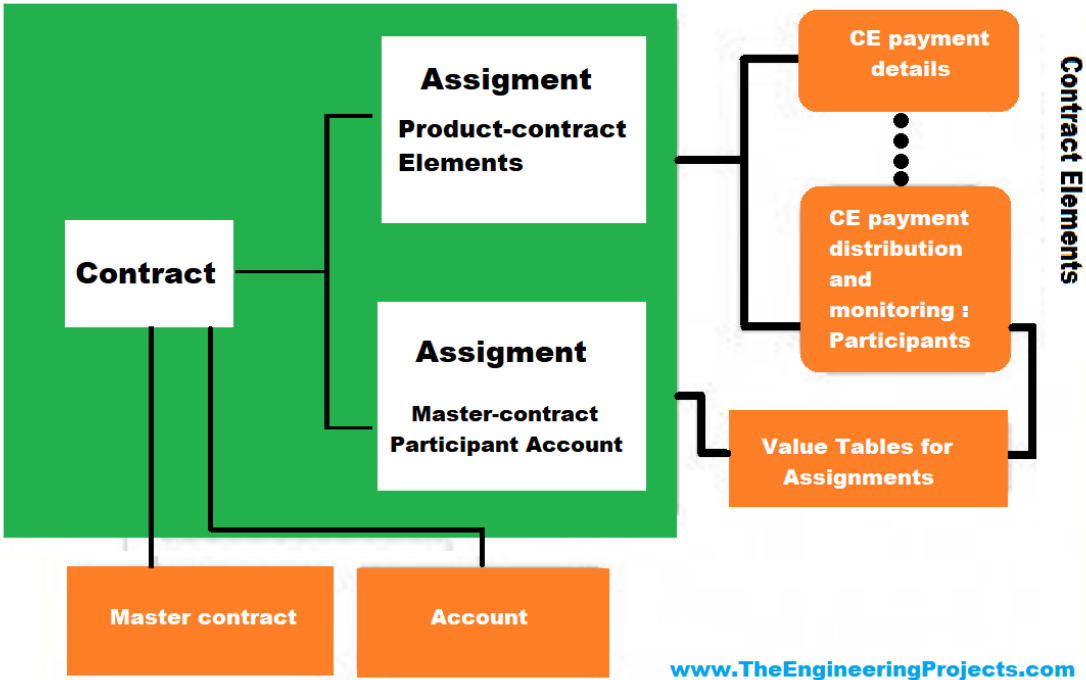
- **Limited Options:** *An enum restricts the possible values of a variable to a predefined set of options. This can be helpful when you want to ensure that a variable only takes on specific values and prevents invalid or unexpected inputs.*
- **Improved Readability:** *By using meaningful names for the values within an enum, you can enhance the readability of your code. Instead of using arbitrary numeric values, you can use descriptive names that make the code more self-explanatory.*
- **Type Safety:** *Enums provide type safety because the variable can only take on one of the predefined values. This reduces the likelihood of errors caused by assigning incorrect values or incompatible types.*
- **Compiler Checks:** *The compiler performs checks to ensure that the assigned value is within the range of the enum options. If you*

*attempt to assign a value that is not part of the enum, the compiler will generate an error.*

- **Semantic Meaning:** *An enum allows you to convey semantic meaning through the named values. For example, if you have an enum called Color, the values Red, Green, and Blue convey more meaning than arbitrary numeric values.*
- **Iteration Support:** *Some programming languages provide built-in support for iterating over the values of an enum. This can be useful when you need to perform operations or validations on all possible values.*

**Structure in solidity:-**

## Structure of a Contract



*In Solidity, the programming language for Ethereum smart contracts, you can define structures using the struct keyword. A structure, also known as a struct, is a user-defined data type that allows you to group together multiple variables of different types.*

## Smart Contract for understanding Struct:

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

pragma solidity ^0.8.13;

contract StructType
{
    struct Car
    {
        string model;
        uint year;
        address owner;
    }

    Car[] public cars;

    function carmodels() external
    {
        // 3 ways to initialize a struct
        Car memory Than = Car("Than",2016,msg.sender);
        // This is the first way to initialize struct

        Car memory BMW = Car(
            {
                model:"BMW",
                year:1995,
                owner:msg.sender
            }
        );
        // This is the second way to initialize struct
        Car memory Honda;
        Honda.model = "Honda";
        Honda.year = 1949;
        Honda.owner = msg.sender;
        // This is the third way to initialize struct
    }
}
```

## Data location in solidity:-



## Data Locations in Solidity



*In Solidity, you can declare and store data in different locations depending on your requirements. Solidity provides several data location keywords that specify where the data should be stored. The data location keywords are as follows:*

**Memory:** *Data stored in the memory location is temporary and exists only for the duration of a function execution. It is suitable for variables that are created and used within a function and do not need to persist between function calls. By default, function parameters and local variables are stored in memory.*

```
function add(uint256 a, uint256 b) public pure returns (uint256) {  
    uint256 result = a + b; // stored in memory  
    return result;  
}
```

**Storage:** *Data stored in the storage location is persistent and is stored on the blockchain. It is suitable for variables that need to persist between function calls or contract invocations. State variables declared outside of any function are stored in storage by default.*

```
contract MyContract {  
    uint256 public myVariable; // stored in storage  
  
    function setMyVariable(uint256 newValue) public {  
        myVariable = newValue;  
    }  
}
```

**Calldata:** *Data stored in the calldata location refers to the input arguments and function parameters passed to a function. Calldata*

*is read-only and cannot be modified within the function. It is suitable for function parameters that are not modified and only used for reading data.*

```
function getValueFromData(uint256[] calldata data) public pure returns (uint256)
    return data[0]; // access calldata
}
```

### **Todo list in solidity:-**

*In Solidity, a "todo list" can refer to a smart contract that allows users to create and manage a list of tasks or notes. Each note typically consists of some content or description and may have additional properties like a timestamp, priority, or status.*

#### **Smart Contract for understanding To Do list:**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract TodoList
{
    struct ToDo
    {
        string username;
        bool completed;
    }

    ToDo[] public Tasks;

    function CreateUser(string calldata _username) external
    {
        Tasks.push(ToDo(
            {
                username: _username,
                completed: false
            }
        ));
    }

    function UpdateUser(uint _index, string memory _username) external
    {
        ToDo storage NewTasks = Tasks[_index];
        NewTasks.username = _username;
    }

    function ToggleCompleted(uint _index) external
    {
        ToDo storage NewTasks = Tasks[_index];
        NewTasks.completed = !NewTasks.completed;
    }

    function getUserTaskInfo(uint _index) external view returns (string memory, bool)
    {

```