

The background of the slide is a blurred image of a computer screen displaying code. The code is written in a light green color on a dark background. Some legible snippets include "data into a dictionary", "like any other Python object", "the magnitude and each event name", "greater than 4", and "properties".

<Solidity>

Programming Language

Getting Started with Solidity?

Solidity is a high-level programming language primarily used for writing smart contracts on the **Ethereum blockchain**.

It is **statically typed** and designed to enable the creation of decentralized applications (DApps) and autonomous programs that run on the **Ethereum Virtual Machine (EVM)**.

Solidity was specifically developed for Ethereum and draws inspiration from various programming languages such as **C++, JavaScript, and Python**.

It provides a way to define the rules and logic of smart contracts, which are **self-executing agreements** with the terms of the agreement directly written into code.

Smart contracts can handle the transfer of digital assets (such as cryptocurrencies) and execute predefined actions based on certain conditions.

What is EVM?



EVM stands for Ethereum Virtual Machine. It is a runtime environment that executes the bytecode of smart contracts on the Ethereum blockchain.

Smart contracts written in languages like Solidity are compiled into bytecode that is understood and executed by the EVM.

When a transaction is sent to the Ethereum network, the EVM processes the transaction, verifies the validity of the smart contract code, and executes the specified operations.

The results of the execution, along with any state changes, are stored on the blockchain.

Why Solidity?

Solidity was specifically designed for Ethereum and is the most widely used programming language for Ethereum smart contracts.

It is well-integrated with the Ethereum ecosystem, providing direct access to Ethereum-specific features like account management, transaction handling, and contract interactions.

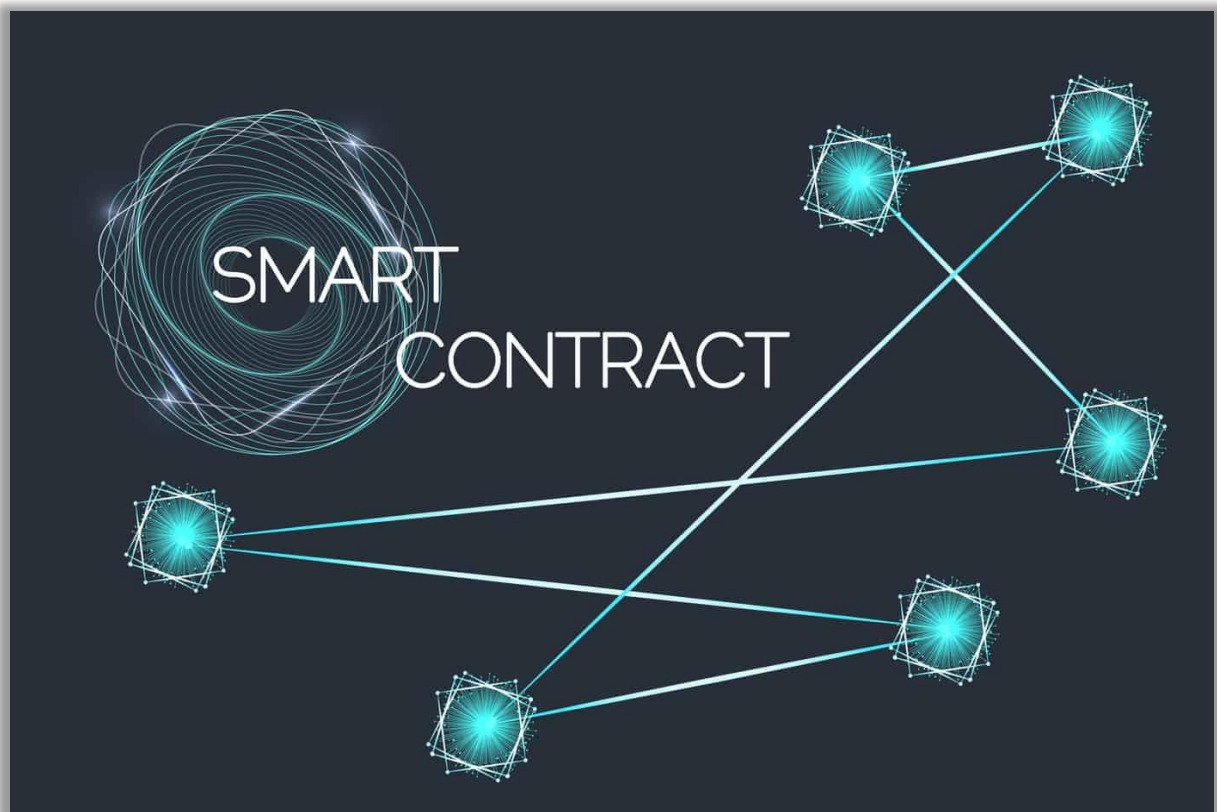
Solidity has been around since 2014 and has matured over the years. It has a large and active developer community, which means there are plenty of resources, libraries, and frameworks available to support Solidity development.

Solidity has a robust ecosystem of tools, development frameworks, and integrated development environments (IDEs) that streamline the development process.

Tools like Truffle, Remix, and Hardhat provide features for testing, debugging, deployment, and interacting with smart

contracts, making development more efficient and developer-friendly.

What are Smart Contracts?



Smart contracts are computer programs that automatically execute predefined actions when certain conditions are met, without the need for intermediaries or human intervention.

Smart contracts are typically deployed on a blockchain, such as Ethereum. They

leverage the decentralized nature of the blockchain to ensure transparency, security, and immutability.

Smart contracts operate on digital assets, such as cryptocurrencies or tokens, and their associated data.

Smart contracts are often the backbone of decentralized applications. They define the business logic and rules of operation for DApps, enabling trust and automation within the application.

How does Smart Contract Work?

Smart contracts work through a combination of blockchain technology, decentralized consensus mechanisms, and code execution.

- Deployment
- Blockchain Execution
- Triggering Events
- Consensus and Validation
- Contract Execution
- Validation and Confirmation
- Immutable Record

what happen if we don't have smart contracts:

If we don't have smart contracts, several aspects of blockchain-based systems and decentralized applications (DApps) would be affected:

- Lack of Automation
- Need for Intermediaries
- Limited Trust and Transparency
- Increased Vulnerability to Fraud
- Complexity in Multi-Party Transactions
- Reduced Efficiency in Decentralized Applications
- Limited Flexibility and Programmability

Introduction to Remix IDE:

While learning Solidity, it is very handy to use an **Integrated development environment(IDE)**.

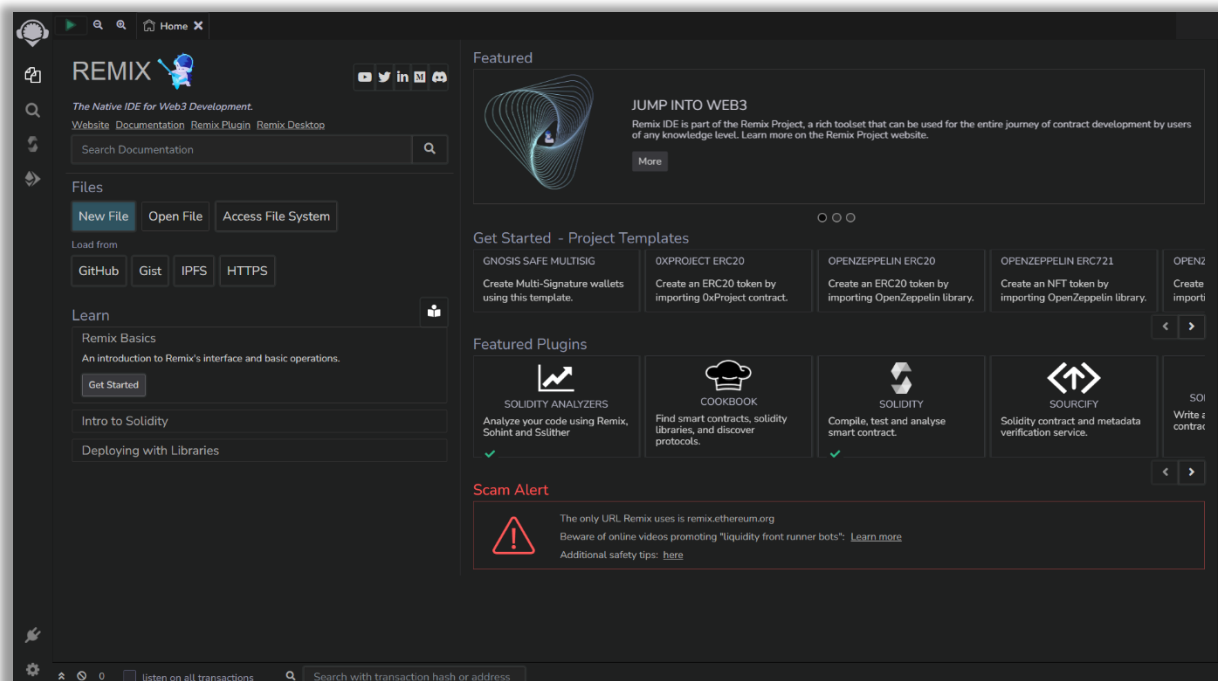
Most of the people use Remix IDE for writing Smart Contract for support testing

and help us debugging our Smart Contracts as well.

Lets start writing our first smart contract on new IDE called "The Remix online Compiler".

It is open-source and written in JavaScript. Remix Ethereum IDE is not the only option available, it is considered to be the best for Solidity learners.

Open the browser of your wish and type, <https://remix.ethereum.org/>



On the left, you have the Icon Panel. In it, you can choose a plugin you wish to see in the Swap Panel.

In this IDE, here are some of the most common plugins:

A Compiler: which compiles a smart contract.

Run & Deploy: Sends a transaction.

Debugger: Debugs a transaction.

Analysis: Presents data of the last compilation.

Main Panel: It allows you to view and edit files in multiple tabs. It highlights Solidity keywords, making it easier to grasp the syntax.

The panel recompiles the code after each change and saves it a few seconds after the last one is finished.

Terminal: You can see the result and run your code in the terminal. It shows all the important operations and transactions mined currently.

You can also search for specific data and clear logs in the terminal.

History of Solidity:

Development Initiation (2014): Solidity was created by Gavin Wood, Christian Reitwiessner, and others as part of the initial development of the Ethereum project, led by Vitalik Buterin.

Its development aimed to provide a high-level language for writing smart contracts on Ethereum.

Initial Release (August 2014): The first version of Solidity, known as "0.1.0," was released alongside the Ethereum Yellow Paper, which described the technical specifications of the Ethereum blockchain.

Evolution and Improvements: Solidity has undergone significant development and improvements over time.

New versions have been released with added features, bug fixes, and security

enhancements, driven by the contributions from the Solidity development team and the wider Ethereum community.

Formal Verification Integration (2015): Solidity added support for formal verification tools, allowing developers to mathematically prove the correctness of their contracts. This integration aimed to enhance the security and reliability of smart contracts.

Language Maturity (2016-2017): Solidity evolved to become more stable and mature as a programming language. It received updates to improve its syntax, semantics, and error handling.

During this period, many developers started adopting Solidity for building decentralized applications (DApps) on the Ethereum blockchain.

Introduction of the Solidity Assembly (2017): Solidity introduced the Assembly

feature, which allowed developers to write low-level, efficient code directly in the Ethereum Virtual Machine (EVM) bytecode. This feature provided developers with more control and optimization opportunities.

Ongoing Development and Updates:

Solidity has continued to evolve with regular updates, addressing security vulnerabilities, adding new features, and improving the overall development experience.

The Solidity development team actively maintains the language and engages with the community to incorporate feedback and suggestions.

Present-Day Usage: Solidity has become the dominant programming language for writing smart contracts on the Ethereum blockchain. It is widely used by developers to create decentralized applications, launch initial coin offerings

(ICOs), build decentralized finance (DeFi) protocols, and more.

First Smart Contract:

TestNet:

The choice of TestNet on which you have to work on, remember to use some specified network ID.

Compiler:

Choosing the compiler as per the requirement, For example, solc is a solidity compiler. It is included in most of the nodes and also available as a standalone package.

Web3.js:

Web3.js is a library. It helps in the connection between the Ethereum network and our App via HTTP or the IPC network connection.

EXAMPLE:



```
//SPDX-License-Identifier:GPL-3.0
pragma solidity ^0.8.18;
contract first_contract{
    string public a;
    int public b;
    uint public c;
    bool public d;
    address public e;
    bytes32 public f;
}
```

Value Types in Solidity:

Value types are used to store and manipulate individual values directly in memory.

Solidity has several built-in value types, including:

Boolean: Represents the values true and false.

Integer types: Signed and unsigned integers of various sizes (e.g., int8, uint256).

Address: Represents Ethereum addresses.

Fixed-size byte arrays: Arrays with a fixed number of bytes.

Enum: User-defined types that represent a set of named values.

Value types are passed by value, meaning that when you assign a value type to a variable or pass it as a function argument, a copy of the value is created.

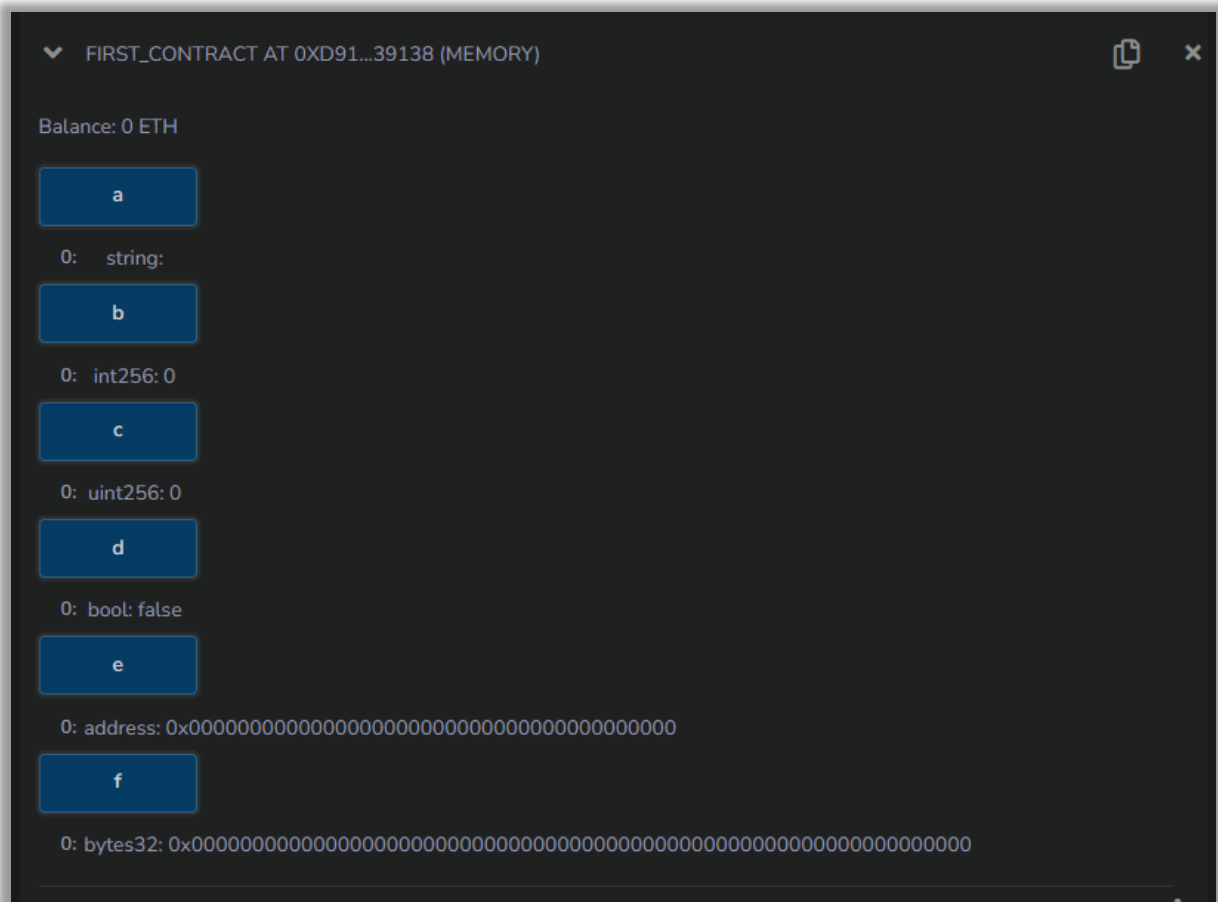
Value types are stored directly in memory and are not assigned to a storage location, such as the blockchain. They are typically used for local variables or function parameters.

Value types have a limited size, which is known at compile-time. This makes them more efficient in terms of gas costs compared to reference types (such as arrays or structs).

Value types have predefined operations and methods that can be performed on them. For example, you can perform arithmetic operations on integers or compare two boolean values.

Value types can be assigned default values. For example, uninitialized boolean

variables have a default value of false, and uninitialized integer variables have a default value of 0.



Function in solidity Programming:

In Solidity, a function is a fundamental building block that allows you to define reusable code blocks with a specific purpose.

Functions in Solidity can be defined within contracts and are used to perform actions and return values.

e.g.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract fun
{
    uint public age= 10;

    //Getter function
    function Get() public view returns(uint) 2459 gas
    {
        return age;
    }

    //Simple Function
    function change(uint _age) public 22520 gas
    {
        age=_age;
    }
}
```

State Variable in Solidity Programming:

In Solidity, state variables are variables that are permanently stored in the contract's storage.

They represent the contract's state and can hold values that persist between function calls and throughout the lifetime of the contract.

State variables are defined at the contract level, outside of any function.

State variables are automatically initialized with default values based on their data type. For example, uninitialized uint256 variables are set to 0, bool variables are set to false, and address variables are set to address(0).

Local Variables in Solidity Programming:

In Solidity, local variables are variables that are declared and used within the scope of a function or a code block.

They are temporary and exist only for the duration of the execution of the function or block.

Local variables can have any of the available data types in Solidity, including bool, uint, int, address, etc.

You can also define local variables of user-defined types, such as structs or enums.

Local variables are only accessible within the block or function where they are declared. They cannot be accessed outside that scope.

```
pragma solidity ^0.5.0;

contract SolidityArticle {
    uint someData;        // State variable
    constructor() public {
        someData = 5;    // Using State variable
    }
}
```

Global Variables in Solidity Programming:

In Solidity, global variables refer to variables that are accessible and can be used throughout the contract. These

variables have a wider scope than local variables and can be accessed from any function within the contract.

Here are some important **global variables** in Solidity:

msg.sender



msg.value

block.timestamp

block.difficulty

block.number

block.coinbase

```
pragma solidity ^0.8.18;
contract GlobalVariables
{
    function MyglobalVariables() external view returns (address, uint, uint, bytes32)  infinite gas
    {
        address sender = msg.sender;
        // address that called this function
        uint timeStamps = block.timestamp;
        // timestamp (in seconds) of current block uint blockNum = block.number;
        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, timeStamps, blockNum, blockHash);
    }
    function ret() external view returns(address){  363 gas
    {
        return msg.sender;
    }
}
```


View and Pure Function in solidity:

In Solidity, the **view** and **pure** function modifiers are used to specify the behavior and guarantees of a function.

These modifiers provide additional information to the compiler and enhance code clarity.

view:


The view modifier is used to indicate that a function **does not modify** the state of the contract.

A view function cannot modify any state variables or emit events within its execution.

It can read the contract's state and other view or pure functions, but it cannot call other functions that are not view or pure.

It **does not consume any gas** when called externally but consumes gas when called internally from a non-view or non-pure function.

e.g.

```
// SPDX-License-Identifier:GPL-3.0
pragma solidity ^0.8.18;
contract start{
    uint public a=5; //state variable
    function happy(uint b) public view returns(uint)  infinite gas
    {
        return a+b;
    }
}
```


pure:

The pure modifier is used to indicate that a function **does not read or modify** the state of the contract.

A pure function is restricted to performing computations and returning a value based solely on its inputs.

It cannot read or modify any state variables or call any other functions, except for other pure functions or view functions.

It does not consume any gas when called externally or internally.

```
// SPDX-License-Identifier:GPL-3.0
pragma solidity ^0.8.18;
contract start{
    uint public a=5;
    function sad(uint c) public pure returns(uint)  infinite gas
    {
        return c+5;
    }
}
```

Counter Contract in Solidity Programming:

The contract is named Counter and is defined using the **contract keyword**.



The state variable count is declared as uint private, which means it can only be accessed within the contract.

The constructor is a special function that is executed once when the contract is deployed. In this case, it sets the initial value of count to 0.

The **getCount()** function is a view function that returns the current value of count without modifying the contract state.

The **increment()** function increments the value of count by 1.

The **decrement()** function decrements the value of count by 1.

```
//SPDX-License-Identifier:GPL-3.0
pragma solidity ^0.8.18;
contract Second{
    uint public a=1;
    function increment() external {  infinite gas
        a+=1;
    }
    function decrement() external  infinite gas
    {
        a-=1;
    }
}
```

Default Values in Solidity Programming:

In Solidity, you can assign default values to variables, both for state variables and function parameters.

Type	Default value	Example
int and uint (all sizes)	0	int32 a; //0
bool	false	bool flag; //false
bytes1 to bytes32	All bytes set to 0	bytes4 byteArray; // 0x00000000
Static array	All items set to zero value	bool [3] flags; // [false, false, false]
bytes	Empty byte array	[]
Dynamic array	Empty array	int [] values; // []
string	Empty string	""
struct	Each element set to the default value	

```
//SPDX-License-Identifier:GPL-3.0
pragma solidity ^0.8.18;
contract first_contract{
    string public a="Tushar";
    int public b=10;
    uint public c=22;
    bool public d=true;
    address public e=msg.sender;
    bytes32 public f="ab";
}
```

Constants in Solidity Programming:

In Solidity, constants are used to define variables whose values cannot be changed once they are assigned.

They are typically used for storing values that are known and will not change during the execution of the contract.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract counter{
    address public constant myAddress=0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;
    string public constant myName="Tushar";
}
```

If and Else Statement in Solidity:

In Solidity, if and else statements are used for conditional execution of code based on certain conditions.

They allow you to control the flow of execution within a contract based on the evaluation of boolean expressions.


```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.18;
contract vote{
    function enter_age(uint age) public pure returns(string memory) {
        if(age>=18)
        {
            return "You are Eligible for Vote!!!";
        }
        else if(age<=17)
        {
            return "You are NOT Eligible for Vote!!!";
        }
        else
        {
            return "Enter valil input";
        }
    }
}
```

Iterative Statement in Solidity:

Solidity provides array and mapping data structures that you can use to iterate over elements.

You can use a for loop to iterate over array elements or key-value pairs in a mapping.

It's important to note that gas costs and other limitations should be considered when designing iterations in Solidity.

Excessive iterations or complex computations can lead to high gas costs

and potentially cause transactions to fail due to gas limits.

For loop:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract check_for{
    function for_loof() public pure returns(uint)  infinite gas
    {
        uint count=0;
        for(uint i=1;i<=10;i++)
        {
            count++;
        }
        return count;
    }
}
```

While loop:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract checking_loop{
    function while_check() public pure returns(uint){  infinite gas
        uint i=1;
        uint count=0;
        while(i<=10)
        {
            i++;
            count++;
        }
        return count;
    }
}
```

ERROR Handling in Solidity:

Error handling in Solidity can be done using **require**, **assert**, **revert** statements.

<Require>

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mycontract{
    function myreq(uint a, uint b) public pure returns(uint){
        require(b>0,"The denominator should be greater than 0");
        return a/b;
    }
}
```

<Revert>

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mycontract{
    function myfun(uint a, uint b) public pure returns(string memory){
        uint sum=a+b;
        if(sum>10 || sum>20)
        {
            revert("Revert is running");
        }
        else {
            return("Revert is not running");
        }
    }
}
```

<Assert>

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mycontract{
    bool result;
    function check_assert(uint a, uint b) public{    infinite gas
        uint sum=a+b;
        assert(sum>=120);
        result=true;
    }
    function checking() public view returns(string memory)    infinite gas
    {
        if(result==true)
        {
            return "The number is greater than or equals to 120";
        }
        else{
            return "Number is less than 120";
        }
    }
}
```

Function Modifier in Solidity:

In Solidity, function modifiers are used to add additional behaviour or restrictions to functions. Modifiers allow you to reuse code and apply it to multiple functions within a contract.

They can be used to enforce access control, validate inputs, modify the behaviour of functions, or perform other checks or operations.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mycontract{
    uint public sum;
    modifier mymod(uint a)
    {
        sum=sum+a;
        _;
    }
    function myfun(uint x) public mymod(x) infinite gas
    {

    }
    function myfun2(uint y) public mymod(y) infinite gas
    {

    }
}
```

Constructor in Solidity Programming:

In Solidity, the constructor is a special function that is executed only once during the contract's deployment. It is used to initialize the state variables of the contract and perform any necessary setup operations.

```
1 //SPDX-License-Identifier:GPL-3.0
2 pragma solidity ^0.8.18;
3 contract First{
4     uint public a;
5     constructor (){ 63607 gas 41400 gas
6         a=100;
7     }
8     function age() public{ 22258 gas
9         a=22;
10    }
11 }
```

Ownable in Solidity Programming:

In Solidity programming, the Ownable contract is a commonly used contract template that provides basic access control functionalities.

It is often used as a base contract for other contracts to enforce ownership and restrict access to certain functions.


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mycontract{
    address owner;
    constructor() 76772 gas 52400 gas
    {
        owner=0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
    }
    function check() public view returns(bool) 2474 gas
    {
        return msg.sender==owner;
    }
}
```

Arrays in Solidity Programming:

Arrays are a fundamental data structure in Solidity that allow you to store and manipulate collections of elements of the same type. Solidity supports both fixed-size arrays and dynamic arrays.

Here's an overview of arrays in Solidity programming:

Fixed-Size Arrays:

- A fixed-size array has a predetermined length that cannot be changed after declaration.
- Elements in a fixed-size array are stored sequentially in memory.
- The length of a fixed-size array is part of its type.
- Elements in a fixed-size array are initialized to their default values upon contract deployment.

Example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract Types {
    uint[6] data1;

    function fixed_array() public returns (    infinite gas
    int[5] memory, uint[6] memory){

        int[5] memory data= [int(50), -63, 77, -28, 90];
        data1= [uint(10), 20, 30, 40, 50, 60];

        return (data, data1);
    }
}
```


Dynamic Arrays:

- A dynamic array has a variable length that can be changed during runtime.
- Elements in a dynamic array are stored in a separate memory area, and the

array itself stores a reference to that memory area.

- Dynamic arrays can grow or shrink in size using the push and pop functions or by directly assigning new values.

Example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.18;
contract my_arr{
    uint []public arr=[10,7,43,5,35,0];
    //for change the index value of array
    function set(uint index, uint value) public{  infinite gas
        arr[index]=value;
    }
}
```

Mapping in Solidity Programming:

In Solidity programming, a mapping is a key-value data structure used to store and retrieve data efficiently.

It is similar to a hash table or a dictionary in other programming languages.

A mapping is declared using the **mapping keyword**, followed by the key and value types enclosed in parentheses.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mymap{
    mapping(uint id =>string) public map;
    function input(uint id, string memory name) public infinite gas
    {
        map[id]=name;
    }
}
```

Enum in Solidity Programming:

In Solidity programming, the **enum keyword** is used to define user-defined types that represent a set of named values.

Enums are used to create a finite list of options or states that a variable can take.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract banking{
    enum Status{
        debit_card,
        credit_card,
        silver_card,
        gold_card,
        taitanium_card,
        platinum_card
    }
    Status public mystatus;
    function debit_card() public{ 24475 gas
        mystatus=Status.debit_card;
    }
    function credit_card() public{ 24432 gas
        mystatus=Status.credit_card;
    }
    function silver_card() public{ 24431 gas
        mystatus=Status.silver_card;
    }
    function gold_card() public{ 24476 gas
        mystatus=Status.gold_card;
    }
    function taitanium_card() public{ 24453 gas
        mystatus=Status.taitanium_card;
    }
    function platinum_card() public{ 24454 gas
        mystatus=Status.platinum_card;
    }
}
```

Struct in Solidity Programming:

In Solidity programming, structs are user-defined data structures that allow you to define custom composite types.

Structs allow you to group together multiple variables of different types under a single name.

Structs can be used to define more complex data structures, represent entities in the application domain, or group related data together.

They can also be used as array elements, mapping values, or as return types for

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.18;
contract mtstu{
    struct Book{
        string title;
        string author;
        uint price;
        bool available;
    }
    Book b1;
    Book b2;
    Book b3;
    function set_value() public{ Infinite gas
        b1=Book("Life_of_Soul","Sheetal",190,false);
        b2=Book("Good Things Takes Time","Tushar",150,true);
        b3=Book("Life_of_Programmer","Unknown",2000,true);
    }
    function get_value() public view returns(string memory, string memory, uint, bool, string memory, string memory, uint, bool, string memory, string memory, uint, bool)
    {
        return(b1.author, b1.title, b1.price, b1.available, b2.author, b2.title, b2.price, b2.available, b3.author, b3.title, b3.price, b3.available);
    }
}
```

functions.

Data Location in Solidity Programming:

In Solidity programming, data location refers to where a variable's data is stored:

in memory, in storage, or as a function parameter.

Understanding data location is crucial for optimizing gas usage and avoiding unintended behaviour.

Memory (memory):

- Memory is used to store temporary data within a function's execution context.
- Variables declared inside functions are typically stored in memory.
- Function parameters are also stored in memory by default.
- Memory data is non-persistent and is cleared after the function finishes executing.

Storage (storage):

- Storage is used to store permanent data that persists between function calls and contract deployments.
- State variables declared outside functions are stored in storage.
- Data stored in storage remains intact across function calls and contract executions.

Calldata (calldata):

- Calldata is a special data location used to store function arguments when a contract is called from outside.
- Calldata is read-only, meaning the data cannot be modified within the contract.
- Function parameters prefixed with calldata are stored in calldata, and they are automatically handled by the EVM.

<Storage Vs Memory>

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mycontract{
    string [] public myarr=["Sachin","Tushar","Mohit","Uttkarsh","Mahima"];
    function mystorage(uint i, string memory v) public infinite gas
    {
        string [] storage z=myarr;
        z[i]=v;
    }
    function mymemory(uint i, string memory v) public view{ infinite gas
        string [] memory x=myarr;
        x[i]=v;
    }
}
```


<Memory Vs calldata>

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract mycontract{
    function mymemory(string memory a)public pure returns(string memory)
    {
        a="Tushar";
        return a;
    }
    function mycalldata(string memory b) public pure returns(string calldata)
    {
        return b;
    }
}
```