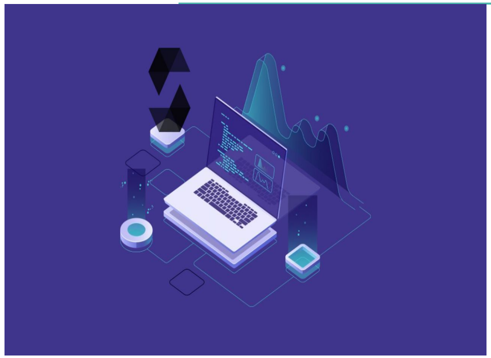


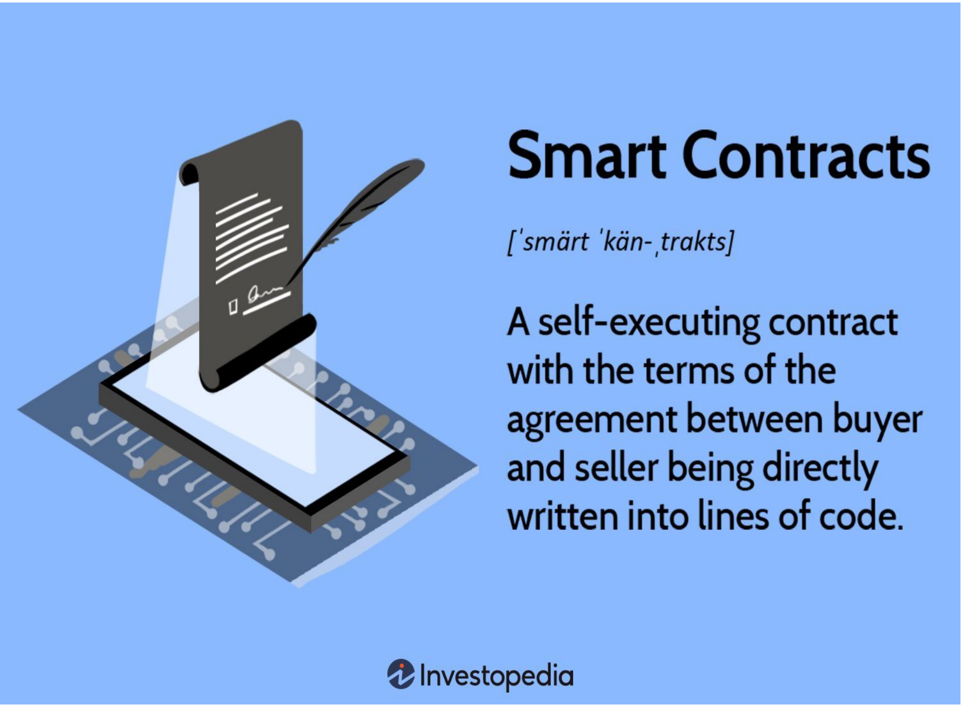
Solidity

The Language of Smart Contract

Dayanand Ambawade

Courtesy: Google Images

Smart Contracts

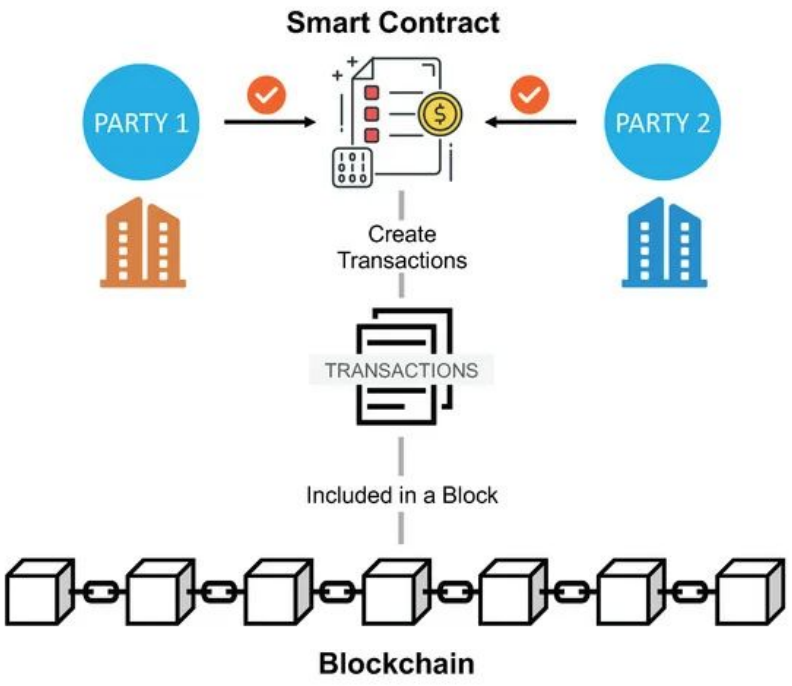


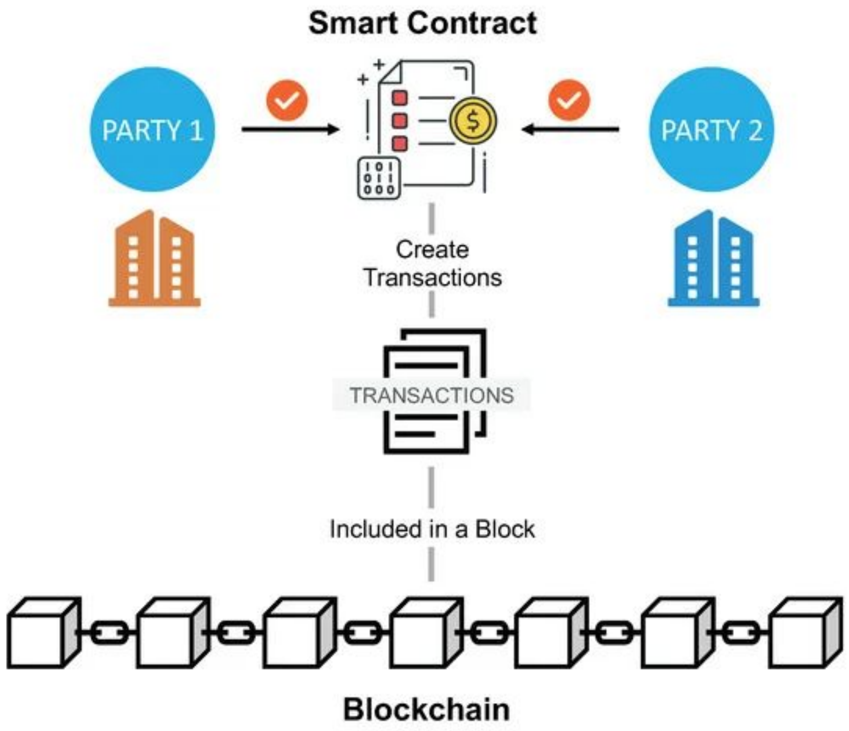
Smart Contracts

['smärt 'kän-ˌtrakts]

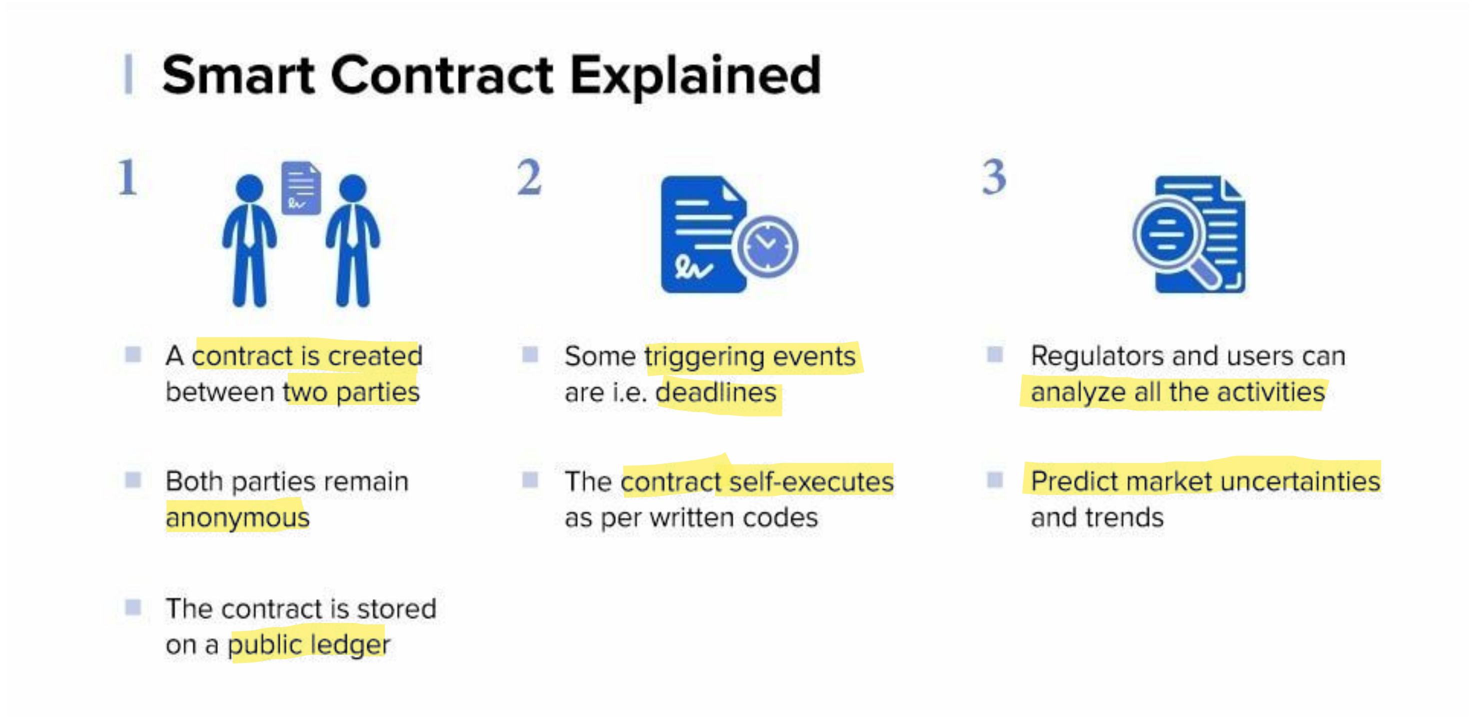
A self-executing contract with the terms of the agreement between buyer and seller being directly written into lines of code.

Investopedia




Smart Contract
PARTY 1
PARTY 2
Create Transactions
TRANSACTIONS
Included in a Block
Blockchain

Working of smart contract



Smart Contract Explained

1

- A contract is created between two parties
- Both parties remain anonymous
- The contract is stored on a public ledger

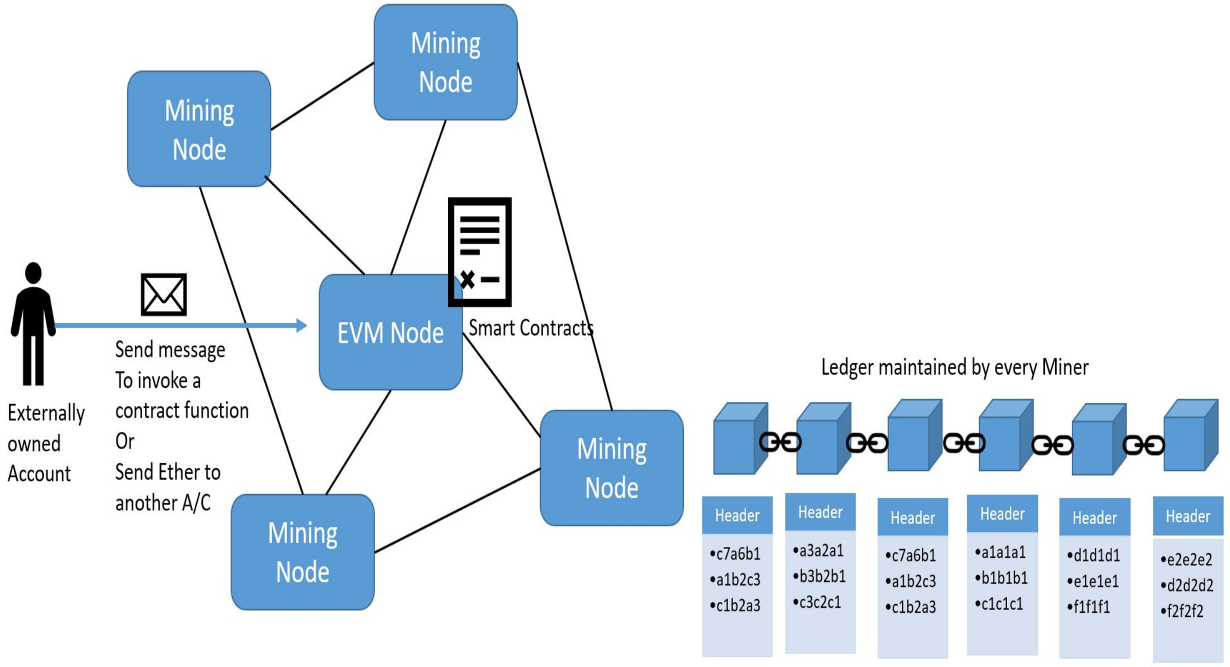
2

- Some triggering events are i.e. deadlines
- The contract self-executes as per written codes

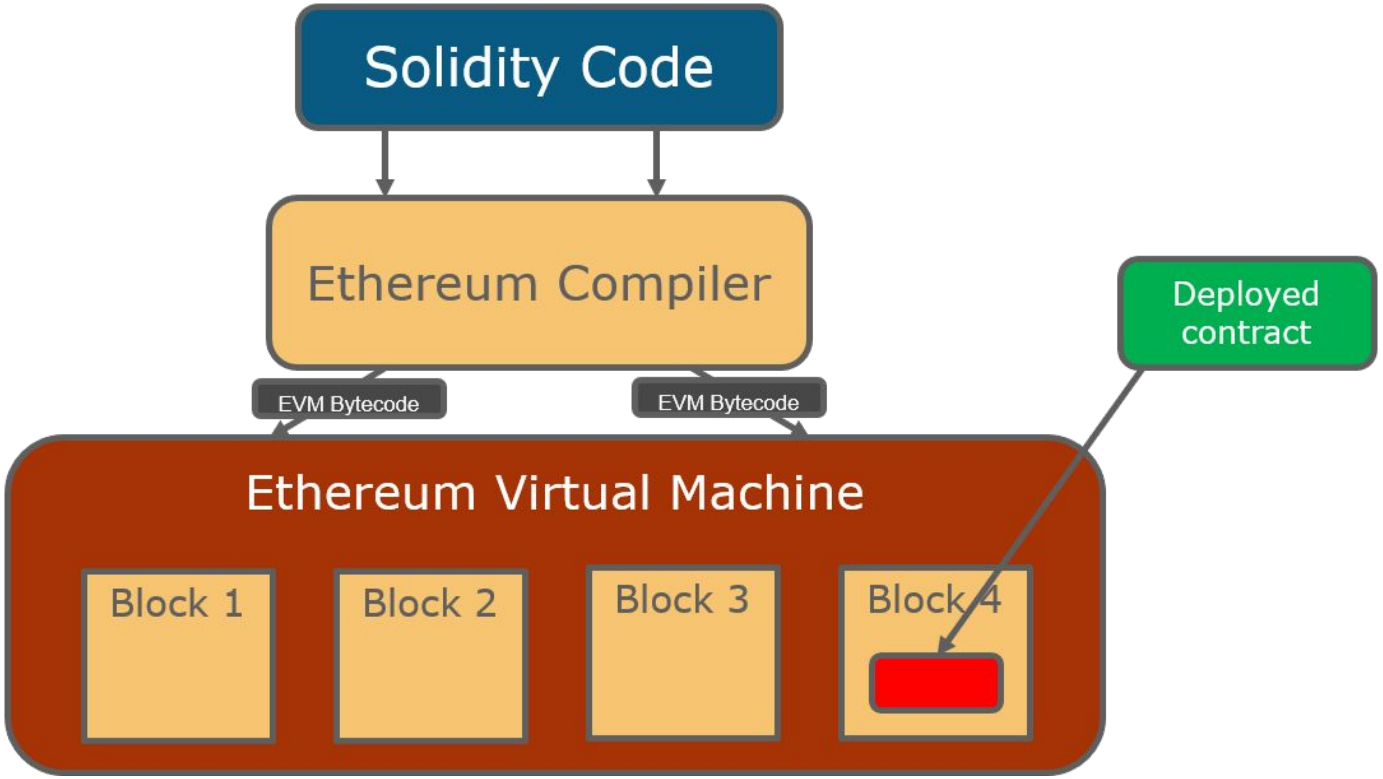
3

- Regulators and users can analyze all the activities
- Predict market uncertainties and trends

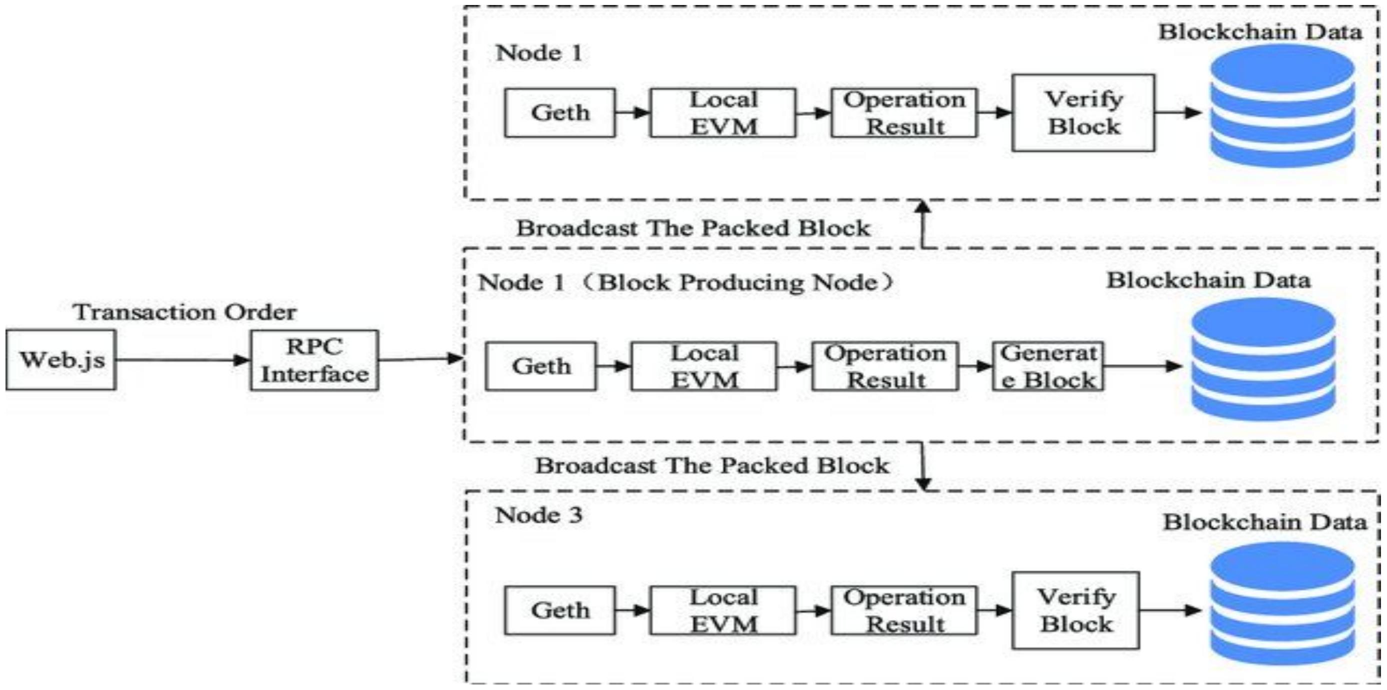
Smart contract and blockchain

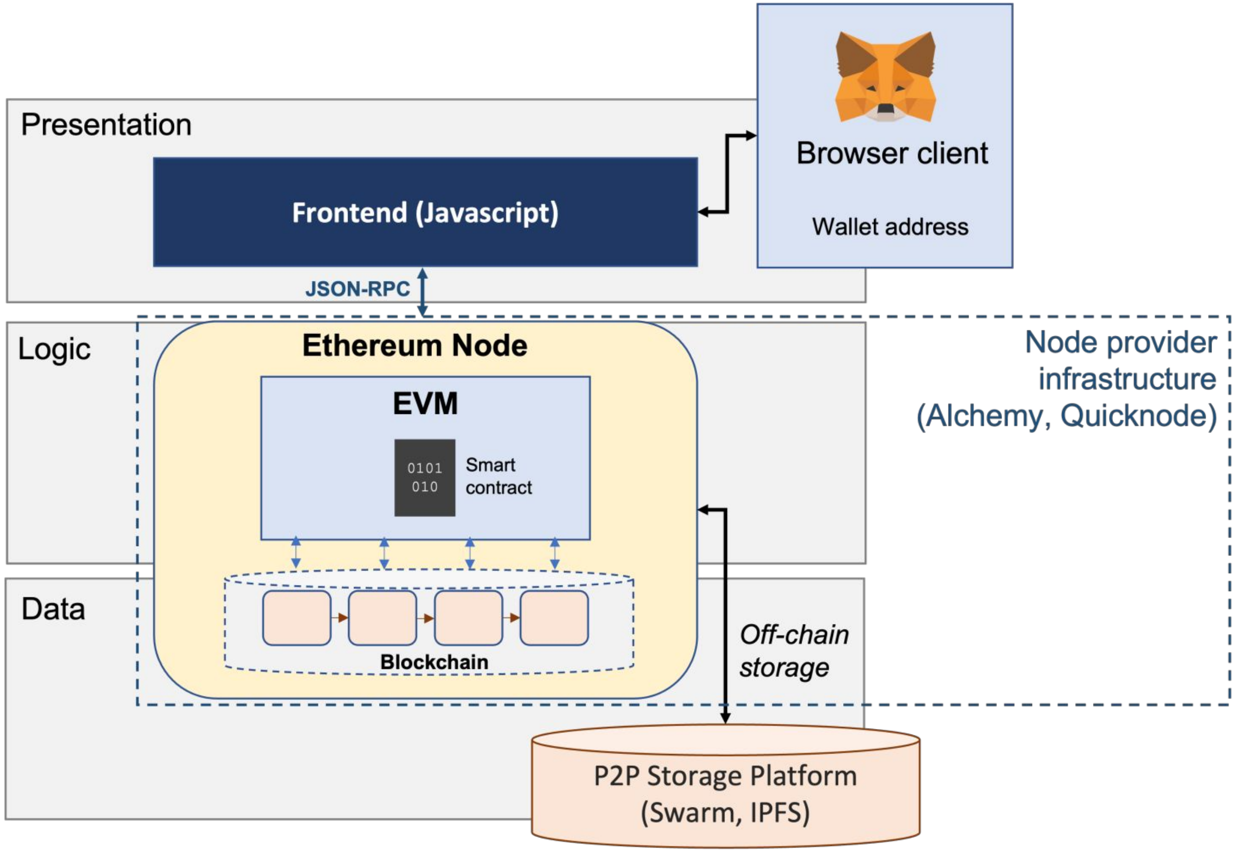


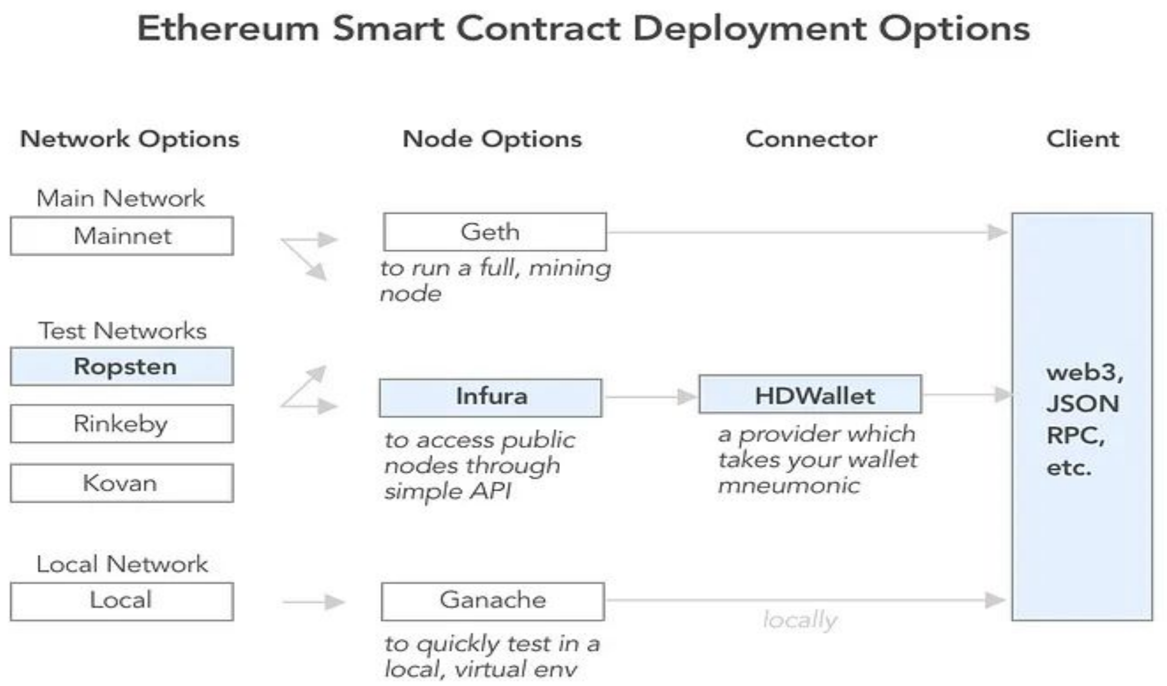
Smart contract and EVM



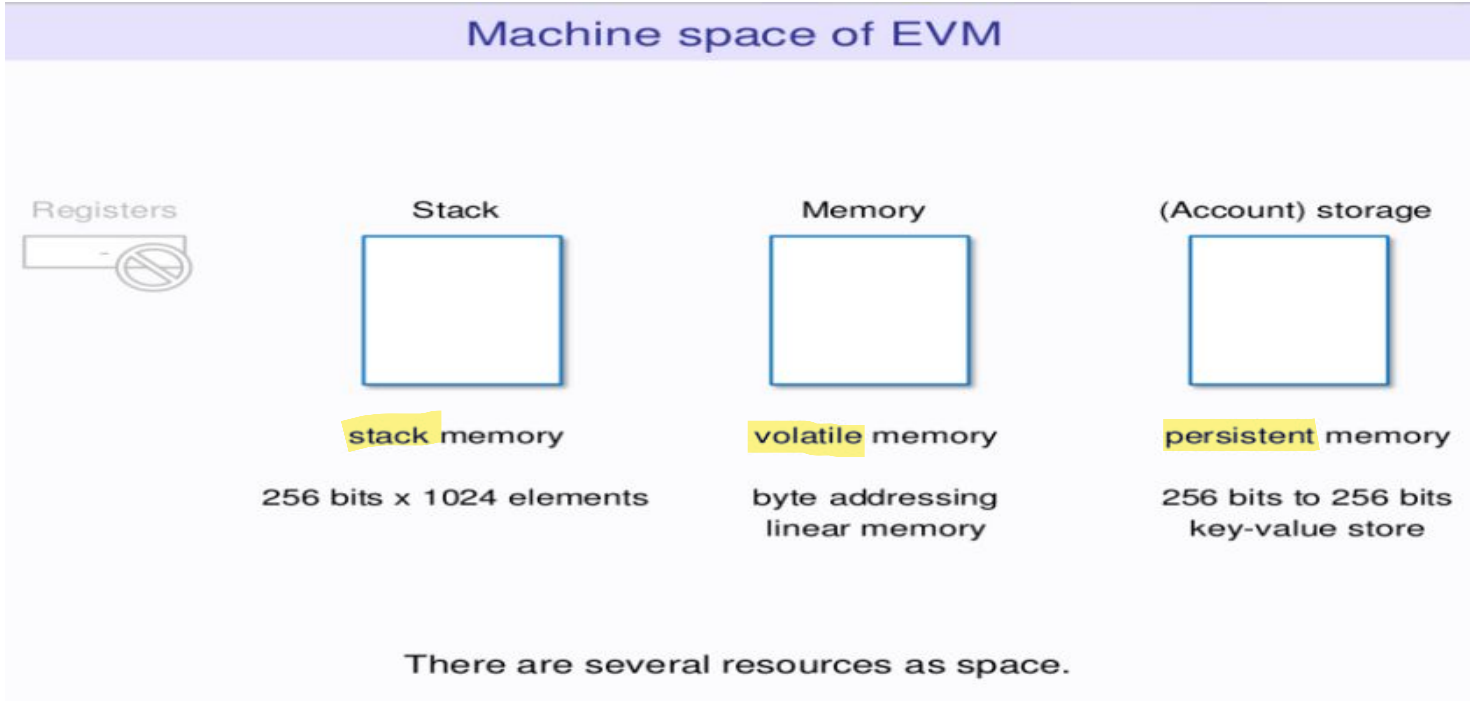
Blockchain P2P nodes




Presentation
Frontend (Javascript)
Browser client
Wallet address
JSON-RPC
Logic
Ethereum Node
EVM
0101
010
Smart contract
Node provider infrastructure (Alchemy, Quicknode)
Data
Blockchain
Off-chain storage
P2P Storage Platform (Swarm, IPFS)


Ethereum Smart Contract Deployment Options
Network Options
Node Options
Connector
Client
Main Network
Mainnet
Geth
to run a full, mining node
Test Networks
Ropsten
Rinkeby
Kovan
Infura
to access public nodes through simple API
HDWallet
a provider which takes your wallet mneumonic
web3, JSON RPC, etc.
Local Network
Local
Ganache
to quickly test in a local, virtual env
locally

Types of Memory in Solidity



STORAGE

- Storage is a location that stores state variables that exists permanently in the contract.
- Storage is a key-value store that maps 256-bit words to 256-bit words. Storage data is costly(consuming much gas) because it is written in blockchain(like HDD/SDD). Therefore,
- It should be used when it is indispensable such as a money balance.
- Data in storage is written in the blockchain (hence they change the state.)are available between calls and transaction that’s why are expensive.

MEMORY

Memory is a byte-array data that exists only during the function call. Every message call starts with a cleared memory.

- Memory is a volatile data and not recorded in the blockchain, so it is much cheaper than storage.
- Memory can be used for every non-permanent data.

Introduction to Solidity

- Solidity is an ***object-oriented, high-level language*** for implementing smart contracts.
- Smart contracts are programs which govern the behavior of accounts within the Ethereum state..
- Solidity is highly influenced by Python, c++, and JavaScript which runs on the ***Ethereum Virtual Machine(EVM).***
- Solidity supports complex ***user-defined programming, libraries and inherita***
- Solidity is primary language for blockchains running platforms.
- Solidity can be used to create contracts like ***voting, blind auctions, crowdfur multi-signature wallets, etc***.

Solidity file extension

A solidity file is created with a file name along with an extension “.sol” .

• You can create and edit solidity contracts in any editor.

• For example, ***HelloWorld.sol*** is the file name of smart contract where Hello World is the name and .sol is the extension.

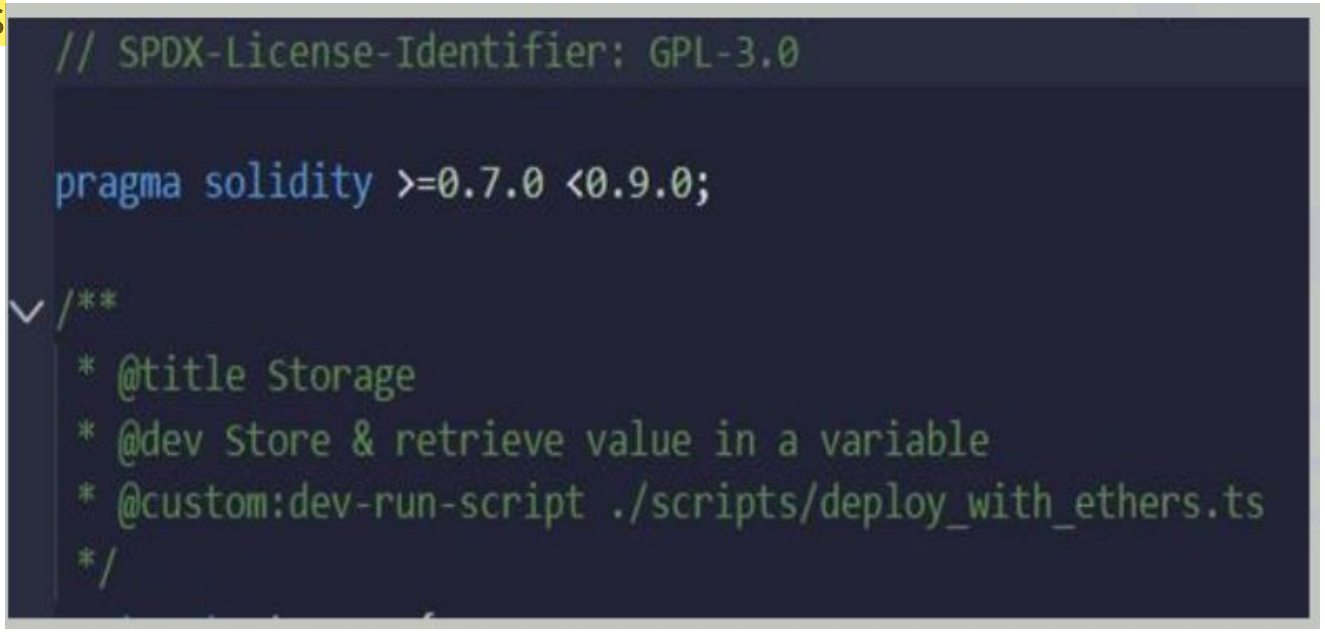
Layout of solidity source code

Every .sol file or smart contract file has the following main components:

1. ***Pragma statements***
2. ***Comments***
3. ***Import statements***
4. ***Contracts***

Pragma Statement

The pragma keyword can be used to enable certain compiler features or checks

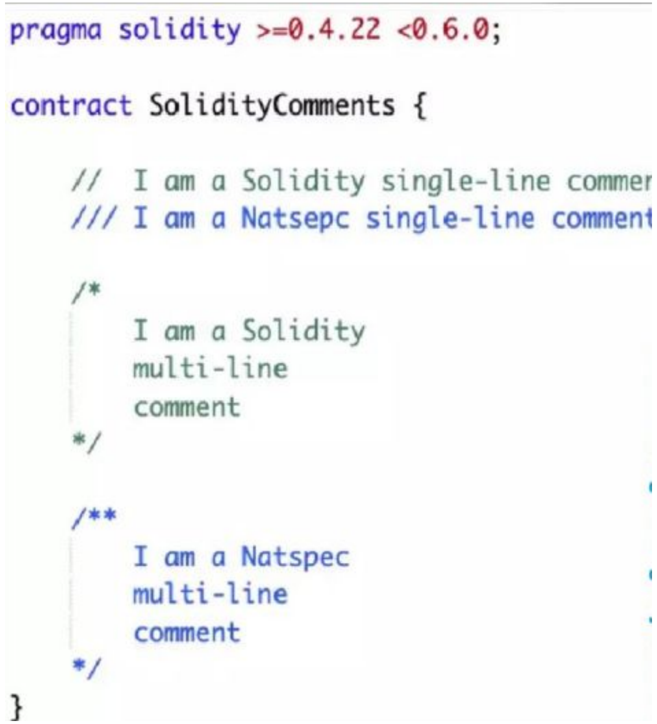


```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 * @custom:dev-run-script ./scripts/deploy_with_ethers.ts
 */
```

Comments in solidity

There are two types of comment in solidity:

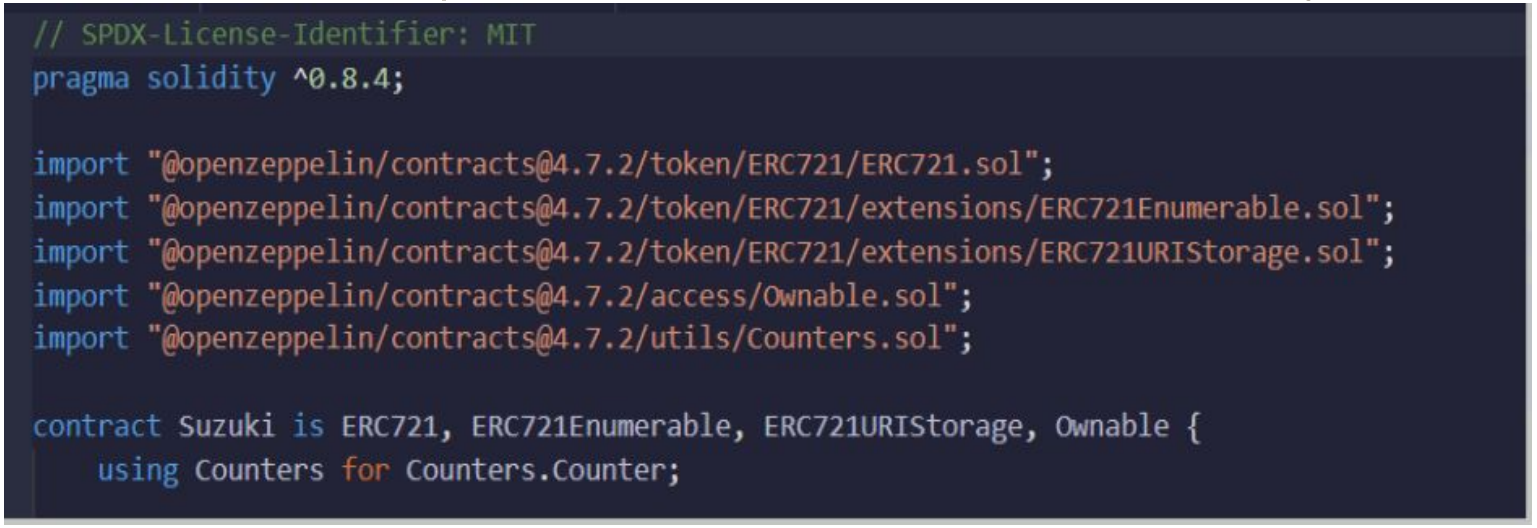
- Single-Line Comments.
- Multi-Line Comments.



```
pragma solidity >=0.4.22 <0.6.0;

contract SolidityComments {

    //  I am a Solidity single-line commer
    /// I am a Natsepc single-line comment

    /*
        I am a Solidity
        multi-line
        comment
    */

    /**
        I am a Natspec
        multi-line
        comment
    */
}
```

Import in solidity

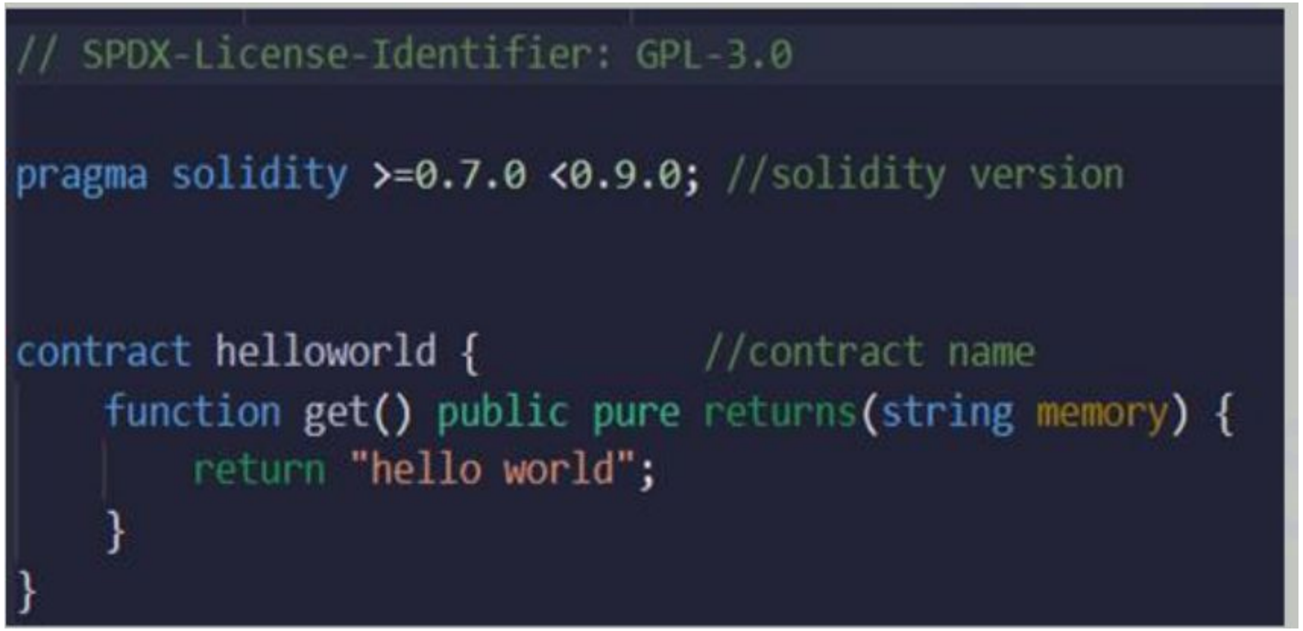
- The import keyword helps in importing other solidity files.
- These files can then be used in smart contract code
- The functions of the imported file can then be used in a solidity file.



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts@4.7.2/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts@4.7.2/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts@4.7.2/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts@4.7.2/access/Ownable.sol";
import "@openzeppelin/contracts@4.7.2/utils/Counters.sol";

contract Suzuki is ERC721, ERC721Enumerable, ERC721URIStorage, Ownable {
    using Counters for Counters.Counter;
```

Contract in solidity

Sample hello world smart contract.



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

pragma solidity >=0.7.0 <0.9.0; //solidity version


contract helloworld {             //contract name
    function get() public pure returns(string memory) {
        return "hello world";
    }
}
```

State transition in Ethereum process

A transaction on Ethereum is executed in the following manner:

- ❖ First, the upfront cost of execution is deducted from the sender’s balance.
- ❖ Then, the nonce of the sender’s account is increased by 1 to account for the current transaction.
- ❖ Next, the transaction starts executing. Throughout the execution, Ethereum keeps track of the 'substate', which is a way to record information accrued during the transaction that will be needed immediately after the transaction is complete. Specifically, it contains:
    - ○ Self-destruct set: a set of accounts (if any) that will be discarded after the transaction completes
    - ○ Log series: archived and indexable checkpoints of the virtual machine’s code execution
    - ○ Refund balance: the amount to be refunded to the sender's account after the transaction
- ❖ The various computations required by the transaction are processed.
- ❖ Once all the steps required by the transaction have been processed and assuming there is no invalid state, the state is finalised by determining the amount of unused gas to be refunded to the sender.
- ❖ The sender is also refunded some allowance from the 'refund balance'.

Gas,Gas Price And Gas Limit ?

ETHER and its applications

In Ethereum, ether can be used for the following things:

- ***Payments***
- ***Powering decentralized applications***
- ***Transactions fees***

How is Gas Calculated ?

- There are two elements that impact the cost of any given transaction:

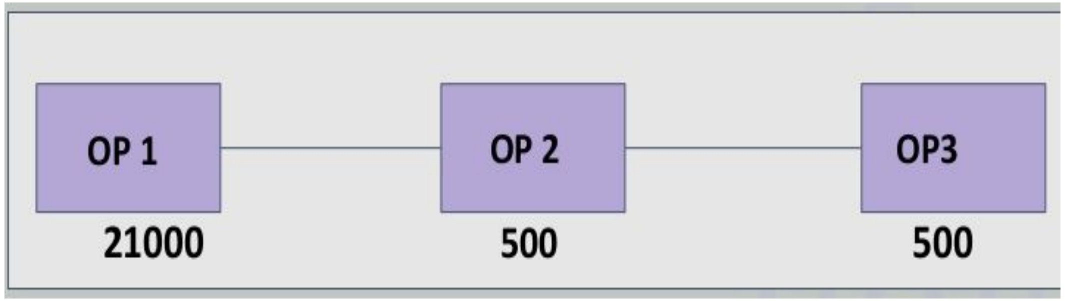
➢ ***Gas price*** at the time of transaction
➢ **Gas required** for a particular transaction.

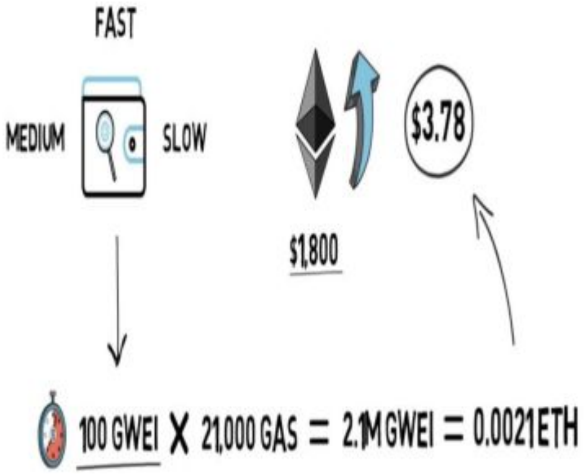
- Each action costs an amount of gas based on the computational power required and how long it takes to run.

- It is expressed as, ***Gas unit (limits) * (Base fee + Tip)***.

How is Gas Calculated ?

•



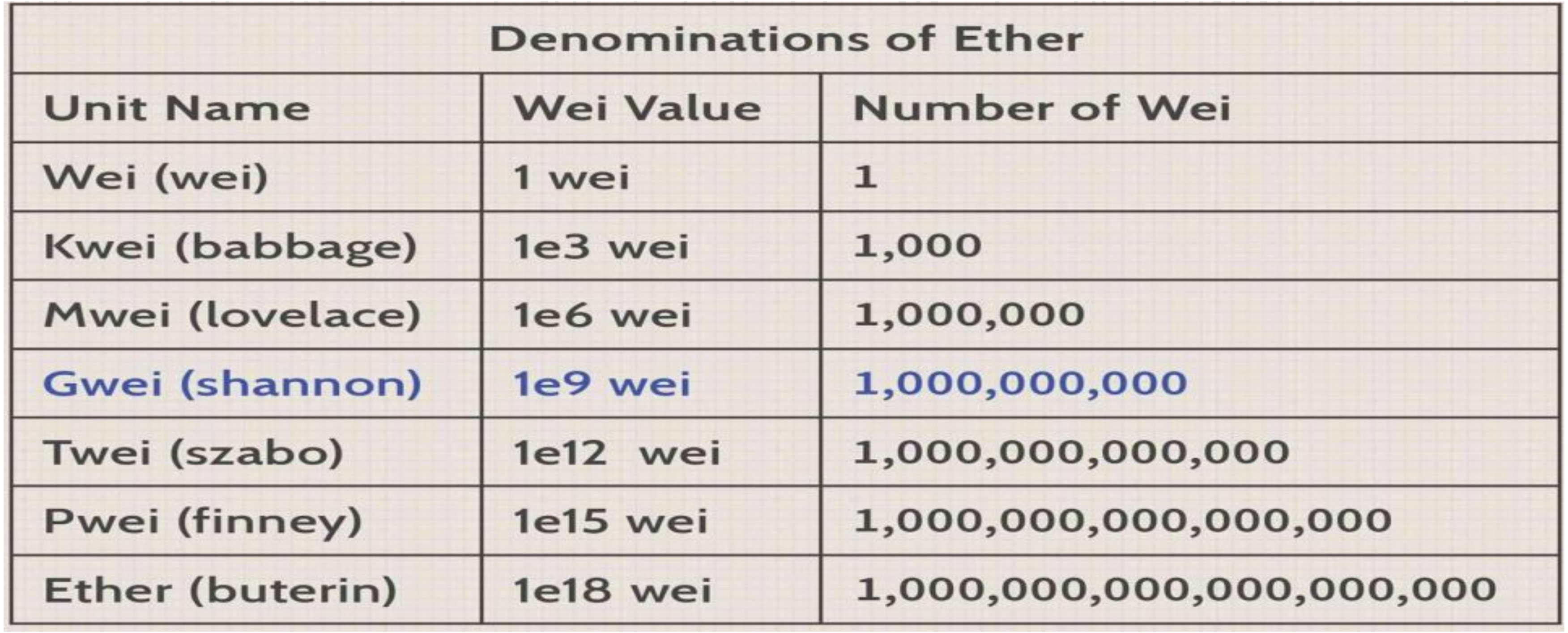


Fixed Cost = 21000

Variable Cost =500

Total = 21000+500+500 = 22000

Different units of Ether Gas



Denominations of Ether		
Unit Name	Wei Value	Number of Wei
Wei (wei)	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
Twei (szabo)	1e12 wei	1,000,000,000,000
Pwei (finney)	1e15 wei	1,000,000,000,000,000
Ether (buterin)	1e18 wei	1,000,000,000,000,000,000

Solidity Operators

•"An operator is a symbol (+,-,*,/) that directs the computer to perform certain mathematical or logical manipulations and is usually used to manipulate data and variables".

•Operators can be classified as :

i.***Arithmetic Operators: (+,-,*,/,%,++,--)***

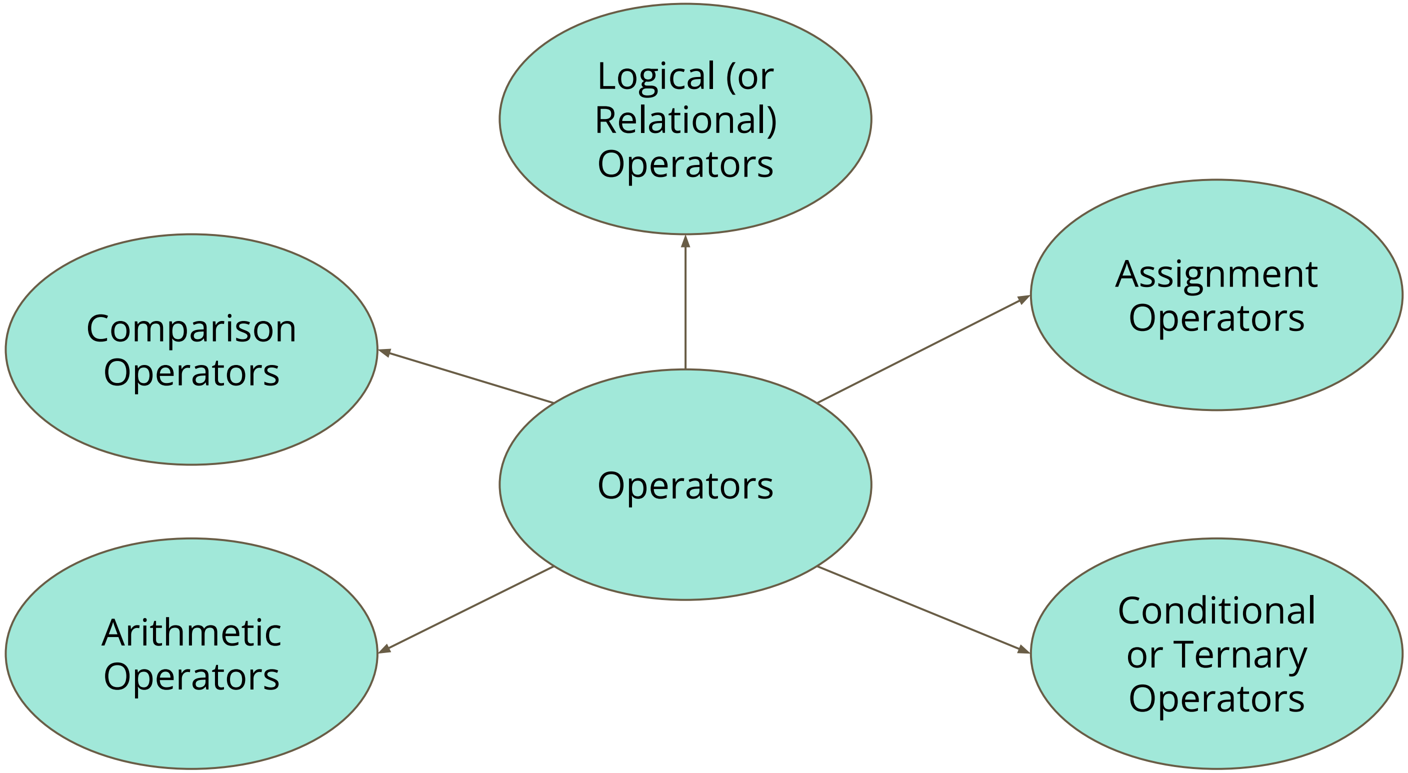
ii.Comparison Operators: (==,!=,>,<,>=,<=)

iii.Logical Operators: (&& (Logical AND), || (Logical OR), ! (Logical NOT))

iv.Assignment Operators: (= (Simple Assignment), += (Add and Assignment) etc.

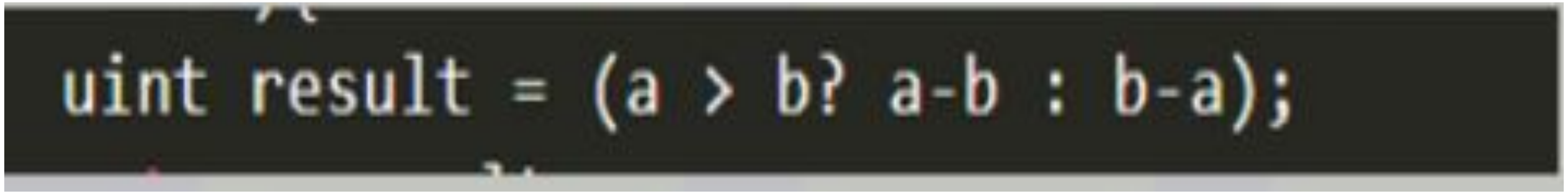
• Example: Let us take a simple expression 4 + 5 is equal to 9.

Here 4 and 5 are called operands and '+' is called the operator


Logical (or Relational) Operators
Comparison Operators
Assignment Operators
Operators
Arithmetic Operators
Conditional or Ternary Operators

Condition Operators

- It is a ternary operator that evaluates the expression first then checks the condition for return values corresponding to true or false
- (test expression) ? true : false



```
uint result = (a > b? a-b : b-a);
```

Value Types in Solidity

Value type variables ***store*** their ***own data***.

• These are the basic data types provided by solidity.

• These types of variables are always ***passed by value.***

• The variables are copied wherever they are ***used in function arguments or assignment.***

1. **Bool :** The possible values are constants true and false

2. **Address** : A special value type designed to hold up to 20 Bytes, or 160 bits, which is the size of an Ethereum address

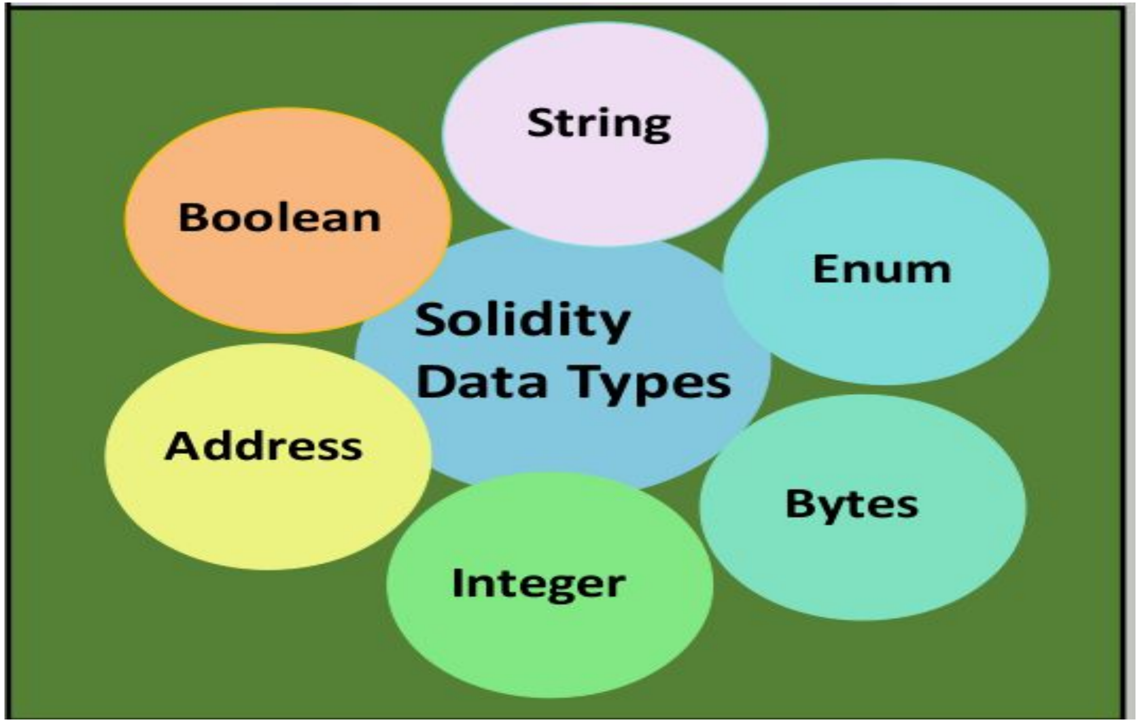
3. **Integer :** This value type variables store a integer value and has two types signed(int) and unsigned(uint) integers

4. **Bytes :** 8-bit signed integer. Everything in memory is stored in bits with binary values 0 & 1. It is stores info in binary format. E.g. bytes by = 0x4 ; bytes1 ff = -62 ; bytes1 ee ='d‘ ;

5. **Enum :** Consists of user-defined data types. It limits a variable to a few predefined values and each copy

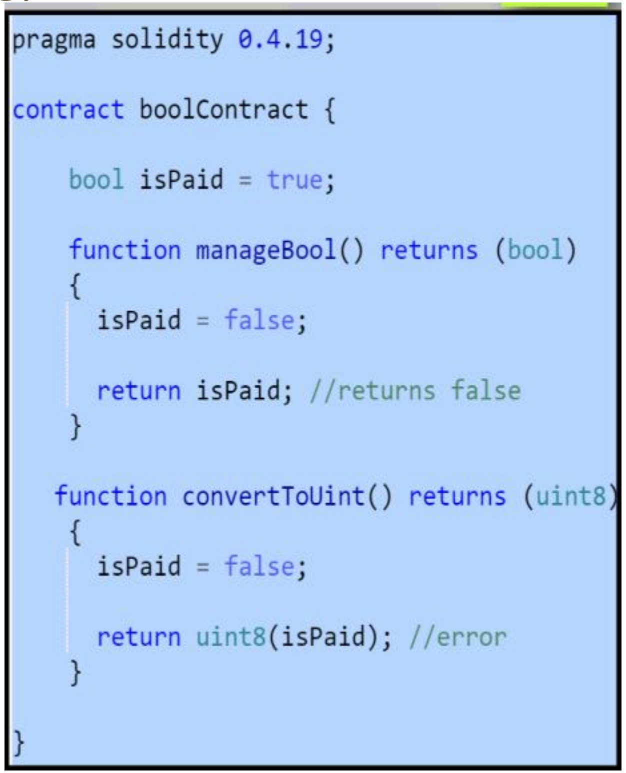
maintains its value. E.g. enum FreshJuiceSize{ SMALL,MEDIUM, LARGE }

Value Types in Solidity



Boolean

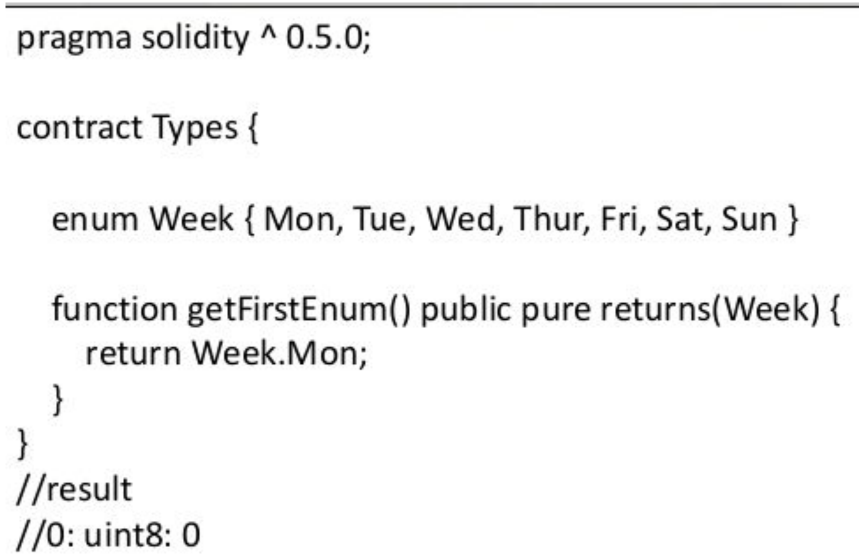
This data type accepts only two values ***True or False***.



```
pragma solidity 0.4.19;

contract boolContract {

    bool isPaid = true;

    function manageBool() returns (bool)
    {
      isPaid = false;

      return isPaid; //returns false
    }

   function convertToUint() returns (uint8)
    {
      isPaid = false;

      return uint8(isPaid); //error
    }

}
```

Enum

- The values in this enumerated list are called enums.
- With the use of enums it is possible to reduce the number of bugs in your code
- Enums are the way of creating user-defined data types, it is usually used to provide names for integral constants which makes the contract better for maintenance and reading.



```
pragma solidity ^ 0.5.0;

contract Types {

    enum Week { Mon, Tue, Wed, Thur, Fri, Sat, Sun }

    function getFirstEnum() public pure returns(Week) {
        return Week.Mon;
    }
}
//result
//0: uint8: 0
```

Address

• Address hold a 20-byte value which represents the size of an Ethereum address.

An address can be used to get balance or to transfer a balance

by balance and transfer method respectively.

• Solidity actually offers two address value types: address and address payable

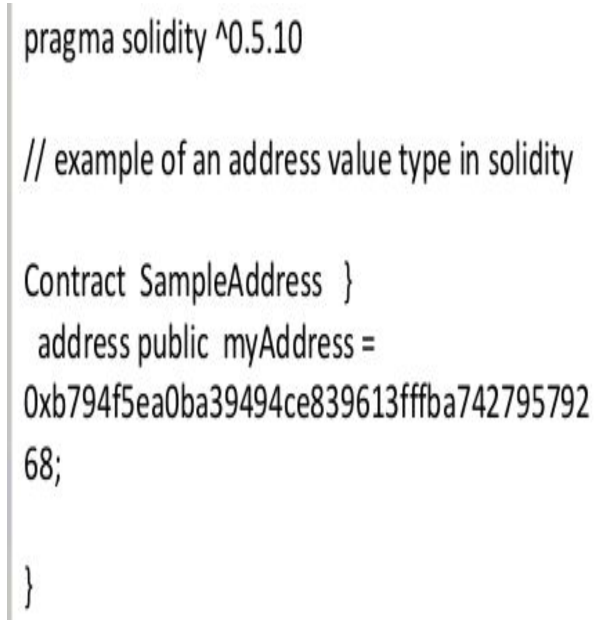
• . The difference between the two is

that address payable can send and transfer Ether.

• We can use an address to acquire a balance

using the .balance method and to transfer a

balance using the .transfer method.

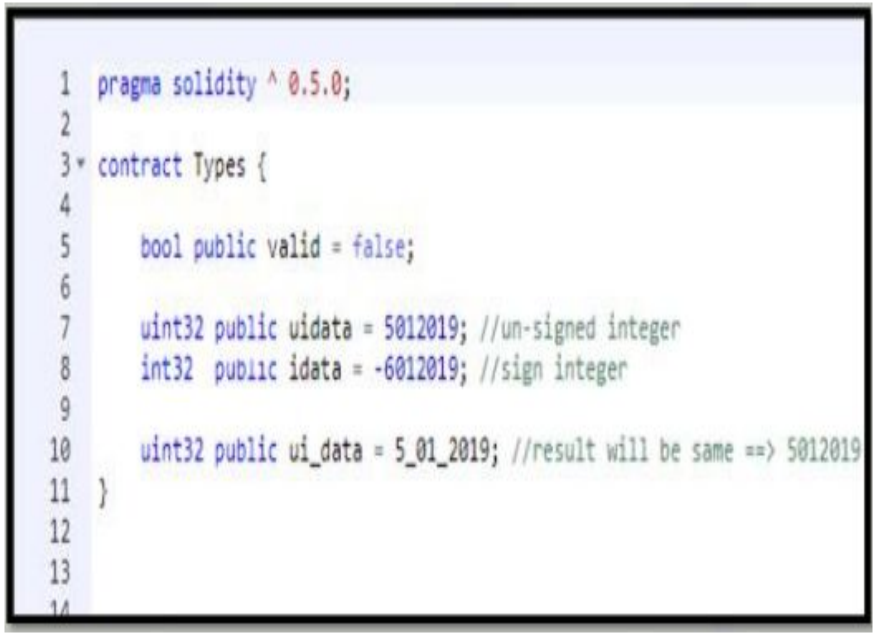


```
pragma solidity ^0.5.10

// example of an address value type in solidity

Contract  SampleAddress  }
 address public  myAddress =
0xb794f5ea0ba39494ce839613fffba742795792
68;

}
```

Integer

Integers help in storing numbers in contracts. Solidity provides the following two types of integer:

- ***Signed integers:*** Signed integers can hold both negative and positive values.
- ***Unsigned integers:*** Unsigned integers can hold only positive values along with zero. They can also hold negative values apart from positive and zero values.



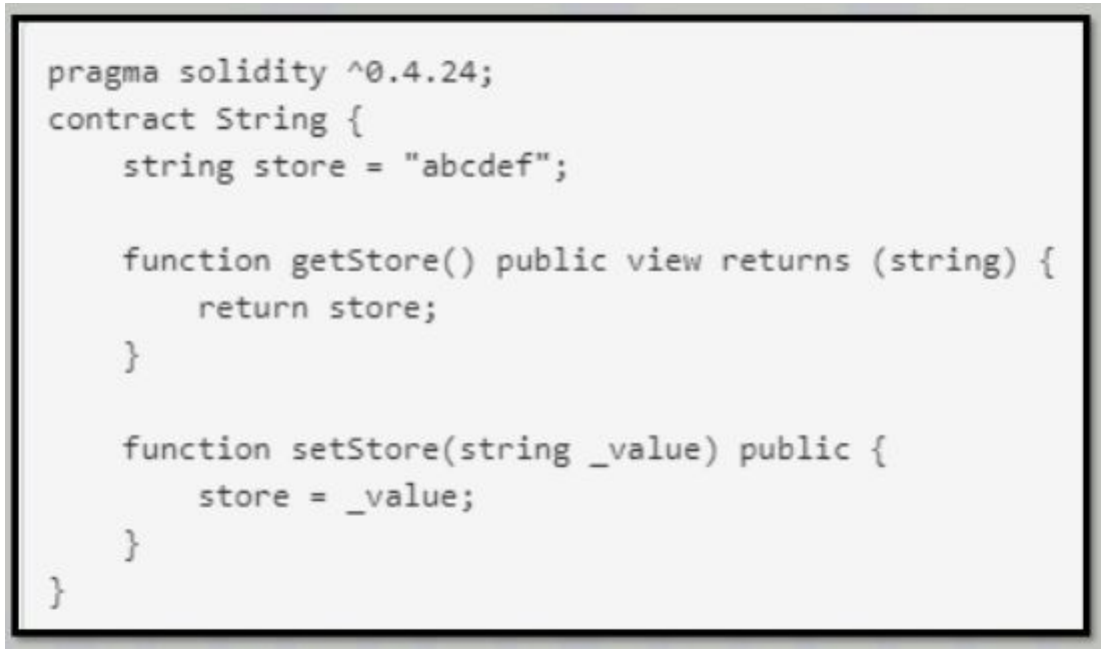
```
pragma solidity ^ 0.5.0;

contract Types {

    bool public valid = false;

    uint32 public uidata = 5012019; //un-signed integer
    int32  public idata = -6012019; //sign integer

    uint32 public ui_data = 5_01_2019; //result will be same ==> 5012019
}
```

String

·Strings in Solidity is a reference type of data type which stores the location of the data instead of directly storing the data into the variable.

• They are dynamic arrays that store a set of characters that can consist of numbers, special characters, spaces, and alphabets.

• Like JavaScript, both Double quote(" ") and Single quote(' ') can be used to represent strings in solidity.



```
pragma solidity ^0.4.24;
contract String {
    string store = "abcdef";

    function getStore() public view returns (string) {
        return store;
    }

    function setStore(string _value) public {
        store = _value;
    }
}
```

References Types in Solidity

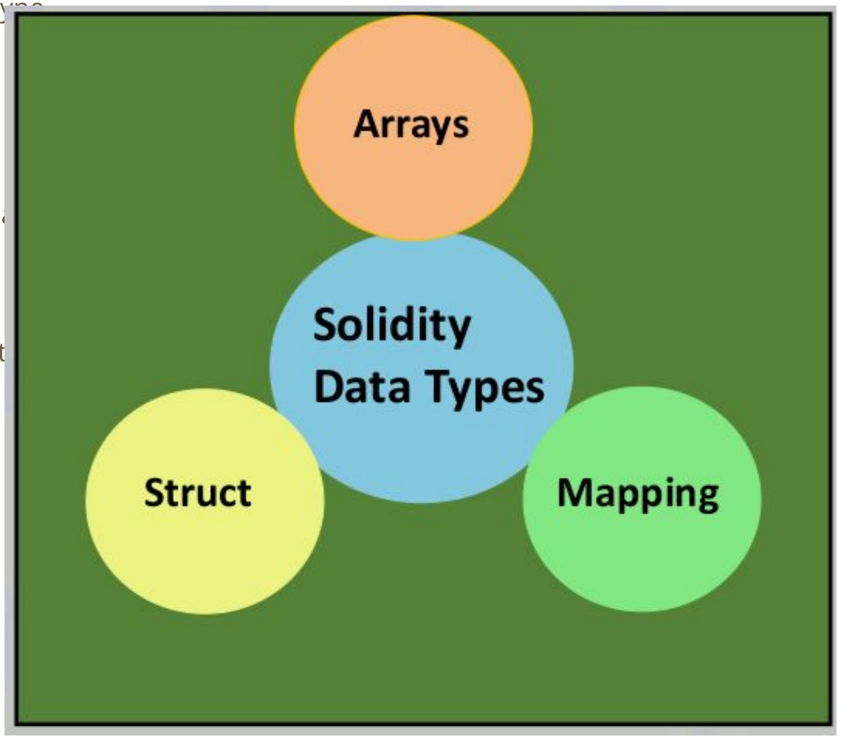
Two separate variables can refer to the exact location, and any change in one variable can affect the other. Several variables that point to the same address can be used to effect a change in a reference data ty

1. **Arrays :** A group of variables of the same data type, with each variable having a unique index. Array size can be fixed or dynamic.

 E.g. uint[6] arrayName;-(Fixed) int[ ] arrayName;- (Dynamic) . In Solidity, bytes

2. **Struct :** It refers to a new (or custom) data type. The struct keyword can be used to define a structure, made up of multiple variables, which can be bot reference type. E.g. struct patient{ string name; string age; uint16 gender; }

3. **Mapping :** Mapping functions similarly to a hashtable or dictionary in other programming languages. It used to store data in the form of key-value pairs. It can retrieve stored data using the supplied key. E.g. **mapping (**unit => address) Variables;

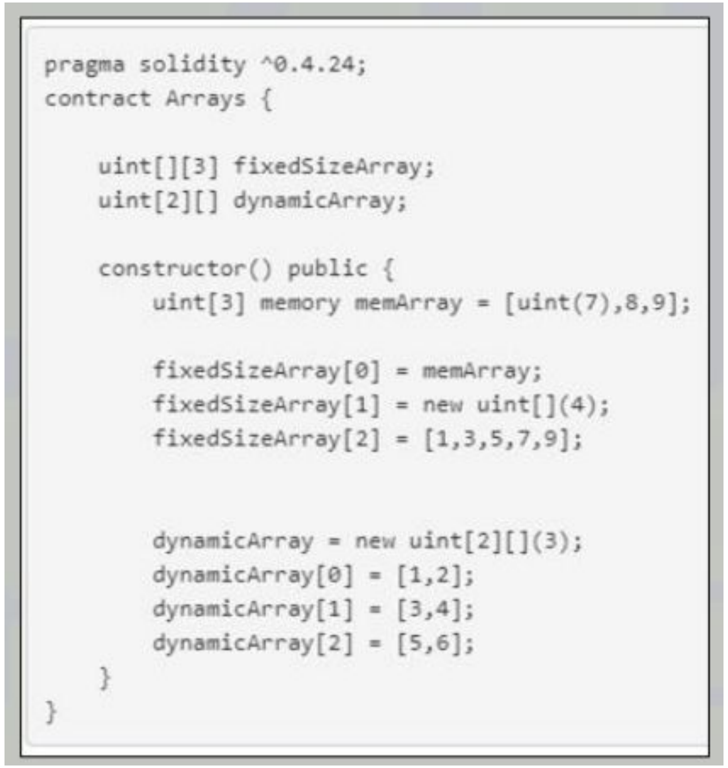


Array

• The array is a data structure to store multiple variable values under a single name.

• An array is a group of variables of the same data type in which variable has a particular location known as an index.

• By using the index location, the desired variable can be accessed. The array size can be fix or dynamic.



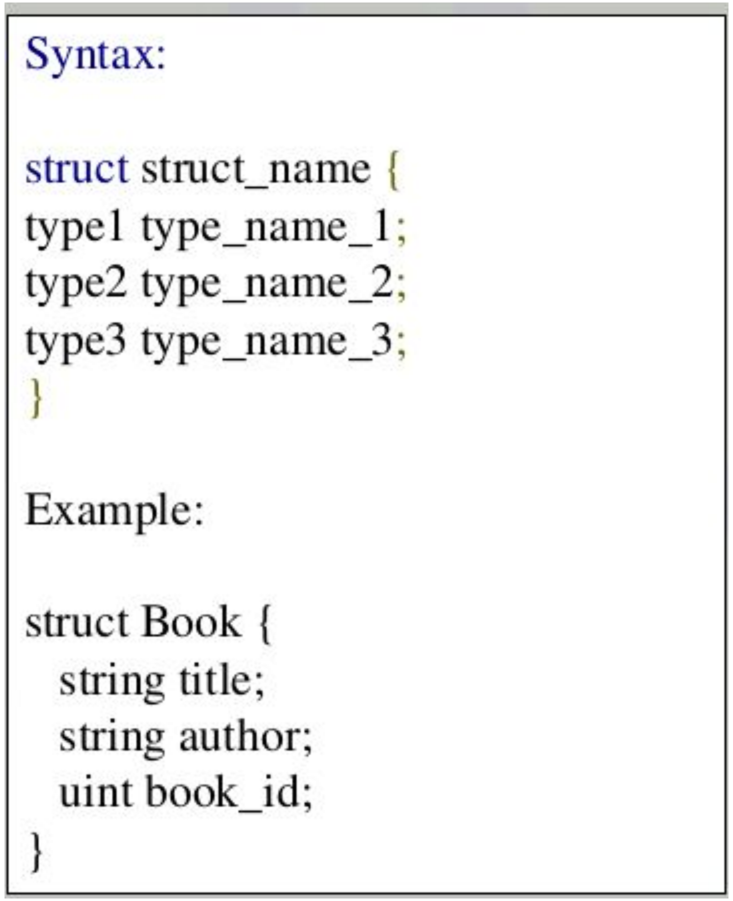
```
pragma solidity ^0.4.24;
contract Arrays {

    uint[][3] fixedSizeArray;
    uint[2][] dynamicArray;

    constructor() public {
        uint[3] memory memArray = [uint(7),8,9];

        fixedSizeArray[0] = memArray;
        fixedSizeArray[1] = new uint[](4);
        fixedSizeArray[2] = [1,3,5,7,9];


        dynamicArray = new uint[2][](3);
        dynamicArray[0] = [1,2];
        dynamicArray[1] = [3,4];
        dynamicArray[2] = [5,6];
    }
}
```

Struct

• Solidity allows users to create and define their own type in the form of structures.

• The structure is a group of different types even though it's not possible to contain a member of its own type.

• The structure is a reference type variable which can contain both value type and reference type



Syntax:

```
struct struct_name {
type1 type_name_1;
type2 type_name_2;
type3 type_name_3;
}
```

Example:

```
struct Book {
   string title;
   string author;
   uint book_id;
}
```

Mapping

• Mapping is a reference type as arrays and structs.

• Following is the syntax to declare a mapping type.

• mapping(_KeyType => _ValueType)

• Mapping is a most used reference type, that stores

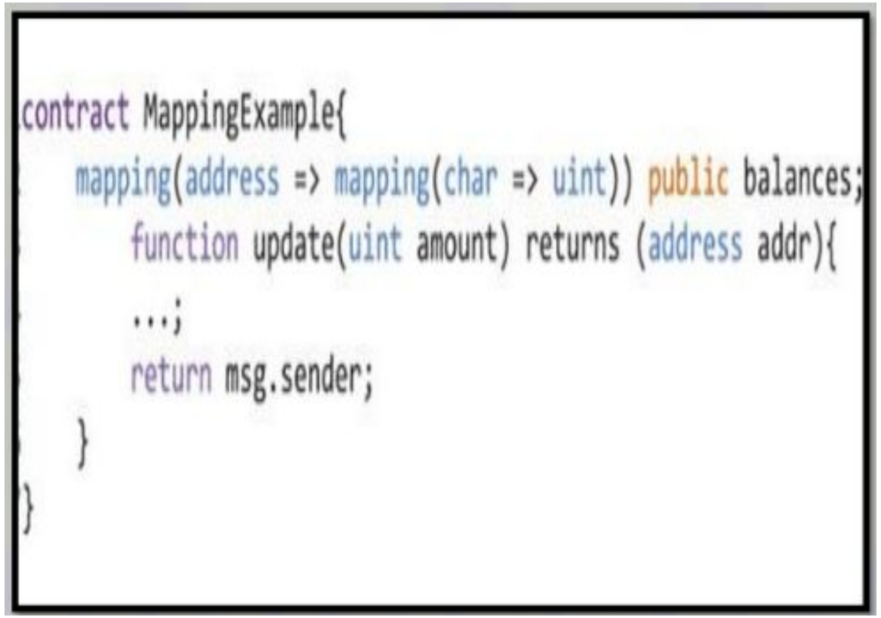
the data in a key-value pair

where a key can be any value types.

• It is like a hash table or dictionary as in any other

programming language, where data can be

retrieved by key.



```
contract MappingExample{
    mapping(address => mapping(char => uint)) public balances;
        function update(uint amount) returns (address addr){
        ...;
        return msg.sender;
    }
}
```

Learning controls flow in Solidity

While Loop

• Loops are used when we have to perform

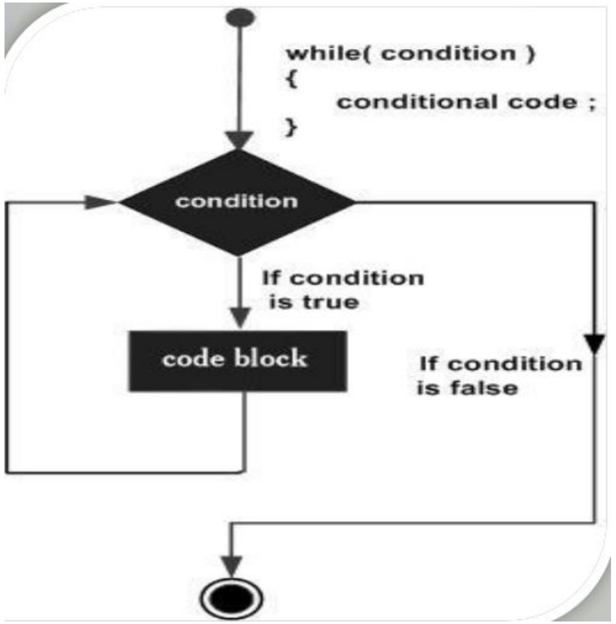
an action over and over again.

• While writing a contract there might be a situation

when we have to perform some task repeatedly.

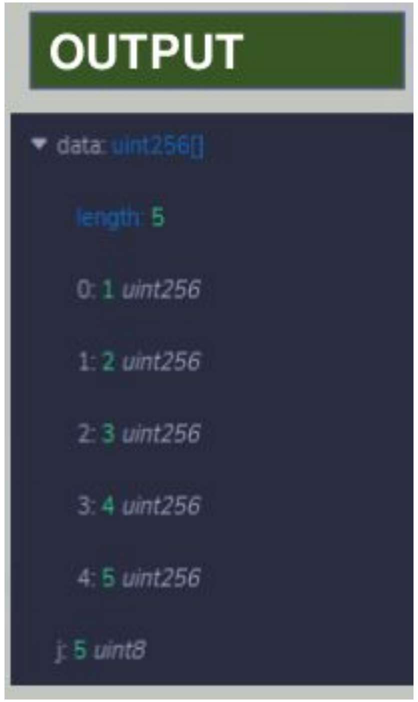
In this situation, loops are implemented to reduce the

number of lines of the statements.



Example of While Loop:

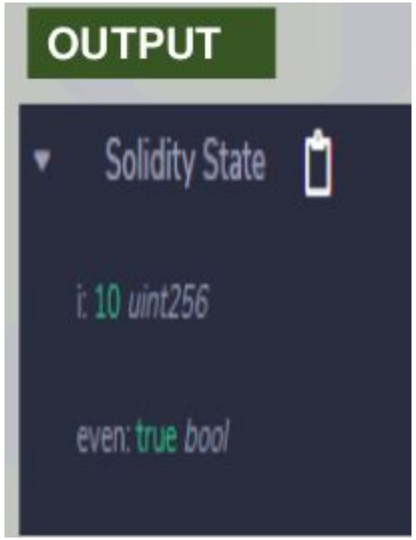
```
function loop(
) public returns( uint [] memory){
while(j < 5) { j++ ;
data.push (j) ;
}
return data;
}
}
```



```
OUTPUT
▾ data: uint256[]
    length: 5
    0: 1 uint256
    1: 2 uint256
    2: 3 uint256
    3: 4 uint256
    4: 5 uint256
  j: 5 uint8
```

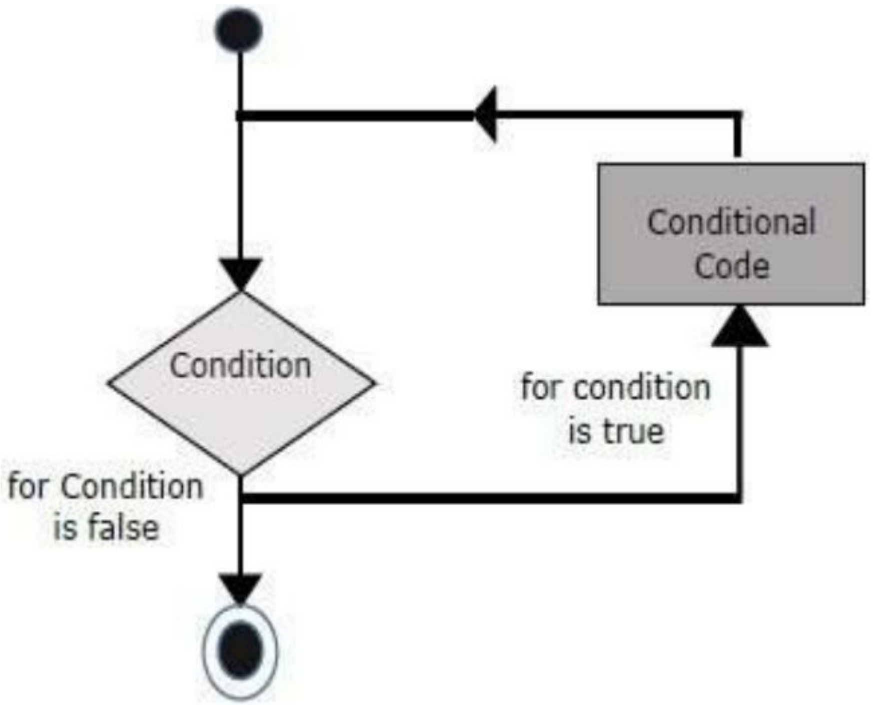
If else

```
function decision making(
) public payable returns(bool){
if (i % 2 == 0){
even = true;
}
else{
even = false;
}
return even;
}
}
```



```
i: 10 uint256
even: true bool
```

For Loop

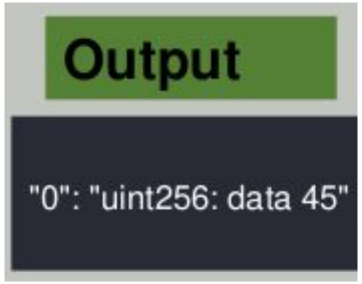

Conditional Code
Condition
for condition is true
for Condition is false

Example of for Loop

The syntax of for loop is Solidity is as follows :

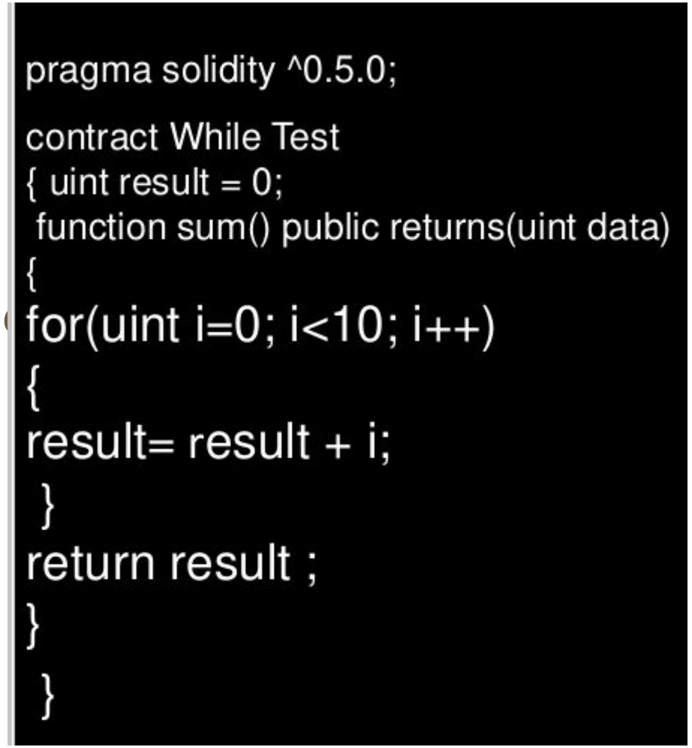
for (initialization; test condition; iteration statement) { Statement(s) to b

executed if test condition is true }



Output

```
"0": "uint256: data 45"
```



```
pragma solidity ^0.5.0;
contract While Test
{ uint result = 0;
 function sum() public returns(uint data)
{
for(uint i=0; i<10; i++)
{
result= result + i;
 }
return result ;
}
 }
```

Solidity Variables

What is a Variable ?

• A variable is basically a placeholder for the data which can be manipulated at runtime

• Variables allow users to retrieve and change the stored information

• Variables in solidity are used to store the data in blockchain as well as used in functions for processing.

• Each declared variable always has a default value based on type.

• Solidity is a statically typed lang , which means that variable type needs to be specified during declaration.

Variable Syntax

Syntax : DataType accessModifier variableName ;

e.g. uint public X = 100 ; string private Y = “Name” ; int Z = -99 ;

• **DataType** is an inbuilt or custom user-defined data type such as boolean, unsigned-integer, integer, string

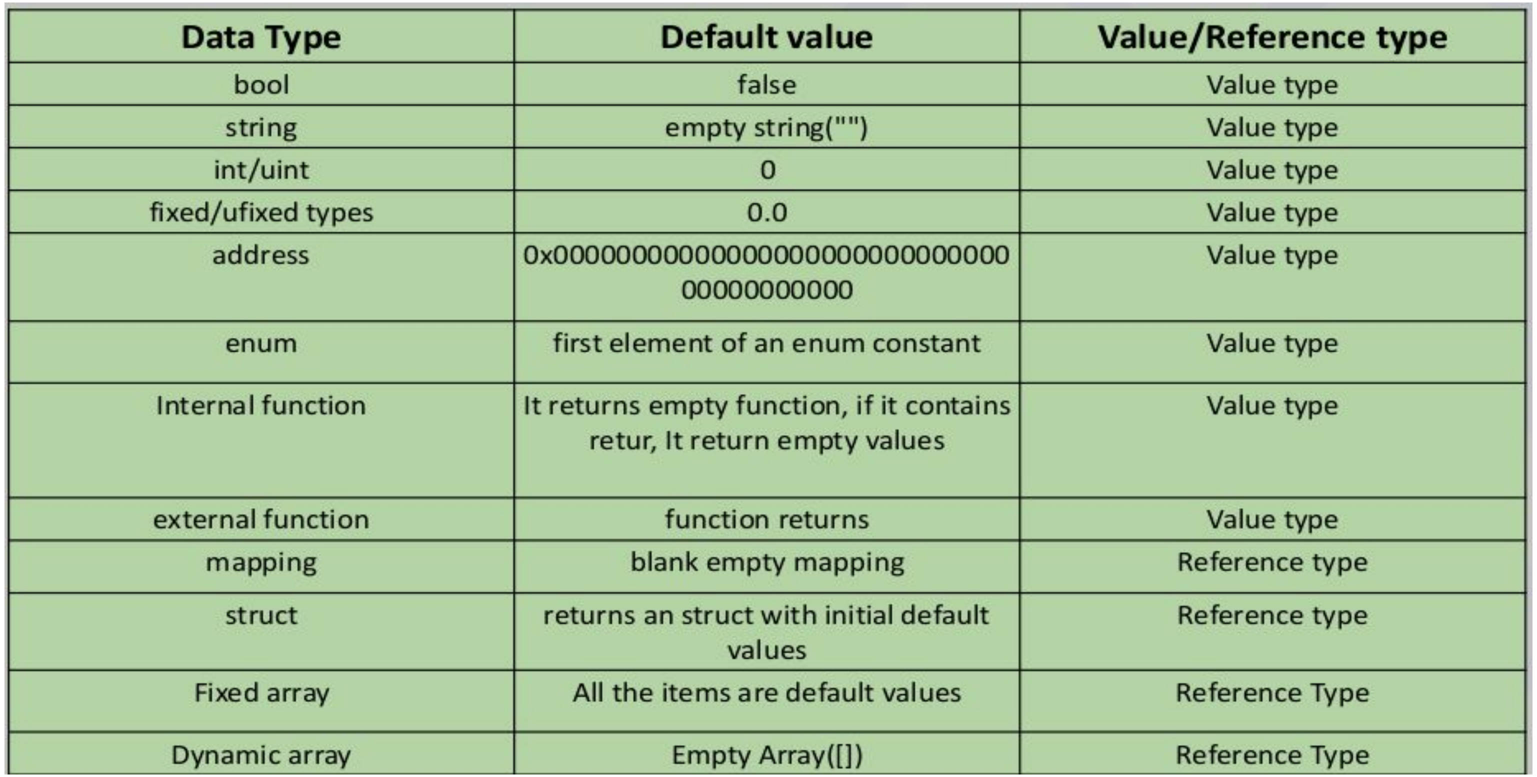
• **Access-Modifier(**Variable Visibility) also called visibility and how and where the variable values

are accessible. Possible values are public, private, internal and external

• **Variable Name** is a valid identifier in solid, it contains a combination of alphabets and numerics.

The variables can be declared in Contracts or functions

Variable – Data Type , Default Values and Type



Data Type	Default value	Value/Reference type
bool	false	Value type
string	empty string("")	Value type
int/uint	0	Value type
fixed/ufixed types	0.0	Value type
address	0x00000000000000000000000000000000000000000	Value type
enum	first element of an enum constant	Value type
Internal function	It returns empty function, if it contains retur, It return empty values	Value type
external function	function returns	Value type
mapping	blank empty mapping	Reference type
struct	returns an struct with initial default values	Reference type
Fixed array	All the items are default values	Reference Type
Dynamic array	Empty Array([])	Reference Type

Variable Mutability(Scope)

1. State Variables − Variables whose values are permanently stored in a contract storage
2. Local Variables − Variables whose values are present till function is executing
3. Global Variables − Special variables exists in the global namespace used to get information about the blockchain

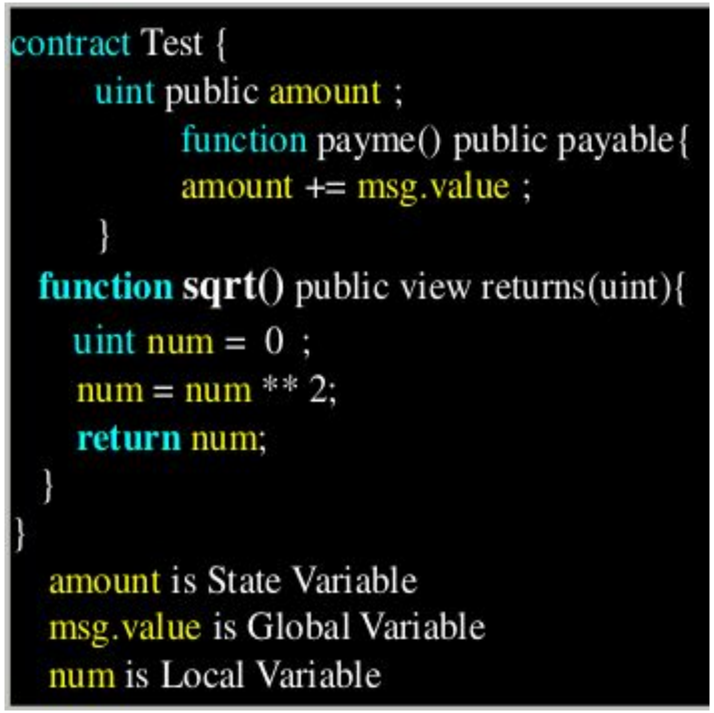
Variable Mutability(Scope)

1. **State Variables** − Variables whose values are permanently stored in a contract storage

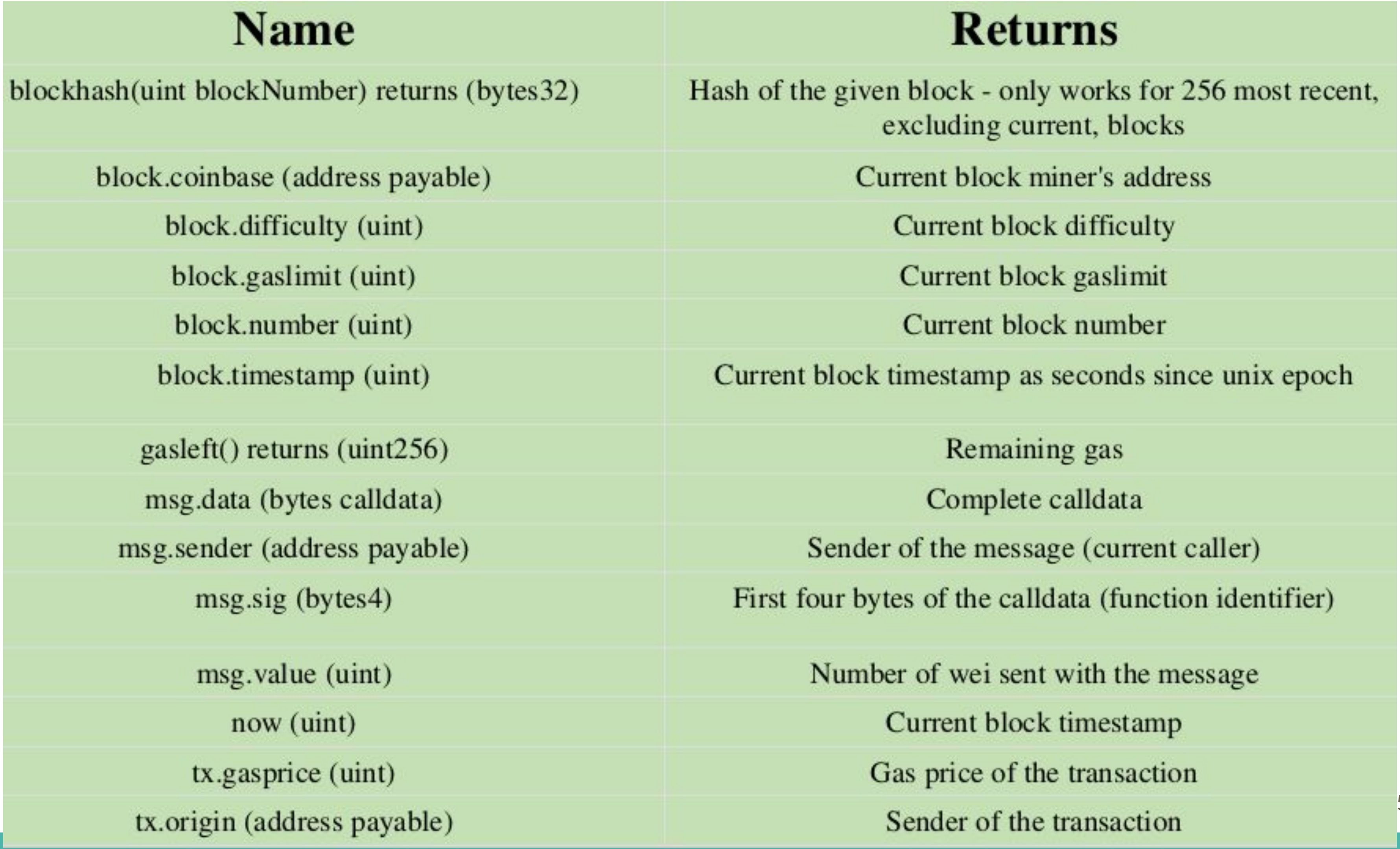
2. **Local Variables** − Variables whose values are present till function is executing

3. **Global Variables** − Special variables exists in the global namespace used to

get information about the blockchain



```
contract Test {
        uint public amount ;
                function payme() public payable{
                amount += msg.value ;
        }
  function sqrt() public view returns(uint){
     uint num =  0  ;
     num = num ** 2;
     return num;
  }
}
   amount is State Variable
   msg.value is Global Variable
   num is Local Variable
```



Name	Returns
blockhash(uint blockNumber) returns (bytes32)	Hash of the given block - only works for 256 most recent, excluding current, blocks
block.coinbase (address payable)	Current block miner's address
block.difficulty (uint)	Current block difficulty
block.gaslimit (uint)	Current block gaslimit
block.number (uint)	Current block number
block.timestamp (uint)	Current block timestamp as seconds since unix epoch
gasleft() returns (uint256)	Remaining gas
msg.data (bytes calldata)	Complete calldata
msg.sender (address payable)	Sender of the message (current caller)
msg.sig (bytes4)	First four bytes of the calldata (function identifier)
msg.value (uint)	Number of wei sent with the message
now (uint)	Current block timestamp
tx.gasprice (uint)	Gas price of the transaction
tx.origin (address payable)	Sender of the transaction

State Variables Visibility

Scope of local variables is limited to function in which they are defined but State variables can have three types of scopes

1. Public − Public state variables can be accessed internally as well as via messages. For a public state variable, an automatic getter function is generated.

2. Internal − Internal state variables can be accessed only internally from the current contract or contract deriving from it without using this keyword.

3. Private − Private state variables can be accessed only internally from the current contract they are defined not in the derived contract from it.

State Variables Visibility Comparison

Visibility / Scope	Inline Functions	Same Contract	Derived Contract	Other Contract
Public				
Internal(Default)				
Private				

Solidity Functions

What is function ?

- A function is a group of reusable code which can be called anywhere in your program
- This eliminates the need of writing the same code again and again
- It helps programmers in writing modular(smaller units) codes and separating bigger problems in separate parts
- Functions allow a programmer to divide a big program into a number of small and manageable parts(functions)

1. Function Terminologies : Function Declaration, Function Calling, Function Parameters,

Function Arguments, Return Statement, Function Behaviour (Special in Solidity), Function

Modifiers and Function Visibility

2. Function Visibility : Public , Private , Internal and External

3. Function Mutability : Pure, View, Payable, Fallback

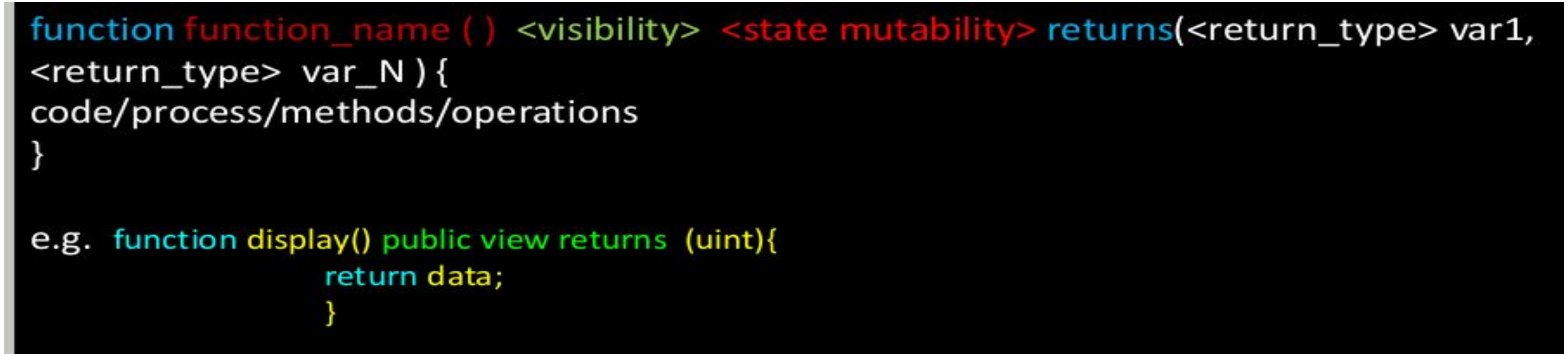
Functions Syntax(Function Declaration)

- Function is a keyword in solidity
- Function-name: Name of the function and valid identifier in solidity, it is case sensitive and keywords cannot be used as names
- Visibility: function visibility how the function is visible to the caller
- Modifier <state mutability> : declare the function to change the behavior of contract data
- Returns: Type of data is being returned(int, uint, boolean, string, address)

Functions Syntax(Function Declaration)

- Function is a keyword in solidity

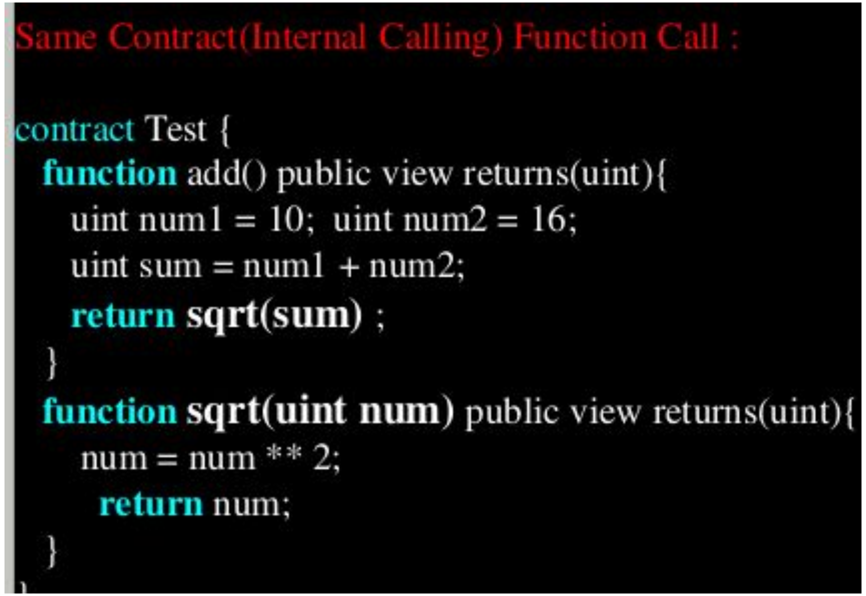
- Function-name: Name of the function and valid identifier in solidity, it is case sensitive and keywords cannot be used as names
- Visibility: function visibility how the function is visible to the caller
- Modifier <state mutability> : declare the function to change the behavior of contract data
- Returns: Type of data is being returned(int, uint, boolean, string, address)



```
function function_name ( )  <visibility>  <state mutability> returns(<return_type> var1,
<return_type>  var_N ) {
code/process/methods/operations
}

e.g.  function display() public view returns  (uint){
                    return data;
                    }
```

Function Call

When a function is invoked in another contract or in the same contract by any other function/method.



Same Contract(Internal Calling) Function Call :

```
contract Test {
  function add() public view returns(uint){
    uint num1 = 10;  uint num2 = 16;
    uint sum = num1 + num2;
    return sqrt(sum) ;
  }
  function sqrt(uint num) public view returns(uint){
     num = num ** 2;
      return num;
  }
```

Pure Function

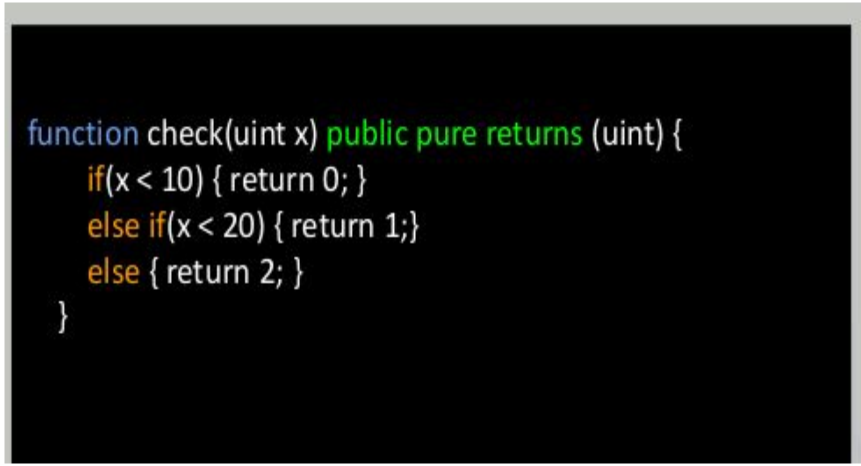
It has following features –

- Pure functions does not access data from blockchain
- Does not allow read or write operations
- No access to state data or transaction data
- Return type based on the data from a function

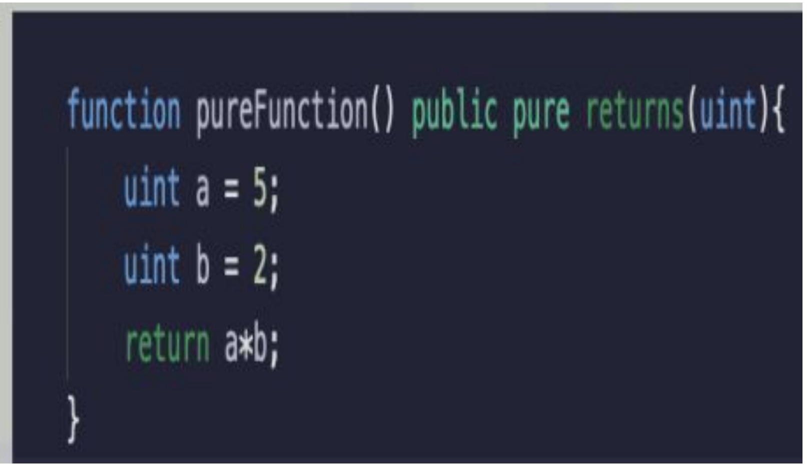
Pure Function

It has following features −

- Pure functions does not access data from blockchain
- Does not allow read or write operations
- No access to state data or transaction data
- Return type based on the data from a function



```
function check(uint x) public pure returns (uint) {
      if(x < 10) { return 0; }
      else if(x < 20) { return 1;}
      else { return 2; }
   }
```



```
function pureFunction() public pure returns(uint){
    uint a = 5;
    uint b = 2;
    return a*b;
}
```

View Function

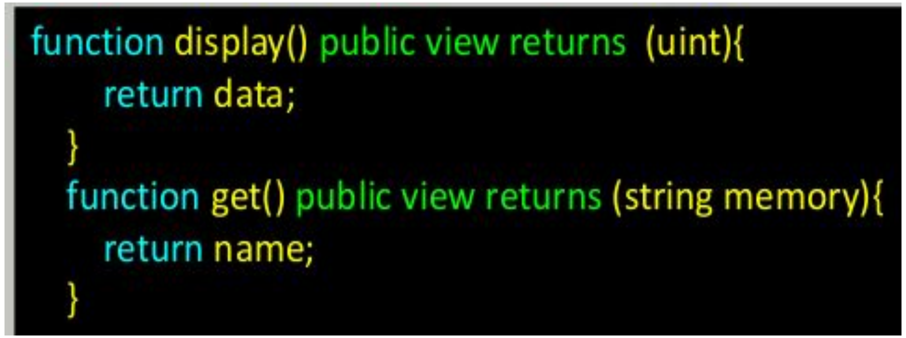
It has following features −

The view functions display the data.

functions do not modify the state variable,

Suggest using for read data operations

A view function only reads but doesn’t modify the variables of the state.



```
function display() public view returns  (uint){
      return data;
   }
   function get() public view returns (string memory){
      return name;
   }
```

Payable Function

It has following features –

• payable functions allows accepting eth

from a caller

• Function fails if the sender has not pro

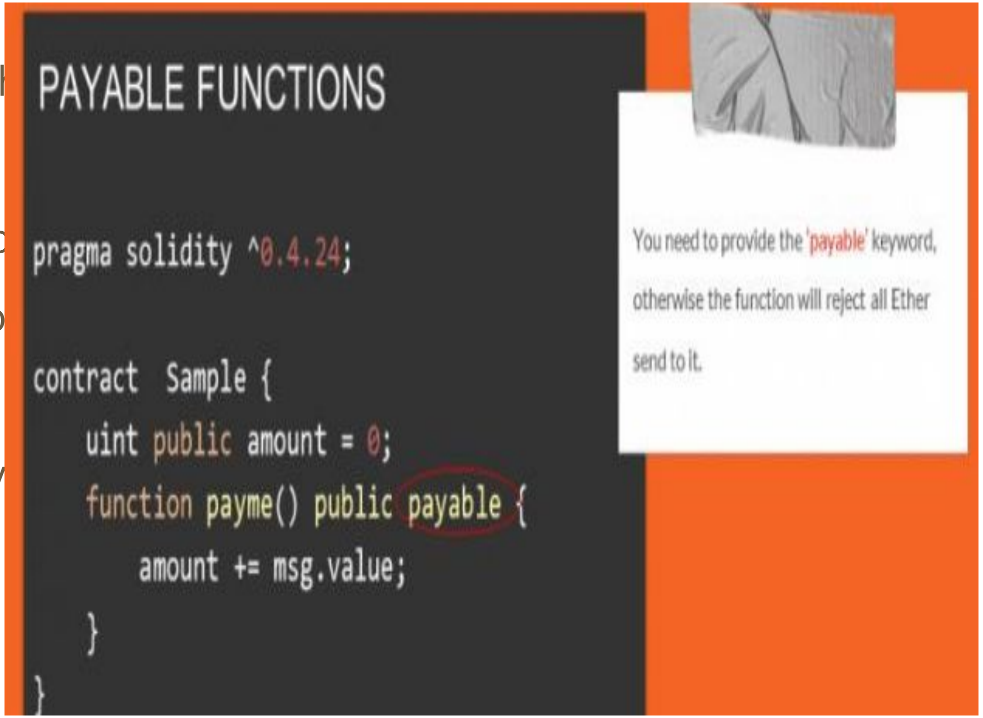
• Other functions do not accept ether, o

functions allow

• Functions and addresses declared pay

receive ether into the

contract



PAYABLE FUNCTIONS

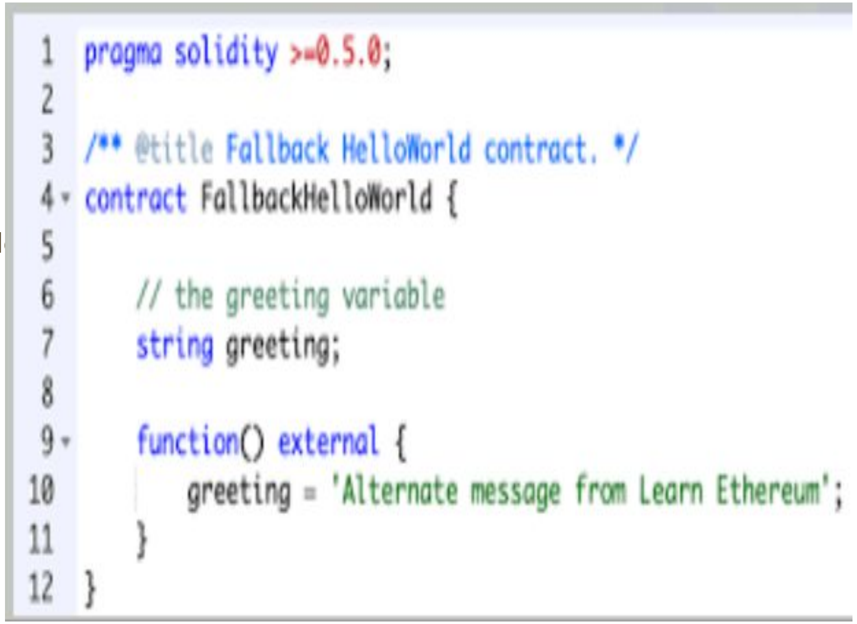
```
pragma solidity ^0.4.24;

contract  Sample {
    uint public amount = 0;
    function payme() public payable {
        amount += msg.value;
    }
}
```

You need to provide the 'payable' keyword, otherwise the function will reject all Ether send to it.

Fallback Functions

It has following features −

- It is called when a non-existent function is called on the contract
- It is required to be marked external
- It has no name, no arguments, no return and cannot be marked payabl
- It can be defined one per contract
- It will throw exception if contract receives plain ether without data



```
pragma solidity >=0.5.0;

/** @title Fallback HelloWorld contract. */
contract FallbackHelloWorld {

    // the greeting variable
    string greeting;

    function() external {
        greeting = 'Alternate message from Learn Ethereum';
    }
}
```

Functions Visibility and Scope

•There are 4 types of function visibilities and scopes :

1. Public − The function can be accessed from everywhere and anywhere.Public functions are visible from internal and external

contracts. Can be called inside a contract or outside contract - Default is public - It is visible to public, so security concern.

2. Internal − The function can be accessed from inside the contract as well as the child(derived, inherited) contracts that inherit it

3. Private − The function can be accessed only from inside the contract. Can be called inside a contract, Not able to call from an outside contract

4. External - This function can be accessed only from outside the contract. Other functions of the contract cannot invoke it. External functions are visible to public functions.

State Variables Visibility Comparison

Visibility / Scope	Same Contract	Derived Contract	Another Contracts
Public			
Internal(Default)			
Private			

Solidity-Inheritance

Solidity inheritance lets us combine multiple contracts into a single one.

• The base contracts are the ones from which other inherit.

• Those contracts that inherit data are derived.

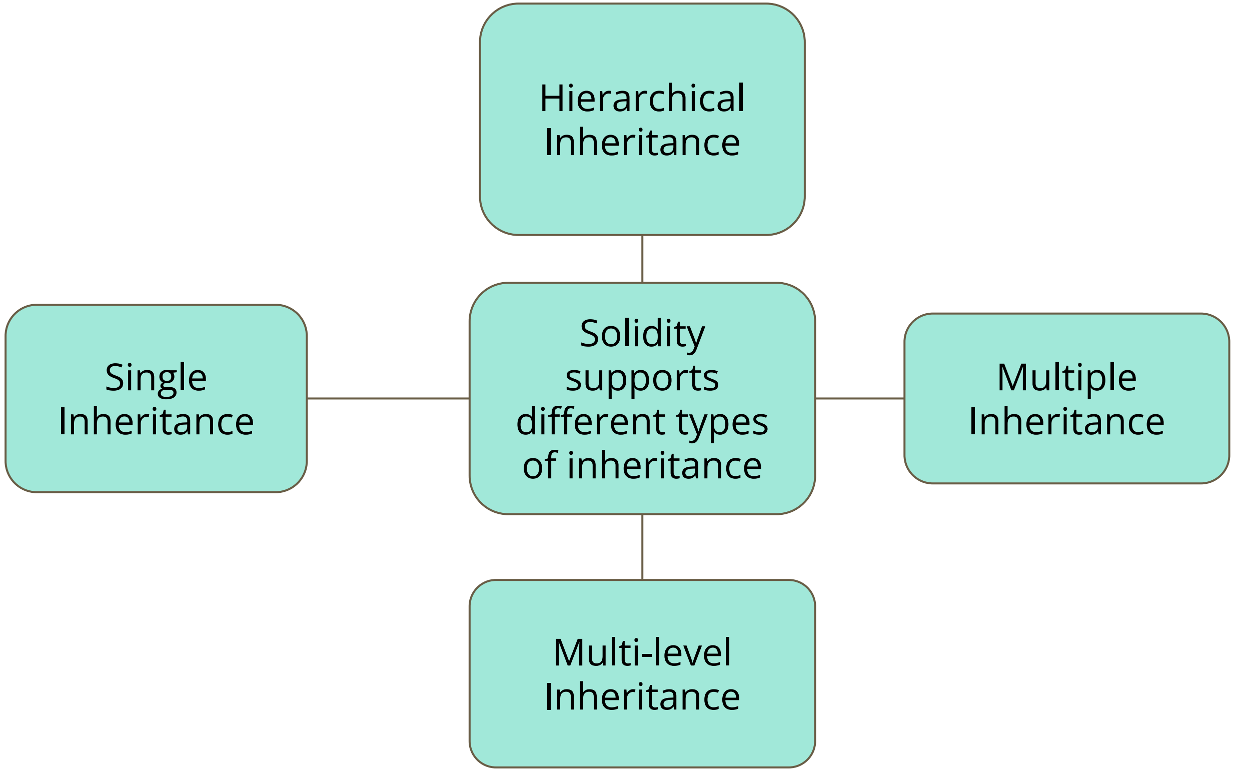
• This is a process resulting in parent-child relationships between contracts.

• Solidity supports both single as well as multiple inheritance

• Syntax Solidity supports different types of inheritance

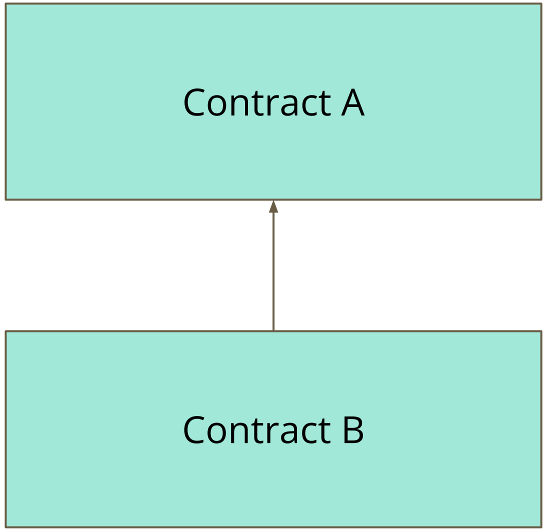
```
contract baseContract {

//parent contract

}

contract derivedContract is baseContract {

//child contract

}
```

Inheritance

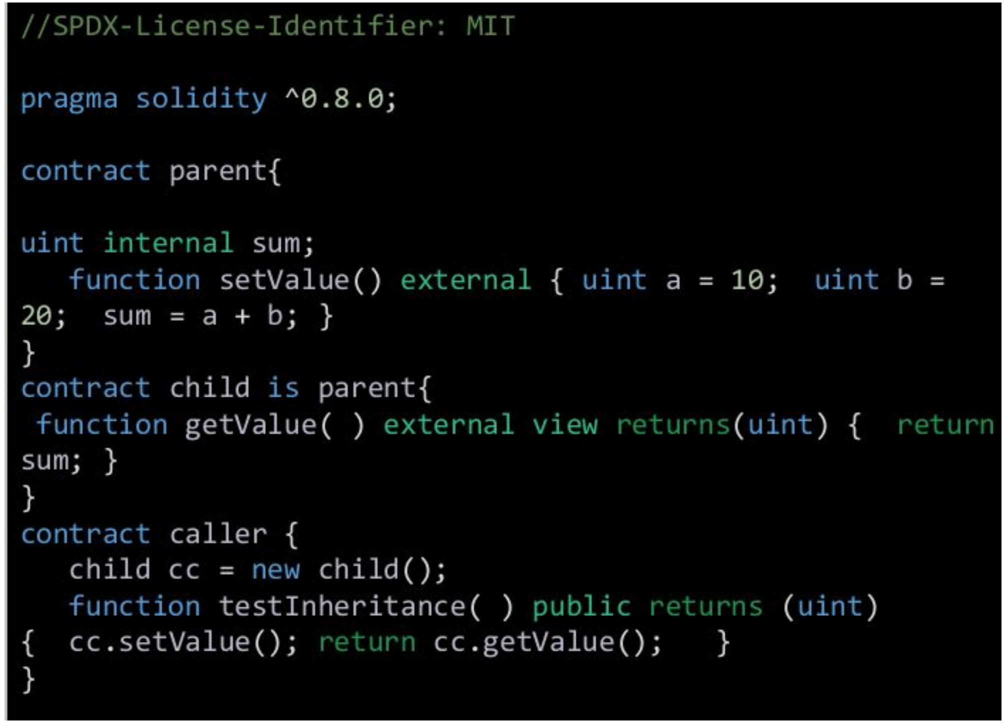


Single Inheritance

- Single or single-level inheritance helps to use variables and functions of the base contract to the derived contract.



Single Inheritance Example

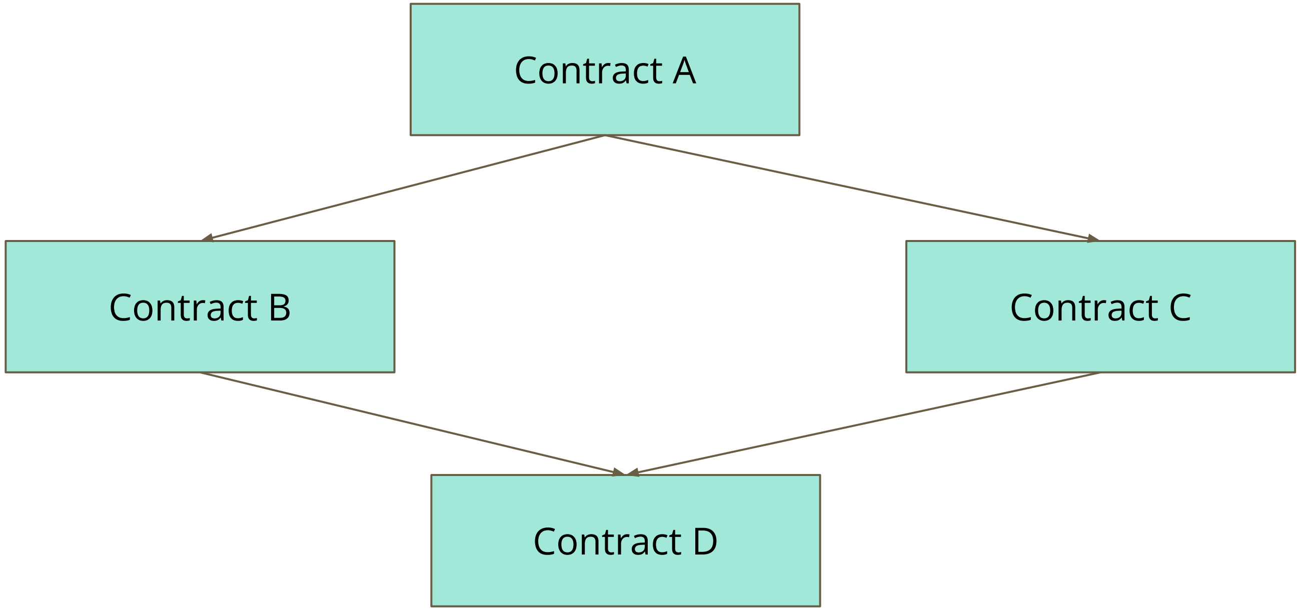


```
//SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract parent{

uint internal sum;
   function setValue() external { uint a = 10;  uint b =
20;  sum = a + b; }
}
contract child is parent{
 function getValue( ) external view returns(uint) {  return
sum; }
}
contract caller {
   child cc = new child();
   function testInheritance( ) public returns (uint)
{  cc.setValue(); return cc.getValue();    }
}
```

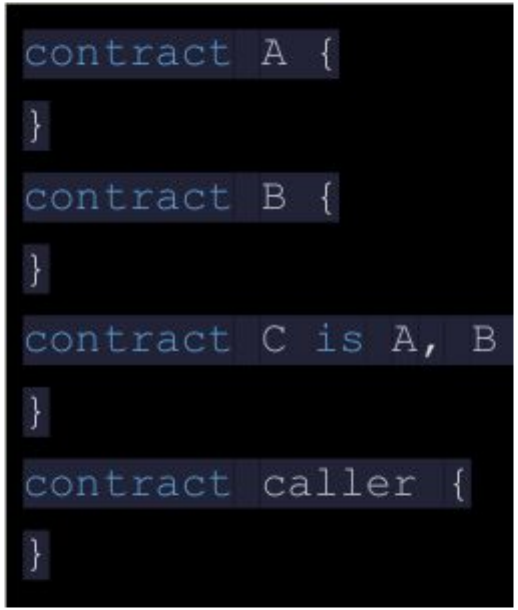
Multiple Inheritance

- Unlike C# and Java, Solidity supports multiple-inheritance. The single contract can be inherited from more than one contract. Here, in the following diagram, shown is the multiple-inheritance.

Multiple Inheritance



Multiple Inheritance Example

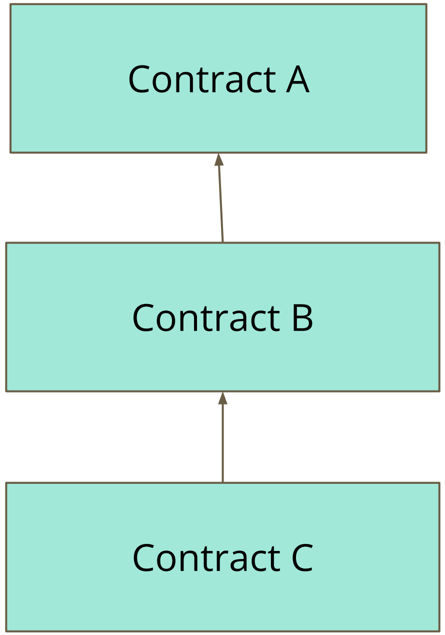


```
contract A {
}
contract B {
}
contract C is A, B
}
contract caller {
}
```

Multi-level Inheritance

- Multilevel inheritance is similar to the single level, but the difference is, it has levels of parent-child relationships. The child contract—derived from a parent contract—also acts as a parent contract of another contract.

Multi-level Inheritance

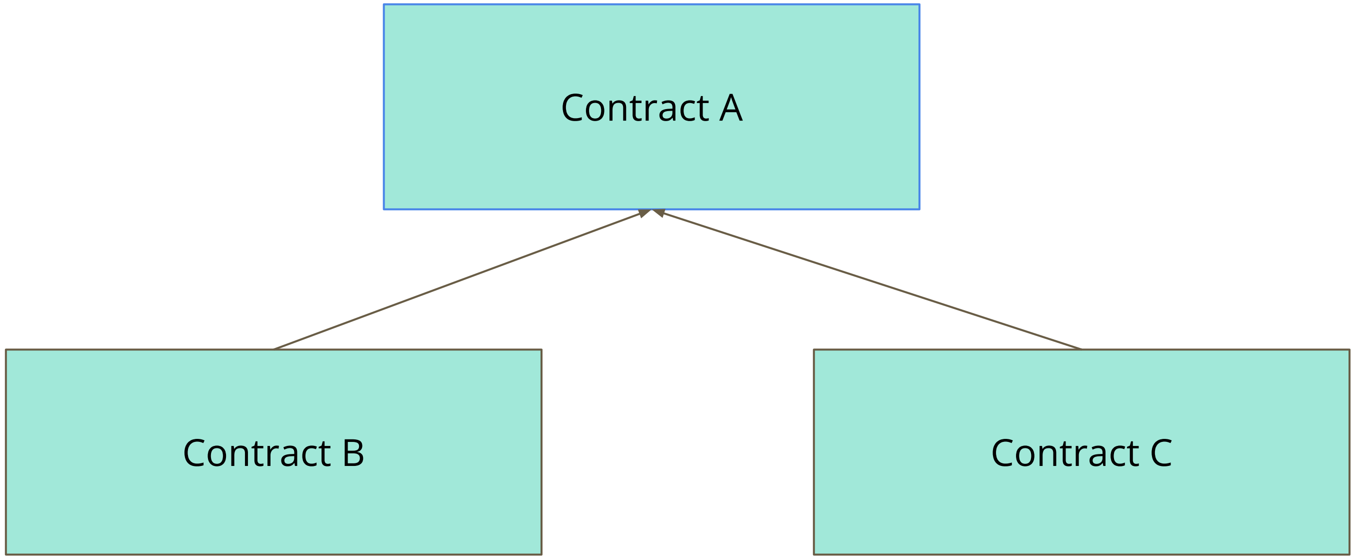


Multi-level Inheritance Example

```
contract A {

}

contract B is A {

}

contract C is B {

}

contract caller {

}
```

Hierarchical Inheritance

- In hierarchical inheritance, the base contract has over one derived contract.
- This inheritance is useful when one common functionality is used in multiple places.
- The following diagram is displayed the type hierarchical, where a single contract is a parent of multiple child contracts.


Contract A
Contract B
Contract C

Hierarchical Inheritance Example

```
contract A {

}

contract B is A {

}

contract C is A {

}
```

Error Handling in Solidity

Error Handling in Solidity

• Solidity has many functions for error handling. Errors can occur at compile time or runtime. Solidity is compiled to byte code and if there is any syntax error then check happens at compile-time, while runtime errors are difficult to catch and occurs mainly while executing the contracts.

• Some of the runtime errors are out-of-gas error, data type overflow error, divide by zero error, array-out-of-index error, etc.

• Until version 4.10 a single throw statement was there in solidity to handle errors, so to handle errors multiple if…else statements, one has to implement for checking the values and throw errors which consume more gas.

• After version 4.10 new error handling construct assert, require, revert statements were introduced and the throw was made obsolete.

Require

- Here we check the conditions and revert the state in case condition do not meet.
- This is most commonly used in solidity for error handling.
- Require statement does not consume any gas.

Syntax

require(condition, errorMessage)

Require

Example

require(_input >= 0, "invalid uint8");

require(_input <= 255, "invalid uint8");

return "Input is Uint8";

Assert

• The assert function, like require, is a convenience function that checks for conditions.

• If a condition fails, then the function execution is terminated with an error message.

• assert() takes only one parameter as input.

• If we pass a condition to assert(), and if the condition is true, then the function

execution continues and the execution jumps to the next statement in the function.

Syntax

assert(<condition to be checked/validated>);

Example

assert(balance %2 == 0)

Revert

• The revert function can be used to flag an error and revert the current call. You can as well provide a message containing details about the error, and the

message will be passed back to the caller. However, the message, like in require(), is an optional parameter.

• revert() causes the EVM to revert all the changes made to the state, and things

return to the initial state or the state before the function call was made.

Syntax

revert("failure reason")

Example

revert("you do not have sufficient balance")

Global function

The global functions are grouped into four major contexts.

These contexts are listed below :

- Address
- Block
- Transaction
- Message

Address Context

• balance: The balance function returns the balance of the Address in Wei.

• function getBalance() public view returns (uint) {

return address(this).balance;

• transfer(): The transfer function sends the given amount of Wei from the current account to the Address mentioned

• function sendViaTransfer(address payable _to) public payable

_to.transfer(msg.value);

• send(): The send function sends the given amount of Wei from the current account to the Address mentioned.

• function sendViaSend(address payable _to) public payable bool sent =

_to.send(msg.value); require(sent, "Failed to send Ether");

• call(), staticcall() and delegatecall(): These are low-level functions. These functions don't go through the checks by

the Solidity compiler, and, hence, it is advised not to use them unless absolutely necessary.

• (bool sent, bytes memory data) = _to.call{value: msg.value}(""); require(sent, "Failed to send Ether");

Block Context

- block.coinbase: It returns the address of the miner than mined the current block.
- block.difficulty: It returns the difficulty at the time when the current block was mined.
- block.timestamp: It returns the timestamp at which the current block was mined.
- block.gaslimit: It returns the total gaslimit of all the transactions mined in the current block.
- block.number: It returns the number of the newest block in the blockchain

Transaction Context

• tx.gasprice: It returns the gas price of the transaction sent by the sender as part of the transaction.

• tx.origin: It returns the address of the original sender of the transaction.

Message Context

• msg.value: This function returns the number of Wei that was sent with the message or the transaction.

• function _msgValue() internal view virtual returns (bytes calldata) { return msg.data; }

• msg.sender: This function returns the immediate sender of the message or the transaction. Unlike tx.origin, msg.sender returns the address of the previous account in the flow of the message. If A

sends the message to B and then B sends it to C, then if C calls msg.sender on that message, it will receive the address of B as the return value

• function _msgSender() internal view virtual returns (address) { return msg.sender; }

• msg.gasleft: This function returns the remaining gas for the transaction. If an account feels that the gas remaining is inadequate or insufficient for a transaction to complete, then it will fail the transaction.

• function _msgGasleft() internal view virtual returns (bytes calldata) { return msg.data; }

Important Global Function

• now(): This function returns the timestamp when the last block in the blockchain was created. Since the block creation rate in Ethereum is approximately 15 seconds, the timestamp returned will be approximately 15-30 seconds prior to the current time. The timestamp returned by now() is the number of milliseconds since the Unix Epoch time.

• selfdestruct(): As the name suggests, this function is used by a contract to destroy or kill itself. You need to pass an address as argument to this function, and all the balance ethers that are present on the

Events & Logging

Event

- Events allow us to “print” information on the block chain in a way that is more searchable and gas efficient than just saving to public storage variables in our smart contracts.
- Event is an inheritable member of a contract. An event is emitted, it stores the arguments passed in transaction logs.
- These logs are stored on block chain and are accessible using address of the contract till the contract is present on the block chain.
- An event generated is not accessible from within contracts, not even the one which have created and emitted them.

Syntax of Events

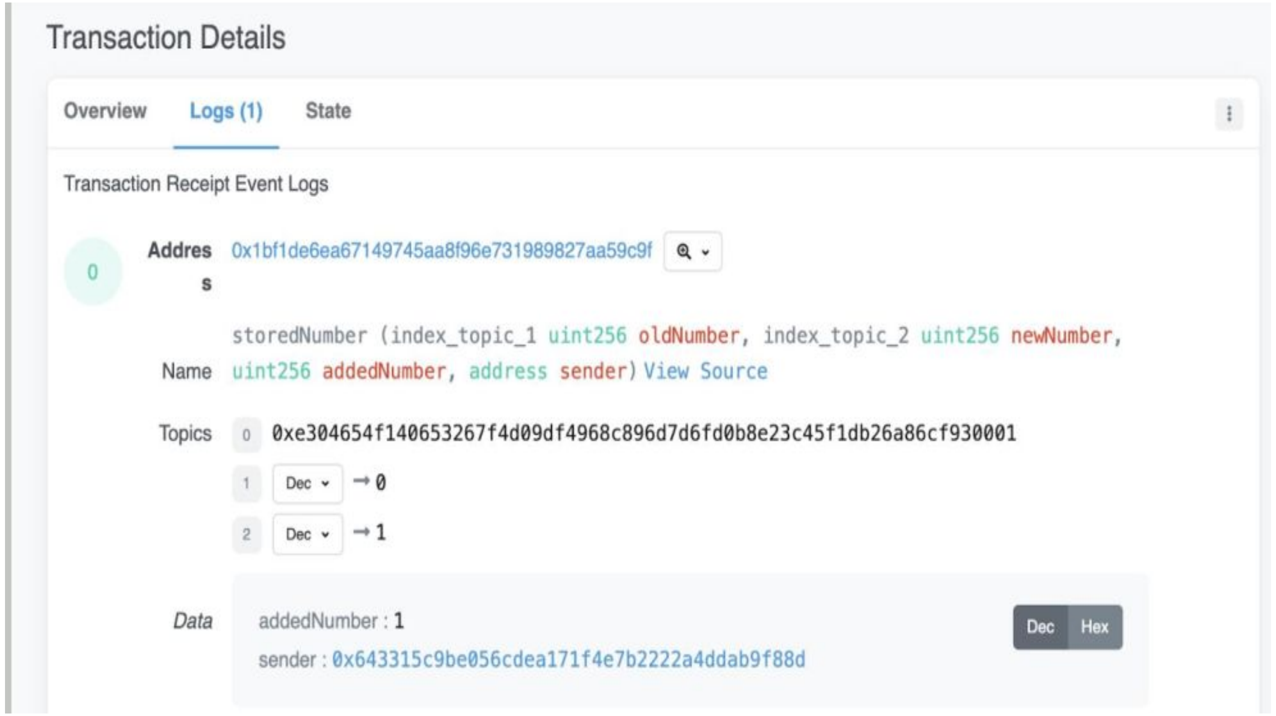
//Declare an Event

event Deposit(address indexed _from, bytes32 indexed _id, uint _value);

//Emit an event emit Deposit(msg.sender, _id,msg.value);

Logging

- The EVM is what makes Ethereum and many other Blockchain tick.
- The EVM has a loging functionality used to “write” data to a structure outside smart contracts.
- Logs are a special data structure on the blockchain
- They cannot be accessed by smart contracts, and gives information about what goes on in the transactions and blocks.



Transaction Details

Overview | Logs (1) | State

Transaction Receipt Event Logs

0

Address 0x1bf1de6ea67149745aa8f96e731989827aa59c9f

Name storedNumber (index_topic_1 uint256 oldNumber, index_topic_2 uint256 newNumber, uint256 addedNumber, address sender) View Source

Topics
0 0xe304654f140653267f4d09df4968c896d7d6fd0b8e23c45f1db26a86cf930001
1 Dec → 0
2 Dec → 1

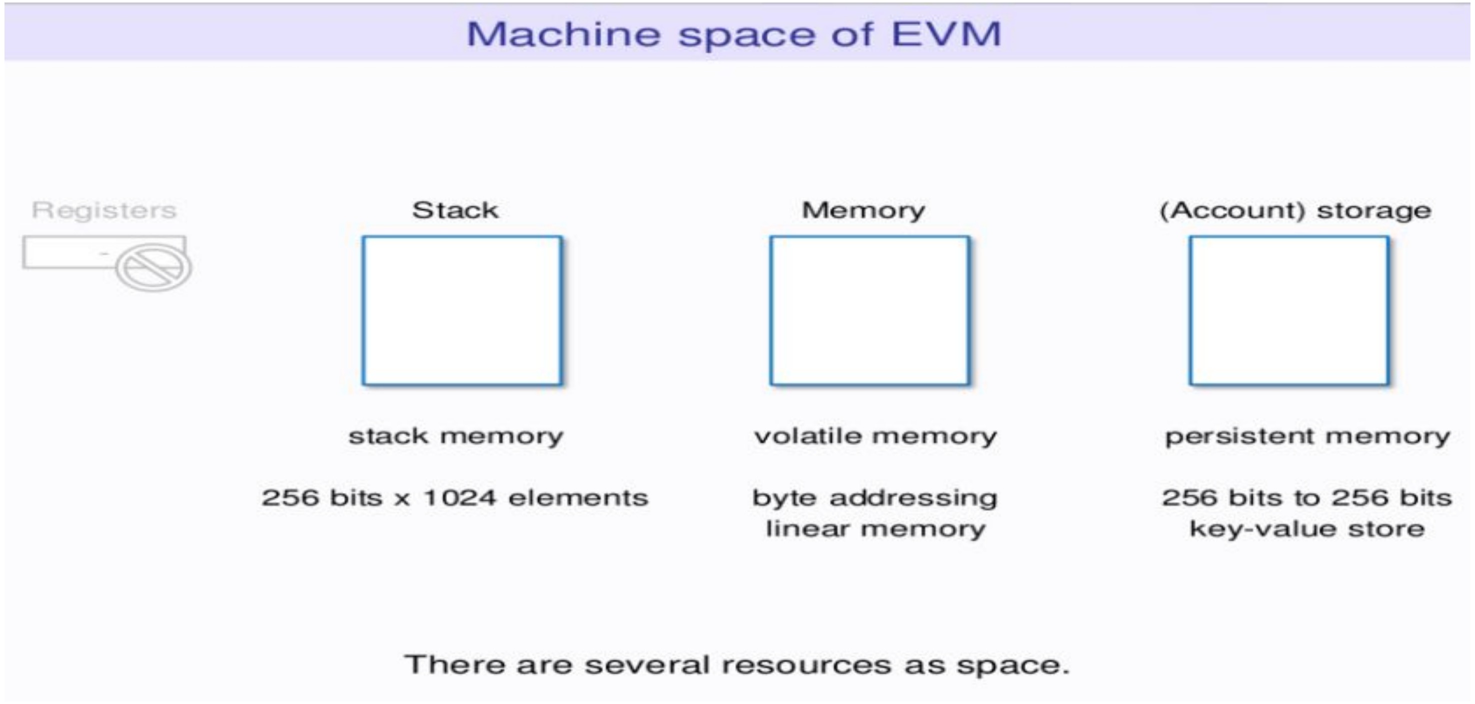
Data
addedNumber : 1
sender : 0x643315c9be056cdea171f4e7b2222a4ddab9f88d

Dec Hex

Transaction Details

- Address: The address of the contract or account the event is emitted from.
- Topics: The indexed parameters of the event.
- Data: The “hashed” non-indexed parameters of the event. Since we know the ABI of the contract (since we verified the contract on Etherscan),
- We can view it in “Dec” or “decoded” mode, or in its raw “hex”, “Hexidecimal”, or “Encoded” mode.

Types of Memory in Solidity



STORAGE

• Storage is a location that stores state variables that exists permanently in the contract.

• Storage is a key-value store that maps 256-bit words to 256-bit words. Storage data is costly(consuming much gas) because it is written in blockchain(like HDD/SDD). Therefore,

• It should be used when it is indispensable such as a money balance.

• Data in storage is written in the blockchain (hence they change the state.)are available between calls and transaction that's why are expensive.

MEMORY

Memory is a byte-array data that exists only during the function call. Every message call starts with a cleared memory.

- Memory is a volatile data and not recorded in the blockchain, so it is much cheaper than storage.
- Memory can be used for every non-permanent data.

STACK

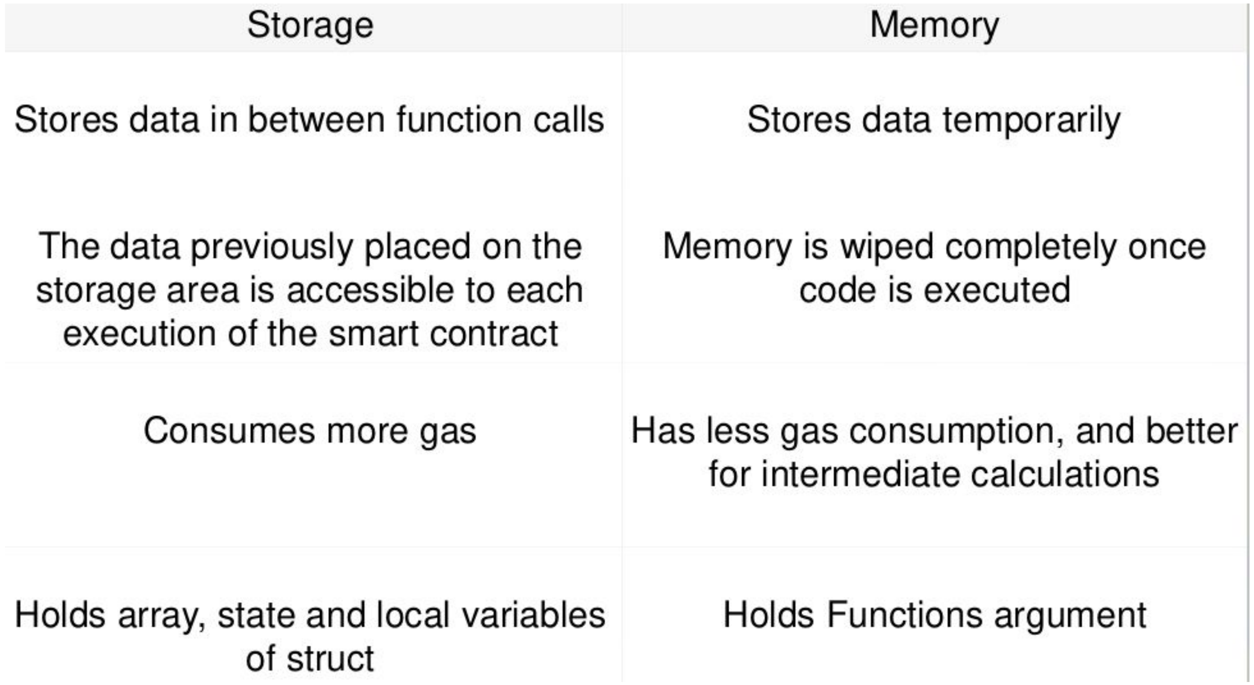
Stack holds small local variables. It has a maximum size of 1024 elements and contains words of 256 bits.

• It means maximum size of stack is 1024 * 256 bits(262,144bits). If you run out of the stack, contract execution will fail.

• EVM provides many opcodes to change stack directly such as POP, PUSH, DUP(duplicate), and SWAP.

• Stack is internal place where temporary variables are stored in 32 bits slots it is usually used for value types

Storage vs Memory



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

References

[1] https://docs.soliditylang.org/en/v0.8.24/

[2] Mastering Blockchain Technology 4th Edition by Imran Bashir, Packt Publications