SCHOOL MATE
Page No.
Date

Introduction :-

→ Solidity is an object-oriented programming language for writing smart contracts on various blockchain platforms.

→ Learning Solidity doesn't mean that you are restricted to only the Ethereum Blockchain; it will serve you well on other Blockchains. Solidity is the primary programming language for writing smart contracts for the Ethereum blockchain.

→ A great way to experiment with Solidity is to use an online IDE called Remix. With remix, you can load the website, start coding and run your first smart contract.

→ Don't worry if you are a beginner and have no idea about how Solidity works, this cheat sheet will give you a quick reference of the keywords, variables, syntax and basics that you must know to get started.

→ Solidity માં Null or nun જેવો કોઈ concept નથી.

Data Types :-

→ Data type is a particular kind of data defined by the values it can take.

Boolean :-

* Logical :-

!	logical negation
&&	AND
||	OR

* Comparisons :-

==	equality
!=	inequality

* Bitwise operators :-

&	AND
|	OR
^	Bitwise XOR
~	Bitwise negation
<<	left shift
>>	Right shift

* Arithmetic Operators :-

+	Addition
-	Subtraction

* multiplication

/ Division

% Modulo

++ increament

-- Decrement

* Relational Operators :-

<= less than or equal to

< less than

== equal to

!= Not equal to

>= Greater than or equal to

> Greater than

* Assignment Operators :-

= Simple Assignment

+= Add Assignment

-= Subtract Assignment

*= Multiply Assignment

/= Divide Assignment

%= modulus Assignment

* Value Types :-

* Boolean :-

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

* Integer :-

→ This data type is used to store integer values. int and uint are used to declare signed and unsigned integers respectively.

* 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.

* Bytes and Strings :-

→ Bytes are used to store a fixed-sized character set while the string is used

to store the characters set equal to or more than a byte. The length of bytes is from 1 to 32, while the string has a dynamic length.

* Enums :-

→ It is used to create user-defined data types, used to assign a name to an integral constant which makes the contract more readable, maintainable and less prone to errors. Options of enums can be represented by unsigned integer values starting from 0.

* Reference Types :-

→ Arrays :-

— An array is a group of variables of the same data type in which variable has a particular location known as its index. By using the index location, the desired variable can be accessed.

— Array can be dynamic or fixed size array.

```
uint [] dynamicSizeArray;

uint [7] fixedSizeArray;
```

* Struct :-

→ Solidity allows users to create and define their own types 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. Also types can be declared using struct.

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

* Mapping :-

→ 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.

mapping (keyType => ValueType)

- keyType :- can be any built-in types plus bytes and string. No reference type or complex objects are allowed.

* Value Type :- can be any type.

* Import Files :-

* Syntax to import the Files.

- import "Filename";
- import * as jsmlogo from "filename";
- or
- import "Filename" as jsmlogo;
- import { jsmlogo1 as alias, jsmlogo2 } from "filename";

* Function Visibility Specifiers :-

function myFunction() <visibility specifiers> returns(bool) {
    return true;
}

* Public :-

→ Visible externally and internally (creates a getter Function for storage/state variables)

* Private :-

→ only visible in the current contract.

* External :-

→ only visible externally (only For Functions) - i.e can only be message - called (via this.Func)

* Internal :-

→ only visible internally.

o Modifiers :-

* Pure :-

→ For Functions : Disallows modification or access of State

* View :-

→ For Functions : Disallows modification of state

* Payable :-

→ For Functions : Allows them to receive Ether together with a call.

* anonymous :-

→ for events : Does not store event signature as topic.

* Indexed :-

→ For event Parameters : Stores the parameter as topic.

* Virtual :-

→ For functions and modifiers: Allows the function's or modifier's behaviour to be changed in derived contracts.

* override :-

→ States that this function, modifier or public state variable changes the behavior of a function or modifier in a base contract.

* Constant :-

→ for state variables : Disallows assignment, does not occupy storage slot.

* Immutable :-

→ for state variables : Allows exactly one assignment at construction time and is constant afterwards is stored in code.

0 Global Variables :-

- block.baseFee (uint)
→ current block's base fee

- block.chainid (uint)
→ current chain id

- block.coinbase (address payable)
→ current block miner's address

- block.difficulty (uint)
→ current block difficulty.

- block.gaslimit (uint)
→ current block gas limit.

- block.number (uint)
→ current block number

- block.timestamp (uint)
→ current block time stamp

- gasleft() return (uint 256)
→ remaining gas

- msg.data (bytes)
→ Complete calldata

- msg.sender (address)
→ sender of the message (current call).

- msg.value (uint)
→ number of wei sent with the message.

- tx.gasprice (uint)
→ gas price of the transaction

- tx.origin (address)
→ Sender of the transaction (full call chain)

- assert (bool condition)
→ abort execution and revert state changes if condition is false (use for internal error)

- require (bool condition)
→ abort execution and revert state changes if condition is false.

- require (bool condition, string memory message)
→ abort execution and revert state changes if condition is false

- revert()
→ abort execution and revert state changes

- revert (string memory message)
→ abort execution and revert state changes providing an explanatory string.

- blockhash (uint blockNumber) returns (bytes32)
→ hash of the given block - only works for 256 most recent blocks

- keccak256 (bytes memory) returns (bytes32)
→ Compute the keccak-256 hash of the input

- sha256 (bytes memory) returns (bytes32)
→ Compute the SHA-256 hash of the input

- ripemd160 (bytes memory) returns (bytes20)
→ Compute the RIPEMD-160 hash of the input

SCHOOL MATE
Page No.
Date

- addmod (uint x, uint y, uint k) returns (uint)
→ abort execution and revert state changes if condition is false.

- mulmod (uint x, uint y, uint k) return (uint)
→ compute (x*y) % k where the multiplication is performed with arbitrary precision & does not wrap around at 2** 256.

- this
→ (current contract's type) : the current contract, explicitly convertible to address or address payable.

- super
→ the contract one level higher in the inheritance hierarchy.

- selfdestruct (address payable recipient)
→ destroy the current contract, sending its funds to the given address.

- <address>.balance (uint256)
→ balance of the Address in wei

- <address>.code (bytes memory)
→ code at the Address (can be empty)

- <address>.codehash (bytes32)
→ the codehash of the Address

- type(C).name (string)
→ the name of the contract

- type(C).creationCode(bytes memory)
→ creation bytecode of the given contract.

- <address payable>.send(uint256 amount) returns (bool)
→ Send given amount of Wei to Address, returns False on failure.

- type(C).runtimeCode(bytes memory)
→ runtime bytecode of the given contract

- type(I).interfaceId(bytes4)
→ Value containing the EIP-165 interface identifier of the given interface.

- type(T).min(T)
→ the minimum value representable by the integer type T.

- type(T).max(T)
→ the maximum value representible by the integer type T.

- abi.decode(bytes memory encodedData, (...)) returns (...)
→ ABI-decodes the provided data. The types are given in parentheses as second argument.

- abi.encode(...) returns (bytes memory)
→ ABI-encodes the given arguments

- abi.encodePacked(...) returns (bytes memory)
→ Performs packed encoding of the given arguments.

SCHOOL MATE

Page No.

Date

- abi.encode(with Selector (bytes4 selector, ...) returns (bytes memory)

→ ABI - encodes the given arguments starting from the second and depends the given four-byte selector

- abi.encodeCall (function functionPointer, (...)) returns (bytes memory)

→ ABI - encodes a call to functionPointer with the arguments found in the tuple, performs a full type-check, ensuring the types match the function signature.

- abi.encodeWithSignature (string memory signature, ...) returns (bytes memory)

→ Equivalent to

- abi.encodeWithSelector (bytes4(keccak256(bytes(signature))), ...)

- bytes.concat (...) returns (bytes memory)

→ Concatenates variable number of arguments to one byte array.

- string.concat (...) returns (string memory)

→ Concatenates variable number of arguments to one string array.

SCHOOL MATE
Page No.
Date

0 Reserved Keywords :-

- after
- alias
- apply
- auto
- byte
- case
- catch of
- default
- define
- final
- implements
- in
- inline
- let
- macro
- match
- mutable
- null
- of
- partial
- promise
- reference
- relocatable
- sealed
- sizeof
- static
- supports
- switch
- typedef
- typeof
- var

↓ uses :-

→ In a computer language, a reserved word (also known as a reserved identifier) is a word that cannot be used as an identifier, such as the name of a variable function, or label - it is "reserved from use".

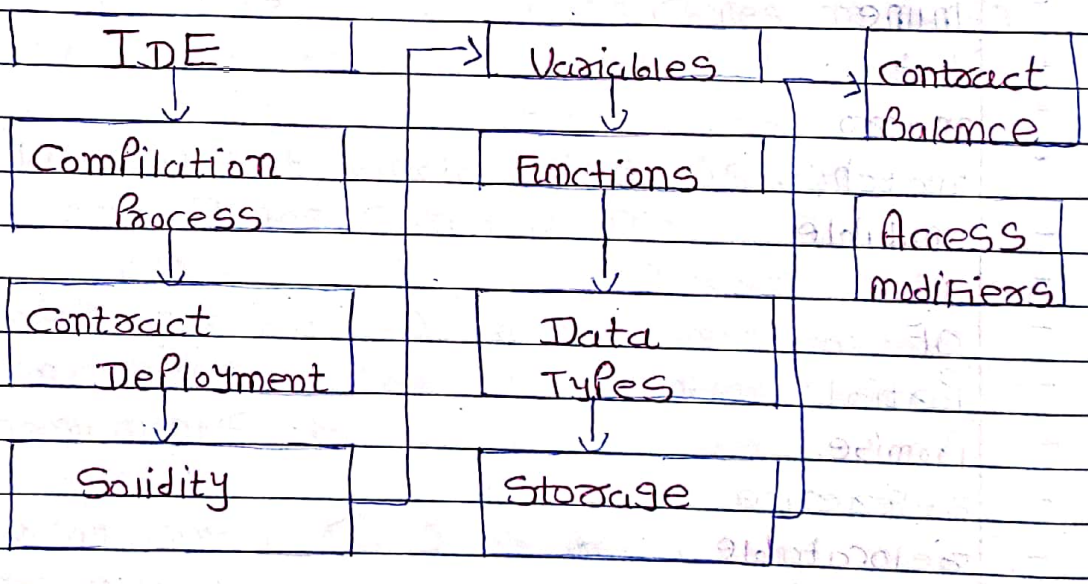
* "Helloworld" Smart contract :-

→ Pragma Solidity >= 0.5.0 < 0.7.0;

Contract Helloworld {
    function get() public pure return(string memory){
        return 'Hello Contracts';
    }
}

→ as you can see in here we're defining a smart contract called hello world.

* Solidity Full Course :-



* Prerequisite :-

- Blockchain
- Ethereum
- Know any Programming Language

SCHOOL MATE
Page No.
Date

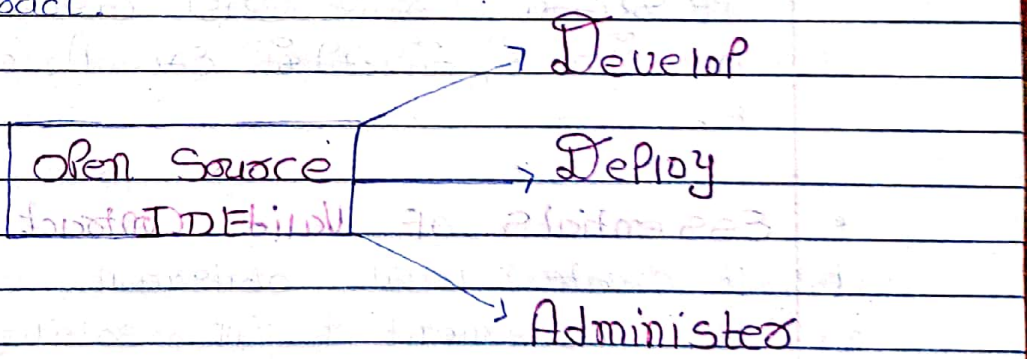
- # Remix IDE :-

→ Remix IDE is generally used to Compile and run Solidity Smart contracts.

→ Solidity online compiler : Summary Remix compiler helps you write, debug, and run Smart Contracts. You can run Remix IDE both online and OFFline. Two Versions are available to use in your browser. One is the latest stable version, and the other is still in the development stage.

→ File extension of Solidity is .Sol

→ Remix IDE is an Open Source IDE. With the help of Remix IDE we can Develop, Deploy, Administer, For Solidity Program as well as Smart contract.



- Some important to know about Remix IDE :-

- Language Support :- Solidity and Vyper
- Written in :- JavaScript
- Modules :- Testing, Debugging, Deploy.

• Smart Contract Compilation :-

→ What is contract?

→ An Agreement which is Enforceable By Law.

→ એસે સભી Agreement જો કી law મે enforceable કરાયે જા સકતે હૈ. જૈસે કિ Against આપ કોર્ટ મે કેસ કર સકતે હૈ કેસ ડાલ સકતે હૈ વો સારે કે સારે Agreement હો ક્યા બન જાતે હૈ? Contract બન જાતે હૈ.

→ Formula :- Agreement + Enforceable by Law

Agreement = OFFER + Acceptance (Consideration)

→ Agreement વો હૈ જહાં પર એક Person offer કરતા હૈ ઔર દૂસરા Person ઉસ offer કો Accept કરતા હૈ ઔર ઉસકે બીચ મેં કુછ Consideration હોતા હૈ, તો ઉસી કો હમ Agreement બોલતે હૈ.

• Essentials of Valid Contract.

1. it create legal obligation
2. An agreement + Enforceability - Contract
3. Only some agreement Forms into a contract
4. All contracts are agreement
5. It is Final, Concaluding and Binding on the Parties.
6. Types :- 1. Valid Contract (legal)
   2. Void contract (inlegal)

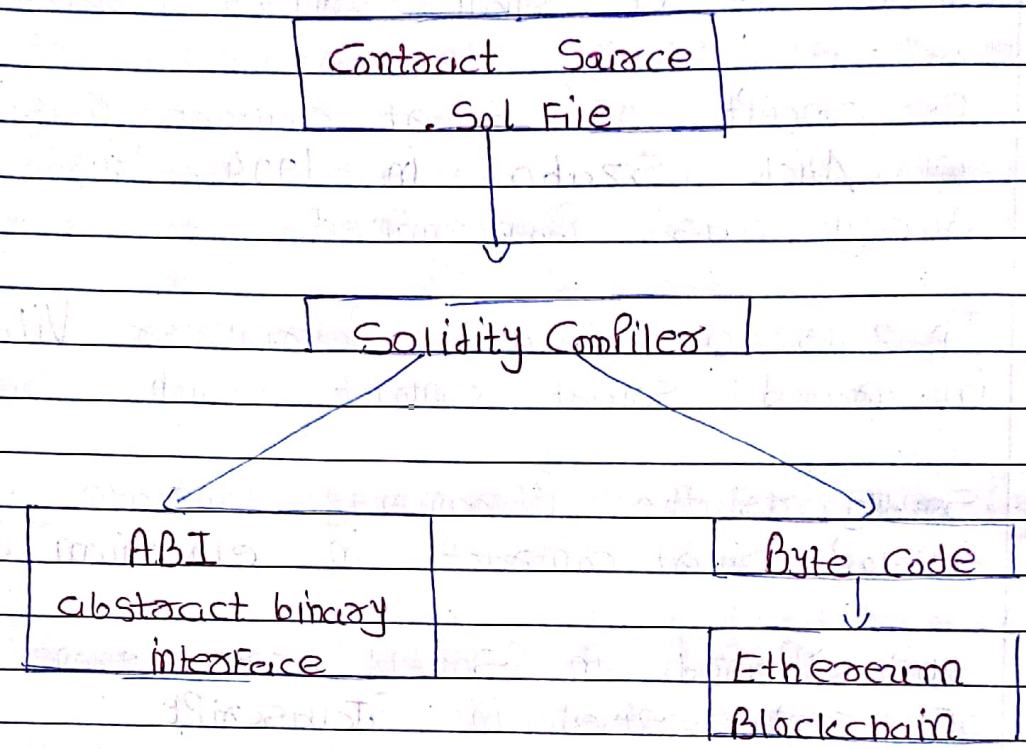
SCHOOL MATE
Page No.
Date

→ What is smart contract?

- Contract is in paper form where as smart contract is a small computer programme.
- The concept of smart contract was given by Nick Szabo in 1994, long before Bitcoin was programmed.
- In 2013, a 19 old programmer Vitalik Buterin integrated smart contract with Blockchain.
- Solidity is the programming language used to develop smart contract in ethereum Blockchain.
- Syntax used in Solidity are same to that of syntax used in Javascript.
- Smart contract is a computer programme where all terms and rules of the contract are specified and stored in Blockchain.
- Decentralised
- Any two strangers can transfer assets through smart contract with no trust issue.
- No trust issue in smart contract.
- No involvement of third party.
- As the smart contract is distributed in open distributed ledger. There is no chance of loss or hacking of smart contract.

SCHOOL MATE
Page No.
Date

SCC :-



→ When the contract source .sol file is generated then Solidity compiler convert that into 2 parts 1. ABI(C abstract binary interface) and 2. Byte code

→ if two smart contract want to communicate at that time abi is acquired like function call, Request of the Data etc. basically its a kind of bridge where any application or smart contracts can intract.

→ ethereum blockchain pe pura ka pura smart contract display nahi hota hai usme size uska jo particular byte code hota hai wahi generate hota hai ethereum blockchain ke nodes pe.

→ abi and byte code copy karke online (bytecode to opcode) pe put karo.

→ abi is fully formated is Json.

— Bytecode to opcode Disassembler.
— ethereum Opcodes :- cryptic/evm-opcodes : Ethereum opcodes and ... - GitHub.

* Some Important Points to note :-

- Contract bytecode is public in readable form.
- Contract doesn't have to be Public.
- Bytecode is immutable
- ABI act as a bridge between application and smart contract.
- ABI and Bytecode cannot be generated without source code.
- Opcodes :- are just like instructions that are given to EVM.

* Mainnet VS Testnet :-

— Used for actual transactions with value	— Used for testing smart contracts and decentralized applications.
— Mainnet's network ID is 1.	— Testnets have network IDs of 3, 4, and 42.
— example :- Ethereum	— Examples :- Rinkeby Test Network

SCHOOL MATE
Page No.
Date

* Metamask :- (Digital wallet)

→ It is a very Popular crypto wallet and it's Provide gateway to the all Decentralized application.

* Function/uses :-
- Store & thers
- Send Ether
- Receive Ether
- Run Dapps
- Swap Tokens

* Testnet Faucets :-

o ETH on testnets has no real value ; these fore we get testnet ETH from faucets.
Example - Rinkeby faucet, Ropsten faucet etc.

* Free Ethers :- Rinkeby Authenticated Faucet (Twitter Account)

* Contract Deployment :-

— Environment :- Javascript VM (Berlin)
Javascript VM (London)
Injected web3
web3 Provider

* JavaScript Virtual Machine :-

- Transaction will be executed in a sandbox.
- Own memory blockchain.
- Ideal for testing.

* Injected Web3 :-

— Deploy a contract or run a transaction on Ethereum main or test net.

* Web3 Provider :-

— Connect to a remote node and Ethereum client.

* Solidity :-

- High-level statically typed programming language.
- With Solidity you can create contracts for uses such as Voting, CrowdFunding, blind auctions, and multi-signature wallets.
- Case sensitive.
- For latest update - Visit Solidity documentation.

> Note :- You should follow established development best-practies when writing your smart contracts.

* Simple Program :-

— Identity.sol :-

```
Pragma solidity >0.5.0 <0.9.0;
```

SCHOOL MATE
Page No.
Date

```
Contract Identity
{
    string name;
    uint age;

  constructor() public
{
    name = "Akshay";
    age = 17;
}
  function getName() view public returns
                                (string memory)
{
   return name;
}
  function getAge() view public returns
                                (string memory)
{
   return age;
}
  function setAge() public
{
   age = age + 1;
}
}
```

SCHOOL MATE
Page No.
Date

* State Variables :-

→ State Variables are variables whose values are Permanently stored in contract storage. it uses contract storage, we have to pay the gas Fees to use state Variables.

→ It is compile Time.

- To change the default values of the state variable.

Using the Contracts Constructor

Initializing the variable at declaration

Using the Setter function

- Permanently stored in contract storage.
- cost gas (expensive).
- Storage not dynamically allocated.
- Instance of the contract cannot have other state variables besides those already declared.

```
Pragma Solidity ^0.5.0;

Contract Sample
{
```

Public

Public वाला automatically get function create होता है।

```
    uint age; // State Variable
}
2.
Pragma Solidity >=0.5.0 <0.9.0;

Contract State
{
  uint Public age; // State Variable

  Constructor() Public
  {
    age=10; // Using State Variable
  }
}

3.

Pragma solidity >=0.5.0 <0.9.0;

Contract State
{
  uint Public age; // State Variable

  function setAge() Public
  {
    age=10;
  }
}
```

* Local Variables :-

→ Local Variables જ્યારે function ની body ની અંદર Declare હોય છે ત્યારે તેને કહેવાય છે Local Variables. એ stack માં જ store હોય છે, નહીં કે contract storage માં.

→ Variables whose values are available only within a function where it is defined. Function parameters are always local to that function.

- string datatype storage માં use હોય છે. contract level પેહી declare હોતા છે?

- with the help of memory keyword we can use the string.

* Ex:- 1 :-

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract local
{
function store() pure returns(uint)
{
    uint age= 10;
    return age;
}
}
```

* Ex:- 2 :-

```
pragma solidity >=0.5.0 <0.9.0;

contract local
{
function store() pure public returns(uint)
{
```

string memory Name = "Akshay";
uint age = 11;
return age;
}
}

Ex:- 3 :-

- Pragma Solidity ^0.5.0;

Contract SolidityTest {
uint storedData; // State Variable
constructor() public {
storedData = 10;
}
function getResult() public view returns (uint) {
uint a = 1; // Local Variable
uint b = 2;
uint result = a+b;
return result; // access the local variable
}
}

→ output :-

0: uint256: 10

- Declared inside functions and are kept on the stack, not on storage,
- Don't cost gas.
- These are some types that reference the storage by default.

- Memory keyword can't be used at contract level.

- Pragma Solidity >=0.5.0 <0.9.0;

Contract local
{
    String memory name = "Akshay"; //State Variable
    function Store() Pure Public returns (uint)
    {
        uint age = 11; //Local Variable
        return age;
    }
}

→ It will give the error.

* Functions - Setters and Getters :-

→ 1. When you call a setter function it creates a transaction that needs to be mined and costs gas because it changes the blockchain. Vice versa for getter function.

2. When you declare a Public State Variable a getter function is automatically created.

3. By default Variable Visibility is Private.

Ex:-

```
Pragma Solidity >= 0.5.0 <0.9.0;

Contract Local
{
   uint age = 10;

  function getter() Public View returns(uint)
{
   return age;
}
  function setter (uint newage) public
{
   age= newage;
}

}
```

* Important :- If we create Public State Variable than no need to make getter function (i.e :- uint Public age=10;)

* View vs Pure :-

→ View માં આપણ read કરી શકતે કે State variable કો.

→ Pure માં આપણ બાતો read કરી શકતે કે બાકી write કરી શકતે કે આપણ State Variable કો.

SCHOOL MATE
Page No.
Date

Constructor :-

→ It will execute only one time whenever we create our contract.
→ State variable initialization. } use.
→ Decide owner of a contract }
→ It is a key word.
→ Executed only once.
→ You can create only one constructor and that is optional.
→ A default constructor is created by the compiler if there is no explicitly defined constructor.

Ex :-

```
Pragma Solidity >= 0.5.0 < 0.9.0;

Contract local
{
   uint Public count;

Constructor () Public
{
      count = 8;
}
}
```

Ex :- Constructer with argument

```
Pragma Solidity >= 0.5.0 < 0.9.0;

Contract local
{
   uint Public count;
```

```
Constructor (uint new_count)
{
    count = new_count;
}
}
```

* **Integers :-**

int	uint

signed and unsigned integers can be of various sizes.

int8 to int256	uint8 to uint256
int alias to int256	uint alias to uint 256

By default an int is initialized to 0.

Overflow get detected at compile time

Range	
int8 : -128 to +127	uint8 : 0 to 255
int16 : -32768 to +32767	uint16 : 0 to 65535
$-2^{(n-1)}$ to $2^{(n-1)}-1$	0 to $2^{(n)}-1$

SCHOOL MATE
Page No.
Date

Operators :-

- Comparisons : <=, <, ==, !=, >=, > (evaluate to bool)
- Bit operators : &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
- Arithmetic operators : +, unary -, unary +, *, /, % (remainder), ** (exponentiation), << (left shift), >> (right shift)

→ Artical :- New batch Overflow Bug.

Integer Overflow.

* Arrays :-

→ Fixed size Array

Dynamic Size Array

* Fixed Size Array :-

Ex :-

```
Pragma Solidity >= 0.5.0 < 0.9.0;
Contract Array
{
 uint[4] Public arr = [10, 20, 30, 40];
}
```

Exc:-2

```
Pragma solidity >=0.5.0 <0.9.0;

Contract Array
{
  uint [4] Public arr = [10,20,30,40];

function Setter (uint index, uint Value)
                                  Public
{
   arr [index] = Value;
}
 function length () Public view returns (uint)
{
   return arr.length;
}
}
```

* Dynamic Size Array:-

→

```
Pragma solidity >=0.5.0 <0.9.0;

Contract Array
{
  uint [] Public arr;

function
name  function PushElement (uint item) Public
{
   arr.Push (item);
}

}
```

→ pragma solidity >=0.5.0 <0.9.0;

contract Array
{
uint[] public arr;

function pushElement(uint item) public
{
arr.push(item);
}
function length() public view returns(uint)
{
return arr.length;
}
function PopElement() public
{
arr.pop();
}
}

→ arr.pop() = it will remove the last element from the array.

- **Byte Arrays :-**
- 1byte = 8bits
- 1 hexadecimal digit = 4 bits
- Everythings that will be stored in the byte array will in the form of hexadecimal digits.

Ex:- pragma solidity >=0.5.0<0.9.0;

contract Array
{

bytes3 Public b3; // 3 bytes array

bytes2 Public b2; // 2 bytes array

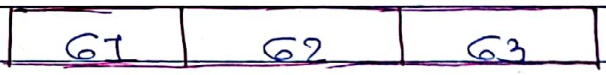
}

Ex:- Pragma Solidity >=0.5.0<0.9.0;

Contract Array
{
bytes3 Public b3; // 3 bytes array
bytes2 Public b2; // 2 bytes array

function setter() Public
{
b3='abc';
b2='ab';
}
}

→ bytes3 Public b1='abc'



- Byte arrays cannot be modified.
- Padding of 0 is added at the end if the value (by which the array is initialized) does not occupy the entire space.

* bytes [] arrays :-

→ Pragma solidity >=0.5.0 <0.9.0;

Contract Array
{
  bytes Public bt = "abc";
Function PushElement () Public
{
  bt.push('d');
}
function getElement (uint i) Public view returns (bytes1)
{
  return bt[i];
}
function getlength () Public view returns (uint)
{
  return bt.length;
}
}

* Loops :-

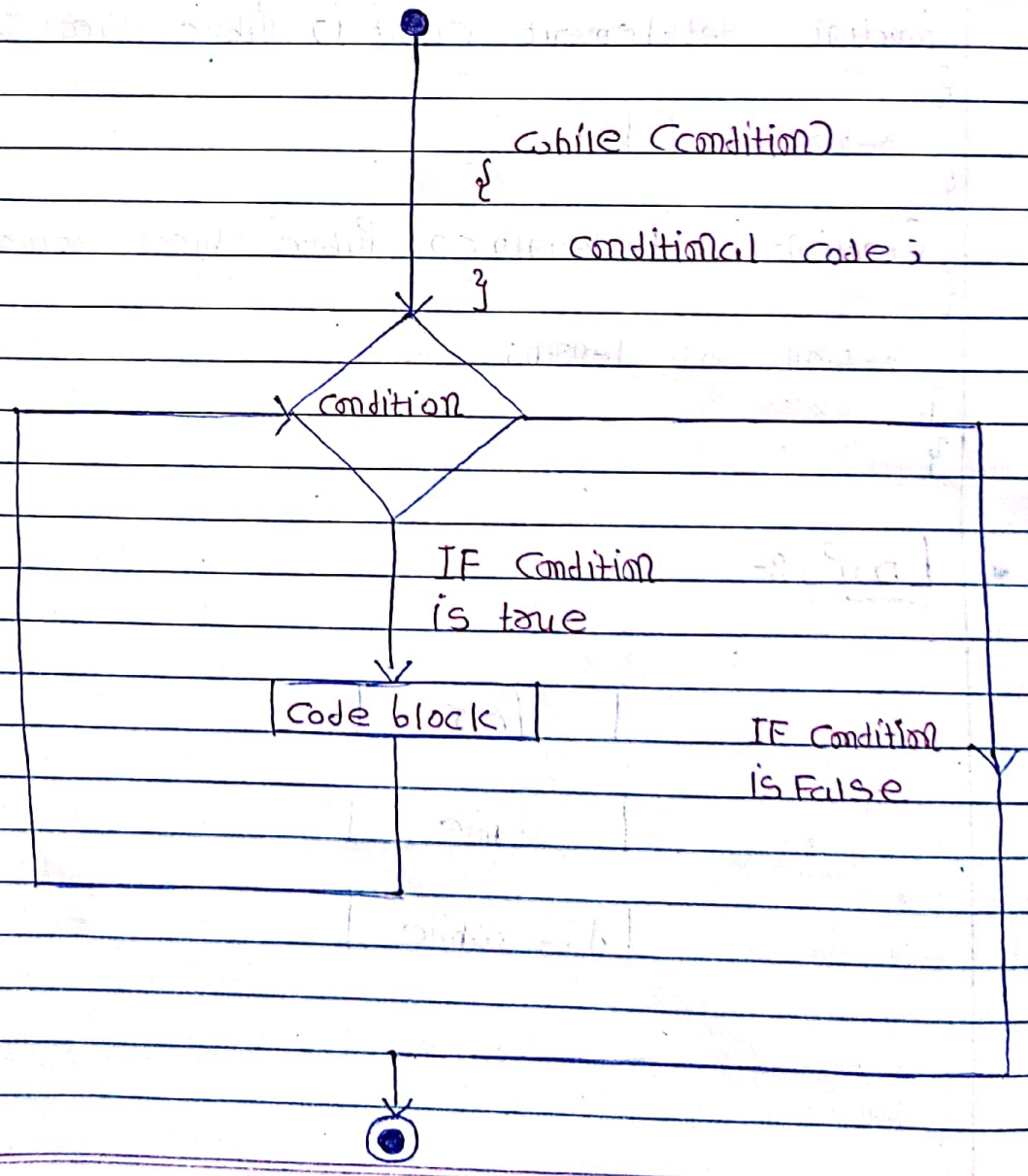
- for
- while
- do-while

* **While-loop :-**

→ The purpose of a while loop is to execute a statement or code block repeatedly as long as an expression is true.

→ Once the expression becomes false, the loop terminates.

* **Flow chart :-**



SCHOOL MATE
Page No.
Date

* Syntax :-

→ while (expression) {

Statement(s) to be executed if expression is true

}

• Ex :-

Pragma Solidity >=0.5.0 < 0.9.0

Contract Array
{
uint [3] Public arr;
uint Public count;

function loop() Public {
while (count < arr.length)
{
arr[count] = count;
count ++;
}
}
}

* for loop :-

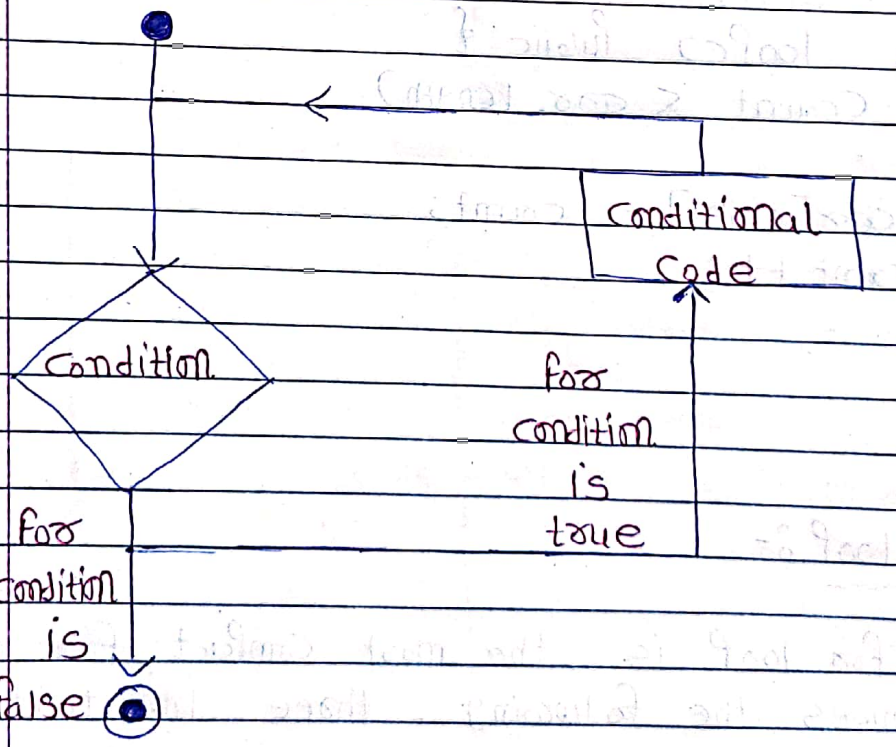
→ The for loop is the most compact form of looping. It includes the following three important parts-

• The loop initialization where we initialize our counter to a ~~string~~ starting value. The initialization statement is executed before the loop begins.

- The test statement which will test if a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.
- The Iteration statement where you can increase or decrease your counter.
- You can put all three parts in a single line separated by semicolons.

Flow chart :-

- The flow chart of a for loop in solidity would be as follows :-



SCHOOL MATE
Page No.
Date

* Syntax :-

- for (initialization ; test condition ; iteration statement)
{
Statement(s) to be executed if test condition is true
}

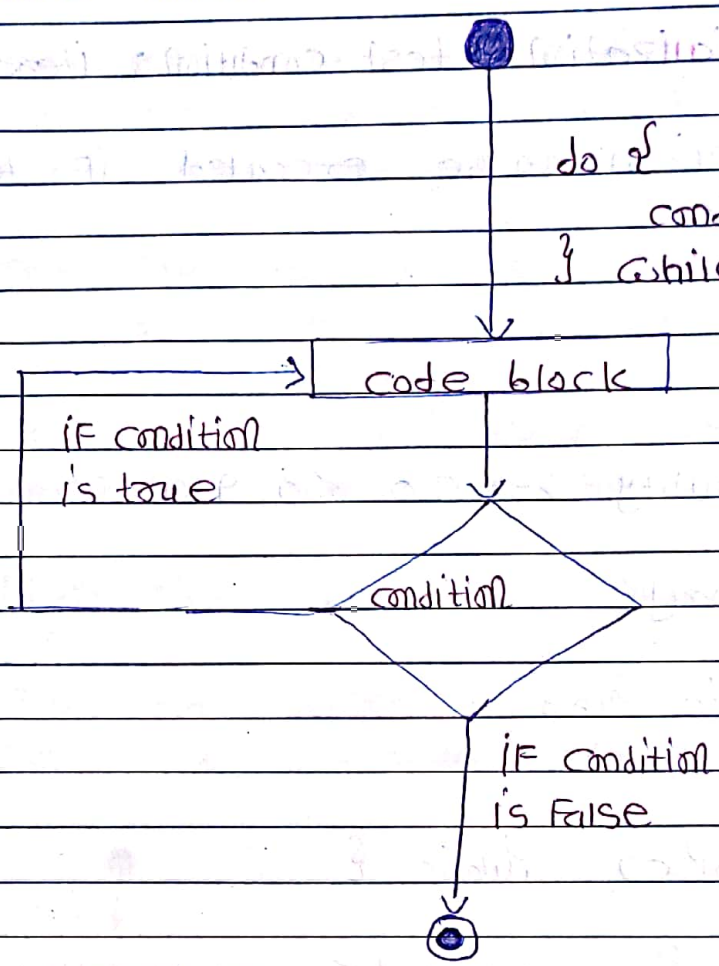
Ex :-

```
Pragma Solidity >=0.5.0 <0.9.0;

Contract Array
{
uint [3] Public arr;
uint Public count;

function loop() Public {

for (uint i= count; i< arr.length; i++)
{
    arr[count] = count;
    count ++;
}
}
}
```

* Do - while :-

→ The do...while loop is similar to the while loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once even if the condition is false.

* Flow chart :-



* Syntax :-

—> do {

Statement (s) to be executed ;

} while (expression) ;

* Ex :-

Pragma Solidity >=0.5.0 <0.9.0 ;

Contract Array

{

```
uint [3] Public arr;
uint Public count;

function loop() Public {
do {
    arr [count] = count;
    count ++;
} while (count < arr.length);
}
```

* **IF-Else :-**

→ The 'if-else' statement is the next form of control statement that allows Solidity to execute statements in a more controlled way.

* **Syntax :-**

```
if (expression) {
statement (s) to be executed if expression is
  true
}
else {
statement (s) to be executed if expression is
false
}
```

Ex:-

```
Pragma solidity >= 0.5.0 < 0.9.0;

Contract Array
{
 function check (int a) Public Pure returns
                        (string memory)
 {
    string memory Value;

   IF (a>0)
 {
    Value = "greater than zero";

 }
  else IF (a==0)
 {
   Value = "equal to zero";

 }
   else
 {
    Value = "less than zero";
 }
  return Value;

  }
}
```

SCHOOL MATE
Page No.
Date

* Booleans :-

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

- Ex :-

```
Pragma solidity >=0.5.0 <0.9.0;

Contract Array
{
  bool Public value = true;
}
```

- Ex :-

```
Pragma solidity >=0.5.0 <0.9.0;

contract Array
{
bool Public value = true;

function check (uint a) Public return s (bool)
{
  if(a >100)
{
      value = true;
      return value;
}
 else
{
    value = false;
      return value;
}
}
```

- bool :- The possible values are constants true and false.

* Operators :-

- ! (logical negation)
- && (logical conjunction, "and")
- || (logical disjunction, "or")
- == (equality)
- != (inequality)

* **Struct :-**

→ Struct types are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title → ઇસ structure કો construct storage કે અંદર ભી જાના ચાહતે હૈં ઔર બાહર ભી જાના ચાહતે હૈં.
- Author
- Subject
- Book ID

* **Defining a Struct :-**

– To define a struct, you must use the struct keyword. The struct keyword defines new data type, with more than one member.

SCHOOL MATE
Page No.
Date

```
Struct Struct_name {

    type1 type_name_1;
    type2 type_name_2;
    type3 type_name_3;

}
```

* Example :-

```
Struct Book {

    String title;
    String author;
    String subject;
    uint book_id;
}
```

* Example :-

```
Pragma solidity ^0.5.0;

contract test {
    struct Book {
        string title;
        string author;
        uint book_id;
    }
    Book book;

    function setBook() public {

        book= Book('Learn Java', 'TP', 1);
    }
```

```
function getBookId() Public View returns(uint) {
    return book.book_id;
  }
}
```

- Example :-

```
Pragma solidity >=0.5.0 <0.9.0

Struct Student {

    uint roll;
    String name;
}

Contract Demo
{
    Student Public S1;

  Constructor(uint _roll, String memory _name)
{
    S1.roll = _roll;
    S1.name = _name;
}
  function change(uint _roll, String memory name)
{                                        Public
  Student memory new_Student = Student({

  roll: _roll;
  name: _name
  });
```

```
S1= new student;
}
}
```

* Enums :-

→ Enums restrict a variable to have one of only a few predefined values. 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.

→ For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or, large.

* Example :-

→
```
Pragma solidity ^0.5.0;

contract test {
    enum FreshJuiceSize { SMALL, MEDIUM, LARGE }

    FreshJuiceSize choice;
    FreshJuiceSize constant defaultchoice = FreshJuice.
                                            MEDIUM;

    function setLarge() public {
        choice = FreshJuiceSize.LARGE;
```

```
}
function getchoice() public view returns
                        (FreshJuiceSize){
    return choice;
}
function getDefaultchoice() public pure
                        return(uint){
    return uint(defaultchoice);
    }
}
```

* output :-

uint8 : 2

— Click getDefaultchoice Button to get the default choice.

* output :-

uint256 : 1

* Example :-

— Pragma solidity >=0.5.0 <0.9.0;

```
contract State
{
    enum user { allowed, not_allowed, wait }

    user public u1 = user.allowed;
    uint public lottery = 1000;
    function owner() public {
```

```
    if (x == user.allowed)
    {
        letter = 0;
    }
}

function changeOwner() public {
    x = user.not allowed;
}
}
```

→ Enum used in Readability of code & Global.

* **Mapping** :- (uint or string datatype)

→ concept of keys and values.
→ mapping (key => value).

→ Mapping is a reference type as arrays and structs.

* **Syntax**:-

→ mapping (_keyType => _valueType)

• Where
– _keyType - can be any built-in types plus bytes and string. No reference type or complex objects are allowed.

– _valueType - can be any type.

SCHOOL MATE
Page No.
Date

* Mapping :- (uint or Structure)

roll		name	class
0	→	Ravi	8
5	→	Akash	9
100	→	Rita	11

* The key cannot be types mapping, dynamic array, enum and Struct,

* The values can be of any type.

* Mappings are always Stored in storage irrespective of whether they are declared in contract storage or not.

* Ex :-

```
Pragma Solidity >=0.5.0 <0.9.0;

Contract demo
{
   Struct student {
        String name;
        uint class;
   }
   mapping (uint => student) Public data;

   function Setter (uint roll, string memory name,
        uint class) Public
   {
      data[roll] = Student (name, class);
   }
}
```

Storage VS Memory :-

Storage	Memory
1. Holds, State Variables	1. Holds local Variables defined inside functions if they are reference types.
2. Persistent	2. Not Persistent.
3. Costly	3. Cheap
4. Like a computer HDD	4. Like a computer RAM

* Example :-

```
Pragma Solidity >= 0.5.0 < 0.9.0;

Contract demo
{
  string [] Public Student = ['Ravi', 'Rita', Aman'];

  function memco Public View
  {
    String [] memory S1 = Student;
    S1 [0] = 'Akash';
  }
  function sto co Public
  {
    String [] Storage S1 = Student;
    S1 [0] = 'Akash';
  }
}
```

SCHOOL MATE
Page No.
Date

* Global Variable :-

→ Special variables exists in the global name space used to get information about the blockchain.

* Example :-

```
Pragma Solidity >=0.5.0 <0.9.0;

Contract demo
{
    function getter() Public View returns (uint
                block_no, uint timestamp, address
                                msg.Sender)
    {
        return (block.number, block.timestamp, msg.
                                Sender);
    }
}
```

→ Output :-

| getter |

0 : uint256 : block no : 7

1 : uint256 : timestamp 1627580751

2 : address : msg.Sender Account के आईडी का Code

* Contract Balance :-

- Payable keyword :-

- A Payable function is a function that can receive either being called. It is mandatory to include the Payable keyword if you wish your function to receive ethers. If you try to send ethers to a function that is not mentioned as Payable, the transaction will be rejected and fail.

→ It's mandatory to include the Payable keyword from Solidity 0.4x. If you try to send ether using call; as follows:-

→ The function below, taken from the Blind Auction contract in the Solidity docs is a good example of the Payable keyword.

* Ex:-

```
function bid( bytes32 blindedBid)
    Rublic
    Payable
    onlyBefore (biddingEnd)
{
    bids [msg.Sender].Push (Bid({
        blindedBid : blindedBid,
        deposit : msg.value
    }));
}
```

SCHOOL MATE
Page No.
Date

* Ex:-

→ Pragma Solidity >=0.5.0<0.9.0;

Contract Pay
{
  function PayEther() Public Payable
{
}
  function getBalance() Public View returns (uint)
{
  return address(this).balance;
}
}

* Ex:-

→ Pragma Solidity >=0.5.0<0.9.0;

Contract Pay
{
  address Payable User = Payable (Account address);
  function PayEther() Public Payable
{

}
  function getbalance() Public View returns (uint)
{
  return address(this).balance;
}
  function sendEther Account() Public
{
  User.transfer (1 ether);
}
}

SCHOOL MATE
Page No.
Date

* Visibility :-

	Public	Private	internal	external
	outside	X	X	outside
	within	within	within	X
Inheritance →	Derived	X	Derived	Derived
	other	X	X	other

* Public :- Visible everywhere (within the contract itself and other contracts or addresses).

→ Therefore, it is part of the contract interface (ABI). It can be called internally or via messages.

* Private :- Visible only by the contract it is defined in, not derived contracts.

→ These functions are not part of the contract interface (ABI).

* Internal :- Visible by the contract itself and contracts deriving from it.

→ Those functions can be accessed internally, without using this. Moreover, they are not part of the contract interface (ABI).

* External :- Visible only by external Contracts / addresses.

→ Like Public functions, external functions are part of the contract interface (ABI).

→ However, they can't be called internally by a function within the contract. for instance, calling fc() does not work, but call this.fc() works.

→ The visibility specifier is given after the the type for state variables and between parameter list and return parameter list for functions.