Smart Contracts:

Overview:

- **What are Smart Contracts?**
- **Solidity.**
- **Remix IDE.**
- **Development and deployment of smart contracts.**

Smart Contracts:

- **Smart contracts are computer programs that act as agreements where the terms of the agreement can be pre programmed with the ability to be executed and enforced.**

- **Ethereum supports smart contracts and it is run in Ethereum Virtual Machine (EVM).**

- **Ethereum smart contract languages:-**
  - ✔ **LLL (Based on LISP).**
  - ✔ **Serpent (Based on Python).**
  - ✔ **Solidity (Influenced by C++, Python, and Java-Script).**

Solidity:

- **Solidity is a statically-typed, contract-oriented, high-level language for implementing smart contracts.**

- **It was influenced by C++. Python, and Java Script and it is designed to target the Ethereum virtual machine (EVM).**

- **Remix IDE (Integrated Development Environment) is a web application that can be used to write, debug, and deploy Ethereum Smart Contracts (under solidity language).**

Cont…

- The smart contracts can be developed locally on a computer with the help of a simple text editor and a solidity compiler.

- To do this, SOLC compiler is required which is installed on computer.

- The SOLC compiler is installed by using the "Docker (or) Node.js Package Manager (NPM).

- The alternative way to write the code is "REMIX IDE" where software is not required to install on computer.

Cont…

Layout of a contract:

```
pragma solidity ^ 0.5.10;

Contract demo
{

//this is a single-line comment.

/* this is
   a multiline
   comment  */

}
```

pragma tells the compiler which version are used to write and compile the contract (code).

Define the contract with the **contract** keyword. (demo is contract name)

Comments in solidity

Data types in solidity:

Integer:-

Declaration:-

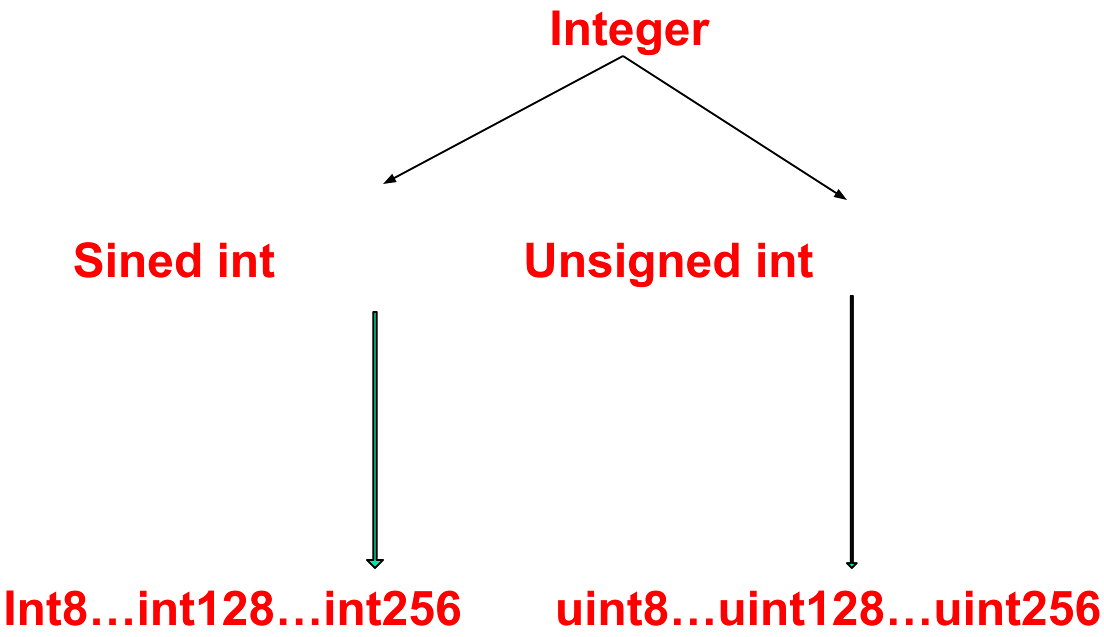
```
uint256 variable_name = value;
```

Example:-

```
uint256 marks = 39;
```

Note: uint256 and uint are same.

Cont…



Integer

Sined int

Unsigned int

Int8…int128…int256

uint8…uint128…uint256

Cont…

Boolean:-

Declaration:-

```
        bool variable_name = true / false;
```

Example:-

```
        bool pass = true;
```

Note: logical operator always return Boolean values.

Cont…

Address:-

- It holds a 20 bytes value (size of an Ethereum address).

- Address types also have members and serves as a base for all contracts.

Declaration:-

```
address variable_name = value;
```

Example:-

```
address home = 0x72ba7d8e73fe8e2;
```

Cont…

Important members of address:-

- address.balance(); → check the balance of an account .
- address.send (uint) returns (bool); → Use to transfer money.
- address.transfer (uint amount); → Use to transfer money

Note:-Address is the very specific feature of solidity. It specify the address of every component that is residing on the blockchain.

Cont…

String:-

Declaration:-

```
string variable_name = value;
```

Example:-

```
string name = “IIITNR”;
```

Data structures in solidity:

Array:-

Static array:-

```
data_type [array_size] array_name;

eg. uint [32] demo;
```

dynamic array:-

```
data_type [ ] array_name;

eg. uint [ ] demo;
```

Data structures in solidity:

struct:-

```
srtuct struct_name
{
type type_name
……
……
}
```

Members are accessed via (.) operator.

Data structures in solidity:

struct:-

```
Pragma solidity ^ 0.5.10;
Contract game
{
        struct player
        {
        string name;
        Unit age;
        Uint stats;
        }
Player captain;
Captain.name = “aarya”;
Captain.age = 22;
Captain.stats = 88;
}
```

Data structures in solidity:

mapping:-

mapping can be seen as hash table which are virtually initialized such that every possible key exists and is mapped to a value.

Declaration:-

```
      Mapping (_keytype => _valuetype)
```

Example:-

```
      Mapping (address => uint) public records
```

Note:-_keytype can be almost ant type except a mapping, a dynamically sized array, a contract and a struct.

_valuetype can actually be any type including mapping.

Functions :-

Functions are generally used for code reusability.

Declaration:-

Function function_name (<parameter types>) <visibility_specifier> <function modifier> returns (<return type>).

Example:-

Function sum (uint a, uint b) public view returns (uint)

Visibility Specifier :-

- **Public:- anyone can access (inside from contract or from outside the contract).**

- **External:- can not access internally (only externally accessible).**

- **Private:- can be accessed only form this contract (also not visible when we deploy it).**

- **Internal:- only from this contract and contracts deriving from it can access. (also accessible form another functions).**

Function modifier:-

- **Pure:- can not read (or) modify the state of function.**

- **View:- can not modify state of function.**

- **Constant:- only for the state variable.**

- **Payable:- Allow functions to receive ether.**

Variables in solidity:-

- **State variables:- Permanently stored in contract storage.**

- **Local variables:- till the function is executing.**

Global variables:- Used to get information about the Blockchain.

- ✔ **now (uint): Current block timestamp.**
- ✔ **msg.value (uint): Number of wei sent with the message**
- ✔ **msg.sender (address payable): Address of the caller who invoked the transaction.**

Operators in solidity:-

- ✔ **Arithmetic Operators:** +, -, *, /, %.
- ✔ **Comparison Operators:** ==, >, <, >=, <=.
- ✔ **Logical Operators:** &&, ||, !.
- ✔ **Bitwise Operators:** &, |, ^, <<, >>.
- ✔ **Assignment Operators:** =, +=, -=.
- ✔ **Conditional Operator:** ? : .

Print “string” in solidity.

```
pragma solidity ^0.5.10;
contract printstring{
  string a="welcome to learn the solidity language";
    function display() public view returns (string memory )
    {

    return a;
    }
}
```

Here data location is memory. for string, array and structures, we have to specify where it is stored either in storage or call data or in memory.

For uint data, data location is not required.

output:

```
display
0: string: welcome to learn the solidity language
```

Use of “bytes” in solidity:-

Bytes provide the hexadecimal value of a string (or) integer.

Range:- bytes1 to bytes32.

```
pragma solidity ^0.5.10;
contract usebytes{
    bytes32 a ="Hello";
    function display() public view returns (bytes32 )
    {
    return a;
    }
}
output:
display
0: bytes32:
0x48656c6c6f000000000000000000000000000000000000000000000000
000
```

Develop a smart contract for simple calculator where all inputs are at run time. (return multiple values)

```
 pragma solidity ^0.5.10;
contract run_time_input{
    uint public c=0;
    uint public d=0;
    uint public e=0;
    uint public f=0;

    function add(uint a, uint b)
public
    {
        c=a+b;
    }

    function sub(uint a, uint b)
public
    {
        d=a-b;
    }
```

```
Pragma  function mul(uint a, uint b)
public
    {
        e=a*b;
    }

    function div(uint a, uint b) public
    {
        f=a/b;
    }

    function display1() public view
returns(uint,uint,uint,uint ){
        return (c,d,e,f);

    }

}
```

Return single and multiple values in a single statement:-

Single value:-

```
function display() public view returns(uint ){
        return (c);


   }
```

Multiple values:-

```
function display1() public view returns(uint,uint,uint,uint ){
        return (c,d,e,f);


   }
```

Use of view and pure :-

- ***view:- used when value is not changing (we just return the value).***

- ***Pure:-used when function is using its own variable to change the value.***

```
pragma solidity ^0.5.10;
contract fun{
   uint a=100;
function display() public view returns(uint)
{
 return a;
}
function show(uint b, uint c) public pure returns(uint)
  {
uint d=b*c;
 return d;
}
}
```

Type casting in solidity:-

```
pragma solidity ^0.5.10;
contract fun{
    uint a=100;
    uint8 b=20;
    uint16 c=30;
    uint16 d=0;
    function display() public view returns(uint)
    {
        return d;
    }
    function add() public
    {
        d=uint16(a)+uint16(b)+c;

    }
    function mul() public view returns(uint)
    {
        uint e= a*uint(b)*uint(c);
        return e;
}
}
```

All integer values are different type.

Since d is uint16 type. So we Have to convert remaining data types to uint16 types, except c.

Loops & Decision making:

- For Loop
- While Loop
- Do..while Loop
- If statement
- If..else statement
- If..else if.. statement

Example: (if-else) (take two values from user and compare it and return true or false

```
pragma solidity ^0.5.10;
contract if_else{
    function setvalue(uint a,uint b) public pure returns(bool)
    {
        if (a>=b)
        {
        return true;

    }
    else{
        return false;
    }
}
}
```

```
output:-

setvalue
a:
12
b:
25
call
0: bool: false
```

Example: (loop) (sum of “n” numbers)

```
pragma solidity ^0.5.10;
contract loop_exampple
{
   function sumof(uint n)public pure returns (uint)
   {

      uint sum=0;
      for (uint i=0; i<=n; i++)
      {
         sum= sum+i;
      }
      return sum;
   }
}
```

```
output:
sumof
n:
10
call
0: uint256: 55
```

Constructor in solidity:

- Constructor are defined using the “constructor” keyword in solidity.
- Constructors are called when we deploy the contracts.
- If constructor is used to set the initial values of variable, every account see the same values of variable.
- If any account changes these values, the reflected value is shown in all the contracts.

Develop a smart contract which uses constructor to set the initial values of three variable a,b,c. write a function which doubles the value of a,b,c and display it.

```
pragma solidity ^ 0.5.10;
contract demo
{
   uint private a;
   uint private b;          Global variables
   uint private c;

    constructor (uint x, uint y, uint z) public
   {
      a=x;
      b=y;       Local variables
      c=z;

   }
```

```
 function double () public
   {
      a=a*2;
      b=b*2;
      c=c*2;
   }

    function show() public view
returns (uint, uint, uint)
   {
      return (a,b,c);
   }
}
```

Inheritance:

- if the contract B inherit the properties of contract A, then contract A is called “base (or) parent” class and contract B is called “derive (or) child” class.

- layout of single-level inheritance is:

```
Contract A          contract B is A

{          {

…..                 ….

…..                 ….
}          }
```

Develop a smart contract called demo1 which accept values from user. Develop another smart contract called demo2 which doubles the value which is given by the user in contract demo1 and display the double value in another smart contract demo3.

```
pragma solidity ^0.5.10;
contract demo1
{
   uint a;
   function set(uint b) public
   {
      a=b;
   }

}

contract demo2 is demo1
{
   function double() public
   {
      a=a*2;
   }

}
```

```
contract demo3 is demo2
{
   function display() public view
returns (uint)

   {
      return (a);
   }
}
```

Develop a smart contract called demo1 which accept values from user. Develop another smart contract called demo2 which doubles the value which is given by the user in contract demo1. similarly develop one more smart contract called demo3 which triple the value of demo1 and display it.

```
pragma solidity ^0.5.10;
contract demo1
{
    uint a;
    function set(uint b) public
    {
        a=b;
    }

}
contract demo2 is demo1
{
    uint b;
    function double() public{
        b=a*2;
    }
     function disply() public view returns (uint){
        return (b);
    }
}
```

```
contract demo3 is demo1
{
    uint b;
    function double() public
    {
        b=a*2;
    }
     function disply() public view
returns (uint)

    {
        return (b);
    }

}
```

Polymorphism:

- **Function overloading (or) method overloading.**
- **Contract overloading**
- **In function overloading, write the same function name but differentiate according to the numbers of parameter (or) types of parameters.**

Example of function overloading:-

```
 pragma solidity ^0.5.10;
contract demo1
{
    uint8 public a;

    uint16 public b;

    function square(uint8 x) public
    {
        a=x;

    }

     function square(uint16 y) public
    {
        b=y;

    }
function square(uint8 x, uint8 y) public
    {
        a=x+y;

    }
      function square(uint16 x, uint16 y,
    uint16 z) public
    {
        b=x+y+z;

    }
}
```

Example of contract overloading:-

```
pragma solidity ^ 0.5.10;
contract poly1
{
   uint internal a;

   function set(uint b) public
   {
      a=b;

   }

   function display() public view returns
(uint)
   {
      return 5;
   }
}
```

```
contract poly2 is poly1
{
    function display() public view returns
    (uint)
   {
      return a;
   }
}

contract poly3
{
   poly1 obj1= new poly1();
   poly1 obj2= new poly2(); → Contract
            overloading

   function call() public returns (uint,uint)
   {
      obj2.set(100);
      return (obj2.display(),obj1.display());
   }
}
```

Abstract contracts in solidity:

- abstract contracts are contracts that have partial function definition.
- We can not create an instance of an abstract contract.
- An abstract contract must be inherited by a child contract for utilizing its function.
- The function signature terminates using the semi-colon (;) character.
- There is no solidity keyword to mark a contract as abstract.
- A contract becomes an abstract class if it has function without implementation.

example:

```
Pragma solidity ^0.5.10;
Contract demo
{
Function set() public returns (bytes32);
}

Contract demo1 is demo
{
Function set() public returns (bytes32)
{
Return “hello”;
}
}
```

Develop a smart contract using an abstract contract for set a value and display it. Also set the value 100 externally.

```
pragma solidity ^ 0.5.10;
contract demo
{
    function set(uint x) public;
    function show() public returns (uint);
}

contract demo1 is demo
{
uint a;
  function set(uint x) public
  {
      a=x;
  }
 function show() public returns (uint)
 {
     return a;
 }
}
```

```
contract demo2
{
    demo1 obj= new demo1();
    function call() public returns
(uint)
    {
    obj.set(100);
    return obj.show();
}
}
```

```
output:-
set
20
show
a
0: uint256: 20

demo2 at 0x5fa...65a6d (memory)
call
100
```

Global Variables:-

- msg.sender = return the address of caller who invoked the function.

- msg.value = amount of wei sent along with transaction.

- if the account “A” id the owner of contract and account “A” is called the some function than msg.sender return the address of “A”.

- if the account “A” id the owner of contract and account “B” is called the some function than msg.sender return the address of “B”.

Develop a smart contract in which only owner is able to perform the arithmetic operations.

```
pragma solidity ^ 0.5.10;
contract demo
{
    address owner;
    uint public a=10;

    constructor() public
    {
        owner=msg.sender;
    }

    function demo1(uint b) public  returns (uint)
    {
        if (msg.sender == owner)
        {
            a=a*b;

        }
    }
}
```

Develop a smart contract in which only owner is able to perform the arithmetic operations.

Output 1:-

```
deploy using address- 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c

call using address- 0x14723a09acff6d2a60dcdf7aa4aff308fddc160c

demo1
uint256 b
a
0: uint256: 10

demo1
b:
2
transact
a
0: uint256: 20
```

Output 2:-

```
deploy using address-
0x14723a09acff6d2a60dcdf7aa4aff308fddc160c

call using address-
0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db

demo1
uint256 b
a
0: uint256: 10

demo1
b:
2
transact
a
0: uint256: 10
```

Modifier in solidity:-

- Modifier helps in modifying the behaviour of a function.
- Modifiers are defined using the “modifier” keyword.
- The code for modifier is replaced by curly bracket { }.
- The code within a modifier can validate the incoming value and can conditionally execute the called function after evaluation.
- The same modifier can be applied to multiple function.
- Modifier can only be applied to functions.
- The (_;) identifier is of special importance. Here (_) purpose is to replace itself with the function code that is invoked by the caller.

Develop a smart contract in which only owner is able to perform the arithmetic operations using the modifier.

```
pragma solidity ^ 0.5.10;
contract demo
{
    uint public a=10;
    address owner;


    constructor () public
    {
       owner=msg.sender;
    }

    modifier isowner
    {
       if(msg.sender==owner)
       {
       _;
       }
    }

  function set1 (uint b) public isowner
     {
        a=a*b;
     }

      function set2 (uint b) public
     {
        a=a*b;
     }


}
```

Note:- by calling the function, modifier is called.

Develop a smart contract in which only owner is able to perform the arithmetic operations using modifier.

Output 1:-

```
deploy at address:-
0xca35b7d915458ef540ade6068dfe2f44e8fa733c
call at address:-
0xca35b7d915458ef540ade6068dfe2f44e8fa733c

Set1 uint256 b
Set2 uint256 b
a 0: uint256: 10

Set1 b:2
transact
Set2 uint256 b
a 0: uint256: 20

Set1 b:2
transact
Set2 b:2
transact
a 0: uint256: 40
```

Output 2:-

```
deploy at address:-
0xca35b7d915458ef540ade6068dfe2f44e8fa733c
call at address:-
0x14723a09acff6d2a60dcdf7aa4aff308fddc160c

Set1 uint256 b
Set2 uint256 b
a 0: uint256: 10

Set1 b:2
Transact
Set2 uint256 b
a 0: uint256: 10

Set1 b:2
transact
Set2 b:2
transact
a 0: uint256: 20
```