MODULE 3:

PROGRAMMING IN BLOCKCHAIN

Prepared By: Prof. Swapnil S. Sonawane

INTRODUCTION TO SMART CONTRACT

A Smart Contract is a computer program working on top of a blockchain, which has a set of rules (conditions) based on which, the parties to that Smart Contract, agree to interact with each other.

The agreement will be automatically executed when the pre-defined rules are met.

Real-life agreements to record the purchase and sale of assets can be automated using Smart Contracts.

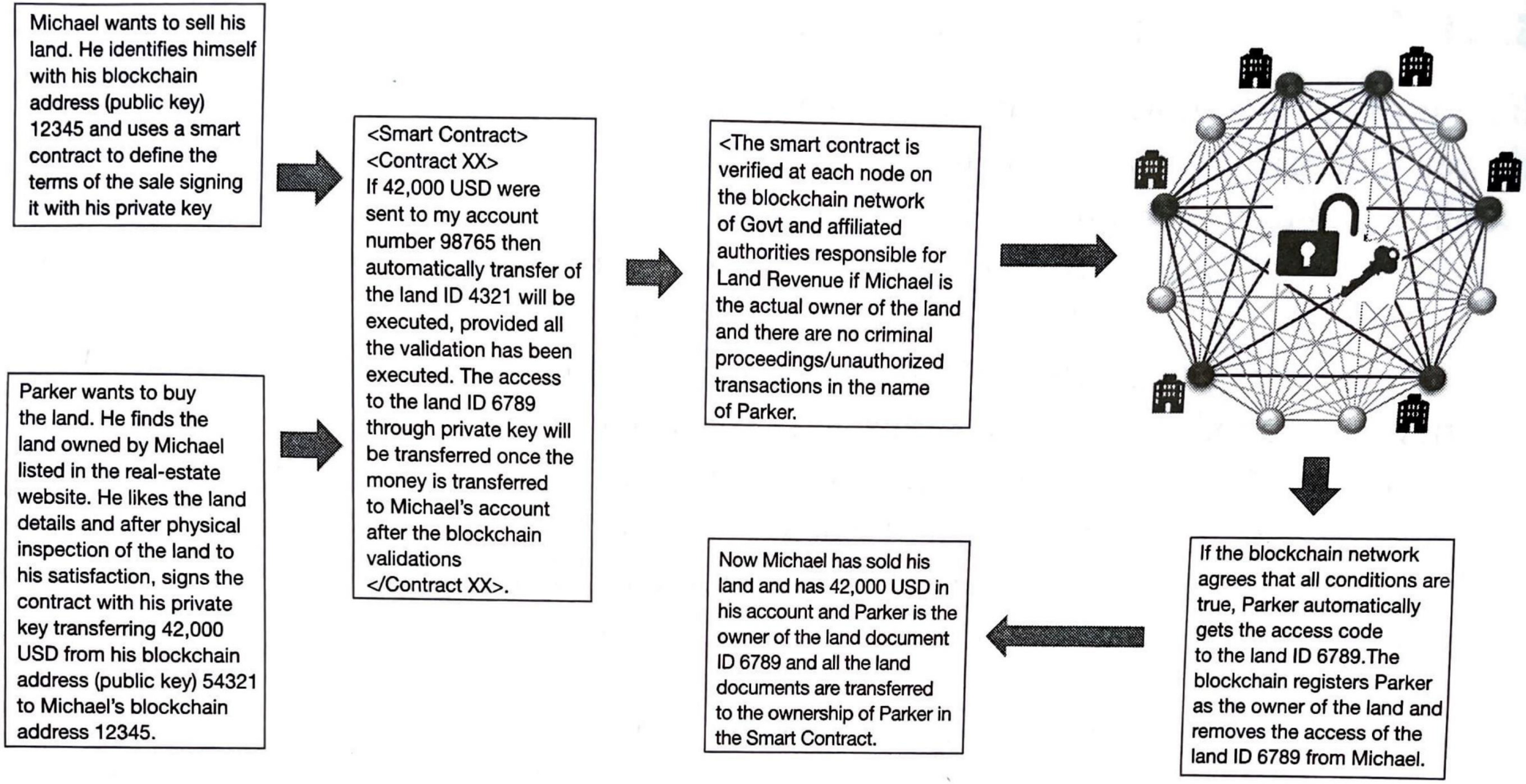
When the smart contracts are programmed, compiled, and migrated on to the blockchain, they are provided an address depicting the location of the contract on the blockchain, and the application is recorded in a block and distributed across all the nodes.

Hence, the applications are called decentralized applications. Decentralized applications interact with external users through a web browser.

A Smart Contract reduces transaction costs drastically, including cost of Reaching an agreement, Formalizing an agreement and Enforcing an agreement

The benefits of smart contracts are:

- Simple, faster to execute, and availability of updates in real-time
- No requirement of mediators and centralized entities
- Lesser cost because there is no need to pay fees to middlemen
- There is no delay in delivering outcomes.



Michael wants to sell his land. He identifies himself with his blockchain address (public key) 12345 and uses a smart contract to define the terms of the sale signing it with his private key

Parker wants to buy the land. He finds the land owned by Michael listed in the real-estate website. He likes the land details and after physical inspection of the land to his satisfaction, signs the contract with his private key transferring 42,000 USD from his blockchain address (public key) 54321 to Michael's blockchain address 12345.

<Smart Contract>
<Contract XX>
If 42,000 USD were sent to my account number 98765 then automatically transfer of the land ID 4321 will be executed, provided all the validation has been executed. The access to the land ID 6789 through private key will be transferred once the money is transferred to Michael's account after the blockchain validations
</Contract XX>.

<The smart contract is verified at each node on the blockchain network of Govt and affiliated authorities responsible for Land Revenue if Michael is the actual owner of the land and there are no criminal proceedings/unauthorized transactions in the name of Parker.

If the blockchain network agrees that all conditions are true, Parker automatically gets the access code to the land ID 6789.The blockchain registers Parker as the owner of the land and removes the access of the land ID 6789 from Michael.

Now Michael has sold his land and has 42,000 USD in his account and Parker is the owner of the land document ID 6789 and all the land documents are transferred to the ownership of Parker in the Smart Contract.

STRUCTURE OF SMART CONTRACT

How Smart Contract Work

A smart contract works through automated conditional performance. When obligation is met, the corresponding obligation is triggered.

For example, an obligation could be triggered by:

- a specific event ("if X happens, then action Y")
- a specific date or at the expiration of time ("at X date, actionY")

Different platforms can host smart contracts, but generally, smart contracts are implemented on a blockchain where the validation logic sits inside a "block". All the messages needed for a smart contract are bundled within the block.

High-level programming languages like Solidity are used to create applications to automate business processes and record them on the blockchain in the form of smart contracts.

Blockchain technologies use public-key encryption infrastructure (PKI). The initiator who wishes to participate in a smart contract hosted on a permissionless blockchain can download the software from open sources and publish the public key on the system (software) publicly.

When publishing the public key, the blockchain will also generate a corresponding private key for the initiator's address.

If the initiator wants to trigger a smart contract transaction on the relevant ledger, the software will use its address to send an initiating message, encrypted with its private key to the other active participants.

That message is picked up by the nodes; however, it can be signed only by nodes having the private key. Participants with access to the public key received from the software can use it to verify the smart contract transaction and also to authenticate the message contents.

Whenever a sufficient number of other participants or nodes is reached in a permissionless blockchain (typically more than 50%), the consensus protocol determines that message related to the smart contract should be added to the blockchain.

TYPES OF SMART CONTRACT

Smart contracts have the potential to influence many industries. Smart Contracts can be categorized into four types based on the applications

1. Smart Legal Contracts

They are smart contracts with various legal contract templates. They just execute the contracts as per template used

2. DApps (Decentralized Applications)

Decentralised Applications (DApps) are blockchain-based applications that allow users to interact with smart contracts deployed on the blockchain.

It is an application built on a decentralized network that combines a smart contract and a frontend user interface.

Decentralized applications are blockchain-powered websites, whereas smart contracts work as API connectors that connect DApps with blockchain.

3. DAO (Distributed Autonomous Organization)

A Decentralized Autonomous Organization is represented by logic encoded as transparent program controlled by stakeholders and not influenced by a central authority.

DAOs use blockchain technology to maintain a secure advanced record in the form of Digital Ledger (DL) to trace the financial transaction over the internet.

4. Smart Contracting Devices (Combined with IoT)

Currently, any failure in an IoT (Internet of Things) ecosystem on a web internet will reveal critical personal data such as our shopping data, census data, etc., that are being used by multiple devices.

By using Blockchain to manage and access data from IoT devices, the hacker has to bypass an additional layer of security coded with some of the most robust encryption standards available. Also, there is no worry of a single-point failure, due to decentralized authority.

COMPONENTS OF SMART CONTRACT

Variables

Any contract data must be assigned to a location: either to storage or memory.

Storage

Persistent data is referred to as storage and is represented by state variables. These values get stored permanently on the blockchain.

```
contract SimpleStorage
{
    uint storedData; // State variable
}
```

Other types include:

Boolean, integer, fixed point numbers, fixed-size byte arrays, dynamically-sized byte arrays, integer literals, String literals, Hexadecimal literals

Memory

Values that are only stored for the lifetime of a contract function's execution are called memory variables.

Environment variables

In addition to the variables, we define on our contract, there are some special global variables. They are primarily used to provide information about the blockchain or current transaction.

Functions:

Functions can get information or set information in response to incoming transactions.

There are two types of function calls:

internal – these don't create an EVM call

Internal functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it)

external – these do create an EVM call

External functions are part of the contract interface, which means they can be called from other contracts and via transactions.

They can also be public or private

public functions can be called internally from within the contract or externally via messages

private functions are only visible for the contract they are defined in and not in derived contracts

View functions

These functions promise not to modify the state of the contract's data.

Constructor functions

Constructor functions are only executed once when the contract is first deployed. Like constructor in many class-based programming languages, these functions often initialize state variables to their specified values.

Built-in functions

In addition to the variables and functions you define on your contract, there are some special built-in functions. The most obvious example is:

address.send() – Solidity

send(address) – Vyper

Events

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 blockchain and are accessible using address of the contract till the contract is present on the blockchain.

SMART CONTRACT APPROACH

Smart contracts are set of rules and protocols which two parties agree upon and have to follow.

IDE

To write and execute solidity codes, the most common IDE used is an online IDE known as REMIX.

MetaMask

MetaMask is a type of Ethereum wallet that bridges the gap between the user interfaces for Ethereum and the regular web.

MetaMask is mainly used as a plugin in chrome. You can add it from

https://chrome.google.com/webstore/detail/metamask

After adding MetaMask as an extension in chrome and creating an account, set up our account as follows –

Step 1: Select Ropsten Test Network from a list of available networks

Step 2: Request test ether

Step 3: MetaMask is ready for deployment.

Steps to deploy your contract

Step 1: Open Remix IDE in your browser.

Step 2: Write code for testing and compile by clicking on the compile button

Step 3: After compilation, move to deploy section and select Injected Web3 in place of JavaScriptVM

Step 4: Now the contract is ready to be deployed. Click on deploy button.

Step 5: Expand the deployed contract as below and get the output using the get_output() function

Step 6: Now, to verify whether the transaction (process) executed successfully, we can check our balance on MetaMask.

DATA TYPES IN SOLIDITY

Solidity is a statically typed language, which implies that the type of each of the variables should be specified.

Value Types

Value type variables store their own data. These types of variables are always passed by value.

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.

Fixed Point Numbers: They can be declared as fixed and unfixed for signed and unsigned fixed-point numbers of varying sizes 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 using 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 character set equal to or more than a byte. The length of bytes is from 1 to 32, while the string has a dynamic length. Byte has an advantage that it uses less gas, so better to use when we know the length of data.

Enums: It is used to create user-defined data types, used to assign a name to an integral constant. Options of enums can be represented by unsigned integer values starting from 0.

```
pragma solidity ^ 0.8.0;
contract Types
{
    bool public boolean = false;
    int32 public int_var = -60313;
    string public str = “Vidyalankar";
    bytes1 public b = "a";
    enum enum1 {apple, orange, mango}
    function Enum() public pure returns(enum1)
    {
        return enum1.mango;
    }
}
```

Reference Types

Reference type variables store the location of the data. They don’t share the data directly. With the help of reference type, two different variables can refer to the same location where any change in one variable can affect the other one. Reference type in solidity are listed below:

Arrays: 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.

Struct: Solidity allows users to create and define their own type in the form of structures. The structure is a group of different types of elements

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.

```
pragma solidity ^0.4.18;
contract example1
{
    uint[5] public array= [1, 2, 3, 4, 5] ;
    struct student
    {
        string name;
        string subject;
        uint8 marks;
    }
    student public std1;
    function structure() public returns(string memory, string memory, uint)
    {
        std1.name = "AAA";
        std1.subject = "Chemistry";
        std1.marks = 88;
        return (std1.name, std1.subject, std1.marks);
    }
}
```

FUNCTIONS IN SOLIDITY

A function is a group of reusable code which can be called anywhere in the program.

Function Definition

Syntax:

```
function function-name(parameter-list) scope returns()
{
   //statements
}
```

```
pragma solidity ^0.8.0;
contract Test
{
   function getResult() public view returns(uint)
  {
     uint a = 1; // local variable
     uint b = 2;
     uint result = a + b;
     return result;
   }
}
```

View functions ensure that they will not modify the state. A function can be declared as view. The following statements if present in the function are considered modifying the state and compiler will throw warning in such cases.

- Modifying state variables.
- Emitting events.
- Creating other contracts.
- Using selfdestruct.
- Sending Ether via calls.
- Calling any function which is not marked view or pure.

Fallback function is a special function available to a contract. 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 argument and no return value

Pure functions ensure that they not read or modify the state. The following statements if present in the function are considered reading the state and compiler will throw warning in such cases.

- Reading state variables.
- Accessing address(this).balance or <address>.balance.
- Accessing any of the special variable of block, tx, msg
- Calling any function not marked pure.

Example:

```
pragma solidity ^0.8.0;
contract Test {
   function() external { x = 1; }
   function getResult() public pure returns(uint product, uint sum){
      uint a = 1;
      uint b = 2;
      product = a * b;
      sum = a + b;
   }
}
```

Visibility Quantifiers

Following are various visibility quantifiers for functions/state variables of a contract.

external: External functions are meant to be called by other contracts. They cannot be used for internal call. State variables cannot be marked as external.

public: Public functions/ Variables can be used both externally and internally. For public state variable, Solidity automatically creates a getter function.

internal: Internal functions/ Variables can only be used internally or by derived contracts.

private: Private functions/ Variables can only be used internally and not even by derived contracts.

Example:

INHERITANCE

Inheritance is a way to extend functionality of a contract. Solidity supports both single as well as multiple inheritance.

A derived contract can access all non-private members including internal methods and state variables.

Function overriding is allowed provided function signature remains same.

We can call a super contract's function using super keyword or using super contract name.

In case of multiple inheritance, function call using super gives preference to most derived contract.

ERROR HANDLING

Following are some of the important methods used in error handling:

assert(bool condition): In case condition is not met, this method call causes an invalid opcode and any changes done to state got reverted. This method is to be used for internal errors.

require(bool condition): In case condition is not met, this method call reverts to original state. This method is to be used for errors in inputs or external components.

require(bool condition, string memory message): In case condition is not met, this method call reverts to original state. This method is to be used for errors in inputs or external components. It provides an option to provide a custom message.

revert(): This method aborts the execution and revert any changes done to the state.

revert(string memory reason): This method aborts the execution and revert any changes done to the state. It provides an option to provide a custom message.