Smart Contracts

Module 3

Trust in conventional contracts

- Alice and Bob are having a bicycle race. Let's say Alice bets Bob $10 that she will win the race. Bob is confident he'll be the winner and agrees to the bet. In the end, Alice finishes the race well ahead of Bob and is the clear winner. But Bob refuses to pay out on the bet, claiming Alice must have cheated.

- This silly example illustrates the problem with any non-smart agreement. Even if the conditions of the agreement get met (i.e. you are the winner of the race), you must still trust another person to fulfill the agreement (i.e. payout on the bet).

A digital vending machine

- A simple metaphor for a smart contract is a vending machine, which works somewhat similarly to a smart contract - specific inputs guarantee predetermined outputs.
  - You select a product
  - The vending machine displays the price
  - You pay the price
  - The vending machine verifies that you paid the right amount
  - The vending machine gives you your item
- The vending machine will only dispense your desired product after all requirements are met. If you don't select a product or insert enough money, the vending machine won't give out your product.

Automatic execution

- The main benefit of a smart contract is that it deterministically executes unambiguous code when certain conditions are met. There is no need to wait for a human to interpret or negotiate the result. This removes the need for trusted intermediaries.
- For example, you could write a smart contract that holds funds in escrow for a child, allowing them to withdraw funds after a specific date. If they try to withdraw before that date, the smart contract won't execute. Or you could write a contract that automatically gives you a digital version of a car's title when you pay the dealer.

Predictable outcomes

- Traditional contracts are ambiguous because they rely on humans to interpret and implement them. For example, two judges might interpret a contract differently, which could lead to inconsistent decisions and unequal outcomes. Smart contracts remove this possibility. Instead, smart contracts execute precisely based on the conditions written within the contract's code. This precision means that given the same circumstances, the smart contract will produce the same result.

Public record

- Smart contracts are useful for audits and tracking. Since Ethereum smart contracts are on a public blockchain, anyone can instantly track asset transfers and other related information. For example, you can check to see that someone sent money to your address.

Privacy protection

- Smart contracts also protect your privacy. Since Ethereum is a pseudonymous network (your transactions are tied publicly to a unique cryptographic address, not your identity), you can protect your privacy from observers.

Visible terms

- Finally, like traditional contracts, you can check what's in a smart contract before you sign it (or otherwise interact with it). A smart contract's transparency guarantees that anyone can scrutinize it.

WHAT IS SMART CONTRACT

Overview

- Smart contracts are contracts that are coded and stored on the blockchain.
- They automate agreements between the creator and recipient, making them immutable and irreversible.
- Their primary purpose is to automate the execution of an agreement without intermediaries, ensuring that all parties can confirm the conclusion instantly.
- Additionally, they can be programmed to initiate a workflow based on specific circumstances.

- A Smart Contract (or cryptocontract) is a computer program that directly and automatically controls the transfer of digital assets between the parties under certain conditions.
- A smart contract works in the same way as a traditional contract while also automatically enforcing the contract.
- Smart contracts are programs that execute exactly as they are set up(coded, programmed) by their creators.
- Just like a traditional contract is enforceable by law, smart contracts are enforceable by code.

- Smart contracts are the fundamental building blocks of Ethereum's application layer.
- They are computer programs stored on the blockchain that follow "if this then that" logic, and are guaranteed to execute according to the rules defined by its code, which cannot be changed once created.

- The bitcoin network was the first to use some sort of smart contract by using them to transfer value from one person to another.
- The smart contract involved employs basic conditions like checking if the amount of value to transfer is actually available in the sender account.
- Later, the Ethereum platform emerged which was considered more powerful, precisely because the developers/programmers could make custom contracts in a Turing-complete language.
- It is to be noted that the contracts written in the case of the bitcoin network were written in a Turing-incomplete language, restricting the potential of smart contracts implementation in the bitcoin network.
- There are some common smart contract platforms like Ethereum, Solana, Polkadot, Hyperledger fabric, etc.

Features of Smart Contracts

- **Distributed:** Everyone on the network is guaranteed to have a copy of all the conditions of the smart contract and they cannot be changed by one of the parties. A smart contract is replicated and distributed by all the nodes connected to the network.
- **Deterministic:** Smart contracts can only perform functions for which they are designed only when the required conditions are met. The final outcome will not vary, no matter who executes the smart contract.
- **Immutable:** Once deployed smart contract cannot be changed, it can only be removed as long as the functionality is implemented previously.

- **Autonomy:** There is no third party involved. The contract is made by you and shared between the parties. No intermediaries are involved which minimizes bullying and grants full authority to the dealing parties. Also, the smart contract is maintained and executed by all the nodes on the network, thus removing all the controlling power from any one party's hand.
- **Customizable:** Smart contracts have the ability for modification or we can say customization before being launched to do what the user wants it to do.

- **Transparent:** Smart contracts are always stored on a public distributed ledger called blockchain due to which the code is visible to everyone, whether or not they are participants in the smart contract.
- **Trustless:** These are not required by third parties to verify the integrity of the process or to check whether the required conditions are met.
- **Self-verifying:** These are self-verifying due to automated possibilities.
- **Self-enforcing:** These are self-enforcing when the conditions and rules are met at all stages.

Capabilities of Smart Contracts

- **Accuracy:** Smart contracts are accurate to the limit a programmer has accurately coded them for execution.
- **Automation:** Smart contracts can automate the tasks/ processes that are done manually.
- **Speed:** Smart contracts use software code to automate tasks, thereby reducing the time it takes to maneuver through all the human interaction-related processes. Because everything is coded, the time taken to do all the work is the time taken for the code in the smart contract to execute.

- **Backup:** Every node in the blockchain maintains the shared ledger, providing probably the best backup facility.
- **Security:** Cryptography can make sure that the assets are safe and sound. Even if someone breaks the encryption, the hacker will have to modify all the blocks that come after the block which has been modified. Please note that this is a highly difficult and computation-intensive task and is practically impossible for a small or medium-sized organization to do.

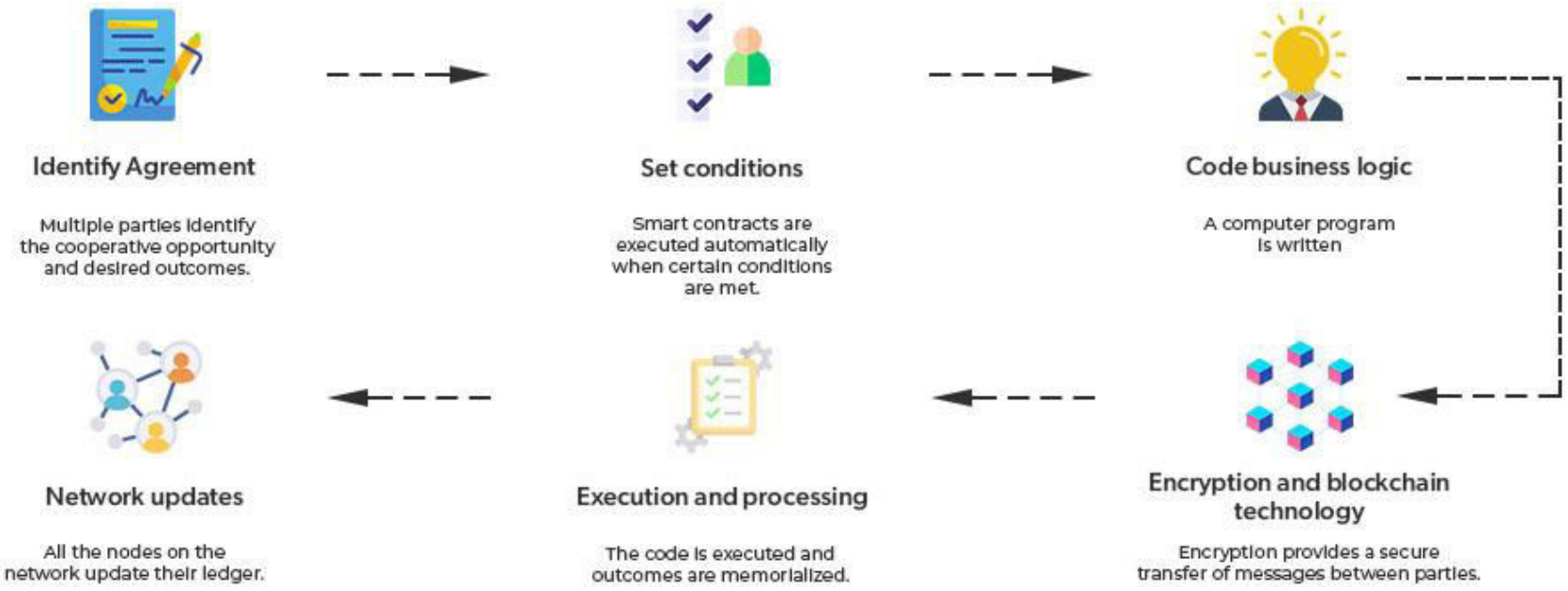
- **Savings:** Smart contracts save money as they eliminate the presence of intermediaries in the process. Also, the money spent on the paperwork is minimal to zero.
- **Manages information:** Smart contract manages users' agreement, and stores information about an application like domain registration, membership records, etc.
- **Multi-signature accounts:** Smart contracts support multi-signature accounts to distribute funds as soon as all the parties involved confirm the agreement.

How Do Smart Contracts Work?

- A smart contract is just a digital contract with the security coding of the blockchain.
- It has details and permissions written in code that require an exact sequence of events to take place to trigger the agreement of the terms mentioned in the smart contract.
- It can also include the time constraints that can introduce deadlines in the contract.
- Every smart contract has its address in the blockchain. The contract can be interacted with by using its address presuming the contract has been broadcasted on the network.

- The idea behind smart contracts is pretty simple. They are executed on a basis of simple logic, IF-THEN for example:
- **IF** you send object A, **THEN** the sum (of money, in cryptocurrency) will be transferred to you.
- **IF** you transfer a certain amount of digital assets (cryptocurrency, for example, ether, bitcoin), **THEN** the A object will be transferred to you.
- **IF** I finish the work, **THEN** the digital assets mentioned in the contract will be transferred to me.

How does a Smart Contract Work?



- **Identify Agreement:** Multiple parties identify the cooperative opportunity and desired outcomes and agreements could include business processes, asset swaps, etc.
- **Set conditions:** Smart contracts could be initiated by parties themselves or when certain conditions are met like financial market indices, events like GPS locations, etc.
- **Code business logic:** A computer program is written that will be executed automatically when the conditional parameters are met.
- **Encryption and blockchain technology:** Encryption provides secure authentication and transfer of messages between parties relating to smart contracts.
- **Execution and processing:** In blockchain iteration, whenever consensus is reached between the parties regarding authentication and verification then the code is executed and the outcomes are memorialized for compliance and verification.
- **Network updates:** After smart contracts are executed, all the nodes on the network update their ledger to reflect the new state. Once the record is posted and verified on the blockchain network, it cannot be modified, it is in append mode only.

Applications of Smart Contracts

- **Real Estate:** Reduce money paid to the middleman and distribute between the parties actually involved. For example, a smart contract to transfer ownership of an apartment once a certain amount of resources have been transferred to the seller's account(or wallet).
- **Vehicle ownership:** A smart contract can be deployed in a blockchain that keeps track of vehicle maintenance and ownership. The smart contract can, for example, enforce vehicle maintenance service every six months; failure of which will lead to suspension of driving license.
- **Music Industry:** The music industry could record the ownership of music in a blockchain. A smart contract can be embedded in the blockchain and royalties can be credited to the owner's account when the song is used for commercial purposes. It can also work in resolving ownership disputes.
- **Government elections:** Once the votes are logged in the blockchain, it would be very hard to decrypt the voter address and modify the vote leading to more confidence against the ill practices.
- **Management:** The blockchain application in management can streamline and automate many decisions that are taken late or deferred. Every decision is transparent and available to any party who has the authority(an application on the private blockchain). For example, a smart contract can be deployed to trigger the supply of raw materials when 10 tonnes of plastic bags are produced.
- **Healthcare:** Automating healthcare payment processes using smart contracts can prevent fraud. Every treatment is registered on the ledger and in the end, the smart contract can calculate the sum of all the transactions. The patient can't be discharged from the hospital until the bill has been paid and can be coded in the smart contract.

Advantages of Smart Contracts

- **Recordkeeping:** All contract transactions are stored in chronological order in the blockchain and can be accessed along with the complete audit trail. However, the parties involved can be secured cryptographically for full privacy.
- **Autonomy:** There are direct dealings between parties. Smart contracts remove the need for intermediaries and allow for transparent, direct relationships with customers.
- **Reduce fraud:** Fraudulent activity detection and reduction. Smart contracts are stored in the blockchain. Forcefully modifying the blockchain is very difficult as it's computation-intensive. Also, a violation of the smart contract can be detected by the nodes in the network and such a violation attempt is marked invalid and not stored in the blockchain.
- **Fault-tolerance:** Since no single person or entity is in control of the digital assets, one-party domination and situation of one part backing out do not happen as the platform is decentralized and so even if one node detaches itself from the network, the contract remains intact.
- **Enhanced trust:** Business agreements are automatically executed and enforced. Plus, these agreements are immutable and therefore unbreakable and undeniable.
- **Cost-efficiency:** The application of smart contracts eliminates the need for intermediaries(brokers, lawyers, notaries, witnesses, etc.) leading to reduced costs. Also eliminates paperwork leading to paper saving and money-saving.

Challenges of Smart Contracts

- **No regulations:** A lack of international regulations focusing on blockchain technology(and related technology like smart contracts, mining, and use cases like cryptocurrency) makes these technologies difficult to oversee.
- **Difficult to implement:** Smart contracts are also complicated to implement because it's still a relatively new concept and research is still going on to understand the smart contract and its implications fully.
- **Immutable:** They are practically immutable. Whenever there is a change that has to be incorporated into the contract, a new contract has to be made and implemented in the blockchain.
- **Alignment:** Smart contracts can speed the execution of the process that span multiple parties irrespective of the fact whether the smart contracts are in alignment with all the parties' intention and understanding.

ANATOMY OF SMART CONTRACTS

Program#1: Helloworld

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;
contract HelloWorld {
    string public greet = "Hello World!";
}
```

```
// Specifies the version of Solidity, using semantic versioning.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma
pragma solidity ^0.5.10;
// Defines a contract named `HelloWorld`.
// A contract is a collection of functions and data (its state).
// Once deployed, a contract resides at a specific address on the Ethereum blockchain.
contract HelloWorld {
    // Declares a state variable `message` of type `string`.
    // State variables are variables whose values are permanently stored in contract storage.
    // The keyword `public` makes variables accessible from outside a contract and creates a function that other contracts or clients can call to access the value.
    string public message;
    // Similar to many class-based object-oriented languages, a constructor is a special function that is only executed upon contract creation.
    // Constructors are used to initialize the contract's data.
constructor(string memory initMessage) public {
        // Accepts a string argument `initMessage` and sets the value into the contract's `message` storage variable).
        message = initMessage;
    }
    // A public function that accepts a string argument and updates the `message` storage variable.
    function update(string memory newMessage) public {
        message = newMessage;
    }
}
```

SMART CONTRACTS

- A smart contract is a program that runs at an address on Ethereum.
- They're made up of data and functions that can execute upon receiving a transaction.

DATA

- Any contract data must be assigned to a location: either to **storage or memory**.
- It's costly to modify storage in a smart contract so you need to consider where your data should live

Storage

- Persistent data is referred to as storage and is represented by state variables.
- These values get stored permanently on the blockchain.
- You need to declare the type so that the contract can keep track of how much storage on the blockchain it needs when it compiles.

```
// Solidity example
contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

Storage

- types include:
  - boolean
  - integer
  - fixed point numbers
  - fixed-size byte arrays
  - dynamically-sized byte arrays
  - Rational and integer literals
  - String literals
  - Hexadecimal literals
  - Enums
- An ***address*** type can hold an Ethereum address which equates to 20 bytes or 160 bits. It returns in hexadecimal notation with a leading 0x.

Address

- The address type comes in two largely identical flavors:
  - address: Holds a 20 byte value (size of an Ethereum address).
  - address payable: Same as address, but with the additional members transfer and send.

- The idea behind this distinction is that address payable is an address you can send Ether to, while you are not supposed to send Ether to a plain address, for example because it might be a smart contract that was not built to accept Ether.

- Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable must be explicit via payable(<address>).

Members of Addresses

- **balance and transfer**
- It is possible to query the balance of an address using the property **balance** and to send Ether (in units of wei) to a payable address using the **transfer** function:

```
address payable x = payable(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10)
x.transfer(10);
```

- **send**
- Send is the low-level counterpart of transfer. If the execution fails, the current contract will not stop with an exception, but send will return false.

Storage

- Storage is a key-value store that maps 256-bit words to 256-bit words.
- It is not possible to enumerate storage from within a contract, it is comparatively costly to read, and even more to initialise and modify storage.
- Because of this cost, **you should minimize what you store in persistent storage to what the contract needs to run.**
- Store data like derived calculations, caching, and aggregates outside of the contract.
- A contract can neither read nor write to any storage apart from its own.

Memory

- Values that are only stored for the lifetime of a contract function's execution are called memory variables.
- Since these are not stored permanently on the blockchain, they are much cheaper to use.
- **Memory** is linear and can be addressed at byte level, but reads are limited to a width of 256 bits, while writes can be either 8 bits or 256 bits wide.
- Memory is expanded by a word (256-bit), when accessing (either reading or writing) a previously untouched memory word (i.e. any offset within a word).
- At the time of expansion, the cost in gas must be paid.
- Memory is more costly the larger it grows (it scales quadratically).

Stack

- The EVM is not a register machine but a stack machine, so all computations are performed on a data area called the **stack**.
- It has a maximum size of 1024 elements and contains words of 256 bits.
- Access to the stack is limited to the top end in the following way:
  - It is possible to copy one of the topmost 16 elements to the top of the stack or swap the topmost element with one of the 16 elements below it.
  - All other operations take the topmost two (or one, or more, depending on the operation) elements from the stack and push the result onto the stack.
  - Of course it is possible to move stack elements to storage or **memory** in order to get deeper access to the stack, but it is not possible to just access arbitrary elements deeper in the stack without first removing the top of the stack.

Instruction Set

- The instruction set of the EVM is kept minimal in order to avoid incorrect or inconsistent implementations which could cause consensus problems.
- All instructions operate on the basic data type, 256-bit words or on slices of **memory** (or other byte arrays).
- The usual arithmetic, bit, logical and comparison operations are present.
- Conditional and unconditional jumps are possible.
- Furthermore, contracts can access relevant properties of the current block like its number and timestamp.

Message Calls

- Contracts can call other contracts or send Ether to non-contract accounts by the means of message calls.
- Message calls are similar to transactions, in that they have a source, a target, data payload, Ether, gas and return data.
- In fact, every transaction consists of a top-level message call which in turn can create further message calls.
- A contract can decide how much of its remaining **gas** should be sent with the inner message call and how much it wants to retain.
- If an out-of-gas exception happens in the inner call (or any other exception), this will be signaled by an error value put onto the stack.
- In this case, only the gas sent together with the call is used up.
- In Solidity, the calling contract causes a manual exception by default in such situations, so that exceptions “bubble up” the call stack.

Environment variables

- there are some special global variables.
- They are primarily used to provide information about the blockchain or current transaction.

Examples:

Prop	State variable	Description
block.timestamp	uint256	Current block epoch timestamp
msg.sender	address	Sender of the message (current call)

Control Structures

- Most of the control structures known from curly-braces languages are available in Solidity:
- There is: **if, else, while, do, for, break, continue, return**, with the usual semantics known from C or JavaScript.
- Solidity also supports **exception handling** in the form of try/catch-statements, but only for external function calls and contract creation calls. Errors can be created using the revert statement.
- Parentheses can *not* be omitted for conditionals, but curly braces can be omitted around single-statement bodies.
- Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so if (1) { ... } is *not* valid Solidity.

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. An external function f cannot be called internally (i.e. f() does not work, but this.f() works).
- 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
  - Both functions and state variables can be made public or private

function for updating a state variable on a contract:

```
// Solidity example
function update_name(string value) public {
    dapp_name = value;
}
```

- The parameter value of type string is passed into the function: update_name
- It's declared public, meaning anyone can access it
- It's not declared view, so it can modify the contract state

View functions

- These functions promise not to modify the state of the contract's data.
- Common examples are "getter" functions – you might use this to receive a user's balance for example.

```
// Solidity example
function balanceOf(address _owner) public view returns (uint256 _balance) {
    return ownerPizzaCount[_owner];
}
```

What is considered modifying state:

- Writing to state variables.
- Emitting events.
- Creating other contracts.
- Using selfdestruct.
- Sending ether via calls.
- Calling any function not marked view or pure.
- Using low-level calls.
- Using inline assembly that contains certain opcodes

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.

```
// Solidity example
// Initializes the contract's data, setting the `owner`
// to the address of the contract creator.
constructor() public {
    // All smart contracts rely on external transactions to trigger its functions.
    // `msg` is a global variable that includes relevant data on the given transaction,
    // such as the address of the sender and the ETH value included in the transaction.

    owner = msg.sender;
}
```

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()***
- These allow contracts to send ETH to other accounts

WRITING FUNCTIONS

- Your function needs:
  - parameter variable and type (if it accepts parameters)
  - declaration of internal/external
  - declaration of pure/view/payable
  - returns type (if it returns a value)

```
contract ExampleDapp {
        string dapp_name; // state variable
        constructor() public {
        dapp_name = "My Example dapp";
        }
        // Get Function
        function read_name() public view returns(string) {
        return dapp_name;
        }
        // Set Function
        function update_name(string value) public {
          dapp_name = value;
        }
}
```

Function Calls with Named Parameters

- Function call arguments can be given by name, in any order, if they are enclosed in { } as can be seen in the following example. The argument list has to coincide by name with the list of parameters from the function declaration, but can be in arbitrary order.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
contract C {
    mapping(uint => uint) data;
    function f() public {
        set({value: 2, key: 3});
    }
    function set(uint key, uint value) public {
        data[key] = value;
    }
}
```

Arrays

- Arrays can have a compile-time fixed size, or they can have a dynamic size.
- The type of an array of fixed size k and element type T is written as T[k], and an array of dynamic size as T[].
- Indices are zero-based, and access is in the opposite direction of the declaration.
- Array elements can be of any type
- **bytes and string as Arrays**
  - Variables of type bytes and string are special arrays. The bytes type is similar to bytes1[], but it is packed tightly in calldata and memory.
  - string is equal to bytes but does not allow length or index access.

Array Members

- **length**:Arrays have a length member that contains their number of elements. The length of memory arrays is fixed (but dynamic, i.e. it can depend on runtime parameters) once they are created.
- **push()**:Dynamic storage arrays and bytes (not string) have a member function called push() that you can use to append a zero-initialised element at the end of the array. It returns a reference to the element, so that it can be used like x.push().t = 2 or x.push() = b.
- **push(x)**:Dynamic storage arrays and bytes (not string) have a member function called push(x) that you can use to append a given element at the end of the array. The function returns nothing.
- **pop()**:Dynamic storage arrays and bytes (not string) have a member function called pop() that you can use to remove an element from the end of the array. This also implicitly calls delete on the removed element. The function returns nothing.

Structs

- Structs are custom defined types that can group several variables.

```
pragma solidity ^0.4.0;
contract Ballot {
struct Voter { // Struct
uint weight1, weight2, weight3;
bool voted;
address delegate1, delegate2, delegate3, delegate4;
string name;
uint vote1, vote2, vote3, vote4, vote5;
uint height1, height2, height3   } }
```

Mappings

- Mappings can be seen as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type's default value.
- Mappings are declared as:
  - Mapping(_Keytype => _ValueType)
- Keytype can be almost any type except for a dynamically sized array, a contract, an enum and a struct.
  - mapping(address => uint) public balances;

Mappings

- Declare Mappings
- CRUD
- Default Values
- Exotic Mappings
    - nested Mapping
    - Array inside Mapping

Declare mappings

```
mapping(address => uint) balances;
```

C.R.U.D

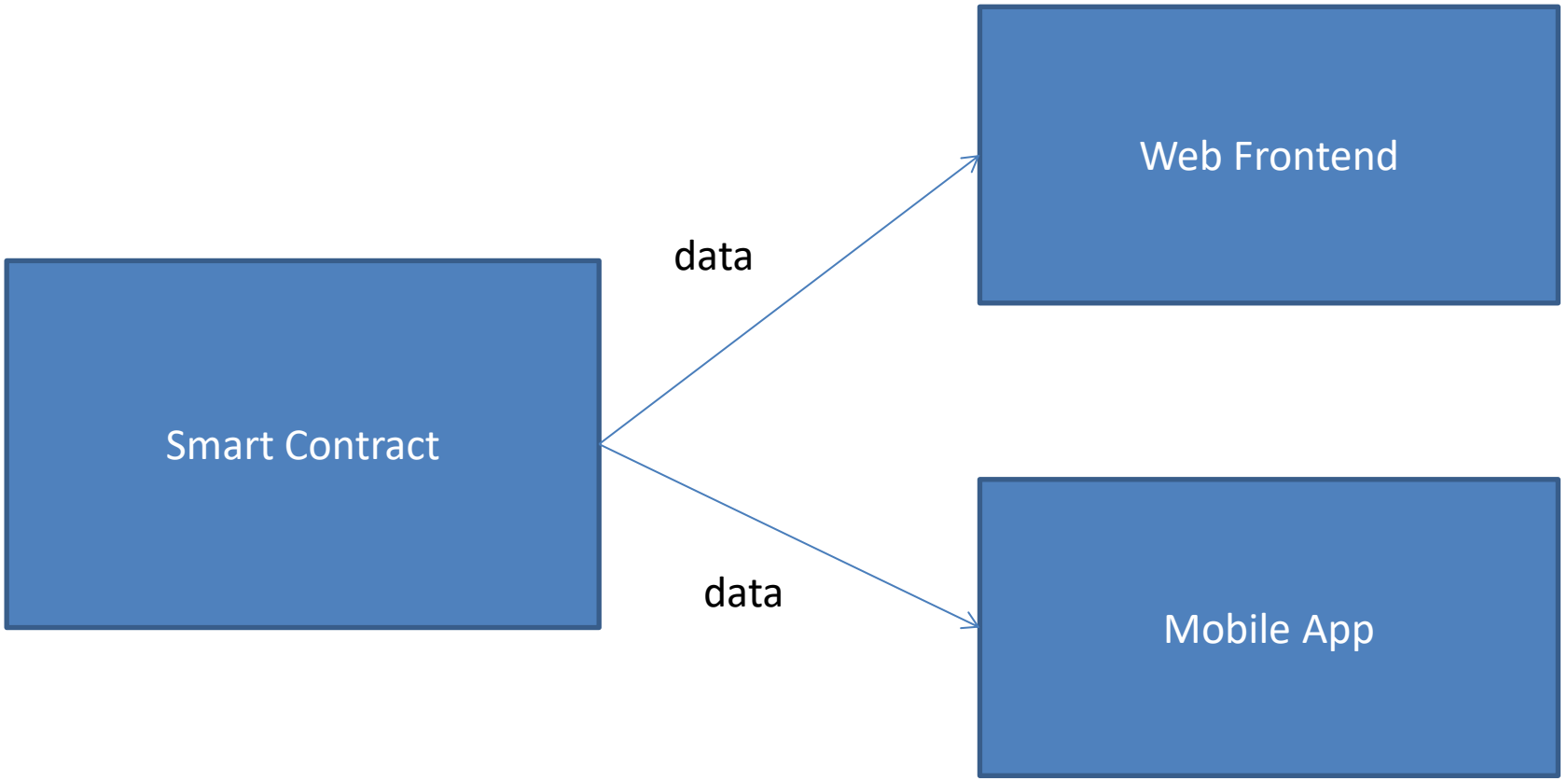
```
function foo() external
{
        balances[msg.sender] = 100;     //Create
        balances[msg.sender];   //Read
        balances[msg.sender] = 200;  //Update
        delete balances[msg.sender];  //Delete
        balances[addressnotexit] =>0;  //Default
}
```

Exotic Mappings

```
mapping(address => mapping(address => bool) approved;
//declaration
        approved[msg.sender][spender] = true;   //create
        approved[msg.sender][spender] ;  //read
        approved[msg.sender][spender] = false;  //update
        delete approved[msg.sender][spender];  //delete


mapping(address => uint[]) scores;
        // scores[msg.sender] = new uint[](2);
        scores[msg.sender].push(1);  //create
        scores[msg.sender].push(2);  // create
        scores[msg.sender][0];  //read
        scores[msg.sender][0] = 10;  //update
        delete scores[msg.sender][0];  //delete
```

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. An event generated is not accessible from within contracts, not even the one which have created and emitted them.

Events

```
event NewTrade(
        uint date,
        address from,      // use indexed for filtering
        address to,        // only 3 indexed in any event
        uint amount
);

function trade(address to, uint amount) external {
        emit NewTrade(now, msg.sender, to, amount);
}
```

Sending ether to smart contract

```
mapping(address => uint) balances;

function invest() external payable
{
If(msg.value < 1000)
        {
        revert();
        }
        balances[msg.sender] += msg.value;
}

function balanceOf() external view returns(uint)
{
return address(this).balance;
}
```

Sending Ether FROM a smart contract to another contract / address

```
address payable[] recipients;

function sendEther(address payable recipient) external
{
recipient.transfer(1 ether);

address a;
a = recipient;
a.transfer(1 ether);

msg.sender.transfer(1 ether);

recipient.send(1 ether);
}
```

Solidity

Code Layout

- **Indentation** − Use 4 spaces instead of tab to maintain indentation level. Avoid mixing spaces with tabs.
- **Two Blank Lines Rule** − Use 2 Blank lines between two contract definitions.
- **One Blank Line Rule** − Use 1 Blank line between two functions. In case of only declaration, no need to have blank lines.
- **Maximum Line Length** − A single line should not cross 79 characters so that readers can easily parse the code.
- **Wrapping rules** − First argument be in new line without opening parenthesis. Use single indent per argument. Terminating element); should be the last one.
- **Source Code Encoding** − UTF-8 or ASCII encoding is to be used preferably.
- **Imports** − Import statements should be placed at the top of the file just after pragma declaration.
- **Order of Functions** − Functions should be grouped as per their visibility.
- **Avoid extra whitespaces** − Avoid whitespaces immediately inside parenthesis, brackets or braces.
- **Control structures** − Braces should open on same line as declaration. Close on their own line maintaining the same indentation. Use a space with opening brace.
- **Function Declaration** − Use the above rule for braces. Always add a visibility label. Visibility label should come first before any custom modifier.
- **Mappings** − Avoid whitespaces while declaring mapping variables.
- **Variable declaration** − Avoid whitespaces while declaring array variables.
- **String declaration** − Use double quotes to declare a string instead of single quote.

Order of Layout

- Elements should be layout in following order.
  - Pragma statements
  - Import statements
  - Interfaces
  - Libraries
  - Contracts
- Within Interfaces, libraries or contracts the order should be as −
  - Type declarations
  - State variables
  - Events
  - Functions

Naming conventions

- Contract and Library should be named using CapWords Style. For example, SmartContract, Owner etc.
- Contract and Library name should match their file names.
  - In case of multiple contracts/libraries in a file, use name of core contract/library.
- Struct Names
- – Use CapWords Style like SmartCoin.
- Event Names
  - Use CapWords Style like Deposit, AfterTransfer.
- Function Names
  - Use mixedCase Style like initiateSupply.
- Local and State variables
  - Use mixedCase Style like creatorAddress, supply.
- Constants
  - Use all capital letters with underscore to seperate words like MAX_BLOCKS.
- Modifier Names
  - Use mixCase Style like onlyAfter.
- Enum Names
  - Use CapWords Style like TokenGroup.

Constructors

- A constructor is an optional function declared with the constructor keyword which is executed upon contract creation, and where you can run contract **initialisation code**.
- Before the constructor code is executed, state variables are initialised to their specified value if you initialise them inline, or their default value if you do not.
- After the constructor has run, the final code of the contract is deployed to the blockchain. The deployment of the code costs **additional gas** linear to the length of the code.
- If there is no constructor, the contract will assume the default constructor, which is equivalent to constructor() {}.

JSON Output

- The storage layout of a contract can be requested via the standard JSON interface.
- The output is a JSON object containing **two keys, storage and types**. The storage object is an array where each element has the following form:

```
{
"astId": 2,
"contract": "fileA:A",
"label": "x",
"offset": 0,
"slot": "0",
"type": "t_uint256"
}
```

- The example above is the storage layout of **contract A { uint x; }** from source unit **fileA**

- **astId** is the id of the AST node of the state variable's declaration
- **contract** is the name of the contract including its path as prefix
- **label** is the name of the state variable
- **offset** is the offset in bytes within the storage slot according to the encoding
- **slot** is the storage slot where the state variable resides or starts. This number may be very large and therefore its JSON value is represented as a string.
- **type** is an identifier used as key to the variable's type information
  - The given type, in this case t_uint256 represents an element in types, which has the form:

```
{
  "encoding": "inplace",
  "label": "uint256",
  "numberOfBytes": "32",
}
```

Functions

- Functions can be defined inside and outside of contracts.
- Functions outside of a contract, also called “free functions”, always have implicit internal visibility. Their code is included in all contracts that call them, similar to internal library functions.
- Functions defined outside a contract are still always executed in the context of a contract.
- They still can call other contracts, send them Ether and destroy the contract that called them, among other things.
- The main difference to functions defined inside a contract is that free functions do not have direct access to the variable **this**, storage variables and functions not in their scope.

Function Parameters and Return Variables

- Functions take typed parameters as input and may, unlike in many other languages, also return an arbitrary number of values as output.

Function Parameters

- Function parameters are declared the same way as variables, and the name of unused parameters can be omitted.

Return Variables

- Function return variables are declared with the same syntax after the **returns** keyword.

Returning Multiple Values

- When a function has multiple return types, the statement return (v0, v1, ..., vn) can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an implicit conversion.

Pure Functions

- Functions can be declared pure in which case they promise not to read from or modify the state.
- In particular, it should be possible to evaluate a pure function at compile-time given only its inputs and msg.data, but without any knowledge of the current blockchain state.
- This means that reading from immutable variables can be a non-pure operation.
- In addition to the list of state modifying statements explained above, the following are considered reading from the state:
  - Reading from state variables.
  - Accessing address(this).balance or <address>.balance.
  - Accessing any of the members of block, tx, msg (with the exception of msg.sig and msg.data).
  - Calling any function not marked pure.
  - Using inline assembly that contains certain opcodes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C
{
function f(uint a, uint b) public pure returns (uint)
{
return a * (b + 42);
}
}
```

Special Functions

Receive Ether Function

- A contract can have at most one receive function, declared using receive() external payable { ... } (without the function keyword).
- This function cannot have arguments, cannot return anything and must have external visibility and payable state mutability. It can be virtual, can override and can have modifiers.
- The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers (e.g. via .send() or .transfer()). If no such function exists, but a payable fallback function exists, the fallback function will be called on a plain Ether transfer. If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through regular transactions and throws an exception.

Fallback Function

- 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.
  - It has no arguments
  - It can not return any thing.
  - It can be defined one per contract.
  - If not marked payable, it will throw exception if contract receives plain ether without data.

```
pragma solidity ^0.5.0;

contract Test {
   uint public x ;
   function() external { x = 1; }
}

contract Sink {
   function() external payable { }
}
```

Libraries

- Libraries are similar to Contracts but are mainly intended for reuse.
- A Library contains functions which other contracts can call.
- Solidity have certain restrictions on use of a Library.
- Following are the key characteristics of a Solidity Library.
  - Library functions can be called directly if they do not modify the state. That means pure or view functions only can be called from outside the library.
  - Library can not be destroyed as it is assumed to be stateless.
  - A Library cannot have state variables.
  - A Library cannot inherit any element.
  - A Library cannot be inherited.

Using For

- The directive **using A for B**; can be used to attach functions (A) as member functions to any type (B).
- These functions will receive the object they are called on as their first parameter.
- It is valid either at file level or inside a contract, at contract level.
- The first part, A, can be one of:
  - a list of file-level or library functions (using {f, g, h, L.t} for uint;) - only those functions will be attached to the type.
  - the name of a library (using L for uint;) - all functions (both public and internal ones) of the library are attached to the type
- At file level, the second part, B, has to be an explicit type (without data location specifier).
  - Inside contracts, you can also use using L for *;, which has the effect that all functions of the library L are attached to *all* types.

Error Handling

Solidity provides various functions for error handling. Generally when an error occurs, the state is reverted back to its original state. Other checks are to prevent unauthorized code access. 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.

Use Cases of Solidity

Voting

- Currently, voting deals with numerous issues including manipulation of data, fake voters, alteration in voting machines and booth capturing. Solidity smart contacts can be created and deployed to make the voting process transparent and streamlined.

Blind Auctions

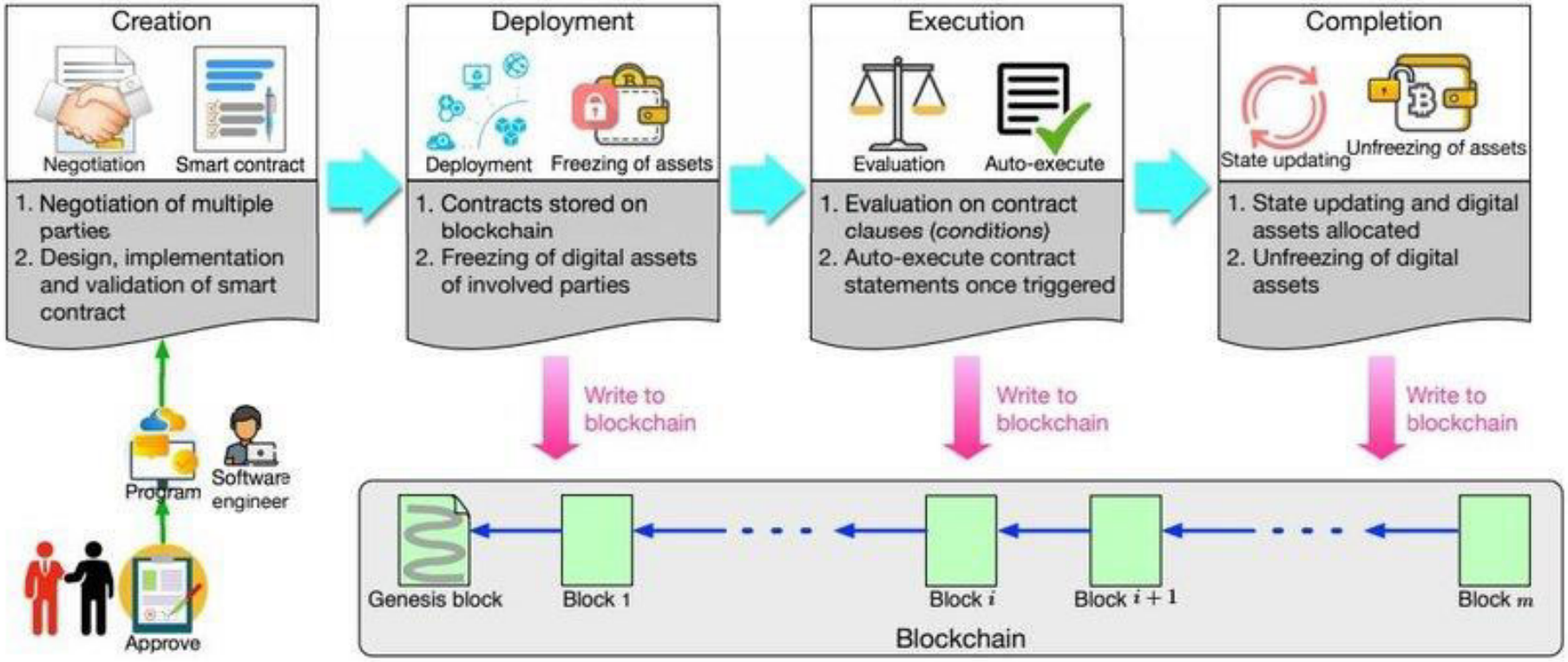
- In an open auction, individuals can view each other's bid which leads to disputes and frauds. Using Solidity Smart Contracts, the blind auction can be designed where users cannot see what someone bid until it ends.

Crowdfunding

- Crowdfunding done via smart contracts can solve issues like the commission of a third party and management of data. Solidity smart contracts for crowdfunding do not require centralized systems to build trust, thereby reducing additional costs.

SMART CONTRACT LIFE CYCLE

Smart Contract Life Cycle



- **Creation Phase:** The creation phase comprises iterative contract negotiations and a phase of implementation. First, the parties must agree on the scope and objectives of the contract. This is comparable to standard contract negotiations and can take place online or in person. Contracts. During this phase, the following are carried out: (a) Multiple-party bargaining; (b) Smart system design, implementation, and validation of Smart contracts
- **Freeze / Deployment :** A network of computers known as nodes perform the validation of transactions on a blockchain. The miners of the blockchain are at these locations. To prevent the ecosystem from being flooded with smart contracts, miners must be compensated for this service with a small fee. A network of computers known as nodes performs the validation of transactions on a blockchain. The miners of the blockchain are at these locations. To prevent the ecosystem from being flooded with smart contracts, miners must be compensated for this service with a small fee.
- **Execution:** Contracts recorded on the distributed ledger are read by all nodes participating in the network. The authentication nodes validate the integrity of a smart contract, while the code is executed by the interference engine (or compiler) of the smart contract. When inputs for execution from one party are received in the form of coins (commitment to products via coins), the interference engine generates a transaction triggered by the met criterion. The execution of the smart contract generates a new set of transactions and a new contract state. Discoveries and new state data are added to the distributed ledger and validated using the consensus procedure.
- **Finalize/Complete:** Following the execution of the smart contract, the resulting transactions and updated state information are recorded on the distributed ledger and validated through the consensus procedure. The pledged digital assets are transferred (the assets are unfrozen), and the contract is signed to affirm all transactions.

USE CASES

Real Estate – Property Transfers

Scenario for the Contract Owner:

- Contract Deployment: The government, as the contract owner, deploys the smart contract on the blockchain to create a decentralized land registry system.
- Land Addition: The contract owner adds lands to the contract by calling the addLand function with the location and cost of each land.
- Agreement on Approvers: The contract owner facilitates the agreement between buyers and sellers on a common approver for each land transfer by calling the agreeOnApprover function.

- **Scenario for a Buyer:**
- Finding a Land: The buyer identifies a land available for sale in the contract by checking the land details using the getLand function.
- Agreeing on an Approver: The buyer communicates with the seller, and both parties agree on a common approver for the land transfer.
- Initiating Transfer: The buyer calls the initiateTransfer function, providing the land ID and the agreed approver's address as arguments.
- Approval: The buyer then approves the transfer by calling the approveTransferAsBuyer function.
- Completion of Transfer: If the seller also approves the transfer by calling the approveTransferAsSeller function, the transfer is completed, and the land ownership is transferred to the approver.

- **Scenario for a Seller:**
- Ownership Verification: The seller confirms ownership of the land by checking the token representing the land and ensuring it belongs to them.
- Agreeing on an Approver: The seller communicates with the buyer, and both parties agree on a common approver for the land transfer.
- Initiating Transfer: The seller calls the initiateTransfer function, providing the land ID and the agreed approver's address as arguments.
- Approval: The seller then approves the transfer by calling the approveTransferAsSeller function.
- Completion of Transfer: If the buyer also approves the transfer by calling the approveTransferAsBuyer function, the transfer is completed, and the land ownership is transferred to the approver.

The selling and transferring of a land token works in the smart contract:

1. Selling the Land (Agreeing on an Approver)

– Let's say a seller wants to sell their land to a buyer, and both parties agree on an approver before starting the transfer process.

– The seller calls the `agreeOnApprover` function and provides the `landID` of the land they wish to sell, along with the address of the agreed-upon `approver`.

– The `agreeOnApprover` function verifies that the caller is the owner of the land (the seller) and sets the `approver` for that specific land.

2. Initiating the Transfer (Buyer Initiates Transfer)

– After agreeing on the approver, the buyer takes the initiative to start the land transfer:

– The buyer calls the `initiateTransfer` function and provides the `landID` of the land they intend to buy, along with the address of the agreed-upon `approver`.

– The `initiateTransfer` function checks if the caller is the buyer and whether the land is available for sale.

– The function updates the `buyerApproved` flag to indicate that the buyer has initiated the transfer.

3. Approval from Seller and Buyer

– Both the seller and the buyer need to approve the transfer before it can be completed:

– The seller approves the transfer by calling the `approveTransferAsSeller` function and providing the `landID`.

– The buyer approves the transfer by calling the `approveTransferAsBuyer` function and providing the `landID`.

4. Completing the Transfer

– Once both the buyer and the seller have approved the transfer, the transfer is completed:

– The `checkAndCompleteTransfer` function is invoked, checking whether both `buyerApproved` and `sellerApproved` are set to true.

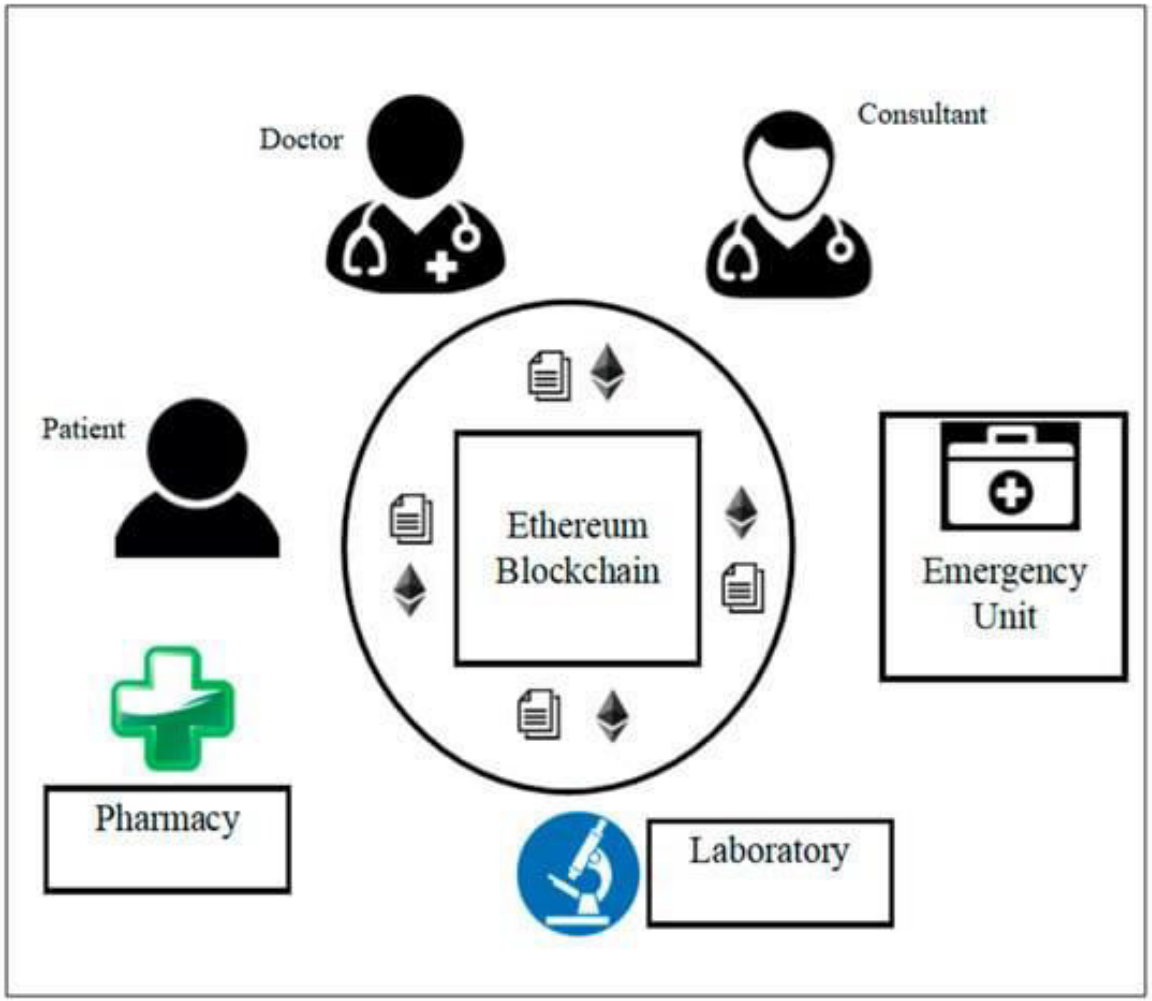
– If both approvals are true, the function transfers the ownership of the land from the seller to the agreed-upon `approver`.

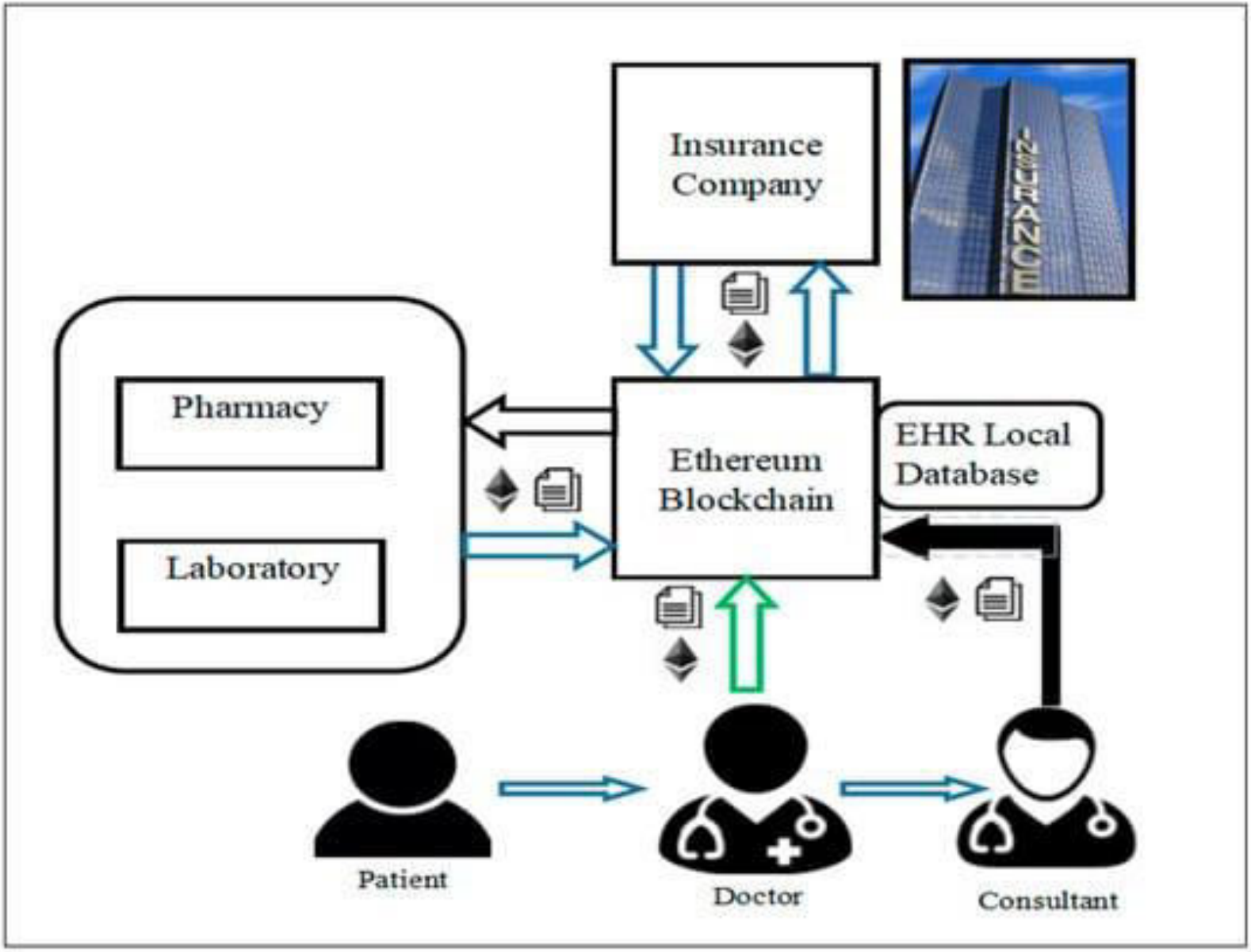
– The `wantSell` flag for the land is set to 0, indicating that the land is no longer available for sales.

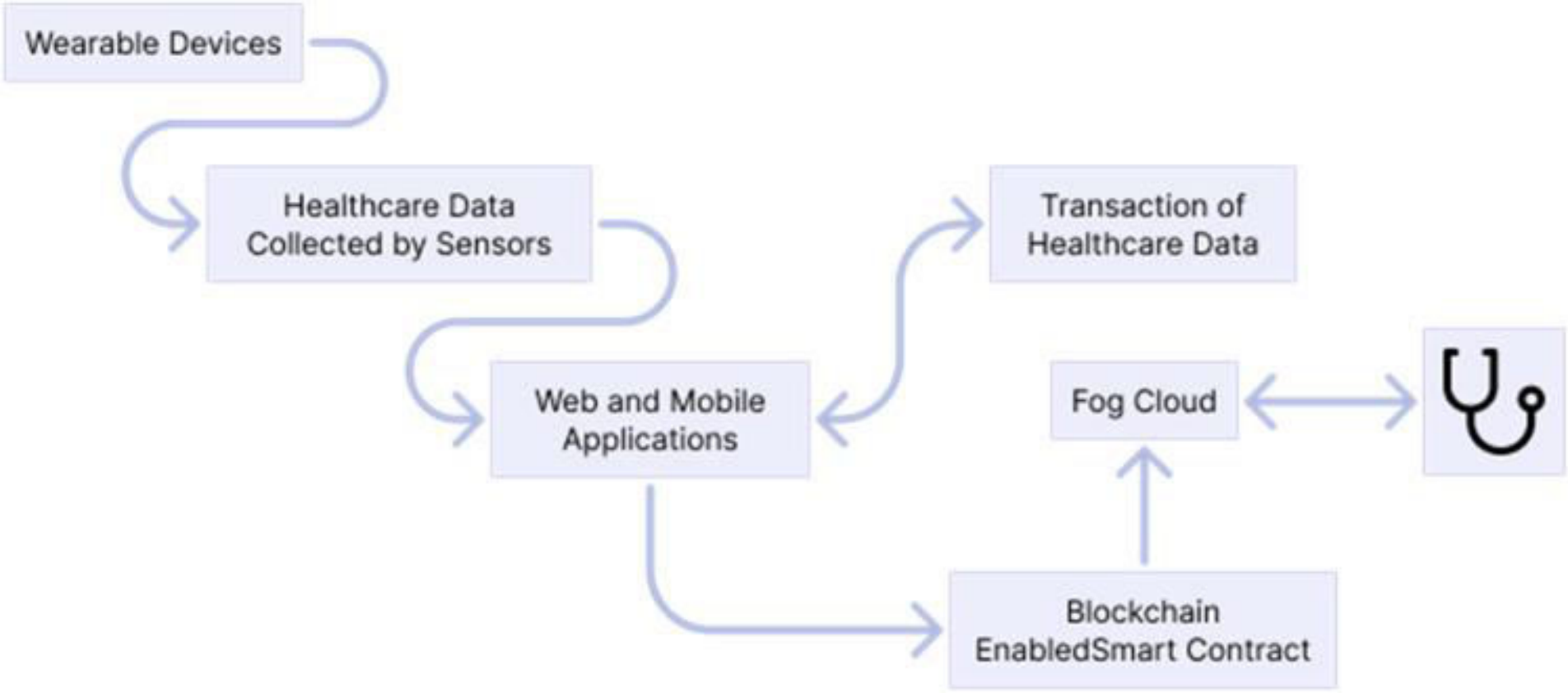
– The agreed-upon `approver` takes ownership of the land token, and the cost of the land is transferred to the seller.

In this way, the land token is successfully sold and transferred from the seller to the buyer, with the involvement of the agreed-upon approver.

Use Case – Healthcare Industry




Insurance Company
Pharmacy
Laboratory
Ethereum Blockchain
EHR Local Database
Patient
Doctor
Consultant


Wearable Devices
Healthcare Data Collected by Sensors
Transaction of Healthcare Data
Web and Mobile Applications
Fog Cloud
Blockchain EnabledSmart Contract

Policy sale (insurer and customer)

Sale of policy to the customer on basis of:

- policy as a 'smart contract'
- customised pricing offered to the customer (based on medical history/ behavioural traits/wearables, etc.)
- clearly explained policy terms
- loading charges, policy exclusions
- benefits entitlement, payout limits.

Underwriting (insurer)

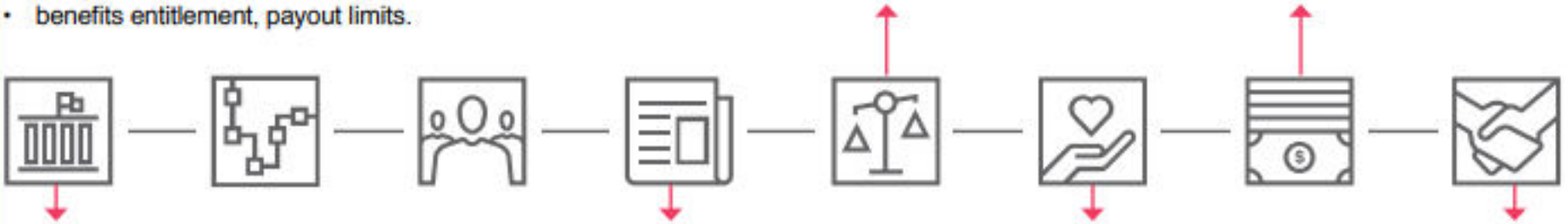
Swift and automated process which includes:

- rule-based underwriting with immutable and approved guidelines
- mapping customer data to policy conditions in real time and offering customised pricing.

Underwriting (insurer)

Cashless payout settlement based on:

- smart contract driven settlement through auto e-invoicing generation
- - customer treatment and agreed tariffs.



Insurer-hospital agreement

The insurer and the hospital agree on an arrangement provided they have in place:

- a tariff agreement
- a packages agreement
- the International Classification of Diseases (ICD) coding system
- a contract validity.

E-health records (insurer)

Accessing the customer's digital identity using:

- KYC details
- demographic and medical history details
- nominee information
- payout bank details.

Claims (customer and provider)

Fast and efficient claims by accessing customer-related information through blockchain, using:

- automated customer authentication
- customer entitlement verification
- updated customer records.

Reinsurance (insurer and reinsurer)

Treaties and claims in accordance with:

- registering various treaties with different reinsurers (smart contract)
- automation of straightforward claims triggered by smart reinsurance contracts.