Course Outline

1. Cryptocurrency and Block chain
2. Delving into BlockChain
3. Bitcoin and Block chain
4. Bitcoin Mining
5. Ethereum
6. Setting up private Blockchain Environment using Ethereum Platform
7. Hyperledger
8. Setting up Development Program using Hyperledger composer
9. Create or Deploy our private Blockchain on Multi chain
10. Prospect of Blockchain

IntelliPaat

Ethereum

Agenda

At the end of this session you will be able to:

- Understand Ethereum
- Define Smart Contracts
- Identify Cryptocurrency used in Ethereum
- Describe Transactions in Ethereum
- Define Consensus Mechanism in Ethereum
- List Development Technologies
- Identify Ethereum Clients
- Define Platform Functions
- Understand Solidity
- Describe Solidity Operators and Functions

Ethereum

Ethereum

Let us know more about Ethereum.

Ethereum

- Ethereum was conceptualized by Vitalik Buterin in November 2013
- Ethereum is a distributed open blockchain network
- The key idea proposed was the development of a Turing-complete language that allows the development of smart contracts for blockchain and decentralized applications

Ethereum

Ethereum aims to enable innovations in four key areas:

- Currency Issuance
- Decentralized Autonomous Organizations (DAO)

- Smart Contracts
- Smart Property

Bitcoin-Ethereum: comparison

	Bitcoin	Ethereum
Idea	Digital money	World Computer
Founder	Satoshi Nakamoto(unknown)	Vitalik Buterin and team
Scripting language	Turing incomplete	Turing complete
Release date	Jan 2009	July 2015
Coin Release Technique	Early mining	Through ICO
Average block time	~10 minutes	~12-15 seconds

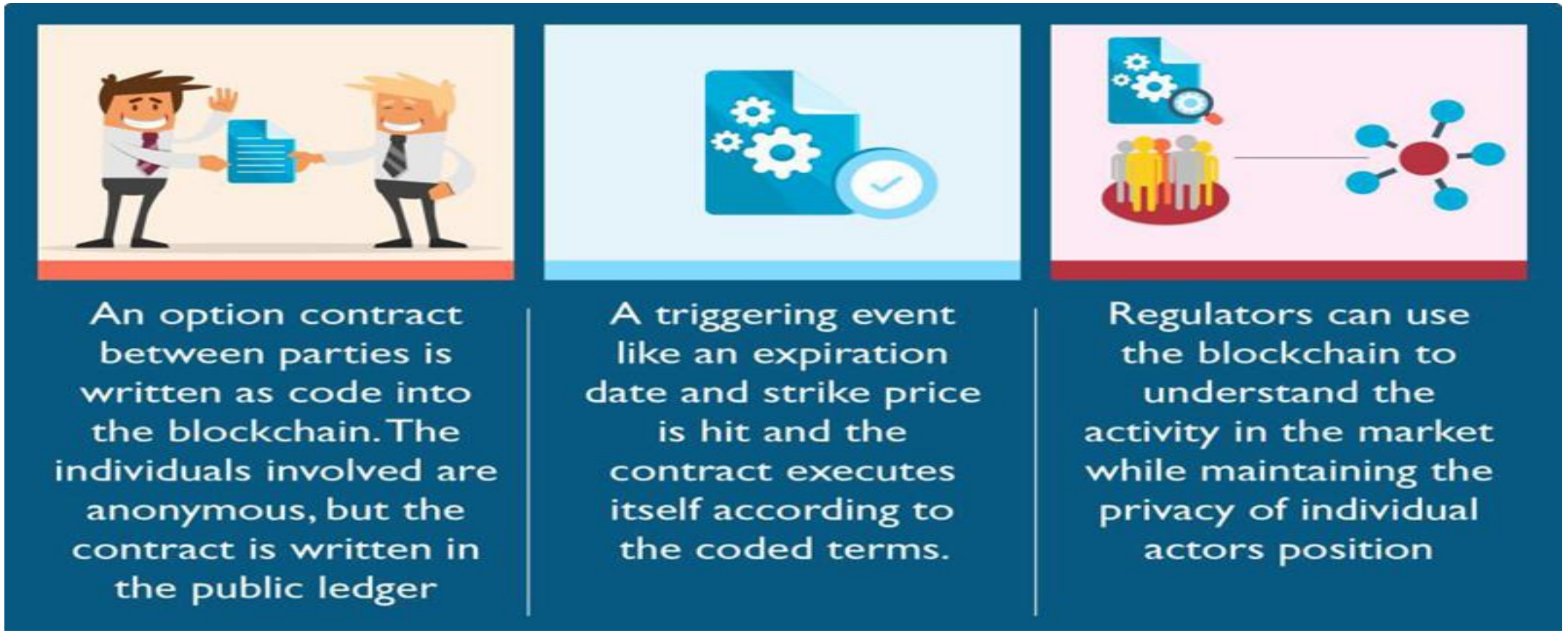
Smart Contracts

Smart Contracts

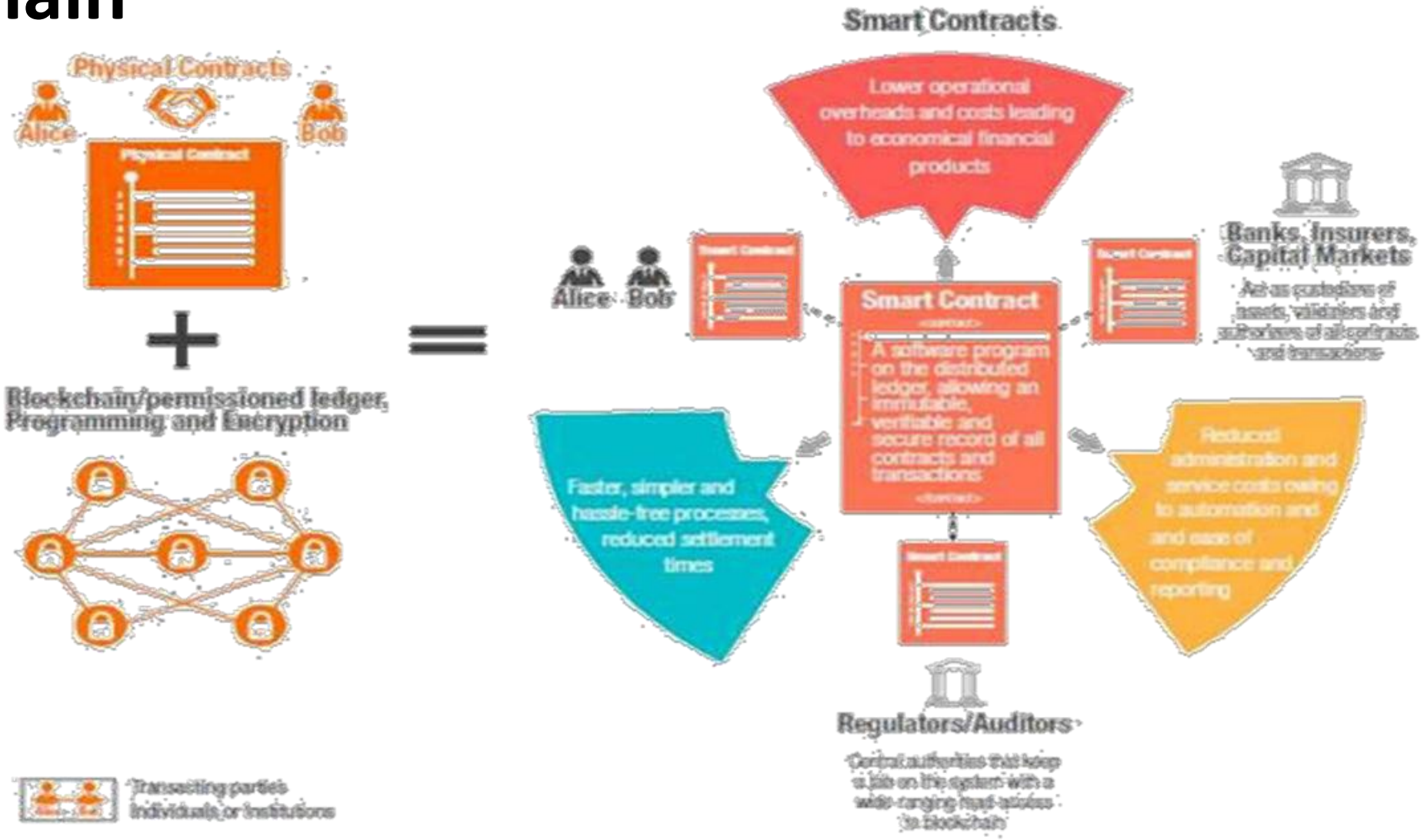
> The basic building blocks of programs written for the ethereum platform are called “smart contracts”. Let’s understand what are smart contracts.

Smart Contracts

- A smart contract is a computerized transaction protocol that executes the terms of a contract
- When running on the blockchain a smart contract becomes like a self-operating computer program that automatically executes when specific conditions are met



How Smart contracts work in a permission Blockchain



Cryptocurrency used in Ethereum

Ether

Ethereum is incomplete without cryptocurrency.
Let’s have a look at the cryptocurrency used in Ethereum.

Ether

The esteem token of the Ethereum blockchain is called

Ether

It is recorded under the code ETH and exchanged on digital currency trades

It is also used to pay for transaction fees and computational services on the Ethereum network

Every time a contract is executed, Ethereum consumes tokens which is termed as ‘gas’ to run the computations

Denomination Table of Ether

Unit	Wei Value	Wei
Wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

Ether is used to purchase gas, a crypto fuel for Ethereum

> Let’s understand what gas is in the Ethereum system and why is it necessary.

Gas in Ethereum

- Gas is required to be paid for each activity performed on the ethereum blockchain
- A transaction fee is charged as some amount of Ether and is
- taken from the account balance of the transaction originator
- A fee is paid for transactions to be included by miners
- The more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block
- Providing too little gas will result in failed transaction

Ether buys Gas to fuel up the EVM

Assessing Transaction Cost

Transaction cost can be estimated using the following formula:

Total cost = gasUsed * gasPrice

gasUsed is the total gas that is supposed to be used by the transaction during the execution

gasPrice is specified by the transaction originator as an incentive to the miners

Transactions in Ethereum

Transactions in Ethereum

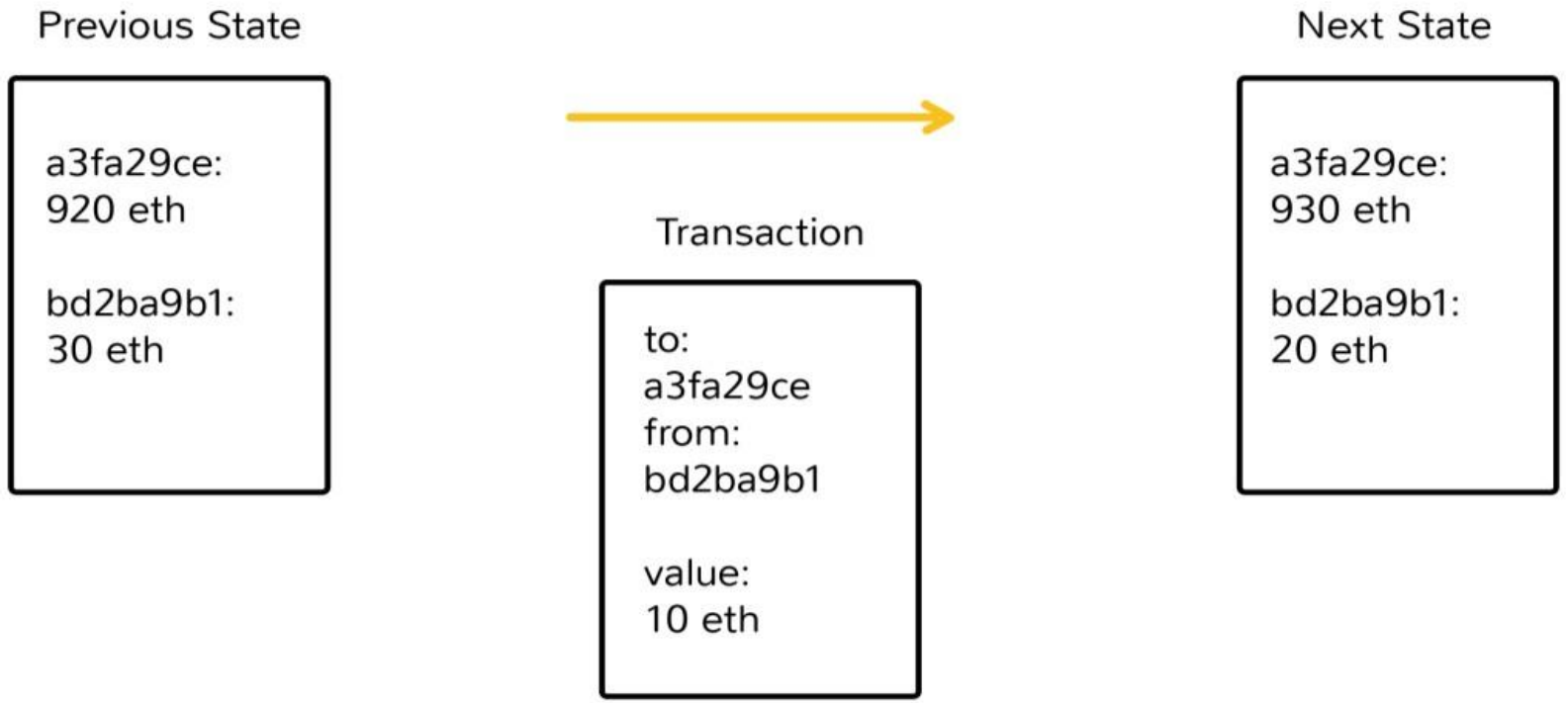
The most notable difference between the Bitcoin and Ethereum blockchain is that ethereum blocks contain both a transaction list and the most recent “state” of the ledger of these transactions

Let’s have a look at the transactions and Accounts in Ethereum Blockchain.

Transactions

The term “transaction” is used in Ethereum to refer to the signed data package that contains a message to be sent from an externally owned account to another account on the blockchain

Ethereum blocks contain both a transaction list and the most recent “state” of the ledger of these transactions



Accounts are one of the main building blocks of the Ethereum blockchain

Types of Accounts

Accounts play a central role in Ethereum. There are two types of accounts: externally owned accounts (EOAs) and contract accounts.

- Externally owned accounts (EOAs): Defined as the basic form of account, EOAs interact with and generate updates on the Ethereum blockchain
- Contracts: Programmatically execute when they receive instructions in the form of a transaction from an EOA. Contracts can push or pull funds, and request these actions from other contracts, calling on the code to perform dynamic actions

- If we restrict Ethereum to only externally owned accounts and allow only transactions between them, we arrive at an “altcoin” system that is less powerful than bitcoin itself and can only be used to transfer ether.

- The state of all accounts is the state of the Ethereum network, which is updated with every block and which the network really needs to reach a consensus about

How approval occurs in Ethereum Blockchain

Let’s see how transactions happens and a block is created in Ethereum Blockchain

Validators (or Miners) in ethereum

Just like in Bitcoin, transactions are approved by the Miners
Anyone can take on the role of a miner and approve the block of transactions
It takes around 12-15 seconds approximately for a miner to approve or mine a block

At a theoretical level, a miner performs the following functions:

- Listens for the transactions broadcasted on the Ethereum network and determines the transactions to be processed
- Determines stale blocks called Uncles or Omners and includes them in the block
- Updates the account balance with the reward earned from successfully mining the block
- Finally, a valid state is computed and block is finalized, which defines the result of all state transitions

Mining Reward

- In order to incentivise miners for supporting the Ethereum network, a block reward is granted to the lucky miners who generate the correct block
- Currently, the block reward is set at 5 Ether for every successful block that is mined
- This will contribute to the inflation in the number of Ether available and hence, the value of each Ether

Consensus Mechanism in Ethereum

Consensus Mechanism in Ethereum

For any distributed computing system to properly function, there needs to be a mechanism by which the entire network can come to an agreement on its state, or how its token supply is divided among registered addresses on the network

Let’s have a look at the consensus mechanism used in Ethereum.

Consensus Algorithm

- At the time of writing, Ethereum uses a similar Proof-of-work protocol known as Ethash
- Similar to bitcoin, the core idea behind mining is to find a nonce that once hashe the result in a predetermined difficulty level
- It uses different cryptographic primitive for its hashing function, known as SHA-3, rather than SHA-256

Ethash is designed to make Ethereum both resistant to the high-powered mining chips that currently dominate the bitcoin industry, and more accessible to "light" client implementations that allow users to use Ethereum without needing to first download the Ethereum blockchain to their device, however light clients downloads headers of the blocks for reference

Ethereum does far more than processing transactions

The Ethereum Blockchain is intended to do far more than process shared transactions. It is designed to execute complex code, where the functionality is only restricted by the imagination of its developers and accessible resources

> The complex codes are executed in Ethereum Virtual Machine.

Ethereum – Releases

Metropolis

Metropolis is next major Ethereum upgrade. Metropolis was introduced on github for testing and its just a matter of time before it goes live. Not only introducing further protocol upgrades and opening the door for casper, but also new interfaces for non technical users. This will help gain further adaptation and also it will help developers create easy to use apps

Serenity

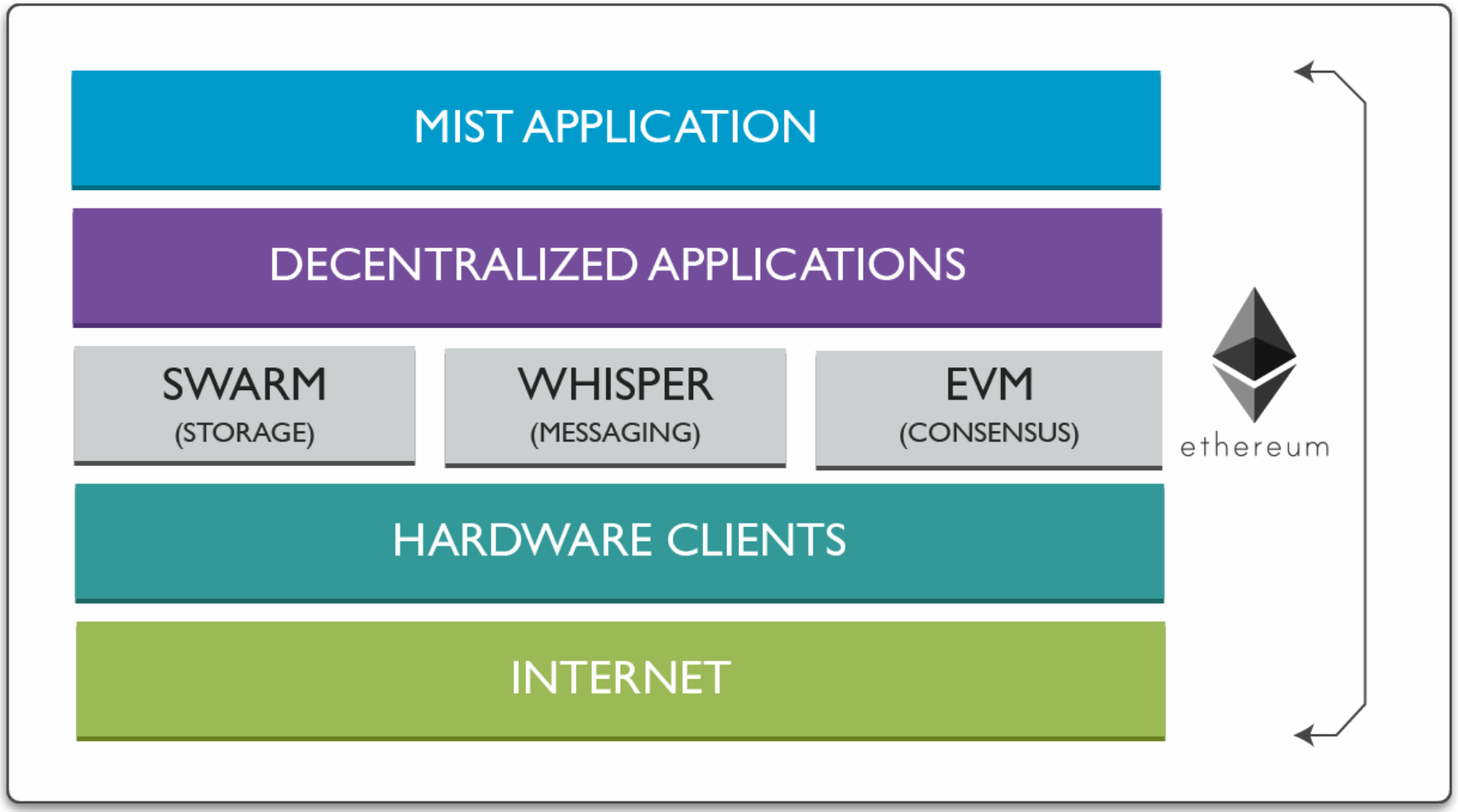
Serenity is the last phase in Ethereum network. Release was planned for the start of 2018, but we might have some delays, as usually. Serenity goal is to switch from proof-of work to proof-of-stake, by using a method called Casper

Development Technologies

Development Technologies that seek to help Network

> In addition to the main Ethereum blockchain protocol, there are also supporting technologies in development that seek to help the network, and components built on the network, run more efficiently

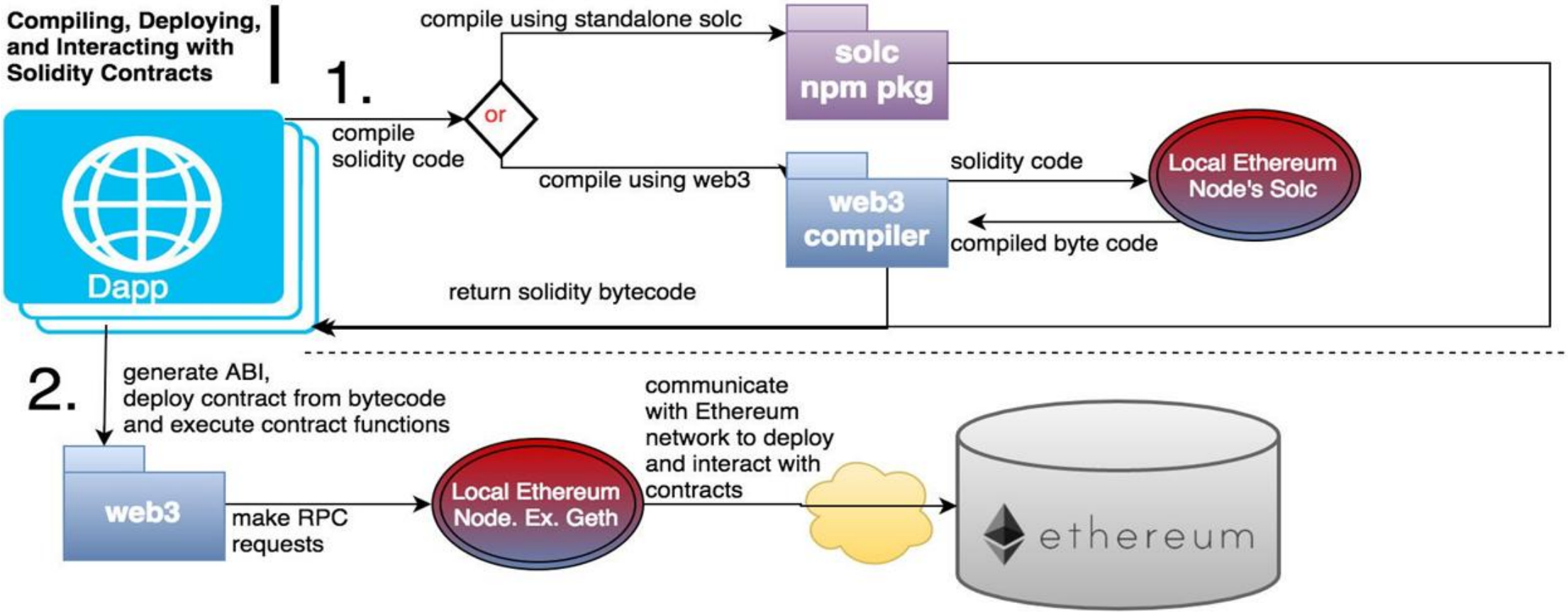
Supporting Technologies



Web 3.0 Technology Stack

Web3.js

- The key connection between the Ethereum network and your DApp
- Web3 allows you to compile, deploy, and interact with your smart contracts



Mist Application

- Styled as a decentralized application discovery tool, Mist is meant to serve as a wallet for smart contracts that features a graphical user interface
- A desktop application is used to communicate with your node
- Allows users to dynamically set transaction fees and manage custom tokens

Mist Browser v/s Ethereum Wallet

Mist DApp Browser

- Mist is the browser for decentralized web apps
- Mist is still in heavy development (it's not recommended to visit untrusted DApps until the full security audit is done).. The current release allows you to open any Ethereum DApp with the Mist Browser

Ethereum Wallet DApp

- The Ethereum Wallet, also known as the Meteor DApp Wallet
- These releases are therefore called Ethereum Wallet as it only offers a bundle of the Mist browser with a single DApp: the wallet
- The future, with Metropolis release, will provide a full Mist Browser able to open any DApp available out there

Ethereum Clients

Ethereum Clients to Interact with Contracts

Developers write up the contracts, which are then converted into the EVM bytecode via a EVM complier and uploaded onto the blockchain using an Ethereum client.

“There are various Clients supporting the Ethereum network. Let’s see what Clients are and what purpose they serve.”

Clients of Ethereum

> Since the start of the Ethereum project, there have been multiple client implementations across a range of different operating systems and programming languages. This client diversity is essential for the long-term health of the Ethereum network

Ethereum clients are softwares that allow users to:

- Approves transactions/blocks
- Create/manage accounts on Ethereum
- Send/receive transactions from/to your Ethereum accounts
- Deploy smart contracts onto the blockchain
- "Mine" Ether on the Ethereum Blockchain

List of Ethereum Clients

Client	Language	Developers
Go-Ethereum (Geth)	Go	Ethereum Foundation
Parity	Rust	Ethcore
Cpp-Ethereum	C++	Ethereum Foundation
Pyethapp	Python	Ethereum Foundation
Ethereumjs-lib	Javascript	Ethereum Foundation
Ethereum(J)	Java	<ether.camp>
Ruby- ethereum	Ruby	Jan Xie
EthereumH	Haskell	Blockapps

Go-Ethereum

The go-ethereum client is commonly referred to as geth, which supports the running a full Ethereum node implemented in the Go language.

Current features:

- **Integrated with mist browser**
- **Light client (Beta)**
- **Swarm (POC)**
- **Whisper (POC)**

Platform Functions

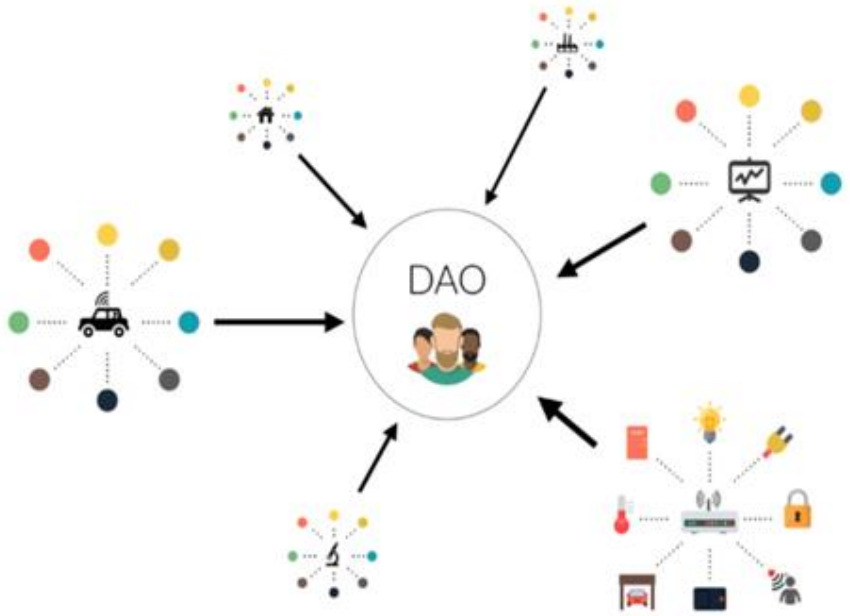
Platform Functions in Ethereum

While the architecture of the network is certainly impressive, it's what's built on these intricate components that truly illustrates Ethereum's potential

Let's have a look at the various platform functions in Ethereum

DAO (Decentralized Autonomous Organization)

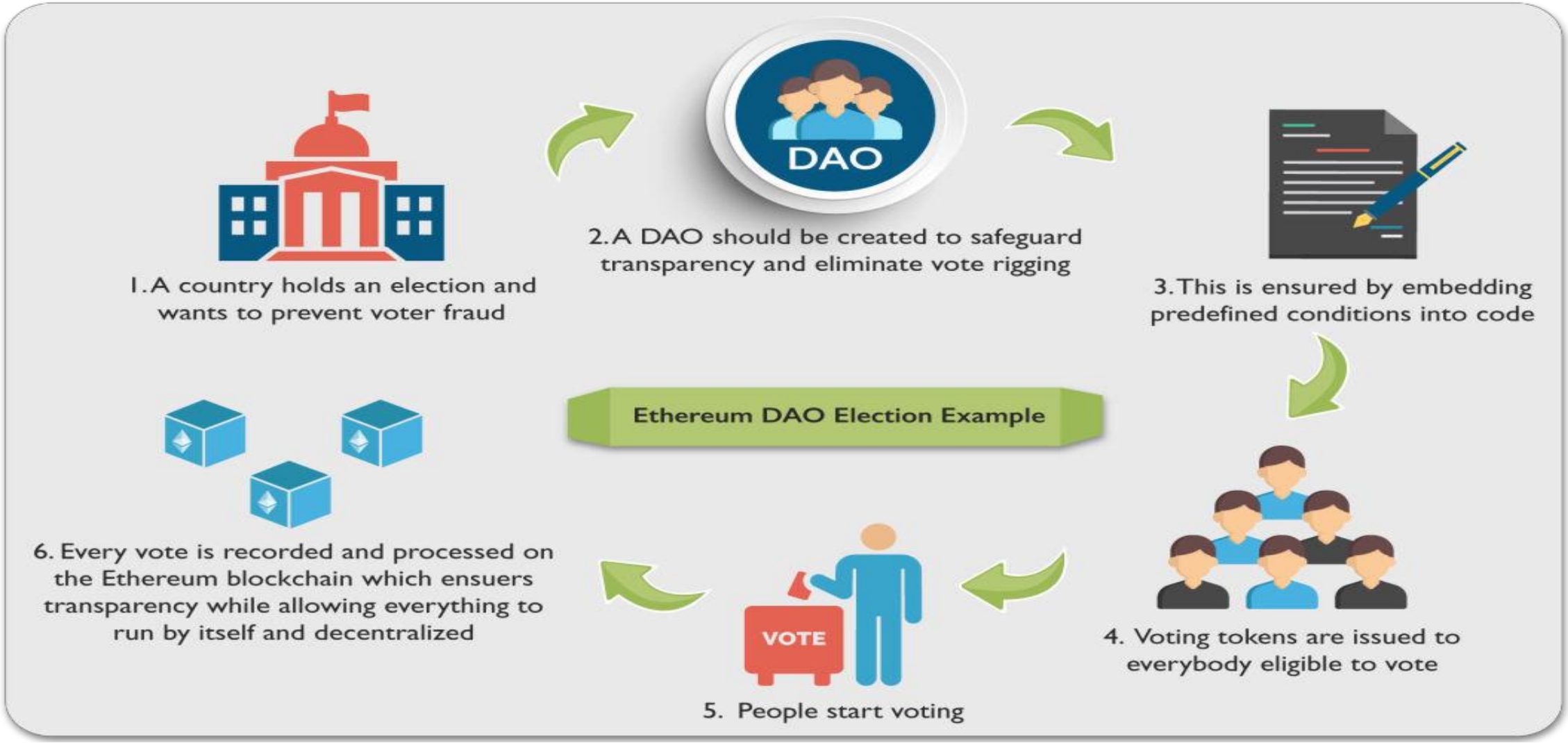
- DAOs are organizations that exist entirely on a blockchain and are governed by its protocols
- A unison of many long-term smart contract between many people
- DAOs are designed to hold onto assets and use a kind of voting system to manage their distribution
- Two or more entities in DAO can interact with each other in a fully decentralized and automated fashion
- DAOs comprise a global network of nodes and members that all work together
- In the DAO, Each action or vote is represented by some form of transaction in the Blockchain



Working of DAO

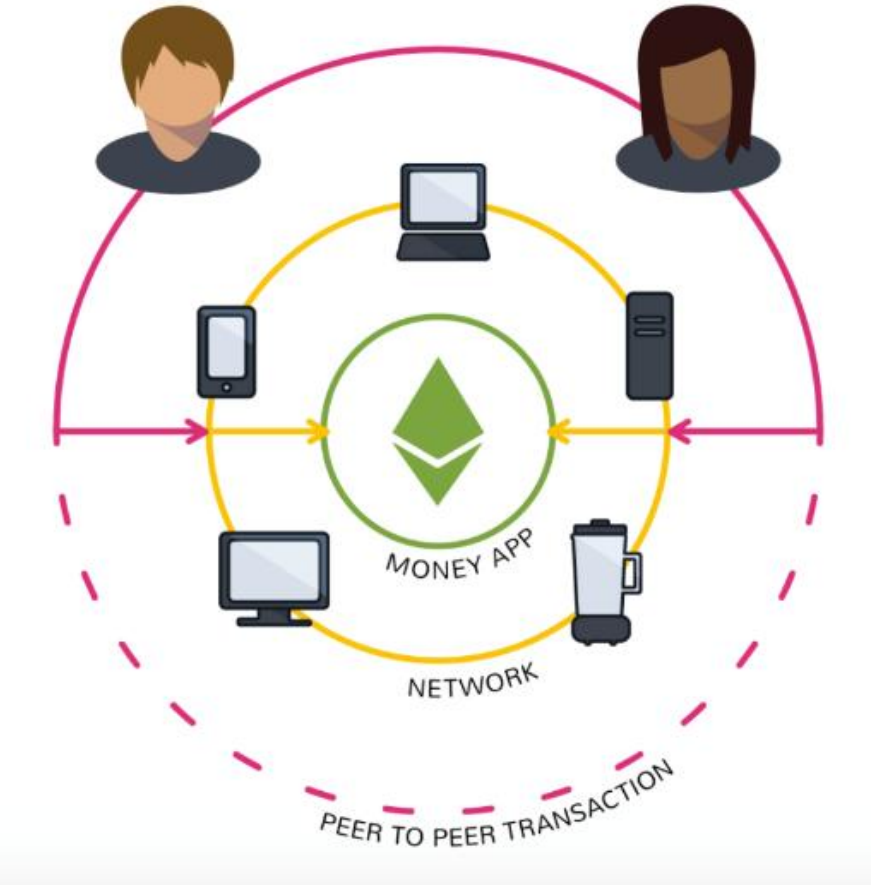
- A group of people writes a smart contract to govern the organization
- People add funds to the DAO and are given tokens that represent ownership
- The DAO begins to operate by having members to propose how to spend the money
- The members vote on these proposals
- When the predetermined time has passed and the predetermined number of votes has accrued, the proposals passes or fails
- Individuals act as contractors to service the DAO

Example of a DAO



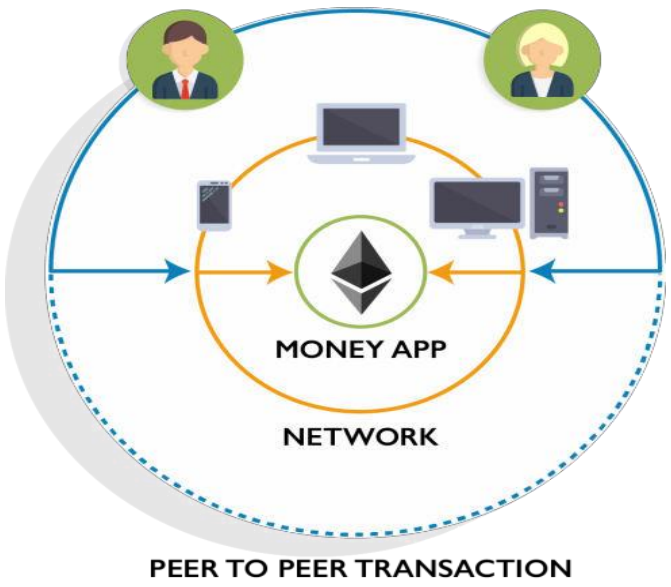
DApps (Decentralized Applications)

- Decentralized applications or DApps are computer applications that operate over a blockchain enabling direct interaction between end users and providers
- Dapps can be comprised of single DAO or even a series of DAOs that work together to create an application
- A decentralized application has an unbounded number of participants on all sides of the market

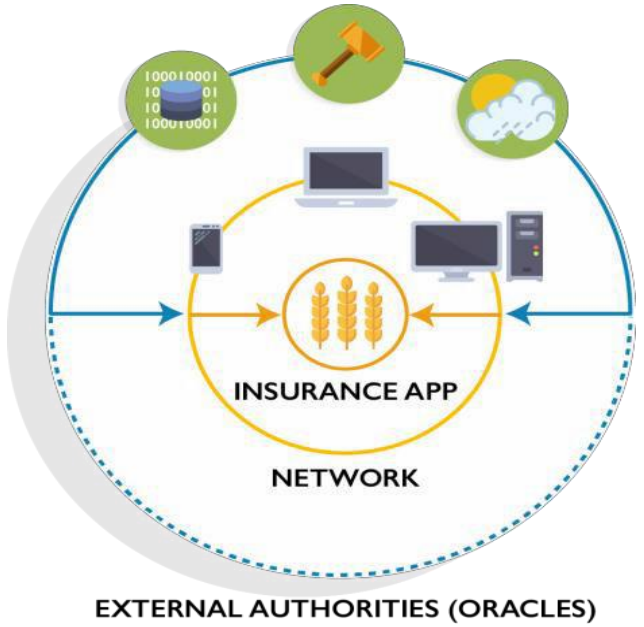


ETHERIA (www.etheria.world), a Minecraft-like game

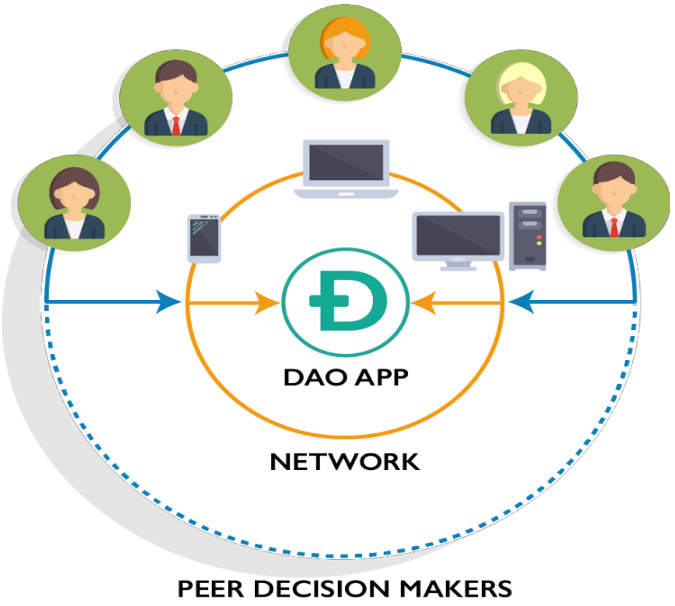
Types of DApps



PEER TO PEER TRANSACTION



EXTERNAL AUTHORITIES (ORACLES)



PEER DECISION MAKERS

1st Type: A user may need to exchange ether as a way to settle a contract with another user, using the network's distributed computer nodes as a way to facilitate the distribution of this data

2nd Type: The app mixes money with information from outside the blockchain. For example, a crop insurance application that's dependent on an outside weather feed. (Say a farmer buys a derivative that automatically pays out if there's a drought that impacts his work.)

3rd Type: DAO
The goal is form a leaderless company, program rules at the beginning about how members can vote and how to release company funds and then... let it go

Future of DApps

As Ethereum and other projects have made writing Dapp protocols quicker and more accessible, a number of possibly disruptive Dapps have appeared

KYC-Chain allows users to maintain private “identity wallet” which can be used to authenticate their identification in finance, legal, or commerce settings

WeiFund uses smart contracts to enhance crowdfunding services including GoFundMe and Kickstarter

Storj provides censorship-free, secure, and zero-downtime distributed cloud storage by sharing data and storing it among a decentralized network of computers

4G Capital — provides micro loans for small businesses in Africa by utilizing smart contracts..

Note: A curated collection of various Dapps can be seen at this address: https://www.stateofthedapps.com/

Solidity: Introduction

Solidity: Introduction

> Ethereum would be incomplete without a native programming language – and that language is Solidity.

Solidity

- Solidity is a contract-oriented, high-level language whose syntax is similar to that of JavaScript
- It is designed to target the Ethereum Virtual Machine (EVM)
- Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features
- In contrast to an “object-oriented” language like, Solidity is “contract-oriented”
- It compiles instructions into bytecode so that they can then be read by the network

Layout of a Solidity Source File

Source files can contain an arbitrary number of contract definitions, include directives and pragma directives

Version Pragma

- Source files can (and should) be annotated with a so-called version pragma to reject being compiled with future compiler versions that might introduce incompatible change

The version pragma is used as follows:

```
pragma  solidity  ^0.4.0;
```

Such a source file will not compile with a compiler earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0

Comparison between Memory and Storage in Solidity

They are analogous to memory and hard drive storage in a computer

Memory	Storage
The contract can use any amount of memory while executing its code, but when execution stops, the entire content of the memory is wiped, and the next execution will start fresh	The storage on the other hand is persisted into the blockchain itself, so the next time the contract executes some code, it has access to all the data it previously stored into its storage area

General Value Types

The following types are also called value types because variables of these types will always be passed by value:

Boolean

Bool: The possible values are constants i.e., true and false

Integers

int / uint: Signed and unsigned integers of various sizes

Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256. uint and int are aliases for uint256 and int256, respectively

General Value Types(Contd…)

Address

address: Holds a 20 byte value (size of an Ethereum address). Address types also have members and serve as a base for all contracts

Members of Addresses

Balance and transfer

It is possible to query the balance of an address using the property balance and to send Ether to an address using the transfer function:

```
pragma solidity ^0.4.0;

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

General Value Types(Contd…)

String Literals:

String literals are written with either double or single-quotes “foo” or ‘bar’

They do not imply trailing zeroes as in C; ” foo” represents three bytes not four

As with integer literals their type can vary, but they are implicitly convertible to bytes1, …, bytes32, if they fit, to bytes and to string

String literals support escape characters, such as \n, \xNN and \uNNNN

\xNN takes a hex value and inserts the appropriate byte, while \uNNN takes a Unicode codepoint and inserts an UTF-8 sequence

Arrays

Arrays can have a compile-time fixed size or they can be dynamic

For storage arrays, the element type can be arbitrary

For memory arrays, it cannot be mapping

An array of fixed size k and element type T is written as T[k]

An array of dynamic size as T[]

Variables of type bytes and string are special arrays

Arrays have a length member to hold their number of elements

Dynamic storage arrays and bytes have a member function called push that can be used to append an element at the end of the array

Apportioning Memory Arrays

- Creating arrays with variable length in memory can be done using new keyword
- As opposed to storage arrays, it is not possible to resize memory arrays by assigning to the length member

```
pragma solidity ^0.4.0; contract C {
    function f(uint len) {
        uint[] memory a = new uint[](7); bytes memory b =
        new bytes(len);
        // Here we have a.length == 7 and b.length == len a[6] = 8;
    }
}
```

Allotting Storage Variable to Memory

Arrays

```
// unnamed array in storage, but storage is “


        is no sensible location it could point to.

       delete y;
    g(x); // calls g, handing over a reference to x
    h(x); // calls h and creates an independent, temporary
    memory
  }
  function g(uint[] storage storageArray) internal {}
  function h(uint[] memoryArray) {}
}
```

Apportioning Storage Variable to Memory Array (Contd...)

```
pragma solidity ^0.4.0; contract C {
                    uint[] x; // the data location of x is storage
                       // the data location of memoryArray is memory function f(uint[]
    memoryArray) {
        x = memoryArray; // works, copies the whole array to storage var y = x; // works,
        assigns a pointer, data location of y is
storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
```

Array Literals/ Inline Arrays

- Array literals are arrays that are written as an expression and are not assigned to a variable right away

- The type of an array literal is a memory array of fixed size whose base type is the common type of the given elements

```
pragma solidity ^0.4.0;

contract C {
    function f() { g([uint(1), 2, 3]);
    }
    function g(uint[3] _data) {
                    // ...
    } }
```

Mappings

Mapping types are declared as :

Mapping(_Keytype => _ValueType)

Here, _KeyType can be almost any type except for a dynamically sized array, a contract, an enum and a struct

_ValueType can actually be any type, including 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

The similarity ends here, though: The key data is not actually stored in a mapping, only its hash is used to look up the value

Example of Mappings

```
pragma solidity ^0.4.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) { balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() returns (uint) { MappingExample m = new MappingExample(); m.update(100);
        return m.balances(this);
    }
}
```

Enums

- Enums are one way to create a user-defined type in Solidity

- They are explicitly convertible to and from all integer types but implicit conversion is not allowed
- The explicit conversions check the value ranges at runtime and a failure causes an exception Enums needs at least one member

```
pragma solidity ^0.4.0; contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill } ActionChoices
    choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() {
        choice = ActionChoices.GoStraight;
    }
```

Structs

- Solidity provides a way to define new types in the form of structs
- Structs are custom defined types that can group several variables
- Struct types can be used inside mappings and arrays and they can itself contain mappings and arrays
- It is not possible for a struct to contain a member of its own type, although the struct itself can be the value

```
pragma solidity ^0.4.0;
contract Ballot {
    struct Voter { // Struct
        uint weight1, weight2, weight3; bool voted;
        address delegate1, delegate2, delegate3, delegate4; uint vote1, vote2,
        vote3, vote4, vote5;
        uint height1, height2, height3 //structs can only have 16 members, exceeding which
the following error might occur [stack too deep]
    } }
```

Internationally available Variables & Functions

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain

Block and Transaction Properties

Block.blockhash(uint blockNumber) returns (bytes32): hash of the given block- only works for 256 most recent blocks excluding current

Block.coinbase (address): current block miner's address

Block.difficulty (uint): current block difficulty

Block.gaslimit(uint): current block gaslimit

Block.number (uint): current block number

Block.timestamp (uint): current block timestamp as seconds since unix epoch

Internationally available Variables & Functions

- **msg.data(bytes):** complete calldata
- **msg.sender(address):** sender of the message call (current call)
- **msg.sig(bytes4):** first four bytes of the message (current call)
- **msg.value (uint):** number of wei sent with the message
- **now (uint):** current block timestamp (alias for block.timestamp)
- **tx.gasprice(uint):** gas price of the transaction
- **tx.origin(address):** sender of the transaction (full call chain)

Mathematical Functions

addmod(uint x, uint y, uint k) returns (uint):

Compute (x+y)%k where the addition is performed with arbitrary precision and does not wrap around at

2**256

mulmod(uint x, uint y, uint k) returns (uint):

Compute (x*y)%k where the multiplication is performed with arbitrary precision and does not wrap around at 2**256

Address Related Functions

<address>.balance(uint256):

Balance of the Address in Wei

address>.delegatecall(…)returns(bool):

Issue low-level DELEGATECALL,return false on failure

<address>.send(uint256 amount) returns (bool):

Send given amount of Wei to Address, returns false on failure

<address>.transfer(uint256 amount):

Send given amount of Wei to Address, throw on failure

Error Handling

Assert(bool condition):

Throws if the condition is not met- to be used for internal errors

Require(bool condition):

Throws if the condition is not met - to be used for errors in inputs or external components

Revert():

Abort execution and revert state changes

Operators and Functions in Solidity

Operators and Functions in Solidity

Let’s learn about Operators and functions in Solidity

Operators: Arithmetic, Logical & Bitwise

Arithmetic Operators: operators in Solidity are same as in JavaScript

- Operators such as addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%), have the same effect in Solidity like any other programming language

Incremental Operators:

- Incremental operators in solidity: a++, a--, ++a, --a, a+=1, a=a+1
- Rules applicable to other programming languages are similar in solidity also

Bitwise Operators:

- Following are the operators: (Bitwise OR) '|', (Bitwise XOR), (Bitwise negation) '~' , (Bitwise right shift) '>>', (Bitwise left shift) '<

Logical Operators:

- Logical operators in Solidity: ! (logical negation), && (logical and), || (logical or), ==(equality), != (not equal)

Various Operators in Solidity

```
pragma solidity ^0.4.0; contract operators {
          // Arithmetic Operators
         // +,-,*,/, %, **
       // Incremental Operators
       // a++, a--, a+=1, a=a+1,++a,--a;
        a=10;
        a= a++; //here, output will be 10, returned and then then increment is done
         a=++a;
      //Logical Operators
!, &&, ||, ==, !=
       isOwner = true && false;
        var orValue= 0x02 | 0x01; // output
    //Bitwise Operators~,>>, <<; function Operators() {
                                // Initialize state variables here}}
```

Control Structures

- Most of the control structures from JavaScript are available in Solidity except for switch and goto.
- So there is**: if, else, while, do, for, break, continue, return, ? :**, with the usual semantics known from C or JavaScript
- Parentheses can not be omitted for conditionals, but curly braces can be omitted around single-statement bodies
- There is no type conversion from non-boolean to boolean types as there is in C and JavaScript

Scoping & Declarations

A variable which is declared will have an initial default value whose byte-representation is all zeros

Example:

- Default value for a bool is false
- Default value for the uint or int type is 0
- For statically-sized arrays and bytes1 to bytes32, each individual element will be initialized to the default value corresponding to its type
- For dynamically-sized arrays, bytes and string, the default value is an empty array or string

Declaration:

A variable declared anywhere within a function will be in scope for the entire function, regardless of where it is declared

Input Parameters & Output Parameters

- In solidity, functions may take parameters as input
- It may also return arbitrary number of parameters as output

Input Parameters

The input parameters are declared the same way as variables are

```
pragma solidity ^0.4.0; contract Simple {
        function taker(uint _a, uint _b){
// do something with a and b
} }
```

Input Parameters & Output Parameters(cont'd)

Output Parameters

- The output parameters can be declared with the same syntax after the returns keyword

Example: we wished to return two results: the sum and the product of the two given integers, then we would

write:

```
pragma solidity ^0.4.0; contract

Simple {
function arithmetics(uint _a, uint _b) returns (uint o_sum, uint
0_product){
O_sum = _a + _b; O_product
= _a* _b;
}
}
```

Function Calls & Return types

```
 pragma solidity ^0.4.0;
  contract FunctionCall {
// Constructor calls are also a function calls and are defined like this
function FunctionCall(uint param1) {
// Initialize state variables here
}
// you can create a contract object with a name & then use it inside the
function calls like this
Miner m;
function setMiner(address addr) {
 m = Miner(addr); // type casted the addr to Miner type and stored in m
    }//function setMiner(Miner _m) { m = _m; } is also correct
// Now you can use the Miner's function which is info to sent
// some ether with optionally specifying the gas like this
function callMinerInfo() {
m.info.value(10).gas(800)();
 }
```

Function Calls & Return types(Cont'd)

```
//function can also be called as json object as parameters
// below function can be called by using the json object as  shown in demo
function below
function someFunction(uint key, uint value) {
// Do something }
function demoFunction() {
// named arguments
someFunction({value: 2, key: 3}); }
//also note that variable names are optional in parameters  & in returns
function someFunction2(uint key, uint) returns (uint){
return key; // Do something} }
contract Miner{
 //The modifier payable has to be used for info,
// because otherwise, we would not be able to
//send Ether to it in the call m.info.value(10).gas(800)()
function info() payable returns (uint ret) {
return 42; } }
```

Function Modifiers

Modifiers can be used to easily change the behaviour of functions

For example, they can automatically check a condition prior to executing the function

These conditions can be checked before even making the function calls because they have been declared in the function definitions in the smart contracts

Modifiers are the inheritable properties of the contract and may be overridden by the derived contracts

Function Modifier Example

Example: If you want to call a kill contract function through only the owner or creator of the function

```
pragma solidity ^0.4.0;
contract FunctionModifiers{
     address public creator;
    function FunctionModifiers(){
         creator= msg.sender; }
Modifier onlyCreator(){
if(msg.sender!=creator){
   throw;
}
_; // resumes the function execution wherever the access modifier is used
}
function killContract() onlyCreator{ //this function will not execute if an
exception occurs
      self-destruct(creator); }}
```

Fallback Function

A contract can have exactly one unnamed function

This function cannot have arguments and cannot return anything

It is executed on a call to the contract if none of the other functions match the given function identifier

Furthermore, this function is executed whenever the contract receives plain Ether

Contracts that receive Ether directly (without a function call, i.e. using send or transfer), but do not define a fallback function throw an exception, sending back the ether. So if you want your contract to receive Ether, you have to implement a fallback function

Fallback Function Example

```
pragma solidity ^0.4.0;
contract Test {
    // This function is called for all messages sent to
    // this contract (there is no other function).
    // Sending Ether to this contract will cause an exception,
    // because the fallback function does not have the "payable"
    // modifier.
    function() { x = 1; }
    uint x;
}
// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    function() payable { }
}
```

Fallback Function Example (Cont'd)

```
contract Caller {
    function callTest(Test test) {
        test.call(0xabcdef01); // hash does not exist
        // results in test.x becoming == 1.
        // The following will not compile, but even
        // if someone sends ether to that contract,
        // the transaction will fail and reject the
        // Ether.
        //test.send(2 ether);
  }
}
```

Inheritance

Solidity supports multiple inheritance by copying code including polymorphism

All function calls are virtual, which means that the most derived function is called, except when the contract name is explicitly given

When a contract inherits from multiple contracts, only a single contract is created on the blockchain, and the code from all the base contracts is copied into the created contract

Inheritance Example

```
 pragma solidity ^0.4.0;

  contract owned {
      address owner;
     function owned(){
       owner = msg.sender;
}}
contract mortal is owned{     //'is' keyword is used for Inheritance
       function kill(){
           self-destruct(owner);}}
contract User is owned, mortal //Multiple  Inheritance
{
String public UserName;
function User(String _name){
UserName = _name;
}
}
```

Abstract Contracts

Contract functions can lack an implementation as in the following example:

```
 pragma solidity ^0.4.0;

 contract Feline {
      function utterance() returns (bytes32);
}
```

Such contracts cannot be compiled , but they can be used as base contracts:

```
 pragma solidity ^0.4.0;

  contract Feline {
      function utterance() returns (bytes32);
}
contract Cat is feline{
        function utterance() returns (bytes32) {
return "miaow;
}
```

Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented

There are further restrictions:

- Cannot inherit other contracts or interfaces
- Cannot define constructor
- Cannot define variables
- Cannot define structs
- Cannot define enums

```
 pragma solidity ^0.4.0;

 interface Token {
      function transfer(address recipient, uint amount);
}
```

Events

Events allow the convenient usage of the EVM logging facilities, which in turn can be used to “call” JavaScript callbacks in the user interface of a dapp, which listen for these events

Events are inheritable members of contracts

When they are called, they cause the arguments to be stored in the transaction’s log - a special data structure in the blockchain

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Event

    function bid() payable {
        // ...
        HighestBidIncreased(msg.sender, msg.value); // Triggering event
    } }
```

Events Example

```
pragma solidity ^0.4.0;

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

    function deposit(bytes32  id) payable {
        // Any call to this function (even deeply nested) can
        // be detected from the JavaScript API by filtering
        // for `Deposit` to be called.
        Deposit(msg.sender, _id, msg.value);
    }
}
```

Events Example (Contd…)

```
pragma solidity ^0.4.0;
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at(0x123 /* address */);

var event = clientReceipt.Deposit();
// watch for changes
event.watch(function(error, result){
    // result will contain various information
    // including the argumets given to the Deposit  call.
    if (!error)
        console.log(result);
});
// Or pass a callback to start watching immediately
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result); });
```

Creating Contracts via new

- A contract can create a new contract using the new keyword
- The full code of the contract being created has to be known in advance, so recursive creation- dependencies are not possible
- As seen in the example, it is possible to forward Ether to the creation using the .value()

```
 pragma solidity ^0.4.0;

contract D{
       uint x;
function D(uint a) payable {
          x=a;
} }
contract C {
   D d= new D(4); // will be executed
as part of C's constructor

function created(uint arg) {
D newD =new D(arg);
}
function createAndEndowD(uint arg,
uint amount){
D newD= (new D).value(amount)(arg);}
}
```

Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the DELEGATECALL feature of the EVM

This means that if library functions are called, their code is executed in the context of the calling contract

As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied

Thank You

Email us – support@intellipaat.com

Visit us - https://intellipaat.com