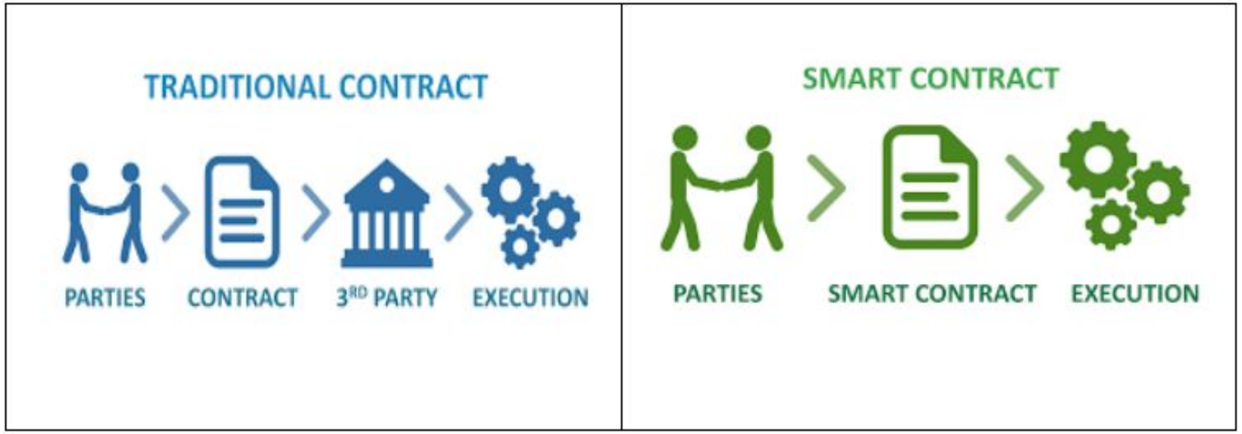
Ethereum Smart Contract

Outline

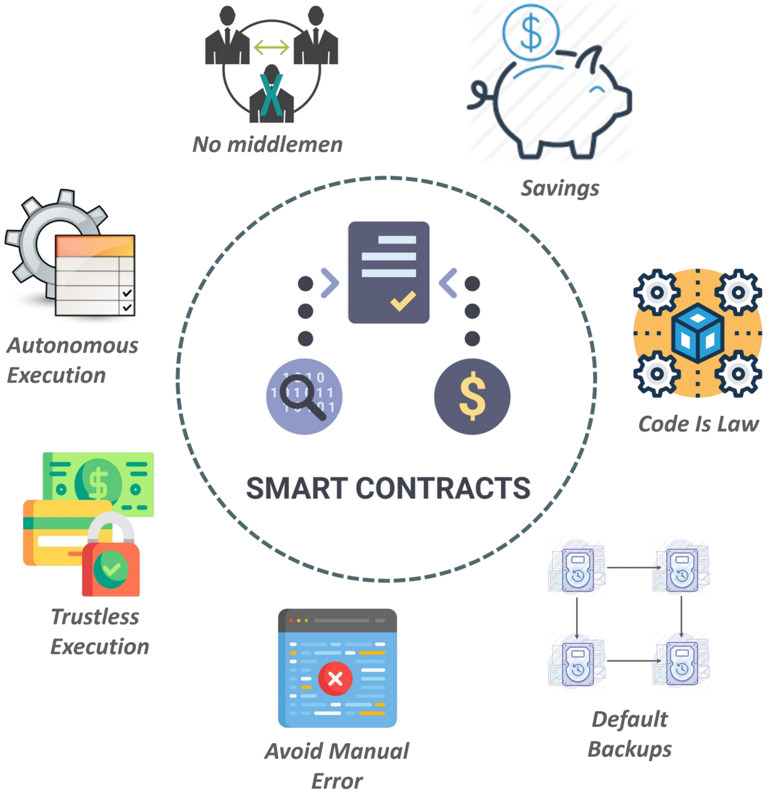
Smart Contract



Smart Contract

- Smart contracts are lines of code that are stored on a blockchain and automatically execute when predetermined terms and conditions are met.
- The code and conditions in the contract are publically available on the ledger.
- When event triggered then the code executes.

Smart Contract



source: https://www.edureka.co/blog/smart-contracts/

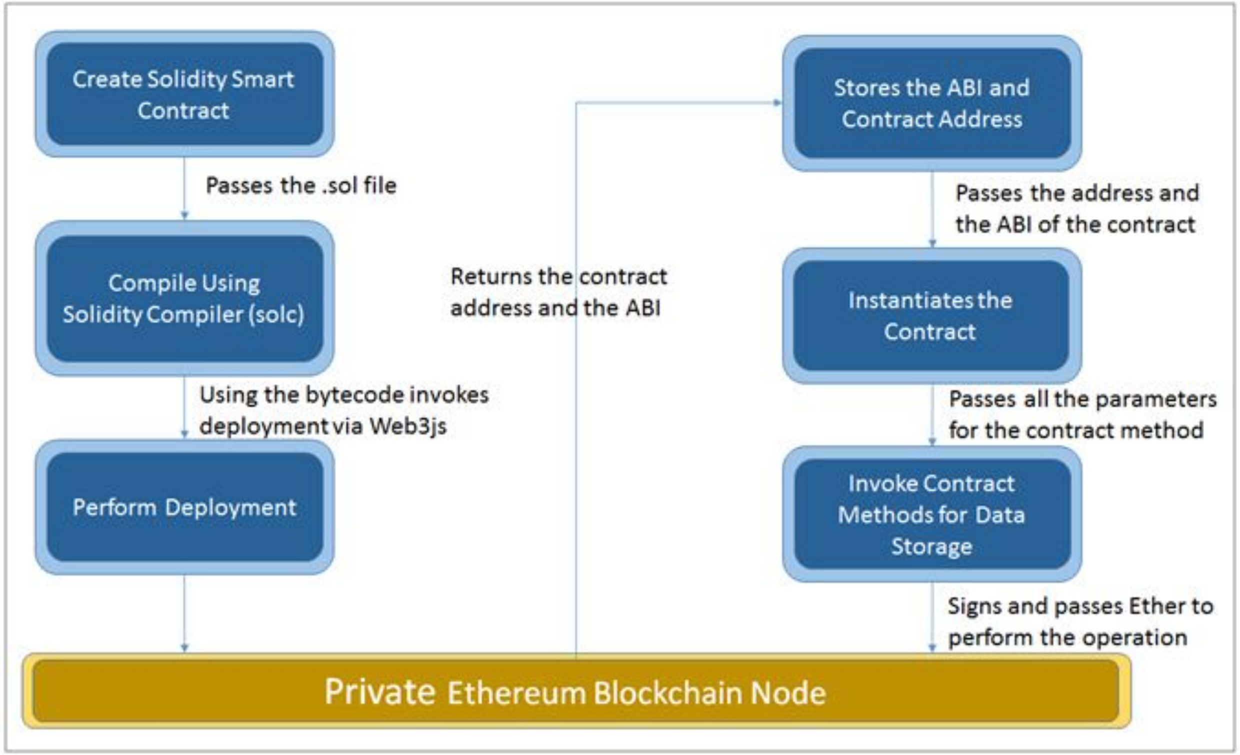
Solidity

- Solidity is a contract-oriented, high-level programming language for implementing smart contracts.
- Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the Ethereum Virtual Machine (EVM).
- You can use Solidity to create contracts for uses such as voting, crowdfunding and multi-signature wallets.

Layout of a Solidity Source File

- Pragmas (Solidity version)
- Importing Source Files
- Contract

Demo Flow



Create Solidity Smart Contract
Passes the .sol file
Compile Using Solidity Compiler (solc)
Using the bytecode invokes deployment via Web3js
Perform Deployment
Returns the contract address and the ABI
Stores the ABI and Contract Address
Passes the address and the ABI of the contract
Instantiates the Contract
Passes all the parameters for the contract method
Invoke Contract Methods for Data Storage
Signs and passes Ether to perform the operation
Private Ethereum Blockchain Node

Value Types

- Booleans
  - **bool**: The possible values are constants true and false.
- Integers
  - signed: int8 to int256
  - unsigned: uint8 to uint256

Address

- Address holds the 20 byte value (size of an Ethereum address).
- An address can be used to get the balance using .balance method and can be used to transfer balance to another address using .transfer method.
- address payable: Same as address, but with the additional members transfer and send.

String

- Solidity supports String literal using both double quote ("cdac") and single quote ('cdac').
- It provides string as a data type to declare a variable of type String.
- Solidity provides inbuilt conversion between bytes to string and vice versa.

Variables

- **State Variables**: Permanently stored in a contract.
- **Local Variables**: Available till function is executing.
- **Global Variables**: Special variables exists in the global namespace used to get information about the blockchain.

block.number (uint) - Current block number

block.timestamp (uint) - Current block timestamp as seconds since unix epoch

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

Variables Scope

State variables can have three types of scopes.

- **Public state variables** can be accessed internally. For a public state variable, an automatic getter function is generated.
- **Internal state variables** can be accessed only internally from the current contract or contract deriving from it without using this.
- **Private state variables** can be accessed only internally from the current contract.

Operators

- Arithmetic Operators (+, -, *, /, %, ++, --)
- Comparison Operators (==, !=, >, <, >=, <=)
- Logical (or Relational) Operators (&&, ||, !)
- Assignment Operators (=, +=, -=, *=, /=, %=)
- Conditional (or ternary) Operators (? :)

Loops

When you need to perform an action over and over again, In such situations, you would need to write loop statements to reduce the number of lines.

- While loop
- Do...while loop
- For loop

Decision making

Solidity supports conditional statements which are used to perform different actions based on different conditions.

- if statement
- if...else statement
- if...else...if statement

Array

An array is used to store a collection of data of same type.

- Fixed Size Array
  - `uint[5] public fixedArray;`
- Dynamic Array
  - `uint[] public dynamicArray;`

Members of an Array

- **length** – length returns the size of the array.
- **push** – push allows to append an element to a dynamic storage array at the end. It returns the new length of the array.
- **pop** - pop removes the last element of an array, and returns that element.

Structs

- Solidity provides a way to define new types in the form of structs.
- To define a Struct, you must use the struct keyword.

```
struct myStruct {
   type1 var1;
   type2 var2;
}
```

Structs

- To access any member of a structure, we use the member access operator (.).
- The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.

```
structVar.structMember
```

Mapping

- Mapping is a reference type as arrays and structs.
- Mappings can only have a data location of storage and thus are allowed for state variables

  mapping(_KeyType => _ValueType) _VariableName.

Error Handling

Solidity provides various functions for error handling.

- assert – 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.

```
assert(bool condition)
```

Error Handling

- require – 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.

```
require(bool condition, string memory message)
```

- 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.

```
revert(string memory reason)
```

Developing and Deploying the Sample Application

CDAC Contract

```
pragma solidity ^0.6.0;
contract CDACContract {

   uint myVar;

   function setVariable(uint newVar) public{
        myVar = newVar;
   }

   function getVariable() public view returns(uint){
        return myVar;
    }
}
```

Sample Application

```
pragma solidity ^0.6.0;
contract balanceTransfer{
   uint public totalRecievedMoney;
   uint public totalTransferredMoney;
   address owner;
   constructor()public{
       owner = msg.sender;
   }
   function receiveMoney() public payable{
       totalRecievedMoney+=msg.value;
   }
   function getBalance()public view returns(uint){
       return address(this).balance;
   }
   function transferMoneyFromContract(address payable toAccount, uint amount)public{
       require(msg.sender==owner, "you are not owner");
       toAccount.transfer(amount);
   }
}
```

Thank You