Creating Smart Contracts with Solidity

Introduction to Solidity

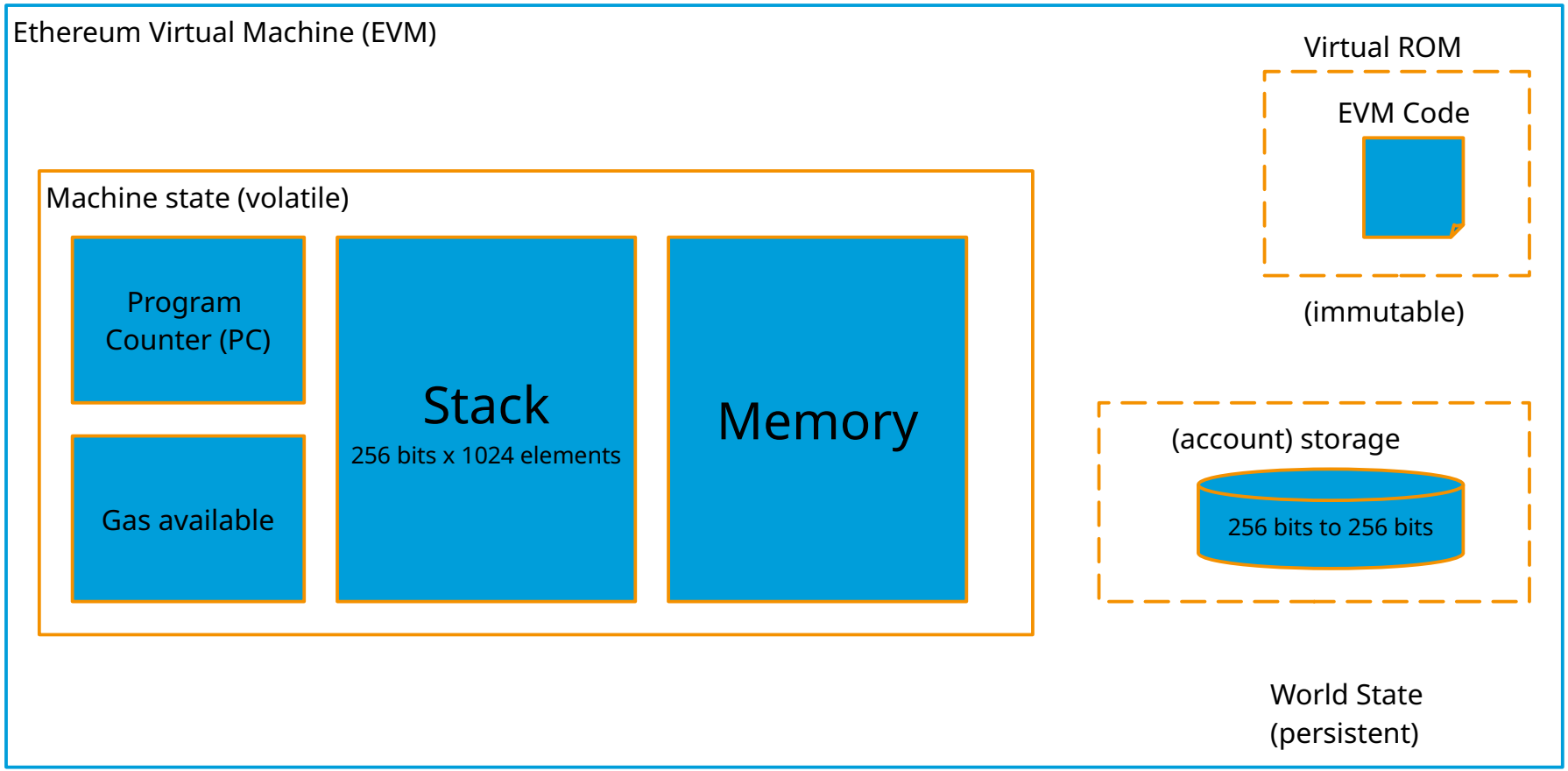
Ethereum Virtual Machine

Ethereum and EVM

- Ethereum is a blockchain with a computer embedded in it
- The computers is specified as Ethereum Virtual machine – EVM
- Each node stores the state of this computer
- Each network participant can send computation requests
- After finishing computation, the state is updated and propagated to all participants in the network

Ethereum Virtual Machine

- A really small (and slow) virtual computer
- Has the bare minimum to do computation
  - Memory
  - Stack
  - Program Counter
  - EVM Microcode (ROM)


Ethereum Virtual Machine (EVM)
Virtual ROM
EVM Code
(immutable)
Machine state (volatile)
Program Counter (PC)
Gas available
Stack
256 bits x 1024 elements
Memory
(account) storage
256 bits to 256 bits
World State
(persistent)

EVM and transactions

- Two types of EVM transactions
  - Smart Contract creation transactions
  - Message calls transactions - execution of smart contract methods

EVM opcodes

- Like any processor it has a list of opcodes (operation code) which allow for computation
  - MUL
  - SUB
  - LT
  - Full list at https://www.evm.codes/?fork=shanghai

Solidity

What is Solidity

- Object-oriented, high-level language for implementing smart contracts
- Targets EVM
- Influenced by C++, Python, and JavaScript
- Statically typed
- Supports inheritance
- Supports libraries and complex user-defined types

Smart Contract Example

```
// Declare contract name and body
contract  Example {
    // Declare an event
    event StateChanged(uint _old, uint _new);
    // Declare state variable
    uint someState;
    // The contract constructor
    constructor() {
    }
    // Declare contract method
    function changeState(uint newState) public {
        emit StateChanged(someState, newState);
        someState = newState;
    }
}
```

Syntax: Literals

- address literals
- rational and integer numbers
- string literals
- unicode
- hexadecimal – represent binary data in hexadecimal

Literal Values Examples

```
// Address , exactly 40 digits hexadecimal
0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF
// Integer and rational
5 , 1.3 , 3e10 , 1_000_000 , 0xBEEF
// String
“sol”
// Unicode string literal
unicode"Hello 😀"
// Hexadecimal
hex"CAFECAFECAFE"
```

Syntax: Operators

- Arithmetic operators , equality and assignment
  - + , - , * , / , % , ++ , -- , ** , == , != , > , < , >= , <= , = , += , -= , *= , /= , %=
- Logical
  - || && !
- Bitwise
  - & | ^ ~ << >>
- Conditional
  - ? :

Syntax: Expressions & Control Sructures

- Control structures (the same as in C / Javascript)
  - if, else, while, do, for, break, continue, return
- Limited support for **try/catch**, error handling is done mostly with **revert/require** and **assert**
- Function calls
  - Internal - doSomething()
  - External – this.doSomething() or contract.doSomething() , can’t be used in the constructor as contract is not created yet. Options for gas and value can be sent additionally

Control structure Examples

```
contract Question {
    function answer() public payable returns (uint ret) { return 42; }
}
contract Person {
    Question q;
    function checkAnswer() public {
        // External call specifying bot gas & value
        uint answer = q.answer{value: 10, gas: 800}();
        // 3 different way to ensure correctness
        assert( answer == 42 ); // panic if not executed
        if( answer != 42) revert("Answer not correct")
        require(answer == 42 , "Answer not correct");
     }
}
```

Smart Contract building blocks

- State Variables
- Struct Types
- Functions
- Events
- Pragma directives – mostly to control compilation

State Variables

- Variables declared in the body of the smart contract are state variables because their state is automatically persisted
- Visibility of state variables can be
  - **public** - state is modifiable externally, the compiler automatically generates a getter for the variable
  - **internal** – can be accessed only from the contract and derived contracts (default)
  - **private** – can be accesses only from the current contract
- Note that the visibility doesn't prevent reading the state from the blockhain, it only regulates how other smart contracts interact with the state variables

Solidity Types

- Solidity is a strictly typed language and each variable must have explicitly defined type
- There are mainly 3 categories of types
  - Value Types
  - Reference Types
  - Mapping Types

Value Types

- bool
- int/uint - uint8, uint16, uint24 … uint256 , uint is alias for uint256
- fixed/ufixed – ufixedMxN M[8,56] - integer, N[0,80] – fractional
- address – representing Ethereum address (20 byte)
- bytes – bytes1, bytes2, …., bytes32 – sequence of bytes
- contract – each contract defines it’s own type
- enum – named integer values

Value Types Examples

```
contract Example {

    bool private flag; // Boolean flag
    uint public largeInt; // 256bit unsigned integer
    int8 smallInt; // 8bit signed integer
    ufixed128x18 fractional; // 128bits integer with 18 bits
fractional
    address owner; // Ethereum address
    bytes16 internal binaryData; // 16 bytes of binary data

    enum Colors { Blue, Red, Green }
    Colors constant fgColor = Colors.Red;

//...
```

Reference Types

- array
  - value type array - e.g. uint[]
  - bytes – like bytes1[], but with more efficient memory representation
  - string – like bytes but has no index access and no legth
- struct
- When using a reference type a data location MUST always be specified. Data locations are:
  - memory - limited to the lifetime of a function call
  - storage – saved in state, limited to the lifetime of the contract
  - calldata – special location with function arguments

Refence Types Examples

```
contract Example {
    // When declaring array as state , data location can be
omitted because it MUST be storage
    uint[] someData;
    function doSomething(uint[] memory a) public {
        someData = a; // will copy the array to storage
        uint[] storage refToSomeData = someData; // only
points to someData
        delete someData; // clears someData and refToSomeData

    }
//...
```

Mapping Types

- mapping(KeyType => ValueType)
- Like {} in Javascript or dict() in Python. It basically is like a hash table mapping a key to a certain value.
- They are also a Reference type but can only be in storage
- KeyType can be any value type, bytes, string, contract or address
- ValueType can be any type

Mapping Types Examples

```
contract Example {
    // Map address to balance
    // The public modifier will automatically create a public
get method
     mapping(address => uint) public balances;
    // Declare nested mapping looking like the following JSON
    // {
    //   address1 : {
    //            “str1” : 5,
    //            “str2” : 8,
    //   },
    //   ….
    mapping(address => mapping(string => uint)) selection;
//...
```

Structs

- Provide a way to create new types with structured data (composition)
- They are reference types, so do not copy by default

Struct Examples

```
contract Example {
    enum Colors { Blue, Red, Green }

    struct Model {
        string manufacturer;
        string model;
        uint16 year;
    }

    struct Car {
            address owner;
            Colors color;
            Model model;
    }

    mapping(string => Car) registry;
//...
```

Functions

- Can take parameters
  - **function sum(uint a, uint b) returns(uint sum) {}**
- Can return Multiple values
  - **function intdiv(uint a, uint b) returns(uint div, uint reminder) {}**
- Can mutate state
  - **No declaration** – function can mutate the state
  - **view** – declares that it will only read the state, no modification
  - **pure** – declares that it will neither write or read the state
- Can be defined inside or outside of the contract, outside implies

Functions Visibility

- **external** - part of the contract interface, can be called only outside of the contract
- **public** – part of the contract interface, can be called both externally and internally
- **internal** – can be called only inside the contract or from derived contract. Functions defined outside of the contract are implicitly internal
- **private** – can be called only from the current contract

Special Functions

- constructor() - the smart contract constructor, gets called automatically on smart contract creation
- receive - one function in the contract can be marked as receive, and it gets called when somebody sends currency to the smart contract
- fallback – one function in the contract can be marked as fallback and it gets called when there is a call to the contract and none of the functions matches the call signature

Functions Example

```
// Defined outside the contract so it's internally visible
only
function sum(uint a, uint b) pure returns (uint sum) {
    sum = a + b;
}

contract Example {
    mapping(address => uint) allBalances;

    // Only reads from the state so we declare it's a view
    function getBalance(address owner)
        public view returns (uint balance, uint interest) {
        balance = allBalances[owner];
        interest = balance * 0.1;
    }
}
```

Events

- Events are structured log messages, that are publicly available for reading
- They have type, and arguments
- Provide way for outside world to listen for events that happen on the Ethereum blockchain

Events Example

```
contract  EventLogger {
    // Declare an event
    event MethodCalled(address caller);

    function doSomething() public {
        // emit the event
        emit MethodCalled(msg.sender);
        // …
    }
}
```

Pragma Directives

- Directive that is put at the beginning of the file controlling compiler options
- Most important one is controlling the version of solidity required
  - pragma solidity ^0.8.0; // require certain version(semver style) of Solidity

Let’s write some code