Essence of Solidity in Depth

Smart Contracts #2

목차

목차

Things You Can Declare within a Contract

- State Variables
- Functions
- Function Modifiers
- Events
- Struct Types
- Enum Types

실습#1에서 사용한 Count 스마트 컨트랙트

```
pragma solidity ^0.5.6;
contract Count {
  uint public count = 0;
  address public lastParticipant;

  function plus() public {
    count++;
    lastParticipant = msg.sender;
  }

  function minus() public {
    count--;
    lastParticipant = msg.sender;
  }
}
```

State Variables

- 블록체인에 영구히 저장할 값들은 상태변수(State Variable)로 선언
  - 어떤 값들은 반드시 State Variable로 선언되어야 함 (e.g., mapping)
- `public` 키워드를 사용하여 변수를 외부에 노출가능
  - 이 경우 자동으로 해당 변수 값을 돌려주는 Getter 함수가 생성됨

```
contract Count {
  uint public count = 0;
  address public lastParticipant;
}
```

Functions

- 함수(Function)은 실행 가능한 코드를 정의한 것
  - external, public, internal, private 중 하나로 visibility를 설정 가능
  - payable, view, pure등 함수의 유형을 정의 가능

```
contract Count {
  function plus() public {
    count++;
    lastParticipant = msg.sender;
  }
}
```

Function Modifiers

- 함수의 실행 전, 후의 성격을 정의
    - 대부분의 경우 함수의 실행조건을 정의하는데 사용됨

```
contract Ballot {
    constructor() public { chairperson = msg.sender; }
    address chairperson;
    modifier onlyChair {
        require(msg.sender == chairperson, "Only the chairperson can call this function.");
        _;
    }
    function giveRightToVote(address to) public onlyChair {
        // `onlyChair` modifier ensures that this function is called by the chairperson
    }
}
```

Events

- 이벤트는 EVM 로깅을 활용한 시스템
  - 이벤트가 실행될 때마다 트랜잭션 로그에 저장
  - 저장된 로그는 컨트랙트 주소와 연동되어 클라이언트가 RPC로 조회 가능

Contract

```
contract Ballot {
    event Voted (address voter, uint proposal);
    function vote(uint proposal) public {
        …
        emit Voted(msg.sender, proposal);
    }
}
```

Client using caver-js

```
const BallotContract = new caver.klay.Contract(abi, address);
BallotContract.events.Voted(
    { fromBlock: 0 },
    function(error, event) {
        console.log(event);
    }
).on('error', console.error);
```

Struct Types

- Solidity에서 제공하지 않는 새로운 형식의 자료를 만들 때 사용
  - 여러 자료를 묶어 복잡한 자료형(complex type)을 만들 때 유용

```
contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

```
contract SocialMedia {
    struct Friend {
        uint id;
        mapping (uint => address) friends;
    }
}
```

Enum Types

- Enum은 임의의 상수를 정의하기에 적합
  - e.g., 상태 ⇒ Active, Inactive
  - e.g., 요일 ⇒ Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

```
contract Ballot {
    enum Status {
        Open,
        Closed
    }
}
```

목차

Data Types

- Solidity는 다음과 같은 자료형(Data Types)을 지원
  - Booleans (bool)
  - Integers (int / int8 / int16 / … / uint256 / uint / uint8 / uint16 / … / uint256)
  - **Address**
  - **Fixed-size byte arrays (bytes1, bytes2, …, bytes32)**
  - **Reference Types**
  - **Arrays**
  - **Mapping Types**
  - Contract Types

Address

- 어카운트 주소를 표현
  - Klaytn 주소의 길이는 20바이트; address ⇒ bytes20 형변환 가능
  - address vs. address payable
    - address payable ⇒ address (O)
    - address ⇒ address payable (X); uint160을 거쳐서만 가능

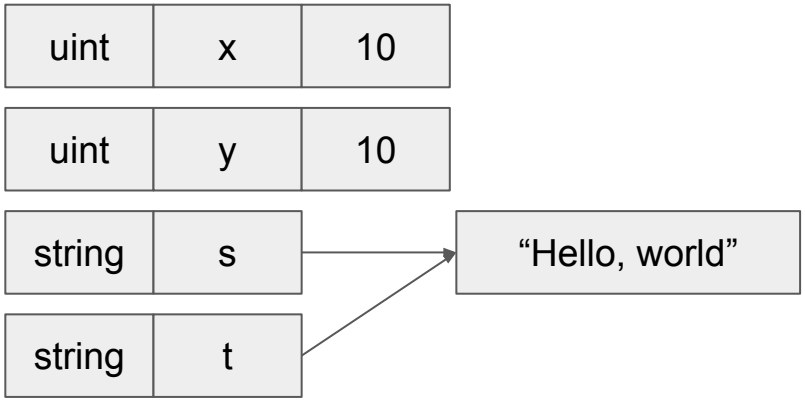
Reference Types

```
uint x = 10;

uint y = x;

string memory s = "Hello, world";

string memory t = s;
```



- Reference Type은 structs, arrays, mapping과 같이 크기가 정해지지 않은 데이터를 위해 사용
  - Value Type들은 변수끼리 대입할 경우 값을 복사
  - Reference Type들은 (같은 영역을 사용하는) 변수끼리 대입할 경우 같은 값을 참조

Reference Types

- Reference Type 데이터는 저장되는 위치를 반드시 명시
  - memory (함수 내에서 유효한 영역에 저장)
  - storage (state variables와 같이 영속적으로 저장되는 영역에 저장)
  - calldata (external 함수 인자에 사용되는 공간)
- 서로 다른 영역을 참조하는 변수 간 대입이 발생 시 데이터 복사
  - storage ⇒ memory / calldata
  - anything ⇒ storage

Arrays

- Javascript에서 배운 배열과 개념은 같으나 사용법이 상이
  - State Variable로 사용할 때 (i.e., 저장공간 = `storage`)
    - `T[k] x;` ⇒ k개의 T를 가진 배열 x를 선언
    - e.g., `uint[5] arr;` ⇒ arr은 5개의 uint를 가진 배열; arr[0] ⇒ 첫번째 uint
    - `T[] x;` ⇒ x는 T를 담을 수 있는 배열; x의 크기는 변할 수 있음 (dynamic size)
    - `T[][k] x;` ⇒ k개의 T를 담을 수 있는 dynamic size 배열 x를 선언
    - `T[][k] x` 가 주어질 때 `x[i][j]`는 `(i-1)`번째 배열의 `(j-1)`번째 T를 불러옴
- 모든 유형의 데이터를 배열에 담을 수 있음
  - mapping, struct 포함

More on Arrays

- .push(T item) and .length
  - .push(T item)은 배열에 데이터를 추가
  - .length는 배열크기를 반환
- 런타임에 생성되는 (i.e., 함수 내에서) memory 배열은 new 키워드를 써서 선언
  - storage 배열과는 달리 memory 배열은 dynamic array 사용이 불가.

```
contract C {
    function f(uint len) public pure {
        bytes memory b = new bytes(len);
        assert(b.length == len);
    }
}
```

bytesN vs. bytes/string vs. byte[]

- (가능하면) 언제나 bytes를 사용
  - byte[]는 배열 아이템 간 31바이트 패딩이 추가됨
- 기본 룰
  - 임의의 길이의 바이트 데이터를 담을 때는 bytes
  - 임의의 길이의 데이터가 UTF-8과 같이 문자로 인코딩 될 수 있을 때는 string
  - 바이트 데이터의 길이가 정해져있을 때는 value type의 bytes1, …, bytes32를 사용
  - byte[]는 지양

Mapping Types

```
mapping (K => V) table;
```

- 해시테이블과 유사, 배열처럼 사용
  - storage 영역에만 저장 가능 (i.e., state variable로만 선언 가능)
  - 함수 인자, 또는 public 리턴값으로 사용 할 수 없음

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

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

목차

Special Variables and Functions

- Block and Transaction Properties
- Error Handling
  - require(bool cond)
  - require(bool cond, string memory message)
- Cryptographic Functions
  - keccak256(bytes memory input) returns (bytes32)
  - sha256(bytes memory input) returns (bytes32)
  - ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)

Blocks and Transaction Properties

`blockhash(uint blockNumber) returns (bytes32)`: 블록 해시 (최근 256 블록까지만 조회가능)

`block.number (uint)`: 현재 블록 번호

`block.timestamp (uint)`: 현재 블록 타임스탬프

`gasleft() returns (uint256)`: 남은 가스량

`msg.data (bytes calldata)`: 메세지(현재 트랜잭션)에 포함된 실행 데이터 (input)

`msg.sender (address payable)`: 현재 함수 실행 주체의 주소

`msg.sig (bytes4)`: calldata의 첫 4 바이트 (함수 해시)

`msg.value (uint)`: 메세지와 전달된 KLAY (peb 단위) 양

`now (uint)`: `block.timestamp`와 동일

`tx.gasprice (uint)`: 트랜잭션 gas price (25 ston으로 항상 동일)

`tx.origin (address payable)`: 트랜잭션 주체 (sender)

Error Handling

`assert(bool condition)`:

condition이 false일 경우 실행 중인 함수가 변경한 내역을 모두 이전 상태로 되돌림 (로직 체크에 사용)

`require(bool condition)`:

condition이 false일 경우 실행 중인 함수가 변경한 내역을 모두 이전 상태로 되돌림 (외부 변수 검증에 사용)

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

require(bool)과 동일. 추가로 메세지를 전달.

Cryptographic Functions

`keccak256(bytes memory) returns (bytes32)`:

주어진 값으로 Keccak-256 해시를 생성

`sha256(bytes memory) returns (bytes32)`:

주어진 값으로 SHA-256 해시를 생성

`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`:

서명(v, r, s)으로부터 어카운트 주소를 도출 (서명 ⇒ 공개키 ⇒ 주소).

목차

Expressions and Control Structures

- Solidity 제어 구문
  - 대부분의 프로그래밍 언어가 지원하는 제어 구문을 지원
  - if, else, while, do, for, break, continue, return
- 예외처리 기능이 없음
  - i.e., try-catch 없음

Loop example: for

```
function loop(uint repeat) public pure returns (uint) {
    uint sum = 0;
    for (uint i = 0; i < repeat; i++) {
        sum += i;
    }
    return sum;
}
```

```
function fib(uint n) public pure
 returns (uint) {
    uint x = 0;
    uint y = 1;
    uint ret = 0;
    for (uint i = 0; i < n; i++) {
        ret = x + y;
        x = y;
        y = ret;
    }
    return ret;
}
```

Loop example: while

```
function whileloop(uint repeat)
    public pure
    returns (uint)
{
    uint sum = 0; uint i = 0;
    while (i < repeat) {
        sum += i;
        i++;
    }
    return sum;
}
```

목차

Contracts

- Creating Contracts
- Visibility and Getters
- Function Modifiers
- Constant State Variables
- Functions: view, pure, fallback

Creating Contracts

- 일반적인 컨트랙트 생성 ⇒ 배포
- 컨트랙트를 클래스처럼 사용
  - 컨트랙트를 객체지향 프로그래밍에서 사용하는 클래스로 취급할 수 있음
  - new 키워드를 사용하여 컨트랙트를 생성하여 변수에 대입

```
contract A {
    B b;
    constructor() public { b = new B(10); }
    function bar(uint x) public view
        returns (uint)
    {
        return b.foo(x);
    }
}
```

```
contract B {
    uint base;
    constructor(uint _base) public { base = _base; }
    function foo(uint y) public view
        returns (uint)
    {
        return y * base;
    }
}
```

Visibility and Getters

- 함수의 visibility(공개정도)를 목적에 맞게 설정
  - `external`
    - 다른 컨트랙트에서 & 트랜잭션을 통해 호출 가능
    - internal 호출 불가능 (i.e., f()는 안되지만 this.f()는 허용됨)
  - `public`
    - 트랜잭션을 통해 호출 가능, internal 호출 가능
  - `internal`
    - 외부에서 호출 불가능, internal 호출 가능, 상속받은 컨트랙트에서 호출 가능
  - `private`
    - internal 호출 가능

Function Declarations: pure vs. view vs. (none)

- 함수 제약을 설정하여 정해진 scope에서 동작할 수 있도록 설정
  - `pure`
    - State Variable 접근 불가 i.e., READ (x), WRITE (X)
  - `view`
    - State Variable 변경 불가 i.e., READ (O), WRITE (X)
  - (none)
    - 제약 없음 i.e., READ (O), WRITE (O)

Fallback function

- 컨트랙트에 일치하는 함수가 없을 경우 실행 (i.e., no input/calldata)
  - 단 하나만 정의 가능 & 함수명, 함수인자, 반환값 없음
  - 반드시 `external`로 선언
- 컨트랙트가 KLAY를 받으려면 `payable` fallback function이 필요
  - `payable fallback`이 없는 컨트랙트가 KLAY를 전송받으면 오류 발생

```
contract Escrow {
    event Deposited(address sender, uint amount);
    function() external payable {
        emit Deposited(msg.sender, msg.value);
    }
}
```

End of Document