1 Contract

- 스마트 컨트랙트란?
  - `당사자들이 다른 약속에 따라 수행하는 프로토콜을 포함하여 디지털 형식으로 지정된 일련의 약속` - 닉 사보
- 이더리움에서 **컨트랙트(스마트 컨트랙트)란 불변적인 컴퓨터 프로그램**을 말한다.
  - 이 프로그램은 이더리움 네트워크 프로토콜의 일부인 **이더리움 가상 머신의 컨텍스트상에서 결정론적으로 작동**한다.
- 컨트랙트는, 수행되거나 컴파일 되어야 할 어떤 것이라기 보다는이더리움의 실행 환경안에 살아있는 일종의 자율 에이전트이다
- 컨트랙트는 자신이 소유한 ether balance와 key/valuce store 대한 직접적인 통제권을 가지고 있다.

1.1 Contract의 정의

- **컴퓨터 프로그램**
  - 스마트 컨트랙트는 단순히 컴퓨터 프로그램이다
  - `컨트랙트` 라는 단어에 법적인 의미는 없다
- **immutable**
  - 스마트 컨트랙트 코드는 일단 배포되면 변경할 수 없다
  - 스마트 컨트랙트를 수정하는 유일한 방법은 새로운 인스턴스를 배포하는 것이다
- **결정론적(deterministic)**
  - 스마트 컨트랙트를 실행한 결과는 실행한 모든 이에게 동일하다
  - 어떤 노드에서 실행하더라도 스마트 컨트랙트를 실행한 트랜잭션 컨텍스트와 실행 시점에서의 이더리움 블록체인 상태가 같으면 실행 결과는 같습니다.
- **EVM 컨텍스트**
  - 스마트 컨트랙트는 매우 제한적인 실행 컨텍스트에서 작동된다
  - 자신의 상태 호출한 트랜잭션의 컨테스트 및 가장 최근 블록의 일부 정보에만 접근할 수 있다

- **탈중앙화된 월드 컴퓨터(decentralized world computer)**
  - EVM은 모든 이더리움 노드에서 로컬 인스턴스로 실행된다.
  - EVM의 모든 인스턴스 동일한 초기 상태에서 동작하고 동일한 최종 상태를 생성하기 때문에 시스템 전체가 단일 월드 컴퓨터로 작동한다.
    - 결정론적으로 스마트 컨트랙트를 실행하기 때문에 전체가 단일 월드 컴퓨터 처럼 작동합니다.

2 컨트랙트의 생명주기

1. 코드 작성
2. 컴파일
3. 컨트랙트 생성 트랜잭션
4. 컨트랙트 실행
5. 컨트랙트 삭제

2.1 컨트랙트 코드 작성

```
contract Faucet {

    function withdraw(uint withdraw_amount) public {
        require(withdraw_amount <= 100000000000000000);

        msg.sender.transfer(withdraw_amount);
    }

    function () public payable {}
}
```

2.2 컴파일

- 고급 언어(솔리디티)로 작성한 스마트 컨트랙트는 EVM에서 사용되는 바이트 코드로 컴파일되어야 한다.
- solidity로 작성한 컨트랙트는 solc라는 컴파일러로 EVM 바이트 코드로 변환한다.

컴파일된 바이트코드

```
solc --bin Faucet.sol
```

```
60606040523415610000f57600080fd5b60e58061001d6000396000f300606060405260043610603f576000357c0100000000000000000000000000000000000000000000000000000090046300000000ffffffff1680632e1a7d4d146041575b005b3415604b57600080fd5b605f6004808035906020019091905050506061565b005b67016345785d8a0000811115151560775760600080fd5b3373ffffffffffffffffffffffffffffffffffffffff166108fc8290811502906040516000604051808303818588888f193505050501515560b657600080fd5b505600a165627a7a7230582071ed64dc2b534bdf3c7c54a9f8e0e4a29ef9adefe21681a724303852faf5ddb40029
```

```
{
  "linkReferences": {},
  "object":
"6060604052341561000f57600080fd5b60e58061001d6000396000f3006060604052600
43610603f576000357c0100000000000000000000000000000000000000000000000000
00000900463ffffffff1680632e1a7d4d146041575b005b3415604b57600080fd5b605f6
004808035906020019091905050606156 5b005b67016345785d8a0000811115151560775
7600080fd5b3373ffffffffffffffffffffffffffffffffffffffff166108fc829081150
290604051600060405180830381858888f193505050501515 60b657600080fd5b505600a
165627a7a7230582071ed64dc2b534bdf3c7c54a9f8e0e4a29ef9adefe21681a72430385
2faf5ddb40029",
  "opcodes": "PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF
JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1
0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1
0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI PUSH1 0x0 CALLDATALOAD PUSH29
0x100000000000000000000000000000000000000000000000000000 SWAP1 DIV
PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI JUMPDEST
STOP JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20
ADD SWAP1 SWAP2 SWAP1 POP POP PUSH1 0x61 JUMP JUMPDEST STOP JUMPDEST
PUSH8 0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI
PUSH1 0x0 DUP1 REVERT JUMPDEST CALLER PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH2 0x8FC DUP3 SWAP1
DUP2 ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1
DUP4 SUB DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP POP ISZERO ISZERO
PUSH1 0xB6 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 PUSH18 0xED64DC2B534BDF3C7C54A9F8E0E4A29EF9AD
0xef 0xe2 AND DUP2 0xa7 0x24 ADDRESS CODESIZE MSTORE STATICCALL 0xf5
0xdd 0xb4 STOP 0x29 ",
  "sourceMap": "25:221:0:-;;;;;;;;;;;;;;;;;"
}
```

2.3 Contract의 생성

- 컴파일된 컨트랙트 코드는 특수한 형태의 트랜잭션을 통해 이더리움 블록체인에 배포가 되고 이때부터 공개된다.
  - 이 특수한 형태의 트랜잭션을 `contract creation trasaction`이라 한다.
  - `contract creation trasaction`은 to 필드로 고유한 컨트랙트 생성 주소(0x0)를 가지고 있다.
- 트랙트가 생성되면 지갑과 마찬가지로 주소를 가지게된다.
  - 이 주소는 원래 계정 및 논스의 함수로 컨트랙트 생성 트랜잭션에서 파생된다.

```
src = web3.eth.accounts[0]
faucet_code =
"606060405234156100f57600080fd5b60e58061001d6000396000f3006060604052600
43610603f576000357c0100000000000000000000000000000000000000000000000000
00000900463ffffffff1680632e1a7d4d146041575b005b3415604b57600080fd5b605f6
004808035906020019091905050606156 5b005b67016345785d8a0000811115151560775
7600080fd5b3373ffffffffffffffffffffffffffffffffffffffff166108fc829081150
290604051600060405180830381858888f193505050501515 60b657600080fd5b505600a
165627a7a7230582071ed64dc2b534bdf3c7c54a9f8e0e4a29ef9adefe21681a72430385
2faf5ddb40029"

web3.eth.sendTransaction({from: src, to: 0, data: faucet_code,
gas:113558, gasPrice:200000000000})

"0x7babc2939834dfae2966710ab097346861aeb3112c2"
```

2.4 Contract의 실행

- 컨트랙트는 트랜잭션에 의해 호출된 경우에만 실행된다.
  - 모든 컨트랙트는 EOA에서 시작된 트랜잭션으로 시작된다.
  - 컨트랙트가 다른 컨트랙트를 호출하여 체인을 구성하지만 첫 번째 컨트랙트는 항상 EOA의 트랜잭션으로 호출된다.
  - 컨트랙트가 다른 컨트랙트를 호출: 메시지, 인터널 트랜잭션

- 스마트 컨트랙트는 병렬적으로 실행되지 않는다
- 트랜잭션 원자성의 특징을 지닌다.
    - 모든 실행이 성공적으로 종료된 경우에만 글로벌 상태의 모든 변경사항이 기록되고 전체가 실행된다.
    - 오류로 인해 실행이 실패하면 모든 상태 변경이 트랜잭션이 실행되지 않은 것처럼 롤백된다.
    - 실패한 트랜잭션은 시도된 것으로 기록되면 가스로 소비된 이더는 원 계정에서 차감되지만, 컨트랙트 또는 계좌 상태에 영향을 미치지 않는다

2.5 Contract 삭제

- 컨트랙트 코드는 변경할 수 없으나 삭제 할 수 있다.
- 컨트랙트를 삭제하면 해당 주소에서 코드와 내부 상태(스토리지)를 제거하고 빈 계정으로 남김으로 자원을 반환하는 효과가 있다.
- 컨트랙트를 삭제하려면 SELFDESTRUCT라는 EVM 연산코드를 실행해야 한다.
- SELFDESTRUCT 기능은 컨트랙트 작성자가 해당 기능을 프로그래밍한 경우에만 사용할 수 있다.
- 컨트랙트를 삭제하면 가스 환불이 일어난다.

---

1 Solidity

> 스마트 컨트랙트를 작성하는데에는 부작용(side-effect)이 없는 선언형 프로그래밍 언어가 더 적당합니다. 하지만 많은 수의 개발자들이 명령형 프로그래밍 언어를 사용하고 있는 현실을 무시하기는 쉽지 않습니다. 현실을 반영하듯이 대표적인 이더리움 스마트 컨트랙트 프로그래밍 언어인 솔리디티 또한 명령형 프로그래밍 언어입니다.

- 프로그래밍 언어를 저차원(Low-Level)과 고차원으로 구분했을 때 비트코인 스크립트 언어와 달리 고

차원 언어.
- C++, Python, 자바스크립트 등 기존에 널리 사용되고 있던 언어에 영향을 받았기에 어렵지 않게 학습이 가능.
- 스마트 계약 작성을 위해 탄생한 언어로써 언어의 모든 요소가 스마트 계약을 쉽게 작성할 수 있도록 설계됨.
- 튜링 완전성으로 인해 표현의 제약없이 자유롭게 어떠한 형태의 경제 활동이든 프로그래밍이 가능.

1.1 solc

- 솔리디티 언어로 작성된 프로그램을 EVM 바이트 코드로 변환하는 솔리디티 컴파일러이다.

설치(mac)

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

1.2 solc 버전 선택

- 솔리디티 프로그램에는 호환 가능한 솔리디티 최소 및 최대 버전을 지정하고 컨트랙트를 컴파일 하는데 사용할 수 있는 `pragma` 지시문이 포함될 수 있다.
- 솔리디티 컴파일러는 버전 `pragma` 를 읽고 컴파일러 버전이 저번 pragma와 호환되지 않으면 오류가 발생
- pragma 지시문은 비이트코드로 컴파일되지 않는다
    - 호환성 검사를 위해 컴파일러에서만 사용됨

2 단순한 솔리디티 프로그램 작성

```
pragma solidity 0.4.19;

contract Faucet {
    function () public payable {}

    function withdraw(uint withdraw_amount) public {
        require(withdraw_amount <= 100000000000000000);

        msg.sender.transfer(withdraw_amount);
    }
}
```

2.1 컴파일

- 솔리디티로 작성한 스마트 컨트랙트는 EVM에서 사용되는 바이트 코드로 컴파일되어야 한다.
- 16진수로 시리얼라이즈된 바이너리 결과물

```
$ solc --optimize --bin Faucet.sol
======= Faucet.sol:Faucet =======
Binary:
608060405234801561001057600080fd5b5060cc8061001f6000396000f3fe6080604052
600436106
01f5760003560e01c80632e1a7d4d14602a576025565b36602557005b600080fd5b34801
560355760
0080fd5b506050606004803603602081101560 4a57600080fd5b50356052565b005b670163
45785d8a0
000811115606657600080fd5b60405133908215 6108fc029083906000818181858888f19
350505050
1580156092573d6000803e3d6000fd5b505056fea2646970667358221220 5cf23994b22f
7ba19eee5
6c77b5fb127bceec1276b6f76ca71b5f95330ce598564736f6c63430006040033
```

2.2 이더리움 컨트랙트 ABI

- ABI(Application Binary Interface)는 두 프로그램 모듈 간의 인터페이스다
  - 주로 운영체제와 사용자 프로그램 사이
- 데이터 구조와 함수가 어떻게 기계 코드에서 사용되는지 그 방법을 정의한다.
- ABI는 기계 코드와 데이터를 교환하기 위해 인코딩 및 디코딩하는 기본 방법이다.
- ABI는 함수 설명 및 이벤트의 JSON 배열로 지정된다.
  - 함수의 필드: type, name, inputs, outputs, constant, payable
  - 이벤트의 필드: type, name, inputs, anonymous

ABI의 목적

- 컨트랙트에서 호출할 수 있는 함수를 정의하고 각 함수가 인수를 받아들이고 결과를 반환하는 방법을 설명하는 것
- 지갑이나 디앱 브라우저와 같은 애플리케이션은 올바른 인수와 인수 타입으로 컨트랙트의 함수들을 호출하는 트랜잭션을 구성할 수 있다.

Faucet.sol의 ABI

- 이 json은 일단 배포되면 Faucet 컨트랙트에 접근하는 모든 애플리케이션에서 사용할 수 있다.

```
$ solc --abi Faucet.sol
======= Faucet.sol:Faucet =======
Contract JSON ABI
[{"inputs":
[{"internalType":"uint256","name":"withdraw_amount","type":"uint256"}],
\
"name":"withdraw","outputs":
[],"stateMutability":"nonpayable","type":"function"}, \
{"stateMutability":"payable","type":"receive"}]
```

```
[
  {
    "constant": false,
    "inputs": [
      {
        "name": "withdraw_amount",
        "type": "uint256"
      }
    ],
    "name": "withdraw",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "payable": true,
    "stateMutability": "payable",
    "type": "fallback"
```

```
    }
  ]
```

3 데이터 타입

4 사전 정의된 글로벌 변수 및 함수

- 컨트랙트가 EVM에서 실행되면 몇개의 글로벌 객체에 접근할 수 있다.
- 또한 사전 정의된 함수로 다수의 EVM 연산 코드가 제공된다.
- 아래는 컨트랙트 내에서 접근할 수 있는 변수와 함수이다.

4.1 메시지 컨텍스트

msg

- 메시지 객체
- 컨트랙트 실행을 시작한 트랜잭션 호출(EOA 발신) 또는 메시지 호출(컨트랙트 발신)

msg.sender

- 컨트랙트 호출을 시작한 주소
- 컨트랙트도 다른 컨트랙트를 호출할 수 있기 때문에 EOA 주소나 컨트랙트 주소가 될 수 있다.
- 컨트랙트에서 다른 컨트랙트를 호출할 때마다 msg의 모든 속성 값이 새 발신자의 정보를 반영하도록 변경된다는 점에 주의해야 한다.
- 원래 msg 컨텍스트 내에서 다른 컨트랙트/라이브러리의 코드를 실행하는 delegatecall 함수는 예외

다.

msg.value

- 컨트랙트 호출과 함께 전송된 이더의 값(웨이)

msg.gas

- 남은 가스양
- 솔리디티 버전 0.4.21에서는 gasleft로 대체

msg.data

- 데이터 페이로드

msg.sig

- 함수 선택자인 데이터 페이로드의 처음 4바이트

4.2 트랜잭션 컨텍스트

tx

- 트랜잭션 객체

tx.gasprice

- 트랜잭션을 호출하는 데 필요한 가스 가격

tx.origin

- 이 트랜잭션에 대한 원래 EOA 주소
- 안전하지 않다

4.3 블록 컨텍스트

block

- 블록 객체

block.blockhash(blockNumber)

- 지정된 블록 번호의 블록 해시
- 더이상 사용하지 않는다
- blockhash 함수로 대체

bock.coinbase

- 채굴자 주소

block.difficulty

- 현재 블록 난이도

block.gaslimit

- 블록에 포함된 모든 트랜잭션이 사용할 수 있는 최대 가스양

block.number

- 현재 블록 번호

block.timestamp

- 채굴자가 현재 블록에 넣은 타임스탬프

4.4 address

address

- 입력으로 전달되거나 컨트랙트 객체에서 형변환되는 주소 객체

address.balance

- 웨이로 표현된 주소의 잔액
- 현재 컨트랙트의 잔액: `address(this).balance`

address.transfer(amount)

- address로 금액(웨이)를 전송한다.
- 오류가 발생하면 예외를 발생시킨다.

address.send(amount)

- address로 금액(웨이)를 전송한다.
- 오류가 발생하면 false를 리턴한다.

address.call(payload)

- 저수준 CALL함수
- 오류가 발생하면 false를 리턴한다

address.callcode(payload)

- address.call(payload)과 같지만 이 컨트랙트의 코드가 주소의 코드로 대체된 저수준 CALLCODE 함수다
- 오류가 발생하면 false를 리턴한다

address.delegatecall()

- address.callcode(payload)와 같지만 현재 컨트랙트에서 볼 수 있는 전체 msg 컨텍스트가 있는 저수준 DELEGATECALL 함수다
- 오류가 발생하면 false를 리턴한다

4.5 내장 함수

addmod, mulmod

- 모듈로 연산

keccak256, sha256, sha3, ripemd160

- 해시 함수

ecrecover

- 서명에서 주소를 복구

selfdestruct(recipient_address)

- 컨트랙트 삭제, 해당 주소로 이더를 환불해 준다.

this

- 현재 실행 중인 컨트랙트 계정 주소

5 컨트랙트 정의

- 솔리디티의 주요 데이터 타입은 contract이다
- 객체 지향 언어의 객체와 마찬가지로 컨트랙트는 데이터와 메서드가 포함된 컨테이너다

5.1 함수

- 컨트랙트 내에서 EOA 트랜잭션이나 다른 컨트랙트에 의해 호출될 수 있는 함수.
- 함수의 선언 구문은 아래와 같다

```
function FunctionName([parameters]) {public|private|internal|external}
[pure|constant|view|payable] [modifiers] [returns (return types)]
```

FunctionName

- 함수 이름.
- 이름 없이 정의될 수 있는 함수는 fallback 함수라고 부르고 다른 함수 이름이 없을 때 호출되며 인수가 없고 반환할 수도 없다.

parameters

- 인수

함수의 가시성

- `{public|private|internal|external}` 는 함수의 가시성을 나타낸다.
- public : 공개함수. 다른 컨트랙트, 트랜잭션에서 호출 가능
- external : 외부 함수. 키워드 this가 붙지 않으면 컨트랙트 내에서 호출할 수 없음.
- internal : 내부 함수. 컨트랙트 내에서만 접근 가능. 다른 컨트랙트, 트랜잭션에서 호출할 수 없고 파생된 컨트랙트에서는 호출 가능
- private : 비공개 함수. 파생된 컨트랙트에서도 호출할 수 없다.

함수의 동작

- `[pure|constant|view|payable]` 은 함수의 동작을 설명한다.

- constant or view : 상태를 변경하지 않음 솔리디티 v0.5 이상부터 constant 대신 view를 써야한다.
- pure : 순수 함수. 스토리지에서 변수를 읽거나 쓰지 않는다. 인수에 대해서만 작동하고 데이터 반환. 부작용 없음
- payable : payable 선언에 따라 입금 여부 판별.

5.2 컨트랙트 생성자

- 컨트랙트가 생성될 때 생성자 함수가 있는 경우 이를 실행하여 상태를 초기화한다.
- 생성자는 컨트랙트 생성 트랜잭션과 동일한 트랜잭션에서 실행된다.
- 생성자는 선택사항
- 생성자는 오직 한 번만 실행된다.

생성자 방식 1

```
contract MEContract {
  function MEContract(){
    //초기화
  }
}
```

생성자 방식 2

```
contract MEContract {
  constructor (){
    //초기화
  }
}
```

5.3 컨트랙트 삭제

- 컨트랙트는 SELFDESTRUCT라는 특수한 EVM 연산코드에 의해 소멸된다.
- selfdestruct는 SELFDESTRUCT를 포함하는 고수준 내장 함수이다.
  - selfdestruct는 하나의 인수를 받는데 컨트랙트 계정에 남아 있는 이더를 받기위한 주소를 의미한다.
- 삭제 가능한 컨트랙트를 생성하기 위해선 selfdestruct를 명시적으로 추가해야한다.

컨트랙트 삭제 예시 코드

```
pragma solidity ^0.4.22;

contract Faucet {
  address owner;

  constructor() {
    owner = msg.sender;
  }

  function withdraw(uint256 withdraw_amount) public {
    require(withdraw_amount <= 10000000000000);
    msg.sender.transfer(withdraw_amount);
  }


  function() external payable {}

  function destroy() public {
    require(msg.sender == owner);
    selfdestruct(owner);
  }

}
```

5.4 함수 변경자

- 특별한 유형의 함수
- 함수 선언에 modifier라는 이름을 추가하여 함수에 변경자를 적용한다.
- 변경자는 컨트랙트 내에서 함수에 적용되어야 할 조건 생성하기 위해 사용한다.

함수 변경자 예시

- 함수 변경자의 이름은 `onlyOwner` 이다
- 함수 변경자를 적용한 모든 함수에 아래의 조건을 설정한다.
- `_;` 플레이스 홀더로 이 부분에 수정된 코드가 삽입된다.

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

함수 변경자 적용

- 변경자를 적용하려면 함수 선언에 변경자 이름을 추가한다.
- 함수에 둘 이상의 변경자를 적용할 수 있다.
  - 쉼표로 구분된 리스트로 선언된 순서대로 적용된다.
- destroy 함수에 onlyOwner 적용

```
function destroy() public onlyOwner {
     selfdestruct(owner);
}
```

5.5 컨트랙트 상속

- 컨트랙트 객체는 바탕이되는 컨트랙트에 기능들을 추가해서 확하기 위한 메커니즘인 상속을 지원한다.
- 이때 `is` 키워드를 사용한다.
- 다중 상속을 지원한다.
- 아래는 child가 parent1와 parent2의 모든 메소드, 기능, 및 변수를 상속한다.

```
contract child is parent1, parent2{
  ...
}
```

상속 예시

```
pragma solidity ^0.6.4;

contract Owned {
    address payable owner;

    // Contract constructor: set owner
    constructor() public {
        owner = msg.sender;
    }

    // Access control modifier
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}

contract Mortal is Owned {
    // Contract destructor
```

```
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract Faucet is Mortal {
    // Accept any incoming amount
    function () public payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {
        // Limit withdrawal amount
        require(withdraw_amount <= 0.1 ether);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```