Understanding Account-based Blockchain

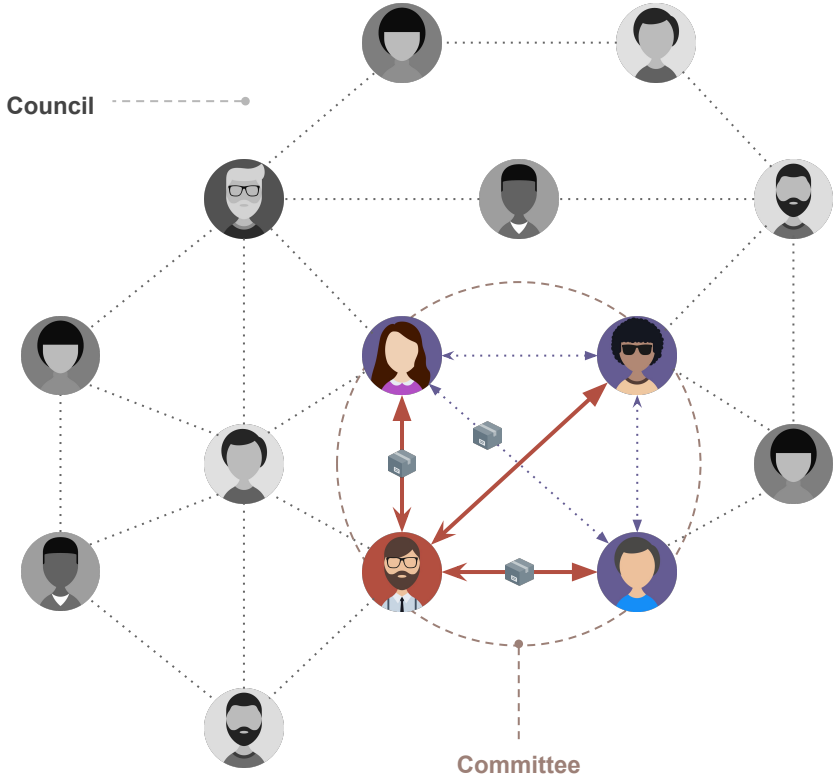
2019.09.19

From Last Class

- 블록체인 101
- 공개키암호, 전자서명
- 합의구조 (PoW, PoS, BFT)

Klaytn BFT

- Klaytn은 확장가능한 BFT를 사용
  - N개의 노드 가운데 S개의 부분노드 집합을 확률적으로 선택 (where N is large, and S is sufficiently small)
  - 전체집합을 거버넌스 카운실(Governance Council),
  - 부분집합을 커미티(Committee)로 정의
  - 커미티 선택은 VRF*로 구해진 무작위값에 기반
- 매 블록마다 새 커미티를 뽑아 BFT를 실행
- 기존의 BFT에 비해 확장성을 크게 개선



* Verifiable Random Function

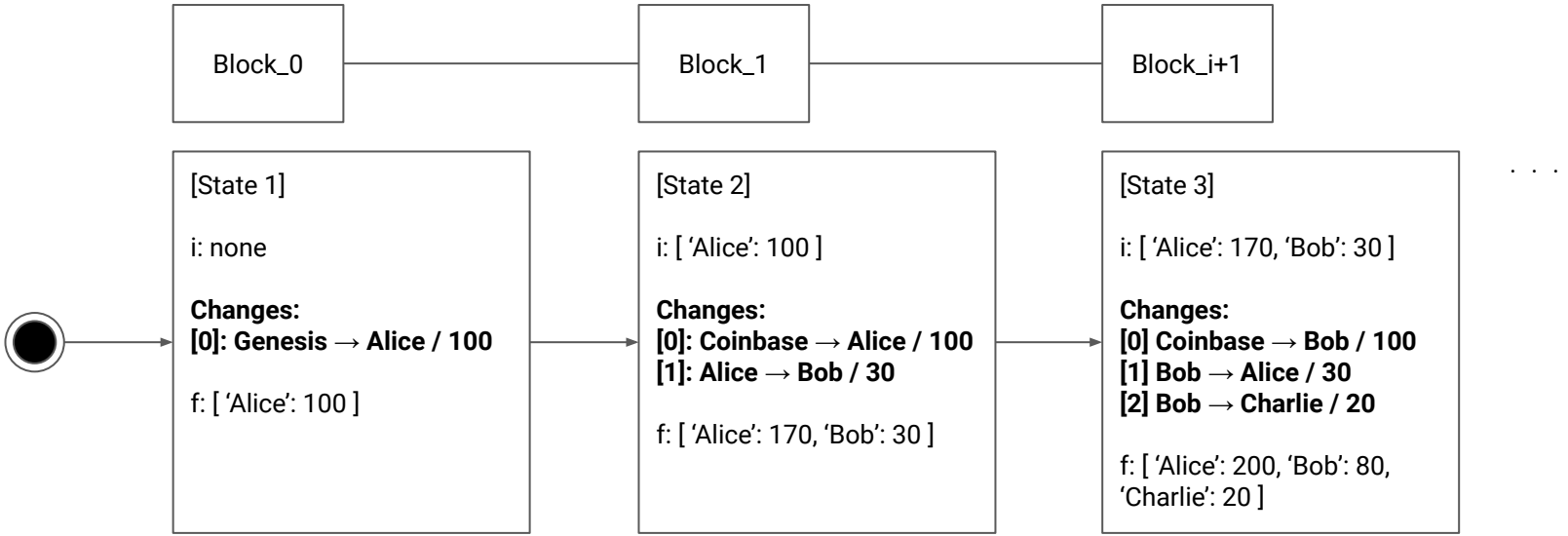
Today’s Agenda

- Blockchain State
- Transaction
- Smart Contract
- Using Klaytn SDK

Blockchain State

(어카운트 기반) 블록체인의 상태

블록체인은 트랜잭션으로 변화하는 상태 기계 (State Machine)



상태 기계

- 블록체인은 **초기 상태**에서 **변경사항**을 적용하여 **최종 상태**로 변화하는 상태 기계
  - 이전 블록의 최종 상태(final state)는 현재 블록의 초기 상태(initial state)
  - Gen block의 경우 임의의 초기값들이 설정되는데 이것이 gen block의 초기상태이자 최종상태
- (어카운트 기반) 블록체인의 상태
  - 블록체인의 상태 = 블록들에 저장된 트랜잭션(TX)들을 순차적으로 실행하여 나온 결과
  - TX는 어카운트를 생성하거나 변경
    - e.g., Alice가 기존에 존재하지 않던 주소 X에 1 ETH를 전송하면 새로운 EOA가 생성
    - e.g., Alice가 새로운 스마트 컨트랙트를 생성/배포하면 새로운 스마트 컨트랙트가 블록체인에 추가
    - e.g., Alice가 Bob에게 5 ETH를 전송하는 TX가 체결되면 Alice의 Bob의 잔고가 변경
  - 항상 같은 결과를 보장하기 위해 하나의 TX가 반영되는 과정에서 다른 TX의 개입은 제한됨

Transactions

(Recall) Ethereum 어카운트의 종류

1. **External Account**: 사용자(end user)가 사용하는 어카운트 (a.k.a. EOA)
2. **Contract Account**: 스마트 컨트랙트를 표현하는 어카운트

- Ethereum은 EOA와 스마트 컨트랙트의 상태를 기록 및 유지
  - 스마트 컨트랙트는 특정주소에 존재하는 **실행 가능한 프로그램**
  - 프로그램은 상태를 가지기 때문에 Ethereum/Klaytn은 스마트 컨트랙트를 어카운트로 표현
- EOA는 블록에 기록되는 TX를 생성
  - 블록에 기록되는 TX들은 명시적인 변경을 일으킴 (e.g., 토큰 전송, 스마트 컨트랙트 배포/실행)

트랜잭션(TX)과 가스(Gas)

- TX의 목적은 블록체인의 상태를 변경하는 것
  - TX는 보내는 사람(sender, from)과 받는 사람(recipient, to)이 지정되어 있으며
  - to가 누구냐에 따라 TX의 목적이 세분화
- Gas: TX를 처리하는데 발생하는 비용
  - TX를 처리하는데 필요한 자원(computing power, storage)을 비용으로 전환한 것이 가스(gas)
  - Sender는 TX의 처리를 위해 필요한 가스의 총량과 같은 가치의 플랫폼 토큰을 제공해야함
  - 이때 지출되는 플랫폼 토큰을 가스비(Gas Fee)라 정의; 가스비는 블록을 생성한 노드가 수집
  - Ethereum은 sender가 임의로 per gas 가격을 설정할 수 있도록 허용
  - Klaytn은 sender가 임의로 per gas 가격을 설정할 수 없도록 gas price를 프로토콜에서 고정

트랜잭션과 서명

- 플랫폼은 sender가 TX가 처리되는데 필요한 가스비를 가지고 있는지 확인
  - 가스비 확인은 구현에 따라 상이
  - Ethereum/Klaytn은 노드가 TX를 수신함과 동시에 가스비 이상의 balance가 있는지 확인
  - TX의 체결과 동시에 sender의 balance에서 가스비를 차감
- TX는 sender의 서명(v, r, s)이 필요
  - 어카운트의 balance를 사용하기 때문
  - 서명의 증명은 구현마다 상이
    - Ethereum: 서명 → 공개키 도출 → 어카운트 주소 도출 → 어카운트 존재유무 확인
    - Klaytn: from 주소 확인 → 저장된 공개키 불러오기 → 서명 직접 검증

트랜잭션 예시

```
{
    nonce: 1,
    from:  ‘0xd0ea3e0eabaea095ea3ba231c043dbf8c0feb40a’,
    to:    ‘0x5e47b195eeb11d72f5e1d27aebb6d341f1a9bedb’,
    value: 10,
    // omitted other fields for brevity
    // platform specific fields are required
}
```

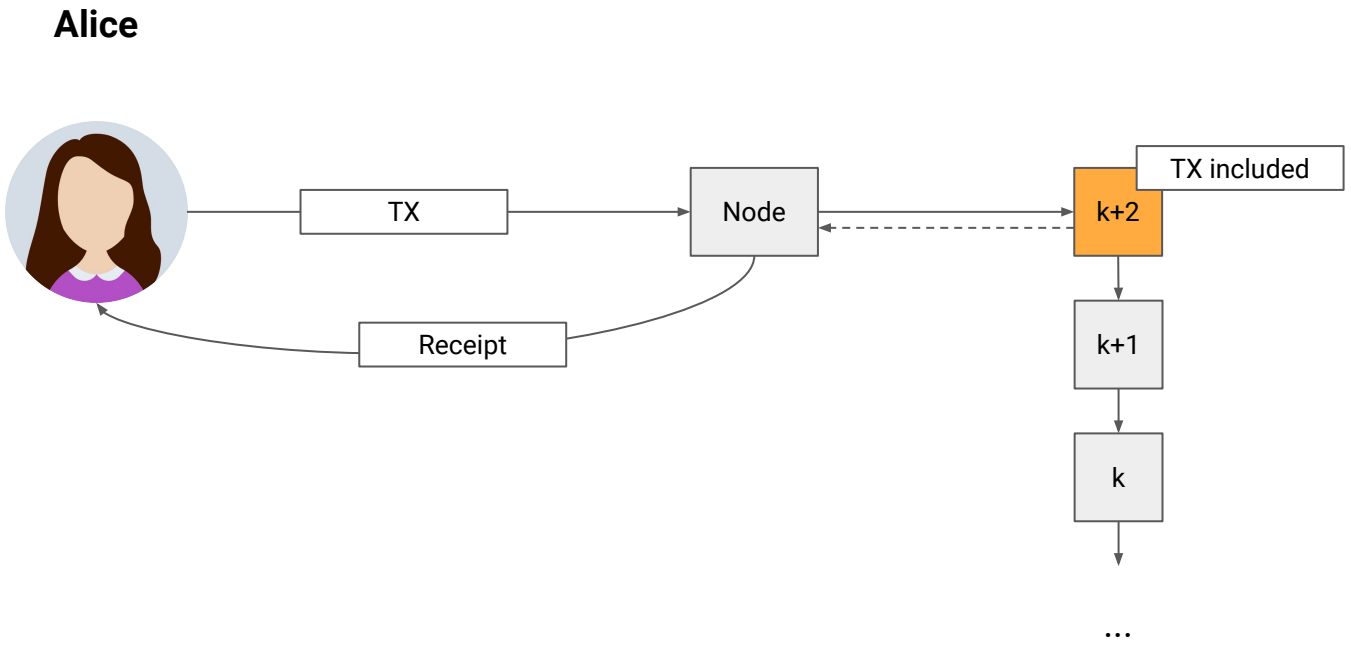
Ethereum 트랜잭션 예시

```
{
    nonce: '0x01',
    gasPrice: '0x4a817c800',         // 20,000,000,000
    gas: '0x5208',                   // 21,000
    value: '0xde0b6b3a7640000',      // 1,000,000,000,000,000,000 = 1 ETH
    to: '0x3535353535353535353535353535353535353535',
    v: '0x25',
    r: '0x4f4c17305743700648bc4f6cd3038ec6f6af0df73e31757007b7f59df7bee88d',
    s: '0x7e1941b264348e80c78c4027afc65a87b0a5e43e86742b8ca0823584c6788fd0'
    // some fields are omitted for brevity
}
```

Klaytn 트랜잭션 예시

```
{
    type: 'VALUE_TRANSFER',
    nonce: '0x01',
    gasPrice: '0x05d21dba00',
    gas: '0x0493e0',
    value: '0x01',
    to: '0xef5cd886c7f8d85fbe8023291761341acbb4da01',
    from: '0xd01b138d49fb248325509771dff44ba25a5e74f1',
    v: '0x07f6',
    r: '0x5f36aad4e3ac680c1541f0306143dbaa96c686e39857931f449a23606fa3b3f3',
    s: '0x709eed912c6b3be6cfd6c185701cb3f736196fb015dfd2a15e0d03f3d74d870e'
}
```

Transaction Journey



Alice와 Node 사이 통신

Alice → Node

- Alice는 TX를 생성, 서명하여 Node에게 전달
- 이때 데이터 구조를 온전하게 전달하고자 RLP 알고리즘으로 TX를 직렬화
- Alice와 Node가 같은 프로토콜로 통신하는 것이 중요

Node → Alice

- 올바른 TX 수신 시 TX 해시를 반환
- TX 체결 시 Receipt를 반환; 소요된 Gas, TX 해시, input, output 등이 기록

Smart Contracts

스마트 컨트랙트

- 특정 주소에 배포되어 있는 TX로 실행 가능한 코드
  - 스마트 컨트랙트 소스코드는 **함수**와 **상태**를 표현
  - 컨트랙트는 어카운트로 취급; 컨트랙트 소스코드는 블록체인에 저장
  - 함수는 상태를 변경하는 함수, 상태를 변경하지 않는 함수로 분류
  - 스마트 컨트랙트는 어카운트이기 때문에 주소를 부여
  - 사용자(end user, EOA owner)가 스마트 컨트랙트 함수를 실행하거나 상태를 읽을 때 주소가 필요
- 스마트 컨트랙트는 사용자가 실행
  - 상태를 변경하는 함수를 실행하려면 그에 맞는 TX를 생성하여 블록에 추가 (TX 체결 = 함수의 실행)
  - 상태를 변경하지 않는 함수, 상태를 읽는 행위는 TX가 필요 없음 (노드에서 실행)

Solidity

- Ethereum/Klaytn에서 지원하는 스마트 컨트랙트 언어
- Klaytn은 Solidity 버전 0.4.24, 0.5.6을 지원
- 일반적인 프로그래밍 언어와 그 문법과 사용이 유사하나 몇가지 제약이 존재
    - e.g., 포인터의 개념이 없기 때문에 recursive type의 선언이 불가능

Contract = Code + Data

- Solidity 컨트랙트는 코드(함수)와 데이터(상태)로 구성
- Solidity 함수는 코드 안에 변수로 선언된 상태를 변경하거나 불러옴
- 아래 예시에서 `set`, `get`은 함수, `storedData`는 상태

```
contract SimpleStorage {
    uint storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() public view returns (uint) {
        return storedData;
    }
}
```

Solidity 예제 - Coin 컨트랙트

```
// Solidity로 간단한 포인트 시스템을 구현

// [Coin 컨트랙트]
// 컨트랙트 생성자가 관리하는 포인트 시스템 컨트랙트로 포인트 시스템 고유의
// 주소공간(address space)을 가지며 각 주소의 포인트 잔고를 기록한다.
// 컨트랙트 생성자는 사용자 주소(e.g., 0xALICE)에 포인트를 부여할 수 있고
// 사용자는 다른 사용자에게 포인트를 전송할 수 있다 (e.g., 0xALICE → 0xBOB, 10 Coins)
contract Coin {
    // [omitted for brevity]
}
```

Solidity 예제 - 상태 (State Variables)

```
// “pragma solidity” 키워드는 Solidity 버전을 지정
pragma solidity ^0.5.6;


// “contract X { … }”는 X라는 컨트랙트를 정의
contract Coin {

      // “minter”는 address 타입으로 선언된 상태; address 타입은 Ethereum에서 사용하는 160-bit 주소
      // “public” 키워드는 상태를 다른 컨트랙트에서 읽을 수 있도록 허용
      address public minter;


      // “balances”는 mapping 타입으로 address 타입 데이터를 key로, uint 타입 데이터를 value로 가지는 key-value mapping
      // mapping은 타 프로그래밍 언어에서 사용하는 해시테이블 자료구조와 유사 (uninitialized 값들은 모두 0으로 초기화되어 있는
상태)
      mapping (address => uint) public balances;


      // [omitted for brevity]
}
```

Solidity 예제 - 이벤트 (Events)

```
contract Coin {
      // [omitted state variables for brevity]

      // event로 정의된 타입은 클라이언트(e.g., application using a platform-specific SDK/library)가
      // listening 할 수 있는 데이터로 emit 키워드로 해당 타입의 객체를 생성하여 클라이언트에게 정보를 전달
      // usage:
      // /* in Solidity */
      // emit Sent(an_address, another_address, 10);
      // /* in Web3.js */
      // Coin.Sent().watch({}, '', function(err, result) { … });
      event Sent(address from, address to, uint amount);

      // [omitted for brevity]
}
```

Solidity 예제 - 생성자 함수 (Constructor)

```
// N.B. 컨트랙트 함수는 함수를 실행한 TX의 정보를 받을 수 있는데 해당 정보를 msg 변수로 접근
// 자세한 정보는 https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html#block-and-transaction-properties를 참조
contract Coin {
	// [omitted state variables and event definitions for brevity]

	// 생성자 함수는 컨트랙트가 생성될 때 한번 실행
	// 아래 함수는 minter 상태변수에 msg.sender값을 대입 (함수를 실행한 사람의 주소)
	constructor() public {
		minter = msg.sender;
	}

	// [omitted for brevity]
}
```

Solidity 예제 - 함수 (Functions) 1/2

```
contract Coin {
      // [omitted state variables, event definitions, and constructors for brevity]

      // receiver 주소에 amount 만큼의 새로운 Coin을 부여
      // require 함수는 입력값이 true일때만 다음으로 진행할 수 있음 (타 언어의 assert와 유사)
      // require 함수는 특정조건을 만족할 경우에만 함수를 실행할 수 있도록 강제할 때 사용
      function mint(address receiver, uint amount) public {
            require(msg.sender == minter);   // 함수를 실행한 사람이 minter(i.e., 컨트랙트 소유자)일때만 진행
            require(amount < 1e60);          // 새로 생성하는 Coin의 양이 1 * 10^60개 미만일때만 진행
            balances[receiver] += amount;    // receiver 주소에 amount만큼을 더함
      }

      // [omitted for brevity]
}
```

Solidity 예제 - 함수 (Functions) 2/2

```
contract Coin {
      // [omitted state variables, event definitions, and constructors for brevity]

      // msg.sender가 receiver에게 amount만큼 Coin을 전송
      function send(address receiver, uint amount) public {
            require(amount <= balances[msg.sender], "Insufficient balance."); // 잔고가 충분한지 확인
            balances[msg.sender] -= amount;              // 잔고 차감
            balances[receiver] += amount;                // 잔고 증가
            emit Sent(msg.sender, receiver, amount);     // 이벤트 생성
      }
}
```

Solidity 소스코드 컴파일링

- Solidity 소스코드는 배포에 앞서 Ethereum Virtual Machine (EVM)에서 실행 가능한 형태로 컴파일 (변환) 되어야 함
- **`solc`** - Solidity 컴파일러
  - npm (linux, macos, Windows)으로 light version을 설치 가능 (**`solcjs`**, partial feature)

```
$ npm install -g solc
```

  - apt (debian linux), brew (macos) 등으로 binary 설치 가능 (**`solc`**, full feature)

```
(debian)
$ sudo add-apt-repository ppa:ethereum/ethereum
$ sudo apt-get update
$ sudo apt-get install solc

(macos)
$ brew tap ethereum/ethereum
$ brew install solidity
```

Solidity 소스코드 컴파일링

- SimpleStorage 컨트랙트를 정의하는 test.sol 이 있다고 가정
- 다음과 같은 방법으로 test.sol을 컴파일

```
$ ls
test.sol
$ solcjs --bin true --abi true -o out test.sol
$ ls
out test.sol
$ tree out
out
├── test_sol_SimpleStorage.abi
└── test_sol_SimpleStorage.bin

0 directories, 2 files
```

Bytecode & ABI

Solidity 소스코드(.sol 파일)를 컴파일하면 Bytecode(.bin 파일)와 ABI(.abi 파일)가 생성

Bytecode

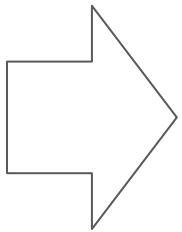
- 컨트랙트를 배포할 때 블록체인에 저장하는 정보
- Bytecode는 Solidity 소스코드를 EVM이 이해할 수 있는 형태로 변환한 것
- 컨트랙트 배포시 HEX로 표현된 Bytecode를 TX에 담아 노드에 전달

ABI (Application Binary Interface) a.k.a. JSON interface

- ABI는 컨트랙트 함수를 JSON 형태로 표현한 정보로 EVM이 컨트랙트 함수를 실행할 때 필요
- 컨트랙트 함수를 실행하려는 사람은 ABI 정보를 노드에 제공

Bytecode 예시

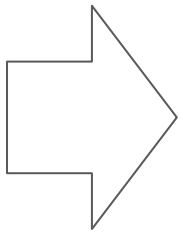
```
contract SimpleStorage {
    uint storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() public view returns (uint) {
        return storedData;
    }
}
```



```
608060405234801561001057600080fd5b5060df8061001
f6000396000f3006080604052600436106049576000357c
0100000000000000000000000000000000000000000000000
0000000000900463ffffffff16806360fe47b114604e57
80636d4ce63c146078575b600080fd5b348015605957600
080fd5b506076600480360381019080803590602001909 2
919050505060a0565b005b34801560835760080fd5b506
08a60aa565b60405180828152602001915050604051809 1
0390f35b80600081905550565b600080549050905600a
165627a7a723058205f8cbe0b8a8e29b1e772c0873cf13b
e7a72f7f6592358e8ceaeeafa367b5896d0029
```

ABI 예시

```
contract SimpleStorage {
    uint storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() public view returns (uint) {
        return storedData;
    }
}
```



```
[{    "constant":false,
      "inputs":[{"name":"x","type":"uint256"}],
      "name":"set",
      "outputs":[],
      "payable":false,
      "stateMutability":"nonpayable",
      "type":"function" }, {
      "constant":true,
      "inputs":[],
      "name":"get",
      "outputs":[{"name":"","type":"uint256"}],
      "payable":false,
      "stateMutability":"view",
      "type":"function" }]
```

Using Klaytn SDK

Klaytn SDK (Software Development Kit)

- Klaytn은 BApp 개발을 위해 필요한 SDK를 제공
- caver-js는 Node.js로 Klaytn BApp을 만들때 필요한 라이브러리를 제공
- 다음 온라인 문서에서 사용방법을 확인: https://docs.klaytn.com/sdk/caverjs

개발환경 셋팅

Node.js 설치

- https://nodejs.org에서 10.16.3 LTS 설치 (installer/pkg 실행)

개발 디렉토리 생성 및 Caver-js 설치

- 성공적으로 Node.js를 설치한 뒤 원하는 위치에 개발 디렉토리를 생성

```
$ mkdir Count && cd Count
```

- 디렉토리 생성 후 npm으로 Node.js 프로젝트를 초기화, caver-js를 설치

```
$ npm init
...
$ npm install caver-js
```

Baobab 테스트넷에 연결

```
const Caver = require('caver-js');
const caver = new Caver('https://api.baobab.klaytn.net:8651/');
```

klay.getBlockNumber()

```
const Caver = require('caver-js');
const caver = new Caver('https://api.baobab.klaytn.net:8651/');
// getBlockNumber() returns a Promise object returning Number
caver.klay.getBlockNumber(function(err, blockNumber) {
    console.log(blockNumber);
});
// alternatively
caver.klay.getBlockNumber().then(console.log);
7092446
```

klay.accounts.wallet

```
const account = caver.klay.accounts.create();

// in-memory wallet
const wallet = caver.klay.accounts.wallet;
wallet.add(account);

console.log(wallet.length);            // wallet에 저장된 어카운트 갯수를 리턴
console.log(wallet[account.address]); // 해당 주소를 가지는 어카운트를 불러옴; 없을 경우 undefined
console.log(wallet[0]);                // 저장된 첫번째 어카운트를 불러옴; 없을 경우 undefined
1
{ address: '0x38aed90665...c4fb53faae', privateKey: '0xfca84955...a96e5c5f', index: 0 }
{ address: '0x38aed90665...c4fb53faae', privateKey: '0xfca84955...a96e5c5f', index: 0 }
```

토큰전송TX 생성 & 서명

```
wallet.clear(); wallet.create(2);     // in-memory wallet 초기화 & 어카운트 2개 생성
const tx = {
    type: "VALUE_TRANSFER",                // Klaytn은 TX 타입을 지정
    from: wallet[0].address,               // 첫번째 어카운트 주소
    to: wallet[1].address,                 // 두번째 어카운트 주소
    value: caver.utils.toPeb('1', 'KLAY'), // 1 KLAY 전송
    gas: 300000                            // TX가 사용할 수 있는 가스총량
};
// 첫번째 어카운트의 비밀키로 서명
caver.klay.accounts.signTransaction(tx, wallet[0].privateKey).then(console.log);
{ messageHash: '0xc95641e147622735347d69e33ce8d5704f1dd240c86ba078d8118b2881b81742',
  v: '0x07f5', r: '0x1de39143f3...19e991d31e', s: '0x53d86c54...bf00e6db',
  rawTransaction: '0x08f87f808505d21d...ed0f06bf00e6db',
  txHash: '0xb57dd4ec...1f509ea2', senderTxHash: '0xb57dd4ec...1f509ea2' }
```

서명된 TX 전송

```
const tx = { ... };
(async () => {
      const signedTransaction = await caver.klay.accounts.signTransaction(tx, sender.privateKey);
      await caver.klay.sendSignedTransaction(signedTransaction.rawTransaction)
            .on('transactionHash', function(txhash) {
                  console.log('hash first', txhash);
            })
            .on('receipt', function(receipt) {
                  console.log('receipt later', receipt);
            })
            .on('error', function(err) {
                  console.error('something went wrong');
            });
})();
```

토큰전송TX + sendTransaction

```
const tx = { … };
caver.klay.sendTransaction(tx) // 서명 + 전송
      .on('transactionHash', function (txhash) {
            console.log("hash first", txhash);
      })
      .on('error', function (err) {
            console.error('something went wrong');
      })
      .on('receipt', function (receipt) {
            console.log("receipt later", receipt);
      });
```

스마트 컨트랙트 배포

```
// 앞서 예제에서 본 SimpleStorage 컨트랙트의 ABI와 Bytecode를 사용
const abi = [ … ];
const contract = new caver.klay.Contract(abi);
contract.deploy({ data: '6080604052348015...0029' })
      .send({
            from: wallet[1].address,
            gas: 3000000,
            value: 0
      })
      .on('receipt', function (receipt) {
            console.log('contract deployed at', receipt.contractAddress); // 컨트랙트 주소가 receipt에 포함
      })
```

스마트 컨트랙트 함수 실행 (mutation)

```
const contract = new caver.klay.Contract(abi, '0x20e199c44768F2C39Cb771D2F96B13fE6D63a411');
contract.methods.set(100)  // SimpleStorage의 set 함수를 실행; 상태를 바꾸는 함수이기 때문에 TX로 실행
    .send({
        from: wallet[1].address,
        gas: 300000
    })
    .on('error', function (hash) { … })
    .on('transactionHash', function (hash) { … })
    .on('receipt', function (receipt) { … });
```

스마트 컨트랙트 함수 실행 (constant)

```
const contract = new caver.klay.Contract(abi, '0x20e199c44768F2C39Cb771D2F96B13fE6D63a411');
// call 함수는 상태를 바꾸는 함수가 아니기 때문에 노드에서 바로 실행
contract.methods.get().call(null, function (err, result) {
    if (err == null) {
        console.log(result);
    } else {
        console.error(err);
    }
});
```

End of Document