# Chapter 12. DeFi

DeFi는 어떤 중개없이 수요자와 공급자가 직접 금융거래가 가능한 탈중화금융이다. 이를 위해 제안된 이더리움 표준 ERC20을 구현해보게 된다. ERC20 코인을 유통하는 증권시장과 같은 거래소에 대해 배운다.

# 1. 디파이는 대체 무엇일까?

DeFi는 Decentralized Finance의 줄여서 쓰는 말로, **탈중앙화금융**이라고 한다. 탈중앙화라고 이름이 붙는다고 하니, 필시 이전의 중앙화된 금융서비스와는 상반된다는 의미일 것이다. 자금을 관리하고 사용자 간의 자동화된 금융 서비스를 제공하는 개념은 동일하지만, 탈중앙화 방식이라는 점에서 차이가 있다.

탈중앙의 대척점에 있는 중앙화 금융을 이름 붙이면 CeFi (Centralized Finance)가 되겠다. 이는 지금까지 우리가 사용해왔던 입금, 출금, 계좌 이체, 보험 계약 등의 금융 서비스를 말한다. 이 모든 금융 서비스는 은행, 증권사, 카드사, 보험사와 같은 모두 중앙의 중개처를 통해서 거래가 이루어지고 있으며, 이를 CeFi라고 구분할 수 있다.

식사를 하고 카드 결제를 한다고 해보자. 카드 회사는 신용에 따라 결제를 승인하는데, 전형적인 CeFi 거래이다. 대출 또한 중앙의 은행을 통한 CeFi의 예이다.

반면에 DeFi는 탈중앙이라고 하는데, 뭔가 중앙의 승인이나 허락없이 직거래한다는 의미이다. 그렇다면 누가 승인을 하고, 지급이 허위가 아니라는 것을 증명할 것인가? 직거래를 생각해보자. 과연 모르는 사람을 신뢰하고 덥석 송금할 수 있겠는가?

블록체인은 매우 혁신적인 틀바뀜이다. 블록체인은 탈중앙을 가능하게 한다고 했으며, 예를 들어 송금을 한다면 블록체인에 참여하는 모두가 이를 승인하는 형식이어서 허위가 발생할 까닭이 없다. 이런 거래가 중앙에서 승인, 결제, 기록되는 것이 아니라 참여자(시스템)에 의해서 처리된다.

DeFi는 이러한 블록체인 상에서 운용되는 금융 상품 및 서비스를 말한다. DeFi는 늘상 일어나는 입출금뿐만 아니라, 암호화폐 대출, 대여, 토큰 교환, 코인을 맡기고, 이자를 코인으로 받는 이자농사 (Yield Farming) 또는 다른 가상자산과 거래하거나 인터넷으로 연결만 될 수 있으면 어느 누구나 사용할 수 있다.

DeFi는 2020년 이전까지만 해도 존재가 무색할 정도였지만, 암호화폐의 인기에 따라 자금의 규모가 급등하고 있다 (https://www.statista.com/statistics/1237821/defi-market-size-value-crypto-locked-usd/). 그렇다고 해서 은행을 대체할 규모는 아니지만, 적다고 무시할 수준은 아니다.

현금을 대체할 수 있어서 함부로 누구나 편한대로 만들도록 방치한다면 큰 혼란이 발생할 수 있다. 그래서 ERC (Ethereum Request for Comments) 표준이 필요한 이유이며, 이를 준수해서 제작하면 금융서비스에 대한 신뢰와 상호운용성을 확보할 수 있다.

ERC(Ethereum Request for Comment)는 이더리움 기술적인 요구사항을 표준화하는 방안이고, 뒤에 특정 방안에 해당하는 일련번호가 붙는다. 몇 가지 ERC를 예를 들면, ERC-20은 토큰 표준으로 여기서 구현하고, ERC-721은 NFT(Non-Fungible Token) 표준으로 다음 장에서 배우고, ERC-1155는 다중 자산 토큰 표준으로 널리 사용된다.

DeFi는 꼭 이더리움일 필요는 없지만, ERC20의 앞글자 Ethereum이 의미하듯 그 환경에서 제작한다. 현재 이더리움 메인넷에는 수십만개의 ERC20 코인이 존재한다 (https://etherscan.io/tokens). ERC20 코인의 가장 좋은 예는 현금을 담보하고 발행하는 Stable 코인을 꼽을 수 있다. Stable 코인은 실제 가치와 연동이 되어 있어, 가격의 변동이 없도록 설계된 암호화폐를 말한다. 현재 매우 많은 코인, 예를 들면 Chainlink (LINK), Tether (USDT), Shiba Inu (SHIB), Wrapped Bitcoin (WBTC), OmiseGO (OMG), 0x (ZRX), Maker (MKR), Augur (REP), Golem (GNT), Loopring (LRC), Basic Attention Token (BAT) 등이 발행되어 있다.

그렇다면 누구나 ERC토큰을 만들 수 있는 것일까? 우리도 예를 들어 시가 수 억원 규모의 디파이를 제작할 수 있는 것일까? 한국 은행에서 화폐를 발행하는 것처럼 우리가 화폐를 만들 수 있다는 건가? 그렇게만 된다면 누구나 쉽게 부자가 될 수 있겠지만, 현실적으로 그렇지는 않다. 충분히 신뢰할 수 없다면, 디파이는 화폐로서 저장, 교환 가치가 있을 수 없다. 가격 변동이 심해서 투자 차익을 노리기만 할 뿐 실생활에서 쓰이지 않는다면 화폐라고 보기 힘들다.

## STO

주식에 국한해서 토큰으로 제작하는 경우 이른바 STO(security token offerings), 즉 '토큰형 주식'이라고 한다.

지금까지의 실물 증권, 전자증권과는 다른 새로운 형태의 블록체인 상의 증권이다.

* 실물 증권은 증서에 기재되어 보관되고,
* 전자증권은 증권회사와 같은 중앙기관에 기재,
* 토큰 증권은 블록체인에 분산 기재되어 존재한다는 차이가 있다. 다만 공중망이 아니라, 허가형 또는 사적망의 블록체인을 적용하면 누구나 거래한다는 본래 취지가 제한될 수 있다.

STO는 주로 미술품, 부동산, 저작권 등을 대상으로 제작되고 있어 NFT와의 경계는 모호하다. 가장 큰 차이는 STO는 법의 적용인데, 우리나라는 자본 시장법의 규제와 감시를 받아야 한다. 부동산에 대해 STO를 만든다고 하자. 그러면 주식과 같이, 실물 부동산을 토큰 주식화하고, 공모하고 등록하고 상장을 해야 한다. 투자자는 증권회사나 허가받은 투자회사에 등록된 실물 부동산에 대해 쪼개기 투자를 하게 된다.

STO를 위한 ERC 표준은 ERC1400과 ERC1410이 제시되고 있다. STO는 자산 소유권이 부여되고, 그 에 따른 이익 배당이나 투표 권한 등의 권리가 되는 등 법적 강제가 있다는 점에서 ERC20와 큰 차이가 있다.

또한 블록체인의 익명성이 문제가 될 수 있다. 블록체인에서는 계정주소에 대해 토큰이 귀속된다. 그러나 키를 유실했다고 하면 토큰의 소유권 또한 실권된다. 기존 증권은 키를 잃어버려도 소유를 증명할 방법이 있다. 블록체인의 계정주소는 익명이라서, 유실되면 재발행할 수 있어야 한다.

# 2. 제작하는 방법

블록체인은 금융서비스에 특화된 플랫폼이고, 따라서 디파이는 매우 자연스러운 후보일 수 밖에 없다.

디파이는 ERC20을 구현해서 제작하게 된다. 프로그래밍 관점에서 보면, ERC20를 상속하는 기법을 통해 구현한다.

ERC20는 이더리움 토큰을 생성하고 관리하기 위한 가장 일반적인 표준으로, 대부분의 토큰은 이 요구사항을 지키면서 코딩하면 된다. 2015년 처음 Fabian Vogelsteller가 처음 표준화에 필요한 기능을 제안한 것으로 알려져 있다 (https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md). 이 표준을 준수하지 않으면 ERC20 토큰이라고 볼 수 없다.

# 3. 표준 인터페이스 IERC20

IERC20는 ERC20 표준을 준수하는 토큰을 다루기 위한 인터페이스이다.

IERC20는 잔고, 허가, 이체와 관련하여 다음 6개의 함수를 정의하고 있다.

## 3.1 잔고

* totalSupply() external view returns (uint) 토큰의 총 잔고를 출력한다.
* balanceOf(address account) external view returns (uint) account주소의 잔고를 출력한다.

## 3.2 허가

* approve(address spender, uint amount) external returns (bool) spender에게 amount를 허용하고, 성공 여부를 출력한다. 권리가 부여되고 나면, 그 금액은 owner의 소유일지라도 owner의 사적키(private key)가 없어도 거래를 수행할 수 있다. 그러고 나면, spender는 transferFrom으로 금액 한도 내에서 전송할 수 있다.
* allowance(address owner, address spender) external view returns (uint) owner가 spender에게 지출 허가한 금액을 반환한다.
* event Approval(address indexed owner, address indexed spender, uint value) 허가 함수가 호출되면 발생할 수 있도록 고안된 이벤트이다.

## 3.3 이체

* transfer(address recipient, uint amount) external returns (bool) recipient로 토큰amount 잔고를 전송한다. 앞서 설명한 approve는 허용만 하지만, transfer는 실제 잔고의 차감이 발생한다. 따라서 자신의 잔고가 금액의 잔고가 정해진 금액 이상 이체되지 않도록 통제해야 한다.
* transferFrom(address sender, address recipient, uint amount) external returns (bool) transfer와 유사한 기능을 수행하지만, sender를 지정한다는 점이 다르다. 따라서 sender는 사전의 허가가 반드시 필요하다.
* event Transfer(address indexed from, address indexed to, uint value) 이체하면 발생하는 이벤트이다

# 4. ERC20

ERC20은 표준을 준수하는 스마트 계약을 직접 작성해도 된다.

안전하고 튼튼한 코드가 이미 존재한다면 이를 사용해도 좋다. 많이 쓰이고 있는 오픈 소스 라이브러리의 하나인 OpenZeppelin은 ERC20을 비롯한 여러 가지 표준 및 기능을 구현하는 Solidity 라이브러리를 제공하고 있다. 여기서는 Openzeppelin에서 제공하고 있는 ERC20을 살펴보자.

프로그래밍으로 ERC20을 제작하려면, 객체 지향 프로그래밍 개념을 이해해야 한다. ERC20을 구현하려면 상위 컨트랙을 필요로 한다. 앞서 설명한 IERC20 인터페이스는 당연히 필요하고, 그 외에도 상위의 Context, IERC20Metadata을 구현하고 있다.

* IERC20Metadata에서는 name, symbol, decimals를 정의한다.
* Context는 msgSender, msgData를 정의한다.

## 4.1 변수

ERC20는 통화이기 때문에 필연적으로 잔고가 있기 마련이다. 잔고와 통화의 명칭과 관련한 멤버 변수가 다음과 같이 선언되어 있다.

* mapping(address => uint256) private \_balances 각 주소의 잔고를 저장한다.
* mapping(address => mapping(address => uint256)) private \_allowances 소유자가 제3자에게 허용한 인출금액을 저장한다. 첫 번째는 소유자의 address, 두 번째는 제3자 address이고 여기에 uint256 유형의 값은 소유자가 제3자에 대해 허용한 인출 금액이다.
* uint256 private - \_totalSupply 총잔고를 저장한다.
* string private \_name DeFi 명칭
* string private \_symbol DeFi 심볼

## 4.2 생성자

ERC20 객체를 생성하려면 생성자가 필요하고, 명칭과 심볼을 정해주어야 한다.

* constructor(string memory name\_, string memory symbol\_)

## 4.3 명칭과 심볼

명칭과 심볼을 읽는 함수가 제공된다. 또한 자릿수를 알아 보는 함수가 있다.

* function name() 명칭을 출력
* function symbol() 심볼을 출력
* function decimals() ether의 wei로 표현하는 자릿수. 잔고 또는 이체하는 실제 금액과는 무관하다.

## 4.4 잔고

잔고 관련 함수인 totalSupply()와 balanceOf() 그리고 자금 이체에 사용되는 transfer()함수가 제공된다. 내부 함수는 식별하기 좋게 \_로 시작하도록 작명하고 있다. 외부에서는 호출될 수 없도록 내부로 제한하는 것이다. 예를 들어, \_mint()는 핵심적인 함수로, 누군가 발권하기 위해 이 함수를 호출하는 것은 생각만 해도 매우 끔찍한 결과를 불러올 수 있다. 내부 함수로 제한하면 그런 외부 호출은 막을 수 있게 된다.

* function totalSupply() 잔고 총액을 출력
* function balanceOf(address account) account의 잔고를 출력
* function transfer(address to, uint256 amount) 내부함수인 \_transfer(owner, to, amount) 호출.
* 내부함수 function \_mint(address account, uint256 amount) amount만큼 발권하여 account에 지급. totalSupply도 증액된다.
* 내부함수 function \_burn(address account, uint256 amount) amount만큼 소각하고 account에서 차감. totalSupply도 차감된다.

## 4.5 허가

제3자가 자금의 이체 권한을 가지도록 하는 함수들이다.

* function allowance(address owner, address spender) 허용금액 \_allowances[owner][spender] 출력
* function approve(address spender, uint256 amount) 내부함수 \_approve(owner, spender, amount) 호출 (아래 설명 참조)
* function increaseAllowance(address spender, uint256 addedValue) 내부함수 \_approve(owner, spender, allowance(owner, spender) + addedValue) 호출 (아래 설명 참조)
* function decreaseAllowance(address spender, uint256 subtractedValue) increaseAllowance의 반대 기능, 내부함수 \_approve(owner, spender, currentAllowance - subtractedValue) 호출 (아래 설명 참조)
* function \_approve(address owner, address spender, uint256 amount) 매핑변수 \_allowances[owner][spender] = amount를 설정, 즉 소유자 owner가 제3자 spender에게 해당 금액의 권한을 부여하는 함수이다.
* function \_spendAllowance(address owner, address spender, uint256 amount) 내부함수 \_approve(owner, spender, currentAllowance - amount)를 호출한다. 즉 허용한도내인지 확인하고 그 한도내에서 매핑변수 \_allowances[owner][spender] = 현재잔고 currentAllowance - 요청잔고 amount를 설정

## 4.6 이체

다른 사람의 토큰을 누군가에게 transfer하는 기능이다.

* function transferFrom(address from, address to, uint256 amount)

그렇다고 해서 다른 사람의 토큰을 마음대로 보내도록 하면 매우 위험한 것이며, 따라서 허용이 되어서는 안된다. 제3자가 나의 주소로부터 잔고를 자신에게 전송하는 것도 가능해지기 때문이다.

따라서 2단계 절차, approve후 transferFrom을 사용해야 한다. 즉 제3자에게 일정 금액에 대해 허가를 한 후, 제 3자는 그 금액 한도내에서의 이체할 수 있다.

* 단계 1: approve(address spender, uint256 \_amount)로 spender를 지정해서 일정량을 이체하도록 허가한다. 그러면 allowance[owner][spender] = \_amount로 지정한다.
* 단계 2: transfer가 가능하다. 물론 정해진 한도 내에서 가능하다. 제 3자, 즉 다른 컨트랙 또는 다른 외부 주소에게 자신의 잔고를 transfer할 수 있는 \_spendAllowance(from, spender, amount), \_transfer(from, to, amount) 등을 사용한다.

A를 소유권자라고 하자. B가 A의 100 토큰을 C로 보내려고 하면, B는 A로부터 허가를 받아야 한다. A는 B에게 일정 한도를 approve(B주소, 100) 한다. A에게서 B로 허용된 한도는 allowance(A주소, B주소)로 확인할 수 있게 된다. 그러고 나서야, 송금 transferFrom(B주소, C주소, 100)을 할 수 있다. 금액을 적게 transferFrom(B주소, C주소, 85)하면, allowance(A주소, B주소)는 15의 잔고가 남아 있다는 것을 알려준다.

* function \_transfer(address from, address to, uint256 amount) 내부에서만 사용하는 함수, from에서 to로 amount 이체

## 5. 디파이 개발

이미 OpenZeppelin의 ERC20이 있으므로, 이를 상속받아서 MyToken을 만들어보자. 코인의 이름만 짓고, 발행하는 것으로 단순하게 만들 수 있다.

[파일명: src/MyToken.sol]  
줄1 // SPDX-License-Identifier: MIT  
줄2 pragma solidity ^0.8.0;  
  
줄3 import "@openzeppelin/contracts/token/ERC20/ERC20.sol";  
  
줄4 contract MyToken is ERC20 {  
줄5 constructor() ERC20("MyToken", "JSL"){ // 토큰의 이름을 MyToken, 심볼(이름의 축약 버전)은 JSL로 지정  
줄6 \_mint(msg.sender, 1000000000000000000000000);  
줄7 //\_mint(msg.sender, 1000000\*decimals());  
줄8 }  
줄9 }

### 줄3 OpenZeppelin 라이브러리

OpenZeppelin에서 표준을 지켜서 작성해 놓은 ERC20.sol 라이브러리를 가져온다. NPM에서 설치하거나 github에서 직접 가져올 수 있다.

#### 방법 1: npm에서 OpenZeppelin 설치

로컬에서 파일을 import 하려면 npm에서 OpenZeppelin을 설치해야 한다.

pjt\_dir> npm install @openzeppelin/contracts  
특정버전을 설치하려면, npm install @openzeppelin/contracts@2.5.1

설치하고 나면, 프로젝트 디렉토리 아래 node\_modules를 가서 확인해보자. 그 아래 디렉토리가 생성되고 라이브러리가 설치되어 있다.

pjt\_dir> dir node\_modules\@openzeppelin  
02/18/2022 05:40 PM <DIR> contracts

@기호는 NPM 패키지 그룹을 동일 명칭으로 사용할 수 있게 해준다. @openzeppelin/이라고 적어주면 된다.

더 알아보기: 자바스크립트의 경로 재지정

자바스크립트에서 @기호는 경로를 재지정하는 module-alias 기능을 의미한다. import할 때 디렉토리가 있는 곳을 상대적으로 나타내면 길어질 수 밖에 없다. @를 사용하면 경로를 그 라이브러리가 있는 시작점으로 재지정하게 된다. 따라서 예를 들면 import MyLib from '../../../my/MyLib' 을 줄여서 import MyLib from '@my/MyLib'로 경로를 재지정할 수 있다.

#### 방법 2: Github에서 OpenZeppelin 가져오기

github에 라이브러리가 배포되어 있으니, 해당 파일을 내려받아서 사용해도 된다.

### 줄3 OpenZeppelin의 import  
  
REMIX에서는 설치한 @openzeppelin의 경로를 그대로 적어주면 된다.  
@openzeppelin 라이브러리를 찾아 import 한다.

import "@openzeppelin/contracts/token/ERC20/ERC20.sol"; 버전을 정해주려면 import "@openzeppelin/contracts@4.2.0/token/ERC20/ERC20.sol";

그러나 로컬에서 컴파일하게 되면 그 파일의 상대적 명칭을 적어 주어야 한다.  
현재 컨트랙 기준으로 contracts/abc.sol 코드 안에 다음과 같이 적어준다.

import "./my/home.sol" --> 이 경로는 contracts/my/home.sol로 변환된다.

### 줄4 ERC20의 상속  
  
앞서 배운 ERC20을 가져와 상속을 적용한다. ERC20은 OpenZeppelin에서 제공되는 라이브러리이고, 앞서 설치하고 있다.  
이 컨트랙을 상속받으면, NFT를 제작하는데 필요한 앞서 설명한 멤버 속성, 멤버 함수를 모두 적용할 수 있게 된다. 직접 코딩해야 하는 분량이 줄어들 뿐만 아니라, 앞서 설명한 바와 같이 NFT 표준을 준수하게 된다.  
  
### 줄5 명칭과 심볼의 지정  
  
앞서 설명한 명칭과 심볼을 constructor() ERC20("MyToken", "JSL") 정한다. 본인만의 이름과 심볼을 나타내는 문자열로 바꿔 지정해도 된다.   
  
### 줄6 잔고의 생성  
  
생성자에서 잔고를 생성하고 있다.  
  
- 줄3 생성자에는 줄4 ```\_mint(msg.sender, 1000)``` 함수를 호출하여 msg.sender의 잔고를 1000 채우고 있다. 금액은 고정으로 설정하고 있고 물론 필요에 따라 늘리거나, 줄일 수 있다.  
- 줄6 ```\_mint()``` 함수를 보자. 줄6 인자의 주소와 금액을 줄7에서 줄2 ```\_totalSupply```, 줄1 ```\_balances```에 입력하고 있다.  
  
즉, 상태변수 totalSupply, balances를 선언하고, 잔고의 증감을 수정하고 있다. 주의해서 보아야 하는 것이, 이들 변수는 private로 지정되어 직접 수정할 수 없다. 대신 내부함수 ```\_mint()```를 이용해서 잔고를 수정한다.

줄1 mapping(address => uint256) private \_balances; 줄2 uint256 private \_totalSupply; 줄3 constructor() public { 줄4 \_mint(msg.sender, 1000); 줄5 } 줄6 function \_mint(address account, uint256 amount) internal virtual { ... 줄7 \_totalSupply += amount; ... 줄8 \_balances[account] += amount; ... 줄9 } ```

## 6. 컴파일

컴파일을 해보자.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/MyToken.sol > src/MyToken.json  
  
Error: Source "@openzeppelin/contracts/token/ERC20/ERC20.sol" not found: File outside of allowed directories.  
 --> src/MyToken.sol:4:1:  
 |  
4 | import "@openzeppelin/contracts/token/ERC20/ERC20.sol";  
 | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

컴파일하고 발생하는 오류를 읽어보자. 현재 디렉토리 밖의 파일을 사용하는 경우에 컴파일 오류가 발생하고 있다. 왜 그럴까? 오류의 마지막 줄을 보면 ERC20.sol 이 파일이 없다고 하는데, 이 파일이 허용되지 않은 경로에 위치하기 때문에 그렇다.

--allow-path로 경로를 지정해 주어도 오류가 발생하고 있다.

pjt\_dir> solc-windows.exe --allow-paths "C:\\Users\\jsl\\Code\\201711111\\node\_modules\\@openzeppelin" --optimize --combined-json abi,bin src/MyToken.sol > src/MyToken.json  
  
Error: Source "@openzeppelin/contracts/token/ERC20/ERC20.sol" not found: File outside of allowed directories.  
 --> src/MyToken.sol:4:1:  
 |  
4 | import "@openzeppelin/contracts/token/ERC20/ERC20.sol";  
 | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

다음과 같이 경로를 재지정해주면 오류가 발생하지 않는다.

pjt\_dir> solc-windows.exe @openzeppelin/="C:\\Users\\jsl\\Code\\201711111\\node\_modules\\@openzeppelin\\" --optimize --combined-json abi,bin src/MyToken.sol > src/MyToken.json

컴파일한 후, MyToken.json 파일을 열어보면 (type src\MyToken.json) 다음과 같이 여러 파일이 포함되어 있다. 내용이 길어서, ...으로 생략하고 있다.

{"contracts":{"C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/token/ERC20/ERC20.sol:ERC20":{"abi":[...],"bin":"..."},  
"C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/token/ERC20/IERC20.sol:IERC20":{"abi":[...],"bin":""},  
"C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol:IERC20Metadata":{"abi":[...],"bin":""},  
"C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/utils/Context.sol:Context":{"abi":[],"bin":""},  
"src/MyToken.sol:MyToken":{"abi":[...],"bin":"..."}},  
"version":"0.8.1+commit.df193b15.Windows.msvc"}

총 5개의 파일이 포함되어 있고, 인터페이스는 바이트코드가 생성이 되지 않고 abi만 생성한다는 것을 알 수 있다.

|  |  |  |  |
| --- | --- | --- | --- |
| 파일 | 구분 | abi | bytecode |
| ERC20 | 구현 컨트랙 | abi 생성 | 바이트 코드 생성 |
| IERC20 | 인터페이스 | abi만 생성 | 없음 |
| IERC20Metadata | 인터페이스 | abi만 생성 | 없음 |
| Context | 추상 컨트랙 | 없음 | 없음 |
| MyToken | 구현 컨트랙 | abi 생성 | bytecode 생성 |

# 7. 배포

복수의 컨트랙이 import되었으므로, 컴파일한 abi, 바이트 코드도 포함된 파일의 수만큼 생성되었다. 순서를 지정하여 contractName[4]를 읽고 있다.

[파일명: src/MyTokenTest.js]  
var \_abiJson = require('./MyToken.json');  
contractName=Object.keys(\_abiJson.contracts);   
console.log("- contract name: ", contractName[4]); //reading src/MyToken.sol:MyToken

pjt\_dir> node src/MyTokenTest.js  
- contract name: src/MyToken.sol:MyToken 컨트랙 명을 맞게 읽어 내고 있다.

실행해 보면 올바르게 컨트랙명을 읽어 오는 것을 알 수가 있다. 주의할 것이 있다. contractName은 4개의 컨트랙, import한 모든 컨트랙의 명칭을 가지고 있다.

* 'C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/token/ERC20/ERC20.sol:ERC20',
* 'C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/token/ERC20/IERC20.sol:IERC20',
* 'C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol:IERC20Metadata',
* 'C://Users//jsl//Code//201711111//node\_modules//@openzeppelin/contracts/utils/Context.sol:Context', 'src/MyToken.sol:MyToken' ]

우리가 필요한 것은 5번째 MyToken이므로, 인덱스로 contractName[4]을 가져온다.

[파일명: src/MyTokenTest1.js]  
var \_abiJson = require('./MyToken.json');  
contractName=Object.keys(\_abiJson.contracts);  
console.log("- contract name: ", contractName[4]);  
  
\_abi=\_abiJson.contracts[contractName[4]].abi  
\_bin=\_abiJson.contracts[contractName[4]].bin  
console.log("- ABI: ", \_abi);  
console.log("- Bytecode: ", \_bin);

abi, 바이트 코드가 잘 읽혀졌는지 다음을 실행해서 확인해보자.

pjt\_dir> node src/MyTokenTest1.js

## Unexpected token o in JSON at position 1 Error

이 오류는 일반적으로 JSON 문자열이 잘못되었거나 유효하지 않은 형식을 가지고 있을 때 발생한다. "Unexpected token o"는 JSON 문자열이 "{" 또는 "["로 시작해야 하는데 "o"로 시작했다는 의미이다.

JSON.parse 함수는 문자열(string)을 인자로 받고, 이를 Javascript 객체(object)로 변환한다. 객체를 전달하면 오류가 발생한다. 반드시 문자열로 변환해서 인자로 넘겨주어야 한다.

node> new Object().toString()  
'[object Object]' --> Object를 생성해서 문자열로 출력한다.  
> JSON.parse(new Object()) 문자열이 아니라 객체를 인자로 전달하고 있다. 오류가 발생하게 된다.  
SyntaxError: Unexpected token o in JSON at position 1  
--> Object를 파싱하면 첫 글자가 기대하는 토큰이 아니고 [object Object]이라서 오류가 발생한다  
> JSON.parse(JSON.stringify(new Object()))  
{} --> 문자열로 변환 stringify해서 출력하면 오류가 없어진다.

이런 코드를 반영해서 아래 프로그램을 작성한다.

[파일명: src/MyTokenDeployAbiBinFromFile.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./MyToken.json');  
  
var web3;  
if (typeof web3 !== 'undefined') {  
 web3 = new Web3(web3.currentProvider);  
} else {  
 web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
}  
  
contractName=Object.keys(\_abiBinJson.contracts); // reading ['src//Timer.sol:Timer']  
console.log("- contract name: ", contractName[4]); //or console.log(contractName);  
  
\_abi=\_abiBinJson.contracts[contractName[4]].abi  
\_bin=\_abiBinJson.contracts[contractName[4]].bin  
  
  
//contractName=Object.keys(\_abiJson.contracts); // reading ['src//Timer.sol:Timer'];  
//console.log("- contract name: ", contractName[4]); //or console.log(contractName);  
//\_abiArray=JSON.parse(\_abiBinJson.contracts[contractName[4]].abi); //JSON parsing needed!!  
//\_bin=\_abiBinJson.contracts[contractName[4]].bin;  
console.log("- ABI: " + \_abi);  
console.log("- Bytecode: " + \_bin);  
  
console.log("- ABI Stringify: " + JSON.stringify(\_abi));  
console.log("- ABI JSON Stringify: " + JSON.parse(JSON.stringify(\_abi)));

실행해서, 올바르게 abi, 바이트 코드가 출력되는지 확인하자.

pjt\_dir> node src/MyTokenDeployAbiBinFromFile.js

자 그러면 배포 프로그램을 완성해보자.

[파일명: src/MyTokenDeployAbiBinFromFile.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./MyToken.json');  
  
var web3;  
if (typeof web3 !== 'undefined') {  
 web3 = new Web3(web3.currentProvider);  
} else {  
 web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
}  
  
contractName=Object.keys(\_abiBinJson.contracts);  
console.log("- contract name: ", contractName[4]); //or console.log(contractName);  
  
//\_abi=\_abiBinJson.contracts[contractName[4]].abi  
//\_bin=\_abiBinJson.contracts[contractName[4]].bin  
  
\_abiArray=JSON.parse(JSON.stringify(\_abiBinJson.contracts[contractName[4]].abi)); //JSON parsing needed!!  
//\_abiArray=JSON.stringify(\_abi)  
\_bin="0x"+\_abiBinJson.contracts[contractName[4]].bin;  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin})  
 .send({from: accounts[0], gas: 1000000}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

실행하면, 해시와 주소가 출력되고 있다.

pjt\_dir> node src/MyTokenDeployAbiBinFromFile.js  
  
- contract name: src/MyToken.sol:MyToken  
Deploying the contract from 0x442c2ca937788E7C76b232f8822B016b8A253282  
hash: 0x4e27c54456e4a58274278be5de7c00511bfd70f7010f4daf78cacc915d432ced  
---> The contract deployed to: 0x8d8A11930c7C25554eD0EE8138E720206437af92

# 8. 사용

앞서 정의한 함수들을 호출하는 프로그램을 작성해보자. 출력을 보고 쉽게 분별할 수 있도록 일련번호를 붙여가면서 출력문을 작성하고 있다.

[파일명: src/MyTokenUse.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./MyToken.json');  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts);   
console.log("- contract name: ", contractName[4]);// reading ['src//Timer.sol:Timer']  
var \_abiArray=JSON.parse(JSON.stringify(\_abiBinJson.contracts[contractName[4]].abi)); //JSON parsing needed!!  
  
var \_test = new web3.eth.Contract(\_abiArray, '0x8d8A11930c7C25554eD0EE8138E720206437af92');  
  
async function doIt() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account: " + accounts[0]);  
 const balanceBefore = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance before: " + balanceBefore);  
   
 const name = await \_test.methods.name().call();  
 console.log("name: " + name);  
 const decimals = await \_test.methods.decimals().call();  
 console.log("decimals: " + decimals);  
 const value = await \_test.methods.totalSupply().call();  
 console.log("totalSupply: " + value);  
  
 console.log("-----1 before transfer-----");  
 console.log("1-1 balanceOfAccount0: " + await \_test.methods.balanceOf(accounts[0]).call());  
 console.log("1-2 balanceOfAccount1: " + await \_test.methods.balanceOf(accounts[1]).call());  
 await \_test.methods.transfer(accounts[1], 1)  
 .send({from: accounts[0], gas: 364124, gasPrice: '1000000000'})  
 console.log("-----2 after transfer-----");  
 console.log("2-1 balanceOfAccount0: " + await \_test.methods.balanceOf(accounts[0]).call());  
 console.log("2-2 balanceOfAccount1: " + await \_test.methods.balanceOf(accounts[1]).call());  
 console.log("-----3 after approve and transferFrom-----");   
 var addrTo = accounts[0];  
 console.log("3-1 allowanceOfAddrTo: "+ await \_test.methods.allowance(accounts[0], addrTo).call());  
 await \_test.methods.approve(addrTo, 111)  
 .send({from: accounts[0], gas: 100000});  
 console.log("3-2 allowanceOfAddrTo 111: "+ await \_test.methods.allowance(accounts[0], addrTo).call());  
 await \_test.methods.transferFrom(addrTo, accounts[1], 1)  
 .send({from: accounts[0], gas: 3641240, gasPrice: '1000000000'})  
 console.log("3-3 balanceOfAccount0: " + await \_test.methods.balanceOf(accounts[0]).call());  
 console.log("3-4 balanceOfAccount1: " + await \_test.methods.balanceOf(accounts[1]).call());  
 //const value = await \_test.methods.myFunction()  
 // .send({from: accounts[0], gas: 364124, gasPrice: '1000000000'})  
 //.then(function(value) {console.log("---> myFunction called " + JSON.stringify(value.events.MyLog.returnValues));});  
 //console.log("---> myFunction called " + JSON.stringify(value.events.MyLog.returnValues));  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));  
}  
doIt()

* 토큰의 소유는 토큰을 배포한 accounts[0]이다. 발권한 금액은 accounts[0]의 잔고로 잡히게 된다.
* call 함수 출력을 위해 await를 해준다. 해주지 않으면 Object가 출력된다.

\_test.methods.balanceOf(accounts[0]) ---> [object Object]  
await \_test.methods.balanceOf(accounts[0]) ---> [object Object]  
await \_test.methods.balanceOf(accounts[0]).call()); ---> 반환값이 올바르게 출력된다.

* 1. transfer하기 전의 잔고

-----before transfer-----  
balanceOfAccount0: 1000000000000000000000000  
balanceOfAccount1: 0

* 1. 1을 transfer한 후 잔고는 서로 1만큼씩 증감이 있다. transfer(address to, uint256 amount)는 to에게 amount를 전송하는 기능이다.
  2. approve하고 나서 1을 transfer한 후 잔고 역시 서로 1만큼씩 증감이 있다. transferFrom()은 2단계가 필요한 송금이다. 이 함수는 from에서 to로 전송하지만 반드시 사전 출금허락이 필요한 기능이다.
  + 따라서 먼저 approve(address owner, address spender, uint256 amount)를 실행하여 어떤 계좌로부터 그 금액 amount만큼 인출할 때 허락을 받고,
  + 그리고 나서 transfer(address from, address to, uint256 amount) 해야 한다.

pjt\_dir> node src/MyTokenUse.js  
  
- contract name: src/MyToken.sol:MyToken  
Account: 0x442c2ca937788E7C76b232f8822B016b8A253282  
Balance before: 999995671865000000000  
name: MyToken  
decimals: 18  
totalSupply: 1000000000000000000000000  
-----1 before transfer-----  
1-1 balanceOfAccount0: 999999999999999999999980  
1-2 balanceOfAccount1: 20  
-----2 after transfer-----  
2-1 balanceOfAccount0: 999999999999999999999979  
2-2 balanceOfAccount1: 21  
-----3 after approve and transferFrom-----  
3-1 allowanceOfAddrTo: 110  
3-2 allowanceOfAddrTo 111: 111  
3-3 balanceOfAccount0: 999999999999999999999978  
3-4 balanceOfAccount1: 22  
Balance after: 999995536446000000000  
Balance diff: 135418999996416

# 8. 나의 토큰을 상장해보자

지금까지 우리의 토큰을 제작하였다. 이렇게 ERC20 토큰을 제작하면 완성된 것일까? 어떠한 화폐를 발행해도 실제 시장에서 쓰이지 않는다면 어떤 효용성이 있을까?

자 우리가 만든 토큰에 교환 가치를 부여해보자. 우리가 제작한 토큰이 실제 사용되려면 **교환 가치**가 있어야 한다.

## 이더리움 메인넷에 토큰을 등록한다.

우선 메인넷에 배포해야 한다. 사적망에 배포해봐야 교환가치는 없다.

메인넷에 배포하려면 제 3자 서비스인 Infura를 통해서 REMIX에서 하는 것이 편하다. 메타마스크(MetaMask) 지갑을 연결하고, 키를 읽어서 쉽게 배포할 수 있다.

## 토큰을 거래소에 상장하자.

ERC20 토큰을 상장하려면 중앙화된 거래소나 탈중앙화 거래소에 등록해야 한다. 상장 과정은 거래소마다 다를 수 있으므로, 각 거래소의 공식 웹사이트나 문서를 참고하는 것이 좋다.

1. 중앙화 거래소 코인거래가 가능한 해외거래소는 코인베이스, 바이낸스 (Binance) 등이 있다. 국내는 빗썸, 업비트 등이 있다. 이들은 중앙화 거래소 (Cex, Centralized Exchange)에는 신청서를 제출한후, 적격성을 심사해야 상장이 된다.
2. 탈중앙화 거래소 중앙화 거래소는 중앙에서 거래가 이루어지는데 반해, 탈중앙화 거래소 DEX(Distrbuted Exchange)는 기존의 암호화폐 거래소와 달리, 분산거래소이다.

국내에서도 많은 코인들이 탈중화화 거래소에 상장하고 있다. 중앙화 거래소와는 달리, 적격성 심사와 같은 절차가 생략되고 아무나 상장할 수 있다.

따라서 상장이라는 단어보다는 등록이라는 단어가 더 적합하다. ERC20 토큰이 등록 (list) 되려면 허가가 필요하지는 않다. 비용도 없고, 중간 매개 없이 거래가 이루어진다.

대표적인 탈중앙화 거래소는 유니스왑(Uniswap)이 있다. 유니스왑은 이더리움 기반의 DEX으로서 ERC20 토큰을 서로 교환할 수 있다. 유니스왑에 등록하려면 동등한 교환가치의 이더가 '쌍으로' 제공되어야 한다.

예를 들어 MyToken이라고 하자. 그러면 10만원의 MyToken과 10만원의 이더가 필요하다. 이를 **유동성 풀(Liquidity Pool)**이라고 한다. 이더와 연동하려면 MyToken/ETH의 풀을 구성한다. 가격은 그 교환비율에 따라 결정한다. 10만원어치 이더에 해당되는 내가 만든 토큰인 MyToken의 개수를 결정하면 된다.

## 거래

ERC20 토큰의 거래를 하려면, ERC20을 지원하는 지갑이 있어야 한다. 많이 쓰는 지갑은 메타마스크가 있다. 또한 Coinbase Wallet, ZenGo, Exodus, Metamask, Trezor, WalletTrust 등을 사용할 수도 있다.

그리고 거래를 하기 위해서는 당연히 수수료 gas를 지급해야 하기 때문에 Ether를 구매해서 넣어 놓아야 한다. Ether를 구매하는 방법은 다양하다. 코인베이스와 같은 암호화폐 거래소를 이용할 수도 있고, 로빈후드(Robinhood) 같은 주식 거래 앱으로도 살 수 있다. 여기서는 메타마스크를 사용해 Ether를 구매해도 된다. Ether를 채우기 위해 '구매' 버튼을 클릭하면 작은 창이 뜨며 '와이어(Wyre)로 Ether구매'나 'Ether직접 예치' 중 선택할 수 있다. 와이어는 블록체인 기반 국제 송금 업체다.

거래는 (1) 급행, (2) 보통, (3) 느림이 있고, 수수료를 많이 내면 빠르게 처리된다.

그 교환비율은 항상 변경된다. MyToken을 누군가 구매하면, 공급이 감소할테고 그러면 ETH의 공급은 증가하게 된다. 따라서 MyToken의 가격은 상승하게 된다. 가격이 변동하려면 거래가 발생해야하고, 그러면 토큰의 가격이 재산정된다.

유니스왑에서는 ETH를 구입하거나 등록된 ERC20 토큰을 구매할 수 있다. 그러려면 수수료를 지급해야 한다.

**차익거래(Arbitrage)**를 통해 수익을 낼 수 있다. 예를 들어, 토큰별로 이더와의 교환 비율에 차이가 있다면, 이를 노리고 수익을 올릴 수 있다.

## 연습문제

1. DeFi는 탈중앙화금융으로 중앙의 중개처가 없어 이러한 제3자의 승인이나 허락없이 직거래한다. OX로 답하시오.
2. DeFi를 제작하려면, 준수해야 하는 이더리움의 표준을 적으시오.
3. STO는 부동산, 미술품 등을 토큰 주식화하기 때문에 정부의 규제와 감시를 받지 않다도 된다. OX로 답하시오.
4. DeFi의 표준을 적용하면, 제 3자는 권리가 부여된다고 해도 소유자의 잔고를 이체할 수 없다. OX로 답하시오.
5. DeFi의 표준에 따라 함수를 구현한다고 하고, 각 계정 주소의 잔고를 저장하는 변수를 선언하는 코드를 작성하시오.
6. account0의 잔고를 출력하는 함수를 작성하시오.
7. 함수 transfer()와 transferFrom()의 차이를 설명하시오.
8. 생성자 constructor에서 하는 작업을 적으시오.
9. ERC20을 메인넷에 배포하거나 사적망에 배포해도 그 가치는 변하지 않는다. OX로 답하시오.
10. 중앙화 거래소 DEX(Distrbuted Exchange)는 기존의 암호화폐 거래소와 달리, 분산 거래소이다. OX로 답하시오.
11. 대표적인 탈중앙화 거래소는 유니스왑(Uniswap)이 있다. 유니스왑은 이더리움 기반의 DEX으로서 ERC20 토큰을 서로 교환할 수 있다. ox로 답하시오.
12. ERC20 토큰을 (1) 판매하고, (2) 인출하는 함수를 구현하시오.