# Chapter 9. Solidity 함수, 이벤트

이 장에서는 Solidity 함수를 어떻게 선언하고, view/pure, 가시성, payable을 설정하는 문법을 배운다.

Solidity의 함수는 다른 언어와 비슷하지만, 몇 가지 주의가 필요하다. 생성자는 객체를 생성하는 역할을 하지만, 특이하게 한 번 호출된 후에는 더 이상 호출되지 않는다.

Solidity 함수의 재미있는 특징 가운데 한 가지는 함수로 제약 조건을 지정해서 함수 실행 전/후에 다른 코드를 실행할 수 있다는 점이다.

이벤트도 배우게 되는데, 다른 언어에서는 이를 쉽게 적용할 수 있게 되어 있다. 그에 반해 블록체인에서는 그렇지 않다. 블록체인이라는 특성이 연결되어 있지 않는 원격이라는 점에서 이벤트의 알림이 다른 언어와 다르다는 것을 알고 배워보자.

또한 fallback이라고 하는 어떤 상황에서 자동호출되는 특이한 함수가 있다.

# 1. 함수의 선언

Solidity 함수를 크게 구분하면 다음과 같다.

* **내장 함수(built-in functions)** 함수는 말 그대로 내장되어 있는, 즉 기본적으로 제공되는 함수들이다. 지금까지 배워왔던 web3.eth.getAccounts() 뿐만 아니라 keccak256(), sha256()과 같이 많은 함수가 제공된다.
* 반면에 **사용자 정의함수(user-defined functions)**는 우리가 직접 정의하고 구현한다.

함수를 선언하는 문법을 보면 다음과 같다.

function 함수명(인자) 수식어 returns(데이터타입) {  
 //함수 바디의 코드를 적는다  
}

사용자 정의함수를 만들 때는, 문법에 맞추어, 앞에 function이라고 적어 함수 선언을 한다. 또한 함수의 입력 인자 및 출력 데이터 타입을 정의해서 넣어주는 것이 필요하다. 가시성 internal, external, public 등과 수행하는 기능의 성격에 따라 pure, view, payable의 수식어를 덧붙인다.

# 2. 생성자

생성자는 조금 특별한 형태의 함수이다. 버전 0.5부터는 그냥 constructor라는 별칭을 그대로 적어주고, 이름이 없다. 그러므로 생성자 constructor()를 통해 컨트랙의 객체가 생성된다.

생성자는 컨트랙의 소유자를 정하거나, 필요한 변수를 설정하고 초기화시킬 때 사용한다. 생성자는 객체를 생성할 때 실행되며, 배포하는 시점에 딱 한 번만 실행이 되고, 배포된 후에는 다시 호출할 수 없다. 다른 시점에 다른 코드에서 객체를 생성해도 생성자가 재실행되지 않는다는 의미이다.

다음은 생성자의 특성들을 설명한 것이다.

* 생성자를 구현하지 않았다면, 기본 생성자가 자동으로 만들어진다. 기본 생성자는 인자가 없는 형태로 구현된다.
* 생성자는 1개만 허용된다. 오버로딩(overloading)은 할 수 없다. 나중에 배우겠지만 상속하는 경우, 부모 컨트랙의 생성자가 먼저 호출되고 그 후 자식의 생성자가 호출된다.
* 생성자는 반환 값이 없다. 따라서 return 문을 사용하지 않는다.
* 생성자는 외부에서 사용할 수 있도록 한다. 버전 0.7부터는 public 키워드가 필요하지 않다. 또는 **internal**로 선언하면, 자신말고는 다른 컨트랙이 생성할 수 없게 된다.

contructor() public { ... } --> 이전 방식  
contructor() { ... } ---> 버전 0.7부터는 public 키워드 없이 적어준다.

# 3. 함수의 수식어

## 3.1 view와 pure

Solidity에서는 함수의 실행 결과로 블록체인이 변경되는지 아닌지를 구분한다. 블록체인의 상태를 변경하지 않는 함수는 view 또는 pure로 표기해야 한다.

view는 이전 constant를 대체한 명령어로서, 상태를 수정하지 않는 경우, 즉 읽기 전용(read only) 함수이다.

pure는 view보다 더 제약이 많은 함수에 붙이는 수식어이다. 함수의 지역변수만을 사용하며, 함수 밖의 그 어떤 변수를 수정하지 못할 뿐만 아니라, 상태 값을 읽지 않는 경우에 사용한다.

|  |  |
| --- | --- |
| 구분 | 설명 |
| view | 상태를 수정하지 않는 경우 |
| pure | 상태를 수정하지도 않으며, 읽지도 않는 경우 |

블록체인을 변경하는 **view, pure가 아닌 경우**에는 transaction이 발생하므로 **Transaction hash**가 반환된다. 즉 처리결과 값을 반환(return) 하지 않는다 점에 주의한다. 따라서 결과 값을 확인하는 방법은 이벤트(event)를 사용해서만 가능하다. 이벤트가 로그에 값을 쓰고, 로그에서 그 값을 읽어서 확인할 수 있다.

pure 함수는 다음과 같은 경우에 사용할 수 없다.

* 상태 변수(state variables)를 읽는 경우,
* 잔고 this.balance 또는 <address>.balance를 읽는 경우,
* 전역변수 block, tx, msg (msg.sig, msg.data 제외)를 읽는 경우,
* pure로 표기되지 않은 함수를 호출하는 경우,
* opcodes로 작성된 인라인(inline) 코드

다음 경우에 해당되는 함수들은 블록체인을 수정하기 때문에 view는 사용할 수 없다.

* 상태 변수를 저장하는 경우 (당연히 블록체인에 쓰는 경우이다)
* 이벤트 발생 (이벤트가 발생하면 로그에 기록이 남겨진다.)
* 다른 컨트랙 생성
* selfdestruct 컨트랙을 삭제하는 경우
* call 함수로 송금하는 경우
* view 또는 pure 아닌 함수를 호출하는 경우
* low-level calls을 호출하는 경우
* opcodes로 작성된 인라인 코드가 있는 경우

## 실습문제: 앞서 만들었던 함수의 view, pure 확인

앞서 작성했던 GlobalVarsTest.sol을 다시 살펴보고 pure 또는 view를 사용한 함수들이 위에서 설명한 것과 일치하는지 확인해본다. 송금하는 컨트랙도 잔고를 구할 경우 pure, view 어떤 수식어를 사용했는지 확인해보자.

## 3.2 Payable

payable 함수는 Ether를 입출금하는 경우에 사용된다. web3 호출하는 측에서 value: 필드에 Ether를 입력하면 전역 변수 msg.value로 전달된다. 컨트랙에서 호출하는 경우는 value() 인자에 금액을 적는다.

## 3.3 가시성

* public 내, 외부에서 누구나 사용할 수 있다. 가장 개방적이다.
* internal 기본(default) 가시성으로 내부 또는 상속관계에서만 사용할 수 있다.
* private 자신만 사용할 수 있다.
* external은 internal의 반대로서 외부에서만 사용할 수 있다. 내부에서 호출할 수 있지만 this 키워드를 이용해서 this.f()형태로 사용한다.

## 3.4 함수를 적어주는 순서

컨트랙 내에 함수를 배치할 때는 다음 순서로 구현한다(참조 Solidity Style Guide). 여기에서 설명되는 함수들은 다시 설명된다. 이해가 안되더라도 일단 계속 읽어나가기로 한다.

* 생성자
* fallback 함수
* external 함수
* public 함수
* internal 함수
* private 함수

## 실습문제: 함수

간단한 컨트랙을 만들어 보자. 이미 함수들을 만들어 보았지만, 관련 문법과 의미를 이해하면서 다양한 함수들을 정의하고 구현해보자.

* 생성자 함수를 만들고, 멤버 변수 x를 초기화한다.
* 멤버 변수 x를 변경, 조회하는 함수를 만들어 보자. 멤버 변수를 읽는 경우 pure인지 view인지 설정하고, 아닌 경우에는 제거한다.
* 함수의 가시성에 따라 내부에서 사용할 경우에는 internal, 누구나 사용할 경우에는 public으로 설정해본다.

REMIX에서 버튼 테스트를 하면서, 다음을 생각해보자.

* 버튼의 색은 3종류이다. (1) 빨간 색은 payable, (2) 주황 색은 non payable이면서 view 또는 pure가 아닌 경우, (3) 하늘색은 view 또는 pure로 구분되어 있다.
* getX\_() view --> getX\_()로 view를 제거하면, 어떻게 되는지 생각해보자. 오류는 발생하지 않는다. 읽기 함수에서 변경 함수로 지위가 변경되면서 버튼의 색이 하늘색으로 변경된다. 더 중요한 것은 gas비가 무료에서 --> 유료로 변경된다. 조회는 당연히 무료이고, 그 외는 gas가 발생한다.
* 또한 internal로 선언하면 버튼이 만들어지는지 생각해보자. 버튼이 만들어질까? 내부용이라서 버튼이 만들어지지 않는다.

[파일명: src/FuctionTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract FunctionTest {  
 int x;  
 constructor() { // 이전 버전에서는 constructor() public {  
 x = 0;  
 }  
 function incrementX() public {  
 x += 1; // 상태변수 x의 변경  
 }  
 function doubleX() public {  
 X2(); //아래에 internal로 정의. 현재 public에서 호출.  
 }  
 function divideBy(int by) view public returns(int) {  
 return x/by; // 소수를 지원하지 않는다. 0, 1/3 등을 테스트해보자.  
 }  
 // actually NONE returned  
 function getX\_() view public returns(int) {  
 return x;   
 }  
 // pure로 설정 테스트해보자  
 function getX() view public returns(int) {  
 return x;  
 }  
 function getBalance() view public returns(uint) {  
 return(address(this).balance);  
 }  
 // 'payable'을 붙이면 코드가 없어도 입금가능 (msg.value가 입금된다)  
 function deposit() public payable {  
 }  
 // 외부에서 사용할 수 없다  
 function X2() internal {  
 x \*= 2;  
 }  
 function getBlockNumber() view public returns(uint) {  
 return block.number;  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 7 ~ 9 constructor() | 생성자는 앞에 위치시킨다. |
| 11 ~ 13 increatementX() | x를 1만큼 증가시킴. 상태 변수를 수정하므로 view, pure가 아님. |
| 14 ~ 16 doubleX() | 내부 함수 internal X2()를 호출해서 x값을 두 배 증가시킴. 상태 변수를 수정하므로 view, pure 아님. |
| 18 ~ 20 divideBy(int by) | 0으로 나누어도 오류를 발생하지 않는다. 아직 소수는 지원하지 않으므로 필요한 경우 라이브러리를 사용한다. |
| 22 ~ 24 getX\_() | view, pure가 아니므로returns(int)해도 반환 값을 받지 못함. 해시값을 돌려줌. |
| 26 ~ 28 getX() | state var를 읽으므로 view를 사용한다. pure를 사용하지 않는다. |
| 29 ~ 31 getBalance() | 잔고 조회는 view도 가능함. |
| 33 ~ 34 deposit() | payable로 선언해야 입금이 가능. 함수에 코드가 없어도 msg.value를 입금. |
| 36 ~ 38 X2() | internal로 선언되어서 내부에서만 사용가능. |
| 39 ~ 41 getBlockNumber() | 전역 변수를 읽을 경우 view |

# 4. 함수의 제약 조건을 지정하는 변경자

변경자 함수는 함수에 **제약 조건**을 덧붙일 경우에 사용되며, 그래서 modifier라고 부른다. 여러 함수에서 같은 조건을 확인한다고 가정해보자. 예를 들어 Owner인 경우, 일정한 기간에 포함되는 경우, 짝수에 해당하는 경우 등 조건을 걸어 그런 경우에만 함수를 실행할 수 있게 한다. 그럼 조건을 확인하는 코드를 모든 함수에 넣어야 한다. 그런데 이렇게 같은 코드를 반복해서 여러 함수에 구현하면, 혹시 그 조건이 수정된다면, 모든 함수의 코드를 다시 수정해야 한다.

변경자는 이렇게 많은 함수가 실행될 때 만족해야 하는 조건을 따로 구현한 것이다. 조건을 확인해야 하는 함수는 같은 코드를 반복해서 작성하는 대신 변경자를 지정하기만 하면 된다. 만약 확인해야 하는 조건이 변경되면, 여러 함수의 코드를 수정하는 대신 변경자만 수정하면 되므로, **코드의 유지 보수가 쉬워진다**. 또한 변경자를 적용하면 코드 안에서 **if문을 이용해서 제한하는 코드를 없애는 효과**가 있게 된다.

변경자 코드를 구현할 때, 밑줄 underscore \_ 명령어를 이용해서 변경자를 지정한 함수가 변경자의 어느 부분에서 실행될 지 지정할 수 있다.

또한 변경자는 보통 함수가 그런 것처럼 **상속**이 가능하다. 또한 오버라이드할 수 있다.

변경자는 일반적인 함수를 만드는 것처럼 작성할 수 있는데, function 대신 modifier라는 키워드를 사용한다. 변경자를 사용하는 함수는 변경자의 이름을 함수의 헤더 부분에 지정할 수 있다. 다음 실습 문제를 통해 변경자 사용법을 익혀보자.

## 실습문제: 밑줄 명령어 적용

봄, 여름, 가을, 겨울을 설정하는 함수를 만들고, 변경자에 따라 포함되는 시점을 전 또는 후로 정할 수 있다. REMIX에서 버튼을 만들어 실행해보자.

* setSpring()을 실행한 후, getSeason()을 호출하면, summer가 출력된다
  + setSpring()은 setSummerBefore를 변경자로 지정하고 있다. setSummerBefore의 \_ 위치에서 setSpring() 함수가 실행되고, season을 "spring"으로 변경한다.
  + 그런 후에 setSummerBefore()의 나머지 코드가 실행되면서 season에 "summer"가 저장되므로, getSeason()이 호출되면 summer가 출력된다.
* setWinter()가 실행되면, setSummerAfter() 변경자가 실행된다.
  + 변경자에서 season변수값은 "summer"가 되었다가 setWinter() 함수의 내부 코드가 실행되며 season 변수가 "winter"로 변경된다.
  + 따라서 getSeason()을 호출하면 winter가 출력된다.

[파일명: src/UnderscoreTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract UnderscoreTest {  
 string season = "";  
 function getSeason() view public returns(string memory) {  
 return season;  
 }  
 function setWinter() public setSummerAfter { // setWinter()를 setSummerAfter의 \_ 위치에서 실행  
 season = "winter";  
 }  
 function setSpring() public setSummerBefore { // setSpring()을 setSummerBefore의 \_ 위치에서 실행  
 season = "spring";  
 }  
 modifier setSummerAfter() {  
 season = "summer";  
 \_;  
 }  
 modifier setSummerBefore() {  
 \_;  
 season = "summer";  
 }  
}

## 실습문제: 은행 BankV3

은행에는 입출금 관련 제한을 걸어 놓는 경우가 있다. 입금시 최소시간, 출금시 최소한도 제한 규정을 구현해보자.

### 단계 1: 컨트랙 개발

앞서 은행 컨트랙에 forwardTo, deposit, withdrawAll 함수를 구현해 놓았으니 활용하기로 하자. 제한을 적용하기 위해 변경자를 추가해서 앞의 은행 코드를 수정한다.

* onlyOwner: 컨트랙 소유자만 가능하도록 제한한다.
* onlyAfter: 입금시 최소 시간이 지나야 가능하도록 제한. 입금을 연달아 할 경우, 반드시 일정 시간이 지난 후에야 입금을 가능하게 하고 있다. onlyAfter() 변경자는 지난 입금 시점 후 10 초로 설정하고 있다. 코드로 쓰면 block.timestamp + 10 seconds이다.
* minBalance: 출금시 최소 잔고가 있어야 가능하도록 minBalance() 함수( modifier)는 일정 잔고 이상인지 제한하고 있다. 이 변경자를 제한 조건으로 withdrawAll() 함수는 최소 잔고를 초과할 경우에만 출금이 되도록 한다.

나중에 더 배우겠지만, fallback() 함수를 넣어보자. 아무런 함수도 호출하지 않고 컨트랙 주소만으로 호출하면 어떤 일이 일어나겠는가? 컨트랙을 주소만으로 함수도 없이 무슨 호출을 한다고? 존재하지 않는 함수를 호출한다는 의미이다. 그러면 보통은 오류가 발생해야 마땅하다. 이 때 fallback()이 호출된다. 여기서 fallback() 함수는 단순히 이벤트 PrintLog("Fallback called")만 발생하도록 작성한다.

REMIX에서 버튼을 눌러서 기능을 테스트 하면서 진행하자. 조회 버튼과 달리, deposit() 함수를 테스트할 경우에, 상단의 'VALUE'와 버튼 옆의 함수 인자 두 필드에 동일한 금액을 넣어준다. 즉 실제 금액과 인자 금액이 일치하도록 하자.

[파일명: src/BankV3.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract BankV3 {  
 address owner;  
 uint balance;  
 uint256 timeToDeposit;  
   
 event PrintLog(string);  
 event Sent(address from, address to, uint amount );  
 constructor() {  
 owner = msg.sender;  
 balance = 0;  
 }  
 fallback() external {  
 emit PrintLog("Fallback called");  
 }  
 function forwardTo(address payable \_receiver) public payable onlyOwner {  
 //require(msg.sender == owner);  
 \_receiver.transfer(msg.value);  
 emit Sent(msg.sender, \_receiver, msg.value);  
 }  
 function getBalance() public view returns(uint, uint) {  
 return (balance, address(this).balance);  
 }  
 function deposit(uint amount) public payable onlyAfter {  
 timeToDeposit = block.timestamp + 10 seconds; //timeToDeposit = now + 10 seconds;  
 require(msg.value == amount);  
 balance += amount;  
 }  
 function withdrawAll() public onlyOwner minBalance {  
 balance -= address(this).balance;  
 //require(msg.sender == owner);  
 //v0.5.0 msg.sender.transfer(address(this).balance);  
 payable(msg.sender).transfer(address(this).balance);   
 }  
 function getTimeToDeposit() view public returns(uint256) { return timeToDeposit; }  
 modifier onlyOwner {  
 require(msg.sender == owner);  
 \_;  
 }  
 modifier onlyAfter {  
 require(block.timestamp >= timeToDeposit); //require(now >= timeToDeposit);  
 \_;  
 }  
 modifier minBalance {  
 require(address(this).balance>101 wei);  
 \_;  
 }  
}

### 단계 2: 컴파일

컴파일 해서, abi와 바이트 코드를 파일에 저장한다.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/BankV3.sol > src/BankV3.json

### 단계 3: 컨트랙 배포

이번에는 async, await로 변경하여 코드를 작성해보자. 배포 함수를 async function deploy()로 선언한다. 그리고 그 안에 await web3.eth.getAccounts()로 계정을 구해서 이를 활용한다. 컨트랙을 생성하기 위해 블록체인에 전송하는 await send() 함수도 비동기적으로 처리하여, 주소를 출력한다.

[파일명: src/BankV3DeployFromFile.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./BankV3.json'); // abi, 바이트 코드가 저장된 파일 읽기  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙 명 ['src/BankV3.sol:BankV3']  
console.log("- contract name: ", contractName);  
\_abi=\_abiBinJson.contracts[contractName].abi;  
\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); // 오류방지 SyntaxError: Unexpected token o in JSON at position 1  
\_bin=\_abiBinJson.contracts[contractName].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: "0x"+\_bin})  
 .send({from: accounts[0], gas: 259210}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

배포 코드에 gas를 수정하면서 어떤 오류가 발생하는지 경험해보자.

* gas를 쓰지 않으면 gas 한도 관련 "Error check your gas limit" 오류가 발생한다.
* gas를 대략 259210 미만을 적으면 gas가 너무 낮다는 "intrinsic gas too low 오류가 발생한다. gas 용량은 .estimageGas() 함수로 추정한 수치이고, 자신의 환경에서와 차이가 있는지 확인해보자.

pjt\_dir> node src/BankV3DeployFromFile.js  
  
- contract name: [ 'src/BankV3.sol:BankV3' ]  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0xd81a2aceb7edfccfade9b2ff7f94fa5928459e0975275a1615232398f7858425  
---> The contract deployed to: 0x0042048e0e97BA996CA18fC7d027379ed786Af7a

### 단계 4: 사용

node에서 대화형식으로 해보자. 설명은 한글로 적었으니, node 문법과 구별해서 해보자. 주의! geth@8445에서 하려면 deposit(), withdrawAll() 함수들은 마이닝이 필요하다.

프로그램에 설정한 다음의 제약조건이 지켜지는지 살펴보면서 실행해보자.

* 10초 이내 저축
* 잔고 101보다 적은데 출금

다음에서 사용하는 bank 객체는 스스로 만들어 보자 (아래 ```BankV3Use.js``` 코드를 참조해서 만들어도 된다)  
  
node> bank.methods.getBalance().call().then(console.log); 잔고 0  
> 0  
> bank.methods.deposit(100).send({from:"0x4D2fF...", value:100}); 입금 100  
(geth@8445는 마이닝이 필요하다 geth> miner.start(1);admin.sleepBlocks(1);miner.stop())  
> bank.methods.getBalance().call().then(console.log); 입금금액 조회 100.  
> 100   
> bank.methods.deposit(100).send({from:"0x4D2fF...", value:111}); 입금 'value'와 '인자'가 서로 다르면 입금 실패  
(geth@8445는 마이닝이 필요하다)  
> bank.methods.getBalance().call().then(console.log); 실패하였으므로 잔고는 계속 100  
> 100  
> bank.methods.deposit(111).send({from:"0x4D2fF...", value:111}); 입금 111  
(geth@8445는 마이닝이 필요하다)  
> bank.methods.getBalance().call().then(console.log); 입금 100+111 = 211  
> 211  
> bank.methods.deposit(111).send({from:"0x4D2fF...", value:111}); (10초 지나서) 입금 111, 마이닝하면 금액 증가함  
(geth@8445는 마이닝이 필요하다)  
> bank.methods.getBalance().call().then(console.log); 입금 100+111+111 = 322  
> 322  
> bank.methods.deposit(111).send({from:"0x4D2fF...", value:111}); 입금 111, 마이닝하면 금액 증가함  
(geth@8445는 마이닝이 필요하다)  
> bank.methods.deposit(111).send({from:"0x4D2fF...", value:111}); 입금 111, 10초 이내이므로 마이닝해도 금액 증가하지 않음  
(geth@8445는 마이닝이 필요하다)  
> bank.methods.getBalance().call().then(console.log); 여러번 입금해도 10초 이내 거래는 실패하고 잔고가 늘지 않음  
> 433

이번에는 묶어서 일괄실행해보자. 블록체인에 send()가 필요한 함수는 비동기적으로 처리하기 위해 await로 처리한다. 비동기적으로 처리하면, 예를 들어 입금 deposit()하고 getBalance()하면 잔고에 입금분만큼 반영이 되어있는 것을 알 수 있다.

[파일명: src/BankV3Use.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./BankV3.json'); //importing a javascript file  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
contractName=Object.keys(\_abiBinJson.contracts); // reading ['src/BankV3.sol:BankV3']  
//console.log("- contract name: ", contractName); //or console.log(contractName[0]);  
\_abiArray=\_abiBinJson.contracts[contractName].abi; //use just as read from file  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
console.log("- ABI: " + \_abiArray);  
var bank = new web3.eth.Contract(\_abiArray,"0x0042048e0e97BA996CA18fC7d027379ed786Af7a");  
//var filter = bank.events.PrintLog(function (error, result) { // 이벤트 알림은 하지 않는다.  
// if (!error)  
// console.log(result);  
//});  
  
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);  
 bank.methods.getBalance().call().then(console.log);  
 await bank.methods.deposit(111).send({from: accounts[0], value:111});  
 bank.methods.getBalance().call().then(console.log);  
 await bank.methods.withdrawAll().send({from: accounts[0]});  
 bank.methods.getBalance().call().then(console.log);  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));   
}  
doIt()

pjt\_dir> node src/BankV3Use.js  
  
- ABI: [object Object],[object Object],[object Object],[object Object],[object Object],[object Object],[object Object],[object Object]  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Balance before: 999978775921999995002 거래 이전의 잔고  
Result { '0': '0', '1': '0' } 잔고반환은 2개를 하고 있다. 검증목적으로 2개가 동일해야 한다.  
Result { '0': '111', '1': '111' } 111을 입금  
Balance after: 999978617241999995002  
Balance diff: 158679999971328 (1)  
Result { '0': '0', '1': '0' } 전액 출금 (101 제약보다 큰 잔고가 있으므로 가능하다)

위 출력 (1)을 보면 balanceBefore와 balanceAfter의 연산이 정확하지 않다. 눈치를 채었는가? 왜 그런지 자바스크립트 콘솔에서 확인해도 오류가 있다.

Javascript > 999978775921999995002-999978617241999995002  
> 158679999971328

자바스크립트는 정수는 253까지인 9007199254740992가 최대 값이다. 그 수를 넘어가게 되면 연산이 올바르게 되지 않는다. 계산기로 해보면, 출력값과 다르다 (158,680,000,000,000).

# 5. 이벤트

이벤트는 프로그램의 실행 도중 발생하며 그 때마다 프로그램은 이를 감지하고 처리할 수 있다. 보통 키보드에서 어떤 키를 누르거나, 마우스로 클릭할 때 발생한다. 또는 파일 생성, 네트워크 연결, 데이터베이스에 추가, 갱신, 삭제하거나 오류가 발생하는 등 시스템 측에서 이벤트가 발생할 수도 있다.

이 중 키보드나 마우스에서 이벤트가 발생하는 원리는 하드웨어적으로 인터럽트를 발생시키고, 이를 운영체제가 감지해서 처리하는 것이다. 블록체인에는 그 특성상 이러한 마우스 클릭이나 키를 누르거나 하는 이벤트가 발생할 수 없다.

블록체인에서는 스마트 계약의 상태 변화나 특정 동작에 대해 이벤트가 발생하고, **로그**에 기록이 된다. 그렇기 때문에 로그 기록을 계속 확인하면서, 특정 함수가 호출되거나 특정 조건이 충족되는 등이 포착되면 이벤트로 인식된다.

이런 관점에서 보면 이벤트는 알림으로 간주해도 된다. 문제는 어떻게 알림을 받는가이다.

계란은 반숙으로 삶는 경우 8분 정도가 걸린다고 한다. 삶는 과정을 중간 중간 확인하지 않고 그 시간이 되면 '띠링~'하고 알림이 울린다. 요리사는 '띠링' 소리에 자신이 하던 작업을 잠시 중단하고, 불을 끄고 계란을 꺼내는 작업을 하면 된다.

누군가 블록체인을 통한 상품 주문을 감지하는 이벤트를 설치해 놓았다고 하자. 클라이언트에서는 로그를 리스닝하다가, 이벤트를 포착하게 된다. 누군가 주문을 하면, 이러한 이벤트 알림과 리스닝을 통해서 '아 주문이 들어왔구나'하고 알게 되는 것이다. 그러면 상품을 준비해 배송하는 등의 다음 작업으로 연결한다.

## 5.1 이벤트 발생하기

### 단계 1. 이벤트 설정

컨트랙의 멤버변수 자리에 이벤트를 설정한다. 설정은 매우 단순하다. 명령어 event 이벤트명을 아래에 보인 예와 같이 적어준다. 다음 명령은 PrintLog 이벤트를 만드는 것이다.

event PrintLog()

이벤트명은 낙타 표기법으로 작성하되, 단어의 첫 글자를 대문자로 쓴다. 이벤트에는 매개변수를 넣을 수 있다.

### 단계 2. 함수를 호출할 때 이벤트가 발생하도록 연결한다.

함수 코드를 작성할 때, 이벤트가 발생하게 되는 곳에 emit 이벤트명이라고 적어준다. 그렇게 되면 함수의 코드가 실행되다가 **emit 명령어를 만나게 되면 이벤트가 발생**하게 되는데 이런 연결을 **바인딩(binding)**이라고 한다.

function fireEvent() {  
 emit PrintLog() // 이 시점에 이벤트가 발생한다. 즉 로그에 PrintLog()가 적힌다.  
 ...  
}

### 단계 3. 이벤트 발생을 콜백(callback) 함수로 리스닝한다.

이벤트가 발생했는지 알아보는 것을 리스닝(listening)한다고 한다. 주의하자. 이벤트는 블록체인에서 발생하도록 설계되어 있고, 클라이언트에서는 이를 리스닝을 한다.

클라이언트에서 리스닝하는 것은 events.이벤트함수 이런 식으로 적어주어야 한다. 즉, events.PrintLog() 이런 식으로 리스닝하게 된다.

이벤트 함수에 매개변수가 있어도 생략할 수 있다. 이벤트의 발생을 리스닝하는 기간을 정할 수 있다. fromBlock의 시작부터 toBlock의 종료시점까지 특정 블록을 정해서 리스닝할 수 있다.

1. 클라이언트, 즉 노드에서 이벤트를 생성하고 이벤트 발생을 리스닝한다.

var myEvent = myInstance.events.PrintLog({from: web3.eth.accounts[0]}, {  
 fromBlock: 0, //시작 블록 수  
 toBlock: 'latest' //끝 블록 수  
 });  
> myEvent.watch(function (error, result) {  
 if (!error) {  
 console.log("Event triggered ===> ",result);  
 process.exit(1);  
 }  
});

1. 또는 한 명령어로 합쳐서 다음과 같이 이벤트 발생을 확인할 수 있다.

var myEvent = myInstance.events.PrintLog({fromBlock: 0}, fromfunction(error, event) {  
 if (!error)  
 console.log(event);  
});

### 단계 4. 함수호출로 이벤트 발생.

이벤트는 당연히 발생해야 알 수 있다. 이벤트가 설정된 함수가 호출되면 이벤트가 발생하는 것이다. 지금까지 우리가 해왔던 것과 다를 것이 없다. 보통 컨트랙의 함수를 호출하는 식으로 \_instance.MyFunction() 이렇게 하면 된다.

myInstance.fireEvent(); // 이벤트를 발생시킴

### 단계 5. 이벤트 발생 확인(watch) 중지

이벤트를 리스닝하려면 컴퓨팅 자원이 소모된다. 프로세스가 지속적으로 리스닝해야 하기 때문이다. 리스닝을 중단하려면 다음과 같이 한다.

myEvent.stopWatching();

생각해 보면, 대화식에만 중단이 의미가 있고, 일괄 실행하는 경우에는 그렇지 않다. 일괄실행은 한 프로그램에 이벤트의 리스닝과 중단 명령어가 모두 포함되어 있으니, 리스닝하다 곧 이어 중단하는 명령이 실행되어 본래의 의도가 무색하게 된다.

### event 인덱싱(indexing)

매개변수를 인덱싱한다는 것은, 빠른 검색을 위해 색인을 만들어 놓는다는 의미이다. 이벤트에서 사용하는 매개변수 3 개까지는 로그에서 인덱스로 사용할 수 있고(매개 변수를 지정할 때 indexed라고 붙인다) 매개변수의 해시 값이 로그의 topic으로 출력된다.

이렇게 인덱싱된 매개변수들은 블록체인에서 이벤트를 검색할 때 색인 역할을 할 수 있어서, 로그를 다 검색하지 않아도 되기 때문에 빠르게 검색하는 것이 가능하다.

예를 들어 다음의 계좌이체 Transfer 이벤트에 \_from과 \_to에 indexed가 붙었다고 가정해보자.

event Transfer(address indexed \_from, address indexed \_to, uint256 \_value)

이렇게 되면 나중에 블록체인에서 이벤트 정보를 찾을 때 전체 정보를 검색하는 것이 아니라, 송신 주소(\_from)과 수신 주소(\_to)로만 제한시켜 검색할 수 있다.

### event 오버로딩(overloading)

이벤트는 **상속**할 수 있어서, 부모가 정의한 이벤트를 자식이 물려받아 발생시킬 수 있다. 상속은 나중에 배우게 된다. 여기서는 컨트랙가 여러 이벤트를 가질 수 있는지, 즉 이벤트의 **오버로딩**이 가능한지 실습문제에서 해보게 된다 (참조: '실습문제: 이벤트 오버로딩')

## 5.2 웹소켓

### 연결과 해제

이벤트가 발생한 것을 포착하려면 웹소켓에 연결해야 한다. 연결하기 위해서는 먼저 provider 객체를 만들고, 이를 Web3(provider 객체) 생성자에 매개변수로 넘겨주어야 한다. 이 때 ws:// 프로토콜을 사용한다.

node에서 다음과 같이 웹소켓을 연결하고, 해제할 수 있다.

node> var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345")); 웹소켓에 연결 개시  
undefined  
> web3.currentProvider.connected 연결상태 true 출력  
true  
> web3.currentProvider.connection.close() 웹소켓을 해제  
undefined  
> web3.currentProvider.connected 연결상태 false 출력  
false

### 설정

웹소켓의 연결을 설정할 수 있다. 연결 상태를 on() 함수를 이용해서 알아볼 수 있다.

node> var myprovider = new Web3.providers.WebsocketProvider("ws://localhost:8345", {  
 clientConfig: {  
 keepalive:true, keepaliveInterval:10000  
 }   
 });  
> myprovider.on('connect', function() { console.log("--> Websocket connected"); });  
undefined  
> myprovider.on('close', function() { console.log("--> Websocket closed"); });  
undefined  
> myprovider.on('end', function() { console.log("--> Websocket closed"); });  
undefined  
> myprovider.on('error', function(error) { console.error(error); });  
undefined  
> var web3 = new Web3(myprovider);

## 실습문제: 웹소켓 연결과 해제

웹소켓에 연결해보자. 연결한 후 상태를 출력하고, 즉시 해제한다. 해제하면서 어떤 상태가 출력이 되는지 확인해보자.

[파일명: src/webSocketTest.js]  
var Web3 = require('web3');  
const myProvider = new Web3.providers.WebsocketProvider("ws://localhost:8345", {  
 clientConfig: {  
 keepalive:true, keepaliveInterval:10000  
 }   
 });  
var web3 = new Web3(myProvider);  
console.log("(1) websocket url: ", myProvider.connection.url);  
myProvider.on('connect', function() {  
 console.log("(2) connecting websocket: "+web3.currentProvider.connected);  
 //myProvider.disconnect();  
 web3.currentProvider.connection.close();  
 console.log("(3) disconnecting Websocket: "+web3.currentProvider.connected);  
});  
myProvider.on('close', function() { console.log("--> Websocket closed"); });  
myProvider.on('end', function() { console.log("--> Websocket ended"); });  
myProvider.on('error', function(error) { console.error(error); });

위 프로그램을 실행해보자. (1) 웹소켓으로 연결된 url을 출력하고, (2) 웹소켓의 연결 여부를 확인하고, (3) 연결을 끊고 확인하고 있다. 우측의 한글은 출력에 대한 설명이다.

pjt\_dir> node src/webSocketTest.js  
  
(1) websocket url: ws://localhost:8345 현재 연결된 url을 출력한다.  
(2) connecting websocket: true 연결되면, 그 시점에 연결상태를 true로 출력한다.  
(3) disconnecting Websocket: false 연결된 후 연결을 바로 끊고, 연결상태를 출력하면 당연히 false  
--> Websocket ended 연결을 끊으면 'end' 시점으로 인식해서 출력하고 있다.

## 실습문제: 간단한 이벤트 발생

이벤트를 만들고 함수가 호출되는 시점에 발생하도록 해보자. 이벤트는 발생하는 시점에 "Hello World!" 문자열이 출력되도록 한다. 이벤트가 발생하는지는 클라이언트에서 리스닝해서 알아 낸다. 이벤트가 발생하면 로그에 기록이 되고, 리스닝한다는 것은 이런 로그의 기록에 이벤트가 발생했는지를 확인하는 것이다.

### 단계 1: 컨트랙 개발

이벤트 MyLog를 설정하고, myFunction이 호출될 때 그 이벤트가 발생하도록 한다. 매우 단순한 컨트랙으로 몇 줄이 되지 않지만, 이벤트의 작동과정을 이해하도록 하자. REMIX에서 함수 테스트까지 해보기로 한다.

[파일명: src/EventTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract EventTest {  
 event MyLog(string my);  
 function myFunction() public {  
 emit MyLog("Hello World!");  
 }  
}

### 단계 2: 컴파일

컴파일해서 abi, 바이트 코드를 EventTest.jon 파일로 내보내서 저장하자.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/EventTest.sol > src/EventTest.json

### 단계 3: 컨트랙 배포

ganache@8345로 설정하고 배포하고 있으며, 코드는 앞서의 것과 큰 차이가 없다.

[파일명: src/EventTestDeployFromFile.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./EventTest.json'); // 파일 읽어오기  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙 명 ['src/EventTest.sol:EventTest']  
console.log("- contract name: ", contractName);  
\_abi=\_abiBinJson.contracts[contractName].abi;  
\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); // 오류가 발생하면 SyntaxError: Unexpected token o in JSON at position 1  
\_bin=\_abiBinJson.contracts[contractName].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: "0x"+\_bin})  
 .send({from: accounts[0], gas: 259210}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

프로그램을 실행하여, 컨트랙 배포 주소를 출력하고 있다.

pjt\_dir> node src/EventTestDeployFromFile.js  
  
- contract name: [ 'src/EventTest.sol:EventTest' ]  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0xb7095ec9d332c32b6f456692746f8811d1da25deabf5cad5346d47d8592c5bb1  
---> The contract deployed to: 0x8911bA097c812Bf0B3ff22F90eaf2A905112C5a6

사설망 geth@8445에 배포하기 위해서는 잊지 말자. 계정의 암호 풀어야 하고, 마이닝을 해야 한다.

위에서 주어진 transactionHash를 가지고 처리결과를 확인하는 것이 좋다. 출력하고 나면, 컨트랙주소, gas 사용량, 거래 hash도 찾아보도록 하자.

geth> eth.getTransactionReceipt("해시번호를 여기에 적어준다")

### 단계 4: 사용

ganache@8345에서 배포하였으므로, 이를 유지하여 컨트랙을 호출해보자. 이번 프로그램에서는 이벤트를 리스닝하고 발생하는지 확인하려고 있다는 것을 기억하는가.

* 이벤트는 events.MyLog() 코드 부분에서 리스닝하고 있다.
* methods.myFunction()는 그 함수 내에 있는 이벤트를 격발(trigger)하게 된다.

그러나 이벤트는 발생하지 않게 될 것이다. 왜 그럴까? 연결을 http 방식으로 만들어 놓으니, 이벤트를 지속적으로 리스닝하지 못하고, 발생도 되지 않는 것이다.

[파일명: src/EventTestHttpNoEventFiredUse.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./EventTest.json'); //파일에서 읽기  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345")); // (1)  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙 명 ['src/EventTest.sol:EventTest']  
\_abiArray=\_abiBinJson.contracts[contractName].abi; // stringify해서 문자열로 변환하지 않아도 가능  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); //JSON parsing needed!!  
var \_test = new web3.eth.Contract(\_abiArray,"0x8911bA097c812Bf0B3ff22F90eaf2A905112C5a6");  
var event = \_test.events.MyLog({fromBlock: 0}, function (error, result) {  
 if (!error) {  
 console.log("Event fired: " + JSON.stringify(result.returnValues));  
 }  
});  
  
async function doIt() {  
 const accounts = await web3.eth.getAccounts();  
 const value = await \_test.methods.myFunction()  
 .send({from: accounts[0], gas: 364124, gasPrice: '1000000000'})  
 console.log("---> myFunction called " + JSON.stringify(value.events.MyLog.returnValues));  
}  
doIt()

예상대로 HttpProvider는 Event를 호출하지 못하고 있다. 함수 호출하면서 발생한 로그를 value.events.MyLog.returnValues를 통해서 우회적으로 출력하고 있다.

pjt\_dir> node src/EventTestHttpNoEventFiredUse.js  
  
- ABI: [object Object],[object Object]  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Balance before: 999978310622999995002  
---> myFunction called {"0":"Hello World!","my":"Hello World!"}  
Balance after: 999978287795999995002  
Balance diff: 22827000004608

#### WebSocketProvider

이번에는 웹소켓을 사용하는 방법은 어렵지 않다. 위 프로그램의 (1)의 HttpProvider --> WebsocketProvider으로 바꾸어 보자.

//var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
//var web3 = new Web3(new Web3.providers.WebsocketProvider("http://localhost:8345"));  
var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345")); 이렇게 설정한다.

그리고 geth@8345 사설망을 사용하는 경우, WebSocket이 가능하도록 설정해야 한다. 여기서는 ganache@8345를 사용하므로 변경하지 않아도 된다.

#### 파일에 로그 쓰기

이벤트가 발생했을 때 후속 작업이 따라올 수 있다. 그렇다면 계속 이벤트를 리스닝하고 그 발생을 포착해야 한다. 잠깐 멈추어 생각해보자. 이벤트는 블록체인에서 발생한다. 그러나 이벤트가 발생하는지는 리스닝은 로컬에서 하게 된다. 이러한 원격 리스닝, 즉 발생은 원격 블록체인에서 하지만, 리스닝은 로컬에서 한다는 것은 상당히 이례적이다.

블록체인에서의 이벤트의 발생이 있다면, 로컬에서 뭔가 뒤따라 후속작업이 필요할 수 있다. 블록체인에서 주문을 했다고 하자, 로컬에서 주문 상품의 배송이 후속으로 뒤따를 수 있다. 이런 시나리오는 어떤가? 블록체인에서 렌트카를 했다고 하자, 차를 운전하기 위해 차문을 개방하는 것은 로컬에서 해야 한다.

여기서는 이벤트가 발생하면, 그 내용을 파일에 써보는 것으로 후속작업을 해보자. 응용하면 필요한 다른 프로세스를 호출할 수 있다. fs.writeFile(파일명, 데이터, 인코딩방식, callback함수)함수를 사용하여 이벤트 발생 로그를 로컬 파일에 써보자. fs.appendFile()은 파일에 추가하는 기능이다.

[파일명: src/EventTestWsUse.js]  
var Web3=require('web3');  
var fs = require('fs');  
var \_abiBinJson = require('./EventTest.json'); // 파일 읽기  
  
var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345"));  
contractName=Object.keys(\_abiBinJson.contracts); // reading ['src/EventTest.sol:EventTest']  
//console.log("- contract name: ", contractName); //or console.log(contractName[0]);  
\_abiArray=\_abiBinJson.contracts[contractName].abi; // 파일에서 abi를 바로 읽어낸다.  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi);  
  
console.log("- ABI: " + \_abiArray);  
  
async function doIt() {  
 var \_test = new web3.eth.Contract(\_abiArray, '0x8911bA097c812Bf0B3ff22F90eaf2A905112C5a6');  
 var event = \_test.events.MyLog({fromBlock: 0}, function (error, result) {  
 if (!error) {  
 log = JSON.stringify(result.returnValues);  
 console.log("Event fired: " + log);  
 //fs.writeFile("src/EventTestLog.txt", log, "utf-8", function(e) {  
 fs.appendFile("src/EventTestLog.txt", log, "utf-8", function(e) {  
 if(!e) {  
 console.log(">> Writing to file");  
 }  
 });  
 }  
 });  
 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 value = await \_test.methods.myFunction()  
 .send({from: accounts[0], gas: 364124, gasPrice: '1000000000'})  
 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));  
 //process.exit(1); //강제로 웹소켓을 종료한다  
}  
  
doIt()

네트워크 IP 주소가 올바르지 않으면 Error: connection not open 오류가 발생한다. 자, 이번에는 이벤트가 발생하는가? 아래 출력에서 Event fired: {"0":"Hello World!","my":"Hello World!"}가 출력되고 있다.

pjt\_dir> node src/EventTestWsUse.js  
  
  
- ABI: [object Object],[object Object]  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Event fired: {"0":"Hello World!","my":"Hello World!"}  
Balance before: 999978082352999995002  
>> Writing to file  
Event fired: {"0":"Hello World!","my":"Hello World!"}  
>> Writing to file  
---> myFunction called {"0":"Hello World!","my":"Hello World!"}  
Balance after: 999978059525999995002  
Balance diff: 22827000004608

이벤트 발생하면서 파일에는 아래에 보인 것처럼 JSON 형식으로 작성된다. 최초 블록부터 읽기로 설정하였으므로, 아래와 같이 복수의 출력문이 저장되어 있다.

pjt\_dir> type src\EventTestLog.txt  
  
{"0":"Hello World!","my":"Hello World!"}{"0":"Hello World!","my":"Hello World!"}{"0":"Hello World!","my":"Hello World!"}{"0":"Hello World!","my":"Hello World!"}{"0":"Hello World!","my":"Hello World!"}

## 실습문제: 이벤트 오버로딩

블록체인에서의 이벤트는 반환이 없어서 단순하다. 이벤트는 반환은 신경쓰지 않고, 함수명은 동일하지만, 인자가 다르게 해서 오버로딩할 수 있다.

### 단계 1: 컨트랙 개발

아래 프로그램에서 보듯이 OrderLog() 이벤트를 인자만 다르게 설정하고 있다. order() 함수에서 이벤트를 호출하면서, 명칭이 같아도 문제가 되는지 확인해보자. string, bytes, uint, address 데이터 타입으로 변화를 주고, indexed도 적용하고 있다.

REMIX에서 버튼 테스트하면서 이벤트가 로그에 작성이 되는지 하나 하나 확인하자. 변수 \_itemId는 bytes2으로 선언되어있다. 값을 입력할 경우 16진수라서 4자리를 넣는다 (예: "0x1234")

[파일명: src/OrderEvent.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract OrderEvent {  
 uint unitPrice = 10;  
 event OrderLog(string);  
 event OrderLog(bytes2 \_itemId, uint \_value);  
 event OrderLog(uint256 timestamp);  
 event OrderLog(address indexed \_from, bytes2 \_itemId, uint indexed \_value);  
  
 function order(bytes2 \_itemId, uint quantity) public payable {  
 //uint256 orderTime = now;  
 uint256 orderTime = block.timestamp;  
 uint256 orderAmount = quantity \* unitPrice;  
 require(msg.value == orderAmount);  
 emit OrderLog("Ordered");  
 emit OrderLog(orderTime);  
 emit OrderLog(msg.sender, \_itemId, msg.value);  
 }  
}

### 단계 2: 컴파일

아래와 같이 컴파일을 해서 abi, 바이트 코드를 파일에 저장하자.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/OrderEvent.sol > src/OrderEvent.json

### 단계 3: 컨트랙 배포

앞서 작성된 OrderEvent.json 파일에서 abi, bin을 읽어와서 ganache@8345에 배포하고 있다. 나머지 코드는 앞서 여러번 작성하고 있어서 중복되는 느낌이 있다.

[파일명: src/OrderEventDeploy.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./OrderEvent.json'); // 파일에서 읽기  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙명 ['src/OrderEvent.sol:OrderEvent']  
console.log("- contract name: ", contractName);  
\_abiArray=\_abiBinJson.contracts[contractName].abi;  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); //JSON parsing needed!! //SyntaxError: Unexpected token o in JSON at position 1  
\_bin=\_abiBinJson.contracts[contractName].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: "0x"+\_bin})  
 .send({from: accounts[0], gas: 259210}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

위 프로그램을 노드에서 실행하여 배포하고 나면 컨트랙 주소를 획득할 수 있다.

pjt\_dir> node src/OrderEventDeploy.js  
  
- contract name: [ 'src/OrderEvent.sol:OrderEvent' ]  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0xddf0be098411f1e36a9906c953a240eca4f141634853bf15fc5b7d6e1c6a532d  
---> The contract deployed to: 0x3E04292870AA4Ef2bc44A1638B19A50BCD99b04D

사설망(geth@8445)에 배포하려면, 계정을 해제하는 것과 더불어 마이닝을 잊지말자.

geth> personal.unlockAccount(eth.accounts[0]);  
Unlock account 0x21c704354d07f804bab01894e8b4eb4e0eba7451  
Passphrase:

### 단계 4: 사용

설정한 이벤트 오버로딩을 호출해보자.

[파일명: src/OrderEventUse.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./OrderEvent.json'); // 파일 읽기  
  
//var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345"));  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙명 ['src/OrderEvent.sol:OrderEvent']  
\_abiArray=\_abiBinJson.contracts[contractName].abi; //use just as read from file  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); //JSON parsing needed!!  
//\_bin=\_abiBinJson.contracts[contractName].bin;  
console.log("- ABI: " + \_abiArray);  
  
async function doIt() {  
 var \_order = new web3.eth.Contract(\_abiArray, '0x3E04292870AA4Ef2bc44A1638B19A50BCD99b04D');  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account: " + accounts[0]);  
 var event = \_order.events.OrderLog({  
 //filter: {\_from: accounts[0], \_value: [20,50]},  
 filter: {\_from: accounts[0], \_value: 30},  
 fromBlock: 'latest', toBlock: 'pending'  
 }, function (error, result) {  
 if (!error) {  
 console.log("Event fired: " + JSON.stringify(result.returnValues));  
 //process.exit(1);  
 }  
 });  
 var value;  
 const balanceBefore = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance before: " + balanceBefore);  
 // (1) 이벤트가 발생하고 value: 30은 출력한다.  
 my = await \_order.methods.order("0x1234", 3)  
 .send({from: accounts[0], gas: 100000, value:30})  
 console.log("---> MyFunction called " + JSON.stringify(my.events.OrderLog.returnValues));  
 // (2) 이벤트가 발생하지만, value: 40이라서 걸러지게 된다  
 my = await \_order.methods.order("0x1234", 4).send({from: accounts[0], gas: 100000, value:40});  
 console.log("---> MyFunction called " + JSON.stringify(my.events.OrderLog.returnValues));  
 // (3) 이벤트가 발생하지만, value: 100이라서 걸러지게 된다  
 my = await \_order.methods.order("0x1234", 10).send({from: accounts[0], gas: 100000, value:100});  
 console.log("---> MyFunction called " + JSON.stringify(my.events.OrderLog.returnValues));  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));   
 //process.exit(1); //강제로 웹소켓을 종료한다  
}  
  
doIt()

코드에 대한 설명이다. 이벤트의 설정과 발생 관련한 코드는 주의해서 배우도록 하자.

|  |  |
| --- | --- |
| 라인 | 설명 |
| 4 | WebsocketProvider으로 설정한다. |
| 19 | filter: {\_from: accounts[0], \_value: 30} 인덱싱 걸어놓은 이벤트에 대해 해당 조건에 맞는 경우만 출력한다. 여기서는 30으로 설정하고 있어, 예를 들어 40 또는 100은 출력하지 않는다. |
| 20 | fromBlock: 'latest', toBlock: 'pending' 최근부터 대기하는 이벤트를 출력한다. fromBlock의 default는 latest이다. 따라서 생략해도 된다. fromBlock: 0, toBlock: 'latest'는 처음 블록부터 발생한 이벤트를 모두 출력한다. (1) Latest는 체인에 이미 기록된 가장 최근의 블록을 말한다. 당연히 성공적으로 실행된 것만 기록이 된다. (2) Pending은 아직 대기 중인 거래 블록을 말한다. 실패하면 탈락이되고 블록에 포함되지 않을 수 있다. |
| 31 | 이벤트는 order()함수가 호출되면 실행된다. 주문은 블록체인을 변경하는 함수이므로 비동기적으로 실행하기 위해 await를 붙여준다. Order에 넘겨주는 매개변수는 바이트는 2바이트를 채워서 "0x1234", 주문량은 정수 3으로 맞추어 준다.출력을 보면, 이벤트 인자를 확인할 수 있다 (예: result.args.\_from은 주소) |
| 34 ~ 39 | 아래 이벤트 발생 (2) value:40과 (3) value:100은 인덱싱에 해당되지 않는다 (인덱싱은 30으로 걸어 놓았다. my.events.OrderLog.returnValues 는 출력할 내용이 없어 undefined로 출력하고 있다. |

node 실행을 해보자.

pjt\_dir> node src/OrderEventUse.js  
  
- ABI: [object Object],[object Object],[object Object],[object Object],[object Object]  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Balance before: 999977145423999994462  
Event fired: {"0":"0x9357f478d86D9222f4413bFd91C8adb0F4c728b7","1":"0x1234","2":"30","\_from":"0x9357f478d86D9222f4413bFd91C8adb0F4c728b7","\_itemId":"0x1234","\_value":"30"}  
---> MyFunction called undefined  
---> MyFunction called undefined  
---> MyFunction called undefined  
Balance after: 999976974723999994292  
Balance diff: 170700000067584

## 실습문제: Multiply7 연산결과를 확인하려면 이벤트를 통해서 한다.

앞서 Multiply7을 구현해 보았다. 간단한 연산이지만, 연산결과를 어떻게 알아볼 수 있을까? 보통 프로그램에서는 return 명령어로 결과를 돌려주면 쉽게 된다. 블록체인에서는 반환을 하지 않을 수 있다는 점에 주의해야 한다. 단순한 조회인 경우에는 가능하지만, 블록체인을 수정하는 함수는 반환을 할 수 없다.

블록체인에서 단순연산이라 하더라도 그 결과를 클라이언트에서 알고자 한다면, (1) 멤버변수에 저장한 후 이를 읽어오는 함수를 작성하거나, (2) 로그에 결과 값을 쓰거나 하면 된다. 로그에 쓴다는 것은 이벤트의 매개변수에 결과값을 입력하고, 이를 로그에 작성한다는 말이다. 아래 주어진 미완성 코드를 구현해서 노드에서 이벤트를 띄워보자.

* 함수는 param4인자를 받아서 곱하기 7 연산을 한다. 8과 16의 연산을 실행하자. 그 결과는 56과 112가 출력될 것이다.
* 이벤트로 넘겨주는 매개변수 param1은 누가 이 함수를 호출했는지 (address), param2는 언제 호출되었는지 (timestamp), 마지막으로 param3는 연산결과를 출력하도록 한다.

contract Multiply7Event {  
 event Print(param1, param2, param3);  
 function multiply(param4);  
}

### 단계 1: 컨트랙 개발

multiply() 함수는 pure로 선언되어야 할 것 같지만, event가 발생한다면 pure 또는 view가 될 수가 없다. 따라서 값을 반환하는 것이 의미가 없고, function multiply(uint input) public로 선언하는 것이 바람직하다.

미완성 코드를 완성해 보자. 거듭 설명하는 것처럼, REMIX에서 코딩하고 버튼 테스트를 실행해보자. 이 경우 우측 아래 콘솔에서 이벤트가 발생하고 로그에 기록지는지 쉽게 확인할 수 있다.

[파일명: src/Multiply7Event.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Multiply7Event {  
 event Print(address \_addr, uint256 timestamp, uint res);  
 //function multiply(uint input) public returns(uint) {  
 function multiply(uint input) public {  
 uint res = input \* 7;  
 emit Print(msg.sender, block.timestamp, res);  
 //return res; // 다른 언어에서는 이렇게 반환하지만, 블록체인에서는 아니다.  
 }  
}

### 단계 2: 컴파일

컴파일 하고, abi와 바이트 코드를 파일에 작성해보자.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/Multiply7Event.sol > src/Multiply7Event.json

### 단계 3: 컨트랙 배포

배포에 필요한 코드는 앞과 크게 다르지 않다. 파일에서 abi, 바이트 코드를 가져와 ganache@8345에 배포해보자.

[파일명: src/Multiply7EventDeploy.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./Multiply7Event.json'); // 파일 읽기  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙 명 ['src/Multiply7EventDeploy.sol:Multiply7EventDeploy']  
console.log("- contract name: ", contractName);  
\_abiArray=\_abiBinJson.contracts[contractName].abi;  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); //JSON parsing needed!! //SyntaxError: Unexpected token o in JSON at position 1  
\_bin=\_abiBinJson.contracts[contractName].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: "0x"+\_bin})  
 .send({from: accounts[0], gas: 259210}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

프로그램을 실행하면 배포 주소를 출력하고 있다.

pjt\_dir> node src/Multiply7EventDeploy.js  
  
- contract name: [ 'src/Multiply7Event.sol:Multiply7Event' ]  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x0999eeb551e76a473873d2880a1f6c8088a42d3f30e2c439ac1b1d13cb78f35b  
---> The contract deployed to: 0x33add2effA3E32050aCD8446d826b0EFFB93A515

### 단계 4: 사용

컨트랙의 객체 m7을 생성하고, 이를 통해 m7.events.Print() 명령문으로 이벤트를 리스닝하고 있다. 그 함수의 콜백함수를 잘 보자. 이벤트가 발생하면 매개변수 result를 통해 result.returnValues를 출력하고 있다.

m7.events.Print(function (error, result) { // 콜백함수  
 ...  
 console.log("Event fired: " + JSON.stringify(result.returnValues)); // 결과 ```result```를 출력한다.  
 ...  
});

이 이벤트는 multiply() 함수 내에서 발생되고 있다. 그런 까닭에 **multiply() 함수는 pure로 선언할 수 있지만 event를 발생하기 때문에 그렇게 설정하지 않은 것을 잘 기억해야 한다.** 주의하자. call() 함수는 트랜잭션을 발생시키지 않고, **이벤트도 발생하지 않는다**. 따라서 web3에서 send로 m7.methods.multiply(8).send({from: accounts[0]}) 이렇게 호출한다. 이렇게 트랜잭션을 발생시켜야 블록체인에 기록되고, 로그에 이벤트가 기록된다.

[파일명: src/Multiply7EventUse.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./Multiply7Event.json'); // 파일 읽기  
  
//var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345"));  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙 명 ['src/Multiply7Event.sol:Multiply7Event']  
//console.log("- contract name: ", contractName);  
\_abiArray=\_abiBinJson.contracts[contractName].abi; // 파일에서 바로 읽어오기  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi);   
  
var m7 = new web3.eth.Contract(\_abiArray, '0x33add2effA3E32050aCD8446d826b0EFFB93A515');  
var event = m7.events.Print({fromBlock: 0}, function (error, result) {  
 if (!error) {  
 console.log("Event fired: " + JSON.stringify(result.returnValues));  
 //process.exit(1); //이벤트의 무한 대기를 강제 종료하고 싶은 경우  
 }  
});  
  
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 value = m7.methods.multiply(8).call(); //call 방식으로 호출  
 const value = m7.methods.multiply(8)  
 .send({from: accounts[0]}) //send 방식으로 호출  
 console.log("---> Function called " + JSON.stringify(value));  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));  
 //process.exit(1); //force exit to end waiting  
}  
  
doIt()

프로그램을 실행해보자. 이벤트가 발생한다면, Event fired가 출력이 되는지 확인하자.

pjt\_dir> node src/Multiply7EventUse.js  
  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Event fired: {"0":"0x9357f478d86D9222f4413bFd91C8adb0F4c728b7","1":"1651956296","2":"56","\_addr":"0x9357f478d86D9222f4413bFd91C8adb0F4c728b7","timestamp":"1651956296","res":"56"}  
Balance before: 999976287563999994122  
---> Function called {"\_events":{}}  
Balance after: 999976287563999994122  
Balance diff: 0

# 6. fallback 함수

## 왜 필요할까

fallback 함수는 다른 프로그래밍 언어에서는 찾아 보기 힘든 특징을 보이고 있다.

* 암호화폐를 내장한 특징을 여실히 잘 드러내듯이 fallback 함수는 수금을 하기 위해 사용될 수 있다. 스마트 계약 주소로 거래를 전송할 때, 이 함수가 호출되어 이더를 수신한다.
* 조금 이상하게 들리겠지만, 존재하지 않는 함수가 호출될 때, 바로 그 지점에서 fallback이 호출될 수 있다. 따라서 fallback 함수는 익명으로 이름이 필요없고, 이름조차 없는데 오버로딩이 필요할 까닭이 없다.
* 금융 보안의 이유로 fallback 함수에 소모되는 gas는 현재 2300으로 제한되어 있다. 이러한 제한은 fallback 함수가 오직 간단한 작업만을 수행할 수 있도록 하는데, 주로 이더를 전송받는 데 사용된다. 외부에서 자동으로 호출할 수 있는데, 상태 변경이나 출금 등 위험한 작업을 허용한다면 어떻게 되겠는가? 안전을 해치는 복잡한 작업은 fallback 함수에서 수행할 수 없도록 제한을 걸어 놓고 있다.
  + 보통 컨트랙의 상태는 상태 트리에서 직접 읽기 때문에 거의 gas가 소모되지 않고,
  + 이벤트는 내부적으로 로그를 기록하는 것이므로 발생하는 gas 역시 거의 무시할 정도이고,
  + 이더의 송금이나 수금하는 경우 2300 gas 이상 들지 않는다.

## 함수는 이름없이 extern으로 선언한다

fallback() 함수는 다음과 같이 선언한다. 사용자가 직접 호출해서 실행되지 않는다. 아니 외부 사용자가 직접 호출할 수가 없다. 따라서 fallback 함수는 이름을 가질 까닭이 없고 함수의 인자와 반환이 필요할 이유가 없다.

fallback () external [payable]

보통 인자와 반환값이 없지만, 최신 버전 0.7이상에서는 개정되어 입력 bytes calldata, 출력 bytes memeory를 가질 수 있다.

fallback (bytes calldata \_input) external [payable] returns (bytes memory \_output)

컨트랙에는 **한 개의 fallback 함수**만 존재할 수 있다. 아래와 같이 하나 이상의 fallback() 함수를 가져서는 안된다. 사용자가 호출하지 않아서 이름도 없는데, 무슨 오버로딩이 필요하겠는가? 즉 fallback 함수는 컨트랙마다 하나만 존재한다. 최신 버전에서 fallback 함수에 인자를 넣어서 구현을 해도 된다고 설명하였지만, 이러한 오버로딩 역시 가능하지 않다.

//오류 코드. 하나 이상의 fallback.  
contract FallbackTest1 {  
 fallback() external {}  
 fallback() external payable {}

fallback 함수도 진화하고 있고, 잘 못된 선언의 예를 보자. external을 빼먹거나, function을 붙이거나 하지 않도록 한다. receive() 함수는 곧 설명한다.

[파일명: src/FallbackTest1.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract FallbackTest1 {   
 receive() external payable {} //정상  
 //receive() {} //오류. external로 선언되어야 함  
 //receive() external {} //오류. payable로 선언되어야 함  
 //function receive() external payable {} //경고 (키워드 function)  
 fallback() external {} //정상. 송금없는 경우 사용.  
 //fallback() {} //오류. external로 선언되어야 함  
 //fallback() external payable {} //정상. 송금있는 경우 사용.  
 //function fallback() external payable {} //경고 (키워드 function)  
 //function() external payable {} //오류. 버전 0.6.x 이하에서는 이렇게 선언하였다.  
 //fallback(bytes calldata \_input) external {} //오류. 반환 필요  
 //fallback(bytes calldata \_input) external returns(bytes memory \_output) {} //정상. 버전 0.7 이상.  
}

아래 첫 번째 컨트랙 FallbackCounter의 fallback() 함수를 보자. 두 번째 컨트랙 CallFallback의 callFallbackCounter() 함수에서는 FallbackCounter를 호출하고 있다. 존재하지 않는 함수가 호출되었기 때문에 fallback이 작동하게 된다. 그러나 주의 하자. fallback이 payable로 선언되지 않아서, 즉 입금불가능하기 때문에 fallback이 호출되지 않고 입금이 실패하는 예외가 발생하게 된다.

[파일명: src/FallbackTest2.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract FallbackCounter {  
 uint public nOfFallback = 0;  
 fallback() external { nOfFallback += 1; } // 입금 불가능  
}  
  
contract CallFallback {  
 function callFallbackCounter(FallbackCounter fb) public returns (bool) {  
 address payable \_fb = payable(address(fb));  
 return (\_fb.send(11 wei)); // 수신측 fallback이 호출되지 않고, 입금실패의 예외 발생  
 }  
}

fallback은 내외부 어디서나 호출될 수 있도록 넓게 **external**으로 선언한다. 위 프로그램은 컨트랙끼리 서로 내부에서 호출하고 있다.

사용자가 명시적으로 호출할 필요가 없는 데 왜 external 선언이 필요할까? 실패는 물론 내부에서나 (컨트랙 간에 호출하는 경우), 외부에서도 발생할 수 있다. 외부에서 실패하는 경우를 대비해서 external로 선언한다. 아래 실습에서는 클라이언트, 즉 외부에서 fallback이 호출되도록 하고 있다.

## 존재하지 않는 함수를 호출하는 경우 fallback을 실행

fallback 말 그대로 대비하는 함수이다. 컨트랙을 호출하는 경우, 그 함수가 **존재하지 않을 경우**에 fallback 함수가 대신해서 자동 호출되는 것이다. 사용자가 호출하는 함수는 아니다.

이 함수는 이름이 없으니 개발자가 만들 필요가 없다고 생각하지 말자. 함수 코드는 개발자가 채워 넣을 수 있다. 즉 실패하는 조건에 수행할 명령어를 넣어두어야 한다.

스마트 계약에 이더를 전송할 때 fallback 함수가 호출되는데, 이 경우 입금 가능 하도록 선언할 수 있다. 특히 이런 경우에는 상대측의 송금액이 자동으로 입금 처리되기 때문에 신중하게 취급해야 한다.

예를 들어, 일정 금액을 송금했는데 (value 필드에 값을 적었다는 것이다), 수신측에서 이를 받는 함수가 존재하지 않으면, 송금이 거절되어야 하지만 fallback이 호출된다. 이 경우는 **fallback() payable**로 선언해야 한다. 이 경우 그 금액이 환급되지 않고 자동 입금되는 것이다.

송금이 실패한다는 것은 당연히 실패로 끝나야 하고, 송금액은 원래의 위치로 반환되어야 한다. 실패에도 불구하고 이를 가능하게 한다는 그 지점이 바로 주의가 필요하다. fallback payable은 송금되지 않으면 반환되어야 할 금액을 재진입하여 강제 입금하는 결과를 만들어 낼 수 있기 때문이다 (뒷 Chapter '예외'에서 더 설명하게 된다).

## receive 함수가 있으면 fallback에 앞서 호출된다

버전 0.6부터는 receive()함수가 제공되고 있다. 이 함수는 이더리움에서 사용되는 특별한 함수로, 송금을 받을 때마다 자동으로 호출된다.

이 함수는 송금을 하지만 (value 필드에 값이 있는 경우) calldata가 없는 경우 호출된다. 따라서 receive() external payable로 사용해야 한다. receive()함수가 없으면 fallback()함수가 호출되게 된다. calldata는, 앞서 ABI 명세에서 설명한 바와 같이, 함수시그니처와 매개변수를 의미하며 아래 data 필드에 적어준다.

web3.eth.sendTransaction({  
 from: ...,  
 data: "0xcdcd77c000...생략...0000000001",  
 gas: ...}  
)

정리하면, 송금을 할 때:

* msg.data가 없는 경우에는 receive() 함수가 있으면 호출, 없으면 fallback() 이 호출된다.
* msg.data가 있는 경우에는 fallback() 함수가 호출

몇 가지 오류와 정상 코드를 예로 들면 다음과 같다.

//receive() {} //오류 external로 선언되어야 함  
//receive() external {} //오류 payable로 선언되어야 함  
receive() external payable {} //정상 function 키워드 없이 사용됨  
//function receive() external payable {} //경고 function 키워드가 사용됨

아래 송금하는 경우를 보자. 송금을 하면 receive() 또는 fallback()의 payable 함수가 입금을 하게 된다. REMIX에서 실행해보자. 실행하는 순서는:

* deposit 버튼을 눌러 111을 입금한다. REMIX에서 value=111, deposit 버튼과 동일한 금액을 입력해야 한다.
* 입력하고 나면 잔고가 있으니, 송금이 가능하다. sending 버튼을 누르자. 잔고 111에서 11이 송금된다.
* REMIX 우측 하단 콘솔에서 로그를 확인한다. 해당하는 메시지가 로그에 출력되어 있다. receive()가 호출되었다면 'now receiving...', fallback()이 호출되었다면 'now fallback...'이 호출될 것이다.
* 잔고 버튼으로 Receiving의 잔고가 11, Sending의 잔고는 100을 확인한다.

아래 코드 (1)과 (2)를 보자. 이를 번갈아 가며 실행해서 작동하는 방식을 이해하도록 하자.

* (1)은 msg.data가 비어있다. 이 경우 Receving의 어느 함수가 호출될까 receive()일까, fallback()일까? receive()가 호출된다.
* (2)는 msg.data가 비어있지 않다. 존재하지 않는 함수 nonExistingFn()을 호출하고 있다. 즉 데이터는 있다는 뜻이다. 이 경우에는 fallback()이 호출된다.

[파일명: src/FallbackTest3.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Receiving {  
 event PrintLog(string s, address \_from, uint amount);  
 receive() external payable {  
 emit PrintLog("now receiving...", msg.sender, msg.value);  
 }  
 fallback() external payable {  
 emit PrintLog("now fallback...", msg.sender, msg.value);  
 }  
}  
  
contract Sending {  
 Receiving r=new Receiving();  
 function deposit(uint amount) payable public {  
 require(msg.value==amount);  
 }  
 function sending() public payable {  
 //(bool success, bytes memory data) = address(r).call{value:11}(""); // (1) receive 호출  
 (bool success, bytes memory data) = address(r).call{value:11, gas:5000}  
 (abi.encodeWithSignature("nonExistingFn()")); // (2) fallback 호출  
 require(success);  
 }  
 function getBalanceOfThis() public view returns(uint) {  
 return address(this).balance;  
 }  
 function getBalanceOfReceiving() public view returns(uint) {  
 return address(r).balance;  
 }   
}

## 실습문제: fallback

컨트랙의 존재하지 않는 함수를 호출해서, 강제적으로 fallback 함수를 실행해 보자. 앞에서는 블록체인 내부에서 fallback이 호출되는 방식을 보았었다. 여기서는 외부, 즉 웹에서 fallback이 어떻게 호출되는지 학습하자.

### 단계 1: 컨트랙 개발

컨트랙은 간단하게 함수 callA()와 fallback()함수로 구성한다. fallback()은 호출되면 이벤트 PrintLog()를 발생하도록 하자.

[파일명: src/FallbackTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract FallbackTest {  
 event PrintLog(string);  
 function callA () pure public returns(string memory){  
 return "doing callA";  
 }  
 fallback () external {  
 emit PrintLog("fallback called");  
 }  
}

### 단계 2: 컴파일

컴파일해서, abi와 바이트 코드를 파일에 저장한다.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/FallbackTest.sol > src/FallbackTest.json

### 단계 3: 컨트랙 배포

[파일명: src/FallbackTestDeployFromFile.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./FallbackTest.json'); // 파일 읽기  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙 명 ['src/FallbackTest.sol:FallbackTest']  
console.log("- contract name: ", contractName);  
\_abiArray=\_abiBinJson.contracts[contractName].abi;  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); //JSON parsing needed!! //SyntaxError: Unexpected token o in JSON at position 1  
\_bin=\_abiBinJson.contracts[contractName].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: "0x"+\_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()

이 프로그램을 실행하면, 컨트랙의 배포 주소를 출력한다.

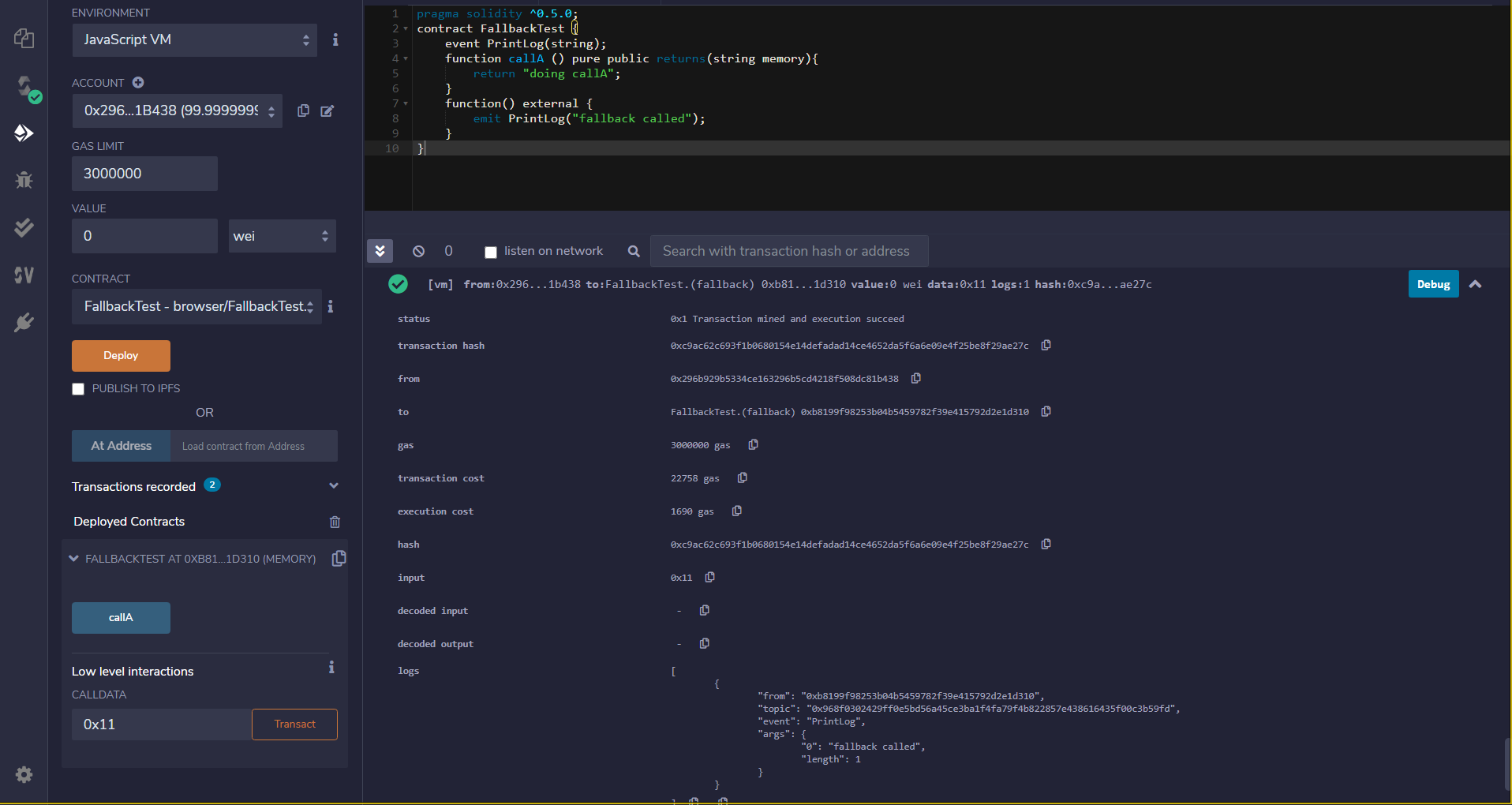
pjt\_dir> node src/FallbackTestDeployFromFile.js  
  
- contract name: [ 'src/FallbackTest.sol:FallbackTest' ]  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0xcd89d730a606179c4c36b84bc2ce58719fce84ecac932af8e2fd3000453fb7da  
---> The contract deployed to: 0x305F89e9b9C91B0b242874d77Ef675b0eBAD437C

### 단계 4: 사용

REMIX에서는 fallback 함수 버튼이 만들어지지 않는다. 몇 번 강조했듯이, 사용자가 호출하는 그런 함수가 아니다.

REMIX에서 배포하고 난 후 그림에서 좌하단을 보자. **Low level interactions**과 바로 그 아래 **CALLDATA**를 발견하게 될 것이다. 이 기능은 REMIX 버전 0.9.3부터 저수준의 상호작용을 지원하고 있는데, CALLDATA를 입력하고 Transact 버튼을 클릭하면 receive 또는 fallback 함수를 호출될 수 있다. 앞서 설명을 기억하면서 해보자.

* CALLDATA가 없으면 receive가 호출된다. 그러니까 CALLDATA에 **아무 데이터도 입력하지 않고** Transact 버튼을 누르면, receive() 함수가 호출된다는 의미이다. 여기서는 fallback 함수가 호출된다 (정확히 말하면, receive() 함수가 없으니까 fallback() 함수가 대신 호출되는 것이다).
* CALLDATA가 있으면, fallback이 호출된다. CALLDATA는 임의의 데이터, **적어도 1바이트 이상을 16진수로 입력 (예를 들면 0x11)**하고 Transact 버튼을 누르면, fallback() 함수가 호출된다는 뜻이다. fallback()이 호출된 것을 어떻게 확인할 수 있을까? REMIX 우하단 콘솔 창에 로그가 출력된다. 로그를 살펴보면 우리가 만들어 놓은 이벤트 출력을 통해 그 발생을 확인할 수 있다.



alt text

[파일명: src/FallbackTestUseFromFile.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./FallbackTest.json'); // 파일 읽기  
  
//var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345")); //이벤트 발생을 위한 웹소켓연결  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙 명 ['src/FallbackTest.sol:FallbackTest']  
console.log("- contract name: ", contractName);  
\_abiArray=\_abiBinJson.contracts[contractName].abi; // abi를 바로 읽어오기. 오류가 발생하면 잘 읽혔는지 확인!  
//\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); //JSON parsing needed!!  
  
console.log("- ABI: " + \_abiArray);  
  
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);  
 var \_instance = new web3.eth.Contract(\_abiArray, "0x305F89e9b9C91B0b242874d77Ef675b0eBAD437C");  
 var event = \_instance.events.PrintLog({fromBlock: 0}, function (error, result) {  
 if (!error) {  
 console.log("Event fired: " + JSON.stringify(result) + "\n---> " + JSON.stringify(result.returnValues));  
 }  
 });  
  
 \_instance.methods.callA().call().then(function(res) { console.log(res); }); //null  
 //존재하지 않는 함수 callB() 호출하면 fallback 작동하지 않는다.  
 //await \_instance.methods.callB().send({from:accounts[0], to: "0x3991e87b71cBFf94aA0718F341d8Ad4bCF969f36"});  
 //await \_instance.methods.callA().send({from:accounts[0], data:"0x1234"}); //fallback 호출 실패  
 //web3의 함수로 호출하면 fallback이 호출된다. 컨트랙 함수를 호출하지 않고, 컨트랙 주소로만 호출하고 있다.  
 web3.eth.sendTransaction({from:accounts[0], to:"0x305F89e9b9C91B0b242874d77Ef675b0eBAD437C"});  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));  
 process.exit(1); //force exit  
}  
doIt()

#### 웹3에서는 존재하지 않는 함수를 호출하면 당연히 오류가 발생한다.

웹3에서는 존재하지 않는 함수를 호출하면, abi에 존재하지 않기 때문에 당연히 오류가 발생하게 된다. abi를 열어보라. 예를 들어 프로그램에서 호출하는 callB() 함수가 정의되어 있는가? 당연히 없을 것이고, 함수 정의가 없는 함수를 어떻게 호출한다는 말인가?

REMIX에서 CALLDATA를 제공하듯이 data 필드를 임의로 수정해서 실행하면 어떻게 될까? 이 역시 abi의 함수의 입출력 정의와 다른 함수는 호출할 수 없다. callA().send({from:accounts[0], data:"0x1234"}) 역시 fallback을 호출하지 못한다.

#### 존재하지 않는 function selector로 fabllback 호출

**존재하지 않는 함수를 호출**하면 fallback함수가 호출된다. 웹3에서는 존재하지 않는 함수를 호출할 수 있는 방법은 없다. 가능한 방법은 web3.eth.sendTransaction()의 data, 즉 msg.data를 수정하면 된다. 앞서 설명한 **function selector** (호출할 함수시그니처의 sha3 해시값으로 만들고, 앞 4바이트)가 존재하지 않도록 수정해야 한다. 참고로 앞서 컨트랙 간에는 msg.data를 수정해서 호출하고 있다 (앞의 FallbackTest2.sol 또는 FallbackTest3.sol 참조)

아쉬운대로 다음과 같이 **함수명을 생략하여 강제적으로 존재하지 않는 함수**를 호출한다. 즉 컨트랙 주소로 그냥 트랜잭션을 전송하여 없는 함수가 호출된 것처럼 한다.

web3.eth.sendTransaction({from:web3.eth.accounts[0], to:컨트랙 주소});

주의하자. 위에서 Http를 사용하면 이벤트가 발생하지 않는다. 웹소켓으로 적용하면 Event fired 메시지가 출력된다.

pjt\_dir> node src/FallbackTestUseFromFile.js  
  
- contract name: src/FallbackTest.sol:FallbackTest  
Account: 0x8218Ee4e07eE1a6000ce4542f6DC9532611A18f1  
Balance before: 99995276566000000000  
doing callA  
Balance after: 99995276566000000000  
Balance diff: 0  
Event fired: {"logIndex":0,"transactionIndex":0,"transactionHash":"0x988203b9673bb10e9ecfed1d9e4a9aad1f6555607be560c40139625153a8ec0b","blockHash":"0x10bf044ab1a16f74a8fd22436b7cdf4cc6c1e82da6b23c8afb5c3a1a13f446f7","blockNumber":13,"address":"0x3991e87b71cBFf94aA0718F341d8Ad4bCF969f36","type":"mined","id":"log\_6023f931","returnValues":{"0":"fallback called"},"event":"PrintLog","signature":"0x968f0302429ff0e5bd56a45ce3ba1f4fa79f4b822857e438616435f00c3b59fd","raw":{"data":"0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000000f66616c6c6261636b2063616c6c65640000000000000000000000000000000000","topics":["0x968f0302429ff0e5bd56a45ce3ba1f4fa79f4b822857e438616435f00c3b59fd"]}}  
 ---> {"0":"fallback called"}

## 실습문제: fallback payable

앞서 존재하지 않는 함수를 호출하는 경우, fallback함수가 호출되었다. 또한 금액없이 송금을 하여도 fallback함수가 호출된다.

송금을 하면서 실패하였다고 하자. 이 경우 fallback이 호출된다 하더라도 그 송금액은 수취할 수 없다. 단 fallback() payable으로 선언하면 가능해진다.

### 단계 1: 컨트랙 개발

fallback() 함수가 호출되려면, web3.js 측에서가 아니라 다른 컨트랙에서 실험하는 편이 좋다. 그렇다면 부득이 컨트랙을 하나 더 만들어야 하고, 두 컨트랙이 연관을 가져야 한다.

컨트랙 Math와 Multiply7을 만들어보자.

Multiply7에는 multiply() 함수와 더불어 receive(), fallback() 함수를 만들고, 어떤 조건에 맞추어 이 함수들이 호출되게 된다.

컨트랙 Math는 Multiply7을 multiply() 함수를 호출하거나 receive() 또는 fallback() 함수를 호출하도록 하자.

[파일명: src/MathMultiply7.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Multiply7 {  
 event PrintLog(string s, address \_from, uint \_amount);  
 event PrintLog(string s, address \_from);  
 event PrintLog(uint);  
 receive() external payable {  
 //emit PrintLog("now receiving in Multiply7", tx.origin, msg.value); //오류  
 //emit PrintLog("now receiving in Multiply7", tx.origin, 11);  
 emit PrintLog("now receiving in Multiply7", tx.origin);  
 }  
 fallback() external payable {  
 //emit PrintLog("now fallback in Multiply7", tx.origin, msg.value); //오류  
 //emit PrintLog("now fallback in Multiply7", tx.origin, 111);  
 emit PrintLog("now fallback in Multiply7", tx.origin);  
 }  
 function multiply(uint input) pure public returns (uint) {  
 //emit Print(input \* 7);  
 return input \* 7;  
 }  
 function getAddress() view public returns(address) {  
 return address(this);  
 }  
}  
  
contract Math {  
 Multiply7 m7=new Multiply7(); //동일한 파일에 존재하는 컨트랙의 ```new``` 객체 생성  
 function deposit(uint amount) payable public {  
 require(msg.value==amount);  
 }  
 //이미 배포된 컨트랙을 사용하려고 하면, 주소를 인자로 받아 사용한다.  
 function setM7(address payable \_addr) public { m7 = Multiply7(\_addr); }  
 function multiply() view public returns(uint) {  
 uint res=m7.multiply(8);  
 return res;  
 }  
 function send11M7() public payable {  
 payable(address(m7)).transfer(11); //컨트랙 주소를 payable로 형변환  
 }  
 function getBalanceOfThis() public view returns(uint) {  
 return address(this).balance;  
 }  
 function getBalanceOfM7() public view returns(uint) {  
 return address(m7).balance;  
 }  
 function getAddressOfM7() view public returns(address) {  
 return address(m7);  
 }  
}

REMIX에서 버튼을 누르며 테스트 해보자.

* 먼저 Math의 적색버튼 deposit을 누른다. 이 때 인자에 111을, 위 VALUE에도 같은 숫자 111을 적는다.
* 그리고 Math의 적색버튼 send11M7을 누른다. 이때 REMIX의 우측 아래 단말를 보면, 녹색의 성공 체크표시가 뜬다 (실패하면 적색 X 표시). 우측의 청색 Debug를 펼치고, logs로 이동하고 args를 보면, 다음과 같은 로그 출력을 볼 수 있다.

"args": {  
 "0": "now receiving in Multiply7",  
 "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  
 "s": "now receiving in Multiply7",  
 "\_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  
}

위 출력을 보면, receive() 함수가 호출되고 있음을 알 수 있다. 소스코드의 도움말을 활성화해가면서 (쓰지 않는 부분은 비활성화), 예를 들어 receive()를 비활성화하면 fallback()을 호출하게 된다.

* fallback이나 receive를 호출되는 이유를 잠깐 살펴보자. payable(address(m7)).transfer(11)는 m7에 송금을 하는 것인데, 함수를 통하지 않고 있다. 즉 없는 함수를 호출하면 어떻게 처리를 할 방법이 없다. 이 때 receive가 호출된다. receive 함수가 없으면 fallback이 호출된다.

### 단계 2: 컴파일

지금은 버전 0.8으로 컴파일해서 ganache@8345에서 배포, 실행하려고 한다. geth@8445에서 실행하려면, 혹시 버전 문제가 있을 수 있다. 그렇다면 버전을 낮추어 적용할 수도 있다 (저자는 0.4.21로 낮추어 한 경험이 있다.) 플랫폼에 맞는 Solidity 컴파일러를 이용해서 컴파일 해본다.

combined-json 명령어를 abi, 바이트 코드 따로 적용해서 해보자.

pjt\_dir> solc src/MathMultiply7.sol --combined-json abi > src/MathMultiply7ABI.json

파일에 2개의 컨트랙이 포함되어, 아래에서 보듯이 abi는 2개가 생성되고 있다.

pjt\_dir> cat src/MathMultiply7ABI.json  
  
{"contracts":{  
 "src/MathMultiply7.sol:Math":{"abi":"[...생략...]"}, (1) Math 컨트랙  
 "src/MathMultiply7.sol:Multiply7":{"abi"[...생략...]"} (2) Multiply7 컨트랙  
 },"version":"0.6.1+commit.e6f7d5a4.Linux.g++"}

이번에는 바이트 코드를 생성한다.

pjt\_dir> solc src/MathMultiply7.sol --combined-json bin > src/MathMultiply7BIN.json

바이트 코드도 역시 2개가 생성되고 있다.

pjt\_dir> cat src/MathMultiply7BIN.json  
  
{"contracts":{  
 "src/MathMultiply7.sol:Math":{"bin":"608060...생략...10033"},  
 "src/MathMultiply7.sol:Multiply7":{"bin":"608060...생략...10033"}  
 },"version":"0.6.1+commit.e6f7d5a4.Linux.g++"}

### 단계 3: 컨트랙 배포

기억하자. 2개의 컨트랙이 있고 이 가운데 외부에서 사용하게 될 Math 하나만 배포하고 있다. abi, 바이트 코드 모두 Math만 포함한다.

[파일명: src/MathMultiply7DeployFromFile.js]  
var Web3=require('web3');  
var \_abiJson = require('./MathMultiply7ABI.json');  
var \_binJson = require('./MathMultiply7BIN.json');  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractNames=Object.keys(\_abiJson.contracts); //Math, Multiply7  
contractName=contractNames[0]; // -> 'src/MathMultiply7.sol:Math', contractNames[1] -> Multiply7  
console.log("- contract name: ", contractName);  
//\_abiArray=JSON.parse(\_abiJson.contracts[contractName].abi); //JSON parsing needed!!  
\_abiArray=JSON.parse(JSON.stringify(\_abiJson.contracts[contractName].abi)); //Unexpected token error  
\_bin="0x"+\_binJson.contracts[contractName].bin;  
//\_bin=\_binJson.contracts[contractName].bin;  
  
async function deploy() {  
 const ac counts = 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()

프로그램을 실행해서 컨트랙 배포 주소를 출력한다.

pjt\_dir> node src/MathMultiply7DeployFromFile.js  
  
- contract name: src/MathMultiply7.sol:Math  
Deploying the contract from 0x83e790081F4117bEfe83d35bcDEC772e066F675D  
hash: 0xed1c5059d1f02cf839fc125810d326a34122369da38f7f298c6def3b0cee07db  
---> The contract deployed to: 0xcD3F2A7eca9E6cF6dA466E0e011858E05B033C65

### 단계 4: 사용

배포가 되었으면 API를 사용해보자. 특히 fallback 함수가 호출되는지 주의해서 살펴보자. 일반적으로 상대측에 입금하는 함수가 있어야 그 함수를 호출하여 송금이 가능하다. 상대측인 Multiply7 소스코드를 보면 어디에도 입금하는 함수는 발견할 수 없다.

그럼에도 불구하고, fallback payable이 작동하여 입금이 되고 있다는 것이다. 여기서는 잔고가 이동하면 fallaback payable이 작동한 결과인 것이다.

이벤트가 발생하도록 Solidity를 프로그래밍하였지만, 웹소켓을 통한 이벤트 발생은 하지 않고 있다. 스스로 이벤트가 발생하도록 코드를 변경하면서 학습해도 좋다.

#### 컨트랙 연관관계에서의 주소 설정

new 명령어는 동일한 파일에 연관관계 컨트랙이 존재할 때 주소를 할당하여 객체를 생성한다. setM7(address) 함수는 Multiply7를 배포하고, 확득한 주소를 할당한다.

#### 입금해서 잔고가 있어야 송금 가능: send11M7()

송금하면서 payable fabllback를 호출되게 하기 위해 의도적으로 아래와 같이 존재하지 않는 함수를 호출한다. 다음 절차에 따라 실행하는 코드를 작성하면 된다.

* 1. 송금은 물론 잔고가 있어야 가능하다. deposit(123)을 먼저 실행해야 한다. value를 반드시 함수인자와 동일하게 입력해야 한다. REMIX에서 테스트할 경우에도 마찬가지이다. Run 탭 상단의 value, Deployed Contracts 함수의 인자 두 필드에 동일한 금액을 넣어준다.
  2. getBalanceOfThis() --> 123이 된다. 입금하고 난 후의 잔고이다.
  3. getBalanceOfM7() ---> 0 아직 Multiply7의 잔고는 비어있다.
  4. send11M7() 함수는 11을 송금한다. 이 때 다른 컨트랙 Multiply7의 payable fallback 함수가 호출되어 11 Wei가 송금된다.
  5. getBalanceOfThis() ---> 112 (11만큼 차감된 결과)
  6. getBalanceOfM7() ---> 11 payable fallback이 실행되어 11 wei 송금 성공.

[파일명: src/MathMultiply7UseFromFile.js]  
var Web3=require('web3');  
var \_abiJson = require('./MathMultiply7ABI.json');  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
//var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345"));  
  
contractNames=Object.keys(\_abiJson.contracts); //Math, Multiply7  
contractName=contractNames[0]; // -> 'src/MathMultiply7.sol:Math', contractNames[1] -> Multiply7  
console.log("- contract name: ", contractName); //or console.log(contractName);  
//\_abiArray=JSON.parse(\_abiJson.contracts[contractName].abi); //JSON parsing needed!!  
\_abiArray=JSON.parse(JSON.stringify(\_abiJson.contracts[contractName].abi)); //Unexpected token error  
//\_bin=\_binJson.contracts[contractName].bin;  
  
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);  
 var \_instance = new web3.eth.Contract(\_abiArray, "0xcD3F2A7eca9E6cF6dA466E0e011858E05B033C65");  
  
 \_instance.methods.multiply().call().then(console.log); //56 출력 (8 곱하기 7 연산)  
 \_instance.methods.deposit(123).send({from:accounts[0], value:123}); //value, 인자에 동일한 123으로 적음  
 \_instance.methods.getBalanceOfM7().call().then(console.log); //0 출력  
 await \_instance.methods.send11M7().send({from:accounts[0]}); //11을 송금  
 \_instance.methods.getBalanceOfM7().call().then(console.log); //잔고 11을 출력  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));  
}  
  
doIt()

프로그램을 실행해보자.

pjt\_dir> node src/MathMultiply7UseFromFile.js  
  
- contract name: src/MathMultiply7.sol:Math  
Account: 0x83e790081F4117bEfe83d35bcDEC772e066F675D  
Balance before: 999997545196000000000  
56  
0  
Balance after: 999997431831999999877  
Balance diff: 113364000047104  
11

한 번 더 실행하고 결과를 보면, 잔고가 증가하고 있다는 것을 알 수 있다. 위에서 11이 송금되었고, 2번째 출력에서 이를 반영해 11을 출력한다.

pjt\_dir> node src/MathMultiply7UseFromFile.js  
  
- contract name: src/MathMultiply7.sol:Math  
Account: 0x83e790081F4117bEfe83d35bcDEC772e066F675D  
Balance before: 99978782245999999877  
56  
11  
22  
Balance after: 99977762925999999754  
Balance diff: 1019320000004096

## 연습문제

1. 생성자 문법이 맞는지 또는 틀린지 답하시오.
   1. constructor() Hello public {}
   2. constructor() internal {}
   3. constructor() private {}
   4. constructor() public {}
   5. constructor() {}
2. Solidity는 객체지향 언어이다. 객체지향에서 가능한 생성자 오버로딩이 Solidity에서도 가능한지 OX로 답하시오.
3. pure함수는 상태변수를 읽을 수 있다. OX로 답하시오.
4. view함수는 이벤트를 발생할 수 있다. OX로 답하시오.
5. web3.js 명령문 가운데, { value: 0 }에 Ether 또는 Wei를 입력할 수 있다. 이 값이 전달 되는 전역변수를 적으시오.
6. public 가시성은 내, 외부에서 모두 호출할 수 있다. OX로 답하시오.
7. external 가시성은 내부에서 호출할 수 없다. OX로 답하시오.
8. 함수의 제약 조건으로 사용할 수 있는 것은?
   1. modifier
   2. underscore
   3. event
   4. returns
9. 블록체인에서의 이벤트는 어디에 기록이 되는가? 여기를 계속 관찰하면서, 이벤트가 발생하는지 포착한다.
10. 이벤트는 오버로딩(overloading)이 가능하다. OX로 답하시오.
11. 이벤트는 http를 통해서 인식할 수 있다. OX로 답하시오.
12. fallback 문법이 맞는지 또는 틀린지 답하시오.
    1. fallback() {}
    2. function() external payable {}
    3. fallback() external payable {}
    4. fallback() external {}

13 컨트랙에는 한 개의 fallback 함수만 존재할 수 있다. OX로 답하시오.

1. msg.data가 있는 경우, fallback() 함수가 없으면 receive() 함수가 먼저 호출된다. OX로 답하시오.
2. 블록체인의 컨트랙을 통해 주문하고, 그 주문 내역은 로컬 파일에 저장하는 프로그램을 작성하시오. 다음 함수를 작성하시오. 그 외 필요한 함수는 추가하시오.

* order(상품항목, 주문개수, 단가, 배송지) 함수: '개수 x 단가'로 주문 금액을 계산하고 이 금액만큼 입금되지 않으면 예외를 발생시킨다. 함수 내에 주문 내역을 이벤트에 첨부하여 발생하도록 한다. 로컬에서 그 이벤트의 주문 내역을 포착하여 파일로 저장한다. (참조: EventTest.sol에서 이벤트가 발생하고 로그를 파일 OrderEvent.txt에 작성하고 있다.)
* getBalance() 함수: 최초 프로그램 생성자 owner만 읽을 수 있게 modfier isOwner 작성.
  1. REMIX에서 생성한 버튼을 눌러 주문하고, 이벤트 발생하는지 보이도록 콘솔 출력을 붙여 넣으시오.
  2. 컴파일해서 abi, 바이트 코드를 파일에 저장하시오.
  3. ganache@8345에서 배포하시오 (프로그램명: 'OrderEventDeploy.js')
  4. 배포를 하고 API를 사용하는 프로그램을 작성하시오 ('OrderEventUse.js'). 주문을 3 건만 하고, 이벤트를 통해 다음과 같이 출력하시오. 동시에 주문 내역을 로컬 파일에 저장하시오. 다음 항목의 순서는 차례대로 주소, 상품 항목 번호, 주문 개수, 단가, 배송지이다.
  + 전송자주소, 1111, 3, 100, 20 2-gil Hongji-dong Jongro-gu Seoul  
    전송자주소, 1111, 5, 100, 20 2-gil Hongji-dong Jongro-gu Seoul  
    전송자주소, 1111, 20, 100, 20 2-gil Hongji-dong Jongro-gu Seoul
  1. 파일에 저장된 주문 내역을 명령창(터미널) 화면에서 출력하시오.
  2. 지금까지는 ganache@8345에서 하였는데, 이번에는 geth@8445로 해보시오. 여러 건을 일괄 처리하기 위해서는 계정의 지급 해제가 단 건으로 끝나지 않고 연속적으로 가능하도록 해야 하고, 마이닝도 그래야 한다. 복수 거래를 일괄 처리하기 어려우면, 한 건씩 발생하도록 'OrderEventUse.js' 파일을 나누어 코딩해도 된다.

1. 앞서 FallbackTest.sol에 receive() 함수를 추가하고, 배포하시오. 그리고 이 함수를 노드에서 호출되도록 프로그램을 작성하여 실행하시오.