# Chapter 8. Solidity 데이터 타입

간단한 스마트 컨트랙을 만들어 보는 것부터 시작해보자.

컨트랙 코드에 앞서 사전에 적어야 할 것이 몇 가지 있다. 먼저 라이센스를 설정에 필요한 SPDX, 버전을 정하는 pragma를 적어야 한다. 그리고 다른 코드를 가져오기 위한 import문이 있다면 다음 줄에 적어 준다.

컨트랙 코드는 객체 지향 클래스와 비슷하다. 컨트랙은 멤버 변수와 멤버 함수로 구성을 한다. 여기에서는 우선 멤버 변수에 대하여 불린, 정수, 문자열, struct, 배열, 주소 데이터 타입을 설명하게 된다.

컴파일하면, 바이트코드가 생성이 되는데, 이러한 어셈블리(assembly) 언어와 비슷한 Opcode에 대해서 설명한다.

# 1. 컨트랙 프로그램

## 1.1 시작하기 두려워? 그렇지 않아!

블록체인에 프로그램을 만들려고 하니까 혹시 두렵지 않아?

새로운 기술이라서 사뭇 다를 것 같고, 인터넷 연결만 되면 누구나 내 프로그램을 호출할 수 있다는 사실에 긴장될 수 있다. 자신감을 가지고 시작하자! 블록체인은 화폐를 다루는 플랫폼이라 조심스러워 하는 것이 당연하지만, 매우 안전하기도 하다.

Solidity는 객체 지향 언어이고, 다른 프로그래밍 언어와 크게 다르지 않다. 클래스를 만들듯이, 컨트랙도 멤버 변수와 멤버 함수를 프로그래밍하면 된다.

## 1.2 컨트랙은 어떤 모습일까?

프로그램에는 컨트랙 자체와 더불어 SPDX, 버전 pragma, 다른 소스코드를 불러오는 import문, 또한 도움말 comments를 포함하게 된다. 프로그램이 완성되면, 확장자 .sol로 컨트랙을 저장한다.

### SPDX

저작권에 대해 어떤 규정을 따르는지 SPDX Software Package Data Exchange를 적는다. Solidity 버전 0.6.8부터 SPDX 라이센스를 넣어야 한다. (어떤 라이센스가 가능한지 자세한 정보는 https://spdx.org/licenses/ 페이지를 찾아 보자).

GPL을 따르는 경우는 아래와 같이 //를 붙여서 도움말로 적는다.

//SPDX-License-Identifier: GPL-3.0-or-later

### pragma

pragma는 컴파일러 버전을 선택할 때 사용한다. 아직 발전하고 있는 언어라서 버전업이 활발하게 되고 있고, 버전이 바뀌면서 언어의 문법적인 내용이 바뀌어서 **하위 호환성**이 문제가 되기도 한다.

버전의 범위를 지정해서 컴파일할 수 있도록 하려면, 예를 들어 다음처럼 지정한다.

pragma solidity ^0.8.1;

여기서 지정한 버전 번호인 ^0.8.1의 '^' 표시는 이 소소 코드를 Solidity 컴파일러 0.8.1 버전 포함해서 그 이상 0.8의 버전에서는 컴파일할 수 있다는 의미이다. 즉 0.8.1 미만 버전과 0.9.0 버전 이상에서는 컴파일 할 수 없음을 나타낸다.

Solidity 컴파일러는 버전이 바뀌면서 기능이 많이 바뀔 수 있고, 기존 버전의 언어와 문법이 달라질 수 있다. 두 번째 버전 번호는 버그 수정 버전을 나타내는 숫자로, 기존 언어와의 호환성을 유지한다.

### import

import는 filename의 다른 컨트랙, 라이브러리를 사용할 수 있게 해준다. 다른 언어에 비해 **라이브러리가 충분하지 못하여** 불편할 수 있다.

import "filename"

"filename"으로 주어진 파일 내에 있는 **컨트랙, 전역 변수** 등을 그대로 가져오기 때문에 주의한다.

파일이 여러 개 import되었는데, 다른 파일에 있는 전역 변수의 이름들이 동일한 경우가 있을 수 있다. 이럴 때는 단순하게 import 해서는 어떤 전역 변수가 어떤 파일에 있는 것을 나타내는지 알 수 없어 문제가 된다. 이렇게 이름 충돌 문제가 발생하는 것을 막기 위해 import ... as ... 구문을 사용하기도 한다.

import "filename" as symbolName

이렇게 사용하면 filename에 있는 모든 변수 등은 "symbolName.변수" 형태로 사용할 수 있다.

### 주석(도움말)

주석은 C 또는 자바 언어와 동일한 형태를 사용한다.

//를 사용하면 그 줄이 끝날 때까지(줄바꿈이 일어날 때까지) 주석으로 처리한다. /\*...\*/를 적으면 /\*와 \*/ 사이에 있는 내용을 모두 주석으로 처리한다.

코멘트를 이용해서 함수 설명서를 생성하는 도구인 DOXygen(https://www.dOXygen.nl/) 스타일의 @태그를 다음과 같이 사용하는 것도 가능하다.

/\*\*  
 @param p1 The first parameter  
 @param p2 The second parameter  
 @returns the returned results  
\*/

### 컨트랙

#### 컨트랙은 멤버 변수, 멤버 함수로 구성된다.

블록체인 상의 컨트랙은 개념이나 구현하는 방법이 객체 지향 언어에서 사용하는 **클래스와 유사**하다.

클래스가 (1) 멤버 **변수**와 (2) 멤버 **함수**로 구성하듯이 컨트랙도 마찬가지이다.

다른 컨트랙을 불러서 결합할 수도 있다.

또한 Solidity는 객체 지향 언어로서, 컨트랙은 상속, 다형성을 사용하여 확장을 할 수도 있다.

#### 컨트랙 예제 프로그램

컨트랙을 구현하면 어떤 모습을 가질지 궁금할 것이다. 다음 SimpleChild.sol은 상속까지 적용된 코드인데, 읽어보면 이런 컨트랙을 만들게 되는구나 하고 느낌이 올 것이다. 어떻게 보면 객체 지향 자바와 닮았고, 메모리를 효율적으로 사용하는 관점에서는 C언어와 비교해도 공통점을 발견하게 된다.

다음 Solidity 코드를 전부 이해할 수는 없을 것이다. 당연히 모르는 명령어들도 있을 것이다. Solidity 언어에 대한 자세한 내용들은 앞으로 몇 장에 걸쳐서 배울 것이다. 여기서는 그냥 간단히 코드를 훑어보고 어떤 형태로 작성하는지 확인한다. 자바같은 다른 프로그래밍 언어를 배운적이 있다면, 의외로 많은 부분에서 비슷함을 발견할 수 있을 것이다.

간단히 설명하면:

* 줄1 저작권을 주석으로 적는다
* 줄2 컴파일러의 버전을 적는다.
* 줄3 컨트랙 코드는 contract으로 시작하고, 그 명칭은 Parent로 한다.
* 줄4 주석은 //로 시작해서 적는다. 줄7, 줄12도 주석을 적고 있다.
* 줄5~6 컨트랙의 멤버 변수 (상태변수라고 한다), int, address, mapping을 선언하고 있다.
* 줄8~11 자바의 생성자 역할을 하는 constructor()를 구현한다. 변수를 초기화한다.
* 줄13~14 add(), getCounter() 함수를 정의하고 있다. Child에서 부모의 함수를 상속받을 수 있다.
* 줄16 상속 관계의 Parent, SimpleChild를 선언하다.
* 두 번째 컨트랙에서 SimpleChild is Parent는 is로 상속 관계를 연결하고 있다.
* 줄17~18 string 변수, mapping의 키는 address, value는 uint를 선언하고 있다.
* 줄19 event PrintLog 이벤트를 선언한다.
* 줄20 자식 컨트랙의 생성자를 적고 있다.
* 줄21~29 deposit(), queryBalance() 등의 함수를 선언하고 있다. 나중에 배우겠지만, 컨트랙은 함수 변경자(function modifier)를 가질 수 있다.
* 줄30 블록체인에서 컨트랙을 제거하는 함수이다.

[파일명: src/SimpleChild.sol]  
줄01 //SPDX-License-Identifier: GPL-3.0-or-later  
줄02 pragma solidity ^0.8.0;  
  
줄03 contract Parent {  
줄04 // 멤버 변수들  
줄05 address owner; // 0.8.0 버전부터 payable을 생략한다  
줄06 uint private counter;  
   
줄07 //생성자. 0.7 버전부터 생성자에 public를 붙이지 않는다  
줄08 constructor() {  
줄09 owner = msg.sender;  
줄10 counter = 0;  
줄11 }  
줄12 //함수들  
줄13 function add() public { counter++; }  
줄14 function getCounter() public view returns(uint) { return counter; }  
줄15 }  
  
줄16 contract SimpleChild is Parent { //상속관계를 구현하고 있다.  
줄17 string nickName;  
줄18 mapping(address => uint) private balances; //딕셔너리 자료  
  
줄19 event PrintLog(address, uint); //이벤트의 설정  
   
줄20 constructor() {} // public으로 선언하지 않아도 된다  
  
줄21 function setNickName(string memory s) public { nickName = s; }  
줄22 function getNickName() public view returns(string memory) { return nickName; }  
줄23 function deposit() public payable {  
줄24 balances[msg.sender] += msg.value;  
줄25 emit PrintLog(msg.sender, msg.value);  
줄26 }  
줄27 function queryBalance() public view returns (uint) {  
줄28 return balances[msg.sender];  
줄29 }  
   
줄30 function kill() public {  
줄31 if (msg.sender == owner) selfdestruct(payable(owner)); //0.6.x 버전에서는 selfdestruct(owner)  
줄32 }  
줄33 }

유닉스 운영체제에서는 solc, 윈도우에서는 다음 solc-windows.exe로 컴파일하자. 오류가 있으면 뭔가 출력이 있지만, 그렇지 않으면 아무런 출력이 없다. 다른 언어에서는 테스트 출력을 하게 되는데, 여기서는 그렇지 않아서 그렇다.

pjt\_dir> solc-windows.exe src\SimpleChild.sol

## 1.3 프로그래밍 스타일

Solidity 프로그래밍 언어를 사용하는 방법은 객체 지향 프로그래밍 언어인 자바의 프로그래밍 방법과 비슷하다. 다음은 Solidity 프로그램을 작성할 때 사용되는 스타일이다. 차츰 배워나가면서 코드에 스타일을 더 익히게 되겠고, 여기에서는 시작에 필요한 수준으로 생각하자.

* Solidity에서 파일명은 컨트랙명과 일치하지 않아도 된다. 하지만 일치시키는 것을 권고한다.
* import문은 프로그램의 시작 부분에 작성하되, 프로그램 위, pragma 다음 줄에 적어준다.
* 컨트랙은 새로운 줄에 시작한다.
* 컨트랙, Event, enum, Struct: 대문자로 시작하고, 낙타 표기법(camel case) 스타일로 단어의 첫글자는 대문자로 작성한다.
* 함수, 함수인자, 변수, 함수 변경자(modifier)는 소문자로 시작하고 낙타 표기법으로 작성한다.
* 상수는 모두 대문자로 작성하되, 단어 사이를 밑줄 문자(underscore)로 연결해서 적어준다 (예: DATE\_OF\_BIRTH).
* 한 파일에 여러 컨트랙을 포함할 경우, 컨트랙 간에는 2줄 띄어쓰기를 해준다.
* 들여쓰기 할 때 탭 문자를 사용하지 말고 공백 문자 4칸을 넣어주자.
* 배열은 int[] x;이라고 적어준다 (int [] x 또는 int x[]가 아니라)
* 문자열은 쌍따옴표를 이용해서 표시한다.
* 함수는 새로운 줄에 적어준다
* 한 줄은 최대 79 문자를 넘지 않도록 하자.
* 괄호에서는 한 칸 띄어쓰기를 하지 않는다. if (x == 1) (if ( x == 1 )이 아니라)
* 연산자 앞 뒤에 공백 1칸을 넣는다.
* 코드 블록을 적을 때, 다음에 보인 것처럼 줄바꿈 없이 중괄호를 연결한다.

if (x == 1) {  
}

더 알아보기

낙타 표기법 또는 카멜 표기법(camel case)

낙타 표기법이란 변수나 함수 등의 이름을 코드로 작성할 때 단어의 시작 글자는 대문자로 나머지는 소문자로 표기하고 모든 단어들을 붙여서 표현한다. 단어의 시작 글자들이 마치 낙타 등처럼 위로 솟아올라 있다고 해서 카멜 표기법이라고 한다.

자바 등을 비롯한 많은 언어들이 관례적으로 낙타 표기법을 사용한다. 자바나 Solidity 같은 언어에서는 경우에 따라 첫 번째 단어의 첫 글자는 소문자로 표기하고 두 번째 단어의 첫 글자부터 대문자로 표기하기도 한다.

다음은 낙타 표기법으로 작성한 몇 가지 예를 보인다.

camelCase, CamelCase, aVariable, AVariable, NewWorld, VariableNumber2

## 1.4 예약어

다음은 Solidity에서 제공하는 예약 명령어(reserved word)를 알파벳 순으로 나열한 것이다. 예약 명령어는 키워드(keyword)라고 부르기도 하며 Solidity 언어에서 특별한 목적으로 사용하겠다고 지정한 단어들이다. 이런 명령어는 컨트랙이나 변수를 명명할 때 사용하지 않는다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 예약어 | 예약어 | 예약어 | 예약어 | 예약어 |
| abstract | after | alias | apply | auto |
| case | catch | copyof | default | define |
| final | immutable | implements | in | inline |
| let | macro | match | mutable | null |
| of | override | partial | promise | reference |
| relocatable | sealed | sizeof | static | supports |
| switch | try | type | typedef | typeof |
| unchecked |  |  |  |  |

# 2. 데이터 타입

Solidity 언어는 공간이 제한된 블록체인에서 실행되는 까닭에 **저장공간을 효율적**으로 사용하는 편이 좋다. 저장공간을 많이 사용할수록 비용이 증가한다.

다른 언어에서 지원하는 데이터타입을 큰 차이 없이 사용할 수 있지만, 메모리를 가급적 적게 사용할 수 있도록 설계되어 있어 불필요한 낭비를 줄이고 있다. 메모리는 실로 중요한 자원이고, 블록체인에서 불필요하게 또는 과도하게 **배열을 검색**하거나 **반복문**을 사용하는 것을 유의해야 한다.

그리고 소수점은 아직 지원되지 않아 float, double과 같은 자료형이 없다. 또한 암호화폐를 가지고 있어, 계정 주소 타입이 있다는 점이 특별나다.

|  |  |
| --- | --- |
| 자료형 | 설명 |
| bool | 내부적으로 uint8로 표현됨(0 또는 1을 나타냄). 거짓(false)은 0으로 참(true)은 1로 표현된다. |
| uint | 양의 정수(unsigned integer), M은 0부터 256비트까지,즉 8로 나누어지도록 하고 최대 256까지 가능하다 0 < M <= 256, M은 8의 배수만 사용 가능. |
| int | 부호가 있는 정수(signed integer), M은 8 ~ 256비트까지 가능하다 0 < M <= 256, M은 8의 배수만 사용 가능. |
| address | 주소. 크기가 20바이트이므로 **uint160** 이다. |
| uint, int | 숫자가 붙지 않은 경우의 타입으로 uint256, int256를 의미 |
| bytes | 바이너리 타잎, M은 32바이트까지 가능하다 0 < M <= 32. 크기를 지정하지 않는 **bytes는 value type이 아니다**. |
| string | UTF-8 문자열, 동적으로 크기가 정해지므로 **value type이 아니다**. |

## 2.1 불린

bool 자료형은 참(true) 또는 거짓(false)으로 나타낸다. 같다, 다르다, 크기 등을 비교하는 비교 연산자 (==, !=, !, >, <)와 논리 연산자 (&& ||)의 결과가 bool 자료형의 결과값을 생성한다. 아래는 bool 자료형의 결혼 여부를 true로 설정하는 코드이다.

bool isMarried=true;

## 2.2 정수

int는 정수를 표현하는 자료형이고 signed, unsigned로 구분할 수 있다.

* signed int는 양수, 0, 음수 모두 허용되며,
* unsigned int는 0 이상의 정수만 사용할 수 있다.

그 크기는 8, 16, 24, 32 이런 식으로 8 단위로 증가하고 **최대 32**바이트까지 사용할 수 있다. 크기를 적지 않으면 256비트로 간주한다. **기본 값(default value)은 0**이다. +, -, \*, /, %, \*\*, ++, --, +=, -=와 같은 산술연산자를 사용할 수 있다.

256비트 크기의 정수를 선언하고 1을 입력해보자.

int256 x=1;

## 2.3 소수

fixed, ufixed는 Solidity 언어에 존재는 하지만 현재 사용할 수 없는 자료형이다. 소수점을 가진 숫자들을 나타낼 것으로 보인다. 변수 선언은 할 수 있지만, 값을 할당하면 UnimplementedFeatureError 오류가 발생한다.

## 실습문제: int, boolean 사용

불린과 정수 변수를 선언하고, get/set 함수를 구현한다.

REMIX에서 코드를 작성하고, 앞 실습에서 그랬던 것처럼 버튼 테스트까지 해보자.

[파일명: src/IntBool.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract IntBoolTest {  
 bool married = true;  
 uint256 xAge = 22;  
 uint256 yAge = 25;  
 //fixed phi; // = 3.14; // 선언만 가능  
 function update() public {  
 xAge = yAge;  
 yAge = 33;  
 }  
 function setXAge(int age) public {  
 xAge = uint(age); //type conversion  
 }  
 function getXAge() public view returns(uint) {  
 return xAge;  
 }  
 function getYAge() public view returns(uint) {  
 return yAge;  
 }  
 function testInt() public view returns(bool) {  
 assert(xAge>=20 && yAge>=20);  
 return true;  
 }  
 function isMarried() public view returns(bool) {  
 return married;  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 2 | 컴파일러 0.8의 최신 버전으로 설정. 다음 명령문 ^0.8.0은 메이저버전 8, 마이너버전 0을 의미한다. 맨 앞 ^ caret는 메이저버전으로 시작하는 최신 버전을 선택한다는 의미이다. 그러나 가급적 정확한 버전을 적는 편이 더 좋다. |
| 8 | fixed 지원 여부를 확인하려면 주석문을 풀어본다. |
| 9 | update() 함수에서 멤버 변수의 값을 갱신할 수 있다. |
| 13 | 매개변수 int를 넘겨준다. int와 uint의 형변환(type conversion)이 필요하다. |
| 23 | assert()가 오류이면 Exception이 발생하고, 다음 줄이 실행이 되지 않는다. 즉 x==1인 경우 true가 반환된다. |

컴파일해서 오류가 있는지 확인해보자.

pjt\_dir> solc-windows.exe src/IntBool.sol

## 2.4 고정크기 바이트 문자열

고정 크기의 바이트 문자열은 bytes1 , bytes2, ..., bytes32 로 선언할 수 있다. **1바이트부터 최대 32바이트까지 1바이트 단위**로 크기를 지정할 수 있다.

bytes1 ~ bytes32는 16진수(hexadecimal) 정수값 뿐만 아니라 정수, 문자, 문자열을 가질 수 있다. bytes1은 1 바이트, 8비트의 길이를 가진다. byte (bytes가 아님)는 0.8.0에서 제거되었고, bytes1을 대신 사용한다.

bytes1 x= 0xFF;  
bytes23 place1 = "7 hongji-dong jongro-gu"; //23글자에 맞추어 bytes23으로 선언  
bytes8 place2 = "7 hongji"; //8글자에 맞추어 bytes8로 선언.

* 줄1 bytes 값은 **따옴표 없이 0x를 붙여서 16진수로 표현**한다. bytes1에는 한 글자 0xFF를 지정할 수 있다. 문자열을 입력할 수 있지만, 16진수값의 형태로 저장된다.
* 줄2 bytes23에는 23자까지 입력할 수 있다.
* 줄3 bytes8은 8글자 "7 hongji"를 입력하고 있다. 입력은 문자열이지만, 실제는 "0x3720686f6e676a69"이다. 참고로 각 문자의 16진수는 다음과 같다. 한 글자씩 맞추어 보자.

|  |  |  |
| --- | --- | --- |
| 문자 | 아스키(ASCII) 코드 | 16진수 |
| 7 | 55 | 37 |
| space | 32 | 20 |
| h | 104 | 68 |
| o | 111 | 6f |
| n | 110 | 6e |
| g | 103 | 67 |
| j | 106 | 6a |
| i | 105 | 69 |

## 2.5 변동 크기 바이트 문자열

bytes는 고정 크기의 바이트 문자열과 혼돈하기 쉬운데, 크기를 뜻하는 **숫자가 없는** 경우를 말한다. string은 그 크기가 정해져 있지 않아서 가변적이다. 길이 제한이 없는 경우 사용한다. string은 UTF8 형식으로 문자열을 아래와 같이 저장한다.

string s = "hello";

크기를 알려주는 bytes.length를 사용할 수 있다. string.length는 불행히도 바로 알 수 없다. **bytes로 형변환을 한 후에** bytes(string).length로 사용할 수 있다.

배열의 인덱스를 이해서 bytes(s)[2]로 3번째 값을 읽을 수 있다.

상수는 constant로 표현한다. 상수 문자열 name을 선언하고 "jsl"을 지정해보자.

string constant name="jsl"; //상수는 constant로 표현한다.

Solidity에서는 다른 언어에서 문자열을 붙이기 위해 사용하는 덧셈 연산자가 없다. 즉 "Hello " + "World" 형태로 문자열이 연결된 "Hello World"같은 문자열이 합성되지 않는다. 문자열을 연결하려면, abi.encodePacked()를 함수를 사용할 수 있다.

string(abi.encodePacked("Hello ", "World")) 이렇게 해야 한다. 다행하게도 버전 0.8.10부터 bytes.concat() 또는 string.concat() 함수를 사용하면 된다.

string.concat("Hello ", "World");

문자열 hello를 예로 들어 인코딩을 설명해보자. abi.encode("hello")는 줄을 띄어서 적고 있지만, 패딩으로 채워서 다음과 같이 출력하고 있다.

0x00000000000000000000 00000000000000000000 00000000000000000000 0020 일부러 10자리씩 한 칸 띄움  
 00000000000000000000 00000000000000000000 00000000000000000000 0005  
 68656c6c6f0000000000 00000000000000000000 00000000000000000000 0000

위 문자열 매 32바이트마다, 숫자가 적혀있다. \* 줄 1, 즉 첫 번째 32바이트의 20: 20은 16진수이고, 10진수로 변환하면 32이다. 즉 처음부터 세어서 32번째에 다음 정보가 있다는 의미이다. \* 줄 2, 즉 두 번째 32바이트의 5: hello 문자열의 길이 5를 말한다. \* 줄 3, 마지막 32바이트의 "68656c6c6f": hello의 16진수 68 65 6c 6c 6f이다.

반면에 abi.encodePacked("hello")를 하면 패딩을 제거하고 0x68656c6c6f를 반환한다.

## 실습문제: 문자열 bytes, string 사용

이번에는 문자열 bytes, string을 선언하고, 함수를 이용해서 설정하거나 값을 읽어오도록 코딩하자. REMIX로 코딩한 후, 함수에 대해 버튼 테스트까지 하도록 한다.

[파일명: src/ByteStringTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract ByteStringTest {  
 //byte b = 0xFF; // bytes1으로 적는다 (버전 0.8.0부터)  
 bytes1 b1 = 0xFF;  
 bytes2 b2 = 0xFFAA;  
 bytes8 place8 = "7 hongji"; // 문자 8개  
 bytes23 place23 = "7 hongji-dong jongro-gu";  
 bytes place = "7 hongji-dong jongro-gu Seoul"; // 가변 길이  
 bytes myBytes = new bytes(3); // 0x000000  
 string constant name = "jsl"; // utf-8 문자열 "jsl"  
 function getB1() public view returns(bytes1) {  
 return b1; //bytes1, so no casting required  
 }  
 function getB2() public view returns(bytes2) {  
 return b2;  
 }  
 function getB23() public view returns(bytes23) {  
 return place23; //fixed size, value type (no memory)  
 }  
 /\*\*@return 가변형 bytes는 참조형이므로 memory로 설정\*/  
 function getBytes() public view returns(bytes memory) {  
 return myBytes; //  
 }  
 function getLengOfBytes23 () view public returns(uint) {  
 return place23.length; // returns 23  
 }  
 function getLenOfBytes() pure public returns(uint) {  
 bytes memory bm = "7 hongji-dong jongro-gu";  
 return bm.length; // returns 23  
 }  
 // 인자는 16진수로 전달한다. 예: bytes1 0x61 bytes2 0x6161  
 // bytes2에 2바이트가 아닌 0x61 또는 0x616161 인자로 전달하는 테스트를 해보자.  
 function setB2(bytes2 \_b2) public {  
 b2 = \_b2;  
 }  
 function setBytes() public {  
 myBytes = "smu"; // smu의 16진수는 0x736d75  
 }  
 function getLenOfString() pure public returns(uint) {  
 string memory nameLocal = "jslLocal";  
 //return nameLocal.length; // 오류 문자열을 bytes로 형변환을 해야한다.  
 return bytes(nameLocal).length;  
 }  
 function getString() pure public returns(string memory) {  
 //function getString() pure public returns(bytes memory) {  
 string memory s = "\xec\x95\x88\xeb\x85\x95"; //"한글";  
 //bytes memory s = "\xec\x95\x88\xeb\x85\x95"; //"한글";  
 return s;  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 2 | 컴파일러 0.8의 최신 버전으로 설정 |
| 6 | bytes1은 1 바이트 길이 (전에는 byte 타입이 있었지만, 0.8.0에서 제거) |
| 8 | bytes8에는 8글자를 입력. |
| 9 | bytes23에는 23글자를 입력. 한 글자를 추가하면 오류가 발생된다. |
| 11 | myBytes의 기본 값은 0이고, 3바이트 이므로 0x000000 |
| 12 | string은 utf-8로 저장되므로, 16진수가 아니라 "jsl" |
| 14 | bytes1로 선언된 b1을 반환한다. (0.8.0 이전까지 byte와 bytes1은 동일한 의미. 0.8.0에서 byte 타입 제거됨) |
| 19 | bytes23과 같이 정해진 경우, memory를 사용하지 않는다. 그러나 bytes와 같은 참조형은 memory를 사용한다. |
| 23 | bytes는 동적배열이다. getBytes()는 3글자로 설정된 동적 배열을 memory로 반환한다. 그 반환 값이 16진수로 출력. |
| 26~28 | bytes23.length는 23이 반환된다. |
| 29~32 | 가변 크기 bytes.length는 23이 반환된다. 같은 값을 가지고 있는 place23과 같은 크기. |
| 36~38 | 매개변수는 hex로 넣어준다. a의 hex는 61이다. bytes1은 "0x61", bytes2 "0x6161"로 입력한다. 2바이트를 넘거나, utf-8 문자열을 넣으면 오류이다. |
| 39-41 | myBytes는 3바이트로 할당되었으므로 크기에 맞추어 넣는다. UTF-8로 넣어도 되고 **"smu"로 넣어주면 0x736d75**로 hex값으로 저장된다 |
| 42~46 | .length는 오류. 반드시 string을 bytes로 변환해서 length를 구한다. |
| 47~52 | 한글을 입력하려면 UTF-8 코드를 16진수 형태로 입력한다. |

REMIX에서 코딩하고 버튼 테스트를 완료한 후, 이제 컴파일 해보자.

pjt\_dir> solc-windows.exe src/ByteStringTest.sol

## 2.6 struct

struct은 서로 관련있는 데이터를 묶어서 새로운 자료형을 만든다. struct의 변수 명은 낙타 표기법을 사용하되, 첫 글자를 대문자로 시작한다.

Student를 uint, string, bool을 struct으로 묶으면 다음과 같이 선언한다.

struct Student {  
 uint num;  
 string name;  
 bool isEnrolled;   
}  
Student s1=Student(201911111,"jslim",true);

## 실습문제: struct 사용

학생 관련 정보를 struct으로 선언하고, 함수를 통해 설정하고 조회해보자. 역시 REMIX에서 코딩하고, 버튼 테스트까지 해보자.

[파일명: src/StructTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract StructTest {  
 struct Student {  
 uint num;  
 string name;  
 bool isEnrolled;  
 }  
 Student s1=Student(201911111,"jslim",true);  
 Student s2;  
 // string 인자는 메모리로 해준다.  
 function setStudent2(uint n, string memory sn, bool e) public {  
 s2.num = n;  
 s2.name = sn;  
 s2.isEnrolled = e;  
 }  
 function getStudent1() public view returns(uint, string memory, bool){  
 return (s1.num, s1.name, s1.isEnrolled); // 항목을 괄호로 묶어서 반환한다  
 }  
 function getStudent2() public view returns(uint, string memory, bool){  
 return (s2.num, s2.name, s2.isEnrolled);  
 }  
 function getStudentName() pure public returns(string memory) {  
 // 우측 값이 함수 내에서 생성되기 때문에, 좌측을 memory를 사용한다고 해준다.  
 Student memory s3 = Student(201911112, "jsl3", true);  
 return s3.name;  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 2 | 컴파일러 0.8의 최신 버전으로 설정 |
| 14 | uint, bool은 value 타입이고 string은 reference 타입이라 memory 공간을 사용한다. |
| 19~24 | struct은 반환할 수 없다. 항목 하나 하나를 반납한다. **web3에서 반환할 경우에는 문자열로 변환해서 toString() 출력**할 수 있다. |
| 26~28 | 새로운 Student를 지역변수 s3로 만들고, 이름만 반환한다. 주의할 점은 s3를 storage, memory 선택할 수 있으나 오른쪽에서 Student를 함수 내 지역에서 생성하고 있으므로 memory를 사용하도록 적는다. |

이제 프로그램을 컴파일 해보자.

pjt\_dir> solc-windows.exe src/StructTest.sol

## 2.7 열거형

enum은 상수로 표현하는 것들 중에서 관련된 것들을 묶어서 쉽게 기억할 수 있는 형태로 구성할 수 있도록 해준다. 예를 들면, 요일을 나타내는 프로그램을 작성한다고 할 때, 일요일을 0 그리고 월요일을 1로 표현하는 것보다는, sunday, monday 같은 형태로 표현하면 기억하기 쉬울 것이다. 이럴 때 enum을 사용하면 좋다.

enum 자료형은 집합 기호안에 단어들을 나열해서 생성한다. enum 자료형의 이름은 관례적으로 대문자로 시작한다.

enum 이름 {단어1, 단어2, ..., 단어n}

enum 요소는 내부적으로 **0부터 시작하여 정수 값**을 가진다. 앞에서 얘기했던 것처럼 상수 대신 기억하기 쉽게 단어를 사용하는 셈이다.

enum요소를 사용하는 것은 이름.단어1 형태로 사용한다.

다음은 성별을 남, 녀로 구성한 enum 자료형이다.

enum Gender {male, female}

요일을 enum으로 구성하면 다음과 같다.

enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}

enum 자료형을 매개 변수로 받는 Solidity 함수를 다른 언어(예: 자바스크립트) 등에서 호출한다면 enum 요소 이름 대신 정수를 전달하면 된다. 예를 들어 Day 자료형은 SUNDAY == 0, MONDAY == 1, ..., SATURDAY == 6이므로, MONDAY ~ SUNDAY 사이의 상수 값이나 0 ~ 6 사이의 정수로 적어줄 수 있고, 그 외 값은 오류가 발생한다.

## 실습문제: enum 사용

요일에 대해 enum을 선언하고, 테스트하는 코드를 작성해보자. REMIX에서 코딩하고 버튼을 생성한 후, 하나 하나 올바르게 작동하는지 확인하자.

[파일명: src/EnumTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract EnumTest {  
 enum Day {MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY}  
 Day myDay = Day.FRIDAY; //index int4  
   
 /\* @return Day의 순서를 반환한다\*/  
 function getMyDay() public view returns(Day) {  
 return myDay; //순서  
 }  
 /\* @param 인자 d는 정수 uint8이면 된다\*/  
 function setMyDay(Day d) public {  
 myDay = d;  
 }  
 //uint를 넘겨주어도 기본 uint8로 변환된다  
 function setMyDayInt(uint d) public {  
 myDay = Day(d);  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 1 | 컴파일러 0.8의 최신 버전으로 설정 |
| 4 | **enum이름.변수명**으로 사용하고, 요소는 index로 읽거나 출력한다. |
| 8 | getMyDay()는 정수 값을 반환한다. |
| 11 | 호출하면서 매개변수를 넘겨줄 때, 0~6의 값을 받을 수 있다. |
| 17 | 요소에 해당하는 값을 index로 설정할 수 있다. |

enum이 적힌 코드를 컴파일 해보자.

pjt\_dir> solc-windows.exe src/EnumTest.sol

## 2.8 배열

1. 고정 배열은 그 크기가 선언할 때 정해지고, 나중에 변동될 수 없다. 나이를 고정 배열로 선언해보자.

uint[3] ages = [15, 25, 35]

1. 반면에 동적 배열은 그 크기가 선언할 때 정해지지 않고, 실행 중 크기가 변동될 수 있다. 점수를 동적 배열로 선언해보자.

int[] marks;  
uint[] marks = [8, 15, 32] //우측 값이 모두 양수라서 int -> uint로 선언한다.  
int[] marks = new int[](5)

배열의 크기는 고정 및 동적 배열 모두 length 속성을 이용해서 알 수 있다. 동적 배열에 대해서 요소를 추가하려면 push, 제거하기 위해서는 pop 함수를 사용할 수 있다.

* length: 배열의 길이, 즉 몇 개의 요소가 포함되어 있는지 출력한다.
* push: 배열에 요소를 추가하는 함수이다. 동적배열, bytes에 사용할 수 있고 string에는 쓰지 못한다.
* pop: 배열에서 요소를 제거하는 함수이다. 동적배열, bytes에 사용할 수 있고 string에는 쓰지 못한다.

## 실습문제: 배열 사용

REMIX에서 고정 및 동적 배열을 선언하고, 사용하는 코드를 작성해보자.

[파일명: src/ArrayTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract ArrayTest {  
 uint[3] ages = [15, 25, 35]; // 고정배열  
 int[] marks; // 동적배열  
   
 /\* @param index 배열 인덱스  
 @param val 인덱스에 입력할 값\*/  
 function updateAges(uint index, uint val) public {  
 if(index>=0 && index <=2)  
 ages[index] = val;  
 }  
 function initMarks() public {  
 marks = new int[](5); // 5개 요소 생성, 기본값은 0으로 설정  
 }  
 function appendMark(int mark) public {  
 marks.push(mark);  
 }  
 function popMark() public {  
 marks.pop();  
 }  
 /\* @return 동적 배열을 반환하는 경우 memory를 사용한다\*/  
 function getMarks() public view returns(int[] memory) {  
 return marks;  
 }  
 /\* @return 고정 배열을 반환하는 경우 memory를 사용한다\*/  
 function getAges() public view returns(uint[3] memory) {  
 return ages;  
 }  
 function getLenOfArr() pure public returns(uint) {  
 //memory is used because locally created array is assigned  
 uint8[3] memory intArr = [0, 1, 2];  
 return intArr.length;  
 }  
}

위 프로그램을 컴파일 해보자.

pjt\_dir> solc-windows.exe src/ArrayTest.sol

## 2.9 계정주소

address는 계정의 주소로서 **20 바이트** 길이를 가지고, 객체로서 잔고를 읽거나 및 입출금이 가능하다.

### 잔고

컨트랙의 현재 잔고는 address(this).balance로 구할 수 있다. 컨트랙 자신의 잔고를 구하려면, this라는 contract 자신의 주소를 의미하는 명령어가 필요하다. 주의 하자. Solidity 0.5.0부터 this.balance는 사용이 금지되었고, address(this).balance로 적어야 한다.

### 송금

**어느 계정으로든 입금이 가능**하다. 자신의 계정에서 그냥 타인의 계정으로 전송하면 된다.

**출금**을 생각해보자. 은행에 빗대어 생각해보면, 타인 계좌에서 임의로 인출을 하는 것은 허용되지 않는다. **출금하려면 자신의 계좌에서만 서명을 한 후 가능**하다.

금액의 단위는 기본 값이 Wei이고, ether 등 다른 단위는 명시하면 된다.

0.5.0 버전부터 주소에서 송금하기 위한 계정은 address payable로 선언해야 한다.

|  |  |  |  |
| --- | --- | --- | --- |
| 차이점 | send() | transfer() | call.value() |
| gas 수정가능 | N | N | Y |
| gas 한도 | 2300 | 2300 | 모든 가용 gas를 넘겨주거나, 일정 gas를 정할 수 있다. |
| 실패하면 | false | throw 예외발생 | false |

#### transfer()는 실패하면 원복된다

수신자에게 amount를 전송하려면 다음과 같이 적는다.

address.transfer(amount) //적은 주소가 수신측이 되고, amount만큼 address로 송금된다.

성공하는 경우, 수신측에서 receive() 또는 fallback() 함수를 통해 수신된다. 또한 수신측에 코드 실행에 필요한 **2,300 gas**가 지급된다.

transfer 뿐만 아니라 어떤 송금이나 실패하는 이유는:

* 송신측에 그만한 잔고가 없거나
* 수신측이 수령을 거절하는 경우이다.

송금이 실패해서 throw가 발생하면:

* 송신자에게 **송금액이 반환**되어 아무런 송금도 없는 것처럼 원래의 상태로 복귀된다 (revert). 따라서 send() 보다 transfer()를 사용하면 예외처리가 용이하다.
* gas는 당연히 소비되었으므로 반환되지 않는다. gas 잔액이 남아도 반환되지 않고 모두 소비된 것으로 한다.
* 여러 계정에 송금하는 중 throw가 발생하면, 나머지 송금계정에는 송금이 아예 발생하지 않을 수 있다.

#### send()는 성공여부를 반환하므로 처리 로직이 필요하다

send() 역시 transfer()와 같이 수신계정으로 송금을 하게 된다. 그러나 송금이 **성공 또는 실패했는지 그 결과를 boolean으로 반환**할 뿐이다. send가 실패하는 경우는 원래의 상태로 복귀하지 않는다는 점에서 transfer와 차이가 있다.

송신측에서 실패하는 경우 어떻게 처리되어야 할지, 금액을 출금계정으로 반환하거나, 적절한 처리 로직을 넣어 주어야 한다. 송금이 실패하는 이유에는 여러 가지가 있을 수 있다. call statck error 또는 out of gas error 등의 이유로 실패하는 경우에도 **예외(Exception)가 발생하지 않고 false를 반환**하게 된다.

따라서 require, revert, assert같은 예외처리를 해주어야 한다 (예외처리는 나중에 배우게 된다) 버전 0.4.10 이전에는 if(!address.send(amount)) throw와 같이 했으나, require(address.send(amount))로 코딩하도록 한다.

성공하는 경우, 마찬가지로 수신측에서 receive() 또는 fallback() 함수를 통해 수신된다. 역시 transfer와 마찬가지로 수신측에 코드실행에 필요한 2,300 gas가 지급된다.

자신의 잔고가 10 wei이상이면, \_receiver 주소에 송금하는 코드이다. require() 예외처리를 강제로 해주고 있다. require()는 나중에 배우겠지만, 괄호 안의 명령이 성공해서 true이어야 한다는 의미이다. 실패하면, 즉 송금이 되지 않으면 그 시점에 예외가 발생하고 중단하게 된다.

address \_receiver = 0x778ea91cb0d0879c22ca20c5aea6fbf8cbeed480;  
if(myAddress.balance>=10) require(\_receiver.send(10));

#### call() 함수는 gas를 명시할 수 있다

이 함수는 송금하기 위해 사용될 수 있다.

앞서 배운 send, transfer 함수가 있는데, 왜 또 송금함수가 필요할까? 이전에는 send, transfer를 사용하는 편이 낫다고 했지만, call() 함수가 보다 안전하게 쓰일 수 있다는 의견도 있다.

그러나 알아두자. call, callcode, delegatecall은 저수준 low-level에서 작동하므로 주의해야 한다. 특히, call함수는 gas를 정할 수 있다는 점에 유의하자. call() 함수에 gas를 명시하지 않으면 실제 사용량이 지급된다. 따라서 받는 측에서 악의적으로 gas의 사용량을 과하게 부과되도록 할 가능성이 열려 있다.

0.5.7 버전부터 call()과 더불어 delegatecall(), staticcall() 함수는 (bool, bytes memory) 형태로 반환값 두 개를 반환한다.

* bool success: 함수 호출이 성공인지
* bytes memory data: 함수가 실제로 반환하는 결과 이전 코딩은
* .call.value(20) 이렇게 했지만, 문법이 다음과 같이 달라졌다.

이미 앞서 배운 적이 있지만, 어떤 함수 뒤에 call, send를 붙여서 사용하는 것을 봤었다. 아래 코드는 그 용법과 다르게, 주소에 붙여서 call이 호출되고 있다는 점에 주의한다. 코드를 보면, call("") 이런 식으로 괄호가 비어 있는데, 존재하는 함수가 없다는 의미이다. 그러면, 나중에 배우겠지만, receive, fallback이 호출되게 된다.

(bool success, ) = <address>.call{value:20}(""); //20 Wei를 송금한다. 빈괄호 ("")는 호출함수가 없다는 의미.  
require(success, "transfer call failed."); //성공이 아니면 메시지 출력 후 실행 중단

gas를 적을 수도 있다. gas 2, 송금 11111 Wei로 설정하고 있다. 주의하자, 이 경우 gas가 낮으면 실패할 수 있다.

(bool success, ) = <address>.call{gas:2, value:11111}(""); //0.6.4이전에는 call.value().gas()()로 적음  
require(success, "transfer call failed.");

수신 컨트랙의 함수 baz(123,true)를 호출하면서 gas 10000, 20 wei를 송금하는 코드를 작성해보자. 이와 같이 call은 상대 컨트랙의 함수를 호출하면서 gas를 설정할 수 있다. 기억하는가. call은 앞서 말한대로 저수준 함수 호출 방식이다. 함수 호출도 abi 인코딩 방식으로 하고 있다. abi.encodeWithSignature("baz(uint32,bool)",123,true)는 상대측 함수 baz()를 저수준으로 호출하고 있다.

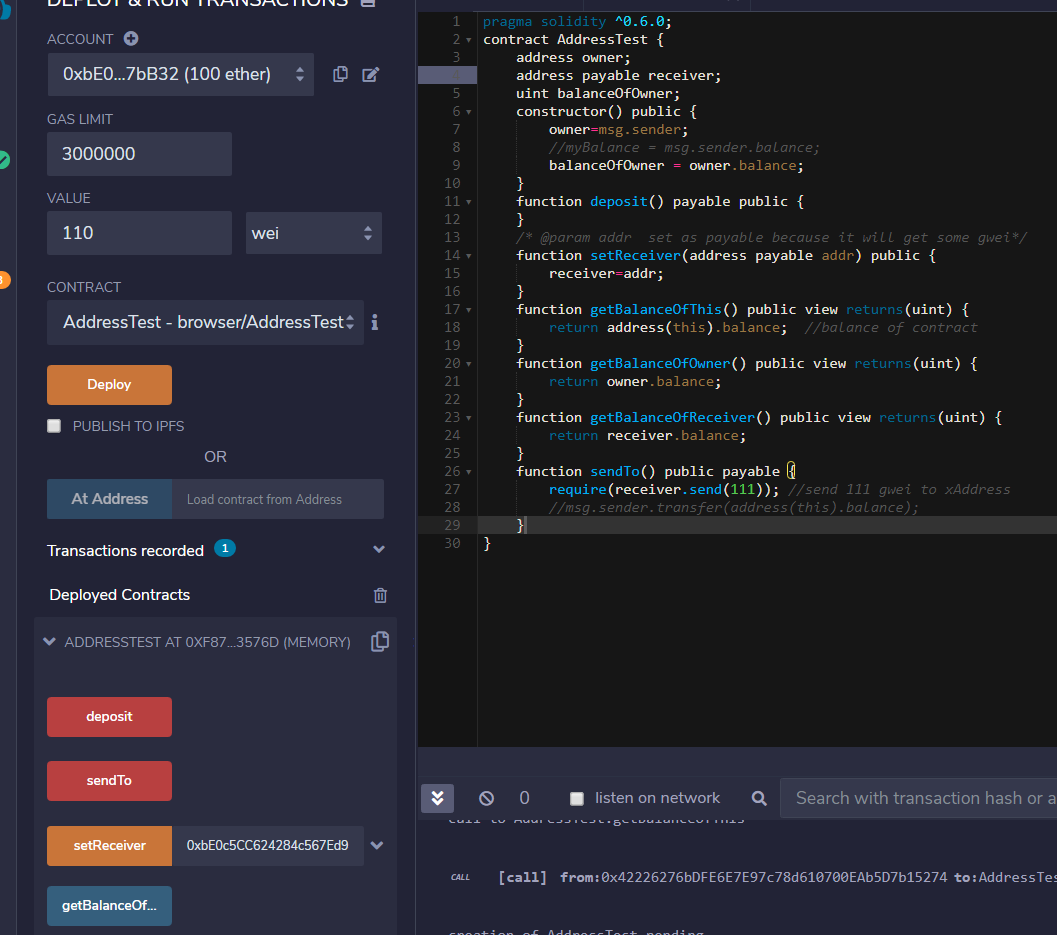
(bool success, bytes memory data) = <address>.call{gas:10000, value: 20}(  
 abi.encodeWithSignature("baz(uint32,bool)",123,true)  
);

## 실습문제: 송금

send, transfer, call 함수를 사용해서 송금하는 프로그램을 작성하자. REMIX에서 코딩하고, 버튼 테스트를 실행할 때는 다음 순서에 따라 해보자.

* 1. 입금할 계정을 설정해야 한다. setReceiver()에 주소를 넣어 주어야 하는데, address는 따옴표 없이 쓴다. 따옴표로 묶으면 문자열 타입으로 인식하게 된다.
  2. send(), transfer, callValue()를 성공적으로 실행하기 위해서는 사전에 deposit()을 호출해서 충분한 금액을 입금해야 한다.

deposit() 함수는 코드가 없다. 잘못한 것 아닌가 하고 착각할 수 있을 수도 있다. 놀랍게도 아무런 코드 없이 입금이 된다. 그 이유는 payable로 선언되어 있어 그렇다. 그러나 잊지 말아야 한다. 호출할 때는 value에 금액을 넣어야 한다 (버튼 테스트할 때, REMIX 윗 부분에서 발견할 수 있다). 출금하기 위해서는 충분히 입금을 해놓는다.

그림: 주소는 따옴표 없이 입력하고 value에 송금액을 적어준다 

* 1. send(), transfer, callValue()를 하나씩 실행해본다.
  + 각 send, transfer, call 함수를 통해 출금하면서 잔고가 그만큼 감소하는지 확인하자.
  + call() 함수에서 gas를 적게 10으로 설정해도 출금이 되는지 확인하자.

[파일명: src/AddressTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract AddressTest {  
 address owner;  
 address payable receiver; // 수신 계정;  
 uint balanceOfOwner;  
 constructor() { // NO! constructor() public {  
 owner=msg.sender;  
 balanceOfOwner = owner.balance; // 잔고 설정  
 }  
 function deposit() payable public {  
 // payable로 선언해서, 코드가 없어도 입금이 된다.  
 }  
 /\* @param 수신 계정을 설정\*/  
 function setReceiver(address payable addr) public {  
 receiver=addr;  
 }  
 function getReceiver() view public returns(address) {  
 return receiver;  
 }  
 function getBalanceOfThis() public view returns(uint) {  
 return address(this).balance; // 컨트랙 자체의 잔고  
 }  
 function getBalanceOfOwner() public view returns(uint) {  
 return owner.balance;  
 }  
 function getBalanceOfReceiver() public view returns(uint) {  
 return receiver.balance;  
 }  
 function send() public payable {  
 require(receiver.send(111)); // 111 wei를 receiver로 송금. require()로 예외처리.  
 }  
 function transfer() public payable {  
 //if !(receiver.transfer(address(this).balance))  
 receiver.transfer(11111); // receiver로 송금  
 }  
 function callValue() public payable {  
 //receiver.call.value(11111)(""); // 과거 문법. 이렇게 작성하지 않는다.  
 (bool success, ) = receiver.call{value: 11111}(""); //gas를 적어주지 않으면 실제 사용량 지급  
 require(success, "transfer call failed.");  
 (success, ) = receiver.call{gas: 10, value: 11111}(""); //gas를 적게 설정하고 테스트  
 require(success, "transfer call failed with gas 10.");  
 }  
}

다음과 같이 컴파일 해보자.

pjt\_dir> solc-windows.exe src\AddressTest.sol

## 2.10 리터럴

변수가 값을 가지려면 할당연산자 다음에 리터럴 상수(literals)를 저장한다. 즉 변수는 명칭이고, 리터럴 상수는 변수에 저장하는 값이다.

* 정수는 소수점이 없는 숫자이고 1, 10, -1, -10 등의 예를 들 수 있다. 또는 16진수 0x14 (십진수 20)도 사용할 수 있다. 16진수는 0x를 붙여서 표현한다. 단 바이트를 제한한 경우, 예를 들어 int8는 범위에 맞게 -127~128 사이의 정수를 입력한다.
* string은 작은 따옴표('') 또는 큰 따옴표("")로 묶인 문자이다. 예를 들어 "jsl", 'hello' 등은 string이다.
* 주소는 객체로서 20바이트 16진수(0x로 시작)로 표현되는 20바이트 즉 40자리 숫자이다. 예를 들면, 0x307A2Db4424E0F651cfa1Dd7C6C418CDf8f5675b는 주소의 예이다.
* bytes는 0x로 시작하는 16진수이다. 따옴표를 해도 또는 하지 않아도 된다. 예: 0x3720686f6e676a69
* 소수점은 1.1 3.14 등이 예이다. 아직 구현되지 않아 값을 할당할 수 없다.
* 배열은 다른 프로그래밍 언어와 유사하게 값을 할당할 수 있다. 단 Solidity에서는 대괄호('['와 ']')를 이용해서 값을 지정한다. 예: [15, 25, 35]

## 2.11 인코딩

인코딩은 사람이 사용하는 문자나 기호들을 블록체인에서 이용할 수 있는 머신코드인 32바이트로 변환하는 것을 말한다. 20바이트의 address, uint256, bytes32 등 크기가 정해진 타입은 32바이트로 변환되지만 string, array와 같은 크기가 정해지지 않은 동적타입은 지정한 방식으로 인코딩된다.

인코딩 방법은 abi.encode() 내장함수를 사용하면 된다. 이 함수는 변환해서 32바이트로 채워지지 않으면 빈 공간을 0으로 채우는 패딩(padding) 작업을 한다.

패딩 작업을 하지 않으려면 abi.encodePacked() 함수를 사용한다.

## 실습문제: 송금

REMIX 에디터에서 코드를 작성하고, 각 데이터 타입이 인코딩하면 어떻게 변환되는지 테스트해보자.

하나씩 int, address, string, bytes에 대해 abi.encode(), abi.encodePacked() 함수를 작성하자.

[파일명: src/AbiTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract AbiTest {  
 function encodeUint(uint256 \_uint) public pure returns(bytes memory) {  
 return abi.encode(\_uint);  
 }  
 function encodeAddress(address addr) public pure returns(bytes memory) {  
 return abi.encode(addr);  
 }  
 function encodeString(string memory s) public pure returns(bytes memory) {  
 return abi.encode(s);  
 }  
 function encodePackedString(string memory s) public pure returns(bytes memory) {  
 return abi.encodePacked(s);  
 }  
 function encodePackedStringConcat(string memory s1, string memory s2) public pure returns(bytes memory) {  
 return abi.encodePacked(string.concat(s1,s2));  
 }  
 function encodeBytes(bytes memory b) public pure returns(bytes memory) {  
 return abi.encode(b);  
 }  
 function encodePackedBytes(bytes memory b) public pure returns(bytes memory) {  
 return abi.encodePacked(b);  
 }  
 function encodePackedBytes2String(bytes memory b) public pure returns(string memory) {  
 return string(abi.encodePacked(b));  
 }  
}

다음을 버튼 테스트해서 확인해보자.

* 계정주소를 encode("0x81b928274EC5c4F366fEb3572252A327474C921d") 인코딩하면, "0x00000000000000000000000081b928274ec5c4f366feb3572252a327474c921d"이 된다.
* 문자열을 인코딩하면 encode("0xab23") 결과는 다음과 같다.

0x0000000000000000000000000000000000000000000000000000000000000020 처음부터 32바이트 지나면 데이터가 있다는 의미  
 0000000000000000000000000000000000000000000000000000000000000002 2글자  
 ab23000000000000000000000000000000000000000000000000000000000000 ab 23

* 바이트를 encodePacked("0xab23") 하면, 그대로 "0xab23"
* 문자열을 encodePacked("hello") 하면, 패딩(padding) 없이 그대로 "0x68656c6c6f". 패딩이란 자릿수에 맞춰 0 등으로 채워넣는 것을 의미한다.
* hello, world 문자열을 encodePackedString("hello", "world")하면 0x68656c6c6f20776f726c64 (앞에서 설명한 것처럼 "hello world"를 아스키(ASCII) 코드 형태로 변환한 것)
* 정수 encode(345) 하면, "0x0000000000000000000000000000000000000000000000000000000000000159"
* 16진수를 string(abi.encodePacked("0x68656c6c6f")) 하면 원 문자열 "hello" 를 반환한다.

# 3. 데이터의 저장

## 3.1 타입 구분

데이터 타입은 다음 두 가지로 구분할 수 있다.

* 값을 가지고 있는 자료형(value type)
* 참조를 가지고 있는 자료형(reference type)

### 값으로 저장되는 자료형

데이터 그 **자체의 값**으로 저장되는 자료형이다. 다른 변수에 할당되거나 함수 인자로 전달될 경우, 값의 복사본이 되어 전달되는 **값 전달(pass by value)**의 특성을 가진다. bool, int/uint, address, bytes (1~32 바이트), enum이 해당된다. int, bytes (1~32 바이트)는 최대 **32바이트**로 그 값을 저장할 수 있다.

### 참조가 저장되는 타입

**32바이트를 초과할 수 있는 크기의 데이터**를 말하며, 그 저장 주소를 통해 사용하는 타입이다. 변수에 할당되거나 매개변수로 전달될 경우, 그 저장 주소가 복사되어 쓰이기 때문에 원본 데이터가 수정되면 따라서 변경된다. 참조 자료형은 bytes, array, string, Struct, mapping과 같이 실제 값이 저장되어 있는 주소를 가리키는 참조를 말한다. 이들 데이터는 32바이트 이상의 크기를 가질 수 있고, 저장 장소를 정의해 줄 수 있다.

## 4.2 저장장소 구분

Solidity는 데이터를 저장하는 장소를 다음과 같이 구분한다.

### storage는 블록체인에 저장한다.

하드디스크에 저장이 되는 것처럼, 일단 storage에 저장이 되면 영원히 저장된다. 따라서 함수 밖에서 사용된다. 컨트랙의 상태 변수는 기본 값이 storage이며, 모든 함수에서 값을 볼 수 있다. 함수가 실행된 후에도 사라지지 않는다. 32바이트 길이의 키-값(key-value) 형식으로 저장되고 계속 유지되므로, 비용이 다른 저장 타입에 비해 높다.

contract IntBoolTest {  
 bool married = true; storage에 저장된다. bool storage married = true;라고 하지 않는다.  
 uint256 xAge = 22; storage에 저장된다. uint256 storage xAge = 22;라고 하지 않는다.  
}

멤버 변수 (또는 상태 변수라고 함)에 지역 변수를 저장하면, 복사본이 저장된다. 복사한 후 지역 변수의 값이 변경되어도, 멤버 변수는 기존 값이 유지된다. 참조값이라 할지라도 참조가 복사되지 않기 때문에 멤버 변수의 값이 유지된다.

[파일명: src/CopyOfStorage.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
   
contract CopyOfStorage {  
 uint stateInt = 111; //storage  
 string stateStr = "hello";  
   
 function copyLocalToStateVar() public {   
 uint localInt = 123;  
 string memory localStr = "hello world";  
 stateInt = localInt; // 복사 후 stateInt = 123  
 stateStr = localStr;   
 localInt = 456; // 456으로 변경하여도 stateInt = 123으로 유지  
 localStr = "hello outer world"; // 참조타입을 변경하여도 유지  
 }  
 function getStateInt() public view returns(uint) { return stateInt; } // 123이 출력된다.  
 function getStateStr() public view returns(string memory ) { return stateStr; } // hello world 출력.  
}

### 메모리는 블록체인에 저장되지 않고, 임시로 저장된다.

memory는 단기간 저장할 때 사용된다. 함수 내에서 함수에 있는 코드가 실행되는 동안 사용되고, 함수를 벗어나면 지워진다. **값을 가지고 있는 자료형(value type)**은 지역 변수, 함수의 인자 또는 반환 값도 memory에 저장되지만 memory 수식어를 붙여주지 않는다.

단, struct, array, mapping과 같은 참조를 가지는 자료형(reference type)은 지역 변수라 하더라도 memory 수식어를 붙인다. memory를 붙이지 않으면 기본으로 디폴트(default)인 공간(storage)에 저장되기 때문에 오류가 발생한다. 이러한 자료형들은 storage라고 명시적으로 적어도 안된다.

function getLenOfString() pure public returns(uint) {   
 int i; // 지역변수 i는 메모리에 저장이 된다.  
 string memory nameLocal="jslLocal"; // 저장구분을 안 적거나, storage라고 하면 오류. memory라고 적어야 함  
}

### 스택은 지역변수의 값을 저장한다.

stack은 지역변수 (참조형 타입 제외)의 값을 저장한다. 1,024 수준까지 저장할 수 있고, 그 이상은 예외처리 된다. 비용은 memory와 동일하다.

### callData는 함수의 매개변수를 저장한다.

calldata는 **외부**에서 함수를 호출할 때 (extern으로 선언된 함수), 사용하는 함수 시그니처(signature. 함수의 헤더(header)라고도 부르며, 이름과 매개 변수, 반환값 등을 나타낸다)와 매개 변수를 calldata로 저장한다. 앞서 ABI 명세에서 설명한 것처럼 1) function selector부호값과 2) 인자 부호값를 합쳐서 data 필드에 적은 "0xcdcd...00001"을 말한다.

web3.eth.sendTransaction({  
 from: ...,  
 data: "0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000000000000000000000000000000000001",  
 gas: ...}  
)

위 설명이 복잡할 수 있으니 요약해보자.

* 상태변수는 storage에 저장된다.
* 함수에 대해서 말하면, value 타입의 지역변수, 함수인자/반환은 memory를 기본 default로 사용한다. 단, 함수에서의 참조타입 (struct, array, mapping, string)은 기본이 storage이고, memory를 사용하려면 선언이 필요하다.
* 조금 복잡하다고 걱정하지 말자. 컴파일러가 이러한 오류를 빼놓지 않고 지적하니까 수정하면 된다.

# 5. 가시성

지역 변수는 선언된 함수 내부에서만 사용할 수 있다. 반면에 멤버 변수는 사용할 수 있는 범위를 제어할 수 있다.

* 가시성을 적어주지 않으면 **기본 값은 internal**로 정의된다. internal은 블록체인 내부에서만 사용할 수 있고, 상속하면 자식이 물려받을 수 있다는 의미이다. library의 함수는 internal로 선언할 수 있다.
* external은 internal의 반대, 외부에서 호출하는 경우만 허용된다. 내부에서는 사용할 수 없지만, this 명령어를 통해서 이런 식으로 this.f() 호출하면 가능하다. fallback 함수는 외부에서만 호출할 수 있도록 extern으로 선언한다.
* 누구나 internal 또는 external 가리지 않고 모두 사용하려면 public으로 선언하면 된다. 객체 지향 public과 동일한 의미이다.
* private은 블록체인 내부에서 컨트랙 자신만 사용할 수 있고, 객체 지향 private과 동일한 의미이다.

|  |  |  |
| --- | --- | --- |
| 구분 | 설명 | default |
| public | 누구나 internal 또는 external 모두 사용 | no |
| private | 컨트랙 자신만 사용 | no |
| internal | 현재 컨트랙 내부에서만 또는 상속의 경우에 사용하는 경우. java의 protected와 비슷한 의미 | yes |
| external | 외부에서 호출하는 경우만 허용 | no |

# 6. 연산자

Solidity의 연산자는 다른 프로그래밍 언어에서 사용되는 연산자와 비슷하다. Solidity에서는 다음 연산자들을 제공한다.

* 산술연산자: + - \* / % (modulus) ++ -- \*\* (exponentiation)
* 비교연산자: == != > < >= <=
* 논리연산자: && || ! (부정)
* 비트연산자: & | ^ ~
* 대입연산자: = += -= \*= /= %= (x = x % y는 x %= y)
* 삼항연산자: ?:

## 실습문제: 주소 계정에서 입출금

앞서 geth 단말에서 계정 간에 서로 송금해 본 적이 있다. 이번에는 컨트랙에서 송금하는 프로그램을 개발해 보자. 입금 deposit(), 출금 withdraw(), 계좌이체 transferTo() 함수를 만들어보자.

코딩하면서, 은행을 경유하지 않고도 송금이 이루어지는 과정을 이해해 보자. 은행없이 송금하는 경우, 서로 어떻게 신뢰할 수 있는지 실패하는 경우에는 어떻게 복구할 수 있는지 생각해 보자.

### 단계 1: 컨트랙 개발

REMIX에서 코드를 작성하자.

코딩을 완성하고 나서, 버튼 테스트를 통해 입금 deposit()하고, 그 후 출금 withdraw() 및 계좌이체 forwardTo()를 해보자. 그러면서 잔고 함수를 실행하면서 입출금이 성공적으로 이루어지는지 확인하자.

[파일명: src/MyBank.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract MyBank {  
 address owner; //address payable owner;  
 uint balance;  
 constructor() { //constructor() public {  
 owner = msg.sender;  
 balance = address(this).balance;  
 }  
 function deposit(uint amount) public payable {  
 require(msg.value == amount);  
 balance += amount;  
 }  
 function withdraw(uint amount) public payable {  
 balance -= amount; // 송금 전 감액  
 payable(owner).transfer(amount); // 이전 owner.transfer(amount)  
 }  
 function transferTo(address payable receiver, uint amount) public payable {  
 balance -= amount; // 송금 전 감액  
 receiver.transfer(amount);  
 }  
 function getBalance() public view returns (uint) {  
 return balance;  
 }  
 function getBalanceOfThis() public view returns (uint) {  
 return address(this).balance;  
 }  
 function getBalanceOfOwner() public view returns (uint) {  
 return owner.balance;  
 }  
}

|  |  |
| --- | --- |
| 줄 | 설명 |
| 11 ~ 14 | 전송자의 msg.value에 입력된 ether를 contract으로 입금하는 함수이다. 주의할 점은 함수인자 amount는 ether가 아니다. 실제 입금되는 금액은 msg.value이다. 단 msg.value와 amount가 동일해야 입금이 실행된다. 그리고 **payable**로 명시해 주어야 한다. |
| 15 ~ 18 | owner 계정으로 컨트랙 자신의 잔고, address(this).balance에서 ammount 만큼을 이체한다. msg.sender는 매번 메시지를 전송할 때마다 변경되는 것이 아니라, 배포하는 시점의 전송자 즉 프로그램을 배포하는 권한을 가진 사람만이 (정확히 말하면 개인키) owner가 된다. |
| 19 ~ 22 | 매개변수로 입력된 주소로 컨트랙 자신의 잔고에서 amount만큼 이체한다. |
| 23 ~ 31 | 컨트랙 자신의 잔고, address(this).balance 및 onwer의 잔고를 조회한다. |

### 단계 2: 컴파일

컴파일 해보자. 디렉토리 구분자를 정슬래시를 사용하고 있지만, 역슬래시도 문제가 없다.

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

### 단계 3: 컨트랙 배포

배포 코드는 앞서 작성한 것과 많이 다르지 않다. 컴파일 결과가 쓰인 자바스크립트 파일 MyBank.json을 읽어서, 그로부터 abi, bin을 읽고 컨트랙 객체를 생성하고, 배포하고 있다.

[파일명: src/MyBankDeployAbiBinFromFile.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./MyBank.json'); // 출력파일을 읽어온다  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // ['src/MyBank.sol:MyBank']를 읽어온다  
console.log("- contract name: ", contractName);  
\_abi=\_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: 364124}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

위 코드를 실행해서, 배포를 완성해보자. 맨 마지막에 출력되는 컨트랙 주소를 다음 단계로 넘겨 활용하게 된다.

pjt\_dir> node src/MyBankDeployAbiBinFromFile.js  
  
- contract name: [ 'src/MyBank.sol:MyBank' ]  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x2525f0be96865507f9e4f892c525e4c8b5385694248b8b412cea80cffb22d632  
---> The contract deployed to: 0x81b928274EC5c4F366fEb3572252A327474C921d

### 단계 4: 사용

deposit()함수의 value:1111 필드를 채워주면 msg.value로 전달이 된다. 이 때 함수의 인자도 동일하게 1111 Wei를 넣어준다.

myBank.deposit(1111,{from:web3.eth.accounts[0],gas:80000,value:1111})

마이닝을 하고나면 잔고가 1111이 된다. 컨트랙의 deposit() 함수에서 잔고에 합산을 하지 않아도 된다.

자신의 잔고에서 1111만큼 빠져나간 것을 확인해보자.

geth> eth.getBalance(eth.accounts[0]);  
94999999999999998889

[파일명: src/MyBankUse.js]  
var Web3=require('web3');  
var \_abiBinJson = require('./MyBank.json'); //importing a javascript file  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
contractName=Object.keys(\_abiBinJson.contracts); // reading ['src/MyBank.sol:MyBank']  
//console.log("- contract name: ", contractName); //or console.log(contractName[0]);  
\_abi=\_abiBinJson.contracts[contractName].abi;  
\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); //JSON parsing needed!!  
//\_bin=\_abiBinJson.contracts[contractName].bin;  
  
var myBank = new web3.eth.Contract(\_abiArray,"0x81b928274EC5c4F366fEb3572252A327474C921d");  
  
async function doIt() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Call from: " + accounts[0]);  
 myBank.methods.getBalance().call().then(console.log);  
 myBank.methods.deposit(1111).send({from:accounts[0],gas:80000,value:1111});  
 myBank.methods.getBalance().call().then(console.log);  
}  
  
doIt()

send, call 함수가 섞여 있어서 프로그램을 나누어서 실행해야 하지만, 편의상 그대로 2회 실행을 해보자. 1111을 deposit() 입금하면 잔고에 반영이 되는 것을 볼 수 있다. 또한 withdraw() 출금하면 잔고가 0이 되면서 전송자의 잔고가 증가한다.

pjt\_dir> node src/MyBankUse.js  
  
Call from: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
1111  
1111

# 7. 형변환

## 정수의 크기를 변환

형변환 하는 경우, 자료형의 크기에 주의해야 한다. 예를 들어 int8에서 int로 변환하려면, 크기를 먼저 확인하자. int는 256비트이므로 오류없이 변환이 된다. 아닐 경우, 괄호 안에 써주어야 한다

int8 i8=20;  
uint i256=int(i8); //8비트를 256비트로 변환한다. 괄호 안에 적어준다.

## 정수를 크기가 같은 byte32로 변환

uint, bytes32는 서로 타입은 다르지만 크기가 동일하므로 변환할 수 있다.

uint x=20;  
bytes32(x); //uint를 bytes32로 변환

bytes는 int로 형변환은 가능하지 않다. 예를 들어, bytes4는 uint32와 크기가 같고 형변환이 가능하다. 그러나 크기가 같더라도 int32로 형변환은 허용되지 않는다.

bytes4 b4=0x68656c6c;  
int32(b4); //크기가 같더라도 int로 형변환 오류  
uint32(b4); //크기가 같은 uint로 형변환 가능

## bytes 간의 변환할 경우 크기에 주의

0x68656c6c은 16진수, 8자리이다. 한 자리에는 0,1,2...D,E,F까지 가지를 표현할 수 있고, 4비트가 소요된다.

이 데이터 bytes4를 bytes2로 변환해보자. 4 바이트 숫자를 2 바이트 공간에 저장하므로 공간이 부족하다. 이럴 때 **오버플로우는 위 4자리부터 잘려 나가게 된다**. 따라서 0x6865가 저장된다.

bytes4 b4 = 0x68656c6c;  
bytes2(b4); //오버플로우가 발생하고 0x6865가 남는다

## string을 bytes로 변환

string을 bytes로 변환하는 경우:

string s = "hello";  
bytes b = bytes(s); //0x68656c6c6f

앞서 설명을 기억한다면, string(abi.encodePacked("0x68656c6c6f")) 하면 "hello"이다. abi 인코딩을 하면 bytes가 되고, 이를 string으로 형변환하는 것이다.

|  |  |
| --- | --- |
| hex | char |
| 68 | h |
| 65 | e |
| 6c | l |
| 6c | l |
| 6f | o |

## payable 형변환

address payable에서 address로의 형변환은 가능하지만, 반대는 허용되지 않는다. address -> uint160 -> address payable로 20바이트의 정수로 변환한 후 하도록 한다.

## 실습문제: 형변환

REMIX에서 코딩하고, 다음 함수에 대해 버튼 테스트를 해서 확인해보자.

* convertInt8ToInt(): int8을 int로 크기가 큰 쪽으로 변환하는 경우이다 (가능하다).
* convertIntToBytes32(): 서로 크기가 동일한 변환한다 (가능하다).
* convertInt64ToBytes8(): int64를 bytes8로 변환한다 (동일한 크기이므로 가능하다).
* convertIntToBytes(): int를 bytes로 변환하는 함수이다. uint를 bytes로 바로 형변환을 할 수 없다. uint는 256비트, 즉 32바이트라서 bytes32로 변환하고, 이를 bytes로 변환해야 한다. 이 경우 바이트마다 옮겨주어야 한다.
* convertStringToBytes(): 문자열을 bytes로 형변환한다 (가능하다)
* convertBytes4ToBytes2(): bytes4를 bytes2로 형변환한다. 크기를 줄이는 경우이므로 오버플로우가 발생한다.
* convertBytes4ToInt32(): bytes4를 int32로 형변환한다 (동일한 크기이므로 가능하다)

[파일명: src/DataConversionTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract DataConversionTest {  
 function convertInt8ToInt() pure public returns(int) {  
 int8 i8=20;  
 return int(i8); //20  
 }  
 function convertIntToBytes32() pure public returns(bytes32) {  
 uint x=20;  
 //0x0000000000000000000000000000000000000000000000000000000000000014  
 return bytes32(x);  
 }  
 function convertInt64ToBytes8() pure public returns(bytes8) {  
 uint64 x=10;  
 return bytes8(x); //0x000000000000000a  
 }  
 function convertIntToBytes() pure public returns(bytes memory) {  
 // can not convert uint -> bytes  
 // convert uint -> bytes32 -> bytes  
 uint x = 20;  
 bytes32 y = bytes32(x); //uint <=> uint256  
 // can not convert from bytes32 -> bytes;  
 bytes memory z = new bytes(32);  
 for (uint i=0; i < 32; i++) {  
 z[i] = y[i];  
 }  
 //0x0000000000000000000000000000000000000000000000000000000000000014  
 return z;  
 }  
 function convertStringToBytes() pure public returns(bytes memory) {  
 string memory s="hello";  
 return bytes(s); //0x68656c6c6f  
 }  
 function convertBytes4ToBytes2() pure public returns(bytes2) {  
 bytes4 b4=0x68656c6c;  
 return bytes2(b4); //0x6865  
 }  
 function convertBytes4ToInt32() pure public returns(uint32) {   
 bytes4 b4=0x68656c6c;  
 //1751477356 = 68656C6C hex = (6 × 16^7) + (8 × 16^6) + ... + (6 × 16^1) + (12 × 16^0)   
 return uint32(b4); //1751477356  
 }  
}

컴파일 해보자.

pjt\_dir> solc-windows.exe src/DataConversionTest.sol

# 8. 전역 변수

Solidity에서는 화폐나 시간의 단위를 숫자 상수 뒤에 붙여서 사용할 수 있다. 그러나 변수 뒤에는 붙일 수 없다.

숫자 뒤에 붙여 쓸 수 있는 화폐 단위는 wei, gwei, ether가 있다. wei는 1, 1 gwei는 wei이고 1 ether는 wei이다.

1 ether == 1000000000000000000 wei  
1 gwei == 1000000000 wei

숫자 뒤에 붙는 시간 단위는 weeks, days, hours, minutes, seconds이다. \* years는 윤년인 경우에 365일이 아닐 수 있으므로, 0.5.0버전 이후부터 삭제되었다. \* now도 0.7.0 버전 이후부터 제거되었다. 현재 시각을 나타내는 block.timestamp을 사용해야 한다.

1 minutes == 60 seconds //1칸 띄워서 시간 단위를 붙여 쓴다.  
1 hours == 60 minutes  
1 days == 24 hours  
1 weeks = 7 days  
uint mytime=block.timestamp //현재 시간을 저장. now는 더 이상 쓰지 못한다.

block은 블록의 속성들을 나타낸다.

* block.coinbase: 현재 블록 마이너의 주소
* block.difficulty: 현재 블록의 난이도
* block.gaslimit: 현재 블록의 gaslimit
* block.number: 현재 블록 수
* block.blockhash: 현재 블록 해쉬 값
* block.timestamp: 현재 블록 타임스탬프 epoch (1970년 1월 1일 0시) 이후 지나간 초, uint256 타입이다.

tx는 컨트랙을 호출하는 트랜잭션 관련 정보들을 가지고 있다.

* tx.origin 트랜잭션에 사인한 계정
* tx.gasprice 트랜잭션 호출자가 명시한 gas price

msg는 컨트랙의 함수를 호출한 전송 관련 정보를 포함한다.

* msg.data: call 데이터 (bytes 값으로 표현)
* msg.gas: gas 분량
* msg.sender: 현재 msg를 전달하는 측 주소
* msg.value: 지급되는 ether (단위는 wei)

얼핏 보면, tx.origin과 msg.sender 구별을 혼돈할 수 있다. 최초사용자 U1 -> 컨트랙 C1 -> 컨트랙 C2의 순서대로 호출된다고 하자.

* C2에서 msg.sender는 바로 직전의 호출자C1을 말한다. 즉 컨트랙도 msg.sender가 될 수 있다.
* 반면에 tx.origin은 U1을 의미한다. 최초로 메시지를 발생한 측을 말한다.

## 실습문제

컨트랙 Order에서 Customer를 호출한다고 하자. getTxOriginMsgSender() 함수를 작성하여, tx.sender, msg.sender를 식별하는 코드를 작성해보자.

REMIX에서 함수의 버튼 테스트를 해서 출력을 확인해보자.

주소 0x69e9a...0c102에서 Order의 getTxOriginMsgSender() 함수를 호출하면:

* Customer의 tx.origin 은 0x69e9a...0c102 (Order 컨트랙 함수호출자인 msg.sender의 주소와 동일)
* Customer의 msg.sender는 0x0b878...E7cD4, 즉 Order 컨트랙 배포주소와 동일하다.

[파일명: src/TxOriginMsgSenderTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Customer {  
 function getTxOriginMsgSender() view public returns(address, address) {  
 return(tx.origin, msg.sender);  
 }  
}  
  
contract Order {  
 Customer c;  
 constructor() { //constructor() public {  
 c = new Customer();  
 }  
 function getTxOriginMsgSender() view public returns(address, address) {  
 return c.getTxOriginMsgSender();  
 }  
}

컴파일해서 오류가 발생하는지 확인한다.

pjt\_dir> solc-windows.exe src/TxOriginMsgSenderTest.sol

## 실습문제: 전역변수

전역변수를 반환하는 함수들을 만들어본다. 코드가 한 줄 정도로 매우 간단한 함수이다.

* getEther(): 1 ether를 반환하지만, wei로 출력하고 있으니, 단위를 확인한다.
* getDays(): 현재 시점의 타임스탬프, 하루의 초 (24시간 x 3600초)를 출력한다.
* 그 외 msg, block 관련 값을 출력하는 함수를 구현한다.

[파일명: src/GlobalVarsTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract GlobalVarsTest {  
 function getEther() pure public returns(uint) {  
 return 1 ether; //1000000000000000000  
 }  
 function getDays() view public returns(uint,uint) {  
 //require(block.timestamp==now); //now deprecated  
 return (block.timestamp, 1 days); // 1558816133 86400 ??web3? 1 days?  
 }  
 //public --> internal, to remove 'payable'  
 function getMsgValue() view internal returns(uint) {  
 return msg.value;  
 }  
 function getMsgSender() view public returns(address) {  
 return msg.sender;  
 }  
 function getCoinbase() view public returns(address) {  
 return block.coinbase;  
 }  
 function getBlockNumber() view public returns(uint) {  
 return block.number;  
 }  
 function getBlockTimeStamp() view public returns(uint) {  
 return block.timestamp;  
 }  
}

REMIX에서 버튼 테스트까지 확인했다면, 당연히 오류가 없겠지만 컴파일해보자.

pjt\_dir> solc-windows.exe src/GlobalVarsTest.sol

# 9. Opcode 읽어보기

Solidity 뿐만 아니라 프로그래밍을 하면서 오류를 만날 수 밖에 없다. 오류는 크게 나누면 세 가지로 분류할 수 있다.

* 1. 문법적으로 맞지 않음 (Syntax Errors),
  2. 논리적으로 부적합 (Logic Errors),
  3. 문법이나 논리적 오류가 없지만, 실행하면서 오류가 발생 (Runtime Errors).

이러한 오류는 컴파일하거나 테스트 과정에서 변수값 등을 확인하면서 코드를 수정해 나가는 방법을 택한다. 우리도 지금까지 이 정도 수준으로 디버깅을 해오고 있다.

난이도가 높지만 Opcode를 실행하면서 오류가 발생하는지 확인할 수도 있다. 그렇게 하려면 Opcode, 즉 기계어 형태의 오브젝트 코드의 이해가 필요하고, 상당한 노력이 필요해서 권하고 싶지는 않다. 이런 방식으로 디버깅을 한다고 하면 아마 컴퓨터를 너무 사랑하는 수준이 아닐까 한다.

여기서는 Opcode와 니모닉(mnemonic)이 이런 것이구나 살펴보고 REMIX에서 디버깅을 Opcode 수준으로 할 수 있다는 정도로만 이해하자. 니모닉은 숫자로 되어 있는 Opcode를 사람이 좀 더 이해하기 쉬운 한 단어 정도의 문자열로 표현하는 것으로 어셈블리 언어라고 생각하면 된다.

앞서 MyBank.sol에서 생성된 바이트코드는 다음과 같다.

608060405234801561001057600080fd5b50336000806101000a81548173ffffffffffffffffffffffffffffffffffffffff021916908373ffffffffffffffffffffffffffffffffffffffff16021790555047600181905550610302806100676000396000f3fe6080604052600436106100555760003560e01c806312065fe01461005a5780632ccb1b30146100855780632e1a7d4d146100...생략...509056fea2646970667358221220febeb9d97631e86ae2847fb122913ad5400485aa2ea08f75f0f1caa649a1ce8164736f6c63430006010033

난해해 보이는 이러한 일련의 바이트코드를 Opcode로 변환할 수 있는데, 앞부터 읽어보면 된다. 바이트 코드는 Opcode와 Opcode에서 사용하는 데이터로 구성된다.

바이트코드는 "6080604052..."로 시작하고 있다. 눈여겨 보았다면, 보통 컨트랙을 컴파일하면 이러한 바이트코드로 동일하게 시작하고 있다. 이 부분에서 첫 번째 바이트는 Opcode를 나타내고, 두 번째 바이트는 Opcode 명령에서 사용하는 데이터이다.

Opcode는 대응되는 니모닉으로 표현할 수 있는데 바이트 코드를 니모닉 형태로 변환하면 "PUSH1 0x80 PUSH1 0x40 MSTORE..."이다. 이런 변환은 개발자가 직접하기에는 매우 복잡하다. REMIX에서 컴파일을하면 바이트코드와 함께 니모닉을 보여주기 때문에 쉽게 알 수 있다.

Opcode를 해석해보자.

* 6080은 PUSH OX80, PUSH1는 stack에 넣는 명령이다 (POP은 데이터를 stack에서 꺼내는 명령이다). PUSH1은 1바이트의 데이터를 사용한다.
* 6040은 PUSH 0x40
* 52는 mstore이다. mstore는 데이터 2개가 필요하다. mstore는 RAM memory에 저장하는 Opcode이다 (참고로 disk storage에 저장하는 Opcode는 sstore이다). 사용하는 예로 덧셈 결과를 메모리에 저장할 때는 mstore(0x10, add(x,y))라고 해준다.

|  |  |  |
| --- | --- | --- |
| 바이트코드 | Opcode | 설명 |
| 6080 | PUSH1 80 | stack에 0x80 넣음. |
| 6040 | PUSH1 40 | stack에 0x40 넣음. |
| 52 | MSTORE |  |

종합적으로 바이트 코드 "6080604052..."는 Opcode "PUSH1 0x80 PUSH1 0x40 MSTORE..."이고, 실행 결과는 메모리 0x80의 공간을 할당하고, 중간인0x40으로 포인터를 이동. 즉 64바이트 scratch space (사용자 데이터를 저장하는 임시 저장 공간)과 temporary memory storage (단기 기억 공간)으로 분할하게 된다.

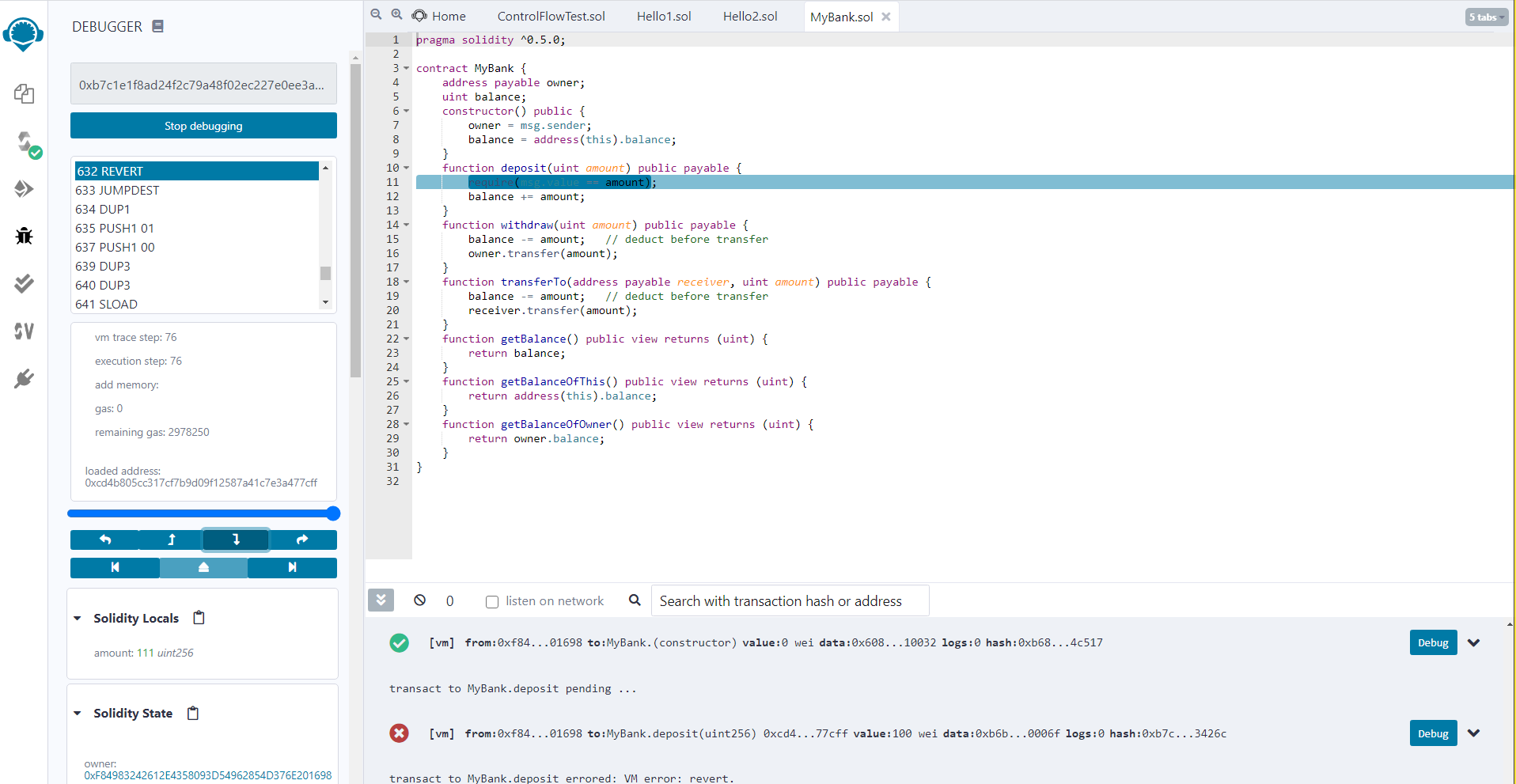
바이트코드 별 Opcode가 있는데, 예를 들면 이렇다. 0x00는 STOP, 0x01은 ADD, 0x02 MUL, ... , 0x52는 MSTORE, 0x55는 SSTORE이다. 이 모든 코드를 어떻게 기억할 필요는 없다. Opcode는 "Ethereum Virtual Machine Opcodes" (https://www.ethervm.io/) 페이지에 상세한 정보가 나와있다. 또한 opcode는 당연히 실행하면 gas 비용이 발생하게 되는데, "eip 150 gas cost"라고 검색하면 구글 스프레드쉬트 문서를 찾을수 있고 상세하게 정리되어 있다.

컨트랙의 함수버튼을 눌러 실행하고 나면, REMIX의 우하단에 있는 콘솔을 보자. 함수의 맨 끝에 하늘색 디버그 버튼이 보인다. 이를 눌러 디버그 모드로 들어가 보자. 그러면 좌측에 디버거가 뜬다. 위에서 설명한 opcode도 보일 것이다.

일부러 오류를 발생시켜 보자. msg.value와 deposit(uint amount)의 amount를 다르게 입력하면 require() 오류가 발생한다.

함수를 추적할 수 있는 step over, step into 등의 버튼이 만들어져 있다.. 다른 언어의 디버거에서 제공되는 기능과 비슷하다. 예를 들어, step into 버튼을 누르면 632 REVERT에서 오류로 인해 중지되고 있다는 것을 알 수 있다. 이 때 우측의 소스에 해당 라인이 하이라이트되는 것을 볼 수 있다.

또한 디버깅을 하면서 아래에 지역변수, 스택, 메모리, 스토리지변수에 프로그램에서 쓰이고 있는 변수들이 올바르게 값을 가지고 있는지 확인하면서 진행할 수 있다.

그림: 디버거에서 step into 버튼을 누르면서 디버깅을 하고 있다 (REMIX가 업그레이드 되면서 화면이 바뀔 수 있다.) 

## 실습 문제: 송금

REMIX에서 은행 컨트랙을 프로그래밍하여 아래 함수들을 만들고, 버튼 테스트를 실행하도록 하자.

### 컨트랙의 함수들

컨트랙에 다음 함수들을 구현하자.

* deposit(uint amount): 입금액을 매개변수로 넣어서 입금하자. amount가 실제 입금액과 일치하지 않으면 오류.
* widthdrawAll(): 전액 인출 함수, 인출 금액을 정하지 않아도 되므로 함수의 인자가 없다.
* getBalance(): 컨트랙 잔고확인 함수 (this를 이용한 잔고, 상태변수 잔고 (this 잔고가 맞는지 확인하는 용도)
* forwardTo(address payable \_receiver): 다른 계정으로 계좌이체 함수

### 이벤트를 설정한다

이벤트도 잠깐 설명하고 가자. 문법은 매우 단순해서, 단 2줄, 줄2와 줄4로 이벤트를 설정할 수 있다.

줄1: contract BankV2 {  
 ...  
줄2: event Sent(address from, address to, uint amount ); // 이벤트를 설정한다  
줄3: function forwardTo(address payable \_receiver) public payable {  
 ...  
줄4: emit Sent(msg.sender, \_receiver, msg.value); // 이벤트 호출  
 }  
 ...  
 }

호출되는 과정은 복잡할 수 있다. 클라이언트, 즉 노드에서 아래 줄5 함수가 이벤트의 발생을 모니터링하다, 이벤트가 발생하면 알리는 줄6 코드가 필요하다.

줄5: var event = bank.events.Sent(function (error, result) { //이벤트가 발생하는지 모니터링하는 함수  
 ...  
줄6: console.log("Event fired: " + JSON.stringify(result.returnValues)); //이벤트의 발생을 알린다  
 ...  
 });

이벤트는 왜 필요할까? 간단하다. 줄3의 forwardTo()가 호출되면 그것을 알려주려는 것이다. 은행에서 출금을 알려주지 않는가? 같은 이유이다. 계좌 이체를 하면 줄4의 Sent() 이벤트가 발생하고 내게 알려지는 것이다.

블록체인에서는 이런 간단한 이벤트의 알림도 간단치 않다. 발생한 줄3 forwardTo를 줄4 Sent에서 이벤트를 발생하고, 클라이언트에서 이벤트의 발생을 모니터링하다, 발생하자마자 띠링~하고 알람이 뜨게 되는 것이다.

그렇게 되면, 클라이언트는 블록체인에서 발생한 계좌이체를 알림을 통해 '누군가 이체를 했구나' 알게되는 것이다.

이러한 이벤트의 설정과 알림은 다음 장에서 더 배우게 된다.

### 버튼 테스트

REMIX에서 코딩하고, 위 함수의 버튼을 만들어 테스트해보자.

* 입금 11111 wei를 하고 컨트랙 잔고 11111 wei 출력 (old: 입금 11111 wei, 222 wei를 하고 잔고 11333 wei 출력)
* 자신의 2번째 계정으로 계좌이체 555 wei하고 컨트랙 잔고는 그대로 11111 wei 출력, 자신의 2번째 계정 잔고 증가분 (+555 wei) 출력
* 전액 인출하고 1111 wei, 자신의 잔고 증가분 출력 (+1111 wei)

ganache@8335에서 해보자. 성공하면 geth@8445에도 배포하고 동일한 기능을 실행해보자.

### 주의

* 계정의 잔고는 gas비 또는 마이닝 보상 등이 결제된다. 따라서 증감분이 정확히 반영되지 않는다.
* 일시에 복수의 거래가 발생한다. 따라서 마이닝, 계정 해제를 복수거래에 대해 해주어야 한다 (geth@8445에서 한다면).
* 이벤트 발생을 위해서는 웹소켓을 사용한다. ganache@8345에는 웹소켓을 별도 설정없이 그대로 사용하지만, geth@8445는 따로 추가되어 있어야 한다. 이벤트는 다음 장에서 배우게 된다. 미리 해보기로 하자. 물론 이벤트를 제외하고 해도 좋다.

### 단계 1: 컨트랙 개발

REMIX에서 코드를 작성한다. 물론 앞서 그랬던 것과 같이 함수의 버튼 테스트는 당연히 실행하자.

[파일명: src/BankV2.sol]  
//SPDX-License-Identifier: UNLICENSED  
pragma solidity ^0.8.0;  
  
contract BankV2 {  
 address owner; //버전 0.8.0 payable로 설정할 필요가 없다  
 uint balance; //잔고를 추적한다.  
 event Sent(address from, address to, uint amount ); // 이벤트를 설정한다  
 constructor() payable {   
 owner = msg.sender;  
 balance = 0;  
 }  
 function forwardTo(address payable \_receiver) public payable {  
 //balance -= msg.value; // 자신의 잔고에서 차감하는 경우  
 require(msg.sender == owner); // owner만 계좌이체 권한을 가진다  
 \_receiver.transfer(msg.value); // 입금액을 단순히 계좌이체  
 emit Sent(msg.sender, \_receiver, msg.value); // 이벤트 호출  
 }  
 function getBalance() public view returns(uint, uint) {  
 return (balance, address(this).balance); // 잔고를 출력  
 }  
 function deposit(uint amount) public payable {  
 require(msg.value == amount); // 인자와 입금액이 다르면 오류가 발생한다  
 balance += amount; // 입금액만큼 잔고 증가  
 }  
 function widthdrawAll() public {  
 balance -= address(this).balance; // 총잔고를 차감  
 require(msg.sender == owner); // owner만 출금 권한을 가진다  
 //owner.transfer(address(this).balance); // payable 오류  
 payable(owner).transfer(address(this).balance); // 0.8.0 부터 payable (이전 0.5에서는 오류)  
 }  
}

코드는 비교적 단순하다. 그러나 이렇게 간단히 완성될 수 있다는 것이 놀라울 수 있다. 블록체인은 금융 플랫폼으로서, 일반 은행의 입출금이 안전한 거래를 위해 다양한 장치를 가지고 있다는 것과 비교될 수 있다.

|  |  |
| --- | --- |
| 줄 | 설명 |
| 4 | address선언 |
| 7 | event 설정하여 송금할 경우 알림 |
| 8 ~ 11 | 생성자에 public을 설정하지 않는다. msg.sender를 owner로 설정, balance 초기화. |
| 12 ~ 17 | 송금. owner만 송금할 수 있고 event 발생. msg.value를 넘겨주는 단순 이체라서 잔고를 줄이지 않음. |
| 18 ~ 20 | 잔고는 (1) address(this).balance와 (2) balance (수작업으로 조정하는 잔고)를 돌려줌. 이들 잔고는 서로 동일해야 함. |
| 25 ~ 29 | owner에게 전액 출금. 버전 0.8에서는 payable(msg.sender)라고 해준다. |

### 단계 2: 컴파일

REMIX에서 버튼 테스트까지 마치고, 컴파일도 성공한다면 컴파일 오류는 없다는 의미이다. 나중에 배포하거나 사용하면서 실행 오류가 발생할 수 있는 가능성이 있다. 이런 점이 블록체인 프로그래밍을 좀 더 어렵게 만드는 요인이기도 하다.

잠깐. 호환성 문제로, 아무리 살펴봐도 오류가 없는 것 같은데 문제가 발생한다면 고민하지 말자. 버전을 낮춰보면 예상 외로 문제가 풀리기도 한다. 저자도 그런 문제로 인해, 버전을 낮추어 0.5로 낮추어 성공한 경험이 있다. 컴파일러 버전 0.5가 설치되지 않았다면, REMIX에서 해당 버전으로 컴파일하고 abi, 바이트 코드를 복사해서 사용한다.

pjt\_dir> solc-windows.exe --optimize --combined-json abi,bin src/BankV2.sol > src/BankV2.json

### 단계 3: 컨트랙 배포

배포 프로그램을 작성해보자. 이전까지는 (1) abi, 바이트 코드, (3) 자신의 계정을 복사-붙여넣고 있다.

(1)에 대해서는 파일에 쓰고, 거기로부터 읽어와서 복사-붙여넣기의 수고를 덜고 있다.

(2)에 대해서는 이제 web3 1.0 버전 이후에 도입된 비동기적 처리를 위해 async 함수를 사용하자. 함수를 async로 설정하고, 비동기 함수에 대해 await를 적용하게 된다. 아래 코드에서 계정을 읽어오는 부분을 보자.

async function deploy() { // 함수를 async로 설정  
 const accounts = await web3.eth.getAccounts(); // web3.js 함수를 await로 설정한 후 결과를 accounts 배열에 저장한다.  
 console.log("Deploying the contract from " + accounts[0]); // 배열 accounts  
}

'커피를 다 갈 때까지 물을 끓이지마'는 순차적, 동기적이다. '커피를 갈고 있을테니 기다리지 말고, 물을 먼저 끓인다'는 커피 가는 작업을 비동기 처리하는 것이다. 그러니까 await web3.eth.getAccounts() 이렇게 적으면, 계정 주소는 가져올테니, 다 끝날 때까지 대기하는 것이 아니라, 다음 작업 contract을 생성하거나 하면서, 계정 주소를 가져오면 그 때 읽을 수 있게 된다. 아래 await를 보면 비동기 작업이 적용되고 있다.

[파일명: src/BankV2Deploy.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./BankV2.json'); // JSON 파일 읽기  
  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙명 ['src/BankV2.sol:BankV2']  
console.log("- contract name: ", contractName);  
\_abi=\_abiBinJson.contracts[contractName].abi;  
\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); // 이런 오류 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: 1000000, gasPrice: '1000000000'}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
  
deploy()

코드를 실행하고 결과를 보자. 기대한 바와 같이, 첫 째 계정을 하드코딩하지 않고도 인식하고 있다. 그리고 배포된 컨트랙 주소도 출력하고 있다.

pjt\_dir> node src/BankV2Deploy.js  
  
- contract name: [ 'src/BankV2.sol:BankV2' ]   
Deploying the contract from 0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE 계정을 비동기적으로 읽어오고 있다  
hash: 0xa36ba52ee3e74f220fe229857b10e12dfde7b1dd3597c320924cfe66a5cc533d  
---> The contract deployed to: 0x8b1A8486fB70E1EF99b32Ac4b45848CF424f0F7B 컨트랙 주소

### 단계 4: 사용

#### 이벤트 발생하지 않는 경우

우선 이벤트를 발생하지 않는 경우로 생각해서 코드들 작성해보자. 그렇다면 연결은 HttpProvider를 통해서 이루어진다.

배포를 ganache@8345에서 했으므로, 일관성있게 ganache@8345에서 사용하는 코드를 작성해보자.

위 코드 줄3과 줄4 가운데 하나는 주석으로 만들고 있다. 줄3을 HttpProvider로 활성화하고 실행하면 이벤트가 발생한 것을 포착하지 못한다.

줄3 var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345")); // http 연결  
줄4 //var web3 = new Web3(new Web3.providers.WebsocketProvider('ws://localhost:8345')); // 웹소켓 연결

코드를 읽어보면, await를 어디는 적고, 어디는 안적고 하고 있다. REMIX 버튼의 색을 떠올리면 쉽다. 즉 블록체인을 변경하는 함수들은 await를 붙이고, 아닌 경우에는 붙이지 않는 것을 원칙으로 한다.

* deposit(), forwardTo(), withdrawAll() 등에는 await를 적고,
* getBalance()에는 await를 붙이지 않고 있다.

참고로 HTML 페이지로 만든다면 어떻게 될지 생각해보자. 이와 같이 모든 함수를 묶어서 일괄처리하지 않고, 사용자가 버튼을 누르든지 요청을 해서 함수가 아마도 개별적으로 실행이 되는 방식으로 만들어지게 된다. 그러니까 입금 버튼을 누르면, 입금함수가 실행되고, 출금 버튼을 누르면 출금이 실행되는 이런 방식이 되겠다.

기억하자. 다음 코드에서 web3.eth.Contract(\_abiArray, "0x8b1A8...f0F7B")의 두 번째 인자에 본인이 배포한 컨트랙 주소를 붙여넣기한다.

[파일명: src/BankV2NoEventUse.js]  
var Web3 = require('web3');  
var \_abiBinJson = require('./BankV2.json'); // JSON 파일 읽기  
  
//var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345")); // http 연결  
var web3 = new Web3(new Web3.providers.WebsocketProvider('ws://localhost:8345')); // 웹소켓 연결  
  
contractName=Object.keys(\_abiBinJson.contracts); // 컨트랙명 ['src/BankV2.sol:BankV2']  
console.log("- contract name: ", contractName);  
\_abi=\_abiBinJson.contracts[contractName].abi;  
\_abiArray=JSON.parse(JSON.stringify(\_abi));  
//\_abiArray=JSON.parse(\_abi); // 이런 오류 발생하면 SyntaxError: Unexpected token o in JSON at position 1  
  
var bank = new web3.eth.Contract(\_abiArray,"0x8b1A8486fB70E1EF99b32Ac4b45848CF424f0F7B"); // 여기에 컨트랙 주소 붙여넣기  
var event = bank.events.Sent(function (error, result) {  
 if (!error) {  
 console.log("Event fired: " + JSON.stringify(result.returnValues));  
 }  
});  
  
async function doIt() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account0: " + accounts[0]);  
 const balance0Before = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance0 before: " + balance0Before);  
 const balance1Before = await web3.eth.getBalance(accounts[1]);  
 console.log("Balance1 before: " + balance1Before);  
  
 bank.methods.getBalance().call().then(function(bal) {  
 console.log("Contract bal before deposit: " + bal[0] + " this.bal:" + bal[1]);  
 });  
 await bank.methods.deposit(1111).send({from: accounts[0],gas:80000,value:1111});  
 bank.methods.getBalance().call().then(function(bal) {  
 console.log("Contract bal after deposit: " + bal[0] + " this.bal:" + bal[1]);  
 });  
 const forward = await bank.methods.forwardTo(accounts[1]).send({from: accounts[0],gas:100000,value:555});  
 //const forward = await \_test.methods.forwardTo().send({from: accounts[0], gas: 364124, gasPrice: '1000000000'})  
 //.then(function(value) {console.log("---> myFunction called " + JSON.stringify(forward.events.Sent.returnValues));});  
   
 console.log("---> forwardTo called " + JSON.stringify(forward.events.Sent.returnValues));  
 bank.methods.getBalance().call().then(function(bal) {  
 console.log("Contract balance after forwardTo: " + bal[0] + " this.bal:" + bal[1]);  
 });  
 const balance0After = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance0 after: " + balance0After);  
 console.log("Balance0 diff: " + (balance0After - balance0Before));  
 const balance1After = await web3.eth.getBalance(accounts[1]);  
 console.log("Balance1 after: " + balance1After);  
 console.log("Balance1 diff: " + (balance1After - balance1Before));  
  
 const withdraw = await bank.methods.widthdrawAll().send({from: accounts[0],gas:100000});  
 bank.methods.getBalance().call().then(function(bal) {  
 console.log("Contract balance after withdrawAll: " + bal[0] + " this.bal:" + bal[1]);  
 });  
 const balance0AfterWithdrawAll = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance0 after withdrawAll: " + balance0AfterWithdrawAll);  
 console.log("Balance0 diff after withdrawAll: " + (balance0After - balance0AfterWithdrawAll));  
}  
doIt()

#### 컨트랙 주소를 잘 못 입력하면 Out of Gas 오류가 발생

위 컨트랙의 주소를 잘 못 넣으면, 오류가 발생한다. 왜 오류가 발생할까? 모르는 계정에서 gas를 지급하려고 하니, Out of Gas 오류가 발생하기 마련이다.

#### Http는 이벤트 리스닝 못한다.

코드를 실행해보자.

위 출력을 단계별로 설명하고 있다. 송금 또는 잔고 변화 등 어떤 결과가 발생했는지 살펴보자. 자바스크립트에서 발생하는 오버플로우(overflow)가 발생해서 계정 잔고는 부정확하지만, 컨트랙의 잔고는 정확히 출력된다.

pjt\_dir> node src/BankV2NoEventUse.js  
  
- contract name: [ 'src/BankV2.sol:BankV2' ]  
Account0: 0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE  
Balance0 before: 999998332955999999445  
Balance1 before: 1000000000000000000555  
Contract bal before deposit: 0 this.bal:0  
Contract bal after deposit: 1111 this.bal:1111 <--- 컨트랙의 잔고가 desposit한 금액만큼 증가  
---> forwardTo called {"0":"0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE","1":"0x32f525923d43DA7532eeE0391654d44a0d3F29eD","2":"555","from":"0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE","to":"0x32f525923d43DA7532eeE0391654d44a0d3F29eD","amount":"555"}  
Balance0 after: 999998175789999997779 <--- 555만큼 감소해야 함. gas비 등을 산정해야 함.  
Balance0 diff: -157166000013312 <--- 오버플로우로 계산 부정확  
Contract balance after forwardTo: 1111 this.bal:1111  
Balance1 after: 1000000000000000001110 <--- 계좌이체 555만큼 증가  
Balance1 diff: 555  
Balance0 after withdrawAll: 999998114419999998890  
Balance0 diff after withdrawAll: 61369999949824 <--- 오버플로우 계산 부정확  
Contract balance after withdrawAll: 0 this.bal:0 <--- 컨트랙에서 전액출금한 후 잔고가 0

위 출력에서 보듯이, HttpProvider는 Event를 리스닝하지 못하고 있다.

HttpProvider는 이벤트를 알아채는 방식이 아니다. HTTP 프로토콜은 원래 연결을 지속하는 방식이 아니고, 이벤트가 발생한다 해도 연결이 끊어져 있으면 알릴 방도가 없기 때문이다. 따라서 이벤트를 자동 알림을 통해서가 알지 못한다는 의미다.

HttpProvider로 연결하면, 이벤트가 발생했는지 아는 방법은 그냥 로그의 events.Sent.returnValues를 출력하는 수 밖에 없다. 프로그램 중간쯤 보면 JSON.stringify(forward.events.Sent.returnValues)에서 emit() 함수 인자의 from, to, acount가 출력되고 있다.

"from":"0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE", "to":"0x32f525923d43DA7532eeE0391654d44a0d3F29eD", "amount":"555"

모닝콜로 비유하면, 아침 일어나는 시각을 프론트에서 자동으로 알림하는 것이 아니라, 투숙객 자신이 그 시간이 되었는지 확인하는 것이라고 볼 수 있다.

geth@8445에서 했다면, 단말에서도 잔고의 변화가 적용되는 것을 알 수 있다. 현재는 ganache@8345에서 작업하고 있으니, ganache@8345 콘솔에서 확인해볼 수 있다.

geth> eth.getBalance(eth.accounts[1]);  
0 <--- 이전 잔고  
> eth.pendingTransactions  
[{  
 blockHash: null,  
 blockNumber: null,  
 from: "0xA3dB...D59CE",  
 gas: 100000,  
 gasPrice: 500000000000,  
 hash: "0xa4c9e...8c94f",  
 input: "0x3e58c...ed480",  
 nonce: 32,  
 r: "0x733d3...47a6d",  
 s: "0x78476...a0213",  
 to: "0x32f52...F29eD",  
 transactionIndex: 0,  
 v: "0x66",  
 value: 555 <--- 계좌이체 전송될 금액  
}]  
> miner.start(1);admin.sleepBlocks(1);miner.stop(); 계좌이체 거래의 마이닝  
null  
> eth.getBalance(eth.accounts[1]);  
555 <--- 이후 잔고

#### 이벤트가 발생하는 경우: 웹소켓

HttpProvider는 이벤트 리스닝을 지원하지 않는다. 이번에는 웹소켓으로 이벤트를 발생해서 해보자. 즉 줄4를 활성화해서 WebsocketProvider를 사용하자.

줄3 //var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345")); // http 연결  
줄4 var web3 = new Web3(new Web3.providers.WebsocketProvider('ws://localhost:8345')); // 웹소켓 연결

이 줄만 수정하면 다른 부분은 그대로 동일하기 때문에 코드는 생략한다. 위 프로그램을 복사해서 쓰도록 하자. 그러나 프로그램은 src\BankV2Use.js로 수정해주자. 덮어쓰면 원파일을 잃기때문에 그렇다.

forwardTo() 함수가 호출되면서, Sent 이벤트가 발생하고, 줄3의 Event fired: 매개변수 값을 출력하고 있다.

줄1 var event = bank.events.Sent(function (error, result) {  
줄2 ...  
줄3 console.log("Event fired: " + JSON.stringify(result.returnValues));  
줄4 ...  
줄5 });

프로그램을 실행해보자. Event fired 부분이 출력되고 있다.

pjt\_dir> node src\BankV2Use.js  
  
- contract name: [ 'src/BankV2.sol:BankV2' ]  
Account0: 0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE  
Balance0 before: 999997895883999998335  
Balance1 before: 1000000000000000001665  
Contract bal before deposit: 0 this.bal:0  
Contract bal after deposit: 1111 this.bal:1111  
Event fired: <--- 이벤트가 발생.{"0":"0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE","1":"0x32f525923d43DA7532eeE0391654d44a0d3F29eD","2":"555","from":"0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE","to":"0x32f525923d43DA7532eeE0391654d44a0d3F29eD","amount":"555"}  
---> forwardTo called {"0":"0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE","1":"0x32f525923d43DA7532eeE0391654d44a0d3F29eD","2":"555","from":"0xA3dB18a0fDF21be547F98CdF09DfDD76B64D59CE","to":"0x32f525923d43DA7532eeE0391654d44a0d3F29eD","amount":"555"}  
Balance0 after: 999997738717999996669  
Balance0 diff: -157165999882240  
Contract balance after forwardTo: 1111 this.bal:1111  
Balance1 after: 1000000000000000002220  
Balance1 diff: 0  
Balance0 after withdrawAll: 999997677347999997780  
Balance0 diff after withdrawAll: 61370000080896  
Contract balance after withdrawAll: 0 this.bal:0

## 연습문제

1. SPDX는 선택적으로 적어도 되고, 쓰지 않아도 된다. OX로 답하시오.
2. import myContract과 같이 읽어 들일 컨트랙 명을 써준다. OX로 답하시오.
3. 배열은 int x[] 이렇게 적는다. int[] x;이라고 적지 않는다.
4. 데이터타입 bool의 용량은 몇 바이트인지 답하시오.
5. int라고 선언하는데, M % 8 == 0을 지켜야 한다. 무슨 뜻인지 설명하시오.
6. byte1의 용량은 몇 바이트인지 답하시오.
7. 문자열 "hello"의 길이를 출력하도록 코드를 작성하시오.
8. abi.encodePacked("hello")하면 결과는 바이트인지 문자열인지 고르시오. 출력결과를 적으시오.
9. Struct에 데이터를 입력하는 코드를 작성하시오. 데이터는 (학번 20171111, 이름 lim, 등록).

* struct Student {  
   uint num;  
   string name;  
   bool isEnrolled;   
  }

1. 요일을 enum으로 선언하는 코드를 작성하시오.
2. 나이가 15, 25, 35인 3 명이 있다고 하자. 나이를 저장하는 배열을 선언하는 코드를 작성하시오.
3. 20 Wei를 송금한다고 하자. 자신의 계정을 복사해서 사용하고, 그 계정에서 송금하는 다음 코드를 작성하시오.
   1. transfer 함수
   2. send 함수
   3. call 함수
4. 문자열을 encodePacked("hello") 하고 그 결과를 적으시오.
5. 다음이 값으로 저장되는 타입인지 또는 참조타입인지 구별하여 적으시오.
   1. int
   2. address
   3. bytes1
   4. bytes
   5. string
   6. mapping
6. 다음 괄호를 채우시오.

* function sayHello() public pure returns( ) { return "hello"; }

1. public은 외부에서만 사용할 수 있다는 의미이다. OX로 답하시오.
2. 다음 중 default 가시성을 고르시오.
   1. public
   2. private
   3. internal
   4. external
3. string s = "hello"이라고 하자. 다음을 코드로 작성하시오.
   1. bytes로 변환
   2. 위 1의 결과를 string으로 다시 변환
4. 전역 변수로 다음을 코드로 적으시오.
   1. 전송 금액
   2. 전송 계정 주소
   3. 트랜잭션에 사인한 계정
5. Hello 컨트랙을 수정하여, 인사하는 대상을 지정해보자.

* 멤버 변수에는 toWhom을 지정한다.
* 멤버 함수에는 아래 함수를 구현한다. 일부러 함수의 입출력 인자는 생략하니 자신이 생각해서 구현한다.

function sayHello() view public returns() 입력: 없다, 출력: Hello smu (bytes 또는 string로 출력)  
function setWhom() public 입력: 인사하는 대상 (toWhom을 지정, bytes 또는 string로 입력)  
function getWhom() view public returns() 출력: 인사하는 대상 (toWhom을 읽음, bytes 또는 string로 출력)

문자열 합성을 하려면, 이전에는 abi.encodePacked()를 함수를 사용했으나 버전 0.8.10부터 bytes.concat() 또는 string.concat() 함수를 사용하면 된다.

2 개의 컨트랙을 구현한다 - HelloInString.sol, HelloInBytes.sol.

* 1. REMIX에서 실행하고, 각 함수의 로그 결과를 붙여 넣는다 (입력, 출력, gas가 보이도록).
  2. HelloInString과 HelloInBytes의 배포할 때와 sayHello() 함수의 gas를 비교한다.

//SPDX-License-Identifier: GPL3.0  
pragma solidity ^0.8.0;  
  
contract HelloInBytes {  
 bytes toWhom;  
 function sayHello() view public returns(bytes memory) {  
 bytes memory hello = bytes("Hello "); // or 0x776f726c64; //hello  
 return bytes.concat(hello, toWhom);  
 }  
 function setWhom(bytes memory whom) public {  
 toWhom = whom;  
 }  
 function getWhom() view public returns(bytes memory) {  
 return toWhom;  
 }  
}

//SPDX-License-Identifier: GPL3.0  
pragma solidity ^0.8.0;  
contract HelloInString {  
 string toWhom;  
 function sayHello() view public returns(string memory) {  
 //return "Hello " + toWhom; //not this way  
 //return string(abi.encodePacked("Hello ", toWhom)); //string concat  
 return string.concat("Hello ", toWhom); //ok from 0.8.12  
 }  
 function setWhom(string memory whom) public {  
 toWhom = whom;  
 }  
 function getWhom() view public returns(string memory) {  
 return toWhom;  
 }  
}

1. 컨트랙 '학생'을 구현하시오. 컨트랙의 이름은 'StudentsCrud.sol'로 명명하고, 학생들을 배열로 구성하고, 다음 기능이 제공되도록 하자.

* 입력: '학번', '이름', '등록 여부'를 입력한다.
* 검색: 조건에 해당하는 항목을 찾기 위해서는 반복문이 필수적이다.

그러나 gas비용이 급증할 수 있으므로, 배열의 인덱스로 검색하도록 하자.

* 삭제: 배열에서는 어떤 항목의 데이터를 지우기 위해서는 검색이 필요하다. 그러나 역시 gas비용이 급증할 수 있으므로, 배열의 인덱스에 해당하는 항목을 지우기로 한다. 해당 항목을 지우고 나서는, 그 항목을 제거하고 배열의 크기도 같이 줄어야 한다.

함수의 시그니처는 다음과 같다.

\* 입력함수 - function insert(uint n, string memory sn, bool e) public  
\* 첫 데이터 조회 - function getFirstStudent() public view returns(uint, string memory, bool)  
\* 검색함수 - function findBy(uint8 index) view public returns(uint, string memory, bool)  
\* 삭제함수 - function remove(uint index) public  
\* 배열크기 조회함수 - function getLength() view public returns(uint)  
\* 삭제함수 - function pop() public

컨트랙을 구현하고, REMIX에서 DEPLOY한 후, 제공되는 함수 버튼을 활용하여 다음 작업을 수행해보자. 항목별 답안으로는 수행 결과가 출력된 REMIX 단말의 로그를 붙여 넣기 하면 된다. (선택적으로 노드에서 배포하고 실행해도 된다)

* 1. 아래 항목을 입력
  + 201711111,"kim",false  
    201711112, "park", true  
    201711113, "lee", false  
    201711114, "lim", false
  1. 2 번째 데이터 201711112 제거
  2. 배열 크기 조회 (4 개 항목에서 1 개가 제거되었으니 3이 출력)
  3. 2 번째 데이터 조회를 한다. 20171112는 삭제되어서 출력할 수 없고, 다른 데이터 항목이 출력.
  4. 첫 데이터 조회

1. 배열의 부분을 삭제할 필요가 있다고 하자. 방법은 여러 가지가 가능하다. 즉 정답이 있는 것이 아니라 설계의 문제라는 의미이다. 다음 방법으로 구현해보자.

* 마지막 요소를 삭제 요소로 복사하고, 마지막 요소를 pop하거나,
* 삭제한 인덱스 자리를 비워놓거나,
* 삭제하고 비우지 않고, 삭제 표시만 하거나,
* 삭제하고 뒤를 한 개씩 순차적 이동

다음은 샘플 프로그램이다. 원하는 방식에 따라 수정해보자.

[파일명: src/StudentsCrud.sol]  
  
pragma solidity ^0.6.0;  
contract StudentsCrud {  
 struct Student {  
 uint num;  
 string name;  
 bool isEnrolled;  
 }  
 Student[] sDB  
 function insert(uint n, string memory sn, bool e) public {  
 Student memory s = Student(n, sn, e);  
 sDB.push(s);  
 }  
 function getFirstStudent() public view returns(uint, string memory, bool) {  
 return (sDB[0].num, sDB[0].name, sDB[0].isEnrolled);  
 }  
 function findBy(uint8 index) view public returns(uint, string memory, bool) {  
 return (sDB[index].num, sDB[index].name, sDB[index].isEnrolled);  
 }  
 function remove(uint index) public {  
 if (index < sDB.length) {  
 sDB[index]=sDB[sDB.length-1];  
 sDB.pop(); //just pop() worked  
 }  
 }  
 function getLength() view public returns(uint) {  
 return sDB.length;  
 }  
 function pop() public {  
 sDB.pop();  
 }  
}