# Chapter 11. 컨트랙의 결합

컨트랙은 다른 컨트랙과의 결합이 당연히 필요할 수 밖에 없다. 필요한 컨트랙은 import문으로 가져올 수 있고, new 명령어를 통해 객체를 생성한다. 그리고 객체 지향 프로그래밍의 상속도 가능하다. 다른 언어와 같이 라이브러리 역시 만들어서 활용할 수 있다.

# 1. 컨트랙 간의 관련

## 1.1 관련의 구분

지금까지는 독립적으로 존재하는 컨트랙을 개발했다. 컨트랙이 2 개 이상이 되는 경우, 이들 간의 결합이 필요하다. 객체 지향에서의 결합하는 방식으로 is-a, has-a 관계를 구현할 수 있다. 객체 지향에서의 has-a는 (1) 의존(dependency), (2) 연관(association), (3) 포함(aggregation), (4) 구성(composition)로 구분하고 있다. is-a는 상속 관계로 설명할 수 있는데 뒤에서 설명한다.

어떤 결합을 어느 경우에 적용해야 하는지 객체 지향에 있어서는 성능에 영향을 끼친다. 제한된 메모리를 효율적으로 사용하기 위해, 객체는 생성되고 어느 정도 사용되지 않으면 메모리에서 해제되는 것이 일반적이다. 그래서 사용하는 기한에 따라, 서로 집합의 관계인지, 사용하는 시점에 따라 결합의 방식을 결정하기도 한다.

그러나 블록체인은 그렇지 않고, 객체가 생성되고 나면 메모리에 항상 존재하는 구조이다. 이런 면을 고려하면, 객체와의 결합은 블록체인 아닌 경우와 비교해 상대적으로 그리 중요하지 않을 수 있다.

어차피 메모리에 상주하고 메모리를 할당하고 해제하는 방식이 아니라면, 언제 결합을 맺고, 끊고 등의 메모리를 효율적으로 사용하려는 노력은 무의미할 수 있다. Solidity에서는 코드 재사용과 더불어 gas 비용의 관점에서 어떤 관련을 적용할 것인지 선택하는 것이 오히려 합리적일 수 있다.

### 1.1.1 의존은 제한된 시간 동안의 has-a 관계이다.

의존 관계(Dependency)는 객체 간의 관계가 있지만, 그 관계가 오래가지 않고 한시적이다. 온라인 쇼핑의 예를 들어보자. 어떤 상품을 주문하면서 이름, 전화번호, 배송지 등의 정보를 필요로 한다. 아래 줄2 placeOrder() 주문 함수는 고객 정보를 매개변수로 받고 있다. 그 관계가 함수 내에서는 제한적으로 존재하고, 함수 밖에서는 끊어진다.

줄1 Contract Order {  
줄2 function placeOrder(Customer c, Product p) //컨트랙이 함수의 인자로 쓰이고, 함수 내에서 제한적 관련  
줄3 }

### 1.1.2 연관은 장기간의 has-a 관계이다.

연관(Association)은 의존 관계에서 보는 has-a의 관계이지만, 그 관계가 오래 지속된다는 점에서 차이가 있다. 객체 간 관계가 있고, 그 관계가 거의 그 객체의 생애 주기(lifetime)에 가깝게 지속되는 경우이다.

주문의 예를 다시 들어보면, 고객과 주문, 배송, 문의, 결재 등 지속적인 상호작용이 필요하며 이런 경우 의존 관계 보다는 연관 관계가 적합하다.

즉, 아래 줄2에서 Customer와의 관계가 멤버변수 자리에 위치하고 있어서 그 관계는 줄3의 주문, 줄4의 결재, 줄4의 배송 등 오랫동안 그 관계가 필요하고 지속된다.

줄1 contract Order {  
줄2 Customer customer //멤버변수 자리의 컨트랙이고, 장기적 관계  
줄3 function placeOrder()  
줄4 function pay()  
줄5 function deliver()  
줄6 }

### 1.1.3 포함은 전체와 부분의 관계이다.

포함(Aggregation)은 has-a의 관계이면서 전체-부분으로 구성된다. 장기적인 has-a관계라는 점에서 연관과 유사하지만, 전체-부분이라는 차이가 있다.

줄1의 강의는 줄2 학생을 부분으로 구성한다. 학생은 그 자체로 강의의 부분이 아니어도 스스로 존속할 수 있다. 줄2에서 보듯이 강의는 학생을 배열과 같은 구성으로 가지게 된다.

줄1 contract Course {  
줄2 Student[] students; //학생은 강의를 구성하는 부분, 그러나 그 자체로 존재할 수 있다.  
줄3 }

### 1.1.4 구성은 전체와 부분이고 전체가 부분을 소유한다.

구성(composition)이라는 한글 단어는 객체 지향의 그 개념을 온전하게 나타내는 단어는 아니라고 생각한다. 포함(aggregation)과 유사하지만, 전체-부분의 관계가 강하다. 부분은 전체와 생성-제거의 동일한 시간 범위를 가진다. 즉 전체가 부분을 생성하고, 소유하고 있어 전체가 부분을 해제하면 그 자체로 존재하지 못하는 것으로 이해하자.

줄1 Vehicle은 줄2 Engine이 필수적이다. 엔진 없는 자동차가 가능하기나 한가? 그렇지 않다. 그 관계는 자동차가 만들어지는 시점에, 즉 줄3의 생성자에서 제작시점부터 줄4와 같이 반드시 장착되어야 한다.

줄1 contract Vehicle {  
줄2 Engine engine;  
줄3 constructor Vehicle() {  
줄4 engine = new Engine(); //생성될 때부터 전체의 부분이고, 독립적으로 존재하지 못한다.  
줄5 }  
줄6 }

## 1.2 상대측 컨트랙 객체의 생성

객체 지향에서 결합은 메모리 효율적인 대안을 고려하여 이루어지곤 한다. 컨트랙을 결합할 경우, 그 대상 컨트랙이 (1) 배포되어 있지 않은 경우, (2) 배포되어 있는 경우로 나누어 구분할 필요가 있다. 배포되어 있지 않는 경우에는 단순하다. 반면에 배포되어 있는 컨트랙을 결합하려면 주소를 알아야 한다. 이와 같이 구분하여 그 객체를 어떻게 가져와 사용하는지 설명해 보자.

### 1.2.1 동일한 파일의 컨트랙과 결합

new() 명령어로 컨트랙을 생성하려면 컴파일 시점에 그 소스코드를 가져올 수 있어야 한다.

즉 **대상 컨트랙이 동일한 파일에 존재**하거나 또는 곧 배우게 될 **import 문으로 다른 컨트랙이 포함**되어야 한다. 그러면 해당 컨트랙이 컴파일되어 바이트코드가 포함된다.

줄1의 Customer, 줄2의 Order가 한 파일에 포함되어 있다. 이 경우 new 명령어로 **인스턴스**를 만들 수 있다. 줄3에서 보듯이 멤버변수를 선언하면서, 아예 객체를 생성해 놓을 수 있다.

줄1 contract Customer { }  
줄2 contract Order {  
줄3 Customer c = new Customer(); //객체를 처음부터 생성  
줄4 constructor() { }  
줄5 }

또는 객체 생성을 연기할 수 있다. 줄5에서 생성자 함수를 호출하면서 비로서 객체를 생성하므로 메모리가 나중에 할당이 되는 효과가 있다. 생성자가 아닌 함수에서 호출하면 시점이 더욱 대조적으로 늦춰지는 효과가 있다.

줄1 contract Customer { }  
줄2 contract Order {  
줄3 Customer c;  
줄4 constructor() {  
줄5 c = new Customer(); //생성자 함수를 호출할 때 비로서 객체를 생성  
줄6 }

## 실습: 자동차와 엔진의 구성 - new 명령어 사용

자동차, 엔진 컨트랙을 구현해보자.

자동차는 시동을 걸고, 속도를 높이거나 낮추거나, 색상을 조회하는 함수를 구현한다. 실제 자동차의 구동은 아니라서 함수들은 간단히 구현하기로 하자.

자동차는 엔진을 통해 작동을 하기 때문에 별도로 엔진 컨트랙을 만들어 함수가 호출되고 있다.

이들 컨트랙들이 서로 어떻게 has-a의 결합을 하는지 판단해보자.

[파일명: src/Car.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Car {  
 Engine engineObj;  
 string private color;  
 event PrintLog(address sender, string msg);  
  
 constructor() {  
 engineObj = new Engine();  
 }  
 function setColor(string memory \_color) public {  
 color=\_color;  
 }  
 function getColor() public view returns(string memory) {  
 return color;  
 }  
 function getSpeed() public view returns(uint) {  
 return engineObj.getSpeed();  
 }  
 function speedUpBy10() public {  
 engineObj.setSpeedUpBy(10);  
 }  
 function speedDownBy10() public {  
 engineObj.setSpeedDownBy(10);   
 }  
 function start() public {  
 engineObj.on();  
 string memory engineStateStr=engineObj.getEngineState()? "on" : "off"; //출력문자열로 만든다  
 emit PrintLog(msg.sender, engineStateStr);  
 }  
}  
  
contract Engine {  
 uint constant private MAXSPEED = 200;  
 uint private speed;  
 bool private engineState;  
  
 constructor() {  
 speed = 0;  
 off();  
 }  
 function on() public {  
 engineState = true;  
 }  
 function off() public {  
 engineState = false;  
 }  
 function getEngineState() public view returns(bool){  
 return engineState;  
 }  
 function setSpeedUpBy(uint \_speed) public {  
 if(speed < (MAXSPEED - 10) && engineState == true)  
 speed += \_speed;  
 }  
 function setSpeedDownBy(uint \_speed) public {  
 if((speed - 10) > 0 && engineState == true)  
 speed -= \_speed;  
 }  
 function getSpeed() public view returns(uint) {  
 return speed;  
 }  
}

REMIX에서 코드를 작성하고, 버튼 테스트를 해보자.

|  |  |
| --- | --- |
| Line | 설명 |
| 4 | contract 명. 컨트랙이 2 개 포함된 경우, 파일명과 반드시 일치할 필요가 없다. |
| 5~11 | Engine을 포함하고, constructor에서 할당한다. 동일한 파일에 포함하고 있어 import문을 사용하지 않고 있다. |
| 27 ~ 32 | start()은 engineObj.on() 엔진을 켠다. 그리고 엔진 상태를 이벤트로 발생한다. |
| 53 ~ 61 | speed가 0 ~ 최대 속도 범위 내에서 속도를 증감하도록 한다. |

### 1.2.2 다른 파일의 컨트랙과 결합

상대 컨트랙을 한 파일에 포함하지 않는 경우라면, import 문으로 그 컨트랙을 포함하여야 한다. import문 다음에는 파일명을 적고 (컨트랙 명이 아니라), **현재 컨트랙을 기준으로 상대경로**를 적어준다.

import <<filename>>

상대 컨트랙을 import하게 되면 코드가 주입되기 때문에, 그 컨트랙은 별도로 배포하지 않아도 된다.

### 1.2.3 이미 배포된 컨트랙과 결합

이미 배포된 컨트랙을 포함되는 경우, 그 **주소**를 찾아서 전달하게 되고, 이 경우 new 명령어는 필요없다.

배포된 컨트랙이기 때문에, 현재 상태를 유지한채 그대로 사용이 된다는 점에 주의한다.

C1 c1 = C1(\_addressOfC1) // 배포주소를 인자로 넣어서 결합한다

## 1.3. 함수의 호출

### 1.3.1 컨트랙에서 호출

상대측 객체를 만들고 나면, 함수를 호출한다. 함수는 객체 지향에서 하는 방식으로 점연산자 dot operator를 사용하면 된다.

객체.functionMethod()

### 1.3.2 web3.js에서 호출

web3.js에서는 객체.methods를 통해 함수명, 그리고 뒤에 call() 또는 send()를 붙여 호출한다. 예를 들면 hello객체에서 함수 호출하는 코드를 보면 다음과 같다.

hello.methods.setLength(10).send() 블록체인을 수정하는 함수의 호출  
hello.methods.getLength().call() 값을 조회하는 함수의 호출

호출한 함수가 존재하지 않는 경우에는 어떻게 될까? 앞서 설명하였듯이 그런 경우에는, receive 또는 fallback 함수가 호출된다. (예를 들어, 계정주소.call.gas(200000).value(msg.value)(""), 괄호 안의 calldata가 비워 있다)

다른 컨트랙의 함수를 호출하는 경우, 컴파일해보면 gas 비용이 infinite라고 계산된다. 그 이유는 다른 컨트랙이 얼마나 gas를 사용하게 될지 모르기 때문이다. gas비용은 전송측에서 차감된다는 점에 주의하자.

## 실습: 컨트랙의 연관

### (1) new 명령어로 객체를 생성하고 구성 (composition)

매우 단순한 컨트랙 C1, C2를 만들어 보자.

C1에는 간단한 set(), get() 함수를 구현하고, C2는 이 함수들을 호출하게 된다. 컨트랙 C1과 C2는 서로 강한 관계를 가지고 있다. C2 생성자에서 C1을 가지고 있다. 또는 set() 함수를 만들어 생성 후에 필요한 시점에 C1과의 연관을 만들고 있다.

Solidity 컨트랙들을 결합하는 경우, 버전의 문제가 있을 수 있다. 일부 0.5버전에서 컴파일은 문제가 없었으나, 실행하면 일부 기능이 적절하게 수행되지 않는 경험을 했다. 일부 개발자들 사이에서 0.4.21이하에서는 문제가 없으나 그 이후에는 문제가 있다는 지적이 있다 (2018년 작성된 글 https://github.com/ethereum/solidity/issues/3969). 이미 지난 얘기이지만, 이더리움과 Solidity가 지속적으로 개발되는 중이고 변화하고 있어서, 앞으로도 문제가 될 수 있다.

REMIX에서 C1, C2의 코드를 작성하자. 하나의 파일에 C1, C2가 모두 포함되어서, import문은 사용하지 않는다.

함수들이 한 줄 코드로 완성이 되는 까닭에 어려워 보이는 코드가 없다. 그 중 특히 C2에서 getC1Address()에 주의하자. 실행한 후 출력이 C1의 배포 주소와 동일한지 확인해보자.

[파일명: src/C1C2.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract C1 {  
 uint128 v1;  
 function set(uint128 \_v1) public {  
 v1=\_v1;  
 }  
 function get() public view returns(uint128) {  
 return v1;  
 }  
 function get7() public pure returns(uint128) {  
 return 7;  
 }  
}  
  
contract C2 {  
 C1 c1;  
   
 constructor() {  
 c1=new C1(); //C2에서 C1의 객체를 생성, 저장  
 }  
 function set(uint128 \_v1) public {  
 c1.set(\_v1);  
 }  
 function get() public view returns(uint128) {  
 return c1.get();  
 }  
 function get7() public view returns(uint128) {  
 return c1.get7();  
 }  
 function getC1Address() public view returns(address) {  
 return address(c1); //C2에서 C1의 주소를 읽는다  
 }  
}

### (2) 컨트랙의 주소를 사용하여 컨트랙 결합(association)

앞서 new() 명령어는 소스 코드를 포함할 수 있는 경우에 사용하였다. 이번에는 이미 배포된 컨트랙 C1을 C2에서 호출하여 결합하여 보자.

#### (2-1) C1을 배포하고 주소를 구하기

C1은 앞서 작성한 코드를 그냥 사용한다. C2에 C1의 배포 주소를 넘겨주어야 한다. 즉 C1을 배포하고 그 주소를 알야야 한다.

[파일명: src/C1.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract C1 {  
 uint128 v1;  
 function set(uint128 \_v1) public {  
 v1=\_v1;  
 }  
 function get() public view returns(uint128) {  
 return v1;  
 }  
 function get7() public pure returns(uint128) {  
 return 7;  
 }  
}

C1을 컴파일해서 abi, 바이트 코드를 C1.json에 저장하자.

pjt\_dir> solc-windows.exe src/C1.sol --combined-json abi,bin > src/C1.json

배포해서 주소를 획득하자.

[파일명: src/C1Deploy.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs=require('fs');  
var \_str = fs.readFileSync("src/C1.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/C1.sol:C1"].abi);  
var \_abiArray = \_json.contracts["src/C1.sol:C1"].abi;  
//var \_bin = \_json.contracts.sHello2.bin;  
var \_bin = "0x"+\_json.contracts["src/C1.sol:C1"].bin;  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin})  
 .send({from: accounts[0], gas: 1000000})  
 .on('transactionHash', function(hash){  
 console.log(">>> transactionHash" + hash);  
 })  
 .on('receipt', function(receipt){  
 console.log(">>> RECEPIT hash: " + receipt.transactionHash + "\n>>> address:" + receipt.contractAddress);  
 })  
 .on('error', function(error, receipt) {  
 console.log(">>> ERROR " + error);  
 });  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

C1Deploy.js를 일괄 실행하고 주소를 출력하자.

pjt\_dir> node src/C1Deploy.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
>>> transactionHash0x1e68c99b948f52b7940f4ff02c4e62b0e60ddb5fadecebe07975f55ae6da9bcd  
>>> RECEPIT hash: 0x1e68c99b948f52b7940f4ff02c4e62b0e60ddb5fadecebe07975f55ae6da9bcd  
>>> address:0xc84381615C183B74ba3cd0cb9Ae9455d5236d586  
---> The contract deployed to: 0xc84381615C183B74ba3cd0cb9Ae9455d5236d586

### 단계 1: 컨트랙 개발 (C2)

자, 이제 C2에서 C1의 주소를 사용하여 결합하는 소스 코드를 구현해본다. C1은 이미 배포가 되었고, 그 주소를 받을 수 있는 기능이 필요하다. 이 경우 C1의 ABI를 모르면 C2를 컴파일을 할 수 없다. ABI는 함수의 호출방식을 정의하고 있어서 예를 들어 C1의 함수 c1.get7()의 ABI를 모르면, C2의 get7()을 컴파일할 수 없게 된다.

앞에서 보았던 것처럼 C1.sol, C2.sol을 동일한 파일 안에 넣지 않고 분리된 C1의 코드를 import해서 포함시킨다. 이 경우 줄03 import C1.sol이라고 해주지 않고, 현재 파일의 상대 디렉토리 형태로 적어준다. 즉 import "./C1.sol"로 쓴다.

|  |  |  |
| --- | --- | --- |
| 구분 | 사용 예 | 이유 |
| 올바른 사용 | import "./C1.sol" | 동일한 디렉토리라 하더라도, "./"를 파일 앞에 붙여서 사용한다. |
| 올바르지 않은 사용 | import "C1.sol" | 상대 디렉토리가 아니라서 오류가 발생 |

그리고 줄09의 setC1(C1의 주소) 함수를 통해 앞서 배포해서 획득한 C1의 주소를 설정한다. new 명령어는 쓰지 않고 주소만으로 객체를 생성한다는 것에 주의하자. 줄21 getC1Address() 함수에서 그 주소를 읽어서 확인할 수 있다.

[파일명: src/C2.sol]  
줄01 //SPDX-License-Identifier: GPL-3.0-or-later  
줄02 pragma solidity ^0.8.0;  
줄03 import "./C1.sol";  
  
줄04 contract C2 {  
줄05 C1 c1;  
줄06 constructor() {  
줄07 c1=new C1();  
줄08 }  
줄09 function setC1(address \_addressOfC1) public {  
줄10 c1 = C1(\_addressOfC1);  
줄11 }  
줄12 function set(uint128 \_v1) public {  
줄13 c1.set(\_v1);  
줄14 }  
줄15 function get() public view returns(uint128) {  
줄16 return c1.get();  
줄17 }  
줄18 function get7() public view returns(uint128) {  
줄19 return c1.get7();  
줄20 }  
줄21 function getC1Address() public view returns(address) {  
줄22 return address(c1);  
줄23 }  
줄24 }

### 단계 2: 컴파일 (C2)

C2를 컴파일한다. C2가 C1을 사용하고 있고, C1의 ABI, 바이트 코드는 포함된다.

pjt\_dir> solc-windows.exe src/C2.sol --combined-json abi,bin > src/C2.json

### 단계 3: 배포

[파일명: src/C2Deploy.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs=require('fs');  
var \_str = fs.readFileSync("src/C2.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/C2.sol:C2"].abi);  
var \_abiArray = \_json.contracts["src/C1.sol:C1"].abi;  
//var \_bin = \_json.contracts.sHello2.bin;  
var \_bin = "0x"+\_json.contracts["src/C2.sol:C2"].bin;  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin})  
 .send({from: accounts[0], gas: 1000000})  
 .on('transactionHash', function(hash){  
 console.log(">>> transactionHash" + hash);  
 })  
 .on('receipt', function(receipt){  
 console.log(">>> RECEPIT hash: " + receipt.transactionHash + "\n>>> address:" + receipt.contractAddress);  
 })  
 .on('error', function(error, receipt) {  
 console.log(">>> ERROR " + error);  
 });  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
deploy()

배포 단계에서는 C1을 따로 지정하지 않는다. 실행하면 컨트랙 주소 등을 출력하고 있다.

pjt\_dir> node src/C2Deploy.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
>>> transactionHash0xe6be9e256fa1f82f7aa305b05579662d916abd0b9223ce1c6b9ff991852cf5c3  
>>> RECEPIT hash: 0xe6be9e256fa1f82f7aa305b05579662d916abd0b9223ce1c6b9ff991852cf5c3  
>>> address:0x3B7A1c3e7C223eDd963eE045DaCf4A6860164Cf8  
---> The contract deployed to: 0x3B7A1c3e7C223eDd963eE045DaCf4A6860164Cf8

### 단계 4: 사용

이제 C2의 API를 호출해보자. 기억하자! C1의 주소를 설정하고, 확인하기 위해 이렇게 돌아오고 있다. 코드를 설명하고 있으니, setC1(), getC1Address()를 잘 살펴보자.

|  |  |  |
| --- | --- | --- |
| 줄 | 함수 | 설명 |
| 17 | c2.methods.getC1Address().call() | 생성자에서 new 명령어로 생성된 C1의 주소를 출력. 단, C2를 배포하고 첫 회 실행할 때만 유효하고, 다음에 실행될 때부터는 이전에 실행된 setC1()의 결과인 이전 C1의 주소가 출력된다. |
| 20 | c2.methods.get7().call() | await로 하지 않으면 실행 순서가 늦춰질 수 있다. |
| 22 | c2.methods.setC1() | 생성자에서 설정한 C1을 제거하고, 위에서 블록체인에 배포한 C1의 주소를 사용하여 교체한다. |

[파일명: src/C2Use.js]  
var Web3=require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs=require('fs');  
var \_str = fs.readFileSync("src/C2.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/C2.sol:C2"].abi);  
var \_abiArray = \_json.contracts["src/C2.sol:C2"].abi;  
  
var c2 = new web3.eth.Contract(\_abiArray, "0x3B7A1c3e7C223eDd963eE045DaCf4A6860164Cf8");  
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);  
 console.log("--- new C1 ---");  
 await c2.methods.getC1Address().call(function(err, c1addr) {  
 if(!err) console.log("c1 address by 'new': "+c1addr);  
 });  
 c2.methods.get7().call().then(function(res) { console.log("get7(): "+res) });  
 console.log("--- set the above deployed address of C1 ---");  
 await c2.methods.setC1("0xc84381615C183B74ba3cd0cb9Ae9455d5236d586").send({from:accounts[0], gas:50000});  
 await c2.methods.getC1Address().call(function(err, c1addr) {  
 if(!err) console.log("c1 address by 'setC1()': "+c1addr);  
 });  
 c2.methods.get7().call().then(console.log);  
 await c2.methods.set(222).send({from: accounts[0],gas:50000});  
 c2.methods.get().call().then(console.log);  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));  
 //hello.methods.kill().send({from: accounts[0]})  
}  
  
doIt()

new명령어를 사용하거나, 주소를 사용해서 설정을 하거나 원하는 기능이 올바르게 수행되고 있다.

pjt\_dir> node src/C2Use.js  
  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Balance before: 999937537163999987456  
--- new C1 ---  
c1 address by 'new': 0xc84381615C183B74ba3cd0cb9Ae9455d5236d586  
--- set the above deployed address of C1 ---  
get7(): 7  
c1 address by 'setC1()': 0xc84381615C183B74ba3cd0cb9Ae9455d5236d586  
7  
Balance after: 999937429911999987456  
Balance diff: 107251999965184  
222

편의상 하나의 파일에서 모두 다 하고 있어, 그냥 재호출하기로 한다.

pjt\_dir> node src/C2Use.js  
  
Account: 0xAD4c0912D2562b7072780A2F0FB2749D749B14fB  
Balance before: 99945866560000000000  
--- new C1 ---  
c1 address by 'new': 0x6BA98F59D5E3dDd9a3A38727e04dFDC4C4bE9D16  
7  
--- set the above deployed address of C1 ---  
c1 address by 'setC1()': 0xc71cC556bf5C9c4694062BF4752768F65d9349Dc  
7  
222  
Balance after: 99944817720000000000  
Balance diff: 1048839999995904

2회를 실행하면 아래에서 보는 것처럼 C1의 주소가 같아진다. 2회부터는 이전에 실행된 setC1()의 결과인 이전 C1의 주소로 설정되어 출력된다. 생성자가 최초에만 호출되기 때문이다.

다시 생성자를 호출하여 C1의 주소를 설정하려면, C2를 다시 배포한 후 그 주소로 C2Use.js를 실행하면 된다.

pjt\_dir> node src/C2Use.js  
  
Account: 0xAD4c0912D2562b7072780A2F0FB2749D749B14fB  
Balance before: 99944817720000000000  
--- new C1 ---  
c1 address by 'new': 0xc71cC556bf5C9c4694062BF4752768F65d9349Dc  
7  
--- set the above deployed address of C1 ---  
c1 address by 'setC1()': 0xc71cC556bf5C9c4694062BF4752768F65d9349Dc  
7  
222  
Balance after: 99943864880000000000  
Balance diff: 952839999995904

## 실습: 사각형과 면적의 구성

사각형과 면적을 구현해보자. 면적에서 사각형을 포함하기 위해 import문을 사용하기로 한다. 사각형을 사전에 배포하고 changeSquare(address \_addressOfSquare) 함수를 호출하여, 앞서 획득한 Square.sol의 주소로 '중도에' 교체할 수 있다.

### 단계 1: 컨트랙 개발

#### Square

정사각형은 한 변의 길이만 가지면 충분하다. 그리고 각도는 당연하지만 90도를 반환하도록 구현한다. 그리고 곧 '면적' 컨트랙을 구현하여, 정사각형을 구성(composition) 관계로 가지고, 면적을 계산하자.

Square.sol은 별도로 파일에 저장해서, Area.sol에서 포함하도록 한다.

[파일명: src/Square.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Square {  
 uint128 length;  
 function getLength() public view returns(uint128) {  
 return length;  
 }  
 function setLength(uint128 \_length) public {  
 length=\_length;  
 }  
 function getDegree() public pure returns(uint128) {  
 return 90;  
 }  
}

Square.sol은 별도로 컴파일하지 않아도 된다. 단, 당연한 설명이겠지만, Square의 배포 주소가 따로 필요하다면 컴파일, 배포를 해 주어야 한다. 그렇다 하더라도, REMIX에서 버튼 테스트를 실행하여 작동이 올바르게 되는지 확인하자.

#### Area

Square를 import할 때는 컴파일하는 Area.sol 기준으로 상대 경로 import "./Square.sol"로 적어준다. \* getAddressOfSquare() 정사각형의 주소를 출력하는 함수를 구현한다. Square, Area 두 개의 컨트랙이 한 파일에 있고 우리는 Area 컨트랙만 배포한다. 그럼에도 불구하고 Square의 주소를 획득할 수 있다는 점을 유의하자. \* changeSquare(address \_addressOfSquare) 별도로 사각형을 배포한 후, 그 사각형의 주소로 변경할 수 있다.

[파일명: src/Area.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
import "./Square.sol";  
  
contract Area {  
 Square s;  
 address owner;  
 //event PrintLog(uint128);  
 //function Area() public { //0.4.21  
 constructor() { //constructor() public {  
 s = new Square();  
 owner = msg.sender;  
 }  
 function changeSquare(address \_addressOfSquare) public {  
 s=Square(\_addressOfSquare);  
 }  
 function calcArea() view public returns(uint128) {  
 uint128 length = s.getLength();  
 uint128 area = length\*length;  
 //emit PrintLog(area);  
 return area;  
 }  
 function setLength(uint128 \_length) public {  
 s.setLength(\_length);  
 }  
 function getLength() public view returns(uint128) {  
 return s.getLength();  
 }  
 function getDegree() public view returns(uint128) {  
 return s.getDegree();  
 }  
 function getAddressOfSquare() public view returns(address) {  
 return address(s);  
 }  
}

물론 Area.sol만 구현하고, Area를 컴파일한다면 어떻게 될까? 예상하겠지만, Square를 찾지 못해서 not found Square.sol 오류가 발생한다. REMIX에서도 마찬가지이다.

### 단계 2: 컴파일

Square의 abi, bin를 필요하지 않는다. import 문으로 포함했으니, 그냥 Area.sol만 컴파일하면 된다.

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

### 단계 3: 컨트랙 배포

컴파일하고 Area.sol의 abi, bin만을 넣어준다. Square.sol의 abi,bin은 무시한다.

[파일명: src/AreaDeploy.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs = require('fs');  
var \_str = fs.readFileSync("src/Area.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/Area.sol:Area"].abi);  
var \_abiArray = \_json.contracts["src/Area.sol:Area"].abi;  
//var \_bin = \_json.contracts.sHello2.bin;  
var \_bin = "0x" + \_json.contracts["src/Area.sol:Area"].bin;  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin})  
 .send({from: accounts[0], gas: 1000000}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
  
deploy()

pjt\_dir> node src/AreaDeploy.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x029823d5e240701347231ab1774afc4348f337ef0f9b483979392aa48b76af69  
---> The contract deployed to: 0x01Dc10B774D545926CAC4595aF91Cd4Df11923aB

### 단계 4: 사용

|  |  |
| --- | --- |
| 함수 | 설명 |
| HttpProvider("http://localhost:8345")) | 이벤트를 출력하는 것은 이미 해보았고, 여기서는 사각형 주소 교체가 주목적이므로 그냥 Http를 사용하자. |
| getAddressOfSquare() | 생성자에서 할당된 Square 주소, 즉 import ./Square.sol에서 가져온 new Square()의 주소. 주소 없이 new Square()라고 해도 문제없이 실행이 된다. 즉, import문을 사용하면 주소없이 컨트랙을 생성해서 사용할 수 있다는 의미이다. |
| setLength() | 길이 설정 |
| calcArea() | 설정된 길이로 면적을 계산 |
| changeSquare('0xFD21931acdCccA516682cC853eD03257515302d9') | 주소를 재설정. 위에서 주소를 넣는다. 즉, 앞서 SquareArea.sol을 컴파일하고 getAddressOfSquare()에서 구한 Square의 주소를 넣어 Square를 교체하여 보자. 단 주소에 따옴표를 해야 한다. 그러고 나서 기능이 적절히 수행되는지 확인해 보자. |

[파일명: src/AreaUse.js]  
var Web3=require('web3');  
var web3=new Web3(new Web3.providers.HttpProvider("http://localhost:8345")); //nok  
//var web3 = new Web3(new Web3.providers.WebsocketProvider("http://117.16.44.45:8345")); //ok  
//var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://117.16.44.45:8345")); //ok  
var fs=require('fs');  
var \_str = fs.readFileSync("src/Area.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.Area.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/Area.sol:Area"].abi);  
var \_abiArray = \_json.contracts["src/Area.sol:Area"].abi;  
  
async function doIt() {  
 var hello = new web3.eth.Contract(\_abiArray, "0x01Dc10B774D545926CAC4595aF91Cd4Df11923aB");  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account: " + accounts[0]);  
 const balanceBefore = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance before: " + balanceBefore);  
 // do by the Square address as set in the constructor  
 hello.methods.getAddressOfSquare().call(function(err, c1addr) {  
 if(!err) console.log(">> Square address by 'new': "+c1addr);  
 });  
 await hello.methods.setLength(10).send({from: accounts[0]});  
 hello.methods.getLength().call().then(console.log);  
 hello.methods.calcArea().call().then(console.log);  
 hello.methods.getDegree().call().then(console.log);  
 hello.methods.getAddressOfSquare().call().then(console.log);  
 //redo by the Square address as changed by changeSquare()  
 await hello.methods.changeSquare('0xFD21931acdCccA516682cC853eD03257515302d9').send({from: accounts[0]});  
 hello.methods.getAddressOfSquare().call(function(err, c1addr) {  
 if(!err) console.log(">> Square address by 'changeSquare: "+c1addr);  
 });  
 await hello.methods.setLength(10).send({from: accounts[0]});  
 hello.methods.getLength().call().then(console.log);  
 hello.methods.calcArea().call().then(console.log);  
 hello.methods.getDegree().call().then(console.log);  
 hello.methods.getAddressOfSquare().call().then(console.log);  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("Balance after: " + balanceAfter);  
 console.log("Balance diff: " + (balanceBefore - balanceAfter));  
}  
  
doIt()

* 첫 번째 getAddressOfSquare() 함수는 new 명령어로 생성된 사각형의 주소를 출력한다.
* changeSquare() 후 두 번째 getAddressOfSquare()는 web3에서 주입한 사각형의 주소(앞의 예제에서 획득)를 출력한다.

pjt\_dir> node src/AreaUse.js  
  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
Balance before: 999931522283999987456  
>> Square address by 'new': 0x42ec439A7A362D2BD4FFfd14302fF65741C15Db9  
0x42ec439A7A362D2BD4FFfd14302fF65741C15Db9  
90  
10  
100  
>> Square address by 'changeSquare: 0xFD21931acdCccA516682cC853eD03257515302d9  
Balance after: 999931304999999987456  
Balance diff: 217283999956992  
0xFD21931acdCccA516682cC853eD03257515302d9  
90  
10  
100

다시 한 번 실행해보자. 사각형의 주소에 어떤 변화가 있는지 살펴보자. 생성자는 최초에만 실행되므로, Square의 주소가 변경된 상태로 실행된다.

pjt\_dir> node src/AreaUse.js  
  
Account: 0x0A2aca05EB30707F09C883A4b1881F775ACA4Fa8  
Balance before: 99954709760000000000  
>> Square address by 'new': 0x07cE6901B343abA0Fabe1098E360F7503De920cD  
10  
100  
90  
0x07cE6901B343abA0Fabe1098E360F7503De920cD  
>> Square address by 'changeSquare: 0x748929714418AaF173Cc14cB269BA246573a71eD  
10  
100  
90  
0x748929714418AaF173Cc14cB269BA246573a71eD  
Balance after: 99952778820000000000  
Balance diff: 1930939999993856

# 2. 상속

## 2.1 상속 is-a 관계

Solidity는 객체 지향 프로그래밍의 상속을 사용할 수 있다. 객체 지향 프로그래밍의 상속처럼 Solidity에서도 상속하면, 자식(child)이 부모(parent)의 private을 제외한 멤버 속성과 멤버 함수를 물려받으면서, 이를 확장하거나 개선하게 된다.

단일 상속과 다중 상속을 모두 지원한다.

Solidity에서 A가 B로부터 상속 받는 관계를 A is B 형식으로 표현한다.

contract A is B {  
 ... // 컨트랙 코드  
}

자바에서 사용하는 super, this 명령어는 사용할 수 없다. this는 다른 용도로 사용되며, external 함수를 내부에서 호출할 때 사용된다.

## 실습: 점과 사각형의 상속

### 단계 1: 컨트랙 개발

#### 부모 컨트랙 Point

점은 좌표 x, y를 가진다. 이 좌표를 출력하는 getX(), getY()를 멤버함수로 가진다.

[파일명: src/Point.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
contract Point {  
 int x;  
 int y;  
 constructor(int \_x, int \_y) {  
 x = \_x;  
 y = \_y;  
 }  
 function getX() view public returns(int) { return x; }  
 function getY() view public returns(int) { return y; }  
}

#### 자식 컨트랙 Rectangle

상속할 부모 컨트랙을 import 한다. 컨트랙 명이 아니라 파일 이름을 경로와 함께 적어주는 것에 주의한다.

사각형은 점을 상속받는다. 상속 관계를 앞에서 보인 것처럼 contract Rectangle is Point로 나타낸다.

생성자는 좌표와 너비, 높이를 가진다. 사각형의 생성자로 전달되는 좌표는 부모의 생성자로 전달되어야 한다. Solidity에서 부모의 생성자를 호출하는 것은 자식의 생성자 함수 헤더 옆에 부모의 생성자를 쓰면 된다. 자바에서와 같이 super 명령어로 부모생성자를 호출하지 못한다.

다음은 Rectangle의 생성자에서 부모인 Point의 생성자를 호출한다. 이 때 자식의 생성자에 전달받은 매개 변수 \_x, \_y값들을 부모의 생성자에 전달하고 있다.

constructor(int \_w, int \_h, int \_x, int \_y) Point(\_x, \_y)

Rectangle의 생성자는 매개 변수를 가지고 있으므로, 컨트랙을 배포할 때 생성자의 매개 변수로 전달할 값들을 넣어주어야 한다.

부모 함수의 호출은 물려받은 getX(), getY() 함수를 그냥 호출하면 된다.

[파일명: src/Rectangle.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
import "./Point.sol"; //컨트랙의 경로포함 파일명을 적는다  
  
contract Rectangle is Point {  
 int w;  
 int h;  
 constructor(int \_w, int \_h, int \_x, int \_y) Point(\_x, \_y) {  
 //super(\_x, \_y); //이렇게 하지 않는다  
 w = \_w; //this.w 이렇게 하지 않는다  
 h = \_h;  
 }  
 function getPerimeter() view public returns(int) {  
 return 2\*(w+h);  
 }  
 function getXOpposite() view public returns(int) { return getX() + w; }  
 function getYOpposite() view public returns(int) { return getY() + h; }  
}

### 단계 2: 컴파일

부모와 자식 중에서 어떤 컨트랙을 컴파일하면 될지 생각해보자. 자식 컨트랙은 부모를 import하므로 컴파일할 때 그 코드도 포함하게 된다. 따라서 자식인 Rectangle 컨트랙만 다음과 같이 컴파일하면 된다.

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

### 단계 3: 컨트랙 배포

배포할 때도 자식 컨트랙에 부모 컨트랙은 이미 포함되므로 자식의 abi, bin만 넣어서 배포한다.

생성자가 문자열 매개 변수 4개를 가지고 있으므로, 코드의 (1)처럼 4개의 인자를 넣어준다.

.deploy({data: \_bin, arguments: [10, 20, 30, 40]})

[파일명: src/RectangleDeploy.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs = require('fs');  
var \_str = fs.readFileSync("src/Rectangle.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/Rectangle.sol:Rectangle"].abi);  
var \_abiArray = \_json.contracts["src/Rectangle.sol:Rectangle"].abi;  
//var \_bin = \_json.contracts.sHello2.bin;  
var \_bin = "0x" + \_json.contracts["src/Rectangle.sol:Rectangle"].bin;  
  
//unlock the account with a password provided  
//web3.personal.unlockAccount(web3.eth.accounts[0],'password');  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin, arguments: [10, 20, 30, 40]}) // (1) 생성자의 인자를 넣어준다.  
 .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()

pjt\_dir> node src/RectangleDeploy.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x6e3c0f430b2f745c1cf785f9f28a490b4358660866dbccdbe721d7389496b151  
---> The contract deployed to: 0xA6D9e53c4E10ae1Af52F647C417E1F41eBEe9B1A

### 단계 4: 사용

[파일명: src/RectangleUse.js]  
var Web3=require('web3');  
var web3=new Web3(new Web3.providers.HttpProvider("http://localhost:8345")); //ok  
//var web3 = new Web3(new Web3.providers.WebsocketProvider("http://localhost:8345")); //ok  
//var web3 = new Web3(new Web3.providers.WebsocketProvider("ws://localhost:8345")); //ok  
var fs = require('fs');  
var \_str = fs.readFileSync("src/Rectangle.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/Rectangle.sol:Rectangle"].abi);  
var \_abiArray = \_json.contracts["src/Rectangle.sol:Rectangle"].abi;  
  
async function doIt() {  
 var rect = new web3.eth.Contract(\_abiArray, "0xA6D9e53c4E10ae1Af52F647C417E1F41eBEe9B1A");  
 const accounts = await web3.eth.getAccounts();  
 console.log("(1) Account: " + accounts[0]);  
 const balanceBefore = await web3.eth.getBalance(accounts[0]);  
 console.log("(2) Balance before: " + balanceBefore);  
 rect.methods.getPerimeter().call().then(function(res) {console.log("(3) Perimeter: "+res)});  
 rect.methods.getXOpposite().call().then(function(res) {console.log("(4) X opp: "+res)});  
 rect.methods.getYOpposite().call().then(function(res) {console.log("(5) Y opp: "+res)});  
 const balanceAfter = await web3.eth.getBalance(accounts[0]);  
 console.log("(6) Balance after: " + balanceAfter);  
 console.log("(7) Balance diff: " + (balanceBefore - balanceAfter));  
   
}  
  
doIt()

자바스크립트의 비동기적 특성으로 출력 순서가 예상과 다르다는 점에 주의한다. 배포할 때 생성자에 주었던 인자의 값으로 올바르게 출력되고 있다. \* 너비는 2 \* (10 + 20) = 60 \* XOpposite는 30 + 10 = 40 \* YOpposite는 40 + 20 = 60

pjt\_dir> node src/RectangleUse.js  
  
(1) Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
(2) Balance before: 999929177179999987456  
(6) Balance after: 999929177179999987456  
(7) Balance diff: 0  
(3) Perimeter: 60  
(5) Y opp: 60  
(4) X opp: 40

## 2.2 추상컨트랙

추상 컨트랙은 멤버 함수를 구현하지 않은 추상 함수를 최소한 한 개 이상 포함한다. 자식이 부모 컨트랙을 상속받아서 구현이 생략된 부분을 구현한다. 자식 컨트랙 역시 모든 멤버 함수를 구현하지 않고, 추상 함수를 가질 수 있다. 이런 경우에는 자식 컨트랙도 추상 컨트랙이 된다.

* 줄3 추상 컨트랙은 abstract이라고 적는다. 구현되지 않은 추상 함수가 있으면, 추상 컨트랙이 되고 객체를 생성하지 못한다.
* 줄4 추상 컨트랙의 변수는 private으로 선언하면 상속되지 않는다. virtual 키워드도 적용하지 않는다.
* 줄5 추상 함수는 virtual이라고 적는다. abstract은 컨트랙에, virtual은 함수에 적용한다. private으로 설정하면 당연히 상속도 될 수 없기 때문에 virtual로 선언될 수 없다.
* 줄7 추상 컨트랙을 상속받고 있다.
* 줄8 부모의 virtual 함수는 자식 컨트랙에서 override로 적고 부모의 함수를 재정의(override)할 수 있다. override 함수는 당연히 함수명, 매개변수의 수와 데이터타입이 똑 같이 일치해야 한다.
* 줄8 재미있게도 override와 virtual을 같이 한 함수에 동시에 적용할 수도 있다. 자신이 부모의 virtual 함수를 재정의하면서, 동시에 자신을 상속하는 자식들이 재정의 하도록 허용하는 의미가 있다.

다음 장에서 DeFi와 NFT를 만들 때 상속을 활용하게 되므로, 여기에서는 간단히 작성하기로 한다. REMIX에서 버튼을 누르면 처음에는 "Hello good morning", 다음부터는 "Hello good morning again"이 출력된다.

[파일명: src/HelloMorning.sol]  
줄01 //SPDX-License-Identifier: GPL-3.0-or-later  
줄02 pragma solidity ^0.8.0;  
  
줄03 abstract contract Hello {  
줄04 bool helloed=false;  
줄05 function sayHello() public virtual returns (string memory);  
줄06 }  
  
줄07 contract HelloMorning is Hello {  
줄08 function sayHello() public override returns (string memory) {  
줄09 string memory message;  
줄10 if(helloed)  
줄11 message = "Hello good morning again!";  
줄12 else {  
줄13 message = "Hello good morning";  
줄14 helloed=true;  
줄15 }  
줄16 return message;  
줄17 }  
줄18 }

## 2.3 인터페이스

인터페이스는 함수의 선언과 구현을 분리할 수 있는 방법으로 사용된다.

contract 명령어 자리에 interface를 적어주면 된다.

* 인터페이스는 모든 함수가 virtual이고, 함수는 구현하지 않는다.
* 인터페이스는 본래 constructor를 가지지 못한다
* 역시 state variable을 가지지 못한다. enum, struct은 가질 수 있다.

sayHello() 함수의 인터페이스를 만들어보자. 나중에 DeFi와 NFT를 만들 때 인터페이스를 적용하게 된다. 그 때 다시 자세하게 보기로 한다.

* 줄4 virtual로 선언하지 않는다. 인터페이스의 함수는 기본적으로 virtual이다. external로 가시성을 선언한다. internal 또는 private이라고 선언하면 TypeError가 발생한다. public 역시 선언하지 못한다.
* 줄6 인터페이스를 구현한다.
* 줄7 구현할 때는 오버라이드하더라도 굳이 override를 적어주지 않아도 된다 (버전 0.8.8부터)

[파일명: src/HelloImpl.sol]  
줄01 //SPDX-License-Identifier: GPL-3.0-or-later  
줄02 pragma solidity ^0.8.0;  
  
줄03 interface Hello {  
줄04 function sayHello() external returns (bytes32);  
줄05 }  
  
줄06 contract HelloImpl is Hello {  
줄07 function sayHello() pure public virtual override returns (bytes32) { return "Hello"; }  
줄08 }

## 2.4 다형성

동일한 함수가 있다고 하자. 함수명이 같고, 매개변수 타입도 동일하다. 그러나 다른 컨트랙에 속해 있기 때문에 서로 충돌하지는 않을 때, 이들 함수를 오버라이딩(overriding) 함수라고 한다.

다형성은 이런 오버라이딩 함수를 호출하여, 다른 작용을 할 수 있게 하는 것이다. 특히 실행 중에 다른 컨트랙으로 대체해서 사용할 수 있게 해준다.

아래 코드는 상속을 받아 인사하는 sayHello()를 다형적으로 호출하고 있다. 즉, 서로 동일한 부모를 두고, 동일한 함수를 오버라이딩해서 (함수명과 매개변수의 자료형이 동일) 호출하고 있다

[파일명: src/HelloPolyTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
abstract contract Hello {  
 function sayHello() public virtual returns (bytes32);  
}  
  
contract HelloMorning is Hello {  
 function sayHello() public pure override returns (bytes32) { return "Hello good morning!"; }  
}  
  
contract HelloAfternoon is Hello {  
 function sayHello() public pure override returns (bytes32) { return "Hello good afternoon!"; }   
}  
  
contract Client {  
 function getAllHellos() public returns(string memory) {  
 Hello[2] memory hellos; //no dynamic Hello[]   
 string memory \_allStr;  
   
 hellos[0] = new HelloMorning();  
 hellos[1] = new HelloAfternoon();  
   
 for(uint i = 0; i < hellos.length; i++) {  
 //bytes32 -> string -> concat  
 \_allStr=string.concat(\_allStr, string(abi.encodePacked(hellos[i].sayHello())));  
 }  
 //return \_allStr;  
 return string(abi.encodePacked(\_allStr));  
 }  
}

sayHello() 함수는 bytes32로 반환하기 때문에 string으로 변환해주어야 할 필요가 있다. 그래서 abi.encodePacked()로 만들어 주고 (32바이트 길이를 맞추기위해 0으로 채워주는 것), 문자열 연결을 하고 있다. 0.8.0이상에서는 간편하게 string.concat() 함수를 이용해서 문자열을 연결할 수 있다.

마지막에는 문자를 합쳐 32바이트로 인코딩하고 있다. 그 결과는 다음과 같다.

19 (Hello good morning!) + 13 바이트 패딩 (0으로 채움)   
Hello good morning!\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000  
  
21 (Hello good afternoon!)+ 11 바이트 패딩  
Hello good afternoon!\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000"

# 3. 라이브러리

## 3.1 라이브러리는 무엇일까?

라이브러리는 빈번하게 사용되는 서비스를 모듈화하고, 인터페이스를 잘 정의해서 재사용하기 위해 만든다.

기본으로 제공되는 라이브러리가 아직은 충분하지 않다. 다른 언어에서는 당연히 제공되지만 없어서, 때로는 인터넷을 뒤져서 누군가 개발해놓은 라이브러리를 활용해야 할 경우도 있을 것이다. 우리도 컨트랙에서 빈번하게 사용하는 코드가 있다면 라이브러리로 만들어 보는 것도 좋은 대안이 될 수 있다.

어떻게 보면 컨트랙을 작성하는 것과 많이 다르지 않지만, 몇 가지 주의해야 한다.

* 컨트랙과 달리 라이브러리는 storage 공간을 가질 수 없어서 멤버 속성을 포함할 수 없다.
* 라이브러리는 ether를 가질 수 없고, payable을 적용할 수 없고, fallback 함수 역시 가질 수 없다.
* 또한 상속도 사용할 수 없다.

라이브러리는 당연히 외부에서 사용되리라는 것을 전제로 하지만, 가시성에 따라 내장형으로 국한할 수도 있다.

* 내장형 라이브러리는 internal로 선언하여 배포하지 않아도 사용할 수 있다.
* 반면에 외부 라이브러리는 external 또는 public으로 선언하고, 배포하고 난 후 사용할 수 있다. 사용하기 위해서는 그 주소가 필요하며, 당연히 여러 컨트랙이 공유할 수 있다.

## 3.2 라이브러리 개발

라이브러리를 만들기 위해서는 다음과 같이 contract 대신 library라는 키워드를 사용해서 선언한다.

library LibrayName {  
}

컨트랙 파일에 라이브러리를 그냥 내장하여 사용할 수 있지만, 재사용하기 위해서는 별도의 파일에 저장하는 편이 좋겠다. 라이브러리 MyLib.sol 파일을 작성해보자.

* 줄3~5는 libray Library1을 코딩하고 있다.
* 줄4는 multiply7() 라이브러리 함수를 external로 선언하여 외부에서 사용할 수 있게 하고 있다.
* 줄8, 줄11은 2, 3번째 라이브러리를 바디(body)없이 선언만 하고 있다. 보통 한 파일에 라이브러리를 하나씩 가지는 것이 좋겠다.

[파일명: src/MyLib.sol]  
줄01 //SPDX-License-Identifier: GPL-3.0-or-later  
줄02 pragma solidity ^0.8.0;  
  
줄03 library Library1 {  
줄04 function multiply7(uint n) pure external returns (uint) {  
줄05 return n\*7;  
줄06 }  
줄07 }  
줄08 library Library2 {  
줄09 // 라이브러리2 코드를 여기에 적는다.  
줄10 }  
줄11 library Library3 {  
줄12 // 라이브러리3 코드를 여기에 적는다.  
줄13 }

위 라이브러리를 사용한다고 하자.

* 줄3에서 Libray2를 제외하고 나머지를 import하고 있다. import문 다음에 괄호로 라이브러리 명을 from 뒤에 상대경로의 파일 명을 적는다.
* 줄6에서 Library1.multiply7()을 호출하고 있다. 컨트랙은 객체를 생성하고, 이를 통해 함수를 호출하지만, 라이브러리의 호출은 그렇지 않다. Library1을 직접 사용하고 있다는 점에 주의하자.

[파일명: src/LibTest1.sol]  
줄1 //SPDX-License-Identifier: GPL-3.0-or-later  
줄2 pragma solidity ^0.8.0;  
  
줄3 import {Library1, Library3} from "./MyLib.sol"; //라이브러리의 경로를 적어준다  
줄4 contract LibTest1 {  
줄5 function mul(uint num) pure public returns (uint) {  
줄6 return Library1.multiply7(num); // Library1의 multiply7()함수를 호출하며 3을 전달  
줄7 }  
줄8 }

## 3.3 using ... for

라이브러리를 적용하는 방식에 변화를 줄 수 있다. using MyLib for 데이터타입는 MyLib에 있는 함수를 '데이터타입'에 붙여서 사용하겠다는 의미이다. 즉 데이터타입.함수() 형태로 사용하겠다는 뜻이다.

using을 사용하는 다음 코드를 통해 이해해보자.

* 줄3 library MyLib의 선언이다.
* 줄4 MyLib 라이브러리에 multiply7() 함수를 만들고 있다. 가시성 internal로 선언하고 있다. 공유하려면 물론 external, public으로 선언해도 된다.
* 줄9 using MyLib for uint 의 선언으로 MyLib의 함수는 uint에 부착하여 적용한다는 의미이다.
* 줄11 num은 uint 타입이다. MyLib을 적용해서 num.multiply7()라고 적는다. 잘 생각해보자, num은 multiply7()이라는 함수를 가지고 있지 않지만, 이런 문법이 어떻게 가능할까? 이유는 using ... for 코드 덕분에 MyLib.multiply7(num)으로 적지 않아도 된다. 즉 num의 uint에 대해 줄4의 라이브러리 함수 multiply7가 적용되는 것이다.

[파일명: src/LibTest2.sol]  
줄01 //SPDX-License-Identifier: GPL-3.0-or-later  
줄02 pragma solidity ^0.8.0;  
  
줄03 library MyLib {  
줄04 function multiply7(uint n) pure internal returns (uint) {  
줄05 return n\*7;  
줄06 }  
줄07 }  
  
줄08 contract LibTest2 {  
줄09 using MyLib for uint;  
   
줄10 function mul(uint num) pure public returns (uint) {  
줄11 return num.multiply7(); // MyLib의 multiply7()함수를 호출  
줄12 }  
줄13 }

데이터타입을 확장해서 적용할 수도 있다.

* 줄3에서 MyLib3 라이브러리를 선언한다.
* 줄4 multiply7() 함수를 internal 선언하고 있다.
* 줄7 add5() 함수를 internal 선언하고 있다.

그리고 나서 LibTest3는 MyLib을 사용하고 있다.

* 줄12 using MyLib3 for \*는 MyLib3 라이브러리의 함수를 어떤 데이터타입이든 적용하겠다는 의미이다.
* 이에 따라 줄14 uint형을 인자로 받는 multiply7() 함수뿐만 아니라, 줄17 int 자료형을 인자로 받는 add5()함수도 비슷한 형태로 호출하는 것을 확인할 수 있다.

잠깐, 그렇다면 줄13을 수정해서 아래의 string이나 int에게도 적용할 수 있을까? 오류가 발생하게 된다. 라이브러리에서 선언된 데이터타입과 다르기 때문이다.

function mul(string memory s) pure public returns (string memory) { return s.multiply7(); }  
function mul(int n) pure public returns (int) { return n.multiply7(); }

[파일명: src/LibTest3.sol]  
줄01 //SPDX-License-Identifier: GPL-3.0-or-later  
줄02 pragma solidity ^0.8.0;  
줄03 library MyLib3 {  
줄04 function multiply7(uint n) pure internal returns (uint) {  
줄05 return n\*7;  
줄06 }  
   
줄07 function add5(int n) pure internal returns (int) {  
줄08 return n+5;  
줄09 }   
줄10 }  
  
줄11 contract LibTest3 {  
줄12 using MyLib3 for \*;  
   
줄13 function mul(uint num) pure public returns (uint) {  
줄14 return num.multiply7();   
줄15 }  
   
줄16 function add(int num) pure public returns (int) {  
줄17 return num.add5();  
줄18 }  
줄19 }

## 실습: 내장형 연산 라이브러리

더하기, 빼기, 곱하기, 나누기 연산을 하는 라이브러리를 만들어 보자.

* add(int a, int b): 두 수를 더하는 연산
* sub(int a, int b): 두 수를 빼는 연산
* mul(int a, int b): 두 수를 곱하는 연산
* div(int a, int b): 두 수를 나누는 연산, 단 0으로 나누는 경우를 감안하자.

[파일명: src/LibraryTest.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
library MyLib {  
 function add(int a, int b) internal pure returns (int) {  
 return a+b;  
 }  
 function sub(int a, int b) internal pure returns (int) {  
 return a-b;  
 }   
 function mul(int a, int b) internal pure returns (int) {  
 if (a == 0) return 0;  
 return a \* b;  
 }  
 function div(int a, int b) internal pure returns (bool, int) {  
 if (b == 0) return (false, 0);  
 return (true, a / b);  
 }  
}  
  
contract LibrayTest {  
 using MyLib for int;  
 function add(int a, int b) pure public returns(int) { return a.add(b); }  
 function sub(int a, int b) pure public returns(int) { return a.sub(b); }  
 function mul(int a, int b) pure public returns(int) { return a.mul(b); }  
 function div(int a, int b) pure public returns(bool, int) { return a.div(b); }  
}

라이브러리는 컨트랙 파일에 포함되어 있고, internal로 선언되어 외부에서 호출할 수 없다. 라이브러리가 내부용으로 선언되었기 때문에 주소 없이 그냥 사용하면 된다. internal로 선언한 까닭에 myLib은 ABI가 없다. 생성된 JSON 파일을 열어 확인해보자.

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

## 3.4 외부에서 사용하는 라이브러리

라이브러리는 구현해서 배포해 놓으면, 필요할 때 그대로 재사용할 수 있다. 외부에서 사용할 것이라면 배포를 하고 별도의 주소만 있으면 된다.

필요하면 주소를 통해서 연결한다고 해서, **연결 라이브러리(linked library)**라고 한다. 외부에서 사용하려면, 라이브러리 함수가 **public 또는 external**로 선언해야 한다.

이미 배포된 컨트랙은 그 주소를 직접 주입해서 사용할 수 있었다. 그러나 라이브러리는 좀 독특하게 주소를 주입하고 있다.

라이브러리를 사전에 **별도로 배포해서 link**해야 한다 (Library linking). 이는 연결부에 **placeholder**를 표시해 놓게 되는데, 그 연결부에 placeholder를 표시해놓게 되는데, 그 곳에 실제 **라이브러리의 배포주소를 대체**해서 넣은 바이트코드를 배포하게 된다.

## 실습: 연결 라이브러리

### 단계 1: 컨트랙 개발

앞서 작성한 라이브러리를 그대로 사용하자. 특히 multiply7() 함수의 가시성을 public 으로 선언하고 있는지 주의하자.

[파일명: src/myLib.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
library myLib {  
 function multiply7(uint num) public pure returns (uint) {  
 return num \* 7;  
 }  
}

library를 사용하기 위해서는 import "./myLib.sol"로 포함한다. 그리고 그 사용 타입을 uint256에 대해서 using myLib for uint256 적용한다.

[파일명: src/LibraryTestPublic.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
  
import "./myLib.sol";  
  
contract LibrayTestPublic {  
 using myLib for uint256;  
 function multiply7By(uint num) public pure returns(uint) {  
 return num.multiply7();  
 }  
}

### 단계 2: 컴파일

#### 2-1 전체 컴파일

LibraryTestPublic.sol 파일에 라이브러리를 import하고 있다. 그렇다고 라이브러리를 따로 컴파일할 필요는 없다. 그냥 컨트랙만 아래와 같이 컴파일해도 된다. 그리고 나서 LibraryTestPublic.json를 열어보면 바이트 코드에 라이브러리의 주소를 넣을 수 있는 자리 placeholder를 표시해 놓는다.

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

컴파일 하고 앞서 internal 라이브러리의 abi와 바이트 코드를 서로 비교하면 차이가 있다.

* **내장형 라이브러리(embedded library)**는 라이브러리 함수가 **internal**로 선언된 경우를 말한다. 내장형 라이브러리는 컨트랙이 배포될 때도 내장, 즉 포함되기 때문에 주소를 적어야 하는 **placeholder가 없다**. 그냥 **library 코드가 컨트랙에 포함**되기 때문에 별도의 link 작업이 필요없다는 의미이다. 따라서 library를 호출할 때 JUMP문을 사용하여 보통 함수를 부르는 것처럼 처리가 된다.
* 반면에 연결 라이브러리, 즉 public/external으로 선언한 라이브러리는 컴파일하면 abi와 바이트 코드를 생성하고 있는데, LibraryTestPublic.json의 아래 출력을 보면 **placeholder**를 찾을 수 있다. 바이트코드에 \_\_$2eb0f...473de$\_\_ 이런 식으로 라이브러리주소 넣을 자리 placeholder를 마련해 놓는다. 이 문자열은 34자리, 앞 뒤 underbar 4자리, 두 자리 ampersand를 포함하여 총 **40 자리**로 구성되어 있다. 그 placeholder는 **library 명의 keccak256 hash의 hex 부호 34자리**이다.

pjt\_dir> type src\LibraryTestPublic.json  
  
{"contracts":{"src/LibraryTestPublic.sol:LibrayTestPublic":{"abi":[{"inputs":[{"internalType":"uint256","name":"num","type":"uint256"}],"name":"multiply7By","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"pure","type":"function"}],"bin":"608060405234801561001057600080fd5b50610219806100206000396000f3fe608060405234801561001057600080fd5b506004361061002b5760003560e01c80633e65296914610030575b600080fd5b61004a6004803603810190610045919061011c565b610060565b60405161005791906101a7565b60405180910390f35b60008173\_\_$2eb0fc2ba9c1b20950fd715a17616473de$\_\_"},"src/myLib.sol:myLib":{"abi":[{"inputs":[{"internalType":"uint256","name":"num","type":"uint256"}],"name":"multiply7","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"pure","type":"function"}],"bin":""}},"version":"0.8.1+commit.df193b15.Windows.msvc"}

#### 2-2 library 주소 구하기

컴파일하고 나면 import한 myLib.sol 라이브러리 abi, 바이트 코드를 뒷부분에서 발견할 수 있는데, 이를 이용해서 배포해보자.

[파일명: src/myLibDeploy.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs = require('fs');  
var \_str = fs.readFileSync("src/LibraryTestPublic.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/myLib.sol:myLib"].abi);  
var \_abiArray = \_json.contracts["src/myLib.sol:myLib"].abi;  
//var \_bin = \_json.contracts.sHello2.bin;  
var \_bin = "0x" + \_json.contracts["src/myLib.sol:myLib"].bin;  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin, arguments: ["Hello from web3"]})  
 .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()

실행하고 나면 라이브러리의 주소를 출력하게 된다. 이 주소를 위 placeholder에 넣어 연결하게 된다.

!node src/myLibDeploy.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x100335e84a7dafb3c5b31253b1e775e694aa9301ce987cc6e71e3afb04f269cf  
---> The contract deployed to: 0x19307aB5C14062A109C8cb91B2e742F15cA7707a

#### 2.3 링크

외부 라이브리러가 이미 배포되어 있거나, 동시에 작업하더라도 자리를 표시해 놓는 것은 동일하다. 이를 Library linking 작업을 통해 해야 한다. solc-windows.exe --libraries 명령어를 사용하여 라이브러리를 링크할 수 있다.

라이브러리를 링크해서 컴파일한다. solc 명령어로 링크를 하면 placeholder 그 자리에 라이브러리 주소를 넣어 연결하게 된다.

컴파일할 때 solc에 --libraries "file.sol:Math:0x1234567890123456789012345678901234567890

#### 자동

myLib은 간편하게 라이브러리 명만 넣어도 되고,

pjt\_dir> solc-windows.exe src/LibraryTestPublic.sol --libraries "myLib:0x19307aB5C14062A109C8cb91B2e742F15cA7707a" --combined-json abi,bin > src/LibraryTestPublicLink.json

또는 다음과 같이 경로 src/myLib.sol와 함께 라이브러리명 myLib을 같이 적어도 된다.

pjt\_dir> solc src/LibraryTestPublic.sol --libraries "src/myLib.sol:myLib:0x19307aB5C14062A109C8cb91B2e742F15cA7707a" --combined-json abi,bin > src/LibraryTestPublicLink\_\_.json

출력을 확인해보면, 다음에서 보듯이 (일부러 \_\_로 구분해 놓았다) 앞서 배포한 myLib 주소를 placeholder에 넣고 있다.

"608060405234801561001057600080fd5b50610219806100206000396000f3fe608060405234801561001057600080fd5b506004361061002b5760003560e01c80633e65296914610030575b600080fd5b61004a6004803603810190610045919061011c565b610060565b60405161005791906101a7565b60405180910390f35b60008173\_\_19307ab5c14062a109c8cb91b2e742f15ca7707a\_\_6350785a..."

pjt\_dir> type src\LibraryTestPublicLink.json  
  
{"contracts":{"src/LibraryTestPublic.sol:LibrayTestPublic":{"abi":[{"inputs":[{"internalType":"uint256","name":"num","type":"uint256"}],"name":"multiply7By","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"pure","type":"function"}],"bin":""},"src/myLib.sol:myLib":{"abi":[{"inputs":[{"internalType":"uint256","name":"num","type":"uint256"}],"name":"multiply7","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"pure","type":"function"}],"bin":""}},"version":"0.8.1+commit.df193b15.Windows.msvc"}

#### 수동

컴파일을 하고 나면, myLib 라이브러리가 public함수라서 binary에 placeholder가 위치하고 있다는 것을 알 수 있다. 작업순서:

* 1. 위 2.1에서 생성한 bytecode를 복사해오자.

608060405234801561001057600080fd5b50610134806100206000396000f3fe6080604052348015600f57600080fd5b506004361060285760003560e01c80633e65296914602d575b600080fd5b605660048036036020811015604157600080fd5b8101908080359060200190929190505050606c565b6040518082815260200191505060405180910390f35b60008173\_\_$2eb0fc2ba9c1b20950fd715a17616473de$\_\_6350785a4b90916040518263ffffffff1660e01b81526004018082815260200191505060206040518083038186803b15801560be57600080fd5b505af415801560d1573d6000803e3d6000fd5b505050506040513d602081101560e657600080fd5b8101908080519060200190929190505050905091905056fea2646970667358221220f42ce135fbefe1ffc4eefc7bc0320c33dbdf257910573c611e2adf7c738edb8b64736f6c63430006010033

* 1. 위 2.2에서 library myLib 배포 후 얻은 주소를 복사해오자.

0xFFd1f58a808ab6001133eA3a60e07f4991E6012e

* 1. 수작업으로 placeholder에 myLib주소 복사

위 주소에서 0x를 제거하고, placeholder에 복사해 넣자. 아래는 예시로 일부러 underbar를 남겨 놓았지만 당연히 제거되어야 한다.

608060405234801561001057600080fd5b50610134806100206000396000f3fe6080604052348015600f57600080fd5b506004361060285760003560e01c80633e65296914602d575b600080fd5b605660048036036020811015604157600080fd5b8101908080359060200190929190505050606c565b6040518082815260200191505060405180910390f35b60008173\_\_19307aB5C14062A109C8cb91B2e742F15cA7707a\_\_6350785a4b90916040518263ffffffff1660e01b81526004018082815260200191505060206040518083038186803b15801560be57600080fd5b505af415801560d1573d6000803e3d6000fd5b505050506040513d602081101560e657600080fd5b8101908080519060200190929190505050905091905056fea2646970667358221220f42ce135fbefe1ffc4eefc7bc0320c33dbdf257910573c611e2adf7c738edb8b64736f6c63430006010033

underbar를 제거하고 실제로 사용할 (library의 주소가 포함된) 바이트 코드이다.



### 단계 3: 배포

위에서 작성된 LibraryTestPublicLink.json의 abi, bin을 사용하여 배포하자.

[파일명: src/LibraryTestPublicDeploy.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs = require('fs');  
//var \_str = fs.readFileSync("src/LibraryTestPublicLink\_\_.json");  
var \_str = fs.readFileSync("src/LibraryTestPublicLink.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/LibraryTestPublic.sol:LibrayTestPublic"].abi);  
var \_abiArray = \_json.contracts["src/LibraryTestPublic.sol:LibrayTestPublic"].abi;  
//var \_bin = \_json.contracts.sHello2.bin;  
var \_bin = "0x" + \_json.contracts["src/LibraryTestPublic.sol:LibrayTestPublic"].bin;  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin, arguments: ["Hello from web3"]})  
 .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()

위 프로그램을 실행하면, 주소를 얻을 수 있다.

pjt\_dir> node src/LibraryTestPublicDeploy.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x23d4988819361344fa12399a5f89ee778ae437a17abd39222564f1017b2ff593  
---> The contract deployed to: 0x27B4E66C995AB06bF7cB20F26AfCacA654C3914B

### 단계 4: 사용

코드의 대부분은 재등장하고 있다. 라이브러리를 사용하는 .multiply7By() 함수 호출을 유의하자. 그 함수 내에서 라이브러리 .multiply7()를 불러 쓰고 있다.

[파일명: src/LibraryTestPublicUse.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs = require('fs');  
//var \_str = fs.readFileSync("src/LibraryTestPublicLink\_\_.json");  
var \_str = fs.readFileSync("src/LibraryTestPublicLink.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/LibraryTestPublic.sol:LibrayTestPublic"].abi);  
var \_abiArray = \_json.contracts["src/LibraryTestPublic.sol:LibrayTestPublic"].abi;  
  
async function doIt() {  
 var lib = new web3.eth.Contract(\_abiArray, "0x27B4E66C995AB06bF7cB20F26AfCacA654C3914B");  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account: " + accounts[0]);  
 lib.methods.multiply7By(5).estimateGas(function(err,gas) {  
 if(!err) console.log(">> gas: "+ gas);  
 });  
 lib.methods.multiply7By(5).call().then(console.log);  
}  
  
doIt()

실행하면 7 x 5 = 35 연산이 출력되고 있다. 그리고 비동기적 특성으로 인해, 앞선 코드의 gas 소요량이 뒤에 출력되고 있다.

pjt\_dir> node src/LibraryTestPublicUse.js  
  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
35  
>> gas: 25832

## 3.5 외부 라이브러리

### github 라이브러리들

Solidity는 아직 발전하고 있는 언어라서, 라이브러리가 충분하지 않다. 그럼에도 불구하고 꽤 많은 라이브러리를 외부의 제3자가 제공하고 있다. 그 중 몇 가지만 예를 들어보면:

* OpenZeppelin: https://github.com/OpenZeppelin/openzeppelin-contracts 사이트에서 유용한 라이브러리를 제공하고 있고, 우리도 다음 챕터에서 DeFi, NFT 관련 라이브러리를 사용하게 된다.
* Modular network (https://github.com/modular-network/ethereum-libraries)에서도 ArrayUtils, BasicMath, CrowdSale, LinkedList, StringUtils, Token, Vesting, Wallet 관련 라이브러리가 제공되고 있다.
* Dapp-bin by Ethereum https://github.com/ethereum/dapp-bin: IterableMapping, DoublyLinkedList, StringUtils

### github 경로 재지정

오픈 소스 리포지토리 github 사이트에 사용하고 싶은 라이브러리가 있다고 하자. 이러한 온라인 라이브러리를 직접 import할 수 있을까?

재미있게도 REMIX에서는 소스 코드를 다운로드 받아서 사용할 능력이 있으므로, URL을 그냥 적으면 가능하다.

그렇지 않고 우리가 우리가 코딩할 때는 라이브러리를 내려받아야 하고, 경로를 재정의해야 한다.

예를 들어, github.com/ethereum/dapp-bin/ 를 사용한다고 하자.

#### import 문

import문에는 다음과 같이 github.com까지 포함해서 직접 적어준다. https:// 또는 blob/master 같은 용어는 적지 않는다.

import "github.com/ethereum/dapp-bin/library/iterable\_mapping.sol" as it\_mapping;

원래의 URL은 https://github.com/ethereum/dapp-bin/blob/master/library/stringUtils.sol이다. 여기서 https://와 blob/master/ 등을 제거하고 실제 github.com/ethereum/dapp-bin/library/stringUtils.sol을 적어준다. 이 url을 웹브라우저에 입력하면 해당 페이지가 뜨면 된다.

#### 경로 재지정 remapping\*\*

컴파일할 할 경우, github 소스코드를 import하고 그 디렉토리에 사용허가 권한을 부여해야 한다.

solc 컴파일할 때 github에서 직접 파일을 가져오지 못한다. 해당 github 리포지토리를 로컬에 다운로드하고 경로를 재지정해주어야 한다.

* 1. 라이브러리 소스 코드를 git clone한다. 예를 들어, github.com/ethereum/dapp-bin/ 를 어디에든, 여기에서는 /usr/local/dapp-bin으로 clone한다고 하자.
  2. 소스의 URL을 로컬디렉토리로 대체해 주고 컴파일할 때는 다음과 같이 적어준다.

solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol

import 문에 쓰인 경로를 재지정하여 파일을 읽을 수 있게 하는 것으로, github.com/ethereum/dapp-bin/ 으로 **시작하는 선행 경로 prefix**를 /usr/local/lib/dapp-bin/로 **로컬 경로 path**로 변경하고, 단순히 해당 파일을 찾는 디렉토리를 재지정하는 것이다. 즉 prefix=path로 재지정 remapping한다.

## 실습: OpenZeppelin를 설치하고 SafeMath 사용

라이브러리 소스코드가 github에 있는 경우는 다운로드 받아서, 노드에서 설치할 수 있으면 설치해서 사용하면 된다. 이 가운데 연산 기능을 제공하는 OpenZeppelin의 SafeMath.sol을 사용해보자. SafeMath 라이브러리는 숫자형 변수의 범위를 넘치는 오버플로우(overflow)나 언더플로우(underflow) 등이 발생하는 것을 막아주는 함수들을 제공한다. https://github.com/OpenZeppelin/openzeppelin-contracts 의 contracts/utils/math로 이동해보자. 이 파일을 로컬에 저장하고 사용하자.

사용하면서 반드시 일치하는 버전을 사용하도록 하자,컴파일은 0.6 이상에서 하고, 함수는 internal로 선언되어 있다.

### 단계 1: 컨트랙 개발

* **설치** 노드에서 npm install @openzeppelin/contracts를 설치하면, node\_modules/@openzeppelin/contracts/math 아래 디렉토리에 설치되어 있다.
* **import 문** import "@openzeppelin/contracts/math/SafeMath.sol"; 이 파일은 pragma solidity ^0.5.0; 버전이 다르기 때문에 맞추어 주어야 한다. 위 @openzeppelin/contracts/math/SafeMath.sol";는 인식이 되지 않으므로 현재 컴파일하는 파일 기준의 **상대경로**를 넣어준다.
* **경로 재지정 및 접근 경로 제한 풀어주기** 상대 경로로 지정하였으므로, solc에서는 보통 해주던 것처럼 하면 된다. 단 그 경로를 사용하는 것이 보안상 제한되므로 풀어준다. solc 컴파일하는 소스 코드가 있는 경로, **재지정된 경로 외에는 접근 할 수 없도록 보안 제한**을 하게 된다. --allow-paths /sample/path,/another/sample/path 스위치를 지정한다.

[파일명: src/TestSafeMathZeppelin.sol]  
//SPDX-License-Identifier: GPL-3.0-or-later  
pragma solidity ^0.8.0;  
//pragma solidity ^0.6.0;  
  
import "../node\_modules/@openzeppelin/contracts/utils/math/SafeMath.sol";  
//import "github.com/OpenZeppelin/openzeppelin-contracts/contracts/math/SafeMath.sol";  
  
contract TestSafeMath {   
 using SafeMath for uint256;  
 function add(uint256 x, uint256 y) public pure returns (uint256) {  
 uint256 z = x.add(y);  
 return z;  
 }  
}

### 단계 2: 컴파일

보안의 이유로 Solidity는 디렉토리 접근 권한을 제한하고 있다. Error: Source "@openzeppelin/contracts/math/SafeMath.sol" not foundFile outside of allowed directories.. 이를 허용하기 위해 **--allow-paths**를 넣어준다. 뒤에 절대 경로를 넣어주면 된다.

SafeMath 경로로 import문에 사용되었던 경로 node\_modules/@openzeppelin/contracts/math/SafeMath.sol:SafeMath가 쓰이고 있다는 것을 발견할 수 있다.

pjt\_dir> solc-windows.exe src/TestSafeMathZeppelin.sol --allow-paths . --combined-json abi,bin > src/TestSafeMathZeppelin.json

pjt\_dir> type src\TestSafeMathZeppelin.json  
  
{"contracts":{"node\_modules/@openzeppelin/contracts/utils/math/SafeMath.sol:SafeMath":{"abi":[],"bin":"60566050600b82828239805160001a6073146043577f4e487b7100000000000000000000000000000000000000000000000000000000600052600060045260246000fd5b30600052607381538281f3fe73000000000000000000000000000000000000000030146080604052600080fdfea264697066735822122068115c4a0549c3053713c9a89288ba4e8cc1583d887f1810d41b7081d5e630df64736f6c63430008010033"},"src/TestSafeMathZeppelin.sol:TestSafeMath":{"abi":[{"inputs":[{"internalType":"uint256","name":"x","type":"uint256"},{"internalType":"uint256","name":"y","type":"uint256"}],"name":"add","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"stateMutability":"pure","type":"function"}],"bin":""}},"version":"0.8.1+commit.df193b15.Windows.msvc"}

### 단계 3: 배포

위 TestSafeMathZeppelin의 abi, bin을 사용해서, 객체를 생성하는데 넣어준다.

[파일명: src/TestSafeMathZeppelinDeploy.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs = require('fs');  
var \_str = fs.readFileSync("src/TestSafeMathZeppelin.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/TestSafeMathZeppelin.sol:TestSafeMath"].abi);  
var \_abiArray = \_json.contracts["src/TestSafeMathZeppelin.sol:TestSafeMath"].abi;  
//var \_bin = \_json.contracts.sHello2.bin;  
var \_bin = "0x" + \_json.contracts["src/TestSafeMathZeppelin.sol:TestSafeMath"].bin;  
  
async function deploy() {  
 const accounts = await web3.eth.getAccounts();  
 console.log("Deploying the contract from " + accounts[0]);  
 var deployed = await new web3.eth.Contract(\_abiArray)  
 .deploy({data: \_bin})  
 .send({from: accounts[0], gas: 1000000}, function(err, transactionHash) {  
 if(!err) console.log("hash: " + transactionHash);   
 })  
 console.log("---> The contract deployed to: " + deployed.options.address)  
}  
  
deploy()

실행하고 나면, 배포된 주소를 출력하고 있다.

pjt\_dir> node src/TestSafeMathZeppelinDeploy.js  
  
Deploying the contract from 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
hash: 0x3c212353dc911c8a84d0f053f6f64d4cb04f34c64c5ebab78944b6919be07d51  
---> The contract deployed to: 0x235C5eb4C4ec5A0FB30e4CdAA8AA28f9812fF67D

### 단계 4: 사용

[파일명: src/TestSafeMathZeppelinUse.js]  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8345"));  
var fs = require('fs');  
var \_str = fs.readFileSync("src/TestSafeMathZeppelin.json");  
var \_json = JSON.parse(\_str)  
//var \_abiArray = JSON.parse(\_json.contracts.sHello2.abi);  
//var \_abiArray = JSON.parse(\_json.contracts["src/TestSafeMathZeppelin.sol:TestSafeMath"].abi);  
var \_abiArray = \_json.contracts["src/TestSafeMathZeppelin.sol:TestSafeMath"].abi;  
  
async function doIt() {  
 var lib = new web3.eth.Contract(\_abiArray, "0x235C5eb4C4ec5A0FB30e4CdAA8AA28f9812fF67D");  
 const accounts = await web3.eth.getAccounts();  
 console.log("Account: " + accounts[0]);  
 lib.methods.add(11,20).estimateGas(function(err,gas) {  
 if(!err) console.log(">> gas: "+ gas);  
 });  
 lib.methods.add(11,20).call().then(console.log);  
}  
  
doIt()

라이브러리의 .add() 함수 호출이 성공적으로 실행되어 31이라는 결과가 출력되고 있다.

pjt\_dir> node src/TestSafeMathZeppelinUse.js  
  
Account: 0x9357f478d86D9222f4413bFd91C8adb0F4c728b7  
31  
>> gas: 22367

## 연습문제

1. 다음은 부모 컨트랙이다. 이 부모로부터 상속받는 자식 컨트랙 HelloWorld을 적어보세요. HelloWorld의 sayHello()는 "Hello World!"를 출력하면 된다.

* abstract contract Hello {  
  function sayHello() public virtual returns (bytes32);  
  }

1. 다음 Customer 컨트랙이 있다. 컨트랙 Order을 작성하는데, addOrder() 함수에서 Customer의 hello() 함수를 호출하는 코드를 작성하시오.

contract Customer {  
 function hello() public  
}

1. 위 Customer 컨트랙을 컨트랙 Order가 상속받는 코드를 작성하시오. 부모의 함수도 상속받을 수 있도록 수정하시오.
2. 인터페이스도 함수를 virtual로 선언할 수 있다. OX로 답하시오.
3. 인터페이스는 생성자를 가질 수 있다. OX로 답하시오.
4. 다음 인터페이스에 오류가 있는지 찾아내시오.

interface Hello {  
 bool helloed;  
 function sayHello() external returns (bytes32);  
}

1. 라이브러리는 payable로 선언할 수 있다. OX로 답하시오.
2. 아래 라이브러리가 있다고 하자.

library myLib {  
 function multiply7(uint num) public pure returns (uint) {  
 return num \* 7;  
 }  
}

* 1. myLib이 연결형인가 내부용인지 답하시오.
  2. myLib 라이브러리를 데이터타입 uint에 붙여서 사용하겠다는 코드를 작성하세요.
  3. 라이브러리를 호출하여, uint x에 대해 위 라이브러리 함수를 적용하는 코드를 작성하시오.
  4. Hello.sol 파일이 myLib을 사용한다고 하자. 컴파일하려면 어떤 스위치를 넣어야 하는지 작성해보세요.

1. Solidity는 문자열 라이브러리를 제공하지 않아서, 문자열을 자르고, 비교하는 작업을 하기 불편하다. github에 배포한 라이브러리를 사용해서, 문자열 비교를 해보자 (https://github.com/Arachnid/solidity-stringutils). 또는 NPM에서도 제공하고 있으니 설치해도 된다 npm intall solidity-stringutils