# Solidity Documentation

***출시/ 0.5.10***

**Ethereum**

**2019년 01월 11일**

# Contents

Solidity는 스마트 컨트랙트를 구현하기 위한 컨트랙트 기반의 고급 프로그래밍 언어입니다. Solidity는 C++, Python, 그리고 JavaScript의 영향을 받아 만들어졌습니다. 그리고 Ethereum Virtual Machine(EVM)에서 구동되도록 설계되었습니다.

Solidity는 정적 타입이며, 상속, 라이브러리 그리고 복잡한 사용자 정의 자료형을 지원합니다.

문서에서 살펴볼 수 있듯이 투표, 크라우드 펀딩, 블라인드 옥션, 멀티 시그 월랫 등 다양한 컨트랙트를 작성할 수 있습니다.

---

**주석:** Solidity를 연습하기 가장 좋은 방법은 현재 Remix (로딩되는데 다소 시간이 걸릴 수 있습니다.)를 사용하는 것입니다. Remix는 Solidity 스마트 컨트랙트를 작성하고, 배포하고, 실행할 수 있는 웹 브라우저 기반의 IDE입니다.

---

**경고:** 소프트웨어는 사람에 의해 만들어지기 때문에 버그가 생길 수 있습니다. 따라서 스마트 컨트랙트는 잘 알려진 모범사례들을 참고하여 작성되어야합니다. 스마트 컨트랙트를 작성할 때는 코드리뷰, 테스팅, 회고 그리고 정확성 증명을 해야합니다. 또한 사용자가 코드 작성자보다 코드를 더 신뢰하는 경우가 있다는 것을 기억해야합니다. 마지막으로, 블록체인 자체적으로 주의해야할 사항들이 있습니다. 다음 섹션을 참조해 주세요. *보안 측면 고려사항*.

CHAPTER 1

# Notice for Korean

아직 번역이 진행중입니다. 누구나 참여하실 수 있으며 해당 solidity-korea/solidity-docs-kr repo 에 편하게 Pull Request 주셔서 참여하실 수 있습니다.

CHAPTER 2

# 번역

This documentation is translated into several languages by community volunteers, but the English version stands as a reference.

- Simplified Chinese (in progress)
- Spanish
- Russian (rather outdated)

CHAPTER 3

# 유용한 링크

- Ethereum
- Changelog
- Story Backlog
- Source Code
- Ethereum Stackexchange
- Gitter Chat

CHAPTER 4

# Solidity 통합 도구들

- **Remix** 별도의 서버없이 컴파일러와 런타임 환경을 제공하는 브라우저 기반의 IDE
- **IntelliJ IDEA plugin** IntelliJ IDEA 를 위한 Solidity 플러그인 (기타 모든 JetBrains IDE 포함)
- **Visual Studio Extension** Solidity 컴파일러가 포함된 Microsoft Visual Studio 플러그인
- **Package for SublimeText — Solidity language syntax** Sublime Text 를 위한 Solidity 문법 강조기
- **Etheratom** 문법 강조, 편집, 실행 환경 (백엔드 노드 및 VM 과 호환 가능한) 을 제공하는 Atom editor 플러그인
- **Atom Solidity Linter** Solidity linting 을 제공하는 Atom editor 플러그인
- **Atom Solium Linter** Solium 기반으로, 사용자 설정이 가능한 Atom editor 용 Solidity linter
- **Solium** Solidity 에서 코드 스타일이나 보안 이슈를 수정하고 확인하기 위한 linter
- **Solhint** Smart Contract 검증을 위한 Solidity linter. 보안 사항 및 스타일 가이드, 최적의 관례 사항(역주: for-loop 에서 index 변수명을 i 로 축약하는 것 등)을 제공함.
- **Visual Studio Code extension** 문법 강조 기능과 컴파일러를 제공하는 Microsoft Visual Studio Code 플러그인
- **Emacs Solidity** 문법 강조 기능과 편집 에러 알림을 제공하는 Emacs editor 플러그인
- **Vim Solidity** 문법 강조 기능을 제공하는 Vim editor 플러그인
- **Vim Syntastic** 컴파일 확인이 가능한 Vim editor 플러그인

지원이 중지된 도구들:

- **Mix IDE** 스마트 컨트랙에 대해 디자인, 디버깅, 테스팅이 가능한 Qt 기반의 IDE
- **Ethereum Studio** 완벽한 Ethereum 환경에 대한 shell 액세스를 제공하는 특수(특화된) 웹 IDE. Specialized web IDE that also provides shell access to a complete Ethereum environment.

CHAPTER 5

# Solidity 도구들

- **Dapp** Solidity 를 위한 빌드 도구, 패키지 매니저, 배포 도우미 도구
- **Solidity REPL** 커맨드 라인 기반으로 Solidity 를 바로 사용해볼 수 있는 도구
- **solgraph** Solidity 흐름을 시각화 하고, 잠재적인 보안 위협을 강조해주는 도구
- **evmdis** Raw EVM operations 보다 높은 추상화를 제공하기 위해 바이트 코드에 직접 정적 분석을 수행하는 EVM Disassembler
- **Doxity** Solidity 를 위한 문서 생성기

CHAPTER 6

# 서드파티 Solidity 파서와 문법

- **solidity-parser** Javascript 를 위한 Solidity 파서
- **Solidity Grammar for ANTLR 4** Solidity grammar for the ANTLR 4 parser generator

CHAPTER 7

# Language Documentation

다음 페이지들 부터, Solidity 로 작성된 *간단한 smart contract* 과 blockchains. 에 대해서 알아보도록 하겠습니다.

다음 섹션은 Solidity 에서 제공하는 몇가지 유용한 *기능*을 살펴보겠습니다. *example contracts* 또한 지금 사용하시는 브라우저! 에서도 저희 코드를 실행시켜볼 수 있다는 것을 잊지마세요!

마지막 섹션에서는 Solidity 의 모든 측면에 대해서 심도 있게 다룹니다.

이외에 질문이 있으시다면, Ethereum Stackexchange 에서 검색이나 직접 질문하실 수 있으며 gitter 채널 에서도 가능합니다.

Solidity 나 이 문서에 대해 발전을 위한 아이디어는 항상 환영합니다. :)

CHAPTER 8

# Contents

Keyword Index, Search Page

## 8.1 스마트 컨트랙트 소개

### 8.1.1 간단한 스마트 컨트랙트

변수의 값을 설정하고 다른 컨트랙트에 액세스 할 수 있도록 노출시키는 기본 예제부터 시작해봅시다. 나중에 더 자세히 살펴볼 것이기 때문에 지금 모든 걸 이해하지 않아도 괜찮습니다.

#### Storage

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

첫 줄은 코드가 Solidity 0.4.0 버전을 기반으로 작성되었다는 것을 뜻하며, 이후 버전(0.6.0 버전 직전까지)에서도 정상 동작할 수 있게 합니다. 이 줄을 통해 컨트랙트가 다르게 동작 할 수 있는 새로운(깨지기 쉬운) 컴파일러 버전에서 컴파일 할 수 없도록 보장합니다. pragma 라는 키워드는 컴파일러가 소스 코드를 어떻게 처리해야하는지를 알려줍니다. (참고. pragma once).

Solidity의 관점에서 컨트랙트란 무수한 코드들(*함수*)과 데이터(*상태*)가 Ethereum 블록체인의 특정 주소에 존재하는 것입니다. 다음 줄의 `uint storedData;` 는 `uint` (*256* 비트의 부호없는 양의 *정수*) 타입의 `storedData` 로 불리는 변수를 선언한 것입니다. 이것은 데이터베이스에서 함수를 호출함으로써 값을 조회하거나 변경할 수 있는 하나의 영역으로 생각할 수 있습니다. Ethereum에서, 변수들은 컨트랙트에 포함되어 있으며 `set` 과 `get` 함수로 변수의 값을 변경하거나 조회할 수 있습니다.

상태 변수에 접근할 때 다른 프로그래밍 언어에서 일반적으로 사용되는 `this.` 키워드를 사용하지 않습니다.

이 컨트랙트는 누구나 접근 가능한 숫자를 저장하는 단순한 일 외에는 아직 할 수 있는게 많지 않습니다. 물론 누구나 `set` 을 호출하여 다른 값으로 덮어쓰는 것이 가능합니다. 하지만 이전 숫자는 블록체인 히스토리 안에 여전히 저장됩니다. 이후에, 숫자를 바꿀 수 있는 접근 제한을 어떻게 둘 수 있는지를 알아볼 것입니다.

---

**주석:** 모든 지시자(컨트랙트, 함수, 변수 이름들)는 ASCII 문자열로 제한됩니다. UTF-8로 인코딩된 데이터도 string 변수로 저장할 수 있습니다.

---

**경고:** 문자열처럼 보이는(혹은 동일한) 유니코드 텍스트 사용은 다른 코드 지점을 가지고 다른 바이트 배열로 인코딩될 수 있다는 점에 주의하세요.

### Subcurrency 예제

다음으로는 간단한 가상화폐를 만들어보겠습니다. 코인 발행은 컨트랙트를 만든 사람만이 할 수 있습니다. 코인을 전송할 땐 아이디와 비밀번호 등이 필요하지 않습니다. 오직 필요한 것은 Ethereum 키 쌍 뿐입니다.

```
pragma solidity ^0.5.0;

contract Coin {
    // The keyword "public" makes those variables
    // easily readable from outside.
    address public minter;
    mapping (address => uint) public balances;

    // Events allow light clients to react to
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    constructor() public {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
```

(continues on next page)

(이전 페이지에서 계속)

```
    }
}
```

이번 컨트랙트는 좀 다릅니다. 하나씩 차근히 살펴보죠.

`address public minter;` 로 누구나 접근 가능한 address 타입의 변수를 선언했습니다. `address` 타입은 160 비트의 값으로 그 어떤 산술 연산을 허용하지 않습니다. 이 타입은 컨트랙트 주소나 외부 사용자들의 키 쌍을 저장하는 데 적합합니다. `public` 키워드는 변수의 현재 값을 컨트랙트 바깥에서 접근할 수 있도록 하는 함수를 자동으로 만들어줍니다.

이 키워드 없이는 다른 컨트랙트가 이 변수에 접근할 방법이 없습니다. 키워드 사용 결과로 컴파일러가 자동으로 만든 함수 코드는 대강 다음과 같습니다 (당분간은 `external` 과 `view` 키워드는 무시합니다.):

```
function minter() returns (address) { return minter; }
```

물론, 위 함수를 정확하게 입력해도 이름이 같아서 제대로 동작하지는 않을 것입니다. 그러나 컴파일러가 이런 식으로 동작한다는 것을 알아두세요.

다음 줄의 `mapping (address => uint) public balances;` 또한 public 상태의 변수를 선언하지만 조금 더 복잡한 데이터 타입입니다. 이 타입은 주소와 양의 정수를 연결(매핑) 짓습니다.

매핑은 가상으로 초기화되는 해시테이블 로 볼 수 있습니다. 그래서 모든 가능한 키값은 처음부터 존재하며, 이 키값들은 바이트 표현이 모두 0인 값에 매핑됩니다. 그렇다고 모든 키와 값들을 쉽게 가져올 수 있다고 생각해서는 안 되며, 내가 추가한 게 무엇인지 알고(리스트를 유지하거나 더 나은 데이터 타입을 사용하면 더 좋습니다) 전체를 가져오지 않는 상황에서 사용해야 합니다. `public` 키워드를 통해 만들어진 *getter function* 은 조금더 복잡합니다. 대략 이런 형태인데요:

```
function balances(address _account) external view returns (uint) {
    return balances[_account];
}
```

보시는 것처럼, 특정 계좌의 잔액이 어떤지 알아내는 데 이 함수을 사용할 수 있습니다.

다음 줄의 `event Sent(address from, address to, uint amount);` 는 소위 "이벤트" 로 불리며 `send` 함수 마지막 줄에서 발생됩니다. 유저 인터페이스(서버 애플리케이션 포함) 는 블록체인 상에서 발생한 이벤트들을 큰 비용을 들이지 않고 받아볼 수 있습니다. 이벤트가 발생되었을 때 이를 받는 곳에서는 `from`, `to`, `amount` 의 인자를 함께 받으며, 이는 트랜잭션을 파악하는데 도움을 줍니다. 이벤트를 받아보기 위해, 다음의 JavaScript 코드(`Coin` 이 web3.js나 비슷한 모듈을 통해 만들어진 콘트랙트 객체라고 가정합니다) 를 사용합니다:

```
Coin.Sent().watch({}, "", function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

유저 인터페이스 상에서 자동으로 만들어진 함수 `balances` 가 어떻게 불리고 있는지 함께 알아두세요.

생성자는 컨트랙트 생성 시 실행되는 특별한 함수이고, 이후에는 사용되지 않습니다. 이것은 컨트랙트를 만든 사람의 주소를 영구적으로 저장합니다: `msg` (`tx` 와 `block` 포함)는 유용한 전역 변수로 블록체인에 접근할 수 있는 다양한 속성들을 담고 있습니다. `msg.sender` 는 외부에서 지금 함수를 호출한 주소를 나타냅니다.

마지막으로, 사용자나 컨트랙트가 호출할 수 있는 함수들은 `mint` 와 `send` 입니다. 만약 `mint` 를 호출한 사용자가 컨트랙트를 만든 사람이 아니면 아무일도 일어나지 않습니다. 이는 인수가 false로 평가될 경우 모든 변경 사항이

원래대로 되돌아가도록 하는 특수 함수 `require` 에 의해 보장됩니다. `require` 를 두 번째로 호출하면 코인이 너무 많아지게 되고, 이는 차후에 오버플로우 에러의 원인이 될 수 있습니다.

반대로 `send` 는 어디든 코인을 보낼 사람이면 (이미 이 코인을 가진) 누구나 호출 가능합니다. 전송하려고 하는 코인의 양이 충분하지 않을 경우, `require` 호출은 실패하게 되며, 적절한 에러메세지를 사용자에게 제공합니다.

---

**주석:** 코인을 전송하려고 이 컨트랙트를 사용해도 블록체인 탐색기로 본 해당 주소에는 변화가 없을 것 입니다. 코인을 보낸 것과 잔액이 변경된 사실은 이 코인 컨트랙트 내의 데이터 저장소에만 저장되어 있기 때문입니다. 이벤트를 사용하면 새 코인의 트랜잭션과 잔액을 추적하는 "블록체인 탐색기"를 만드는것이 상대적으로 쉽습니다. 하지만, 여러분은 주인의 주소가 아닌 코인 컨트랙트의 주소를 조사해야 합니다.

---

### 8.1.2 블록체인 개론

블록체인의 개념은 개발자들에게는 그리 어려운 건 아닙니다. 그 이유는 대부분의 복잡한 것들(mining, hashing, elliptic-curve cryptography, peer-to-peer networks, etc.) 은 단지 일련의 플랫폼에 대한 약속들로 정해져 있기 때문입니다. 이러한 개념들을 받아들일 때 여러분은 그 기반이 되는 기술에 대해 걱정할 필요는 없습니다. 아마존의 AWS가 내부적으로 어떻게 동작하는지를 알고 쓰는 건 아닌 것처럼 말입니다.

#### 트랜잭션

블록체인은 전세계적으로 공유되어 트랜잭션이 일어나는 데이터베이스입니다. 이것은 네트워크에 참여하면 누구나 데이터베이스를 살펴볼 수 있다는 것을 뜻합니다. 만약 여러분이 데이터베이스의 어떤 것을 변경하려고 한다면, 소위 트랜잭션을 만들어야 하며 이는 다른 모두가 동의해야만 합니다. 트랜잭션이라는 단어는 당신이 만드려는 어떤 변화(동시에 두 값을 바꾸려 할 때)가 모두 안 되었거나, 모두 되었다는 것을 뜻합니다. 그리고 여러분의 트랜잭션이 데이터베이스에 적용되는 동안 어떤 트랜잭션도 그 값을 바꿀 수 없습니다.

예를 들어, 모든 계좌의 전자 화폐 잔액을 나타내는 도표를 상상해봅시다. 한 계좌에서 다른 계좌로 이체하는 작업이 필요할 때, 데이터베이스의 트랜잭션은 한 계좌에서 돈이 빠져나갔으면 다른 계좌에 그 금액만큼 추가가 되있어야 한다는 걸 보장해야 합니다. 어떤 이유로 금액 추가가 되지 않으면 돈도 빠져나가지 않아야겠죠.

그리고 트랜잭션은 항상 만든 사용자에 의해 암호화됩니다. 그래서 데이터베이스를 직접 수정하려는 것을 차단할 수 있습니다. 전자화폐의 경우 이 간단한 검사가 계좌의 키를 소유한 사용자만이 이체할 권한을 가지는 것을 보장합니다.

#### 블록

비트코인이 극복해야할 가장 큰 장애물은 "이중 지불 공격" 입니다. 계정을 초기화할 2개의 트랜잭션이 함께 일어난다면 어떻게 될까요? 하나의 트랜잭션만이 유효할 것이고, 둘 중 처음으로 수용되는 쪽일 것입니다. 문제는 "첫 번째"가 Peer-to-Peer 네트워크에서 객관적인 용어가 아니라는 점입니다.

그에 대한 추상적인 답은 여러분이 딱히 신경 쓸 필요는 없다는 것입니다. 전반적으로 수용되는 트랜잭션들의 순서는 여러분이 설정한대로 선택되고, 이는 충돌을 해결해 줄 것입니다. 트랜잭션들은 "블록" 이라 불리는 곳에 합쳐집니다. 그리고 네트워크에 참여한 모든 노드들에 전파됩니다. 만약 두 개의 트랜잭션이 충돌한다면, 두 번째가 되는 트랜잭션은 거절될 것이며 블록의 일부가 되지 않습니다.

이러한 블록들은 시간에 따라 선형의 순서를 가진 형태를 띄며 "블록체인"의 어원이 되었습니다. 블록들은 일정한 간격에 의해 체인으로 연결됩니다. Ethereum은 약 17초마다 만들어지고요.

("채굴" 이라 불리는) "순서 선택 메커니즘"의 일환으로 블록들의 순서가 바뀌는 경우도 있는데, 이는 블록의 끝 부분에서만 일어납니다. 이런 현상은 특정 블록 위에 더 많은 블록이 생길수록 되돌릴 가능성도 점점 낮아집니다. 따라서 여러분의 트랜잭션이 블록체인에서 바뀌거나 제거되는 경우도 있지만, 시간이 지날수록 그럴 가능성은 낮아집니다.

**주석:** 트랜잭션은 다음 블록이나 향후 특정 블록을 포함하지 않을 수도 있습니다. 어떤 트랜잭션 블록이 포함될지 결정하는 것은 트랜잭션의 제출자가 아니라 채굴자에게 달려있기 때문입니다.

향후 컨트랙트 호출을 예약하길 원한다면, 알람시계 나 이와 비슷한 오라클 서비스를 사용할 수 있습니다.

### 8.1.3 Ethereum 가상 머신

#### 소개

Ethereum 가상머신, EVM은 Ethereum의 스마트 컨트랙트를 위한 런타임 환경입니다. 이것은 완전히 독립되어 있기 때문에 EVM 에서 실행되는 코드는 네트워크나 파일 시스템, 기타 프로세스들에 접근할 수 없습니다. 심지어 스마트 컨트랙트는 다른 스마트 컨트랙트에 접근이 제한적으로 불가능합니다.

#### 계정

Ethereum 내에는 같은 공간을 공유하는 2가지의 계정 종류가 있습니다: **외부 계정** 은 사람이 가지고 있는 공개키, 비밀키 쌍으로 동작되며, **컨트랙트 계정** 은 계정과 함께 저장된 코드에 의해 동작됩니다.

외부 계정의 주소는 공개키에 의해 정해지는 반면 컨트랙트의 주소는 생성되는 시점에 정해집니다. (생성한 사용자의 주소와 주소로부터 보내진 트랜잭션의 수, "논스"에 기반합니다.)

계정이 코드를 저장하든 아니든 상관없이 두 종류는 모두 EVM 내에서는 동일하게 다뤄집니다.

모든 계정은 256비트의 문자열들이 서로 키-값으로 영구히 매핑된 **스토리지** 를 가지고 있습니다. 그리고 모든 계정은 트랜잭션으로 바뀔 수 있는 Ether(정확히는 "Wei", *1 ether* 는 *10**18 wei*) 잔액을 가지고 있습니다.

#### 트랜잭션

트랜잭션은 한 계정에서 다른 계정(같을수도 있고, 비어있을 수도 있습니다. 아래 참조)으로 보내지는 일종의 메시지입니다. 그리고 바이너리 데이터("페이로드"라고 불림)와 Ether 양을 포함할 수 있습니다.

대상 계정이 코드를 포함하고 있으면 코드는 실행되고 페이로드는 입력 데이터로 제공됩니다.

대상 계정이 설정되지 않은 경우(트랜잭션에 받는 사람이 없거나 받는 사람이 `null` 로 설정된 경우) 일 땐, 트랜잭션은 **새로운 컨트랙트** 를 생성하며 앞서 말씀드렸던 것처럼 사용자와 "논스"로 불리는 트랜잭션의 수에 의해 주소가 결정됩니다. 각 컨트랙트 생성 트랜잭션 페이로드는 EVM 바이트코드로 실행되기 위해 사용됩니다. 이 실행 데이터는 컨트랙트의 코드로 영구히 저장됩니다. 즉, 컨트랙트를 만들기 위해 실제 코드를 보내는 대신, 실행될 때의 코드를 리턴하는 코드를 보내야 한다는 것을 뜻합니다.

**주석:** 컨트랙트가 생성되는 동안, 컨트랙트의 코드는 비어있습니다. 이 때문에, 생성자가 실행을 끝낼 때까지 컨트랙트를 다시 호출해서는 안됩니다.

#### 가스

트랜잭션 발생 시, 일정량의 **가스** 가 동시에 사용되며 이는 트랜잭션 실행에 필요한 작업의 양을 제한하는 목적을 가지고 있습니다. 그리고 특별한 규칙에 의해 작업 중 가스는 조금씩 고갈되게 됩니다.

**가스 가격** 은 트랜잭션을 만든 사용자가 정하고 최대 가스 가격 * 가스 양 을 지불합니다. 실행이 끝난 이후에도 가스가 남았다면 이는 같은 방식으로 사용자에게 다시 환불됩니다.

만약 가스가 모두 사용되었다면(음수가 되었다면), 가스 부족 예외 오류가 발생하며 현재 단계에서 발생하는 모든 변화를 되돌립니다.

### 스토리지, 메모리와 스택

Ethereum 가상 머신은 데이터 스토리지, 메모리, 스택이라 불리는 3가지 영역이 있습니다. 이는 다음 문단에서 설명합니다.

각 계정에는 **스토리지** 라 불리는 데이터 영역이 있습니다. 해당 영역은 함수호출과 트랜잭션 사이에서 영구적으로 존재합니다.

스토리지는 256비트 문자가 키-값 형태로 연결된 저장소입니다. 컨트랙트 내의 스토리지를 탐색하는 건 불가능하며 읽고 수정하는데 비용이 많이 듭니다. 컨트랙트가 소유하지 않은 스토리지는 읽거나 쓸 수 없습니다.

두번째 영역은 **메모리** 이며 각 메시지 콜에 대해 새로 초기화된 인스턴스를 가지고 있습니다. 메모리는 선형이며 바이트 레벨로 다뤄집니다. 쓰기가 8 비트나 256 비트가 될 수 있는 반면 읽기는 256 비트로 한정됩니다. 이전에 변경되지 않은 메모리 워드 영역(즉, 워드 내 오프셋) 에 액세스할 때(읽기, 쓰기 모두) 메모리는 256비트 워드 영역으로 확장됩니다. 확장되는 시점에 가스 비용이 지불되어야 합니다. 메모리는 커질수록 비용도 커집니다. (2차식으로 증가합니다)

EVM은 레지스터 머신이 아니라 스택 머신입니다. 모든 연산은 **스택** 이라 불리는 영역에서 처리됩니다. 최대 1024개의 요소를 가질 수 있고 256비트의 단어들을 포함합니다. 스택은 상단 꼭대기에서 접근이 일어납니다:

스택 최상단의 16개 요소들 중 하나를 최상단에 복사하거나 최상단의 요소를 밑의 16개 요소 중 하나와 교체하는 것이 가능합니다. 연산들은 스택의 최상단 2개(어떤 연산이냐에 따라 하나일수도, 더 많을수도) 를 가져오며 그 결과를 스택에 푸시합니다. 물론 더 깊은 스택의 액세스를 위해 스택 요소들을 스토리지나 메모리로 옮기는 것도 가능합니다. 하지만 스택의 상단 요소를 제거하지 않으면 그 밑에 존재하는 요소를 임의로 접근하는 건 불가능합니다.

### 명령어 집합

EVM의 명령어들은 최소로 구성되며 합의 문제를 야기할 수 있는 잘못된 구현을 방지합니다. 모든 명령어는 기본 데이터 타입, 256비트 단어나 메모리 조각(혹은 다른 바이트 배열)을 기반으로 동작합니다. 일반적인 산술, 비트, 논리, 비교 연산이 있습니다. 조건과 조건 없는 점프도 가능합니다. 그리고 컨트랙트는 현재 블록의 수나 타임스탬프 관련 속성에도 접근할 수 있습니다.

전체 목록은 인라인 어셈블리 문서 리스트를 참조하시기 바랍니다. *list of opcodes*

### 메시지 콜

메시지 콜을 사용하면 컨트랙트는 다른 컨트랙트를 호출하거나 컨트랙트가 아닌 계정으로 Ether를 송금할 수 있습니다. 메시지 콜은 송신자, 수신자, 데이터 페이로드, Ether, 가스와 리턴 값 등을 가지고 있어 트랜잭션과 유사합니다. 실제로 모든 트랜잭션은 상위 메시지 콜로 구성되며 추가 메시지 콜도 만들 수 있습니다.

컨트랙트는 내부 메시지 호출과 함께 보내고 남길 가스량을 정할 수 있습니다. 만약 내부 호출 중 가스 부족 오류(아니면 다른 오류) 가 발생하면 스택에 에러 값이 추가되며 알리게 됩니다. 이 경우 호출을 위해 사용된 가스만 소모됩니다. Solidity에서 호출하는 계약은 이런 상황에서 기본적으로 수동 예외를 발생시키므로 호출 스택의 우선순위를 올립니다.

앞서 말했듯, 호출된 컨트랙트는 깨끗이 비워진 메모리 인스턴스와 **호출 데이터** 라는 격리된 공간의 호출 페이로드 접근 권한을 가집니다. 실행이 완료되면 호출자에 의해 이미 할당된 메모리 영역 안에 저장될 데이터를 리턴받을 수 있습니다. 이런 호출은 모두 완전한 동기식입니다.

호출은 1024개의 깊이로 제한되며 이는 복잡한 연산일수록 재귀호출보다 반복문이 선호된다는 것을 뜻합니다. 게다가, 가스의 63/64 만이 메세지콜에서 포워딩 될 수 있으며, 이는 실질적으로 1000 이하의 깊이 제한 원인이 될 수 있습니다.

### 델리게이트 콜 / 콜코드와 라이브러리

메시지 콜은 다양한 변형이 있는데, **델리게이트 콜** 의 경우는 대상 주소의 코드가 호출하는 컨트랙트의 컨텍스트 내에서 실행된다는 것과 `msg.sender` 와 `msg.value` 가 값이 바뀌지 않는다는 것 외에는 메시지 콜과 동일합니다.

이것은 컨트랙트가 실행 중 다양한 주소의 코드를 동적으로 불러온다는 것을 뜻합니다. 스토리지, 현재 주소와 잔액은 여전히 호출하는 컨트랙트를 참조하지만 코드는 호출된 주소에서 가져옵니다.

이것은 Solidity에서 복잡한 데이터 구조 구현이 가능한 컨트랙트의 스토리지에 적용 가능한 재사용 가능한 코드, "라이브러리"의 구현을 가능하도록 합니다.

### 로그

블록 레벨까지의 모든 절차를 매핑하며 특별히 인덱싱된 데이터 구조 데이터를 저장하는 것도 가능합니다. 이 기능은 **로그** 라 부르며 Solidity에서 *이벤트* 를 구현하기 위해 사용됩니다. 컨트랙트들은 로그 데이터를 만들고 접근할 수는 없지만 블록체인 바깥에서 효율적으로 접근 가능합니다.

일부 로그 데이터들은 bloom filters 안에 저장되기 때문에, 효율적이고 암호화되어 안전한 방법으로 데이터를 찾는게 가능합니다. 따라서 모든 블록체인(라이트 클라이언트라고 불리는)을 다운받지 않은 네트워크 피어들도 로그들을 여전히 찾을 수 있습니다.

### 생성

컨트랙트들은 특별한 연산 부호(단순히 트랜잭션으로 0 주소를 호출하지 않습니다)를 사용하여 다른 컨트랙트들을 생성할 수 있습니다. 이러한 **생성 콜** 과 일반 메시지 콜의 차이는 페이로드 데이터가 실행된다는 것과 결과가 코드로 저장된다는 점, 호출자와 생성자가 스택의 새 컨트랙트 주소를 받는다는 점 입니다.

### 비활성화와 자기 소멸

코드가 블록체인에서 코드가 지워지는 유일한 방법은 주소의 컨트랙트가 `selfdestruct` 연산을 사용했을 때 입니다. 주소에 저장된 남은 Ether는 지정된 타겟으로 옮겨지고 스토리지와 코드는 해당 상태에서 지워집니다. 이론적으로 컨트랙트를 제거하는 것은 좋은 아이디어로 들릴지도 모르겠습니다만, 잠재적으로 위험한 행위입니다. 만일 누군가가 제거된 컨트랙트에 Ether를 전송하면, 해당 Ether는 영구적으로 손실되게 됩니다.

---

**주석:** 컨트랙트 코드가 `selfdestruct` 를 포함하지 않더라도, `delegatecall` 이나 `callcode` 를 실행해 그 작업을 수행할 수 있습니다.

---

여러분의 컨트랙트를 비활성화하려면, 내부상태를 바꿈으로써 **disable** 해야 합니다. 이때, 내부 상태는 모든 함수를 되돌리는 원인이 됩니다. 이로인해 Ether가 즉시 반환되므로 컨트랙트를 사용할 수 없게 됩니다.

**경고:** "자기 소멸자"에 의해 컨트랙트가 제거되었더라도, 블록체인의 히스토리에 남아있게됩니다. 그리고, 대부분의 Ethereum 노드들이 이를 보유하게 될 것입니다. 그래서, "자기 소멸자"를 사용하는 것은 데이터를 하드디스크에서 삭제하는 것과는 다릅니다.

## 8.2 Solidity 컴파일러 설치하기

### 8.2.1 버전 관리

Solidity 버전 관리는 semantic versioning 를 따르며 **nightly development builds** 가 배포될 수 도 있습니다. nightly builds는 작동이 보장되지 않고 최선의 노력을 다하겠지만 문서화되지 않거나 잘못된 변경사항이 있을 수 있습니다. 우리는 최신 배포판을 사용하길 권장합니다. 아래의 패키지 인스톨러는 최신판을 사용할 예정입니다.

### 8.2.2 Remix

*간단한 컨트랙트 작성과 빠른 Solidity 학습에는 Remix를 추천합니다.*

Remix 온라인에 접속하세요, 아무것도 설치 할 필요 없습니다. 인터넷 접속이 안되는 환경에서 Remix를 사용하길 원한다면, https://github.com/ethereum/remix-live/tree/gh-pages 로 가서 `.zip` 파일을 내려받으세요.

이 페이지의 다음 항목들은 커맨드라인 Solidity 컴파일러 소프트웨어를 설치하는 내용을 열거하고 있습니다. 큰 규모의 컨트랙트를 작업하거나 더 많은 편집 옵션이 필요하다면 커맨드라인 컴파일러를 사용하세요.

### 8.2.3 npm / Node.js

Solidity 컴파일러인 *solcjs* 를 설치하려면 간편한 *npm* 을 사용하세요. *solcjs* 프로그램은 현 페이지 아래에 설명 된 컴파일러에 접근하는 방법보다 더 적은 기능을 가지고 있습니다. *명령행 컴파일러 사용하기* 문서는 여러분이 모든 기능을 갖춘 컴파일러 *solc* 을 사용한다고 가정합니다. *solcjs* 의 사용법은 저장소 에 문서화 되어 있습니다.

주석: solc-js 프로젝트는 Emscripten을 사용하는 C++ *solc* 에서 파생되었습니다. solc-js와 Emscripten는 동일한 컴파일러 소스 코드를 사용합니다. *solc-js* 는 Javascript 프로젝트에서 사용될 수 있습니다(예: Remix). 자세한 내용은 solc-js 저장소를 참조하십시오.

```
npm install -g solc
```

---

**주석:** 실행가능한 커맨드라인의 이름은 *solcjs* 입니다.

*solc* 의 명령어가 *solcjs* 에서 작동하지 않듯이 *solcjs* 의 커맨드라인 옵션은 *solc* 그리고 *geth* 같은 도구들에서 호환되지 않습니다.

---

### 8.2.4 Docker

우리는 컴파일러가 포함된 최신 Docker 빌드를 제공합니다. `stable` 저장소에는 배포된 버전이 포함되어 있으며 `nightly` 저장소에는 잠재적으로 불안정한 변경사항이 포함된 개발 브랜치의 버전이 포함되어 있습니다.

```
docker run ethereum/solc:stable --version
```

현재, Docker 이미지에는 컴파일러 실행 파일만 포함되어 있습니다, 그러므로 소스와 출력 디렉터리를 연결하기 위해선 몇 가지 추가 작업을 해야 합니다.

### 8.2.5 바이너리 패키지

Solidity 바이너리 패키지는 solidity/releases 에서 이용 가능합니다.

Ubuntu에서 사용 가능한 PPA도 있습니다. 안정된 최신 버전은 아래 명령어를 통해 설치가능합니다.

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

개발중인 최신 버전을 사용하고 싶다면 아래 명령어를 이용하세요:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install solc
```

우리는 또한 snap package 를 배포하고 있습니다. 이 패키지는 지원되는 모든 Linux 배포판 에 설치할 수 있습니다. solc의 안정된 최신 버전을 설치하려면 다음 명령어를 이용하세요:

```
sudo snap install solc
```

최신 변경사항이 포함된 Solidity의 최신 개발버전을 테스트하는 데 도움을 주고 싶다면 다음을 따르세요:

```
sudo snap install solc --edge
```

개발 중인 최신 버전뿐이지만 Arch Linux 역시 패키지가 있습니다:

```
pacman -S solidity
```

우리는 Homebrew를 통해 Solidity 컴파일러를 build-from-source 버전으로 배포합니다. pre-built bottles 는 현재 지원되지 않습니다.

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

Solidity의 특정 버전이 필요한 경우, 깃허브에서 직접 Homebrew formula를 설치할 수 있습니다.

깃허브의 solidity.rb 커밋 내역 을 참조하세요.

`solidity.rb` 의 특정 커밋의 raw file 링크를 찾을 때까지 히스토리 링크를 따라가세요.

`brew` 를 사용하여 설치하십시오:

```
brew unlink solidity
# Install 0.4.8
brew install https://raw.githubusercontent.com/ethereum/homebrew-ethereum/
↪77cce03da9f289e5a3ffe579840d3c5dc0a62717/solidity.rb
```

Gentoo Linux 또한 `emerge` 를 이용해 설치할 수 있는 Solidity 패키지를 제공합니다:

```
emerge dev-lang/solidity
```

### 8.2.6 소스에서 빌드하기

#### 필수 설치 항목 - Linux

Solidity를 Linux에서 빌드 하기위해 다음 의존 항목을 설치해야 합니다:

| 소프트웨어 | 설명 |
|---|---|
| Git for Linux | Github에서 소스를 검색하기 위한 커맨드라인 툴 |

### 필수 설치 항목 - macOS

macOS의 경우, 반드시 최신 버전의 **'Xcode<https://developer.apple.com/xcode/download/>'_** 가 설치되어야 합니다. 여기에는 Clang C++ compiler, Xcode IDE 와 그 외 OS X에서 C++ 애플리케이션을 빌드하기 위한 애플 개발도구들이 포함되어 있습니다. Xcode를 처음 설치하거나 새 버전을 설치했다면, 커맨드라인에서 빌드하기 전 라이선스에 동의해야 합니다:

```
sudo xcodebuild -license accept
```

외부 의존 항목을 설치하기 위해 OSX 빌드는 Homebrew 패키지 매니저를 필요로 합니다. 혹시 처음부터 다시 시작하고 싶다면, 여기 Homebrew 삭제 하는 방법입니다.

### 필수 설치 항목 - Windows

Solidity의 Windows 빌드를 위해 아래의 의존 항목들을 설치해야 합니다:

| 소프트웨어 | 설명 |
|---|---|
| Git for Windows | Github에서 소스를 검색하기 위한 커맨드라인 도구. |
| CMake | 크로스 플랫폼 빌드 파일 생성기. |
| Visual Studio 2017 Build Tools | C++ 컴파일러 |
| Visual Studio 2017 (Optional) | C++ 컴파일러 및 개발 환경. |

IDE가 하나 뿐이고, 컴파일러와 라이브러리만 있으면, Visual Studio 2017 Build Tools를 설치할 수 있습니다.

Visual Studio 2017은 IDE와 필요한 컴파일러와 라이브러리를 모두 제공합니다. 따라서 IDE가 없는 상황에서 Solidity 개발을 생각하고 있다면, Visual Studio 2017이 모든 setup을 쉽게 해 줄 수 있는 선택이 될 것입니다.

다음은 Visual Studio 2017 Build Tools나 Visual Studio 2017에서 설치해야하는 구성요소 목록입니다.

- Visual Studio C++ core features
- VC++ 2017 v141 toolset (x86,x64)
- Windows Universal CRT SDK
- Windows 8.1 SDK
- C++/CLI support

### 저장소 복제

소스코드를 복제하기 위해서는, 아래의 명령어를 실행하세요:

```
git clone --recursive https://github.com/ethereum/solidity.git
cd solidity
```

Solidity 개발을 돕고싶다면, Solidity 프로젝트를 포크하고, 두 번째 원격 저장소로 자신의 저장소를 추가하세요:

```
git remote add personal git@github.com:[username]/solidity.git
```

### 외부 의존 항목

macOS, Windows 외 수많은 Linux 배포판에 필요한 모든 외부 의존 항목을 설치하는 도우미 스크립트가 있습니다.

```
./scripts/install_deps.sh
```

Windows에선 아래와 같습니다:

```
scripts\install_deps.bat
```

#### 커맨드라인 빌드

** 빌드하기 전 외부 의존 항목을(윗부분 참조) 반드시 설치해야 합니다.**

Solidity 프로젝트는 빌드를 구성하기 위해 CMake를 사용합니다. 반복된 빌드 속도를 높이기 위해서 ccache를 설치하는 것이 좋습니다. CMake가 자동으로 ccache를 선택할것입니다. Solidity 빌드는 Linux, macOS 및 기타 Unix에서 매우 유사하게 진행됩니다:

```
mkdir build
cd build
cmake .. && make
```

또는 조금 더 쉬운 방법:

Windows에서는:

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
```

이 명령어의 결과로 해당 빌드 디렉터리에 **solidity.sln** 가 생성됩니다. 이 파일을 더블클릭하면 Visual Studio가 실행됩니다. 우리는 **Release** 환경설정을 빌드하는 걸 제안합니다.

또 다른 방법으로는 Windows 커맨드라인에서 아래와같이 빌드를 진행할 수 있습니다:

```
cmake --build . --config Release
```

### 8.2.7 CMake 옵션

CMake 옵션을 알고 싶다면 `cmake .. -LH` 명령어를 실행하십시오.

#### STM Solvers

Solidity는 SMT solvers에 대해 빌드 될 수 있으며, 시스템에서 발견되면 디폴트로 수행될것입니다. 각 solver는 *cmake* 옵션에 의해 비활성화 될 수 있습니다.

*주석 : 경우에 따라 빌드 실패의 잠재적인 해결방법이 될 수도 있습니다.*

빌드 폴더에서는 디폴트로 사용하도록 설정되어 있기 때문에, 사용하지 않도록 설정 할 수 있습니다.

```
# Z3 SMT Solver 만 비활성화
cmake .. -DUSE_Z3=OFF

# CVC4 SMT Solver 만 비활성화
cmake .. -DUSE_CVC4=OFF

# Z3와 CVC4 모두 비활성화
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

### 8.2.8 버전 문자열 상세하게 보기

Solidity 버전 문자열은 네 부분으로 구성됩니다:

- 버전 숫자
- pre-release 태그, 대개 `develop.YYYY.MM.DD` 나 `nightly.YYYY.MM.DD` 형태를 지님
- 다음과 같은 형태의 커밋 `commit.GITHASH`
- 플랫폼 및 컴파일러에 대한 세부 정보를 포함하는 몇 가지 항목

로컬에서 수정된 부분이 있다면, 커밋 뒤에 `.mod` 가 붙습니다.

이 부분들은 Semver(Semantic Versioning)에 따라 필요에 의해 결합됩니다. 여기서 Solidity pre-release 태그는 Semver의 pre-release 태그와 같고 Solidity 커밋 및 플랫폼은 결합되어 Semver 빌드 메타데이터를 구성합니다.

release 예: `0.4.8+commit.60cc1668.Emscripten.clang`.

pre-release 예: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

### 8.2.9 버전 관리에 대한 중요한 정보

릴리즈가 일어난 후에, 패치 버전은 변경됩니다. 변경사항이 합쳐질 때, 버전은 semver와 변경 정도에 따라 변경됩니다. 따라서, 배포는 항상 `prerelease` 태그를 제외한 현재의 nightly build버전으로 이루어집니다.

예:

0. 0.4.0가 배포된다
1. 지금부터 nightly build는 0.4.1 버전이다
2. 어떠한 변경사항이 없을 경우 - 버전은 변화가 없다
3. 변경사항이 있을 경우 - 버전은 0.5.0이 된다
4. 0.5.0가 배포된다

이 동작은 *version pragma* 와 함께 작동합니다.

## 8.3 예제를 통한 솔리디티

### 8.3.1 투표

다음의 콘트렉트는 상당히 복잡하지만 solidity의 많은 특징들을 보여줍니다. 이는 투표에 관한 콘트렉트를 구현한 것입니다. 물론 전자 투표의 핵심 문제는 어떻게 투표권을 올바르게 할당할것이며 이에 대한 조작을 어떻게 예방하느냐에 대한 것입니다. 여기서 모든 문제를 해결할 수 없겠지만 적어도 우리는 이 예제를 통해서 개표가 " 자동적이고 동시에 완벽히 투명한지" 위임투표가 어떻게 이루어지는지 보여줄 것입니다.

이 아이디어는 한개의 투표 당 하나의 컨트랙트를 작성하여 각각의 선택마다 짧은 이름을 제공합니다. 그리고 의장 역할을 수행하는 컨트렉트 작성자는 각 주소에 개별적인 투표권을 부여할 것입니다.

투표권을 받은 주소의 사람들은 자신이 직접 투표하거나 자신의 투표권을 신뢰할 수 있는 다른 사람에게 위임할 수 있습니다.

투표 시간이 끝나면, ‘‘winningProposal()‘‘함수는 가장 많은 득표를 한 안건을 반환할 것입니다.

```
pragma solidity ^0.4.16;

/// @title 위임 투표.
contract Ballot {
    // 이것은 나중에 변수에 사용될 새로운
    // 복합 유형을 선언합니다.
    // 그것은 단일 유권자를 대표할 것입니다.
    struct Voter {
        uint weight; // weight 는 대표단에 의해 누적됩니다.
        bool voted;  // 만약 이 값이 true라면, 그 사람은 이미 투표한 것 입니다.
        address delegate; // 투표에 위임된 사람
        uint vote;   // 투표된 제안의 인덱스 데이터 값
    }

    // 이것은 단일 제안에 대한 유형입니다.
    struct Proposal {
        bytes32 name;   // 간단한 명칭 (최대 32바이트)
        uint voteCount; // 누적 투표 수
    }

    address public chairperson;

    // 이것은 각각의 가능한 주소에 대해
    // `Voter` 구조체를 저장하는 상태변수를 선언합니다.
    mapping(address => Voter) public voters;

    // 동적으로 크기가 지정된 `Proposal` 구조체의 배열입니다.
    Proposal[] public proposals;

    /// `proposalNames` 중 하나를 선택하기 위한 새로운 투표권을 생성십시오.
    function Ballot(bytes32[] proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // 각각의 제공된 제안서 이름에 대해,
        // 새로운 제안서 개체를 만들어 배열 끝에 추가합니다.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // `voter` 에게 이 투표권에 대한 권한을 부여하십시오.
    // 오직 `chairperson` 으로부터 호출받을 수 있습니다.
    function giveRightToVote(address voter) public {
        // `require`의 인수가 `false`로 평가되면,
        // 그것은 종료되고 모든 변경내용을 state와
        // Ether Balance로 되돌립니다.
        // 함수가 잘못 호출되면 이것을 사용하는 것이 좋습니다.
        // 그러나 조심하십시오,
        // 이것은 현재 제공된 모든 가스를 소비할 것입니다.
        // (이것은 앞으로 바뀌게 될 예정입니다)
```

(continues on next page)

(이전 페이지에서 계속)

```
        require(
            (msg.sender == chairperson) &&
            !voters[voter].voted &&
            (voters[voter].weight == 0)
        );
        voters[voter].weight = 1;
    }

    /// `to` 로 유권자에게 투표를 위임하십시오.
    function delegate(address to) public {
        // 참조를 지정하십시오.
        Voter storage sender = voters[msg.sender];
        require(!sender.voted);

        // 자체 위임은 허용되지 않습니다.
        require(to != msg.sender);

        // `to`가 위임하는 동안 delegation을 전달하십시오.
        // 일반적으로 이런 루프는 매우 위험하기 때문에,
        // 너무 오래 실행되면 블록에서 사용가능한 가스보다
        // 더 많은 가스가 필요하게 될지도 모릅니다.
        // 이 경우 위임(delegation)은 실행되지 않지만,
        // 다른 상황에서는 이러한 루프로 인해
        // 스마트 컨트랙트가 완전히 "고착"될 수 있습니다.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // 우리는 delegation에 루프가 있음을 확인 했고 허용하지 않았습니다.
            require(to != msg.sender);
        }

        // `sender` 는 참조이므로,
        // `voters[msg.sender].voted` 를 수정합니다.
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // 대표가 이미 투표한 경우,
            // 투표 수에 직접 추가 하십시오
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // 대표가 아직 투표하지 않았다면,
            // weight에 추가하십시오.
            delegate_.weight += sender.weight;
        }
    }

    /// (당신에게 위임된 투표권을 포함하여)
    /// `proposals[proposal].name` 제안서에 투표 하십시오.
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted);
        sender.voted = true;
        sender.vote = proposal;

        // 만약 `proposal` 이 배열의 범위를 벗어나면
        // 자동으로 throw 하고 모든 변경사항을 되돌릴 것입니다.
```

(continues on next page)

(이전 페이지에서 계속)

```
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev 모든 이전 득표를 고려하여 승리한 제안서를 계산합니다.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // winningProposal() 함수를 호출하여
    // 제안 배열에 포함된 승자의 index를 가져온 다음
    // 승자의 이름을 반환합니다.
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

### 개선가능 한 사안들

현재 모든 참가자에게 투표권을 부여하기 위해 많은 거래가 필요 합니다. 더 나은 방법을 생각해 볼 수 있습니까?

## 8.3.2 블라인드 경매

이 섹션에서는, 이더리움 블라인드 경매 콘트렉트를 완전하게 만드는것이 얼마나 쉬운지 보여줄 것입니다. 우리는 누구나 제시되어진 가격을 볼수 있는 공개 경매로 시작하여 이 콘트렉트를 입찰기간이 끝나기전까지 실제 입찰가를 보는 것이 불가능한 블라인드 경매로 확장시킬것입니다.

### 간단한 공개 경매

아래의 간단한 경매 콘트랙트에 대한 일반적인 아이디어는 경매 기간동안 모든 사람이 그들의 가격제시를 전송할 수 있다는 것입니다. 가격 제시는 이미 가격제시자(bidder)와 그들의 가격제시(bid)를 묶기위해서 돈이나 이더를 송금하는 것을 포함하고 있습니다. 만약 최고 제시 가격이 올라간다면, 그 이전의 최고 제시자는 그녀의 돈을 돌려받습니다. 가격제시 기간이 끝난후, 그 콘트렉트는 그의 돈을 받는 수혜자를 위해 수동으로 호출 되어져야만 합니다 - 콘트렉트는 그들 스스로 활성화 시킬 수 없습니다.

```
pragma solidity ^0.4.21;

contract SimpleAuction {
    // 옥션의 파라미터. 시간은 아래 둘중 하나입니다.
    // 앱솔루트 유닉스 타임스탬프 (seconds since 1970-01-01)
    // 혹은 시한(time period) in seconds.
    address public beneficiary;
    uint public auctionEnd;
```

(continues on next page)

(이전 페이지에서 계속)

```
    // 옥션의 현재 상황.
    address public highestBidder;
    uint public highestBid;

    // 이전 가격 제시들의 수락된 출금.
    mapping(address => uint) pendingReturns;

    // 마지막에 true 로 설정, 어떠한 변경도 허락되지 않습니다.
    bool ended;

    // 변경에 발생하는 이벤트
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // 아래의 것은 소위 "natspec"이라고 불리우는 코멘트,
    // 3개의 슬래시에 의해 알아볼 수 있습니다.
    // 이것을 유저가 트렌젝션에 대한 확인을 요청 받을때
    // 보여집니다.

    /// 수혜자의 주소를 대신하여 두번째 가격제시 기간 '_biddingTime'과
    /// 수혜자의 주소 '_beneficiary' 를 포함하는
    /// 간단한 옥션을 제작합니다.
    function SimpleAuction(
        uint _biddingTime,
        address _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }

    /// 경매에 대한 가격제시와 값은
    /// 이 transaction과 함께 보내집니다.
    /// 값은 경매에서 이기지 못했을 경우만
    /// 반환 받을 수 있습니다.
    function bid() public payable {
        // 어떤 인자도 필요하지 않음, 모든
        // 모든 정보는 이미 트렌젝션의
        // 일부이다. 'payable' 키워드는
        // 이더를 지급는 것이 가능 하도록
        // 하기 위하여 함수에게 요구됩니다.

        // 경매 기간이 끝났으면
        // 되돌아 갑니다.
        require(now <= auctionEnd);

        // 만약 이 가격제시가 더 높지 않다면, 돈을
        // 되돌려 보냅니다.
        require(msg.value > highestBid);

        if (highestBid != 0) {
            // 간단히 highestBidder.send(highestBid)를 사용하여
            // 돈을 돌려 보내는 것은 보안상의 리스크가 있습니다.
            // 그것은 신뢰되지 않은 콘트렉트를 실행 시킬수 있기 때문입니다.
            // 받는 사람이 그들의 돈을 그들 스스로 출금 하도록 하는 것이
            // 항상 더 안전합니다.
            pendingReturns[highestBidder] += highestBid;
```

(continues on next page)

(이전 페이지에서 계속)

```
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// 비싸게 값이 불러진 가격제시 출금.
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // 받는 사람이 이 `send` 반환 이전에 받는 호출의 일부로써
            // 이 함수를 다시 호출할 수 있기 때문에
            // 이것을 0으로 설정하는 것은 중요하다.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // 여기서 throw를 호출할 필요가 없습니다, 빚진 양만 초기화.
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// 이 경매를 끝내고 최고 가격 제시를
    /// 수혜자에게 송금.
    function auctionEnd() public {

        // 이것은 다른 콘트렉트와 상호작용하는 함수의 구조를 잡는 좋은 가이드 라인입니다.
        // (i.e. 그것들은 이더를 보내거나 함수를 호출합니다.)
        // 3가지 단계:
        // 1. 조건을 확인
        // 2. 동작을 수행  (잠재적으로 변경되는 조건)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could call
        // back into the current contract and modify the state or cause
        // effects (ether payout) to be performed multiple times.
        // If functions called internally include interaction with external
        // contracts, they also have to be considered interaction with
        // external contracts.

        // 1. 조건
        require(now >= auctionEnd); // auction did not yet end
        require(!ended); // this function has already been called

        // 2. 영향
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. 상호작용
        beneficiary.transfer(highestBid);
    }
}
```

### 블라인드 경매

이전의 공개 경매는 블라인드 경매로 확장 되었습니다. 블라인드 경매의 이점은 경매 기간 마감에 대한 시간 압박이 없다는 것입니다. 블라인드 경매를 트렌젝션 컴퓨팅 플렛폼에 만드는 것은 모순 되는 말 처럼 들릴지도 모르지만, 그러나 암호학이 이를 구조했습니다.

**경매 기간** 동안, 가격 제시자들은 실제로 그녀의 가격제시를 보내는 것이 아니라, 그 버전의 해쉬 버전입니다. 두 해쉬 값이 같은 두개의 (충분히 긴)값을 찾는 것은 현재 실용적으로 불가능 하다고 여겨지기 때문에, 가격 제시자들은 그것에 의해 가격을 제시합니다. 경매 기간이 끝난 후, 가격 제시자들은 그들이 가격을 제시한 것들을 드러내야만 합니다: 그들은 그들의 암호화되지 않은 값을 보내고 콘트렉트는 해쉬 값이 경매 기간동안 제공되어진 것과 같은지 확인합니다.

또다른 도전은 어떻게 경매를 **묶고 블라인드**로 동시에 만드는가 입니다 : 가격 제시자를 그가 경매에서 이긴 후에 돈을 보내는 것을 막는 유일한 방법은 그녀가 돈을 가격 제시와 함께 보내도록 하는 것입니다. 값 전달이 이더리움 안에서 블라인드가 될 수 없기 때문에, 누구든 그 값을 볼 수 있습니다.

아래의 콘트렉트는 이 문제를 가장 큰 가격제시보다 더큰 어떤 값이든 수락 함으로써 해결했습니다. 이것은 당연히 드러내는 단계에서만 확인되어 질 수 있기 때문에, 몇몇 가격 제시는 **유효하지 않을**지도 모릅니다, 그리고 이것은 고의 입니다. (그것은 심지어 높은 가치의 이전과 함께 유효하지 않은 입찰에 배치할 명시적인 깃발을 제공합니다.): 가격 제시자들은 몇몇의 높고 낮은 유효하지 않은 가격 제시를 함으로써 경쟁을 혼란 스럽게 할 수 있습니다.

```
pragma solidity ^0.4.21;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // 이전 가격 제시의 허락된 출금
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Modifier는 함수 입력값을 입증하는 편리한 방법입니다.
    /// `onlyBefore`은 아래의 'bid'에 적용 되어 질 수 있습니다:
    /// 이 새로운 함수 몸체는 `_`이 오래된 함수 몸체를 대체하는
    /// Modifier의 몸체 입니다.
    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }

    function BlindAuction(
        uint _biddingTime,
        uint _revealTime,
        address _beneficiary
    ) public {
        beneficiary = _beneficiary;
```

(continues on next page)

(이전 페이지에서 계속)

```
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    /// `_blindedBid` = keccak256(value, fake, secret)와 함께
    /// 가려진(blinded) 가격을 제시합니다.
    /// 만약 가격 제시가 드러내는 단계에서 올바르게 보여진다면
    /// 보내진 이더는 환급받을 수 만 있습니다.
    /// 가격 제시와 함께 보내진 이더는 적어도 "value"와 "fake" 는 참입니다.
    /// "fake"를 참으로 설정하고 정확하지 않은 양을 보내는 것은 진짜 가격 제시를
    /// 숨기는 방법들입니다. 그러나 여전히 요구되는 출금을 합니다. 같은
    /// 주소는 여러 가격 제시들을 둘 수 있습니다.
    function bid(bytes32 _blindedBid)
        public
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

    /// Reveal your blinded bids. You will get a refund for all
    /// correctly blinded invalid bids and for all bids except for
    /// the totally highest.
    /// 가려진 가격 제시를 드러냅니다. 너는 알맞게 가려진 유효하지 않은
    /// 가격 제시들을 되돌려 받을 것입니다. 그리고 가장 높은 가격 제시를 제외하고
    /// 모든 가격 제시도 돌려 받을 것입니다.
    function reveal(
        uint[] _values,
        bool[] _fake,
        bytes32[] _secret
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            var bid = bids[msg.sender][i];
            var (value, fake, secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bid.blindedBid != keccak256(value, fake, secret)) {
                // 가격 제시는 실제로 드러나지 않습니다.
                // Do not refund deposit.
                continue;
            }
            refund += bid.deposit;
            if (!fake && bid.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
```

(continues on next page)

(이전 페이지에서 계속)

```
            }
            // Make it impossible for the sender to re-claim
            // the same deposit.
            bid.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }

    // 이것은 이 함수가 이 콘트렉트 안에서 이것 스스로만 호출 될 수
    // 있다는 의미를 가지는 "internal" 함수입니다.
    // (혹은 파생된 콘트렉트들에서)
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != 0) {
            // 이전에 가장 높은 가격 제시를 환급
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the recipient
            // can call this function again as part of the receiving call
            // before `transfer` returns (see the remark above about
            // conditions -> effects -> interaction).
            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    /// 경매를 끝내고 가장 높은 가격 제시를 수혜자에게
    /// 송금합니다.
    function auctionEnd()
        public
        onlyAfter(revealEnd)
    {
        require(!ended);
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

### 8.3.3 Safe Remote Purchase

```
pragma solidity ^0.4.21;

contract Purchase {
    uint public value;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    // `msg.value`가 짝수임을 확실히 하세요.
    // 만약 홀수라면 분할은 길이를 줄일 것입니다.
    // 곱셈을 통해 이것이 홀수가 아닌지 확인하세요.
    function Purchase() public payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value);
    }

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(msg.sender == buyer);
        _;
    }

    modifier onlySeller() {
        require(msg.sender == seller);
        _;
    }

    modifier inState(State _state) {
        require(state == _state);
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();

    /// 구매를 중단하고 이더를 회수합니다.
    /// 콘트렉트가 잠기기 전에 판매자에 의해서만
    /// 호출 되어 질 수 있습니다.
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        seller.transfer(this.balance);
    }
```

(continues on next page)

(이전 페이지에서 계속)

```
    /// 구매자로서 구매를 확정합니다.
    /// 트렌젝션은 `2 * value` ether 을 포함해야 한다.
    /// 이 이더는 confirmReceived()가 호출 될때까지
    /// 잠길것입니다.
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// 구매자가 아이템을 받았다고 확인.
    /// 이것은 잠긴 이더를 풀어줄것입니다.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        // 먼저 상태를 변경하는 것이 중요합니다.
        // 그렇지 않으면, `send`를 사용하며 호출된 콘트렉트는
        // 다시 여기를 호출할 수 있기 때문입니다.
        state = State.Inactive;

        // NOTE: 이것은 실제로 구매자와 판매자 둘다 환급 하는 것을 막을 수
        // 있도록 합니다. - 출금 패턴이 사용되어야만 합니다.

        buyer.transfer(value);
        seller.transfer(this.balance);
    }
}
```

### 8.3.4 Micropayment Channel

앞으로 쓰여질 예정

## 8.4 솔리디티 파고들기

이 섹션에서는 솔리디티에 대해 당신이 알아야할 모든 정보를 제공합니다. 만약 누락된 정보가 있다면 Gitter 나 Github 을 통해 Pull request를 요청해 주십시오.

### 8.4.1 Layout of a Solidity Source File

Source files can contain an arbitrary number of *contract definitions*, *import* directives and *pragma directives*.

### Pragmas

The `pragma` keyword can be used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want enable it in all of your project. If you *import* another file, the pragma from that file will not automatically apply to the importing file.

### Version Pragma

Source files can (and should) be annotated with a so-called version pragma to reject being compiled with future compiler versions that might introduce incompatible changes. We try to keep such changes to an absolute minimum and especially introduce changes in a way that changes in semantics will also require changes in the syntax, but this is of course not always possible. Because of that, it is always a good idea to read through the changelog at least for releases that contain breaking changes, those releases will always have versions of the form `0.x.0` or `x.0.0`.

The version pragma is used as follows:

```
pragma solidity ^0.4.0;
```

Such a source file will not compile with a compiler earlier than version 0.4.0 and it will also not work on a compiler starting from version 0.5.0 (this second condition is added by using `^`). The idea behind this is that there will be no breaking changes until version `0.5.0`, so we can always be sure that our code will compile the way we intended it to. We do not fix the exact version of the compiler, so that bugfix releases are still possible.

It is possible to specify much more complex rules for the compiler version, the expression follows those used by npm.

---

**주석:** Using the version pragma will *not* change the version of the compiler. It will also *not* enable or disable features of the compiler. It will just instruct the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler will issue an error.

---

### Experimental Pragma

The second pragma is the experimental pragma. It can be used to enable features of the compiler or language that are not yet enabled by default. The following experimental pragmas are currently supported:

### ABIEncoderV2

The new ABI encoder is able to encode and decode arbitrarily nested arrays and structs. It produces less optimal code (the optimizer for this part of the code is still under development) and has not received as much testing as the old encoder. You can activate it using `pragma experimental ABIEncoderV2;`.

### SMTChecker

This component has to be enabled when the Solidity compiler is built and therefore it is not available in all Solidity binaries. The build instructions explain how to activate this option. It is activated for the Ubuntu PPA releases in most versions, but not for solc-js, the Docker images, Windows binaries or the statically-built Linux binaries.

If you use `pragma experimental SMTChecker;`, then you get additional safety warnings which are obtained by querying an SMT solver. The component does not yet support all features of the Solidity language and likely outputs many warnings. In case it reports unsupported features, the analysis may not be fully sound.

### Importing other Source Files

#### Syntax and Semantics

Solidity supports import statements that are very similar to those available in JavaScript (from ES6 on), although Solidity does not know the concept of a "default export".

At a global level, you can use import statements of the following form:

```
import "filename";
```

This statement imports all global symbols from "filename" (and symbols imported there) into the current global scope (different than in ES6 but backwards-compatible for Solidity). This simple form is not recommended for use, because it pollutes the namespace in an unpredictable way: If you add new top-level items inside "filename", they will automatically appear in all files that import like this from "filename". It is better to import specific symbols explicitly.

The following example creates a new global symbol `symbolName` whose members are all the global symbols from `"filename"`.

```
import * as symbolName from "filename";
```

If there is a naming collision, you can also rename symbols while importing. This code creates new global symbols `alias` and `symbol2` which reference `symbol1` and `symbol2` from inside `"filename"`, respectively.

```
import {symbol1 as alias, symbol2} from "filename";
```

Another syntax is not part of ES6, but probably convenient:

```
import "filename" as symbolName;
```

which is equivalent to `import * as symbolName from "filename";`.

#### Paths

In the above, `filename` is always treated as a path with `/` as directory separator, `.` as the current and `..` as the parent directory. When `.` or `..` is followed by a character except `/`, it is not considered as the current or the parent directory. All path names are treated as absolute paths unless they start with the current `.` or the parent directory `..`.

To import a file `x` from the same directory as the current file, use `import "./x" as x;`. If you use `import "x" as x;` instead, a different file could be referenced (in a global "include directory").

It depends on the compiler (see below) how to actually resolve the paths. In general, the directory hierarchy does not need to strictly map onto your local filesystem, it can also map to resources discovered via e.g. ipfs, http or git.

---

**주석:** Always use relative imports like `import "./filename.sol";` and avoid using `..` in path specifiers. In the latter case, it is probably better to use global paths and set up remappings as explained below.

---

#### Use in Actual Compilers

When invoking the compiler, you can specify how to discover the first element of a path, and also path prefix remappings. For example you can setup a remapping so that everything imported from the virtual directory `github.com/ethereum/dapp-bin/library` would actually be read from your local directory `/usr/local/dapp-bin/library`. If multiple remappings apply, the one with the longest key is tried first. An empty prefix is not allowed.

The remappings can depend on a context, which allows you to configure packages to import e.g., different versions of a library of the same name.

**solc**:

For solc (the commandline compiler), you provide these path remappings as `context:prefix=target` arguments, where both the `context:` and the `=target` parts are optional (`target` defaults to `prefix` in this case). All remapping values that are regular files are compiled (including their dependencies).

This mechanism is backwards-compatible (as long as no filename contains = or `:`) and thus not a breaking change. All files in or below the `context` directory that import a file that starts with `prefix` are redirected by replacing `prefix` by `target`.

For example, if you clone `github.com/ethereum/dapp-bin/` locally to `/usr/local/dapp-bin`, you can use the following in your source file:

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

Then run the compiler:

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

As a more complex example, suppose you rely on a module that uses an old version of dapp-bin that you checked out to `/usr/local/dapp-bin_old`, then you can run:

```
solc module1:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin_old/ \
     source.sol
```

This means that all imports in `module2` point to the old version but imports in `module1` point to the new version.

---

**주석:** `solc` only allows you to include files from certain directories. They have to be in the directory (or subdirectory) of one of the explicitly specified source files or in the directory (or subdirectory) of a remapping target. If you want to allow direct absolute includes, add the remapping `/=/`.

---

If there are multiple remappings that lead to a valid file, the remapping with the longest common prefix is chosen.

**Remix**:

Remix provides an automatic remapping for GitHub and automatically retrieves the file over the network. You can import the iterable mapping as above, e.g.

**::** import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;

Remix may add other source code providers in the future.

### Comments

Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

**주석:** A single-line comment is terminated by any unicode line terminator (LF, VF, FF, CR, NEL, LS or PS) in utf8 encoding. The terminator is still part of the source code after the comment, so if it is not an ascii symbol (these are NEL, LS and PS), it will lead to a parser error.

Additionally, there is another type of comment called a natspec comment, for which the documentation is not yet written. They are written with a triple slash (`///`) or a double asterisk block(`/** ... */`) and they should be used directly above function declarations or statements. You can use Doxygen-style tags inside these comments to document functions, annotate conditions for formal verification, and provide a **confirmation text** which is shown to users when they attempt to invoke a function.

In the following example we document the title of the contract, the explanation for the two input parameters and two returned values.

```
pragma solidity >=0.4.0 <0.6.0;

/** @title Shape calculator. */
contract ShapeCalculator {
    /** @dev Calculates a rectangle's surface and perimeter.
      * @param w Width of the rectangle.
      * @param h Height of the rectangle.
      * @return s The calculated surface.
      * @return p The calculated perimeter.
      */
    function rectangle(uint w, uint h) public pure returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

### 8.4.2 Structure of a Contract

Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of *State Variables*, *Functions*, *Function Modifiers*, *Events*, *Struct Types* and *Enum Types*. Furthermore, contracts can inherit from other contracts.

There are also special kinds of contracts called *libraries* and *interfaces*.

The section about *contracts* contains more details than this section, which serves to provide a quick overview.

#### State Variables

State variables are variables whose values are permanently stored in contract storage.

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

See the *타입* section for valid state variable types and *Visibility and Getters* for possible choices for visibility.

### Functions

Functions are the executable units of code within a contract.

```
pragma solidity >=0.4.0 <0.6.0;

contract SimpleAuction {
    function bid() public payable { // Function
        // ...
    }
}
```

*함수 호출* can happen internally or externally and have different levels of *visibility* towards other contracts.

### Function Modifiers

Function modifiers can be used to amend the semantics of functions in a declarative way (see *Function Modifiers* in the contracts section).

```
pragma solidity >=0.4.22 <0.6.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modifier
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public view onlySeller { // Modifier usage
        // ...
    }
}
```

### Events

Events are convenience interfaces with the EVM logging facilities.

```
pragma solidity >=0.4.21 <0.6.0;

contract SimpleAuction {
    event HighestBidIncreased(address bidder, uint amount); // Event

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // Triggering event
    }
}
```

See *Events* in contracts section for information on how events are declared and can be used from within a dapp.

#### Struct Types

Structs are custom defined types that can group several variables (see *구조체* in types section).

```
pragma solidity >=0.4.0 <0.6.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

#### Enum Types

Enums can be used to create custom types with a finite set of 'constant values' (see *Enums* in types section).

```
pragma solidity >=0.4.0 <0.6.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

### 8.4.3 타입

solidity는 컴파일 시점에 각 변수(상태변수와 지역변수)의 타입이 명시되어야하는 (또는 최소한 추론가능해야하는 - *타입 추론* 참조) 정적 타입 언어입니다. solidity는 몇 가지의 기본 타입을 제공하며 이를 조합해서 복합 타입을 만들 수 있습니다.

또한, 타입은 연산자가 포함된 표현식에서 서로 상호작용할 수 있습니다. 여러 가지 연산자에 대한 내용은 *Order of Precedence of Operators* 를 참조하세요.

#### 값 타입

다음의 타입들은 변수가 전달될 때 값(value)이 전달되므로 값 타입이라고도 불립니다. 즉, 이 타입이 함수의 인자로 사용되거나 할당값으로 사용될 땐, 값이 복사됩니다.

#### Booleans

`bool`: 가능한 값은 상수 `true` 그리고 `false` 입니다.

연산자:

- `!` (논리 부정)
- `&&` (논리 AND, "and")
- `||` (논리 OR, "or")
- `==` (같음)
- `!=` (같지 않음)

`||` 과 `&&` 에는 일반적인 short-circuiting rules이 적용됩니다. 이것은 `f(x) || g(y)` 에서 만약 `f(x)` 가 `true` 라면, `g(y)` 의 값을 확인하지 않는다면 부작용이 있을 수 있음에도 불구하고 값을 확인하지 않는것을 의미합니다.

### 정수

`int` / `uint`: 다양한 크기의 부호있는 정수 타입, 부호없는 정수 타입이 존재합니다. `uint8` 에서 `uint256` 까지, 그리고 `int8` 부터 `int256` 까지 8비트 단위로 키워드가 존재합니다. `uint` 와 `int` 는 각각 `uint256` 와 `int256` 의 별칭입니다.

연산자:

- 비교 연산자: <=, <, ==, !=, >=, > (`bool` 결과값을 가짐)
- 비트 연산자: `&`, `|`, ^ (배타적 비트 or), ~ (비트 보수)
- 산술 연산자: +, -, 단항 -, 단항 +, *, /, `%` (나머지), `**` (거듭제곱), << (왼쪽 시프트), >> (오른쪽 시프트)

나눗셈의 결과는 항상 정수이며 소수부분은 절사됩니다(EVM의 `DIV` opcode로 컴파일 됩니다). 그러나 두 연산자가 *literals* (또는 리터럴 표현식)인 경우 소수부분은 절사되지 않습니다.

0으로 나누거나 0으로 모듈로 연산을 하면 런타임 예외가 발생합니다.

시프트 연산 결과의 타입은 왼쪽 피연산자의 타입을 따릅니다. `x << y` 는 `x * 2**y` 와 동일하며, `x >> y` 는 `x / 2**y` 와 동일합니다. 이는 음수를 시프트하는 경우 부호가 확장됨을 의미합니다.(This means that shifting negative numbers sign extends.) 음수만큼 시프트 연산을 실행하는 경우 런타임 예외가 발생합니다.

**경고:** 부호있는 음의 정수를 우측 시프트 연산 한 결과값은 다른 프로그래밍 언어에서의 결과값과 다릅니다. solidity에서는, 우측 시프트는 나눗셈과 매핑되며 그로 인해 시프트된 음의 값은 0으로 반올림되어 갑니다(절사). 다른 프로그래밍 언어에서는, 음의 값을 우측 시프트연산 하는 경우, 나눗셈과 소수점 이하 버림이 동시에 작동하는것과 유사하게 동작합니다(음의 무한대 방향).

### 부동 소수점 숫자

**경고:** 고정 소수점 수는 아직 solidity에서 완벽하게 지원되지 않습니다. 고정 소수점 수는 선언될 수는 있지만 할당될 수는 없습니다.

`fixed` / `ufixed`: 다양한 크기의 부호있는 고정 소수점, 부호없는 고정 소수점 타입이 존재합니다. 키워드 `ufixedMxN` 와 `fixedMxN` 에서 `M` 은 타입에 의해 취해진 비트의 수를 나타내며 `N` 은 소수점이하 자리수를 나타냅니다. `M` 은 8에서 256비트 사이의 값이며 반드시 8로 나누어 떨어져야 합니다. `N` 은 0과 80 사이의 값이어야만 합니다. `ufixed` 와 `fixed` 는 각각 `ufixed128x19` 와 `fixed128x19` 의 별칭입니다.

연산자:

- 비교 연산자: <=, <, ==, !=, >=, > (`bool` 결과값을 가짐)
- 산술 연산자: +, -, 단항 -, 단항 +, *, /, `%` (나머지)

**주석:** 부동 소수점 수와 고정 소수점 수의 주요한 차이점은, 부동 소수점 수는 정수와 소수점 부분을 표현하기 위해 사용되는 비트의 수가 유동적인데 반해, 고정 소수점의 경우 엄격히 정의되어 있습니다. 일반적으로, 부동 소수점 방식에서는 거의 모든 공간이 소수 부분을 나타내기 위해 사용되지만 고정 소수점 방식에서는 적은 수의 비트만이 소수 부분을 정의하는데 사용됩니다.

### Address

`address` : 20바이트(이더리움 address의 크기)를 담을 수 있습니다. address 타입에는 멤버가 있으며 모든 컨트랙트의 기반이 됩니다.

연산자:

- <=, <, ==, !=, >= and >

---

**주석:** 0.5.0으로 시작하는 버전의 컨트랙트는 address 타입에서 파생되지 않았지만, address 타입으로 명시적 변환될 수 있습니다.

---

### address의 members

- `balance` 와 `transfer`

빠르게 훑으려면 *Address 관련* 를 참조하세요.

`balance` 속성을 이용하여 address의 잔고를 조회하고 `transfer` 함수를 이용하여 다른 address에 Ether를 (wei 단위로) 보낼 수 있습니다:

```
address x = 0x123;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

---

**주석:**

**x 가 컨트랙트 address인 경우, 코드(더 구체적으로는: fallback 함수가 존재하는 경우)는 `transfer` 호출과 함께** 실행될 것입니다(이건 EVM의 특성이며 막을 수 없습니다). 코드가 실행될때 가스가 부족하거나 어떤식으로든 실패한다면, Ether 전송은 원상복구되며 현재의 컨트랙트는 예외를 발생하며 중지됩니다.

---

- `send`

Send는 low-level 수준에서 `transfer` 에 대응됩니다. 실행이 실패하면 컨트랙트는 중단되지 않고 대신 `send` 가 `false` 를 반환할 것입니다.

**경고:** `send` 를 사용할 땐 몇가지 주의사항이 있습니다: call stack의 깊이가 1024라면 전송은 실패하며(이것은 항상 호출자에 의해 강제 될 수 있습니다) 그리고 수신자의 gas가 전부 소모되어도 실패합니다. 그러므로 안전한 Ether 전송을 위해서, 항상 `send` 의 반환값을 확인하고, `transfer` 를 사용하세요: 혹은 더 좋은 방법은 수신자가 돈을 인출하는 패턴을 사용하는 것입니다.

- `call`, `callcode` 그리고 `delegatecall`

또한, ABI를 준수하지 않는 컨트랙트와 상호작용하기 위하여 임의 숫자의 인자를 취하는 `call` 함수가 제공되며 인자의 타입 역시 모든 타입을 취할 수 있습니다. 이러한 인자는 32바이트가 될 때까지 채워지고 연결됩니다. 한 가지 예외는 첫 번째 인자가 정확히 4바이트로 인코딩 되는 경우입니다. 이 경우에는, 함수 서명이 사용되도록 하기 위해 인자가 채워지지 않습니다.

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
nameReg.call("register", "MyName");
nameReg.call(bytes4(keccak256("fun(uint256)")), a);
```

`call` 은 호출된 함수가 종료되었는지(`true`) 아니면 EVM 예외를 발생시켰는지를(`false`) 나타내는 boolean을 반환합니다. 반환된 데이터 자체에 접근하는건 불가능합니다(이를 위해선 우리는 인코딩과 크기를 미리 알고 있어야 합니다).

`.gas ()` 제어자를 사용하여 제공된 가스를 조절할 수 있습니다:

```
namReg.call.gas(1000000)("register", "MyName");
```

이와 유사하게, 제공된 Ether의 값도 역시 조절할 수 있습니다:

```
nameReg.call.value(1 ether)("register", "MyName");
```

마지막으로, 이 제한자들은 함께 사용할 수 있으며 순서는 상관 없습니다:

```
nameReg.call.gas(1000000).value(1 ether)("register", "MyName");
```

---

**주석:** 현재로서는 오버로딩된 함수에서 가스 제한자나 값 제한자를 사용할 수 없습니다.

이를 위한 해결책은 가스와 값에 관해 특수한 경우가 있다는걸 소개하고 다시 한번 더 문제를 일으키지 않는지 확인하는 것입니다.

---

이와 유사하게, 함수 `delegatecall` 을 사용할 수 있습니다: 차이점은, 주어진 주소에선 오직 코드만 사용되고, 다른 것들(저장소, 잔고, ...)은 현재 컨트랙트의 것을 사용합니다. `delegatecall` 의 목적은 다른 컨트랙트에 저장된 라이브러리 코드를 사용하기 위함입니다. 사용자는 양쪽 컨트랙트의 저장소 레이아웃이 delegatecall을 통해 사용되기 적합한지 반드시 확인해야 합니다. homestead 단계 전까지는, `callcode` 라고 불리는 delegatecall의 제한된 변형형태만이 이용가능했는데, 이 변형된 형태는 `msg.sender` 와 `msg.value` 에 접근하는 기능을 제공하지 않았습니다.

세 가지 함수 `call`, `delegatecall` 및 `callcode` 는 매우 low-level 함수이므로 Solidity의 타입 안전성을 깨뜨리니 *최후의 수단* 으로서만 사용해야합니다.

`.gas ()` 옵션은 세 가지 메소드 모두에서 사용할 수 있지만, `.value ()` 옵션은 `delegatecall` 에서 사용할 수 없습니다.

---

**주석:** 모든 컨트랙트는 address의 멤버를 상속하므로, `this.balance` 를 이용하여 현재 컨트랙트의 잔액을 조회하는것이 가능합니다.

---

---

**주석:** `callcode` 는 추후 버전에서 제거 될 예정이라 사용을 권장하지 않습니다.

---

**경고:** 이 함수들은 low-level 함수이므로 주의해서 사용해야 합니다. 특히 알지 못하는 컨트랙트는 위험할 수 있으며 만약 이를 호출할 경우 해당 컨트랙트에 대한 제어 권한을 넘겨주므로 호출이 반환될 때 상태 변수를 변경할 수 있습니다.

### 고정 크기 바이트 배열

`bytes1`, `bytes2`, `bytes3`, ..., `bytes32`. `byte` is an alias for `bytes1`.

연산자:

- 비교 연산자: <=, <, ==, !=, >=, > (`bool` 결과값을 가짐)

- 비트 연산자: `&`, `|`, `^` (배타적 비트 or), ~ (비트 보수), << (왼쪽 시프트), >> (오른쪽 시프트)
- 인덱스 접근: 만약 `x` 가 `bytesI` 타입이라면, `0 <= k < I` 일때 `x[k]` 는 `k` 번째 바이트를 반환한다(읽기 전용).

시프트 연산자는 몇 비트만큼 이동할건지를 나타내는 오른쪽 피연산자로 모든 정수를 취할 수 있습니다(그렇지만 왼쪽 피연산자의 타입을 반환합니다). 음수만큼 시프트하는 경우 런타임 예외가 발생합니다.

Members:

- `.length` 는 바이트 배열의 고정된 길이를 반환합니다(읽기 전용).

---

**주석:** 바이트 배열은 `byte[]` 로도 사용이 가능하지만, 이럴 경우 각 요소마다 정확히 31바이트의 공간을 낭비하게됩니다. `bytes` 를 사용하는것이 더 낫습니다.

---

### 동적 크기 바이트 배열

**`bytes`:** 동적 크기 바이트 배열, *배열* 을 참조하세요. 값 타입이 아닙니다!

**`string`:** 동적 크기의 UTF-8 인코딩된 문자열, *배열* 을 참조하세요. 값 타입이 아닙니다!

경험에 따르면 임의 길이의 원시 바이트 데이터의 경우에는 `bytes` 를 사용하고 임의 길이의 문자열(UTF-8) 데이터의 경우에는 `string` 을 사용하세요. 만약 길이를 특정 바이트만큼 제한할수 있다면, 항상 `bytes1` 에서 `bytes32` 중 하나를 사용하세요. 왜냐하면 공간을 더 절약할 수 있기 때문입니다.

### Address 리터럴

address 체크섬 테스트를 통과한 16진수 리터럴(예를 들면 `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF`)은 `address` 타입입니다. 체크섬 테스트를 통과하지 못한 39자리 ~ 41자리 길이의 16진수 리터럴은 경고를 발생시키고 일반적인 유리수 리터럴로 취급됩니다.

---

**주석:** 혼합 케이스 address의 체크섬 형식은 EIP-55 에 정의되어 있습니다.

---

### 유리수 리터럴 및 정수 리터럴

정수 리터럴은 0-9 범위의 일련의 숫자로 구성됩니다. 정수 리터럴은 십진법으로 나타내어집니다. 예를 들어, `69` 는 육십구를 의미합니다. 8진법 리터럴은 solidity에 존재하지 않으며 선행 0은 유효하지 않습니다.

소수점 이하 리터럴은 한쪽에 적어도 하나의 숫자가 있을때 `.` 에 의해 구성됩니다. 예로는 `1.`, `.1`, `1.3` 이 있습니다.

소수가 밑이 될 수 있지만 지수는 될 수 없는 과학적 표기법 또한 지원됩니다. `2e10`, `-2e10`, `2e-10`, `2.5e1` 같은 예가 있습니다.

숫자 리터럴 표현식은 리터럴이 아닌 타입으로 변환될때까지(즉, 리터럴이 아닌 표현식과 함께 사용되는 경우) 임의 정밀도를 유지합니다. 이는 계산이 수행될때 오버플로우가 발생하지 않으며 나눗셈이 수행될때 자릿수를 잘라내지 않는걸 의미합니다.

예를 들어, `(2**800 + 1) - 2**800` 의 결과는 비록 중간 결과값이 machine word size에 적합하지 않을지라도 상수 `1` (`uint8` 타입)입니다. 게다가 `.5 * 8` 의 결과값은 (비록 중간에 정수가 아닌 숫자가 사용되었을지라도) 정수 `4` 입니다.

정수에 사용할 수 있는 연산자는 피연산자가 정수인 숫자 리터럴 표현식에도 사용할 수 있습니다. (피연산자) 둘 중 하나라도 소수일 경우에는, 비트 연산이 허용되지 않으며 지수가 소수일 경우에도 지수 연산이 허용되지 않습니다 (무리수가 발생할 수 있으므로).

---

**주석:** solidity는 각 유리수에 대해 숫자 리터럴 타입을 가집니다. 정수 리터럴과 유리수 리터럴은 숫자 리터럴 타입에 속합니다. 또한 모든 숫자 리터럴 표현식(즉, 오직 숫자 리터럴과 연산자로만 구성된 표현식)은 숫자 리터럴 타입에 속합니다. 그러므로 숫자 리터럴 표현식 `1 + 2` 와 `2 + 1` 모두 유리수 3에 대해 같은 숫자 리터럴 타입에 속합니다.

---

**경고:** 이전 버전에서는 정수 리터럴에 대한 나눗셈의 결과에서 자릿수를 버렸지만, 현재는 유리수로 변환됩니다. 즉 `5 / 2` 는 `2` 가 아니라 `2.5` 입니다.

---

**주석:** 숫자 리터럴 표현식은 리터럴이 아닌 표현식과 함께 사용되는 즉시 리터럴이 아닌 타입으로 변환됩니다. 비록 우리는 다음 예제에서 `b` 에 할당된 값이 정수로 평가된다는걸 알고 있지만, 부분 표현식 `2.5 + a` 은 타입 검사를 하지 않으며 코드는 컴파일되지 않습니다.

---

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

### 문자열 리터럴

문자열 리터럴은 큰따옴표나 작은따옴표와 함께 작성됩니다(`"foo"` 또는 `'bar'`). solidity에선 C에서 처럼 trailing zeroes를 포함하진 않습니다; `"foo"` 는 4바이트가 아닌 3바이트를 차지합니다. 정수 리터럴과 마찬가지로, 문자열 리터럴의 타입은 다양하며 `bytes1`, ..., `bytes32` 로 암시적 변환될 수 있습니다. 적합한 크기라면 `bytes` 와 `string` 으로도 변환될 수 있습니다.

문자열 리터럴은 `\n`, `\xNN`, `\uNNNN` 와 같은 escape characters을 지원합니다. `\xNN` 은 16진수 값을 취해 적절한 바이트를 삽입하는 반면 `\uNNNN` 은 Unicode codepoint를 취해 UTF-8 sequence를 삽입합니다.

### 16진수 리터럴

16진수 리터럴은 키워드 `hex` 가 접두사로 붙고 큰따옴표나 작은따옴표로 둘러싸여집니다(`hex"001122FF"`). 내용은 16진수 문자열이어야 하며 값은 바이너리로 표현됩니다.

16진수 리터럴은 문자열 리터럴과 같이 동작하기에 동일하게 변경에 제한이 있습니다.

### Enums

열거형은 solidity에서 사용자 정의 타입을 만드는 한 가지 방법입니다. 열거형은 모든 정수타입으로/정수타입에서 명시적 변환이 가능하지만 암시적 변환은 허용되지 않습니다. 명시적 변환은 런타임때 값 범위를 체크하고 실패시 예외를 발생시킵니다. 열거형은 최소 하나의 멤버를 필요로 합니다.

```
pragma solidity ^0.4.16;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
```

(continues on next page)

(이전 페이지에서 계속)

```
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of "getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity. The integer type used is just
    // large enough to hold all enum values, i.e. if you have more values,
    // `uint16` will be used and so on.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
}
```

### 함수 타입

함수 타입입니다. 함수 타입의 변수는 함수에서 할당 될 수 있으며 함수 타입의 함수매개변수는 함수가 호출될 때, 함수를 전달하거나 반환하는데 사용될 수 있습니다. 함수 타입에는 두 종류가 있습니다 - *내부* 및 *외부* 함수입니다:

내부 함수는 오직 현재 컨트랙트의 내부에서만(더 구체적으로는, 내부 라이브러리 함수와 상속받은 함수를 포함한 현재 코드 유닛 내부에서만) 호출될 수 있습니다. 왜냐하면 내부 함수는 현재 컨트랙트의 컨텍스트 밖에서 실행될 수 없기 때문입니다. 내부 함수를 호출하는것은 마치 현재 컨트랙트의 함수를 내부적으로 호출할때와 마찬가지로 entry label로 가서 실현됩니다.

외부 함수는 address와 함수 서명으로 구성되며 외부 함수 호출을 통해 전달되고 반환 될 수 있습니다.

함수 타입은 다음과 같이 표기됩니다:

```
function (<parameter types>) {internal|external} [pure|constant|view|payable]␣
↪[returns (<return types>)]
```

매개변수 타입과 달리, 반환 타입은 비워 둘 수 없습니다. 함수 타입이 아무것도 반환하지 않는다면 `returns (<return types>)` 이 부분 전체를 생략해야 합니다.

기본적으로, 함수 타입은 내부함수이므로, `internal` 키워드는 생략 가능합니다. 반대로, 컨트랙트 함수 자체는 기본적으로 public이며 타입의 이름으로 사용될 때만 기본값이 internal 입니다.

현재 컨트랙트에서 함수에 접근하는 방법은 두가지가 있습니다: `f` 이렇게 직접 이름을 사용하거나, `this.f` 이런 식으로 접근할 수 있습니다. 전자는 내부함수, 후자는 외부함수가 될 것입니다.

함수 타입 변수가 초기화 되지 않은 상태에서, 이를 호출하면 예외가 발생합니다. 함수에 `delete` 를 사용 후 그 함수를 호출하는 경우에도 동일하게 예외가 발생합니다.

외부 함수 타입이 solidity 컨텍스트의 외부에서 사용되는 경우, 이들은 뒤에 함수 식별자가 붙는 address를 단일 `bytes24` 타입으로 인코딩하는 `function` 으로 취급됩니다.

현재 컨트랙트의 퍼블릭 함수는 내부 함수로도 외부함수로도 사용될 수 있습니다. `f` 를 내부 함수로 사용하려면, `f` 만 사용하고 외부 함수로 사용하려면 `this.f` 로 사용하세요.

또한 퍼블릭 (또는 external) 함수에는 *ABI function selector* 를 반환하는 특수한 멤버 *selector* 가 있습니다.

pragma solidity ^0.4.16;

**contract Selector {**

**function f() public view returns (bytes4) {** return this.f.selector;

}

}

내부 함수 타입을 사용하는 방법을 보여주는 예제:

```
pragma solidity ^0.4.16;

library ArrayUtils {
  // internal functions can be used in internal library functions because
  // they will be part of the same code context
  function map(uint[] memory self, function (uint) pure returns (uint) f)
    internal
    pure
    returns (uint[] memory r)
  {
    r = new uint[](self.length);
    for (uint i = 0; i < self.length; i++) {
      r[i] = f(self[i]);
    }
  }
  function reduce(
    uint[] memory self,
    function (uint, uint) pure returns (uint) f
  )
    internal
    pure
    returns (uint r)
  {
    r = self[0];
    for (uint i = 1; i < self.length; i++) {
      r = f(r, self[i]);
    }
  }
  function range(uint length) internal pure returns (uint[] memory r) {
    r = new uint[](length);
    for (uint i = 0; i < r.length; i++) {
      r[i] = i;
    }
  }
}

contract Pyramid {
  using ArrayUtils for *;
  function pyramid(uint l) public pure returns (uint) {
    return ArrayUtils.range(l).map(square).reduce(sum);
  }
  function square(uint x) internal pure returns (uint) {
    return x * x;
  }
  function sum(uint x, uint y) internal pure returns (uint) {
    return x + y;
  }
}
```

외부 함수 타입을 사용하는 또 다른 예제:

```
pragma solidity ^0.4.21;

contract Oracle {
  struct Request {
    bytes data;
    function(bytes memory) external callback;
  }
  Request[] requests;
  event NewRequest(uint);
  function query(bytes data, function(bytes memory) external callback) public {
    requests.push(Request(data, callback));
    emit NewRequest(requests.length - 1);
  }
  function reply(uint requestID, bytes response) public {
    // Here goes the check that the reply comes from a trusted source
    requests[requestID].callback(response);
  }
}

contract OracleUser {
  Oracle constant oracle = Oracle(0x1234567); // known contract
  function buySomething() {
    oracle.query("USD", this.oracleResponse);
  }
  function oracleResponse(bytes response) public {
    require(msg.sender == address(oracle));
    // Use the data
  }
}
```

**주석:** 람다함수나 인라인함수 역시 계획되어있지만 아직 지원되지 않습니다.

### 참조 타입

복합 타입, 즉 항상 256비트에 들어맞지 않는 타입은 우리가 이제까지 다뤘던 타입보다 더욱 신중하게 다뤄야 합니다. 복합 타입을 복사하는 것은 비싼 연산이 될 수 있으므로, 우리는 복합 타입을 **메모리** (지속성이 없음) 와 **스토리지** (상태 변수가 저장되는 곳)중 어디에 저장하고 싶은지 생각해보아야 합니다.

### 데이터 위치

모든 복합 타입은 자신이 *메모리* 나 *스토리지* 중 어디에 저장되었는지를 나타내는 "데이터 위치"가 추가적으로 존재합니다. 컨텍스트에 따라 항상 기본값이 존재하지만, 타입에 스토리지 나 메모리 를 추가하여 재정의 할 수 있습니다. 함수 매개 변수(반환 매개 변수도 포함)의 기본값은 메모리 이고, 지역 변수의 기본값은 스토리지 이며 상태 변수의 위치는 스토리지 로 강제되어 있습니다.

또한 세 번째 데이터 위치인 `calldata` 가 있으며, 여기에는 함수 인자가 저장되고 수정 불가능하며 지속성이 없습니다. 외부 함수의 함수 매개 변수(반환 매개변수 제외)는 `calldata` 에 강제 저장되며 거의 `memory` 처럼 작동합니다.

데이터 위치는 변수가 할당되는 방식을 변경하기 때문에 중요합니다: assignments between storage and memory and also to a state variable (even from other state variables) always create an independent copy. Assignments to local storage variables only assign a reference though, and this reference always points to the state variable even if the latter

is changed in the meantime. 반면, 메모리에 저장된 참조 타입에서 다른 메모리에 저장된 참조 타입을 할당할땐 복사본을 만들지 않습니다.

```
pragma solidity ^0.4.0;

contract C {
    uint[] x; // the data location of x is storage

    // the data location of memoryArray is memory
    function f(uint[] memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        var y = x; // works, assigns a pointer, data location of y is storage
        y[7]; // fine, returns the 8th element
        y.length = 2; // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new temporary /
        // unnamed array in storage, but storage is "statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would "reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in memory
    }

    function g(uint[] storage storageArray) internal {}
    function h(uint[] memoryArray) public {}
}
```

### 요약

**강제 데이터 위치:**

- 외부 함수의 매개 변수(반환값 미포함): calldata
- 상태 변수: 스토리지

**기본 데이터 위치:**

- 함수의 매개변수(반환값 포함): 메모리
- 모든 지역 변수: 스토리지

### 배열

배열은 컴파일시 고정 크기를 가질 수도 있고 동적인 크기를 가질 수도 있습니다. 스토리지 배열의 경우, 요소의 타입은 임의적일 수(즉, 다른 배열이 될 수도 있고, 매핑이나 구조체일수도 있음) 있습니다. 메모리 배열의 경우, 매핑이 될 수 없으며 만약 공개적으로 보여지는 함수의 인자라면 ABI 타입이어야 합니다.

크기는 `k` 로 고정되었고 요소의 타입은 `T` 인 배열은 `T[k]` 로 표시하며, 동적인 크기의 배열은 `T[]` 로 표시합니다. 예를 들자면, `uint` 타입을 요소로 하는 동적 크기 배열 5개로 구성된 배열은 `uint[][5]` 입니다(다른 언어들과는 달리 행과 열의 표현이 바뀌어있음에 유의하세요). 세번째 동적 크기 배열의 두번째 uint에 접근하려면, `x[2][1]` 이렇게 하세요 (인덱스는 0부터 시작하며 접근은 선언과는 반대되는 방식으로 작동합니다. i.e. `x[2]` shaves off one level in the type from the right)

`bytes` 와 `string` 타입의 변수는 특별한 형태의 배열입니다. `bytes` 는 `byte[]` 와 유사하지만 calldata로 꽉 차여져 있습니다. `string` 은 `bytes` 와 동일하지만 (현재로서는) 길이나 인덱스 접근을 허용하지 않습니다

그러므로 `bytes` 는 언제나 `byte[]` 보다 우선순위로 고려되어야합니다. 더 저렴하기 때문입니다.

---

**주석:** 문자열 `s` 의 byte-representation에 접근하고자 한다면, `bytes(s).length` / `bytes(s)[7] = 'x';` 이렇게 하세요. 개별 문자가 아닌 UTF-8의 low-level 바이트에 접근하고 있다는걸 명심하세요!

---

배열을 `public` 으로 생성하고 solidity가 *getter* 를 생성하도록 할 수 있습니다. 숫자 인덱스는 getter의 필수 매개변수가 됩니다.

### 메모리 배열 할당

`new` 키워드를 사용해 크기 변경이 가능한 배열을 메모리에 생성할 수 있습니다. 스토리지 배열과는 달리, `.length` 멤버에 값을 할당함으로써 메모리 배열의 크기를 변경하는것은 **불가능** 합니다.

```
pragma solidity ^0.4.16;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        // Here we have a.length == 7 and b.length == len
        a[6] = 8;
    }
}
```

### 배열 리터럴 / 인라인 배열

배열 리터럴은 표현식으로 작성된 배열이며 즉각적으로 변수에 할당되지 않습니다.

```
pragma solidity ^0.4.16;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] _data) public pure {
        // ...
    }
}
```

배열 리터럴의 타입은 고정된 크기를 가지는 메모리 배열이며 base type은 주어진 요소들의 공통 타입을 따릅니다. `[1, 2, 3]` 의 타입은 `uint8[3] memory` 입니다. 왜냐하면 정수 각각의 타입이 `uint8` 이기 때문입니다. 그렇기 때문에, 위 예제의 첫 번째 요소를 `uint` 로 변환해야 했습니다. 현재로서는, 고정된 크기의 메모리 배열을 동적 크기의 메모리 배열에 할당할 수 없습니다. 즉, 다음과 같은 것은 불가능합니다:

```
// This will not compile.

pragma solidity ^0.4.0;

contract C {
    function f() public {
        // The next line creates a type error because uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] x = [uint(1), 3, 4];
```

(continues on next page)

(이전 페이지에서 계속)

```
    }
}
```

이 제약사항은 추후에 제거될 예정이지만, 현재 이러한 제약사항으로 인해 배열이 ABI로 전달되는 방식에 따라 몇가지 문제가 발생합니다.

### 멤버

**length:** 배열에는 요소의 갯수를 저장하기 위한 `length` 멤버가 존재합니다. (메모리가 아닌) 스토리지에 저장된 동적 배열은 `length` 멤버의 값을 변경하여 크기를 조절할 수 있습니다. 현재 length를 벗어나는 요소에 접근을 시도한다고해서 크기의 조절이 자동으로 되는건 아닙니다. 메모리 배열은 생성될 때 크기가 결정되며 크기의 변경은 불가능합니다(그러나 동적 배열의 경우, 런타임 매개변수에 따라 달라질 수 있습니다.).

**push:** 동적 크기의 스토리지 배열과 `bytes` (`string` 은 제외)는 `push` 라는 멤버 함수를 가지고 있습니다. 이 함수는 배열의 끝에 요소를 추가하는데 사용됩니다. 이 함수는 새로운 length를 반환합니다.

**경고:** It is not yet possible to use arrays of arrays in external functions. 외부 함수에서 배열의 배열을 사용하는건 아직 불가능합니다.

**경고:** EVM의 한계로 인해, 외부함수를 호출했을때 동적인 것을 반환하는건 불가능합니다. `contract C { function f() returns (uint[]) { ... } }` 내부의 함수 `f` 는 web3.js에서 호출될 경우 무언가를 반환하겠지만 solidity에서 호출될 경우 반환이 불가능합니다.

현재로선 유일한 해결 방법은 크기가 고정된(동적이지 않은) 거대한 크기의 배열을 사용하는 것입니다.

```
pragma solidity ^0.4.16;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of dynamic arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of length two).
    bool[2][] m_pairsOfFlags;
    // newPairs is stored in memory - the default for function arguments

    function setAllFlagPairs(bool[2][] newPairs) public {
        // assignment to a storage array replaces the complete array
        m_pairsOfFlags = newPairs;
    }

    function setFlagPair(uint index, bool flagA, bool flagB) public {
        // access to a non-existing index will throw an exception
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // if the new size is smaller, removed array elements will be cleared
        m_pairsOfFlags.length = newSize;
    }

    function clear() public {
```

(continues on next page)

(이전 페이지에서 계속)

```
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags.length = 0;
    }

    bytes m_byteData;

    function byteArrays(bytes data) public {
        // byte arrays ("bytes") are different as they are stored without padding,
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        m_byteData.length += 7;
        m_byteData[3] = byte(8);
        delete m_byteData[2];
    }

    function addFlag(bool[2] flag) public returns (uint) {
        return m_pairsOfFlags.push(flag);
    }

    function createMemoryArray(uint size) public pure returns (bytes) {
        // Dynamic memory arrays are created using `new`:
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(i);
        return b;
    }
}
```

### 구조체

solidity는 아래의 예시처럼 구조체의 형식으로 새로운 타입을 정의하는 방법을 제공합니다.

```
pragma solidity ^0.4.11;

contract CrowdFunding {
    // Defines a new type with two fields.
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
```

(continues on next page)

(이전 페이지에서 계속)

```
    mapping (uint => Campaign) campaigns;

    function newCampaign(address beneficiary, uint goal) public returns (uint␣
↪campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
        // Creates new struct and saves in storage. We leave out the mapping type.
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // Creates a new temporary memory struct, initialised with the given values
        // and copies it over to storage.
        // Note that you can also use Funder(msg.sender, msg.value) to initialise.
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

컨트랙트는 크라우드펀딩에서의 계약에 필요한 모든 기능을 제공하진 않지만 구조체를 이해하는데 필요한 기본적인 개념을 포함합니다. 구조체 타입은 매핑과 배열의 내부에서 사용될 수 있으며 구조체 역시 내부에 매핑과 배열을 포함할 수 있습니다.

구조체는 매핑 멤버의 값 타입이 될 순 있지만, 구조체가 동일한 구조체 타입의 멤버를 포함할 순 없습니다. 구조체의 크기는 유한해야 하므로 이러한 제약이 필요한것이죠.

모든 종류의 함수에서, 어떻게 구조체 타입이 (기본 스토리지 데이터 위치의) 지역 변수에 할당되는지 유의하십시오. 이는 구조체를 복사(copy)하지 않고 참조(reference)만 저장하므로 지역 변수의 멤버에 할당하는것은 실제로 상태에 기록됩니다.

물론 `campaigns[campaignID].amount = 0` 처럼 지역 변수에 할당하지 않고도 구조체의 멤버에 직접 접근할 수도 있습니다.

### 매핑

매핑 타입은 `mapping(_KeyType => _ValueType)` 와 같이 선언됩니다. 여기서 `_KeyType` 은 매핑, 동적 크기 배열, 컨트랙트, 열거형, 구조체를 제외한 거의 모든 유형이 될 수 있습니다. `_ValueType` 은 매핑 타입을 포함한 어떤 타입이든 될 수 있습니다.

매핑은 사실상 모든 가능한 키가 초기화되고 byte-representation이 모두 0인 값(타입의 *기본 값*)에 매핑되는 해시 테이블 로 볼 수 있습니다. 이는 매핑과 해시테이블의 유사한 점이며 차이점은, 키 데이터는 실제로 매핑에 저장되지 않고 오직 `keccak256` 해시만이 값을 찾기 위해 사용됩니다.

이로 인해, 매핑에는 길이 또는 집합(set)을 이루는 키나 값의 개념을 가지고 있지 않습니다.

매핑은 상태변수(또는 내부 함수에서의 스토리지 참조 타입)에만 허용됩니다.

매핑을 `public` 으로 표시하고 solidity가 *getter* 를 생성토록 할 수 있습니다. `_KeyType` 은 getter의 필수 매개변수이며 `_ValueType` 을 반환 합니다.

매핑 역시 `_ValueType` 이 될 수 있습니다. getter는 각각의 `_KeyType` 에 대하여 하나의 매개변수를 재귀적으로 가집니다.

```
pragma solidity ^0.4.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(this);
    }
}
```

**주석:** 매핑은 iterable하진 않지만, 그 위에 자료구조(data structure)를 구현하는건 가능합니다. 예시는 iterable mapping 을 참조하세요.

### Operators Involving LValues

만약 `a` 가 LValue라면(즉, 할당 될 수 있는 변수 또는 무언가), 다음의 연산자를 약자로 사용할 수 있습니다:

`a += e` is equivalent to `a = a + e`. The operators `-=`, `*=`, `/=`, `%=`, `|=`, `&=` and `^=` are defined accordingly. `a++` and `a--` are equivalent to `a += 1` / `a -= 1` but the expression itself still has the previous value of `a`. In contrast, `--a` and `++a` have the same effect on `a` but return the value after the change.

`a += e` 는 `a = a + e` 와 동일합니다. 연산자 `-=`, `*=`, `/=`, `%=`, `|=`, `&=`, `^=` 역시 동일한 방식으로 정의됩니다. `a++` 와 `a--` 는 `a += 1` / `a -= 1``와 동일하게 값을 변경시키지만, 표현식 자체는 여전히 ``a``의 변경이 일어나지 않은 값을 반환합니다. 이와 반대로, ``--a` 와 `++a` 역시 ‘‘a‘‘의 값을 변화시키지만, 이 표현식은 변경된 값을 반환합니다.

### delete

`delete a` 는 타입의 초기 값을 `a` 에 할당합니다. 즉, 정수의 경우라면 `a = 0` 입니다. 배열에서도 사용될 수 있는데 이 경우, 길이가 0인 동적 배열이나 동일한 길이의 정적 배열의 모든 요소를 초기화합니다. 구조체에 사용할 경우, 구조체의 모든 멤버를 초기화합니다.

`delete` 는 매핑에 아무런 영향을 미치지 못합니다(매핑의 키는 임의적이며 일반적으로 알려져있지 않기 때문입니다). 따라서 구조체를 delete할 경우, 매핑이 아닌 모든 멤버를 초기화하며 멤버의 내부도 매핑이 아니라면 재귀적으로 초기화합니다. 그러나, 개별 키 그리고 그 키가 어디에 매핑되었는지는 삭제될 수 있습니다.

`delete a` 는 실제로 `a` 에 초기값을 할당하는것처럼 동작합니다. 즉, `a` 에 새로운 객체를 저장합니다.

```
pragma solidity ^0.4.0;
```

(continues on next page)

(이전 페이지에서 계속)

```
contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x which still holds a copy
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to zero, but as uint[] is a␣
↪complex object, also
        // y is affected which is an alias to the storage object
        // On the other hand: "delete y" is not valid, as assignments to local␣
↪variables
        // referencing storage objects can only be made from existing storage␣
↪objects.
    }
}
```

### 기본 타입간의 변환

#### 암시적 형변환

피연산자가 서로 다른 타입이라면, 컴파일러는 하나의 피연산자를 다른 피연산자의 타입으로 암시적 형변환을 시도합니다(할당의 경우에도 마찬가지입니다). 일반적으로, 의미가 통하며 손실되는 정보가 없다면 value-type간 암시적 형변환이 가능합니다: `uint8` 는 `uint16` 로 암시적 형변환되며 `int128` 는 `int256` 로 암시적 형변환됩니다, 그러나 `int8` 는 `uint256` 으로 암시적 형변환될 수 없습니다(왜냐하면 `uint256` 는 `-1` 같은 값을 표현할 수 없기 때문입니다). 더욱이, 부호없는 정수는 같거나 큰 크기의 바이트로 변환 될 수 있지만 그 반대는 불가능합니다. `uint160` 로 변환 가능한 타입이라면 `address` 로도 변환될 수 있습니다.

#### 명시적 형변환

컴파일러가 암시적 형변환을 허용하지 않더라도 당신이 현재 무엇을 하고있는지 알고 있다면 명시적 형변환이 때때로 가능할 수 있습니다. 이는 예상치 않은 작동을 불러일으킬수 있으므로 확실히 당신이 원하는 결과가 나오는지 테스트해봐야 합니다! 음수 `int8` 을 `uint` 로 변환하는 다음의 예제를 보겠습니다:

```
int8 y = -3;
uint x = uint(y);
```

이 코드 조각의 끝에서, `x` 는 `0xfffff..fd` 값을 가질것이고, (64 hex characters) 이는 256 비트의 2의 보수 표현에서 -3입니다.

크기가 더 작은 타입으로 명시적 형변환 될 경우, 상위 비트가 잘려져 나갑니다:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

### 타입 추론

편의상, 항상 변수의 타입을 명시적으로 지정할 필요는 없으며 컴파일러는 변수에 할당된 첫번째 표현식의 타입에서 자동으로 타입을 추론합니다:

```
uint24 x = 0x123;
var y = x;
```

여기서, y 의 타입은 uint24 가 될겁니다. var 은 함수 매개 변수나 반환 매개 변수에선 사용될 수 없습니다.

**경고:** 첫 번째 할당에서만 타입이 추론되기에, i는 uint8 타입이고 이 타입의 가장 큰 값이 2000 보다 작기에 아래 코드 조각의 반복문은 무한반복문입니다. `for (var i = 0; i < 2000; i++) { ... }`

### 8.4.4 단위 및 전역 변수

#### 이더 단위

Ether를 더 작은 단위로 변환하기 위해 숫자리터럴 뒤에 wei, finney, szabo, ether 라는 접미사가 붙을 수 있습니다. Ether통화를 나타내는 숫자리터럴에 접미사가 붙지 않으면 Wei가 붙어있다고 간주합니다. 예. `2 ether == 2000 finney` 는 true 로 평가됩니다.

#### 시간 단위

숫자리터럴 뒤에 붙는 seconds, minutes, hours, days, weeks, years 와 같은 접미사는 시간 단위를 변환하는데 사용될 수 있으며 기본 단위는 seconds이고 다음과 같이 변환됩니다:

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`
- `1 years == 365 days`

이 단위들을 사용해 달력에서 계산을 할 땐 주의가 필요합니다. 왜냐하면 윤초 로 인해 모든 1 years가 항상 365 days와 동일한건 아니며 모든 1 days가 항상 24 hours와 동일한건 아니니까요. 윤초는 예측할 수 없기 때문에, 정확한 달력 라이브러리는 신뢰할수 있는 외부로부터 업데이트 되어야 합니다.

이 접미사들을 변수 뒤에 붙일순 없습니다. 만약 days를 포함하는 입력변수를 반복하고싶다면 다음과 같은 방식으로 할 수 있습니다:

```
function f(uint start, uint daysAfter) public {
    if (now >= start + daysAfter * 1 days) {
      // ...
    }
}
```

#### 특수한 변수와 함수

전역 네임스페이스에는(global namespace) 특수한 변수와 함수가 존재하며 이들은 주로 블록체인에 관한 정보를 제공하는데 사용됩니다.

### 블록 및 트랜잭션 속성

- `block.blockhash(uint blockNumber) returns (bytes32)`: 주어진 블록의 해시 - 현재 블록을 제외한 가장 최근 256개의 블록에 대해서만 작동함
- `block.coinbase` (`address`): 현재 블록 채굴자의 address
- `block.difficulty` (`uint`): 현재 블록 난이도
- `block.gaslimit` (`uint`): 현재 블록 gaslimit
- `block.number` (`uint`): 현재 블록 번호
- `block.timestamp` (`uint`): unix epoch 이후의 현재 블록 타임스탬프
- `gasleft() returns (uint256)`: 잔여 가스
- `msg.data` (`bytes`): 완전한 calldata
- `msg.gas` (`uint`): 잔여 가스 - 0.4.21버전에서 제거되었으며 `gasleft()` 로 대체됨
- `msg.sender` (`address`): 메세지 발신자 (현재 호출)
- `msg.sig` (`bytes4`): calldata의 첫 4바이트(즉, 함수 식별자)
- `msg.value` (`uint`): 메세지와 함께 전송된 wei 수
- `now` (`uint`): 현재 블록 타임스탬프(`block.timestamp` 의 별칭)
- `tx.gasprice` (`uint`): 트랜잭션의 가스 가격
- `tx.origin` (`address`): 트랜잭션의 발신자 (full call chain)

**주석:** `msg.sender` 와 `msg.value` 를 포함한 모든 `msg` 의 멤버 값은 **외부** 함수 호출에 의해 바뀔 수 있습니다. 외부함수 호출에는 라이브러리 함수 호출도 포함됩니다.

**주석:** 랜덤한 값을 선택하기 위해 `block.timestamp`, `now`, `block.blockhash` 를 사용하지 마세요.

채굴 커뮤니티의 악의적인 행위를 예로 들어보자면 특정 해시에서 casino payout function을 실행시키고 돈을 받지 못한다면 다시 다른 해시에 대해서 동일한 행위를 시도하는것이 가능합니다.

현재 블록의 타임스탬프는 마지막 블록의 타임스태프보다 무조건적으로 커야 합니다만, 유일하게 보장되는건, 현재 블록의 타임스탬프는 canonical chain의 연속된 두 블록의 타임스탬프 사이에 있을것이라는 것입니다.

**주석:** 확장성 문제로 인해 모든 블록의 블록해시가 사용가능한것은 아닙니다. 가장 최근 256개 블록의 해시에 대해서만 접근이 가능하며 다른 해시값은 0이 됩니다.

### 에러 처리

**`assert(bool condition)`:** 조건이 충족되지 않으면 예외를 발생시킵니다 - 내부 에러에 사용됩니다.

**`require(bool condition)`:** 조건이 충족되지 않으면 예외를 발생시킵니다 - 입력 또는 외부 요소의 에러에 사용됩니다.

**`revert()`:** 실행을 중단하고 변경된 상태를 되돌립니다.

### 수학 및 암호화 함수

**addmod(uint x, uint y, uint k) returns (uint):** compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

**mulmod(uint x, uint y, uint k) returns (uint):** compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.

**keccak256(...) returns (bytes32):** *(tightly packed) 인자* 의 Ethereum-SHA-3 (Keccak-256) 해시를 계산합니다.

**sha256(...) returns (bytes32):** *(tightly packed) 인자* 의 SHA-256 해시를 계산합니다

**sha3(...) returns (bytes32):** `keccak256` 의 별칭

**ripemd160(...) returns (bytes20):** *(tightly packed) 인자* 의 RIPEMD-160 해시를 계산합니다

**ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address):** 타원 곡선 서명으로부터 address와 연관된 공개 키를 복구하며 오류시엔 0을 반환합니다. (사용 예시)

위에서 "tightly packed"는 인자가 패딩(padding)없이 연결됨을 의미합니다. 이것은 다음이 모두 동일하다는 것을 의미합니다:

```
keccak256("ab", "c")
keccak256("abc")
keccak256(0x616263)
keccak256(6382179)
keccak256(97, 98, 99)
```

패딩이 필요하다면, 명시적 형변환을 통해 패딩을 할 수 있습니다: `keccak256("\x00\x12")` 는 `keccak256(uint16(0x12))` 와 동일합니다.

상수는 상수를 저장하는데 필요한 최소 바이트 수만을 사용하여 압축됩니다. `keccak256(0) == keccak256(uint8(0))` 와 `keccak256(0x12345678) == keccak256(uint32(0x12345678))` 는 이에 대한 예시입니다.

*프라이빗 블록체인* 에서 `sha256`, `ripemd160` 또는 `ecrecover` 를 실행할때 Out-of-Gas상태가 될 수 있습니다. 그 이유는 이러한 것들이 precompiled contracts라고 불리우는 형태로 구현되었기 때문입니다. 그리고 이런 컨트랙트는 오직 그들이 첫번째 메세지를 받은 이후에만 실제로 존재합니다(비록 컨트랙트 코드가 하드코딩되었을지라도). 존재하지 않는 컨트랙트로 메세지를 보내는건 훨씬 비싸며 그렇기에 Out-of-Gas 에러를 발생시킵니다. 이 문제에 대한 해결책은 실제 컨트랙트를 사용하기 전에 1 wei를 각 컨트랙트에 전송해보는것입니다. 이렇게 해도 문제될건 없습니다.

### Address 관련

**<address>.balance (uint256):** *Address* 의 잔액(Wei 단위)

**<address>.transfer(uint256 amount):** 주어진 양만큼의 Wei를 *Address* 로 전송합니다. 실패시 에러를 발생시키고 2300 gas를 전달하며 이 값은 변경할 수 없습니다.

**<address>.send(uint256 amount) returns (bool):** 주어진 양만큼의 Wei를 *Address* 로 전송합니다. 실패시 `false` 를 반환하고 2300 gas를 전달하며 이 값은 변경할 수 없습니다.

**<address>.call(...) returns (bool):** 로우 레벨 수준에서의 `CALL` 을 수행합니다. 실패시 `false` 를 반환하고 모든 gas를 전달하며 이 값은 변경가능합니다.

**<address>.callcode(...) returns (bool):** 로우 레벨 수준에서의 `CALLCODE` 를 수행합니다. 실패시 `false` 를 반환하고 모든 gas를 전달하며 이 값은 변경가능합니다.

**<address>.delegatecall(...) returns (bool):** 로우 레벨 수준에서의 DELEGATECALL 을 수행합니다. 실패시 false 를 반환하고 모든 gas를 전달하며 이 값은 변경가능합니다.

자세한 내용은 *Address* 섹션을 참조하십시오.

**경고:** send 를 사용할 땐 몇가지 주의사항이 있습니다: call stack의 깊이가 1024라면 전송은 실패하며(이것은 항상 호출자에 의해 강제 될 수 있습니다) 그리고 수신자의 gas가 전부 소모되어도 실패합니다. 그러므로 안전한 Ether 전송을 위해서, 항상 send 의 반환값을 확인하고, transfer 를 사용하세요: 혹은 더 좋은 방법은 수신자가 돈을 인출하는 패턴을 사용하는 것입니다.

**주석:** callcode 의 사용은 권장되지 않으며 추후 제거 될 예정입니다.

#### 컨트랙트 관련

**this (current contract's type):** 현재의 컨트랙트, *Address* 로 명시적 변환 가능합니다.

**selfdestruct(address recipient):** 현재의 컨트랙트를 파기하고 자금을 주어진 *Address* 로 전송합니다.

**suicide(address recipient):** selfdestruct 의 별칭

뿐만 아니라, 현재 컨트랙트의 모든 함수는 직접적으로 호출 가능합니다.

### 8.4.5 Expressions and Control Structures

#### Input Parameters and Output Parameters

As in Javascript, functions may take parameters as input; unlike in Javascript and C, they may also return arbitrary number of parameters as output.

#### Input Parameters

The input parameters are declared the same way as variables are. The name of unused parameters can be omitted. For example, suppose we want our contract to accept one kind of external calls with two integers, we would write something like:

```
pragma solidity >=0.4.16 <0.6.0;

contract Simple {
    uint sum;
    function taker(uint _a, uint _b) public {
        sum = _a + _b;
    }
}
```

Input parameters can be used just as any other local variable can be used, they can also be assigned to.

### Output Parameters

The output parameters can be declared with the same syntax after the `returns` keyword. For example, suppose we wished to return two results: the sum and the product of the two given integers, then we would write:

```
pragma solidity >=0.4.16 <0.6.0;

contract Simple {
    function arithmetic(uint _a, uint _b)
        public
        pure
        returns (uint o_sum, uint o_product)
    {
        o_sum = _a + _b;
        o_product = _a * _b;
    }
}
```

The names of output parameters can be omitted. The output values can also be specified using `return` statements, which are also capable of *returning multiple values*. Return parameters can be used as any other local variable and they are zero-initialized; if they are not explicitly set, they stay zero.

### Control Structures

Most of the control structures known from curly-braces languages are available in Solidity:

There is: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, with the usual semantics known from C or JavaScript.

Parentheses can *not* be omitted for conditionals, but curly brances can be omitted around single-statement bodies.

Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so `if (1) { ... }` is *not* valid Solidity.

### Returning Multiple Values

When a function has multiple output parameters, `return (v0, v1, ..., vn)` can return multiple values. The number of components must be the same as the number of output parameters.

### 함수 호출

### 내부 함수 호출

예제에서 처럼 현재 contract의 함수는 직접적으로(내부적으로) 또는 재귀적으로 호출 될 수 있습니다.:

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

이런 함수 calls은 EVM 내부의 간단한 jumps로 번역 될 수 있습니다. 이것은 현재 돈이 정리되지 않았을 때 내부적으로 호출된 함수에 대한 메모리 참조 변환이 매우 효과적이다. 같은 contract 함수에 대해서만 내부적으로 호출 될 수 있습니다.

You should still avoid excessive recursion, as every internal function call uses up at least one stack slot and there are at most 1024 slots available.

### 외부 함수 호출

표현식 `this.g(8);` 와 `c.g(2);` (g는 contract 인스턴스)은 유효한 함수 calls입니다. 그러나 함수는 jumps가 아닌 메시지 call을 통해 외부에서 불려집니다. 실제 contrat가 아직 생성되지 않았기 때문에 생성자에서 함수 calls은 사용 될 수 없습니다.

다른 contracts의 함수는 외부적으로 호출되어야 합니다. 외부 호출을 위해 모든 함수 arguments는 메모리에 복사되어야 합니다.

---

**주석:** A function call from one contract to another does not create its own transaction, it is a message call as part of the overall transaction.

---

다른 contracts의 함수를 부를 때, call과 함께 보내진 Wei와 gas는 각각 `.value()` 와 `.gas()` 로 명시될 수 있습니다.:

```
pragma solidity >=0.4.0 <0.6.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info.value(10).gas(800)(); }
}
```

You need to use the modifier `payable` with the `info` function because otherwise, the `.value()` option would not be available.

**경고:** Be careful that `feed.info.value(10).gas(800)` only locally sets the `value` and amount of `gas` sent with the function call, and the parentheses at the end perform the actual call. So in this case, the function is not called.

함수 호출은 호출된 contract가 존재하지 않거나(계좌가 코드를 포함하지 않는 다는 점에서), 호출된 contract가 스스로 예외처리를 하거나 gas가 없으면 예외를 발생시킵니다.

**경고:** Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the

calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.

### 지정 호출과 익명 함수 parameters

다음의 예에서 볼 수 있는 것처럼 `{ }` 로 묶여 있다면, 함수 호출 arguments는 순서와 상관없이 이름으로 지정 될 수 있습니다. argument list는 함수 선언에서 parameters 리스트와 이름이 일치해야 하지만 순서는 일치 되지 않을 수 있습니다.

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }

}
```

### 제거된 함수 parameter 이름

사용되지 않을 parameters(특히 반환 parameters)의 이름은 제거될 수 있습니다. 이런 parameters의 이름은 스택에 존재하지만 접근할 수 없습니다.

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    // omitted name for parameter
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

### Creating Contracts via `new`

A contract can create other contracts using the `new` keyword. The full code of the contract being created has to be known when the creating contract is compiled so recursive creation-dependencies are not possible.

```
pragma solidity >0.4.99 <0.6.0;

contract D {
    uint public x;
    constructor(uint a) public payable {
        x = a;
    }
}
```

(continues on next page)

(이전 페이지에서 계속)

```
contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = (new D).value(amount)(arg);
        newD.x();
    }
}
```

As seen in the example, it is possible to send Ether while creating an instance of `D` using the `.value()` option, but it is not possible to limit the amount of gas. If the creation fails (due to out-of-stack, not enough balance or other problems), an exception is thrown.

### Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done. See *Order of Precedence of Operators* for more information.

### Assignment

### Destructuring Assignments and Returning Multiple Values

Solidity internally allows tuple types, i.e. a list of objects of potentially different types whose number is a constant at compile-time. Those tuples can be used to return multiple values at the same time. These can then either be assigned to newly declared variables or to pre-existing variables (or LValues in general).

Tuples are not proper types in Solidity, they can only be used to form syntactic groupings of expressions.

```
pragma solidity >0.4.23 <0.6.0;

contract C {
    uint[] data;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // Common trick to swap values -- does not work for non-value storage types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
```

(continues on next page)

(이전 페이지에서 계속)

```
        (data.length, , ) = f(); // Sets the length to 7
    }
}
```

It is not possible to mix variable declarations and non-declaration assignments, i.e. the following is not valid: `(x, uint y) = (1, 2);`

**주석:** Prior to version 0.5.0 it was possible to assign to tuples of smaller size, either filling up on the left or on the right side (which ever was empty). This is now disallowed, so both sides have to have the same number of components.

**경고:** Be careful when assigning to multiple variables at the same time when reference types are involved, because it could lead to unexpected copying behaviour.

### Complications for Arrays and Structs

The semantics of assignments are a bit more complicated for non-value types like arrays and structs. Assigning *to* a state variable always creates an independent copy. On the other hand, assigning to a local variable creates an independent copy only for elementary types, i.e. static types that fit into 32 bytes. If structs or arrays (including `bytes` and `string`) are assigned from a state variable to a local variable, the local variable holds a reference to the original state variable. A second assignment to the local variable does not modify the state but only changes the reference. Assignments to members (or elements) of the local variable *do* change the state.

### Scoping and Declarations

A variable which is declared will have an initial default value whose byte-representation is all zeros. The "default values" of variables are the typical "zero-state" of whatever the type is. For example, the default value for a `bool` is `false`. The default value for the `uint` or `int` types is `0`. For statically-sized arrays and `bytes1` to `bytes32`, each individual element will be initialized to the default value corresponding to its type. Finally, for dynamically-sized arrays, `bytes` and `string`, the default value is an empty array or string.

Scoping in Solidity follows the widespread scoping rules of C99 (and many other languages): Variables are visible from the point right after their declaration until the end of the smallest `{ }`-block that contains the declaration. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.

Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

As a consequence, the following examples will compile without warnings, since the two variables have the same name but disjoint scopes.

```
pragma solidity >0.4.99 <0.6.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }
```

(continues on next page)

(이전 페이지에서 계속)

```
        {
            uint same;
            same = 3;
        }
    }
}
```

As a special example of the C99 scoping rules, note that in the following, the first assignment to `x` will actually assign the outer and not the inner variable. In any case, you will get a warning about the outer variable being shadowed.

```
pragma solidity >0.4.99 <0.6.0;
// This will report a warning
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // this will assign to the outer variable
            uint x;
        }
        return x; // x has value 2
    }
}
```

**경고:**

Before version 0.5.0 Solidity followed the same scoping rules as JavaScript, that is, a variable declared anywhere within a function would be in scope for the entire function, regardless where it was declared. The following example shows a code snippet that used to compile but leads to an error starting from version 0.5.0.

```
pragma solidity >0.4.99 <0.6.0;
// This will not compile
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

### Error handling: Assert, Require, Revert and Exceptions

Solidity uses state-reverting exceptions to handle errors. Such an exception will undo all changes made to the state in the current call (and all its sub-calls) and also flag an error to the caller. The convenience functions `assert` and `require` can be used to check for conditions and throw an exception if the condition is not met. The `assert` function should only be used to test for internal errors, and to check invariants. The `require` function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts. If used properly, analysis tools can evaluate your contract to identify the conditions and function calls which will reach a failing `assert`. Properly functioning code should never reach a failing assert statement; if this happens there is a bug in your contract which you should fix.

There are two other ways to trigger exceptions: The `revert` function can be used to flag an error and revert the current call. It is possible to provide a string message containing details about the error that will be passed back to the

caller.

---

**주석:** There used to be a keyword called `throw` with the same semantics as `revert()` which was deprecated in version 0.4.13 and removed in version 0.5.0.

---

When exceptions happen in a sub-call, they "bubble up" (i.e. exceptions are rethrown) automatically. Exceptions to this rule are `send` and the low-level functions `call`, `delegatecall` and `staticcall` – those return `false` as their first return value in case of an exception instead of "bubbling up".

**경고:** The low-level functions `call`, `delegatecall` and `staticcall` return `true` as their first return value if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

Catching exceptions is not yet possible.

In the following example, you can see how `require` can be used to easily check conditions on inputs and how `assert` can be used for internal error checking. Note that you can optionally provide a message string for `require`, but not for `assert`.

```
pragma solidity >0.4.99 <0.6.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

An `assert`-style exception is generated in the following situations:

1. If you access an array at a too large or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).
2. If you access a fixed-length `bytesN` at a too large or negative index.
3. If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).
4. If you shift by a negative amount.
5. If you convert a value too big or negative into an enum type.
6. If you call a zero-initialized variable of internal function type.
7. If you call `assert` with an argument that evaluates to false.

A `require`-style exception is generated in the following situations:

1. Calling `require` with an argument that evaluates to `false`.
2. If you call a function via a message call but it does not finish properly (i.e. it runs out of gas, has no matching function, or throws an exception itself), except when a low level operation `call`, `send`, `delegatecall`, `callcode` or `staticcall` is used. The low level operations never throw exceptions but indicate failures by returning `false`.

3. If you create a contract using the `new` keyword but the contract creation does not finish properly (see above for the definition of "not finish properly").
4. If you perform an external function call targeting a contract that contains no code.
5. If your contract receives Ether via a public function without `payable` modifier (including the constructor and the fallback function).
6. If your contract receives Ether via a public getter function.
7. If a `.transfer()` fails.

Internally, Solidity performs a revert operation (instruction `0xfd`) for a `require`-style exception and executes an invalid operation (instruction `0xfe`) to throw an `assert`-style exception. In both cases, this causes the EVM to revert all changes made to the state. The reason for reverting is that there is no safe way to continue execution, because an expected effect did not occur. Because we want to retain the atomicity of transactions, the safest thing to do is to revert all changes and make the whole transaction (or at least call) without effect. Note that `assert`-style exceptions consume all gas available to the call, while `require`-style exceptions will not consume any gas starting from the Metropolis release.

The following example shows how an error string can be used together with revert and require:

```
pragma solidity >0.4.99 <0.6.0;

contract VendingMachine {
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
}
```

The provided string will be *abi-encoded* as if it were a call to a function `Error(string)`. In the above example, `revert("Not enough Ether provided.");` will cause the following hexadecimal data be set as error return data:

```
0x08c379a0                                                         // Function␣
↪selector for Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Data offset
0x000000000000000000000000000000000000000000000000000000000000001a // String length
0x4e6f7420656e6f756768204574686572207072696465642e000000000000 // String data
```

### 8.4.6 Contracts

Contracts in Solidity are similar to classes in object-oriented languages. They contain persistent data in state variables and functions that can modify these variables. Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables are inaccessible.

#### Creating Contracts

Contracts can be created "from outside" via Ethereum transactions or from within Solidity contracts.

IDEs, such as Remix, make the creation process seamless using UI elements.

Creating contracts programmatically on Ethereum is best done via using the JavaScript API web3.js. It has a function called web3.eth.Contract to facilitate contract creation.

When a contract is created, its *constructor* (a function declared with the `constructor` keyword) is executed once.

A constructor is optional. Only one constructor is allowed, which means overloading is not supported.

After the constructor has executed, the final code of the contract is deployed to the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.

Internally, constructor arguments are passed *ABI encoded* after the code of the contract itself, but you do not have to care about this if you use `web3.js`.

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. This means that cyclic creation dependencies are impossible.

```
pragma solidity >=0.4.22 <0.6.0;

contract OwnedToken {
    // TokenCreator is a contract type that is defined below.
    // It is fine to reference it as long as it is not used
    // to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;

    // This is the constructor which registers the
    // creator and the assigned name.
    constructor(bytes32 _name) public {
        // State variables are accessed via their name
        // and not via e.g. this.owner. This also applies
        // to functions and especially in the constructors,
        // you can only call them like that ("internally"),
        // because the contract itself does not exist yet.
        owner = msg.sender;
        // We do an explicit type conversion from `address`
        // to `TokenCreator` and assume that the type of
        // the calling contract is TokenCreator, there is
        // no real way to check that.
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) public {
        // Only the creator can alter the name --
        // the comparison is possible since contracts
        // are explicitly convertible to addresses.
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;

        // We also want to ask the creator if the transfer
        // is fine. Note that this calls a function of the
```

(continues on next page)

(이전 페이지에서 계속)

```
        // contract defined below. If the call fails (e.g.
        // due to out-of-gas), the execution also fails here.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

contract TokenCreator {
    function createToken(bytes32 name)
       public
       returns (OwnedToken tokenAddress)
    {
        // Create a new Token contract and return its address.
        // From the JavaScript side, the return type is simply
        // `address`, as this is the closest type available in
        // the ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
        // Again, the external type of `tokenAddress` is
        // simply `address`.
        tokenAddress.changeName(name);
    }

    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // Check some arbitrary condition.
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

### Visibility and Getters

Since Solidity knows two kinds of function calls (internal ones that do not create an actual EVM call (also called a "message call") and external ones that do), there are four types of visibilities for functions and state variables.

Functions have to be specified as being `external`, `public`, `internal` or `private`. For state variables, `external` is not possible.

**`external`:** External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works). External functions are sometimes more efficient when they receive large arrays of data.

**`public`:** Public functions are part of the contract interface and can be either called internally or via messages. For public state variables, an automatic getter function (see below) is generated.

**`internal`:** Those functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it), without using `this`.

**`private`:** Private functions and state variables are only visible for the contract they are defined in and not in derived contracts.

**주석:** Everything that is inside a contract is visible to all observers external to the blockchain. Making something `private` only prevents other contracts from accessing and modifying the information, but it will still be visible to the whole world outside of the blockchain.

The visibility specifier is given after the type for state variables and between parameter list and return parameter list for functions.

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

In the following example, `D`, can call `c.getData()` to retrieve the value of `data` in state storage, but is not able to call `f`. Contract `E` is derived from `C` and, thus, can call `compute`.

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }
    function setData(uint a) public { data = a; }
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
}

// This will not compile
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // error: member `f` is not visible
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // error: member `compute` is not visible
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // access to internal member (from derived to␣
↪parent contract)
    }
}
```

### Getter Functions

The compiler automatically creates getter functions for all **public** state variables. For the contract given below, the compiler will generate a function called `data` that does not take any arguments and returns a `uint`, the value of the state variable `data`. State variables can be initialized when they are declared.

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

The getter functions have external visibility. If the symbol is accessed internally (i.e. without `this.`), it evaluates to a state variable. If it is accessed externally (i.e. with `this.`), it evaluates to a function.

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // internal access
        return this.data(); // external access
    }
}
```

If you have a `public` state variable of array type, then you can only retrieve single elements of the array via the generated getter function. This mechanism exists to avoid high gas costs when returning an entire array. You can use arguments to specify which individual element to return, for example `data(0)`. If you want to return an entire array in one call, then you need to write a function, for example:

```
pragma solidity >=0.4.0 <0.6.0;

contract arrayExample {
  // public state variable
  uint[] public myArray;

  // Getter function generated by the compiler
  /*
  function myArray(uint i) returns (uint) {
      return myArray[i];
  }
  */

  // function that returns entire array
  function getArray() returns (uint[] memory) {
      return myArray;
  }
}
```

Now you can use `getArray()` to retrieve the entire array, instead of `myArray(i)`, which returns a single element per call.

The next example is more complex:

```
pragma solidity >=0.4.0 <0.6.0;
```

(continues on next page)

(이전 페이지에서 계속)

```
contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping (uint => uint) map;
    }
    mapping (uint => mapping(bool => Data[])) public data;
}
```

It generates a function of the following form. The mapping in the struct is omitted because there is no good way to provide the key for the mapping:

```
function data(uint arg1, bool arg2, uint arg3) public returns (uint a, bytes3 b) {
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
}
```

### Function Modifiers

Modifiers can be used to easily change the behaviour of functions. For example, they can automatically check a condition prior to executing the function. Modifiers are inheritable properties of contracts and may be overridden by derived contracts.

```
pragma solidity >0.4.99 <0.6.0;

contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;

    // This contract only defines a modifier but does not use
    // it: it will be used in derived contracts.
    // The function body is inserted where the special symbol
    // `_;` in the definition of a modifier appears.
    // This means that if the owner calls this function, the
    // function is executed and otherwise, an exception is
    // thrown.
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract mortal is owned {
    // This contract inherits the `onlyOwner` modifier from
    // `owned` and applies it to the `close` function, which
    // causes that calls to `close` only have an effect if
    // they are made by the stored owner.
    function close() public onlyOwner {
        selfdestruct(owner);
    }
}
```

(continues on next page)

(이전 페이지에서 계속)

```
contract priced {
    // Modifiers can receive arguments:
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is priced, owned {
    mapping (address => bool) registeredAddresses;
    uint price;

    constructor(uint initialPrice) public { price = initialPrice; }

    // It is important to also provide the
    // `payable` keyword here, otherwise the function will
    // automatically reject all Ether sent to it.
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// This function is protected by a mutex, which means that
    /// reentrant calls from within `msg.sender.call` cannot call `f` again.
    /// The `return 7` statement assigns 7 to the return value but still
    /// executes the statement `locked = false` in the modifier.
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.

**경고:** In an earlier version of Solidity, `return` statements in functions having modifiers behaved differently.

Explicit returns from a modifier or function body only leave the current modifier or function body. Return variables are assigned and control flow continues after the "_" in the preceding modifier.

Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

### Constant State Variables

State variables can be declared as `constant`. In this case, they have to be assigned from an expression which is a constant at compile time. Any expression that accesses storage, blockchain data (e.g. `now`, `address(this).balance` or `block.number`) or execution data (`msg.value` or `gasleft()`) or makes calls to external contracts is disallowed. Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not. The built-in functions `keccak256`, `sha256`, `ripemd160`, `ecrecover`, `addmod` and `mulmod` are allowed (even though they do call external contracts).

The reason behind allowing side-effects on the memory allocator is that it should be possible to construct complex objects like e.g. lookup-tables. This feature is not yet fully usable.

The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective constant expression (which might be computed to a single value by the optimizer).

Not all types for constants are implemented at this time. The only supported types are value types and strings.

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    uint constant x = 32**22 + 8;
    string constant text = "abc";
    bytes32 constant myHash = keccak256("abc");
}
```

## Functions

### View Functions

Functions can be declared `view` in which case they promise not to modify the state.

**주석:** If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used for `view` functions which enforces the state to stay unmodified as part of the EVM execution. For library `view` functions `DELEGATECALL` is used, because there is no combined `DELEGATECALL` and `STATICCALL`. This means library `view` functions do not have run-time checks that prevent state modifications. This should not impact security negatively because library code is usually known at compile-time and the static checker performs compile-time checks.

The following statements are considered modifying the state:

1. Writing to state variables.
2. *Emitting events*.
3. *Creating other contracts*.

4. Using `selfdestruct`.
5. Sending Ether via calls.
6. Calling any function not marked `view` or `pure`.
7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

```
pragma solidity >0.4.99 <0.6.0;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + now;
    }
}
```

**주석:** `constant` on functions used to be an alias to `view`, but this was dropped in version 0.5.0.

**주석:** Getter methods are automatically marked `view`.

**주석:** Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `view` functions. This enabled state modifications in `view` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `view` functions, modifications to the state are prevented on the level of the EVM.

### Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state.

**주석:** If the compiler's EVM target is Byzantium or newer (default) the opcode `STATICCALL` is used, which does not guarantee that the state is not read, but at least that it is not modified.

In addition to the list of state modifying statements explained above, the following are considered reading from the state:

1. Reading from state variables.
2. Accessing `address(this).balance` or `<address>.balance`.
3. Accessing any of the members of `block`, `tx`, `msg` (with the exception of `msg.sig` and `msg.data`).
4. Calling any function not marked `pure`.
5. Using inline assembly that contains certain opcodes.

```
pragma solidity >0.4.99 <0.6.0;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

**주석:** Prior to version 0.5.0, the compiler did not use the `STATICCALL` opcode for `pure` functions. This enabled state modifications in `pure` functions through the use of invalid explicit type conversions. By using `STATICCALL` for `pure` functions, modifications to the state are prevented on the level of the EVM.

**경고:** It is not possible to prevent functions from reading the state at the level of the EVM, it is only possible to prevent them from writing to the state (i.e. only `view` can be enforced at the EVM level, `pure` can not).

**경고:** Before version 0.4.17 the compiler did not enforce that `pure` is not reading the state. It is a compile-time type check, which can be circumvented doing invalid explicit conversions between contract types, because the compiler can verify that the type of the contract does not do state-changing operations, but it cannot check that the contract that will be called at runtime is actually of that type.

### Fallback Function

A contract can have exactly one unnamed function. This function cannot have arguments, cannot return anything and has to have `external` visibility. It is executed on a call to the contract if none of the other functions match the given function identifier (or if no data was supplied at all).

Furthermore, this function is executed whenever the contract receives plain Ether (without data). Additionally, in order to receive Ether, the fallback function must be marked `payable`. If no such function exists, the contract cannot receive Ether through regular transactions.

In the worst case, the fallback function can only rely on 2300 gas being available (for example when *send* or *transfer* is used), leaving little room to perform other operations except basic logging. The following operations will consume more gas than the 2300 gas stipend:

- Writing to storage
- Creating a contract
- Calling an external function which consumes a large amount of gas
- Sending Ether

Like any function, the fallback function can execute complex operations as long as there is enough gas passed on to it.

**주석:** Even though the fallback function cannot have arguments, one can still use `msg.data` to retrieve any payload supplied with the call.

**경고:** The fallback function is also executed if the caller meant to call a function that is not available. If you want to implement the fallback function only to receive ether, you should add a check like `require(msg.data.length == 0)` to prevent invalid calls.

**경고:** Contracts that receive Ether directly (without a function call, i.e. using `send` or `transfer`) but do not define a fallback function throw an exception, sending back the Ether (this was different before Solidity v0.4.0). So if you want your contract to receive Ether, you have to implement a payable fallback function.

**경고:** A contract without a payable fallback function can receive Ether as a recipient of a *coinbase transaction* (aka *miner block reward*) or as a destination of a `selfdestruct`.

A contract cannot react to such Ether transfers and thus also cannot reject them. This is a design choice of the EVM and Solidity cannot work around it.

It also means that `address(this).balance` can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the fallback function).

```
pragma solidity >0.4.99 <0.6.0;

contract Test {
    // This function is called for all messages sent to
    // this contract (there is no other function).
    // Sending Ether to this contract will cause an exception,
    // because the fallback function does not have the `payable`
    // modifier.
    function() external { x = 1; }
    uint x;
}


// This contract keeps all Ether sent to it with no way
// to get it back.
contract Sink {
    function() external payable { }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // results in test.x becoming == 1.

        // address(test) will not allow to call ``send`` directly, since ``test`` has␣
↪no payable
        // fallback function. It has to be converted to the ``address payable`` type␣
↪via an
        // intermediate conversion to ``uint160`` to even allow calling ``send`` on␣
↪it.
        address payable testPayable = address(uint160(address(test)));

        // If someone sends ether to that contract,
        // the transfer will fail, i.e. this returns false here.
        return testPayable.send(2 ether);
    }
}
```

### Function Overloading

A contract can have multiple functions of the same name but with different parameter types. This process is called "overloading" and also applies to inherited functions. The following example shows overloading of the function `f` in the scope of contract `A`.

```
pragma solidity >=0.4.16 <0.6.0;

contract A {
    function f(uint _in) public pure returns (uint out) {
        out = _in;
    }

    function f(uint _in, bool _really) public pure returns (uint out) {
        if (_really)
            out = _in;
    }
}
```

Overloaded functions are also present in the external interface. It is an error if two externally visible functions differ by their Solidity types but not by their external types.

```
pragma solidity >=0.4.16 <0.6.0;

// This will not compile
contract A {
    function f(B _in) public pure returns (B out) {
        out = _in;
    }

    function f(address _in) public pure returns (address out) {
        out = _in;
    }
}

contract B {
}
```

Both `f` function overloads above end up accepting the address type for the ABI although they are considered different inside Solidity.

### Overload resolution and Argument matching

Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call. Functions are selected as overload candidates if all arguments can be implicitly converted to the expected types. If there is not exactly one candidate, resolution fails.

**주석:** Return parameters are not taken into account for overload resolution.

```
pragma solidity >=0.4.16 <0.6.0;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

Calling `f(50)` would create a type error since `50` can be implicitly converted both to `uint8` and `uint256` types. On another hand `f(256)` would resolve to `f(uint256)` overload as `256` cannot be implicitly converted to `uint8`.

### Events

Solidity events give an abstraction on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible (forever as of the Frontier and Homestead releases, but this might change with Serenity). The Log and its event data is not accessible from within contracts (not even from the contract that created them).

It is possible to request a simple payment verification (SPV) for logs, so if an external entity supplies a contract with such a verification, it can check that the log actually exists inside the blockchain. You have to supply block headers because the contract can only see the last 256 block hashes.

You can add the attribute `indexed` to up to three parameters which adds them to a special data structure known as "topics" instead of the data part of the log. If you use arrays (including `string` and `bytes`) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a topic can only hold a single word (32 bytes).

All parameters without the `indexed` attribute are *ABI-encoded* into the data part of the log.

Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

For example, the code below uses the web3.js `subscribe("logs")` method to filter logs that match a topic with a certain address value:

```
var options = {
    fromBlock: 0,
    address: web3.eth.defaultAccount,
    topics: ["0x0000000000000000000000000000000000000000000000000000000000000000",␣
↪null, null]
};
web3.eth.subscribe('logs', options, function (error, result) {
    if (!error)
        console.log(result);
})
    .on("data", function (log) {
        console.log(log);
    })
    .on("changed", function (log) {
});
```

The hash of the signature of the event is one of the topics, except if you declared the event with the `anonymous` specifier. This means that it is not possible to filter for specific anonymous events by name.

```
pragma solidity >=0.4.21 <0.6.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );
```

(continues on next page)

(이전 페이지에서 계속)

```
    function deposit(bytes32 _id) public payable {
        // Events are emitted using `emit`, followed by
        // the name of the event and the arguments
        // (if any) in parentheses. Any such invocation
        // (even deeply nested) can be detected from
        // the JavaScript API by filtering for `Deposit`.
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

The use in the JavaScript API is as follows:

```
var abi = /* abi as generated by the compiler */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* address */);

var event = clientReceipt.Deposit();

// watch for changes
event.watch(function(error, result){
    // result contains non-indexed arguments and topics
    // given to the `Deposit` call.
    if (!error)
        console.log(result);
});


// Or pass a callback to start watching immediately
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

The output of the above looks like the following (trimmed):

```
{
   "returnValues": {
       "_from": "0x1111...FFFFCCCC",
       "_id": "0x50...sd5adb20",
       "_value": "0x420042"
   },
   "raw": {
       "data": "0x7f...91385",
       "topics": ["0xfd4...b4ead7", "0x7f...1a91385"]
   }
}
```

### Low-Level Interface to Logs

It is also possible to access the low-level interface to the logging mechanism via the functions `log0`, `log1`, `log2`, `log3` and `log4`. `logi` takes `i + 1` parameter of type `bytes32`, where the first argument will be used for the data part of the log and the others as topics. The event call above can be performed in the same way as

```
pragma solidity >=0.4.10 <0.6.0;
```

(continues on next page)

(이전 페이지에서 계속)

```
contract C {
    function f() public payable {
        uint256 _id = 0x420042;
        log3(
            bytes32(msg.value),
            bytes32(0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20),
            bytes32(uint256(msg.sender)),
            bytes32(_id)
        );
    }
}
```

where the long hexadecimal number is equal to `keccak256("Deposit(address,bytes32,uint256)")`, the signature of the event.

### Additional Resources for Understanding Events

- Javascript documentation
- Example usage of events
- How to access them in js

### Inheritance

Solidity supports multiple inheritance by copying code including polymorphism.

All function calls are virtual, which means that the most derived function is called, except when the contract name is explicitly given.

When a contract inherits from other contracts, only a single contract is created on the blockchain, and the code from all the base contracts is copied into the created contract.

The general inheritance system is very similar to Python's, especially concerning multiple inheritance, but there are also some *differences*.

Details are given in the following example.

```
pragma solidity >0.4.99 <0.6.0;

contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;
}

// Use `is` to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
```

(continues on next page)

(이전 페이지에서 계속)

```
// These abstract contracts are only provided to make the
// interface known to the compiler. Note the function
// without body. If a contract does not implement all
// functions it can only be used as an interface.
contract Config {
    function lookup(uint id) public returns (address adr);
}

contract NameReg {
    function register(bytes32 name) public;
    function unregister() public;
 }

// Multiple inheritance is possible. Note that `owned` is
// also a base class of `mortal`, yet there is only a single
// instance of `owned` (as for virtual inheritance in C++).
contract named is owned, mortal {
    constructor(bytes32 name) public {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // Functions can be overridden by another function with the same name and
    // the same number/types of inputs.  If the overriding function has different
    // types of output parameters, that causes an error.
    // Both local and message-based function calls take these overrides
    // into account.
    function kill() public {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // It is still possible to call a specific
            // overridden function.
            mortal.kill();
        }
    }
}

// If a constructor takes an argument, it needs to be
// provided in the header (or modifier-invocation-style at
// the constructor of the derived contract (see below)).
contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) public {
      if (msg.sender == owner) info = newInfo;
   }

   function get() public view returns(uint r) { return info; }

   uint info;
}
```

Note that above, we call `mortal.kill()` to "forward" the destruction request. The way this is done is problematic, as seen in the following example:

```
pragma solidity >=0.4.22 <0.6.0;
```

(continues on next page)

(이전 페이지에서 계속)

```
contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* do cleanup 1 */ mortal.kill(); }
}

contract Base2 is mortal {
    function kill() public { /* do cleanup 2 */ mortal.kill(); }
}

contract Final is Base1, Base2 {
}
```

A call to `Final.kill()` will call `Base2.kill` as the most derived override, but this function will bypass `Base1.kill`, basically because it does not even know about `Base1`. The way around this is to use `super`:

```
pragma solidity >=0.4.22 <0.6.0;

contract owned {
    constructor() public { owner = msg.sender; }
    address payable owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* do cleanup 1 */ super.kill(); }
}


contract Base2 is mortal {
    function kill() public { /* do cleanup 2 */ super.kill(); }
}

contract Final is Base1, Base2 {
}
```

If `Base2` calls a function of `super`, it does not simply call this function on one of its base contracts. Rather, it calls this function on the next base contract in the final inheritance graph, so it will call `Base1.kill()` (note that the final inheritance sequence is – starting with the most derived contract: Final, Base2, Base1, mortal, owned). The actual function that is called when using super is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

### Constructors

A constructor is an optional function declared with the `constructor` keyword which is executed upon contract creation, and where you can run contract initialisation code.

Before the constructor code is executed, state variables are initialised to their specified value if you initialise them inline, or zero if you do not.

After the constructor has run, the final code of the contract is deployed to the blockchain. The deployment of the code costs additional gas linear to the length of the code. This code includes all functions that are part of the public interface and all functions that are reachable from there through function calls. It does not include the constructor code or internal functions that are only called from the constructor.

Constructor functions can be either `public` or `internal`. If there is no constructor, the contract will assume the default constructor, which is equivalent to `constructor() public {}`. For example:

```
pragma solidity >0.4.99 <0.6.0;

contract A {
    uint public a;

    constructor(uint _a) internal {
        a = _a;
    }
}

contract B is A(1) {
    constructor() public {}
}
```

A constructor set as `internal` causes the contract to be marked as *abstract*.

**경고:** Prior to version 0.4.22, constructors were defined as functions with the same name as the contract. This syntax was deprecated and is not allowed anymore in version 0.5.0.

### Arguments for Base Constructors

The constructors of all the base contracts will be called following the linearization rules explained below. If the base constructors have arguments, derived contracts need to specify all of them. This can be done in two ways:

```
pragma solidity >=0.4.22 <0.6.0;

contract Base {
    uint x;
    constructor(uint _x) public { x = _x; }
}

// Either directly specify in the inheritance list...
contract Derived1 is Base(7) {
    constructor() public {}
}

// or through a "modifier" of the derived constructor.
contract Derived2 is Base {
```

(continues on next page)

(이전 페이지에서 계속)

```
    constructor(uint _y) Base(_y * _y) public {}
}
```

One way is directly in the inheritance list (`is Base(7)`). The other is in the way a modifier is invoked as part of the derived constructor (`Base(_y * _y)`). The first way to do it is more convenient if the constructor argument is a constant and defines the behaviour of the contract or describes it. The second way has to be used if the constructor arguments of the base depend on those of the derived contract. Arguments have to be given either in the inheritance list or in modifier-style in the derived constructor. Specifying arguments in both places is an error.

If a derived contract does not specify the arguments to all of its base contracts' constructors, it will be abstract.

### Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems. One is the Diamond Problem. Solidity is similar to Python in that it uses "C3 Linearization" to force a specific order in the directed acyclic graph (DAG) of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the `is` directive is important: You have to list the direct base contracts in the order from "most base-like" to "most derived". Note that this order is the reverse of the one used in Python.

Another simplifying way to explain this is that when a function is called that is defined multiple times in different contracts, the given bases are searched from right to left (left to right in Python) in a depth-first manner, stopping at the first match. If a base contract has already been searched, it is skipped.

In the following code, Solidity will give the error "Linearization of inheritance graph impossible".

```
pragma solidity >=0.4.0 <0.6.0;

contract X {}
contract A is X {}
// This will not compile
contract C is A, X {}
```

The reason for this is that `C` requests `X` to override `A` (by specifying `A, X` in this order), but `A` itself requests to override `X`, which is a contradiction that cannot be resolved.

### Inheriting Different Kinds of Members of the Same Name

When the inheritance results in a contract with a function and a modifier of the same name, it is considered as an error. This error is produced also by an event and a modifier of the same name, and a function and an event of the same name. As an exception, a state variable getter can override a public function.

### Abstract Contracts

Contracts are marked as abstract when at least one of their functions lacks an implementation as in the following example (note that the function declaration header is terminated by `;`):

```
pragma solidity >=0.4.0 <0.6.0;

contract Feline {
    function utterance() public returns (bytes32);
}
```

Such contracts cannot be compiled (even if they contain implemented functions alongside non-implemented functions), but they can be used as base contracts:

```
pragma solidity >=0.4.0 <0.6.0;

contract Feline {
    function utterance() public returns (bytes32);
}

contract Cat is Feline {
    function utterance() public returns (bytes32) { return "miaow"; }
}
```

If a contract inherits from an abstract contract and does not implement all non-implemented functions by overriding, it will itself be abstract.

Note that a function without implementation is different from a *Function Type* even though their syntax looks very similar.

Example of function without implementation (a function declaration):

```
function foo(address) external returns (address);
```

Example of a Function Type (a variable declaration, where the variable is of type `function`):

```
function(address) external returns (address) foo;
```

Abstract contracts decouple the definition of a contract from its implementation providing better extensibility and self-documentation and facilitating patterns like the Template method and removing code duplication. Abstract contracts are useful in the same way that defining methods in an interface is useful. It is a way for the designer of the abstract contract to say "any child of mine must implement this method".

### Interfaces

Interfaces are similar to abstract contracts, but they cannot have any functions implemented. There are further restrictions:

- They cannot inherit other contracts or interfaces.
- All declared functions must be external.
- They cannot declare a constructor.
- They cannot declare state variables.

Some of these restrictions might be lifted in the future.

Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

Interfaces are denoted by their own keyword:

```
pragma solidity >=0.4.11 <0.6.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

Contracts can inherit interfaces as they would inherit other contracts.

Types defined inside interfaces and other contract-like structures can be accessed from other contracts: `Token.TokenType` or `Token.Coin`.

### Libraries

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused using the `DELEGATECALL` (`CALLCODE` until Homestead) feature of the EVM. This means that if library functions are called, their code is executed in the context of the calling contract, i.e. `this` points to the calling contract, and especially the storage from the calling contract can be accessed. As a library is an isolated piece of source code, it can only access state variables of the calling contract if they are explicitly supplied (it would have no way to name them, otherwise). Library functions can only be called directly (i.e. without the use of `DELEGATECALL`) if they do not modify the state (i.e. if they are `view` or `pure` functions), because libraries are assumed to be stateless. In particular, it is not possible to destroy a library.

---

**주석:** Until version 0.4.20, it was possible to destroy libraries by circumventing Solidity's type system. Starting from that version, libraries contain a *mechanism* that disallows state-modifying functions to be called directly (i.e. without `DELEGATECALL`).

---

Libraries can be seen as implicit base contracts of the contracts that use them. They will not be explicitly visible in the inheritance hierarchy, but calls to library functions look just like calls to functions of explicit base contracts (`L.f()` if `L` is the name of the library). Furthermore, `internal` functions of libraries are visible in all contracts, just as if the library were a base contract. Of course, calls to internal functions use the internal calling convention, which means that all internal types can be passed and types stored in memory will be passed by reference and not copied. To realize this in the EVM, code of internal library functions and all functions called from therein will at compile time be pulled into the calling contract, and a regular `JUMP` call will be used instead of a `DELEGATECALL`.

The following example illustrates how to use libraries (but manual method be sure to check out *using for* for a more advanced example to implement a set).

```
pragma solidity >=0.4.22 <0.6.0;

library Set {
  // We define a new struct datatype that will be used to
  // hold its data in the calling contract.
  struct Data { mapping(uint => bool) flags; }

  // Note that the first parameter is of type "storage
  // reference" and thus only its storage address and not
  // its contents is passed as part of the call.  This is a
  // special feature of library functions.  It is idiomatic
  // to call the first parameter `self`, if the function can
  // be seen as a method of that object.
  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
```

(continues on next page)

(이전 페이지에서 계속)

```
        public
        returns (bool)
    {
        if (!self.flags[value])
            return false; // not there
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}

contract C {
    Set.Data knownValues;

    function register(uint value) public {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        require(Set.insert(knownValues, value));
    }
    // In this contract, we can also directly access knownValues.flags, if we want.
}
```

Of course, you do not have to follow this way to use libraries: they can also be used without defining struct data types. Functions also work without any storage reference parameters, and they can have multiple storage reference parameters and in any position.

The calls to `Set.contains`, `Set.insert` and `Set.remove` are all compiled as calls (`DELEGATECALL`) to an external contract/library. If you use libraries, be aware that an actual external function call is performed. `msg.sender`, `msg.value` and `this` will retain their values in this call, though (prior to Homestead, because of the use of `CALLCODE`, `msg.sender` and `msg.value` changed, though).

The following example shows how to use types stored in memory and internal functions in libraries in order to implement custom types without the overhead of external function calls:

```
pragma solidity >=0.4.16 <0.6.0;

library BigInt {
    struct bigint {
        uint[] limbs;
    }

    function fromUint(uint x) internal pure returns (bigint memory r) {
        r.limbs = new uint[](1);
        r.limbs[0] = x;
    }

    function add(bigint memory _a, bigint memory _b) internal pure returns (bigint␣
↪memory r) {
        r.limbs = new uint[](max(_a.limbs.length, _b.limbs.length));
```

(continues on next page)

(이전 페이지에서 계속)

```
        uint carry = 0;
        for (uint i = 0; i < r.limbs.length; ++i) {
            uint a = limb(_a, i);
            uint b = limb(_b, i);
            r.limbs[i] = a + b + carry;
            if (a + b < a || (a + b == uint(-1) && carry > 0))
                carry = 1;
            else
                carry = 0;
        }
        if (carry > 0) {
            // too bad, we have to add a limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            uint i;
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

    function limb(bigint memory _a, uint _limb) internal pure returns (uint) {
        return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
    }

    function max(uint a, uint b) private pure returns (uint) {
        return a > b ? a : b;
    }
}

contract C {
    using BigInt for BigInt.bigint;

    function f() public pure {
        BigInt.bigint memory x = BigInt.fromUint(7);
        BigInt.bigint memory y = BigInt.fromUint(uint(-1));
        BigInt.bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}
```

As the compiler cannot know where the library will be deployed at, these addresses have to be filled into the final bytecode by a linker (see *명령행 컴파일러 사용하기* for how to use the commandline compiler for linking). If the addresses are not given as arguments to the compiler, the compiled hex code will contain placeholders of the form `__Set_______` (where `Set` is the name of the library). The address can be filled manually by replacing all those 40 symbols by the hex encoding of the address of the library contract.

**주석:** Manually linking libraries on the generated bytecode is discouraged, because it is restricted to 36 characters. You should ask the compiler to link the libraries at the time a contract is compiled by either using the `--libraries` option of `solc` or the `libraries` key if you use the standard-JSON interface to the compiler.

Restrictions for libraries in comparison to contracts:

- No state variables
- Cannot inherit nor be inherited

- Cannot receive Ether

(These might be lifted at a later point.)

### Call Protection For Libraries

As mentioned in the introduction, if a library's code is executed using a `CALL` instead of a `DELEGATECALL` or `CALLCODE`, it will revert unless a `view` or `pure` function is called.

The EVM does not provide a direct way for a contract to detect whether it was called using `CALL` or not, but a contract can use the `ADDRESS` opcode to find out "where" it is currently running. The generated code compares this address to the address used at construction time to determine the mode of calling.

More specifically, the runtime code of a library always starts with a push instruction, which is a zero of 20 bytes at compilation time. When the deploy code runs, this constant is replaced in memory by the current address and this modified code is stored in the contract. At runtime, this causes the deploy time address to be the first constant to be pushed onto the stack and the dispatcher code compares the current address against this constant for any non-view and non-pure function.

### Using For

The directive `using A for B;` can be used to attach library functions (from the library `A`) to any type (`B`). These functions will receive the object they are called on as their first parameter (like the `self` variable in Python).

The effect of `using A for *;` is that the functions from the library `A` are attached to *any* type.

In both situations, *all* functions in the library are attached, even those where the type of the first parameter does not match the type of the object. The type is checked at the point the function is called and function overload resolution is performed.

The `using A for B;` directive is active only within the current contract, including within all of its functions, and has no effect outside of the contract in which it is used. The directive may only be used inside a contract, not inside any of its functions.

By including a library, its data types including library functions are available without having to add further code.

Let us rewrite the set example from the *Libraries* in this way:

```
pragma solidity >=0.4.16 <0.6.0;

// This is the same code as before, just without comments
library Set {
  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
        return false; // already there
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
```

(continues on next page)

(이전 페이지에서 계속)

```
  {
      if (!self.flags[value])
          return false; // not there
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      public
      view
      returns (bool)
  {
      return self.flags[value];
  }
}

contract C {
    using Set for Set.Data; // this is the crucial change
    Set.Data knownValues;

    function register(uint value) public {
        // Here, all variables of type Set.Data have
        // corresponding member functions.
        // The following function call is identical to
        // `Set.insert(knownValues, value)`
        require(knownValues.insert(value));
    }
}
```

It is also possible to extend elementary types in that way:

```
pragma solidity >=0.4.16 <0.6.0;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return uint(-1);
    }
}

contract C {
    using Search for uint[];
    uint[] data;

    function append(uint value) public {
        data.push(value);
    }

    function replace(uint _old, uint _new) public {
        // This performs the library function call
        uint index = data.indexOf(_old);
        if (index == uint(-1))
```

(continues on next page)

(이전 페이지에서 계속)

```
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Note that all library calls are actual EVM function calls. This means that if you pass memory or value types, a copy will be performed, even of the `self` variable. The only situation where no copy will be performed is when storage reference variables are used.

### 8.4.7 Solidity Assembly

Solidity defines an assembly language that you can use without Solidity and also as "inline assembly" inside Solidity source code. This guide starts with describing how to use inline assembly, how it differs from standalone assembly, and specifies assembly itself.

#### Inline Assembly

You can interleave Solidity statements with inline assembly in a language close to the one of the virtual machine. This gives you more fine-grained control, especially when you are enhancing the language by writing libraries.

As the EVM is a stack machine, it is often hard to address the correct stack slot and provide arguments to opcodes at the correct point on the stack. Solidity's inline assembly helps you do this, and with other issues that arise when writing manual assembly.

Inline assembly has the following features:

- functional-style opcodes: `mul(1, add(2, 3))`
- assembly-local variables: `let x := add(2, 3) let y := mload(0x40) x := add(x, y)`
- access to external variables: `function f(uint x) public { assembly { x := sub(x, 1) } }`
- loops: `for { let i := 0 } lt(i, x) { i := add(i, 1) } { y := mul(2, y) }`
- if statements: `if slt(x, 0) { x := sub(0, x) }`
- switch statements: `switch x case 0 { y := mul(x, 2) } default { y := 0 }`
- function calls: `function f(x) -> y { switch x case 0 { y := 1 } default { y := mul(x, f(sub(x, 1))) } }`

**경고:** Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.

#### Syntax

Assembly parses comments, literals and identifiers in the same way as Solidity, so you can use the usual `//` and `/* */` comments. Inline assembly is marked by `assembly { ... }` and inside these curly braces, you can use the following (see the later sections for more details):

- literals, i.e. `0x123`, `42` or `"abc"` (strings up to 32 characters)

- opcodes in functional style, e.g. `add(1, mlod(0))`
- variable declarations, e.g. `let x := 7`, `let x := add(y, 3)` or `let x` (initial value of empty (0) is assigned)
- identifiers (assembly-local variables and externals if used as inline assembly), e.g. `add(3, x)`, `sstore(x_slot, 2)`
- assignments, e.g. `x := add(y, 3)`
- blocks where local variables are scoped inside, e.g. `{ let x := 3 { let y := add(x, 1) } }`

The following features are only available for standalone assembly:

- direct stack control via `dup1`, `swap1`, ...
- direct stack assignments (in "instruction style"), e.g. `3 =: x`
- labels, e.g. `name:`
- jump opcodes

**주석:** Standalone assembly is supported for backwards compatibility but is not documented here anymore.

At the end of the `assembly { ... }` block, the stack must be balanced, unless you require it otherwise. If it is not balanced, the compiler generates a warning.

### Example

The following example provides library code to access the code of another contract and load it into a `bytes` variable. This is not possible with "plain Solidity" and the idea is that assembly libraries will be used to enhance the Solidity language.

```
pragma solidity >=0.4.0 <0.6.0;

library GetCode {
    function at(address _addr) public view returns (bytes memory o_code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(_addr)
            // allocate output byte array - this could also be done without assembly
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // store length in memory
            mstore(o_code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)
        }
    }
}
```

Inline assembly is also beneficial in cases where the optimizer fails to produce efficient code, for example:

```
pragma solidity >=0.4.16 <0.6.0;

library VectorSum {
```

(continues on next page)

(이전 페이지에서 계속)

```
    // This function is less efficient because the optimizer currently fails to
    // remove the bounds checks in array access.
    function sumSolidity(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i)
            o_sum += _data[i];
    }

    // We know that we only access the array in bounds, so we can avoid the check.
    // 0x20 needs to be added to an array because the first slot contains the
    // array length.
    function sumAsm(uint[] memory _data) public pure returns (uint o_sum) {
        for (uint i = 0; i < _data.length; ++i) {
            assembly {
                o_sum := add(o_sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
            }
        }
    }

    // Same as above, but accomplish the entire code within inline assembly.
    function sumPureAsm(uint[] memory _data) public pure returns (uint o_sum) {
        assembly {
            // Load the length (first 32 bytes)
            let len := mload(_data)

            // Skip over the length field.
            //
            // Keep temporary variable so it can be incremented in place.
            //
            // NOTE: incrementing _data would result in an unusable
            //       _data variable after this assembly block
            let data := add(_data, 0x20)

            // Iterate until the bound is not met.
            for
                { let end := add(data, mul(len, 0x20)) }
                lt(data, end)
                { data := add(data, 0x20) }
            {
                o_sum := add(o_sum, mload(data))
            }
        }
    }
}
```

## Opcodes

This document does not want to be a full description of the Ethereum virtual machine, but the following list can be used as a reference of its opcodes.

If an opcode takes arguments (always from the top of the stack), they are given in parentheses. Note that the order of arguments can be seen to be reversed in non-functional style (explained below). Opcodes marked with `-` do not push an item onto the stack, those marked with `*` are special and all others push exactly one item onto the stack. Opcodes marked with `F`, `H`, `B` or `C` are present since Frontier, Homestead, Byzantium or Constantinople, respectively. Constantinople is still in planning and all instructions marked as such will result in an invalid instruction exception.

In the following, `mem[a...b)` signifies the bytes of memory starting at position `a` up to but not including position `b`

and `storage[p]` signifies the storage contents at position `p`.

The opcodes `pushi` and `jumpdest` cannot be used directly.

In the grammar, opcodes are represented as pre-defined identifiers.

| Instruction | | | Explanation |
|---|---|---|---|
| stop | - | F | stop execution, identical to return(0,0) |
| add(x, y) | | F | x + y |
| sub(x, y) | | F | x - y |
| mul(x, y) | | F | x * y |
| div(x, y) | | F | x / y |
| sdiv(x, y) | | F | x / y, for signed numbers in two's complement |
| mod(x, y) | | F | x % y |
| smod(x, y) | | F | x % y, for signed numbers in two's complement |
| exp(x, y) | | F | x to the power of y |
| not(x) | | F | ~x, every bit of x is negated |
| lt(x, y) | | F | 1 if x < y, 0 otherwise |
| gt(x, y) | | F | 1 if x > y, 0 otherwise |
| slt(x, y) | | F | 1 if x < y, 0 otherwise, for signed numbers in two's complement |
| sgt(x, y) | | F | 1 if x > y, 0 otherwise, for signed numbers in two's complement |
| eq(x, y) | | F | 1 if x == y, 0 otherwise |
| iszero(x) | | F | 1 if x == 0, 0 otherwise |
| and(x, y) | | F | bitwise and of x and y |
| or(x, y) | | F | bitwise or of x and y |
| xor(x, y) | | F | bitwise xor of x and y |
| byte(n, x) | | F | nth byte of x, where the most significant byte is the 0th byte |
| shl(x, y) | | C | logical shift left y by x bits |
| shr(x, y) | | C | logical shift right y by x bits |
| sar(x, y) | | C | arithmetic shift right y by x bits |
| addmod(x, y, m) | | F | (x + y) % m with arbitrary precision arithmetic |
| mulmod(x, y, m) | | F | (x * y) % m with arbitrary precision arithmetic |
| signextend(i, x) | | F | sign extend from (i*8+7)th bit counting from least significant |
| keccak256(p, n) | | F | keccak(mem[p...(p+n))) |
| jump(label) | - | F | jump to label / code position |
| jumpi(label, cond) | - | F | jump to label if cond is nonzero |
| pc | | F | current position in code |
| pop(x) | - | F | remove the element pushed by x |
| dup1 ... dup16 | | F | copy nth stack slot to the top (counting from top) |
| swap1 ... swap16 | * | F | swap topmost and nth stack slot below it |
| mload(p) | | F | mem[p...(p+32)) |
| mstore(p, v) | - | F | mem[p...(p+32)) := v |
| mstore8(p, v) | - | F | mem[p] := v & 0xff (only modifies a single byte) |
| sload(p) | | F | storage[p] |
| sstore(p, v) | - | F | storage[p] := v |
| msize | | F | size of memory, i.e. largest accessed memory index |
| gas | | F | gas still available to execution |
| address | | F | address of the current contract / execution context |
| balance(a) | | F | wei balance at address a |
| caller | | F | call sender (excluding `delegatecall`) |
| callvalue | | F | wei sent together with the current call |

| Instruction | | | Explanation |
|---|---|---|---|
| calldataload(p) | | F | call data starting from position p (32 bytes) |
| calldatasize | | F | size of call data in bytes |
| calldatacopy(t, f, s) | - | F | copy s bytes from calldata at position f to mem at position t |
| codesize | | F | size of the code of the current contract / execution context |
| codecopy(t, f, s) | - | F | copy s bytes from code at position f to mem at position t |
| extcodesize(a) | | F | size of the code at address a |
| extcodecopy(a, t, f, s) | - | F | like codecopy(t, f, s) but take code at address a |
| returndatasize | | B | size of the last returndata |
| returndatacopy(t, f, s) | - | B | copy s bytes from returndata at position f to mem at position t |
| extcodehash(a) | | C | code hash of address a |
| create(v, p, n) | | F | create new contract with code mem[p...(p+n)) and send v wei and return the new |
| create2(v, p, n, s) | | C | create new contract with code mem[p...(p+n)) at address keccak256(0xff . this . s |
| call(g, a, v, in, insize, out, outsize) | | F | call contract at address a with input mem[in...(in+insize)) providing g gas and v w |
| callcode(g, a, v, in, insize, out, outsize) | | F | identical to `call` but only use the code from a and stay in the context of the curr |
| delegatecall(g, a, in, insize, out, outsize) | | H | identical to `callcode` but also keep `caller` and `callvalue` |
| staticcall(g, a, in, insize, out, outsize) | | B | identical to `call(g, a, 0, in, insize, out, outsize)` but do not |
| return(p, s) | - | F | end execution, return data mem[p...(p+s)) |
| revert(p, s) | - | B | end execution, revert state changes, return data mem[p...(p+s)) |
| selfdestruct(a) | - | F | end execution, destroy current contract and send funds to a |
| invalid | - | F | end execution with invalid instruction |
| log0(p, s) | - | F | log without topics and data mem[p...(p+s)) |
| log1(p, s, t1) | - | F | log with topic t1 and data mem[p...(p+s)) |
| log2(p, s, t1, t2) | - | F | log with topics t1, t2 and data mem[p...(p+s)) |
| log3(p, s, t1, t2, t3) | - | F | log with topics t1, t2, t3 and data mem[p...(p+s)) |
| log4(p, s, t1, t2, t3, t4) | - | F | log with topics t1, t2, t3, t4 and data mem[p...(p+s)) |
| origin | | F | transaction sender |
| gasprice | | F | gas price of the transaction |
| blockhash(b) | | F | hash of block nr b - only for last 256 blocks excluding current |
| coinbase | | F | current mining beneficiary |
| timestamp | | F | timestamp of the current block in seconds since the epoch |
| number | | F | current block number |
| difficulty | | F | difficulty of the current block |
| gaslimit | | F | block gas limit of the current block |

### Literals

You can use integer constants by typing them in decimal or hexadecimal notation and an appropriate `PUSHi` instruction will automatically be generated. The following creates code to add 2 and 3 resulting in 5 and then computes the bitwise and with the string "abc". The final value is assigned to a local variable called `x`. Strings are stored left-aligned and cannot be longer than 32 bytes.

```
assembly { let x := and("abc", add(3, 2)) }
```

### Functional Style

For a sequence of opcodes, it is often hard to see what the actual arguments for certain opcodes are. In the following example, `3` is added to the contents in memory at position `0x80`.

```
3 0x80 mload add 0x80 mstore
```

Solidity inline assembly has a "functional style" notation where the same code would be written as follows:

```
mstore(0x80, add(mload(0x80), 3))
```

If you read the code from right to left, you end up with exactly the same sequence of constants and opcodes, but it is much clearer where the values end up.

If you care about the exact stack layout, just note that the syntactically first argument for a function or opcode will be put at the top of the stack.

### Access to External Variables, Functions and Libraries

You can access Solidity variables and other identifiers by using their name. For variables stored in the memory data location, this pushes the address, and not the value onto the stack. Variables stored in the storage data location are different, as they might not occupy a full storage slot, so their "address" is composed of a slot and a byte-offset inside that slot. To retrieve the slot pointed to by the variable `x`, you use `x_slot`, and to retrieve the byte-offset you use `x_offset`.

Local Solidity variables are available for assignments, for example:

```
pragma solidity >=0.4.11 <0.6.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            r := mul(x, sload(b_slot)) // ignore the offset, we know it is zero
        }
    }
}
```

**경고:** If you access variables of a type that spans less than 256 bits (for example `uint64`, `address`, `bytes16` or `byte`), you cannot make any assumptions about bits not part of the encoding of the type. Especially, do not assume them to be zero. To be safe, always clear the data properly before you use it in a context where this is important: `uint32 x = f(); assembly { x := and(x, 0xffffffff) /* now use x */ }` To clean signed types, you can use the `signextend` opcode.

### Labels

Support for labels has been removed in version 0.5.0 of Solidity. Please use functions, loops, if or switch statements instead.

### Declaring Assembly-Local Variables

You can use the `let` keyword to declare variables that are only visible in inline assembly and actually only in the current `{...}`-block. What happens is that the `let` instruction will create a new stack slot that is reserved for the variable and automatically removed again when the end of the block is reached. You need to provide an initial value for the variable which can be just `0`, but it can also be a complex functional-style expression.

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    function f(uint x) public view returns (uint b) {
        assembly {
            let v := add(x, 1)
            mstore(0x80, v)
            {
                let y := add(sload(v), 1)
                b := y
            } // y is "deallocated" here
            b := add(b, v)
        } // v is "deallocated" here
    }
}
```

### Assignments

Assignments are possible to assembly-local variables and to function-local variables. Take care that when you assign to variables that point to memory or storage, you will only change the pointer and not the data.

Variables can only be assigned expressions that result in exactly one value. If you want to assign the values returned from a function that has multiple return parameters, you have to provide multiple variables.

```
{
    let v := 0
    let g := add(v, 2)
    function f() -> a, b { }
    let c, d := f()
}
```

### If

The if statement can be used for conditionally executing code. There is no "else" part, consider using "switch" (see below) if you need multiple alternatives.

```
{
    if eq(value, 0) { revert(0, 0) }
}
```

The curly braces for the body are required.

### Switch

You can use a switch statement as a very basic version of "if/else". It takes the value of an expression and compares it to several constants. The branch corresponding to the matching constant is taken. Contrary to the error-prone behaviour of some programming languages, control flow does not continue from one case to the next. There can be a fallback or default case called `default`.

```
{
    let x := 0
    switch calldataload(4)
```

(continues on next page)

(이전 페이지에서 계속)

```
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

The list of cases does not require curly braces, but the body of a case does require them.

### Loops

Assembly supports a simple for-style loop. For-style loops have a header containing an initializing part, a condition and a post-iteration part. The condition has to be a functional-style expression, while the other two are blocks. If the initializing part declares any variables, the scope of these variables is extended into the body (including the condition and the post-iteration part).

The following example computes the sum of an area in memory.

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

For loops can also be written so that they behave like while loops: Simply leave the initialization and post-iteration parts empty.

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {      // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

### Functions

Assembly allows the definition of low-level functions. These take their arguments (and a return PC) from the stack and also put the results onto the stack. Calling a function looks the same way as executing a functional-style opcode.

Functions can be defined anywhere and are visible in the block they are declared in. Inside a function, you cannot access local variables defined outside of that function. There is no explicit `return` statement.

If you call a function that returns multiple values, you have to assign them to a tuple using `a, b := f(x)` or `let a, b := f(x)`.

The following example implements the power function by square-and-multiply.

```
{
    function power(base, exponent) -> result {
```

(continues on next page)

(이전 페이지에서 계속)

```
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

### Things to Avoid

Inline assembly might have a quite high-level look, but it actually is extremely low-level. Function calls, loops, ifs and switches are converted by simple rewriting rules and after that, the only thing the assembler does for you is re-arranging functional-style opcodes, counting stack height for variable access and removing stack slots for assembly-local variables when the end of their block is reached.

### Conventions in Solidity

In contrast to EVM assembly, Solidity knows types which are narrower than 256 bits, e.g. `uint24`. In order to make them more efficient, most arithmetic operations just treat them as 256-bit numbers and the higher-order bits are only cleaned at the point where it is necessary, i.e. just shortly before they are written to memory or before comparisons are performed. This means that if you access such a variable from within inline assembly, you might have to manually clean the higher order bits first.

Solidity manages memory in a very simple way: There is a "free memory pointer" at position `0x40` in memory. If you want to allocate memory, just use the memory starting from where this pointer points at and update it accordingly. There is no guarantee that the memory has not been used before and thus you cannot assume that its contents are zero bytes. There is no built-in mechanism to release or free allocated memory. Here is an assembly snippet that can be used for allocating memory:

```
function allocate(length) -> pos {
  pos := mload(0x40)
  mstore(0x40, add(pos, length))
}
```

The first 64 bytes of memory can be used as "scratch space" for short-term allocation. The 32 bytes after the free memory pointer (i.e. starting at `0x60`) is meant to be zero permanently and is used as the initial value for empty dynamic memory arrays. This means that the allocatable memory starts at `0x80`, which is the initial value of the free memory pointer.

Elements in memory arrays in Solidity always occupy multiples of 32 bytes (yes, this is even true for `byte[]`, but not for `bytes` and `string`). Multi-dimensional memory arrays are pointers to memory arrays. The length of a dynamic array is stored at the first slot of the array and followed by the array elements.

**경고:** Statically-sized memory arrays do not have a length field, but it might be added later to allow better convertibility between statically- and dynamically-sized arrays, so please do not rely on that.

### Standalone Assembly

The assembly language described as inline assembly above can also be used standalone and in fact, the plan is to use it as an intermediate language for the Solidity compiler. In this form, it tries to achieve several goals:

1. Programs written in it should be readable, even if the code is generated by a compiler from Solidity.
2. The translation from assembly to bytecode should contain as few "surprises" as possible.
3. Control flow should be easy to detect to help in formal verification and optimization.

In order to achieve the first and last goal, assembly provides high-level constructs like `for` loops, `if` and `switch` statements and function calls. It should be possible to write assembly programs that do not make use of explicit `SWAP`, `DUP`, `JUMP` and `JUMPI` statements, because the first two obfuscate the data flow and the last two obfuscate control flow. Furthermore, functional statements of the form `mul(add(x, y), 7)` are preferred over pure opcode statements like `7 y x add mul` because in the first form, it is much easier to see which operand is used for which opcode.

The second goal is achieved by compiling the higher level constructs to bytecode in a very regular way. The only non-local operation performed by the assembler is name lookup of user-defined identifiers (functions, variables, ...), which follow very simple and regular scoping rules and cleanup of local variables from the stack.

Scoping: An identifier that is declared (label, variable, function, assembly) is only visible in the block where it was declared (including nested blocks inside the current block). It is not legal to access local variables across function borders, even if they would be in scope. Shadowing is not allowed. Local variables cannot be accessed before they were declared, but functions and assemblies can. Assemblies are special blocks that are used for e.g. returning runtime code or creating contracts. No identifier from an outer assembly is visible in a sub-assembly.

If control flow passes over the end of a block, pop instructions are inserted that match the number of local variables declared in that block. Whenever a local variable is referenced, the code generator needs to know its current relative position in the stack and thus it needs to keep track of the current so-called stack height. Since all local variables are removed at the end of a block, the stack height before and after the block should be the same. If this is not the case, compilation fails.

Using `switch`, `for` and functions, it should be possible to write complex code without using `jump` or `jumpi` manually. This makes it much easier to analyze the control flow, which allows for improved formal verification and optimization.

Furthermore, if manual jumps are allowed, computing the stack height is rather complicated. The position of all local variables on the stack needs to be known, otherwise neither references to local variables nor removing local variables automatically from the stack at the end of a block will work properly.

Example:

We will follow an example compilation from Solidity to assembly. We consider the runtime bytecode of the following Solidity program:

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
  function f(uint x) public pure returns (uint y) {
    y = 1;
    for (uint i = 0; i < x; i++)
      y = 2 * y;
  }
}
```

The following assembly will be generated:

```
{
  mstore(0x40, 0x80) // store the "free memory pointer"
  // function dispatcher
  switch div(calldataload(0), exp(2, 226))
  case 0xb3de648b {
    let r := f(calldataload(4))
    let ret := $allocate(0x20)
    mstore(ret, r)
    return(ret, 0x20)
  }
  default { revert(0, 0) }
  // memory allocator
  function $allocate(size) -> pos {
    pos := mload(0x40)
    mstore(0x40, add(pos, size))
  }
  // the contract function
  function f(x) -> y {
    y := 1
    for { let i := 0 } lt(i, x) { i := add(i, 1) } {
      y := mul(2, y)
    }
  }
}
```

### Assembly Grammar

The tasks of the parser are the following:

- Turn the byte stream into a token stream, discarding C++-style comments (a special comment exists for source references, but we will not explain it here).
- Turn the token stream into an AST according to the grammar below
- Register identifiers with the block they are defined in (annotation to the AST node) and note from which point on, variables can be accessed.

The assembly lexer follows the one defined by Solidity itself.

Whitespace is used to delimit tokens and it consists of the characters Space, Tab and Linefeed. Comments are regular JavaScript/C++ comments and are interpreted in the same way as Whitespace.

Grammar:

```
AssemblyBlock = '{' AssemblyItem* '}'
AssemblyItem =
    Identifier |
    AssemblyBlock |
    AssemblyExpression |
    AssemblyLocalDefinition |
    AssemblyAssignment |
    AssemblyStackAssignment |
    LabelDefinition |
    AssemblyIf |
    AssemblySwitch |
    AssemblyFunctionDefinition |
    AssemblyFor |
```

(continues on next page)

(이전 페이지에서 계속)

```
    'break' |
    'continue' |
    SubAssembly
AssemblyExpression = AssemblyCall | Identifier | AssemblyLiteral
AssemblyLiteral = NumberLiteral | StringLiteral | HexLiteral
Identifier = [a-zA-Z_$] [a-zA-Z_0-9]*
AssemblyCall = Identifier '(' ( AssemblyExpression ( ',' AssemblyExpression )* )? ')'
AssemblyLocalDefinition = 'let' IdentifierOrList ( ':=' AssemblyExpression )?
AssemblyAssignment = IdentifierOrList ':=' AssemblyExpression
IdentifierOrList = Identifier | '(' IdentifierList ')'
IdentifierList = Identifier ( ',' Identifier)*
AssemblyStackAssignment = '=:' Identifier
LabelDefinition = Identifier ':'
AssemblyIf = 'if' AssemblyExpression AssemblyBlock
AssemblySwitch = 'switch' AssemblyExpression AssemblyCase*
    ( 'default' AssemblyBlock )?
AssemblyCase = 'case' AssemblyExpression AssemblyBlock
AssemblyFunctionDefinition = 'function' Identifier '(' IdentifierList? ')'
    ( '->' '(' IdentifierList ')' )? AssemblyBlock
AssemblyFor = 'for' ( AssemblyBlock | AssemblyExpression )
    AssemblyExpression ( AssemblyBlock | AssemblyExpression ) AssemblyBlock
SubAssembly = 'assembly' Identifier AssemblyBlock
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

### 8.4.8 Miscellaneous

#### Layout of State Variables in Storage

Statically-sized variables (everything except mapping and dynamically-sized array types) are laid out contiguously in storage starting from position `0`. Multiple items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

- The first item in a storage slot is stored lower-order aligned.
- Elementary types use only that many bytes that are necessary to store them.
- If an elementary type does not fit the remaining part of a storage slot, it is moved to the next storage slot.
- Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

**경고:** When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

It is only beneficial to use reduced-size arguments if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. When dealing with function arguments or memory values, there is no inherent benefit because the compiler does not pack these values.

Finally, in order to allow the EVM to optimize for this, ensure that you try to order your storage variables and `struct` members such that they can be packed tightly. For example, declaring your storage variables in the order of `uint128, uint128, uint256` instead of `uint128, uint256, uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

The elements of structs and arrays are stored after each other, just as if they were given explicitly.

### Mappings and Dynamic Arrays

Due to their unpredictable size, mapping and dynamically-sized array types use a Keccak-256 hash computation to find the starting position of the value or the array data. These starting positions are always full stack slots.

The mapping or the dynamic array itself occupies a slot in storage at some position `p` according to the above rule (or by recursively applying this rule for mappings of mappings or arrays of arrays). For dynamic arrays, this slot stores the number of elements in the array (byte arrays and strings are an exception, see *below*). For mappings, the slot is unused (but it is needed so that two equal mappings after each other will use a different hash distribution). Array data is located at `keccak256(p)` and the value corresponding to a mapping key `k` is located at `keccak256(k . p)` where `.` is concatenation. If the value is again a non-elementary type, the positions are found by adding an offset of `keccak256(k . p)`.

So for the following contract snippet:

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
  struct s { uint a; uint b; }
  uint x;
  mapping(uint => mapping(uint => s)) data;
}
```

The position of `data[4][9].b` is at `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`.

#### `bytes` and `string`

`bytes` and `string` are encoded identically. For short byte arrays, they store their data in the same slot where the length is also stored. In particular: if the data is at most `31` bytes long, it is stored in the higher-order bytes (left aligned) and the lowest-order byte stores `length * 2`. For byte arrays that store data which is `32` or more bytes long, the main slot stores `length * 2 + 1` and the data is stored as usual in `keccak256(slot)`. This means that you can distinguish a short array from a long array by checking if the lowest bit is set: short (not set) and long (set).

**주석:** Handling invalidly encoded slots is currently not supported but may be added in the future.

### Layout in Memory

Solidity reserves four 32-byte slots, with specific byte ranges (inclusive of endpoints) being used as follows:

- `0x00` - `0x3f` (64 bytes): scratch space for hashing methods
- `0x40` - `0x5f` (32 bytes): currently allocated memory size (aka. free memory pointer)

- `0x60` - `0x7f` (32 bytes): zero slot

Scratch space can be used between statements (i.e. within inline assembly). The zero slot is used as initial value for dynamic memory arrays and should never be written to (the free memory pointer points to `0x80` initially).

Solidity always places new objects at the free memory pointer and memory is never freed (this might change in the future).

**경고:** There are some operations in Solidity that need a temporary memory area larger than 64 bytes and therefore will not fit into the scratch space. They will be placed where the free memory points to, but given their short lifetime, the pointer is not updated. The memory may or may not be zeroed out. Because of this, one shouldn't expect the free memory to point to zeroed out memory.

While it may seem like a good idea to use `msize` to arrive at a definitely zeroed out memory area, using such a pointer non-temporarily without updating the free memory pointer can have adverse results.

### Layout of Call Data

The input data for a function call is assumed to be in the format defined by the *ABI specification*. Among others, the ABI specification requires arguments to be padded to multiples of 32 bytes. The internal function calls use a different convention.

Arguments for the constructor of a contract are directly appended at the end of the contract's code, also in ABI encoding. The constructor will access them through a hard-coded offset, and not by using the `codesize` opcode, since this of course changes when appending data to the code.

### Internals - Cleaning Up Variables

When a value is shorter than 256-bit, in some cases the remaining bits must be cleaned. The Solidity compiler is designed to clean such remaining bits before any operations that might be adversely affected by the potential garbage in the remaining bits. For example, before writing a value to the memory, the remaining bits need to be cleared because the memory contents can be used for computing hashes or sent as the data of a message call. Similarly, before storing a value in the storage, the remaining bits need to be cleaned because otherwise the garbled value can be observed.

On the other hand, we do not clean the bits if the immediately following operation is not affected. For instance, since any non-zero value is considered `true` by `JUMPI` instruction, we do not clean the boolean values before they are used as the condition for `JUMPI`.

In addition to the design principle above, the Solidity compiler cleans input data when it is loaded onto the stack.

Different types have different rules for cleaning up invalid values:

| Type | Valid Values | Invalid Values Mean |
|---|---|---|
| enum of n members | 0 until n - 1 | exception |
| bool | 0 or 1 | 1 |
| signed integers | sign-extended word | currently silently wraps; in the future exceptions will be thrown |
| unsigned integers | higher bits zeroed | currently silently wraps; in the future exceptions will be thrown |

### Internals - The Optimizer

The Solidity optimizer operates on assembly, so it can be and also is used by other languages. It splits the sequence of instructions into basic blocks at `JUMPs` and `JUMPDESTs`. Inside these blocks, the instructions are analysed and every modification to the stack, to memory or storage is recorded as an expression which consists of an instruction

and a list of arguments which are essentially pointers to other expressions. The main idea is now to find expressions that are always equal (on every input) and combine them into an expression class. The optimizer first tries to find each new expression in a list of already known expressions. If this does not work, the expression is simplified according to rules like `constant + constant = sum_of_constants` or `X * 1 = X`. Since this is done recursively, we can also apply the latter rule if the second factor is a more complex expression where we know that it will always evaluate to one. Modifications to storage and memory locations have to erase knowledge about storage and memory locations which are not known to be different: If we first write to location x and then to location y and both are input variables, the second could overwrite the first, so we actually do not know what is stored at x after we wrote to y. On the other hand, if a simplification of the expression x - y evaluates to a non-zero constant, we know that we can keep our knowledge about what is stored at x.

At the end of this process, we know which expressions have to be on the stack in the end and have a list of modifications to memory and storage. This information is stored together with the basic blocks and is used to link them. Furthermore, knowledge about the stack, storage and memory configuration is forwarded to the next block(s). If we know the targets of all `JUMP` and `JUMPI` instructions, we can build a complete control flow graph of the program. If there is only one target we do not know (this can happen as in principle, jump targets can be computed from inputs), we have to erase all knowledge about the input state of a block as it can be the target of the unknown `JUMP`. If a `JUMPI` is found whose condition evaluates to a constant, it is transformed to an unconditional jump.

As the last step, the code in each block is completely re-generated. A dependency graph is created from the expressions on the stack at the end of the block and every operation that is not part of this graph is essentially dropped. Now code is generated that applies the modifications to memory and storage in the order they were made in the original code (dropping modifications which were found not to be needed) and finally, generates all values that are required to be on the stack in the correct place.

These steps are applied to each basic block and the newly generated code is used as replacement if it is smaller. If a basic block is split at a `JUMPI` and during the analysis, the condition evaluates to a constant, the `JUMPI` is replaced depending on the value of the constant, and thus code like

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2)
  return 2;
else
  return 1;
```

is simplified to code which can also be compiled from

```
data[7] = 9;
return 1;
```

even though the instructions contained a jump in the beginning.

### Source Mappings

As part of the AST output, the compiler provides the range of the source code that is represented by the respective node in the AST. This can be used for various purposes ranging from static analysis tools that report errors based on the AST and debugging tools that highlight local variables and their uses.

Furthermore, the compiler can also generate a mapping from the bytecode to the range in the source code that generated the instruction. This is again important for static analysis tools that operate on bytecode level and for displaying the current position in the source code inside a debugger or for breakpoint handling.

Both kinds of source mappings use integer identifiers to refer to source files. These are regular array indices into a list of source files usually called `"sourceList"`, which is part of the combined-json and the output of the json / npm compiler.

주석: In the case of instructions that are not associated with any particular source file, the source mapping assigns an integer identifier of `-1`. This may happen for bytecode sections stemming from compiler-generated inline assembly statements.

The source mappings inside the AST use the following notation:

`s:l:f`

Where `s` is the byte-offset to the start of the range in the source file, `l` is the length of the source range in bytes and `f` is the source index mentioned above.

The encoding in the source mapping for the bytecode is more complicated: It is a list of `s:l:f:j` separated by `;`. Each of these elements corresponds to an instruction, i.e. you cannot use the byte offset but have to use the instruction offset (push instructions are longer than a single byte). The fields `s`, `l` and `f` are as above and `j` can be either `i`, `o` or `-` signifying whether a jump instruction goes into a function, returns from a function or is a regular jump as part of e.g. a loop.

In order to compress these source mappings especially for bytecode, the following rules are used:

- If a field is empty, the value of the preceding element is used.
- If a `:` is missing, all following fields are considered empty.

This means the following source mappings represent the same information:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2:1:2;;`

### Tips and Tricks

- Use `delete` on arrays to delete all its elements.
- Use shorter types for struct elements and sort them such that short types are grouped together. This can lower the gas costs as multiple `SSTORE` operations might be combined into a single (`SSTORE` costs 5000 or 20000 gas, so this is what you want to optimise). Use the gas price estimator (with optimiser enabled) to check!
- Make your state variables public - the compiler will create *getters* for you automatically.
- If you end up checking conditions on input or state a lot at the beginning of your functions, try using *Function Modifiers*.
- Initialize storage structs with a single assignment: `x = MyStruct({a: 1, b: 2});`

주석: If the storage struct has tightly packed properties, initialize it with separate assignments: `x.a = 1; x.b = 2;`. In this way it will be easier for the optimizer to update storage in one go, thus making assignment cheaper.

### Cheatsheet

### Order of Precedence of Operators

The following is the order of precedence for operators, listed in order of evaluation.

| Precedence | Description | Operator |
|---|---|---|
| *1* | Postfix increment and decrement | `++`, `--` |
| | New expression | `new <typename>` |
| | Array subscripting | `<array>[<index>]` |
| | Member access | `<object>.<member>` |
| | Function-like call | `<func>(<args...>)` |
| | Parentheses | `(<statement>)` |
| *2* | Prefix increment and decrement | `++`, `--` |
| | Unary minus | `-` |
| | Unary operations | `delete` |
| | Logical NOT | `!` |
| | Bitwise NOT | `~` |
| *3* | Exponentiation | `**` |
| *4* | Multiplication, division and modulo | `*`, `/`, `%` |
| *5* | Addition and subtraction | `+`, `-` |
| *6* | Bitwise shift operators | `<<`, `>>` |
| *7* | Bitwise AND | `&` |
| *8* | Bitwise XOR | `^` |
| *9* | Bitwise OR | `\|` |
| *10* | Inequality operators | `<`, `>`, `<=`, `>=` |
| *11* | Equality operators | `==`, `!=` |
| *12* | Logical AND | `&&` |
| *13* | Logical OR | `\|\|` |
| *14* | Ternary operator | `<conditional> ? <if-true> : <if-false>` |
| *15* | Assignment operators | `=`, `\|=`, `^=`, `&=`, `<<=`, `>>=`, `+=`, `-=`, `*=`, `/=`, `%=` |
| *16* | Comma operator | `,` |

### Global Variables

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: *ABI*-decodes the provided data. The types are given in parentheses as second argument. Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: *ABI*-encodes the given arguments
- `abi.encodePacked(...) returns (bytes memory)`: Performs *packed encoding* of the given arguments
- **`abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`: *ABI*-encodes the given arg** starting from the second and prepends the given four-byte selector
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)`: Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature)), ...)` `
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp
- `gasleft() returns (uint256)`: remaining gas

- `msg.data` (`bytes`): complete calldata
- `msg.sender` (`address payable`): sender of the message (current call)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address payable`): sender of the transaction (full call chain)
- `assert(bool condition)`: abort execution and revert state changes if condition is `false` (use for internal error)
- `require(bool condition)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component)
- `require(bool condition, string memory message)`: abort execution and revert state changes if condition is `false` (use for malformed input or error in external component). Also provide error message.
- `revert()`: abort execution and revert state changes
- `revert(string memory message)`: abort execution and revert state changes providing an explanatory string
- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block - only works for 256 most recent blocks
- `keccak256(bytes memory) returns (bytes32)`: compute the Keccak-256 hash of the input
- `sha256(bytes memory) returns (bytes32)`: compute the SHA-256 hash of the input
- `ripemd160(bytes memory) returns (bytes20)`: compute the RIPEMD-160 hash of the input
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: recover address associated with the public key from elliptic curve signature, return zero on error
- `addmod(uint x, uint y, uint k) returns (uint)`: compute `(x + y) % k` where the addition is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `mulmod(uint x, uint y, uint k) returns (uint)`: compute `(x * y) % k` where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`. Assert that `k != 0` starting from version 0.5.0.
- `this` (current contract's type): the current contract, explicitly convertible to `address` or `address payable`
- `super`: the contract one level higher in the inheritance hierarchy
- `selfdestruct(address payable recipient)`: destroy the current contract, sending its funds to the given address
- `<address>.balance` (`uint256`): balance of the *Address* in Wei
- `<address payable>.send(uint256 amount) returns (bool)`: send given amount of Wei to *Address*, returns `false` on failure
- `<address payable>.transfer(uint256 amount)`: send given amount of Wei to *Address*, throws on failure

**주석:** Do not rely on `block.timestamp`, `now` and `blockhash` as a source of randomness, unless you know what you are doing.

Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.

The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

---

**주석:** The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

---

**주석:** In version 0.5.0, the following aliases were removed: `suicide` as alias for `selfdestruct`, `msg.gas` as alias for `gasleft`, `block.blockhash` as alias for `blockhash` and `sha3` as alias for `keccak256`.

---

### Function Visibility Specifiers

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`: visible externally and internally (creates a *getter function* for storage/state variables)
- `private`: only visible in the current contract
- `external`: only visible externally (only for functions) - i.e. can only be message-called (via `this.func`)
- `internal`: only visible internally

### Modifiers

- `pure` for functions: Disallows modification or access of state.
- `view` for functions: Disallows modification of state.
- `payable` for functions: Allows them to receive Ether together with a call.
- `constant` for state variables: Disallows assignment (except initialisation), does not occupy storage slot.
- `anonymous` for events: Does not store event signature as topic.
- `indexed` for event parameters: Stores the parameter as topic.

### Reserved Keywords

These keywords are reserved in Solidity. They might become part of the syntax in the future:

`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`.

### Language Grammar

```
SourceUnit = (PragmaDirective | ImportDirective | ContractDefinition)*

// Pragma actually parses anything up to the trailing ';' to be fully forward-
↪compatible.
PragmaDirective = 'pragma' Identifier ([^;]+) ';'

ImportDirective = 'import' StringLiteral ('as' Identifier)? ';'
        | 'import' ('*' | Identifier) ('as' Identifier)? 'from' StringLiteral ';'
        | 'import' '{' Identifier ('as' Identifier)? ( ',' Identifier ('as'␣
↪Identifier)? )* '}' 'from' StringLiteral ';'

ContractDefinition = ( 'contract' | 'library' | 'interface' ) Identifier
                     ( 'is' InheritanceSpecifier (',' InheritanceSpecifier )* )?
                     '{' ContractPart* '}'

ContractPart = StateVariableDeclaration | UsingForDeclaration
             | StructDefinition | ModifierDefinition | FunctionDefinition |␣
↪EventDefinition | EnumDefinition

InheritanceSpecifier = UserDefinedTypeName ( '(' Expression ( ',' Expression )* ')' )?

StateVariableDeclaration = TypeName ( 'public' | 'internal' | 'private' | 'constant'␣
↪)* Identifier ('=' Expression)? ';'
UsingForDeclaration = 'using' Identifier 'for' ('*' | TypeName) ';'
StructDefinition = 'struct' Identifier '{'
                     ( VariableDeclaration ';' (VariableDeclaration ';')* ) '}'

ModifierDefinition = 'modifier' Identifier ParameterList? Block
ModifierInvocation = Identifier ( '(' ExpressionList? ')' )?

FunctionDefinition = 'function' Identifier? ParameterList
                     ( ModifierInvocation | StateMutability | 'external' | 'public' |
↪'internal' | 'private' )*
                     ( 'returns' ParameterList )? ( ';' | Block )
EventDefinition = 'event' Identifier EventParameterList 'anonymous'? ';'

EnumValue = Identifier
EnumDefinition = 'enum' Identifier '{' EnumValue? (',' EnumValue)* '}'

ParameterList = '(' ( Parameter (',' Parameter)* )? ')'
Parameter = TypeName StorageLocation? Identifier?

EventParameterList = '(' ( EventParameter (',' EventParameter )* )? ')'
EventParameter = TypeName 'indexed'? Identifier?

FunctionTypeParameterList = '(' ( FunctionTypeParameter (',' FunctionTypeParameter )*␣
↪)? ')'
FunctionTypeParameter = TypeName StorageLocation?

// semantic restriction: mappings and structs (recursively) containing mappings
// are not allowed in argument lists
VariableDeclaration = TypeName StorageLocation? Identifier

TypeName = ElementaryTypeName
         | UserDefinedTypeName
```

(continues on next page)

(이전 페이지에서 계속)

```
        | Mapping
        | ArrayTypeName
        | FunctionTypeName
        | ( 'address' 'payable' )

UserDefinedTypeName = Identifier ( '.' Identifier )*

Mapping = 'mapping' '(' ElementaryTypeName '=>' TypeName ')'
ArrayTypeName = TypeName '[' Expression? ']'
FunctionTypeName = 'function' FunctionTypeParameterList ( 'internal' | 'external' |␣
↪StateMutability )*
                   ( 'returns' FunctionTypeParameterList )?
StorageLocation = 'memory' | 'storage' | 'calldata'
StateMutability = 'pure' | 'view' | 'payable'

Block = '{' Statement* '}'
Statement = IfStatement | WhileStatement | ForStatement | Block |␣
↪InlineAssemblyStatement |
            ( DoWhileStatement | PlaceholderStatement | Continue | Break | Return |
              Throw | EmitStatement | SimpleStatement ) ';'

ExpressionStatement = Expression
IfStatement = 'if' '(' Expression ')' Statement ( 'else' Statement )?
WhileStatement = 'while' '(' Expression ')' Statement
PlaceholderStatement = '_'
SimpleStatement = VariableDefinition | ExpressionStatement
ForStatement = 'for' '(' (SimpleStatement)? ';' (Expression)? ';'␣
↪(ExpressionStatement)? ')' Statement
InlineAssemblyStatement = 'assembly' StringLiteral? InlineAssemblyBlock
DoWhileStatement = 'do' Statement 'while' '(' Expression ')'
Continue = 'continue'
Break = 'break'
Return = 'return' Expression?
Throw = 'throw'
EmitStatement = 'emit' FunctionCall
VariableDefinition = (VariableDeclaration | '(' VariableDeclaration? (','␣
↪VariableDeclaration? )* ')' ) ( '=' Expression )?

// Precedence by order (see github.com/ethereum/solidity/pull/732)
Expression
  = Expression ('++' | '--')
  | NewExpression
  | IndexAccess
  | MemberAccess
  | FunctionCall
  | '(' Expression ')'
  | ('!' | '~' | 'delete' | '++' | '--' | '+' | '-') Expression
  | Expression '**' Expression
  | Expression ('*' | '/' | '%') Expression
  | Expression ('+' | '-') Expression
  | Expression ('<<' | '>>') Expression
  | Expression '&' Expression
  | Expression '^' Expression
  | Expression '|' Expression
  | Expression ('<' | '>' | '<=' | '>=') Expression
  | Expression ('==' | '!=') Expression
  | Expression '&&' Expression
```

(continues on next page)

(이전 페이지에서 계속)

```
  | Expression '||' Expression
  | Expression '?' Expression ':' Expression
  | Expression ('=' | '|=' | '^=' | '&=' | '<<=' | '>>=' | '+=' | '-=' | '*=' | '/='␣
↪| '%=') Expression
  | PrimaryExpression

PrimaryExpression = BooleanLiteral
                  | NumberLiteral
                  | HexLiteral
                  | StringLiteral
                  | TupleExpression
                  | Identifier
                  | ElementaryTypeNameExpression

ExpressionList = Expression ( ',' Expression )*
NameValueList = Identifier ':' Expression ( ',' Identifier ':' Expression )*

FunctionCall = Expression '(' FunctionCallArguments ')'
FunctionCallArguments = '{' NameValueList? '}'
                      | ExpressionList?

NewExpression = 'new' TypeName
MemberAccess = Expression '.' Identifier
IndexAccess = Expression '[' Expression? ']'

BooleanLiteral = 'true' | 'false'
NumberLiteral = ( HexNumber | DecimalNumber ) (' ' NumberUnit)?
NumberUnit = 'wei' | 'szabo' | 'finney' | 'ether'
           | 'seconds' | 'minutes' | 'hours' | 'days' | 'weeks' | 'years'
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*

HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+ ( '.' [0-9]* )? ( [eE] [0-9]+ )?

TupleExpression = '(' ( Expression? ( ',' Expression? )*  )? ')'
                | '[' ( Expression  ( ',' Expression  )*  )? ']'

ElementaryTypeNameExpression = ElementaryTypeName

ElementaryTypeName = 'address' | 'bool' | 'string' | Int | Uint | Byte | Fixed |␣
↪Ufixed

Int = 'int' | 'int8' | 'int16' | 'int24' | 'int32' | 'int40' | 'int48' | 'int56' |
↪'int64' | 'int72' | 'int80' | 'int88' | 'int96' | 'int104' | 'int112' | 'int120' |
↪'int128' | 'int136' | 'int144' | 'int152' | 'int160' | 'int168' | 'int176' | 'int184
↪' | 'int192' | 'int200' | 'int208' | 'int216' | 'int224' | 'int232' | 'int240' |
↪'int248' | 'int256'

Uint = 'uint' | 'uint8' | 'uint16' | 'uint24' | 'uint32' | 'uint40' | 'uint48' |
↪'uint56' | 'uint64' | 'uint72' | 'uint80' | 'uint88' | 'uint96' | 'uint104' |
↪'uint112' | 'uint120' | 'uint128' | 'uint136' | 'uint144' | 'uint152' | 'uint160' |
↪'uint168' | 'uint176' | 'uint184' | 'uint192' | 'uint200' | 'uint208' | 'uint216' |
↪'uint224' | 'uint232' | 'uint240' | 'uint248' | 'uint256'

Byte = 'byte' | 'bytes' | 'bytes1' | 'bytes2' | 'bytes3' | 'bytes4' | 'bytes5' |
↪'bytes6' | 'bytes7' | 'bytes8' | 'bytes9' | 'bytes10' | 'bytes11' | 'bytes12' |
↪'bytes13' | 'bytes14' | 'bytes15' | 'bytes16' | 'bytes17' | 'bytes18' | 'bytes19' |
↪'bytes20' | 'bytes21' | 'bytes22' | 'bytes23' | 'bytes24' | 'bytes25' | 'bytes26' |
↪'bytes27' | 'bytes28' | 'bytes29' | 'bytes30' | 'bytes31' | 'bytes32'
```

(continues on next page)

(이전 페이지에서 계속)

```
Fixed = 'fixed' | ( 'fixed' [0-9]+ 'x' [0-9]+ )

Ufixed = 'ufixed' | ( 'ufixed' [0-9]+ 'x' [0-9]+ )

InlineAssemblyBlock = '{' AssemblyItem* '}'

AssemblyItem = Identifier | FunctionalAssemblyExpression | InlineAssemblyBlock |␣
↪AssemblyVariableDeclaration | AssemblyAssignment | AssemblyLabel | NumberLiteral |␣
↪StringLiteral | HexLiteral
AssemblyExpression = Identifier | FunctionalAssemblyExpression | NumberLiteral |␣
↪StringLiteral | HexLiteral
AssemblyVariableDeclaration = 'let' Identifier ':=' AssemblyExpression
AssemblyAssignment = ( Identifier ':=' AssemblyExpression ) | ( '=:' Identifier )
AssemblyLabel = Identifier ':'
FunctionalAssemblyExpression = Identifier '(' AssemblyItem? ( ',' AssemblyItem )* ')'
```

## 8.5 보안 측면 고려사항

소프트웨어를 원하는대로 개발하는 것은 어렵지 않지만, 다른 사람이 아예 다른 의도로 작동시키는걸 막는 것은 어렵습니다.

솔리디티에서는 모든 스마트 컨트랙트가 공개적으로 실행되고 대부분의 소스코드를 확인할 수 있는 경우가 많습니다. 따라서 이런 보안 측면의 고려사항은 특히 더 중요합니다.

물론 보안에 얼마나 신경을 써야하는 지는 상황에 따라 다릅니다. 가령, 웹 서비스는 공격자를 포함한 대중에게 공개되어야하고, 누구나 접근할 수 있어야하며, 어떤 때에는 오픈소스로 관리되는 경우도 있습니다. 만약 웹 서비스에 중요치 않은 정보만 저장한다면 문제가 되지 않지만, 은행 계좌와 같은 정보를 관리한다면 더욱 조심할 필요가 있죠.

이 장에서는 조심해야할 문제들과 일반적인 보안관련 패턴들을 다룹니다. 하지만 이는 완벽한 해결법이 아닙니다. 즉, 스마트 컨트랙트 상에는 버그가 없더라도, 컴파일러나 플랫폼 자체에 버그가 있을 수 있다는 얘기죠.

언제나 그렇듯이, 이 문서는 오픈 소스 기반의 문서이기 때문에, 보안에 대한 문제가 생긴다면 주저없이 내용을 추가해주시기 바랍니다.

### 8.5.1 유의사항

#### 개인 정보와 무작위성

스마트 컨트랙트 상의 모든 정보는 공개적으로 보여집니다. 심지어 지역 변수 및 상태 변수가 ‘‘private‘‘으로 선언되었다고해도 마찬가지죠.

만약 당신이 채굴자의 부정 행위를 막고자 한다면, 난수를 생성하는 것이 어느정도 유용할 수 있습니다.

#### 재진입 문제

(A)콘트랙트에서 (B)콘트랙트로 연결되는 어떠한 상호작용 및 Ether의 전송은 제어권을 (B)에게 넘겨주게 됩니다. 이 때문에 B의 상호작용이 끝나기 전에 다시 A를 호출할 수 있는 상황이 벌어질 수 있습니다. 예를 들어, 다음 코드는 버그를 포함하고 있습니다(요약된 코드입니다).

```
pragma solidity ^0.4.0;

// 버그가 포함된 코드입니다. 사용하지 마세요!
contract Fund {
    /// 컨트랙트의 Ether 정보 mapping
    mapping(address => uint) shares;
    /// 지분을 인출하는 함수
    function withdraw() public {
        if (msg.sender.send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

“send“함수 자체에서 gas의 소비량을 제어하기 때문에, 큰 문제는 되지 않지만, 그럼에도 이 코드는 보안 상의 문제를 가지고 있습니다. Ether의 전송은 항상 코드의 실행을 포함하기에, 수신자는 반복적으로 “withdraw“를 실행할 수 있게되죠. 결과적으로 중복된 “withdraw“함수의 실행을 통해 컨트랙트 상의 모든 Ether를 가져갈 수 있다는 의미입니다. 상황에 따라, 공격자는 아래 코드 속 “call“을 통해 남은 gas를 모두 가져올 수 있을지도 모릅니다.

```
pragma solidity ^0.4.0;

// 버그가 포함된 코드입니다. 사용하지 마세요!
contract Fund {
    /// 컨트랙트의 Ether 정보 mapping
    mapping(address => uint) shares;
    /// 지분을 인출하는 함수
    function withdraw() public {
        if (msg.sender.call.value(shares[msg.sender])())
            shares[msg.sender] = 0;
    }
}
```

재진입 공격을 막기 위해서는 아래와 같이 Checks-Effects-Interactions 패턴을 사용할 수 있습니다.

```
pragma solidity ^0.4.11;

contract Fund {
    /// 컨트랙트의 Ether 정보 mapping
    mapping(address => uint) shares;
    /// 지분을 인출하는 함수
    function withdraw() public {
        var share = shares[msg.sender];
        shares[msg.sender] = 0;
        msg.sender.transfer(share);
    }
}
```

재진입 공격은 Ether 전송에서 뿐만 아니라 함수를 호출하는 어떤 상황에서도 수행될 수 있습니다. 나아가, 여러분은 하나의 계정에 많은 컨트랙트를 가질 수도 있을 텐데요, 이 때, 하나의 컨트랙트가 다른 컨트랙트를 호출할 수 있다는 것도 알아둬야합니다.

### 가스 제한 및 루프

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully: Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. This may not apply to `constant` functions that are only executed

to read data from the blockchain. Still, such functions may be called by other contracts as part of on-chain operations and stall those. Please be explicit about such cases in the documentation of your contracts.

### Ether 보내고 받기

- Neither contracts nor "external accounts" are currently able to prevent that someone sends them Ether. Contracts can react on and reject a regular transfer, but there are ways to move Ether without creating a message call. One way is to simply "mine to" the contract address and the second way is using `selfdestruct(x)`.
- If a contract receives Ether (without a function being called), the fallback function is executed. If it does not have a fallback function, the Ether will be rejected (by throwing an exception). During the execution of the fallback function, the contract can only rely on the "gas stipend" (2300 gas) being available to it at that time. This stipend is not enough to access storage in any way. To be sure that your contract can receive Ether in that way, check the gas requirements of the fallback function (for example in the "details" section in Remix).
- There is a way to forward more gas to the receiving contract using `addr.call.value(x)()`. This is essentially the same as `addr.transfer(x)`, only that it forwards all remaining gas and opens up the ability for the recipient to perform more expensive actions (and it only returns a failure code and does not automatically propagate the error). This might include calling back into the sending contract or other state changes you might not have thought of. So it allows for great flexibility for honest users but also for malicious actors.
- If you want to send Ether using `address.transfer`, there are certain details to be aware of:
  1. If the recipient is a contract, it causes its fallback function to be executed which can, in turn, call back the sending contract.
  2. Sending Ether can fail due to the call depth going above 1024. Since the caller is in total control of the call depth, they can force the transfer to fail; take this possibility into account or use `send` and make sure to always check its return value. Better yet, write your contract using a pattern where the recipient can withdraw Ether instead.
  3. Sending Ether can also fail because the execution of the recipient contract requires more than the allotted amount of gas (explicitly by using `require`, `assert`, `revert`, `throw` or because the operation is just too expensive) - it "runs out of gas" (OOG). If you use `transfer` or `send` with a return value check, this might provide a means for the recipient to block progress in the sending contract. Again, the best practice here is to use a *"withdraw" pattern instead of a "send" pattern*.

### 콜스택 깊이

External function calls can fail any time because they exceed the maximum call stack of 1024. In such situations, Solidity throws an exception. Malicious actors might be able to force the call stack to a high value before they interact with your contract.

Note that `.send()` does **not** throw an exception if the call stack is depleted but rather returns `false` in that case. The low-level functions `.call()`, `.callcode()` and `.delegatecall()` behave in the same way.

### tx.origin

Never use tx.origin for authorization. Let's say you have a wallet contract like this:

```
pragma solidity ^0.4.11;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;
```

(continues on next page)

(이전 페이지에서 계속)

```
    function TxUserWallet() public {
        owner = msg.sender;
    }

    function transferTo(address dest, uint amount) public {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Now someone tricks you into sending ether to the address of this attack wallet:

```
pragma solidity ^0.4.11;

interface TxUserWallet {
    function transferTo(address dest, uint amount) public;
}

contract TxAttackWallet {
    address owner;

    function TxAttackWallet() public {
        owner = msg.sender;
    }

    function() public {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

If your wallet had checked `msg.sender` for authorization, it would get the address of the attack wallet, instead of the owner address. But by checking `tx.origin`, it gets the original address that kicked off the transaction, which is still the owner address. The attack wallet instantly drains all your funds.

### Minor Details

- In `for (var i = 0; i < arrayName.length; i++) { ... }`, the type of `i` will be `uint8`, because this is the smallest type that is required to hold the value `0`. If the array has more than 255 elements, the loop will not terminate.
- The `constant` keyword for functions is currently not enforced by the compiler. Furthermore, it is not enforced by the EVM, so a contract function that "claims" to be constant might still cause changes to the state.
- Types that do not occupy the full 32 bytes might contain "dirty higher order bits". This is especially important if you access `msg.data` - it poses a malleability risk: You can craft transactions that call a function `f(uint8 x)` with a raw byte argument of `0xff000001` and with `0x00000001`. Both are fed to the contract and both will look like the number `1` as far as `x` is concerned, but `msg.data` will be different, so if you use `keccak256(msg.data)` for anything, you will get different results.

### 8.5.2 Recommendations

#### Restrict the Amount of Ether

Restrict the amount of Ether (or other tokens) that can be stored in a smart contract. If your source code, the compiler or the platform has a bug, these funds may be lost. If you want to limit your loss, limit the amount of Ether.

#### Keep it Small and Modular

Keep your contracts small and easily understandable. Single out unrelated functionality in other contracts or into libraries. General recommendations about source code quality of course apply: Limit the amount of local variables, the length of functions and so on. Document your functions so that others can see what your intention was and whether it is different than what the code does.

#### Use the Checks-Effects-Interactions Pattern

Most functions will first perform some checks (who called the function, are the arguments in range, did they send enough Ether, does the person have tokens, etc.). These checks should be done first.

As the second step, if all checks passed, effects to the state variables of the current contract should be made. Interaction with other contracts should be the very last step in any function.

Early contracts delayed some effects and waited for external function calls to return in a non-error state. This is often a serious mistake because of the re-entrancy problem explained above.

Note that, also, calls to known contracts might in turn cause calls to unknown contracts, so it is probably better to just always apply this pattern.

#### Include a Fail-Safe Mode

While making your system fully decentralised will remove any intermediary, it might be a good idea, especially for new code, to include some kind of fail-safe mechanism:

You can add a function in your smart contract that performs some self-checks like "Has any Ether leaked?", "Is the sum of the tokens equal to the balance of the contract?" or similar things. Keep in mind that you cannot use too much gas for that, so help through off-chain computations might be needed there.

If the self-check fails, the contract automatically switches into some kind of "failsafe" mode, which, for example, disables most of the features, hands over control to a fixed and trusted third party or just converts the contract into a simple "give me back my money" contract.

### 8.5.3 Formal Verification

Using formal verification, it is possible to perform an automated mathematical proof that your source code fulfills a certain formal specification. The specification is still formal (just as the source code), but usually much simpler.

Note that formal verification itself can only help you understand the difference between what you did (the specification) and how you did it (the actual implementation). You still need to check whether the specification is what you wanted and that you did not miss any unintended effects of it.

## 8.6 컴파일러 사용하기

### 8.6.1 명령행 컴파일러 사용하기

**주석:** 이 섹션은 :ref:‘solcjs <solcjs>‘에 적용 되지 않습니다.

Solidity 저장소의 빌드 대상 중 하나는 Solidity 명령행 컴파일러인 `solc``입니다, ``solc --help``를 사용하면 모든 옵션에 대한 설명을 제공합니다. 컴파일러는 간단한 구문 트리 (구문 분석 트리)를 통한 간단한 바이너리 및 어셈블리부터 가스 사용량 평가에 이르기까지 다양한 출력을 생성 할 수 있습니다. 만약 단일 파일만 컴파일 하기를 원한다면, ``solc --bin sourceFile.sol``를 실행하세요. 바이너리가 출력 될 것입니다. 당신의 컨트렉트를 배치(deploy)하기전에 ``solc --optimize --bin sourceFile.sol``를 이용하여 컴파일하는 동안 최적화기(Optimizer)를 활성화 시키세요. 만약 조금더 진보된 다른 형태의 ``solc``의 결과를 얻기를 원한다면, 아마도 ``solc -o outputDirectory --bin --ast --asm sourceFile.sol 를 사용하여 분할 파일들을 모두 출력하도록 명령하는 것이 좋을 것입니다.

명령행 컴파일러는 자동적으로 파일 시스템으로 부터 수입된(imported) 파일들을 읽습니다. 그러나 아래의 방법으로 ‘‘prefix=path‘‘를 사용해 리다이렉트할 경로를 제공하는 것 또한 가능합니다.

```
solc github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/ =/usr/local/lib/fallback␣
↪file.sol
```

이것은 본질적으로 /usr/local/lib/dapp-bin``아래에 있는 ``github.com/ethereum/dapp-bin/``로 시작하는 것을 검색하라고 컴파일러에게 지시합니다. 그리고 만약 거기에 있는 파일을 찾지 못한다면, 컴파일러는 ``/usr/local/lib/fallback``를 살펴 볼것입니다. (공백의 접두사는 항상 일치한다.) ``solc``은 외부에 재 맵핑 대상 외부와 명시적으로 소스파일 위치가 명시된 디렉터리 외부에 있는 파일 시스템으로 부터 파일 시스템로 부터 파일들을 읽지 않습니다. 그러므로 ``import "/etc/passwd";``와 같은 것들은 오직 재맵 핑(remapping)으로서 ``=/ 추가하는 경우에만 작동합니다.

만약 재맵핑으로 인해 여러 일치 항목이 있는 경우 가장 긴 공통 접두사가 있는 항목이 선택됩니다.

보안상의 이유들로 컴파일러는 액세스 할 수 있는 디렉터리를 제한합니다. 명령행에 명시된 소스파일 경로(및 해당 하위 디렉터리)와 재맵핑에 의해 정의 된 경로들은 import 문에서 허용됩니다, 그러나 다른 모든 것들은 거절됩니다. 추가적인 경로(및 그것의 하위 디렉터리)는 --allow-paths /sample/path,/another/sample/path 스위치를 통해 허용 될 수 있습니다.

만약 컨트렉트가 :ref:‘libraries <libraries>‘를 사용한다면, 바이트코드가 ‘‘__LibraryName_____‘‘의 하위 문자열(substrings)를 포함한다는 것을 알아야합니다. 그 지점에 당신의 라이브러리 주소가 삽입될것을 의미하는 링커로써 ‘‘solc‘‘을 사용 할 수 있습니다.

각각의 라이브러리 주소를 제공하기 위해서 커맨드에 ‘‘–libraries "Math:0x12345678901234567890 Heap:0xabcdef0123456"‘‘를 추가하던지 파일(한줄에 하나의 라이브러리)에 문자열을 저장하고 ‘‘–libraries fileName‘‘를 사용하여 ‘‘solc‘‘를 실행하세요

만약 solc``가 ``--link``옵션과 함께 호출 된다면, 모든 입력 파일 들은 주어진 ``__LibraryName_____ 형식의 연결(link)되지 않은 (16진수로 인코드 된)바이더리들로 해석되어 현재 위치에 연결 됩니다(만약 입력이 stdin으로 부터 읽어온것이라면, 그것은 stdout으로 쓰여집니다). --libraries``를 제외한 모든 옵션 은 이 경우에 무시됩니다(-o‘‘ 포함)

만약 ‘‘solc‘‘가 옵션 ‘‘–standard-json‘‘과 함께 호출 되었다면, 그것은 표준 입력에서 JSON 입력(아래에 설명)을 기대합니다 그리고 표준출력의 JSON출력을 반환합니다..

### 8.6.2 컴파일러 입력과 출력 JSON 설명

이러한 JSON 형식은 ‘‘solc‘‘를 통해 가능한것과 같이 컴파일러 API에 의해 사용됩니다. 이러한 것들은 변화될 수 있고, 몇몇 필드는 선택적이지만(앞에서 말한대로), 이전 버전과의 호환성을 위해서만 사용됩니다.

컴파일러 API는 JSON 형식으로 입력을 기대하고 JSON 형식으로 컴파일의 결과를 출력합니다.

주석은 당연히 설명목적으로만 허용되고 여기에 사용되지 않습니다.

#### 입력 설명

```
{
  // Required: Source code language, such as "Solidity", "serpent", "lll", "assembly",
↪ etc.
  language: "Solidity",
  // Required
  sources:
  {
            // 여기에 이 키는 소스코드 파일들의 "전역(global)" 이름들입니다.
    // 임포트는 재맵핑을 통해 다른 파일을 사용할 수 있습니다.
    "myFile.sol":
    {
      // 선택적 : 소스 파일의 keccak256 해시
      // 이것은 URL을 통해 임포트된 경우 내용을 검증하기 위해 사용됩니다.
      "keccak256": "0x123...",
      // Required (만약 "content"가 사용되지 않았다면 아래를 보세요): 소스파일의 URL
      // URL(s) should be imported in this order and the result checked against the
      // keccak256 hash (if available). If the hash doesn't match or none of the
      // URL은 여기 안에 임포트 되어야만 합니다. 그리고 결과는 keccak256 해시에 대해
      // 확인해야 합니다. (가능한 경우에). 만약 해시가 맞지 않거나 성공한 URL이 없다면
      // 에러가 발생해야 합니다.
      "urls":
      [
        "bzzr://56ab...",
        "ipfs://Qma...",
        "file:///tmp/path/to/file.sol"
      ]
    },
    "mortal":
    {
      // Optional: 소스파일의 keccak256 해시
      "keccak256": "0x234...",
      // Required (만약 "urls"가 사용되지 않으면): 소스 파일의 리터럴 내용
      "content": "contract mortal is owned { function kill() { if (msg.sender ==␣
↪owner) selfdestruct(owner); } }"
    }
  },
  // Optional
  settings:
  {
    // Optional: 재맵핑의 정렬된 리스트
    remappings: [ ":g/dir" ],
    // Optional: 최적화기 (enabled defaults to false)
    optimizer: {
      enabled: true,
      runs: 500
    },
```

(continues on next page)

(이전 페이지에서 계속)

```
    evmVersion: "byzantium", // Version of the EVM to compile for. Affects type␣
↪checking and code generation. Can be homestead, tangerineWhistle, spuriousDragon,␣
↪byzantium or constantinople
    // Metadata settings (optional)
    metadata: {
      // URL이 아닌 리터럴 내용만 사용하세요. (기본값 : false)
      useLiteralContent: true
    },
    // 라이브러리들의 주소. 만약 모든 라이브러리가 여기에 주어지지 않는다면, 그것은 출력 데이터가 다른 연
결되지 않은 객체를 초례할 수 있습니다.
    libraries: {
      // 최상위 레벨 키는 라이브러리가 사용된 소스파일의 이름입니다.
      // 만약 재맵핑이 사용되었다면, 재 맵핑이 적용된 후에, 이 소스 파일은 전역 경로가 일치해야 합니다.
      // 만약 이 키가 빈 문자열이라면, 그것은 전역 수준을 참조합니다.
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    }
    // The following can be used to select desired outputs.
    // 아래의 코드는 원하는 출력을 선택하는데 사용할 수 있습니다.
    // 만약 이 필드가 누락 된다면, 컴파일러는 불러오고 타입을 체크 합니다. 그러나 에러러부터 어떠한 에러
도 생성하지 않습니다.

    // 첫번째 레벨의 키는 파일 이름이고 두번재는 컨트렉트 이름입니다. 여기서 빈 계약이름은 파일 자체를 나
타냅니다,
    // star가 컨트렉트의 모든 내용을 참조하는 동안.
    //
    // 아래는 가능한 출력 타입입니다.
    //   abi - ABI
    //   ast - 모든 소스파일의 AST
    //   legacyAST - 모든 소스파일의 legacy AST
    //   devdoc - 개발자 문서 (natspec)
    //   userdoc - 사용자 문서 (natspec)
    //   metadata - 메타데이터
    //   ir - desugaring이전의 새로운 어셈블리 형식
    //   evm.assembly - desugaring이후의 새로운 어셈블리 형식
    //   evm.legacyAssembly - 이전 스타일의 JSON형식 어셈블리
    //   evm.bytecode.object - 바이트 코드 객체
    //   evm.bytecode.opcodes - Opcodes 리스트
    //   evm.bytecode.sourceMap - 소스 맵핑 (디버그에 유용함)
    //   evm.bytecode.linkReferences - 링크 참조 (if unlinked object)
    //   evm.deployedBytecode* - 배포된 바이트코드 (evm.bytecode과 동일한 옵션을 가짐)
    //   evm.methodIdentifiers - 해시함수 리스트
    //   evm.gasEstimates - 가스 측정함수
    //   ewasm.wast - eWASM S-expressions format (not supported atm)
    //   ewasm.wasm - eWASM 바이터리 데이터 (not supported atm)
    //
    // Note that using a using `evm`, `evm.bytecode`, `ewasm`, etc. will select every
    // target part of that output. Additionally, `*` can be used as a wildcard to␣
↪request everything.
    //
    outputSelection: {
      // Enable the metadata and bytecode outputs of every single contract.
      "*": {
        "*": [ "metadata", "evm.bytecode" ]
      },
      // Enable the abi and opcodes output of MyContract defined in file def.
```

(continues on next page)

(이전 페이지에서 계속)

```
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      },
      // Enable the source map output of every single contract.
      "*": {
        "*": [ "evm.bytecode.sourceMap" ]
      },
      // Enable the legacy AST output of every single file.
      "*": {
        "": [ "legacyAST" ]
      }
    }
  }
}
```

### 출력 설명

```
{
  // 선택적 : 에러나 경고가 발생 했는지 나타내지 않습니다.
  errors: [
    {
      // Optional: 소스 파일안 위치.
      sourceLocation: {
        file: "sourceFile.sol",
        start: 0,
        end: 100
      ],
      // 의무적 : "TypeError", "InternalCompilerError", "Exception"등 과 같은 에러 타입
      // 아래 타입 리스트를 보세요.
      type: "TypeError",
      // 의무적 : "general", "ewasm"등과 같은 에러가 발생한 컴포넌트
      component: "general",
      // 의무적 ("error" or "warning")
      severity: "error",
      // 의무적
      message: "Invalid keyword"
      // 선택적 : 소스 위치를 포함한 형식을 갖춘 메세지
      formattedMessage: "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // 이것은 파일 수준 출력을 포함합니다. 이것은 outputSelection 설정에 의해 제한되고 걸러질 수 있습니
다.
  sources: {
    "sourceFile.sol": {
      // Identifier (used in source maps)
      id: 1,
      // The AST object
      ast: {},
      // The legacy AST object
      legacyAST: {}
    }
  },
  // 이것은 컨트렉트 수준 출력을 포함합니다. 이것은 outputSelection설정에 의해 제한되고 걸러질 수 있습
니다.
  contracts: {
```

(continues on next page)

(이전 페이지에서 계속)

```
    "sourceFile.sol": {
      // 만약 사용된 언어가 컨트렉트 이름을 가지고 있지 않다면, 이 필드는 빈 문자열과 같아야 합니다.
      "ContractName": {
        // 이더리움 컨트렉트 ABI. 만약 비어있다면, 이것은 빈 배열을 나타냅니다.
        // https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI 를 확인해 보세요.
        abi: [],
        // 메타데이터 출력 문서를 확인해보세요.
        metadata: "{...}",
        // 사용자 문서 (natspec)
        userdoc: {},
        // 개발자 문서 (natspec)
        devdoc: {},
        // 중간 표현 (string)
        ir: "",
        // EVM-related outputs
        evm: {
          // Assembly (string)
          assembly: "",
          // 이전 스타일의 어셈블리 (object)입니다.
          legacyAssembly: {},
          // 바이트 코드와 자세한 내용.
          bytecode: {
            // 16진수 인 바이트 코드입니다.
            object: "00fe",
            // OPcodes 리스트 (string)입니다.
            opcodes: "",
            // 문자열로써 소스 맵핑입니다. 소스 맵핑 정의를 확인해 보세요.
            sourceMap: "",
            // 주어졌다면, 이것은 연결되지 않은 객체입니다.
            linkReferences: {
              "libraryFile.sol": {
                // Byte offsets into the bytecode. Linking replaces the 20 bytes␣
↪located there.
                "Library1": [
                  { start: 0, length: 20 },
                  { start: 200, length: 20 }
                ]
              }
            }
          },
          // 위와 같은 레이아웃 입니다.
          deployedBytecode: { },
          // 해시 함수 리스트 입니다.
          methodIdentifiers: {
            "delegate(address)": "5c19a95c"
          },
          // Function gas estimates
          // 가스 예측 함수 입니다.
          gasEstimates: {
            creation: {
              codeDepositCost: "420000",
              executionCost: "infinite",
              totalCost: "infinite"
            },
            external: {
              "delegate(address)": "25000"
            },
```

(continues on next page)

(이전 페이지에서 계속)

```
            internal: {
              "heavyLifting()": "infinite"
            }
          }
        },
        // 출력과 연관된 eWASM입니다.
        ewasm: {
          // S-expressions 형식입니다.
          wast: "",
          // Binary format (hex string)
          // 바이너리 형식 (hex string)
          wasm: ""
        }
      }
    }
  }
}
```

### 에러 타입

1. "JSONError" : JSON 입력은 요구된 형식에 일치하지 않습니다. 예시) 입력이 json 오브젝트가 아닙니다. 그 언어는 지원되지 않습니다. 등.
2. "IOError" : IO와 임포트 과정에서의 에러들입니다, 분석될 수 없는 URL이나 공급된 소스에서의 해시 불일치와 같은것들이 있습니다.
3. "ParserError" : 소스코드는 언어 원칙에 일치하지 않습니다.
4. "DocstringParsingError" : 커맨드 블록에서 NatSpec 태그는 분석될 수 없습니다.
5. "SyntaxError" : Syntactical error는 "continue"가 "for" 반복 외부에서 사용 되는것등이 있습니다.
6. "DeclarationError" : 유효하지 않거나 혹은 의결 할수 없는(unresolvable), 식별자 이름충돌입니다. 예시 "identifier not found" 식별자가 발견되지 않음
7. "TypeError" : 유효하지 않은 타입 변경, 유효하지 않은 할당(assignment) 등과 같은 type system내의 에러입니다.
8. "UnimplementedFeatureError" : 기능이 컴파일러에 의해 지원되지 않습니다. 하지만 미래 번전에서는 지원될 것으로 예상됩니다.
9. "internalCompilerError" : 컴파일러에 의해 촉발되는 내부의 버그 - 이것은 문제로서 보고되어져야 합니다.
10. "Exception" : 컴파일러 도중에 알려지지 않은 실패 - 이것은 문제로서 보고되어야 합니다.
11. "CompilerError" : 유효하지 않은 컴파일러 스택의 사용 - 이것은 문제로서 되어야 합니다.
12. "FatalError" : 치명적 오류가 바르게 처리되지 않음 - 이는 문제로서 기록되어야 합니다.
13. "Warning" : 단순 경고, 컴파일러를 중단 하지는 않지만, 가능하다면 다뤄져야 합니다.

## 8.7 컨트랙트 메타데이터

Solidity 컴파일러는 현재 컨트랙트에 대한 정보를 포함하고 있는 컨트랙트 메타데이터 JSON파일을 자동적으로 생성합니다. 이 파일을 사용하여 컴파일러 버전, 사용된 소스, ABI와 NatSpec 문서를 질의하여 보다 안전하게 컨트랙트와 상호작용하고 소스코드를 검증 할 수 있습니다.

이 컴파일러는 메타데이터 파일의 Swarm 해시를 각 컨트랙트의 byte코드 (자세한 내용은 아래를 참조) 끝에 추가하므로 중앙 집중식 데이터 공급자가 없어도 인증된 방식으로 파일을 검색 할 수 있습니다.

다른 사용자가 액세스 할 수 있도록 Swarm(또는 다른 서비스)에 메타데이터 파일을 게시해야 합니다. `ContractName_meta.Json` 라 불리는 파일을 생성하는 `solc --metadata` 명령어를 사용하여 파일을 만듭니다. 이 파일에는 소스코드에 대한 Swarm 참조가 포함되어 있으므로, 모든 소스파일과 메타데이터 파일을 업로드해야 합니다.

메타데이터 파일은 다음과 같은 형식을 가집니다. 아래의 예제는 사람이 읽을 수 있는 방식으로 제공됩니다. 적절한 형식의 메타데이터는 따옴표를 올바르게 사용해야 하고, 공백을 최소한으로 줄이며 모든 객체의 키를 고유한 형식으로 정렬해야 합니다. 주석은 사용할 수 없습니다. 여기서는 오로지 설명의 목적으로 사용되었습니다.

```
{
  // 필수 : 메타데이터 형식의 버전
  version: "1",
  // 필수 : 소스코드 언어는 기본적으로 사양의
  // "하위 버전(sub-version)"을 선택합니다.
  language: "Solidity",
  // 필수 : 컴파일러에 대한 세부사항, 내용은
  // 언어에 따라 다릅니다.
  compiler: {
    // Solidity에 필수 : 컴파일러의 버전
    version: "0.4.6+commit.2dabbdf0.Emscripten.clang",
    // 선택 사항 : 이 출력을 생성한 컴파일러 바이너리의 해시
    keccak256: "0x123..."
  },
  // 필수 : 컴파일 소스 파일 / 소스 유닛, 키는 파일 이름입니다.
  sources:
  {
    "myFile.sol": {
      // 필수 : 소스 파일의 keccak256 해시
      "keccak256": "0x123...",
      // 필수 ("content"를 사용하지 않는 한 아래를 참조) : 소스파일에
      // 정렬된 URL, 프로토콜은 다소 제멋대로지만, Swarm URL을
      // 권장합니다.
      "urls": [ "bzzr://56ab..." ]
    },
    "mortal": {
      // 필수 : 소스 파일의 keccak256 해시
      "keccak256": "0x234...",
      // 필수("url"을 사용하지 않는 한) : 소스 파일의 literal 내용 // TODO : Review needed
      "content": "contract mortal is owned { function kill() { if (msg.sender ==␣
↪owner) selfdestruct(owner); } }"
    }
  },
  // 필수 : 컴파일러 세팅
  settings:
  {
    // Solidity에 필수 : 재맵핑의 정렬된 목록 // TODO : Review needed
    remappings: [ ":g/dir" ],
    // 선택 사항 : 최적화 설정 (기본값 : false)
    optimizer: {
      enabled: true,
      runs: 500
    },
    // Solidity에 필수 : 이 메타데이터가 작성된
    // 컨트랙트나 라이브러리의 파일과 이름
    compilationTarget: {
```

(continues on next page)

(이전 페이지에서 계속)

```
      "myFile.sol": "MyContract"
    },
    // Solidity에 필수 : 사용된 라이브러리의 주소
    libraries: {
      "MyLib": "0x123123..."
    }
  },
  // 필수 : 컨트랙트에 대한 생성된 정보
  output:
  {
    // 필수 : 컨트랙트의 ABI 정의
    abi: [ ... ],
    // 필수 : 컨트랙트의 NatSpec 사용자 문서
    userdoc: [ ... ],
    // 필수 : 컨트랙트의 NatSpec 개발자 문서
    devdoc: [ ... ],
  }
}
```

### 8.7.1 Byte코드에서 메타데이터 해시의 인코딩

앞으로 메타데이터 파일을 검색하는 다른 방법을 지원할 수 있기 때문에, 매핑 `{"bzzr0": <Swarm hash>}` 는 *CBOR* <*https://tools.ietf.org/html/rfc7049*> 인코딩으로 저장됩니다. 인코딩의 시작부분을 찾기가 쉽지 않기 때문에, 길이가 2 바이트 빅엔디안 인코딩으로 추가됩니다. Solidity 컴파일러의 현재 버전은 배포된 byte코드의 끝에 다음을 추가합니다:

```
0xa1 0x65 'b' 'z' 'z' 'r' '0' 0x58 0x20 <32 bytes swarm hash> 0x00 0x29
```

따라서 데이터를 검색하기 위해, 배치된 byte코드의 끝을 검사하여 위의 패턴과 일치시키고 Swarm 해시를 사용하여 파일을 검색 할 수 있습니다.

### 8.7.2 자동 인터페이스 생성 및 NatSpec의 사용

메타데이터는 다음과 같은 방식으로 사용됩니다: 컨트랙트와 상호작용하려는 구성요소(예:Mist 혹은 지갑)는 검색된 파일의 Swarm 해시에서 컨트랙트의 코드를 검색합니다. 이 파일은 위와 같은 구조로 JSON으로 디코딩 됩니다.

구성요소는 ABI를 사용하여 컨트랙트에 대한 기본 사용자 인터페이스를 자동으로 생성 할 수 있습니다.

또한, Wallet은 NatSpec 사용자 문서를 사용하여 컨트랙트와 상호작용 할 때마다 트랜잭션 서명에 대한 승인 요청과 함께 사용자에게 확인 메세지를 표시 할 수 있습니다.

이더리움 Natural Specification (NatSpec) 에 대한 추가정보는 여기 에서 찾을 수 있습니다.

### 8.7.3 소스코드 검증 사용법

컴파일을 확인하기 위해, 메타데이터 파일의 링크를 통해 Swarm에서 소스를 검색 할 수 있습니다. 올바른 버전의 컴파일러("공식" 컴파일러의 일부로 확인 됨) 는 특별한 설정으로 해당 입력에서 호출됩니다. 결과 byte코드는 작성 트랜잭션 또는 `CREATE` opcode 데이터의 데이터와 비교됩니다. 이는 해시가 byte코드의 일부이기 때문에 메타데이터가 자동으로 증명됩니다. 잉여데이터는 생성자 입력데이터와 동일해야하며, 이는 인터페이스에 따라 디코딩되어 사용자에게 제공되어야 합니다.

## 8.8 어플리케이션 바이너리 인터페이스 설명

### 8.8.1 기본 디자인

어플리케이션 바이너리 인터페이스는 블럭체인 밖에서든 컨트렉트 대 컨트렉트 상호작용이든, 이더리움 생태계에서 컨트렉트들과 상호작용 하는 표준 방법 입니다. 이 설명에 나와있는 것 같이, 타입에 따라 데이터는 인코딩되어 집니다. 인코딩은 스스로 설명하지 않기때문에 디코드 하기 위해 스키마를 요구합니다.

컨트렉트의 인터페이스 함수는 컴파일 타임 및 정적으로 알려진 강력한 형식이라고 가정합니다. 내성 메커니즘은 제공되지 않습니다. 모든 컨트렉트에는 컴파일 타임에 사용할 수있는 모든 컨트렉트의 인터페이스 정의가 있다고 가정합니다.

이 설명은 인터페이스가 동적이거나 달리 런타임에만 알려져있는 계약을 다루지 않습니다. 이러한 사례가 중요해지면 Ethereum 생태계 내에 구축 된 시설로 적절하게 처리 할 수 있습니다.

### 8.8.2 함수 선택자

함수 호출을 위한 호출 데이터의 첫 4바이트는 호출될 함수를 지정합니다. 그것은 함수 시그니쳐의 Keccak (SHA-3) 해시의 첫 번째(빅 엔디안)에서 4바이트 입니다. 이 시그니쳐는 기본 프로토콜의 정식 표현으로써 정의됩니다. 매개변수 타입의 괄호로 묶인 목록이 있는 함수 이름. 매개 변수 타입은 공백을 사용하지 않고 단일 쉼표로 구분합니다.

---

**주석:** 함수의 반환 타입은 이 시그니쳐의 일부분이 아닙니다. 솔리디티 함수 오버로딩)를 확인하세요.

---

### 8.8.3 인자 인코딩

다섯 번째 바이트부터 인코딩 된 인수가옵니다. 이 인코딩은 다른 장소에서도 사용됩니다 (예 : 반환 값과 이벤트 인수는 함수를 지정하는 4 바이트가없는 동일한 방식으로 인코딩됩니다.

### 8.8.4 Types

The following elementary types exist:

- `uint<M>`: unsigned integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`. e.g. `uint32`, `uint8`, `uint256`.
- `int<M>`: two's complement signed integer type of `M` bits, `0 < M <= 256`, `M % 8 == 0`.
- `address`: equivalent to `uint160`, except for the assumed interpretation and language typing. For computing the function selector, `address` is used.
- `uint`, `int`: synonyms for `uint256`, `int256` respectively. For computing the function selector, `uint256` and `int256` have to be used.
- `bool`: equivalent to `uint8` restricted to the values 0 and 1. For computing the function selector, `bool` is used.
- `fixed<M>x<N>`: signed fixed-point decimal number of `M` bits, `8 <= M <= 256`, `M % 8 ==0`, and `0 < N <= 80`, which denotes the value `v` as `v / (10 ** N)`.
- `ufixed<M>x<N>`: unsigned variant of `fixed<M>x<N>`.
- `fixed`, `ufixed`: synonyms for `fixed128x19`, `ufixed128x19` respectively. For computing the function selector, `fixed128x19` and `ufixed128x19` have to be used.

- `bytes<M>`: binary type of `M` bytes, `0 < M <= 32`.
- `function`: an address (20 bytes) folled by a function selector (4 bytes). Encoded identical to `bytes24`.

The following (fixed-size) array type exists:

- `<type>[M]`: a fixed-length array of `M` elements, `M > 0`, of the given type.

The following non-fixed-size types exist:

- `bytes`: dynamic sized byte sequence.
- `string`: dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]`: a variable-length array of elements of the given type.

Types can be combined to a tuple by enclosing a finite non-negative number of them inside parentheses, separated by commas:

- `(T1,T2,...,Tn)`: tuple consisting of the types `T1`, ..., `Tn`, `n >= 0`

It is possible to form tuples of tuples, arrays of tuples and so on.

---

**주석:** Solidity supports all the types presented above with the same names with the exception of tuples. The ABI tuple type is utilised for encoding Solidity `structs`.

---

### 8.8.5 Formal Specification of the Encoding

We will now formally specify the encoding, such that it will have the following properties, which are especially useful if some arguments are nested arrays:

Properties:

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]`. In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.
2. The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative "addresses"

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

**Definition:** The following types are called "dynamic":

- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k > 0`
- `(T1,...,Tk)` if any `Ti` is dynamic for `1 <= i <= k`

All other types are called "static".

**Definition:** `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

**Definition:** For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being

- `(T1,...,Tk)` for `k >= 0` and any types `T1`, ..., `Tk`

  `enc(X) = head(X(1)) ... head(X(k-1)) tail(X(0)) ... tail(X(k-1))`

  where `X(i)` is the `i`th component of the value, and `head` and `tail` are defined for `Ti` being a static type as

  `head(X(i)) = enc(X(i))` and `tail(X(i)) = ""` (the empty string)

  and as

  `head(X(i)) = enc(len(head(X(0)) ... head(X(k-1)) tail(X(0)) ... tail(X(i-1)))) tail(X(i)) = enc(X(i))`

  otherwise, i.e. if `Ti` is a dynamic type.

  Note that in the dynamic case, `head(X(i))` is well-defined since the lengths of the head parts only depend on the types and not the values. Its value is the offset of the beginning of `tail(X(i))` relative to the start of `enc(X)`.

- `T[k]` for any `T` and `k`:

  `enc(X) = enc((X[0], ..., X[k-1]))`

  i.e. it is encoded as if it were a tuple with `k` elements of the same type.

- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):

  `enc(X) = enc(k) enc([X[1], ..., X[k]])`

  i.e. it is encoded as if it were an array of static size `k`, prefixed with the number of elements.

- `bytes`, of length `k` (which is assumed to be of type `uint256`):

  **`enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a** `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.

- `string`:

  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is utf-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the utf-8 encoded string, not its number of characters.

- `uint<M>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is 32 bytes.

- `address`: as in the `uint160` case

- `int<M>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-order (left) side with `0xff` for negative `X` and with zero bytes for positive `X` such that the length is 32 bytes.

- `bool`: as in the `uint8` case, where `1` is used for `true` and `0` for `false`

- `fixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `int256`.

- `fixed`: as in the `fixed128x19` case

- `ufixed<M>x<N>`: `enc(X)` is `enc(X * 10**N)` where `X * 10**N` is interpreted as a `uint256`.

- `ufixed`: as in the `ufixed128x19` case

- `bytes<M>`: `enc(X)` is the sequence of bytes in `X` padded with trailing zero-bytes to a length of 32 bytes.

Note that for any `X`, `len(enc(X))` is a multiple of 32.

### 8.8.6 Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

```
function_selector(f) enc((a_1, ..., a_n))
```

and the return values `v_1, ..., v_k` of `f` are encoded as

```
enc((v_1, ..., v_k))
```

i.e. the values are combined into a tuple and encoded.

### 8.8.7 Examples

Given the contract:

```
pragma solidity ^0.4.16;

contract Foo {
  function bar(bytes3[2]) public pure {}
  function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
  function sam(bytes, bool, uint[]) public pure {}
}
```

Thus for our `Foo` example if we wanted to call `baz` with the parameters `69` and `true`, we would pass 68 bytes total, which can be broken down into:

- `0xcdcd77c0`: the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature `baz(uint32,bool)`.
- `0x0000000000000000000000000000000000000000000000000000000000000045`: the first parameter, a uint32 value `69` padded to 32 bytes
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the second parameter - boolean `true`, padded to 32 bytes

In total:

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000045000000000000000000000000000000
```

It returns a single `bool`. If, for example, it were to return `false`, its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000`, a single bool.

If we wanted to call `bar` with the argument `["abc", "def"]`, we would pass 68 bytes total, broken down into:

- `0xfce353f6`: the Method ID. This is derived from the signature `bar(bytes3[2])`.
- `0x6162630000000000000000000000000000000000000000000000000000000000`: the first part of the first parameter, a `bytes3` value `"abc"` (left-aligned).
- `0x6465660000000000000000000000000000000000000000000000000000000000`: the second part of the first parameter, a `bytes3` value `"def"` (left-aligned).

In total:

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000000064656600000000000000000000000
```

If we wanted to call `sam` with the arguments `"dave"`, `true` and `[1,2,3]`, we would pass 292 bytes total, broken down into:

- `0xa5643bf2`: the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])`. Note that `uint` is replaced with its canonical representation `uint256`.
- `0x0000000000000000000000000000000000000000000000000000000000000060`: the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60`.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the second parameter: boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000a0`: the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xa0`.
- `0x0000000000000000000000000000000000000000000000000000000000000004`: the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.
- `0x6461766500000000000000000000000000000000000000000000000000000000`: the contents of the first argument: the UTF-8 (equal to ASCII in this case) encoding of `"dave"`, padded on the right to 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001`: the first entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000002`: the second entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000003`: the third entry of the third parameter.

In total:

```
0xa5643bf20000000000000000000000000000000000000000000000000000000000000060000000000000000000000000000000
```

### 8.8.8 Use of Dynamic Types

A call to a function with the signature `f(uint,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way:

We take the first four bytes of `sha3("f(uint256,uint32[],bytes10,bytes)")`, i.e. `0x8be65246`. Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10`, these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes`, we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32 + 3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)
- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"`:

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case): 13)
- `0x48656c6c6f2c20776f726c6421000000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (newline after function selector and each 32-bytes for clarity):

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
  0000000000000000000000000000000000000000000000000000000000000080
  3132333435363738393000000000000000000000000000000000000000000000
  00000000000000000000000000000000000000000000000000000000000000e0
  0000000000000000000000000000000000000000000000000000000000000002
  0000000000000000000000000000000000000000000000000000000000000456
  0000000000000000000000000000000000000000000000000000000000000789
  000000000000000000000000000000000000000000000000000000000000000d
  48656c6c6f2c20776f726c6421000000000000000000000000000000000000000
```

### 8.8.9 Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3, are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which are not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as:

- `address`: the address of the contract (intrinsically provided by Ethereum);
- `topics[0]: keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo`, it would return `uint256`). If the event is declared as `anonymous` the `topics[0]` is not generated;
- `topics[n]`: `EVENT_INDEXED_ARGS[n - 1]` (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
- `data`: `abi_serialise(EVENT_NON_INDEXED_ARGS)` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_serialise` is the ABI serialisation function used for returning a series of typed values from a function, as described above).

For all fixed-length Solidity types, the `EVENT_INDEXED_ARGS` array contains the 32-byte encoded value directly. However, for *types of dynamic length*, which include `string`, `bytes`, and arrays, `EVENT_INDEXED_ARGS` will contain the *Keccak hash* of the encoded value, rather than the encoded value directly. This allows applications to efficiently query for values of dynamic-length types (by setting the hash of the encoded value as the topic), but leaves

applications unable to decode indexed values they have not queried for. For dynamic-length types, application developers face a trade-off between fast search for predetermined values (if the argument is indexed) and legibility of arbitrary values (which requires that the arguments not be indexed). Developers may overcome this tradeoff and achieve both efficient search and arbitrary legibility by defining events with two arguments — one indexed, one not — intended to hold the same value.

### 8.8.10 JSON

The JSON format for a contract's interface is given by an array of function and/or event descriptions. A function description is a JSON object with the fields:

- `type`: `"function"`, `"constructor"`, or `"fallback"` (the *unnamed "default" function*);
- `name`: the name of the function;
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter;
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
- `outputs`: an array of objects similar to `inputs`, can be omitted if function doesn't return anything;
- `payable`: `true` if function accepts ether, defaults to `false`;
- `stateMutability`: a string with one of the following values: `pure` (*specified to not read blockchain state*), `view` (*specified to not modify the blockchain state*), `nonpayable` and `payable` (same as `payable` above).
- `constant`: `true` if function is either `pure` or `view`

`type` can be omitted, defaulting to `"function"`.

Constructor and fallback function never have `name` or `outputs`. Fallback function doesn't have `inputs` either.

Sending non-zero ether to non-payable function will throw. Don't do it.

An event description is a JSON object with fairly similar fields:

- `type`: always `"event"`
- `name`: the name of the event;
- `inputs`: an array of objects, each of which contains:
  - `name`: the name of the parameter;
  - `type`: the canonical type of the parameter (more below).
  - `components`: used for tuple types (more below).
  - `indexed`: `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous`: `true` if the event was declared as `anonymous`.

For example,

```
pragma solidity ^0.4.0;

contract Test {
  function Test() public { b = 0x1234567890123456789012345678901234567890123456789012; }
  event Event(uint indexed a, bytes32 b);
  event Event2(uint indexed a, bytes32 b);
```

(continues on next page)

(이전 페이지에서 계속)

```
  function foo(uint a) public { Event(a, b); }
  bytes32 b;
}
```

would result in the JSON:

```
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

### Handling tuple types

Despite that names are intentionally not part of the ABI encoding they do make a lot of sense to be included in the JSON to enable displaying it to the end user. The structure is nested in the following way:

An object with members `name`, `type` and potentially `components` describes a typed variable. The canonical type is determined until a tuple type is reached and the string description up to that point is stored in `type` prefix with the word `tuple`, i.e. it will be `tuple` followed by a sequence of `[]` and `[k]` with integers `k`. The components of the tuple are then stored in the member `components`, which is of array type and has the same structure as the top-level object except that `indexed` is not allowed there.

As an example, the code

```
pragma solidity ^0.4.19;
pragma experimental ABIEncoderV2;

contract Test {
  struct S { uint a; uint[] b; T[] c; }
  struct T { uint x; uint y; }
  function f(S s, T t, uint a) public { }
  function g() public returns (S s, T t, uint a) {}
}
```

would result in the JSON:

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
```

(continues on next page)

(이전 페이지에서 계속)

```
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
              }
            ]
          }
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

### 8.8.11 Non-standard Packed Mode

Solidity supports a non-standard packed mode where:

- no *function selector* is encoded,
- types shorter than 32 bytes are neither zero padded nor sign extended and

- dynamic types are encoded in-place and without the length.

As an example encoding `int1, bytes1, uint16, string` with values `-1, 0x42, 0x2424, "Hello, world!"` results in

```
0xff42242448656c6c6f2c20776f726c6421
  ^^                                  int1(-1)
    ^^                                bytes1(0x42)
      ^^^^                            uint16(0x2424)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^ string("Hello, world!") without a length field
```

More specifically, each statically-sized type takes as many bytes as its range has and dynamically-sized types like `string`, `bytes` or `uint[]` are encoded without their length field. This means that the encoding is ambiguous as soon as there are two dynamically-sized elements.

## 8.9 Yul

Yul (previously also called JULIA or IULIA) is an intermediate language that can compile to various different backends (EVM 1.0, EVM 1.5 and eWASM are planned). Because of that, it is designed to be a usable common denominator of all three platforms. It can already be used for "inline assembly" inside Solidity and future versions of the Solidity compiler will even use Yul as intermediate language. It should also be easy to build high-level optimizer stages for Yul.

**주석:** Note that the flavour used for "inline assembly" does not have types (everything is `u256`) and the built-in functions are identical to the EVM opcodes. Please resort to the inline assembly documentation for details.

The core components of Yul are functions, blocks, variables, literals, for-loops, if-statements, switch-statements, expressions and assignments to variables.

Yul is typed, both variables and literals must specify the type with postfix notation. The supported types are `bool`, `u8`, `s8`, `u32`, `s32`, `u64`, `s64`, `u128`, `s128`, `u256` and `s256`.

Yul in itself does not even provide operators. If the EVM is targeted, opcodes will be available as built-in functions, but they can be reimplemented if the backend changes. For a list of mandatory built-in functions, see the section below.

The following example program assumes that the EVM opcodes `mul`, `div` and `mod` are available either natively or as functions and computes exponentiation.

```
{
    function power(base:u256, exponent:u256) -> result:u256
    {
        switch exponent
        case 0:u256 { result := 1:u256 }
        case 1:u256 { result := base }
        default
        {
            result := power(mul(base, base), div(exponent, 2:u256))
            switch mod(exponent, 2:u256)
                case 1:u256 { result := mul(base, result) }
        }
    }
}
```

It is also possible to implement the same function using a for-loop instead of with recursion. Here, we need the EVM opcodes `lt` (less-than) and `add` to be available.

```
{
    function power(base:u256, exponent:u256) -> result:u256
    {
        result := 1:u256
        for { let i := 0:u256 } lt(i, exponent) { i := add(i, 1:u256) }
        {
            result := mul(result, base)
        }
    }
}
```

### 8.9.1 Specification of Yul

This chapter describes Yul code. It is usually placed inside a Yul object, which is described in the following chapter.

Grammar:

```
Block = '{' Statement* '}'
Statement =
    Block |
    FunctionDefinition |
    VariableDeclaration |
    Assignment |
    If |
    Expression |
    Switch |
    ForLoop |
    BreakContinue
FunctionDefinition =
    'function' Identifier '(' TypedIdentifierList? ')'
    ( '->' TypedIdentifierList )? Block
VariableDeclaration =
    'let' TypedIdentifierList ( ':=' Expression )?
Assignment =
    IdentifierList ':=' Expression
Expression =
    FunctionCall | Identifier | Literal
If =
    'if' Expression Block
Switch =
    'switch' Expression ( Case+ Default? | Default )
Case =
    'case' Literal Block
Default =
    'default' Block
ForLoop =
    'for' Block Expression Block Block
BreakContinue =
    'break' | 'continue'
FunctionCall =
    Identifier '(' ( Expression ( ',' Expression )* )? ')'
Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*
IdentifierList = Identifier ( ',' Identifier)*
TypeName = Identifier | BuiltinTypeName
BuiltinTypeName = 'bool' | [us] ( '8' | '32' | '64' | '128' | '256' )
TypedIdentifierList = Identifier ':' TypeName ( ',' Identifier ':' TypeName )*
```

(continues on next page)

(이전 페이지에서 계속)

```
Literal =
    (NumberLiteral | StringLiteral | HexLiteral | TrueLiteral | FalseLiteral) ':'␣
↪TypeName
NumberLiteral = HexNumber | DecimalNumber
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
TrueLiteral = 'true'
FalseLiteral = 'false'
HexNumber = '0x' [0-9a-fA-F]+
DecimalNumber = [0-9]+
```

### Restrictions on the Grammar

Switches must have at least one case (including the default case). If all possible values of the expression is covered, the default case should not be allowed (i.e. a switch with a `bool` expression and having both a true and false case should not allow a default case).

Every expression evaluates to zero or more values. Identifiers and Literals evaluate to exactly one value and function calls evaluate to a number of values equal to the number of return values of the function called.

In variable declarations and assignments, the right-hand-side expression (if present) has to evaluate to a number of values equal to the number of variables on the left-hand-side. This is the only situation where an expression evaluating to more than one value is allowed.

Expressions that are also statements (i.e. at the block level) have to evaluate to zero values.

In all other situations, expressions have to evaluate to exactly one value.

The `continue` and `break` statements can only be used inside loop bodies and have to be in the same function as the loop (or both have to be at the top level). The condition part of the for-loop has to evaluate to exactly one value.

Literals cannot be larger than the their type. The largest type defined is 256-bit wide.

### Scoping Rules

Scopes in Yul are tied to Blocks (exceptions are functions and the for loop as explained below) and all declarations (`FunctionDefinition`, `VariableDeclaration`) introduce new identifiers into these scopes.

Identifiers are visible in the block they are defined in (including all sub-nodes and sub-blocks). As an exception, identifiers defined in the "init" part of the for-loop (the first block) are visible in all other parts of the for-loop (but not outside of the loop). Identifiers declared in the other parts of the for loop respect the regular syntatical scoping rules. The parameters and return parameters of functions are visible in the function body and their names cannot overlap.

Variables can only be referenced after their declaration. In particular, variables cannot be referenced in the right hand side of their own variable declaration. Functions can be referenced already before their declaration (if they are visible).

Shadowing is disallowed, i.e. you cannot declare an identifier at a point where another identifier with the same name is also visible, even if it is not accessible.

Inside functions, it is not possible to access a variable that was declared outside of that function.

### Formal Specification

We formally specify Yul by providing an evaluation function E overloaded on the various nodes of the AST. Any functions can have side effects, so E takes two state objects and the AST node and returns two new state objects and a variable number of other values. The two state objects are the global state object (which in the context of the EVM is

the memory, storage and state of the blockchain) and the local state object (the state of local variables, i.e. a segment of the stack in the EVM). If the AST node is a statement, E returns the two state objects and a "mode", which is used for the `break` and `continue` statements. If the AST node is an expression, E returns the two state objects and as many values as the expression evaluates to.

The exact nature of the global state is unspecified for this high level description. The local state `L` is a mapping of identifiers `i` to values `v`, denoted as `L[i] = v`.

For an identifier `v`, let `$v` be the name of the identifier.

We will use a destructuring notation for the AST nodes.

```
E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    let L2 be a restriction of L1 to the identifiers of L
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
            E(G1, L1, St2, ..., Stn)
        otherwise
            G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var1, ..., varn := rhs>: VariableDeclaration) =
    E(G, L, <var1, ..., varn := rhs>: Assignment)
E(G, L, <let var1, ..., varn>: VariableDeclaration) =
    let L1 be a copy of L where L1[$vari] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var1, ..., varn := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$vari] = vi for i = 1, ..., n
    G, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L1, mode = E(G, L, i1, ..., in)
        // mode has to be regular due to the syntactic restrictions
        let G2, L2, mode = E(G1, L1, for {} condition post body)
        // mode has to be regular due to the syntactic restrictions
        let L3 be the restriction of L2 to only variables of L
        G2, L3, regular
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            else:
                G3, L3, mode = E(G2, L2, post)
                E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
    G, L, continue
```

(continues on next page)

(이전 페이지에서 계속)

```
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // Evaluate literals, context doesn't matter
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    if there exists smallest i such that vi = v:
        E(G0, L0, sti)
    else:
        E(G0, L0, st')

E(G, L, <name>: Identifier) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
    ...
    G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
    Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
    Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
    be the function of name $fname visible at the point of the call.
    Let L' be a new local state such that
    L'[$parami] = vi and L'[$reti] = 0 for all i.
    Let G'', L'', mode = E(Gn, L', block)
    G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: HexLiteral) = G, L, hexString(l),
    where hexString decodes l from hex and left-aligns it into 32 bytes
E(G, L, l: StringLiteral) = G, L, utf8EncodeLeftAligned(l),
    where utf8EncodeLeftAligned performs a utf8 encoding of l
    and aligns it left into 32 bytes
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal decoding function
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal decoding function
```

### Type Conversion Functions

Yul has no support for implicit type conversion and therefore functions exist to provide explicit conversion. When converting a larger type to a shorter type a runtime exception can occur in case of an overflow.

**Truncating conversions are supported between the following types:**

- `bool`
- `u32`
- `u64`
- `u256`

- `s256`

For each of these a type conversion function exists having the prototype in the form of `<input_type>to<output_type>(x:<input_type>) -> y:<output_type>`, such as `u32tobool(x:u32) -> y:bool`, `u256tou32(x:u256) -> y:u32` or `s256tou256(x:s256) -> y:u256`.

---

**주석:** `u32tobool(x:u32) -> y:bool` can be implemented as `y := not(iszerou256(x))` and `booltou32(x:bool) -> y:u32` can be implemented as `switch x case true:bool { y := 1:u32 } case false:bool { y := 0:u32 }`

---

### Low-level Functions

The following functions must be available:

| | |
|---|---|
| *Logic* | |
| not(x:bool) -> z:bool | logical not |
| and(x:bool, y:bool) -> z:bool | logical and |
| or(x:bool, y:bool) -> z:bool | logical or |
| xor(x:bool, y:bool) -> z:bool | xor |
| *Arithmetic* | |
| addu256(x:u256, y:u256) -> z:u256 | x + y |
| subu256(x:u256, y:u256) -> z:u256 | x - y |
| mulu256(x:u256, y:u256) -> z:u256 | x * y |
| divu256(x:u256, y:u256) -> z:u256 | x / y |
| divs256(x:s256, y:s256) -> z:s256 | x / y, for signed numbers in two's comple |
| modu256(x:u256, y:u256) -> z:u256 | x % y |
| mods256(x:s256, y:s256) -> z:s256 | x % y, for signed numbers in two's compl |
| signextendu256(i:u256, x:u256) -> z:u256 | sign extend from (i*8+7)th bit counting fr |
| expu256(x:u256, y:u256) -> z:u256 | x to the power of y |
| addmodu256(x:u256, y:u256, m:u256) -> z:u256 | (x + y) % m with arbitrary precision arithr |
| mulmodu256(x:u256, y:u256, m:u256) -> z:u256 | (x * y) % m with arbitrary precision arithr |
| ltu256(x:u256, y:u256) -> z:bool | true if x < y, false otherwise |
| gtu256(x:u256, y:u256) -> z:bool | true if x > y, false otherwise |
| sltu256(x:s256, y:s256) -> z:bool | true if x < y, false otherwise (for signed nu |
| sgtu256(x:s256, y:s256) -> z:bool | true if x > y, false otherwise (for signed nu |
| equ256(x:u256, y:u256) -> z:bool | true if x == y, false otherwise |
| iszerou256(x:u256) -> z:bool | true if x == 0, false otherwise |
| notu256(x:u256) -> z:u256 | ~x, every bit of x is negated |
| andu256(x:u256, y:u256) -> z:u256 | bitwise and of x and y |
| oru256(x:u256, y:u256) -> z:u256 | bitwise or of x and y |
| xoru256(x:u256, y:u256) -> z:u256 | bitwise xor of x and y |
| shlu256(x:u256, y:u256) -> z:u256 | logical left shift of x by y |
| shru256(x:u256, y:u256) -> z:u256 | logical right shift of x by y |
| saru256(x:u256, y:u256) -> z:u256 | arithmetic right shift of x by y |
| byte(n:u256, x:u256) -> v:u256 | nth byte of x, where the most significant b |
| *Memory and storage* | |
| mload(p:u256) -> v:u256 | mem[p..(p+32)) |
| mstore(p:u256, v:u256) | mem[p..(p+32)) := v |
| mstore8(p:u256, v:u256) | mem[p] := v & 0xff - only modifies a sing |

| | |
|---|---|
| sload(p:u256) -> v:u256 | storage[p] |
| sstore(p:u256, v:u256) | storage[p] := v |
| msize() -> size:u256 | size of memory, i.e. largest accessed mem |
| *Execution control* | |
| create(v:u256, p:u256, n:u256) | create new contract with code mem[p..(p+ |
| create2(v:u256, p:u256, n:u256, s:u256) | create new contract with code mem[p...(p- |
| call(g:u256, a:u256, v:u256, in:u256, insize:u256, out:u256, outsize:u256) -> r:u256 | call contract at address a with input mem[ |
| callcode(g:u256, a:u256, v:u256, in:u256, insize:u256, out:u256, outsize:u256) -> r:u256 | identical to `call` but only use the code fr |
| delegatecall(g:u256, a:u256, in:u256, insize:u256, out:u256, outsize:u256) -> r:u256 | identical to `callcode`, but also keep `ca` |
| abort() | abort (equals to invalid instruction on EVI |
| return(p:u256, s:u256) | end execution, return data mem[p..(p+s)) |
| revert(p:u256, s:u256) | end execution, revert state changes, return |
| selfdestruct(a:u256) | end execution, destroy current contract an |
| log0(p:u256, s:u256) | log without topics and data mem[p..(p+s)) |
| log1(p:u256, s:u256, t1:u256) | log with topic t1 and data mem[p..(p+s)) |
| log2(p:u256, s:u256, t1:u256, t2:u256) | log with topics t1, t2 and data mem[p..(p+ |
| log3(p:u256, s:u256, t1:u256, t2:u256, t3:u256) | log with topics t, t2, t3 and data mem[p..( |
| log4(p:u256, s:u256, t1:u256, t2:u256, t3:u256, t4:u256) | log with topics t1, t2, t3, t4 and data mem |
| *State queries* | |
| blockcoinbase() -> address:u256 | current mining beneficiary |
| blockdifficulty() -> difficulty:u256 | difficulty of the current block |
| blockgaslimit() -> limit:u256 | block gas limit of the current block |
| blockhash(b:u256) -> hash:u256 | hash of block nr b - only for last 256 bloc |
| blocknumber() -> block:u256 | current block number |
| blocktimestamp() -> timestamp:u256 | timestamp of the current block in seconds |
| txorigin() -> address:u256 | transaction sender |
| txgasprice() -> price:u256 | gas price of the transaction |
| gasleft() -> gas:u256 | gas still available to execution |
| balance(a:u256) -> v:u256 | wei balance at address a |
| this() -> address:u256 | address of the current contract / execution |
| caller() -> address:u256 | call sender (excluding delegatecall) |
| callvalue() -> v:u256 | wei sent together with the current call |
| calldataload(p:u256) -> v:u256 | call data starting from position p (32 bytes |
| calldatasize() -> v:u256 | size of call data in bytes |
| calldatacopy(t:u256, f:u256, s:u256) | copy s bytes from calldata at position f to |
| codesize() -> size:u256 | size of the code of the current contract / ex |
| codecopy(t:u256, f:u256, s:u256) | copy s bytes from code at position f to me |
| extcodesize(a:u256) -> size:u256 | size of the code at address a |
| extcodecopy(a:u256, t:u256, f:u256, s:u256) | like codecopy(t, f, s) but take code at addr |
| extcodehash(a:u256) | code hash of address a |
| *Others* | |
| discard(unused:bool) | discard value |
| discardu256(unused:u256) | discard value |
| splitu256tou64(x:u256) -> (x1:u64, x2:u64, x3:u64, x4:u64) | split u256 to four u64's |
| combineu64tou256(x1:u64, x2:u64, x3:u64, x4:u64) -> (x:u256) | combine four u64's into a single u256 |
| keccak256(p:u256, s:u256) -> v:u256 | keccak(mem[p...(p+s))) |

### Backends

Backends or targets are the translators from Yul to a specific bytecode. Each of the backends can expose functions prefixed with the name of the backend. We reserve `evm_` and `ewasm_` prefixes for the two proposed backends.

#### Backend: EVM

The EVM target will have all the underlying EVM opcodes exposed with the *evm_* prefix.

#### Backend: "EVM 1.5"

TBD

#### Backend: eWASM

TBD

### 8.9.2 Specification of Yul Object

Grammar:

```
TopLevelObject = 'object' '{' Code? ( Object | Data )* '}'
Object = 'object' StringLiteral '{' Code? ( Object | Data )* '}'
Code = 'code' Block
Data = 'data' StringLiteral HexLiteral
HexLiteral = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
StringLiteral = '"' ([^"\r\n\\] | '\\' .)* '"'
```

Above, `Block` refers to `Block` in the Yul code grammar explained in the previous chapter.

An example Yul Object is shown below:

```
// Code consists of a single object. A single "code" node is the code of the object.
// Every (other) named object or data section is serialized and
// made accessible to the special built-in functions datacopy / dataoffset / datasize
object {
    code {
        let size = datasize("runtime")
        let offset = allocate(size)
        // This will turn into a memory->memory copy for eWASM and
        // a codecopy for EVM
        datacopy(dataoffset("runtime"), offset, size)
        // this is a constructor and the runtime code is returned
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "runtime" {
        code {
            // runtime code

            let size = datasize("Contract2")
            let offset = allocate(size)
            // This will turn into a memory->memory copy for eWASM and
            // a codecopy for EVM
            datacopy(dataoffset("Contract2"), offset, size)
            // constructor parameter is a single number 0x1234
            mstore(add(offset, size), 0x1234)
            create(offset, add(size, 32))
```

(continues on next page)

(이전 페이지에서 계속)

```
        }

        // Embedded object. Use case is that the outside is a factory contract,
        // and Contract2 is the code to be created by the factory
        object "Contract2" {
            code {
                // code here ...
            }

            object "runtime" {
                code {
                    // code here ...
                }
             }

             data "Table1" hex"4123"
        }
    }
}
```

## 8.10 스타일 가이드

### 8.10.1 소개

이 가이드는 Solidity 코드를 작성할 때 지켜야 할 코딩 규칙에 대해 설명합니다. 좀더 나은 규칙이 발견되면 기존의 규칙은 삭제될 수 있고 시간이 지나면서 가이드는 바뀔 수 있습니다.

많은 프로젝트들이 각각의 스타일 가이드를 만들 것입니다. 충돌하는 지점이 발생하면, 프로젝트의 스타일 가이드를 우선적으로 따릅니다.

이 가이드 권장사항 규격은 파이썬의 pep8 style guide 를 따왔습니다.

가이드의 목표는 Solidity 코드를 작성할 때 최고의 방법을 알려드리는 것이 아닙니다. 가이드의 목표는 **일관성** 을 유지하는 것입니다. 파이썬의 pep8 인용구는 이 철학을 잘 소개하고 있습니다.

> 스타일 가이드는 일관성에 관한 것입니다. 일관성있는 스타일 가이드는 매우 중요합니다. 일관성있는 프로젝트, 모듈, 함수 또한 매우 중요합니다. 하지만 제일 중요한 것은 일관성을 가지지 않는 경우가 언제인지를 아는 것입니다. 때때로 스타일 가이드가 적용되지 않을 때가 있습니다. 의심을 갖고, 최선의 판단을 내리십시오. 다른 예를 보고 제일 나은 방향을 결정하십시오. 그리고 물어보는 걸 주저하지 마세요!

### 8.10.2 코드 레이아웃

#### 들여쓰기

들여쓰기마다 4 스페이스를 사용합니다.

#### 탭 or 스페이스

스페이스를 선호합니다.

탭과 스페이스를 섞어 사용하는 것을 피하세요.

### 빈 줄

상위 선언 괄호는 2개의 빈 줄을 사이로 둡니다.

Yes:

```
contract A {
    ...
}


contract B {
    ...
}


contract C {
    ...
}
```

No:

```
contract A {
    ...
}
contract B {
    ...
}

contract C {
    ...
}
```

컨트랙트 내의 선언 괄호는 1개의 빈 줄을 사이로 둡니다.

함수 스텁이나 추상 컨트랙트를 만드는 등 1줄 코드 사이에는 빈 줄을 생략합니다.

Yes:

```
contract A {
    function spam() public;
    function ham() public;
}


contract B is A {
    function spam() public {
        ...
    }

    function ham() public {
        ...
    }
}
```

No:

```
contract A {
    function spam() public {
```

(continues on next page)

(이전 페이지에서 계속)

```
        ...
    }
    function ham() public {
        ...
    }
}
```

### 최대 행 길이

PEP 8 recommendation of 79 (or 99) 줄의 내용들을 숙지하시면 도움이 될 것입니다.

줄바꿈은 다음과 같은 가이드라인을 따르세요.

1. 첫 번째 인자는 여는 괄호 옆에 붙이지 않습니다.
2. 한 번의 들여쓰기를 사용합니다.
3. 각각의 인자는 각 줄마다 위치합니다.
4. 끝을 내는 `);` 는 마지막 줄에 혼자 남겨질 수 있게 하세요.

함수 호출

Yes:

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

No:

```
thisFunctionCallIsReallyLong(longArgument1,
                             longArgument2,
                             longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3);
```

값 할당

Yes:

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

No:

```
thisIsALongNestedMapping[being][set][to_some_value] = someFunction(argument1,
                                                                   argument2,
                                                                   argument3,
                                                                   argument4);
```

이벤트 정의와 구현

Yes:

```
event LongAndLotsOfArgs(
    adress sender,
    adress recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

No:

```
event LongAndLotsOfArgs(adress sender,
                        adress recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

### 소스 파일 인코딩

UTF-8 이나 ASCII 인코딩을 선호합니다.

### 임포트

import 문법은 항상 파일 최상단에 위치합니다.

Yes:

```
import "owned";


contract A {
    ...
}


contract B is owned {
    ...
}
```

No:

```
contract A {
    ...
}


import "owned";


contract B is owned {
    ...
}
```

### 함수 순서

순서를 지키면 호출 가능한 함수들이 무엇인지, 생성자나 실패 예외처리가 어디있는지를 쉽게 찾을 수 있게 도움을 줍니다.

함수들은 다음과 같은 순서로 묶습니다.

- 생성자
- (있다면) fallback function
- external
- public
- internal
- private

그룹 내에서는 `constant` 함수는 마지막에 두세요.

Yes:

```
contract A {
    function A() public {
        ...
    }
```

(continues on next page)

(이전 페이지에서 계속)

```
    function() public {
        ...
    }

    // External functions
    // ...

    // External functions that are constant
    // ...

    // Public functions
    // ...

    // Internal functions
    // ...

    // Private functions
    // ...
}
```

No:

```
contract A {

    // External functions
    // ...

    // Private functions
    // ...

    // Public functions
    // ...

    function A() public {
        ...
    }

    function() public {
        ...
    }

    // Internal functions
    // ...
}
```

### 표현식 내 공백

다음과 같은 상황일 때 여분의 공백을 두는 걸 피하세요: 소, 중, 대괄호 안 바로 옆에 붙은 경우. 단, 한 줄의 함수 선언은 예외

Yes:

```
spam(ham[1], Coin({name: "ham"}));
```

No:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

Exception:

```
function singleLine() public { spam(); }
```

콤마나 세미콜론 바로 전에 붙은 경우:

Yes:

```
function spam(uint i, Coin coin) public;
```

No:

```
function spam(uint i , Coin coin) public ;
```

값을 할당할 때나 다른 연산자와 정렬을 맞추려는 공백이 1개를 넘는 경우:

Yes:

```
x = 1;
y = 2;
long_variable = 3;
```

No:

```
x             = 1;
y             = 2;
long_variable = 3;
```

fallback 함수 안에 공백을 넣지 마세요:

Yes:

```
function() public {
    ...
}
```

No:

```
function () public {
    ...
}
```

### 구조 잡기

컨트랙트, 라이브러리, 함수나 구조체의 내용을 담는 중괄호는 다음을 따라야 합니다:

- 선언부의 같은 줄에 괄호를 여세요
- 선언부와 같은 들여쓰기면서 고유한 마지막 한 줄에 괄호를 닫으세요
- 여는 괄호는 한 칸의 공백이 들어갑니다

Yes:

```
contract Coin {
    struct Bank {
        address owner;
        uint balance;
    }
}
```

No:

```
contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

같은 권장 형태가 `if`, `else`, `while` 그리고 `for` 구문을 구성할때도 쓰입니다. 특히 `if`, `while`, `for` 에서 소괄호로 쌓인 조건부의 경우 앞뒤로 공백 한 칸씩을 추가해야 합니다.

Yes:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

No:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

*한 줄* 과 하나의 구현체만 가지고 있는 경우엔 괄호를 생략할 수 있습니다.

Yes:

```
if (x < 10)
    x += 1;
```

No:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

`else` 나 `else if` 구문을 가진 `if` 문의 경우 `if` 문의 닫는 괄호와 같은 줄에 `else` 문이 위치해야 합니다. 이것은 다른 블록 형태의 구조 규칙과는 예외입니다.

Yes:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

No:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

### 함수 선언

짧은 함수 선언의 경우 함수 내용부의 여는 괄호는 선언부와 같은 줄에 위치합니다. 닫는 괄호는 함수 선언부와 같은 들여쓰기를 가집니다. 여는 괄호는 한 칸의 공백 뒤에 위치합니다.

Yes:

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyowner returns (uint) {
    return x + 1;
}
```

No:

```
function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
    }
```

(continues on next page)

(이전 페이지에서 계속)

```
function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

생성자를 포함한 모든 함수에 대해 접근 제어자를 명시적으로 적습니다.

Yes:

```
function explicitlyPublic(uint val) public {
    doSomething();
}
```

No:

```
function implicitlyPublic(uint val) {
    doSomething();
}
```

함수의 접근 제어자는 커스텀 키워드보다 앞에 옵니다.

Yes:

```
function kill() public onlyowner {
    selfdestruct(owner);
}
```

No:

```
function kill() onlyowner public {
    selfdestruct(owner);
}
```

함수가 많은 인자를 갖고 있을 땐, 각 인자마다 같은 들여쓰기로 한 줄씩 작성하는 것을 권장합니다. 열고 닫는 중괄호 또한 같은 규칙으로 한 줄씩 작성합니다.

Yes:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
{
    doSomething();
}
```

No:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}
```

(continues on next page)

(이전 페이지에서 계속)

```
function thisFunctionHasLotsOfArguments(address a,
                                        address b,
                                        address c,
                                        address d,
                                        address e,
                                        address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

제어자를 가진 긴 함수의 경우 각 제어자는 각각의 줄을 가지며 작성합니다.

Yes:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z,
)
    public
    onlyowner
    priced
    returns (address)
{
    doSomething();
}
```

No:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyowner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyowner priced returns (address)
```

(continues on next page)

(이전 페이지에서 계속)

```
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyowner
    priced
    returns (address) {
    doSomething();
}
```

많은 줄의 출력 파라미터와 리턴문은 *최대 행 길이* 섹션에서 언급된 긴 줄을 묶는 권장사항을 따라야 합니다.

Yes:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
}
```

No:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
             uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

인자가 필요한 상속 생성자의 경우, 선언부가 길거나 읽기 어려울 때 제어자와 같은 방식으로 새 줄에 기본 생성자를 떨어뜨려 놓는 것을 권장합니다.

Yes:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
        public
    {
        // do something with param5
    }
}
```

No:

```
contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
    B(param1)
    C(param2, param3)
    D(param4)
    public
    {
        // do something with param5
    }
}

contract A is B, C, D {
    function A(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
        public {
        // do something with param5
    }
}
```

단일문의 짧은 함수를 선언할 때 한 줄에 모두 작성할 수 있습니다.

허용:

```
function shortFunction() public { doSomething(); }
```

함수 선언에 대한 이 가이드라인들은 가독성을 높이기 위함입니다. 이 가이드는 함수 선언의 모든 경우를 다루지는 않으므로 작성자는 최선의 판단을 내려야 합니다.

### 매핑

TODO

### 변수 선언

배열 변수를 선언할 때, 타입과 중괄호 사이 공백을 두지 않습니다.

Yes:

```
uint[] x;
```

No:

```
uint [] x;
```

#### 기타 권장사항

- 문자열은 작은 따옴표보다 큰 따옴표를 사용합니다.

Yes:

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

No:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- 연산자 사이에 공백을 둡니다.

Yes:

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

No:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- 우선순위를 나타내기 위해 우선순위가 더 높은 연산자는 공백을 제외할 수 있습니다. 이로 인해 가독성을 늘릴 수 있으며, 연산자 양쪽에는 같은 공백을 사용해야 합니다.

Yes:

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

No:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

### 8.10.3 명명 규칙

명명 규칙은 넓게 사용될 때 더욱 강력합니다. 이를 통해 중요하고 많은 *메타* 정보들을 전달할 수 있습니다. 이 권장사항은 가독성을 높이기 위한 것으로, 규칙이 아니라 사물의 이름을 통해 정보를 전달하기 위함입니다. 마지막으로 코드의 일관성은 이 문서의 규칙들로 항상 대체할 수 있어야 합니다.

### 명명 스타일

혼동을 줄이기 위한 예로, 다음 이름들은 모두 다른 명명 스타일을 가지고 있습니다.

- `b` (소문자 한 글자)
- `B` (대문자 한 글자)
- `lowercase`
- `lower_case_with_underscores`
- `UPPERCASE`
- `UPPER_CASE_WITH_UNDERSCORES`
- `CapitalizedWords` (or CapWords)
- `mixedCase` (첫 글자가 소문자로 CapitalizedWords와 다르다!)
- `Capitalized_Words_With_Underscores`

**주석:** CapWords 스타일에서 이니셜을 사용할 땐 이니셜 모두를 대문자로 씁니다. 즉, HttpServerError 대신 HTTPServerError를 사용하는 것입니다. mixedCase 스타일에서 이니셜을 사용할 땐 이니셜을 대문자로 쓰되, 이름의 처음으로 쓰일 땐 소문자를 사용합니다. 즉, XMLHTTPRequest 보다 xmlHTTPRequest이 낫습니다.

### 피해야 할 명명법

- `l` - 문자열 el 의 소문자
- `O` - 문자열 oh 의 대문자
- `I` - 문자열 eye의 대문자

한 글자 변수 이름은 사용하지 마세요. 종종 숫자 1과 0과도 구별하기 어려울 때가 있습니다.

### 컨트랙트와 라이브러리 이름

컨트랙트와 라이브러리는 CapWords 스타일을 사용합니다. 예: `SimpleToken`, `SmartBank`, `CertificateHashRepository`, `Player`.

### 구조체 이름

구조체는 CapWords 스타일을 사용합니다. 예: `MyCoin`, `Position`, `PositionXY`.

### 이벤트 이름

이벤트는 CapWords 스타일을 사용합니다. 예: `Deposit`, `Transfer`, `Approval`, `BeforeTransfer`, `AfterTransfer`.

### 함수 이름

생성자가 아닌 함수는 mixedCase 스타일을 사용합니다. 예: `getBalance`, `transfer`, `verifyOwner`, `addMember`, `changeOwner`.

#### 함수 인자 이름

함수 인자는 mixedCase 스타일을 사용합니다. 예: `initialSupply`, `account`, `recipientAddress`, `senderAddress`, `newOwner`.

커스텀 구조체에서 동작하는 라이브러리 함수를 작성할 때, 구조체는 함수의 첫 번째 인자여야 하며 `self` 라는 이름을 가집니다.

#### 지역, 상태 변수 이름

mixedCase 스타일을 사용합니다. 예: `totalSupply`, `remainingSupply`, `balancesOf`, `creatorAddress`, `isPreSale`, `tokenExchangeRate`.

#### 상수

상수는 밑줄과 함께 대문자를 사용합니다. 예: `MAX_BLOCKS`, `TOKEN_NAME`, `TOKEN_TICKER`, `CONTRACT_VERSION`.

#### 제어자 이름

mixedCase 스타일을 사용합니다. 예: `onlyBy`, `onlyAfter`, `onlyDuringThePreSale`.

#### 열거형

열거형은 CapWords 스타일을 사용합니다. 예: `TokenGroup`, `Frame`, `HashStyle`, `CharacterLocation`.

#### 명명 충돌의 방지

- `single_trailing_underscore_`

이 규칙은 원하는 이름이 예약어와 충돌할 때 제안됩니다.

#### 일반 권장사항

TODO

## 8.11 자주 쓰이는 패턴

### 8.11.1 컨트랙트에서의 출금

Effect 이후 기금송금에 있어 가장 권장되는 방법은 출금 패턴을 사용하는 것입니다. Effect의 결과로 Ether를 송금하는 가장 직관적인 방법은 직접 `transfer` 를 호출하는 것이겠지만, 잠재적인 보안위협을 초래 할 수 있으므로 권장하지 않습니다. security_consideration 페이지에서 보안에 대해 더 알아 볼 수 있습니다.

다음은 King of the Ether 에서 영감을 받아 작성된 "richest"가 되기 위해 가장 많은 돈을 컨트랙트로 송금하는 실제 출금패턴 예제입니다.

다음의 컨트랙트에서 당신이 "richest"를 빼앗긴다면, 새롭게 "richest" 가 된 사람으로부터 기금을 돌려 받을 것입니다.

```
pragma solidity >0.4.99 <0.6.0

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping (address => uint) pendingWithdrawals;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable returns (bool) {
        if (msg.value > mostSent) {
            pendingWithdrawals[richest] += msg.value;
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // 리엔트란시(re-entrancy) 공격을 예방하기 위해
        // 송금하기 전에 보류중인 환불을 0으로 기억해 두십시오.
        pendingWithdrawals[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}
```

다음은 직관적인 송금패턴과 정반대인 패턴입니다.

```
pragma solidity >0.4.99 <0.6.0;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    constructor() public payable {
        richest = msg.sender;
        mostSent = msg.value;
    }

    function becomeRichest() public payable returns (bool) {
        if (msg.value > mostSent) {
            // 현재의 라인이 문제의 원인이 될 수 있습니다. (아래에서 설명됨)
            richest.transfer(msg.value);
            richest = msg.sender;
            mostSent = msg.value;
            return true;
        } else {
            return false;
        }
    }
}
```

본 예제에서, 반드시 알아둬야 할 것은, 공격자가 실패 fallback함수를 가진 컨트랙트 주소를 `richest` 로 만들어 컨트랙트를 할 수 없는 상태로 만들 수 있다는 점입니다. (예를들어 `revert()` 를 사용하거나 단순히 2300 이상의 가스를 그들에게 전송 시켜 소비함으로써) 그렇게하면, "poisoned" 컨트랙트에 기금을 `transfer` 하기위해 송금이 요청 될 때마다 컨트랙트와 더불어 `becomeRichest` 도 실패 할 것이고, 이는 컨트랙트가 영원히 진행되지 않게 만들 것입니다.

반대로, 첫번째 예제에서 "withdraw" 패턴을 사용한다면 공격자의 출금만 실패할 것이고, 나머지 컨트랙트는 제대로 동작 할 것입니다.

### 8.11.2 제한된 액세스

제한된 액세스는 컨트랙트에서의 일반적인 패턴입니다. 알아둬야 할 것은, 다른사람이나 컴퓨터가 당신의 컨트랙트 상태의 내용을 읽는것을 결코 제한 할 수 없다는 것입니다. 암호화를 사용함으로써 컨트랙트를 더 읽기 어렵게 만들 수 있습니다. 하지만 컨트랙트가 데이터를 읽으려고 한다면, 다른 모든 사람들 또한 당신의 데이터를 읽을 수 있을것입니다.

**다른 컨트랙트들** 이 컨트랙트 상태를 읽지 못 하도록 권한을 제한 할 수 있습니다. 상태변수를 `public` 으로 선언하지 않는 한, 이 제한은 디폴트로 제공됩니다.

게다가, 컨트랙트 상태를 수정하거나 컨트랙트 함수를 호출 할 수 있는 사람을 제한 할 수 있습니다. 다음이 바로 본 섹션에 대한 내용입니다.

**function modifiers** 를 사용하면 이런 제한을 매우 알아보기 쉽게 할 수 있습니다.

```
pragma solidity >=0.4.22 <0.6.0;

contract AccessRestriction {
    // 이것들은 건설단계에서 할당됩니다.
    // 여기서, `msg.sender` 는
    // 이 계약을 생성하는 계정입니다.
    address public owner = msg.sender;
    uint public creationTime = now;

    // 수정자를 사용하여 함수의
    // 본문을 변경할 수 있습니다.
    // 이 수정자가 사용되면,
    // 함수가 특정 주소에서 호출 된
    // 경우에만 통과하는 검사가
    // 추가됩니다.
    modifier onlyBy(address _account)
    {
        require(
            msg.sender == _account,
            "Sender not authorized."
        );
        // "_;" 를 깜빡하지 마세요! 수정자가
        // 사용 될 때, "_;"가 실제 함수
        // 본문으로 대체됩니다.
        _;
    }

    /// `_newOwner` 를 이 컨트랙트의
    /// 새 소유자로 만듭니다.
    function changeOwner(address _newOwner)
        public
        onlyBy(owner)
    {
```

(continues on next page)

(이전 페이지에서 계속)

```
        owner = _newOwner;
    }

    modifier onlyAfter(uint _time) {
        require(
            now >= _time,
            "Function called too early."
        );
        _;
    }

    /// 소유권 정보를 지우십시오.
    /// 컨트랙트가 생성된 후 6주가
    /// 지나야 호출 될 수 있습니다.
    function disown()
        public
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // 이 수정자는 함수 호출과 관련된
    // 특정 요금을 요구합니다.
    // 호출자가 너무 많은 금액을 송금했을시,
    // 함수 본문 이후에만 환급이 됩니다.
    // 이는 Solidity 0.4.0 이전의 버전에서는 위험하였습니다.
    // `_;` 이후의 부분은 스킵될 가능성이 있습니다.
    modifier costs(uint _amount) {
        require(
            msg.value >= _amount,
            "Not enough Ether provided."
        );
        _;
        if (msg.value > _amount)
            msg.sender.transfer(msg.value - _amount);
    }

    function forceOwnerChange(address _newOwner)
        public
        payable
        costs(200 ether)
    {
        owner = _newOwner;
        // 몇 가지의 예제 조건
        if (uint(owner) & 0 == 1)
            // 이는 Solidity 0.4.0 이전의
            // 버전에서는 환불되지 않았습니다.
            return;
        // 초과 요금에 대한 환불
    }
}
```

함수호출에 대한 액세스를 제한 할 수 있는 보다 특별한 방법에 대해서는 다음 예제에서 설명합니다.

### 8.11.3 상태 머신

컨트랙트는 종종 상태머신인 것처럼 동작합니다. 다시 말해, 컨트랙트들이 다르게 동작하거나 다른 함수들에 의해 호출되는 어떠한 **단계** 를 가지고 있습니다. 함수 호출은 종종 단계를 끝내고 컨트랙트를 다음 단계로 전환 시킵니다. (특히 컨트랙트 모델이 **상호작용** 인 경우에) 또한, **시간** 의 특정 지점에서 일부 단계에 자동으로 도달 하는 것이 일반적입니다.

예를 들어 "블라인드 입찰을 수락하는" 단계에서 시작하여 "옥션 결과 결정"으로 끝나는 "공개 입찰"로 전환하는 블라인드 옥션 컨트랙트가 있습니다.

이 상황에서 상태를 모델링하고 컨트랙트의 잘못된 사용을 방지하기 위해 함수 수정자를 사용할 수 있습니다.

#### 예제

다음의 예제에서, 수정자 `atStage` 는 함수가 특정 단계에서만 호출되도록 보장해 줍니다.

자동 timed transitions 는 모든 함수에서 사용되는 수정자 `timeTransitions` 에 의해 처리됩니다.

마지막으로, 수정자 `transitionNext` 는 함수가 끝났을 때, 자동적으로 다음 단계로 넘어가도록 하기 위해 사용될 수 있습니다.

```
pragma solidity >=0.4.22 <0.6.0;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }

    // 이 부분이 현재의 단계입니다.
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(
            stage == _stage,
            "Function cannot be called at this time."
        );
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // timed transitions를 수행하십시오.
    // 이 수정자를 먼저 언급해야 합니다. 그렇지 않으면,
    // Guards가 새로운 단계를 고려하지 않을 것 입니다.
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    now >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
                now >= creationTime + 12 days)
```

(continues on next page)

(이전 페이지에서 계속)

```
            nextStage();
        // 다른 단계는 거래에 의해 전환됩니다.
        _;
    }

    // 수정자의 순서가 중요합니다!
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // 우리는 여기서 그것을 구현하지 않을 것입니다.
    }

    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // 이 수정자는 함수가 완료된 후
    // 다음 단계로 이동합니다.
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
        transitionNext
    {
    }

    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

## 8.12 알려진 버그 리스트

아래를 확인해보면, Solidity 컴파일러에서 알려진 보안 관련 버그의 JSON 포맷형식 리스트를 찾을 수 있습니다. 파일 자체는 Github repository 에서 호스팅되었습니다. 이 리스트는 0.3.3 버전까지 거슬러 올라가며, 그 이전의 버전에서 존재했으나 지금은 사라진 알려진 버그들은 리스트에 넣지 않았습니다.

bugs_by_version.json 라고 불리는 다른 파일이 존재하며, 이는 특정 컴파일러 버전에 영향을 주는 버그를 확인하는데 사용할 수 있습니다.

컨트랙트 소스 확인 툴과 컨트랙트와 상호작용하는 툴들은 다음 기준에 따라 다음 리스트를 참조해야 합니다.

- 컨트랙트가 릴리즈된 버전이 아닌 알파테스트 컴파일러 버전(Nightly compiler version) 으로 컴파일 된 경우 다소 의심스럽습니다. 이 리스트는 릴리즈 되지 않았거나 알파테스트 버전을 트래킹하지 않습니다.
- 또한 컨트랙트를 작성한 시점에서 가장 최신 버전이 아닌 컴파일러로 컴파일 되었다면 다소 의심스럽습니다. 다른 컨트랙트들에 의해 만들어진 컨트랙트의 경우, 트랜젝션으로 돌아가 생성체인을 따르고 그 트렌젝션의 날짜를 생성 날짜로써 사용해야 합니다.
- 알려진 버그를 포함하고 있는 컴파일러로 컨트랙트를 컴파일 했을 경우, 그리고 수정본이 포함된 최신 컴파일러 버전이 이미 릴리즈 된 시점에서 컨트랙트가 만들어졌을 경우, 매우 의심스럽습니다.

아래에 나열된 알려진 버그가 있는 JSON 파일은 각 버그마다 하나씩 다음 키가 있는 객체 배열입니다.

**이름** 버그에 부여된 고유한 이름

**요약** 버그에 대한 짧은 설명

**설명** 버그에 대한 자세한 설명

**링크** 더 자세한 정보가 있는 웹사이트의 URL, 선택사항

**도입성** 버그가 포함된 최초의 컴파일러 버전, 선택사항

**고정된(fixed)** 어떠한 버그도 포함되지 않은 최초의 컴파일러 버전

**퍼블리쉬** 버그가 공개적으로 알려지게 된 날짜, 선택사항

**심각도** 버그의 심각도: 매우 낮음, 낮음, 보통, 높음. 컨트랙트 테스트에서의 발견가능성과 발생 확률, 악용으로 인한 잠재적 손상들을 고려합니다.

**조건** 버그를 유발하는 조건. 현재 이 객체는 부울값 `optimizer` 를 포함 할 수 있습니다. 즉, 버그를 활성화하려면 optimizer를 켜야 합니다. 조건이 주어지지 않으면, 버그가 있다고 간주됩니다.

**검사** 이 필드에는 스마트 컨트랙트가 버그를 가지고 있는지 없는지를 보고하는 여러가지 검사가 있습니다. 첫번째 검사 유형은 Javascript 정규식입니다. 이는 버그가 있는 경우 소스 코드("source-regex")와 일치해야합니다. 일치하는 항목이 없다면, 버그가 존재하지 않을 가능성이 높습니다. 반대로, 일치하는 항목이 있다면, 버그가 존재할 수 있습니다. 정확성을 높이기 위해서 stripping 주석을 지운 뒤, 소스 코드에 검사를 적용해야 합니다. 두번째 검사 유형은 Solidity 프로그램의 소형 AST("ast-compact-json-path")에서 검사 할 패턴입니다. 특별한 검색 쿼리는 JsonPath 표현식입니다. Solidity AST의 경로가 쿼리와 하나라도 일치하지 않는 경우, 버그가 존재 할 수 있습니다.

```
[
    {
        "name": "ExpExponentCleanup",
        "summary": "Using the ** operator with an exponent of type shorter than 256␣
↪bits can result in unexpected values.",
        "description": "Higher order bits in the exponent are not properly cleaned␣
↪before the EXP opcode is applied if the type of the exponent expression is smaller␣
↪than 256 bits and not smaller than the type of the base. In that case, the result␣
↪might be larger than expected if the exponent is assumed to lie within the value␣
↪range of the type. Literal numbers as exponents are unaffected as are exponents or␣
↪bases of type uint256.",
```

(continues on next page)

(이전 페이지에서 계속)

```
        "fixed": "0.4.25",
        "severity": "medium/high",
        "check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
    },
    {
        "name": "EventStructWrongData",
        "summary": "Using structs in events logged wrong data.",
        "description": "If a struct is used in an event, the address of the struct is␣
↪logged instead of the actual data.",
        "introduced": "0.4.17",
        "fixed": "0.4.25",
        "severity": "very low",
        "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')]..
↪[?(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
↪'struct'))]"}
    },
    {
        "name": "NestedArrayFunctionCallDecoder",
        "summary": "Calling functions that return multi-dimensional fixed-size arrays␣
↪can result in memory corruption.",
        "description": "If Solidity code calls a function that returns a multi-
↪dimensional fixed-size array, array elements are incorrectly interpreted as memory␣
↪pointers and thus can cause memory corruption if the return values are accessed.␣
↪Calling functions with multi-dimensional fixed-size arrays is unaffected as is␣
↪returning fixed-size arrays from function calls. The regular expression only checks␣
↪if such functions are present, not if they are called, which is required for the␣
↪contract to be affected.",
        "introduced": "0.1.4",
        "fixed": "0.4.22",
        "severity": "medium",
        "check": {"regex-source": "returns[^;{]*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^
↪\\]]*\\]\\s*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\][^{;]*[;{]"}
    },
    {
        "name": "OneOfTwoConstructorsSkipped",
        "summary": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored.",
        "description": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored. There will be a␣
↪compiler warning about the old-style constructor, so contracts only using new-style␣
↪constructors are fine.",
        "introduced": "0.4.22",
        "fixed": "0.4.23",
        "severity": "very low"
    },
    {
        "name": "ZeroFunctionSelector",
        "summary": "It is possible to craft the name of a function such that it is␣
↪executed instead of the fallback function in very specific circumstances.",
        "description": "If a function has a selector consisting only of zeros, is␣
↪payable and part of a contract that does not have a fallback function and at most␣
↪five external functions in total, this function is called instead of the fallback␣
↪function if Ether is sent to the contract without data.",
        "fixed": "0.4.18",
        "severity": "very low"
```

(continues on next page)

(이전 페이지에서 계속)

```
    },
    {
        "name": "DelegateCallReturnValue",
        "summary": "The low-level .delegatecall() does not return the execution␣
↪outcome, but converts the value returned by the functioned called to a boolean␣
↪instead.",
        "description": "The return value of the low-level .delegatecall() function is␣
↪taken from a position in memory, where the call data or the return data resides.␣
↪This value is interpreted as a boolean and put onto the stack. This means if the␣
↪called function returns at least 32 zero bytes, .delegatecall() returns false even␣
↪if the call was successful.",
        "introduced": "0.3.0",
        "fixed": "0.4.15",
        "severity": "low"
    },
    {
        "name": "ECRecoverMalformedInput",
        "summary": "The ecrecover() builtin can return garbage for malformed input.",
        "description": "The ecrecover precompile does not properly signal failure for␣
↪malformed input (especially in the 'v' argument) and thus the Solidity function can␣
↪return data that was previously present in the return area in memory.",
        "fixed": "0.4.14",
        "severity": "medium"
    },
    {
        "name": "SkipEmptyStringLiteral",
        "summary": "If \"\" is used in a function call, the following function␣
↪arguments will not be correctly passed to the function.",
        "description": "If the empty string literal \"\" is used as an argument in a␣
↪function call, it is skipped by the encoder. This has the effect that the encoding␣
↪of all arguments following this is shifted left by 32 bytes and thus the function␣
↪call data is corrupted.",
        "fixed": "0.4.12",
        "severity": "low"
    },
    {
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode␣
↪by routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.ethereum.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying␣
↪memory, ignored its return value. On the public chain, calls to the identity␣
↪precompile can be made in a way that they never fail, but this might be different␣
↪on private chains.",
```

(continues on next page)

(이전 페이지에서 계속)

```
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was simplified to just␣
↪use the empty state, but this implementation was not done properly. This bug can␣
↪cause data corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly␣
↪and could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same␣
↪32 byte storage slot, but storage writes always write 32 bytes. For some types, the␣
↪higher order bytes were not cleaned properly, which made it sometimes possible to␣
↪overwrite a variable in storage when writing to another one.",
        "link": "https://blog.ethereum.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "name": "OptimizerStaleKnowledgeAboutSHA3",
        "summary": "The optimizer did not properly reset its knowledge about SHA3␣
↪operations resulting in some hashes (also used for storage variable positions) not␣
↪being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-
↪evaluating expressions whose value is already known. This knowledge was not␣
↪properly reset across control flow paths and thus the optimizer sometimes thought␣
↪that the result of a SHA3 operation is already present on the stack. This could␣
↪result in data corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that␣
↪received Ether.",
        "description": "Library functions are protected against sending them Ether␣
↪through a call. Since the DELEGATECALL opcode forwards the information about how␣
↪much Ether was sent with a call, the library function incorrectly assumed that␣
↪Ether was sent to the library and threw an exception.",
```

(continues on next page)

(이전 페이지에서 계속)

```
        "severity": "low",
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if␣
↪no Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a␣
↪certain amount of gas from the EVM to handle the transfer. In the case of a zero-
↪transfer, this gas is not provided which causes the recipient to throw an exception.
↪",
        "severity": "low",
        "fixed": "0.4.0"
    },
    {
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite␣
↪loop and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this␣
↪length is zero, code was generated that did not terminate and thus consumed all gas.
↪",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was not done correctly.␣
↪This bug can cause data corruption, but it is probably quite hard to use for␣
↪targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned␣
↪before comparison.",
        "description": "Two variables of type bytesNN were considered different if␣
↪their higher order bits, which are not part of the actual value, were different. An␣
↪attacker might use this to reach seemingly unreachable code paths by providing␣
↪incorrectly formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32␣
↪bytes did not correctly clean the higher order bits, causing corruption in other␣
↪array elements.",
        "description": "Multiple elements of an array of values that are shorter than␣
↪17 bytes are packed into the same storage slot. Writing to a single element of such␣
↪an array did not properly clean the higher order bytes and thus could lead to data␣
↪corruption.",
```

(continues on next page)

(이전 페이지에서 계속)

```
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several␣
↪undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting␣
↪from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
]
```

## 8.13 Contributing

Help is always appreciated!

To get started, you can try *소스에서 빌드하기* in order to familiarize yourself with the components of Solidity and the build process. Also, it may be useful to become well-versed at writing smart-contracts in Solidity.

In particular, we need help in the following areas:

- Improving the documentation
- Responding to questions from other users on StackExchange and the Solidity Gitter
- Fixing and responding to Solidity's GitHub issues, especially those tagged as up-for-grabs which are meant as introductory issues for external contributors.

Please note that this project is released with a Contributor Code of Conduct. By participating in this project - in the issues, pull requests, or Gitter channels - you agree to abide by its terms.

### 8.13.1 How to Report Issues

To report an issue, please use the GitHub issues tracker. When reporting issues, please mention the following details:

- Which version of Solidity you are using
- What was the source code (if applicable)
- Which platform are you running on
- How to reproduce the issue
- What was the result of the issue
- What the expected behaviour is

Reducing the source code that caused the issue to a bare minimum is always very helpful and sometimes even clarifies a misunderstanding.

### 8.13.2 Workflow for Pull Requests

In order to contribute, please fork off of the `develop` branch and make your changes there. Your commit messages should detail *why* you made your change in addition to *what* you did (unless it is a tiny change).

If you need to pull in any changes from `develop` after making your fork (for example, to resolve potential merge conflicts), please avoid using `git merge` and instead, `git rebase` your branch. This will help us review your change more easily.

Additionally, if you are writing a new feature, please ensure you add appropriate test cases under `test/` (see below).

However, if you are making a larger change, please consult with the Solidity Development Gitter channel (different from the one mentioned above, this one is focused on compiler and language development instead of language use) first.

New features and bugfixes should be added to the `Changelog.md` file: please follow the style of previous entries, when applicable.

Finally, please make sure you respect the coding style for this project. Also, even though we do CI testing, please test your code and ensure that it builds locally before submitting a pull request.

Thank you for your help!

### 8.13.3 Running the compiler tests

There is a script at `scripts/tests.sh` which executes most of the tests and runs `aleth` automatically if it is in the path, but does not download it, so it most likely will not work right away. Please read on for the details.

Solidity includes different types of tests. Most of them are bundled in the application called `soltest`. Some of them require the `aleth` client in testing mode, some others require `libz3` to be installed.

To run a basic set of tests that neither require `aleth` nor `libz3`, run `./scripts/soltest.sh --no-ipc --no-smt`. This script will run `build/test/soltest` internally.

---

**주석:** Those working in a Windows environment wanting to run the above basic sets without aleth or libz3 in Git Bash, you would have to do: `./build/test/RelWithDebInfo/soltest.exe -- --no-ipc --no-smt`. If you're running this in plain Command Prompt, use `.\build\test\RelWithDebInfo\soltest.exe -- --no-ipc --no-smt`.

---

The option `--no-smt` disables the tests that require `libz3` and `--no-ipc` disables those that require `aleth`.

If you want to run the ipc tests (those test the semantics of the generated code), you need to install aleth and run it in testing mode: `aleth --test -d /tmp/testeth` (make sure to rename it).

Then you run the actual tests: `./scripts/soltest.sh --ipcpath /tmp/testeth/geth.ipc`.

To run a subset of tests, filters can be used: `./scripts/soltest.sh -t TestSuite/TestName --ipcpath /tmp/testeth/geth.ipc`, where `TestName` can be a wildcard `*`.

The script `scripts/tests.sh` also runs commandline tests and compilation tests in addition to those found in `soltest`.

The CI even runs some additional tests (including `solc-js` and testing third party Solidity frameworks) that require compiling the Emscripten target.

---

**주석:** Some versions of `aleth` cannot be used for testing. We suggest using the same version that is used by the Solidity continuous integration tests. Currently the CI uses `d661ac4fec0aeffbedcdc195f67f5ded0c798278` of `aleth`.

---

### Writing and running syntax tests

Syntax tests check that the compiler generates the correct error messages for invalid code and properly accepts valid code. They are stored in individual files inside `tests/libsolidity/syntaxTests`. These files must contain annotations, stating the expected result(s) of the respective test. The test suite will compile and check them against the given expectations.

Example: `./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

A syntax test must contain at least the contract under test itself, followed by the separator `// ----`. The following comments are used to describe the expected compiler errors or warnings. The number range denotes the location in the source where the error occurred. In case the contract should compile without any errors or warning, the section after the separator has to be empty and the separator can be left out completely.

In the above example, the state variable `variable` was declared twice, which is not allowed. This will result in a `DeclarationError` stating that the identifier was already declared.

The tool that is being used for those tests is called `isoltest` and can be found under `./test/tools/`. It is an interactive tool which allows editing of failing contracts using your preferred text editor. Let's try to break this test by removing the second declaration of `variable`:

```
contract test {
    uint256 variable;
}
// ----
// DeclarationError: (36-52): Identifier already declared.
```

Running `./test/isoltest` again will result in a test failure:

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
        contract test {
            uint256 variable;
        }

    Expected result:
        DeclarationError: (36-52): Identifier already declared.
    Obtained result:
        Success
```

`isoltest` prints the expected result next to the obtained result, but also provides a way to change edit / update / skip the current contract or to even quit. It offers several options for failing tests:

- edit: `isoltest` tries to open the contract in an editor so you can adjust it. It either uses the editor given on the command line (as `isoltest --editor /path/to/editor`), in the environment variable `EDITOR` or just `/usr/bin/editor` (in this order).
- update: Updates the contract under test. This either removes the annotation which contains the exception not met or adds missing expectations. The test will then be run again.
- skip: Skips the execution of this particular test.
- quit: Quits `isoltest`.

Automatically updating the test above will change it to

```
contract test {
    uint256 variable;
}
// ----
```

and re-run the test. It will now pass again:

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

**주석:** Please choose a name for the contract file that explains what it tests, e.g. `double_variable_declaration.sol`. Do not put more than one contract into a single file, unless you are testing inheritance or cross-contract calls. Each file should test one aspect of your new feature.

### 8.13.4 Running the Fuzzer via AFL

Fuzzing is a technique that runs programs on more or less random inputs to find exceptional execution states (segmentation faults, exceptions, etc). Modern fuzzers are clever and do a directed search inside the input. We have a specialized binary called `solfuzzer` which takes source code as input and fails whenever it encounters an internal compiler error, segmentation fault or similar, but does not fail if e.g. the code contains an error. This way, internal problems in the compiler can be found by fuzzing tools.

We mainly use AFL for fuzzing. You need to download and install AFL packages from your repos (afl, afl-clang) or build them manually. Next, build Solidity (or just the `solfuzzer` binary) with AFL as your compiler:

```
cd build
# if needed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

At this stage you should be able to see a message similar to the following:

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

If the instrumentation messages did not appear, try switching the cmake flags pointing to AFL's clang binaries:

```
# if previously failed
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

Othwerise, upon execution the fuzzer will halt with an error saying binary is not instrumented:

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...
```

(continues on next page)

(이전 페이지에서 계속)

```
[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6920
```

Next, you need some example source files. This will make it much easier for the fuzzer to find errors. You can either copy some files from the syntax tests or extract test files from the documentation or the other tests:

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# extract from tests:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↪SolidityEndToEndTest.cpp
# extract from documentation:
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs docs
```

The AFL documentation states that the corpus (the initial input files) should not be too large. The files themselves should not be larger than 1 kB and there should be at most one input file per functionality, so better start with a small number of input files. There is also a tool called `afl-cmin` that can trim input files that result in similar behaviour of the binary.

Now run the fuzzer (the `-m` extends the size of memory to 60 MB):

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

The fuzzer will create source files that lead to failures in `/tmp/fuzzer_reports`. Often it finds many similar source files that produce the same error. You can use the tool `scripts/uniqueErrors.sh` to filter out the unique errors.

### 8.13.5 Whiskers

*Whiskers* is a string templating system similar to Mustache. It is used by the compiler in various places to aid readability, and thus maintainability and verifiability, of the code.

The syntax comes with a substantial difference to Mustache: the template markers `{{` and `}}` are replaced by < and > in order to aid parsing and avoid conflicts with *Inline Assembly* (The symbols < and > are invalid in inline assembly, while `{` and `}` are used to delimit blocks). Another limitation is that lists are only resolved one depth and they will not recurse. This may change in the future.

A rough specification is the following:

Any occurrence of `<name>` is replaced by the string-value of the supplied variable `name` without any escaping and without iterated replacements. An area can be delimited by `<#name>...</name>`. It is replaced by as many concatenations of its contents as there were sets of variables supplied to the template system, each time replacing any `<inner>` items by their respective value. Top-level variables can also be used inside such areas.

## 8.14 Frequently Asked Questions

This list was originally compiled by fivedogit.

### 8.14.1 Basic Questions

#### What is the transaction "payload"?

This is just the bytecode "data" sent along with the request.

#### Create a contract that can be killed and return funds

First, a word of warning: Killing contracts sounds like a good idea, because "cleaning up" is always good, but as seen above, it does not really clean up. Furthermore, if Ether is sent to removed contracts, the Ether will be forever lost.

If you want to deactivate your contracts, it is preferable to **disable** them by changing some internal state which causes all functions to throw. This will make it impossible to use the contract and ether sent to the contract will be returned automatically.

Now to answering the question: Inside a constructor, `msg.sender` is the creator. Save it. Then `selfdestruct(creator);` to kill and return funds.

example

Note that if you `import "mortal"` at the top of your contracts and declare `contract SomeContract is mortal { ...` and compile with a compiler that already has it (which includes Remix), then `kill()` is taken care of for you. Once a contract is "mortal", then you can `contractname.kill.sendTransaction({from:eth.coinbase})`, just the same as my examples.

#### Can you return an array or a `string` from a solidity function call?

Yes. See array_receiver_and_returner.sol.

#### Is it possible to in-line initialize an array like so: `string[] myarray = ["a", "b"];`

Yes. However it should be noted that this currently only works with statically sized memory arrays. You can even create an inline memory array in the return statement.

Example:

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    function f() public pure returns (uint8[5] memory) {
        string[4] memory adaArr = ["This", "is", "an", "array"];
        adaArr[0] = "That";
        return [1, 2, 3, 4, 5];
    }
}
```

#### Can a contract function return a `struct`?

Yes, but only in `internal` function calls or if `pragma experimental "ABIEncoderV2";` is used.

### If I return an `enum`, I only get integer values in web3.js. How to get the named values?

Enums are not supported by the ABI, they are just supported by Solidity. You have to do the mapping yourself for now, we might provide some help later.

### Can state variables be initialized in-line?

Yes, this is possible for all types (even for structs). However, for arrays it should be noted that you must declare them as static memory arrays.

Examples:

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    struct S {
        uint a;
        uint b;
    }

    S public x = S(1, 2);
    string name = "Ada";
    string[4] adaArr = ["This", "is", "an", "array"];
}

contract D {
    C c = new C();
}
```

### How do structs work?

See struct_and_for_loop_tester.sol.

### How do for loops work?

Very similar to JavaScript. Such as the following example:

```
for (uint i = 0; i < a.length; i ++) { a[i] = i; }
```

See struct_and_for_loop_tester.sol.

### What are some examples of basic string manipulation (`substring`, `indexOf`, `charAt`, etc)?

There are some string utility functions at stringUtils.sol which will be extended in the future. In addition, Arachnid has written solidity-stringutils.

For now, if you want to modify a string (even when you only want to know its length), you should always convert it to a `bytes` first:

```
pragma solidity >=0.4.0 <0.6.0;

contract C {
    string s;
```

(continues on next page)

(이전 페이지에서 계속)

```
    function append(byte c) public {
        bytes(s).push(c);
    }

    function set(uint i, byte c) public {
        bytes(s)[i] = c;
    }
}
```

### Can I concatenate two strings?

Yes, you can use `abi.encodePacked`:

```
pragma solidity >=0.4.0 <0.6.0;

library ConcatHelper {
    function concat(bytes memory a, bytes memory b)
            internal pure returns (bytes memory) {
        return abi.encodePacked(a, b);
    }
}
```

### Why is the low-level function `.call()` less favorable than instantiating a contract with a variable (`ContractB b;`) and executing its functions (`b.doSomething();`)?

If you use actual functions, the compiler will tell you if the types or your arguments do not match, if the function does not exist or is not visible and it will do the packing of the arguments for you.

See ping.sol and pong.sol.

### When returning a value of say `uint` type, is it possible to return an `undefined` or "null"-like value?

This is not possible, because all types use up the full value range.

You have the option to `throw` on error, which will also revert the whole transaction, which might be a good idea if you ran into an unexpected situation.

If you do not want to throw, you can return a pair:

```
pragma solidity >0.4.23 <0.6.0;

contract C {
    uint[] counters;

    function getCounter(uint index)
        public
        view
        returns (uint counter, bool error) {
            if (index >= counters.length)
                return (0, true);
            else
                return (counters[index], false);
    }
```

(continues on next page)

(이전 페이지에서 계속)

```
    function checkCounter(uint index) public view {
        (uint counter, bool error) = getCounter(index);
        if (error) {
            // Handle the error
        } else {
            // Do something with counter.
            require(counter > 7, "Invalid counter value");
        }
    }
}
```

#### Are comments included with deployed contracts and do they increase deployment gas?

No, everything that is not needed for execution is removed during compilation. This includes, among others, comments, variable names and type names.

#### What happens if you send ether along with a function call to a contract?

It gets added to the total balance of the contract, just like when you send ether when creating a contract. You can only send ether along to a function that has the `payable` modifier, otherwise an exception is thrown.

### 8.14.2 Advanced Questions

#### How do you get a random number in a contract? (Implement a self-returning gambling contract.)

Getting randomness right is often the crucial part in a crypto project and most failures result from bad random number generators.

If you do not want it to be safe, you build something similar to the coin flipper but otherwise, rather use a contract that supplies randomness, like the RANDAO.

#### Get return value from non-constant function from another contract

The key point is that the calling contract needs to know about the function it intends to call.

See ping.sol and pong.sol.

#### How do you create 2-dimensional arrays?

See 2D_array.sol.

Note that filling a 10x10 square of `uint8` + contract creation took more than `800,000` gas at the time of this writing. 17x17 took `2,000,000` gas. With the limit at 3.14 million... well, there's a pretty low ceiling for what you can create right now.

Note that merely "creating" the array is free, the costs are in filling it.

Note2: Optimizing storage access can pull the gas costs down considerably, because 32 `uint8` values can be stored in a single slot. The problem is that these optimizations currently do not work across loops and also have a problem with bounds checking. You might get much better results in the future, though.

### What happens to a `struct`'s mapping when copying over a `struct`?

This is a very interesting question. Suppose that we have a contract field set up like such:

```
struct User {
    mapping(string => string) comments;
}

function somefunction public {
   User user1;
   user1.comments["Hello"] = "World";
   User user2 = user1;
}
```

In this case, the mapping of the struct being copied over into `user2` is ignored as there is no "list of mapped keys". Therefore it is not possible to find out which values should be copied over.

### How do I initialize a contract with only a specific amount of wei?

Currently the approach is a little ugly, but there is little that can be done to improve it. In the case of a `contract A` calling a new instance of `contract B`, parentheses have to be used around `new B` because `B.value` would refer to a member of `B` called `value`. You will need to make sure that you have both contracts aware of each other's presence and that `contract B` has a `payable` constructor. In this example:

```
pragma solidity >0.4.99 <0.6.0;

contract B {
    constructor() public payable {}
}

contract A {
    B child;

    function test() public {
        child = (new B).value(10)(); //construct a new B with 10 wei
    }
}
```

### Can a contract function accept a two-dimensional array?

If you want to pass two-dimensional arrays across non-internal functions, you most likely need to use `pragma experimental "ABIEncoderV2";`.

### What is the relationship between `bytes32` and `string`? Why is it that `bytes32 somevar = "stringliteral";` works and what does the saved 32-byte hex value mean?

The type `bytes32` can hold 32 (raw) bytes. In the assignment `bytes32 samevar = "stringliteral";`, the string literal is interpreted in its raw byte form and if you inspect `somevar` and see a 32-byte hex value, this is just `"stringliteral"` in hex.

The type `bytes` is similar, only that it can change its length.

Finally, `string` is basically identical to `bytes` only that it is assumed to hold the UTF-8 encoding of a real string. Since `string` stores the data in UTF-8 encoding it is quite expensive to compute the number of characters in the string (the encoding of some characters takes more than a single byte). Because of that, `string s; s.length`

is not yet supported and not even index access `s[2]`. But if you want to access the low-level byte encoding of the string, you can use `bytes(s).length` and `bytes(s)[2]` which will result in the number of bytes in the UTF-8 encoding of the string (not the number of characters) and the second byte (not character) of the UTF-8 encoded string, respectively.

### Can a contract pass an array (static size) or string or `bytes` (dynamic size) to another contract?

Sure. Take care that if you cross the memory / storage boundary, independent copies will be created:

```
pragma solidity >=0.4.16 <0.6.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

The call to `g(x)` will not have an effect on `x` because it needs to create an independent copy of the storage value in memory. On the other hand, `h(x)` successfully modifies `x` because only a reference and not a copy is passed.

### Sometimes, when I try to change the length of an array with ex: `arrayname.length = 7;` I get a compiler error `Value must be an lvalue`. Why?

You can resize a dynamic array in storage (i.e. an array declared at the contract level) with `arrayname.length = <some new length>;`. If you get the "lvalue" error, you are probably doing one of two things wrong.

1. You might be trying to resize an array in "memory", or
2. You might be trying to resize a non-dynamic array.

```
pragma solidity >=0.4.18 <0.6.0;

// This will not compile
contract C {
    int8[] dynamicStorageArray;
    int8[5] fixedStorageArray;

    function f() public {
        int8[] memory memArr;        // Case 1
        memArr.length++;             // illegal

        int8[5] storage storageArr = fixedStorageArray;    // Case 2
        storageArr.length++;                              // illegal

        int8[] storage storageArr2 = dynamicStorageArray;
```

(continues on next page)

(이전 페이지에서 계속)

```
        storageArr2.length++;                       // legal


    }
}
```

**Important note:** In Solidity, array dimensions are declared backwards from the way you might be used to declaring them in C or Java, but they are access as in C or Java.

For example, `int8[][5] somearray;` are 5 dynamic `int8` arrays.

The reason for this is that `T[5]` is always an array of 5 `T`'s, no matter whether `T` itself is an array or not (this is not the case in C or Java).

### Is it possible to return an array of strings (`string[]`) from a Solidity function?

Only when `pragma experimental "ABIEncoderV2";` is used.

### What does the following strange check do in the Custom Token contract?

```
require((balanceOf[_to] + _value) >= balanceOf[_to]);
```

Integers in Solidity (and most other machine-related programming languages) are restricted to a certain range. For `uint256`, this is `0` up to `2**256 - 1`. If the result of some operation on those numbers does not fit inside this range, it is truncated. These truncations can have serious consequences, so code like the one above is necessary to avoid certain attacks.

### Why are explicit conversions between fixed-size bytes types and integer types failing?

Since version 0.5.0 explicit conversions between fixed-size byte arrays and integers are only allowed, if both types have the same size. This prevents unexpected behaviour when truncating or padding. Such conversions are still possible, but intermediate casts are required that make the desired truncation and padding convention explicit. See types-conversion-elementary-types for a full explanation and examples.

### Why can number literals not be converted to fixed-size bytes types?

Since version 0.5.0 only hexadecimal number literals can be converted to fixed-size bytes types and only if the number of hex digits matches the size of the type. See types-conversion-literals for a full explanation and examples.

### More Questions?

If you have more questions or your question is not answered here, please talk to us on gitter or file an issue.

## 8.15 LLL

LLL은 s-expressions 문법을 사용하는 EVM의 저수준 언어입니다.

Solidity 저장소는 LLL 컴파일러가 포함되어 있으며 어셈블러 하위시스템을 Solidity와 공유합니다. 하지만, 컴파일을 유지하는 것과 별개로 다른 개선점은 없습니다.

특별히 요청하지 않는 한 빌드되지 않습니다.

```
$ cmake -DLLL=ON ..
$ cmake --build .
```

**경고:** LLL 코드베이스는 제공되지 않을것이며, 향후 Solidity 저장소에서 삭제될 예정입니다.

# 색인

## F



## G



## H



## I



## J



## K



## L



## M



## N



## O



## P

## R



## S



## T



## U



## V



## W



## Y