Introduction to Solidity (Part 2)

Disarankan sudah membaca Introduction to Solidity (Part 1) untuk pemahaman dasar seputar bahasa pemrograman Solidity. Syntax di sini mengikuti versi Solidity terbaru.

Contents

Loop 3

While Loop 3

Do-While Loop 3

For Loop 3

Break and Continue Statement 4

Decision-Making Statement 5

If Statement 5

Else If - Else Statement 5

Assert Statement 6

Revert Statement 6

Event 7

Inheritance 7

Inheritance Keyword 9

Super Keyword 9

Override Keyword 9

Virtual Keyword 9

Abstract Contract 9

Interface 9

Library 10

Loop

Seperti bahasa pemrograman lainnya, Solidity juga memiliki loop untuk mengulang eksekusi code.

While Loop

While loop akan melakukan pengulangan code selama kondisi yang ditentukan dalam while bernilai **true**. Pengecekan kondisi dilakukan di tahap awal sebelum code dieksekusi.

Syntax:

```
while (kondisi) {
   // code untuk dieksekusi
}
```

Contoh:

```
function whileLoop() public pure returns(uint8) {
   uint8 num = 0;
   while (num < 5) {
   num++;
   }
   return num;
}
```

Do-While Loop

Do-while loop akan menjalankan code setidaknya 1 kali sebelum mengecek kondisi di akhir iterasi. Jadi walau dari awal kondisinya **false**, codenya tetap akan jalan dulu.

Syntax:

```
do {
   // code untuk dieksekusi
} while (kondisi);
```

Contoh:

```
function doWhileLoop() public pure returns(uint8) {
   uint8 num = 0;
   do {
   num++;
   }
   while (num < 5);
   return num;
}
```

For Loop

For loop adalah format loop yang paling compact. Loop ini memiliki beberapa komponen, yaitu:

- Inisialisasi loop counter
- Kondisi yang perlu dipenuhi
- Statement iterasi untuk menaik-turunkan counter

Syntax:

```
for (inisialisasi; kondisi; statement iterasi) {
   // code untuk dieksekusi
}
```

Contoh:

```
function forLoop() public pure returns(uint8) {
   uint8 num = 0;
   for(uint i = 0; num < 5; i++){
   num++;
   }
   return num;
}
```

Contoh di atas menginisialisasi variable counter **i** dari 0 lalu cek kondisi nilai variable **num** kurang dari 5, jika sesuai maka code di dalam for dieksekusi, kemudian variable **i** dinaikkan nilainya dengan 1.

Break and Continue Statement

Seperti bahasa pemrograman lainnya, Solidity juga ada **break** dan **continue** statement untuk mengontrol flow dari loop. Perbedaannya, **break** digunakan untuk keluar dari loop lebih awal jika suatu kondisi terpenuhi, sedangkan **continue** digunakan untuk melewati eksekusi sisa code dalam suatu iterasi jika kondisi terpenuhi.

Contoh:

```
function forLoop() public pure returns(uint) {
   uint num = 0;
   while (num < 10) {
     if(num == 6) {
       break; // saat num bernilai 6, keluar dari loop
     }
     num += 2;
   }
   return num;
}
```

```
function forLoop() public pure returns(uint) {
   uint num = 0;
   for (uint i = 0; i < 10; i++) {
   if(i % 2 == 0) {
      continue; // setiap i genap, maka num++ tidak dieksekusi
   }
   num++;
   }
   return num;
}
```

Decision-Making Statement

Decision-making statement digunakan untuk mengatur flow dari eksekusi code berdasarkan kondisi tertentu. Statement yang digunakan ada 3: **if**, **else**, dan **else if**. Di contoh code sebelumnya, kita sudah menggunakan **if**.

If Statement

If statement adalah kondisial statement yang paling dasar. Ia hanya akan mengeksekusi code di dalamnya jika kondisinya bernilai **true**.

Syntax:

```
if (kondisi) {
   // code untuk dieksekusi
}
```

Else If- Else Statement

If statement bisa dilanjutkan dengan elses (**else if** atau **else**). Gunanya untuk menambahkan kontrol ke flow eksekusi code. **Else if** digunakan jika ada kondisi lain yang perlu dicek kebenarannya untuk mengeksekusi code dan **else** untuk mengeksekusi code yang kondisinya tidak dipenuhi dalam **if** atau **else if** sebelumnya. Else statement harus diletakkan terakhir jika digunakan.

Syntax:

```
if (kondisi 1) {
   // code untuk dieksekusi jika kondisi 1 true
} else if (kondisi 2) {
   // code untuk dieksekusi jika kondisi 2 true
} else if (kondisi 3) {
   // code untuk dieksekusi jika kondisi 3 true
} else {
   // code untuk dieksekusi jika tidak ada kondisi di atas yang terpenuhi
}
```

Contoh:

```
function check(uint a) public pure returns(string memory) {
if (a >= 0 && a <= 10) {
   return "A di antara 0 dan 10";
}
else if (a > 50) {
   return "A lebih besar dari 50";
}
else {
   return "A angka positif";
}
```

Assert Statement

Assert statement mirip dengan require statement. Keduanya sama-sama mengembalikan boolean atas hasil evaluasi suatu kondisi dan me-revert perubahan yang telah terjadi jika **false**. Berikut perbedaan assert dengan require.

Require Statement	**Assert Statement**
Cek user input dan kondisi sebelum eksekusi function.	Cek kemungkinan adanya internal error (sesuatu yang seharusnya tidak terjadi) dan kondisi setelah suatu perubahan.
Umumnya digunakan di awal function.	Umumnya digunakan di akhir function.
Umum digunakan di tahap production.	**Tidak disarankan** digunakan di production karena untuk **testing dan debugging**.
Refund gas yang tidak terpakai jika kondisi **false**.	Menghabiskan semua gas yang ada walau kondisi **false**.
Ada error message.	Tidak ada error message.

Karena perbedaan di atas, bisa dibilang code yang berfungsi dengan benar seharusnya tidak akan mendapat assert yang false.

Syntax:

```
assert(condition);
```

Contoh:

```
function add(uint a, uint b) public pure returns(uint) {
   uint num = a + b;
   assert(num > a);
return num;
}
```

Revert Statement

Revert digunakan untuk **revert** (kembali ke kondisi sebelum eksekusi function) dan memunculkan error. Statement ini mengandung string yang mengindikasikan penyebab suatu exception. Revert mirip dengan require dimana gas yang tidak terpakai akan di-refund dan handle error dengan **error message yang bersifat optional**. Bedanya, revert digunakan untuk error handling dengan logic lebih kompleks seperti nested if (decision-making statement yang kompleks). Biasanya digunakan bersamaan dengan **require statement**.

Syntax:

```
revert(errorMessage);
```

Contoh:

```
function withdraw(uint amount) public {
   require(balances[msg.sender] >= amount, "Insufficient balance");
   balances[msg.sender] -= amount;

   if (!msg.sender.send(amount)) {
      revert("Failed to send funds");
   }
}
```

Event

Event di Solidity digunakan untuk logging ke blockchain saat di emit. Event dapat dimanfaatkan untuk event listening dan storage yang lebih murah. Data yang di-emit dari event tidak disimpan di contract dan tidak dapat diakses contract. Event dapat memberikan detail lebih jelas terkait suatu transaksi. Dalam mendefinisikan event, nama parameter bersifat optional, tapi nama parameter membuat informasi lebih jelas.

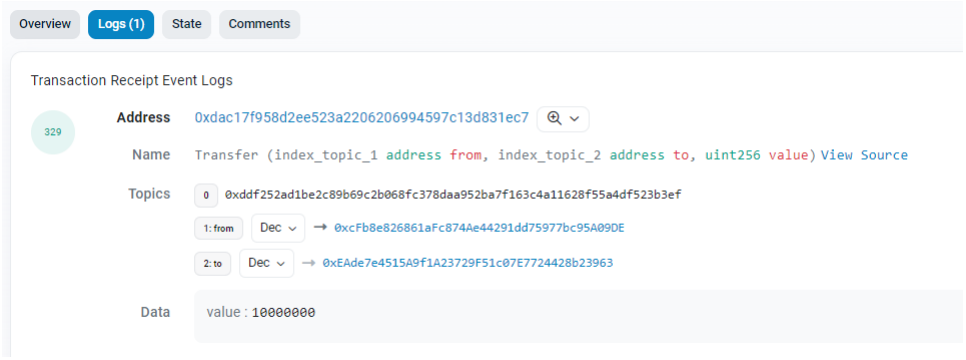
Syntax:

```
event <eventName>(datatype parameterName);
```

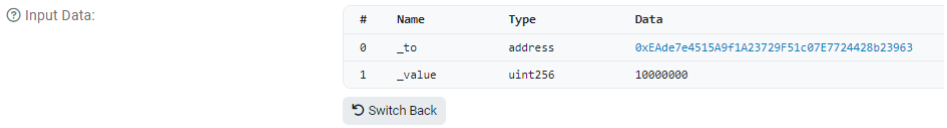
Contoh:

```
event Transfer(address indexed from, address indexed to, uint256 value);
```

Tampilan log event yang di emit di etherscan:



1 Tampilan dari tab Logs



2 Tampilan dari tab Overview setelah decode

Parameter event dapat berupa **indexed parameter** agar pencarian log event lebih mudah dari aplikasi. Satu event bisa punya maksimal 3 index. Aggap aja index parameter seperti key dari mapping dimana valuenya adalah log-nya.

Inheritance

Inheritance adalah sebuah konsep yang hadir dalam OOP (object-oriented programming) language. Inheritance membuat reusability code meningkat, struktur project lebih rapih, dan dependency berkurang. Istilah kerennya, **codebase menjadi lebih modular**. Simplenya, dengan inheritance kita bisa menggunakan code dari contract lain di contract kita tanpa harus copy-paste isi codenya.

Solidity mendukung inheritance beberapa contract ke satu contract dimana implementasi inheritance menggunakan keyword **is**. Inget visibility dari variable dan function? Semua yang bersifat **private** tidak bisa diakses oleh derived contract (contract yang meng-inherit) dan **external function** juga tidak dapat dipanggil dari dalam function. Berikut beberapa terminology dalam membahas inheritance:

- Parent atau base contract adalah contract yang di-inherit
- Child atau derived contract adalah contract yang meng-inherit

- Base function adalah function dari parent contract

Inheritance ada beberapa jenis:

- Single inheritance: child contract inherit satu parent contract. Parent-child contracts.

```
contract A is B {}
```

- Multiple inheritance: child contract inherit 2 atau lebih parent contract.

```
contract A is B, C {}
```

- Multi-level inheritance: ada level hubungan inheritance.

```
contract B is A {}
contract C is B {}
```

Syntax:

```
contract ChildContract is MomContract, DadContract;
```

Contoh:

```
abstract contract Inherited {
   function returnString() public virtual pure returns(string memory str) {
       str = 'Returned from Inherited';
   }
}

contract Inherits is Inherited {
   function returnString() public override pure returns(string memory str) {
       str = super.returnString(); // returns 'Returned from Inherited'
   }

   function returnStr() public pure returns(string memory str) {
       str = super.returnString(); // returns 'Returned from Inherited'
   }
}
```

Inheritance Keyword

Dalam menggunakan inheritance, kalian akan bertemu keyword dan function type baru.

Super Keyword

Mirip dengan bahasa pemrograman Java, Solidity juga ada **super** keyword. Intinya, keyword ini ngasih tau kalau function yang dieksekusi adalah dari contract yang di-inherit.

Override Keyword

Setiap kali kita meng-override function dari parent contract (menggunakan **function signature/name yang sama**), maka kita perlu menggunakan keyword **override**. Kalau function signature-nya berbeda, keyword ini tidak diperlukan.

Virtual Keyword

Function dari parent contract perlu keyword override agar dapat di-override. Jika tidak ada keyword ini, function dari parent contract masih dapat di akses dengan **super** tapi inisialisasi function signature yang sama tidak bisa dilakukan di derived contract.

Abstract Contract

Abstract contract adalah contract yang tidak dapat di-deploy sendirian karena function-nya kurang implementasi code, ia harus di-deploy dengan contract yang meng-inheritnya. Contohnya adalah abstract contract ERC20 dari OpenZeppelin.

Interface

Interface mirip dengan abstract contract dimana keduanya sama-sama bisa di-inherit. Semua function yang didefinisikan dalam interface secara automatis adalah **virtual function** jadi keyword virtual tidak perlu ditambahkan lagi. Perbedaan interface dengan abstract contract adalah interface hanya mengandung **definisi function tanpa implementasi** (function body) sama sekali. Kegunaannya untuk menjadi blueprint, mendefinisikan sekumpulan function dan event yang menjadi acuan struktur contract untuk development yang kompleks di blockchain. Contract yang meng-inherit sebuah interface harus mendefinisikan seluruh functionnya di dalam contract itu. Contohnya adalah interface contract IERC20 dari OpenZeppelin. Interface dan abstract smart contract dapat saling melengkapi seperti ERC20 dengan IERC20.

Interface standar token menunjukkan seberapa bergunanya interface agar kita dapat dengan mudah berinteraksi dengan banyaknya token ERC20 yang ada, misal untuk sekedar mengecek saldo suatu address.

Penamaan interface sebaiknya diawali huruf ‘I’ kapital.

Syntax:

```
Interface IInterfaceName {
   // functions, events
}
```

Contoh:

```
interface IERC20 {
   function balanceOf(address account) external view returns (uint256);
   function approve(address spender, uint256 value) external returns (bool);
   function transfer(address to, uint256 value) external returns (bool);
   function transferFrom(address from, address to, uint256 value) external returns (bool);
}
```

Interface juga dapat digunakan untuk import contract. Jadi misalnya kita mau memerlukan sebagian function dari IERC20 di contract kita karena ada interaksi dengan ERC20 token, maka kita bisa mendefinisikan interface seperti contoh di atas dan memanggilnya dengan cara berikut jika contract sudah di-deploy.

Syntax:

```
IInterfaceName(contractAddress).functionName();
```

Contoh:

```
IERC20(ERC20TokenAddress).balanceOf(address(this));
```

ERC20TokenAddress adalah address dari ERC20 token contract.

Cara lain untuk import contract adalah dengan keyword **import** jika file contract dimiliki.

Syntax:

```
import "filepath/contractFilename.sol";
```

Contoh:

```
import "./IERC20.sol";
```

Library

Library adalah jenis contract yang spesial. Mirip dengan abstract contract, gunanya adalah untuk meningkatkan reusability code. Isi dari library adalah sekelompok function berupa helper function (function pembantu) yang mengefisiensikan smart contract development, mengoptimisasi gas dengan mengurangi code duplikat, dan mempermudah organisasi code (readability, maintainability, consistency). Library tidak bisa di inherit seperti abstract contract dan interface. Contohnya adalah library SafeMath dari OpenZeppelin.

Syntax:

```
library MyLibrary {
     // functions
}
```

Contoh:

```
library SafeMath {

   /**
   * @dev Multiplies two numbers, throws on overflow.
   */
   function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
     if (a == 0) {
       return 0;
     }
     c = a * b;
     assert(c / a == b);
     return c;
   }
}
```

Cara importnya bisa dengan import file contract berisi library dengan keyword import seperti yang dicontohkan di bagian interface. Cara lain import external contract adalah dengan Node module dan URL file GitHub.

Misalnya library SafeMath di atas disimpan dalam file Lib.sol, maka kita bisa menggunakan cara berikut untuk menggunakan library-nya.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import "./Lib.sol";
// Defining calling contract
contract LibraryExample {

   address owner = address(this);

   function Multiply(
     uint num1, uint num2) public pure returns (
     uint) {
      return SafeMath.mul(num1, num2);
      // akses function library dengan memanggil nama contract
   }
}
```

Cara lain bisa menggunakan **using … for …** seperti berikut. Gaya syntax ini memiliki istilah **syntactic sugar**.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import "./Lib.sol";
// Defining calling contract
contract LibraryExample {
    using SafeMath for uint256;
    address owner = address(this);

    function Multiply(
      uint num1, uint num2) public pure returns (
      uint) {
        return num1.mul(num2);
    }
}
```