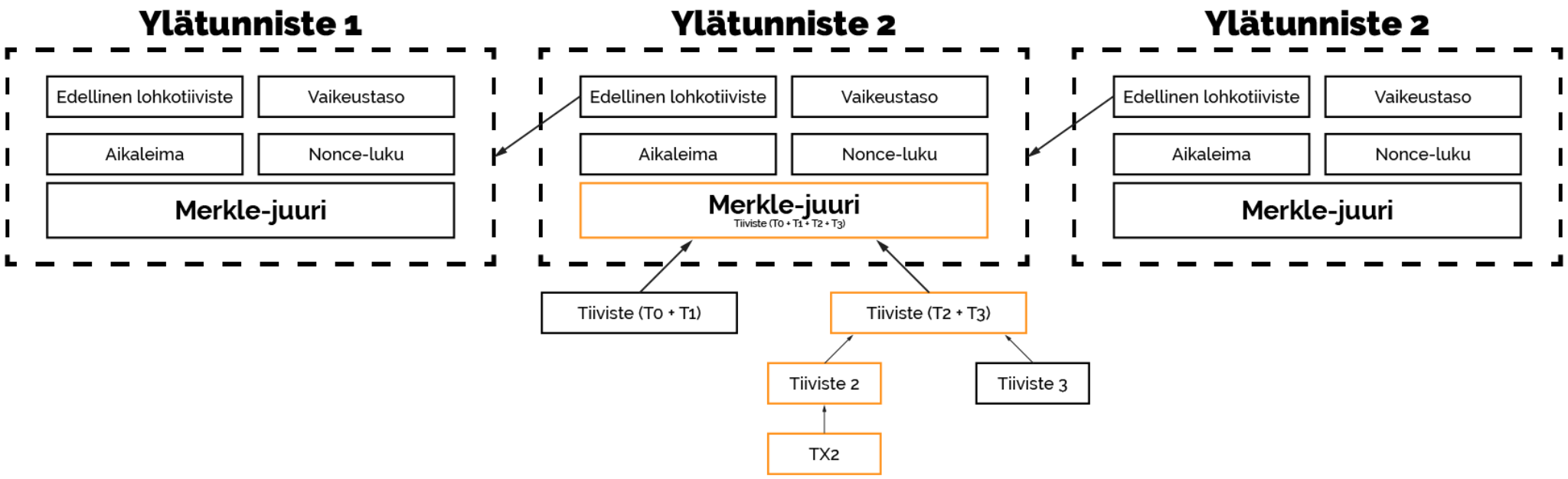
Solidity älysopimus ohjelmointi

Sopimus suuntautunut ohjelmointi

Merkle puu


Ylätunniste 1
Edellinen lohkotiiviste
Vaikeustaso
Aikaleima
Nonce-luku
Merkle-juuri
Ylätunniste 2
Edellinen lohkotiiviste
Vaikeustaso
Aikaleima
Nonce-luku
Merkle-juuri
Tiiviste (T0 + T1 + T2 + T3)
Ylätunniste 2
Edellinen lohkotiiviste
Vaikeustaso
Aikaleima
Nonce-luku
Merkle-juuri
Tiiviste (T0 + T1)
Tiiviste (T2 + T3)
Tiiviste 2
Tiiviste 3
TX2

Kertausta eiliseltä

- Solidity on korkean tason älysopimus ohjelmointikieli
  - Muistuttaa olio-ohjelmointia Javalla
- Sopimuskoodi on suunniteltu ajettavaksi Ethereum virtuaalikoneessa (EVM)
- Sopimukset itsessään toimivat yleensä tilakoneina
  - Käyttäytyvät ja toimivat erilailla eri vaiheissa
  - Funktiokutsut yleensä siirtävät sopimuksen eri vaiheeseen
  - Vaiheet voivat myös muuttua automaattisesti ilman herätteitä

Sopimuksen rakenne

- Sopimus siis muistuttavat rakenteeltaan hyvinkin paljon luokkaa olio-ohjelmoinnista
- Sopimuksen pystyy myös luokkien tavalla perimään toisesta sopimuksesta
- Sopimuksessa on mahdollista määrittää:
  - tilamuuttujia (state variable)
  - funktioita (function)
  - funktiomääritteitä (function modifier)
  - tapahtumia (event)
  - rakenteita (struct)
  - arvojoukkoja (enum)

Tilamuuttuja

- Tilamuuttujan arvot tallennetaan pysyvästi sopimuksen tallennustilaan
- Solidityssä on useita eri perustyyppejä muuttujille
  - Tyyppejä voidaan yhdistellä myös monimutkaisemmiksi muuttujiksi
- Tyyppien käytössä on myös vertailu- ja aritmeettisia operaatioita, jotka ovat tuttuja muista ohjelmointikielistä

```
address osoite;
bool totuusArvo;
int kokonaisluku; // int8 - int256
uint8 etuMerkiton; //uint8 - uint256
bytes32 fixdTavut; // bytes1 - bytes32
bytes dynaamisetTavut; // raaka
string merkkijono; // utf-8
int[] taulukkoNumeroita;
int[] allokoituTaulukko = new int[](5);
```

Funktiot

- Funktiot ovat sopimuksessa suoritettavia koodipätkiä
- Funktiokutsut voivat tapahtua sisäisesti tai ulkoisesti
- Sisäisesti funktiokutsut tapahtuvat sopimuksen sisällä
- Ulkoiset funktiokutsut kutsuvat toista sopimusta

```
contract A {

    int private kokonaisluku;

    function setData(int a) public {
        kokonaisluku = a;
    }

    function getData() public view returns(int) {
        return kokonaisluku;
    }
}

contract B {

    A a = new A();

    function setA(int kokonaisluku) public {
        a.setData(kokonaisluku);
    }

    function getA() public view returns(int) {
        return a.getData();
    }
}
```

Näkyvyys

- Funktiolle voidaan asettaa neljä eri näkyvyysmuotoa
  - *external*
  - *public*
  - *private*
  - *Internal*
- Tilamuuttujille voidaan näistä asettaa *public, private* tai *internal*

```
contract A {

    int public kokonaisluku = 32;

    function setData(int a) private {
        kokonaisluku = a;
    }
}

contract B {

    A a = new A();

    function setA(int kokonaisluku) public {
        a.setData(kokonaisluku); // setData -funktio ei ole näkyvissä
    }

    function getA() public view returns(int) {
        return a.kokonaisluku();
    }
}
```

Perintä ja konstruointi

- Sopimukset pystyvät periytymään yhdestä tai useammasta sopimuksesta
  - Useammasta sopimuksesta perittäessä luodaan yksi sopimus, johon on kopioitu perittyjen sopimuksien sisältö
- Konstruoinnissa voidaan alustaa sopimuksen tilamuuttujia

```
contract A {
    uint public a;

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

contract B is A(3) {

}
```

Funktio määritteet

- Funktio määritteiden avulla pystytään tarkistamaan jokin ehto ennen koodin suorittamista
- Määritteissä alaviiva merkkaa aluetta mistä määritettä käyttävän funktion koodi alkaa
- Määritteitä voidaan myös periä ja korvata uusilla ehdoilla johdetuissa sopimuksissa

```
contract Ownable {
    address public owner;

    function Ownable() public {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        owner = newOwner;
    }

}
```

Tapahtumat

- EVM lokin avulla pystytään lähettämään herätteitä lohkoketjussa tapahtuneista tapahtumista esimerkiksi käyttöliittymän puolelle
- *Indexed* avainsanalla varustettujen parametrien saamia arvoja voidaan käyttää hyväksi toimintalokin *suodattamisessa*

```
contract Ownable {
    address public owner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    function Ownable() public {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }

}
```

Rakenteet ja kuvaajat

- Rakenteet ovat muuttujien yhdistelmiä
- Yhdistelemällä eri muuttujia voidaan muodostaa uudenlaisia tyyppejä
- Rakenteita voidaan käyttää niin taulukoissa kuin kuvaajissa
- Kuvaajista voi myös saada yksinkertaisesti getterin lisäämällä siihen avainsanan *public*

```
contract A {
    struct Investor {
        address owner;
        uint amount;
    }

    mapping(string=>Investor) investors;
}
```

Arvojoukot

- Arvojoukkojen avulla voidaan luoda omia tyyppejä Solidityssä.
- Arvojoukossa pitää olla ainakin yksi arvo
- Arvojoukot soveltuu hyvin kuvaamaan sopimuksen tilan siirtymisiä

```
enum state {Pending, Approved, Canceled}
```

Abstrakti sopimus

- Sopimukset ovat abstrakteja mikäli yksi tai useampi funktio ei ole toteutettu sopimuksessa
- Sopimuksia missä kaikkia funktioita ei ole toteutettu ei voida kääntää
- Abstraktit sopimukset toimivat hyvin pohjana sopimuksille
- Johdettu sopimus on myös abstrakti jos se ei toteuta kaikkia funktioita peritystä abstraktista sopimuksesta

```
contract ERC20Basic {
    function totalSupply() public view returns (uint256);
    function balanceOf(address who) public view returns (uint256);
    function transfer(address to, uint256 value) public returns (bool);
    event Transfer(address indexed from, address indexed to, uint256 value);
}
```

Rajapinta

- Rajapinta käyttäytyy samalla tavalla kuin abstraktit sopimukset, mutta rajapinnoissa ei saada toteuttaa yhtäkään funktiota
- Tämän lisäksi rajapintoja koskee seuraavat säännöt:
  - Ei voi periä toisia sopimuksia tai rajapintoja
  - Ei voi konstruoida
  - Ei voi alustaa muuttujia
  - Ei voi alustaa rakenteita
  - Ei voi alustaa arvojoukkoja

```
interface ERC20Basic {
    function totalSupply() external view returns (uint256);
    function balanceOf(address who) external view returns (uint256);
    function transfer(address to, uint256 value) external returns (bool);
    event Transfer(address indexed from, address indexed to, uint256 value);
}
```

Kirjastot

- Kirjasto on sopimus jolla ei ole omaa tallennustilaa ja ei voi omistaa Etheriä
- Kirjasto on pala koodia EVM:n sisällä jota jokainen sopimus voi kutsua ilman että sitä tarvitsee uudelleen luoda
  - Tämä säästää esimerkiksi kaasukustannuksissa
  - Hyvä tapa parantaa koodin uudelleenkäytettävyyttä
- SafeMath kirjasto voidaan käyttää esimerkiksi uint tyyppisissä laskutoimituksissa:
  - using SafeMathLib for uint;

```
library SafeMath {

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }
        uint256 c = a * b;
        assert(c / a == b);
        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // assert(b > 0); // Solidity automatically throws when dividing by 0
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        assert(b <= a);
        return a - b;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        return c;
    }
}
```

Error Handling

`assert(bool condition)`:
throws if the condition is not met - to be used for internal errors.

`require(bool condition)`:
throws if the condition is not met - to be used for errors in inputs or external components.

`require(bool condition, string message)`:
throws if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.

`revert()`:
abort execution and revert state changes

`revert(string reason)`:
abort execution and revert state changes, providing an explanatory string

Tutustu Solidityn dokumentointiin

Enemmän tavaraa, esimerkkejä ja tietoa

?

Koodataan!

tai ainakin yritetään…