Solidity çalışma notları – Part 2

Adresler ve Mappingler:

Adresler sadece spesifik bir kullanıcıya özeldir ve uniquedir.

Mappingler pythondaki dictionarylere benzetilebilir. Key-value ilişkisi vardır.

Örnek Kod:

// For a financial app, storing a uint that holds the user's account balance:

mapping (address => uint) public accountBalance;

// Or could be used to store / lookup usernames based on userId

mapping (uint => string) userIdToName;

Msg.sender:

Solidityde belirli global değerler vardır ve bütün fonksiyonlarda ulaşılabilirler. Msg.sender göndericinin adresini geri döndürür. Bu güvenliği arttırmaya yardımcı olur.

Örnek kullanım şekli:

function setMyNumber(uint \_myNumber) public {

// Update our `favoriteNumber` mapping to store `\_myNumber` under `msg.sender`

favoriteNumber[msg.sender] = \_myNumber;

// ^ The syntax for storing data in a mapping is just like with arrays

}

Require:

Türkçesi gereklilik olan bu method Soliditynin kodun içinde bir gerekliliğini savunur. Böylelikle bu şart sağlanmadan kod devam etmez ve hata verir.

Örnek kullanım:

function createRandomZombie(string memory \_name) public {

require(ownerZombieCount[msg.sender] == 0);

uint randDna = \_generateRandomDna(\_name);

\_createZombie(\_name, randDna);

}

Inheritance:

Inheritence olaylarında inherit ettiğimiz kontratın fonksiyonlarını kullanabilme özelliğine sahip oluruz.

Örnek kullanım:

contract ZombieFeeding is ZombieFactory {

}

Storage ve Memory (Data lokasyonları):

Solidityde iki farklı lokasyonda datanızı saklayabilirsiniz – storage ve memoryde.

Storagede depolanan değerler kalıcı olarak depolanır fakat Memorydekiler external fonksiyon çağrıları arasında silinirler. Bunlar HardDisk ve Ram’e benzetilebilir.

Genel olarak bunlar default olarak atanır. State değerler yani fonksiyon dışı declare edilmiş olanların default Storage’dir. Fonksiyon içinde declare edilmiş değerler Memorydir. Bazı zamanlarda kendimiz atammak isteyebiliriz. Örneğin structlar ve arrayler üzerinde çalışırken olabilir.

Örnek kullanım:

function eatSandwich(uint \_index) public {

// Sandwich mySandwich = sandwiches[\_index];

// ^ Seems pretty straightforward, but solidity will give you a warning

// telling you that you should explicitly declare `storage` or `memory` here.

// So instead, you should declare with the `storage` keyword, like:

Sandwich storage mySandwich = sandwiches[\_index];

// ...in which case `mySandwich` is a pointer to `sandwiches[\_index]`

// in storage, and...

mySandwich.status = "Eaten!";

// ...this will permanently change `sandwiches[\_index]` on the blockchain.

// If you just want a copy, you can use `memory`:

Sandwich memory anotherSandwich = sandwiches[\_index + 1];

// ...in which case `anotherSandwich` will simply be a copy of the

// data in memory, and...

anotherSandwich.status = "Eaten!";

// ...will just modify the temporary variable and have no effect

// on `sandwiches[\_index + 1]`. But you can do this:

sandwiches[\_index + 1] = anotherSandwich;

// ...if you want to copy the changes back into blockchain storage.

}

Internal ve External fonksiyonlar:

Internal private ile aynı özelliklere sahip fakat buna dahil olarak inherit edildiği kontratlarda çalışabilir.

External ise public’e benzerdir. Fakat bunlar sadece kontrat dışı çağırılabilirler. Aynı kontratın içindeki bir fonksiyonda çalıştırılamazlar.

contract Sandwich {

uint private sandwichesEaten = 0;

function eat() internal {

sandwichesEaten++;

}

}

contract BLT is Sandwich {

uint private baconSandwichesEaten = 0;

function eatWithBacon() public returns (string memory) {

baconSandwichesEaten++;

// We can call this here because it's internal

eat();

}