Développeur blockchain

Solidity

Approfondissements

ALYRA

cyril@alyra.fr / benjamin.brucher@alyra.fr

Promo Buterin

Topo du live

- Revenir sur le transfert d'eth
- Revenir sur les contracts et les events
- Comprendre le view et pure
- Comprendre les visibilités
- Comprendre et manipuler l'héritage
- Comprendre le constructeur
- Comprendre les conditions
- Un exercice - sa correction
- Maitriser struct - mappings - arrays - enum
- Exercices pour demain

Avec ça: toutes les connaissances pour le projet

But du live: mélanger théorie et code ensemble

ALYRA

Solidity

Transferts d'eth - Envoyer

***Rappel**: Quant on parle de transferts d'eth, en réalité, on parle en wei, soit 10^-18 eth.*

3 manières d'envoyer via solidity des Eth:

```
function sendEth(address _to, uint amount) public
payable{
  _to.transfer(amount)
  _to.send(amount)
  (bool sent, bytes memory data) = _to.call{value:
amount, gas : 25000}("");
}
```

*(Cette fonction existe dans un contexte de contrat qui possède 3*amount eth dans sa balance)*

ALYRA

Solidity

Transferts d’eth - Envoyer

Bonne pratique à ce sujet:
Send et *transfer* coûtent un coût fixe au transfert d’Eth (24 000 gas).
De fait, il vaut mieux utiliser .call pour éviter des soucis, et par volonté d'être "*futur proof*".

Send et *call* vont retourner un booléen *false* si le transfert ne se fait pas.
Transfer fait revert la fonction, ce qui est pratique.
Ainsi, pour que la fonction revert si le booléen est false, on utilise un *require*:

```
require(_to.send(amount))
```

ou pour le call, voir correction:

```
(bool sent, ) = _to.call{value: msg.value}("");
require(sent, "Failed to send Ether");
```

ALYRA

Solidity

Transferts d’eth - Recevoir

2 manières de recevoir de l’eth sur un contrat:

1) Utiliser receive() {} ou fallback() {}
Pour une automatisation des opérations à faire lors d’un dépôt d’eth sur le contrat:
Receive s'active quand on envoie des Eth au contrat, fallback quand on envoit des eth avec des données en plus

2) Utiliser payable comme tag de fonction:
Lors de l’appel d’une fonction en particulier, permet d’envoyer des eth au contrat en meme temps
Ces Eth peuvent être traités dans le corps de fonction

ALYRA

Nous pouvons transférer de l'ETH à des contrats intelligents sans appeler une fonction de paiement.

Pour gérer ces scénarios, lorsque quelqu'un envoie de l'ETH directement à un contrat, nous utilisons des fonctions de réception (receive) et de repli (fallback).

Si quelqu'un envoie de l'ETH à un contrat intelligent sans passer de calldata et qu'il y a une fonction receive(), la fonction receive() est exécutée. S'il n'y a pas de fonction receive(), la fonction fallback() est exécutée (et s'il n'y a ni receive ni fallback, la transaction est annulée).

Si des ETH sont envoyés à des smart contracts avec des calldata et que le sélecteur de fonction ne correspond à aucune fonction existante dans le contrat, la fonction fallback() est exécutée (là encore, si aucune fonction fallback n'est mise en œuvre, la transaction est annulée).

Untitled-1

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract Receive_FallBack {
    event ReceivedEth(uint256 amount);

    function fundme() public payable {
        emit ReceivedEth(msg.value);
    }

    receive() external payable  {
        fundme();
    }

    fallback() external payable {
        fundme();
    }

    //            is msg.data empty?
    //           /                 \
    //         yes                  no
    //        /                      \
    //  receive() exists?       is the function selector fundMe()?
    //      /      \                    /      \
    //    yes       no                no       yes
    //    /          \                /          \
    // receive()     fallback() exists?          fundMe()
    //                  /       \
    //                yes       no
    //               /            \
    //          fallback()    transaction is reverted
}
```

Try Pitch

Abstract vs contract vs interface

Contract
Toutes fonctions implémentées
Tout est compilé

Abstract:
Au moins une fonction n'est pas implémentée
Pas compilé

Interface:
Aucune fonction implémentée
Pas compilée

Abstract et interface ont leur fonction par défaut overridable
Dans un contrat, pour overrider une fonction, il faut qu'elle soit taguée « virtual »

ALYRA

Solidity

Untitled-1

```sol
pragma solidity 0.8.17;

interface interfaceB {
    function getNombre() external view returns(uint);
    function setNombre(uint _nombre) external;
}

contract B {
    uint nombre;

    function getNombre() external view returns(uint) {
        return nombre;
    }

    function setNombre(uint _nombre) external {
        nombre = _nombre;
    }
}
```

Untitled-1

```sol
pragma solidity 0.8.17;

import './B.sol';

contract A {
    interfaceB addressB;

    constructor(interfaceB _addressB) {
        addressB = _addressB;
    }

    function appelGetNombre() external view returns(uint) {
        return addressB.getNombre();
    }

    function appelSetNombre(uint _nombre) external {
        addressB.setNombre(_nombre);
    }
}
```

ALYRA

Abstract vs Interface

C'est la même chose que dans la plupart des autres langages de programmation orientés objet :

- L'interface ne déclare que des fonctions. Elle ne peut pas les mettre en œuvre.
- La classe abstraite peut déclarer des fonctions (comme l'interface) et les mettre en œuvre.

Les deux ne peuvent pas être instanciées et doivent être implémentées/héritées.

Untitled-1

```
interface IMyContract {
    // can declare, cannot implement
    function foo() external returns (bool);
}
```

Untitled-1

```
abstract contract MyContract {
    // can declare
    function foo() virtual external returns (bool);

    // can implement
    function hello() external pure returns (uint8) {
        return 1;
    }
}
```

Les contrats sont considérés comme des contrats abstraits si au moins une de leurs fonctions n'est pas implémentée. C'est la seule condition requise pour une classe abstraite. Par conséquent, elles ne peuvent pas être compilées. Elles peuvent cependant être utilisées comme contrats de base dont d'autres contrats peuvent hériter.

- Contrairement à d'autres langages, les contrats Solidity n'ont pas besoin d'un mot-clé abstract pour être marqués comme abstraits. Au contraire, tout contrat qui possède au moins une fonction non implémentée est traité comme abstrait dans Solidity. Un contrat abstrait ne peut être ni compilé ni déployé s'il n'a pas de contrat d'implémentation

Untitled-1

```
contract MyAbstractContract {
  function myAbstractFunction() public pure returns (string);
}
```

- Si un contrat hérite d'un contrat abstrait et n'implémente pas toutes les fonctions non implémentées, ce contrat sera également considéré comme abstrait

Untitled-1

```
//MyContract is also abstract
 contract MyContract is MyAbstractContract {
     function myAbstractFunction() public pure returns (string)
}
```

Ci-dessous, ce n'est pas abstrait car nous mettons en œuvre la fonction.

Untitled-1

```
contract MyContract is MyAbstractContract {
        function myAbstractFunction() public pure returns (string)
         { return "string value to return"; }
    }
```

- Un contrat abstrait peut avoir des fonctions implémentées et non implémentées.

Abstract vs Interface

Les interfaces ne peuvent avoir que des fonctions non implémentées. En outre, elles ne sont ni compilées ni déployées. Elles sont également appelées contrats abstraits purs.

- Les interfaces ne peuvent implémenter aucune de leurs fonctions. Toutes les fonctions d'interface sont implicitement virtuelles
- Les interfaces sont définies à l'aide du mot-clé Interface.
- Les interfaces ne peuvent pas hériter d'autres contrats ou interfaces (après Solidity 6.0.0, les interfaces peuvent hériter d'interfaces), mais d'autres contrats peuvent hériter d'interfaces.
- Les interfaces ne peuvent pas définir de constructeur
- Les fonctions d'une interface ne peuvent être que de type externe.
- Les interfaces ne peuvent pas avoir de variables d'état
- Pour l'instant, les interfaces ne peuvent pas définir de structs et d'enums, mais cela pourrait changer bientôt.

Solidity

Events

Déclaration des évènements comme suit
event nomEvent(params)

Et doivent être émis de cette manière
emit nomEvent(paramètres)

Ils servent à inscrire des logs dans la blockchain

Logs faciles à récupérer dans une application web via web3

ALYRA

Solidity

Tags de fonctions
On a vu le tag payable, il en existe deux autres

View:
Est inscrit dans une fonction qui ne fait aucune transaction sur la blockchain:
aucune modification d'état, donc de variables stockées, ni d'ajouts, pas d'écriture
En revanche il peut y avoir lecture dans la blockchain.
Très souvent utilisés dans des cas de getter

Pure:
Ne fait appel à rien dans la blockchain.
Souvent utilisés dans des librairies de computation, de calcul...

Ces deux tags indiquent que les fonctions n'ont pas besoin de valider la transaction via une clef privée

Solidity

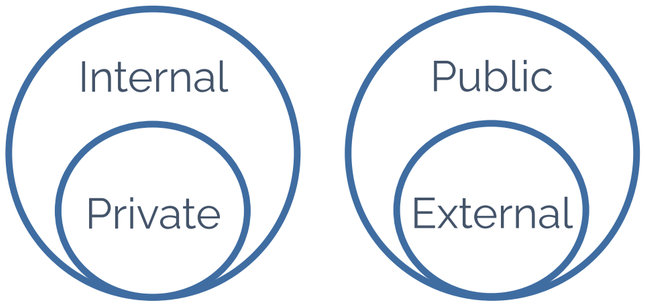
Visibilités

4 types de visibilités différentes: 4 types d'utilisations différentes

Valables pour fonctions et variables
1) Public
Comme son nom l'indique, cette visibilité permet à une fonction d'être publique, utilisable par tout le monde, visible de tous, de ce contrat et d'autres contrats.
Une variable avec cette visibilité possède un getter.

2) External
Toutes fonctions/var ayant cette visibilité n'est appelable qu'à l'exterieur du contrat.
Très utile à la création d'un contrat pour une librairie ou une interface, moins coûteuse qu'une public.

Solidity



Visibilités

3) Private
Fonctions et variables dont la visibilité ne se fait que pour les fonctions du contrat.
Plus grande sécurité en utilisant cette fonctions: si tout est public dans la blockchain, aucun autre contrat peut appeler directement ces fonctions

4) Internal
Fonctions accessibles depuis ce contrat, et tous les contrats dérivés (enfants).

On peut voir les choses de la manière suivante:
Private est un sous groupe d'internal
External est un sous groupe de public

Solidity

Héritage

Afin d'utiliser des contrats déjà existant
Ou des contrats que l'on utilise dans d'autres fichiers .sol
On utilise l'héritage :

1) Déclaration de l'import de fichier: import 'file.sol';

Cet import peut se faire depuis un chemin relatif, ou absolu :

Exemple:

- import 'path/to/file.sol';

ou

- import 'http://github.com/file.sol';

Solidity

Untitled-1

```
contract Owner {
    address i_owner;

    constructor() {
        i_owner = msg.sender;
    }

    modifier isOwner() {
        require(msg.sender == i_owner, "Not the owner");
        _;
    }
}

contract Test is Owner {
    uint nombre;

    function setNombre(uint _nombre) external isOwner {
        nombre = _nombre;
    }
}
```

Héritage

2) Utilisation de l'héritage:

2 cas possibles :

¤ contract son is contractFather{ }
On indique au contrat utilisé (le fils) qu'on hérite d'un contrat parent.
Ainsi, notre contrat va posséder toutes les fonctions du contrat parent.

¤ Using A for B;
où A est une librairie, B est un type
Exemple, librairie counters de open zeppelin
On indique qu'un type de variable hérite de fonctions déterminées par le contrat dont il hérite

ALYRA

Solidity

Constructor:

Sert à agir sur le contrat au moment du déploiement
Permet de lancer une fonction, initialiser une variable avec une certaine valeur, minter par exemple des ERC20...
Sont disponibles toutes les variables d'environnement

Syntaxe:

```
constructor() payable{
    depot = msg.value;
}
```

Par convention, on place le constructor entre la déclaration des variables et les premières fonctions.

```
Contrat
    Variables
    events
    constructor
    modifier
    fonctions
```

Contrat général

ALYRA

Solidity

Conditions

Deux moyens pour indiquer des conditions sur nos contrats

Require:
syntaxe: require(condition(s), « message si condition pas atteinte»);
Le require necessite que la ou les conditions soient validées, et si ça ne passe pas, la fonction revert, et émet un message d'erreur (optionnel).

Très utilisées pour verifier la sécurité des smart contracts.

If:

syntaxe: if(condition) { fonctionnalités }

Elle peut etre suivie d'un else.
Les fonctionnalités de votre fonction s'executent si les conditions sont validées

Solidity

Exercice

Faire un compte d'épargne sur la blockchain!

On a le droit de transférer au compte de l'argent quand on le souhaite

1- Ajouter un admin au déploiement du contrat.
2- Ajouter la condition suivante : l'admin ne peut récupérer les fonds qu'après 3 mois après la première transaction
3- On peut évidemment rajouter de l'argent sur le contrat régulièrement. Faire une fonction pour ça, et garder un historique (simple, d'un numéro vers une quantité) des dépôts dans un mapping.
4 – Mettre en commentaire les fonctions d'admin, et rajouter onlyOwner

ALYRA

Corrections

"Compte épargne "

Comment faire cet exercice?

- savoir faire un mapping, et lui allouer un id externe,
- comprendre le require sur le temps,

- envoyer des eth contenus dans le contrat vers son adresse,
- savoir utiliser onlyOwner

Penser à faire le dépot et le retrait

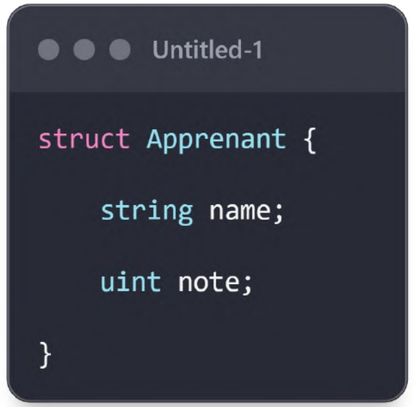
Untitled-1

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import "@openzeppelin/contracts/access/Ownable.sol";

contract Epargne is Ownable {

    mapping(uint => uint) deposits;
    uint depositId;
    uint time;
    // address immutable i_owner;

    // constructor() {
    //     i_owner = msg.sender;
    // }

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

    function deposit() external payable onlyOwner {
        require(msg.value > 0, "Not enough funds provided");
        depositId++;
        if(time == 0) {
            time = block.timestamp + 12 weeks;
        }
    }

    function withdraw() external onlyOwner {
        require(block.timestamp >= time, "Wait 3 months after the first deposit to withdraw");
        (bool sent, ) = msg.sender.call{value: address(this).balance}("");
        require(sent, "An error occured");
    }

}
```

Solidity

Struct

Une structure est composée de sous-types:



On considère cette déclaration comme un schéma de type personnalisé

Ainsi, de ce type personnalisé on peut sortir de nouveaux objets, un tableau de ces objets, les mettre dans des mappings...

ALYRA

Solidity

Struct

Plusieurs moyens de faire ces nouveaux objets

Apprenant beauPrenom= Apprenant("Cyril", 20); par exemple

Un objet est souvent amené à être stocké

apprenants[id]=beauPrenom;

ALYRA

Solidity

Mapping

Premier moyen de stockage

C'est une paire de clef valeur: mapping (address => uint) balance

Pour toute clef est retournée une valeur

Ces mapping peuvent associer n'importe quel type vers n'importe quel type (dont les struct)

On ne peut pas itérer sur les mappings. Mais moins couteux que les arrays

On peut mettre en cascade les mappings pour des structures complexes:

mapping (string => mapping (address=> Apprenant)) Satoshi

Ce mapping prends comme clé primaire le type de formation (par exemple "dev"), l'adresse d'un apprenant en clef secondaire, et renvoie les informations de l'apprenant

ALYRA

Solidity

Arrays

On peut faire des tableaux de

- uint : uint[]
- string: string[]
- booléen: bool[]
- ...
- de structures: struct[]

Déclaré comme suit: uint[] values : tableau dynamique

Si uint[3] values, alors c'est un tableau comprenant 3 uint, taille fixe

Il existe des tableaux multi dimensionnels (uint[][])

Ces tableaux multi dimensionnels (en taille fixe ici) sont écrits de manière inversée:

uint [2][3] est un tableau de taille 3 composé de tableaux de taille 2 en uint

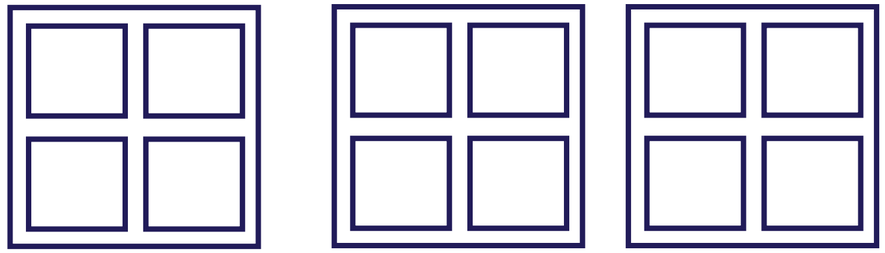
Solidity

Arrays

Premier index 0.

Il possède les attributs:
length, qui retourne la taille du tableau (ne pas oublier que le dernier index est donc length-1),
pop(), qui enlève du tableau le dernier élément,
push(x), qui ajoute x au tableau

uint [4][3]



ALYRA

Solidity

Arrays

- Deux types de tableaux
  - storage (persistant dans la Blockchain) (déclaré en dehors d'une fonction)
  - memory (dans le cadre d'une fonction)

Untitled-1

```
contract Test {
    uint[] public nombres;

    function addNombre(uint _nombre) public {
        nombres.push(_nombre);
    }

    function updateNombre(uint _index, uint _nombre) public {
        nombres[_index] = _nombre;
    }

    function deleteValue(uint _index) public {
        delete nombres[_index];
    }

    function getValue(uint _index) public view returns(uint) {
        return nombres[_index];
    }

    function deleteLastValue() public {
        nombres.pop();
    }

    function getLength() public view returns(uint) {
        return nombres.length;
    }
}
```

Untitled-1

```
pragma solidity 0.8.17;

contract Test {
    uint[] public nombres;

    function addNombre(uint _nombre) public {
        nombres.push(_nombre);
    }

    function getNombreX2() external view returns(uint[] memory) {
        uint longueurTableau = nombres.length;
        uint[] memory nombresX2 = new uint[](longueurTableau);
        for(uint i = 0 ; i < longueurTableau ; i++) {
            nombresX2[i] = nombres[i] * 2;
        }
        return nombresX2;
    }

    function sommeTableau(uint[] memory monTableau) external pure returns(uint) {
        uint longueurTableau = monTableau.length;
        uint somme = 0;
        for(uint  i = 0 ; i < longueurTableau ; i++) {
            somme += monTableau[i];
        }
        return somme;
    }
}
```

Solidity

Untitled-1

```
contract Test {
    enum etape { commande, expedie, livre } //pas de ;

    struct produit {
        uint _SKU;
        etape _etape;
    }

    mapping(address => produit) public CommandesClient;

    function commander(uint _SKU) external {
        produit memory p = produit(_SKU, etape.commande);
        CommandesClient[msg.sender] = p;
    }

    function expedier(address _client) external {
        CommandesClient[_client]._etape = etape.expedie;
    }
}
```

Try Pitch

Enum

Une dernière manière de gérer de la donnée:

L'énumération est un sélecteur dans une liste de possibilité données

Très utilisée pour gérer l'état d'un contrat ou d'un attribut

On la déclare comme suit:

enum Etat {etat1, etat2, etat3}

(sans le ;)

Puis ensuite, on peut créer une variable de cette énumeration:

Etat etatDuContrat

Cette variable prends par défaut la 1ere valeur, indexée 0, et on accède aux différents états par leur représentation en uint ou en expression littérale

require(uint(Etat.etat2) >= value);

etatDuContrat= Etat(_value);

ALYRA

Solidity

Exercice 1:

Faire un "deviner c'est gagné!"

Un administrateur va placer un mot, et un indice sur le mot
Les joueurs vont tenter de découvrir ce mot en faisant un essai

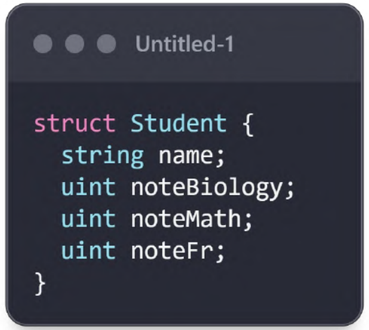
Le jeu doit donc
1) instancier un owner
2) permettre a l'owner de mettre un mot et un indice
3) les autres joueurs vont avoir un getter sur l'indice
4) ils peuvent proposer un mot, qui sera comparé au mot référence, return un boolean
5) les joueurs seront inscrit dans un mapping qui permet de savoir si il a déjà joué
6) avoir un getter, qui donne si il existe le gagnant.
7) facultatif (nécessite un array): faire un reset du jeu pour relancer une instance

ALYRA

Solidity

Exercice 2: Système de notation élève

Ecrire un smart contract qui gère un système de notation d'une classe d'étudiants avec addNote, getNote, setNote. Un élève est défini par une structure de données :



Les professeurs adéquats (rentrés en "dur") peuvent rajouter des notes. Chaque élève est stocké de manière pertinente. On doit pouvoir récupérer:

- la moyenne générale d'un élève
- la moyenne de la classe sur une matière
- la moyenne générale de la classe au global

On doit avoir un getter pour savoir si l'élève valide ou non son année.

ALYRA

23/05/2023

Demain on corrigera et on reverra l'héritage et les derniers détails

Merci de votre attention!

ALYRA