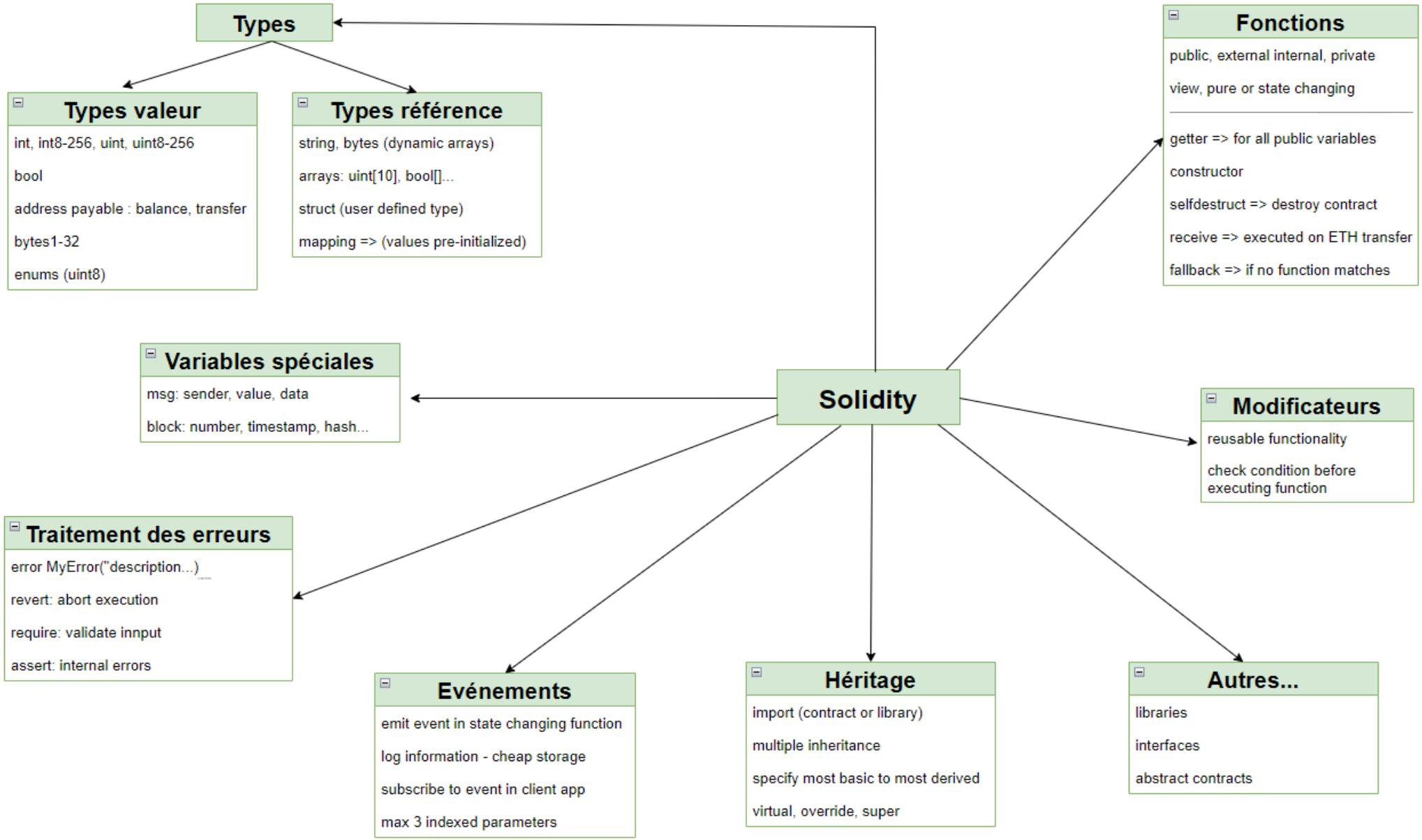

Développer et déployer des contrats intelligents sur Ethereum
Solidity

Solidity docs: https://solidity-fr.readthedocs.io/_/downloads/fr/latest/pdf/



Exemple d’un contrat intelligent

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import "../SomeOtherContract.sol";

contract Voting is SomeOtherContract {
    address public owner;

    struct Proposal {
        string name;
        uint voteCount;
    }

    struct Voter {
        bool allowedToVote;
        bool voted;
    }

    mapping(address => Voter) public voters;

    Proposal[] public proposals;

    event UserHasVoted(address indexed voter, uint proposal);
```

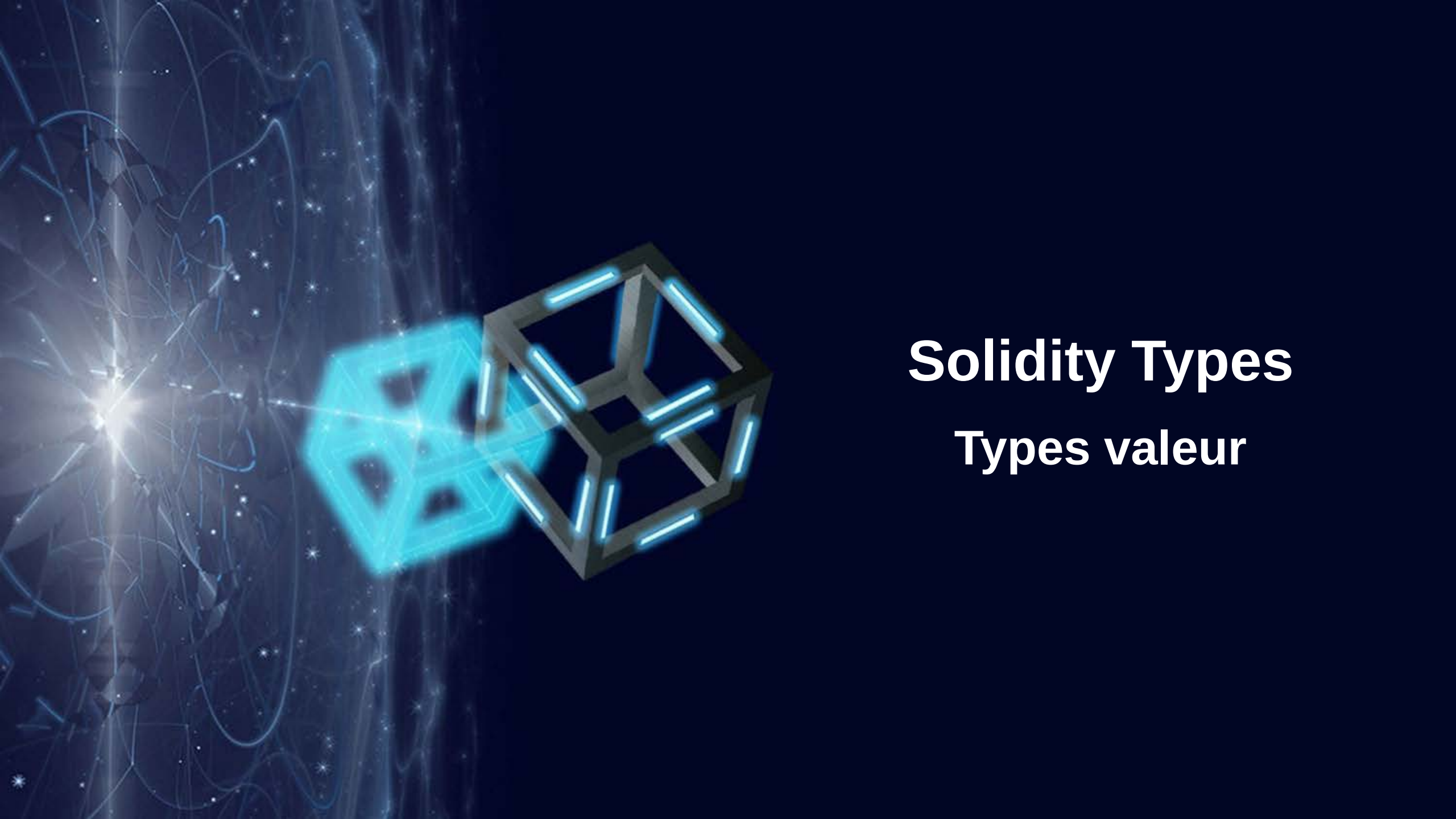
```solidity
    ///The user has already voted
    error UserHasAlreadyVoted(address voter);

    modifier onlyOwner() {
        require(msg.sender == owner, "You are not authorized");
        _;
    }

    constructor(string[] memory proposalNames) {
        owner = msg.sender;
        voters[owner].allowedToVote = true;
    }

    function giveRightToVote(address voter) external onlyOwner {
        if (voters[voter].voted) revert UserHasAlreadyVoted(voter);
        voters[voter].allowedToVote = true;
    }
}
```

Caractéristiques principales d'un contrat intelligent 1/2

- ***SPDX license identifier*** … – pas requis mais recommandé
- ***pragma solidity*** … – définit la version du compiler
- ***import*** … – importer d'autres fichiers
- Commentaires : // & /// - commentaires standard et commentaires NatSpec
- Variables d'état – stocké en permanence sur la blockchain
- Types valeur & référence : int, uint, bool, address, arrays, string, bytes…
- ***mapping*** (…) – pair clé-valeur pré-initialisée (comme table de hachage)
- ***struct*** … – type personnalisée qui regroupe plusieurs variables

Caractéristiques principales d’un contrat intelligent 2/2

- ***enum*** … - attribuer des noms à des constantes entières
- ***event*** … - pour avertir des applications clients
- ***modifier*** … - modifier la sémantique des fonctions d'une manière déclarative, appliquer certains vérifications et/ou validations à diverses fonctions
- ***error*** … - noms descriptifs pour les situations de défaillance, utilisé avec revert
- ***constructor*** (…) – exécuté une fois après le déploiement du contrat
- Fonctions
- Structure de contrôle (if / else) et boucles (for / while)
- Héritage - l'héritage multiple est supporté



Solidity Types

Types valeur

Solidity – les types de données

- Solidity est un langage statiquement typé - le type de chaque variable doit être spécifié
- Toutes les variables sont initialisées par défaut : int = 0 ; bool = false ; string = ""...
- Il n'y a pas de valeur nulle ou indéfinie
- Il existe 2 catégories de types : Types valeur et type référence
- Pour les types de référence, il faut aussi spécifier l'emplacement de la variable :
  - ***memory***
  - ***storage***
  - ***calldata***

Visibilité / types de valeur

Visibilité des variables d'état (stockées en permanence sur la blockchain):

- ***public:*** Une fonction getter est générée automatiquement par le compilateur. Peut être appelé en interne via le nom de la variable ou en externe via le getter
- ***internal:*** Accessible uniquement en interne (depuis le contrat on cours ou des contrats qui en dérivent). C'est le niveau de visibilité par défaut
- ***private:*** Visibles que pour le contrat dans lequel elles sont définies

Types de valeur:

- int, uint, bool, address, fixed size byte arrays (bytes1, bytes2... bytes32)
- Stockent directement la valeur des données & passent une copie de la valeur
- Déclaration: ***uint public myInt = 5; ... bool myBool = true***;

Entiers et boolean

- Valeur par défaut pour int/uint : 0
- Valeur par défaut pour bool : false
- uint8 de uint256 par incrément de 8 bits
- uint8 de 0 à 255 (2 ** 8 = 256) ... uint256 de 0 à (2 ** 256) – 1
- int8 de -128 à 127
- uint / int est un alias pour uint256 / int256
- L’exponentiation n’est disponible que pour les types non signé (uint) via le paramètre **

Dépassement / soupassement – mode sans contrôle

- Solidity (depuis version 0.8.0) lance une erreur et annule tous les modifications des variables d'états lorsque nous quittons la plage de valeurs définie (overflow ou underflow)

- Pour obtenir le comportement précédent (mode « wrapping ») il faut utiliser : ***"unchecked {...}"***

```
uint8 public myInt;

function decrement() public {
    myInt = 0;
    myInt--; //throws an error and reverts

    unchecked {
        myInt--;
        console.logUint(myInt);
    }
}
```

Type d'adresse

- Déclaration: ***address public myAddress;***
- Valeur par défaut : 0x0000... address(0)
- Chaque interaction sur Ethereum est basée sur des adresses
- Longueur d'une adresse : 20 octet (40 caractères)
- 2 types: ***address*** et ***address payable***
- Conversion de ***address*** à ***address payable*** doit être explicit via : payable(myAddress)
- Le type « contrat » peut être converti en address ou address payable : ***payable(address(this))*** => si le contrat peut recevoir de l'ETH (il doit avoir une fonction : ***payable fallback()*** ou ***receive()***)

Type d'adresse

- Membre pour address : ***balance***
- Membres pour address payable : ***transfer*** (annulation sur erreur), ***send*** (retourne false sur erreur)
- send et transfer fournis seulement 2300 gaz => utilise : ***receiverAddress.call{value: 1 ether}(“”);***

```
function transferEther() public {
    address payable address2 = payable(0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2);
    address addressContract = address(this);

    console.log("Balance of contract: ", addressContract.balance);

    if (addressContract.balance >= 1 ether) {
        //address2.transfer(1 ether);
        (bool success, ) = address2.call{value: 1 ether}("");
        require(success, "Transfer failed.");
    }
}
```

Tableaux d’octets de taille fixe et enum

Tableaux d’octets de taille fixe :

- bytes1... bytes32: contiennent une séquence de 1 - 32 octets
- Accès par indexage : myByte32[5] …
- Membre : length
- Exemple : ***bytes4 functionSig = bytes4(payload);***

Enum :

- Peut être utilisé pour créer un type défini par l’utilisateur
- Example: ***enum Directions { Left, Right };***
- Les options sont représentées par des valeurs entières non signées à partir de 0

Exemple : types de valeur

Créer un contrat intelligent qui contient les fonctionnalités suivantes:

- Créer diverses variables d’état pour différents types de valeurs
- Créer un constructeur qui peut recevoir de l'ETH et qui initialise certaines variables d’état
- Créer une fonction qui modifie l’une des variables d’état
- Créer une fonction qui renvoie le solde du contrat
- Créer une fonction qui permet d’envoyer de l'ETH en utilisant "**transfer**" ou "**send**"
- Créer une fonction qui permet d’envoyer de l'ETH en utilisant "**call**"

Exercice

Créer un contrat intelligent qui contient les fonctionnalités suivantes :

- Créer une variable d'état **public** de type **uint256** avec le nom "**myValue**"
- Créer une variable d'état **privé** de type **address** avec le nom "**myAddress**"
- Ajoutez un constructeur qui initialise "**myValue**" à 5 et "**myAddress**" à l'adresse du déployeur du contrat.
- Assurez-vous également que le contrat vous permet de recevoir des ETH pendant le déploiement
- Ajoutez une fonction qui vous permet de modifier la valeur de "**myValue**" et qui ne peut être appelée qu'en **externe**
- Créez une fonction qui contient un argument avec le nom "**withdrawAddress**" et qui vous permet d'envoyer l'intégralité du solde du contrat à "withdrawAddress". Assurez-vous d'utiliser un appel de bas niveau (**call**) pour transférer le solde du contrat.

Solidity Types

Types de référence

Types de référence

- structures, tableaux, mappages - bytes, strings (tableaux spéciaux)
- Les valeurs du type de référence peuvent être modifiées par plusieurs noms différents
- Il y a trois emplacements des données : stockage, mémoire et calldata. L'emplacement de données doit être toujours spécifié – sauf pour les variables d'état (stockage par défaut):
  - mémoire (memory) : temporaire, la durée de vie est limitée à l'appel de fonction externe, faible coût en gaz, utiliser pour le calcul intermédiaire et stocker le résultat en stockage
  - calldata : emplacement de données pour les arguments de fonction, non persistant, non modifiable - pareil comme mémoire
  - stockage (storage) : emplacement pour stocker les variables d'état, stockage persistant (durée de vie du contrat), coût élevé du gaz – éviter si possible

Types de référence – tableaux

- Peuvent avoir une taille fixe à la compilation (uint[3] myArr) ou dynamiques (uint[ ] myArr)
- Pour les tableaux public, le compilateur crée un getter. Seul un élément de tableau peut être retourné - l'indice doit être spécifié sur le getter
- Membre pour tous les tableaux : ***.length*** (retourne le nombre d'éléments)
- Membres supplémentaires pour les tableaux de stockage dynamiques et bytes: ***.push(), .push(value) et .pop()***
- Les tableaux de mémoire peuvent être crée avec le mot-clé « new » - ils ne peuvent pas être redimensionnés (la fonction .push() n'est pas disponible) :

  uint[ ] memory myArr = new uint[ ](7)
  myArr[0] = 1...

- Des littéraux peuvent être utilisées pour assigner des valeurs à une tableaux mémoire de taille fixe : ***uint8[3] memory myArr = [1, 2, 3]***

Types de référence - affectation

- Les affectations entre le stockage et la mémoire (ou calldata) créent toujours une copie
- Les affectations de mémoire à mémoire créent des références
- Les affectations du stockage à une variable de stockage local créent des références

```
uint[] arr1; //data location is storge - can be omitted

function arrayTest(uint[] memory arr2) public {
    arr1 = arr2; //this creates an independant copy

    uint[] storage arr3 = arr1; //assigns a reference
    arr3[0] = 5; //modifies arr1 through arr3
    console.log("Value of arr1[0]: ", arr1[0]);
    arr1[1] = 10; //modifies arr3 through arr1
    console.log("Value of arr3[1]: ", arr3[1]);
}
```

Types de référence – string & bytes

- Sont des tableaux dynamiques spéciaux
- bytes est semblable à bytes1[ ], mais moins cher (les données sont condensé en mémoire)
- bytes est utilisé pour les données en octets bruts et string pour les données de chaîne de caractères (UTF-8)
- Si la longueur peut être limité à un certain nombre d'octets, il est préférable d'utiliser les types bytes1 à bytes32, car ils sont moins chers
- String ne permet pas l'accès à la longueur ***(.length)*** ou aux éléments individuels par index
- A part de ***string.concat(s1, s2...)***, il n'y a pas d'autres fonctions de manipulation de chaîne
- Limiter l'utilisation du type string, car c'est très cher

Types de référence – structures & mappages

struct :

- Type défini par l'utilisateur qui regroupent plusieurs variables
- Les types d'une structure sont initialisés avec leurs valeurs par défaut

mapping :

- Les mappages sont comme des tables de hachage - chaque valeur est pré-initialisée
- Utilisation : mapping(typeClé => typeValeur) myMapping
- Exemple : ***mapping(address => uint) public balances;***
- Les mappages sont accessibles comme des tableaux, mais il n'y a pas d'exceptions d'index hors limite et toutes les paires clé/valeur possibles sont déjà initialisées => toutes les clés possible sont accessible

Types de référence – mappages

- Utilisez des mappages plutôt que des tableaux, car ils sont moins chers
- Le type clé ne peut pas être une structure, mappage ou tableaux (bytes et string sont autorisés)
- Les mappages ont toujours le stockage (storage) comme emplacement de données
- Les mappages ne peuvent pas être utilisés comme paramètre ou valeur de retour pour les fonctions publiques
- Un getter est créé pour les mappages qui sont utilisés comme variable d'état public – le type clé doit être utilisé comme paramètre pour le getter
- Les mappages et les structures sont souvent utilisés ensemble
- Le mappages ne peuvent pas être itéré

Variables spéciales

msg :

- Fournit des données sur l'appel de message courant
- msg.sender - expéditeur du message
- msg.value - nombre de wei envoyés avec le message
- msg.data - calldata complet

block :

- Fournit des informations générales sur la blockchain
- block.timestamp - timestamp du bloc en temps unix (secondes)
- block.chainid - id de la blockchain utilisé
- block.basefee - frais de base courants
- block.number - numéro du bloc courant
- blockhash(uint blocknumber) - hash du numéro de bloc indiqué

Types de référence – exemple de structures & mappages

```
contract Bank {
    struct Deposit {
        uint amount;
        uint timestamp;
    }

    struct AccountDetail {
        uint balance;
        uint numDeposits;
        mapping(uint => Deposit) deposits;
    }

    mapping(address => AccountDetail) public accounts;

    function deposit() payable public {
        ...
    }

    function withdraw(uint amount) public {
        ...
    }
}
```

Exemple : types de référence

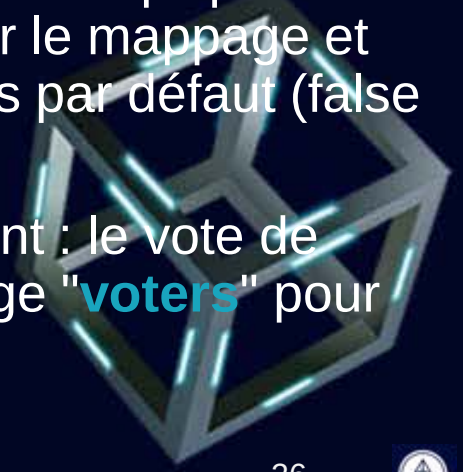
Créer un contrat intelligent qui contient les fonctionnalités suivantes:

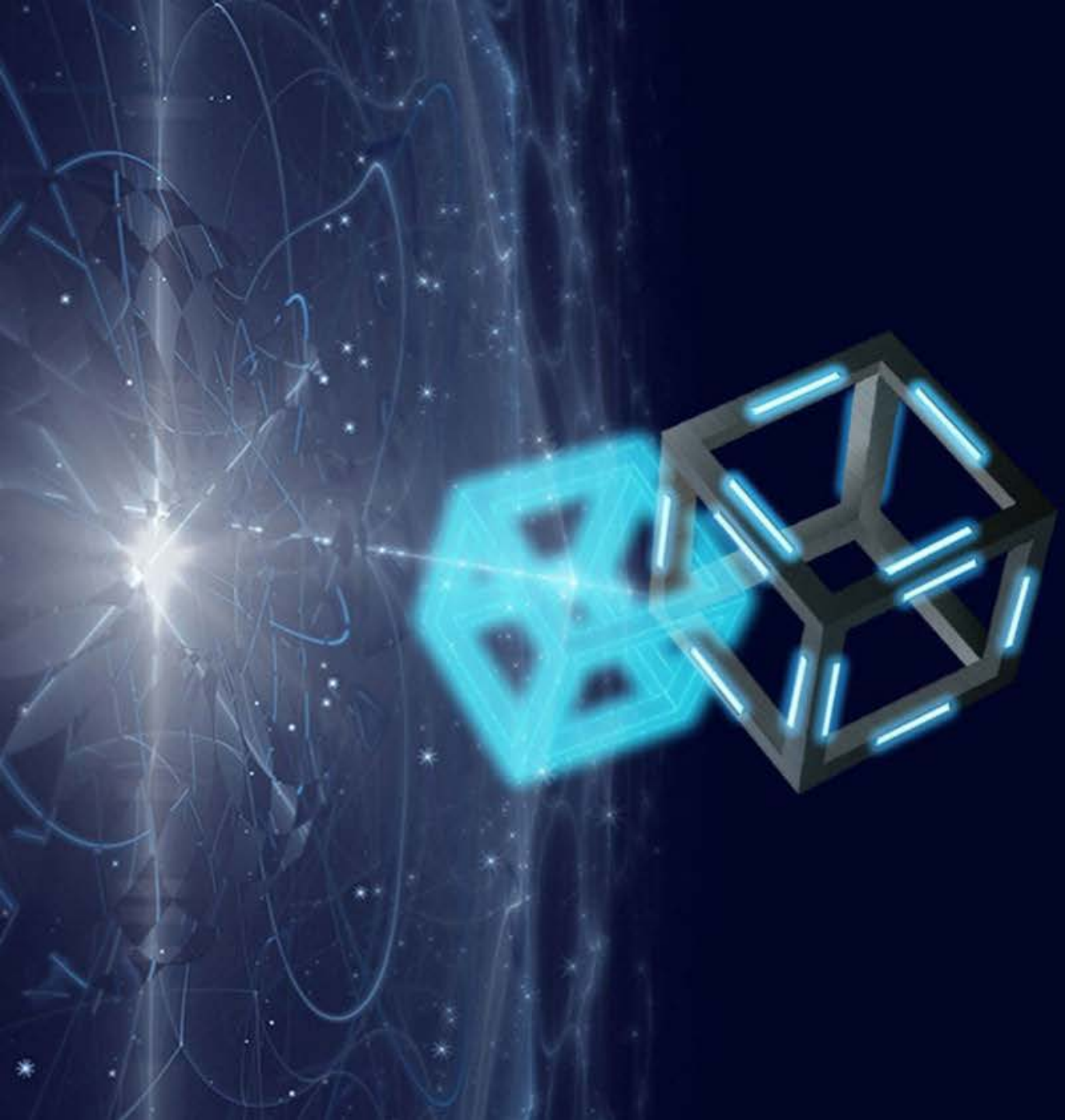
- Créer 2 variables d'état : une chaîne et un tableau dynamique qui est initialisé avec les valeurs 1, 2 et 3
- Créer une structure (**Deposit)** qui contient 2 types (**amount et timestamp**)
- Créer un mappage qui gère les soldes pour tous les utilisateurs et créer une deuxième mappage qui stocke le dernier dépôt pour chaque utilisateur
- Créer une fonction (**changeValues**) qui modifie la variable d'état de la chaîne et qui permet de modifier la première valeur de la variable d'état du tableau via un autre tableau défini localement.
- Créer une fonction (**deposit**) qui permet d'effectuer un dépôt et qui met à jour le le solde de l'utilisateur et le dernier dépôt.

Exercice

Créer un contrat intelligent qui contient les fonctionnalités suivantes :

- Créer une variable d'état public (**arr1**) qui est un tableau dynamique d'entiers signés avec les valeurs -1, 0 et 1
- Créer une fonction qui vous permet de modifier les valeurs d'arr1 via un autre tableau qui doit être déclaré dans cette fonction. Vous n'êtes pas autorisé à manipuler les valeurs d'arr1 en accédant directement aux valeurs d'arr1, comme : arr1[0] = 5;
- Créer une structure (**Voter**) qui définit 2 types : un type qui indique si l'électeur a déjà voté (**hasAlreadyVoted**), et un type **uint** qui indique le **vote** (vote = 1, 2, 3...)
- Créer un mappage (**voters**) qui assigne une structure **Voter** à toute adresse possible qui pourrait exister sur le réseau Ethereum. Toutes ces adresses doivent être accessibles sur le mappage et chaque adresse doit pointer vers une structure qui est initialisée avec ses valeurs par défaut (false pour booléen...)
- Créer une fonction qui permet à quiconque de voter. La fonction prend 1 argument : le vote de l'utilisateur (**uint userVote**) et met à jour les valeurs correspondantes du mappage "**voters**" pour cet utilisateur





Gestion des erreurs, fonctions, modificateurs & événements

Gestion des erreurs

- Les transactions sont atomiques - elles échouent ou réussissent dans leur ensemble
- Toutes les erreurs rétablissent l'état du contrat à ses valeurs initiales avant la transaction
- ***require(bool condition, [string memory message])*** - généralement utilisé pour valider les paramètres d'entrée - annulation des toutes les modifications apportées si la condition n'est pas remplie. Renvoie le gaz restant
- ***revert(string memory reason)*** ou ***revert MyCustomError()*** - annule l'exécution et annule les changements d'état. Il est préférable d'utiliser des erreurs personnalisées car c'est moins cher que de fournir des informations dans une chaîne => une explication détaillée de l'erreur peut être fournie via les commentaires NatSpec. Renvoie le gaz restant
- ***assert(bool condition)*** - utilisé pour les erreurs internes. Si une telle erreur se produit, le code est probablement défectueux et doit être corrigé. Annulation des modifications si la condition n'est pas remplie. Consomme tout le gaz

  Assert est également déclenché, si : division ou modolo par zéro, un index hors limites est accédé...

Exemple de gestion d'erreurs

```
/// the provided amount must be 1 ETH
/// @param provided The amount provided by the user
error WrongAmount(uint provided);

function buySomethingFor1ETH() public payable {
    if (msg.value != 1 ether) revert WrongAmount(msg.value);

    // Alternative way to do it:
    require(msg.value != 1 ether, "Incorrect amount.");

    // Perform the purchase...
}
```

Exemple : gestion d'erreurs

Créer un contrat intelligent qui contient les fonctionnalités suivantes:

- Créer une erreur personnalisée (**WrongAmount**) qui renvoie le montant fourni par l'utilisateur
- Créer une fonction qui retourne (en utilisant **require**) si la valeur transférée est différente de 1 ETH
- Créer une fonction qui retourne (en utilisant **l'erreur personnalisée**) si la valeur transférée est différent de 1 ETH
- Créer une fonction qui utilise une instruction **assert** pour vérifier la valeur d'un invariant

Fonctions

- Modification de l’état => nécessite une transaction – il faut payer des frais en gaz
- Lecture de l’état => simple appel de fonction – pas de frais
- Une fonction peut être déclarées ***view***, dans ce cas elle promet de ne pas modifier l’état
- Une fonction peut être déclarées ***pure***, dans ce cas elles promet de ne pas lire ou modifier l’état
- Les fonctions view peuvent appeler d’autres fonctions view et pure
- Les fonctions pure ne peuvent appeler que d’autres fonctions pure
- Les noms des paramètres de retour peuvent être omis
- Soit assigner explicitement les variables de retour, puis quitter la fonction (sans instruction de retour) ou fournir une ou plusieurs valeurs de retour avec le mot-clé ***return***

Fonctions lecture/écriture

Instructions considérées comme modifiant l'état (la fonction ne peut pas être vue ou pure) :

- Modification d'une variable d'état
- Emettre un événement
- Création d'un contrat
- Utilisation de l'autodestruction (**selfdestruct**)
- Envoi d'éther
- Appeler une autre fonction qui modifie l'état

Instructions considérés comme lisant l'états (la fonction ne peut pas être pure) :

- Accès aux variables d'état
- Accéder à ***address.balance***
- Accès à l'un des membres de **block ou msg**

Exemple de fonctions

```
function funcWrite(uint8 val) public {
    myInt = val;
}

function funcPure(uint a, uint b) internal pure returns (uint sum, uint prod) {
    sum = a + b;
    prod = a * b;
    //return (a+b, a*b);
}

function funcView() public view returns (uint) {
    (uint add, uint multiply) = funcPure(3, 5);
    return myInt + add + multiply;
}
```

Visibilité des fonctions & surcharge de fonction

Visibilité :

- ***public :*** peut être appelé en interne et en externe
- ***external :*** peut être appelé à partir d'autres contrats et via des transactions. Ne peut pas être appelée en interne. Moins cher que public => utiliser externe si vous êtes sûr que la fonction ne sera appelée qu'externe.
- ***private :*** accessible uniquement à partir du contrat en cours
- ***internal :*** accessible à partir du contrat en cours et des contrats dérivés. Appeler les fonctions en interne est moins cher et plus efficace que de les appeler en externe

Surcharge de fonction :

- Un contrat peut avoir plusieurs fonctions avec le même nom mais avec des paramètres différents

```
function myFunc(uint value) public view returns (uint) {...
function myFunc(uint value, bool flag) public view returns (uint) {...
```

Fonctions spéciales – getter & constructeur

getter :

- Le compilateur crée automatiquement une fonction getter pour toutes les variables d'état public
- Une fonction getter pour un tableau ne renvoie qu'un élément du tableau => l'élément correspondant à l'indice fourni
- Pour appeler une fonction getter d'un autre contrat, utilisez : ***someContract.myStateVar();***

constructor :

- Appelé une seule fois lors du déploiement du contrat
- Des arguments peuvent être fournis
- Pour initialiser les variables d'état - par exemple le propriétaire du contrat…
- Doit être marqué comme ***payable*** si les fonds doivent être transférés pendant le déploiement

Fonctions spéciales – autodestruction & réception d'Ether

selfdestruct :

- rend le contrat inutilisable
- l'historique des transactions reste sur la blockchain
- préciser l'adresse à laquelle les fonds restants seront envoyés => ***selfdestruct(owner);***

receive :

- Un contrat peut avoir une fonction de réception. Elle ne peut pas avoir d'arguments, ne peut rien retourner, doit avoir une visibilité externe et doit être payable
- ***receive() external payable {...}***
- La fonction est exécuté sur les transferts d'Ether
- Seulement 2300 unités de gaz sont disponible avec **send ou transfer** => peu d'opérations peuvent être effectuée (par exemple, l'émission d'un événement)

Fonctions spéciales – fonction de repli (fallback)

- Un contrat peut avoir une fonction de repli ayant une visibilité externe
- ***fallback() external [payable] - or:***
  fallback (bytes calldata input) external [payable] returns (bytes memory output)
- La fonction de repli est exécutée sur un appel au contrat si aucune des autres fonctions correspondent à la signature de la fonction spécifié
- La fonction de repli peut recevoir des données. Pour recevoir également d'Ether elle doit être marqué comme **payable**
- Si la fonction ***receive()*** n'existe pas, mais qu'une fonction de repli payable existe, elle sera appelé sur un transfert d'Ether
- Si ni ***receive()*** ni ***fallback() payable*** ne sont présents, le contrat ne peut pas recevoir d'Ether

Exemples de fonctions spéciales

```
constructor(uint8 valueForMyInt) payable {
    owner = payable(msg.sender);
    myInt = valueForMyInt;
    direction = Directions.Right;
}

function destroySmartContract() public onlyOwner {
    selfdestruct(owner);
}

receive() external payable {
    emit Received(msg.sender, msg.value);
}

fallback() external {
    myInt = 55;
}
```

Modificateurs (modifier)

- Les modificateurs sont utilisés pour ajouter des fonctionnalités réutilisable dans un manière déclarative
- Généralement utilisé pour vérifier certains conditions préalables à l'exécution de la fonction
- Des arguments peuvent être fournis
- Plusieurs modificateurs peuvent être appliqués à une fonction – séparée par des espaces blancs et exécuté dans l'ordre spécifié
- Le symbole _ dans le modificateur est remplacé par le corps de la fonction

```
address payable public owner;

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

function destroySmartContract() public onlyOwner {
    selfdestruct(owner);
}
```

Exemple : fonctions et modificateurs

Créer un contrat intelligent qui contient les fonctionnalités suivantes:

- Créer 2 modificateurs : Un qui donne accès uniquement au propriétaire du contrat et un qui permet l'exécution d'une fonction seulement après un certain temps dans le futur.
- Ajouter un constructeur qui initialise le propriétaire du contrat et une variable **timeLimit** à 1 minute après le déploiement du contrat.
- Créer une fonction qui modifie la valeur d'une variable d'état (**value**) et qui ne peut être exécutée que par le propriétaire du contrat et au plus tôt 1 minute après le déploiement du contrat.
- Créer une fonction qui prend un **uint** comme argument et retourne le produit de l'argument et la variable d'état "**value**".
- Créer une fonction qui prend **2 uint** comme arguments et retourne le produit et la somme de ces arguments.

Exercice

Créer un contrat intelligent qui contient les fonctionnalités suivantes :

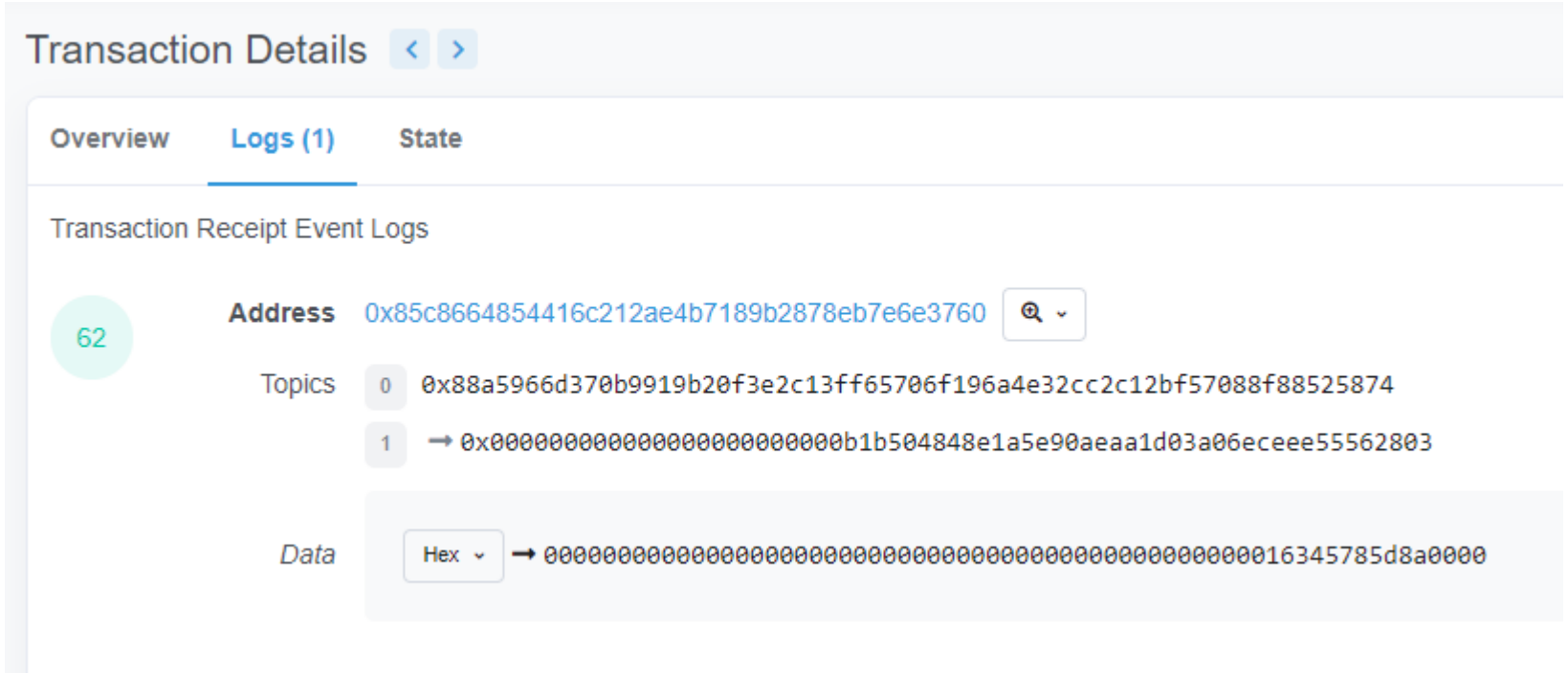
- Créer une variable d'état de type string avec le nom "**myString**". N'importe qui est autorisé à lire la valeur de "**myString**"
- Créer une fonction qui permet de modifier la valeur de "**myString**". Seul le titulaire du contrat est autorisé à appeler cette fonction et il ne devrait être possible d'appeler cette fonction que dans les 2 premières minutes après le déploiement du contrat.
- Si le propriétaire du contrat tente d'appeler la fonction 2 minutes (ou plus tard) après le déploiement du contrat, annulez la transaction et annulez toutes les modifications d'état à l'aide d'une erreur personnalisée. Fournir également une description détaillée de l'erreur à l'aide des commentaires **NatSpec**.
- Créer une fonction qui prend 2 arguments de type **uint** et qui retourne la somme de ces 2 arguments. La fonction doit être exécutable en interne et en externe et la fonction n'est pas autorisée à lire des variables d'état.

Evénements

- Les fonctions qui change l'état ne peuvent pas renvoyer de valeurs (elles renvoient uniquement le hash de la transaction) => par contre on peut utiliser des événements pour avertir des applications clients
- Les événements permettent d'enregistrer des informations spécifiques.
- Les arguments d'événement sont stockés dans le journal de transaction (**txn log**), une structure de données spéciale sur la blockchain. C'est une forme de stockage pas cher – en comparaison, le stockage des données sur la blockchain est très coûteux
- Ces journaux (logs) sont associés à l'adresse du contrat
- Il n'est pas possible d'accéder aux données du journal à partir du contrat
- Les applications clientes peuvent s'abonner aux événements et lire les données du journal
- Jusqu'à 3 paramètres peuvent être indexés et recherchés plus tard - ils sont ajoutés à une structure de données spéciale appelée « **topics** »

Exemple d'événement

```
event Received(address indexed from, uint amount);

receive() external payable {
    emit Received(msg.sender, msg.value);
}
```



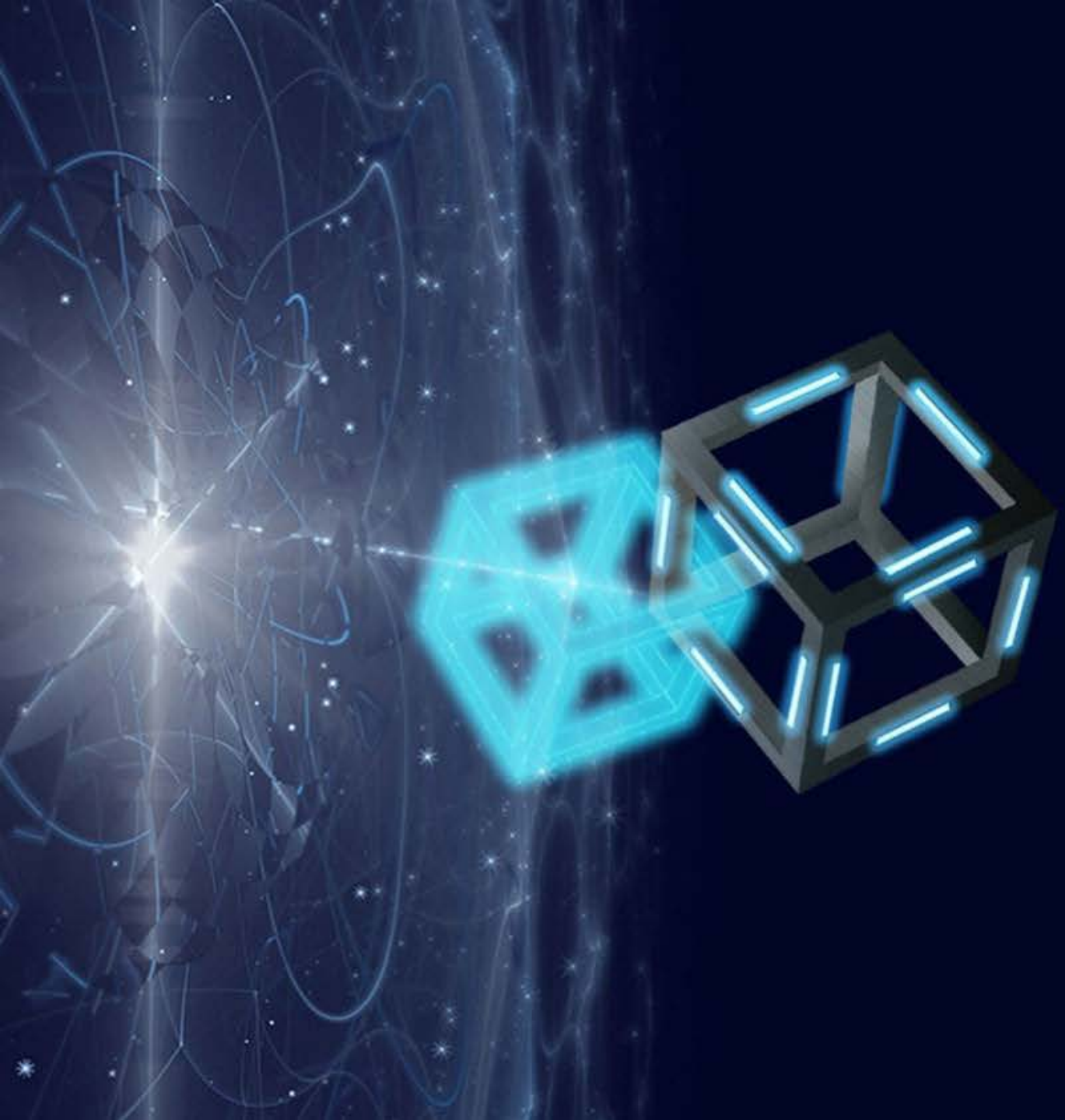
L'adresse « ***from*** » est indexée et donc listée comme "**Topic**". Topic 0 est pour la signature de l'événement

https://goerli.etherscan.io/tx/0x62c667f1cf3128756d115b31a1a8782b59a33c8333380f090ac92a71bca21064

Exemple : événements

Créer un contrat intelligent qui contient les fonctionnalités suivantes:

- Créer une variable d'état privé qui peut stocker les soldes de milliers d'adresses
- Créer un événement (**Deposit**) qui sera émis chaque fois qu'un utilisateur dépose des fonds. L'événement doit enregistrer l'adresse de l'utilisateur, qui doit être consultable et le montant déposé.
- Créer une fonction (**deposit**) qui permet aux utilisateurs de déposer des fonds. La fonction retourne si aucun fonds n'est déposé. Mettez à jour le solde de l'utilisateur et émettez l'événement "**Deposit**".



Héritage, bibliothèques, interfaces et contrats abstraits

Importer des fichiers

- Réutiliser les contrats et les bibliothèques existants, mieux organiser le code, faciliter la maintenance du code…
- ***import "filename"*** => tous les membres publiques du fichier sont importés => attention, cela peut pollue l'espace de nom global !
- ***import * as someNamespace from "filename"*** => tous les membres publiques sont accessibles via le nom spécifié
- ***import {function1 as alias, function2} from "filename"*** => importer uniquement les membres requis - un alias peut être spécifié

Héritage

- L'héritage multiple est supporté
- Utiliser ***"is"*** pour dériver d'un ou plusieurs contrats :
  contract C is A, B => B est le contrat le plus dérivé
- Un contrat dérivé peut accéder à tous les membres non privés
- Utiliser le mot-clé ***virtual*** sur une fonction pour permettre la modification de la fonction (overriding)
- Utilisez le mot-clé ***override*** pour remplacer une fonction virtuelle d'un contrat dérivé
- Une fonction dans un contrat dérivé peut être appelée en spécifiant le contrat :
  ContractName.functionName()
- Pour appeler la fonction un niveau plus haut dans la hiérarchie d'héritage, utilisez :
  super.functionName()

Exemple d'héritage

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract TestSolidity is ERC20 {

    constructor() payable ERC20("name", "symbol") {
        _mint(msg.sender, 1000 * 10**decimals());
    }

    // overide function from ERC20
    function name() public view virtual override returns (string memory) {
        return "TEST";
    }
}
```

Exemple : héritage

Créer un contrat intelligent qui contient les fonctionnalités suivantes:

- Créer un contrat **C2** qui hérite du contrat **C1** existant et qui surcharge la fonction **f3**
- Créer un contrat **C3** qui définit une fonction **f2** avec la même signature que celle définie dans **C1**
- Créer un contrat **C4** qui hérite de **C2 et C3** (dans cet ordre)
- Ajoutez une fonction (**callF3**) à **C4** qui renvoie le résultat de **f3** – quelle est la valeur de retour et pourquoi ?
- Ajouter une fonction (**callSuperF3**) à **C4** qui renvoie le résultat de l'appel à : **super.f3()** – quelle est la valeur de retour et pourquoi ?

Les bibliothèques

- Utiliser le mot-clé ***"library"*** pour crée une bibliothèque
- Fournit un ensemble de fonctions réutilisables - étendre la fonctionnalité des contrats intelligents
- Ne peuvent pas avoir des variables d'état
- Ne peuvent pas recevoir l'Ether
- Le code est exécuté dans le contexte du contrat appelant => le stockage du contrat appelant peut être accédé (seulement si les variables d'état sont explicitement fournies) !
- ***using libraryName for someType;*** => attache les fonctions de la bibliothèque au type spécifié.
- Le premier paramètre d'une fonction de la bibliothèque est souvent appelé ***"self"*** si la fonction peut être considérée comme une méthode de ce type

Exemple de bibliothèque

```
library Search {
    // usage: uint[] data = [1,2,3]; => data.indexOf(3);
    function indexOf(uint[] memory self, uint value) internal pure returns(uint) {
        for(uint i = 0; i < self.length; i++)
            if(self[i] == value) return i;

        return type(uint).max;
    }
}

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

    function ReplceOrAddValue(uint oldValue, uint newValue) public {
        uint index = data.indexOf(oldValue);
        if(index == type(uint).max)
            data.push(newValue);
        else
            data[index] = newValue;
    }
```

Les interfaces

- Défini par le mot-clé ***"interface"***
- Aucune fonction ne peut être implémenter => uniquement la déclaration de la fonction
- Toutes les fonctions doivent être externes - elles sont virtuelles par défaut
- Ne peuvent pas avoir des variables d'état, constructeur et modificateur
- Peut hériter d'autres interfaces
- Un contrat implémente une interface en utilisant le mot-clé ***"is"*** :
  contract SomeContract is SomeInterface

Exemple d'interface

```
interface IERC20 {
    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);

    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol

Les contrats abstraits

- Défini par le mot-clé ***"abstract"***
- Un contrat doit être marqué comme abstrait lorsqu'une ou plusieurs fonctions ne sont pas implémentées
- Un contrat peut être marqué comme abstrait même si toutes les fonctions sont implémentées
- Un contrat abstrait ne peut pas être instancié directement
- Les contrats abstraits sont utilisés comme classe de base
- Un contrat hérite d'un contrat abstrait en utilisant le mot-clé ***"is"*** :
  contract SomeContract is MyAbstractContract

Exemple d'un contrat abstrait

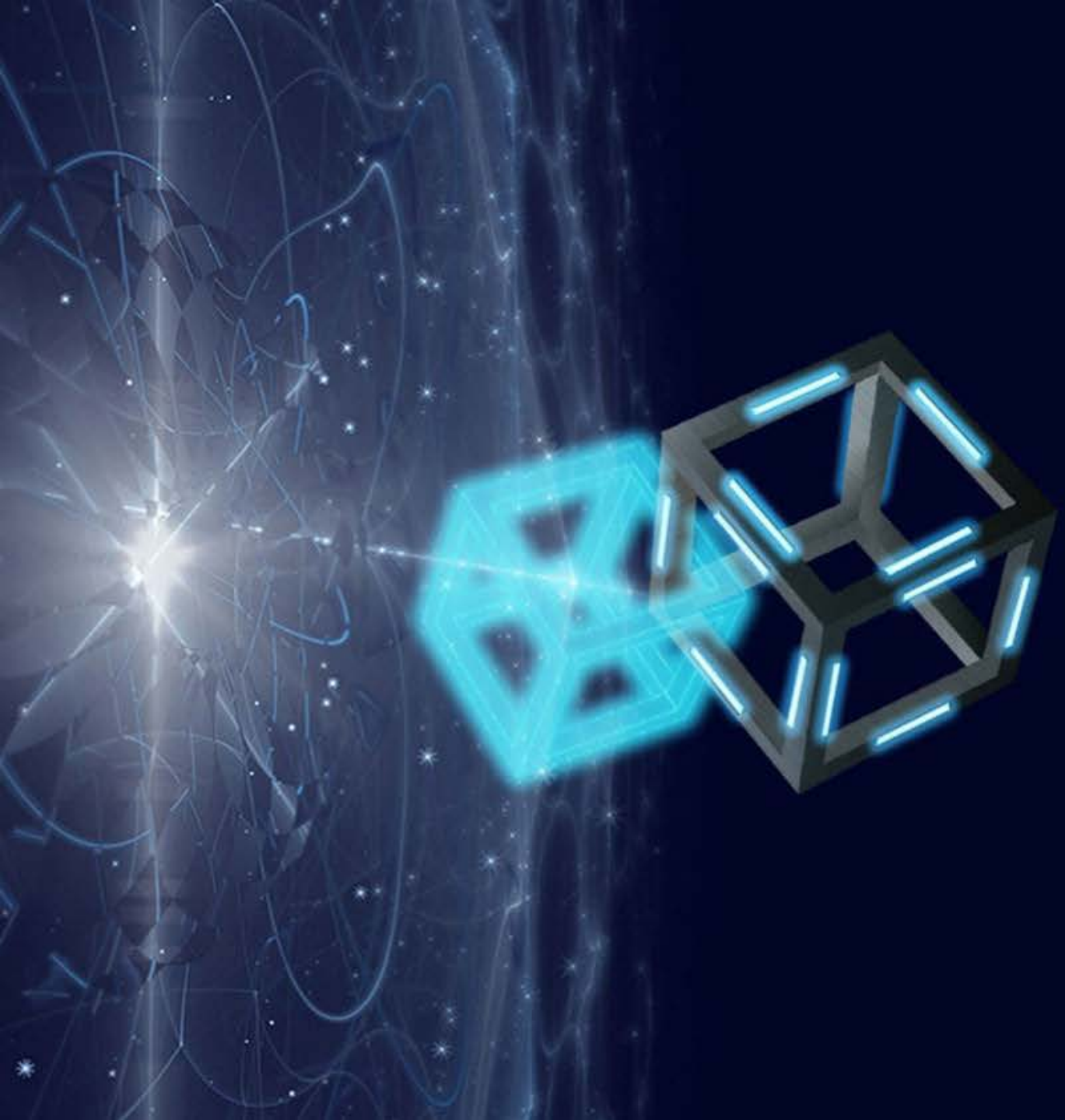
```
contract ERC20 is Context, IERC20, IERC20Metadata {
    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;

    uint256 private _totalSupply;
```

****

```
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Context.sol



Contrat de vote

Contrat de vote

Besoins du projet:

- Pendant la création du contrat, toutes les propositions sont crées et la fin des élections est défini
- Chaque proposition a un Id et détient le nombre de votes reçus
- Le contrat a un propriétaire (l'adresse qui à crée le contrat) qui peut donner à d'autres utilisateurs le droit de vote
- Chaque utilisateur (qui a le droit de vote) ne peut voter qu'une seule fois et uniquement avant la fin des élections
- Après chaque vote, un événement est émis avec l'adresse de l'électeur et la proposition votée
- À tout moment, n'importe qui peut vérifier la proposition actuelle et le nombre de votes actuel

Contrat de vote

Exercice:

- Ajouter une fonction ***endElection*** qui ne peut être appelée que par le propriétaire du contrat quand l'élection est terminée

- Retourner une erreur personnalisée de type : ***EndElectionAlreadyCalled*** si la fonction ***endElection*** est appelé plus d'une fois

- Émettre un événement ***ElectionEnded*** lorsque la fonction ***endElection*** est appelée et fournir le nom de la proposition gagnante et le nombre des votes comme arguments de l'événement