Construction d'un token Ethereum

Le smart contract minimum pour créer son token est le suivant :

```
pragma solidity ^0.4.16;

contract MyToken {
    // This creates an array with all balances
    mapping (address => uint256) public balanceOf;

    // Initializes contract with initial supply tokens to the creator
of the contract
    function MyToken(
        uint256 initialSupply
    ) {
        balanceOf[msg.sender] = initialSupply;              // Give the
creator all initial tokens
    }

    // Send coins
    function transfer(address _to, uint256 _value) {
        require(balanceOf[msg.sender] >= _value);           // Check if
the sender has enough
        require(balanceOf[_to] + _value >= balanceOf[_to]); // Check
for overflows
        balanceOf[msg.sender] -= _value;                    // Subtract
from the sender
        balanceOf[_to] += _value;                           // Add the
same to the recipient
    }
}
```

1) Mapping

```
mapping (address => uint256) public balanceOf;
```

Cette ligne de code va attribuer à chaque adresse ethereum un compte de notre token, vide au début. Le mot clé « public » sert à signifier que les comptes sont public, donc n'importe qui peut savoir le montant de tokens sur chaque adresses (et donc le propriétaire du compte)

2) Création du montant de tokens

```
    function MyToken(
        uint256 initialSupply
    ) {
        balanceOf[msg.sender] = initialSupply;              // Give the
creator all initial tokens
    }
```

Cette fonction va donner au créateur des tokens tout les tokens créer initialement

```
3) Transfère de tokens
   // Send coins
    function transfer(address _to, uint256 _value) {
        require(balanceOf[msg.sender] >= _value);           // Check
if the sender has enough
          require(balanceOf[_to] + _value >= balanceOf[_to]); //
   Check for overflows
        balanceOf[msg.sender] -= _value;                    //
Subtract from the sender
          balanceOf[_to] += _value;                           // Add
   the same to the recipient
       }
   }
```

Cette fonction permet de transférer des tokens de notre compte à une autre adresse. On précise comme paramètre l'adresse et le montant. On vérifie que nous avons suffisamment de tokens et on vérifie que l'adresse n'a pas trop de tokens (si l'adresse à déjà atteint le maximum de tokens et que l'on lui envoie encore des tokens, on risque d'avoir un problème de dépassement de mémoire et donc l'adresse qui recevrai les tokens risque de perdre tout ses tokens.

Ensuite on diminue de notre compte le nombre de tokens et on augmente réciproquement le compte de l'adresse où on envoie les tokens.

Ce code est le code minimum mais en l'état il ne permet pas de faire grand-chose.

La communauté Ethereum a défini un standard , le ERC20 Token Standard pour définir un token qui peut être utiliser et être échanger contre d'autres tokens sur le réseau Ethereum.

https://github.com/ethereum/EIPs/issues/20

Un token qui respecte le ERC20 Token Standard doit avoir les 6 fonctions et ses 2 events suivantes:

Fonctions :

- **totalSupply** : Le montant total de tokens en circulation
  function totalSupply() constant returns (uint256 totalSupply)
- **balanceOf** : Pour connaître le montant de token sur le compte d'un utilisateur avec comme adresse "_owner"
  function balanceOf(address _owner) constant returns (uint256 balance)
- **transfer** : Transfère la somme de token "_value" de notre compte vers l'adresse "_to"
  function transfer(address _to, uint256 _value) returns (bool success)

- **transferFrom** : Transfère la somme de token "_value" de l'adresse "_from" à l'adresse "_to". L'adresse "_from" doit au préalable avoir autoriser le transfère de cette somme (ou +) vers l'adresse "_to" avec la fonction "approve"
  function transferFrom(address _from, address _to, uint256 _value) returns (bool success)
- **approve** : autorise le transfère de la somme "_value" de notre compte vers l'adresse "_spender"
  function approve(address _spender, uint256 _value) returns (bool success)
- **allowance** : Pour connaître le montant restant de token que le compte "_spender" peut encore
  prendre du compte "_owner"
  function allowance(address _owner, address _spender) constant returns (uint256 remaining)

Events :

- **Transfer** : S'active lors d'un transfère de tokens de l'adresse "_from" l'adresse "_to" de la somme "_value".
  event Transfer(address indexed _from, address indexed _to, uint256 _value)
- **Approval** : S'active lors de l'autorisation de transfère de l'adresse "owner" vers l'adresse "_spender"
  event Approval(address indexed _owner, address indexed _spender, uint256 _value)

https://github.com/ConsenSys/Tokens/blob/master/contracts/Token.sol

et

https://github.com/ConsenSys/Tokens/blob/master/contracts/StandardToken.sol

Token.sol + StandardToken.sol est une implémentation de ERC20 Token Standard.

Ces 6 fonctions et ces 2 events permettent de faire un minimum de chose mais ce n'est pas suffisant pour notre token car on souhaite qu'il ait un nom etc

C'est pour cela que notre token doit également avoir des options supplémentaires pour être utilisable par des humains.

HumanStandardToken

- **Un nom** pour le token, dans notre cas Peculium
  string public name;
- **Un nombre de decimal** à afficher
  uint8 public decimals;
- **Un symbole**, dans notre cas PCX
  string public symbol;

- **version** du token
  string public version = 'H0.1';

Ajout de la fonction approveAndCall

Cette fonction autorise l'adresse "_spender" à prendre "_value" token sur notre compte (fonction allowed) et envoie un message à cette adresse pour le lui signaler (event Approval).

function approveAndCall(address _spender, uint256 _value, bytes _extraData) returns (bool success)

Avec ceci, on peut déjà faire pas mal de choses mais on peut aller plus loin.

Choix du contrôle du token :

```
contract owned {
    address public owner;

    function owned() {
        owner = msg.sender;
    }

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

    function transferOwnership(address newOwner) onlyOwner {
        owner = newOwner;
    }
}
```

Ce contract basique possède une variable owner, et une fonction de type « modifier » onlyOwner . C'est un modificateur de fonctions.

Nous avons ensuite une fonction transferOwnership qui transfère le propriétaire du contract de token à une autre personne. Cette fonction ne peut être activée que par le propriétaire actuel.

On peut ensuite faire en sorte que notre smart contract hérite de ce contract comme ceci :

```
contract MyToken is owned
```

Il faut bien entendu changer les fonctions en conséquence.

Gel de compte :

Cela peut être pratique de pouvoir geler un compte de tokens, et que seul le propriétaire du smart-contrat puisse dégeler un compte.

```
mapping (address => bool) public frozenAccount;
event FrozenFunds(address target, bool frozen);

function freezeAccount(address target, bool freeze) onlyOwner {
```

```
    frozenAccount[target] = freeze;
    FrozenFunds(target, freeze);
}
```

On créer un compte geler pour chaque adresse sur la blockchain (comme précédemment avec les comptes classiques

Puis on créer une fonction qui gel un compte et le signal à son propriétaire.

On peut construire une fonction de dégel de la même manière.

Système automatique de vente et d'achat contre de l'ether

On peut souhaiter que notre token ai une vrai valeur en ether. Il est possible de faire cela et de paramétrer notre smart-contrat pour qu'il achète et vente automatiquement nos token contre une valeur en ether prédéfini.

```
uint256 public sellPrice;
uint256 public buyPrice;

function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner
{
    sellPrice = newSellPrice;
    buyPrice = newBuyPrice;
}
```

Tout d'abord on définit un prix de vente et un prix d'achat. Ils sont définis en wei. 1 ether = 1000000000000000000 wei (1 suivit de 18 zéros). On définit également une fonction qui permet de changer les prix de vente et d'achat.

On peut également mettre en place un système qui change les prix de manière dynamique mais cela est plus compliqué.

On définit ensuite des fonctions de vente et d'achat :

```
function buy() payable returns (uint amount){

    amount = msg.value / buyPrice;                    // calculates the
amount
    require(balanceOf[this] >= amount);               // checks if it
has enough to sell
    balanceOf[msg.sender] += amount;                  // adds the
amount to buyer's balance
    balanceOf[this] -= amount;                        // subtracts
amount from seller's balance
    Transfer(this, msg.sender, amount);               // execute an
event reflecting the change
    return amount;                                    // ends function
and returns
}
```

Cette fonction permet à un utilisateur d'acheter des tokens en échange d'ether à notre smart-contrat.

```
function sell(uint amount) returns (uint revenue){
    require(balanceOf[msg.sender] >= amount);         // checks if the
sender has enough to sell
    balanceOf[this] += amount;                        // adds the
amount to owner's balance
    balanceOf[msg.sender] -= amount;                  // subtracts the
amount from seller's balance
    revenue = amount * sellPrice;
    require(msg.sender.send(revenue));                // sends ether to
the seller: it's important to do this last to prevent recursion attacks
    Transfer(msg.sender, this, amount);               // executes an
event reflecting on the change
    return revenue;                                   // ends function
and returns
}
```

Cette fonction permet à un utilisateur de vendre des tokens en échange d'ether à notre smart-contrat.

Nous devons donner suffisament d'ether à notre smart-contract pour qu'il puisse acheter tous les tokens en circulation.

Payement des frais de transactions

Les frais de transactions sont payés en ether sur le réseau Ethereum. Si on souhaite que les utilisateurs n'ai pas besoin de se casser la tête avec tout ça, on peut faire un système qui va automatiquement calculer les frais de transactions et les payé à chaque fois que l'utilisateur va utiliser ses tokens sur le réseau.

```
uint minBalanceForAccounts;

function setMinBalance(uint minimumBalanceInFinney) onlyOwner {
     minBalanceForAccounts = minimumBalanceInFinney * 1 finney;
}
```

On définit un montant minimum sur une adresse et une fonction pour changer ce montant minimum.

```
// Send coins
function transfer(address _to, uint256 _value) {
    // ...
    if (msg.sender.balance < minBalanceForAccounts)
        sell((minBalanceForAccounts - msg.sender.balance) / sellPrice);
}
```

On change la fonction de transfère de tel manière qu'un utilisateur ne peut pas envoyer de token si il ne peut pas payer les frais en ether (il paye les frais en ether en échangeant ses tokens contre des ethers avec le système vu au-dessus par exemple)

Nouveau tokens

On peut être amené à vouloir créer plus de tokens que ceux créer de base. Il existe plusieurs manières pour cela :

- Créer un système de proof of work, qui donne des tokens à chaque utilisateur qui fait un calcul informatique
- Faire que son token soit mintable, donc que le propriétaire du smart-contrat du token puisse ajouter des token comme il le souhaite.
- On peut également faire en sorte de donner des frais de transactions en token (en plus de ceux en ether) aux mineurs

```
contract token { function mintToken(address receiver, uint amount){  }
}
// ...
    function () {
        // ...
        tokenReward.mintToken(msg.sender, amount / price);
        // ...
    }
```

Ce code permet de rendre notre token mintToken.

Système d'horloge

Pour pouvoir faire des interactions avec la blockchain de manière plus évolué (par exemple ne vendre nos tokens qu'entre une certaine date et une autre) on veut pouvoir gérer les variables temporelles. On peut faire cela grâce à http://www.ethereum-alarm-clock.com/ , un smart contract qui indique l'heure en permanence.

On fait appel à ce smart-contract à l'adresse
0xe109EcB193841aF9dA3110c80FDd365D1C23Be2a

Ou bien *0xb8Da699d7FB01289D4EF718A55C3174971092BEf*

Sur le serveur de test

Ce smart-contrat permet d'executer notre smart-contrat plus loin dans le temps. On lui envoie l'adresse de notre smart-contrat, une signature (calculé avec la clé privée de notre smart-contrat) et une date dans le future (ou plutôt un nombre de block calculé dans le futur). Nous devons également payer les frais de transactions et le smart-contract ethereum-alarm-clock

Nous avons cette approximation :

- 1 hour duration (60 minutes): 212 blocks
- 1 day duration (1440 minutes): 5082 blocks
- 1 week duration (10,800 minutes): 38,117 blocks
- 1 month duration (44,640 minutes): 157,553 blocks

Ethereum Natural Specification Format

La dernière chose importante est le style de format de commentaires pour la documentation.

On va ajouter les variables suivantes en tant que commentaire dans le code de notre smart-contrat. Cela va permettre à la blockchain, au portefeuille et aux développeurs de mieux comprendre notre smart-contrat

@title: Un titre et une définition

@author: Le nom de l'auteur du contrat

@notice: description de comment marche le contrat et ce qu'il fait

@dev: documentation pour les développeurs.

@param: suivit du nom du paramètre, permet de décrire un paramètre du smart-contrat

@return: permet de décrire ce que retourne une fonction du smart-contrat