Lab 3 – Estructuras de datos en Solidity: *arrays* y *mappings*

En esta sesión vamos a ver cómo almacenar datos en Solidity utilizando arrays y mappings. Estudiaremos cómo se comportan respecto al consumo de gas. Recuerda que en Solidity **ejecutar código cuesta dinero:** intenta mantener el código tan simple y minimalista como sea posible y hacerlo tan eficiente como puedas. Los accesos a *storage* son particularmente costosos.

1. Una hucha multiusuario

Vamos a extender la hucha que creamos en el ejercicio anterior proporcionando funcionalidades adicionales. Será una *hucha multiusuario*. Utilizaremos un único contrato para almacenar ether de varios usuarios (a los que llamaremos *clientes*), garantizando que ningún cliente pueda retirar dinero depositado por otro cliente en la hucha. Para ello debemos mantener la información sobre los clientes en una estructura de datos en *storage*.

El contrato que vamos a desarrollar debe tener las siguientes funciones **`external`**:

- **`function`** `addClient(`**`string memory`** `name)`**`external payable`**
  Esta función se utiliza para añadir un nuevo cliente a la hucha. El contrato debe almacenar la siguiente información sobre los clientes: sus **nombres, direcciones** (obtenidas de **`msg.sender`**) **y la cantidad** depositada en la hucha (obtenida de **`msg.value`**). Esta función es **`payable`** porque los clientes pueden depositar una cantidad de ether inicial en su cuenta cuando se registran en la hucha.

  Esta función debe comprobar que el nombre es un string no vacío (es decir, su longitud es mayor a cero) antes de añadir un cliente, revirtiendo la ejecución en caso contrario. Utiliza una sentencia **`require`** para esta comprobación.

- **`function`** `deposit()`**`external payable`**
  Esta función la utilizan los clientes para depositar cantidades adicionales de ether en sus cuentas. Los clientes se identifican con el valor de **`msg.sender`** y la cantidad a depositar se proporciona en **`msg.value`**.

- **`function`** `withdraw(`**`uint`** `amountInWei)`**`external`**
  Esta función la utilizan los clientes para retirar su dinero de la hucha. Debe comprobar que el cliente que la invoca existe en la estructura de datos interna y que ha depositado suficiente ether en la hucha. Si las comprobaciones son correctas, debe transferir la cantidad al remitente (**`msg.sender`**), o revertir la ejecución en caso contrario.

- **function** getBalance()**external view returns** (**uint**)
  Esta función debe devolver el saldo del cliente que invoca esta función en la hucha. Si **msg.sender** no es un cliente registrado en la hucha, debe revertir la ejecución.

Debes crear **dos implementaciones de este contrato: una utilizando un array y otra utilizando un mapping.**

Ejercicio 1. *Implementa las funciones anteriores en un contrato llamado* ***PiggyArray*** *que utilice un array para almacenar la información sobre clientes. Puedes implementar una función* ***internal*** *para realizar la búsqueda secuencial de un* `address` *en el array, y llamarla desde las funciones externas que lo necesiten.*

Ejercicio 2. *Implementa estas mismas funciones en otro contrato llamado* ***PiggyMapping*** *que utilice un mapping para almacenar la información sobre los clientes. Recuerda que un mapping es una lista asociativa: piensa qué dato utilizarás para la clave del mapping y cómo se almacenarán los datos en el mapping.*

Nota: los elementos no inicializados en un mapping contienen strings de longitud cero. Puedes comprobar si existe un elemento para un valor de clave `key` *en un mapping de esta forma:*

```
bytes(mymapping[key].name).length > 0
```

donde `name` *es un campo de tipo string en el contenido de* `mymapping`*.*

Si has escrito los dos contratos en el mismo fichero fuente, debes desplegarlos (deploy) uno a uno. Puedes seleccionar el contrato a desplegar utilizando una lista que está justo encima del botón "deploy".

2. Estudio del consumo de gas

En esta sección vamos a estudiar el consumo de gas en ambas implementaciones. Compila y despliega ambos contratos y completa los ejercicios siguientes:

Ejercicio 3. *Añade tres usuarios a ambos contratos mediante la función* ***addClient*** ***utilizando cuentas Ethereum diferentes para cada una de las llamadas.*** *Rellena una tabla como la que aparece a continuación. Explica los resultados que has obtenido respecto al código que has programado.*

addClient			***Execution cost***	
Name	***Amount***	***Address***	***PiggyArray***	***PiggyMapping***
Juanito	*10.000*			
Jorgito	*20.000*			
Jaimito	*30.000*			

Ejercicio 4. *Ahora utiliza la función* ***getBalance*** *para obtener el saldo de algunos clientes y registra el coste de ejecución en gas de esta función. Utiliza una nueva cuenta Ethereum que no hayas utilizado antes para obtener el balance de "Silvestre", un usuario que no se ha añadido a ninguna de las dos huchas. Rellena la siguiente tabla:*

getBalance		***Execution cost***	
Name	***Address***	***PiggyArray***	***PiggyMapping***
Juanito			
Jaimito			
Silvestre			

Explica los resultados obtenidos respecto al código que has programado. Qué piensas que ocurriría si tuvieras 100 clientes registrados en las huchas? Y si tuvieras 1000 clientes?

3. Recorriendo los datos de un mapping

Ahora debes adaptar el código de `PiggyMapping` para que se puedan recorrer los datos almacenados en el contrato. Como los mappings no son recorribles, debes añadir un array que solo se utilizará cuando se necesite recorrer los datos almacenados. Si el mapping se define como **`mapping`**`(KeyType => ValueType)`, lo habitual es tener un array de elementos de tipo `KeyType`.

Ejercicio 5. *Haz una copia del contrato `PiggyMapping` y renómbrala como `PiggyMapping2`. Después añade la siguiente función a `PiggyMapping2`:*

- ***`function`*** *`checkBalances()`****`external view returns`*** *`(`****`bool`***`)`*
  Esta función debe calcular la suma de los saldos de todos los clientes almacenados en la hucha. Debe devolver ***`true`*** *si la suma es igual al saldo del propio contrato (`address(`****`this`***`).balance`*, *o* ***`false`*** *en caso contrario.*

Para que esta función funcione correctamente, se debe actualizar el contenido del array cada vez que un cliente se añada a la hucha.