El dia de hoy comenzamos a codear en Solidity, en forma de prueba, hicimos un código que nos permite hacer transacciones entre wallets, permitiendo seleccionar el monto que deseemos, junto a esto, agradecerá al usuario si el monto a transferir es mayor a 10 ether, en caso contrario, no le permitirá realizar la transacción.

Explicación de la sintaxis de Solidity (Código completo al final del documento):

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;
```

Al crear un documento .sol, necesitaremos aclarar la licencia y la versión en cuestión, en el caso de la versión podemos poner un rango de versiones, por ej aquí declaramos que usaremos una versión entre la 0.7.0 como mínimo, hasta un máximo de la 0.9.0

```
contract CuentaComunitaria {
}
```

Esta línea define nuestro smart contract, en este caso llamado “CuentaComunitaria”. Este contrato es como una clase en otros lenguajes como Java, esta va a contener todo el código que va a ejecutar nuestro programa en la blockchain. Aquí es donde vamos a escribir las funciones y las variables que definirán cómo funciona nuestro contrato y cómo interactúa con la blockchain.

Explicación de las **posibles variables**/datos básicos que utilizaremos y que hace cada una:

```
bool estado;

int256 número;

uint256 número;

address myAddress;
```

```
string saludo;
```

(Aclaro, el bool no lo vimos en clase, al igual que el int, pero los pongo para ya conocerlos)

bool: Almacena un valor booleano, en otras palabras, si es Falso o Verdadero, también dicho 0 y 1, siendo este último el verdadero.

int256: Almacena un número positivo o negativo.

uint256: Almacena un número positivo.

(Por lo que encontré, como dato, el 256 representa el largo máximo que puede contener de números, en este caso, es 2 a la 256)

address: Almacena una dirección de Ethereum, normalmente wallets.

string: Almacena cadenas de texto.

Funciones (Pero no literalmente Functions) básicas:

```
function miFuncion() public {
    // código
}
```

Puedes almacenar funciones como variables y pasarlas como parámetros.

Return

La palabra **return** se utiliza para devolver un valor desde una función. Es como en otros lenguajes de programación.

```
function getBalance() public view returns(uint256) {
    return address(this).balance;
}
```

Visibilidad de las **funciones** (external, public, internal, private):

Estas palabras clave definen quién puede llamar a las funciones o acceder a las variables:

external: Solo se puede llamar desde fuera del contrato (por otros contratos o usuarios).
public: Se puede llamar tanto dentro como fuera del contrato.
internal: Sólo puede ser llamada desde dentro del contrato o contratos derivados.
private: Sólo puede ser usada dentro del mismo contrato.

Modificadores de **funciones** (**view**, **pure**):

Estos modificadores indican si una función puede modificar o leer el estado del contrato, en estos casos tenemos view, pure y no poner un modificador.

view:La función **puede leer** el estado del contrato, pero **no lo modifica**. Se usa cuando solo necesitas consultar variables o balances.
pure:La función **no puede leer ni modificar** el estado del contrato. Solo trabaja con los parámetros que recibe y no interactúa con variables o datos del contrato.
nada: Cuando una función no tiene ningún modificador (pure o view), significa que puede modificar el estado del contrato.

msg.:

msg.sender: Identifica al remitente de la transacción.
msg.value: Indica la cantidad de Ether enviada con la transacción.
msg.data: Contiene los datos enviados en la transacción en formato de bytes.
msg.sig: es un valor de 4 bytes que representa el "signature" (firma) del mensaje.

Errores

Se pueden definir errores personalizados utilizando la palabra clave **error**. Esto permite crear errores con mensajes.

```
error FaltaPlata(uint256 cantidaFaltante);
```

Uso:

```
if (msg.value < CANTIDAD_MINIMA) revert FaltaPlata(CANTIDAD_MINIMA - msg.value);
        depositar();
        emit Agradecimiento(msg.sender);
```

En este caso, por ejemplo, lo que ocurre es que en caso de que se envíen menos de la cantidad que declaramos como mínima (10 ether), hace que se detenga la función, y da nuestro código de error personalizado, en este caso seria el monto restante faltante para poder realizar la transacción.

receive() external payable { }

Es una función especial que permite que el contrato reciba Ether. Esta función no tiene nombre ni argumentos, y se ejecuta cuando el contrato recibe fondos directamente (sin llamar a una función específica).

external significa que solo puede ser llamada desde fuera del contrato.
payable permite que la función reciba Ether.

Explicación estructuras compuestas:

```
mapping(X => Y) Z;
```

Mapping: Esta estructura lo que hace es asignar a, en este caso, todas las X, que tendrán una característica Y. Luego, la Z será el nombre, aquí un ejemplo “real”:

```
mapping (address depositante => uint256 monto) contribuyentes;
```

En este caso, el mapping indica que para cada dirección (**address**, que representa al depositante), habrá un valor numérico (**uint256**, que representa el monto depositado). El nombre del **mapping** es **contribuyentes**, que nos permitirá consultar cuánto ha depositado cada persona.
Esto lo podemos imaginar como una tabla, Ej:

Clave	Valor
K1	X1
K2	X2
K3	X3
Kn	Xn

```
event Agradecimiento(address depositante);

emit Agradecimiento(msg.sender);
```

Los **event** permiten a los contratos informar a los usuarios o a otros contratos sobre acciones específicas que han ocurrido.

Explicación del **Ejemplo**: Aquí, se define un evento llamado Agradecimiento con un parámetro:

address depositante: La dirección de la persona que está realizo el depósito.

La función “emit” se usa para lanzar un event previamente definido. Esto significa que el evento se "emite" y los datos asociados se registran en la blockchain.

Explicación del **Ejemplo**:

emit: Esta palabra indica que estamos lanzando un evento.

Agradecimiento: Este es el nombre del evento que estamos lanzando. Se debe haber definido previamente con event.

msg.sender: Es una variable especial en Solidity que representa la address del remitente que ha llamado a la función. En este contexto, el evento **Agradecimiento** está registrando la address de la persona que ha realizado una acción que queremos registrar (por ejemplo, un depósito).

Struct:

```
struct Producto {
    string nombre;
    uint256 precio;
    uint256 cantidad;
}
```

Un struct te permite definir un nuevo tipo de dato que puede contener varios atributos.

Constructor:

```
constructor(uint256 _cantidadMinima){
    CantidadMinima = _cantidadMinima;
    }
```

Características del **Constructor**

Ejecución Única: El constructor se ejecuta solo una vez, al momento de desplegar el contrato. No puede ser llamado después del despliegue.

Inicialización: Se utiliza para configurar el estado inicial del contrato, como establecer valores predeterminados para variables o realizar configuraciones necesarias.

No Tiene Tipo de Retorno: A diferencia de las funciones normales, el constructor no tiene un tipo de retorno y no se especifica la palabra clave function. Simplemente se declara con el nombre del contrato.

En este caso por ejemplo, lo que estamos definiendo es la cantidad mínima que se necesita para poder realizar una transferencia, pues, la define el usuario al momento de hacer el pago.

Codigo completo:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

error FaltaPlata(uint256 cantidaFaltante);

contract CuentaComunitaria {

    mapping (address depositante => uint256 monto) contribuyentes;


    event Depostio(address depositante, uint256 monto);

    event Agradecimiento(address depositante);

    receive() external payable { }

    uint256 public immutable CantidadMinima;
    uint256 public constant CANTIDAD_MINIMA = 10 ether;

    function depostiar() payable public{
        emit Deposito(msg.sender, msg.value);
        contribuyentes[msg.sender] += msg.value;

    }

    constructor(uint256 _cantidadMinima){
    CantidadMinima = _cantidadMinima;
    }
```

```
    function depositoGeneroso() payable public {

        require(msg.value >= CANTIDAD_MINIMA, "Cantidad insuficiente"
);

        if (msg.value < CANTIDAD_MINIMA) revert
FaltaPlata(CANTIDAD_MINIMA - msg.value);
        depositar();
        emit Agradecimiento(msg.sender);
    }

    function balanceDeUnaAddress(address unaAddress) public view
returns(uint256) {
        return unaAddress.balance;
    }

    function balanceContrato() public view returns(uint256){
        return address(this).balance;
    }

}
```