

# Desarrollo Blockchain Ethereum con Solidity

Módulo 2 - Modificadores, Eventos y Arrays

# Modificadores, Eventos y Arrays

# Validaciones (if vs. required)

Imaginemos que tenemos el siguiente contrato:

```
browser/educacionItContract.sol x
1 pragma solidity ^0.4.17;
2 //import "./jds.sol";
3
4 contract JuegoDeApuestas {
5     address[] public apostadores;
6
7     function apostar() public payable {
8         apostadores.push(msg.sender);
9     }
10
11     function obtenerListadoDeApostadores() public view returns (address[]) {
12         return apostadores;
13     }
14 }
```

¿Qué sucede si no enviamos nada de ether a la función "apostar"?

## Validaciones (if vs. required)













Necesitaremos entonces agregar una llamada a una función global para validar que se está enviando la cantidad deseada de Ether. De manera que la función ha de quedar como en el ejemplo a la derecha.

*Require* impide que la ejecución continúe si la condición no se cumple. En este caso, cualquier intento por llamar a la función con menos de 1 ether terminará en error. Estos errores pueden ser vistos en la consola de debug. *(Ejemplo a continuación)*

```
function apostar() public payable {  
    require(msg.value >= 1 ether);  
    apostadores.push(msg.sender);  
}
```

## Validaciones (if vs. required)

✖ [vm] from:0xca3...a733c to:JuegoDeApuestas.apostar() 0x9dd...d44dd value:1 wei data:0xc18...11269 logs:0 hash:0x6b0...2196a

status	0x0 Transaction mined but execution failed
transaction hash	0x6b0622441d0a062cef16ba50370a758a5a9669e76316c61ac84a6d79d652196a 
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c 
to	JuegoDeApuestas.apostar() 0x9dd1e8169e76a9226b07ab9f85cc20a5e1ed44dd 
gas	3000000 gas 
transaction cost	21408 gas 
execution cost	136 gas 
hash	0x6b0622441d0a062cef16ba50370a758a5a9669e76316c61ac84a6d79d652196a 
input	0xc18...11269 
decoded input	{ } 
decoded output	{ } 
logs	[ ]  
value	1 wei 

## Validaciones (if vs. required)

Supongamos que usamos un *if* para validar que se haya enviado el ether deseado.

Tendríamos que controlar que sucede en los casos en que no, dado que la transacción se ejecutaría sin inconvenientes.

```
function apostar() public payable {  
    if(msg.value >= 1 ether) {  
        apostadores.push(msg.sender);  
    }  
    else {  
        //Qué hacemos acá?  
    }  
}
```

## Validaciones (if vs. required)

🟢 [vm] from:0xca3...a733c to:JuegoDeApuestas.apostar() 0x0c2...739ef value:0 wei data:0xc18...11269 logs:0 hash:0xfec...d4696

status	0x1 Transaction mined and execution succeed
transaction hash	0xfec154a36bf4a1dfba4803bdd2fe0e3b7b32d9cddb9a9e12f47978005a3d4696 📄
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c 📄
to	JuegoDeApuestas.apostar() 0x0c2e77121daf0270d26bf0a7e9ab0faa8bf739ef 📄
gas	3000000 gas 📄
transaction cost	21410 gas 📄
execution cost	138 gas 📄
hash	0xfec154a36bf4a1dfba4803bdd2fe0e3b7b32d9cddb9a9e12f47978005a3d4696 📄
input	0xc18...11269 📄
decoded input	{ } 📄
decoded output	{ } 📄
logs	[ ] 📄 📄
value	0 wei 📄

# Sentencia Assert

La sentencia **Assert** se utiliza para validar valores con fines de testing, por eso los utilizaremos más adelante cuando analicemos cómo escribir pruebas sobre los contratos ya que no tiene ningún efecto en los mismos como lo tiene la sentencia **require**.





# Modificadores de función

Entre una de las buenas prácticas que se deben tener en cuenta al desarrollar contratos inteligentes es la de no repetir el código siempre que sea posible.

```
browser/educacionItContract.sol x
1  pragma solidity ^0.4.17;
2  //import "./jds.sol";
3
4  contract JuegoDeApuestas {
5      address[] public apostadores;
6      address public owner;
7
8      constructor() public {
9          owner = msg.sender;
10     }
11
12     function apostar() public payable {
13         if(msg.value >= 1 ether) {
14             apostadores.push(msg.sender);
15         }
16         else {
17             //Qué hacemos acá?
18         }
19     }
20
21     function elegirGanador() public view miModificadorDeFuncion {
22
23     }
24
25     function obtenerListadoDeApostadores() public view returns (address[]) {
26         return apostadores;
27     }
28
29
30     modifier miModificadorDeFuncion() {
31         require(msg.sender == owner);
32         _;
33     }
34 }
```

## Modificadores de función

- Un modificador de función, en Solidity, se define con la palabra reservada **modifier**.
- Puede o no recibir parámetros.
- No le aplican los modificadores de accesibilidad.
- Deben terminar en guión bajo, punto y coma (`_;`) para que pueda continuar la ejecución.
- Puede o no tener llamadas a `require`.
- Puede tener más de un `require`.
- Se ejecutará siempre antes que la primer línea de la función sobre la que se aplique.

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

# Logging & Eventos

El siguiente es un ejemplo de uso de Logging y Eventos en un contrato dentro de Ethereum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // Events are emitted using `emit`, followed by
        // the name of the event and the arguments
        // (if any) in parentheses. Any such invocation
        // (even deeply nested) can be detected from
        // the JavaScript API by filtering for `Deposit`.
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

## Logging & Eventos

Para definir un evento en Solidity para Ethereum, basta con utilizar la palabra clave **"event"** de la siguiente manera:

```
event Deposit(  
    address indexed _from,  
    bytes32 indexed _id,  
    uint _value  
);
```

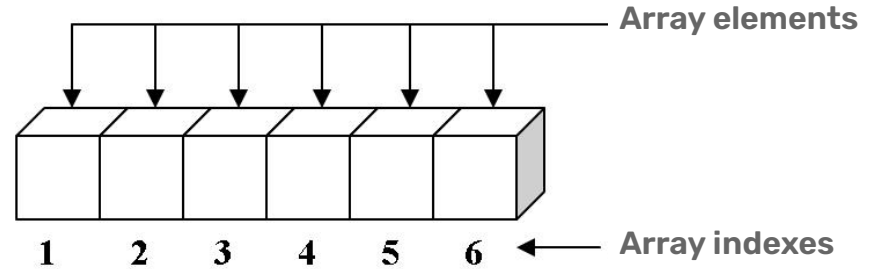
Luego, para invocar al evento, en cualquier función, sea pública o privada, basta con llamar al evento con el nombre definido con anterioridad. De manera que la llamada, dentro de una función se vería como la siguiente:

```
emit Deposit(msg.sender, _id, msg.value);
```

Por supuesto, el valor enviado al evento ha de ser del mismo tipo que el definido y puede o no ser el recibido por la función.

# Arrays

- Recordemos que un **array** es un tipo de estructura que contiene un grupo de elementos.
- Los arrays en Solidity son tipos de referencia.
- Existen diferentes tipos de arrays, entre los cuales se encuentran los **dynamic array** y los **fixed array**
- Son de base cero, es decir que el primer elemento se encuentra en la posición 0 y el último en la longitud menos uno.



**One-directional array with six elements**

# Arrays

## Fixed array

En Solidity, un array fijo se declara de la siguiente manera:

```
uint256[] ejemploDeArrayFijo = new uint256[](5);
```

Y los valores pueden ser asignados de la siguiente forma:

```
ejemploDeArrayFijo[0] = 115;  
ejemploDeArrayFijo[1] = 125;  
ejemploDeArrayFijo[2] = 145;  
ejemploDeArrayFijo[3] = 165;  
ejemploDeArrayFijo[4] = 185;  
  
ejemploDeArrayFijo[5] = 200; //ERROR!
```

Si el índice al que se quiere acceder está fuera del array, esto provocará un **error** que será devuelto y no continuará la ejecución del contrato.

# Arrays

## Dynamic array

En Solidity, un array dinámico se declara de la siguiente manera:

```
uint256[] ejemploDeArrayDinamico;
```

Y los valores pueden ser asignados de la siguiente manera:

```
ejemploDeArrayDinamico.push(120);  
ejemploDeArrayDinamico.push(1150);  
ejemploDeArrayDinamico.push(50);  
ejemploDeArrayDinamico.push(1900);  
ejemploDeArrayDinamico.push(333);
```

Es posible recuperar la longitud del array en todo momento usando la propiedad **length**.

```
ejemploDeArrayDinamico.length;
```

# ¡Muchas gracias!

¡Sigamos trabajando!