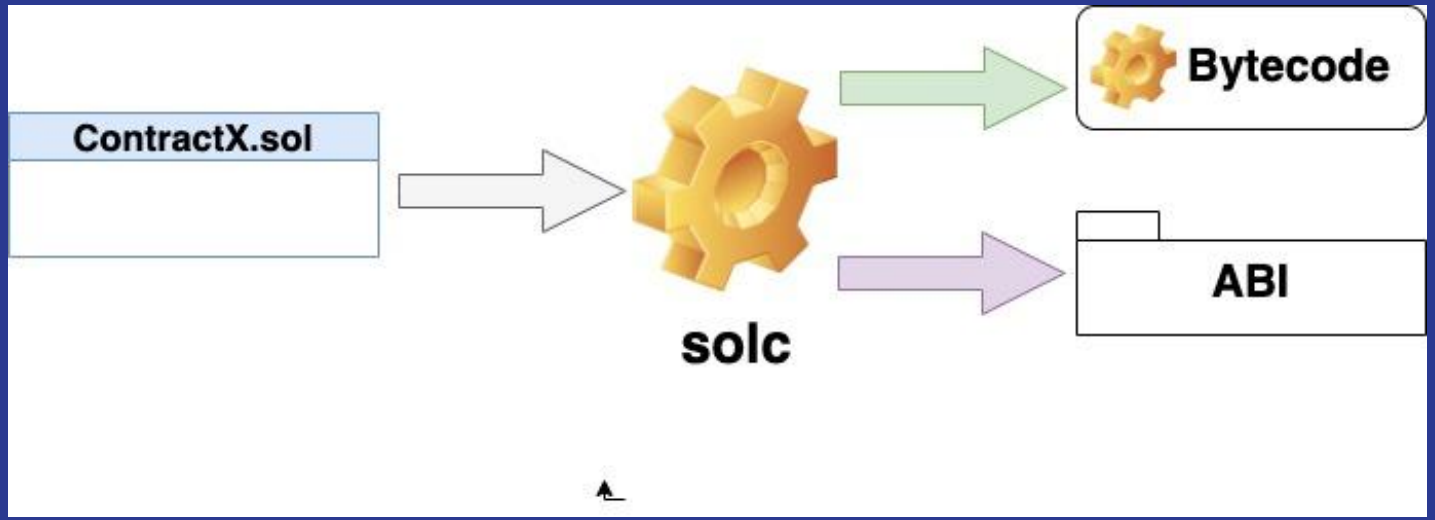
Desarrollo de software para blockchain 2024

Clase 03

Solidity

Solidity

Es un lenguaje de programación de alto nivel compilado diseñado específicamente para escribir smart contracts en la plataforma Ethereum. Fue propuesto por Gavin Wood en 2014 y desde entonces se ha convertido en el lenguaje de programación principal para el desarrollo de smart contracts en Ethereum.



Solidity: tipos de datos

- Primitivos:
  - Booleanos: bool (con valores true o false).
  - Enteros: int y uint (para números enteros con y sin signo, respectivamente). Ambos pueden tener tamaños múltiples de 8 bits hasta 256 bits, como int8, int16... int256 y uint8, uint16... uint256.
- Direcciones:
  - address: Representa una dirección Ethereum de 20 bytes.
  - address payable: Similar a address, pero permite recibir y enviar ether.
- Bytes:
  - bytes1, bytes2,..., bytes32 (secuencias de bytes de longitud fija).
- Tipos de Punto Flotante:
  - No hay soporte nativo para tipos de punto flotante en Solidity debido a complicaciones con la precisión y la determinística en la EVM.

Solidity: tipos de datos

- Arrays:
  - Arrays de longitud fija y dinámica, por ejemplo, uint[5] (fijo) y uint[] (dinámico).
- Bytes Dinámicos: bytes (similares a bytes[], pero de longitud dinámica).
- Strings: string (representa una cadena de caracteres UTF-8 de longitud dinámica).
- Maps: mapping(key => value). Funcionan como diccionarios o tablas hash, donde key puede ser casi cualquier tipo excepto un mapping, y value puede ser cualquier tipo, incluido otro mapping.
- Structs: Definen estructuras personalizadas.
- Enums: Se utilizan para definir variables que pueden tener uno de los valores predefinidos.

Solidity: tipos de datos

```
bool miBool = true;
int miInt = -123456;
uint256 miUint = 123456;
address miDireccion = 0x5A0b54D5dc17e0AadC383d2db43B0a0D3E029c4c;
address payable miDireccionPayable = payable(0x5A0b54D5dc17e0AadC383d2db43B0a0D3E029c4c);
bytes32 miBytes32 = "mi cadena de bytes"; // Esto es solo un ejemplo de valor
address[] listaDirecciones;
bytes miBytesDinamico = "Ejemplo bytes dinamico";
string miString = "Ejemplo string";
mapping(address => uint256) miMapping;
```

Solidity: tipos de datos

```
struct User {
    uint id;
    string username;
    Rol rol;
}
enum Rol {
    Admin,
    Staff,
    Inactive
}
```

Solidity: estructuras de control

```
if (condicion) {
    // código a ejecutar si la condición es verdadera
} else {
    // código a ejecutar si la condición es falsa
}
```

Solidity: estructuras de control

Hay que tener cuidado con los bucles en Solidity, ya que los bucles que consuman demasiado gas pueden resultar en transacciones que no se puedan confirmar debido a superar el límite de gas del

```
for (uint i = 0; i < num; i++) {
    // código a ejecutar en cada iteración
}
while (condicion) {
    // código a ejecutar mientras la condición sea verdadera
}
do {
    // código a ejecutar
} while (condicion);
```

Solidity: definición de un contrato

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

contract SimpleContract {
    uint256 public storedData;

    constructor(uint256 initialValue) {   infinite gas 76600 gas
        storedData = initialValue;
    }

    function set(uint256 x) public {   22520 gas
        storedData = x;
    }

    function get() public view returns (uint256) {   2459 gas
        return storedData;
    }
}
```

Solidity: funciones de contrato

se pueden clasificar según su visibilidad y comportamiento.

- **Funciones de Estado (State Functions):** Estas son funciones que pueden modificar el estado del contrato (es decir, cambiar el valor de una o más variables de estado). Estas funciones consumen gas cuando son llamadas y las transacciones que las invocan deben ser validadas en un bloque.

- **Funciones de Sólo Lectura (View Functions):** Estas funciones no pueden modificar el estado del contrato y sólo leen valores. Se pueden llamar externamente y no consumen gas cuando se invocan de esta manera (porque no generan una transacción real en la cadena de bloques). Se declaran con el modificador view.

- **Funciones Pura (Pure Functions):** Son similares a las funciones view, pero son aún más restrictivas: no pueden leer ni modificar el estado del contrato. Están diseñadas para retornar un valor basado únicamente en sus argumentos. Se declaran con el modificador pure.

Solidity: funciones de contrato cont...

- **Funciones de Pago (Payable Functions):** Son funciones que pueden recibir Ether. Si envías Ether a una función que no está marcada con el modificador payable, la transacción fallará. Las funciones payable son esenciales para contratos que deben manejar Ether directamente, como un contrato de crowdfunding.

- **Funciones Internas (Internal Functions):** Estas funciones sólo pueden ser llamadas desde el propio contrato o desde contratos derivados. No pueden ser llamadas directamente por transacciones externas. Son útiles para dividir la lógica del contrato en partes más pequeñas, para reutilizar código, o para implementar detalles de bajo nivel que no deberían ser expuestos a los usuarios del contrato.

- **Funciones Externas (External Functions):** Estas funciones están destinadas a ser llamadas desde transacciones externas y no desde otras funciones dentro del contrato. En algunos casos, pueden ser más eficientes en términos de gas cuando se llaman desde el exterior en comparación con las funciones public.

Solidity: funciones de contrato cont II

- Funciones Fallback y Receive:

  - **Fallback:** Antes de la versión 0.6.0 de Solidity, si un contrato recibía Ether sin que ninguna función fuese llamada (es decir, una transacción de Ether a la dirección del contrato sin datos adicionales), y si el contrato no tenía una función marcada como payable, entonces la función fallback se ejecutaba. Esta función no puede tener argumentos ni retornar nada.

  - **Receive:** Introducido en la versión 0.6.0, es una variante especializada de la función fallback que sólo se ejecuta cuando no se llama a ninguna otra función y no se proporcionan datos.

La visibilidad de las funciones (y también de las variables de estado) en Solidity se define mediante los modificadores public, private, internal y external.

Solidity: funciones de contrato cont III

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;
contract EjemploFunciones {
    uint256 public contador = 0; // Variable de estado publica.
    address private owner; // variable de estado internal por defecto
    // Constructor: Se ejecuta una vez cuando se despliega el contrato.
    constructor() {    242729 gas 213200 gas
        owner = msg.sender; // Asigna al creador del contrato como propietario.
    }
    // 1. Función de Estado: Modifica el estado del contrato.
    function incrementar() public {    infinite gas
        contador += 1;
    }
    // 2. Función View: Sólo lee el estado y no lo modifica.
    function obtenerContador() public view returns (uint256) {    2481 gas
        return contador;
    }
    // 3. Función Pura: No accede ni modifica el estado, solo trabaja con sus argumentos.
    function sumar(uint256 a, uint256 b) public pure returns (uint256) {    infinite gas
        return a + b;
    }
    // 4. Función Payable: Puede recibir Ether.
    function depositar() public payable {    409 gas
        require(msg.value > 0, "Debe enviar algo de Ether");
    }
    // 5a. Función Receive: Se ejecuta cuando se envía Ether sin datos adicionales.
    receive() external payable {}    undefined gas
    // 5b. Función Fallback: Se ejecuta si ninguna otra función coincide o si los datos enviados no corresponden.
    fallback() external {}    undefined gas
    // 6. Función Internal: No puede ser llamada desde fuera, útil para lógica interna.
    function esPropietario() internal view returns (bool) {    2176 gas
        return msg.sender == owner;
    }
    // 7. Función External: Solo puede ser llamada desde fuera, no desde otras funciones dentro del contrato.
    function comprobarPropietario() external view returns (bool) {    2553 gas
        return esPropietario();  // Error: una función external no puede llamar a una función internal.
    }
```

Solidity: modificadores

Los modificadores en Solidity son herramientas que se utilizan para cambiar el comportamiento de funciones o para verificar ciertas condiciones antes de que una función sea ejecutada. En esencia, permiten añadir una lógica adicional a la ejecución de una función de forma reutilizable y legible. Es común utilizarlos para verificar permisos, restricciones o estados específicos del contrato.

- **Definición:** Un modificador se define de manera similar a una función, pero con la palabra clave modifier en lugar de function.
- **Uso del placeholder _:** Dentro del cuerpo del modificador, el placeholder _ se utiliza para indicar dónde debe insertarse el código de la función que se modifica. El código antes del _ se ejecuta antes de la función, y el código después del _ se ejecuta después de la función (si lo hay).
- **Aplicación a funciones:** Una vez definido, el modificador puede ser añadido a una función añadiendo su nombre después de la declaración de la función, antes del {.

Solidity: modificadores cont …

```
pragma solidity ^0.8.0;

contract Example {

    address public owner;

    // Constructor para establecer el propietario como la dirección que despliega el contrato
    constructor() {   185874 gas 161400 gas
        owner = msg.sender;
    }

    // Modificador para restringir el acceso solo al propietario
    modifier onlyOwner() {
        require(msg.sender == owner, "Solo el propietario puede ejecutar esto");
        _;  // Placeholder: aquí es donde se insertará el código de la función modificada
    }

    // Una variable de estado
    uint256 public data;

    // Función que solo puede ser llamada por el propietario debido al modificador
    function setData(uint256 _data) public onlyOwner {   24665 gas
        data = _data;
    }
}
```

Solidity: import y herencia

En Solidity, la directiva import se utiliza para incluir código de otros archivos. Esto es especialmente útil para la organización de código y para reutilizar bibliotecas o contratos existentes.

Definición Básica de Herencia:
Un contrato puede heredar de uno o más contratos usando la palabra clave **is.**

```
contract BaseContract {
    uint public data;

    function setData(uint _data) public {
        data = _data;
    }
}

contract DerivedContract is BaseContract {
    // Hereda la variable de estado "data" y la función "setData"
}
```

Solidity: import y herencia

Orden de Herencia:
Cuando un contrato hereda de múltiples contratos, el orden en el que se enumeran es importante porque determina el orden de inicialización y el orden de disposición del almacenamiento.

```
contract A {
    uint public a = 1;
}

contract B {
    uint public b = 2;
}

contract C is A, B {
    // El orden de inicialización es A, luego B, por lo que "a" se inicializa antes que "b".
}
```

Solidity: import y herencia

Sobrescritura:
Las funciones, las variables de estado y los modificadores en los contratos derivados pueden sobrescribir las definiciones de los contratos base.

```
contract Base {
    function getValue() public pure returns (uint) {
        return 1;
    }
}

contract Derived is Base {
    // Sobrescribe la función getValue de Base
    function getValue() public pure override returns (uint) {
        return 2;
    }
}
```

Solidity: import y herencia

Super:Si un contrato deriva de múltiples contratos y varios de ellos definen una función (o un modificador) con el mismo nombre, super se utiliza para acceder a la función o modificador de la siguiente base en la cadena de herencia.

```
contract A {
    function foo() public pure returns (string memory) {
        return "A";
    }
}

contract B is A {
    function foo() public pure override returns (string memory) {
        return super.foo();  // Devuelve "A"
    }
}

contract C is B {
    function foo() public pure override returns (string memory) {
        return super.foo();  // Devuelve "A", ya que super en C apunta a B, y super en B apunta a A
    }
}
```

Solidity: import y herencia

Constructores y Herencia:Si un contrato base tiene un constructor, este debe ser llamado en el constructor del contrato derivado. Si el contrato base tiene un constructor sin argumentos, este se llamará automáticamente.

```
pragma solidity ^0.8.0;

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

contract Example is EjemploFunciones, ERC20{

    address public owner;

    constructor(string memory name_, string memory symbol_) ERC20(name_, symbol_){
        owner = msg.sender;
    }

    // Modificador para restringir el acceso solo al propietario
```

Solidity: herencia e interfaces

Interfaces y Herencia:
Además de heredar de otros contratos, un contrato también puede heredar de interfaces, que son como contratos pero no pueden tener implementaciones de funciones ni variables de estado.

```
interface IExample {
    function getValue() external returns (uint);
}

contract MyContract is IExample {
    uint private _value = 1;

    function getValue() external override returns (uint) {
        return _value;
    }
}
```

Solidity: Modificadores de Tipo de Datos

- **memory**: Indica que la variable se almacena temporalmente y desaparecerá entre llamadas de función externa.
- **storage**: Indica que la variable se almacena de forma persistente en el almacenamiento del contrato.
- **calldata**: Es similar a memory, pero es inmutable y se utiliza principalmente para parámetros de funciones externas.
- **constant / immutable**:Estos modificadores se utilizan para variables de estado que no cambiarán después de que se asignen. Mientras que constant se usaba para variables cuyo valor se conoce en tiempo de compilación, immutable es para variables que se asignan una vez durante la construcción y no cambian después.

Solidity: Modificadores de Tipo de Datos

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleContract {

    // Variable de estado en almacenamiento (storage)
    uint256 private storageData;

    // Constante (valor que no cambia y es conocido en tiempo de compilación)
    uint256 public constant CONSTANTE_VALOR = 123456;

    // Valor asignado en el constructor que no cambia después (inmutable)
    uint256 public immutable creationTimestamp;

    // Constructor para inicializar el valor inmutable
    constructor() {     infinite gas 266400 gas
        creationTimestamp = block.timestamp;
    }

    // Función que utiliza parámetros calldata, memory y modifica storage
    function setData(uint256 nuevoData, bytes calldata datosAdicionales) external {     infinite gas
        uint256[] memory arrayMemory = new uint256[](3);
        arrayMemory[0] = nuevoData;
        arrayMemory[1] = storageData;
        arrayMemory[2] = bytesToUint256(datosAdicionales);  // Conversión simplificada, solo para propósitos de demostración.

        storageData = arrayMemory[0] + arrayMemory[1] + arrayMemory[2];
    }

    // Función que devuelve un valor desde el almacenamiento(storage)
    function getData() external view returns (uint256) {     2437 gas
        return storageData;
    }

    // Función helper para convertir bytes a uint256 (simplificada)
    function bytesToUint256(bytes memory input) internal pure returns (uint256) {     undefined gas
        uint256 output;
        for (uint256 i = 0; i < input.length; i++) {
            output = output << 8 | uint256(uint8(input[i]));
        }
```

Solidity: palabras clave o reservadas

Algunas importantes, aparte de las ya vistas:

`**abstract**`: Indica que un contrato no puede ser desplegado por sí mismo y debe ser heredado por otro contrato.
`**delete**`: Usado para borrar contenido en storage o para resetear una variable a su valor predeterminado.
`**library**`: Define una biblioteca, que es similar a un contrato pero no tiene variables de estado.
`**override**`: Indica que una función sobrescribe una función heredada.
`**this**`: Refiere a la instancia actual del contrato.
`**try-catch**`: Utilizado para manejar fallos en llamadas externas.
`**type**`: Obtiene el tipo de una variable.
`**using**`: Indica que un contrato usa una biblioteca específica.

Solidity: Eventos

Un evento es una característica que permite a los contratos inteligentes emitir registros que las dapps front-end o las aplicaciones externas pueden "escuchar" y reaccionar en consecuencia. Estos registros se almacenan en los logs de la blockchain, lo que significa que son menos costosos en términos de gas en comparación con el almacenamiento de datos en el propio contrato.

```
contract Storage {

    event DataChanged(address indexed by, uint256 oldValue, uint256 newValue);
    uint256 public data;

    function setData(uint256 _data) public {
        emit DataChanged(msg.sender, data, _data);  // Emitir el evento
        data = _data;  // Cambiar la data
    }
```

En este ejemplo, cada vez que se llama a la función setData, se emite el evento DataChanged registrando quién cambió los datos, el valor anterior y el valor nuevo. La palabra indexed en el evento significa que ese argumento se puede usar como un filtro para buscar o escuchar eventos específicos.

Links

- remix: http://remix.ethereum.org/
- faucet: pedirme en un mensaje de discord
- metamask: chrome-extension://nkbihfbeogaeaoehlefnkodbefgpgknn/home.html#onboarding/welcome
- explorador de bloques
- validar un contrato