

## 1. ¿Qué es Solidity y por qué importa en IoT?

**Solidity** es un lenguaje estático inspirado en JS/TypeScript, con tipado fuerte y orientado a contratos para la EVM (Ethereum Virtual Machine). Sirve para escribir *smart contracts*: programas que viven en la blockchain.

### IoT + Blockchain:

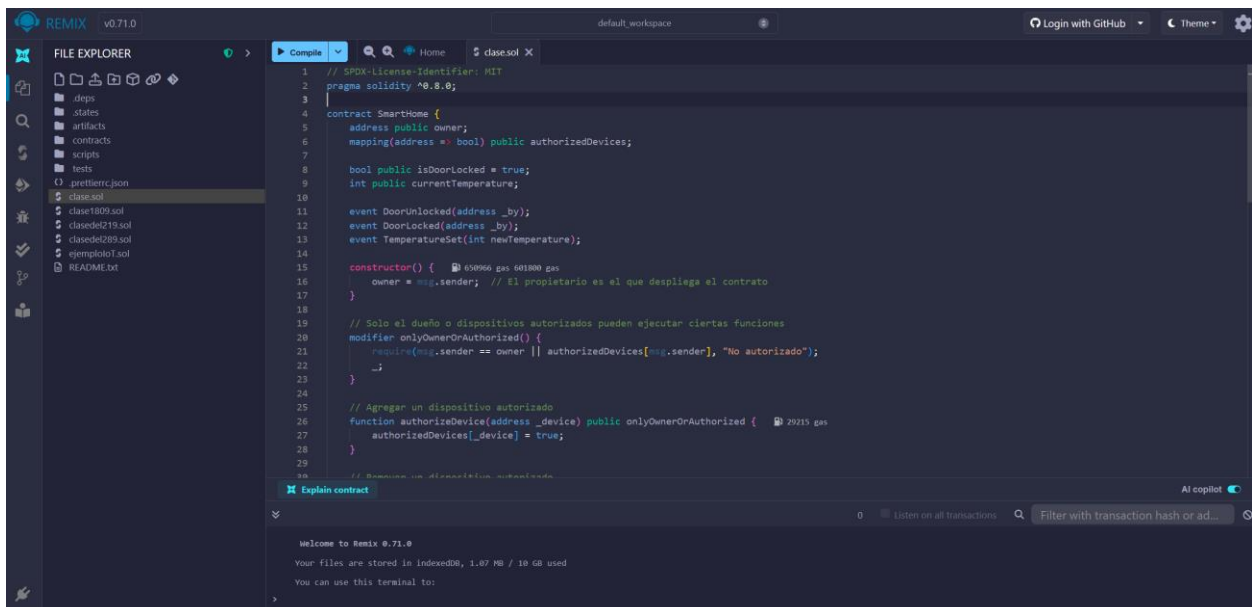
- **Trazabilidad** de lecturas (temperatura, humedad, consumo energético) con sello de tiempo inmutable.
- **Automatización** (pagos o alertas) cuando un sensor reporta un valor fuera de umbral.
- **Mercados de datos** (quién puede leer o pagar por datos).

Idea clave: la blockchain no “lee sensores”. Solo ejecuta transacciones. Para llevar datos del mundo al contrato se usa un **oráculo** (off-chain) que firma/transmite lecturas.

## 2. Tour rápido por Remix IDE

Abrí [remix.ethereum.org](https://remix.ethereum.org). Verás:

- **File Explorers:** crea archivos **.sol**.
- **Solidity Compiler:** elegí versión ^0.8.x.
- **Deploy & Run:** ambiente de ejecución (*JavaScript VM* para pruebas, *Injected Provider* para MetaMask), despliegue y panel de funciones.
- **Terminal:** logs y errores.



### 3. Tu primer contrato

Objetivo: compilar, desplegar e invocar funciones.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract HelloSensor {
    string private message;

    constructor(string memory initialMessage) {
        message = initialMessage;
    }

    function setMessage(string calldata newMessage) external {
        message = newMessage;
    }

    function getMessage() external view returns (string memory) {
        return message;
    }
}
```

#### Pasos:

1. Crear `HelloSensor.sol` → pegar el código.
2. Compilar (Solidity Compiler → ^0.8.20).
3. Deploy (Deploy & Run → Environment: *JavaScript VM* → en *constructor* ingresar "`Hola IoT`").
4. Probar `getMessage()` y `setMessage("Nuevo")`.

### 4. Estructura mínima de un contrato

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract EjemploBasico {
    // estado (variables persistentes)
    uint256 public contador;

    // constructor (se ejecuta 1 vez al deploy)
    constructor(uint256 inicial) {
        contador = inicial;
    }

    // función que modifica estado (consume gas)
    function inc() external {
        contador += 1;
    }

    // función de solo lectura (no consume gas llamada como call)
    function get() external view returns (uint256) {
        return contador;
    }
}
```

## 5. Solidity

### Primitivos

- **bool**: `true` / `false`
- **uintX** / **intX**: enteros sin/signed (8,16,...,256 bits). Por defecto usamos `uint256`.
- **address**: 20 bytes (dirección de cuenta/contrato). `address payable` permite `.transfer/.send/.call{value:...}`.
- **bytes1..bytes32**: arreglos de bytes de tamaño fijo (baratos).
- **bytes**: arreglo dinámico de bytes (más caro).
- **string**: cadena UTF-8 dinámica (costosa en almacenamiento).
- **enum**: tipo enumerado (internamente uint).

```

bool flag = true;
uint256 precio = 1 ether;
int128 delta = -5;
address owner = msg.sender;
bytes32 hashFijo = keccak256(abi.encodePacked("hola"));
enum Estado { Creado, Activo, Pausado, Cerrado }

```

### Compuestos

- **array**: fijo (`uint[3]`) o dinámico (`uint[]`).
- **struct**: registro con campos nombrados.
- **mapping**( $K \Rightarrow V$ ): diccionario (no iterable nativamente).

```

uint[] public lista;           // dinámico
uint[3] public fijo;          // tamaño fijo
struct Usuario { address cuenta; uint40 creado; string nombre; }
mapping(address => Usuario) public usuarios;

```

### Ubicaciones de datos: storage, memory, calldata

- **storage**: estado persistente en la blockchain (caro). Variables de estado y referencias a estructuras en estado.
- **memory**: temporal durante la ejecución; copia de datos (barato comparado con storage).
- **calldata**: solo lectura para parámetros en funciones `external`; evita copias.

```

function setNombre(string calldata nom) external {
    // 'nom' está en calldata (no se copia). Si necesitas mutar, copialo a 'memory'.
    usuarios[msg.sender].nombre = nom;
}

function getUsuario(address who) external view returns (Usuario memory) {
    return usuarios[who]; // copia a memory para el retorno
}

```

### *Variables de estado, locales y constantes*

- **estado**: definidas fuera de funciones (viven en **storage**).
- **locales**: dentro de funciones (en stack o memory).
- **constant / immutable**:
  - **constant**: se fija en compilación.
  - **immutable**: se fija en el constructor.

```
uint256 public contador;           // estado
uint256 public constant FEE_BP = 25; // 0.25% en basis points
address public immutable OWNER;

constructor() {
    OWNER = msg.sender; // se fija aquí
}
```

### *Visibilidad y mutabilidad de funciones*

- **Visibilidad**: **public**, **external**, **internal**, **private**
  - **external**: mejor para funciones que solo se llaman desde fuera (parámetros en **calldata**).
- **Mutabilidad**: **view** (solo lectura), **pure** (no lee estado), **payable** (puede recibir ETH).

### *Modificadores, eventos y errores*

#### Modificadores

Reutilizan *pre/post-checks* en funciones.

```

address public owner = msg.sender;
modifier onlyOwner() {
    require(msg.sender == owner, "not owner");
    _;
}

function cambiarOwner(address nuevo) external onlyOwner {
    require(nuevo != address(0), "zero addr");
    owner = nuevo;
}

```

### *Eventos (logs)*

Baratos para consultar off-chain (analítica, IoT).

### *Errores y manejo de fallas*

- `require(cond, "msj")`: valida precondiciones.
- `revert ErrorPersonalizado(args)`: más barato que strings largas.
- `assert(cond)`: invariantes internas (si falla, bug).

```

error NoDueno(bytes32 id, address quien);
if (msg.sender != dueno[id]) revert NoDueno(id, msg.sender);

```

### *Globales útiles y unidades*

- `msg.sender`, `msg.value`, `tx.origin` (no recomendado para auth), `block.timestamp`, `block.number`.
- Unidades:
  - Moneda: `wei`, `gwei`, `ether`.
  - Tiempo: `seconds`, `minutes`, `hours`, `days`, `weeks`.

*Funciones especiales: constructor, receive y fallback*

*Mappings, arrays y estructuras (patrones comunes IoT)*

```
struct Lectura { int32 valor; uint40 t; } // packing eficiente
mapping(bytes32 => Lectura[]) private lecturas;

function pushLectura(bytes32 id, int32 val, uint40 t) external {
    // validaciones...
    lecturas[id].push(Lectura(val, t));
}

function ultimaLectura(bytes32 id) external view returns (Lectura memory) {
    Lectura[] storage arr = lecturas[id];
    require(arr.length > 0, "sin datos");
    return arr[arr.length - 1];
}
```

*Gas y optimización básica*

- **Tipos más pequeños** y *packing* en **struct** (ordenar de mayor a menor o compatibles).
- Usar **calldata** en funciones **external**.
- Evitar **string/bytes** en loops.
- Desacoplar operaciones costosas.
- Medir con Remix: *Compile* → *Gas Estimates* y comparará cambios.

## 6. Mini-ejemplo integral

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
contract IoTMini {
    struct Sensor { address dueno; string nombre; bool activo; }
    struct Lectura { int32 v; uint40 t; }
```

```

mapping(bytes32 => Sensor) public sensores;
mapping(bytes32 => Lectura[]) private datos;
mapping(address => bool) public updaters;

event SensorReg(bytes32 indexed id, address indexed dueno, string nombre);
event LecturaPush(bytes32 indexed id, int32 v, uint40 t);

modifier onlyUpdater() { require(updaters[msg.sender], "not updater"); _; }

function setUpdater(address who, bool isU) external {
    // demo: cualquiera puede setear (en prod: onlyOwner)
    updaters[who] = isU;
}

function sensorId(address dueno, string memory nombre) public pure returns (bytes32) {
    return keccak256(abi.encode(dueno, nombre));
}

function registrar(string calldata nombre) external {
    bytes32 id = sensorId(msg.sender, nombre);
    require(sensores[id].dueno == address(0), "existe");
    sensores[id] = Sensor(msg.sender, nombre, true);
    emit SensorReg(id, msg.sender, nombre);
}

function toggle(bytes32 id) external {
    Sensor storage s = sensores[id];
    require(s.dueno == msg.sender, "no dueno");
    s.activo = !s.activo;
}

function pushLectura(bytes32 id, int32 v, uint40 t) external onlyUpdater {
    Sensor storage s = sensores[id];
    require(s.dueno != address(0) && s.activo, "invalido");
    datos[id].push(Lectura(v, t));
    emit LecturaPush(id, v, t);
}

function ultima(bytes32 id) external view returns (Lectura memory) {
    Lectura[] storage arr = datos[id];
    require(arr.length > 0, "sin datos");
}

```

```
        return arr[arr.length - 1];  
    }  
}
```