Clase 2

Intro a Solidity

Leer y escribir un contrato

Leer y escribir un contrato Solidity simple

Leer y escribir un contrato Solidity simple:

- tipos y **data location**
- **visibilidad** y **mutabilidad** de funciones
- manejo de **errores** y **eventos**
- el ciclo **deploy → call**

Objetivos de aprendizaje

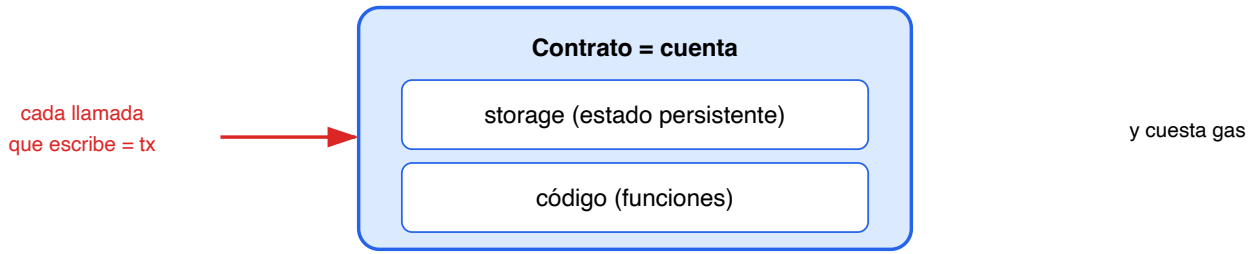
Al terminar, vas a poder:

- Leer un contrato y decir **qué hace cada parte**
- Distinguir `storage` / `memory` / `calldata` y **por qué importa** (estado y gas)
- Elegir la **visibilidad** correcta de una función
- Usar `require` / `revert` / **custom errors**
- Emitir y entender **eventos**
- **Desplegar y llamar** un contrato en Remix

SEGMENTO 1

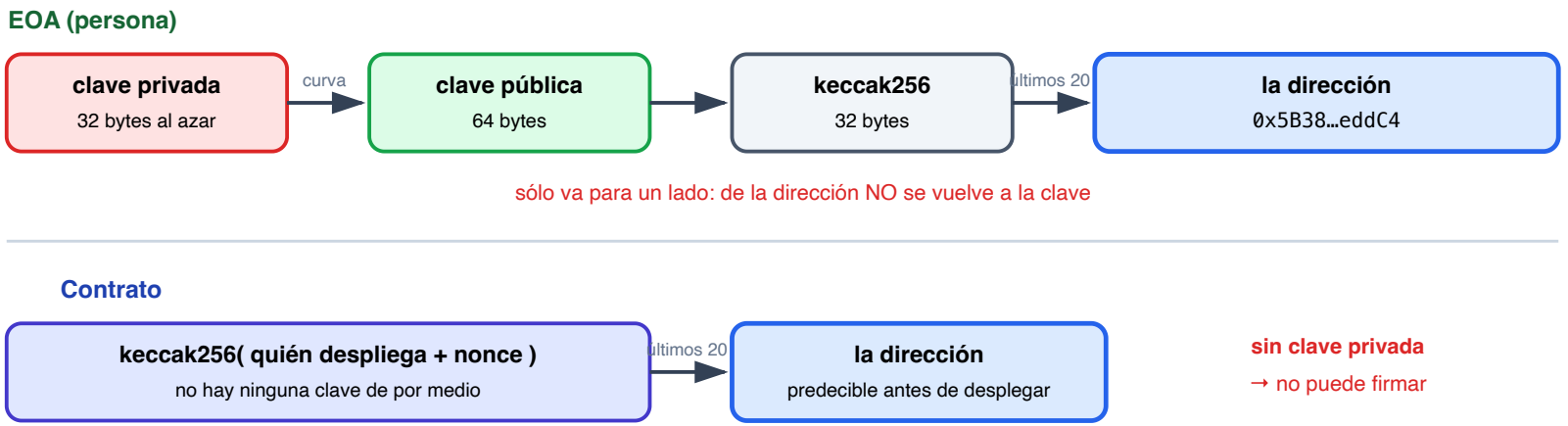
Contexto EVM

Un contrato es una cuenta con código



Solidity → **bytecode** → lo ejecuta la **EVM**.

¿De dónde sale una dirección?



> Un contrato **no puede iniciar una transacción**: no tiene con qué firmarla.
> En el origen de toda cadena de llamadas hay **siempre una persona**. *

Como un objeto — pero que cuesta dinero y persiste

> Un contrato es como un **objeto** con campos persistentes (storage) y métodos (funciones)...
>
> ...pero cada método que **escribe** estado **cuesta dinero** y **queda registrado** en la blockchain.

Esa diferencia con un objeto común —el costo y la permanencia— condiciona **cómo** se escribe el código.

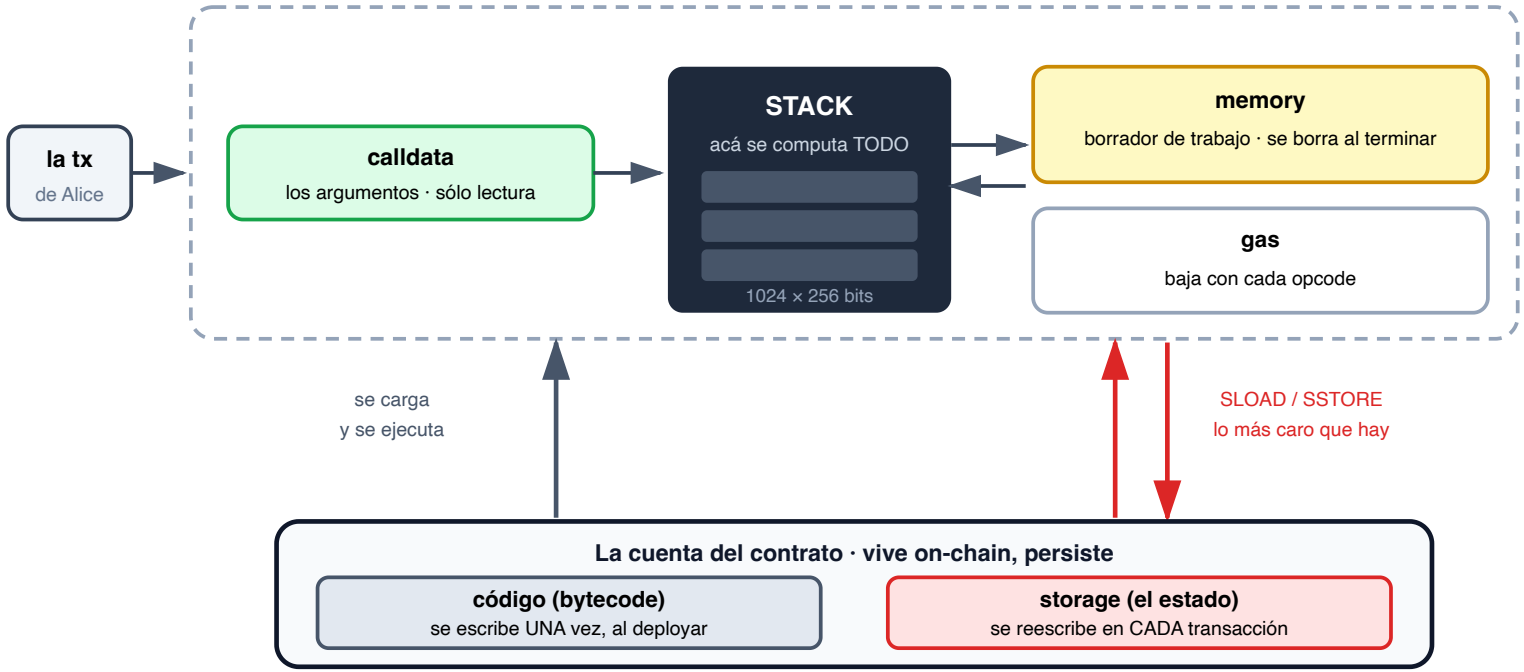
La EVM: máquina de pila de 256 bits

La palabra nativa de la EVM es de **256 bits** → por eso `uint256` es el tipo "por defecto" de todo el ecosistema.

> Usar `uint8` **no ahorra gas** por sí solo — la EVM opera palabras completas. Ahorra sólo cuando el compilador puede **empaquetar** varios campos chicos en un mismo slot de storage.

Y no hay punto flotante: **todos los montos son enteros** (en wei). Es deliberado — determinismo primero.

Las piezas de la EVM



Lo de arriba **existe sólo durante la llamada**. Lo de abajo **persiste**: el código (inmutable) y el storage de la cuenta (mutable). *

El gas cuenta una historia

Números orientativos — lo que importa es la **proporción**:

Operación	Gas (aprox.)
Suma en la pila (`ADD`)	3
Leer un slot de storage (`SLOAD` , primera vez)	~2.100
Escribir un slot NUEVO de storage (`SSTORE`)	**~20.000**
Sobrescribir un slot existente	~5.000
Costo base de cualquier tx	21.000

> Escribir estado es **~4 órdenes de magnitud** más caro que computar.
> **Diseñar las estructuras de datos = diseñar los costos.**

Unidades de ETH

Las tres que vas a ver todo el tiempo:

```
1 ether = 1.000.000.000 gwei = 10^18 wei

msg.value  → siempre en wei
precio del gas → se suele expresar en gwei
```

Todos los montos on-chain son **enteros en wei** — nunca "0.5 ETH" dentro del contrato, sino `500000000000000000`.

SEGMENTO 2

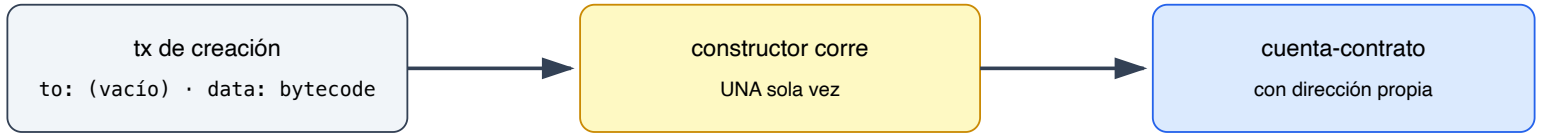
Anatomía de un contrato

Las partes mínimas

```
// SPDX-License-Identifier: MIT       ← identificador de licencia
pragma solidity ^0.8.26;              ← versión del compilador

contract Contador {
    uint256 public total;             // variable de estado (storage)
    address public owner;             // quién desplegó

    constructor() {
        owner = msg.sender;           // corre UNA vez, en el deploy
    }

    function incrementar() external {
        total += 1;                   // muta estado → es una tx, cuesta gas
    }
}
```

Qué aporta cada pieza

Qué es "deployar"



Desde entonces **el código es inmutable** — no hay "subir un fix".
Esta es la razón de ser de todo el bloque de análisis del curso
(y del ecosistema de proxies — Clase 6).

El getter automático

```
uint256 public total;   // ← `public` genera un getter automático
```

Es equivalente a escribir a mano:

```
uint256 private total;

function total() external view returns (uint256) {
    return total;
}
```

> `public` en una **variable de estado** crea una función de lectura con el mismo nombre.

`constant` e `immutable`

Dos calificadores que **evitan storage** (y por lo tanto, gas):

```
uint256 public constant MAX = 1000;   // fijado al COMPILAR; vive en el bytecode
address public immutable owner;       // fijado UNA vez, en el constructor
```

	Se fija...	Vive en...
`constant`	al compilar	el bytecode
`immutable`	una vez, en el constructor	el código desplegado

> Idioma muy común: `owner` como `immutable`. Leerlas **no toca storage** → mucho más barato que una variable de estado normal.

Comentarios NatSpec

La convención de documentación del ecosistema — la van a ver en cualquier contrato real:

```
/// @title  Contador — lleva una cuenta pública
/// @notice Suma `cuantos` al total de una vez.  ← para el usuario
/// @dev    Sin control de acceso: cualquiera    ← para el desarrollador
///         puede llamarla.
/// @param  cuantos  cuánto sumar al total
function incrementarEn(uint256 cuantos) external { ... }
```

Las herramientas (exploradores, wallets, generadores de docs) los leen y los muestran.

`import` y herencia (`is`)

Cierran la anatomía del archivo: arriba del `contract` van los **imports**; al lado del nombre, **de quién hereda**.

```
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";

contract MiToken is Ownable {      // `is` = herencia (el `extends` de Solidity)
    // hereda owner, onlyOwner, transferOwnership…
}
```

`Ownable` (OpenZeppelin) resuelve *"sólo el dueño puede llamar a esto"*. Heredarlo evita reescribir el control de acceso a mano — **error muy común**.

Una `interface` es otra cosa: sólo las firmas, sin cuerpo, para hablarle a un contrato ajeno.

> En la Clase 3 vamos a ver los tests que empiezan con `contract MiTest is Test`: el contrato de test **hereda** del framework de Foundry.

SEGMENTO 3

Tipos y data location

Los tipos

De valor (se copian):

```
uint256 n;      int256 i;      bool flag;
address cuenta; bytes32 hash;
```

De referencia (apuntan a una ubicación):

```
string  texto;
bytes   datos;
uint256[] lista;        // array (dinámico: push/pop)
struct Persona { ... }  // struct
mapping(address => uint256) balances;  // mapping
```

Detalles de tipos que importan

`address` vs `address payable`:

```
address cuenta;                         // NO se le puede enviar ETH
address payable cobrador;               // sí se puede
cobrador = payable(cuenta);             // el cast es EXPLÍCITO
```

`bytes32`: el tipo de los hashes e identificadores —

```
bytes32 huella = keccak256(abi.encodePacked(dato));
```

No hay `null`

Toda variable nace **zero-inicializada**: `0`, `false`, `address(0)`, `""`.

Consecuencias directas:

- "No inicializado" y "vale cero" son **indistinguibles** — si cero es un valor válido del dominio, hace falta un flag aparte
- `address(0)` como valor especial: mandar fondos u ownership a `address(0)` es un bug clásico — muchos contratos lo chequean explícitamente

El mapping no es un diccionario común

```
mapping(address => uint256) private balances;
```

- **Todas las claves "existen" siempre**, con valor cero:
  `balances[cualquiera]` nunca falla — devuelve `0`
  (es la zero-inicialización otra vez)
- **No es iterable**: no hay `keys()` ni `length`.
  ¿"Cómo listo todos los balances"?
  → **off-chain**, con eventos o un indexer
  → **on-chain**, con un array (no recomendable)
- `delete balances[x]` sólo resetea esa clave a cero

Es *la* estructura de estado: casi todo contrato que lleva **saldos por usuario** es, en el fondo, un mapping.

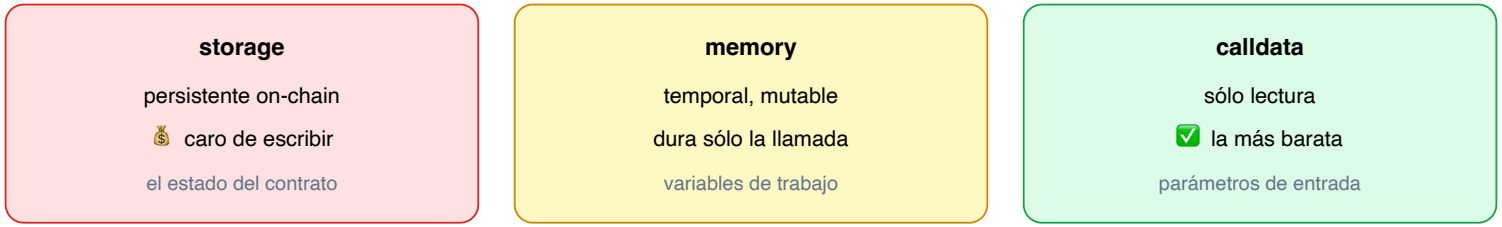
`private` no es privacidad

`private` controla el acceso **desde otros contratos** —
el contenido del storage es **legible por cualquiera** desde afuera de la blockchain.

> El estado es público por diseño (Clase 1). **Nunca guardar secretos en storage**,
> ni "escondidos" en variables `private`.

Data location: ¿dónde vive el dato?

Fuente clásica de **bugs** y de **costos de gas**:



Data location en código

```
mapping(address => uint256) private balances;  // sólo existe en storage

function nombres(string calldata n)
    external pure
    returns (string memory)
{
    return n;   // calldata entra (lectura barata), memory sale
}
```

> Equivocarse de location —p. ej. copiar a `memory` creyendo que se modifica el estado—
> es un error **muy común** al empezar.

Ejemplo del error clásico de location

```
struct Usuario { uint256 saldo; }
mapping(address => Usuario) usuarios;

function malRomper() external {
    Usuario memory u = usuarios[msg.sender];  // ← COPIA a memory
    u.saldo += 10;                            // modifica la copia...
}                                             // ...el storage NO cambia ✗

function bienHacer() external {
    Usuario storage u = usuarios[msg.sender]; // ← REFERENCIA a storage
    u.saldo += 10;                            // modifica el estado real ✓
}
```

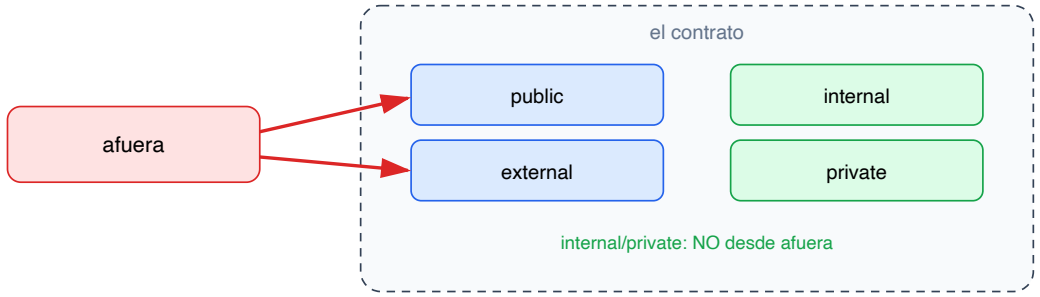
`memory` vs `storage` en una variable local **cambia por completo** el comportamiento.

SEGMENTO 4

Funciones

visibilidad · mutabilidad · errores

Visibilidad



	Quién puede llamar
`public`	interna **y** externa
`external`	sólo desde afuera (más barata para args grandes)
`internal`	este contrato y herederos
`private`	sólo este contrato

> Regla práctica para empezar: `external` para la API, `internal` para helpers; `public` sólo cuando de verdad hace falta llamar desde adentro **y** desde afuera.

Mutabilidad

Modificador	Qué puede hacer
`view`	**lee** estado, no escribe
`pure`	**ni** lee **ni** escribe
`payable`	puede **recibir ETH**

```
function leer() external view returns (uint256) { return total; }
function sumar(uint256 a, uint256 b) external pure returns (uint256) { return a + b; }
function depositar() external payable { /* msg.value trae ETH */ }
```

El compilador **verifica** estas promesas: si una `view` intenta escribir, no compila.

> Y una función **sin** `payable` **rechaza** cualquier ETH que le manden (la tx revierte).
> Por eso `depositar()` de arriba lleva `payable` — y `leer()` no.

Funciones especiales

Función	Cuándo corre
`constructor()`	una vez, al deploy
`receive() external payable`	llega **ETH sin datos** (transferencia "pelada")
`fallback() external [payable]`	la llamada **no matchea ninguna función**

`receive` **tiene que ser** `payable`. `fallback` puede serlo **o no**:

El contrato tiene...	Transferencia "pelada" (sin datos)
`receive`	la atiende `receive`
sólo `fallback` **payable**	la atiende `fallback` — **no revierte**
sólo `fallback` **sin** `payable`	**revierte**
ninguno de los dos	**revierte**

Con datos que no matchean ninguna función, siempre va a `fallback`.*

Errores: require, revert, custom errors

```
error NoAutorizado();              // custom error (desde 0.8.4)

modifier soloOwner() {
    if (msg.sender != owner) revert NoAutorizado();
    _;   // ← acá se inserta el cuerpo de la función
}

function reset() external soloOwner {
    total = 0;
}
```

- `require` / `revert` **revierten** todo el cambio si la condición falla
- **Custom errors**: más **baratos** en gas que strings de `require(...)`
- **Modifiers**: pre-condición **reutilizable**

require vs custom error

```
// Estilo viejo: string (cuesta más gas, el string vive en el bytecode)
require(msg.sender == owner, "No autorizado");

// Estilo moderno: custom error (más barato)
if (msg.sender != owner) revert NoAutorizado();
```

> Convención: nombrar los errores como **sustantivos** descriptivos (`NoAutorizado`, `SaldoInsuficiente`), no como frases.

`assert(...)`, en una línea: es para **invariantes internas** ("esto no puede pasar jamás") — si falla, es un bug del contrato, no un input inválido. Los analizadores de la Clase 6 lo tratan distinto de `require` por esa semántica.

Los custom errors pueden llevar datos

Y ahí está buena parte de su valor:

```
error SaldoInsuficiente(uint256 pedido, uint256 disponible);

if (monto > saldo) revert SaldoInsuficiente(monto, saldo);
```

Quien recibe el revert puede **decodificar esos valores** —el test, el frontend, el explorador—: mucho más útil para diagnosticar que un string, y aún así **más barato**.

> Cada error tiene un **selector** que lo identifica. En la Clase 3 los tests van a esperar errores por nombre usándolo: `vm.expectRevert(MiContrato.MiError.selector)`.

Qué significa "revertir", exactamente

La tx se deshace **entera y atómicamente**:

- todos los cambios de estado
- todas las transferencias de ETH
- todos los eventos emitidos

No hay "quedó por la mitad". Y vale **a través de toda la cadena de llamadas**:
si A llamó a B y B revierte, lo de A también se deshace
(salvo manejo explícito con `try/catch` *— mención, no lo desarrollamos)*.

> **Pero el gas consumido hasta ahí se paga igual.** Revertir no es gratis —
> la diferencia con Cardano que quedó planteada en la Clase 1.

— intervalo —

SEGMENTO 5

Eventos y contexto de la transacción

Eventos: el puente al off-chain

```
event Incrementado(address indexed quien, uint256 nuevoTotal);

function incrementar() external {
    total += 1;
    emit Incrementado(msg.sender, total);
}
```

Los contratos **no** pueden leer logs. `indexed` (hasta 3 por evento) manda ese campo a los *topics* del log, que es lo que se puede **filtrar**: acá `quien` es `indexed`, así que un frontend puede pedir *"todos los `Incrementado` de esta cuenta"*. `nuevoTotal` **no** es `indexed` — viaja en el log, pero no se puede buscar por él.

¿Por qué eventos y no storage para el historial?

	Costo aprox.
Emitir un evento	**cientos** de gas
Guardar una entrada de historial en storage	**~20.000** de gas

...para datos que el contrato **nunca necesita releer**.

> **Storage para lo que el contrato necesita decidir;**
> **eventos para lo que los humanos y las UIs necesitan saber.**

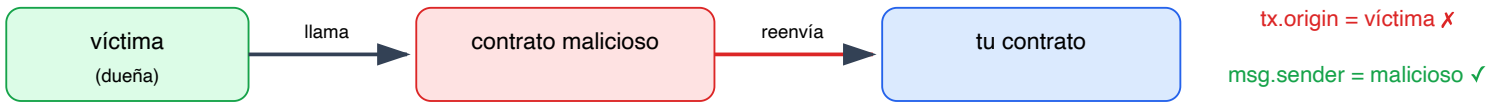
El contexto de la transacción

Variables globales disponibles en cualquier función:

Global	Qué es
`msg.sender`	quién llama (dirección **directa**: EOA o contrato)
`msg.value`	ETH enviado con la llamada (en wei)
`msg.data`	el calldata crudo de la llamada (*)
`block.timestamp`	tiempo del bloque (lo pone el **productor** del bloque)
`block.number`	número de bloque actual
`tx.origin`	la **EOA** que originó toda la cadena de llamadas

Ownership básico: guardar `owner` en el constructor (idealmente `immutable`) y chequearlo con un modifier.

⚠️ Aviso temprano: `tx.origin`



No usar `tx.origin` para autorización — usar `msg.sender`

(Se profundiza en la Clase 6.)

Enviar ETH *desde* un contrato

La forma actual — `call` con valor, chequeando el resultado:

```
error TransferFailed();          // un custom error, como los de recién

(bool ok, ) = destinatario.call{value: monto}("");
if (!ok) revert TransferFailed();
```

Ese `("")` es el **calldata vacío**: sólo plata. Si además querés **llamar a una función**, ahí va la llamada — o, si conocés el contrato, `Otro(dir).f{value: monto}(arg)`.

Existen `transfer` y `send` (heredadas, con límite fijo de gas que las volvió frágiles); hoy la recomendación es **`call` + chequear `ok`**.

> Si el destinatario es un **contrato**, esa línea **ejecuta código ajeno en el medio de tu función**.

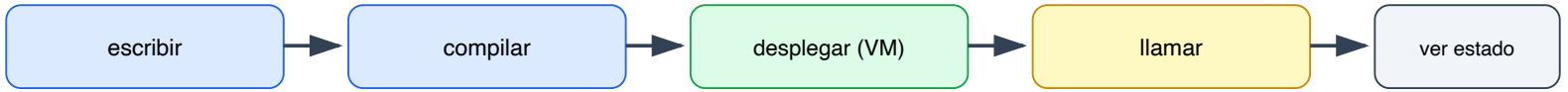
De ahí salen **checks-effects-interactions** (Clase 3) y **reentrancy** (Clases 6–7).

SEGMENTO 6

Tooling + mini-demo

Empezamos con Remix

`https://remix.ethereum.org` — editor en el navegador, **sin instalar nada**.
Ideal para el primer contacto.



La demo en vivo (1/2) — deploy y primera tx

Contrato listo para pegar: **`ethereum/src/Contador.sol`** — junta las piezas que fueron apareciendo sueltas (estado, evento, custom error + modifier, `incrementar()`, `reset()`).

1. Crear `Contador.sol`, **compilar** (Ctrl+S compila solo; la versión debe matchear el `pragma`)
2. **Desplegar en la "Remix VM"**: blockchain simulada en el navegador — cuentas de juguete con 100 ETH falsos. Señalar el **dropdown de cuentas**: cada una es una EOA
3. Leer el contrato desplegado:
   - botones **azules** = `view` / getters → **gratis**, no crean tx
   - botones **naranjas** = escriben → **cada click es una transacción**
4. Llamar `incrementar()` y abrir la tx en la consola: ver el **gas usado** y, en *logs*, el evento `Incrementado` decodificado

La demo en vivo (2/2) — el contexto en acción

5. En la consola, la tx trae **`data: 0x…`** — ése es el **`msg.data`**. Los primeros **4 bytes** son el **selector** de la función; **el resto, los parámetros**. Acá no hay resto: `incrementar()` no toma nada, y por eso **`decoded input`** dice `{}`
6. Llamar `reset()` y **comparar los dos** **`data`**: distinto **selector** → es lo que dice **qué función** se llamó
7. **Cambiar de cuenta** y llamar `reset()` → **revierte** con `NoAutorizado()` y `total` **no cambió**. Volver a la cuenta original → funciona
8. Mandar ETH "pelado" desde el campo **Value** → **revierte**: no hay `receive()`

> **Llamar a una función no es "llamar"**: es mandarle **bytes** a una dirección, y el contrato decide qué hacer con ellos. Y con el paso 7: los **mismos** bytes, **distinto** **`msg.sender`** → **distinto resultado**.

Actividad — leer un contrato que no vieron

En parejas, "leerlo completo" (si no llega el tiempo, queda de tarea):

```
contract Alcancia {
    address public immutable owner;
    mapping(address => uint256) private ahorros;
    bool public cerrada;

    event Ahorro(address indexed quien,
                 uint256 monto);
    event Cierre(uint256 totalFinal);

    error EstaCerrada();
    error NoEsOwner();

    constructor() { owner = msg.sender; }

    modifier soloOwner() {
        if (msg.sender != owner)
            revert NoEsOwner();
        _;
    }

    function ahorrar() external payable {
        if (cerrada) revert EstaCerrada();
        ahorros[msg.sender] += msg.value;
        emit Ahorro(msg.sender, msg.value);
    }

    function miAhorro() external view
            returns (uint256) {
        return ahorros[msg.sender];
    }

    function cerrar() external soloOwner {
        cerrada = true;
        emit Cierre(address(this).balance);
    }
}
```

Son **tres** funciones. Contalas — hace falta para la consigna 2, y para la pregunta que cierra la clase.

Actividad — las consignas

1. Variables de estado y **dónde vive** cada una
2. **Visibilidad y mutabilidad** de cada función
3. ¿**Quién** puede llamar qué? (¿y quién es `owner`?)
4. ¿Qué **eventos** se emiten y cuándo?
5. Si alguien llama `ahorrar()` con 1 ETH **después del cierre**, ¿qué pasa con su ETH?
6. ¿`ahorros` es **secreto**, dado que es `private`?

(Las respuestas las vemos juntos al cierre.) *

¿Cómo saco la plata de la `Alcancia`?

> **No se puede: falta una función de retiro.** El contrato acepta ETH pero no tiene ninguna función que lo envíe — quedó **atrapado para siempre** (el código es inmutable; no hay a quién pedirle un fix).

Escribir esa función de retiro es **exactamente** el trabajo de la **Clase 3** — y por qué escribirla **bien** es mucho más delicado de lo que parece.

Un dato que se ve en la Clase 6

Desde **Solidity 0.8** el overflow/underflow **se chequea por defecto** (revierte).

> La vulnerabilidad clásica **no desapareció**: ahora vive en bloques `unchecked` y en *casts*.

```
unchecked {
    balances[msg.sender] += amount;   // puede dar wraparound (sin chequeo)
}

uint8 chico = uint8(valorGrande);     // trunca en silencio si no entra
```

Lo retomamos al ver el **modelo de amenazas**.

Lo que se ve más adelante

Hoy apareció…	Vuelve en…
`(bool ok, ) = dest.call{value: monto}("")`	**Clase 3** — la función de retiro y el orden estado/interacción (CEI)
"esa línea ejecuta código ajeno en el medio de tu función"	**Clases 6–7** — reentrancy
Overflow en `unchecked` y casts	**Clase 6** — modelo de amenazas
`tx.origin` · código inmutable (proxies) · `block.timestamp` lo pone el productor	**Clase 6**
Remix	→ **Foundry** desde la Clase 3

Lecturas

- *Mastering Ethereum*, **2ª ed. (2025) — cap. 7**, *Smart Contracts and Solidity (actualizada post-Merge; gratis online y en* `ethereumbook` *)*
- Documentación oficial: `docs.soliditylang.org` (versión actual)
- Para practicar con ejemplos cortos: `solidity-by-example.org` *(ejemplos por tema, misma filosofía que esta clase)*

Próxima clase

Clase 3 — Taller Solidity con Foundry

Construir una bóveda (`Vault.sol`) paso a paso, tests con `forge test` ,
y el patrón **checks-effects-interactions**.