Clase 3

Taller Solidity

Foundry, Vault y tests

Tesis de la clase

> Un contrato es **software**: necesita entorno de desarrollo serio, pruebas automatizadas y un proceso de deploy reproducible.

Todo lo que se construye hoy se **auditará y explotará en la Clase 7**.

Objetivos de aprendizaje

Al terminar, vas a poder:

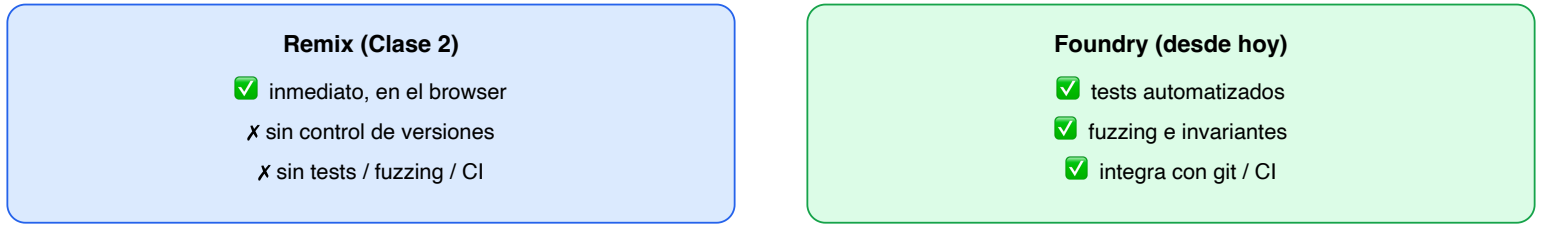
- Instalar y usar **Foundry** (`forge` , `anvil`)
- Leer y navegar la estructura de un proyecto Foundry
- **Construir `Vault.sol` paso a paso**, entendiendo cada decisión (custom errors, events, checks-effects-interactions)
- Escribir **tests** en Solidity con `forge test`
- **Desplegar** en una blockchain local con `anvil` e **interactuar** con `cast`

SEGMENTO 1 · ~20 min

De Remix a Foundry

¿por qué cambiar?

Remix sirve para explorar, no para un proyecto serio



> Y el análisis automatizado que viene después del taller corre sobre Foundry:
> empezar hoy evita cambiar de entorno a mitad de camino.

Qué trae Foundry

Toolkit de línea de comandos, escrito en **Rust**:

Herramienta	Para qué
`forge`	compilar, testear, desplegar, fuzzear
`anvil`	nodo local de Ethereum (reemplaza Ganache/Hardhat node)
`cast`	interactuar con contratos desde la terminal ← **lo usamos hoy**

Instalación

```
# instala foundryup (el instalador de Foundry)
curl -L https://foundry.paradigm.xyz | bash

# recargá el shell o abrí una terminal nueva, luego:
foundryup

# verificar
forge --version   # forge Version: 1.7.1 (o la vigente)
anvil --version   # anvil Version: 1.7.1
```

Estructura de un proyecto Foundry

```
forge init mi-proyecto
cd mi-proyecto
```

```
src/         → contratos
test/        → tests (son contratos Solidity)
script/      → scripts de deploy
lib/         → dependencias (forge-std, OpenZeppelin, ...)
foundry.toml → configuración
```

En la clase trabajamos sobre `ethereum/` (ya tiene `foundry.toml` y `Vault.sol` con TODOs):

```
cd ethereum/
forge build   # debe compilar sin errores
```

Cómo resuelve Foundry los `import`

En los tests escribimos `import "forge-std/Test.sol"`. Ese prefijo es un **remapping**:

```
forge-std/  →  lib/forge-std/src/
```

- Las dependencias viven en `lib/`, instaladas como **submódulos de git** (`forge install`)
- Si `forge build` dice `forge-std/Test.sol not found` en un clon limpio: faltan los submódulos → `git submodule update --init --recursive`

> Es el **problema de setup #1** al clonar el repo: si `forge build` no encuentra `forge-std`, empezá por acá. *

Tres comandos útiles desde hoy

```
forge fmt                # formatea el código (como prettier / gofmt)
forge test --gas-report  # tabla de gas por función
forge build --sizes      # tamaño del bytecode (límite: 24 KB por contrato)
```

SEGMENTO 2 · ~40 min

Construcción guiada

`Vault.sol`

Qué construimos

Una **bóveda de ETH**: los usuarios **depositan**, la bóveda lleva su balance, y pueden **retirar**.

Simple, pero incluye **todos** los patrones importantes:

- estado en un `mapping`
- eventos con `indexed`
- custom errors
- **checks-effects-interactions** ← patrón muy importante

> Archivo completo: `ethereum/src/Vault.sol`

2a. Estado y eventos

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract Vault {
    mapping(address => uint256) private balances;

    event Deposit(address indexed account, uint256 amount);
    event Withdraw(address indexed account, uint256 amount);
```

- `private` el `mapping`: no necesitamos que otro contrato lo lea directo
  → lo exponemos sólo con `balanceOf`
- `indexed` permite **filtrar** los eventos por cuenta desde un indexer

2b. Custom errors

```
    error InsufficientBalance();
    error TransferFailed();
```

- Desde **Solidity 0.8.4**, los custom errors son **más baratos** en gas que los strings de `require`
- Convención: **sustantivos** descriptivos, no frases

2c. `deposit()`

```
    function deposit() external payable {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }
```

- `payable` es **obligatorio** para recibir ETH
- `msg.value` es el ETH enviado con la llamada
- El evento se emite **después** de actualizar el estado

2d. `withdraw()` — checks-effects-interactions

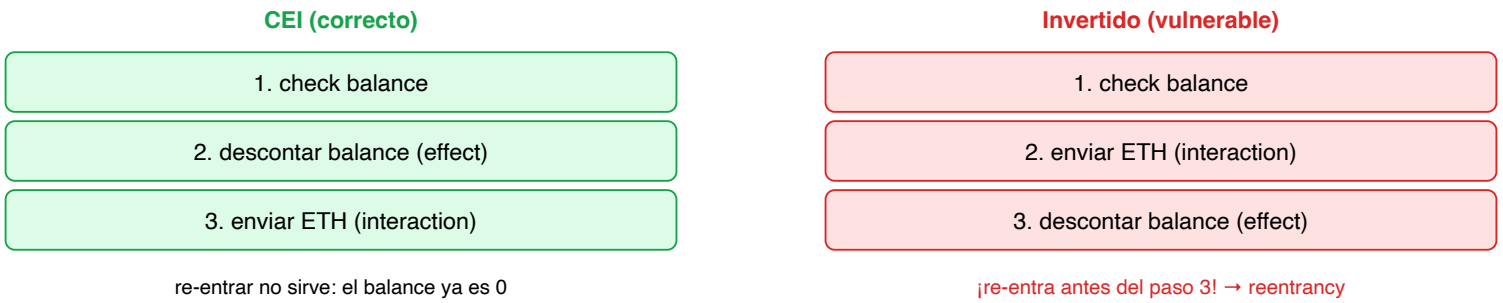
```
    function withdraw(uint256 amount) external {
        // CHECKS: precondiciones antes de tocar nada
        if (balances[msg.sender] < amount) revert InsufficientBalance();

        // EFFECTS: mutamos el estado ANTES de salir del contrato
        balances[msg.sender] -= amount;

        // INTERACTION: ahora sí llamamos a código externo
        (bool ok, ) = msg.sender.call{value: amount}("");
        if (!ok) revert TransferFailed();

        emit Withdraw(msg.sender, amount);
    }
```

¿Por qué el orden importa?



La regla: todo lo que muta estado propio va **antes** de cualquier llamada externa.

¿Por qué `call` y no `transfer`?

Tres formas de mandar ETH: `transfer`, `send`, `call`.

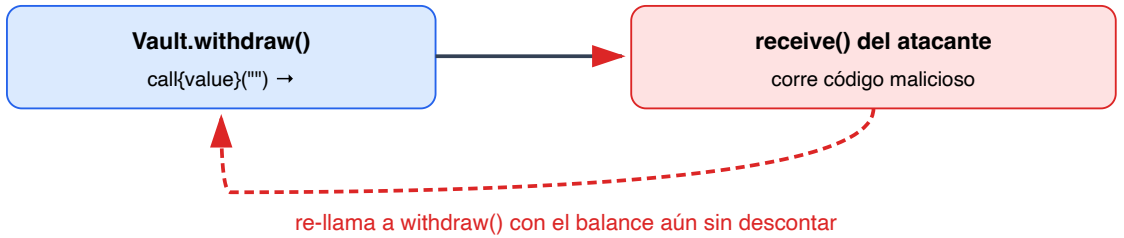
	Gas que reenvía	Falla
`transfer`	**2300 fijo**	revierte solo
`call{value:}`	**todo el disponible**	devuelve `false` → hay que chequear `ok`

- `transfer` con 2300 quedó corto tras cambios de costos (EIP-1884) → rompía contratos legítimos
- Hoy se recomienda `(bool ok,) = destino.call{value: monto}("")` **chequeando** `ok`

> El trade-off es toda la clase: `call` reenvía gas suficiente para **re-entrar**.
> Esa flexibilidad es la que CEI cierra. **Chequear** `ok` **es obligatorio.**

`call{value: amount}("")` — ¿qué es ese `("")`?

El segundo argumento es el **calldata**. Al pasar `""` (vacío), mandamos ETH "a secas", sin invocar ninguna función.



Recibir ETH sin datos ejecuta el `receive()` del destinatario —la función que vieron en la Clase 2—. Si ese destinatario es un **contrato**, ahí corre **código ajeno antes de que `withdraw` termine**. Por eso el orden de las tres etapas no es decorativo.

Nota sobre el `-=` (Solidity 0.8)

```
balances[msg.sender] -= amount;   // no puede hacer underflow: revierte solo
```

- En 0.8 la resta está **protegida**: si `amount > balance`, revierte
- Pero **no dependemos de eso** — el `check` de arriba ya lo garantiza *

> El criterio: que la corrección no dependa de una red de seguridad del compilador.
> El `check` explícito dice **qué** condición tiene que valer; el underflow sólo evita un desastre si esa condición ya falló.

2e. `withdrawAll()` — lo escribimos juntos

Retira **todo** el saldo del que llama. Mismo patrón, con una diferencia.

```
    function withdrawAll() external {
        // 1. CHECK   — ¿tiene algo? si no, revert InsufficientBalance()
        // 2. ???     — ¿qué hay que hacer ANTES de poner el balance en cero?
        // 3. EFFECT  — balance a cero
        // 4. INTERACTION — enviar y chequear ok
        // 5. emitir Withdraw
    }
```

> La diferencia: en `withdraw(amount)` el monto **te lo pasan**. Acá hay que **leerlo del estado**… y ese mismo estado es el que vas a poner en cero. **El orden importa.**

> Antes de escribirlo: *¿qué pasa si ponés el balance en cero primero y después leés el monto a transferir?*

2f. `balanceOf()` — getter de sólo lectura

```
    function balanceOf(address account) external view returns (uint256) {
        return balances[account];
    }
}
```

- `view` garantiza que **no muta** estado (lo verifica el compilador)
- Podríamos haber hecho `balances` `public`, pero `balanceOf(address)` es la firma **estándar** (la usa ERC-20)

SEGMENTO 3 · ~40 min

Tests con `forge test`

Los tests son contratos Solidity

Heredan de `forge-std/Test.sol`. Cada función que empieza con `test` es un caso; `setUp` corre **antes de cada uno** (aislamiento total).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "forge-std/Test.sol";
import "../src/Vault.sol";

contract VaultTest is Test {
    Vault vault;
    address alice = makeAddr("alice");
    address bob   = makeAddr("bob");

    function setUp() public {
        vault = new Vault();
        vm.deal(alice, 10 ether);   // le damos ETH a alice
        vm.deal(bob,   10 ether);
    }
}
```

¿Quién es `msg.sender` en un test?

El **contrato de test** (`VaultTest`) es el que llama al `Vault` .

- Sin hacer nada → `msg.sender` dentro del `Vault` es la dirección de `VaultTest` , **no** **`alice`**
- Por eso existe `vm.prank(alice)` : suplanta el sender de la **siguiente** llamada

> Sin `prank` , **todos** los depósitos vendrían de una sola dirección (la del test)
> y no podríamos probar el aislamiento entre usuarios.

Cheatcodes esenciales

`vm.*` son **cheatcodes**: funciones especiales de `forge-std` que manipulan el entorno de test sin una tx real.

Cheatcode	Qué hace
`vm.deal(addr, x)`	fija el balance ETH de `addr`
`vm.prank(addr)`	la **siguiente** llamada viene de `addr`
`vm.startPrank/stopPrank`	bloque de llamadas como `addr`
`vm.expectRevert(...)`	la siguiente llamada debe revertir con ese error
`vm.expectEmit(...)`	la siguiente llamada debe emitir ese evento
`vm.warp(ts)`	cambia `block.timestamp`

`makeAddr("alice")` genera una dirección **determinista** (aparece como "alice" en los logs).

3c. Tests de `deposit`

```
function test_Deposit() public {
    vm.prank(alice);
    vault.deposit{value: 1 ether}();
    assertEq(vault.balanceOf(alice), 1 ether);
}

function test_Deposit_EmitsEvent() public {
    vm.expectEmit(true, false, false, true); // qué campos verificar
    emit Vault.Deposit(alice, 1 ether);       // el evento esperado
    vm.prank(alice);
    vault.deposit{value: 1 ether}();
}
```

`expectEmit(i1, i2, i3, data)`: los tres bools son los parámetros `indexed`; el cuarto compara el data no-indexed.

3d. Tests de `withdraw`

```
function test_Withdraw() public {
    vm.startPrank(alice);
    vault.deposit{value: 3 ether}();
    uint256 antes = alice.balance;
    vault.withdraw(1 ether);
    assertEq(vault.balanceOf(alice), 2 ether);
    assertEq(alice.balance, antes + 1 ether);
    vm.stopPrank();
}

function test_Withdraw_RevertsOnInsufficientBalance() public {
    vm.prank(alice);
    vault.deposit{value: 1 ether}();
    vm.prank(alice);
    vm.expectRevert(Vault.InsufficientBalance.selector);
    vault.withdraw(2 ether);
}
```

3e. Tests de `withdrawAll` — el par que no puede faltar

Todo comportamiento se testea **de a dos**: el caso feliz y el que debe fallar.

```
function test_WithdrawAll() public {
    vm.startPrank(alice);
    vault.deposit{value: 3 ether}();
    uint256 antes = alice.balance;
    vault.withdrawAll();
    assertEq(vault.balanceOf(alice), 0);          // ← el saldo interno quedó en 0
    assertEq(alice.balance, antes + 3 ether);     // ← y el ETH volvió de verdad
    vm.stopPrank();
}
```

Las dos aserciones son distintas y hacen falta las dos: la primera mira la contabilidad, la segunda mira que el ETH se haya movido. Un bug puede pasar una y fallar la otra.

> **Falta el par:** el test de que `withdrawAll()` con saldo cero **revierte**.

3f. Aislamiento entre usuarios

```
function test_BalancesAreIsolated() public {
    vm.prank(alice);
    vault.deposit{value: 2 ether}();
    vm.prank(bob);
    vault.deposit{value: 5 ether}();

    assertEq(vault.balanceOf(alice), 2 ether);
    assertEq(vault.balanceOf(bob),   5 ether);
}
```

El balance de uno **no afecta** al del otro — propiedad básica de una bóveda.

Correr los tests — la `v` gradúa la verbosidad

```
forge test              # solo conteo pasa/falla
forge test -vv          # + nombres y valores de assert que fallan
forge test -vvv         # + traces de llamadas y eventos (para debuggear)
forge test -vvvv        # + todo, incluido setUp y stack de cada llamada
forge test --match-test test_Withdraw   # filtrar por nombre
```

> Con `-vv` alcanza para el día a día. Cuando algo se rompe, `-vvv` muestra la cascada completa de llamadas, que es donde se ve qué llamó a qué.

Los tests prueban lo que se te ocurrió testear

Nuestra suite cubre los caminos que **nosotros** imaginamos: depositar, retirar, retirar de más, aislamiento entre usuarios.

> Un **atacante** no juega a lo que vos imaginaste. Por eso el análisis de seguridad no es "escribir más tests": es **otra disciplina**.

Recordar esta recomendación: **cada comportamiento se testea de a dos** — el caso que funciona y el que debe fallar.

SEGMENTO 4 · ~20 min

Deploy con `anvil`

anvil: una blockchain de juguete, en tu máquina

```
anvil     # y dejalo corriendo en esa terminal
```

```
Available Accounts
(0) 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 (10000.000000000000000000 ETH)
(1) 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000.000000000000000000 ETH)
...                                    (10 en total)

Private Keys
(0) 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
...

Listening on 127.0.0.1:8545
```

- **10 cuentas con 10.000 ETH** y claves **fijas** (siempre las mismas) — para desarrollo, **jamás** en mainnet
- **RPC** en `127.0.0.1:8545` · **chain id 31337**
- **Minado instantáneo**: cada tx entra en su propio bloque, sin esperar

Dos terminales, un mismo contenedor

`anvil` ocupa una terminal entera. Para tener la segunda **sin salir del contenedor**:

```
# TERMINAL 1 — arrancar el contenedor CON NOMBRE
podman run -it --rm --name curso --userns=keep-id  -v "$PWD":/curso:z curso-sc:arm64
cd ethereum && anvil
```

```
# TERMINAL 2 — entrar al MISMO contenedor
podman exec -it curso bash
cd /curso/ethereum
```

> La clave es el **`--name`**: sin él no hay cómo volver a entrar.
> Los dos shells comparten el mismo `127.0.0.1:8545`, así que no hace falta publicar puertos.

El script de deploy

```
// ethereum/script/DeployVault.s.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import "forge-std/Script.sol";
import "../src/Vault.sol";

contract DeployVault is Script {
    function run() external {
        vm.startBroadcast();
        new Vault();
        vm.stopBroadcast();
    }
}
```

`vm` viene de `is Script` (como `is Test` en los tests). Pero `startBroadcast` **no** es un cheatcode: lo que va entre `start` y `stop` se vuelve **transacciones reales**.

```
# terminal 2 · sin --broadcast sólo simula (dry run)
forge script script/DeployVault.s.sol --rpc-url http://127.0.0.1:8545 \
  --private-key 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80 --broadcast
```

El script imprime la dirección; también queda en `broadcast/DeployVault.s.sol/31337/run-latest.json`.

Ahora sí: hablarle al contrato con `cast`

`cast` es el cliente de línea de comandos. **Leer** no cuesta gas; **escribir** es una tx.

```
export VAULT=0x5FbDB2315678afecb367f032d93F642f64180aa3   # la dirección del deploy
export PK=0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
export YO=$(cast wallet address --private-key $PK)         # mi dirección, derivada de la clave
```

Comando	Qué hace
`cast call`	**lee**: simula, **devuelve el valor** y no toca la cadena
`cast send`	**escribe**: arma, firma y manda una tx — devuelve un **recibo**, no el valor
`cast balance`	ETH de una dirección
`cast from-wei` / `to-wei`	convertir sin contar ceros

> **La regla:** si la función es `view`, `call`. Si cambia estado, `send`.
> Y como `send` **no te devuelve el valor de retorno**, después hay que **leer con `call`**
> para ver qué pasó. Por eso el ciclo de abajo alterna.

El ciclo completo, en vivo

```
cast call $VAULT "balanceOf(address)(uint256)" $YO          # → 0
cast send $VAULT "deposit()" --value 2ether --private-key $PK
```

```
blockNumber          2
gasUsed              45194
status               1 (success)      ← 1 es éxito, 0 es revert
```

```
cast call $VAULT "balanceOf(address)(uint256)" $YO          # → 2000000000000000000 [2e18]
cast balance $VAULT                                        # → 2000000000000000000
cast send $VAULT "withdraw(uint256)" $(cast to-wei 0.5 ether) --private-key $PK
cast call $VAULT "balanceOf(address)(uint256)" $YO          # → 1500000000000000000 [1.5e18]
```

El saldo interno y el ETH del contrato coinciden. Esa igualdad tiene nombre: es una **propiedad** del Vault, algo que debe valer siempre, para cualquier secuencia de operaciones.

Y qué pasa cuando algo sale mal

```
cast send $VAULT "withdraw(uint256)" $(cast to-wei 99 ether) --private-key $PK
```

```
Error: Failed to estimate gas: server returned an error response:
error code 3: execution reverted: custom error 0xf4d678b8: InsufficientBalance
```

- El **custom error viaja por la red**: `cast` lo decodifica y lo muestra por nombre
- Ese `0xf4d678b8` son los **4 bytes del selector** — es todo lo que va on-chain
- La tx **ni siquiera se envía**: `cast` estima el gas primero, la simulación revierte, y aborta **antes de firmar**. No se paga nada, ni se consume el nonce *

> Es el mismo `InsufficientBalance` que escribieron en el contrato y que testearon con `vm.expectRevert`. **Tres vistas del mismo error**: el código, el test, y la red.

Manos al teclado (~5 min)

Con `anvil` corriendo en una terminal y el Vault desplegado:

1. Depositar **1 ETH** desde la cuenta `(0)` y verificar el saldo
2. Depositar **3 ETH** desde la cuenta `(1)` — otra `--private-key`
3. Leer los **dos** saldos: ¿se mezclaron?
4. `cast balance $VAULT` → ¿cuánto custodia el contrato? ¿coincide con la suma?
5. Intentar que la cuenta `(1)` retire **5 ETH** → ¿qué error sale?

> Si algo no anda: `cast block-number` dice si anvil está vivo, y `cast from-wei <n>` convierte cualquier monto a ETH legible.

SEGMENTO 5 · ~10 min

Cierre

y qué se ve más adelante

Lo que queda hecho hoy

> Un contrato **construido con criterio**, con tests que lo respaldan y un deploy reproducible. Eso es la línea base de cualquier proyecto serio.

- El estado se actualiza **antes** de cualquier llamada externa
- Cada comportamiento tiene su **par** de tests
- El deploy es un **script versionado**, no clicks

> En la **Clase 7** vamos a tomar este mismo Vault, romperlo a propósito, y ver cuánto de lo que hicimos hoy lo salva — y cuánto no.

Bonus opcional — los mismos tests, desde afuera

La misma suite del `Vault`, escrita en **JavaScript** con el stack clásico de Ethereum (**Hardhat + ethers + chai**). El contrato es el mismo — no hay copia.

```
it("withdraw de mas revierte con InsufficientBalance", async () => {
  const { vault, alice } = await loadFixture(desplegar);
  await vault.connect(alice).deposit({ value: ethers.parseEther("1") });
  await expect(vault.connect(alice).withdraw(ethers.parseEther("2")))
    .to.be.revertedWithCustomError(vault, "InsufficientBalance");
});
```

	Foundry (Solidity)	Off-chain (JS)
Actuar como otro usuario	`vm.prank(alice)`	`vault.connect(alice)`
Estado limpio por test	automático	`loadFixture(desplegar)`
Revert tipado	`vm.expectRevert(...selector)`	`.to.be.revertedWithCustomError(...)`
Fuzzing	incluido	no viene
Velocidad	13 tests en **20 ms**	6 tests en 55 ms

> **No viene preinstalado** (pesa 372 MB, un tercio de la imagen). Se instala con `cd ethereum/offchain && npm install`. Pasos en `ethereum/offchain/README.md`.

¿Para qué sirve cada uno?

> No compiten: **cubren cosas distintas.**

- **En Solidity** se escriben los tests para **auditar y razonar sobre el contrato**: corren dentro de la EVM, tienen cheatcodes para manipular el estado, y traen fuzzing e invariantes gratis.
- **Desde el off-chain** se atrapa lo que vive **en la frontera**: que el ABI sea el que la app espera, que los argumentos se codifiquen bien. Los tests de Solidity **nunca cruzan esa frontera** — ahí el compilador les garantiza los tipos. Un frontend con el ABI viejo pasa **todos** los tests de Foundry y falla en producción.

> Y sirve para lo que viene: contá **qué necesita** este off-chain para mandar una tx —dirección, ABI, argumentos, firma—. En la **Clase 4** vas a ver los **6 pasos** que hace falta en Cardano.

SEGMENTO 5 · ~30 min

Actividad

la Alcancía, terminada

Se acuerdan de la `Alcancia`?

En la **Clase 2** la leyeron entera y llegaron a una conclusión:

> *"¿Cómo saco la plata?"* → **No se puede: falta una función de retiro.**
> El contrato acepta ETH y no tiene ninguna función que lo envíe.

Está en el repo, en `src/Alcancia.sol`, **con ese agujero todavía abierto**.

Hoy lo cierran ustedes.

La consigna

```
cd ethereum/
forge test --match-contract AlcanciaTest
#   1 passed · 4 failed     ← los 4 son suyos
```

1. **Implementar `retirar()`** en `src/Alcancia.sol` — recupera todo lo ahorrado, aplicando **checks-effects-interactions**
2. **Escribir los 4 tests** de `test/Alcancia.t.sol` (hay uno resuelto de modelo)
3. Llegar a **5 passed**

> No copien el `withdraw` del Vault sin pensar: la Alcancía tiene algo que el Vault no tiene. *

La decisión que no tiene respuesta única

La `Alcancia` tiene un estado que el Vault no tiene: **`cerrada`**.

> **¿Se puede retirar con la alcancía cerrada?**

Si decís NO	Si decís SÍ
`cerrar()` congela los fondos **para siempre**	¿Para qué sirve `cerrar()`?
¿Es una alcancía o una trampa?	¿Qué impide exactamente?
¿Quién se queda con el ETH?	¿Alcanza con bloquear depósitos?

Las dos son defendibles. Lo que no es defendible es no haberlo pensado.
Su implementación y sus tests tienen que ser **coherentes** entre sí.

> En la puesta en común: cada grupo dice qué eligió y **por qué**.

Lecturas

- **Foundry Book**: `book.getfoundry.sh` (caps. "Getting Started" y "Writing Tests")
- *Mastering Ethereum*, **2ª ed. (2025)** — **cap. 9**, *Smart Contract Security*
- **OWASP Smart Contract Top 10 (2026)** — la clasificación vigente
- **SWC-107** (Reentrancy): `swcregistry.io/docs/SWC-107` — discontinuado, pero es la nomenclatura de los reportes viejos

Próxima clase

Clase 4 — Intro Aiken/Plinth

Cambiamos de blockchain: validators en EUTXO, datum/redeemer/contexto,
y el flujo on-chain / off-chain de Cardano.