Clase 1

Fundamentos de blockchain y comparación de modelos

Profesor: Diego Garbervetsky

Cuentas (Ethereum) vs EUTXO (Cardano)

Tesis de la clase — y del curso

> Cómo se **escribe** y cómo se **rompe** un contrato
> es una consecuencia directa del
> **modelo de ejecución** de la blockchain.

Todo lo que viene después **cuelga de la distinción que se enseña hoy**.

Objetivos de aprendizaje

Al terminar, vas a poder:

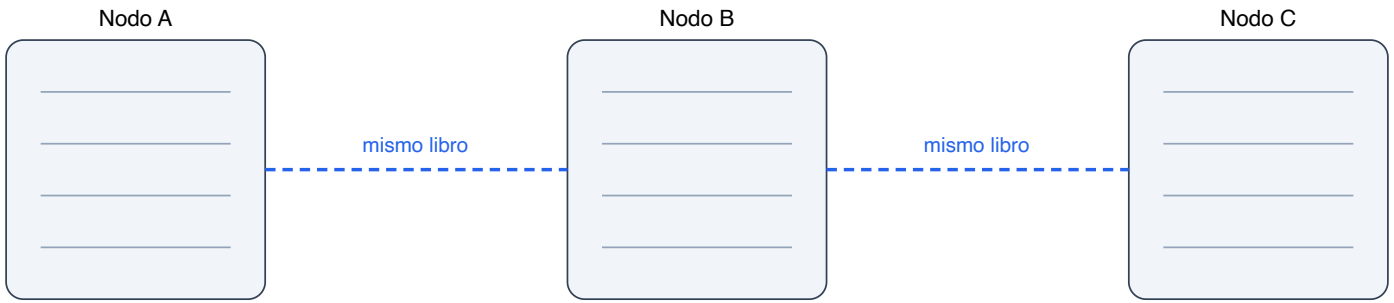
- Explicar **qué problema resuelve** una blockchain y por qué necesita **consenso**
- Describir la **anatomía** de una transacción y un bloque
- **Distinguir el modelo de cuentas del modelo (E)UTXO**
- Anticipar por qué cada uno genera **clases de bugs distintas**
- Identificar **dónde corre el código**: EVM vs UPLC, on-chain vs off-chain

SEGMENTO 1 · ~25 min

¿Qué problema resuelve una blockchain?

Un libro de cuentas (*ledger*) compartido

Replicado entre partes que **no confían** entre sí, **sin autoridad central**.



Cada nodo guarda una copia idéntica; todos tienen que **acordar** qué dice el libro.

Blockchain vs BD tradicional

Una blockchain **es** una base de datos — pero con **muchas más restricciones**.

	BD tradicional	Blockchain
Operaciones	CRUD: `INSERT` / `UPDATE` / `DELETE`	**sólo agregar** (append-only)
Quién escribe	quien autoriza el **admin**	cualquiera con una clave, si cumple las reglas
Quién valida	el servidor — le **creés**	**todos los nodos** re-ejecutan
Rendimiento	decenas de miles de ops/s	**decenas de tx/s**
Costo	barato	caro: se replica **en todos**
¿Te pueden apagar o censurar?	sí, el dueño	no hay dueño a quien pedirle permiso

> Más lenta, más cara y con menos operaciones. Todo ese sacrificio compra **una sola cosa**:
> que no haga falta **confiar en un administrador**.

> Ese "y aún así todos acuerdan" es **todo el problema**. De ahí salen tres preguntas.

Problema 1 — Ordenar las transacciones

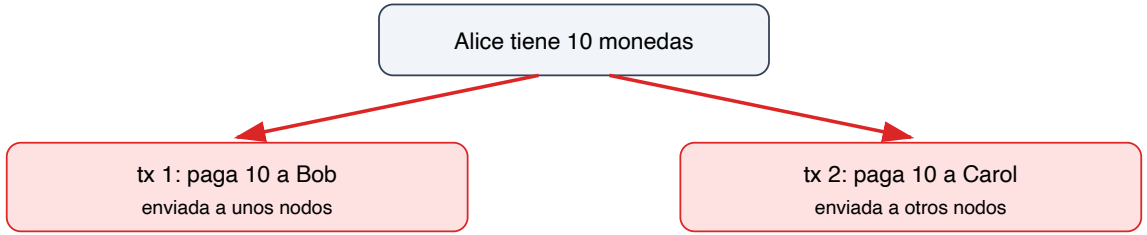
Si dos personas mandan transacciones "al mismo tiempo", ¿cuál va primero?

> No hay un reloj global ni un árbitro. Hay que **acordar un orden** entre nodos que no confían entre sí → eso es el **consenso**.

El orden no es un detalle: como veremos, **quién decide el orden** puede explotarlo (front-running, MEV).

Problema 2 — Evitar el doble gasto

Ejemplo concreto:



Sin acuerdo, **ambas** podrían aceptarse: Alice gastó **20** monedas que no tenía.

El consenso + el orden garantizan que **sólo una** sea válida.

Problema 3 — Que el pasado no se reescriba

Inmutabilidad / finalidad: una vez que una transacción es vieja, debe ser prácticamente imposible borrarla o cambiarla.

- Si cualquiera pudiera reescribir el pasado, podría "des-gastar" sus monedas
- La blockchain está diseñada para que **alterar el pasado sea muy costoso** (lo vemos en la anatomía, con el encadenamiento por hash)

> Tres problemas, una misma raíz: **acordar sin confiar**.
> (El nombre clásico del problema: tolerancia a **fallas bizantinas** — algunos participantes pueden mentir activamente, y el conjunto igual converge.)

Qué NO resuelve una blockchain

Para calibrar expectativas desde el día uno:

No resuelve	Por qué
Privacidad	Al contrario: el libro es **público por diseño** — cualquiera ve cada tx de cada dirección
Veracidad de datos externos	La blockchain garantiza *consistencia interna*, no que "el dólar cotiza a X" sea cierto → **oráculos** (Clase 6)
Escala barata	Replicar todo en todos los nodos es lo opuesto a eficiente — es el precio de no tener admin

> Los oráculos —el puente con el mundo real— son a su vez una **superficie de ataque**.
> Se ven en la Clase 6.

¿Cuándo NO usar blockchain?

> Si existe un tercero en quien **todas** las partes pueden confiar → **usá una base de datos**.
> Va a ser más rápida, más barata y más simple.

Una blockchain recién se justifica cuando el **costo de confiar** en alguien supera al **costo de replicar todo en todos**: partes que no se conocen, que compiten entre sí, o un árbitro que podría hacer trampa.

> El error más común del rubro es exactamente ese: usar una blockchain como base de datos lenta para un problema que no la necesitaba.

SEGMENTO 2 · ~20 min

Anatomía

tx · bloque · hash · estado

Herramienta 1 — Hash criptográfico

Comprime cualquier dato a una **huella de tamaño fijo**:

```
sha256("hola") = b221d9dbb083a7f33428d7c2a3c3198ae925614d70210e28716ccaa7cd4ddb79
sha256("Hola") = e633f4fc79badea1dc5db970cf397c8248bac47cc3acf9915ba60b5d76b0e88f
```

Tres propiedades que usamos todo el tiempo:

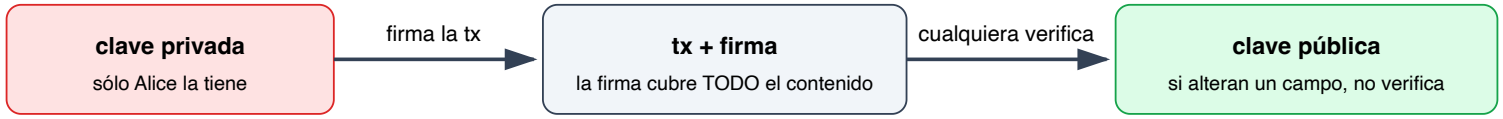
1. **Determinista** — mismo input, misma huella
2. **Avalancha** — un bit distinto cambia TODO (mirá: sólo cambió una mayúscula)
3. **Unidireccional** — de la huella no se recupera el input

> Un hash funciona como **compromiso**: si publico el hash de algo,
> no puedo cambiar ese algo sin que se note.

(Cada blockchain elige su función: Ethereum keccak256, Cardano blake2b — mismas garantías.)

Herramienta 2 — Firma digital

Un **par de claves**: con la **privada** se firma, con la **pública** cualquiera verifica.



- Una tx "de Alice" = una tx **firmada con la clave privada de Alice**. La red no sabe quién es Alice: sabe que la firma verifica.

> **Tu clave privada ES tu identidad.** No hay "recuperar contraseña".
> Quien tiene la clave, tiene los fondos. *(Modelo de amenazas — Clase 6.)*

La transacción: unidad de cambio de estado

Una **transacción** es la instrucción que pide modificar el libro.

```
tx:
  de:      Alice
  a:       Bob
  monto:   3 monedas
  fee:     0.01          ← se paga por procesarla
  firma:   <firma de Alice>
```

Nada cambia en el libro hasta que una tx **válida y firmada** se incluye en un bloque.

Una tx real en cada blockchain (anticipo)

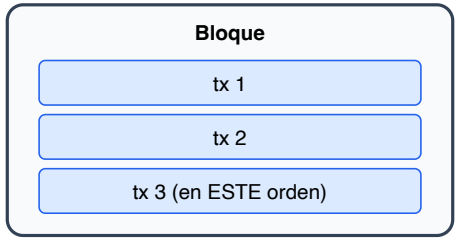
No hace falta entender cada campo — sólo reconocer la **forma**:

```
Ethereum                              Cardano
──────────────────────────────        ──────────────────────────────
nonce:     7                          inputs:   [UTXO#a, UTXO#b]   ← qué consume
to:        0xAB...                    outputs:  [8 → Bob, 1.8 → Alice]
value:     3 ETH                      fee:      0.17 ADA
data:      (llamada a función)        validez:  slot < 12345678
gasLimit / maxFeePerGas               firmas:   [vkey de Alice]
firma:     (v, r, s)
```

Dos cosas para señalar (se retoman en el Segmento 4):

- Ethereum dice **a quién llamar**; Cardano dice **qué consume y qué crea**
- El `nonce` (contador anti-replay por cuenta) en Cardano **no existe**: consumir los inputs *ya* hace la tx irrepetible — anti-replay **por diseño**

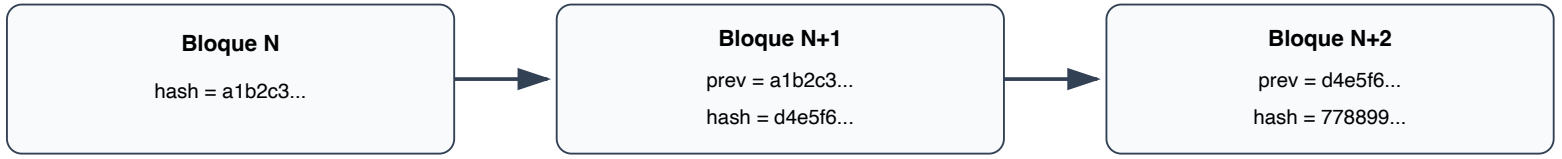
El bloque: un lote ordenado de tx



El bloque fija **cuáles** transacciones entran y **en qué orden** — el output del consenso.

Encadenamiento por hash

Cada bloque incluye el **hash** del bloque anterior:



Si alguien cambia una tx vieja → cambia el hash de ese bloque → **se rompe la cadena** de todos los bloques siguientes. Por eso reescribir el pasado es carísimo.

Fees / gas: el recurso económico

- Procesar una tx **cuesta** (cómputo, almacenamiento)
- Se paga con **fees** (Cardano) / **gas** (Ethereum)
- Sirve para dos cosas:
  - Compensar a quien produce bloques
  - **Evitar el spam** (atacar la red cuesta dinero)

> En Ethereum, cada operación de la EVM tiene un precio en gas — lo veremos en la Clase 2. *Cómo se calcula* difiere entre blockchains — lo vemos en el Segmento 5.

Mini-demo — un bloque real en el explorador

Exploradores de bloques: la blockchain, en vivo.
`etherscan.io` · `cardanoscan.io`

Vamos a abrir **los dos bloques del Merge** (15/09/2022) y buscar tres cosas:
las **transacciones**, el **hash**, y el **prev** que los encadena.

```
Bloque 15537393  ← último bloque Proof of Work
  Hash        0x55b11b91...bb286
  Difficulty  11.055.787.484.078.698

Bloque 15537394  ← primer bloque Proof of Stake
  Parent      0x55b11b91...bb286   ← el hash del anterior. Eso ES la cadena.
  Difficulty  0                    ← el Merge, en un solo número
  Txs         80 · Gas 29.983.006 / 30.000.000  (99,9% lleno)
```

La misma pieza en Cardano: entradas y salidas

Una transacción real, en el explorador:

```
INPUTS  (2)                          OUTPUTS (2)
  524,805026 ADA                       2,000000 ADA
    2,000000 ADA                     524,630549 ADA
  ──────────────                     ──────────────
  526,805026            =            526,630549   +   0,174477 de fee
```

Cierra exacta, hasta el último lovelace — es una regla que el ledger **verifica**: si no cierra, la tx no entra.

- Cada input dice **de qué transacción viene** — un UTXO no es un saldo
- Los inputs se consumen **enteros** → por eso hay vuelto
- **En ningún lado hay un campo "saldo"**: es la suma de tus UTXOs

> Nada de esto es un diagrama teórico: es un sistema corriendo **ahora mismo**, y todo lo que vimos recién está ahí para mirarlo.

La pregunta que vamos a perseguir

toda la clase

¿Dónde vive el estado?

La respuesta es **distinta** en cada blockchain —
y de ahí sale **todo** lo demás.

SEGMENTO 3 · ~15 min

Consenso, sin perderse

PoW vs PoS, a alto nivel

Proof of Work

gastar cómputo (energía)
para ganar el derecho a proponer

Proof of Stake

poner capital en juego (stake)
si hacés trampa, lo perdés

Ambos resuelven lo mismo: **quién** propone el próximo bloque, de forma que hacer trampa sea caro.

Hoy ambas blockchains son Proof of Stake

Ethereum	Cardano
PoS desde el **Merge** (2022)	PoS desde el inicio (**Ouroboros**)

La diferencia entre las dos **no** está en el consenso.

Está en el **modelo de estado** (lo que viene ahora).

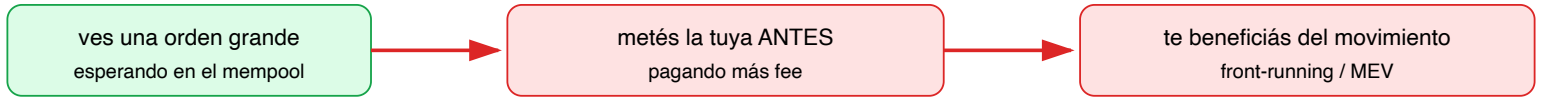
Cadencia y finalidad (para que los números no floten)

	Ethereum	Cardano
Un bloque cada…	12 segundos	~20 segundos (promedio)
Finalidad	**Explícita** por checkpoints: a los ~13 min el bloque queda finalizado	**Probabilística**: cada bloque encima hace exponencialmente más improbable revertir
En la práctica	Esperar la finalización	Esperar confirmaciones acordes al monto

> Lo que hay que retener: **"apareció en un bloque" ≠ "es final"**.
> Recién después de un rato una tx es irreversible.

El orden importa para la seguridad

El consenso decide **qué** tx entran y **en qué orden**. Eso tiene consecuencias:



Lo retomamos en la **Clase 6** (modelo de amenazas).

SEGMENTO 4 · ~50 min · EL NÚCLEO

Modelo de cuentas

vs EUTXO

La idea: dibujar el estado antes y después

La **misma** operación —cambiar quién tiene qué— se modela de forma **radicalmente distinta** en cada blockchain.

Vamos a ver, en cada modelo:

1. **Dónde** vive el estado
2. **Qué hace** una transacción con él
3. **Quién valida** y con qué información

> Estas tres preguntas ordenan **el resto del curso**: cada bug que veamos va a ser consecuencia de cómo las responde cada modelo.

Ethereum: dos tipos de cuenta

La distinción reaparece en **cada** clase de Ethereum:

	EOA (externally owned account)	Cuenta-contrato
Controlada por	Un par de claves (persona/wallet)	**Su propio código**
Tiene código	No	Sí (inmutable una vez deployado)
Tiene storage	No	Sí (estado mutable persistente)
Puede iniciar una tx	**Sí** — toda tx nace en una EOA	No (sólo reacciona a llamadas)
Tiene balance de ETH	Sí	Sí

> Un contrato es **una cuenta con código y storage**.

Ethereum: modelo de cuentas

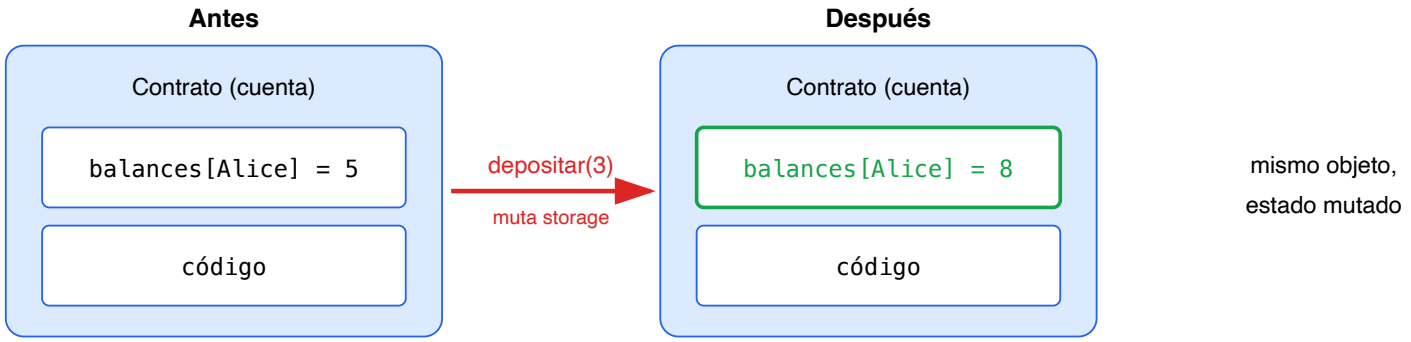
Cuentas con **balance**, y si son contratos, con **storage** (estado mutable persistente) y **código**.

```
contract Banco {
    mapping(address => uint256) balances;   // ← el estado, mutable

    function depositar() public payable {
        balances[msg.sender] += msg.value;  // muta el estado global
    }
}
```

Una tx **llama a una función** que **muta el estado global compartido**.

Ethereum: el estado antes y después



El contrato es **persistente**: vive en la blockchain y su storage cambia con cada tx.

Ethereum: consecuencias del modelo

Ejecución **secuencial** sobre **estado común y mutable**:

- El storage **persiste** entre transacciones
- Una función puede **llamar a otro contrato** en el medio de su ejecución
- Ese contrato puede **volver a llamar** (re-entrar) antes de que la primera termine
- El **orden** de las tx puede cambiar el resultado

> → De acá nacen **reentrancy**, condiciones de **carrera**, y dependencia del **orden** (MEV).

Cardano, paso 1 — UTXO "a secas" (el modelo de Bitcoin)

Construimos EUTXO en dos pasos, porque la "E" sólo se aprecia contra el modelo base.

- **No hay cuentas ni balances** como dato primario: hay **outputs no gastados** (UTXOs), cada uno con un valor y una **condición de gasto** (típicamente: "firma de tal clave")
- "Tu balance" es un concepto **derivado**: la suma de los UTXOs que podés gastar
- Una tx **consume inputs enteros** y **crea outputs nuevos**

> Vocabulario: "output" e "input" son **el mismo UTXO en dos momentos de su vida** — nace como output de una tx, flota como "no gastado", muere como input de otra.
> *(Se desarrolla a fondo en la Clase 4.)*

Cardano: consumir y crear

El estado vive en los **outputs no gastados** (UTXOs). No hay un "objeto contrato" que se mute: una tx **consume inputs** y **crea outputs**.

5 + 3 entran, 6 + 2 salen. Los UTXOs viejos **dejan de existir**; nacen otros.
No se muta: **se reemplaza**. *(En la práctica una parte se va en fee.)*

Cardano, paso 2 — la "E" de Extended

El UTXO de Bitcoin sólo exige condiciones muy simples.
Cardano lo **extiende en tres direcciones** — y eso habilita los smart contracts:

	UTXO (Bitcoin)	EUTXO (Cardano)
Condición de gasto	Script simple (casi siempre: una firma)	**Validator**: lógica arbitraria
Estado pegado al output	No	**Datum** (dato arbitrario adjunto)
Qué ve la condición al validar	Casi nada	**El contexto completo de la tx** (inputs, outputs, firmas, rango de validez)

> EUTXO = UTXO + **datum** + **validator arbitrario** + **visión de toda la tx**.

Cardano: el validator decide sí/no

Para gastar un output **protegido por un script**, un **validator** responde `True` / `False` a partir de tres entradas:

Ejemplo de condición: *"apruebo sólo si la tx está firmada por el dueño guardado en el datum"*.

Punto fino: el validator **no decide a dónde van los fondos** — sólo aprueba o veta que *ese* input se gaste en *esa* tx. Qué hace la tx con los fondos es responsabilidad de quien la armó *(se retoma en las Clases 8 y 9)*.

Cardano: validación local y determinista

El validator es una **función pura** que sólo mira **esa** transacción.

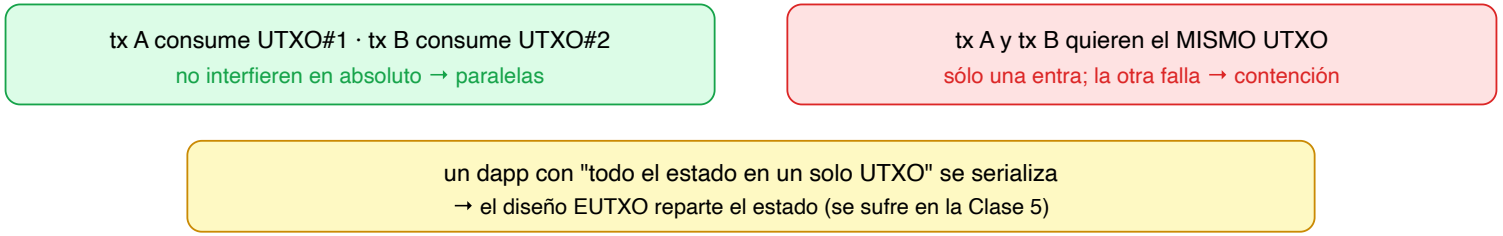
> Dada la misma tx, el validator **siempre devuelve lo mismo** → el veredicto y el costo se pueden calcular **antes** de enviarla.

Contraste con Ethereum: ahí la tx se ejecuta contra el **estado global del momento**, que pudo cambiar entre que la armaste y que se incluyó → puede revertir inesperadamente.

> ⚠️ **Determinismo no es garantía de que la tx entre.** Puede ser rechazada igual si **otro gastó tu UTXO primero** o si llega **fuera de su** `validity_range`. Lo predecible es el **veredicto del script**, no la aceptación de la transacción. *

Cardano: concurrencia y anti-replay

Concurrencia explícita:



Anti-replay estructural: los inputs consumidos ya no existen → la misma tx no puede aplicarse dos veces. **No hace falta nonce.**

Los dos modelos, lado a lado

	Ethereum (cuentas)	Cardano (EUTXO)
¿Dónde vive el estado?	Storage global, **mutable**	En los **UTXOs** (se consumen/crean)
Rol del contrato	Código que **muta** estado	Validator: **aprueba/rechaza** gastar
Entradas a la lógica	Llamada + estado actual	Datum + redeemer + contexto
Ejecución	Secuencial, compartida	**Local, determinista**
Conocer el resultado	Recién al ejecutar on-chain	**Antes** de enviar
Concurrencia	Orden global obligatorio	**Paralela** si no comparten UTXOs
Anti-replay	Nonce por cuenta	Consumo de inputs (**estructural**)
Lógica off-chain	Mínima	**Mucha**
Bugs típicos	Reentrancy, control de acceso	Double satisfaction, validación de outputs

Anticipo: dos catálogos de bugs

Columna vertebral de la segunda mitad del curso — **vuelve completa en las Clases 6 y 8**.
Hoy alcanza con que suenen los nombres; no hay que entender cada fila todavía:

Ethereum (cuentas)	Cardano (EUTXO)
Reentrancy	Double satisfaction
Control de acceso	Validación de outputs insuficiente (destino/monto)
Overflow en `unchecked` / casts	Datum no autorizado / no validado
`delegatecall` / proxies	Dust / min-ADA
Front-running / MEV	Seguridad de minting policies
Oráculos, DoS por gas	Asunciones sobre el contexto de la tx

Los bugs no se traducen entre modelos

Reentrancy

casi un no-concepto en EUTXO

(no hay estado mutable que re-entrar)

Double satisfaction

ni siquiera existe en Ethereum

(nace de componer varios inputs)

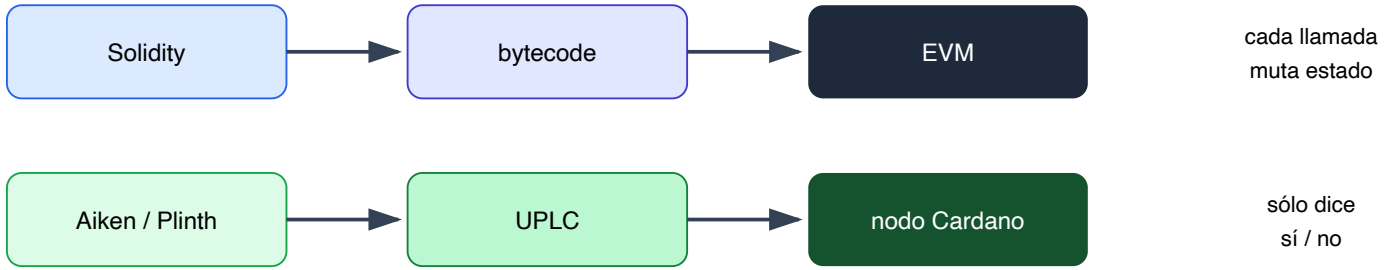
Distinta arquitectura → distinto catálogo de bugs.

(Se profundiza en Clases 6–7 para Ethereum y 8–9 para Cardano.)

SEGMENTO 5 · ~20 min

¿Dónde corre el código?

Dos pipelines de compilación



UPLC = Untyped Plutus Core, el "assembly" que ejecutan los nodos de Cardano. Aiken y Plinth (ex-PlutusTx, Haskell) compilan **al mismo objetivo** — cambia la ergonomía, no el destino *(Clase 4)*.

Qué significa "deployar"

	Ethereum	Cardano
El código…	Se sube a la blockchain y **vive** en una cuenta-contrato con su dirección	Se **referencia por hash**: la dirección del script se deriva del hash del validator
Las tx…	Le "hablan" a esa dirección	Mandan fondos a la dirección; el código se aporta **al gastar** (en la tx, o publicado antes como *reference script*)

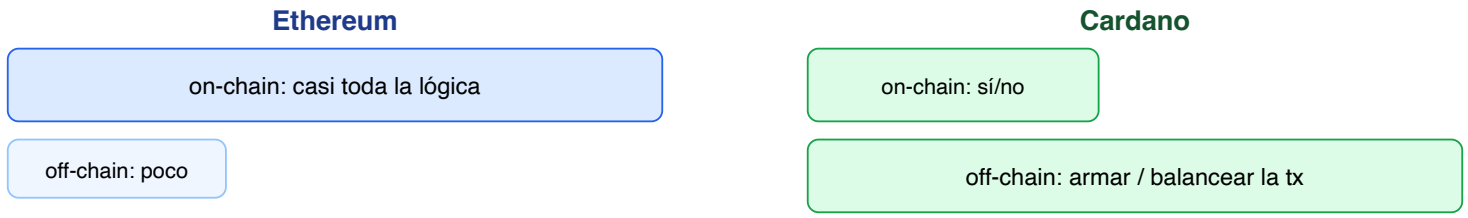
> Otra cara de la misma idea: **no hay objeto contrato — hay outputs protegidos.**

Cómo se cobra el cómputo

Consecuencia directa del modelo (cierra el círculo del Segmento 2):

	Ethereum (gas)	Cardano (ExUnits)
Unidad	Gas por opcode ejecutado	Memoria + pasos de CPU
¿Cuándo se conoce el costo?	Recién al ejecutar (depende del estado global)	**Antes de enviar** (ejecución determinista)
¿Puede fallar on-chain y cobrar igual?	**Sí**: una tx que revierte paga el gas consumido	El validator se corre localmente primero; una wallet honesta no envía tx que fallan

La asimetría on-chain / off-chain



En Cardano, decidir **qué inputs juntar**, **cómo balancear** y **armar datum/redeemer** vive **fuera** de la blockchain. El validator on-chain sólo dice **sí o no**.

SEGMENTO 6 · ~25 min

Actividad

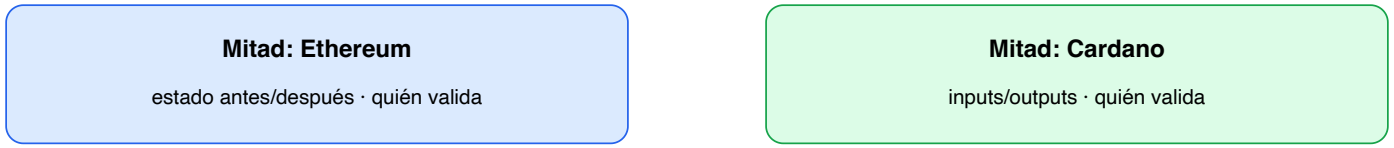
"Seguí la transacción"

Actividad — en parejas

La **misma** operación, modelada en los dos mundos:

Alice le transfiere 10 unidades de un token a Bob

(el token ya existe; no importa cómo se creó)



Cada pareja entrega, en papel o pizarra:

1. **El estado ANTES** — cuentas/storage (Ethereum) · UTXOs existentes (Cardano)
2. **La transacción** — ¿a quién llama? / ¿qué consume y qué crea? · quién firma
3. **El estado DESPUÉS**
4. **¿Quién valida y con qué información?** ← la pregunta central

Después se **cruzan y comparan**.

> Al terminar tendrían que poder explicar la diferencia **sin decir la palabra**

Lo que tiene que salir de la actividad

- En **Ethereum**: una llamada a una función muta un `mapping` de balances;
  valida **el código del contrato** contra el estado actual.
- En **Cardano**: la tx consume el UTXO con el token y crea uno nuevo para el destinatario;
  valida **el validator** con datum + redeemer + contexto.

> Misma intención, **dos máquinas distintas** — y por eso, dos formas de fallar.

Errores esperables (dejarlos aparecer y usarlos)

Lado Cardano — olvidar el cambio:

Si Alice tiene un UTXO con **25** tokens y manda **10**, los otros **15** tienen que salir en un output de vuelta a ella — los inputs se consumen **enteros**.

Lado Ethereum — dibujar "monedas que viajan":

En cuentas **no viaja nada**: se editan dos números de un `mapping`.

```
balances[Alice]: 25 → 15
balances[Bob]:    0 → 10
```

Lo que se ve más adelante

Hoy apareció…	Vuelve en…
El contrato `Banco` y su `mapping`	**Clase 2** — primer contrato leído en serio; gas por operación
El diagrama EUTXO y **datum / redeemer / contexto**	**Clase 4** — se retoman tal cual
La tabla de los **dos catálogos de bugs**	**Clases 6 y 8** — índice del bloque de análisis
Reentrancy ↔️ double satisfaction	**Clases 7 y 9** — se demuestran con código
Front-running / MEV · oráculos	**Clase 6** — modelo de amenazas

Lecturas

- *Mastering Ethereum*, **2ª ed. (2025)** — **caps. 1–2** (cuentas) y **14** (EVM) *(actualizada post-Merge; gratis online y en* `ethereumbook` *)*
- *Mastering Cardano* — **cap. 4** (*How Cardano works*): EUTXO
- Cardano Developer Portal — página de EUTXO
- Opcional para curiosos: el paper *The Extended UTXO Model* (Chakravarty et al.) — formaliza la "E"; con la introducción alcanza

Próxima clase

Clase 2 — Intro a Solidity

Leer y escribir un contrato; tipos y data location, visibilidad, errores, eventos, y el ciclo deploy → call (en Remix).