Clase 5

Taller Aiken

Escrow con deadline y tests

Tesis de la clase

> La misma operación —un depósito condicional— requiere un enfoque **radicalmente distinto** en EUTXO que en cuentas.

Comparar este taller con el de la Clase 3 hace **visible** la consecuencia de los modelos.

Objetivos de aprendizaje

Al terminar, vas a poder:

- **Instalar Aiken**, verificar el entorno y correr el proyecto del repo
- **Construir `escrow.ak` paso a paso**, entendiendo cada decisión
- Explicar por qué en EUTXO **no existe `block.timestamp`** y cómo `validity_range` lo reemplaza
- Escribir tests unitarios con `transaction.placeholder`
- **Comparar** el escrow con el Vault de la Clase 3

SEGMENTO 1 · ~20 min

Setup y punto de partida

Instalar Aiken

En la **imagen del curso ya está** — verificá con `aiken --version` y saltá a la slide siguiente.

Fuera de la imagen:

```
# aikup: el instalador de Aiken (similar a foundryup)
curl -sSfL https://install.aiken-lang.org | bash
aikup install

aiken --version   # aiken v1.1.21+... (la del repo del curso)
```

Un proyecto nuevo se arma con `aiken new <nombre>`, y queda así:

```
aiken.toml        → configuración (nombre, versión, dependencias)
lib/              → módulos reutilizables
validators/       → los validators on-chain
```

El `aiken.toml` del curso fija `compiler = "v1.1.21"` y `plutus = "v3"`.

Verificar el entorno

```
cd cardano/
aiken check    # Summary 12 checks, 5 errors, 2 warnings
```

Los 5 rojos son correctos. Hay dos validators: `escrow.ak`, terminado, con sus **7 tests en verde**; y `vesting.ak`, **sin implementar**, con 5 tests que arrancan en rojo.

Hoy repasamos el escrow —incluida la parte de tiempo, que es nueva— y después el vesting lo escriben ustedes. Si algo falla en el setup: `cardano/README.md`.

> Pregunta para tener en la cabeza toda la clase:
> *¿qué tiene este validator que todavía le falta?*

SEGMENTO 2 · ~35 min

Construcción guiada

`escrow.ak`

2a. Imports

```
use aiken/collection/list
use aiken/interval
use cardano/transaction.{OutputReference, Transaction, placeholder}
```

- `list` → buscar en listas
- `interval` → manejar rangos de tiempo
- `transaction` → tipos centrales y el `placeholder` para tests

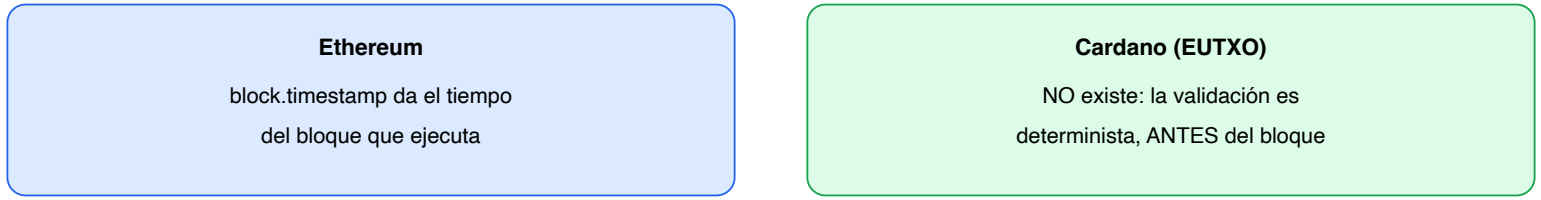
2b. Datum y Redeemer

```
pub type Datum {
  beneficiary: ByteArray,   // hash de clave — quién reclama
  owner:       ByteArray,   // hash de clave — quién cancela
  deadline:    Int,         // POSIX time en ms: límite para reclamar
}

pub type Redeemer {
  Claim    // el beneficiario reclama
  Cancel   // el owner cancela y recupera
}
```

Comparación con el Vault: el Vault guarda balances en un `mapping`; el escrow guarda **todo en el `Datum`**, que viaja pegado al UTXO. No hay storage mutable.

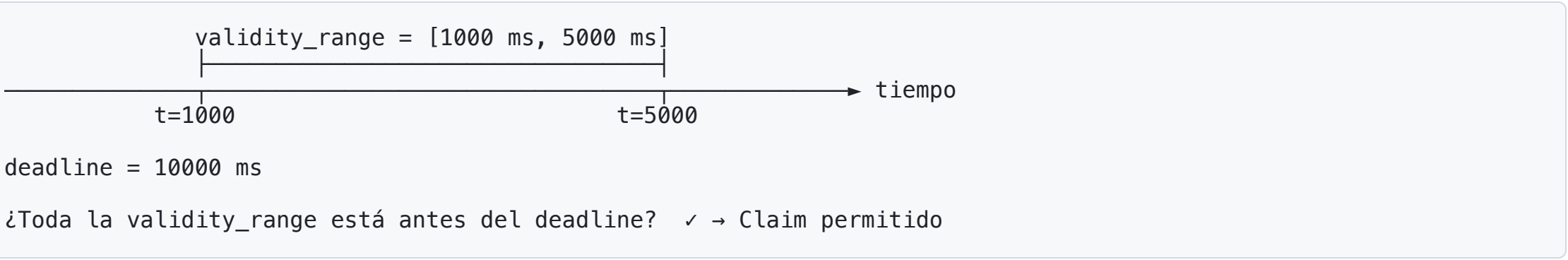
2c. ¿Por qué no hay `block.timestamp`?



La tx lleva un `validity_range`: el intervalo de tiempo en que es válida.
El validator razona sobre ese intervalo para inferir cuándo ocurre.

> Ojo: que la validación sea determinista **no** quiere decir que una tx no pueda fallar al enviarla. *

2c. validity_range vs deadline




```
interval.before(d.deadline) |> interval.includes(tx.validity_range)
```

"El intervalo (-∞, deadline] **incluye** la validity_range" → la tx ocurre antes del deadline.

¿Por qué el intervalo entero y no "comparar con ahora"?

En el validator **no hay un "ahora"**. Lo único que hay es el `validity_range`, y lo declara **quien arma la transacción**.

Con `deadline = 10:00`, y una tx que declara el rango `[09:00, +∞)`:

	chequeo ingenuo	`includes` (el rango entero)
qué mira	sólo el borde de **abajo**	**los dos** bordes
decide	`09:00 < 10:00` → **aprueba**	el rango no termina antes de las 10:00 → **rechaza**
resultado	la tx puede minarse a las **11:00**	obliga a declarar `[09:00, 09:59]`

> El agujero no es el borde de abajo: es **el de arriba, que quedó abierto**. Con el rango sin tope, la transacción sigue siendo válida después del deadline.

¿Y quién garantiza que ese rango se cumpla?

El **ledger**, y **antes** de correr el script. Son dos etapas distintas:

	quién	qué hace
Fase 1	el ledger	comprueba que el slot del bloque caiga dentro del `validity_range`. Si no cae, la tx es **inválida** — el validator ni se ejecuta
Fase 2	el validator	corre, y razona sobre el rango **declarado**

> Por eso el validator puede razonar sobre un rango sin saber "cuándo es ahora": cuando le toca correr, el ledger **ya garantizó** que la transacción está adentro de ese rango.

Declarar un rango angosto no es una formalidad: es lo que vuelve **exigible** el deadline.

Detalle de bordes y unidades

- `interval.before(x)` construye `(−∞, x]` → el borde superior es **inclusivo** ("en o antes")
- Para *estrictamente* antes existe `interval.entirely_before`
- `deadline` y los bordes del rango están en **POSIX time en milisegundos** (no segundos)
- La tx se declara en **slots**, pero el script ve **POSIX**: por eso existe el `slotFrom(ms)` del off-chain *

2d. Funciones auxiliares (testeable aislado)

```
fn can_claim(d: Datum, tx: Transaction) -> Bool {
  let signed_by_beneficiary = list.has(tx.extra_signatories, d.beneficiary)
  let before_deadline =
    interval.before(d.deadline) |> interval.includes(tx.validity_range)
  signed_by_beneficiary && before_deadline
}

fn can_cancel(d: Datum, tx: Transaction) -> Bool {
  list.has(tx.extra_signatories, d.owner)
}
```

`can_claim` exige firma **Y** tiempo. `can_cancel` sólo la firma del owner (puede cancelar antes o después del deadline).

¿Por qué separar en funciones puras?

> Separá la **decisión** (pura, testeable) del **cableado** (el handler del validator).

- `can_claim(datum, tx) -> Bool` se prueba **aislada**: armás una `Transaction` de mentira
- Testear el `validator` entero obliga a construir el `ScriptContext` completo (propósito, redeemer serializado, ref al output…) → mucho más verboso
- El `spend` queda como un simple `when redeemer is` que **delega**

> Decisión de diseño abierta: `can_cancel` **no** mira el tiempo — el owner recupera cuando quiera. En otro producto podrías exigir que cancele *después* del deadline. ¿Qué cambiarías?

2e. El validator

```
validator escrow {
  spend(datum: Option<Datum>, redeemer: Redeemer,
        _utxo: OutputReference, self: Transaction) {
    expect Some(d) = datum
    when redeemer is {
      Claim  -> can_claim(d, self)
      Cancel -> can_cancel(d, self)
    }
  }
  else(_) { fail }
}
```

`expect Some(d)` → falla si no hay datum. `else(_) { fail }` → rechaza otros propósitos.

Vault vs Escrow, lado a lado

	Vault (Ethereum)	Escrow (Cardano)
¿Qué protege?	mapping address→balance	un UTXO con datum
Condición de acceso	`msg.sender == owner`	firma en `extra_signatories`
Tiempo	`block.timestamp`	`validity_range` (inferido)
Patrón clave	CEI	verificar el contexto completo
Bug asociado	reentrancy	double satisfaction

SEGMENTO 3 · ~25 min

Tests en Aiken

`transaction.placeholder`

La clave para tests legibles: una transacción **vacía** con todos los campos en su valor neutro. Se sobreescriben **solo** los que importan.

```
let tx = Transaction {
  ..placeholder,                          // todo en valor neutro
  extra_signatories: [beneficiary_hash],  // solo esto importa
  validity_range:    interval.between(1_000, 5_000),
}
```

`..placeholder` es el operador de actualización de registro (como Rust).

Constantes de test

```
const beneficiary_hash: ByteArray = #"aabbccdd"
const owner_hash:       ByteArray = #"11223344"
const deadline_ms:      Int       = 10_000

fn make_datum() -> Datum {
  Datum { beneficiary: beneficiary_hash, owner: owner_hash, deadline: deadline_ms }
}
```

Constantes descriptivas + helper → tests legibles, sin repetir construcción.

Tests de `Claim`

```
test claim_con_firma_y_antes_del_deadline() {
  let d = make_datum()
  let tx = Transaction { ..placeholder,
    extra_signatories: [beneficiary_hash],
    validity_range: interval.between(1_000, 5_000) }  // ⊆ (-∞, 10000]
  can_claim(d, tx)
}

test claim_despues_del_deadline_falla() {
  let d = make_datum()
  let tx = Transaction { ..placeholder,
    extra_signatories: [beneficiary_hash],
    validity_range: interval.between(10_001, 15_000) }  // fuera del deadline
  !can_claim(d, tx)
}
```

También: `claim_sin_firma_falla`, `claim_con_firma_equivocada_falla`.

Tests de `Cancel`

```
test cancel_con_firma_del_owner() {
  let d = make_datum()
  let tx = Transaction { ..placeholder,
    extra_signatories: [owner_hash],
    validity_range: interval.everything }   // sin restricción de tiempo
  can_cancel(d, tx)
}

test cancel_despues_del_deadline_pasa() {
  let d = make_datum()
  let tx = Transaction { ..placeholder,
    extra_signatories: [owner_hash],
    validity_range: interval.after(50_000) }  // muy después
  can_cancel(d, tx)
}
```

Correr los tests

```
aiken check -m "escrow.{..}"
```

```
┍━ escrow ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
│ PASS [mem:  97.50 K, cpu:  34.15 M] claim_con_firma_y_antes_del_deadline
│ PASS [mem:  29.16 K, cpu:   8.83 M] claim_sin_firma_falla
│ PASS [mem:  97.00 K, cpu:  34.02 M] claim_despues_del_deadline_falla
│ PASS [mem:  32.29 K, cpu:   9.73 M] claim_con_firma_equivocada_falla
│ PASS [mem:  20.73 K, cpu:   6.83 M] cancel_con_firma_del_owner
│ PASS [mem:  22.80 K, cpu:   7.44 M] cancel_con_firma_equivocada_falla
│ PASS [mem:  24.59 K, cpu:   7.91 M] cancel_despues_del_deadline_pasa
┕━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7 tests | 7 passed | 0 failed
```

Ese `[mem, cpu]` son las ExUnits de la Clase 4

```
PASS [mem:  97.50 K, cpu:  34.15 M] claim_con_firma_y_antes_del_deadline
PASS [mem:  29.16 K, cpu:   8.83 M] claim_sin_firma_falla     ← un tercio
```

Cuando **no hay firma**, `list.has` devuelve `False`, el `&&` **corta**,
y el chequeo del intervalo —la parte cara— nunca se evalúa.

- El cortocircuito no es un detalle de estilo: **se paga en ExUnits reales**
- Aiken te dice el costo **test por test**, sin desplegar nada
- En Ethereum el costo se descubre **ejecutando**; acá se conoce **antes** de enviar

```
aiken check -m "escrow.{claim_sin_firma_falla}"   # correr uno solo
```

Tests property-based (anticipo)

Con el paquete `aiken-lang/fuzz`:

```
use aiken/fuzz

test prop_claim_pasa_con_firma_y_tiempo(deadline via fuzz.int()) {
  let d = Datum { beneficiary: beneficiary_hash, owner: owner_hash, deadline }
  let tx = Transaction { ..placeholder,
    extra_signatories: [beneficiary_hash],
    validity_range: interval.before(deadline) }   // siempre dentro
  can_claim(d, tx)
}
```

`via fuzz.int()` → `deadline` toma valores aleatorios (100 corridas).
El análogo en Foundry se escribe `testFuzz_`.

> El paquete **no viene en el repo**: hay que agregarlo con `aiken packages add aiken-lang/fuzz --version main`.

SEGMENTO 4 · ~30 min

Taller

`vesting.ak`

Los fondos se liberan después del deadline

El espejo del escrow: aquel exige que la tx ocurra **antes**; éste, **después**.

```
fn can_unlock(d: Datum, tx: Transaction) -> Bool {
  todo @"implementar can_unlock"
}
```

En `cardano/validators/vesting.ak` ya están el `Datum`, el `validator` y **3 tests que son la especificación**. Arrancan en rojo.

1. Implementar `can_unlock`: firma del beneficiario **y** tx después del deadline.
2. Escribir las **2 consignas** que quedan (las que tienen `False` como cuerpo).

```
al empezar:  12 checks, 5 errors        al terminar:  12 checks, 0 errors
```

> El escrow ya resolvió el problema simétrico. Miralo antes de escribir — pero ojo, no alcanza con copiarlo.

SEGMENTO 5 · ~20 min

Off-chain: construir la tx

El flujo completo de un escrow

```
FASE 1 — LOCK (depositar)
  output → dirección del validator
         → datum { beneficiary, owner, deadline }
         → N ADA + min-ADA

FASE 2 — CLAIM (beneficiario, antes del deadline)
  1. encontrar el UTXO en la dirección del validator
  2. validity_range = [ahora, deadline - margen]
  3. redeemer = Claim
  4. required_signer = beneficiary_hash
  5. firmar con la clave del beneficiario → enviar

FASE 3 — CANCEL (owner, cualquier momento)
  igual, con redeemer = Cancel y firmante = owner
```

Armar el `validity_range`: upper bound ≤ deadline

> El `validity_range` de la tx de Claim debe tener el upper bound **antes o igual** al deadline del datum (si no, el validator falla), y ser **suficientemente amplio** para que la tx llegue a un bloque antes de expirar.

```
tx.spendingPlutusScript("V3")
  .txIn(utxo.txHash, utxo.index)
  .txInScript(validatorCbor)
  .txInInlineDatumPresent()
  .txInRedeemerValue(mConStr0([]))          // Claim
  .requiredSignerHash(beneficiaryPubKeyHash)
  .txInCollateral(col.txHash, col.index)
  .invalidHereafter(slotDelDeadline)        // ← el TOPE del rango
  .changeAddress(beneficiaryAddress);
```

Los dos métodos del rango son `invalidHereafter` (tope) e `invalidBefore` (piso).
El escrow necesita **el tope**, porque exige que la tx ocurra antes del deadline.

> El rango no se puede poner arbitrariamente lejos en el futuro: hay un límite de cuán adelante se puede convertir tiempo a slots. *

SEGMENTO 6 · ~10 min

Blueprint y deploy

Generar el blueprint

```
aiken build     # genera plutus.json
```

`plutus.json` guarda el **código compilado** del validator y su **hash**. Del hash sale la **dirección**; el código viaja **dentro de la transacción**. Es el equivalente del `out/` de Foundry.

Y ahora se manda de verdad, contra un devnet local — **sin faucet y sin esperar**:

```
./scripts/devnet.sh up                # terminal 1
cd cardano/offchain && npm run demo-simple    # terminal 2
```

`demo-simple.ts` bloquea 5 ADA en el escrow y las reclama. Son **40 líneas** y hace exactamente lo de la slide anterior, con hashes de transacción reales.

> Para mandarlo a una testnet pública (Preprod/Preview) hace falta un faucet y un explorador. No cambia nada del código: cambia el provider.

Llevarlo a una red real (Preprod)

El validator no cambia. El hash es el mismo en todas las redes; la dirección sólo cambia de prefijo:

```
hash      72869004128e27a34b734017de9bb37f794b720a41cd47f79d6a44af
preprod   addr_test1wpegdyqyz28z0g6twdqp0h5mkdlhjjmjpfqu63lhn44yftckavq9j
mainnet   addr1w9egdyqyz28z0g6twdqp0h5mkdlhjjmjpfqu63lhn44yftcd4cu2h
                 ↑ el cuerpo es idéntico
```

Lo que cambia es **el off-chain**, y son cuatro cosas:

	devnet	Preprod
provider	`YaciProvider`	`BlockfrostProvider` (project id gratis)
fondos	20 cuentas precargadas	**faucet** de testnet
claves	mnemonic público del devnet	uno propio — **nunca el del curso**
esperas	bloque cada ~1 s	~20 s, y la finalidad tarda mucho más

> Mainnet es el mismo cambio con `networkId: 1`. Ahí el ADA vale plata.

Ejercicio: el off-chain del vesting

Ya escribieron el validator. Falta la transacción que lo invoca.

```
npm run demo-vesting     # tiene dos TODO en el UNLOCK
```

`demo-vesting.ts` es el espejo de `demo-simple.ts`: el LOCK ya está resuelto, y en el UNLOCK faltan dos líneas.

1. Declarar el **firmante requerido**. Sin eso `extra_signatories` llega vacío.
2. Declarar el **rango de validez**. `demo-simple` declara uno; ¿cuál de los dos métodos corresponde acá, y por qué?

> Es la misma decisión que ya tomaron on-chain, ahora del lado off-chain. Si el validator está bien y la transacción mal, **igual falla** — las dos mitades tienen que estar de acuerdo.

SEGMENTO 7 · ~10 min

Cierre

y qué se ve más adelante

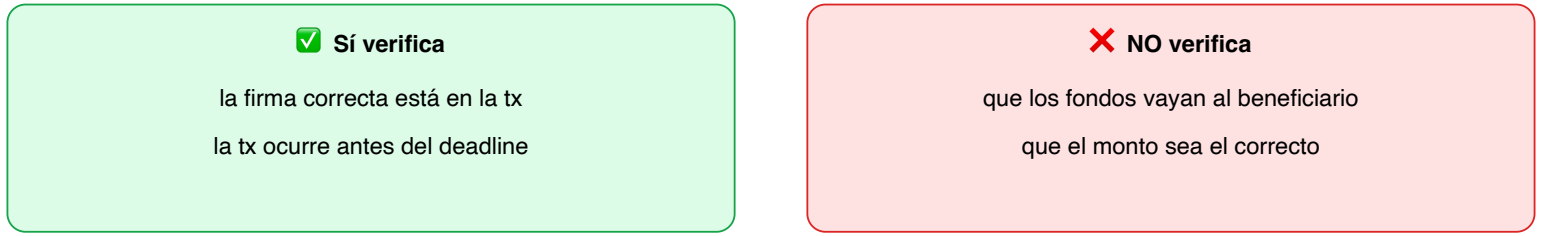
Actividad

En pares, sobre el escrow:

1. **Análisis:** ¿qué habría que cambiar en `can_claim` para verificar que un output va a la dirección del beneficiario? ¿Qué información habría que agregar al `Datum`?
2. Y una pregunta sobre el vesting que acaban de escribir: **¿hacía falta Plutus?** Repasen el criterio de los native scripts.

> Dado que `self.outputs` está disponible, ¿qué debería verificar `can_claim` para ser seguro?

Lo que el escrow verifica… y lo que no



En la **Clase 9** explotamos ese hueco: alguien arma una tx que cumple firma y tiempo pero **redirige los fondos**. Nombre: **double satisfaction**.

Lecturas

- **Aiken**: `aiken-lang.org` — "Language Tour" y "Testing"
- Aiken stdlib — módulo `aiken/interval` (leer los ejemplos del docstring)
- **Mesh.js**: `meshjs.dev` — sección "Plutus scripts"
- *Mastering Cardano* — **cap. 8** (smart contracts) · **cap. 4** (EUTXO)
- **Cardano CTF** (Vacuumlabs): `github.com/vacuumlabs/cardano-ctf` — ~25 retos en Aiken. La práctica ideal para seguir por tu cuenta

Próxima clase

Clase 6 — Intro análisis Solidity

Volvemos a Ethereum: modelo de amenazas EVM, Slither, fuzzing e invariantes con Foundry.