Clase 4

Intro Aiken/Plinth

Validators en EUTXO

Tesis de la clase

> En EUTXO el validator no "corre un contrato": **solo dice sí o no**.
> Toda la complejidad de construir la transacción vive **off-chain**.

Entender esta asimetría es la clave para **leer** —y eventualmente **romper**— cualquier validator.

Objetivos de aprendizaje

Al terminar, vas a poder:

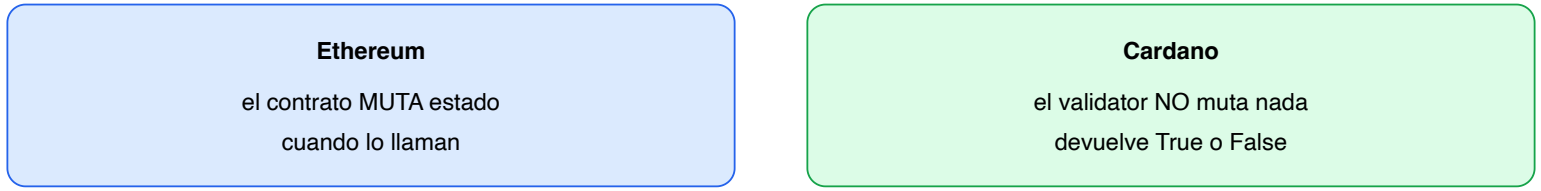
- Recordar **qué rol** cumple un validator en EUTXO
- Leer código Aiken básico (tipos, pattern matching, funciones)
- **Explicar datum, redeemer y contexto** y qué consulta cada uno
- Decidir si un caso necesita un **native script** o un validator Plutus
- Explicar qué cambia al **parametrizar** un validator
- Distinguir **tokens nativos** (Cardano) de contratos ERC-20 (Ethereum)
- Identificar qué va **on-chain** vs **off-chain** y por qué

SEGMENTO 1 · ~15 min

Recap EUTXO

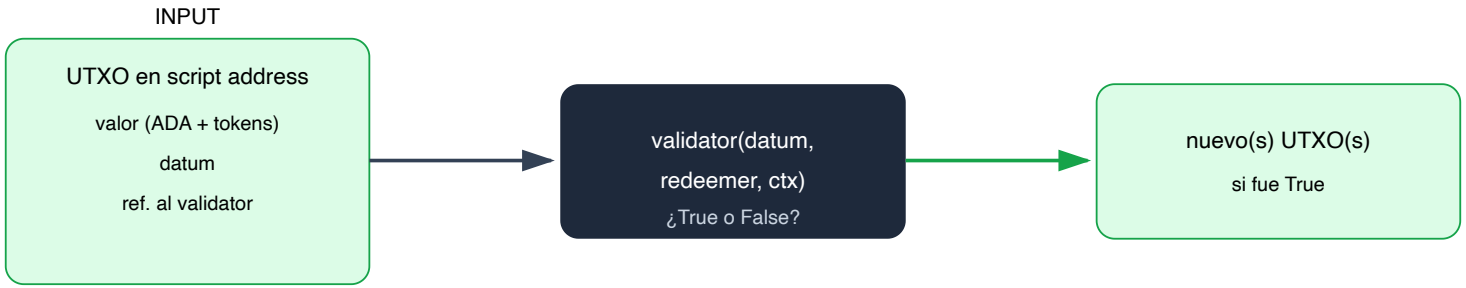
+ rol del validator

El validator no muta estado: devuelve `True` o `False`



Si el validator devuelve `False`, **la tx entera falla** y el UTXO queda intacto.
Si devuelve `True`, el output se consume y se crean los nuevos.

El validator en el flujo de gasto

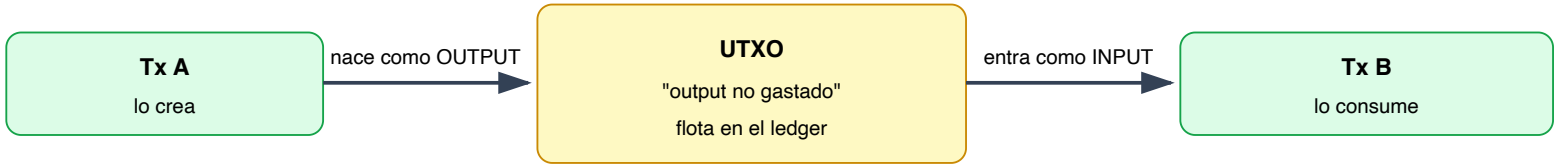


El validator **no elige a dónde van los fondos**. Sólo dice si está bien gastar ese input en este contexto.

Output o input: el mismo UTXO

> **No son dos cosas distintas: es el mismo UTXO en dos momentos de su vida.**

Un UTXO = **U**nspent **T**ransaction **O**utput = un *output que todavía no se gastó*.



Nace como **output** de una tx (el `LOCK`) → flota → otra tx lo toma como **input** y lo consume (el `UNLOCK`).

¿Por qué "consumir el output" y no "el input"?

Las dos frases describen **la misma** operación, vistas desde dos perspectivas:

Frase	Perspectiva	Qué enfatiza
"consumir el output"	desde el pasado / el ledger	agarrás algo que **ya existe** como output pendiente
"gastar el input"	desde la tx que armás ahora	el rol que cumple ese UTXO **dentro de tu tx**

> En EUTXO los UTXOs **no se modifican: se reemplazan**. Una tx **consume** inputs (UTXOs que existían) y **crea** outputs (UTXOs nuevos). El validator `spend` corre justo en esa transición —una vez por cada input de script— para autorizar que un output preexistente se convierta en input consumido.

SEGMENTO 2 · ~25 min

El stack

Aiken, Plinth, off-chain

On-chain: dos lenguajes, un objetivo

Opción	Lenguaje base	Compila a	Quién lo mantiene
Aiken	propio (Rust/Elm-flavored)	UPLC	comunidad (aiken-lang)
Plinth (ex-PlutusTx)	Haskell	UPLC	IOG / IntersectMBO

Ambos compilan a **UPLC** (Untyped Plutus Core), lo que realmente ejecutan los nodos. La versión del lenguaje es **Plutus V3**, vigente desde el hard fork de **Chang (2024)**.

¿Por qué Aiken en el curso? Sintaxis accesible (sin Haskell), errores claros, tests integrados, documentación activa. Plinth existe y es relevante en producción, pero la curva de entrada es mayor.

Antes de Plutus: los *native scripts*

Cardano tiene un **tercer camino**, anterior a Plutus: reglas **declarativas** en JSON.
No es un lenguaje de programación — no hay bucles, no hay funciones, no es Turing-completo.

Regla	Qué exige
`sig`	que **esta clave** firme la transacción
`all`	que se cumplan **todas** las reglas de adentro
`any`	que se cumpla **al menos una**
`atLeast`	que se cumplan **al menos N**
`after` / `before`	que la tx se ejecute **después / antes** de un slot

Con esas seis piezas se define quién puede **gastar un UTXO** o **acuñar un token**.

Dos casos que no necesitan un contrato

Multisig: 3 de 5 tienen que firmar

```
{ "type": "atLeast", "required": 3,
  "scripts": [
    {"type":"sig","keyHash":"78489d…3ae09"},
    {"type":"sig","keyHash":"d3fde9…2f2dd0"},
    {"type":"sig","keyHash":"c3a0bd…5d3e45"},
    …
  ]
}
```

Alcanza con **tres firmas cualesquiera** de la lista.

Vesting: Bob, pero recién a partir del slot N

```
{ "type": "all",
  "scripts": [
    {"type":"sig","keyHash":"d3fde9…2f2dd0"},
    {"type":"after","slot": 52593306}
  ]
}
```

Ese `52593306` **es** el slot N: la tx sólo vale **de ahí en adelante**.

En Ethereum, cada uno de estos dos casos es **un contrato** que hay que escribir, testear y auditar.

¿Cuándo alcanza un native script?

> **Alcanza** si la regla se expresa sólo con **firmas y tiempo**.
> **No alcanza** en cuanto la regla mira **a dónde va el dinero**, **cuánto**, o **un datum**.

	Native script	Validator Plutus
Expresividad	firmas + ventana temporal	cualquier condición sobre la tx
Costo de ejecución	**no consume ExUnits**	consume ExUnits
Superficie de bugs	6 constructores, sin lógica propia	todo el código que escribas

El criterio de ingeniería: el código que no escribís es código que no puede fallar.
Antes de escribir un validator, preguntate si tu regla cabe en estas seis piezas. *

Costo de ejecución: ExUnits, no gas

El UPLC de un validator consume un presupuesto acotado de **ExUnits**:

Dimensión	Análogo
memoria	—
pasos de CPU	~ gas de Ethereum

- Si la ejecución se pasa del presupuesto → la tx es **inválida**
- Diferencia con Ethereum: el costo se **conoce de antemano** (no hay estado global mutable), se calcula off-chain al armar la tx

> Empuja a validators **baratos y simples** → otra razón por la que la lógica pesada vive off-chain.

Off-chain: armar la transacción

Biblioteca	Lenguaje
Mesh.js	TypeScript/JS (usada en las demos)
Lucid Evolution	TypeScript
Blaze	TypeScript
PyCardano	Python

> En Ethereum el off-chain es mínimo. En Cardano, **construir una tx válida** es ingeniería: seleccionar inputs, balancear, calcular min-ADA, adjuntar datum/redeemer, agregar firmantes. Consecuencia directa del modelo EUTXO.

SEGMENTO 3 · ~30 min

Aiken

sintaxis y proyecto

Tipos básicos

```
let n: Int = 42
let flag: Bool = True
let hash: ByteArray = #"deadbeef"   // bytes en hex

type Option<a> { Some(a)  None }        // evita el null
type Result<a, e> { Ok(a)  Err(e) }     // éxito o error
```

> `#"deadbeef"` = bytes en hex, cantidad **par** de dígitos (o no parsea).

`Int` es bignum, no `uint256`

Solidity — uint256

ancho fijo (256 bits), tiene techo

overflow / underflow = preocupación

0.8 los chequea, pero la clase existe

Aiken — Int

precisión arbitraria (bignum)

no hay techo, no hay wraparound

la clase de bug NO existe

> El riesgo no desaparece, **se muda**: en Cardano el problema no es que un número desborde, sino **validar bien los montos**: que el output tenga la plata correcta.

Tipos personalizados: structs y enums

```
// Tipo suma: variantes alternativas (como enum/union)
type Accion {
  Reclamar
  Cancelar
  Extender { nuevo_deadline: Int }   // variante con dato
}

// Tipo producto: campos combinados (como struct)
type Participantes {
  comprador: ByteArray,   // hash de clave pública
  vendedor:  ByteArray,
}
```

Pattern matching: el mecanismo central

```
fn describir(a: Accion) -> ByteArray {
  when a is {
    Reclamar -> "el beneficiario reclama"
    Cancelar -> "el dueño cancela"
    Extender { nuevo_deadline } -> "extender el plazo"
  }
}
```

> `when ... is` es **exhaustivo**: el compilador **falla** si falta una variante.
> Elimina el bug del `default` olvidado de los lenguajes imperativos.

Funciones, pipe, y nada de mutación

```
fn suma(a: Int, b: Int) -> Int {
  a + b   // la última expresión es el retorno (no hay `return`)
}

fn triple(n: Int) -> Int {
  n |> suma(n) |> suma(n)   // pipe: pasa el resultado como primer arg
}
```

- **No** hay mutación
- **No** hay loops imperativos (`for` , `while`)
- Iteración con recursión o `aiken/collection/list` (`map` , `filter` , `has` , …)

Combinar condiciones: `and { }` y el `?`

Un validator suele ser una **conjunción** de condiciones. Se lee de arriba abajo:

```
and {
  list.has(tx.extra_signatories, d.owner)?,   // ¿firmó el owner?
  cantidad_correcta?,                          // ¿el monto es el esperado?
  antes_del_deadline?,                         // ¿está en tiempo?
}
```

- `and { ... }` es `True` solo si **todas** lo son (y existe `or { ... }`)
- El `?` es el *operador de traza*: si esa condición da `False`, imprime su nombre en el log de `aiken check` → te dice **cuál** condición falló

> El `?` es la herramienta de debugging del validator. También está `trace "mensaje"`.

Correr el proyecto

```
aiken check    # compila + corre los tests (funciones test_)
aiken build    # genera plutus.json
```

`plutus.json` es el **blueprint**: por cada validator guarda su **código compilado** —una tira larga de hex, el programa que ejecutan los nodos— y su **hash**.

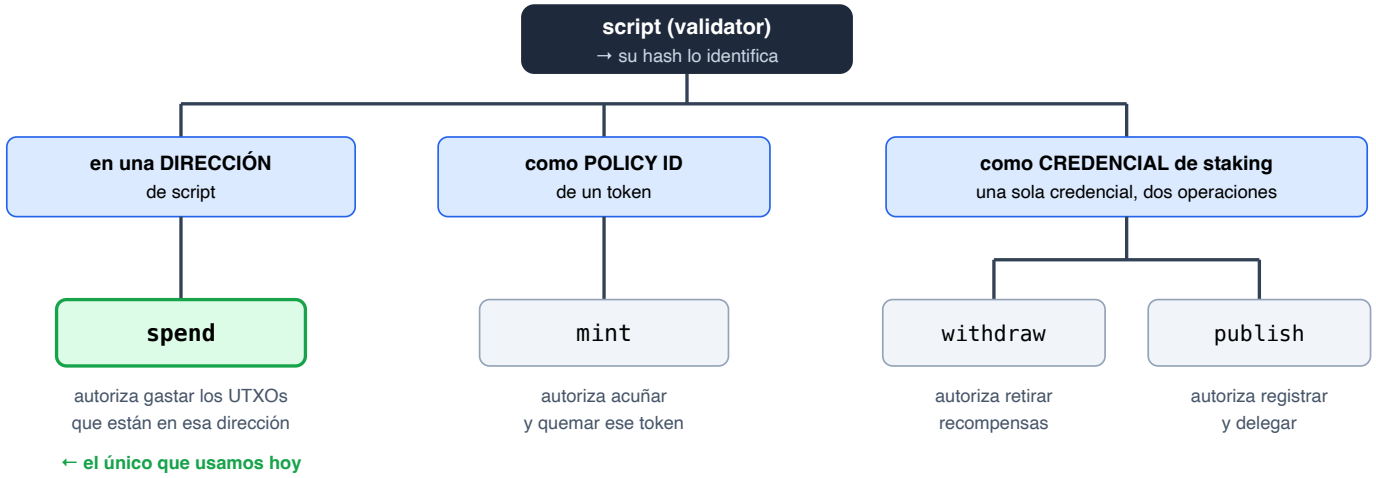
Del hash sale la **dirección** donde se bloquean los fondos. El código viaja **dentro de la transacción** que lo invoca. Al off-chain no le hace falta nada más.

SEGMENTO 4 · ~35 min

Anatomía de un validator

Un mismo script, cuatro propósitos

Se compila **un script** y su **hash** lo identifica. Según en qué lugar del ledger aparezca ese hash, la transacción invoca un **handler** distinto.



Un mismo script puede ocupar los cuatro roles a la vez; el **propósito** dice desde cuál.

Como **dirección**, el hash es un **lugar**: hay UTXOs ahí y se ven en un explorador.
Como **policy id**, es parte del **nombre** del token — no hay ningún lugar que contenga nada.

Tipos de operaciones en EUTXO

¿Qué dispara un script y qué no? En Ethereum casi todo es "una llamada a un contrato".
En Cardano **no**: muchas operaciones comunes **solo requieren la firma**.

Operación	¿Ejecuta script?	Propósito
Transferir ADA/tokens entre wallets	**No** — basta la firma	(ninguno)
Gastar un output bloqueado	**Sí**	`spend`
Acuñar tokens (mint)	**Sí** — minting policy	`mint` (cantidad **> 0**)
Quemar tokens (burn)	**Sí** — la misma policy	`mint` (cantidad **< 0**)
Delegar / retirar staking	**Sí** (si es script)	`publish` / `withdraw`

Un script corre **sólo** cuando la tx toca algo protegido por él. Mover 10 ADA es **consumir UTXOs de Alice y crear UTXOs para Bob**: el ledger sólo pide su firma.
Y no hay un handler `burn` aparte — acuñar y quemar los maneja el mismo, según el signo.

Sobre qué actúa cada uno, con un ejemplo

Propósito	Actúa sobre...	Un ejemplo concreto
`spend`	**un UTXO** que está en la dirección del script	*"quiero gastar los 5 ADA que Alice bloqueó en el escrow"*
`mint`	**el campo** `mint` de la transacción	*"quiero acuñar 100 unidades de mi token"* · con cantidad negativa, quemarlos
`withdraw`	**las recompensas** de una cuenta de staking	*"quiero retirar las recompensas acumuladas"*
`publish`	**certificados** de delegación/registro	*"quiero delegar mi stake a este pool"*

Cada propósito corre **una vez por cada unidad sobre la que actúa**: `spend`, una por input de script; `mint`, una por policy id acuñado o quemado; `withdraw`, una por cuenta de staking; `publish`, una por certificado.

> En este curso sólo escribimos `spend`; los otros se nombran porque el `else(_) { fail }` que van a ver los rechaza explícitamente.

El núcleo de `escrow.ak`: quién firma

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

pub type Datum {
  beneficiary: ByteArray,   // quién puede reclamar
  owner: ByteArray,         // quién puede cancelar
}
pub type Redeemer { Claim  Cancel }

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

Un solo handler: **`spend`** — el que corre al gastar un UTXO que vive en esta dirección. El `else(_) { fail }` rechaza los otros tres propósitos.

> El archivo del repo ya trae `deadline` y `validity_range`: es la versión terminada.
> **Esa parte la construimos en el taller.**

Datum y Redeemer

datum

viaja pegado al UTXO al depositar

info que el validator puede leer

aquí: beneficiary + owner

redeemer

lo declara quien intenta gastar

la "clave" que presenta la tx

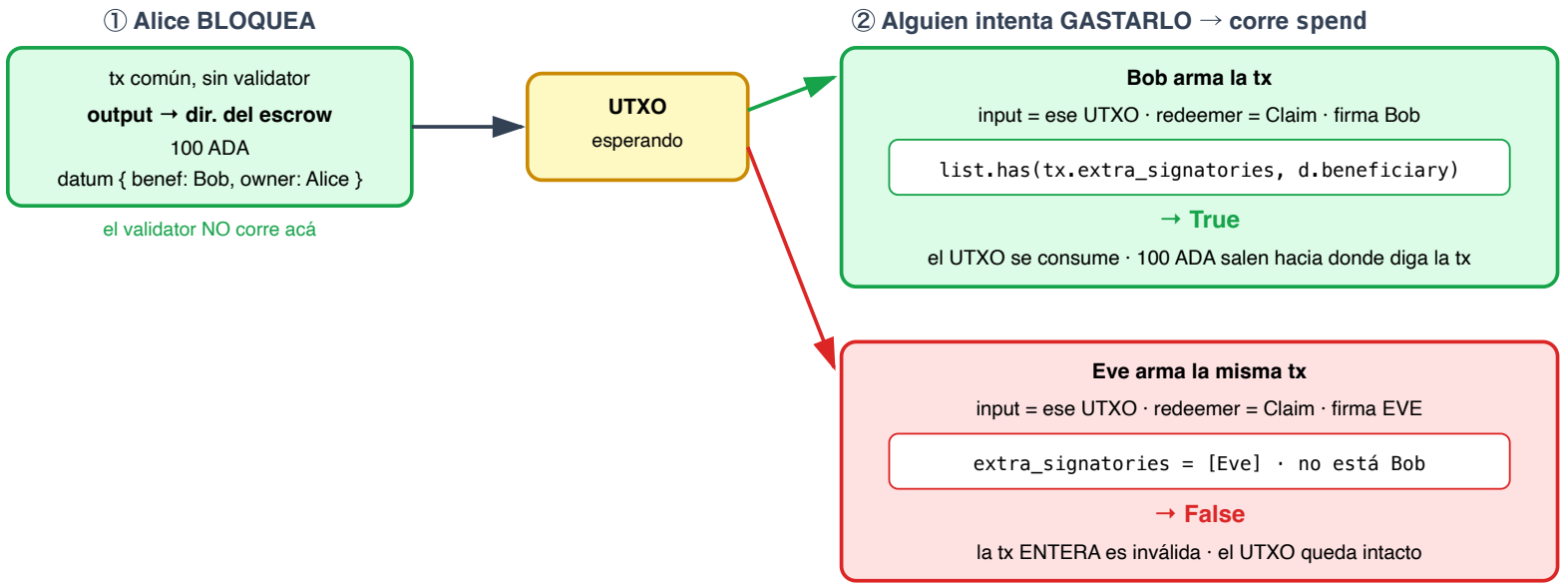
aquí: Claim o Cancel

Los cuatro parámetros de `spend`

Parámetro	Tipo	Viene de
`datum`	`Option<Datum>`	el output que se quiere gastar
`redeemer`	`Redeemer`	la tx de gasto (lo elige quien gasta)
`_utxo`	`OutputReference`	referencia al output (tx_hash + index)
`self`	`Transaction`	**todo** el contexto de la transacción

`expect Some(d) = datum` → si el datum es `None`, falla inmediatamente.
`else(_) { fail }` → rechaza cualquier propósito que no sea `spend`.

El escrow en acción: qué pasa, paso a paso



Qué efecto tiene, exactamente

> El validator **no movió un solo ADA.** Sólo dijo `True` o `False` sobre si ese input podía consumirse. **Quién recibe los 100 ADA lo decidió la transacción**, que armó Bob.

Si devuelve...	Qué pasa
`True`	la tx es válida: el UTXO **desaparece** y nacen los outputs que la tx declaró
`False`	**toda la tx** falla — no sólo ese input. El UTXO queda **intacto**, disponible

> **Y acá está la pregunta:** el validator verificó que **Bob firmó**...
> pero ¿verificó que los 100 ADA **le lleguen a Bob**? *
> Mirá el código otra vez: `list.has(extra_signatories, beneficiary)`. Nada más.

`extra_signatories` y el contexto

`self.extra_signatories` = hashes de claves que se declararon **explícitamente** como firmantes de la tx.

```
Transaction {
  inputs:            List<Input>,     // UTXOs que se consumen
  outputs:           List<Output>,    // UTXOs que se crean
  mint:              MintedValue,     // tokens acuñados/quemados
  extra_signatories: List<ByteArray>, // hashes de firmantes
  validity_range:    ValidityRange,   // rango de slots válido
  ...
}
```

> El validator puede leer **cualquier** campo: verificar destino de outputs, montos, ventana temporal — o todo a la vez.

Inline datum vs. datum hash

El datum se guarda en el output de dos formas:

	Inline datum	Datum hash
Qué guarda el output	el dato **completo**	solo el **hash**
Al gastar	ya está on-chain	quien gasta **aporta el datum** (el ledger verifica el hash)

- **El escrow usa inline**: el dato viaja con el UTXO, no hay que aportarlo al gastar
- Riesgo de auditoría: un datum-hash cuyo preimagen nadie conoce → UTXO **inservible**

> Y un UTXO enviado a una dirección de script **sin datum** cuando el validator hace `expect Some(d) = datum` queda **atascado para siempre**. Bloquear fondos así es un error real.

Un tercer lugar para poner información: el parámetro

Hasta acá el escrow guarda `beneficiary` y `owner` en el **datum**. Hay otra opción: ponerlos como **parámetros del validator**.

Con datum (lo que vimos)

```
validator escrow {
  spend(datum: Option<Datum>, r: Redeemer,
        _u: OutputReference, self: Transaction) {
    expect Some(d) = datum
    when r is {
      Claim  -> list.has(self.extra_signatories,
                         d.beneficiary)
      Cancel -> list.has(self.extra_signatories,
                         d.owner)
    }
  }
  else(_) { fail }
}
```

Parametrizado

```
validator escrow(beneficiary: ByteArray,
                 owner: ByteArray) {
  spend(_datum: Option<Data>, r: Redeemer,
        _u: OutputReference, self: Transaction) {
    when r is {
      Claim  -> list.has(self.extra_signatories,
                         beneficiary)
      Cancel -> list.has(self.extra_signatories,
                         owner)
    }
  }
  else(_) { fail }
}
```

Y eso cambia la dirección

Un **acuerdo** es cada escrow concreto: Alice con Bob, Dana con Carol.
El parámetro se aplica **antes** de compilar → otro script → otro hash → **otra dirección**.

Con Mesh: `applyParamsToScript(codigo, [param1, param2, …])`, **en el mismo orden** en que los declara el validator.

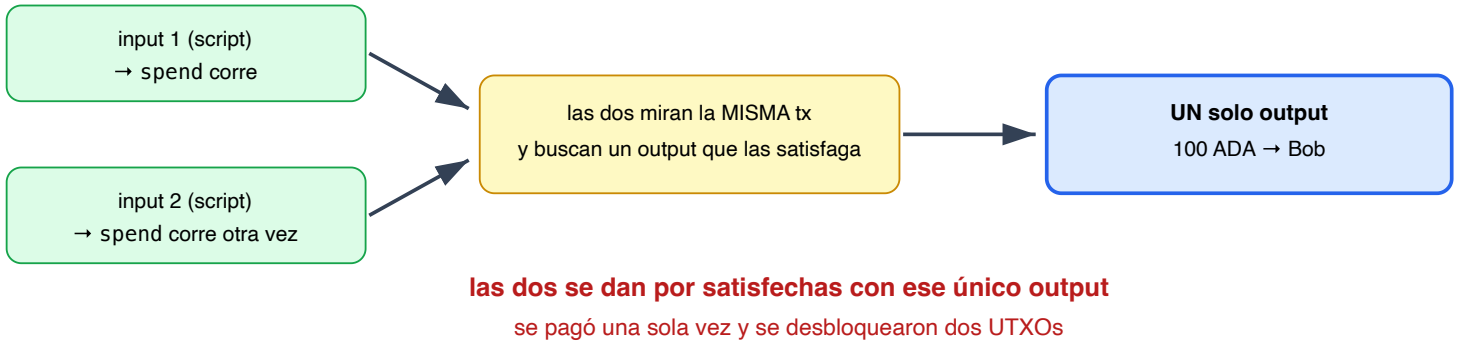
Un validator, muchas instancias: donde Ethereum usaría un `mapping`, EUTXO usa **una dirección por acuerdo** — y dos acuerdos **no comparten UTXOs**. *

Datum, parámetro o hardcodeado: cómo elegir

	Dónde vive	Cuándo se fija	¿Se puede cambiar después?
Datum	pegado al UTXO	al **depositar**	**sí** — una tx que actualice el datum
Parámetro	dentro del script ya compilado	al **aplicar los parámetros**	**no** — cambia el hash, es otra dirección
Hardcodeado	en el código fuente	al **escribirlo**	hay que recompilar, re-testear y **re-auditar**

> Si cambia por UTXO, **datum**. Si identifica a la instancia y no cambia nunca, **parámetro**. Hardcodeado sólo si es igual para todas las instancias y para siempre.

Cómo corre un validator



- Un **input de script** es un input que viene de un UTXO bloqueado en la dirección de un validator. `spend` corre **una vez por cada uno**: dos inputs de script, dos ejecuciones.
- Las dos reciben **la misma transacción** en `self`, y **ninguna sabe que la otra existe** — en EUTXO no hay llamadas entre validators, la composición es por transacción.

> De ahí sale una regla al escribir un validator: verificar **su** output, no que *exista* alguno que lo satisfaga. Si se conforma con que exista, el mismo output sirve para las dos.

SEGMENTO 5 · ~20 min

Tokens nativos y min-ADA

ERC-20 vs token nativo

Ethereum (ERC-20)

"crear token" = desplegar un contrato

mapping address → balance

transferir = mutar storage

Cardano (nativo)

no hay contrato de token

los activos viven en el ledger

un UTXO lleva ADA + varios tokens

Un UTXO con múltiples activos

```
UTXO {
  value: {
    lovelace: 2_000_000,            // 2 ADA (lovelace = unidad mínima)
    "a1b2...policy_id": {
      "MiToken": 100,
      "OtroToken": 1,
    }
  }
}
```

Un token se identifica por el par `(policy_id, asset_name)`:

- **`policy_id`** = hash de la *minting policy* (script que controla acuñar/quemar)
- **`asset_name`** = bytes arbitrarios (el "nombre")

Cómo se acuña un token

```
inputs:   un UTXO tuyo                          ← paga el fee y el min-ADA
mint:     { policy_id: { "MiToken": +100 } }    ← acá nacen los 100
witness:  el script de la policy                ← corre con propósito mint
outputs:  tu dirección → 2 ADA + 100 MiToken    ← acá terminan
signers:  [tu clave]                            ← si la policy la exige
```

No hay paso previo. A diferencia de `spend`, no hay que mandar nada a ninguna dirección antes: la primera transacción que menciona la policy es la que acuña.

- La tx tiene que **balancear**: `inputs + mint = outputs + fee`
- El output que lleva los tokens **necesita ADA**: no viajan solos
- Quemar es lo mismo con cantidad **negativa**, consumiendo un UTXO que los tenga

Minting policy mínima

```
validator mi_token(creador: ByteArray) {
  mint(_redeemer: Void, _policy_id: PolicyId, tx: Transaction) {
    list.has(tx.extra_signatories, creador)
  }
  else(_) { fail }
}
```

Una minting policy corre al **acuñar o quemar** tokens (no al transferirlos).

Está **parametrizada**: `creador` es un parámetro, no un datum — un `mint` no tiene datum de dónde leerlo. Cada `creador` distinto produce **otro policy id**.

> Una policy también se puede escribir como **native script** (`sig` + `before`): así se hacen la mayoría de los NFTs con emisión limitada en el tiempo.

No hay "hook" de transferencia

Como transferir un token nativo **no ejecuta ningún script**...

> **No se puede** poner lógica que corra en cada transferencia — p. ej. cobrar una comisión por envío, como hacen algunos ERC-20.

- Requeriría bloquear los tokens en un validator y que **toda** transferencia pase por él (cambia la UX y la composición)
- No es una limitación de Aiken: es una consecuencia del **modelo**

> Tesis del curso: *el modelo de ejecución condiciona qué es fácil y qué es difícil de construir.*

min-ADA

Todo UTXO debe llevar un **mínimo de ADA** (~1–2 ADA, más si tiene tokens o datum grande).

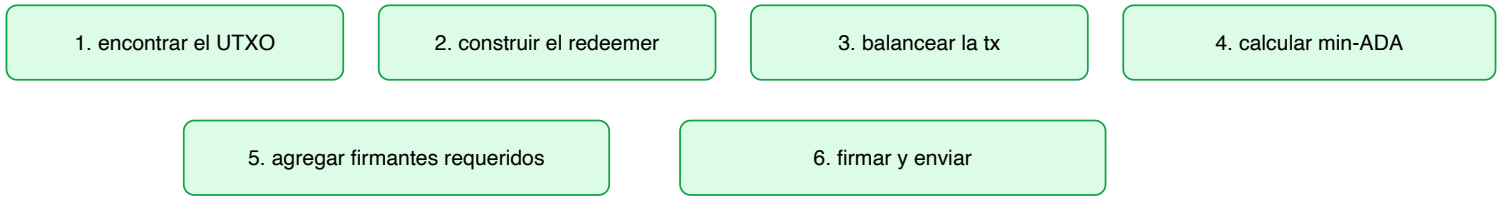
- Existe porque **almacenar UTXOs tiene costo** (memoria de los nodos)
- El ADA actúa como **depósito**: se recupera al gastar el UTXO

> Origen de ataques de **"dust"**: crear UTXOs tan chicos que son más caros de gastar que su valor. El off-chain debe calcular y agregar el min-ADA o la tx falla.

SEGMENTO 6 · ~35 min

Off-chain + mini-demo

Los 6 pasos que el off-chain hace antes del `True`



El validator on-chain tiene **5 líneas**. El off-chain coordina todo lo demás.

Inversión de complejidad respecto a Ethereum.

El flujo lock / unlock

```
[LOCK: Alice bloquea]                    ← el validator NO corre
  output → dirección del escrow
         → datum { beneficiary: Bob, owner: Alice, deadline }
         → 5 ADA

[UNLOCK: Bob reclama]                    ← acá SÍ corre spend
  input  → el UTXO anterior (el datum ya está en la cadena)
  redeemer → Claim
  required_signers → [Bob]
  validity_range → hasta el deadline
  output → dirección de Bob, 5 ADA − fee
  → Bob firma → el validator corre → True → tx válida
```

Depositar no requiere permiso. Mandar ADA a una dirección de script es una transferencia común: **cualquiera** puede hacerlo, y **ningún** validator se ejecuta. El validator sólo controla la **salida** de los fondos, nunca la entrada.

> Por eso el handler se llama `spend`: sólo hay algo que autorizar cuando se **gasta** el UTXO. Y por eso el datum lo pone **quien deposita**, no el validator — un detalle con consecuencias.

¿De dónde sale la billetera? CIP-30

Una **CIP** (Cardano Improvement Proposal) es cómo el ecosistema estandariza cosas.
La **CIP-30** estandariza el puente entre una página web y la billetera del usuario.

La dApp pide...	La billetera responde
UTXOs, dirección, colateral	los datos necesarios para **armar** la tx
firmar una tx	el usuario aprueba en su extensión → tx firmada
enviarla	la billetera la **submitea** a la red

Sin CIP-30 habría que integrar cada billetera por separado. Con una librería compatible (Mesh, Lucid), tu dApp funciona **con todas** — la billetera es intercambiable.

El `wallet` de la próxima slide es exactamente esto: un objeto CIP-30. Nunca entrega tu clave privada — sólo devuelve la tx firmada.

El off-chain (Mesh.js)

```
const tx = new MeshTxBuilder({ fetcher });

tx.spendingPlutusScript("V3")            // la VERSIÓN de Plutus, no el código
  .txIn(utxo.txHash, utxo.index)         // el UTXO bloqueado
  .txInScript(validatorCbor)             // el código: sale de `aiken build`
  .txInInlineDatumPresent()              // el datum ya está en la cadena
  .txInRedeemerValue(mConStr0([]))       // Claim = constructor 0
  .requiredSignerHash(bobPubKeyHash)     // declarar el firmante
  .txInCollateral(col.txHash, col.index) // se pierde si el script falla
  .invalidHereafter(slotDelDeadline)     // sin esto, can_claim rechaza
  .changeAddress(bobAddress);

await tx.complete();                     // balancea, calcula fee y ExUnits
const firmada = await wallet.signTx(tx.txHex);
await wallet.submitTx(firmada);
```

> Este mismo código, corriendo de verdad: `npm run demo-simple` — 40 líneas en `cardano/offchain/src/demo-simple.ts`.

La inversión de complejidad on-chain / off-chain

> El código del validator tiene **5 líneas** significativas. El off-chain coordina inputs, outputs, balanceo, serializaciones y firmantes.

En Ethereum el off-chain sólo arma el calldata y firma. Esta **inversión de complejidad** es consecuencia directa de EUTXO — y explica por qué los bugs en Cardano viven en la **interacción** on-chain/off-chain más que en el validator en sí.

Actividad — leer un validator

En pares, leer `cardano/validators/escrow.ak` y responder:

1. ¿Qué info necesita el datum para funcionar?
2. ¿Bajo qué condición exacta aprueba con `Claim`? (son **dos** condiciones)
3. ¿Y con `Cancel`? ¿Por qué esa asimetría?
4. ¿Qué pasa si alguien intenta gastar **sin** incluir ninguna firma?
5. **Análisis:** ¿el validator verifica que los fondos lleguen al beneficiario?

> La 5 no se contesta de memoria: **busquen en el código** dónde estaría esa verificación. Si no la encuentran, esa ausencia *es* la respuesta. *

El hueco del escrow

El escrow verifica firmas pero **no verifica qué outputs crea la tx**.

> Pregunta para pensar: dado que el validator tiene acceso a `self.outputs`, ¿qué tendría que verificar para que `Claim` sea **seguro**?

Lecturas

- **Aiken**: `aiken-lang.org` — "Getting Started" y "Language Tour"
- Cardano Developer Portal — smart contracts y EUTXO
- **Mesh.js**: `meshjs.dev` — para explorar el off-chain
- *Mastering Cardano* (IOG) — **cap. 8**: smart contracts y tokens nativos
- **"I Can Aiken"** (J. Greene) — libro hands-on, para practicar sintaxis
- **Cardano Academy — "eUTxO Smart Contracts"** — curso oficial con Aiken
- **CIP-30**: `cips.cardano.org/cip/CIP-30` — corta y legible

Próxima clase

Clase 5 — Taller Aiken

Completar `escrow.ak` con la condición temporal (`validity_range` + deadline)
y escribir tests con `transaction.placeholder` .