Module 5

States

corda

Learning outcomes

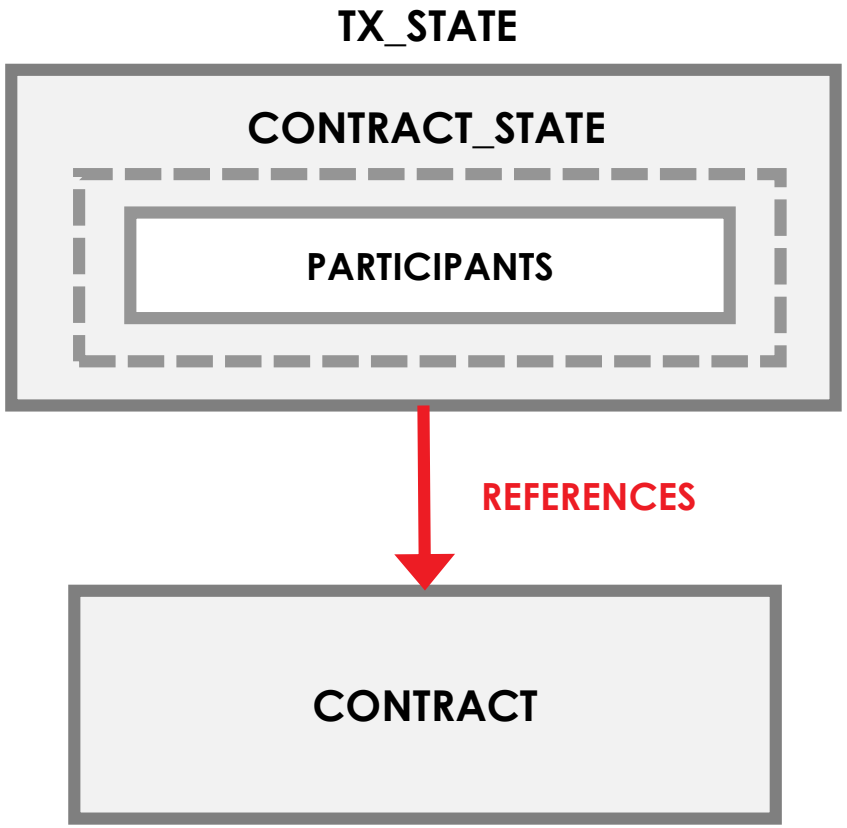
- Learn what key interfaces states need to implement
- Learn how a node's vault decides whether to track a state
- Learn how to design your own states

ContractState and TransactionState

- All states implement the **ContractState** interface:

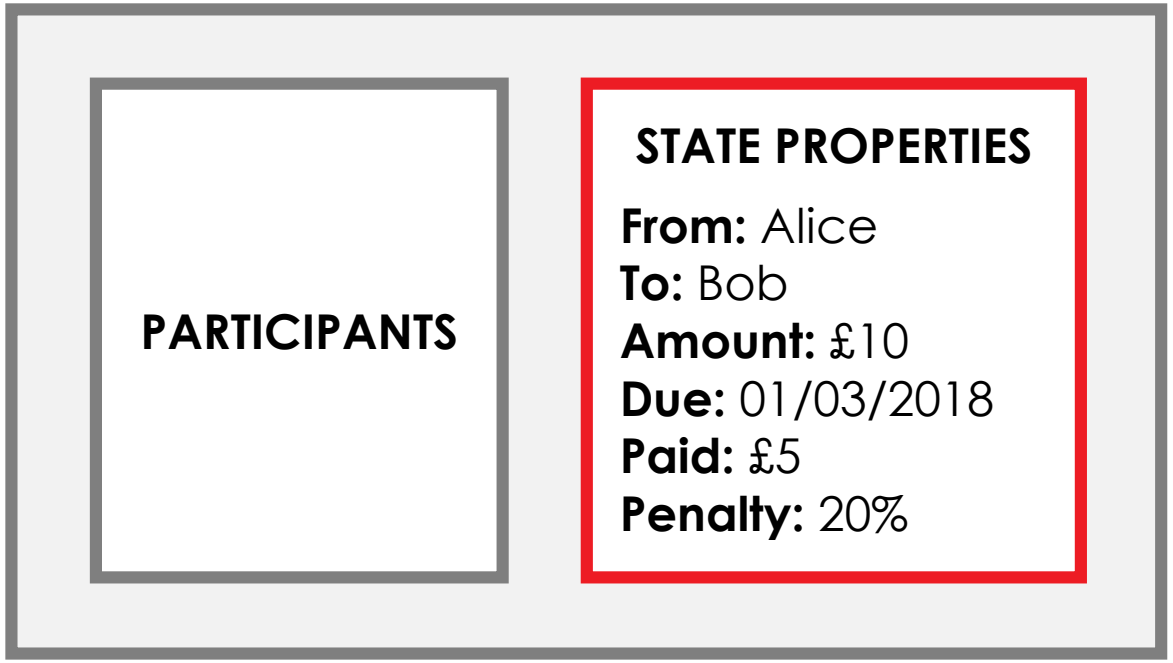
```
interface ContractState {
    val participants:
        List<AbstractParty>
}
```

- Contract state does not refer to a contract directly
- **contract** defines the constraints on the evolution of instances of this state type
- **participants** lists anyone who is able to evolve this state



User-defined fields

Classes implementing the **ContractState** interface can also have unlimited user-defined fields and methods…



AbstractParty

- In Corda, all nodes are identified as **AbstractParty** instances:

```
class AbstractParty(val owningKey: PublicKey)
```

- **owningKey** is a **PublicKey** representing the node's master public key
- **AbstractParty** provides no additional information to identify the node

Party

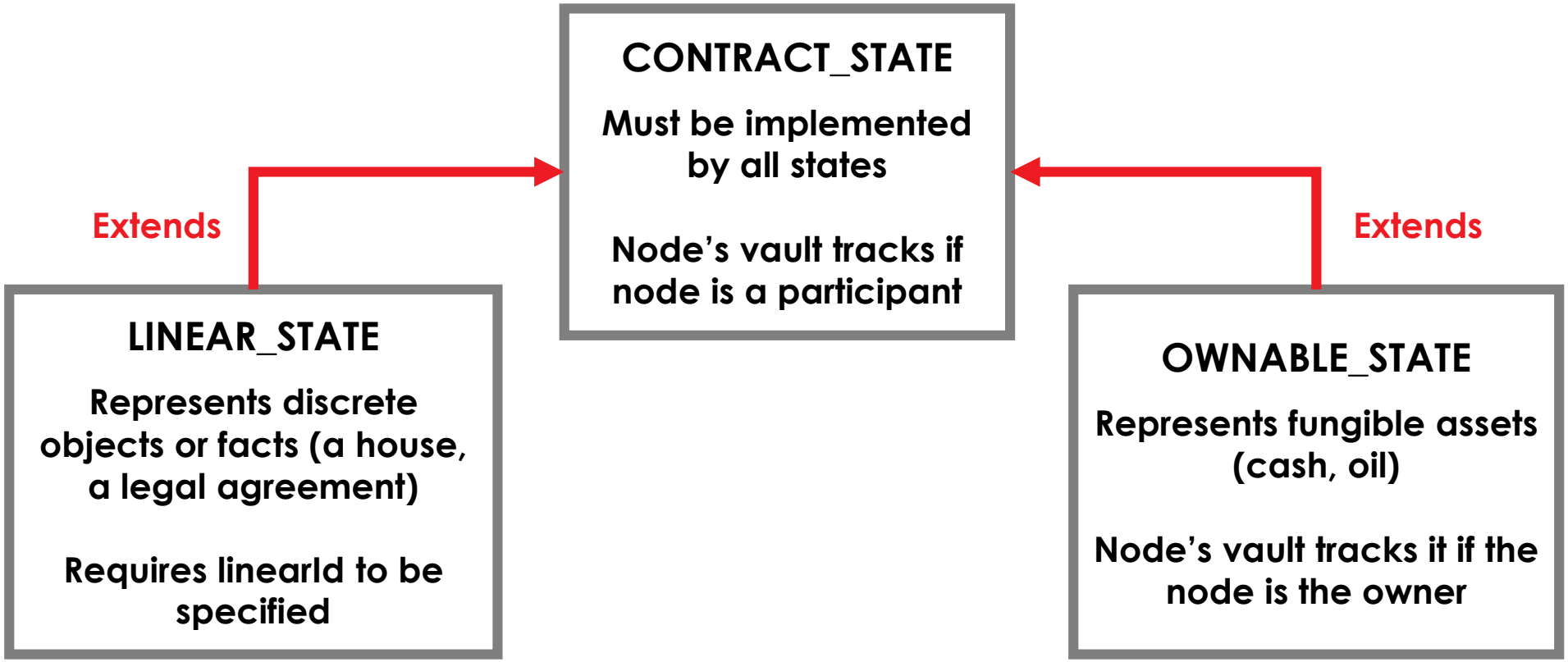
- **Party** is an **AbstractParty** subclass that associates the master public key with an identity:

```
class Party(val name: CordaX500Name, val owningKey: PublicKey)
```

- **name** is the node's X500 name
- **owningKey** is the node's master public key, as before

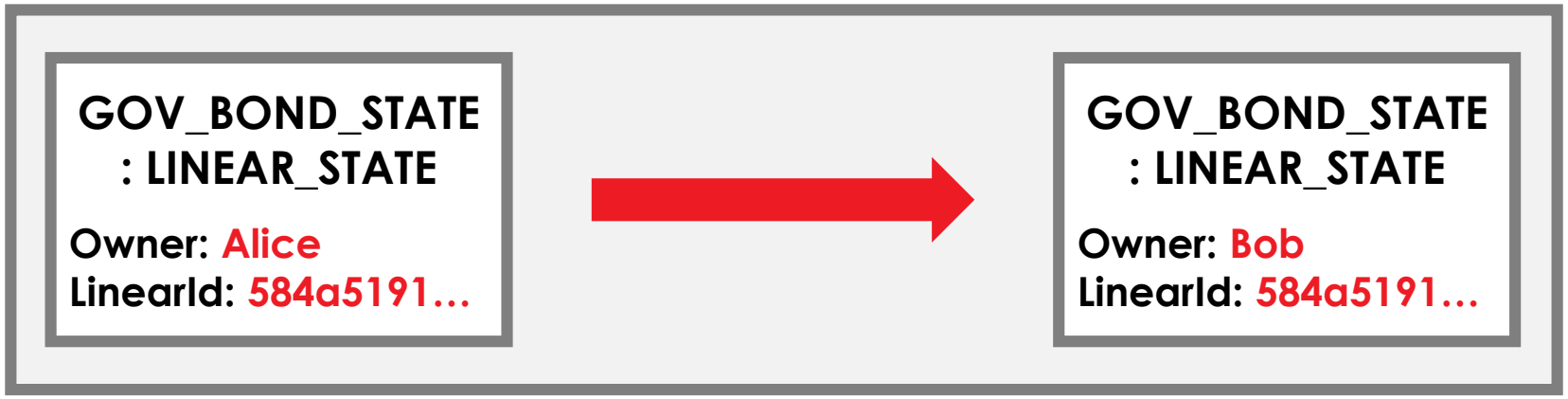
State types

ContractState has several key child interfaces:



LinearState

- **LinearState** is used to represent discrete objects or facts that evolve over time (a specific bond, a trade finance deal…)
- In Corda's UTXO model, we can't modify the state directly
- So we evolve the state through a transaction that consumes the old state and creates an updated one with the same ID:



LinearState

LinearState attributes and methods:

- **linearId: UniqueIdentifier**
  - A unique ID that allows the fact to be tracked over time and referenced in external systems

A common LinearState usecase: agreements

Agreements (contracts, bilateral derivatives, invoices) are a good fit for **LinearState**:

- A given legal agreement has a single identity over time
- Agreements evolve over time by replacing the existing agreement
- We can only evolve the agreement by modifying the most recent version…
- But we still have access to the old versions if required

An example LinearState: the NumberState

- **NumberState** extends **LinearState** to represent a number on ledger:

```
data class NumberState(
    val number: Int,
    val alice: Party,
    val bob: Party,
    override val linearId: UniqueIdentifier =
                                                UniqueIdentifier()
) : LinearState {

    override val contract = NumberContract()

    override val participants
        get() = listOf(alice, bob)

}
```

OwnableState

- **OwnableState** is used to represent fungible assets with an owner (cash, barrels of oil, bushels of corn…)

```
interface OwnableState : ContractState {
    val owner: AbstractParty
    fun withNewOwner(newOwner: AbstractParty):
        Pair<CommandData, OwnableState>
}
```

- **owner** is the **AbstractParty** that owns the state
- **withNewOwner()** creates a copy of the **OwnableState** with a new owner
- There is no unique identifier – **OwnableStates** with the same attributes (e.g. two £100 cash states) are effectively identical

A common OwnableState usecase: cash

Cash will implement **OwnableState**:

- Two cash states with the same currency and the same value are identical
- Cash states can be split and merged
- Instead of spending a specific cash state, we spend an amount of cash (i.e. a set of states) of a given value

An example LinearState: the BondState

- **BondState** extends **OwnableState** to represent a fungible bond:

```
data class BondState(
    val maturityDate: Date,
    val nominal: Int,
    val couponPercent: Int,
    override val owner: AbstractParty
) : OwnableState {

    override val contract = BondContract()

    override val participants
        get() = listOf(alice, bob)   }
```

States in summary

- **States** represent shared facts on the ledger
- States must directly or indirectly implement **ContractState**
- All states have a participants list
- States may also wish to implement some child interfaces:
  - **LinearState**
  - **OwnableState**


r3
Practical

The IOUState Template

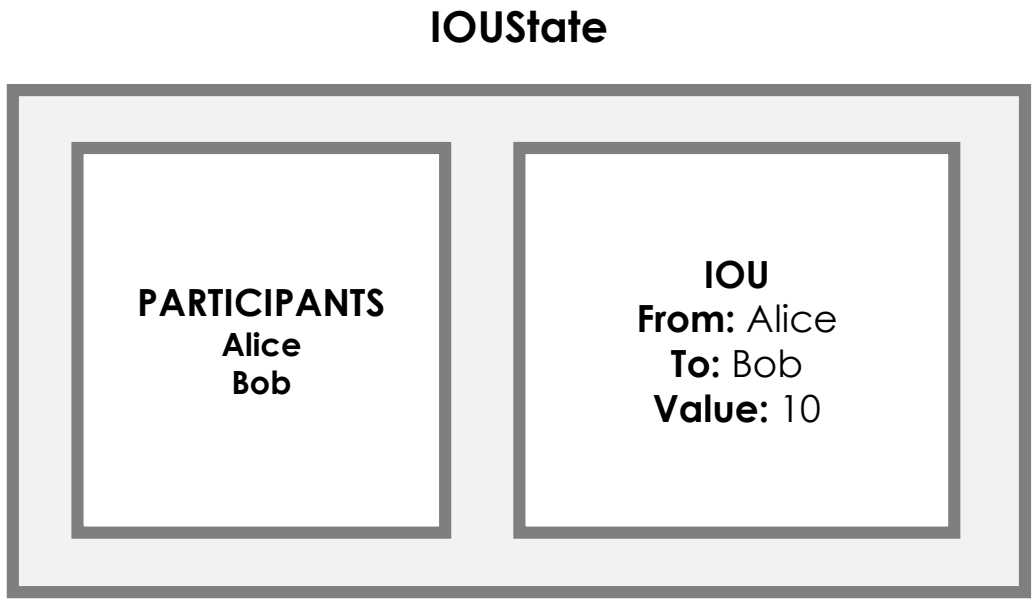
- The IOUState provided in the template is just a skeleton

ContractState:

```
data class IOUState(val data: String = "data"): ContractState {
    override val participants: List<AbstractParty> get() = listOf()
}
```

IOUState

- Whereas our IOUState has the following design:



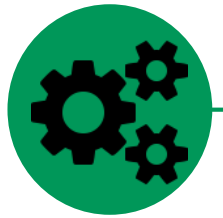
- We are going to use test-driven development to implement this design

Step 1 – The IOU Fields

IOU Fields

- Our IOUState needs the following fields:
  - amount: the Amount<Currency> value of the IOU
  - lender: the Party lending the amount
  - borrower: the Party borrowing the amount
  - paid: the Amount<Currency> of the IOU which has been paid, it should be initialised to an Amount of 0
- Party is a Corda class representing an entity on the network

IOU Fields - Implementation



Goal	Add the amount, lender, borrower and paid fields
Where?	• test/kotlin/states/IOUStateTests.kt • state/IOUState.kt
Steps	1. Uncomment the following tests: • hasIOUAmountFieldOfCorrectType() • hasLenderFieldOfCorrectType() • hasBorrowerFieldOfCorrectType() • hasPaidFieldOfCorrectType() 2. Run the tests: • Press the green arrow/play button next to the IOUStateTests class 3. Modify IOUState.kt to make the tests pass
Key Docs	https://www.jetbrains.com/help/idea/2016.3/run-debug-configurations.html https://docs.corda.net/tutorial-test-dsl.html

IOU Fields - Solution

Goal	Add the amount, lender, borrower and paid fields

Steps	• Add fields of the correct type for:
	  • amount
	  • lender
	  • borrower
	  • paid

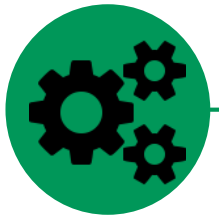
Code	`data class IOUState(`
	`  val amount: Amount<Currency>,`
	`  val lender: Party,`
	`  val borrower: Party,`
	`  val paid: Amount<Currency> = Amount(0, amount.token))`
	`: ContractState {`
	`  ...`
	`}`

Step 2 – participants

getParticipants()

- Remember that all ContractStates have a participants field holding a list of the Partys involved in the state
- For our IOUState, participants should return:
  - The lender
  - The borrower

getParticipants() - Implementation



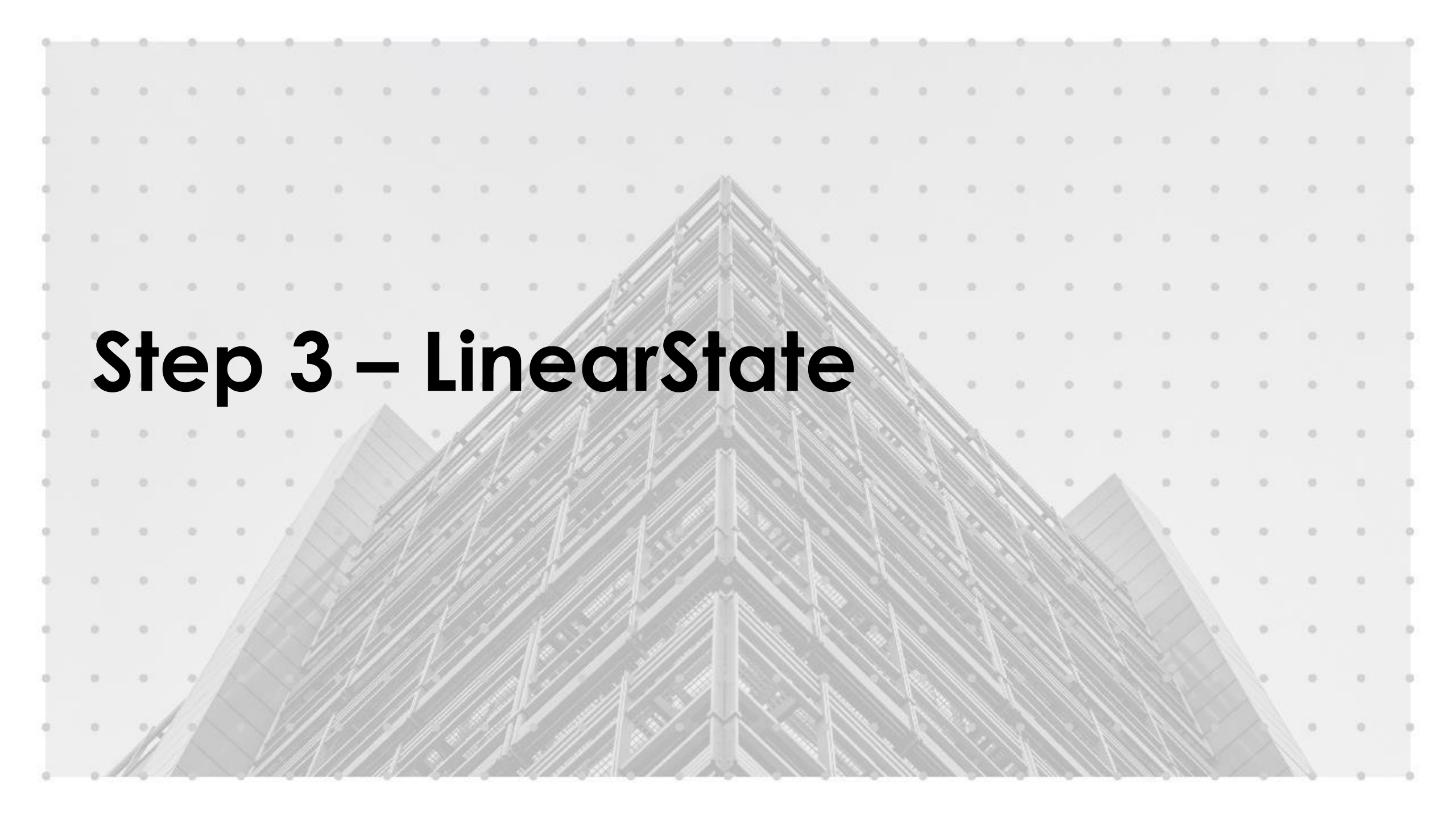
Goal	Return a list of the lender and borrower from participants
Where?	• test/states/IOUStateTests.kt • state/IOUState.kt

Steps	1. Uncomment the following tests:
	  • lenderIsParticipant()
	  • borrowertIsParticipant()
	2. Run the tests:
	  • Press the green arrow/play button
	3. Modify IOUState.kt to make the tests pass

Key Docs	N/A

getParticipants() - Solution

Goal	Return the lender and borrower CompositeKeys from participants
Steps	• Retrieve the sender and the recipient's CompositeKeys • Return them as a list
Code	`override val participants : List<AbstractParty> get() = listOf(lender, borrower)`



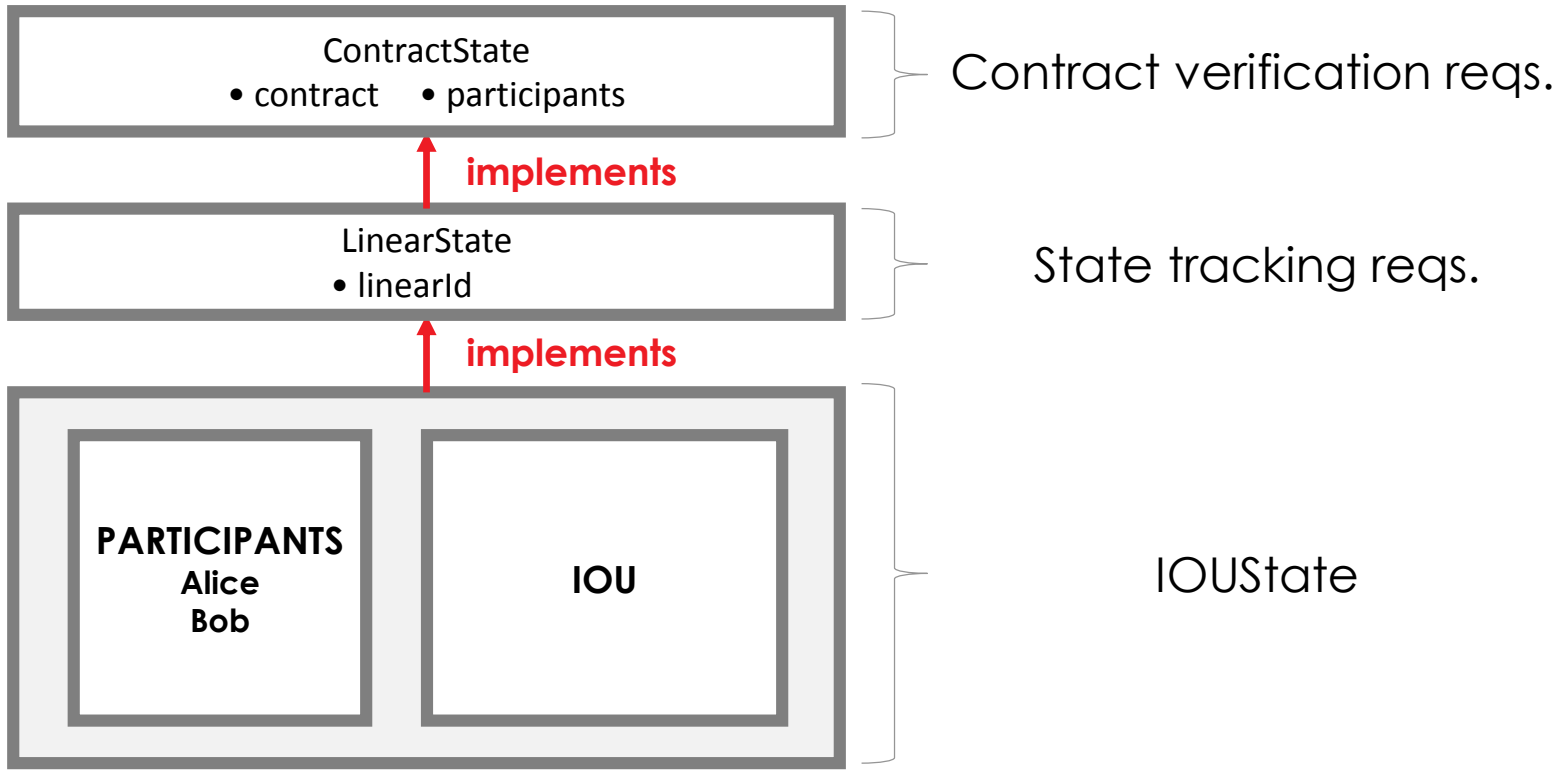
Step 3 – LinearState

LinearState

- Remember that LinearState is a child interface of ContractState that models facts that evolve over time
- Implementing LinearState will allow the same IOU to be tracked across ledger updates
  - Creation, transfer, redemption…

The IOUState Inheritance Tree

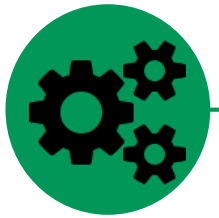
- We need to extend our IOUState to implement LinearState:



LinearID

- The LinearId for our IOU can simply be a new UniqueIdentifier instance

LinearState - Implementation



Goal	Make IOUState implement LinearState
Where?	• test/states/IOUStateTests.kt • state/IOUState.kt

Steps	1. Uncomment the following test:
	  • isLinearState()
	  • hasLinearIdFieldOfCorrectType()
	2. Run the test:
	  • Press the green arrow/play button
	3. Modify IOUState.kt to make the tests pass

Key Docs	N/A

LinearState - Solution

Goal	Make IOUState implement LinearState
Steps	• Implement LinearState • Override the linearId field

Code	**public class** IOUState (
	… **override val** linearId: UniqueIdentifier = UniqueIdentifier())
	implements LinearState {
	…
	}

Checkpoint – Progress So Far

Our progress so far

- Our **IOUState** allows us to model an IOU on the ledger:
  - It has **value**/**sender**/**recipient** fields to store IOU information
  - It references an **IOUContract** governing state evolution
  - It implements **LinearState**:
    - To provide a common ID over time
    - To tell the vault when to track the state
- However, the evolution of **IOUState**s is currently completely uncontrolled
- We need to modify **IOUContract**'s logic to control this evolution