



Module 6

Contracts

c·rda

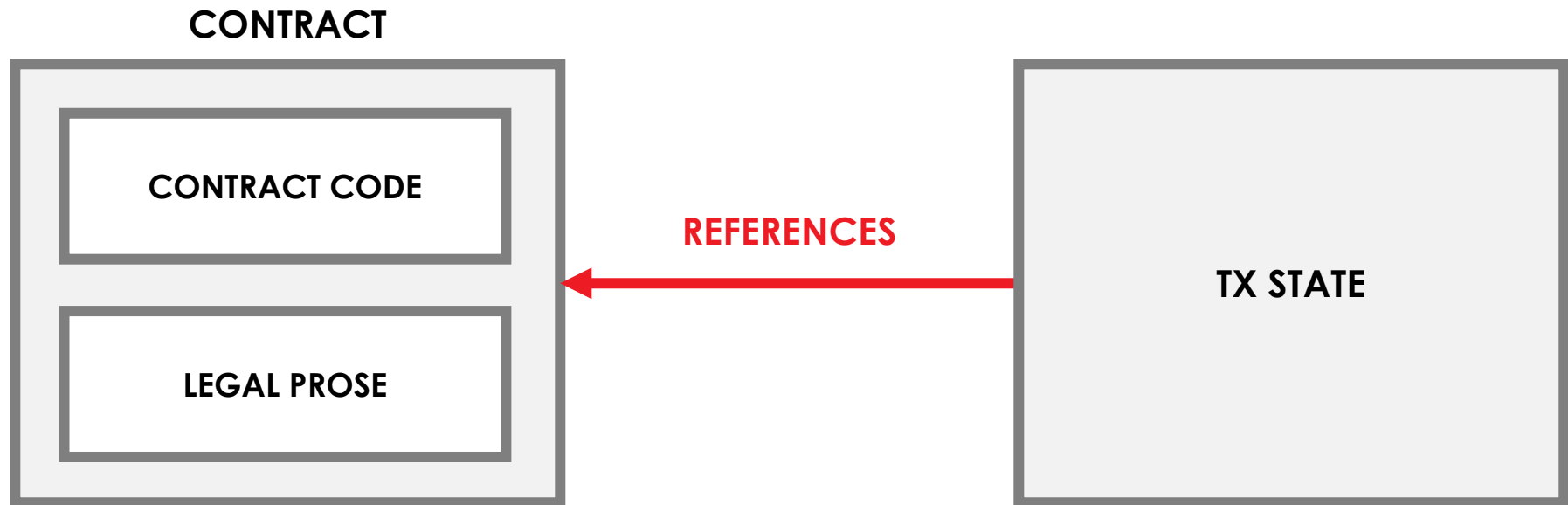


Learning outcomes

- Learn how contracts control the evolution of states
- Learn how a transaction's states are grouped for verification
- Learn the purpose of commands
- Learn how to design your own contract

Contracts

A contract is an object **referenced by a transaction state** that contains **legal prose** and **contract code** governing the state's evolution.



Contracts

All contracts must implement the **Contract** interface:

```
interface Contract {  
    @Throws(IllegalArgumentException::class)  
    fun verify(tx: LedgerTransaction)  
}
```



Contracts

Contracts might optionally be annotated with
@LegalProseReference annotation for a legal prose
reference

The `verify()` method

The `verify()` method takes a `LedgerTransaction` as input and returns either:

- An exception if the supplied transaction is invalid according to the contract's rules
- `Unit` if the supplied transaction is valid

IMPORTANT: In verifying a transaction, the `verify()` method **ONLY HAS ACCESS** to the contents of `LedgerTransaction`.

LedgerTransaction

- **LedgerTransaction** has all the transaction's contents

available for verification:

```
val inputs: List<StateAndRef>
val outputs: List<TransactionState<ContractState>>
val attachments: List<Attachment>
val commands: List<AuthenticatedObject<CommandData>>
val id: SecureHash
val notary: Party?
val signers: List<PublicKey>
val timeWindow: TimeWindow? = null
val type: TransactionType
val privacySalt: PrivacySalt
```

r3.

- It also has methods to easily extract these transaction elements

The simplest contract

The simplest possible contract would be defined as follows:

```
class SimplestContract: Contract {  
    companion object {  
        @JvmStatic  
        val CONTRACT_ID = "com.example.Contract"  
    }  
    override fun verify(tx: LedgerTransaction) {  
        // No constraints, so accepts anything.  
    }  
}
```


An example: Writing a verify function

- Let's write a verify function for the following state:

```
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)  
}
```

The NumberContract

Our **NumberContract** will allow:

- The creation of new, positive-value **NumberStates**
- Adding non-negative amounts to existing **NumberStates**

These two possibilities correspond to two commands:

- Create
- Add

The NumberContract's commands

```
class NumberContract: Contract {  
    // contract id was omitted...  
  
    interface Commands : CommandData {  
        class Create : TypeOnlyCommandData(), Commands  
        class Add : TypeOnlyCommandData(), Commands  
    }  
  
    override fun verify(tx: LedgerTransaction) {  
        // verify() on next page...  
    }  
}
```

The NumberContract's verify function

```
fun verify(tx: LedgerTransaction) {  
    val command = tx.findCommand<NumberContract.Commands> { true }  
  
    when (command.value) {  
        is Commands.Create -> { /* Create verification logic. */ }  
        is Commands.Add -> { /* Add verification logic. */ }  
        else ->  
            throw IllegalArgumentException("Unknown command $command")  
    }  
}
```

verify code for Create command

```
is Commands.Create -> {  
    requireThat {  
        "There are no inputs" using (tx.inputs.isEmpty())  
        "There is only one output" using (tx.outputs.size == 1)  
  
        val out = tx.outputsOf<NumberState>().single()  
        "Number must be positive" using (out.number > 0)  
        "The participants are distinct" using (out.alice != out.bob)  
  
        val participantKeys = out.participants.map { it.owningKey }  
        "All participants must be signers" using  
            (command.signers.containsAll(participantKeys))  
    }  
}
```

verify code for Add command

```
is Commands.Add -> {  
    requireThat {  
        "There is only one input" using (tx.inputs.size == 1)  
        "There is only one output" using (tx.outputs.size == 1)  
  
        val input = tx.inputsOf<NumberState>().single()  
        val out = tx.outputsOf<NumberState>().single()  
        "Amount added is >0" using (input.number < out.number)  
        "The participants are distinct" using (out.alice != out.bob)  
  
        val participantKeys = out.participants.map { it.owningKey }  
        "All participants must be signers" using  
            (command.signers.containsAll(participantKeys))  
    }  
}
```

verify() can be complex!

```
override fun verify(tx: TransactionForContract) {
    val stateGroups = tx.groupStates(UTIMatchingState::class.java, { it.linearId })
    val matchGroups = tx.groupStates(UTIMatchedState::class.java, { it.linearId })
    val command = tx.commands.requireSingleCommand<UTIMatchingContract.Commands>()
    require(tx.timestamp?.midpoint != null) { "must be timestamped" }
    when (command.value) {
        is Commands.Issue -> {
            require(matchGroups.isEmpty()) { "Issue must not contain any UTIMatchedState" }
            requireThat {
                "Issue of new UTIMatchingState must not include any inputs" by (tx.inputs.isEmpty())
                "Issue of new UTIMatchingState must be in a unique transaction" by (tx.outputs.size == 1)
            }
            val issued = tx.outputs.get(0) as UTIMatchingState
            requireThat {
                "Initial Issue state must be INITIAL" by (issued.matchingState == InitialState(issued.matchingState.content, issued.matchingState.submittedBy.ownId))
                "Issue requires the submitting Party as signer" by (command.signers.contains(issued.matchingState.submittedBy.ownId))
            }
        }
        is Commands.Validate -> {
```

Contracts in summary

- Contracts decide which transactions are valid, and therefore control the evolution of states over time
- For verification, you only have access to the contents of **LedgerTransactionForContract**
- Commands provide additional information and are often used to fork the execution of **verify()**
- The **verify()** function can be quite complex



r3

Practical

IOUContract

- In the `IOUContract.kt` template:
 - `legalContractReference` holds a hash of a dummy string
 - `verify` has an empty body
- Currently, the contract accepts every transaction (i.e. `verify` never throws an exception)
- We are now going to add constraints to control the evolution of `IOUStates`

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

Step 1 – Contract Tests

A low-angle, black and white photograph of a modern skyscraper with a complex, geometric facade, viewed through a grid of small dots. The building's structure is composed of many sharp angles and lines, creating a sense of height and architectural complexity. The grid of dots is a light gray, semi-transparent overlay that covers the entire image, adding a technical or digital feel to the composition.

Testing Contracts

- We test contract behavior using **LedgerDSL**
- **LedgerDSL** allows you to:
 - Create mock transactions
 - Test whether these are valid based on contract rules
- **LedgerDSL** also provides:
 - Dummy parties (**MINI_CORP**, **MEGA_CORP**...)
 - Dummy keys (**MINI_CORP_PUBKEY**...)

r3.

1. CorDapp Design

2. State

3. Contract

- **Contract Tests**
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

LedgerDSL Syntax

- Corda's **NodeTestUtils** provide a **ledger** function, which takes a **LedgerDSL** lambda as an argument
- **LedgerDSL** exposes a **transaction** function, which takes a **TransactionDSL** lambda as an argument:

```
// Define your states, etc. here first.
ledger {
    transaction {
        // TODO: Test our transaction
    }
}
```



- **Contract Tests**
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

TransactionDSL Syntax

- **TransactionDSL** is a mock transaction to which we can add inputs, outputs and commands:

```
...
transaction {
    input(INPUT_STATE) // An input state.
    output(OUTPUT_STATE) // An output state.
    command(KEYS, COMMAND) // A transaction command.
});
...
```

- We can then assert whether the contract is valid or not (with a specific message):

```
...
transaction {
    input(INPUT_STATE) // An input state.
    output(OUTPUT_STATE) // An output state.
    command(KEYS, COMMAND) // A transaction command.
    failsWith(FAILURE_MSG) // Assert transaction failure.
    verifies() // Assert transaction success.
}
...
```



- **Contract Tests**
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint



Step 2 – The Create Command

Command Recap

- Remember that commands play two roles in a transaction:
 - Parameterizing the running of a **Contract**'s **verify** function
 - e.g. *executing different constraints for issuances vs. transfers*
 - Attaching signatures to transactions
- We will define a **Issue** command that is only used to attach signatures to IOU transactions
- We will require this command in every transaction involving an **IOUState**

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- **The Create Command**
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

Adding the Command

- We will define the **Issue** command inside the **Commands** interface which has been provided inside **IOUContract**:

```
class Create : TypeOnlyCommandData(), Commands
```

- We also need to require the **Issue** command in the **verify** function:
 - Within **verify**, we access a transaction's commands using **tx.commands**
 - We retrieve the command's type using **Command.value**
- Refer to the unit test instructions for more details

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- **The Create Command**
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

The Command Test - Implementation



Goal	Require the Issue command in valid transactions
Where?	test/contract/IOUIssueTests.kt Contract/IOUContract.kt
Steps	<ol style="list-style-type: none">1. Uncomment the mustIncludeIssueCommand test2. Run the test using the "Kotlin – IOU Transaction Tests" run config3. Modify IOUContract.kt to make the tests pass
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- **The Create Command**
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

Adding the Constraint - Solution



Goal	Impose a constraint on the command type in <code>IOUContract.verify</code>
Steps	<ul style="list-style-type: none">• Check that there is only one command• Check that it is of type <code>IOUContract.Commands</code>
Code	<pre>val command = tx.commands.requireSingleCommand<IOUContract.Commands>()</pre>

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- **The Create Command**
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

A low-angle, black and white photograph of a modern skyscraper with a complex, angular facade, viewed through a grid of small dots. The building's structure is composed of many sharp angles and lines, creating a sense of height and architectural complexity. The grid of dots is a light gray, semi-transparent overlay that covers the entire image, adding a technical or digital feel to the composition.

Step 3 – Further Constraints

Constraint Types

There are three broad types of constraints:

- Constraints on the attributes of the shared facts
 - e.g. no cash states over USD10,000, max 100 items per order...
- Constraints on the types of transactions that are valid
 - e.g. transaction inputs value == transaction outputs value...
- Constraints on the signers of a transaction
 - e.g. a purchase order must be signed by the buyer...



- Contract Tests
- The Create Command
- **Further Constraints**
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

Design Brainstorm



- What additional constraints should we impose on our IOUs to achieve the desired behaviour?

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- **Further Constraints**
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

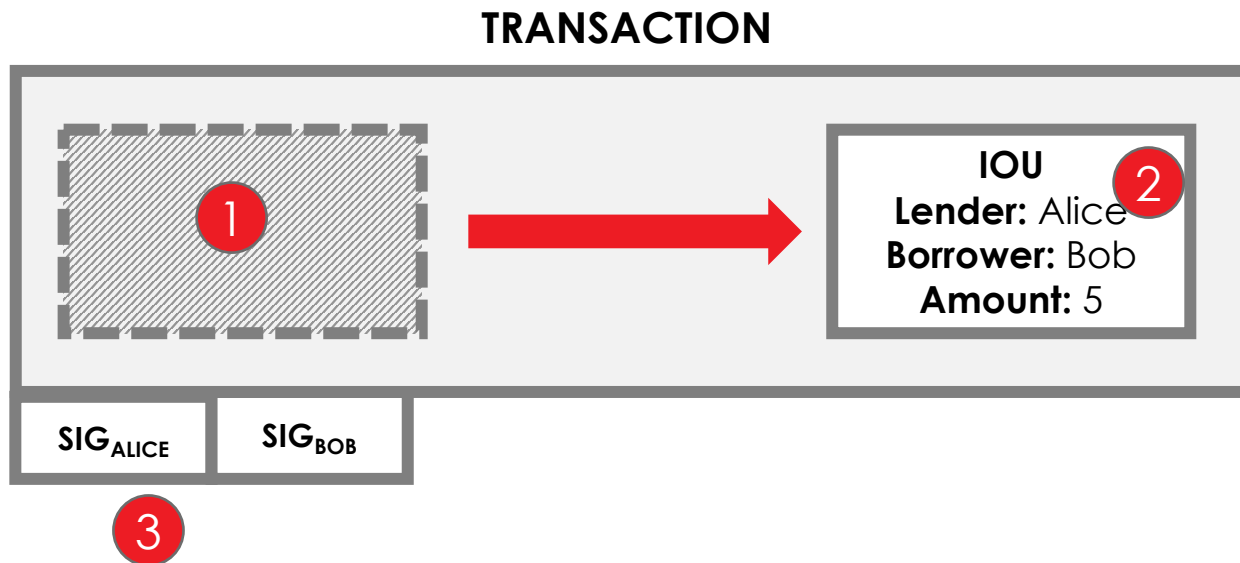
4. Flow

5. Network

6. API

IOU Creation Behavior

- Transactions creating **IOUState** should behave as follows:
 1. No inputs
 2. One output
 3. Signatures from both parties
- **IOUContract** must embody these constraints



1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- **Further Constraints**
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

IOU Creation Constraints

- We can enforce this behaviour with the following using:
 - **mustIncludeIssueCommand**
 - **valueMustBePositive**
 - **transactionMustHaveNoInputs**
i.e. IOUs can be transferred
 - **transactionMustHaveOneOutput**
i.e. only one IOU per transaction
 - **senderMustSignTransaction**
 - **recipientMustSignTransaction**
i.e. both parties must agree to the transaction

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- **Further Constraints**
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

A low-angle, black and white photograph of a modern skyscraper with a complex, angular facade, viewed through a grid of small dots. The building's structure is composed of many sharp angles and lines, creating a sense of height and architectural complexity. The grid of dots is a light gray, semi-transparent overlay that covers the entire image, adding a technical or digital feel to the composition.

Step 3 – Tx-Level Constraints

Transaction-Level Tests

- We need two transaction-level constraints:
 - `issueTransactionMustHaveNoInputs`
 - `issueTransactionMustHaveOneOutput`
- A note on `issueTransactionMustHaveOneOutput`:
 - A mistake would be to test this transaction by passing in no outputs and no inputs
 - With no outputs (and no inputs), there are no states, and thus no contract code to execute, so the transaction can't fail!
 - Instead, we'll test the transaction by giving it two outputs

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- **Tx-Level Constraints**
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

Transaction-Level Constraints - Implementation



Goal	Implement the constraints that transactions must have a single output and no inputs
Where?	contract/IOUContract.kt, inside the verify method test/transactions/IOUIssueTests.kt
Steps	<ol style="list-style-type: none">Uncomment the following tests:<ul style="list-style-type: none">issueTransactionMustHaveNoInputsissueTransactionMustHaveOneOutputRun the testModify IOUContract.kt to make the tests pass
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- **Tx-Level Constraints**
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

Transaction-Level Constraints - Solution



Goal	Constrain the number of inputs (0) and outputs (1) in <code>IOUContract.verify</code>
Steps	<ul style="list-style-type: none">• Test the sizes of the input and output arrays• Make sure the contract error messages match those in the tests
Code	<pre>"No inputs should be consumed when issuing an IOU." using tx.inputs.isEmpty() "Only one output state should be created when issuing an IOU." using (tx.outputs.size == 1)</pre>

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- **Tx-Level Constraints**
- Value Constraints
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

A low-angle, black and white photograph of a modern skyscraper with a complex, angular facade, viewed through a grid of small dots. The building's structure is composed of many sharp angles and lines, creating a sense of height and architectural complexity. The grid of dots is a light gray, semi-transparent overlay that covers the entire image, adding a technical or digital feel to the composition.

Step 4 – Value Constraints

IOU Value Constraint

- We are now going to update our contract code to prevent the creation of negative-valued IOUs
- Constraints are written using the **Requirements** DSL:

```
override fun verify(tx : LedgerTransaction) {  
    requireThat {  
        FAILURE_MSG using BOOLEAN_TEST  
        FAILURE_MSG using BOOLEAN_TEST  
    }  
}
```

- The transaction's inputs and outputs are available as **ContractState** arrays via **tx.inputs** and **tx.outputs**
- The **ContractState** array must then be cast to the actual input/output state type(s)

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- **Value Constraints**
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

IOU Value Constraint - Implementation



r3.

Goal	Impose an “IOU value must be non-negative” constraint
Where?	IOUContract.kt, inside the verify function
Steps	<ul style="list-style-type: none">• Uncomment the cannotCreateZeroValueIOUs test• Run the test• Modify IOUContract.kt to make the test pass:<ul style="list-style-type: none">• Use the syntax on the previous page to create a requireThat block• Retrieve the output ContractState from the transaction• Cast the output to an IOUState• Write a constraint that this output cannot be negatively-valued
Key Docs	N/A

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- **Value Constraints**
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

IOU Value Constraint - Solution



Goal	Impose "IOU value must be non-negative" constraint in IOUContract.verify
Steps	<ul style="list-style-type: none">• Extract the output ContractState and cast it to IOUState• Obtain the IOUState's value using IOUState.amount• Write a failure message matching the message in the test
Code	<pre>override fun verify(tx: TransactionForContract) { "" requireThat { val iou = tx.outputstates.first() as IOUState "A newly issued IOU must have a positive amount." using (iou.amount > Amount(0, iou.amount.token)) } }</pre>

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- **Value Constraints**
- Signer Constraints
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

A low-angle, black and white photograph of a modern skyscraper with a complex, geometric facade, viewed through a grid of small dots. The building's facade is composed of many sharp, angular elements that create a dense, crystalline pattern. The perspective is looking up from the base of the building, making it appear to converge towards a point at the top. The entire image is overlaid with a semi-transparent grid of small, light-colored dots.

Step 5 – Signer Constraints

Signer Tests

- The final constraint is to check for the correct public keys in the transaction:
 - **`lenderAndBorrowerMustSignIssueTransaction`**
- We don't add public keys to transactions directly – we attach them to commands instead

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- **Signer Constraints**
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

Signer Constraints - Implementation



r3.

Goal	Implement the constraints requiring the participants to sign the transaction
Where?	<ul style="list-style-type: none">test/contract/IOUIssueTests.ktcontract/IOUContract.kt
Steps	<ol style="list-style-type: none">Uncomment and run the following test:<ul style="list-style-type: none"><code>lenderAndBorrowerMustSignIssueTransaction</code>The tests should failModify IOUContract.kt to make the tests pass:<ul style="list-style-type: none">Use the <code>Command.signers</code> methodAccess a transaction's participants using <code>tx.participants</code>
Key Docs	N/A

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- **Signer Constraints**
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

Signer Constraints - Solution



Goal	Impose a constraint on the required signatures in <code>IOUContract.verify</code>
Steps	<ul style="list-style-type: none">• Extract the command from the transaction• Compare the command's signers to the transaction's participants
Code	<pre>"Both lender and borrower together only may sign IOU issue transaction." using (command.signers.toSet() == iou.participants.map { it.owningKey }.toSet())</pre>

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- **Signer Constraints**
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API

There's more...

There is one more test to finish – you're on your own!

`LenderAndBorrowerCannotBeTheSame()`

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- **Signer Constraints**
- Another Command
- ✓ Checkpoint

4. Flow

5. Network

6. API



Step 6 – Another Command

The Transfer Command

- IOU creation/evolution is now controlled by a set of rules:
 - Non-zero IOUs only
 - IOUs can only be created (not transferred or destroyed)
 - IOU creation transactions must have:
 - *No inputs*
 - *One output (the new IOU)*
 - IOU creation requires sender and recipient signatures
- Let's write another command, **Transfer**, that will allow the IOU's recipient to transfer it to another party

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- **Another Command**
- ✓ Checkpoint

4. Flow

5. Network

6. API

Design Brainstorm



- What contract constraints should we impose to model the behaviour of transferring an IOU?

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- **Another Command**
- ✓ Checkpoint

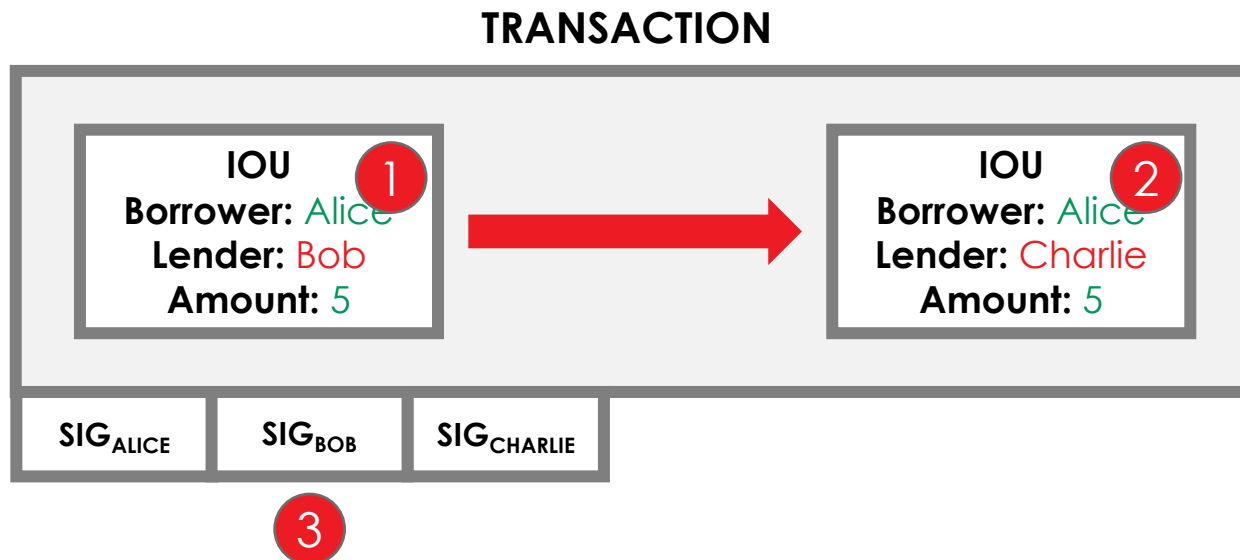
4. Flow

5. Network

6. API

Transfer Command Design

- Transactions transferring **IOUState** should behave as follows:
 - One input
 - One output
 - The amount and borrower should remain the same
 - The lender should be different
 - Signatures from all three parties



r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- Another Command**
- ✓ Checkpoint

4. Flow

5. Network

6. API

Parameterizing Contract Execution

- To implement the **Transfer** command, we need to:
 1. Add a new **CommandData** subclass to **IOUContract**
 2. Fork the execution of **verify** based on the command type
 3. Add the new contract constraints
- We can fork **verify**'s execution using a **when** statement:

```
override fun verify(tx: LedgerTransaction) {  
    val command = tx  
        .commands  
        .requireSingleCommand<IOUContract.Commands>()  
    when (command.value) {  
        is Commands.Issue -> requireThat { }  
        is Commands.Transfer -> { }  
    }  
}
```

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- **Another Command**
- ✓ Checkpoint

4. Flow

5. Network

6. API

Transfer Command - Implementation



Goal	Implement the Transfer command and contract constraints
Where?	<ul style="list-style-type: none">• test/contract/IOUTransferTests.kt• contract/IOUContract.kt
Steps	<ol style="list-style-type: none">1. Uncomment the tests in IOUTransferTests2. Write the code to make the tests pass
Key Docs	N/A

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- **Another Command**
- ✓ Checkpoint

4. Flow

5. Network

6. API

Transfer Command - Solution



Goal	Implement the Transfer command and contract constraints
Steps	<ul style="list-style-type: none">• Define the IOUContract.Transfer class• Define the corresponding constraints
Code	<p>Add the Transfer command to the Commands interface:</p> <pre>interface Commands : CommandData { class Issue : TypeOnlyCommandData(), Commands class Transfer : TypeOnlyCommandData(), Commands }</pre> <p>Add the verify function:</p> <p><i>Over the page...</i></p>

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- **Another Command**
- ✓ Checkpoint

4. Flow

5. Network

6. API

Transfer Command - Solution



Code

```
val command = tx.commands.requireSingleCommand<IOUContract.Commands>()
when (command.value) {
    is Commands.Issue -> requireThat { /* ... */ }
    is Commands.Transfer -> requireThat {
        "An IOU transfer transaction should only consume one input state."
        using (tx.inputs.size == 1)
        "An IOU transfer transaction should only create one output state."
        using (tx.outputs.size == 1)
        val input = tx.inputStates.single() as IOUState
        val output = tx.outputStates.single() as IOUState
        "Only the lender property may change."
        using (input == output.withNewLender(input.lender))
        "The lender property must change in a transfer."
        using (input.lender != output.lender)
        "The borrower, old lender and new lender only must sign an IOU
        transfer transaction"
        using (command.signers.toSet() ==
            (input.participants.map { it.owningKey }.toSet() `union`
            output.participants.map { it.owningKey }.toSet()))
    }
}
```

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- **Another Command**
- ✓ Checkpoint

4. Flow

5. Network

6. API

There are more (advanced) tests to complete!

Check out the tests in:
IOUSettleTests.kt

r3.

1. CorDapp Design

2. State

3. Contract

- Contract Tests
- The Create Command
- Further Constraints
- Tx-Level Constraints
- Value Constraints
- Signer Constraints
- **Another Command**
- ✓ Checkpoint

4. Flow

5. Network

6. API



Checkpoint – Progress So Far

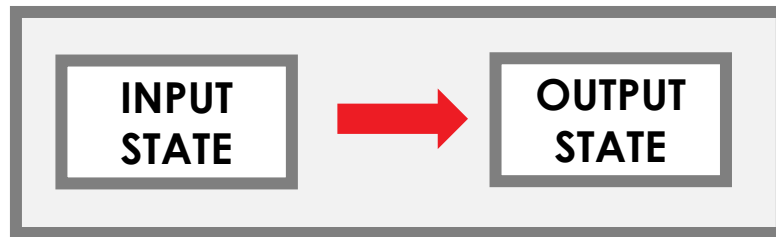
Our progress so far

- We have defined a contract that allows IOU states on the ledger to only evolve in three specific ways:
 - Creation
 - Transfer
 - Settle
- We could further extend the behavior of IOU states by adding additional commands and contract code
- We now need to write the flow that will allow two nodes to speak to each other and agree the creation of IOUs



State grouping

- The simplest way to propose a transaction would be to have zero or one input states and zero or one output states
- This would be easy for the developer, but would prevent many important use cases



State grouping

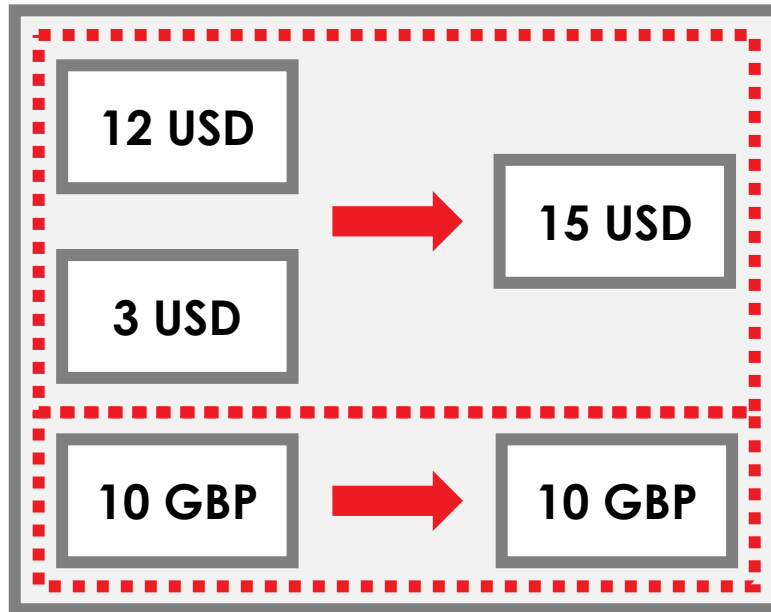
- Another way may be to **iterate over each input state and expect it to have an output state**
- This would make it possible move to two different cash states in different currencies simultaneously
- However, simultaneously dealing with inputs, exits, fungible states (that can split and merge) would make the API overly complex
- There must be another way...

State grouping

- Consider the following simplified currency trade transaction:
 - Input: \$12 owned by Alice
 - Input: \$3 owned by Alice
 - Input: £10 owned by Bob
 - Output: £10 owned by Alice
 - Output: \$15 owned by Bob

State grouping

To verify this transaction, we want to verify two groups of states (the USD states and the GBP states) in isolation:



State grouping

TransactionForContract

has a method which can help:

Where **InOutGroup** is defined as follows:

```
fun <T : ContractState, K : Any>
groupStates(
    ofType: Class<T>,
    selector: (T) -> K
): List<InOutGroup<T, K>>
```

```
data class InOutGroup
<out T : ContractState, out K : Any>(
    val inputs: List<T>,
    val outputs: List<T>,
    val groupingKey: K)
```

State grouping

- Any states for which the selector returns the same value will be placed in the same **InOutGroup**

- In our case, we can use the following grouping function

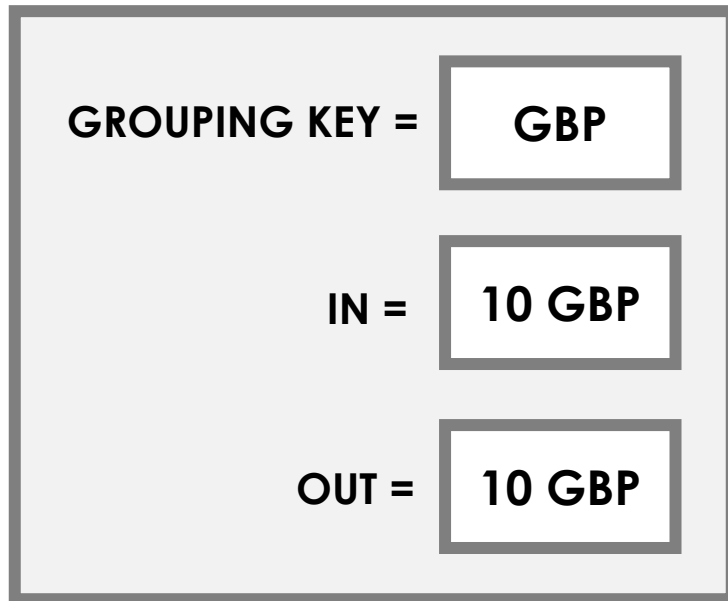
```
val groups = tx.groupStates(Cash.State::class.java) {  
    it -> it.amount.token  
}
```

- Where **amount.token** is the currency of each cash state

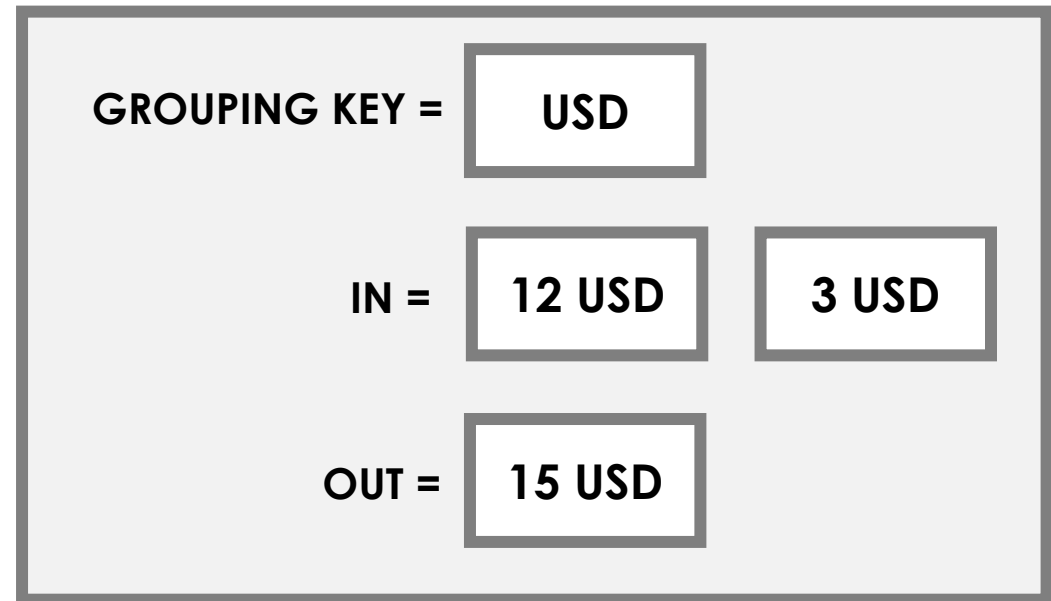
State grouping

`groupStates()` produces the following **InOutGroups**:

INOUTGROUP



INOUTGROUP



State grouping

- You can now apply different verification logic to each group:

```
for ((in, out, key) in groups) {  
    when (key) {  
        is GBP -> { // GBP verification logic. }  
        is USD -> { // USD verification logic. }  
        else -> throw IllegalArgumentException(  
            "Unrecognised currency: $key"  
        )  
    }  
}
```