

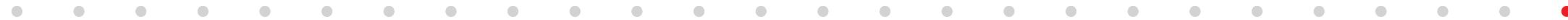
r3.

# Module 6

Contracts



corda



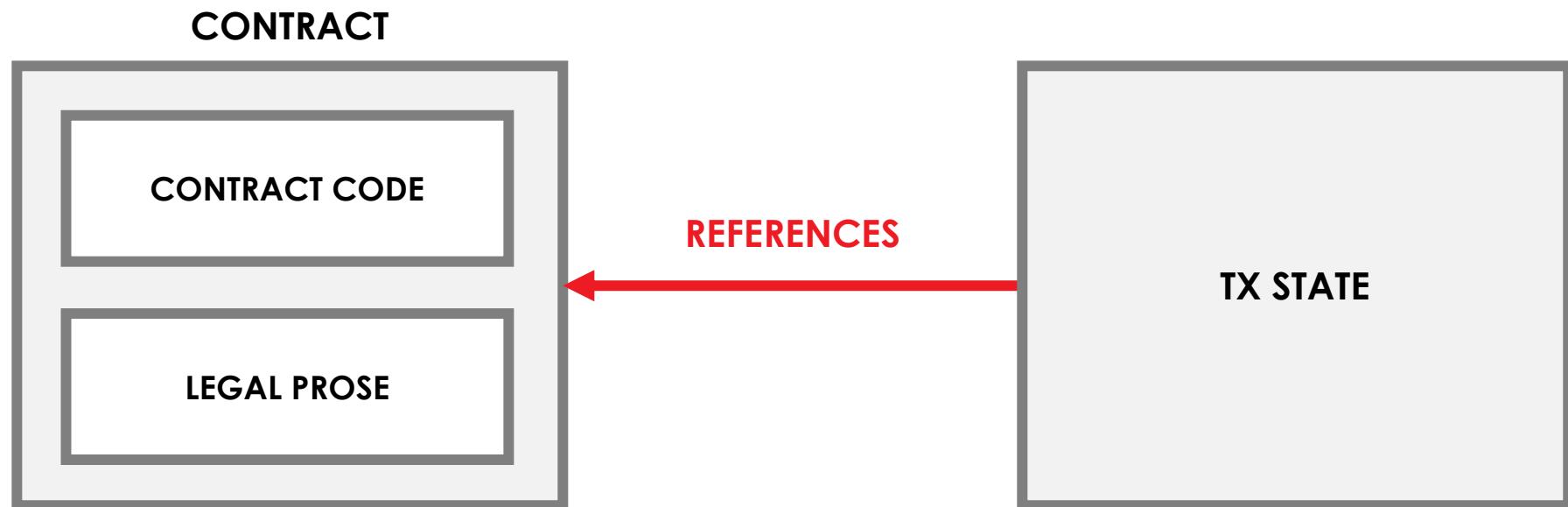
# Learning outcomes

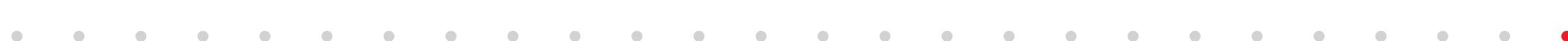
- Learn how contracts control the evolution of states
- Learn how a transaction's states are grouped for verification
- Learn the purpose of commands
- Learn how to design your own contract

r3.

# 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<CommandWithParties<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 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.outputsOfType<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.inputsOfType<NumberState>().single()
        val out = tx.outputsOfType<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))  
                "Issue requires the submitting Party as signer" by (command.signers.contains(issued.matchingState.submittedBy.owner))  
            }  
        }  
        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 **LedgerTransaction**
- Commands provide additional information and are often used to fork the execution of **verify()**
- The **verify()** function can be quite complex

r3.

r<sup>3</sup>

Practical

# IOUContract

1. CorDapp Design

2. State

3. Contract

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

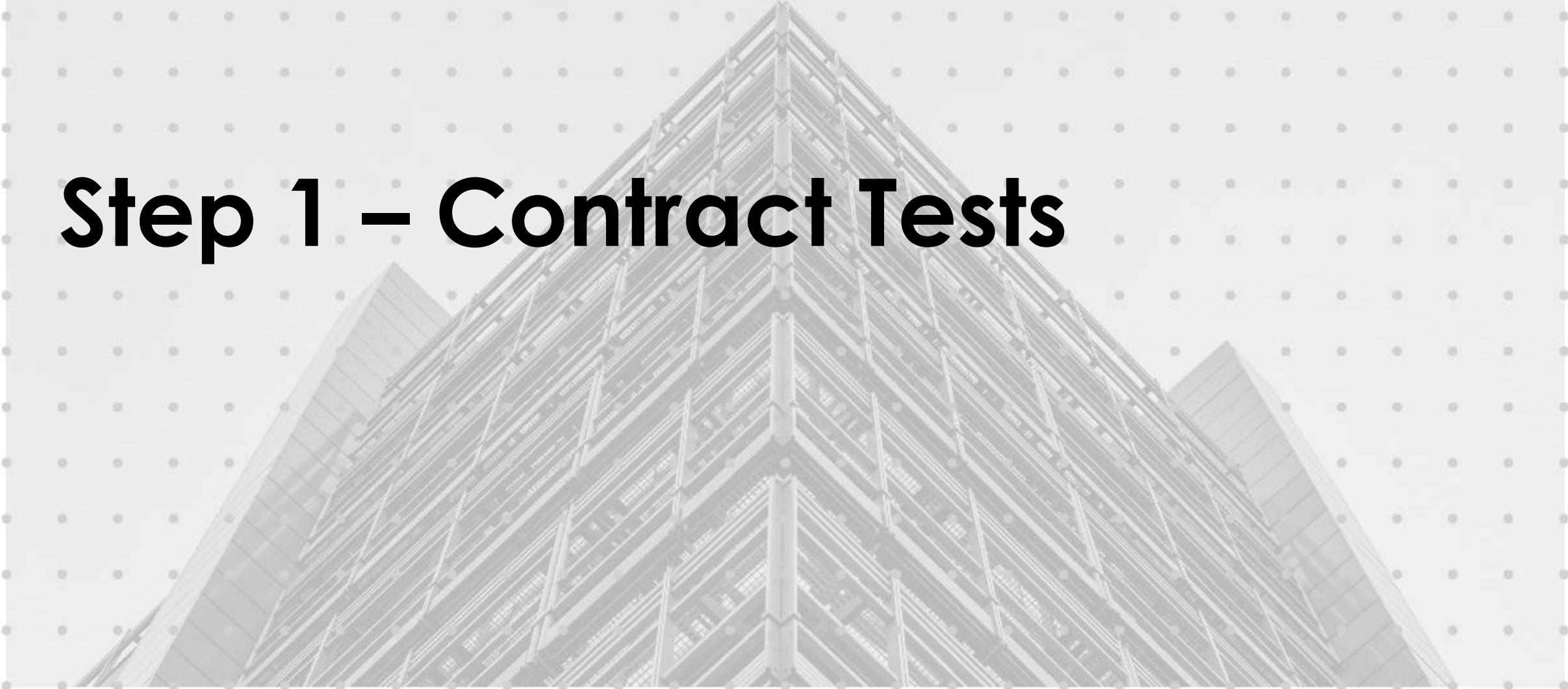
4. Flow

5. Network

6. API



# **Step 1 – Contract Tests**



# 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...**)

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  
    }  
}
```

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



# 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.
}
...
...
```



# **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**

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



<b>Goal</b>	Require the Issue command in valid transactions
<b>Where?</b>	test/contract/IOUissueTests.kt Contract/IOUContract.kt
<b>Steps</b>	<ol style="list-style-type: none"><li>1. Uncomment the <b>mustIncludeIssueCommand</b> test</li><li>2. Run the test using the “Kotlin – IOU Transaction Tests” run config</li><li>3. Modify IOUContract.kt to make the tests pass</li></ol>
<b>Key Docs</b>	N/A



# Adding the Constraint - Solution



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

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 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...

1. CorDapp Design

2. State

3. Contract

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

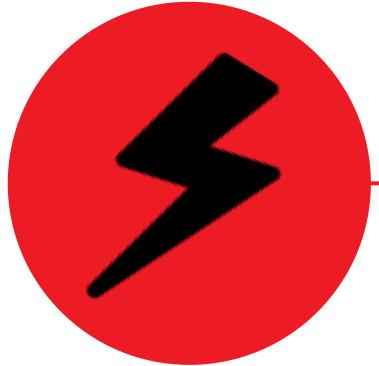
4. Flow

5. Network

6. API



# Design Brainstorm



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

1. CorDapp Design

2. State

3. Contract

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

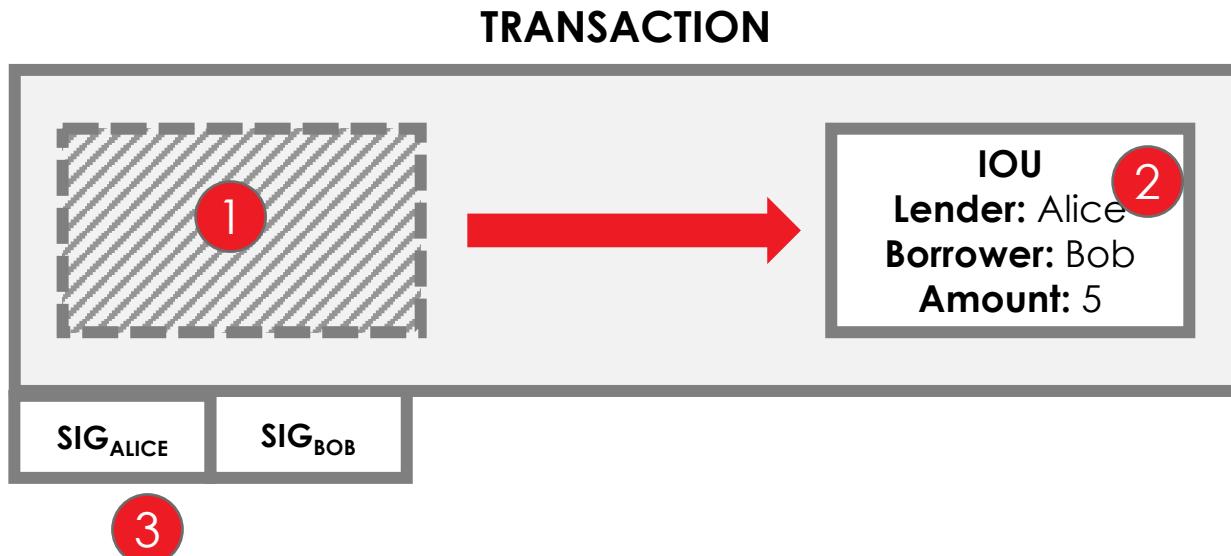
4. Flow

5. Network

6. API

# IOU Creation Behavior

- Transactions creating **IOUState**s 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

1. CorDapp Design

2. State

- 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

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 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

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



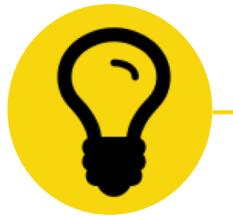
# Transaction-Level Constraints - Implementation



<b>Goal</b>	Implement the constraints that transactions must have a single output and no inputs
<b>Where?</b>	contract/IOUContract.kt, inside the <b>verify</b> method test/transactions/IOUIssueTests.kt
<b>Steps</b>	<ol style="list-style-type: none"><li>1. Uncomment the following tests:<ul style="list-style-type: none"><li>• <b>issueTransactionMustHaveNoInputs</b></li><li>• <b>issueTransactionMustHaveOneOutput</b></li></ul></li><li>2. Run the test</li><li>3. Modify IOUContract.kt to make the tests pass</li></ol>
<b>Key Docs</b>	N/A

r3.

# Transaction-Level Constraints - Solution



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



# **Step 4 – Value Constraints**



# IOU Value Constraint

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

- 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)



# IOU Value Constraint - Implementation



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

r3.

# IOU Value Constraint - Solution

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



<b>Goal</b>	Impose "IOU value must be non-negative" constraint in <b>IOUContract.verify</b>
<b>Steps</b>	<ul style="list-style-type: none"><li>• Extract the output <b>ContractState</b> and cast it to <b>IOUState</b></li><li>• Obtain the <b>IOUState</b>'s value using <b>IOUState.amount</b></li><li>• Write a failure message matching the message in the test</li></ul>
<b>Code</b>	<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 &gt; Amount(0, iou.amount.token))     } }</pre>

r3.

# **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

1. CorDapp Design

2. State

3. Contract

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

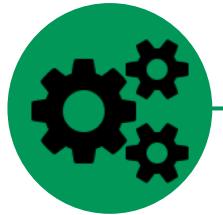
4. Flow

5. Network

6. API



# Signer Constraints - Implementation



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

r3.

# Signer Constraints - Solution



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

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...

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

1. CorDapp Design

2. State

3. Contract

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

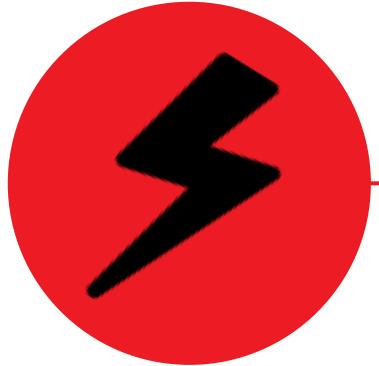
4. Flow

5. Network

6. API



# Design Brainstorm



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

1. CorDapp Design

2. State

3. Contract

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

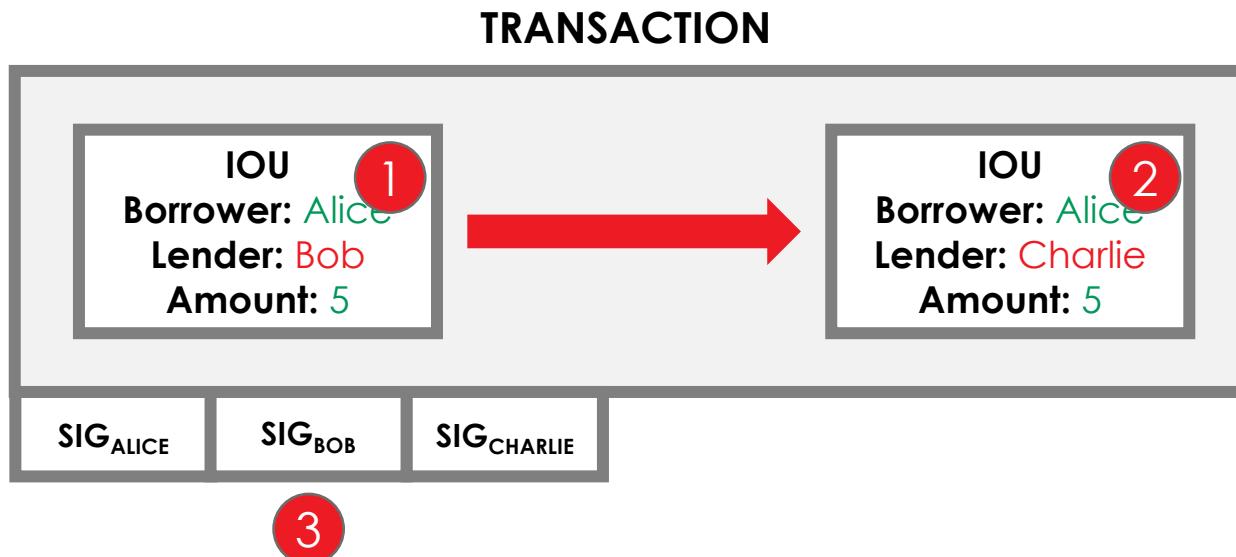
4. Flow

5. Network

6. API

# Transfer Command Design

- Transactions transferring **IOUStates** 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



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

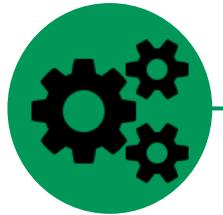
# Parameterizing Contract Execution

- To implement the **Transfer** command, we need to:
  1. Add a new **CommandData** subclass to **I0UContract**
  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<I0UContract.Commands>()  
    when (command.value) {  
        is Commands.Issue -> requireThat { }  
        is Commands.Transfer -> { }  
    }  
}
```



# Transfer Command - Implementation



<b>Goal</b>	Implement the <b>Transfer</b> command and contract constraints
<b>Where?</b>	<ul style="list-style-type: none"><li>test/contract/IOUTransferTests.kt</li><li>contract/IOUContract.kt</li></ul>
<b>Steps</b>	<ol style="list-style-type: none"><li>1. Uncomment the tests in IOUTransferTests</li><li>2. Write the code to make the tests pass</li></ol>
<b>Key Docs</b>	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

# Transfer Command - Solution



<b>Goal</b>	Implement the <b>Transfer</b> command and contract constraints
<b>Steps</b>	<ul style="list-style-type: none"><li>Define the <b>IOUContract.Transfer</b> class</li><li>Define the corresponding constraints</li></ul>
<b>Code</b>	<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 <b>verify</b> function:</p> <p>Over the page...</p>

r3.

# Transfer Command - Solution

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



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.

There are more (advanced) tests to complete!

Check out the tests in:  
**IOUSettleTests.kt**

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

r3.