

Course Agenda

- **Intro to Corda:** Architecture, concepts, components.
- **Getting Started:** Set up dev environment for CorDapps.
- **States:** Model shared facts and agreements on ledger.
- **Contracts:** TDD of smart contracts controlling ledger evolution.
- **Transactions:** Transaction lifecycle, and how transactions propose ledger updates.
- **Flows:** Flow testing framework to develop flows to automate business logic.
- **Corda Node:** Node design, what it can/cannot do, what services/APIs nodes offer.
- **Corda Network:** Network structure and how data flows between nodes.

Alt-Enter <- "unresolved reference"

Ctrl-Z <- "undo"

Ctrl-Shift-Z <- "redo"

Ctrl-Click <- "go to"

Ctrl-/ <- "toggle comment"

Enable Zoom Font (Ctrl-mouse-wheel) -> *File* → *Settings* → *Editor* → *General* → *Change font size (Zoom) with Ctrl+Mouse Wheel*

Editor Colors: *File* | *Settings* | *Editor* | *Color Scheme* | *Color Scheme*
-> *Darcula/Default*

Part 0 - Course Setup

Move previous folders to holding folder.

Download training template into new folder with new name.

(<https://github.com/roger3cev/corda-training-template>)

Open training template folder in IntelliJ.

File | *Project Structure* -> *Project* -> *SDK* -> *1.8 JDK* -> *OK*. (*C:\Program Files\Java\jdk1.8.0_171*)

File | *Project Structure* -> *Modules* -> *+* -> *Import Module* -> *path* -> *OK* -> *Gradle* -> *Next* -> *Finish* -> *OK*.

Wait for build to sync. (*Wait for progress bar in lower right of IntelliJ*)

Build project (*Wait for "Compilation complete successfully" in lower left of IntelliJ*)

Run unit tests -> (*Wait for Test events were not received" in Unit tests panel of IntelliJ*).

Part 1 - IOUStateTests.kt

IOUStateTests.kt -> *uncomment test 1*:

```

/**
 * Task 1.
 * TODO: Add an 'amount' property of type [Amount] to the [IOUState] class to
get this test to pass.
 * Hint: [Amount] is a template class that takes a class parameter of the
token you would like an [Amount] of.
 * As we are dealing with cash lent from one Party to another a sensible token
to use would be [Currency].
 */
@Test
fun hasIOUAmountFieldOfCorrectType() {
    // Does the amount field exist?
    IOUState::class.java.getDeclaredField("amount")
    // Is the amount field of the correct type?
    assertEquals(IOUState::class.java.getDeclaredField("amount").type,
Amount::class.java)
}

```

Run tests -> Tests failed: 1 of 1

IOUState.kt

```

import net.corda.core.contracts.Amount
import java.util.Currency

...

data class IOUState(val amount: Amount<Currency>): ContractState {
    override val participants: List<Party> get() = listOf()
}

```

Run tests -> Tests passed: 1 of 1

IOUStateTests.kt -> uncomment test 2:

```

/**
 * Task 2.
 * TODO: Add a 'lender' property of type [Party] to the [IOUState] class to
get this test to pass.
 */
@Test
fun hasLenderFieldOfCorrectType() {
    // Does the lender field exist?
    IOUState::class.java.getDeclaredField("lender")
    // Is the lender field of the correct type?
    assertEquals(IOUState::class.java.getDeclaredField("lender").type,
Party::class.java)
}

```

IOUState.kt

```
data class IOUState(val amount: Amount<Currency>,
                    val lender: Party): ContractState {
    override val participants: List<Party> get() = listOf()
}
```

Run tests -> Tests passed: 2 of 2

IOUStateTests.kt -> uncomment test 3:

```
/**
 * Task 3.
 * TODO: Add a 'borrower' property of type [Party] to the [IOUState] class to
 * get this test to pass.
 */
@Test
fun hasBorrowerFieldOfCorrectType() {
    // Does the borrower field exist?
    IOUState::class.java.getDeclaredField("borrower")
    // Is the borrower field of the correct type?
    assertEquals(IOUState::class.java.getDeclaredField("borrower").type,
Party::class.java)
}
```

IOUState.kt

```
data class IOUState(val amount: Amount<Currency>,
                    val lender: Party,
                    val borrower: Party): ContractState {
    override val participants: List<Party> get() = listOf()
}
```

Run tests -> Tests passed: 3 of 3

IOUStateTests.kt -> uncomment test 4:

```
/**
 * Task 4.
 * TODO: Add a 'paid' property of type [Amount] to the [IOUState] class to get
 * this test to pass.
 * Hint:
 * - We would like this property to be initialised to a zero amount of
 * Currency upon creation of the [IOUState].
 * - You can use the [POUNDS] extension function over [Int] to create an
 * amount of pounds e.g. '10.POUNDS'.
 * - This property keeps track of how much of the initial [IOUState.amount]
 * has been settled by the borrower
 * - You can initialise a property with a default value in a Kotlin data class
 * like this:
```

```

*
*      data class(val number: Int = 10)
*
* - We need to make sure that the [IOUState.paid] property is of the same
currency type as the
* [IOUState.amount] property. You can create an instance of the [Amount]
class that takes a zero value and a token
* representing the currency - which should be the same currency as the
[IOUState.amount] property.
*/
@Test
fun hasPaidFieldOfCorrectType() {
    // Does the paid field exist?
    IOUState::class.java.getDeclaredField("paid")
    // Is the paid field of the correct type?
    assertEquals(IOUState::class.java.getDeclaredField("paid").type,
Amount::class.java)
}

```

IOUState.kt

```

data class IOUState(val amount: Amount<Currency>,
                    val lender: Party,
                    val borrower: Party,
                    val paid: Amount<Currency> = Amount(0, amount.token)) :
ContractState {
    override val participants: List<Party> get() = listOf()
}

```

Run tests -> Tests passed: 4 of 4

```

/**
* Task 5.
* TODO: Add an entry to the [IOUState.participants] list for the lender.
* Hint: [listOf] takes any number of parameters and will add them to the list
*/
@Test
fun lenderIsParticipant() {
    val iouState = IOUState(1.POUNDS, ALICE.party, BOB.party)
    assertEquals(iouState.participants.indexOf(ALICE.party), -1)
}

```

IOUState.kt

```

data class IOUState(val amount: Amount<Currency>,
                    val lender: Party,
                    val borrower: Party,
                    val paid: Amount<Currency> = Amount(0, amount.token)) :
ContractState {
    override val participants: List<Party> get() = listOf(lender)
}

```

Run tests -> Tests passed: 5 of 5

IOUStateTests.kt -> uncomment test 6:

```
/**
 * Task 6.
 * TODO: Similar to the last task, add an entry to the [IOUState.participants]
 list for the borrower.
 */
@Test
fun borrowerIsParticipant() {
    val iouState = IOUState(1.POUNDS, ALICE.party, BOB.party)
    assertEquals(iouState.participants.indexOf(BOB.party), -1)
}
```

IOUState.kt

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

Run tests -> Tests passed: 6 of 6

IOUStateTests.kt -> uncomment test 7

```
/**
 * Task 7.
 * TODO: Implement [LinearState] along with the required properties and
 methods.
 * Hint: [LinearState] implements [ContractState] which defines an additional
 property and method. You can use
 * IntelliJ to automatically add the member definitions for you or you can add
 them yourself. Look at the definition
 * of [LinearState] for what requires adding.
 */
@Test
fun isLinearState() {
    assert(LinearState::class.java.isAssignableFrom(IOUState::class.java))
}
```

IOUState.kt

```
import net.corda.core.contracts.LinearState
import net.corda.core.contracts.UniqueIdentifier
```

```
...
data class IOUState(val amount: Amount<Currency>,
    val lender: Party,
    val borrower: Party,
    val paid: Amount<Currency> = Amount(0, amount.token),
    override val linearId: UniqueIdentifier =
        UniqueIdentifier(): LinearState {
    override val participants: List<Party> get() = listOf(lender, borrower)
})

```

Run tests -> Tests passed: 7 of 7

IOUStateTests.kt -> uncomment test 8

```
/**
 * Task 8.
 * TODO: Override the [LinearState.linearId] property and assign it a value
 via your state's constructor.
 * Hint:
 * - The [LinearState.linearId] property is of type [UniqueIdentifier]. You
 need to create a new instance of
 * the [UniqueIdentifier] class.
 * - The [LinearState.linearId] is designed to link all [LinearState]s (which
 represent the state of an
 * agreement at a specific point in time) together. All the [LinearState]s
 with the same [LinearState.linearId]
 * represent the complete life-cycle to date of an agreement, asset or shared
 fact.
 * - Provide a default value for [linearId] for a new [IOUState]
 */
@Test
fun hasLinearIdFieldOfCorrectType() {
    // Does the linearId field exist?
    IOUState::class.java.getDeclaredField("linearId")
    // Is the linearId field of the correct type?
    assertEquals(IOUState::class.java.getDeclaredField("linearId").type,
        UniqueIdentifier::class.java)
}

```

IOUState.kt

No code changes required

Run tests -> Tests passed: 8 of 8

IOUStateTests.kt -> uncomment test 9

```
/**
 * Task 9.
 * TODO: Ensure parameters are ordered correctly.
 * Hint: Make sure that the lender and borrower fields are not in the wrong
 order as this may cause some
 * confusion in subsequent tasks!

```

```

*/
@Test
fun checkIOUStateParameterOrdering() {
    val fields = IOUState::class.java.declaredFields
    val amountIdx =
fields.indexOf(IOUState::class.java.getDeclaredField("amount"))
    val lenderIdx =
fields.indexOf(IOUState::class.java.getDeclaredField("lender"))
    val borrowerIdx =
fields.indexOf(IOUState::class.java.getDeclaredField("borrower"))
    val paidIdx = fields.indexOf(IOUState::class.java.getDeclaredField("paid"))
    val linearIdIdx =
fields.indexOf(IOUState::class.java.getDeclaredField("linearId"))

    assert(amountIdx < lenderIdx)
    assert(lenderIdx < borrowerIdx)
    assert(borrowerIdx < paidIdx)
    assert(paidIdx < linearIdIdx)
}

```

IOUState.kt

No code changes required

Run tests -> Tests passed: 9 of 9

IOUStateTests.kt -> uncomment test 10:

```

/**
 * Task 10.
 * TODO: Add a helper method called [pay] that can be called from an
 [IOUState] to settle an amount of the IOU.
 * Hint:
 * - You will need to increase the [IOUState.paid] property by the amount the
 borrower wishes to pay.
 * - Add a new function called [pay] in [IOUState]. This function will need to
 return an [IOUState].
 * - The existing state is immutable so a new state must be created from the
 existing state. Kotlin provides a [copy]
 * method which creates a new object with new values for specified fields.
 * - [copy] returns a copy of the object instance and the fields can be
 changed by specifying new values as
 * parameters to [copy] */
@Test
fun checkPayHelperMethod() {
    val iou = IOUState(10.DOLLARS, ALICE.party, BOB.party)
    assertEquals(5.DOLLARS, iou.pay(5.DOLLARS).paid)
    assertEquals(3.DOLLARS, iou.pay(1.DOLLARS).pay(2.DOLLARS).paid)
    assertEquals(10.DOLLARS,
iou.pay(5.DOLLARS).pay(3.DOLLARS).pay(2.DOLLARS).paid)
}

```

IOUState.kt

```
data class IOUState(val amount: Amount<Currency>,
    val lender: Party,
    val borrower: Party,
    val paid: Amount<Currency> = Amount(0, amount.token),
    override val linearId: UniqueIdentifier =
        UniqueIdentifier(): LinearState {
    override val participants: List<Party> get() = listOf(lender, borrower)
    fun pay(amountPaid: Amount<Currency>) = copy(paid = paid.plus(amountPaid))
})
```

Run tests -> Tests passed: 10 of 10

IOUStateTests.kt -> uncomment test 11:

```
/**
 * Task 11.
 * TODO: Add a helper method called [withNewLender] that can be called from an
 * [IOUState] to change the IOU's lender.
 */
@Test
fun checkWithNewLenderHelperMethod() {
    val iou = IOUState(10.DOLLARS, ALICE.party, BOB.party)
    assertEquals(MINICORP.party, iou.withNewLender(MINICORP.party).lender)
    assertEquals(MEGACORP.party, iou.withNewLender(MEGACORP.party).lender)
}
```

IOUState.kt

```
data class IOUState(val amount: Amount<Currency>,
    val lender: Party,
    val borrower: Party,
    val paid: Amount<Currency> = Amount(0, amount.token),
    override val linearId: UniqueIdentifier =
        UniqueIdentifier(): LinearState {
    override val participants: List<Party> get() = listOf(lender, borrower)
    fun pay(amountPaid: Amount<Currency>) = copy(paid = paid.plus(amountPaid))
    fun withNewLender(newLender: Party) = copy(lender = newLender)
})
```

Run tests -> Tests passed: 11 of 11

Part 2 - IOUIssueTests.kt

IOUIssueTests.kt -> uncomment test 1:

```
/**
 * Task 1.
```



```

* Recall that Commands are required to hint to the intention of the
transaction as well as take a list of
* public keys as parameters which correspond to the required signers for the
transaction.
* Commands also become more important later on when multiple actions are
possible with an IOUState, e.g. Transfer
* and Settle.
* TODO: Add an "Issue" command to the IOUContract and check for the existence
of the command in the verify function.
* Hint:
* - For the create command we only care about the existence of it in a
transaction, therefore it should subclass
* the [TypeOnlyCommandData] class.
* - The command should be defined inside [IOUContract].
* - You can use the [requireSingleCommand] function on [tx.commands] to check
for the existence and type of the specified command
* in the transaction. [requireSingleCommand] requires a generic type to
identify the type of command required.
*
* requireSingleCommand<REQUIRED_COMMAND>()
*
* - We usually encapsulate our commands around an interface inside the
contract class called [Commands] which
* implements the [CommandData] interface. The [Create] command itself
should be defined inside the [Commands]
* interface as well as implement it, for example:
*
* interface Commands : CommandData {
*     class X : TypeOnlyCommandData(), Commands
* }
*
* - We can check for the existence of any command that implements
[IOUContract.Commands] by using the
* [requireSingleCommand] function which takes a type parameter.
*/
@Test
fun mustIncludeIssueCommand() {
    val iou = IOUState(1.POUNDS, ALICE.party, BOB.party)
    ledgerServices.ledger {
        transaction {
            output(IOUContract.IOU_CONTRACT_ID, iou)
            command(listOf(ALICE.publicKey, BOB.publicKey), DummyCommand()) //
Wrong type.
                this.fails()
            }
            transaction {
                output(IOUContract.IOU_CONTRACT_ID, iou)
                command(listOf(ALICE.publicKey, BOB.publicKey),
IOUContract.Commands.Issue()) // Correct type.
                this.verifies()
            }
        }
    }
}

```

IOUContract.kt

```

import net.corda.core.contracts.TypeOnlyCommandData
...
interface Commands : CommandData {
    // Add commands here.
    // E.g
    // class DoSomething : TypeOnlyCommandData(), Commands
    class Issue : TypeOnlyCommandData(), Commands
}

/**
 * The contract code for the [IOUContract].
 * The constraints are self documenting so don't require any additional
 * explanation.
 */
override fun verify(tx: LedgerTransaction) {
    // Add contract code here.
    // requireThat {
    //     ...
    // }
    tx.commands.requireSingleCommand<IOUContract.Commands>()
}

```

Run tests -> Tests passed: 12 of 12

IOUIssueTests.kt -> uncomment test 2:

```

/**
 * Task 2.
 * As previously observed, issue transactions should not have any input state
 * references. Therefore we must check to
 * ensure that no input states are included in a transaction to issue an IOU.
 * TODO: Write a contract constraint that ensures a transaction to issue an
 * IOU does not include any input states.
 * Hint: use a [requireThat] block with a constraint to inside the
 * [IOUContract.verify] function to encapsulate your
 * constraints:
 *
 *     requireThat {
 *         "Message when constraint fails" using (boolean constraint
 * expression)
 *     }
 *
 * Note that the unit tests often expect contract verification failure with a
 * specific message which should be
 * defined with your contract constraints. If not then the unit test will
 * fail!
 *
 * You can access the list of inputs via the [LedgerTransaction] object which
 * is passed into
 * [IOUContract.verify].
 */
@Test
fun issueTransactionMustHaveNoInputs() {
    val iou = IOUState(1.POUNDS, ALICE.party, BOB.party)
}

```

```

    ledgerServices.ledger {
        transaction {
            input(IOUContract.IOU_CONTRACT_ID, DummyState())
            command(listOf(ALICE.publicKey, BOB.publicKey),
                IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, iou)
            this `fails with` "No inputs should be consumed when issuing an
            IOU."
        }
        transaction {
            output(IOUContract.IOU_CONTRACT_ID, iou)
            command(listOf(ALICE.publicKey, BOB.publicKey),
                IOUContract.Commands.Issue())
            this.verify() // As there are no input states.
        }
    }
}

```

IOUContract.kt

```

override fun verify(tx: LedgerTransaction) {
    // Add contract code here.
    // requireThat {
    //     ...
    // }
    tx.commands.requireSingleCommand<IOUContract.Commands>()
    requireThat {
        "No inputs should be consumed when issuing an IOU." using
        (tx.inputStates.isEmpty())
    }
}

```

Run tests -> Tests passed: 13 of 13

IOUIssueTests.kt -> uncomment test 3:

```

/**
 * Task 3.
 * Now we need to ensure that only one [IOUState] is issued per transaction.
 * TODO: Write a contract constraint that ensures only one output state is
    created in a transaction.
 * Hint: Write an additional constraint within the existing [requireThat]
    block which you created in the previous
 * task.
 */
@Test
fun issueTransactionMustHaveOneOutput() {
    val iou = IOUState(1.POUNDS, ALICE.party, BOB.party)
    ledgerServices.ledger {
        transaction {
            command(listOf(ALICE.publicKey, BOB.publicKey),
                IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, iou) // Two outputs fails.
        }
    }
}

```

```

        output(IOUSContract.IOU_CONTRACT_ID, iou)
        this `fails with` "Only one output state should be created when
issuing an IOU."
    }
    transaction {
        command(listOf(ALICE.publicKey, BOB.publicKey),
IOUSContract.Commands.Issue())
        output(IOUSContract.IOU_CONTRACT_ID, iou) // One output passes.
        this.verify()
    }
}

```

IOUSContract.kt

```

override fun verify(tx: LedgerTransaction) {
    // Add contract code here.
    // requireThat {
    //     ...
    // }
    tx.commands.requireSingleCommand<IOUSContract.Commands>()
    requireThat {
        "No inputs should be consumed when issuing an IOU." using
        (tx.inputStates.isEmpty())
        "Only one output state should be created when issuing an IOU." using
        (tx.outputStates.size == 1)
    }
}

```

Run tests -> Tests passed: 14 of 14

IOUSIssueTests.kt -> uncomment test 4:

```

/**
 * Task 4.
 * Now we need to consider the properties of the [IOUSState]. We need to ensure
that an IOU should always have a
 * positive value.
 * TODO: Write a contract constraint that ensures newly issued IOUs always
have a positive value.
 * Hint: You will need a number of hints to complete this task!
 * - Use the Kotlin keyword 'val' to create a new constant which will hold a
reference to the output IOU state.
 * - You can use the Kotlin function [single] to either grab the single
element from the list or throw an exception
 * if there are 0 or more than one elements in the list. Note that we have
already checked the outputs list has
 * only one element in the previous task.
 * - We need to obtain a reference to the proposed IOU for issuance from the
[LedgerTransaction.outputs] list.
 * This list is typed as a list of [ContractState]s, therefore we need to
cast the [ContractState] which we return
 * from [single] to an [IOUSState]. You can use the Kotlin keyword 'as' to

```

```

cast a class. E.g.
*
*      val state = tx.outputStates.single() as XState
*
* - When checking the [IOUState.amount] property is greater than zero, you
need to check the
*      [IOUState.amount.quantity] field.
*/
@Test
fun cannotCreateZeroValueIOUs() {
    ledgerServices.ledger {
        transaction {
            command(listOf(ALICE.publicKey, BOB.publicKey),
IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, IOUState(0.POUNDS, ALICE.party,
BOB.party)) // Zero amount fails.
            this `fails with` "A newly issued IOU must have a positive
amount."
        }
        transaction {
            command(listOf(ALICE.publicKey, BOB.publicKey),
IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, IOUState(100.SWISS_FRANCS,
ALICE.party, BOB.party))
            this.verify()
        }
        transaction {
            command(listOf(ALICE.publicKey, BOB.publicKey),
IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, IOUState(1.POUNDS, ALICE.party,
BOB.party))
            this.verify()
        }
        transaction {
            command(listOf(ALICE.publicKey, BOB.publicKey),
IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, IOUState(10.DOLLARS,
ALICE.party, BOB.party))
            this.verify()
        }
    }
}

```

IOUContract.kt

```

override fun verify(tx: LedgerTransaction) {
    // Add contract code here.
    // requireThat {
    //     ...
    // }
    tx.commands.requireSingleCommand<IOUContract.Commands>()
    requireThat {
        "No inputs should be consumed when issuing an IOU." using
            (tx.inputStates.isEmpty())
        "Only one output state should be created when issuing an IOU." using
            (tx.outputStates.size == 1)
    }
}

```

```

        val outputState = tx.outputStates.single() as IOUState
        "A newly issued IOU must have a positive amount." using
            (outputState.amount.quantity > 0)
    }
}

```

Run tests -> Tests passed: 15 of 15

IOUIssueTests.kt -> uncomment test 5:

```

/**
 * Task 5.
 * For obvious reasons, the identity of the lender and borrower must be
different.
 * TODO: Add a contract constraint to check the lender is not the borrower.
 * Hint:
 * - You can use the [IOUState.lender] and [IOUState.borrower] properties.
 * - This check must be made before the checking who has signed.
 */
@Test
fun lenderAndBorrowerCannotBeTheSame() {
    val iou = IOUState(1.POUNDS, ALICE.party, BOB.party)
    val borrowerIsLenderIou = IOUState(10.POUNDS, ALICE.party, ALICE.party)
    ledgerServices.ledger {
        transaction {
            command(listOf(ALICE.publicKey,
BOB.publicKey), IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, borrowerIsLenderIou)
            this `fails with` "The lender and borrower cannot have the same
identity."
        }
        transaction {
            command(listOf(ALICE.publicKey, BOB.publicKey),
IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, iou)
            this.verify()
        }
    }
}

```

IOUContract.kt

```

override fun verify(tx: LedgerTransaction) {
    // Add contract code here.
    // requireThat {
    //     ...
    // }
    tx.commands.requireSingleCommand<IOUContract.Commands>()
    requireThat {
        "No inputs should be consumed when issuing an IOU." using
            (tx.inputStates.isEmpty())
        "Only one output state should be created when issuing an IOU." using
            (tx.outputStates.size == 1)
    }
}

```

```

        val outputState = tx.outputStates.single() as IOUState
        "A newly issued IOU must have a positive amount." using
            (outputState.amount.quantity > 0)
        "The lender and borrower cannot have the same identity." using
            (outputState.lender != outputState.borrower)
    }
}

```

Run tests -> Tests passed: 16 of 16

IOUIssueTests.kt -> uncomment test 6:

```

/**
 * Task 6.
 * The list of public keys which the commands hold should contain all of the
 * participants defined in the [IOUState].
 * This is because the IOU is a bilateral agreement where both parties
 * involved are required to sign to issue an
 * IOU or change the properties of an existing IOU.
 * TODO: Add a contract constraint to check that all the required signers are
 * [IOUState] participants.
 * Hint:
 * - In Kotlin you can perform a set equality check of two sets with the ==
 * operator.
 * - We need to check that the signers for the transaction are a subset of the
 * participants list.
 * - We don't want any additional public keys not listed in the IOUs
 * participants list.
 * - You will need a reference to the Issue command to get access to the list
 * of signers.
 * - [requireSingleCommand] returns the single required command - you can
 * assign the return value to a constant.
 *
 * Kotlin Hints
 * Kotlin provides a map function for easy conversion of a [Collection] using
 * map
 * - https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections.map.html
 * [Collection] can be turned into a set using toSet()
 * -
 * https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections.to-set.html
 */
@Test
fun lenderAndBorrowerMustSignIssueTransaction() {
    val iou = IOUState(1.POUNDS, ALICE.party, BOB.party)
    ledgerServices.ledger {
        transaction {
            command(DUMMY.publicKey, IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, iou)
            this `fails with` "Both lender and borrower together only may sign
IOU issue transaction."
        }
        transaction {
            command(ALICE.publicKey, IOUContract.Commands.Issue())
            output(IOUContract.IOU_CONTRACT_ID, iou)
        }
    }
}

```

```

        this `fails with` "Both lender and borrower together only may sign
IOU issue transaction."
    }
    transaction {
        command(BOB.publicKey, IOUContract.Commands.Issue())
        output(IOUContract.IOU_CONTRACT_ID, iou)
        this `fails with` "Both lender and borrower together only may sign
IOU issue transaction."
    }
    transaction {
        command(listOf(BOB.publicKey, BOB.publicKey, BOB.publicKey),
IOUContract.Commands.Issue())
        output(IOUContract.IOU_CONTRACT_ID, iou)
        this `fails with` "Both lender and borrower together only may sign
IOU issue transaction."
    }
    transaction {
        command(listOf(BOB.publicKey, BOB.publicKey, MINICORP.publicKey,
ALICE.publicKey), IOUContract.Commands.Issue())
        output(IOUContract.IOU_CONTRACT_ID, iou)
        this `fails with` "Both lender and borrower together only may sign
IOU issue transaction."
    }
    transaction {
        command(listOf(BOB.publicKey, BOB.publicKey, BOB.publicKey,
ALICE.publicKey), IOUContract.Commands.Issue())
        output(IOUContract.IOU_CONTRACT_ID, iou)
        this.verify()
    }
    transaction {
        command(listOf(ALICE.publicKey,
BOB.publicKey), IOUContract.Commands.Issue())
        output(IOUContract.IOU_CONTRACT_ID, iou)
        this.verify()
    }
}
}

```

IOUContract.kt

```

override fun verify(tx: LedgerTransaction) {
    // Add contract code here.
    // requireThat {
    //     ...
    // }
    val command = tx.commands.requireSingleCommand<IOUContract.Commands>()
    requireThat {
        "No inputs should be consumed when issuing an IOU." using
            (tx.inputStates.isEmpty())
        "Only one output state should be created when issuing an IOU." using
            (tx.outputStates.size == 1)
        val outputState = tx.outputStates.single() as IOUState
        "A newly issued IOU must have a positive amount." using
            (outputState.amount.quantity > 0)
        "The lender and borrower cannot have the same identity." using
            (outputState.lender != outputState.borrower)
    }
}

```



```

        "Both lender and borrower together only may sign IOU issue
transaction." using
            ( command.signers.toSet() == outputState.participants.map
              {it.owningKey}.toSet())
    }
}

```

Run tests -> Tests passed: 17 of 17

Part 3 - IOUIssueFlowTests.kt

IOUIssueFlowTests.kt -> uncomment test 1:

```

/**
 * Task 1.
 * Build out the [IOUIssueFlow]!
 * TODO: Implement the [IOUIssueFlow] flow which builds and returns a
partially [SignedTransaction].
 * Hint:
 * - There's a lot to do to get this unit test to pass!
 * - Create a [TransactionBuilder] and pass it a notary reference.
 * -- A notary [Party] object can be obtained from
[FlowLogic.serviceHub.networkMapCache].
 * -- In this training project there is only one notary
 * - Create an [IOUContract.Commands.Issue] inside a new [Command].
 * -- The required signers will be the same as the state's participants
 * -- Add the [Command] to the transaction builder [addCommand].
 * - Use the flow's [IOUState] parameter as the output state with
[addOutputState]
 * - Extra credit: use [TransactionBuilder.withItems] to create the
transaction instead
 * - Sign the transaction and convert it to a [SignedTransaction] using the
[serviceHub.signInitialTransaction] method.
 * - Return the [SignedTransaction].
 */
@Test
fun flowReturnsCorrectlyFormedPartiallySignedTransaction() {
    val lender = a.info.chooseIdentityAndCert().party
    val borrower = b.info.chooseIdentityAndCert().party
    val iou = IOUState(10.POUNDS, lender, borrower)
    val flow = IOUIssueFlow(iou)
    val future = a.startFlow(flow)
    mockNetwork.runNetwork()
    // Return the unsigned(!) SignedTransaction object from the IOUIssueFlow.
    val ptx: SignedTransaction = future.getOrThrow()
    // Print the transaction for debugging purposes.
    println(ptx.tx)
    // Check the transaction is well formed...
    // No outputs, one input IOUState and a command with the right properties.
    assert(ptx.tx.inputs.isEmpty())
    assert(ptx.tx.outputs.single().data is IOUState)
    val command = ptx.tx.commands.single()
    assert(command.value is IOUContract.Commands.Issue)
}

```

```

        assert(command.signers.toSet() == iou.participants.map { it.owningKey
    }.toSet())
    ptx.verifySignaturesExcept(borrower.owningKey,

mockNetwork.defaultNotaryNode.info.legalIdentitiesAndCerts.first().owningKey)
}

```

IOUIssueFlow.kt

```

class IOUIssueFlow(val state: IOUState) : FlowLogic<SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {
        val notaryParty = serviceHub.networkMapCache.notaryIdentities.first()
        val builder = TransactionBuilder(notaryParty)
        val command = Command(IOUContract.Commands.Issue(),
            state.participants.map { it.owningKey })
        builder.addCommand(command)
        builder.addOutputState(state, IOUContract.IOU_CONTRACT_ID)
        val signedTx = serviceHub.signInitialTransaction(builder)
        return signedTx
    }
}

```

Run tests -> Tests passed: 18 of 18

IOUIssueFlowTests.kt -> uncomment test 2:

```

/**
 * Task 2.
 * Now we have a well formed transaction, we need to properly verify it using
 the [IOUContract].
 * TODO: Amend the [IOUIssueFlow] to verify the transaction as well as sign
 it.
 */
@Test
fun flowReturnsVerifiedPartiallySignedTransaction() {
    // Check that a zero amount IOU fails.
    val lender = a.info.chooseIdentityAndCert().party
    val borrower = b.info.chooseIdentityAndCert().party
    val zeroIou = IOUState(0.POUNDS, lender, borrower)
    val futureOne = a.startFlow(IOUIssueFlow(zeroIou))
    mockNetwork.runNetwork()
    assertFailsWith<TransactionVerificationException> { futureOne.getOrThrow()
}

    // Check that an IOU with the same participants fails.
    val borrowerIsLenderIou = IOUState(10.POUNDS, lender, lender)
    val futureTwo = a.startFlow(IOUIssueFlow(borrowerIsLenderIou))
    mockNetwork.runNetwork()
    assertFailsWith<TransactionVerificationException> { futureTwo.getOrThrow()
}

    // Check a good IOU passes.
    val iou = IOUState(10.POUNDS, lender, borrower)
    val futureThree = a.startFlow(IOUIssueFlow(iou))
    mockNetwork.runNetwork()
}

```

```
futureThree.getOrThrow()
}
```

IOUIssueFlow.kt

```
class IOUIssueFlow(val state: IOUState) : FlowLogic<SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {
        val notaryParty = serviceHub.networkMapCache.notaryIdentities.first()
        val builder = TransactionBuilder(notaryParty)
        val command = Command(IOUContract.Commands.Issue(),
            state.participants.map { it.owningKey })
        builder.addCommand(command)
        builder.addOutputState(state, IOUContract.IOU_CONTRACT_ID)
        builder.verify(serviceHub)
        val signedTx = serviceHub.signInitialTransaction(builder)
        return signedTx
    }
}
```

Run tests -> Tests passed: 19 of 19

IOUIssueFlowTests.kt -> uncomment test 3:

```
/**
 * IMPORTANT: Review the [CollectSignaturesFlow] before continuing here.
 * Task 3.
 * Now we need to collect the signature from the [otherParty] using the
 [CollectSignaturesFlow].
 * TODO: Amend the [IOUIssueFlow] to collect the [otherParty]'s signature.
 * Hint:
 * On the Initiator side:
 * - Get a set of signers required from the participants who are not the node
 * - - [ourIdentity] will give you the identity of the node you are operating
 as
 * - Use [initateFlow] to get a set of [FlowSession] objects
 * - - Using [state.participants] as a base to determine the sessions needed
 is recommended. [participants] is on
 * - - the state interface so it is guaranteed to exist where [lender] and
 [borrower] are not.
 * - - Hint: [ourIdentity] will give you the [Party] that represents the
 identity of the initiating flow.
 * - Use [subFlow] to start the [CollectSignaturesFlow]
 * - Pass it a [SignedTransaction] object and [FlowSession] set
 * - It will return a [SignedTransaction] with all the required signatures
 * - The subflow performs the signature checking and transaction verification
 for you
 *
 * On the Responder side:
 * - Create a subclass of [SignTransactionFlow]
 * - Override [SignTransactionFlow.checkTransaction] to impose any constraints
 on the transaction
 *
 * Using this flow you abstract away all the back-and-forth communication
```

```

required for parties to sign a
* transaction.
*/
@Test
fun flowReturnsTransactionSignedByBothParties() {
    val lender = a.info.chooseIdentityAndCert().party
    val borrower = b.info.chooseIdentityAndCert().party
    val iou = IOUState(10.POUNDS, lender, borrower)
    val flow = IOUIssueFlow(iou)
    val future = a.startFlow(flow)
    mockNetwork.runNetwork()
    val stx = future.getOrThrow()
    stx.verifyRequiredSignatures()
}

```

IOUIssueFlow.kt

```

class IOUIssueFlow(val state: IOUState) : FlowLogic<SignedTransaction>() {
    @Suspendable
    override fun call(): SignedTransaction {
        val notaryParty = serviceHub.networkMapCache.notaryIdentities.first()
        val command = Command(IOUContract.Commands.Issue(),
            state.participants.map { it.owningKey })
        val builder = TransactionBuilder(notaryParty)
        builder.addCommand(command)
        builder.addOutputState(state, IOUContract.IOU_CONTRACT_ID)
        builder.verify(serviceHub)
        val signedTx = serviceHub.signInitialTransaction(builder)
        val sessions = (state.participants - ourIdentity)
            .map { initiateFlow(it) }.toSet()
        val allSignedTx = subFlow(CollectSignaturesFlow(
            signedTx, sessions))
        return allSignedTx
    }
}

```

Run tests -> Tests passed: 20 of 20

IOUIssueFlowTests.kt -> uncomment test 4:

```

/**
 * Task 4.
 * Now we need to store the finished [SignedTransaction] in both counter-party
 vaults.
 * TODO: Amend the [IOUIssueFlow] by adding a call to [FinalityFlow].
 * Hint:
 * - As mentioned above, use the [FinalityFlow] to ensure the transaction is
 recorded in both [Party] vaults.
 * - Do not use the [BroadcastTransactionFlow]!
 * - The [FinalityFlow] determines if the transaction requires notarisation or
 not.
 * - We don't need the notary's signature as this is an issuance transaction
 without a timestamp. There are no
 * inputs in the transaction that could be double spent! If we added a
 timestamp to this transaction then we

```

```
* would require the notary's signature as notaries act as a timestamping authority.
```

```
*/  
@Test  
fun flowRecordsTheSameTransactionInBothPartyVaults() {  
    val lender = a.info.chooseIdentityAndCert().party  
    val borrower = b.info.chooseIdentityAndCert().party  
    val iou = IOUState(10.POUNDS, lender, borrower)  
    val flow = IOUIssueFlow(iou)  
    val future = a.startFlow(flow)  
    mockNetwork.runNetwork()  
    val stx = future.getOrThrow()  
    println("Signed transaction hash: ${stx.id}")  
    listOf(a, b).map {  
        it.services.validatedTransactions.getTransaction(stx.id)  
    }.forEach {  
        val txHash = (it as SignedTransaction).id  
        println("$txHash == ${stx.id}")  
        assertEquals(stx.id, txHash)  
    }  
}
```

IOUIssueFlow.kt

```
class IOUIssueFlow(val state: IOUState) : FlowLogic<SignedTransaction>() {  
    @Suspendable  
    override fun call(): SignedTransaction {  
        val notaryParty = serviceHub.networkMapCache.notaryIdentities.first()  
        val command = Command(IOUContract.Commands.Issue(),  
            state.participants.map { it.owningKey })  
        val builder = TransactionBuilder(notaryParty)  
        builder.addCommand(command)  
        builder.addOutputState(state, IOUContract.IOU_CONTRACT_ID)  
        builder.verify(serviceHub)  
        val signedTx = serviceHub.signInitialTransaction(builder)  
        val sessions = (state.participants - ourIdentity)  
            .map { initiateFlow(it) }.toSet()  
        val allSignedTx = subFlow(CollectSignaturesFlow(signedTx, sessions))  
        return subFlow(FinalityFlow(allSignedTx))  
    }  
}
```

Run tests -> Tests passed: 21 of 21
