Tutorial 3: Selling, buying, and trading

With the first two tutorials complete, all the basic elements of Fae necessary to construct an economy have been presented. However, as we saw at the end of Tutorial 2, assembling them into a secure interface that protects value requires the application of idioms that transcend their coding elements. There, we saw the idiom for encapsulating a valuable as a controlled interface to a module containing its implementation, similar to the object-oriented pattern of coding. Actually exchanging these valuables securely requires another object-oriented idiom in the code as well as a meta-transactional bit of logic between the participants.

As in previous tutorials, the sample code will use the conventions of the postTX utility and the sample command lines use the docker images. After following the instructions there, you should execute this first to have a local server running:

```
./faeServer.sh
```

A `Nametag` factory

At the end of Tutorial 2, we saw an example of claiming and storing a "valuable" object, a `Nametag`, so that the owner had possession of it both in the sense of holding the escrow in a contract, and also restricting access to that contract to themselves. The transaction that produced the nametag, however, was rigidly individualized by hard-coding my name, Ryan Reich, as the name on the tag. Though anyone could theoretically duplicate this with the appropriate substitution, it is not possible (as with import statements) to expand an environment variable in the code, so it would have to be true duplication, which is poor practice. In addition, it was a little arbitrary that the `Nametag` module was, itself, only associated with a random particular transaction that also claimed one for the first time. We begin this tutorial with a more organized installation of the interface.

Module source file: Nametag.hs

```
module Nametag (Nametag, getNametag, checkNametag) where

import Blockchain.Fae
import Control.Monad (forever)

data NametagName = NametagName String deriving (Generic)
newtype Nametag = Nametag (EscrowID NametagName) deriving (Generic)

instance ContractName NametagName where
  type ArgType NametagName = ()
  type ValType NametagName = String
  theContract (NametagName name) = \_ -> forever $ release $ "Property of: " ++ name

getNametag :: (MonadTX m) => Reward -> String -> m Nametag
getNametag rwd name = do
  claimReward rwd
  Nametag <$> newEscrow (NametagName name)

checkNametag :: (MonadTX m) => Nametag -> m String
checkNametag (Nametag eID) = useEscrow [] eID ()
```

We reproduce the `Nametag` module from Tutorial 2.

Transaction source file: InstallNametag.hs

```
import Nametag

body :: FaeTX ()
body = newContract OfferNametag

data OfferNametag = OfferNametag deriving (Generic)

instance ContractName OfferNametag where
  type ArgType OfferNametag = String
  type ValType OfferNametag = EscrowID GetNametag
  theContract OfferNametag = \name -> do
    eID <- newEscrow $ GetNametag name
    release eID >>= theContract OfferNametag

data GetNametag = GetNametag String deriving (Generic)

instance ContractName GetNametag where
  type ArgType GetNametag = Reward
  type ValType GetNametag = Nametag
  theContract (GetNametag name) = \rwd -> do
    nt <- getNametag rwd name
    release nt >>= theContract (GetNametag name)
```

This transaction introduces two new contract names, one of which uses the other to create an escrow. These are both public in the sense that other transaction authors might later use them as well, which although pointless in the direct sense, might be useful for constructing more elaborate contracts.

Transaction file: InstallNametag

```
body = InstallNametag
others
  - Nametag
```

Command line:

```
./postTX.sh InstallNametag
```

Transaction results:

```
Transaction 9d6bfb03a0fe6dccc9ebb350830ae9d5156fd947d6c60501272da8f3a9ba2523
  result: ()
  outputs:
    0: 9d6bfb03a0fe6dccc9ebb350830ae9d5156fd947d6c60501272da8f3a9ba2523
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
```

HOW IT WORKS

There are no new syntactic features in this transaction, but it is somewhat more complex than previous ones because it creates a contract that, when called, (repeatedly) creates an escrow that, when called, (repeatedly) cashes in a

`Reward` to create a `Nametag`. What does this accomplish?

The `GetNametag` contract name is a `Nametag` factory, specifically one that endlessly produces nametags with the particular name provided to it upon creation. For the owner of a `GetNametag` escrow, it functions as their personal storefront of nametags describing themselves. Although this sounds reasonable, it raises the question why such a storefront could not be created as a contract directly.

The answer to that question is simply that *contracts cannot accept valuables as arguments*. This is a direct consequence of how they are called: by textual, literal arguments given in the transaction message. A valuable cannot be parsed from plain text because there is no scarcity in this process and the result would not actually *be* valuable. There is also no way of taking a valuable, say a `Reward`, that appears as a transaction `body`'s argument, and pass it as the argument to one of the input contracts. Contracts are *referentially transparent*: they can only act according to the overt form of the input, without secret contributions such as the presence of an escrow backing some valuable.

We use `GetNametag` as the agent for a `Nametag` store; it accepts the payments and delivers the products. This agent can be obtained from a contract that, itself, only acts upon plain data, namely the name that this particular agent will put on its nametags. This contract is, of course, the `OfferNametag` contract, which only needs to be created once and can produce agents for any interested parties that call it.

Running the `Nametag` factory

Here is an example of actually using the contracts above to get nametags.

Transaction source file: RunNametagFactory.hs

```
import Blockchain.Fae.Contracts
import Blockchain.Fae.Transactions.TX$txID

body :: Reward -> EscrowID GetNametag -> FaeTX ()
body rwd ntFactory = do
  nt <- useEscrow [] ntFactory rwd
  deposit nt "self"
```

Transaction message: RunNametagFactory

```
body = RunNametagFactory
reward = True
inputs
  $txID/Body/0/Current = "$name"
keys
  self = $name
```

Command line:

```
./postTX.sh \
  -e txID=<InstallNametag id> \
  -e name=<your choice> \
  RunNametagFactory
```

A simple exchange

As part of this game of collecting personalized nametags, several people may wish to trade them, perhaps so that one individual can prove themselves to be popular and another, to be associated with someone popular. Let's say that two such people, Ryan and Sarah, each have a nametag obtained by running `RunNametagFactory` with their own names, with the following results:

Transaction `RunNametagFactory` "ryan" results

```
Transaction 68da7c58dc105ddbadd1d0af541d69d2d8f2f72df8b80d73e63f13e0b2680c0b
  result: ()
  outputs:
    0: 3dcad91bf3427384e92c003e0249bf95353e2f18a676b00b9df21401a8b6155b
  signers:
    self: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
  input #0: b1a6.../Body/0/Current (Updated)
    version: b1a6ab2c0840f35fb70bd1019bc960f33d389582046f92cfd75d4b300bed0875
```

Transaction `RunNametagFactory` "sarah" results

```
Transaction 4d312001a599fc42a23c8483b8a6ecd6e443e0aa8ef007e9a5d59949b41530e2
  result: ()
  outputs:
    0: 742bc2901d3ef5e21ada485a3aaed3e60b00de11c9def7beef8c21d1333aec89
  signers:
    self: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
  input #0: b1a6.../Body/0/Current (Updated)
    version: a7c0318fe456d68a7f629cbca6a3487eb0b93ed96c17dfdaae95ca3965631e79
```

Note the public keys of the signers of each transaction as well as the two transaction IDs in order to verify the effects of the following transaction that trades the nametags.

Transaction source file: TradeNametags.hs

```
import Blockchain.Fae.Contracts (deposit)
import Blockchain.Fae.Transactions.TX$installTXID.Nametag

body :: Nametag -> Nametag -> FaeTX (String, String)
body nt1 nt2 = do
  message1 <- checkNametag nt1
  message2 <- checkNametag nt2
  deposit nt1 "person2"
  deposit nt2 "person1"
  return (message1, message2)
```

This transaction, which is entirely generic and can trade nametags between any two owners regardless of their names, deposits them in the *opposite* order they were received in, and prints their messages in the order of the inputs to prove that they had the expected owners. Note that we use `$installTXID` in the import statement; this refers to the ID of `InstallNametag`, which uploaded the `Nametag` module.

Transaction message: TradeNametags

```
body = TradeNametags
inputs
```

```
  $tx1ID/Body/0/Current = ()
    self = person1
  $tx2ID/Body/0/Current = ()
    self = person2
keys
  person1 = $person1
  person2 = $person2
```

Here, `$tx1ID` and `$tx2ID` are the transactions in which `$person1` and `$person2` deposited their nametags into the contracts being called.

Command line:

```
./postTX.sh \
  -e installTXID=<InstallNametag ID> \
  -e tx1ID=<RunNametagFactory "ryan" ID> \
  -e tx2ID=<RunNametagFactory "sarah" ID> \
  -e person1=ryan \
  -e person2=sarah \
  TradeNametags
```

This is the *only* place where `ryan` and `sarah` are mentioned; clearly, the rest of the transaction is fully reusable.

Transaction results:

```
Transaction 999cb0ebc145354e835f6bd264828b002b94dc5ebb226a885e2652b96b8d1683
  result: ("Property of: ryan","Property of: sarah")
  outputs:
    0: 999cb0ebc145354e835f6bd264828b002b94dc5ebb226a885e2652b96b8d1683
    1: b77e3f33ffab41f79be1a668799670ae40b4f5c7b1c822d6e3039612f7163ab9
  signers:
    person1: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
    person2: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
  input #0: 68da.../Body/0/Current (Deleted)
  input #1: 4d31.../Body/0/Current (Deleted)
```

Compare the public keys of `person1` and `person2` to those of `ryan` and `sarah` in the calls to `RunNametagFactory`; they are the same.

RENAMING

There is a single new syntax element here, and it is important: each input call runs with the signer role `self` set to a *different* one of the actual signers. The lines `self = person1` and `self = person2` mean "in this call, create a new name called `self` and assign it whatever corresponds to the existing name `person1` (or `person2`)." This is crucial because the `deposit` contract, as defined in this example in Tutorial 2, *always* checks the `self` signer role for the key to be validated against the owner. Since it is called twice in `TradeNametags`, without renaming it would be impossible for both Ryan and Sarah to withdraw their nametags.

(An alternative design decision would have been possible: simply have the `deposit` contract, the `theContract` of `Restricted` in Tutorial 2, take as its argument the name of the role to check. I opted against this because the logic to check an alternative name is completely boilerplate, and therefore the only contribution of passing it off on the user is the possibility of an error in implementing it; in addition, it makes the argument of every

contract that uses it more complicated and also reduces the flexibility of contracts for which it is only realized after they are created that such a renaming scheme is necessary.)

The renaming is *local*; in `input #0`, `self` is the public key of `ryan`; in `input #1` it is that of `sarah`; and in the transaction `body` itself, there is *no* role called `self`.

Renaming is also available for escrow calls within contract or transaction code. It is declared using the previously-mysterious list that is the first argument of `useEscrow`; this list can contain elements of the form `newName <-| oldName`, as in the `inputs` list of the transaction message except for the use of the funny operator `<-|`, which can also be written in UTF-8 as `↤`, the symbol often used in computer science to indicate variable assignment (I couldn't use `=`, as that is taken; also, this syntax more clearly describes what gets assigned to what).

HOW IT WORKS

This transaction creates a swap between two parties that is subject to the veto of either party. Yet, as written it appears to run without actually checking any kind of permission from either one! In fact, `TradeNametags.hs` itself makes no mention of this issue; it simply performs the oblivious logic of verifying that the nametags are valid and re-depositing them.

The permissions are actually contained in the transaction message, `TradeNametags`, in the form of the two signatures indicated under `keys`. In this demo example, the same `postTX` tracks both `ryan` and `sarah`'s private keys; however, this is a proxy for the logic that should actually be understood in a live run, where each one has a protected account (or “wallet” in the terminology of cryptocurrencies) requiring their authorization to open. The fact that this transaction can *be written correctly* is already proof that both parties have consented.

Suppose that the transaction were not correctly signed, as if `sarah` tried to write and send it without `ryan`'s knowledge, effectively to steal his nametag, which we can simulate by using `person1=sarah` in the command line. This produces a rather different transaction result:

```
Transaction aa4e4114835a8260337558cf924898b96842a97b31351a0a358b89c7aa3d7a55
  result: <exception> Signer with public key 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2
  signers:
    person1: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
    person2: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
  input #0: 6ea2.../Body/0/Current (Failed)
    <exception> Signer with public key 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb
  input #1: 814d.../Body/0/Current (Deleted)
```

The first `deposit` contract rejected the attempt to withdraw and, therefore, the call to `checkNametag nt1` threw the exception first raised by that contract. This is the essence of why this function is essential for validation: not only does it protect against forgery, which was its motivation in Tutorial 2, but it also protects against theft. By performing this check on the return value of a restricted-access contract we enforce the restriction before finalizing the change in ownership.

(You might notice that if Sarah is going to go rogue, she can also write a transaction without calling `checkNametag`. This is true, and then she would end up with a `deposit` contract containing, in effect, a time bomb. The `Nametag` it contains is not actually valid, but that hasn't been proven yet. Eventually, in order to benefit from it, say to re-trade it or simply to show it off, she will have to call `checkNametag` and the lie will be exposed. Granted, at that time the error message becomes a little less clear, because it refers to the signer of a transaction that may have occurred long in the past...but the check will still have failed.)

Ultimately, the lesson of this set of transactions is that in Fae, nothing happens as a surprise. Contracts get the arguments they appear to get, and an exchange between two real people only happens if they are both aware of it, its contents, and its consequences.

Earning money

The modern economy is not defined by trades-in-kind like the one above (though that does happen); rather, we have currencies, universal mediums of exchange possessing objective numerical value rather than subjective object value. Fae also has a currency, called `Coin`, though it is not part of the interpreter like `Reward`s are and can be replaced with other currencies that exist on an equal footing. Without going into its implementation (which is in `Blockchain.Fae.Currency`), we will recapitulate the scenario above with an asymmetric twist: Ryan is *selling* a `Nametag`, which Sarah will try to purchase with this currency.

First, Sarah must make some money. While practically, this may involve some activity like mining, we can abstract away that detail using Fae's `Reward`, producing a short transaction that gives her one coin at a time.

Transaction source file: GetCoin.hs

```
import Blockchain.Fae.Contracts
import Blockchain.Fae.Currency

body :: Reward -> Coin -> FaeTX (Valuation Coin)
body rToken coin = do
  rCoin <- reward rToken
  coin' <- add rCoin coin
  val <- value coin'
  deposit coin' "self"
  return val
```

Tranasaction message: GetCoin

```
body = GetCoin
reward = True
inputs
  $coinTX/Body/0/Current = ()
keys
  self = $self
```

Command line:

```
./postTX.sh -e self=sarah -e coinTX=<previous txID> GetCoin
```

This transaction incrementally accumulates `Coin`s, saving each new sum in a `deposit` contract and reporting its value in the transaction results. Being incremental, it needs a place to start, so before we run it, we will also run the following transaction to prime the pump:

Transaction source file: DepositZeroCoin.hs

```
import Blockchain.Fae.Contracts
import Blockchain.Fae.Currency

body :: FaeTX ()
```

```
body = do
  zID <- zero @Coin
  deposit zID "self"
```

Transaction message: DepositZeroCoin

```
body = DepositZeroCoin
keys
  self = $self
```

Command line:

```
./postTX.sh -e self=sarah DepositZeroCoin
```

Transaction results:

```
Transaction e1ef783d15f9a9c9722b75edc73021f1386447307e771fc0cd9b6515c505fb22
  result: ()
  outputs:
    0: ff60727a04e24bf395cb90cba3079ca3cf9ebb54bed518728790d8eb3ecdfe49
  signers:
    self: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
```

ZERO

The `zero` value gives you a free `Coin` with no value, or rather, that contains a value numerically equal to zero. Though useless for monetary purposes, it is useful when, as here, you need a place to start counting. Think of this transaction as "opening an account" for Sarah, and the one above as making deposits to that account.

TYPE ANNOTATIONS

We used a new and weird syntax, `zero @Coin`, whose purpose is to make explicit what type, exactly, we expect to get from `zero`. If `zero` were a monomorphic value (that is, with a specific and concrete type) then this could be deduced from its type, but in fact, it is part of an interface called `Currency` and, therefore, we need to make it clear somehow that we want this *particular* currency. As not much is going on in this transaction, there are no other contextual clues for the interpreter to follow, so we simply annotate `zero` with the type we want; the `@Coin` fills in the first free type variable in its signature, which in this case, happens to be its return type.

As an alternative, we could actually write the full type signature, `zero :: FaeTX Coin`, but clearly, that is longer and may include substantial amounts of detail that are not relevant for fixing just this one type. And either way, we need to know exactly what its type signature is.

Transaction `GetCoin` results:

```
Transaction af6402f6f86a36bfe1ad7c12faad8a6954ef0f9364106ddfbf38c2ee3124293e
  result: 1
  outputs:
    0: ae970f5fa46165a817214bb65e0b7fa5df84b7a3c1943c5cd37e54e9f09d2d74
  signers:
    self: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
  input #0: e1ef.../Body/0/Current (Deleted)
```

VALUATION

The return type of the `GetCoin` transaction is `Valuation Coin`, which is unfamiliar. In fact, this is just a numerical type, and the `Valuation` type family is part of the `Currency` interface typeclass to provide a uniform way to describe "denominations" of money. Having a coin of `1 :: Valuation Coin` is like having one dollar, as opposed to one pound or euro or renminbi or....

REWARD FUNCTION

The function `reward`, not to be confused with the type `Reward`, is an exchange that cashes in a `Reward` for a single `Coin`; it is much the same as `getNametag`. This is not part of the `Currency` typeclass because each currency is allowed to decide if and how to offer rewards, which may in fact come in several forms. This is a mechanism for providing economic incentive for doing whatever it is that yields a `Reward`: say, mining blocks. The strength of the incentive depends on just how valuable a `Coin` is, and the incentive for *providing* the incentive is to both promote the healthy continuation of that `Reward`-producing activity (which should be something that keeps Fae going, benefiting everyone who uses it) and to promote `Coin`s themselves, which makes them more valuable.

ADD FUNCTION

The `add` function *is* part of `Currency`, as it is one of the fundamental operations that make a type act like a currency. Its type signature is not quite the same as the operator `+`, though, which adds data-based numbers; the difference is:

```
(+) :: Int -> Int -> Int
add :: Coin -> Coin -> FaeTX Coin
```

(Both of these type signatures are less polymorphic than their actual definitions.) While data can be manipulated without passing through Fae, a Fae-based valuable needs to use Fae operations such as `useEscrow` to validate and obtain the internal quantities that define it. Hence, we bind the result: `coin' <- add rCoin coin`.

This function is *destructive*. While a data value can never be destroyed while it is in scope, so that the actual `Coin` (which is an alias for an `EscrowID`) always remains available, a Fae valuable's backing escrows *can* be deleted. This is imperative for a function like `add` that actually creates new `Coin`s; in order to maintain a constant supply, it has to eliminate its operands and place their total value into its result. The implementation of `Coin` is, in effect, an implementation of the Bitcoin virtual machine.

VALUE FUNCTION

The `value` function is another member of `Currency` and is responsible for converting the actual valuable `Coin` to its non-valuable but more-numeric `Valuation Coin`. Like `add`, it returns into `FaeTX` (or more generally, a `MonadTX`). The following code snippet illustrates the meaning of this conversion:

```
-- We have coin1, coin2 :: Coin
do
  val1 <- value coin1
  val2 <- value coin2
  coin3 <- add coin1 coin2
  val3 <- value coin3
  -- val3 == val1 + val2
```

Note the order of operations here: we *first* extract the values, *then* add. This is because `add`, as discussed above, is destructive, and it is impossible to find the value of its operands after they have been used. Fortunately, `value` is non-destructive, so there is no conflict if the value is taken first.

COMMENTS

The snippet above employs a few line comments, `-- Comment text`. In this case, they stand in for some actual code that I don't want to include because it would muddy the waters of this example, but in which, say, the comparison `val3 == val1 + val2` would actually be executed. (This style of comments is quite unfamiliar for programmers who are used to languages descended from C, which include virtually all popular imperative languages. Think of the `--` as the decrement operator in those languages: we "decrement" the comment so that it goes away when the line is evaluated.)

The other form of comment is the block comment `{- Comment text -}`, which can span multiple lines. While `--` plays the role of `//`, the block comment `{- -}` plays the role of `/* */` in the C family of languages. (The mnemonic here is that we have a group, which is of course contained in `{ }`, and this group is subtracted, hence the `--` split across the delimiting braces.)

Paying with money

Now that Sarah can accumulate wealth, she can use it. First, run `GetCoin` a few times until she has a `Coin` of value 5 in her account:

Transaction results:

```
Transaction 6db8e12c88d50df46af89ae1b722392b0eefe746be744cab7e7b9524effd91e3
  result: 5
  outputs:
    0: 0d60c3a9592e443df9ea8be008e862b970e4388a67b0d244610dbd3202f408eb
  signers:
    self: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
  input #0: 489e.../Body/0/Current (Deleted)
```

Now she can offer this to Ryan in a transaction like `TradeNametags`. First, run `RunNametagFactory` for him to get a nametag:

Transaction results:

```
Transaction cebe12aacb1967edc4f4d37331502a683ee6a8f4cf003f3fb4d9cada39575089
  result: ()
  outputs:
    0: d9288acdf496effa798d133217027a9878920b3556f9fd810e1512131b58d6dc
  signers:
    self: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
  input #0: 50fe.../Body/0/Current (Updated)
    version: 50fe6913cfd78148f6024e9811a4debc2cc2ef3eb1d1784409e5c0a53fe1fe32
```

Let us say that he wants to sell his nametag for 3 `Coins`; then, an expedient transaction that satisfies both parties is:

Transaction source file: BuyNametag.hs

```
import Blockchain.Fae.Contracts
import Blockchain.Fae.Currency
import Blockchain.Fae.Transactions.TX$installTX.Nametag

import Data.Maybe (fromMaybe)
```

```
body :: Coin -> Nametag -> FaeTX (Valuation Coin, String)
body payment nametag = do
  message <- checkNametag nametag
  changedMay <- change payment 3

  let err = error "Nametag price is 3 coins"
      (priceCoin, changeMay) = fromMaybe err changedMay

  deposit nametag "person1"
  deposit priceCoin "person2"

  changeVal <- case changeMay of
    Nothing -> return 0
    Just changeCoin -> do
      val <- value changeCoin
      deposit changeCoin "person1"
      return val

  return (changeVal, message)
```

Transaction message: BuyNametag

```
body = BuyNametag
inputs
  $coinTX/Body/0/Current = ()
    self = person1
  $ntTX/Body/0/Current = ()
    self = person2
keys
  person1 = $person1
  person2 = $person2
```

Command line:

```
./postTX.sh \
  -e installTX=<InstallNametag transaction ID> \
  -e coinTX=<last GetCoin transaction ID> \
  -e ntTX=<RunNametagFactory transaction ID> \
  -e person1=sarah \
  -e person2=ryan \
  BuyNametag
```

Transaction results:

```
Transaction bdb9f1a4a286e1ea526d60c590eefc2d5aa437def14321b97af32aefef99b7e9
  result: (2,"Property of: ryan")
  outputs:
    0: a9df66888a90373fbc9697965f1d1651feaedf745fbe55cbe77cc98de4831475
    1: bff8033faadbd5c955cc49030ca73aef8066aefa52b1ffb17704bb78f29332da
    2: d7f9b3f1b8330dd96202098e840b17e52a120a1a2e8021919046e2c394e916eb
  signers:
    person1: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
```

```
    person2: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
  input #0: 6db8.../Body/0/Current (Deleted)
  input #1: cebe.../Body/0/Current (Deleted)
```

MAYBE

The (standard, built-in) type `Maybe` makes a few appearances here, starting with `import Data.Maybe`. Actually a type family, its purpose is to stand in for a value that *may be* present, or not. The actual definition is:

```
data Maybe a = Just a | Nothing
```

MULTIPLE CONSTRUCTORS

The definition of `Maybe` contains new syntax, the `|` separating two different constructors for this type. Expressions of the form `Nothing` or `Just x` are both values of a `Maybe` (of whatever type `x` is).

CASE STATEMENTS

The complement to the multiple-constructor syntax is the multiple-deconstructor expression called `case`. Its purpose is to sort through the various possibilities of constructors that may appear in some value and return different results for each one. An expression such

```
case xMay of
  Nothing -> defaultValue
  Just x -> someFunction x
```

will always have a well-defined value, but what that value is depends on how `xMay` turns out; either way, we have an alternative result.

DO BLOCK VALUES

The actual `case` expression that appears here has a more elaborate `Just` alternative, namely a full `do` block. This is actually a perfectly ordinary value of type (in this case) `FaeTX (Valuation Coin)`, and therefore needs to be matched in the `Nothing` alternative by another value of that type, which is why we write `return 0` rather than just `0`.

You may be surprised to see a `return` in the *middle* of the transaction code, with more expected to execute afterwards (namely, the final line, which is a more prosaic appearance of `return` that actually appears to *return* from the `body` function). This is because `return` doesn't actually do anything. Its purpose is to wrap a regular value in the monad that it appears in; if a `do`-block ends with `return`, then the value of the whole block is equal to whatever was returned, inside the monad that the actions are being `do`ne in.

Since in the block we are discussing, a `Valuation Coin` was returned, the block evaluates to `FaeTX (Valuation Coin)`, which means that the `0` in the alternative `Nothing -> return 0` means `0` as a `Valuation Coin` and the `return` means a return into `FaeTX`.

SIDE EFFECTS

The `do`-block inside the `case`'s `Just` alternative has an unusual feature: although if you just look at the outer level where the `case`'s value is bound to `changeVal` you would think that its purpose is to produce a `Valuation Coin`, in fact, it *also* creates a `deposit` contract in doing so. This kind of thing would make it very misleading to say that the `case` has type `Valuation Coin`, which is one of the primary reasons that we actually *have* a monad `FaeTX` to contain that value. As we introduced monads in Tutorial 1, they are an “environment” for computing a value. This

environment happens to contain information about contracts that have been created, and so when you see a `FaeTX`-wrapped value, you should consider the possible creation of contracts as part of it.

Even so, it is still misleading to write the binding of `changeVal` the way we have, because it isn't apparent until two indentations in what its side effects are. It is probably better to write a separate function,

```
changeDeposited :: Maybe Coin -> FaeTX (Valuation Coin)
changeDeposited Nothing = return 0
changeDeposited (Just changeCoin) = do
  val <- value changeCoin
  deposit changeCoin "person1"
  return val
```

and in the `body` just write `changeVal <- changeDeposited changeMay`. This not only has the advantage of having a somewhat self-explanatory name, but also of eliminating several levels of indentation and replacing the `case` alternatives with actual alternative definitions of the function, which is also more self-explanatory.

FROMMAYBE

The function `fromMaybe` that we imported from `Data.Maybe` is a shortcut into writing a full `case` expression if all we want to do is unwrap the `Maybe` and supply a default value in case it is `Nothing`. It's the same as

```
fromMaybe def xMay = case xMay of
  Nothing -> def
  Just x -> x
```

and therefore, clearly a superior alternative all around.

CHANGE

The newest member of the `Currency` interface is the `change` function, which attempts to “make change” for a value from a coin of, potentially, greater value. This can fail in two ways: first, the coin might actually have lesser value, in which case the attempt clearly fails. And second, the coin and the desired value might be equal, in which case the attempt succeeds but no change is necessary. In order to convey the sense of these alternatives, the return value of `change` has two `Maybe`s in it:

```
change :: Coin -> Valuation Coin -> FaeTX (Maybe (Coin, Maybe Coin))
```

The outer `Maybe` supplies a `Nothing` corresponding to the first kind of failure; the inner one's `Nothing` is the second kind. This is why the first `fromMaybe` in `BuyNametag.hs` defaults to throwing an error message, as it unwraps the outer `Maybe`, and of course the transaction is impossible if there wasn't sufficient payment. The `case` statement unwraps the inner `Maybe`, both alternatives of which are acceptable, but whose `Just` alternative indicates that Sarah is owed back part of her payment, which is deposited in her name.

POSSIBLE OUTCOMES

This transaction can have several outcomes. It could, of course, completely fail with the `error` message, which is something we have seen before. It can also succeed and return some change to Sarah, in which case the `case` statement produces a third output (number 2 in the transaction results above) containing the change. Finally, it can succeed but with exact payment, so there is no change and no third output; those transaction results would not have `output` 2, but (because we were careful to put the optional `deposit` last in the `do`-block) outputs 0 and 1 still have the same meaning of the purchased nametag for Sarah and the exact price paid to Ryan.

In this case, there was change, namely 2 `Coin`s, as Sarah paid 5 and Ryan asked for 3. This, along with the proof that the nametag was valid, appears in the `result` field.

Contractual sales

While the previous transaction was actually successful in mediating an exchange of money for goods, it is sub-optimal because it contains so much validation logic in the transaction body itself. This is not reusable and error-prone, and in addition, one of the parties could potentially overlook a flaw in it and get taken for a ride. Far better would be to have a *contract* for Ryan to sell his nametag, so that the only thing that would need to happen in the transaction body would be for Sarah to deposit it after the purchase succeeds, and also possibly deposit her change. This is the same structure that we used all the way back in `InstallNametag` when we defined the `OfferNametag` and `GetNametag` contracts, where the former accepts a request to transact and the latter accepts payment inside the transaction body.

Fae provides for this very common idiom with another function from `Blockchain.Fae.Contracts` that Ryan can use in the following transaction that both obtains a `Nametag` and offers it for sale.

Transaction source file: SellNametag.hs

```
import Blockchain.Fae.Contracts
import Blockchain.Fae.Currency
import Blockchain.Fae.Transactions.TX$installTX

body :: Reward -> EscrowID GetNametag -> FaeTX (Valuation Coin)
body rwd ntFactory = do
  nametag <- useEscrow [] ntFactory rwd
  seller <- signer "name"
  sell nametag price seller
  return price

  where price = 3
```

Transaction message: SellNametag

```
body = SellNametag
reward = True
inputs
  $installID/Body/0/Current = "$name"
keys
  name = $name
```

Command line:

```
./postTX.hs -e installTX=<InstallNametag ID> -e name=ryan SellNametag
```

Transaction results:

```
Transaction 765be2ad06e806a614f25724327f68b1bc98a00eca20e823f07409d09cbbfc01
  result: 3
  outputs:
    0: f9cacd0bc27a50f7dc1c092c8c6210499eb800b33a9b4c1f3cfd0774d0ec67f4
```

```
  signers:
    name: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
  input #0: 6be9.../Body/0/Current (Updated)
    version: 6be90b02aff8b9667b47268c9a4283dec811ffcd4d3948f9b05362a7cfc9a9c5
```

SELL

The function that offers a product for sale is called, appropriately, `sell`, which, as we have used it, has the following signature:

```
sell :: Nametag -> Valuation coin -> PublicKey -> FaeTX ()
```

indicating the product to sell, the price to ask, and the seller to receive it, and returning an action that creates a contract to do just that. This contract, the analogue of `OfferNametag`, is essentially a `deposit` in the seller's name, containing an escrow that accepts a payment and returns the product and maybe the change.

WHERE

The line `where price = 3` at the bottom is not actually part of the `do`-block (hence the whitespace above it), but is part of the definition of `body`; it simply defines a new value to be used inside the function. As is good practice, we avoid the indiscriminate use of the magic number 3 by giving it a meaningful name.

Executing the sales contract

Now that Ryan has consigned his `Nametag` to the care of a contract that will manage its sale, Sarah can call the contract to buy it. First she has to go through the `GetCoin` sequence again to regain her 5 `Coin`s; then she can submit the following transaction.

Transaction source file: BuyNametag2.hs

```
import Blockchain.Fae.Contracts
import Blockchain.Fae.Currency
import Blockchain.Fae.Transactions.TX$installTX.Nametag

body :: Coin -> EscrowID (Sell Nametag Coin) -> FaeTX (Valuation Coin, String)
body payment salesEscrow = do
  (nametag, changeMay) <- useEscrow [] salesEscrow payment
  message <- checkNametag nametag
  deposit nametag "person1"
  changeVal <- changeDeposited changeMay
  return (changeVal, message)

changeDeposited :: Maybe Coin -> FaeTX (Valuation Coin)
changeDeposited Nothing = return 0
changeDeposited (Just changeCoin) = do
  val <- value changeCoin
  deposit changeCoin "person1"
  return val
```

We use the suggested `changeDeposited` function from the discussion above. The format of the `Sell` type should be self-explanatory.

Transaction message: BuyNametag2.hs

```
body = BuyNametag2
inputs
  $coinTX/Body/0/Current = ()
    self = person1
  $salesTX/Body/0/Current = ()
    self = person2
keys
  person1 = $person1
  person2 = $person2
```

Command line:

```
./postTX.hs \
  -e installTX=<InstallNametag ID> \
  -e coinTX=<GetCoin ID> \
  -e salesTX=<SellNametag ID> \
  -e person1=sarah \
  -e person2=ryan \
  BuyNametag2
```

Transaction results:

```
Transaction aa617185c3bad0297552ec125996812ad9ecf2ff85ba54c5ca5fefb28f84e12c
  result: (2,"Property of: ryan")
  outputs:
    0: 8e9e142cfca4bf47b426779347d1b360b7c5c431c0e9b3079dabc0f5519657be
    1: aa617185c3bad0297552ec125996812ad9ecf2ff85ba54c5ca5fefb28f84e12c
    2: d3fbb393f607e3d37a1802761fea2dd803532881b2d248bf893f27760f6770a9
  signers:
    person1: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
    person2: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
  input #0: 6b3f.../Body/0/Current (Deleted)
  input #1: b854.../Body/0/Current (Deleted)
```

This transaction has no new elements either syntactic or of the Fae API. In fact, it functions exactly like a combination of `TradeNametags` and `BuyNametag`, including the meta-transactional logic of the former that ensures that both parties get what they want.

Even with the sales contract, Ryan has to sign this transaction, for exactly the same reason as in `TradeNametags`. Its contribution is not to automate the transaction, but to automate and modularize the transaction *logic*. This `body` requires absolutely no intelligent thought to write; it calls an escrow that is provided to it, checks the result with a function that is provided to it, and deposits it (via a function that is provided to it).

As in `BuyNametag`, the optional change account is output 2; output 1 is now Sarah's nametag, while output 0 is now Ryan's revenue, as calling `salesEscrow` triggers the `deposit` of the latter, and of course, the former can only occur after it returns the nametag.

One point that arises here, and also in `TradeNametags` and especially `BuyNametag`, is the necessity that Ryan be *available* to scrutinize transactions in his name. An actual person will most likely not be able to do this; however, if Ryan is indeed running a `Nametag`-selling business, then that business probably has a Web entry point; like Ethereum, Fae is no substitute for the regular internet. The server that hosts that site, or some other server that it can contact behind the scenes, can automatically run proposed transactions in a sandbox (as Ryan would himself do) and verify

that they terminate with acceptable results. Although Ryan's interests must be protected in these sales, his actual attention need not be occupied with what is, in the end, a repeatable task best performed by a computer.

Summary

This tutorial, which was even more lengthy than Tutorial 2, presented Fae's concepts of economics both as simple trades and more computational monetary payments. The main points were:

- The use of escrows as “sales agents” as well as currencies or non-numerical valuables;
- The use of factories to automate the offering of these sales agents to potential customers;
- The meta-transactional logic of multiple signers who each protect their own interests.

Many, many new syntactic elements were introduced, as well as more of Fae's API:

- Renaming syntax for assigning roles in a contract;
- The `Blockchain.Fae.Currency` module's `Coin` currency, as well as glimpses of the `Currency` typeclass;
- The `Blockchain.Fae.Contracts` module's `sell` function and `Sell` contract name

This content is very practical and addresses what is most likely the vast majority of activities that take place in a blockchain-mediated smart contract system.