Tutorial 2: Escrows

In this second tutorial, we will introduce the remaining feature of Fae: *escrows*, which are the means by which you may create value and perform complex transactions. As in the first tutorial, 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
```

Creating escrows

To present the concepts surrounding escrows accessibly, we will start with not a full transaction, but simply a contract that uses them. To begin, we will revert to the ad-hoc style of defining a contract, and return to contract names when they are seen to be absolutely necessary.

CONTRACT SOURCE CODE

```
c :: Contract String (EscrowID (Contract () String))
c name = newEscrow e where
  e :: Contract () String
  e _ = spend ("Property of: " ++ name)
```

This is a contract that creates an escrow personalized according to the caller's request.

ESCROW CREATION

The escrow creation function, `newEscrow`, has exactly the same argument as `newContract`, including the type `Contract argType valType`: escrows are the same as contracts but have a different role. As we will see later, they can be called as well, with different restrictions; this involves the *escrow ID*, which is returned by `newEscrow`. This particular escrow can be used once, returning a declaration of who created it, and is then deleted, just like contracts that end with `spend` are deleted when that line is executed.

ESCROW ID

An escrow is created with a unique identifier; if the escrow's name has type `name`, then the escrow ID has type `EscrowID name`; `EscrowID` is, like `ArgType` and `ValType`, a family of types. The reason the name is specified is so that contracts that accept escrow IDs can guarantee that they get the right kind of escrow, and users of contracts that return escrow IDs can guarantee that *they* got the right kind of escrow. As we will see, escrows can represent valuable quantities in Fae, so these guarantees amount to an assurance that the contracts deal in a particular kind of valuable (say, a specific currency).

Using escrows

An escrow is of no use in isolation: it must be used, if only to verify it. The transaction below will demonstrate calling it. First, we should commit the contract above to Fae via a transaction.

Transaction source file: Escrow1.hs

```
type EType = Contract () String

body :: FaeTX ()
```

```
body = newContract c where
  c :: Contract String (EscrowID EType)
  c name = do
    eID <- newEscrow e
    spend eID

    where
      e :: EType
      e _ = spend ("Property of: " ++ name)
```

Transaction message: Escrow1

```
body = Escrow1
```

Command line:

```
./postTX.sh Escrow1
```

Transaction results

```
Transaction 49c34b94bc40bf1012ec989bebecea10719c577c3f1c3f8d01ffd4457feba73c
  result: ()
  outputs:
    0: 49c34b94bc40bf1012ec989bebecea10719c577c3f1c3f8d01ffd4457feba73c
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
```

Like most of the examples in Tutorial 1, this transaction just creates a single new contract. This example also demonstrates how additional code beyond the transaction executable can be included in a transaction source file.

TYPE SYNONYMS

The first line of the transaction above is a *type synonym definition*, much like the instantiations of `ArgType` and `ValType` we saw in Tutorial 1. It declares an alias (in this case, a shorter one for convenience) for another type. Not only would the long expression `Contract () String` otherwise occur twice in this transaction, but it would also appear in every other transaction, and we will see that we can reuse the same synonym in all of them.

BINDING THE FAE API

This example has another of the *do block* for creating contract bodies longer than a single line. This block allows more than simple imperative expressions like `newContract contractName`; it also allows *bindings* of values obtained from the Fae API. Bindings in a `do`-block can assign any name (like `eID`) to any value that would otherwise appear on a line (like `newEscrow e`), "extracting" the content of the value from the monad at the present time and making it available for use on subsequent lines.

Transaction source file: CallEscrow1.hs

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

body :: EscrowID EType -> FaeTX String
body eID = useEscrow [] eID ()
```

Transaction message: CallEscrow1

```
body = CallEscrow1
inputs
  $txID/Body/0/Current = "Ryan Reich"
```

Command line:

```
./postTX.sh -e txID=<transaction ID from Escrow1> CallEscrow1
```

Transaction results

```
Transaction 304142b3b74befb0eef7fa3204ede5be2dc4714ada6f66855251d7814eb91d1f
  result: "Property of: Ryan Reich"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: 49c3.../Body/0/Current (Deleted)
```

The contract ID format `$txID/Body/0/Current` should be familiar from the first tutorial, signifying that the contract to be called was the first one created by transaction `$txID`. This transaction calls the contract above, passing it my name as an argument, and then uses it with the ID for that new transaction.

IMPORT STATEMENT

The first line of *this* transaction is the *import statement*, which recalls the source-code module defined in the transaction `$txID`. This environment variable is actually allowed to appear in the code, although it is invalid syntax, because `postTX` scans the import statements for exactly this and substitutes them before sending the transaction. All transaction modules are placed under the hierarchical location `Blockchain.Fae.Transactions` with the specific name `TX<the transaction ID>`. Other core Fae modules are placed under `Blockchain.Fae` as well.

The effect of the import statement is that all of the definitions in `Escrow1.hs` become available by name to `CallEscrow1.hs`. Thus, the name `EType` is available, and means the same thing. This is important when using user-defined types, because simply repeating the definition in the second transaction will not create a type that the interpreter understands to be the same as the original, but rather, another type with the same name and (coincidentally) the same definition.

Speaking of definitions with the same name, this import *also* exposes the `body` function of `Escrow1.hs`; however, since we explicitly define a function of this name in `Escrow2.hs`, the latter "shadows" the former, and is the one that the interpreter sees when running the transaction.

CALLING AN ESCROW

Escrows can be called like functions with `useEscrow`, which takes three arguments: an unexplained list, here the empty list `[]`; the escrow ID; and the argument to the escrow itself, here the empty argument `()`. This returns a string, as indicated by the second type argument to `EscrowID () String`, which is returned from the transaction. The transaction result is therefore `Property of: Ryan Reich`, as shown.

ESCROW OWNERSHIP

One subtle but extremely important point that comes out in this example is the *ownership* of this escrow. Originally, it was owned by its creator, the contract created in `Escrow1`. However, by returning the escrow ID from its call, that contract *transfers* ownership to the transaction that called it, which is `CallEscrow1`. It is only because of this

transfer that the `useEscrow` call is valid; if an escrow is called by a contract or transaction that does not own it, the call will fail with an error indicating that the escrow does not exist. Escrows only exist for their owners.

Saving escrows

The transaction above does not use escrows to their full potential, but merely as glorified functions. In Fae, escrows are the only means of creating persistent value, and in order to persist, they must be held by a contract rather than expended by a transaction. Before running this transaction, you need to re-execute `Escrow1`, because the contract it created was deleted in `CallEscrow1`.

Transaction source file: Escrow2.hs

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

body :: EscrowID EType -> FaeTX ()
body eID = newContract (Spend eID)

data Spend a = Spend a deriving (Generic)

instance (ContractVal a) => ContractName (Spend a) where
  type ArgType (Spend a) = ()
  type ValType (Spend a) = a
  theContract (Spend x) = \_ -> spend x
```

Transaction message: Escrow2

```
body = Escrow2
inputs
  $txID/Body/0/Current = "Ryan Reich"
```

Command line:

```
./postTX.sh -e txID=<transaction ID from Escrow1> Escrow2
```

Transaction results

```
Transaction f75495105860cf227c3eef53049519b08e9e0f7c801f8a144b07503e909b12ce
  result: ()
  outputs:
    0: f75495105860cf227c3eef53049519b08e9e0f7c801f8a144b07503e909b12ce
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: af00.../Body/0/Current (Deleted)
```

This transaction calls the contract from `Escrow1` (for me, the re-execution of this transaction had ID `af00...`, which isn't shown here but does appear in the input contract ID), obtaining an escrow, then creates a contract to hold the escrow rather than calling it. This contract, in turn, will pass the escrow back to its next caller, then be deleted (leaving that caller in possession of the escrow). We'll leave it as an optional exercise to formulate the transaction that does so.

POLYMORPHISM AND CONSTRAINTS

In this transaction, we see for the first time a definition that applies to not one type but a family of types: a *polymorphic* definition. It begins with the `data Spend a` statement, which defines a type that depends on a variable `a`, an unspecified other type. So, `Spend (EscrowID EType)` is a specific example of this family of types; this works much the same way as `ArgType` and `ValType`. This dependence on a free type variable is called *polymorphism*.

Matching this is the `instance ContractName (Spend a)` statement, which declares a uniform instance of the `ContractName` typeclass for all of the specific `Spend a` types. That is, almost all: the instance declaration is actually decorated with a new piece of notation: `(ContractVal a) =>`, which is a *constraint* limiting the particular types `a` that this instance can actually apply to. (You want to read this as the "if" side of an "if...then" statement, as in, "If `a` is a `ContractVal`, then `Spend a` is a `ContractName`.") The constraint contains one or possibly several properties that must hold of the free type variables mentioned elsewhere in the signature; here, a "property" is an interface (typeclass) that the type in the variable must have.

It is important to mention here that the `body` function is *not* polymorphic, even though as mere code it could, apparently, have the signature `body :: (ContractVal a) => a -> FaeTX ()`, because that is all we need to get the `ContractName` instance for the `Spend` value we use to create the new contract. This will not fly with Fae, however, because `body` must be a *monomorphic* type (as opposed to polymorphic; it should have no free type variables at all) in order for the interpreter to know how to call it on the return values of the input contracts, whose types are effectively dynamic in order to accommodate, in a single framework, all the possible types of contract return values. It relies on the specificity of the definition of `body` to cast those dynamic types to the correct static ones. (Optional exercise: change the type signature so that it actually has the wrong type for its input, and see how Fae handles this.)

CONTRACT VALUES

The specific interface mentioned in the constraint above is `ContractVal`, an internally defined collection of constraints that ensure that types that have it are suitable for returning from a Fae contract or escrow. These all, in practice, tend to be implied by the `Generic` property that your data types will derive, but there are some, more primitive, types that have this interface but not because they are `Generic`, so it is good form, when writing a polymorphic definition that needs to return a value from a contract, to use the least restrictive and most descriptive property, which is `ContractVal`.

BEARERS OF VALUE

This transaction finally demonstrates the need to use data types, rather than inline functions, as contract names. This need is simply that *ownership of the escrow `eID` needs to be transferred to the new contract*. This transfer is performed automatically and invisibly simply by scanning the value provided as the contract name for escrow IDs, and in order to do this, Fae needs to be able to introspect into the type of that value. This kind of introspection is provided for data types by the `Generic` property, and is impossible for function types. The analogous contract definition to the above that uses an inline function would fail because the contract code would behave as the owner of an escrow that was never transferred to it.

This general concept, of data values containing escrow IDs, is Fae's notion of a "valuable" *backed by* escrows that, through their contractual behavior, give it value. Such a data value can of course be passed into functions or used in definitions, but Fae ensures that the ownership of its backing escrows is correctly maintained. In this case, the creation of a contract, that is accomplished by introspecting into the type of the contract name, extracting all escrow IDs, and moving them from the creating code's escrow storage into the created contract's escrow storage. This unifies the concept of *program value*, which is just data managed by the interpreter, with *Fae value*, which is data managed by the Fae internals. In other words, just like an escrow ID can't be called except by the owner of the escrow itself, a bearer of value is invalid if its backing escrows are not owned by the contract that makes use of it.

(As a slight aside, the `spend` and `release` functions are the other ways to transfer escrows, as indicated in the previous example; they both also introspect into their arguments.)

The upshot is that the evaluation `theContract` function on a `Spend x` is definitively the owner of the value `x`, whatever it is (here, it's an `EscrowID name`) and can do what it wishes with it, including transferring it again via `spend`.

Restricted-access contracts

The example above is highly flawed if the escrow is to be a bearer of value, since there are no access restrictions on the contract in which it is deposited. Here we show how to rectify this (once again, you need to re-execute `Escrow1`).

Transaction source file: Escrow3.hs

```
import Blockchain.Fae.Transactions.TX$txID
import Control.Monad (unless)

body :: EscrowID EType -> FaeTX ()
body eID = do
 us <- signer "self"
 newContract (Restricted us eID)

data Restricted a = Restricted PublicKey a deriving (Generic)

instance (ContractVal a) => ContractName (Restricted a) where
  type ArgType (Restricted a) = ()
  type ValType (Restricted a) = a
  theContract (Restricted us x) = \_ -> do
    them <- signer "self"
    unless (us == them) (error "Wrong sender")
    spend x
```

Transaction message: Escrow3

```
body = Escrow3
inputs
  $txID/Body/0/Current = "Ryan Reich"
```

Command line:

```
./postTX.sh -e txID=cab24f... Escrow3
```

Transaction results

```
Transaction e0543e9374b1cd1dab03402d3ec206b47734b6f91f1be95cf25b9a02fa1712a7
  result: ()
  outputs:
    0: b4f45d8bd1ba4571688ff0174395d380975f0306c12298d2ad64970a2d38d407
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: 3747.../Body/0/Current (Deleted)
```

This transaction calls the contract of `Escrow1`, deleting it, and passes its return value, which is an escrow ID, to a newly created contract, as shown in the `outputs`. The result is a contract that only returns the escrow ID to the same person (literally person, as in "someone who can send a transaction") that first deposited it. In order to see the restriction logic in action, we present a transaction to call this contract, similar to `CallEscrow1` except that it also sets the signers.

Transaction source file: CallEscrow3.hs

```
import Blockchain.Fae.Transactions.TX$tx1ID

body :: EscrowID EType -> FaeTX String
body eID = useEscrow [] eID ()
```

Transaction message: CallEscrow3

```
body = CallEscrow3
inputs
  $txID/Body/0/Current = "Ryan Reich"
keys
  self = $self
```

Command line:

```
./postTX.sh \
  -e tx1ID=<transaction ID from Escrow1> \
  -e txID=<transaction ID from Escrow3> \
  -e self=<your choice>
```

Transaction results (`self = other`)

```
Transaction 3cf8c74e88c29d00808c61595103f7f3019bdec3af791fc0d308ee5d3106e5a1
  result: <exception> Wrong sender
  signers:
    self: 412535810f40782eb779855a8cadbd5e0c5e7af1715d2f4d9c863c1fc8106987
  input #0: e054.../Body/0/Current (Failed)
    <exception> Wrong sender
```

Transaction results (`self = self`)

```
Transaction 1ba58cd97f17f52ba530b582b9883792ce8e7fa94d53c3bec8bde779240d737d
  result: "Property of: Ryan Reich"
  signers:
    self: 72c52f0acab0128d8ad56866e27bd2866aaa2c54ae754a917fafbb438e5c9420
  input #0: e054.../Body/0/Current (Deleted)
```

Note how `CallEscrow3.hs` imports the transaction module from `$tx1ID`, while the transaction message `CallEscrow3` uses `$txID` in the input call's contract ID. This reflects their different needs: the source file needs the definition of `EType`, which is provided by `Escrow1`; the message file needs the ID of the contract to be called, which is the one created by `Escrow3`. So long as both variables are set it does not matter that they are not even both used in the same files.

The first transaction uses an identity for `self` that is different from the one used in `Escrow1` (which is the default key, also called `self`; see below), and receives an exception. The second one uses the correct identity and receives the expected message as its `result`.

This example is more complex than usual and introduces several new syntax features.

PUBLIC KEYS

Fae uses the type `PublicKey` as a cryptographic proof of identity: when the transaction message is received, the required signers are listed in it and their signatures are attached, and checked. If and when this check passes, the signers have been proven to have the necessary identities (and the message to be unmodified) and can be referred to within the transaction simply by their public key, which (as far as is computationally reasonable) uniquely specifies them. The actual nature of the cryptography involved is irrelevant; Fae happens, for now at least, to use the Ed25519 elliptic curve schema, but that does not need to be known, and cannot be used, in the transaction body.

SIGNERS

The `signer` API function is the interface to an associative array containing the public keys of the various parties that signed the transaction currently being processed when it is bound. Multiple signatures are possible and this enables some intricate ownership schemes, as well as defending against the misuse of one's contracts. Therefore, in `Escrow3.hs`, the binding `us <- signer "self"` binds `us` to the public key of the signer named `self` of the transaction written in this code; however, the binding `them <- signer "self"` that appears in the contract it creates binds the `self` signer of the transaction that *calls* the contract (which is `CallEscrow3`), since the line is not actually executed until that time.

There are two other functions: `lookupSigner`, which returns a `Maybe PublicKey` instead of throwing an error for a missing signer like `signer` does, and `signers`, which returns a `Map` of all the signers.

Fae often uses `self` as the default name for a transaction signer (`postTX` actually supplies it if it is not present in the message, with a private key also called `self`, but this is a demo behavior). Signers can be declared in the transaction message similarly to inputs:

```
keys
  self = ryan
  other = sarah
```

The names on the right indicate named public keys to be used to sign the transaction; the private keys are stored without any attempt at security by `postTX` (again, this is demo behavior; despite this, it is feasible to use `postTX` unmodified in a live system, so long as the machines that run it are secured; this kind of activity has absolutely nothing in particular to do with Fae, so is not attempted by its tools).

DECISIONS AND EXCEPTIONS

The line `unless (us == them) (error "Wrong sender")` is quite novel for us, as it contains the first appearance of one of the staples of programming, the conditional. The `unless` method (provided by `Control.Monad`) takes a boolean value and, if it is false, executes the second argument; if it is true, it does nothing. Here, the boolean is `us == them`, which exploits the fact that `PublicKey` supports the `==` operator; in formal terms, it is an instance of the `Eq` interface (typeclass), like `Control.Monad` a part of the Haskell standard library package `base`.

The action `error "Wrong sender"` raises an *exception*, which causes the entire transaction to end and its return value to be replaced by this exception. In addition, none of the transaction's actions are saved: new contracts are not created, though contract calls are still made normally. An exception in a contract call will roll back just the one call, preventing its update or deletion. Aside from the exceptional return value, it is as though the transaction (respectively, contract call) did not happen. Exceptions may be thrown using `error` or `throw` (which takes a different kind of argument) but may not be caught, and so should not be used for control flow. The appropriate time to throw an exception is when an error occurs that renders the contract meaningless.

Exceptions are raised not only by explicit errors but also by errors of programming such as type mismatches, evaluating an undefined value (for example, `1/0`), calling a contract by an invalid contract ID or with a literal argument that

doesn't parse correctly, or using an escrow that you do not own.

The inability to catch exceptions has one exception in that it is possible to attach recovery routines to a transaction in case it does exit prematurely. This feature will be discussed elsewhere.

LIBRARY ALTERNATIVE

This example is instructive but unnecessary, since Fae provides a contract library that includes a method to save a value in a contract owned by a particular public key. Here is a shorter way of writing `Escrow3.hs`:

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

body :: EscrowID EType -> FaeTX ()
body eID = deposit eID "self"
```

The module `Blockchain.Fae.Contracts` contains a method `deposit` accepting a value-bearing argument and a name, and creates a contract that releases the value to the caller if that caller is the same as the one referred to by the name *in the original transasction*. Its implementation is as shown above.

Restricted-access escrows

This ongoing example still contains a serious issue: the escrow can easily be obtained without calling contract `<Escrow1 transaction ID>/Body/0/Current`. Indeed, *anyone* is free to write the code in the contract definition to create their own escrow that works the same way. This means that the escrows issued by that contract have no value: the access control placed on the contract that saves them does not actually keep the escrow private. Here, we at last present the method that makes escrows actually valuable.

MODULE SOURCE CODE

```
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) => String -> m Nametag
getNametag name = Nametag <$> newEscrow (NametagName name)

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

This code defines not a transaction or contract, but a new module like `Blockchain.Fae.Contracts`. This module provides the interface to escrows like the ones previously created by contract `<Escrow1 transaction ID>/Body/0/Current`.

MODULE DEFINITIONS

A module is just a source file that can be imported by others, including transactions. It *exports* some or all of the names defined in it, which are available to any other source file that imports it, whereas the ones that are not exported, are not available outside the module. Modules occupy a place very similar to that of classes in languages like `C++`: indeed, you can make an easy analogy between the above module and the definition of a class called `Nametag` having a private type `NametagName` with a constructor of that same name, a public type `Nametag`, and two public functions `getNametag` and `checkNametag` defining its interface. The analogy ends there, though, since one does not instantiate the `Nametag` type as a programmatic value `x` that has members such as `x.checkNametag`.

The export list has a subtlety that we exploit: when exporting a type like `Nametag`, just placing that type name in the list only exports the type name itself, and *not* any of its constructors. If you want the constructors you have to write `Nametag(the constructor name)`, or just `Nametag(..)` to get all of them (if there were more than one).

MODULE IMPORTS

This module imports `Blockchain.Fae`, the main Fae module that contains the Fae API. This module is provided implicitly to all transactions, but must be imported explicitly by modules (because, conceivably, a module could provide just a general source code library intended to be used in contracts but not actually using them itself).

NEWTYPES

In addition to the familiar `data NametagName = NametagName String` that defines a data-bearing type with a constructor having a single `String` field, we also see the similar but new `newtype Nametag = Nametag (EscrowID NametagName)`. This defines a data type that only *looks* different from its single field; internally, it has exactly the same values. This construct is typically used to build an abstraction over a more general type and enforcing the use of that abstraction rather than taking a shortcut through the original type. We use it here in just this way, so as to hide the escrow ID from consumers of `Nametag`s.

Although there are some efficiencies both in coding and also execution in using newtypes, for the most part you could replace `newtype` with `data` and nothing would change. I chose to use it here because it signifies my intent to abstract away the fact that a nametag is just an escrow.

THE MONADTX CONSTRAINT

The functions `getNametag` and `checkNametag` return into an unfamiliar context: rather than `Fae` or `FaeTX`, their return value is in some other monad `m`, constrained by the type signature to be a `MonadTX`, which is evidently some interface (typeclass) defined elsewhere. In fact, it is defined in the Fae core, but it doesn't matter how, because you will only ever use it in type signatures as a constraint and never implement it for your own types. It means "`m` is a monad that resembles `FaeTX` as far as the Fae API is concerned". So, we can use `newContract` and `newEscrow`, `useEscrow` and `signer`, but not `release` or `spend`, because those are not allowed in transactions.

This typeclass exists so that the Fae monads can be extended to include more features, which we will demonstrate in a later tutorial. It also serves a more immediate purpose: it simultaneously covers both `FaeTX` and *every* `Fae argType valType`, because this restricted form of the API is available in both. This means that functions that return into a polymorphic `MonadTX` can be used in contracts, transactions, or escrows, a degree of generality that could not be achieved without the constraint.

There is also a typeclass `MonadContract argType valType`, which generalizes `Fae argType valType` and *does* allow `spend` and `release` which, you should recall, require `argType` and `valType` for their type signatures. Using this monad, you can implement reusable contract fragments.

THE DOLLAR SIGN OPERATOR

The code for `getNametag` has an odd syntax, separating functions from their arguments with a dollar sign `$`. This useful operator allows us to avoid writing parentheses: it collects its entire right-hand side and provides it as a single value to its left-hand side. So if `f` is a function, `f $ x = f (x)` for any expression `x`, which is visually cleaner particularly when there are nested parentheses.

The similar operator `<$>` in `checkNametag` is actually used for something different. It allows us to apply the `Nametag` constructor to the Fae value of type `EscrowID NametagName`, which latter type is the one that `Nametag` actually takes, rather than what we have as the result of `useEscrow`, which is a `FaeTX (EscrowID NametagName)`. By using `<$>` we enter the monad and apply the constructor there. It is exactly the same as writing

```
f <$> xMonadic = do
  x <- xMonadic
  return $ f x
```

but, of course, much shorter and more to the point.

HOW IT WORKS

There are no other new syntactical elements in this file, but its construction exploits the semantics of module exports and patterns in a new way. Quite simply, it protects the escrow by making its contract name and its escrow ID private types, usable only within the `Nametag` module. Since consumers of this module are unable to create their own `NametagName` values, they are also unable to create a valid escrow with that contract name type, and so cannot forge a `Nametag`. In addition, if they are given a `Nametag`, they cannot use it to call the backing escrow directly, because only the *type* name `Nametag` was exported; its constructor called `Nametag` was not, and so the pattern match used by `checkNametag` cannot be written in any other code. This means that a nametag is used *exactly* as specified by its interface, or else not used at all: it is a high-level abstraction that prevents the use of the lower-level primitives on which it is built.

The `checkNametag` function has another purpose besides getting the name: validation. Although no one can use the same function definition to create a nametag, it is still possible to write contract code like

```
newNametag :: String -> FaeTX Nametag
newNametag name = Nametag <$> newEscrow undefined
```

where `undefined` is a value that throws an exception when evaluated, as it would be if for instance the escrow were actually called. It can be of any type, so this definition is type-correct, but it can never be used. But `checkNametag` *does* use it, so if one attempts to evaluate it on an attempted forgery as above, the undefined contract name has to be evaluated and this throws an exception. Thus, the *success* of a `checkNametag` call indicates that the nametag is truly valid. This is also why the nametag escrows have been redefined to `release` rather than `spend` their contents, so that they can be checked repeatedly without being deleted.

(Lest you think that I am setting up `undefined` as a strawman, rest assured that the only kind of value that can masquerade as a private type is an erroneous value: one that, when evaluated, either throws an exception or fails to produce a result at all.)

Rewards and payment

Using the module concept, we can lock down access to a particular type of escrow to its public interface. As written, the `Nametag` interface does not restrict the use of `getNametag`, so that although the value of nametags is protected by the module, it does not really exist at all because nametags are free. The solution to this is, of course, to charge for them. The only form of value built in to Fae is the reward, which may be awarded as a participation incentive. To use it, we just alter `getNametag` slightly.

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

Then, a special *reward transaction* can call this function, but no other kind of transaction.

Transaction source file: RewardNametag.hs

```
import Blockchain.Fae.Contracts
import Nametag

body :: Reward -> FaeTX ()
body rwd = do
  nt <- getNametag rwd "Ryan Reich"
  deposit nt "self"
```

Transaction message: RewardNametag

```
body = RewardNametag
others
  - Nametag
reward = True
keys
  self = ryan
```

Command line:

```
./postTX.sh RewardNametag
```

Transaction results:

```
Transaction 0e49d0a3b1ede73524354c3f7b343524ef6a284a67f5e55120bb1a427992ff24
  result: ()
  outputs:
    0: bd11897e5a2950f0311b932395834c29b00a1ed7ed5fd95a789927cc3faedee1
  signers:
    self: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
```

This cashes in the reward to get a nametag, then saves that nametag in a signature-protected contract. We can see that it works as expected using the following transaction:

Transaction source file: CallRewardNametag.hs

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

body :: Nametag -> FaeTX String
body = checkNametag
```

Transaction message: CallRewardNametag

```
body = CallRewardNametag
inputs
  $txID/Body/0/Current = ()
keys
  self = $self
```

Command line:

```
./postTX.sh \
  -e txID=<transaction ID from RewardNametag> \
  -e self=<your choice>
  CallRewardNametag
```

We call this first with `self = sarah` and then `self = ryan` to see the access restriction working:

Transaction "sarah" results:

```
Transaction 83b487a511b20f9011abcf118beff8cbc81ffe2e7d75aca4ff519a702fc62c44
  result: <exception> Signer with public key 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2
  signers:
    self: 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb4bd5c96a4e477077a
  input #0: 0e49.../Body/0/Current (Failed)
    <exception> Signer with public key 260cc5cf5cc36548b1ae701db9d3f3ff478fbd2e2bbeddb
```

Transaction "ryan" results

```
Transaction 7b784d4dbe4e783d9cfdf029cd711e20d74d16898357a00914b9cce8e5c8dbcb
  result: "Property of: Ryan Reich"
  signers:
    self: 738a79bc3b11c200ceb5190a14382cb5aa6514640d0f05adfc93d2bb23139091
  input #0: 0e49.../Body/0/Current (Deleted)
```

SUBMODULE IMPORTS

The `import` line in `CallRewardNametag` specifies `Nametag` *beneath* the transaction module for `RewardNametag`; this is because the latter uploaded this module *as* `Nametag`, and so it is placed hierarchically inside its module namespace. Though `RewardNametag` can use the module just as `Nametag`, any other transaction has to give the full path.

MULTIPLE ARGUMENT FUNCTIONS

The type of `getNametag` was changed to `(MonadTX m) => Reward -> String -> m Nametag`, which denotes a function that takes two arguments: first, a `Reward`, and second, a `String`. This is one way, and the best one, to write a function taking several arguments. This is not always an option; for instance, `theContract` must always define a function taking only one argument, so if you want to write a contract with more than one, you need to use the other option: *pairs*.

PAIRS

A pair is something of the form `(x,y)`. This is the same concept you might see in, e.g. Python, and is definitely *not* the same concept as the function-call syntax you see in C-family languages. A function taking a pair is, more or less, a function taking two arguments: the syntax `f (x,y)` actually means to call `f` on the single value that is a pair `(x,y)`, but can be understood to mean that `f` takes two arguments just like in the C family.

REWARDS AND REWARD TRANSACTIONS

Rewards in Fae have the type `Reward`, which is provided by the module `Blockchain.Fae` (and therefore available implicitly to any transaction). This is a semi-private type like `Nametag` for nametags, available for type signatures but not construction or deconstruction. Values of this type are only created in reward transactions, which in a more realistic deployment might be designated as such in blocks and may be issued, for instance, to miners (the `reward` field in the message indicates to `faeServer` to simulate this designation; obviously, in reality a transaction should not be able to declare itself a reward recipient). A reward transaction gets an extra input value, before the the list of explicit contract calls, of type `Reward`.

CLAIMING REWARDS

As we saw for nametags, the private type must be accompanied by a public interface that can validate an escrow of that type. For rewards, that interface has only one function, `claimReward`, which accepts a valid `Reward` and deletes its backing escrow (i.e. the escrow is defined to `spend` after one call). Therefore, rewards cannot be reused or accumulated (though they can be saved), so are not really a currency; they are just an entrypoint to self-contained Fae value.

Because rewards are scarce and can be validated, they are one way to meaningfully limit access to the `Nametag` type. Essentially, the `getNametag` function is a reward-to-nametag conversion engine, though this does not make nametags the same as rewards, because (for example) they cannot be used to pay `getNametag`.

FALLING OFF THE END

You'll notice that `callRewardNametag` terminated while still holding the `Nametag`, as it did not `deposit` it again. This valuable is actually *destroyed* when the transaction terminates; Fae makes no attempt to preserve dangling escrows.

Summary

This lengthy tutorial gradually developed the machinery of escrows, which are contracts-within-contracts that can be used to represent value in Fae. We saw:

- Escrows can be created like contracts and called like functions using their escrow ID, but only by their owner.
- Escrows can be transferred to new contracts or escrows, or returned by ID from contract or escrow calls.
- To make an escrow type valuable, it should be encapsulated in a module defining a public interface to a private type.

We also learned about some more programming features:

- Many, many new syntactic constructs (linked in the table of contents).
- The `signer` API method.
- The `Blockchain.Fae.Contracts` module and, in particular, `deposit` for saving things in a new contract.