# Lecture Six

> “Unless you try to do something beyond what you have already mastered you will never grow.”
> — Ralph Waldo Emerson

## 1. Introduction

*Note: You always write your introduction last.*

## 2. Oracles

> "Know Thyself."
> The Oracle, The Matrix
> It would be disingenuous not to also attribute this quote to Socrates

For a moment consider applications in general. As we have already concluded, they're typically fairly 'unsafe' and can run off to do some IO without asking (on the odd occasion). Fortunately we're fairly decent programmers *(I think some circle jerking is fairly appropriate at this point in the course, kudos to all those who have stuck through it)* and our programs know how to behave. However, sometimes it is appropriate to accept data from a stranger, and **to be clear**, that's what oracles are. They're strangers, who knows how reliable some random fortune-teller you just met is?[1]

Unfortunately this untrustworthy bunch are the only people who seem to know anything around here. So, that's where can get our IO from the outside world from. Now, there are mitigation schemes. Lars does go into this within the video, so I won't spend too much time discussing them. But we're looking at:

- An Oracle putting down some collateral, which incentivises this 'randomer' to actually be truthful, otherwise: bad things happen to them (they loose their money).
- You could aggregate many different oracles together and take the mean? If there were a number of oracles outside an assigned threshold, the transaction could abort / rollback.
- Stuff like this... c'mon guys, start thinking creatively!

## 2.1 What have we learnt thus far?

We now know that:

- Oracles make data (similarly to OPEN APIs) accessible to smart contracts within the eco-system.
- Oracles are not to be trusted[2].
- Oracles are ONE way of getting data from the outside world into a smart contract, smart contract validators can be (as is the case with Haskell function, I think?) 'overloaded' with parameters (don't confuse the term overloaded with the OOP keyword, in this instance it's simply an adjective), meaning we could inject our own outside data via the redeemer, this would be suitable in cases where, for example, you need multiple signatories to access a smart contract (or an oracle).
- The value provided by an Oracle is an output from the Tx paid to the oracle, the particular output is stored within the datum.

## 2.2 Use Cases

Let's keep this short and sweet (I want to get developing on the #purple test net ASAP!):

- 'one-shot' oracles, where the value does not change over time (for example, the result of a football game) - betting.
- Distributing information about weather conditions to any vehicle that requests it.
- I mean, you be creative!

*Note: this is a SIMPLE APPROACH, you can do things such as aggregate data, process it, run it through functions, etc.*

# 3. How Exactly Do These Things Work Then?

Right.

> *COFFEE TIME.*
> *Take five minutes, have a think - how could they work? It doesn't matter if you're wrong, remember:*
> *"A foolish consistency is the hobgoblin of little minds" ... and "to be great is to be misunderstood"*
> *I would highly recommend some Emerson if you've not read his work.*

Let's work through this methodically:

1. What are our requirements? What are our constraints?
2. Our primary requirements are as follows:
    - At minimum, one wallet can store data on-chain.
    - Smart contracts and DApps can read data using on-chain.
    - We must be able to identify which script address any given oracle sits at.
    - a signatory must be able to update the value of an oracle.

    - Fees must be paid to read from the oracle.
    - Fees may be collected when updating the oracle Datum value.

3. A wallet can store on-chain data by creating an oracle with an output datum which contains a value. However, the script must also contain pass an NFT as input into the duplicated oracle (since there can only be one unique script address for any given NFT). This means you obtain another set of UTxOs: the same NFT and the same Datum (along with paid fee from the reader) as input into the next instance somebody reads from the oracle.
4. Smart contracts and DApps can read from specific oracles because, as mentioned, there can only be a singe script address for a properly implemented oracle.
5. We can update an oracle if we are the signatory. We consume the UTxO at the address of the NFT, and feed the NFT and the Datum back into the script, whilst collecting the fees everybody else who was reading the oracle paid in. Now we have a 'new' oracle with an updated Datum, as we have changed the state of the output Datum - and that is how we update.

## 4. Let's Have A Look At Some Code

It's important to remember that an Oracle is a parameterised script, and the Oracle data structure is defined below within the module below:

*Defining an Oracle and importing the necessary modules...*

```
module Week06.Oracle.Core
    ( Oracle (..)
    , OracleRedeemer (..)
    , oracleTokenName
    , oracleValue
    , oracleAsset
    , typedOracleValidator
    , oracleValidator
    , oracleAddress
    , OracleSchema
    , OracleParams (..)
    , runOracle
    , findOracle
    ) where

import           Control.Monad             hiding (fmap)
import           Data.Aeson                (FromJSON, ToJSON)
import qualified Data.Map                  as Map
import           Data.Monoid               (Last (..))
import           Data.Text                 (Text, pack)
import           GHC.Generics              (Generic)
import           Plutus.Contract           as Contract
import qualified PlutusTx
import           PlutusTx.Prelude          hiding (Semigroup(..), unle
ss)
import           Ledger                    hiding (singleton)
import           Ledger.Constraints        as Constraints
import qualified Ledger.Typed.Scripts      as Scripts
import           Ledger.Value              as Value
import           Ledger.Ada                as Ada
import           Plutus.Contracts.Currency as Currency
import           Prelude                   (Semigroup (..), Show (..),
 String)
import qualified Prelude
```

The code is declaring what an Oracle is and what it can do (abstractly). It's heavily typed, as we need to import various type classes to achieve the end goal.

*Getting Started...*

```
data Oracle = Oracle
    { oSymbol   :: !CurrencySymbol
    , oOperator :: !PubKeyHash
    , oFee      :: !Integer
    , oAsset    :: !AssetClass
    } deriving (Show, Generic, FromJSON, ToJSON, Prelude.Eq, Prelude.Ord

PlutusTx.makeLift ''Oracle

data OracleRedeemer = Update | Use
    deriving Show

PlutusTx.unstableMakeIsData ''OracleRedeemer

{-# INLINABLE oracleTokenName #-}
oracleTokenName :: TokenName
oracleTokenName = TokenName emptyByteString

{-# INLINABLE oracleAsset #-}
oracleAsset :: Oracle -> AssetClass
oracleAsset oracle = AssetClass (oSymbol oracle, oracleTokenName)

{-# INLINABLE oracleValue #-}
oracleValue :: TxOut -> (DatumHash -> Maybe Datum) -> Maybe Integer
oracleValue o f = do
    dh      <- txOutDatum o
    Datum d <- f dh
    PlutusTx.fromBuiltinData d
```

- Data Oracle is the parameter (using the constructor as previously seen)
- The `oSymbol :: !CurrencySymbol` is going to be responsible for allowing for a unique script address, as the TokenName for the NFT is simply going to be 'empty'.
- The `oOperator :: !PubKeyHash` is simply the operator of the oracle, it will allow the wallet with the appropriate key to update the oracle when required.
- The `oFee :: !Integer` is the value to be paid when reading from the oracle.
- Finally, the `oAsset :: !AssetClass` will contain a representation of USD, as in this instance we're pretending to convert ADA to USD. This is *essentially* where the Datum lives.
- Standard practice at this point, making the compiler happy with Lift code.
- The redeemer can be one of two values, as different business logic required to be executed depends on the value of the redeemer.
- The remainder of the code is... at this point, fairly self-explanatory.
- The TokenName is an Empty Byte String.
- The NFT is oracleAsset, as that is where the empty byte string and CurrencySymbol live.

- Then the oracleValue is somewhat more confusing, because of the nature of how Datum works.
    - TxOut is going to be a hash that can be used in the Datum map to pull out the Datum.
    - The second param is the map.
    - Return type is safely declared as an Integer (which makes sense, as we're concerned with exchange rates).
    - The definition of the oracleValue describes a process where:
        - the datum hash is obtained (dh)
        - it is then used to obtain the datum and pushed into a var (d)
        - we then prepare the datum for the compiler

Right, these Oracles are pretty large programs, so I'm going to be a little more concise in my explanation for now, as time is a factor (Since I'm the most academic individual on my team for Catalyst, I'm spending a lot of time reviewing the document, I'm trying to get going with Alonzo Purple and I'm also running a couple of small pet projects, these 16 hour days are going to have to stop soon!).

Right, so here is the Validator:

```
{-# INLINABLE mkOracleValidator #-}
mkOracleValidator :: Oracle -> Integer -> OracleRedeemer -> ScriptCont
ext -> Bool
mkOracleValidator oracle x r ctx =
    traceIfFalse "token missing from input"  inputHasToken  &&
    traceIfFalse "token missing from output" outputHasToken &&
    case r of
        Update -> traceIfFalse "operator signature missing" (txSignedBy
                  traceIfFalse "invalid output datum"       validOutputDa
        Use    -> traceIfFalse "oracle value changed"       (outputDatum
                  traceIfFalse "fees not paid"              feesPaid
  where
    info :: TxInfo
    info = scriptContextTxInfo ctx

    ownInput :: TxOut
    ownInput = case findOwnInput ctx of
        Nothing -> traceError "oracle input missing"
        Just i  -> txInInfoResolved i

    inputHasToken :: Bool
    inputHasToken = assetClassValueOf (txOutValue ownInput) (oracleAsset

    ownOutput :: TxOut
    ownOutput = case getContinuingOutputs ctx of
        [o] -> o
        _   -> traceError "expected exactly one oracle output"

    outputHasToken :: Bool
    outputHasToken = assetClassValueOf (txOutValue ownOutput) (oracleAss

    outputDatum :: Maybe Integer
    outputDatum = oracleValue ownOutput (`findDatum` info)

    validOutputDatum :: Bool
    validOutputDatum = isJust outputDatum

    feesPaid :: Bool
    feesPaid =
      let
        inVal  = txOutValue ownInput
        outVal = txOutValue ownOutput
      in
        outVal `geq` (inVal <> Ada.lovelaceValueOf (oFee oracle))
```

- Standard validator input, except for the additional parameter of the Oracle
- Integer for the Datum

- Remember, we're not in procedural land, this aint assembly lads, we can use functions that have not yet been defined in the code...
- Use of Helper Functions to ensure the NFT is passed in AND out
- 'Switch' statement which either updates or reads based on redeemer input
- `txSignedBy info $ oOperator oracle` checks that the key within the ctx can be signed using the pubhashkey.
- I don't think I need to explain the rest of the initial part of this script...
- info is essentially just a function that follows a path through the data structures we've seen millions of times by now and obtains the requirement for a digital signature
- ownInput just checks for the Oracle data structure
- inputHasToken checks for the NFT
- ownOutput ensures the datum remains the same
- output has token ensures the NFT is part of the UTxO
- outputDatum is looking up the datum which is embedded in... script.context.txInfo.datum -> hashmap of datums, or a normal map, however Haskell people like to define their data structures :)
- validOutputDatum checks, well.. that the output Datum is valid :D
- feesPaid ensures the reader is paying his taxes!

> Death, Haskell and Taxes
> — Jon Dilworth

Remember guys, this is all declarative functional, I think it looks a bit confusing because it looks almost like a procedural script, but Haskell is actually REALLY LAZY, most of this won't even be executed, due to the switch statement based on the redeemer.

*Right, now let's finish up with the remaining on-chain code*

```
data Oracling
instance Scripts.ValidatorTypes Oracling where
    type instance DatumType Oracling = Integer
    type instance RedeemerType Oracling = OracleRedeemer

typedOracleValidator :: Oracle -> Scripts.TypedValidator Oracling
typedOracleValidator oracle = Scripts.mkTypedValidator @Oracling
    ($$(PlutusTx.compile [|| mkOracleValidator ||]) `PlutusTx.applyCode`
    $$(PlutusTx.compile [|| wrap ||])
  where
    wrap = Scripts.wrapValidator @Integer @OracleRedeemer

oracleValidator :: Oracle -> Validator
oracleValidator = Scripts.validatorScript . typedOracleValidator

oracleAddress :: Oracle -> Ledger.Address
oracleAddress = scriptAddress . oracleValidator
```

Again, to be brief, we're essentially just compiling the script to plutus-core, which (as the script is parameterised) requires lift code. Usual deal, compile, create the validator and then use the validator (which, remember - contains the NFT -> CurrencySymbol, so it's always going to be unique, so we get a unique address that, I believe we can reuse. I think next lecture we talk about state machines and this is kind of how it works. Datum holds state, the address remains the same, and you can change the business logic of what is essentially an automaton by changing the Datum state. That's what I got from reading about it, but we'll leave it until L7.

*In this instance, the off-chain code could be anything, so... let's leave it there for now.*

## Footnotes

1. That's why a couple of us have a small pet project in the making, more to come on this soon!

2. See the following funny quote from Snatch by Guy Richie (if you've not watched it, do it!):

> What's in the car?
> Seats and a steering wheel.
> What do you know about gypsies?
> I know they're not to be trusted.