CARDANO DEVELOPMENT COURSE (Week 2-3)

Module 2: Aiken Intro.

2.0. Aiken Types (Primitives; Custom Types)

2.1. Aiken Functions

2.2. Aiken Constants

2.3. Control Flow

2.4. Intro to Aiken Standard Library

2.5. Intro Aiken Validators

2.6. Hello world Aiken Program; Always True Validator

2.0. Aiken Types (Primitives; Custom Types)

2.0.1. Primitive Types

Aiken has 6 primitive types that are built in the language and can be typed as literals: booleans, integers, strings, bytearrays, data and void. The language also includes few base building blocks for associating types together: lists, tuples, options, pairs...

- **Int** is Aiken's number type. It's an arbitrary sized integer.
- **ByteArray** is an array of bytes.
- **String** has extremely narrow use case in Aiken and on-chain code.
- **Bool** is a boolean value that can be either **`True`** or **`False`**.
- **Tuple** is useful for grouping values that are logically related.
- **List** is an ordered collections of values of the same type.
- **Option** is used in situations where you need optional values.
- **Ordering** is a type that can be used when comparing two values of same type.
- **Void** is a type representing the nullary constructor.
- **Never** is needed to refer to an **`Option`** that can only ever be **`None`**.
- **Data** is an opaque compound type that can represent any possible types.

Int

Aiken's number type. It's an arbitrary sized integer.

Examples:

```
let i: Int = 1
let j = 2
```

(type-annotations are optional in Aiken, as long as the type can be inferred)

See also: Let bindings

Literals can also be written with _ as separators to enhance readability:

```
let ada: Lovelace = 3_000_000
```

ByteArray

A ByteArray is an array of bytes.

Examples:

```
let a: ByteArray = "abcd"
let b = "1234"
```

Behind the scene, Aiken decodes the encoded string for you and stores only the raw bytes as a ByteArray.
This is achieved by prefixing a double-quotes byte string with a # like so:

```
let c = #"abcd"
let d = #[0xab, 0xcd]
```

Note that **"abcd"** is not the same as **#"abcd"**:

```
"abcd" == #[0x61, 0x62, 0x63, 0x64]
#"abcd" == #[0xab, 0xcd]
```

String

In Aiken text strings can be written as text surrounded by double quotes, prefixed with @

Examples:

```
trace @"Trace Label": x, y, z
todo @"TODO"
fail @"Error!"
```

The use case for strings is extremely narrow in Aiken and on-chain code.

Bool

A Bool is a boolean value that can be either True or False.

Examples:

```
let y: Bool = True
let n = False
```

Aiken defines a handful of operators that work with booleans:

&&	Logical conjunction (AND)
||	Logical disjunction (OR)
==	Equal
!=	NOT Equal
!	Logical negation (NOT)

Tuple

Tuples are useful for grouping values that are logically related.
Each element in a tuple can have a different type.

Examples:

```
let t2: (Int, Int) = (1, 2)
let t3: (Int, ByteArray, Bool) = (3, "abc", True)
let t4 = (False, "123", 4, #"56")
```

Elements of a tuple can also be accessed by destructuring:

```
let (a, _) = t2       // a == t2.1st
let (_, b, _) = t3    // b == t3.2nd
let (_, _, _, d) = t4 // d == t4.4th
```

List

Lists are ordered collections of values. Each element in a list must be the same type.

Examples:

```
let ls: List<Int> = [1, 2, 3]
let bs = ["abc", "def"]
```

Inserting at the front of a list is very fast, and is the preferred way to add new values:

```
[0, ..ls] == [0, 1, 2, 3]
["123", ..bs] == ["123", "abc", "def"]
```

Note that you cannot spread a list other than at the end:

```
[..ls, 4] // ERROR
```

Option

Options are used in situations where you need optional values.

Examples:

```
let someone: Option<Int> = Some(1)
let someabc = Some("abc")
let neither: Option<Bool> = None
```

You can pattern-match an Option like:

```
when someone is {
  Some(_) -> .. // do something
  None -> .. // do something else
}
```

Ordering

Ordering type can be used when comparing two values of same type.
The value is either **Less**, **Equal**, or **Greater**

Examples:

```
script_purpose.compare(mint, spend) == Less
output_reference.compare(o_ref, o_ref) == Equal
```

```
credential.compare(key, script) == Greater
```

Void

Void is a type representing the nullary constructor, like:

```
type Void {
  Void
}
```

If you think in terms of *tuples*, **Void** is a tuple with no element in it.
Void is only useful in certain situations.

In Aiken everything is a typed **expression** (there's no *statement*),
so you'll rarely end up in a situation where you need it.

Never

In some rare cases, it is needed to refer to an Option that can only ever be None.

This is the case in some parts of the standard library and mainly due to unforeseen bugs in the ledger and backward compatibility concerns.

It is identical in all points to **None** and serialises down to the same binary structure:

```
let x = Never
let o = None
x == o
```

Data

A Data is an opaque compound type that can represent any possible serializable types.

Examples:

```
let a: Data = 1
let b: Data = "abc"
```

```
let c: Data = True
```

See also: Upcasting

Likewise, we can also downcast Data like so:

```
expect i: Int = a       // i == 1
expect s: ByteArray = b // s == "abc"
expect y: Bool = c      // y == True
```

Further Reading: *https://aiken-lang.org/language-tour/primitive-types*

2.0.2. Custom Types

Aiken's custom types are named collections of keys and/or values. They are similar to objects in object-oriented languages, though they don't have methods.

Custom types are defined with the type keyword. They may contain named fields, or not; but they cannot mix.

- **Single Constructors**: are commonly written using the shorthand-notation
- **Multi Constructors**: can also be used as enums, for example: Bool
- **Generics**: are parameterized types, for example: Option<a>
- **Recursive Types**: are types which one or more fields refer to its own type
- **Opaque Types:** are encapsulated-object-like types

Single Constructors

```
Example:
type NamedFields {
  field_name: ByteArray,
  int_field: Int,
}
```

This is a shorthand notation, because single constructors are so common.

Instantiating single constructor variables:

You can instantiate a variable using the let-bindings:

```
let named_fields = NamedFields { field_name: "abcd", int_field: 1 }
let nameless_fields = NamelessFields("abcd", 1)
let void = Void
```

You can also instantiate a named-fields constructor in a nameless-fields notation:

```
let named_fields_actually = NamedFields("abcd", 1)
named_fields_actually == named_fields // True
```

Accessing single constructor fields:

You can access a named-fields single constructor fields by named-accessors:

```
let named_fields = NamedFields { field_name: "abcd", int_field: 1 }
named_fields.field_name == "abcd" // True
named_fields.int_field == 1       // True
```

You can also access them by destructuring:

```
let NamedFields { field_name, int_field } = named_fields
```

```
let nameless_fields = NamelessFields("abcd", 1)
let NamelessFields(a, b) = nameless_fields
```

```
field_name == a // True
int_field == b  // True
```

Updating single constructor values:

Aiken provides a dedicated syntax for updating some of the fields of a custom type record utilizing the spread operator (`..`) by replacing the given binding with their new values.

Examples:

```
let original = NamedFields { field_name: "abcd", int_field: 1 }
let updated = NamedFields { ..original, int_field: 2 }
```

```
original.int_field == 1 // True
updated.int_field == 2  // True
```

The update syntax **creates** a new record with the values of the initial record.
So it does NOT replace the original record values.

Multi Constructors

Custom types in Aiken can be defined with multiple constructors,
making them a way of modeling data that can be one of a few different variants.

Example:

```
type Multi {
  NamedFields { boolean: Bool, maybe: Option<Int> }
  NamelessFields(Bool, Option<Int>)
  Fieldless
}
```

In fact, **`Bool`** and **`Option`** are also multi constructors types:

```
type Bool {
```

```
  False
  True
}
```

Accessing multi constructor fields:

Unlike single constructor, we access multi constructors fields by pattern-matching or "destructuring" using the **`expect`** keyword (non-exhaustive pattern-matching)

Pattern-matching:

```
let recall_our_multi_type: Multi = NamelessFields(True, None)
when recall_our_multi_type is {
  NamelessFields(is_true, _) -> is_true
  NamedFields { .. } | Fieldless -> False
} // this will return True since is_true is True
```

Non-exhaustive pattern-matching:

```
let our_multi_type: Multi = NamelessFields(True, None)
expect NamelessFields(is_true, _) = our_multi_type
// is_true is True
```

Generics

Custom types can be parameterized with other types, making their contents variable.

We have seen that with **`Option`**:

```
type Option<a> {
  Some(a)
  None
}
```

The type inside the constructor **Some** is **a**, which is a parameter of the **Option** type. If it holds an **Int** the **Option** type is **Option<Int>**, for example:

```
let someone: Option<Int> = Some(1)
```

See also: https://aiken-lang.org/language-tour/custom-types#generics

Recursive Types

A type of a field can also refer to its own type, for example our custom Tree type:

```
type Tree<a> {
  Tip
  Node { left: Tree<a>, right: Tree<a>, value: a }
}
```

We can work with **our Tree** type like:

```
let tree: Tree<Int> = Node(Node(Tip, Tip, 2), Node(Tip, Tip, 3), 1)
```

```
expect Node { value: root_value, .. } = tree
root_value == 1 // True
```

Our Tree

```
  1
 / \
2   3
```

Notice that we're destructuring the **tree** variable using the **expect** keyword.

Opaque Types

An **opaque** type is an *encapsulated-object-like* type where the constructors and fields are **private** so that users of this type can only use the type through publicly exported functions (*setter-getters*)

Example:

```
pub opaque type Class {
  i: Int,
  a: ByteArray,
  y: Bool,
}
```

Here, we have a single constructor opaque type called **`Class`** with 3 fields.
Remember, there is no **`Class`** keyword in Aiken.

Accessing opaque types fields:

Since any constructors and fields of an opaque type are private,
we need to provide public functions in order for the type to be useful.

Examples:

```
pub fn new(i: Int, a: ByteArray, y: Bool) { Class(i, a, y) }
let o: Class = new(1, "Hello", True)
```

```
// ## Setter-getters
pub fn get_i(o: Class) -> Int { o.i }
pub fn set_i(o: Class, new_i: Int) { Class { ..o, i: new_i } }
```

and so on...

Updating opaque types fields:

We can modify an opaque type field value by calling its defined setter function.
It will then create a new record with the values of the initial record and NOT replace the original record values:

```
let o: Class = new(1, "Hello", True)
```

```
let o1 = set_i(o, 2)
let o2 = set_a(o, "World")
let o3 = set_y(o, False)
```

```
(get_i(o), get_a(o), get_y(o)) == (1, "Hello", True)     // True
(get_i(o1), get_a(o1), get_y(o1)) == (2, "Hello", True)  // True
(get_i(o2), get_a(o2), get_y(o2)) == (1, "World", True)  // True
(get_i(o3), get_a(o3), get_y(o3)) == (1, "Hello", False) // True
```

Single-constructor single-field opaque types:

There's a *special treatment* for an opaque type with only one constructor, and if that constructor has only 1 field. Under the hood, it behaves like an opaque type-alias. This matters when we're dealing with CBOR.

Example:

```
pub opaque type NewType<a> { field: a }
pub fn new_type(a) -> NewType<a> { NewType(a) }
trace new_type(43) // 43
```

Compare that to a non-opaque type,

```
pub type Constr<a> { field: a }
pub fn new_constr(a) -> Constr<a> { Constr(a) }
trace new_constr(43) // 121([_ 43])
```

Further reading: *https://aiken-lang.org/language-tour/custom-types*

2.1. Aiken Functions

Functions in Aiken are defined using the *fn* keyword. Functions can have (typed) arguments, and always have a return type. Because in Aiken, pretty much everything is an expression, functions do not have an explicit return keyword. Instead, they implicitly return whatever they evaluate to.

```
fn add(x: Int, y: Int) -> Int {
  x + y
}

fn multiply(x: Int, y: Int) -> Int {
  x * y
}
```

Functions are first class values and so can be assigned to variables, passed to other functions, or anything else you might do with any other data type with the exception of: being part of a data-type definition.

Anonymous function

Anonymous functions can be defined with a similar syntax and assigned to an identifier using a let-binding. The identifier then serves as a name to call and pass the function around.

```
fn run() {
  let add = fn(x, y) { x + y }

  add(1, 2)
}
```

Further reading: *https://aiken-lang.org/language-tour/functions*

2.2. Aiken Constants

Constants is a way to use certain fixed values in multiple places of a project.

Examples:

```
const the_ultimate_answer_of_everything: Int = 42
const pkh =
#"00000000000000000000000000000000000000000000000000000056"
pub const key_token = ("PolicyId", "NFT", 1)
```

Constants are immutable. They can hold *almost* any Aiken expression and are fully **evaluated at compile-time**.

Constants can refer to other identifiers such as functions and other constants with one restriction: constants cannot refer to other constants that are defined **after** them in the file. As such, there's no recursive or mutually recursive constants.

2.3. Control Flow

2.3.1. Blocks

Every block in Aiken is an expression. All expressions in the block are executed, and the result of the last expression is returned.

```
let value: Bool = {
    "Hello"
    42 + 12
    False
}

value == False
```

2.3.2. If-Else

Pattern matching on a Bool value is discouraged and if *condition* else expressions should be used instead.

```
let some_bool = True

if some_bool {
  "It's true!"
} else {
  "It's not true."
}
```

Note that, while it may look like an imperative instruction: if this then do that or else do that, it is in fact one single expression. This means, in particular, that the return types of both branches have to match.

Further reading: *https://aiken-lang.org/language-tour/control-flow*

2.4. Intro to Aiken Standard Library

Before diving into the standard library, let's review some Aiken commands.

- **aiken new:** Create a new Aiken project
- **aiken c:** Type-check an Aiken project and run any tests found
- **aiken b:** Build an Aiken project
- **aiken add:** Add a new project package as dependency
- **aiken docs:** Build the documentation for an Aiken project

The official standard library (*stdlib*) for the Aiken Cardano smart-contract language. It extends the language builtins with useful data-types, functions, constants and aliases that make using Aiken a bliss. To add to your project, you run:

```
aiken add aiken-lang/stdlib --version v2
```

Example Usage:

```
use aiken/collection/list
use aiken/crypto.{VerificationKeyHash}
use cardano/transaction.{OutputReference, Transaction}

pub type Datum {
  owner: VerificationKeyHash,
}

pub type Redeemer {
  msg: ByteArray,
}

/// A simple validator which replicates a basic public/private
signature lock.
///
/// - The key (hash) is set as datum when the funds are sent to the
```

```
script address.
/// - The spender is expected to provide a signature, and the string
'Hello, World!' as message
/// - The signature is implicitly verified by the ledger, and
included as 'extra_signatories'
///
validator hello_world {
  spend(datum: Option<Datum>, redeemer: Redeemer, _, self:
Transaction) {
    expect Some(Datum { owner }) = datum

    let must_say_hello = redeemer.msg == "Hello, World!"

    let must_be_signed = list.has(self.extra_signatories, owner)

    and {
      must_say_hello,
      must_be_signed,
    }
  }
}
```

In the line below for example:

```
let must_be_signed = list.has(self.extra_signatories, owner)
```

You can see that *list* from the stdlib is used as a utility function to check if an item is a member of a list.

Checkout The Standard Library: *https://aiken-lang.github.io/stdlib/*

2.5. Intro Aiken Validators

A validator is a named block that contains one or more handlers. The handler name must match Cardano's well-known purposes:

- mint, spend, withdraw, publish, vote or propose.

Every handler is a predicate function: they must return True or False. When True, they authorize the action they are validating. Alternatively, to return false, they can instead halt using the *fail* keyword or via an invalid *expect* assignment.

With the exception of the spend handler, each handler is a function with exactly three arguments:

- A **redeemer**, which is a user-defined type and value.
- A **target**, whose type depends on the purpose.
- A **transaction**, which represents the script execution context.

The spend handler takes an additional first argument which is an optional datum, which is also a user-defined type.

Handler	What for
mint	Minting / burning of assets
spend	Spending of transaction outputs
withdraw	Withdrawing staking rewards
publish	Publishing of delegation certificates
vote	Voting on governance proposals
propose	Constitution guardrails, executed when submitting governance proposals

```
use cardano/address.{Credential}
use cardano/assets.{PolicyId}
use cardano/certificate.{Certificate}
use cardano/governance.{ProposalProcedure, Voter}
use cardano/transaction.{Transaction, OutputReference}

validator my_script {
  mint(redeemer: MyMintRedeemer, policy_id: PolicyId, self: Transaction) {
    todo @"mint logic goes here"
  }

  spend(datum: Option<MyDatum>, redeemer: MySpendRedeemer, utxo:
OutputReference, self: Transaction) {
    todo @"spend logic goes here"
  }

  withdraw(redeemer: MyWithdrawRedeemer, account: Credential, self:
Transaction) {
    todo @"withdraw logic goes here"
  }

  publish(redeemer: MyPublishRedeemer, certificate: Certificate, self:
Transaction) {
    todo @"publish logic goes here"
  }

  vote(redeemer: MyVoteRedeemer, voter: Voter, self: Transaction) {
    todo @"vote logic goes here"
  }

  propose(redeemer: MyProposeRedeemer, proposal: ProposalProcedure, self:
Transaction) {
    todo @"propose logic goes here"
  }
}
```

Further reading: *https://aiken-lang.org/language-tour/validators*

2.6. Hello world Aiken Program; Always True Validator

```
use aiken/collection/list
use aiken/crypto.{VerificationKeyHash}
use cardano/transaction.{OutputReference, Transaction}

pub type Datum {
  owner: VerificationKeyHash,
}

pub type Redeemer {
  msg: ByteArray,
}

validator hello_world {
  spend(
    datum: Option<Datum>,
    redeemer: Redeemer,
    _own_ref: OutputReference,
    self: Transaction,
  ) {
    expect Some(Datum { owner }) = datum
    let must_say_hello = redeemer.msg == "Hello, World!"
    let must_be_signed = list.has(self.extra_signatories, owner)
    must_say_hello && must_be_signed
  }
}
```

Our first validator is rudimentary, yet there's already a lot to say about it.

It looks for a verification key hash (owner) in the datum and a message (msg) in the redeemer. Remember that, in the eUTxO model, the datum is set when locking funds in the contract and can be therefore seen as configuration. Here, we'll indicate the owner of the contract and require a signature from them to unlock funds—very much like it already works on a typical non-script address.

Moreover, because there's no "Hello, World!" without a proper "Hello, World!" our little contract also demands this very message, as a UTF-8-encoded byte array, to be passed as redeemer (i.e. when spending from the contract).

It's now time to build our first contract!

Further reading: *https://aiken-lang.org/example--hello-world/basics*