Universidad de los Andes

ISIS-4217 Paradigmas de programación
**Object orientation**
Date: September 2, 2025

---

As a project for the object oriented programming part of the course we are going to create a new implementation of composable objects using "meta functions".

As we do not want to build a full new object system with its own syntax, we will use the function representation of objects as a record of named functions we can use. Additionally, the composable objects will respond to an specific API of "meta" functions to help create, manage, and modify composable objects.

**Task 1. Object definition:**
To start building composable objects, we first require an implementation of objects. The definition of objects will follow the definition of bundled modularity, with a couple of differences.

First, object attributes will be defined explicitly as modifiable values within the object function. That is upon creating an object you need to define each of the attributes as variables of the functions. Such variables should be defined as cells. Upon definition of an object, then the values for each of its variables should be explicitly given.

Second, object attributes should become an explicit part of the object structure (the record). For facility, attributes will be defined in an `Attributes` function that returns a record (`attributes`) with the names and cell values for each of the attributes.

To test your object definition, implement the following objects:

- An `Employer` object with two attributes `Name` and `Address`, and three functions, `Name` (that returns the name of the employer), `Address` (that returns the address of the employer), and `Display` (that displays the string "Employer" in the following line "Name:" followed by the employer name, and in the following line "Address:" followed by the address of the employer).

- A `Person` object with two attributes `name` and `employer`, and three functions, `PersonName` (that returns the name of the person), `PersonEmployer` (that returns the name of the employer attribute), and `Display` (that displays the string "Person" in the following line "Name:" followed by the person name).

Given multiple object instances we want to compose them by means of the "meta" function `Compose`, that receives any number of objects and returns the composed object. An object is the composition of multiple objects if it contains the attributes and methods from all of its constituents. We show an example of object composition for two objects in Snippet 1.

Note that the composition of objects has a couple of caveats to take into account.

1. The composition should be idempotent (composing the same object twice should return just the object without replicated information

```
local O1 O2 Comp in
O1 = {NewObject1 Val}
O2 = {NewObject2 Val1 Val2}
Comp = {Compose O1 O2}

{Show {O1 getAttribute1} == {Comp getAttribute1}}

end
```

Snippet 1: Task 1.

2. Multiple objects can share or have polymorphic attributes or behavior. Composing them should result in the instance of the attributes or behavior of the first object in which they appear

There are two main ways to implement the `Compose` function, we will implement both of them.

**Task 2.** The first possibility (`ExplicitComposition`) is to, upon receiving the objects to compose, build a brand new object with the attributes and functions that correspond to the composition.

Implement the `ExplicitComposition` function that receives any number of objects, and returns their composition.

**Task 3.** The second possibility (`ImplicitComposition`) is to, upon receiving the objects to compose, manage them as part of a new object, and use the appropriate attributes and methods as nedded.

Implement the `ImplicitComposition` function that receives any number of objects, and returns their composition. **Note:** This is a harder function to implement as it will require more creativity on your part.

Now we will extend the composition of objects with additional functionality (managed at the meta-level). Out of facility, we will use the explicit composition of objects implemented before, but with a couple of modifications.

**Task 4.** Extend the composition of objects so that, whenever there are clashes between methods (methods defined in multiple objects) they all become part of the object. In such cases, the definition of the methods in the object structure should become an ordered list of all the implementations from the composite objects (in the order in which they appear).

Implement the function `ExplicitCompositionPoly` that lets you compose objects with method clashes (keeping all methods)

**Task 5.** For the new definition of objects you will have to implement (explicitly but this is really a meta function) a dispatching function to be able to actually use the methods. Since now method objects are lists of methods, they cannot be applied directly (*e.g.,* {`A .deposit 10`}). These functions will have to be dispatched {`Dispatch A deposit 10`}, calling the dispatch function directly with the receiving object, the function selector to call, and the function parameters (I know this looks awkward, but is the way of doing it without having to build a full new language).

Of course, the implementation of dispatch should work exactly as the functions were working in `ExplicitComposition`, but this will change in a bit.

**Task 6.** Now we will put the dispatch (metafunction) to work. To make sense of keeping all method implementations in the composed object, we want to implement a new function for composed objects `NextFunction` that when used (within a method), will call the next method in the list of applicable methods. To keep this function easy, will manage the list of applicable methods as an indexed list. Therefore the dispatch function needs an index parameter that states which method are we currently calling, to be able to call the next one.

Using the dispatch function the first time we will use it as {`Dispatch A deposit 10 1`}, effectively calling the first implementation of the deposit method in the composed object `A`. The new signature of the dispatch function then becomes: the receiver, the selector, the parameters, and as last argument the index of the function to call. As a convention we will suppose that the index is always a valid number, and that calling an index greater than the available implementations will have no effect.

Now, in the object functions we can call `NextFunction` that will in turn result in calling the dispatch function, as {`Dispatch A deposit 10 2`}. Note that all functions must be called from the dispatch, so the index is available to call the next function.

**Hand-in a file** `composition.oz` with all implemented functions for composable objects and the example objects. You may define additional functions for your code if you need them, but at the very least all specifications described above should be present.

The object system that we just created is interesting as it provides a more flexible object model than the rigid inheritance model. Through composition/delegation it is possible to reuse behavior more freely, flexibly composing objects to have differentiated behavior, even if their constituents are the same.