JOS - Julia Object System

António Menezes Leitão

March, 13, 2023

1 Introduction

A Meta-Object Protocol (MOP) mediates the behavior of an object-oriented system, e.g., the instantiation of objects (and, therefore, classes), inheritance from superclasses, and method invocation. The Smalltalk language was the first to propose a metaobject protocol, which was mostly used to manage the creation of instances of classes. The Common Lisp Object System (CLOS) went much farther, but its MOP is considerably more complex, in particular, because of CLOS' support for multiple inheritance, generic functions, and multiple-dispatch methods.

Unfortunately, it is usually difficult to implement MOPs in programming languages that were not designed to be extended and this has been one of the biggest obstacles for the wider acceptance of MOPs. There are programming languages, however, that allow language extensions and Julia is one of them.

2 Goals

The main goal is to implement JOS, the Julia Object System, an extension to the Julia programming language that supports classes and metaclasses, multiple inheritance, and generic functions with multiple-dispatch methods. The implementation should follow the way those ideas were implemented in CLOS, in particular, in what regards the CLOS MOP.

It is important to note that Julia already provides its own version of generic functions and multiple dispatch methods but it does not support classes or inheritance (much less, multiple inheritance). This means that you will need to implement your own version of these concepts, although you might find it advantageous to use the ones already available in Julia to support your implementation in a meta-circular style.

To implement JOS, it is important to first understand how CLOS operates. At the end of this document you will find multiple references explaining CLOS. You might also find it useful to look at different implementations of CLOS and of its derivatives, such as PCL - The Portable implementation of CLOS, for Common Lisp, Tiny-CLOS, for Scheme, Swindle, for Racket. The Dylan programming language is also an important source of ideas. As is usual nowadays, there is a large amount of information available online.

In the rest of this document, whenever we mention the concepts of *generic function* or *method*, we are referring to the ones you need to implement and not to those of Julia. Whenever we mention the concept of *function* (the non-generic variety), we are referring to Julia's generic functions.

This document also includes multiple examples of the use of JOS. To shorten the output, arrays will be printed in simplified form, avoiding the element type and with an horizontal layout. Moreover, when only a fraction of the array is relevant for the example, we will use `...` for the remaining, unspecified, elements, as in `[a, b, ...]`.

You must implement, at the very least, the features described in the following sections:

2.1 Classes

Classes are created using the macro `@defclass` that, in its simplest form, takes the name of the class to define, a possibly empty vector of superclasses, and a possibly empty vector of slot names.

Here is one class to implement complex numbers in rectangular representation:

```
@defclass(ComplexNumber, [], [real, imag])
```

Note that an empty vector of superclasses is equivalent to a vector containing the `Object` class. Besides creating the class, a `@defclass` form also creates a global variable with the same name of the class that references the class, allowing it to be used to create instances of that class.

2.2 Instances

To create instances of a class, the `new` function is used. This function takes, as mandatory argument, the class to instantiate, and as keyword arguments, the values of the slots to initialize.

For example, we can create an instance of the previous class and assign it to variable `c1` using the following expression:

```
c1 = new(ComplexNumber, real=1, imag=2)
```

2.3 Slot Access

The operations `getproperty` and `setproperty!` are used to retrieve or modify the slots of an object. Note that the Julia language also supports a more user-friendly syntax for these operations. The following interaction demonstrates it:

```
> getproperty(c1, :real)
1
> c1.real
1
> setproperty!(c1, :imag, -1)
-1
> c1.imag += 3
2
```

2.4 Generic Functions and Methods

A generic function is defined using the macro `@defgeneric`, as follows:

```
@defgeneric add(a, b)
```

Note that a generic function does not specify behavior. That is the role of methods.

A method is defined using the macro `@defmethod`, as follows:

```
@defmethod add(a::ComplexNumber, b::ComplexNumber) =
  new(ComplexNumber, real=(a.real + b.real), imag=(a.imag + b.imag))
```

The method definition should have the same parameters as the corresponding generic function, each one typed with the class to which it applies. An omitted type means that the parameter has the most generic type, i.e., `Top`.

When a method is defined, if the corresponding generic function does not exists, it is automatically created using, as parameters, the names of the parameters of the method.

2.5 Pre-defined Generic Functions and Methods

There is a multitude of generic functions (with corresponding methods) already implemented. For example, to support printing objects on a Julia's `IO` stream, there is the generic function:

```
@defgeneric print_object(obj, io)
```

The previous function is automatically called whenever an instance is about to be printed:

```
> c2 = new(ComplexNumber, real=3, imag=4)
<ComplexNumber Azn2z9XrGHH>
```

Note that an instance is printed using the name of its class and a unique identifier for that instance. This happens because the default behavior of the `print_object` generic function is

implemented at the `Object` class level (from which all classes inherit, directly or indirectly) as follows:

```
@defmethod print_object(obj::Object, io) =
  print(io, "<$(class_name(class_of(obj))) $(string(objectid(obj), base=62))>")
```

Naturally, it is possible to add additional methods to that generic function:

```
@defmethod print_object(c::ComplexNumber, io) =
  print(io, "$(c.real)$(c.imag < 0 ? "-" : "+")$(abs(c.imag))i")
```

which gives us the following behavior:

```
> c1
1+2i
> c2
3+4i
> add(c1, c2)
4+6i
```

2.6 MetaObjects

Given an instance of a class, it is possible to retrieve the class to which it belongs by using the (non-generic) function `class_of`:

```
> class_of(c1) === ComplexNumber
true
```

Classes describe the structure of their instances, for example, their direct slots. These are the slots specified on the class definition and, therefore, do not include inherited ones:

```
> ComplexNumber.direct_slots
[:real, :imag]
```

Interestingly, classes are objects, too, and, therefore, also instances of other classes. `Class` is an important one:

```
> class_of(class_of(c1)) === Class
true
> class_of(class_of(class_of(c1))) === Class
true
```

As is visible in the previous interaction, `ComplexNumber` is an instance of `Class`. Additionally, `Class` is also an instance of `Class`, i.e., it is its own class. Given that classes describe the structure of their instances, it is the class `Class` that describes the structure of other classes, including itself. That structure includes, among other relevant information, the direct superclasses or the direct slots that where provided in the class definition:

```
> Class.slots
[:name, :direct_superclasses, :direct_slots, ...]
> ComplexNumber.name
:ComplexNumber
> ComplexNumber.direct_superclasses == [Object]
true
```

There are also additional metaobjects and corresponding metaclasses that describe other concepts, such as generic functions and methods:

```
> add
<GenericFunction add with 1 methods>
> class_of(add) === GenericFunction
true
> GenericFunction.slots
[:name, :methods, ...]
> class_of(add.methods[1]) === MultiMethod
true
> MultiMethod.slots
[:specializers, :procedure, :generic_function, ...]
> add.methods[1]
<MultiMethod add(ComplexNumber, ComplexNumber)>
> add.methods[1].generic_function === add
true
```

2.7 Class Options

As mentioned before, classes are created using the macro `@defclass`. However, the macro supports additional options that are important to know. In fact, the macro takes the name of the class to define, a possibly empty vector of superclasses, a possibly empty vector of slot definitions with corresponding options, and, optionaly, a metaclass specification.

Here is one example:

```
@defclass(Person, [],
  [[name, reader=get_name, writer=set_name!],
   [age, reader=get_age, writer=set_age!, initform=0],
   [friend, reader=get_friend, writer=set_friend!]],
   metaclass=UndoableClass)
```

Once again, note that an empty vector of superclasses is equivalent to a vector containing the `Object` class.

In the class definition, each slot is described by a vector containing the name of the slot followed by a possibly empty sequence of keyword arguments specifying readers, writers, and initforms, as exemplified above. Note that a slot without `initform` is equivalent to a slot with `missing` as `initform`.

To simplify class definitions, an explicit `initform` option can be replaced by an equivalent slot initialization. This means that, instead of writing the explicit slot definition

```
[foo, reader=get_foo, writer=set_foo!, initform=123]
```

one can write the shorter, but equivalent definition

```
[foo=123, reader=get_foo, writer=set_foo!]
```

Additionally, if no other options are provided, the definition can be further simplified by droping the vector syntax. This means that `foo=123` is equivalent to `[foo, initform=123]` and `foo` is equivalent to `[foo]`.

The metaclass option is used to specify the class of the class being defined, i.e., the metaclass of the class' instances.

Besides creating the class, a `@defclass` form also creates a global variable with the same name of the class that references the class, as demonstrated in the following interaction:

```
> Person
<UndoableClass Person>
```

Notice how classes are printed. That happens because there is a default printing method specialized for classes:

```
@defmethod print_object(class::Class, io) =
  print(io, "<$(class_name(class_of(class))) $(class_name(class))>")
```

The previous method definition also explains the following behavior:

```
> class_of(Person)
<Class UndoableClass>
> class_of(class_of(Person))
<Class Class>
```

2.8 Readers and Writers

When a class definition specifies readers and writers, these become methods of the corresponding generic functions, but specialized for that class.

As an example, the previous class definition automatically generates the following methods:

```
@defmethod get_name(o::Person) = o.name
@defmethod set_name!(o::Person, v) = o.name = v
@defmethod get_age(o::Person) = o.age
@defmethod set_age!(o::Person, v) = o.age = v
@defmethod get_friend(o::Person) = o.friend
@defmethod set_friend!(o::Person, v) = o.friend = v
```

As a result, it is possible to use the corresponding generic functions:

```
> get_age(new(Person))
0
> get_name(new(Person))
missing
```

2.9 Generic Function Calls

The application of a generic function to concrete arguments, by default, involves multiple steps. Given a generic function *gf* and a list of arguments *args*, the default behavior executes the following steps: (1) selects the set of applicable methods of *gf*, (2) sorts them according to their specificity, and, finally, (3) applies the most specific method to *args* while allowing that method to call the next most specific method using the `call_next_method` function. If there is no applicable method, the generic function `no_applicable_method` is called, using the *gf* and *args* as its two arguments. The default behavior of that function is to raise an error.

As an example, considering the previous method definition for the generic function `add`, the following interaction should be expected:

```
> add(123, 456)
ERROR: No applicable method for function add with arguments (123, 456)
```

2.10 Multiple Dispatch

Note that the actual method that ends up being called by a generic function is determined using multiple dispatch. As an example, consider the following program:

```
@defclass(Shape, [], [])
@defclass(Device, [], [])

@defgeneric draw(shape, device)

@defclass(Line, [Shape], [from, to])
@defclass(Circle, [Shape], [center, radius])

@defclass(Screen, [Device], [])
@defclass(Printer, [Device], [])

@defmethod draw(shape::Line, device::Screen) = println("Drawing a Line on Screen")
@defmethod draw(shape::Circle, device::Screen) = println("Drawing a Circle on Screen")
@defmethod draw(shape::Line, device::Printer) = println("Drawing a Line on Printer")
@defmethod draw(shape::Circle, device::Printer) = println("Drawing a Circle on Printer")

let devices = [new(Screen), new(Printer)],
    shapes = [new(Line), new(Circle)]
  for device in devices
    for shape in shapes
      draw(shape, device)
    end
  end
end
```

The execution of the previous program produces the following output:

```
Drawing a Line on Screen
Drawing a Circle on Screen
Drawing a Line on Printer
Drawing a Circle on Printer
```

You might assume that precedence among applicable methods is determined by left-to-right consideration of the parameter types. Method m_1 is more specific than method m_2 if the type of the first parameter of m_1 is more specific than the type of the first parameter of m_2. If they are identical types, then specificity is determined by the next parameter, and so on.

2.11 Multiple Inheritance

A class can inherit from multiple superclasses, which is important to support *mixins*. Consider the following extension to the previous example:

```
@defclass(ColorMixin, [],
  [[color, reader=get_color, writer=set_color!]])

@defmethod draw(s::ColorMixin, d::Device) =
  let previous_color = get_device_color(d)
    set_device_color!(d, get_color(s))
    call_next_method()
    set_device_color!(d, previous_color)
  end

@defclass(ColoredLine, [ColorMixin, Line], [])
@defclass(ColoredCircle, [ColorMixin, Circle], [])

@defclass(ColoredPrinter, [Printer],
  [[ink=:black, reader=get_device_color, writer=_set_device_color!]])

@defmethod set_device_color!(d::ColoredPrinter, color) = begin
  println("Changing printer ink color to $color")
  _set_device_color!(d, color)
end

let shapes = [new(Line), new(ColoredCircle, color=:red), new(ColoredLine, color=:blue)],
    printer = new(ColoredPrinter, ink=:black)
  for shape in shapes
    draw(shape, printer)
  end
end
```

The execution of the combined program prints the following:

```
Drawing a Line on Printer
Changing printer ink color to red
Drawing a Circle on Printer
Changing printer ink color to black
Changing printer ink color to blue
Drawing a Line on Printer
Changing printer ink color to black
```

Note that when the generic function `draw` is called using a `ColoredCircle` and a `ColoredPrinter`, there are multiple applicable methods, namely, one specialized for `(Circle, Printer)` and another for `(ColorMixin, Device)`. Due to the local ordering of direct superclasses, the second method is more specific than the first and, thus, is the method that is called. However, after changing the device color, this method calls `call_next_method` which calls the next most specific method, in this case, the first one of the list of applicable methods. When the called method returns, ending the `call_next_method` operation, the caller resumes execution, restoring the device's color.

2.12 Class Hierarchy

Given that each class can inherit from multiple superclasses, the resulting class hierarchy is a graph. We can traverse that graph by following the `direct_superclasses` slots that is contained in each class. However, it is important to know that this graph is finite and, thus, there must be a class that does not inherit from any other class. This is visible in the following script:

```
julia> ColoredCircle.direct_superclasses
[<Class ColorMixin>, <Class Circle>]
> ans[1].direct_superclasses
[<Class Object>]
> ans[1].direct_superclasses
[<Class Top>]
> ans[1].direct_superclasses
[]
```

As is visible, `Object` is a subclass of `Top` which is at the root of the class hierarchy. Directly or indirectly, all (regular) classes inherit from `Object`. There are special classes, however, that support the use of native Julia values as arguments to generic function calls. These special classes do not inherit from `Object` but from `Top` instead.

2.13 Class Precedence List

To support multiple inheritance, it is necessary to compute the *class precedence list* of a class. The generic function `compute_cpl` computes it for any class given as an argument. The result is an array that contains the class itself followed by a linearization of the class's superclass graph, i.e., the set of classes that are reachable from that class by following the superclass relation. The default behavior is to do that linearization in a *breadth-first* manner. As an example, consider the following hierarchy:

```
@defclass(A, [], [])
@defclass(B, [], [])
@defclass(C, [], [])
@defclass(D, [A, B], [])
@defclass(E, [A, C], [])
@defclass(F, [D, E], [])
```

and the following interaction:

```
> compute_cpl(F)
[<Class F>, <Class D>, <Class E>,
 <Class A>, <Class B>, <Class C>,
 <Class Object>, <Class Top>]
```

The class precedence list is critical to determine not only the slots that were inherited from superclasses (while avoiding unnecessary duplications), but also to sort generic function methods according to the types of the arguments.

2.14 Built-In Classes

To be more useful, JOS provides special classes that represent some of Julia's pre-defined types:

```
> class_of(1)
<BuiltInClass _Int64>
> class_of("Foo")
<BuiltInClass _String>
```

Note that, to avoid collisions with the corresponding native Julia types, these classes have names with the underscore `_` prefix. Additionally, note that these classes are instances of the metaclass `BuiltInClass`.

The goal of these classes is not to allow the creation of instances, but, instead, to permit the specialization of generic functions that take primitive Julia values as arguments.

For example:

```
@defmethod add(a::_Int64, b::_Int64) = a + b
@defmethod add(a::_String, b::_String) = a * b

> add(1, 3)
4
> add("Foo", "Bar")
"FooBar"
```

2.15 Introspection

Given that JOS uses metaobjects to store information about itself, it is trivial to introspect it. This was demonstrated above by reading the contents of the slots `direct_superclasses` and `direct_slots` that exist in all classes.

However, to provide a more funcional interface, JOS also provides the following (non-generic) functions:

```
> class_name(Circle)
:Circle
> class_direct_slots(Circle)
[:center, :radius]
> class_direct_slots(ColoredCircle)
[]
> class_slots(ColoredCircle)
[:color, :center, :radius]
> class_direct_superclasses(ColoredCircle)
[<Class ColorMixin>, <Class Circle>]
> class_cpl(ColoredCircle)
[<Class ColoredCircle>, <Class ColorMixin>, <Class Circle>,
 <Class Object>, <Class Shape>, <Class Top>]
> generic_methods(draw)
[<MultiMethod draw(ColorMixin, Device)>, <MultiMethod draw(Circle, Printer)>,
 <MultiMethod draw(Line, Printer)>, <MultiMethod draw(Circle, Screen)>,
 <MultiMethod draw(Line, Screen)>]
> method_specializers(generic_methods(draw)[1])
[<Class ColorMixin>, <Class Device>]
```

2.16 Meta-Object Protocols

The critical aspect of this work is the support for metaobject protocols. In fact, most of the behavior of the system is controled by generic functions specialized in different metaclasses. These generic functions include the allocation of objects (`allocate_instance`), the initialization of objects (`initialize`), the computation of the class precedence list (`compute_cpl`), etc. The following sections illustrate the use of some of these protocols.

2.16.1 Class Instantiation Protocol

The non-generic function `new` is responsible for creating instances of classes (and, therefore, also instances of metaclasses, i.e., classes). It receives the class to instantiate and a series of keyword arguments, calls the generic functions `allocate_instance` to create the non-initialized instance, and then calls the generic function `initialize` to initialize it using the keyword arguments. It is because JOS' generic functions do not support keyword arguments that `new` is non-generic. Its definition is as follows:

```
new(class; initargs...) =
  let instance = allocate_instance(class)
    initialize(instance, initargs)
    instance
  end
```

The pre-defined default behavior of regular classes results from the methods that specialize the above-mentioned generic functions, particularly when the first argument is a direct or indirect instance of `Class` or `Object`. Those methods should have the following (incomplete) definitions:

```
@defmethod allocate_instance(class::Class) = ???

@defmethod initialize(object::Object, initargs) = ???
@defmethod initialize(class::Class, initargs) = ???
```

Given that other metaobjects, such as generic functions and methods, might also need a specialized initialization, it is expected that aditional specializations exist, namely:

```
@defmethod initialize(generic::GenericFunction, initargs) = ???
@defmethod initialize(method::MultiMethod, initargs) = ???
```

The interesting feature, however, is that these default implementations can be extended. As an example, consider a *counting class* that counts how many instances were created from it. To implement this behavior, we start by creating a subclass of `Class` that adds a `counter` slot to each different counting class:

```
@defclass(CountingClass, [Class],
 [counter=0])
```

We now specialize the `allocate_instance` generic function so that each time `allocate_instance` is called using a class that is, directly or indirectly, an instance of `CountingClass`, the class' counter is incremented:

```
@defmethod allocate_instance(class::CountingClass) = begin
  class.counter += 1
  call_next_method()
end
```

It is now possible to create classes that count their instances, as follows:

```
> @defclass(Foo, [], [], metaclass=CountingClass)
<CountingClass Foo>
> @defclass(Bar, [], [], metaclass=CountingClass)
<CountingClass Bar>
> new(Foo)
<Foo GKWqdH65Ot6>
> new(Foo)
<Foo 1bfnBI6LHyS>
> new(Bar)
<Bar 1MOzFoH7eZ5>
> Foo.counter
2
> Bar.counter
1
```

2.16.2 The Compute Slots Protocol

The slots of a class include not only those that are directly declared at the class definition (the so-called *direct slots*) but also those that are inherited from its superclasses. The way the array of slots is computed is mediated by the generic function `compute_slots`, which takes a class as an argument and is automatically invoked at class initialization time. This function is specialized for the class `Class` as follows:

```
@defmethod compute_slots(class::Class) =
  vcat(map(class_direct_slots, class_cpl(class))...)
```

This behavior is sufficient for most cases but it might produce surprising effects when there are name colisions between slots. Consider the following example:

```
> @defclass(Foo, [], [a=1, b=2])
<Class Foo>
> @defclass(Bar, [], [b=3, c=4])
<Class Bar>
> @defclass(FooBar, [Foo, Bar], [a=5, d=6])
<Class FooBar>
> class_slots(FooBar)
[:a, :d, :a, :b, :b, :c]
> foobar1 = new(FooBar)
<FooBar 6i1nVhjh6o>
> foobar1.a
1
> foobar1.b
3
> foobar1.c
4
> foobar1.d
6
```

Unfortunately, it is not easy to decide which of the duplicated slots should be used and one way to handle these situations is to raise an error. This is easily done with a collision-detection metaclass:

```
@defclass(AvoidCollisionsClass, [Class], [])

@defmethod compute_slots(class::AvoidCollisionsClass) =
  let slots = call_next_method(),
      duplicates = symdiff(slots, unique(slots))
    isempty(duplicates) ?
      slots :
      error("Multiple occurrences of slots: $(join(map(string, duplicates), ", "))")
  end
```

Using this metaclass, we now get a different behavior:

```
> @defclass(FooBar2, [Foo, Bar], [a=5, d=6], metaclass=AvoidCollisionsClass)
ERROR: Multiple occurrences of slots: a, b
```

2.16.3 Slot Access Protocol

To allow flexibility in the representation of instances, for example, to support the dictionary-based approach used in Python, it is necessary to have a protocol that mediates the way slots are accessed. Diferently from the CLOS MOP that calls the generic function `slot-value-using-class` every time an instance's slot is accessed, JOS uses a more efficient approach, calling the generic function `compute_getter_and_setter` just once (at class initialization time) for each different slot. This function receives the class object, the slot name, and an index corresponding to the position of that slot in the slots of the class. This index should be used for array-based representations of instances but might be ignored for dictionary-based representations. The generic function must return a tuple of non-generic functions, where the first one gets the value of a slot and the second one sets the value of a slot. The first function takes an instance as an argument while the second one takes an instance and a new value to place in the corresponding slot.

As a complete example of the use of this protocol, consider the problem of creating *undoable classes*. First, we present the support code:

```
undo_trail = []

store_previous(object, slot, value) = push!(undo_trail, (object, slot, value))

current_state() = length(undo_trail)

restore_state(state) =
  while length(undo_trail) != state
    restore(pop!(undo_trail)...)
  end

save_previous_value = true

restore(object, slot, value) =
  let previous_save_previous_value = save_previous_value
    global save_previous_value = false
    try
      setproperty!(object, slot, value)
    finally
      global save_previous_value = previous_save_previous_value
    end
  end
```

Now, we can create an `UndoableClass` metaclass as a subclass of the `Class` metaclass.

```
@defclass(UndoableClass, [Class], [])
```

Finally, we only need to specialize the generic function `compute_getter_and_setter` for the new metaclass:

```
@defmethod compute_getter_and_setter(class::UndoableClass, slot, idx) =
  let (getter, setter) = call_next_method()
    (getter,
     (o, v)->begin
              if save_previous_value
                store_previous(o, slot, getter(o))
              end
              setter(o, v)
            end)
  end
```

We can now use this on a specific example:

```
@defclass(Person, [],
  [name, age, friend],
  metaclass=UndoableClass)

@defmethod print_object(p::Person, io) =
  print(io, "[$(p.name), $(p.age)$(ismissing(p.friend) ? "" : " with friend $(p.friend)")]")

p0 = new(Person, name="John", age=21)
p1 = new(Person, name="Paul", age=23)
#Paul has a friend named John
p1.friend = p0
println(p1) #[Paul,23 with friend [John,21]]
state0 = current_state()
#32 years later, John changed his name to 'Louis' and got a friend
p0.age = 53
p1.age = 55
p0.name = "Louis"
p0.friend = new(Person, name="Mary", age=19)
println(p1) #[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
state1 = current_state()
#15 years later, John (hum, I mean 'Louis') died
p1.age = 70
p1.friend = missing
println(p1) #[Paul,70]
#Let's go back in time
restore_state(state1)
println(p1) #[Paul,55 with friend [Louis,53 with friend [Mary,19]]]
#and even earlier
restore_state(state0)
println(p1) #[Paul,23 with friend [John,21]]
```

2.16.4 Class Precedence List Protocol

When a class is initialized, its class precedence list is computed by calling the generic function `compute_cpl` with the class object and is then cached in the class object. The default behavior, implemented on the specialization of the generic function for instances of `Class`, is to produce the linearization of the class graph that is reachable from the class by following the superclass relation in a *breadth-first* manner. This is similar but not entirely identical to the approaches used in CLOS or in Dylan, which use more complex algorithms.

The good news is that the pre-defined behavior is not the only one that is possible. As an example, we will implement the Flavors approach to the class precedence list, which requires a depth-first walk of the inheritance graph, from left to right, with duplicates removed from the right and with `Object` and `Top` appended on the right.

Given that we just want to change the way the `compute_cpl` operates, while preserving the existing behavior for the remaining protocols, we start by creating a new (meta)class that inherits from `Class`:

```
@defclass(FlavorsClass, [Class], [])
```

We then specialize the generic function `compute_cpl` so that, for classes that are instances of `FlavorsClass`, the class precedence list is computed according to the Flavors strategy:

```
@defmethod compute_cpl(class::FlavorsClass) =
  let depth_first_cpl(class) =
        [class, foldl(vcat, map(depth_first_cpl, class_direct_superclasses(class)), init=[])...],
      base_cpl = [Object, Top]
    vcat(unique(filter(!in(base_cpl), depth_first_cpl(class))), base_cpl)
  end
```

Just to test if everthing is working correctly, we redefine the hierarchy presented in section 2.13 but now using the Flavors metaclass:

```
@defclass(A, [], [], metaclass=FlavorsClass)
@defclass(B, [], [], metaclass=FlavorsClass)
@defclass(C, [], [], metaclass=FlavorsClass)
@defclass(D, [A, B], [], metaclass=FlavorsClass)
@defclass(E, [A, C], [], metaclass=FlavorsClass)
@defclass(F, [D, E], [], metaclass=FlavorsClass)
```

This time, the computed class precedence list is the following:

```
> compute_cpl(F)
[<FlavorsClass F>, <FlavorsClass D>, <FlavorsClass A>,
 <FlavorsClass B>, <FlavorsClass E>, <FlavorsClass C>,
 <Class Object>, <Class Top>]
```

2.17 Multiple Meta-Class Inheritance

In the previous examples, different protocols where specialized by first creating adequate meta-classes. Obviously, nothing prevents us from using multiple inheritance at the metaclass level. For example, to have an undoable, collision-avoiding, counting class, we can write:

```
@defclass(UndoableCollisionAvoidingCountingClass,
  [UndoableClass, AvoidCollisionsClass, CountingClass],
  [])
```

Then, we can create classes that are instances of the `UndoableCollisionAvoidingCountingClass`:

```
> @defclass(NamedThing, [], [name])
<Class NamedThing>
> @defclass(Person, [NamedThing],
    [name, age, friend],
    metaclass=UndoableCollisionAvoidingCountingClass)
ERROR: Multiple occurrences of slots: name
> @defclass(Person, [NamedThing],
    [age, friend],
    metaclass=UndoableCollisionAvoidingCountingClass)
<UndoableCollisionAvoidingCountingClass Person>
> @defmethod print_object(p::Person, io) =
    print(io, "[$(p.name), $(p.age)$(ismissing(p.friend) ? "" : " with friend $(p.friend)")]")
<MultiMethod print_object(Person)>
```

Now, we can create instances of those classes, do some slot changes, and undo those changes, just like we did for `UndoableClasses`.

```
p0 = new(Person, name="John", age=21)
p1 = new(Person, name="Paul", age=23)
#Paul has a friend named John
p1.friend = p0
println(p1) #[Paul,23 with friend [John,21]]
state0 = current_state()
#32 years later, John changed his name to 'Louis' and got a friend
...
#and even earlier
```

```
restore_state(state0)
println(p1) #[Paul,23 with friend [John,21]]
```

The difference, now, is that we can also know how many instances were created from class `Person`:

```
> Person.counter
3
```

2.18 Extensions

You can extend your project to further increase your grade. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections. To ensure this behavior, you should implement all your extensions in a different file.

Some of the potentially interesting extensions include:

- Implement meta-objects for slot definitions.
- Implement CLOS-like method combination for generic functions.
- Implement the strategies used in CLOS or Dylan for computing the class precedence list.
- Implementing additional metaobject protocols.

3 Code

Your implementation must work in Julia, version 1.8.3.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

4 Presentation

For this project, a full report is not required. Instead, a presentation is required. This presentation should be prepared for a hypothetical 20-minute-long slot, should be centered in the architectural decisions taken and might include all the details that you consider relevant. The presentation should be sufficient to "sell" your solution to anyone that watches it.

5 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named as `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing the source code, within subdirectory `/src/`.

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `presentation.pdf`.

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the presentation.

In case of doubt, the teacher might request explanations about the inner working of the developed project, including demonstrations.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues or other sources as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code must be submitted via Fénix, no later than 23:00 of **April**, **16**. Similarly, the presentation must be submitted via Fénix, no later than 23:00 of **April**, **16**.

References

[1] Giuseppe Attardi, Cinzia Bonini, Maria Rosario Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel Programming in CLOS. In S. Cook, editor, *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 243–256, Nottingham, July 1989. Cambridge University Press.

[2] D. G. Bobrow, R. P. Gabriel, and J. L. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[3] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A programmer's guide to CLOS*. Addison-Wesley Publishing Company, Cambridge, MA, 1989.

[4] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[5] Andreas Paepcke. PCLOS: Stress testing CLOS experiencing the metaobject protocol. In *OOPSLA/ECOOP*, pages 194–211, 1990.

[6] Andreas Paepcke. User-level language crafting. In *Object-Oriented Programming: the CLOS perspective*, pages 66–99. MIT Press, 1993.