JOS - Julia Object System

Group 6:
90115 João Pedro Lopes Ivo
95569 Eduardo Miranda
95666 Rodrigo Pinto
105767 Alexandre Serras

1.1 Introduction

The primary goal is to implement JOS, or the Julia Object System, a Julia programming language extension that supports classes and metaclasses, multiple inheritance, and generic functions with multiple-dispatch methods. This implementation should be done in the same way that those ideas were implemented in CLOS.

2.1 Classes

Every class, including the Class class, is an instance of the struct Class1 that basically stores all the information that is passed in the creation of a class.

```
mutable struct Class1
    name::Symbol
    direct_superclasses::Array{Any}
    direct_slots::Array{Symbol}
    metaclass::Union{Class1,Missing}
    default::Union{Any,Nothing}
    getters_and_setters::Union{Dict{Symbol,Any},Nothing}
    cpl::Union{Array{Any},Nothing}
    slots::Array{Any}
end
```

```
Top = Class1(:Top,[],[],missing,nothing)
push!(class1_instances,Top)

Object = Class1(:Object,[Top],[],missing,nothing)
push!(class1_instances,Object)

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

2.2 Instances

To create a new instance of a class, we defined the new function that creates a new instance of the struct Instance with the class that is received in new and then initialize the corresponding arguments, putting them in the fields dictionary.

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

```
mutable struct Instance
    class::Class1
    fields::Dict{Symbol,Any}
end
```

2.3 Slot Access

We had to redefine the Base.getproperty and Base.setproperty! to be able to access the fields dictionary of the instance and find the value of the corresponding field.

```
function Base.getproperty(instance::Instance, field::Symbol)
    return getfield(instance,:class).getters_and_setters[field][1](instance)
end
```

```
function Base.setproperty!(instance::Instance, field::Symbol, value)
    getfield(instance,:class).getters_and_setters[field][2](instance, value)
end
```

```
@defmethod compute_getter_and_setter(class::Class, slot, index) = begin
    function get(instance)
        getfield(instance,:fields)[slot]
    end
    function set!(instance, value)
        getfield(instance,:fields)[slot] = value
    end
    return (get, set!)
end
```

2.4 Generic Functions and Methods

Every generic function , including the generic function class itself, is an instance of the struct GenericFunction1 that basically stores all the information relevant to the generic function.

```
mutable struct GenericFunction1
    name::Symbol
    methods::Array{Any}
    slots::Array{Any}
    last_method::Dict{String,Any}
    sorted_methods::Array{MultiMethod1}
end
GenericFunction = GenericFunction1(:GenericFunction,[],[:name,:methods])
```

2.4 Generic Functions and Methods

Every method, including the MultiMethod class itself, is an instance of the struct MultiMethod1 that stores all the information relevant to the method.

```
mutable struct MultiMethod1
    name::Symbol
    specializers::Array{Any}
    procedure::Function
    generic_function::Any
    slots::Array{Any}
end
```

```
MultiMethod = MultiMethod1(:MultiMethod,[],(x)->nothing,GenericFunction,[:specializers,:procedure,:generic_function])
```

@defclass

This macro takes as input the name of a class, along with its direct superclasses and slots, and an optional metaclass. Upon receiving this input, the macro processes the slots to create necessary data structures for class creation. After creating the data structures, the macro creates the class object, which is made as a global variable. Finally, the macro generates all the required readers and writers based on the information parsed from the slots.

```julia
macro defclass(name, superclasses, slots, metaclass=missing)
    global_sym = name
    simple_slots = []
    init = Dict{Symbol,Any}([])
    readers = Dict()
    writers = Dict()
    #After getting the data now transform the slots
    if slots.args != []
        simple_slots = filter(slot -> isa(slot, Symbol), slots.args) # Slots that are symbols
        complex_slots = filter(slot -> !isa(slot, Symbol), slots.args) # Slots that are expressions
        push!(simple_slots, [isa(slot.args[1], Symbol) ? slot.args[1] : slot.args[1].args[1] for slot in complex_slots]...)
        [init[slot] = missing for slot in simple_slots]
        for slot in complex_slots
            if slot.head === :(=) # Handles cases like @defclass(A, [], [a=1, b=2])
                init[slot.args[1]] = slot.args[2]
                continue
            elseif slot.head === :vect
                field = slot.args[1] # Can be either a symbol (ex. :a) or an expression (ex. :(a=1))
                if !isa(slot.args[1], Symbol) && slot.args[1].head === :(=) # Handles cases like @defclass(A, [], [[a=1, ...], [b=2, ...]])
                    field = slot.args[1].args[1] # Becomes just a symbol (ex. a=1 --> :a)
                    init[field] = slot.args[1].args[2]
                end
                for option in slot.args
                    if !isa(option, Symbol)
                        # Handles cases like @defclass(A, [], [[a, ..., initform=1], [b, ..., initform=2]])
                        if option.args[1] == :initform
                            init[field] = option.args[2]
                        # Handles cases like @defclass(A, [], [[a, reader=get_a, ...], [b, reader=get_b, ...]])
                        elseif option.args[1] == :reader
                            readers[field] = option.args[2]
                        # Handles cases like @defclass(A, [], [[a, writer=set_a!, ...], [b, writer=set_b!, ...]])
                        elseif option.args[1] == :writer
                            writers[field] = option.args[2]
                        end
                    end
                end
            end
        end
    end
    sym_name = Symbol(name)
    # generate the readers and writers expressions to be called on the quote block
    expr_readers = []
    for (field, reader) in readers
        aux = :(@defmethod $(Expr(:function, Expr(:call,reader,:(o::$sym_name)), Expr(:call,:getproperty,:o, QuoteNode(field)))))
        push!(expr_readers, aux)
    end

    expr_writers = []
    for (field, reader) in writers
        aux = :(@defmethod $(Expr(:function, Expr(:call,reader,:(o::$sym_name),:v), Expr(:call,:setproperty!,:o, QuoteNode(field),:v))))
        push!(expr_writers, aux)
    end
    #Generate the class itself
    quote
        if $superclasses == []
            if $metaclass !== missing
                global $global_sym = Class1($(QuoteNode(sym_name)), [$Object], $simple_slots, $metaclass, $init)
            else
                global $global_sym = Class1($(QuoteNode(sym_name)), [$Object], $simple_slots, missing, $init)
            end
        else
            if $metaclass !== missing
                global $global_sym = Class1($(QuoteNode(sym_name)), $superclasses, $simple_slots, $metaclass, $init)
            else
                global $global_sym = Class1($(QuoteNode(sym_name)), $superclasses, $simple_slots, missing, $init)
            end
        end
        push!($class1_instances,$global_sym)
        $(expr_readers...)
        $(expr_writers...)
    end
```

```julia
@defclass(FooBar2, [Foo, Bar], [a=5, d=6], metaclass=AvoidCollisionsClass)
```

@defgeneric

This macro just get the name of the generic function that we want to create and create an instance of a GenericFunction1 and after that we make that new generic function global with that is possible to access from anywhere.

```
generic_functions=[]
macro defgeneric(name...)
    name=name[1].args[1]
    global_sym = name
    class_obj = GenericFunction1(name, [], [:name,:methods])
    push!(generic_functions,class_obj)
    quote
        global $global_sym = $class_obj
    end
end

@defgeneric add(a,b)
```

@defmethod

To define a new method first of all , we want to know if the associated generic function is already created, if not we need to create one. After this we need to get the reference for the type of the Objects that the new method is going to have , like if it is a ComplexNumber we need that reference. When all of this is done we simply create a new instance of a MultiMethod 1 and we associate this method to the generic function.To instance a new method is very simple

```
macro defmethod(name)
    info=name.args[1]
    data=name.args[2]
    tmp_gen =info.args[1]
    exists=false
    #CHECKING IF THE GENERIC FUNCTION EXISTS
    for  generic in generic_functions
        if generic.name == tmp_gen
            tmp_gen=generic
            exists=true
            break
        end
    end
    #GETTING THE GENERIC FUNCTION
    if exists == false
        global_sym = tmp_gen
        class_obj = GenericFunction1(tmp_gen, [], [:name,:methods])
        push!(generic_functions,class_obj)
        tmp_gen=class_obj
    end
```

```
    # putting the arguments into the right types
    slots_name::Vector{Symbol} = []
    slots_type=[]
    for slots in info.args[2:end]
        #println(typeof(slots))
        if string(slots)=="io"
            push!(slots_name,slots)
        elseif typeof(slots) == Symbol
            push!(slots_name,slots)
        else
            slots_data=slots.args
            push!(slots_name,slots_data[1])
            type=string(slots_data[2])
            for data_type in class1_instances
                if type == string(data_type.name)
                    push!(slots_type,data_type)
                end
            end
        end
    end
    res = Tuple(slots_name)
    expr = Expr(:function, Expr(:tuple, Symbol.(res)...), data)

    if exists == false
        quote
            global $global_sym = $class_obj
            push!($tmp_gen.methods,MultiMethod1($tmp_gen.name,$slots_type,
            $expr , $tmp_gen, [:specializers,:procedure,:generic_function]))
        end |> esc
    else
        quote
            push!($tmp_gen.methods,MultiMethod1($tmp_gen.name,$slots_type,
            $expr , $tmp_gen, [:specializers,:procedure,:generic_function]))
        end |> esc
    end
```

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

2.5 Pre-defined Generic Functions and Methods

When we start the program there are some generic functions and methods that are already defined for example the print_object generic function and 3 print_object methods, but we can add new ones by using the @defmethod macro.

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

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

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

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

2.6 MetaObjects

To replicate the behaviour shown we had to implement the class_of function that receives an object as argument. If the object is a class it will return the metaclass if it exists or Class since every class is an instance of the class Class including itself. If the object is an Instance it will return the class stored in the Instance struct.

```
function class_of(instance)
    if typeof(instance) === Int64
        return _Int64
    elseif typeof(instance) === String
        return _String
    elseif typeof(instance) === Symbol
        return _Symbol
    elseif typeof(instance) === Instance
        return getfield(instance,:class)
    elseif typeof(instance) === Class1
        if getfield(instance,:metaclass) === missing
            return Class
        else
            return getfield(instance,:metaclass)
        end
    elseif typeof(instance) === GenericFunction1
        return GenericFunction
    elseif typeof(instance) === MultiMethod1
        return MultiMethod
    end
end
```

2.7 Class Options

We can initialize a class with some optional arguments, for example the metaclass, initform (initial value of a field) and the names of the reader and writer of a field.

```
@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)
```

2.9, 2.10 and 2.11 Generic Function Calls and Multiple Dispatch and Multiple Inheritance

Every time a generic function is called with certain arguments we start by getting the applicable_methods for this arguments and then we sort the applicable methods and call the procedure of the first method in the list (the most specific). If we don't find any applicable methods an error is displayed. Another details is that the last_method and the arguments are stored for the case in which the procedure calls the next method. The gen_funs list is used if a different generic function is called inside this generic function method.

```
function no_aplicable_method(gf, args)
    filter!(x -> x != gf, gen_funs)
    throw("No applicable method for $(gf.name) with arguments $(args)")
end
```

```
add(123,456) | ERROR: "No applicable method for add with arguments (123, 456)"
```

```
gen_funs = []
function (gf::GenericFunction1)(args...)
    global gen_funs
    push!(gen_funs, gf)

    applicable_methods = get_applicable_methods(gf, args...)
    gf.sorted_methods = sortmethods(applicable_methods, args...)
    method = gf.sorted_methods[1]
    gf.last_method["method"] = 1
    gf.last_method["args"] = args

    result = method.procedure(args...)

    filter!(x -> x != gf, gen_funs)
    return result
end
```

2.9, 2.10 and 2.11 (get_applicable_methods)

In this function we start by checking if the generic function is print_object and if it is we decrement the size by 1 because io has no specializers. Then we iterate over the methods in the generic function and using the class precedence list for the class of the current argument we check if the specializer is in the list. If this happens to all the arguments we add the method to the applicable methods list.

```julia
function get_applicable_methods(a::GenericFunction1, args...)
    applicable_methods = []
    size = length(args)
    if a === print_object
        size = size - 1
    end
    for method in a.methods
        k=0
        for i in 1:size
            if i > length(method.specializers)
                continue
            end
            applicable_classes = compute_cpl_normal(class_of(args[i]))
            if method.specializers[i] in applicable_classes
                k+=1
            else
                break
            end
        end
        if k == length(method.specializers)
            push!(applicable_methods, method)
        end
    end

    if length(applicable_methods) == 0
        no_aplicable_method(a, args)
    end
    return applicable_methods
end
```

2.9, 2.10 and 2.11 (sortmethods)

To sort all the applicable methods from most specific to least specific we implemented a bubble sort where the operator of decision is the appears_last function that confirms that the first argument appears after the second argument of the function in the cpl.

```julia
function sortmethods(applicable_methods,args...)
    sorted = []
    sorted = applicable_methods
    n = length(sorted)

    for i in 1:n

        for j in 1:n-i

            index = 1
            while sorted[j].specializers[index] === sorted[j+1].specializers[index]
                index += 1
            end
            if appears_last(sorted[j], sorted[j+1],compute_cpl_normal(class_of(args[index])), index)
                sorted[j], sorted[j+1] = sorted[j+1], sorted[j]
            end
        end
    end
    return sorted
end
```

2.9, 2.10 and 2.11 (call_next_method)

As explained above the generic function stores the last method that it used so in that in mind we can increment the position in the sorted methods vector to call the next applicable method. If there are no more methods in the list this function will call the no_applicable_method function.

```
function call_next_method()
    global gen_funs
    current_gen = gen_funs[end]

    methods = current_gen.sorted_methods
    size = length(methods)

    if current_gen.last_method["method"] == size
        return no_aplicable_method(current_gen, current_gen.last_method["args"])
    else
        current_gen.last_method["method"] += 1
        return methods[current_gen.last_method["method"]].procedure(current_gen.last_method["args"]...)
    end
end
```

2.12 Class Hierarchy

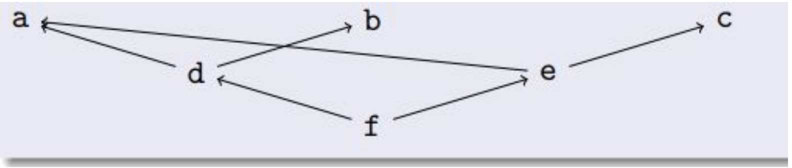
Classes can inherit from multiple classes resulting in a graph that is the class hierarchy however this graph is finite because the class that does not inherit from any other class is the class Top as shown in the image.

```
Top = Class1(:Top,[],[],missing,nothing)
push!(class1_instances,Top)

Object = Class1(:Object,[Top],[],missing,nothing)
push!(class1_instances,Object)

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

2.13 Class Precedence List

The default compute cpl function what it does is simply an breadth-first manner, in order to give the right precedence list, to know the best possible class in the inheritance to use in the situation.

```julia
function compute_cpl_normal(cls::Class1)
    queue=[]
    result=[cls]
    queue = vcat(queue,cls.direct_superclasses)
    result = vcat(result,cls.direct_superclasses)
    while length(queue) > 0
        current = queue[1]
        queue = queue[2:end]
        for superclass in current.direct_superclasses
            if superclass ∉ result
                all_precedents_included = all(x -> x in result, superclass.direct_superclasses[2:end])
                if all_precedents_included
                    push!(result, superclass)
                    push!(queue, superclass)
                end
            end
        end
    end
    tmp_result = []
    a=[]
    for r in result
        if (r.name) == :Top  || (r.name) == :Object
            push!(a,r)
        else
            push!(tmp_result, r)
        end
    end
    tmp_result =vcat(tmp_result,a)

    return tmp_result
end
```



2.14 Built-In Classes

To implement new Built-In classes we have to do 3 things firstly create the class, then we add it to the if of the constructor and finally add a new if to the class_of function.

```
BuiltInClass = Class1(:BuiltInClass,[Top],[])
push!(class1_instances,BuiltInClass)

_Int64 = Class1(:_Int64,[Top],[],BuiltInClass)
push!(class1_instances,_Int64)

_String = Class1(:_String,[Top],[],BuiltInClass)
push!(class1_instances,_String)

_Symbol = Class1(:_Symbol,[Top],[],BuiltInClass)
push!(class1_instances,_Symbol)
```

```
function Class1(name, direct_superclasses, direct_slots, metaclass)
    default_fields = Dict([])

    class = Class1(name, direct_superclasses, direct_slots, metaclas

    if metaclass !== missing && metaclass.default !== nothing
        default_fields = merge!(default_fields,copy(getfield(metacla
    end

    if name !== :_Int64 && name !== :_String && name !== :_Symbol
        class = class_computations(class, default_fields, name, dire
    end
    return class
end
```

```
function class_of(instance)
    if typeof(instance) === Int64
        return _Int64
    elseif typeof(instance) === String
        return _String
    elseif typeof(instance) === Symbol
        return _Symbol
    elseif typeof(instance) === Instance
```

2.15 Introspection

To allow for introspection we defined the function shown in the image.

```
function class_direct_slots(cls::Class1)
    return getfield(cls, :direct_slots)
end

function class_slots(cls::Class1)
    superclasses = compute_cpl(cls)
    slots = []
    for superclass in superclasses
        slots = vcat(slots, class_direct_slots(superclass))
    end
    return slots
end

function class_direct_superclasses(cls::Class1)
    return getfield(cls, :direct_superclasses)
end

function class_cpl(cls::Class1)
    return compute_cpl(cls)
end

function generic_methods(gf::GenericFunction1)
    return getfield(gf, :methods)
end

function method_specializers(method::MultiMethod1)
    return getfield(method, :specializers)
end
```

2.16 Default Protocols

In the image is shown the default behaviour of the JOS protocols. The behaviour can the be extended through metaclasses as we will show. But first let's see the class_computations function that allows for this to happen.

```
@defgeneric allocate_instance(class)

@defmethod allocate_instance(class::Class) = Instance(class)

@defgeneric compute_slots(class)

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

@defgeneric compute_getter_and_setter(class, slot, index)

@defmethod compute_getter_and_setter(class::Class, slot, index) = begin
    function get(instance)
        getfield(instance,:fields)[slot]
    end
    function set!(instance, value)
        getfield(instance,:fields)[slot] = value
    end
    return (get, set!)
end

@defgeneric compute_cpl(cls)

@defmethod compute_cpl(cls::Class) = compute_cpl_normal(cls)
```

2.16 class_computations

The function starts by computing the slots of the class. Then it will see if the superclasses have a default for any field, if yes it joins it to the current defaults. After this creates an Instance to store this defaults. It stores the class precedence list in the cpl field and computes getters and setters for each slot of the class.

```
function class_computations(class, default_fields, name, direct_superclasses, direct_slots)
    slots = compute_slots(class)
    for superclass in direct_superclasses
        if getfield(superclass,:default) !== nothing
            default_fields = merge!(default_fields,copy(getfield(superclass.default,:fields)))

        end
    end
    class.default = Instance(Class1(name, direct_superclasses, direct_slots, missing, nothing, nothing,[],
        [:name,:direct_superclasses,:direct_slots,:metaclass, :default, :getters_and_setters, :cpl]),default_fields)

    class.cpl = compute_cpl(class)
    gs = Dict([])
    for slot in slots
        gs[slot] = compute_getter_and_setter(class, slot, 0)
    end
    class.getters_and_setters = gs
    return class
end
```

2.16.1 Class Instantiation Protocol

By defining a new class called CountingClass and defining a new specific allocate_instance for this class. It is now possible to define new classes with CountingClass as the metaclass now we will be able to count the number of instances in each class.

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

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

@defclass(Foo, [], [], metaclass=CountingClass)
@defclass(Bar, [], [], metaclass=CountingClass)
```

2.16.2 The Compute Slots Protocol

The default compute_slots method doesn't check for collisions between the class fields and the superclasses fields. By defining a new class called AvoidCollisionsClass and defining a new specific compute_slots for this class. It is now possible to check if there will be collisions between fields in a class that has AvoidCollisionsClass as a metaclass before his creation.

```
@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
```

2.16.3 Slot Access Protocol

Following the same logic we can create a new class called UndoableClass, defining a new compute_getter_and_setter method specific for this class and using it as metaclass in new classes. We can using the support code create classes that save and restore state.

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

@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

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

2.16.4 Class Precedence List Protocol

We can also use different strategies for compute_cpl by creating a new class for example FlavorsClass and defining a new compute_cpl method specific to this class. Now every class that has this one has a metaclass will use the new strategy.

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

@defmethod compute_cpl(class::FlavorsClass) = begin
    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
end
```

2.17 Multiple Meta-Class Inheritance

We can also join all this behaviors on a single class by providing the classes in the superclass list. As shown in the following image:

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

@defclass(NamedThing, [], [name])

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

Validation of our results

We used all of the data that the teacher gave us to check our results, and we tried to implement the functions and classes in order to get the same results that he gave us.