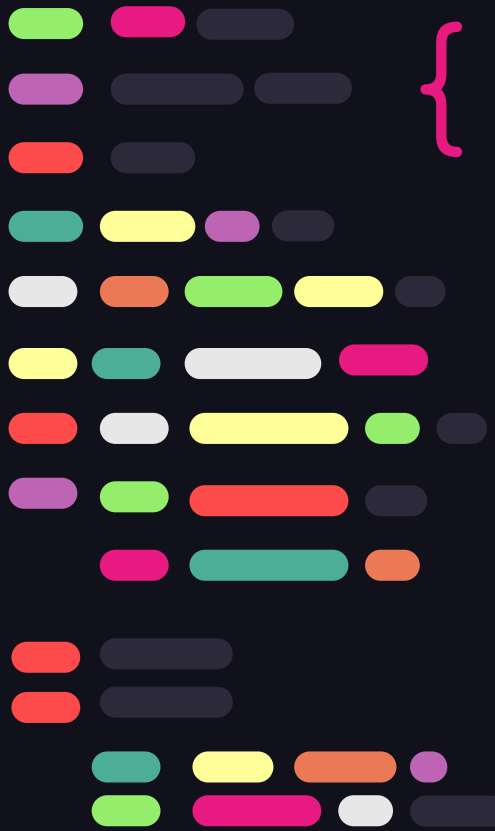
Advanced Programming



Julia Object System

GROUP 9
João Caldeira 93729
João Silveira 95597
Francisco António 105741

Prof. António Menezes Leitão

This talk is an introduction to our **object system in Julia (JOS)**. The main goal of this talk is to show how we can leverage Julia's features to implement a minimal object system.

We will focus on some of the most important **implementation details** and their **trade-offs**.

```
mutable struct JClass
    name::Symbol
    cpl::Vector{JClass}
    slots::Vector{Symbol}
    defaulted::Dict{Symbol,Any}
    meta::Union{Nothing,JClass}
    direct_slots::Vector{Symbol}
    meta_slots::Dict{Symbol,Any}
    getters::Dict{Symbol,Function}
    setters::Dict{Symbol,Function}
    direct_superclasses::Vector{JClass}
end
```

Firstly, we need to able to talk about classes. We chose to represent classes with a special struct type.

This will cause some problems later because classes and instances will not share the same uniform representation.

This representation is more efficient since it avoids having to store all the fields in a dictionary, but sacrifices flexibility.

JClass Getter and Setter

```
function Base.getproperty(cls::JClass, name::Symbol)
    try
        Base.getfield(cls, name)
    catch
        if haskey(cls.meta_slots, name)
            cls.meta_slots[name]
        else
            error("Invalid slot name: $name")
        end
    end
end
```

Next, we need to define Julia's **getproperty** and **setproperty!** for the **JClass** struct. This needs to be done precisely because classes can have slots specified by their metaclass.

The getter and setter will try to access the fields of the class struct first and, in case of failure, will fallback to the metaclass slots.

```
struct JInstance
    class::JClass
    slots::Dict{Symbol,Any}
end
```

We also need to be able to talk about instances of classes. We represent instances with another struct. This struct has a field for storing the class of the instance and a field for storing the slots specified by this class.

JInstacce Getter and Setter

```
function Base.getproperty(obj::JInstance,
slot::Symbol)
    slots = Base.getfield(obj, :slots)
    if haskey(slots, slot)
        class_of(obj).getters[slot](obj)
    else
        error("Invalid slot name: $slot")
    end
end
```

We also need to define Julia's **getproperty** and **setproperty!** for the **JInstance** struct.

This function calls the getter/setter of the class of the instance.

This will allow us to implement the Slot Access Protocol later on.

Bootstrapping Initial Classes

```
Top = _new_base_class(:Top, Symbol[], Symbol[], JClass[])

Object = _new_base_class(:Object, Symbol[], Symbol[], [Top])

Class = _new_base_class(:Class, collect(fieldnames(JClass)), collect(fieldnames(JClass)), [Object])
```

The initial classes of the system will have to be defined by hand. Since there's circularity in the definition of these base classes, we have to define them first and, only then, connect them.

One nice thing about this piece of code is the **collect(fieldnames())** which is a nice way to get the names of the fields of a struct. This way, we can easily change the fields of the **JClass** struct and the **Class** class will always be updated accordingly.

Bootstrapping Initial Classes

```
Top.meta = Class
Top.cpl = [Top]

Object.meta = Class
Object.cpl = [Object, Top]

Class.meta = Class
Class.cpl = [Class, Object, Top]
```

The initial classes of the system will have to be defined by hand. Since there's circularity in the definition of these base classes, we have to define them first and, only then, connect them.

One nice thing about this piece of code is the **collect(fieldnames())** which is a nice way to get the names of the fields of a struct. This way, we can easily change the fields of the **JClass** struct and the **Class** class will always be updated accordingly.

Bootstrapping Remaining Classes

```
function _compute_cpl(cls::JClass)::Vector{JClass}
function _compute_slots(cls::JClass)::Vector{Symbol}
function _compute_defaulted(cls::JClass)::Dict{Symbol,Any}
function _compute_meta_slots(cls::JClass)::Dict{Symbol,Any}
function _compute_getter_and_setter(slot::Symbol)::Tuple{Function, Function}
```

We are now ready to define some helper functions to compute the necessary fields for the next classes we are going to define.

Some of these functions will later be used to define the default behavior of the Meta-Object Protocols of the system.

Bootstrapping Remaining Classes

```
BuiltInClass = _new_default_class(:BuiltInClass, Symbol[], [Class])
MultiMethod = _new_default_class(:MultiMethod, collect(fieldnames(JMultiMethod)), [Object])
GenericFunction = _new_default_class(:GenericFunction, collect(fieldnames(JGenericFunction)), [Object])

_Int64 = _new_default_class(:_Int64, Symbol[], [Top], BuiltInClass)
_String = _new_default_class(:_String, Symbol[], [Top], BuiltInClass)
```

Using the **_new_default_class()** function we can define the remaining classes. Note that this function is used to define classes without using any protocols or generic functions under the hood.

Now is also a good time to define some helper functions that promote functional programming: **class_of()**, **class_name()**, **class_slots()**, **class_direct_slots()** and **class_direct_superclasses()**

```
struct JMultiMethod
    procedure::Function
    specializers::Vector{JClass}
    generic_function::Union{...}
end

struct JGenericFunction
    name::Symbol
    params::Vector{Symbol}
    methods::Vector{JMultiMethod}
End

_add_method(gf, specializers, f)
```

Another crucial part of the system are the generic functions. These are represented in specific structs as well. Generic functions store a list of their methods. Each method stores its specializers.

Additionally, we can define the function **_add_method()** that adds a method to a generic function. This function will check whether the method already exists and, if it does, it will replace it. Otherwise, it will just add it to the list of methods of the generic function.

```
@defgeneric add(x, y)
# Transforms into
add = JGenericFunction(:add, [:x, :y], [])

@defmethod add(x::_Int64, y::_Int64) = x + y
# Transforms into
_add_method(add, [_Int64, _Int64],
      (call_next_method, x, y) -> x + y)
```

We are now able to define the **@defgeneric** macro to define generic functions and the **@defmethod** macro to define methods. We will use them from now on.

One important thing to note is that the **@defmethod** macro will add an extra argument to the method call: the **call_next_method** function. This function will be used to call the next method in the sorted list of methods of the generic function.

Multiple Dispatch

```
@defmethod no_applicable_method(gf::GenericFunction, args) =
    error("No applicable method for function $(gf.name) with arguments ($(join(args, ", ")))")
```

In order for our generic functions to work, we need to be able to dispatch methods. Firstly we can define the generic function **no_applicable_method** which will be called when no method is applicable to the provided specializers.

Multiple Dispatch

```
function (gf::JGenericFunction)(args...)
  applicable_methods = . . .
  sort(applicable_methods)

  method_idx = 1
  function call_next_method()
    if method_idx == length(applicable_methods)
      no_applicable_method(gf, collect(args))
    else
      applicable_methods[method_idx+=1].procedure(call_next_method, args ...)
    End
  End
  applicable_methods[1].procedure(call_next_method, args ...)
End
```

Then we need to specialize the function call on the **JGenericFunction** type. Function calling an instance of this type will filter the applicable methods and sort them by specificity (left to right). Then, it will call the first method in the list. An inner **call_next_method()** function is passed to the method call, so that it can keep track of the next method to call.

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

@defmethod compute_slots(cls::Class) =
_compute_slots(cls)

@defmethod compute_getter_and_setter
(cls::Class, slot, idx) =
_compute_getter_and_setter(slot)
```

To implement Meta-Object Protocols we need to expose some of the already implemented functionality through generic functions in order to provide some default behavior.

```
@defmethod allocate_instance(cls::Class) = begin
    if cls === Top
        error("Cannot instantiate Top class")
    elseif cls === GenericFunction
        JGenericFunction("Null", Symbol[],
MMultiMethod[])
    elseif cls === MultiMethod
        JMultiMethod((call_next_method) -> nothing,
MClass[], nothing)
    elseif cls === Class
        JClass(:Class,
            JClass[],
            Symbol[],
            Dict{Symbol,Any}(),
            Class,
            Symbol[],
            Dict{Symbol,Any}(),
            Dict{Symbol,Function},
            Dict{Symbol,Function},
            [Object])
    else
        JInstance(cls, Dict())
    end
end
```

For the Class Instantiation Protocol, we already start to feel the limitations of our implementation. Since the system doesn't use a uniform representation for classes and instances, we need to define a special case for the **Class** class which restricts the user to only be able to instantiate classes whose meta-class is **Class**.

This is not a big issue since the **@defclass** macro bypasses this protocol to allow the user to define classes whose class is different than **Class**.

```
@defmethod initialize(cls::Class, initargs) =
begin
    cls.cpl = compute_cpl(cls)
    cls.slots = compute_slots(cls)
    cls.defaulted = _compute_defaulted(cls)
    cls.meta_slots = _compute_meta_slots(cls)

    for (slot, idx) in enumerate(cls.slots)
        cls.getters[slot], cls.setters[slot] =
        compute_getter_and_setter(cls, slot, idx)
    end

    . . .
end
```

When defining the **initialize()** method for the **Class** class, we need to call the **compute_cpl()** and **compute_slots()** generic functions we defined earlier, in order to actually implement the protocol.

The **_compute_defaulted()** and **_compute_meta_slots()** functions can also be exposed through generic functions in order add more Meta-Object Protocols. We will discuss this later.

Final Details

```
@defmethod print_object(cls::Class, io) = . . .
@defmethod print_object(obj::Object, io) = . . .
@defmethod print_object(mm::MultiMethod, io) = . . .
@defmethod print_object(gf::GenericFunction, io) = . . .

function Base.show(io::IO, cls::JClass) = . . .
function Base.show(io::IO, obj::JInstance) = . . .
function Base.show(io::IO, mm::JMultiMethod) = . . .
function Base.show(io::IO, gf::JGenericFunction) = . . .
```

For printing objects of the system, we define a generic function **print_object()** and specialize it for the different classes. Besides that, we also specialize Julia's **show** function so that it calls the **print_object** generic function.

Final Details

```
function _new_class(name, direct_slots, direct_superclasses, meta)::JClass
```

```
@defmethod defclass
```

Finally, we can define the **@defclass** macro which makes the definition of classes much easier. This macro will call the **_new_class()** function, which is an improved version of the **_new_default_class()** and calls all the necessary generic functions in order to implement the required protocols.

At this point, we have a fully functional object system with a multitude of features implemented, such as: **Multiple Dispatch**, **Multiple Inheritance**, **Meta-Object Protocols** and **Multiple Meta-Class Inheritance**.

We also have some nice macros that provide a nice **syntactic sugar** for defining classes, generic functions and methods.

Extensions

We further extended the system to support a CLOS-like computation of the class precedence list. Besides that, it’s also very easy to add support for some other Meta-Object Protocols, such as the Compute Class Defaulted Protocol and the Compute Class Meta Slots protocol.

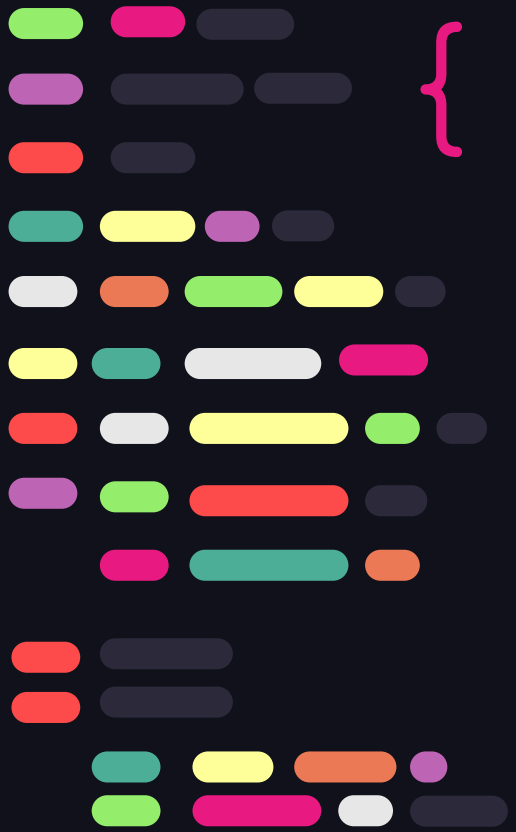
Some proposed extensions are implemented in the **src/extensions.jl** file.

Final Remarks

Not having a uniform representation for classes and instances is a problem that should be fixed in the current implementation. For instance, some of the implemented protocols such as the Slot Access Protocol do not work for classes.

On the other hand, it can also be argued that this implementation is more efficient since it reduces the usage of dictionaries.

Is the trade-off worth is? **Depends on the use case.**



Thank you!