

A simple reflective object kernel

Stéphane Ducasse

September 2, 2018

Copyright 2017 by Stéphane Ducasse.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:
<http://creativecommons.org/licenses/by-sa/3.0/>

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

Contents

Illustrations	iii
1 A class-based reflective minimal kernel	3
1.1 ObjVlisp	3
1.2 ObjVlisp's six postulates	4
1.3 Kernel overview	5
1.4 Instances	7
1.5 Understanding metaclasses	8
1.6 Instance structure	8
1.7 About behavior	9
1.8 Class as an object	10
1.9 Sending a message	12
1.10 Inheritance	15
1.11 Object: defining the minimal behavior of any object	15
1.12 Inheritance and instantiation together	16
1.13 Refresh on self and super semantics	17
1.14 Object creation	19
1.15 Creation of instances of the class Point	19
1.16 Creation of the class Point instance of Class	19
1.17 The Class class	23
1.18 Defining a new Metaclass	24
1.19 About the 6th postulate	25
1.20 Conclusion	27
2 Building a minimal reflective class-based kernel	29
2.1 Objectives	29
2.2 Preparation	30
2.3 Naming conventions	31
2.4 Inheriting from class Array	31
2.5 Facilitating objclass class access	32
2.6 Structure and primitives	33
2.7 Structure of a class	33
2.8 Finding the class of an object	35
2.9 Accessing object instance variable values	36
2.10 Object allocation and initialization	37
2.11 Keywords primitives	38

2.12	Object initialization	39
2.13	Static inheritance of instance variables	39
2.14	Method management	40
2.15	Message passing and dynamic lookup	42
2.16	Method lookup	43
2.17	Managing super	43
2.18	Representing super	44
2.19	Handling not understood messages	45
2.20	Bootstrapping the system	46
2.21	Manually creating ObjClass	47
2.22	Creation of ObjObject	49
2.23	Creation of ObjClass	51
2.24	First User Classes: ObjPoint	53
2.25	First User Classes: ObjColoredPoint	54
2.26	A First User Metaclass: ObjAbstract	55
2.27	New features that you could implement	55
3	Selected definitions	57

Illustrations

1-1	The ObjVlisp kernel: a minimal class-based kernel.	5
1-2	The kernel with specialized metaclasses.	6
1-3	Understanding metaclasses using message passing.	6
1-4	Chain of instantiation: classes are objects too.	7
1-5	Everything is an object. Classes are just objects which can create other objects and metaclasses are just classes whose instances are classes.	9
1-6	Instances of <code>Workstation</code> have two values: their names and their next node.	9
1-7	<code>Point</code> class as an object.	10
1-8	<code>Class</code> as an object.	11
1-9	Through the prism of objects.	11
1-10	Sending a message is two step process: method lookup and execution.	12
1-11	Looking for a method is two step process: first go to the class of receiver then follow inheritance.	13
1-12	When a message is not found, another message is sent to the receiver supporting reflective operation.	14
1-13	Full inheritance graph: Every class ultimately inherits from <code>Object</code>	16
1-14	Kernel with instantiation and inheritance link.	17
1-15	<code>self</code> always represents the receiver.	18
1-16	<code>super</code> represents the receiver but the method lookup starts in the superclass of the class of the method using <code>super</code>	18
1-17	Metaclass role during instance creation: Applying plain message resolution.	20
1-18	Metaclass role during class creation: Applying plain message resolution - the self instantiation link is followed.	21
1-19	Abstract metaclass: its instance (i.e., the class <code>Node</code>) are abstract.	24
1-20	Abstract metaclass at work.	25
2-1	Class structure representation.	34
2-2	Using offset to access information.	34
2-3	Instance variable offset asked to the class.	36
2-4	Instance variable offset asked to the instance itself.	37

Illustrations

This book explains the design and the consequence of having a self-described reflective minimal kernel such as ObjVlisp.

Doing so we will learn about objects, object creation instantiation, message lookup, delegation, inheritance and much more.

A class-based reflective minimal kernel

"The difference between classes and objects has been repeatedly emphasized. In the view presented here, these concepts belong to different worlds: the program text only contains classes; at run-time, only objects exist. This is not the only approach. One of the subcultures of object-oriented programming, influenced by Lisp and exemplified by Smalltalk, views classes as object themselves, which still have an existence at run-time." B. Meyer Object-Oriented Software Construction

As this quote expresses it, there is a realm where classes are true objects, instances of other classes. In such systems such as Smalltalk, Pharo, CLOS, classes are described by other classes and form often reflective architectures each one describing the previous level. In this chapter we will explore a minimal reflective class-based kernel, inspired from ObjVlisp . In the following chapter you will implement step by step such a kernel in less than 30 methods.

1.1 ObjVlisp

ObjVlisp was published the first time in 1986 when the foundation of object-oriented programming was still emerging . ObjVlisp has explicit metaclasses and supports metaclass reuse. It was inspired from the kernel of Smalltalk-78. The IBM SOM-DSOM kernel is similar to ObjVlisp while implemented in C++ . ObjVlisp is a subset of the reflective kernel of CLOS since CLOS reifies instance variables, generic functions, and method combination . In comparison to ObjVlisp, Smalltalk or Pharo have implicit metaclasses and no metaclass reuse except by basic inheritance but it is more stable as explained by Bouraqadi et al .

Studying this kernel is really worth since it has the following properties:

- It unifies class and instances (there is only one data structure to represent all objects, classes included),
- It is composed of only two classes `Class` and `Object` (It relies on existing elements such as booleans, arrays, string of the underlying implementation language),
- It raises the question of meta-circularity infinite regression (a class is instance of another class that is an instance of yet another class ...) and how to resolve it,
- It forces to revisit allocation, class and object initialization, message passing as well as bootstrap,
- It can be implemented in less than 30 methods in Pharo.

Just remember that this kernel is self-described so we will start to explain some aspects and since everything is linked you may have to read the chapter twice to fully get it.

1.2 ObjVLisp's six postulates

The original ObjVLisp kernel is defined by six postulates . Some of them look a bit dated by today standards and the 6th postulate is simply wrong as we will explain later (A solution is simple to design and implement).

Here we report them as stated in the paper for sake of historical perspective.

1. An object represents a piece of knowledge and a set of capabilities.
2. The only protocol to activate an object is message passing: a message specifies which procedure to apply (denoted by its name, the selector) and its arguments.
3. Every object belongs to a class that specifies its data (attributes called fields) and its behavior (procedures called methods). Objects will be dynamically generated from this model, they are called instances of the class. Following Plato, all instances of a class have same structure and shape, but differ through the values of their common instance variables.
4. A class is also an object, instantiated by another class, called its meta-class. Consequently (P3), to each class is associated a metaclass which describes its behavior as an object. The initial primitive metaclass is the class `Class`, built as its own instance.
5. A class can be defined as a subclass of one (or many) other class(es). This subclassing mechanism allows sharing of instance variables and methods, and is called inheritance. The class `Object` represents the most common behavior shared by all objects.

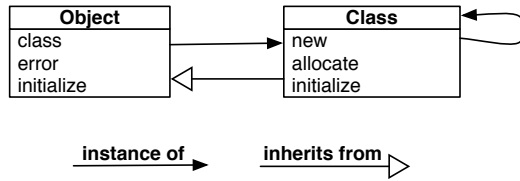


Figure 1-1 The ObjVlisp kernel: a minimal class-based kernel.

6. If the instance variables owned by an object define a local environment, there are also class variables defining a global environment shared by all the instances of a same class. These class variables are defined at the metaclass level according to the following equation: class variable [an-object] = instance variable [an-object’s class].

1.3 Kernel overview

If you do not fully grasp the following overview, do not get worry, this full chapter is here to make sure that you will understand it. Let us get started.

Contrary to a real uniform language kernel, ObjVlisp does not consider arrays, booleans, strings, numbers or any other elementary objects as part of the kernel as this is the case in a real bootstrap such as the one of Pharo. ObjVlisp’s kernel focuses on understanding Class/Object core relationships.

Figure 1-1 shows the two core classes of the kernel:

- Object which is the root of the inheritance graph and is instance of Class.
- Class is the first class and root of the instantiation tree and instance of itself as we will see later.

Figure 1-2 shows that the class Workstation is an instance of the class Class since it is a class and it inherits from Object the default behavior objects should exhibit. The class WithSingleton is an instance of the class Class but in addition it inherits from Class since this is a metaclass: its instances are classes. As such, it changes the behavior of classes. The class Special-Workstation is an instance of the class WithSingleton and inherits from Workstation since its instances exhibits the same behavior as Workstation.

The two diagrams 1-1 and 1-2 will be explained step by step during all this chapter.

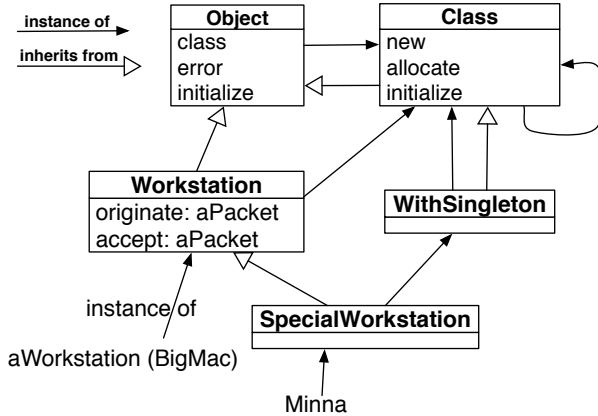


Figure 1-2 The kernel with specialized metaclasses.

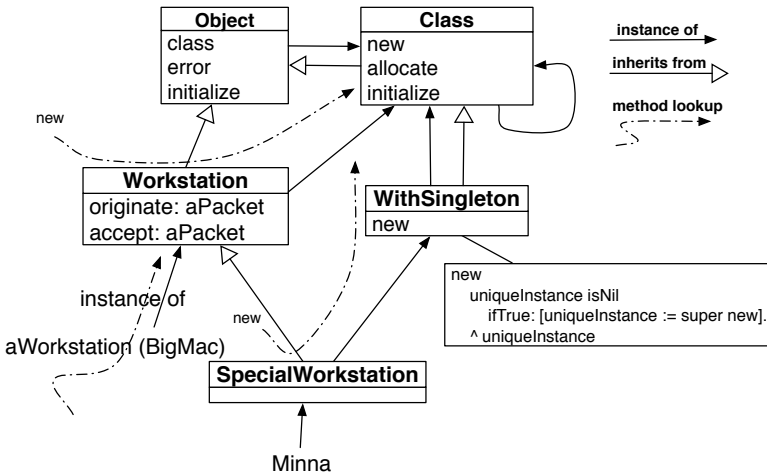


Figure 1-3 Understanding metaclasses using message passing.

Note The key point to understand such a reflective architecture is that message passing always looks up methods in the class of the receiver of the message and then follows the inheritance chain (See Figure 1-3).

Figure 1-3 illustrates two main cases:

- When we send a message to BigMac or Minna, the corresponding method is looked up in their corresponding classes Workstation or Special-Workstation and follows the inheritance link up to Object.
- When we send a message to the classes Workstation or Special-

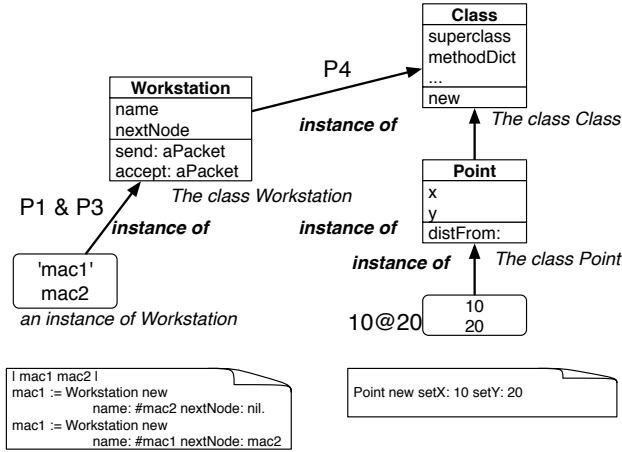


Figure 1-4 Chain of instantiation: classes are objects too.

Workstation, the corresponding method is looked up in their class, the class Class up to Object.

1.4 Instances

In this kernel, there is only one instantiation link, it is applied at all the levels as shown by Figure 1-4:

- Terminal instances are obviously objects: a workstation named mac1 is an instance of the class Workstation, a point 10@20 is instance of the class Point.
- Classes are also objects instances of other classes: the class Workstation is an instance of the class Class, the class Point is an instance of the class Class.

In our diagrams, we represent objects (mainly terminal instances) as round corner rectangles with the list of instance variable values. Since classes are objects, when we want to stress that classes are objects we use the same graphical convention as shown in Figure 1-7.

Handling infinite recursion

A class is an object. Thus it is an instance of another class, its metaclass. This metaclass is an object too, instance of a metametaclass that is an object too instance of another a metametametaclass..... To stop this potential infinite recursion, ObjVlisp uses similar to solutions proposed in many meta circular systems: one instance is instance of itself.

A class is an object. Thus, it is an instance of another class, its metaclass. This metaclass is an object as well, instance of a metametaclass which is itself an instance of another a metametametaclass...

In ObjVLisp:

- `Class` is the initial class and metaclass,
- `Class` is instance of itself and directly or indirectly all other metaclasses are instances of `Class`.

We will see later the implication of this self instantiation at the level of the class structure itself.

1.5 Understanding metaclasses

The model unifies classes and instances. When we follow the instance related postulates of the kernel we get:

- Every object is instance of a class,
- A class is an object instance of a metaclass, and
- A metaclass is only a class that generates classes.

At the implementation level there is only one kind of entity: objects. There is no special treatment for classes. Classes are instantiated following the same process than terminal instances. There are sent messages the same way other objects are sent messages too.

This unification between instances and classes does not mean that we do not distinguish objects and classes. Indeed not all the objects are classes. In particular, the sole difference between a class and an instance is the ability to respond to the creation message: `new`. Only a class knows how to respond to it. Then metaclasses are just classes whose instances are classes as shown in Figure 1-5.

1.6 Instance structure

The model does not really bring anything new about instance structure when compared with languages such as Pharo or Java.

Instance variables are an ordered sequence of instance variables defined by a class. Such instance variables are shared by all instances. The values of such instance variables are specific to each instance. Figure 1-6 shows that instances of `Workstation` have two values: a name and a next node.

In addition we should note that an object has a pointer to its class. As we will see when we will come to inheritance, every object possesses an instance variable `class` (inherited from `Object`) that points to its class.

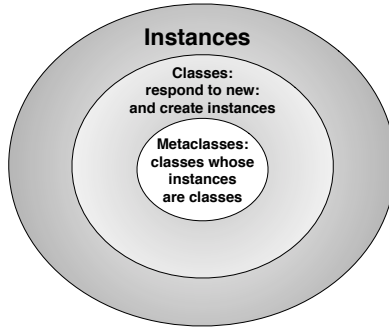


Figure 1-5 Everything is an object. Classes are just objects which can create other objects and metaclasses are just classes whose instances are classes.

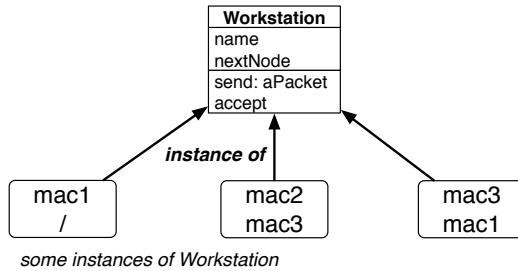


Figure 1-6 Instances of Workstation have two values: their names and their next node.

Note that this management of class instance variable defined in Object is specific to the model. In Pharo for example, the class identification is not managed as a declared instance variable but as an element part of any object. It is an index in a class-table.

1.7 About behavior

Let us continue with basic instance behavior. As in modern class-based languages, this kernel has to represent how methods are stored and looked up.

Methods belong to a class. They define the behavior of all the instances of the class. They are stored into a method dictionary that associates a key (the method selector) and the method body.

Since the methods are stored in a class, the method dictionary should be described in the metaclass. Therefore, the method dictionary of a class is the *value* of the instance variable `methodDict` defined on the metaclass `Class`. Each class will have its own method dictionary.

The class *Point*

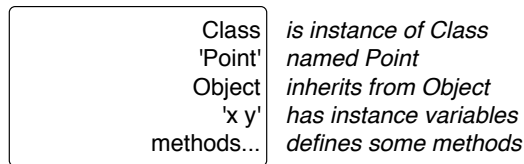


Figure 1-7 Point class as an object.

1.8 Class as an object

Now it is time to ask us about the minimal information that a class should have. Here is the minimal information required:

- A list of instance variables to describe the values that the instances will hold,
- A method dictionary to hold methods,
- A superclass to look up inherited methods.

This minimal state is similar to the one of Pharo: Pharo Behavior class has a format (compact description of instance variables), method dictionary, and superclass link.

In ObjVLisp, we will had a name so that we can identify the class. As an instance factory, the metaclass `Class` possesses 4 instance variables that describe a class:

- name the class name,
- superclass its superclass (we limit to single inheritance),
- i-v the list of its instance variables, and
- methodDict a method dictionary.

Since a class is an object. A class possesses the instance variable `class` inherited from `Object` that refers to its class as any object.

Example: class `Point`

Figure 1-7 shows the instance variable values for the class `Point` as declared by the programmer and before class initialization and inheritance take place.

- It is an instance of class `Class`: indeed this is a class.
- It is named `'Point'`.
- It inherits from class `Object`.

1.8 Class as an object

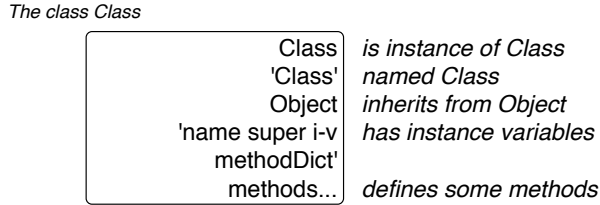


Figure 1-8 Class as an object.

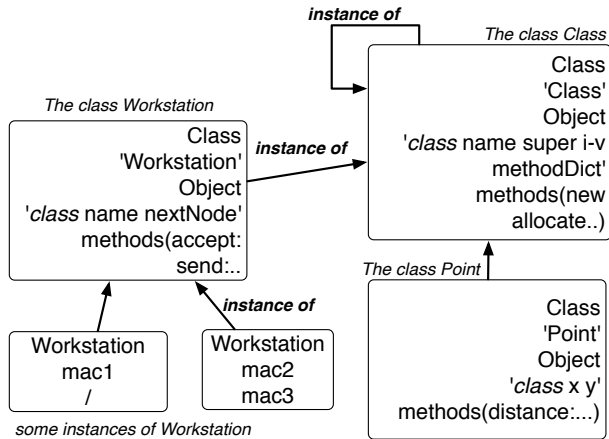


Figure 1-9 Through the prism of objects.

- It has two instance variables: *x* and *y*. After inheritance it will be three instance variables: *class*, *x*, and *y*.
- It has a method dictionary.

Example: class Class

Figure 1-8 describes the class `Class` itself. Indeed it is also an object.

- It is an instance of class `Class`: indeed this is a class.
- It is named `'Class'`.
- It inherits from class `Object`
- It has four locally defined instance variables: *name*, *superclass*, *i-v*, and *methodDict*.
- It has a method dictionary.

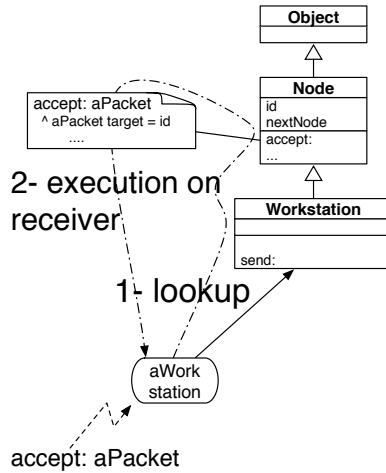


Figure 1-10 Sending a message is two step process: method lookup and execution.

Everything is an object

Figure 1-9 describes a typical situation of terminal instances, class and meta-classes when looked at them using an object perspective. We see three level of instances: terminal objects, instances of Workstation, Workstation and Point classes which are instances of Class and the metaclass Class which is instance of itself.

1.9 Sending a message

In this kernel, the second postulate states that the only way to perform computation is via message passing.

Sending a message is a two step process as shown by Figure 1-10

1. Method lookup: the method corresponding to the selector is looked up in the class of the receiver and its superclasses.
2. Method execution: the method is applied to the receiver. It means that `self` or `this` in the method will be bound to the receiver.

Conceptually, sending a message can be described by the following function composition:

```

[ sending a message (receiver argument)
  return apply (lookup (selector classof(receiver) receiver)
               receiver arguments)

```

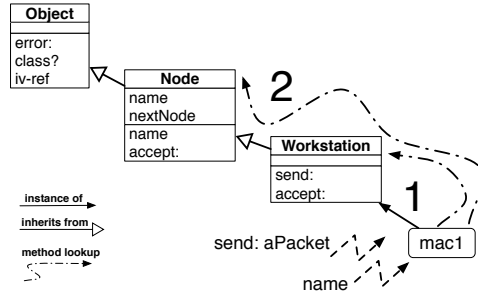


Figure 1-11 Looking for a method is two step process: first go to the class of receiver then follow inheritance.

Method lookup

Now the lookup process is conceptually defined as follows:

1. The lookup starts in the **class** of the **receiver**.
2. If the method is defined in that class (i.e., if the method is defined in the method dictionary), it is returned.
3. Otherwise the search continues in the superclass of the currently explored class.
4. If no method is found and there is no superclass to explore (if we are in the class `Object`), this is an error.

The method lookup walks through the inheritance graph one class at a time using the superclass link. Here is a possible description of the lookup algorithm that will be used for both instance and class methods.

```

lookup (selector class receiver):
  if the method is found in class
    then return it
  else if class == Object
    then send the message error to the receiver
  else lookup (selector superclass(class) receiver)
  
```

Handling errors

When the method is not found, the message `error` is sent as shown in Figure 1-12. Sending a message instead of simply reporting an error using a trace or an exception is a key design decision. It corresponds to the `doesNotUnderstand:` message in Pharo and it is an important reflective hook. Indeed classes can define their own implementation of the method `error` and perform specific actions in reaction of messages that are not understood. For example, it is possible to implement proxies (objects representing other remote objects) or compile code on the fly by redefining locally such message.

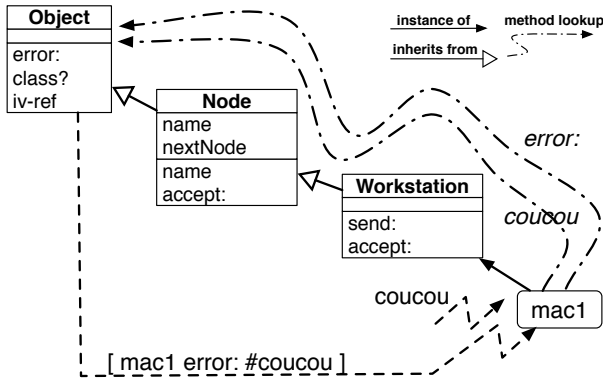


Figure 1-12 When a message is not found, another message is sent to the receiver supporting reflective operation.

Now it should be noted that the previous algorithm is not really good because in case of error there can be a mismatch between the number of arguments of the method we are looking for and the number of arguments of the error message.

A better way to handle error is to decompose the algorithm differently as follows:

```

lookup (selector class):
  if the method is found in class
  then return it
  else if class == Object
  then return nil
  else lookup (selector superclass(class))
  
```

And then we redefined sending a message as follows:

```

sending a message (receiver argument)
  methodOrNil = lookup (selector classof(receiver)).
  if methodOrNil is nil
  then return send the message error to the receiver
  else return apply(methodOrNil receiver arguments)
  
```

Remarks

This lookup is conceptually the same in Pharo where all methods are public and virtual. There is no statically bound method, even class methods are looked up dynamically. This allows the possibility to define really elegant and dynamic registration mechanism.

While the look up happens at runtime, it is often cached. Languages usually have several systems of caches: one global (class, selector), one per call site.

1.10 Inheritance

There are two aspects of inheritance to consider:

- One static for the state where subclasses get superclass state. This instance variable inheritance is static in the sense that it happens only once at class creation time i.e., at compilation-time.
- One dynamic for behavior where methods are looked up during program execution. There the inheritance tree is walked at run-time.

Let's look at these two aspects.

Instance variable inheritance

Instance variable inheritance is done at class creation time and from that perspective static and performed once. When a class C is created, its instance variables are the union of the instance variables of its superclass with the instance variables defined locally in class C. Each language defines the exact semantics of instance variable inheritance, for example if they accept instance variables with the same name or not. In our model, we decide to use the simplest way: there should be no name duplicates.

```
instance-variables(aClass) =
  union (instance-variables(superclass(aClass)),
        local-instance-variables(aClass))
```

A word about union: when the implementation of the language is based on offsets to access instance variables, the union should make sure that the location of inherited instance variables are kept ordered compared to the superclass because in general we want that methods of the superclass can be applied to subclasses without copying them down and recompiling them. Indeed if a method uses a variable at a given position in the instance variable lists, applying this method to instance of subclasses should work. In the implementation proposed next chapter, we will use accessors and will not support direct access to instance variables from method body.

Method lookup

As previously described in Section 1.9, methods are looked up at runtime. Methods defined in superclasses are reused and applied to instances of subclasses. Contrary to instance variable inheritance, this part of inheritance is dynamic, i.e., it happens during program execution.

1.11 Object: defining the minimal behavior of any object

Object represents the minimal behavior that any object should understand. For example, returning the class of an object, being able to handle errors,

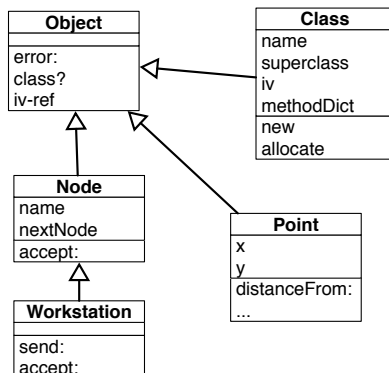


Figure 1-13 Full inheritance graph: Every class ultimately inherits from `Object`.

initializing an object. This is why `Object` is the root of the hierarchy. Depending on language `Object` can be complex. In our kernel it is kept minimal as we will show in the implementation chapter.

Figure 1-13 shows the inheritance graph without the presence of instantiation. A `Workstation` is an object (should at least understand the minimal behavior), so the class `Workstation` inherits directly or indirectly from the class `Object`. A class is an object (it should understand the minimal behavior) so the class `Class` inherits from class `Object`. In particular, the `class` instance variable is inherited from `Object` class.

Remark.

In `Pharo`, the class `Object` is not the root of inheritance. It is `ProtoObject` and `Object` inherits from it. Most of the classes still inherit from `Object`. The design of `ProtoObject` is special: the design goal of `ProtoObject` is to generate as many as errors as possible. Such errors can be then captured via `doesNotUnderstand:` redefinition and can support different scenarios such as proxy implementation.

1.12 Inheritance and instantiation together

Now that we saw independently the instantiation and the inheritance graph we can look at the complete picture. Figure 1-14 shows the graphs and in particular how such graph are used during message resolution:

- the instantiation link is used to find the class where to start to look the method associated with the received message.
- the inheritance link is used to find inherited methods.

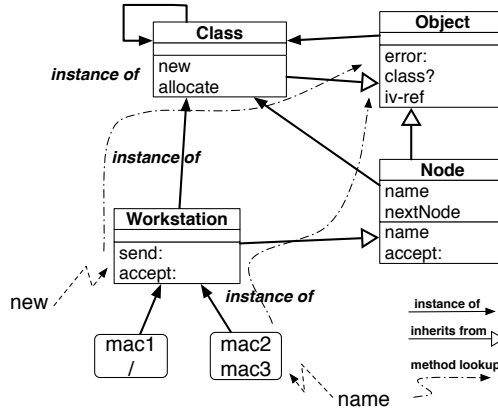


Figure 1-14 Kernel with instantiation and inheritance link.

This process is also true when we send messages to the classes themselves. There is no difference between sending a message to an object or a class. The system *always* performs the same steps.

1.13 Refresh on self and super semantics

Since our experience showed us that even some book writers got key semantics of object-oriented programming wrong, we just refresh some facts that normally programmers familiar with object-oriented programming should fully master. For further readings refer to Pharo By Example or the Pharo Moom available at <http://moom.pharo.org>.

- **self** (also called **this** in languages like Java). self always represents the receiver of the message. The method lookup starts in the class of the receiver.
- **super**. super always represents the receiver of the message (and not the superclass). The method lookup starts in the superclass of the class containing the super expression (and not in the superclass of the class of the receiver: this would mean that it loops forever in case of inheritance tree of three classes - We let you to find how).

Looking at Figure 1-15 we see that the key point is that `B new bar` returns 50 since the method is dynamically looked up and self represents the receiver i.e., the instance of the class B. What is important to see is that `self` sends act as a hook and that subclasses code can be injected in superclass code.

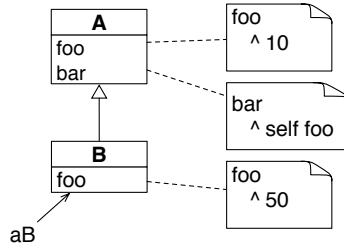


Figure 1-15 self always represents the receiver.

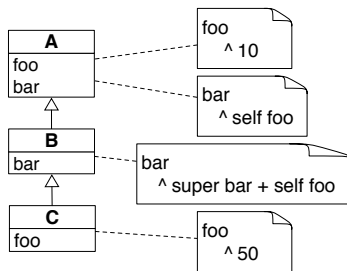


Figure 1-16 super represents the receiver but the method lookup starts in the superclass of the class of the method using super.

```
A new foo
>>> 10
B new foo
>>> 50
A new bar
>>> 10
B new bar
>>> 50
```

For super, the situation depicted in Figure 1-16 shows that super represents the receiver, but that when super is the receiver of a message, the method is looked up differently (starting from the superclass of the class using super) hence C new bar returns 100 and not 20 nor 60.

```
B new bar
>>> 20
C new bar
>>> 100
```

As a conclusion, we can say that self is dynamic and super static. Let us explain this view:

- When sending a message to self the lookup of the method begins in the class of the receiver. self is bound at execution-time. We do not

know its value until execution time.

- `super` is static in the sense that while the object it will point to is only known at execution time, the place to look for the method is known at compile-time: it should start to look in the class above the one containing `super`.

1.14 Object creation

Now we are ready to understand the creation of objects. In this model there is only one way to create instances: we should send the message `new` to the class with a specification of the instance variable values as argument.

1.15 Creation of instances of the class `Point`

The following examples show several point instantiations. What we see is that the model inherits from the Lisp traditional of passing arguments using keys and values, and that the order of arguments is not important.

```
[ Point new :x 24 :y 6
  >>> aPoint (24 6)
  Point new :y 6 :x 24
  >>> aPoint (24 6)
```

When there is no value specified, the value of an instance variable is initialized to `nil`. CLOS provides the notion of default instance variable initialization. It can be added to `ObjVlisp` as an exercise and does not bring conceptual difficulties.

```
[ Point new
  >>> aPoint (nil nil)
```

When the same argument is passed multiple times, then the implementation takes the first occurrence.

```
[ Point new :y 10 :y 15
  >>> aPoint (nil 10)
```

We should not worry too much about such details: The point is that we can pass multiple arguments with a tag to identify it.

1.16 Creation of the class `Point` instance of `Class`

Since the class `Point` is an instance of the class `Class`, to create it, we should send the message `new` to the class as follows:

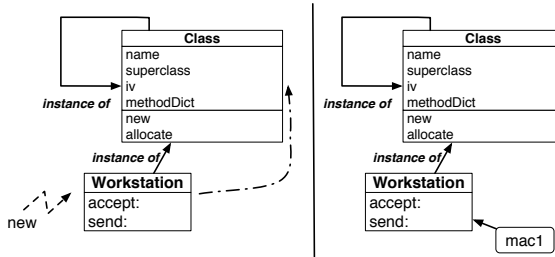


Figure 1-17 Metaclass role during instance creation: Applying plain message resolution.

```

Class new
  :name 'Point'
  :super 'Object'
  :ivs #(x y)
>>> aClass

```

Here what is interesting to see is that we use exactly the same way to create an instance of the class `Point` or the class itself. Note that this single way to create objects is supported by the argument variable list.

An implementation could have two different messages to create instances and classes. As soon as the same `new`, `allocate`, `initialize` methods are involved, the essence of the object creation is similar and uniform.

Instance creation: Role of the metaclass

The following diagram (Figure 1-17) shows that against common expectations, when we create a terminal instance the metaclass `Class` is evolved in the process. Indeed, we send the message `new` to the class, to resolve this message, the system will look for the method in the class of the receiver (here `Workstation`) which the metaclass `Class`. The method `new` is found in the metaclass and applied to the receiver: the class `Workstation`. Its effect is to create an instance of the class `Workstation`.

The same happens when creating a class. Figure 1-18 shows the process. We send a message, now this time, to the class `Class`. The system makes no exception and to resolve the message, it looks for the method in the class of the receiver. The class of the receiver is itself, so the method `new` found in `Class` is applied to `Class` (since it is the receiver of the message), and a new class is created.

`new` = `allocate` and `initialize`

In fact creating an object is a two step process: Creating an instance is the composition of two actions: memory allocation `allocate` message and object

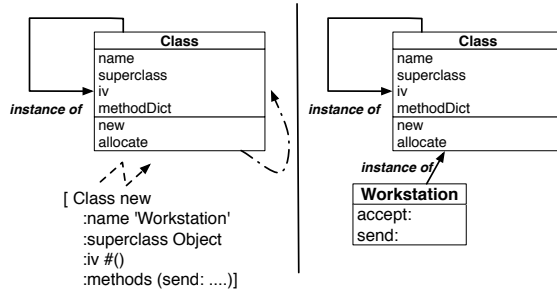


Figure 1-18 Metaclass role during class creation: Applying plain message resolution - the self instantiation link is followed.

initialisation message `initialize`.

In Pharo syntax it means:

```
[ aClass new: args = (aClass allocate) initialize: args
```

What we should see is that:

- The message `new` is a message sent to a class. The method `new` is a class method.
- The message `allocate` is a message sent to a class. The method `allocate` is a class method.
- The message `initialize:` will be executed on any newly created instance. It means that when it will be sent to a class, a class `initialize:` method will be involved. When it will be sent to a terminal object, an instance `initialize:` method will be executed (defined in `Object`).

Object allocation: the message `allocate`

Allocating an object means allocating enough space to the object state but not only: it should mark instances with their class name or id. There is a really strong invariant in the model and in general in object-oriented programming model. Every single object must have an identifier to its class else the system will break when trying to resolve a message.

Object allocation should return:

- A newly created instance with empty instance variables (pointing to `nil` for example).
- But marked with an identifier to its class.

In our model, the marking of an object as instance of a class is performed by setting the value of the instance variable `class` inherited from `Object`. In

Pharo this information is not recorded as a instance variable but encoded in the internal virtual machine object representation.

The `allocate` method is defined on the metaclass `Class`. Here are some examples of allocation.

```
[Point allocate
>>> #(Point nil nil)
```

A point allocation allocates three slots: one for the class and two for `x` and `y` values.

```
[Class allocate
>>>#(Class nil nil nil nil nil)
```

The allocation for an object representing a class allocates six slots: one for class and one for each of the class instance variable: `name`, `super`, `iv`, `key-words`, and `methodDict`.

Object initialization

Object initialization is the process to get the values passed as arguments as key/value pair and assigned the value to the corresponding instance variable.

The following snippet illustrates it. An instance of class `Point` is created and the key/value pairs `(:y 6)` and `(:x 24)` are specified. The instance is created and it received the `initialize:` messages with the key/value pairs. The `initialize:` method is responsible to set the corresponding variables in the receiver.

```
[Point new :y 6 :x 24
>>> #(Point nil nil) initialize: (:y 6 :x 24)]
>>> #(Point 24 6)
```

When an object is initialized as a terminal instance, two actions are performed:

- First we should get the values specified during the creation, i.e., get that `y` value is 6 and `x` value is 24,
- Second we should assign the values to the corresponding instance variables of the created object.

Class initialization

During its initialization a class should perform several steps:

- First as any it should get the arguments and assigned them to their corresponding instance variables. This is basically implemented by invoking the `initialize` method of `Object` via a super call since `Object` is the superclass of `Class`.

- Second the inheritance of instance variables should be performed. Before this step the class `iv` instance variable just contains the instance variables that are locally defined. After this step the instance variable `iv` will contain all the instance variables inherited and local. In particular this is here that the `class` instance variable inherited from `Object` is added to the instance variables list of the subclass of `Object`.
- Third the class should be declared to a class pool or namespaces so that as programmers we can access it via its name.

1.17 The Class class

Now we get a better understanding of what is the class `Class`. The class `Class` is:

- The initial metaclass and initial class.
- It defines the behavior of all the metaclasses.
- It defines the behavior of all the classes.

In particular, metaclasses define three messages related to instance creation.

- The `new` message creates an initialized instance of the class. It allocates the instance using the class message `allocate` and then initializes it by sending the message `initialize:` to this instance.
- The `allocate` message. As message `new` it is a class message. It allocates structure for newly created object.
- Finally the message `initialize:.` This message has two definitions one on `Object` and one on `Class`.

There is a difference between the method `initialize:` executed on any instance creation and the class `initialize:` method only executed when the created instance is a class.

- The first one is a method defined on the class of the object and potentially inherited from `Object`. This `initialize:` method just extracts the value corresponding to each instance variables from the argument list and sets them in the corresponding instance variables.
- The class `initialize:` method is executed when a new instance representing a class is executed. The message `initialize:` is sent to the newly created object but its specialisation for classes will be found during method lookup and it will be executed. Usually this method invokes the default ones because the class parameter should be extracted from the argument list and set in their corresponding instance variables but in addition, instance variable inheritance and class declaration in the class namespace is performed.

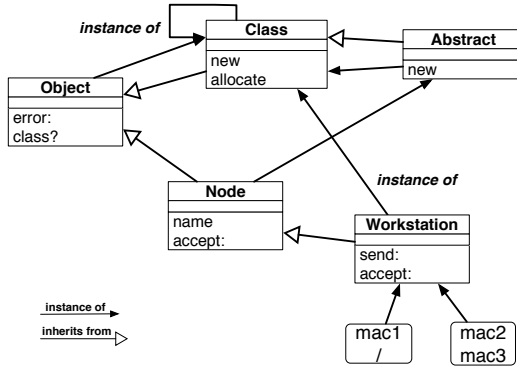


Figure 1-19 Abstract metaclass: its instance (i.e., the class Node) are abstract.

1.18 Defining a new Metaclass

Now we can study how we can add new metaclasses and see how the system handles them. To create a new metaclass is simple, it is enough to inherit from an existing one. Maybe this is obvious to you but this is what we will check now.

Abstract

Imagine that we want to define abstract classes. We set the abstractness of a class as the fact that it cannot create instances. To control the creation of instances of a class, we should define a new metaclass which forbids it. Therefore we will define a metaclass whose instances (abstract classes) cannot create instances.

We create a new metaclass named `AbstractMetaClass` which inherits from `Class` and we redefine the method `new` in this metaclass to raise an error (as shown in Figure 1-19). The following code snippet defines this new metaclass.

```

[
Class new
  :name 'AbstractMetaClass'
  :super 'Class'
]
[
AbstractMetaClass
  addMethod: #new
  body: [ :receiver :initargs | receiver error: 'Cannot create
instance of class' ]
]
  
```

Two facts describe the relations between this metaclass and the class `Class`:

- `AbstractMetaClass` is a class: It is instance of `Class`.
- `AbstractMetaClass` defines class behavior: It inherits from `Class`.

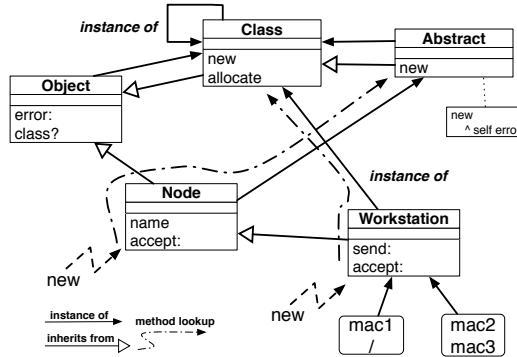


Figure 1-20 Abstract metaclass at work.

Now we can define an abstract class Node.

```
[ AbstractMetaclass new :name 'Node' :super 'Object'
```

Sending a message new to the class Node will raise an error.

```
[ Node new
>>> Cannot create instance of class
```

A subclass of Node, for example Workstation, can be a concrete class by being an instance of Class instead of AbstractMetaclass but still inheriting from Node. What we see in Figure 1-20 is that there are two links: instantiation and inheritance and the method lookup follows them as we presented previously: always start in the class of the receiver and follow the inheritance link.

What is key to understand is that when we send the message new to the class Workstation, we look for methods first in the metaclass Class. When we send the message new to class Node we look in its class: AbstractMetaclass as shown in Figure 1-20. In fact we do what we do for any instances: we look in the class of the receiver.

A class method is just implemented and following the same semantics that instance methods: Sending the message error to the class Node starts in AbstractMetaclass and since we did not redefine it locally, and it is not found there, the lookup will continue in the superclass of AbstractClass: the class Class and then the superclass of class Class, the class Object.

1.19 About the 6th postulate

The 6th postulate of ObjVlisp is wrong. Let us read it again: *If the instance variables owned by an object define a local environment, there are also class variables defining a global environment shared by all the instances of a same class. These*

class variables are defined at the metaclass level according to the following equation: class variable [an-object] = instance variable [an-object's class].

It says that class instance variables are equivalent to shared variables between instances and this is wrong. Let us study this. According to the 6th postulate, a shared variable between instances is equal to an instance variable of the class. The definition is not totally clear so let us look at an example given in the article.

Illustrating the problem

Imagine that we would like the constant character '*' to be a class variable shared by all the points of a same class. We redefine the Point class as before, but metaclass of which (let us call it MetaPoint) specifies this common character. For example if a point has a shared variable named char, this instance variable should be defined in the class of the class Point called MetaPoint. The author proposes to define a new metaclass MetaPoint to hold a new instance variable to represent a shared variable between points.

```
Class new
  :name 'MetaPoint'
  :super 'Class'
  :ivs #(char)
```

Then he proposes to use it as follows:

```
MetaPoint new
  :name Point
  :super 'Object'
  :ivs #(x y)
  :char '*'
```

The class Point can define a method that accesses the character just by going the class level. So why this approach is wrong? Because it mixes levels. The instance variable char is not a class information. It describes the terminal instances and not the instance of the metaclass. Why the metaclass MetaPoint would need a char instance variable.

The solution

The solution is that the shared variable char should be held in a list of the shared variables of the class Point. Any point instance can access this variable. The implication is that a class should have an extra information to describe it: an instance variable sharedVariable holding pairs i.e., variable and its value. We should be able to write:


```
Class new
  :name Point
  :super 'Object'
  :ivs #(x y)
  :sharedivs {#char -> '*'}
```

Therefore the metaclass `Class` should get an extra instance variable named `sharedivs` and each of its instances (the classes `Point`, `Node`, `Object`) can have different *values* and such values can be shared among their instances by the compiler.

What we see is that `sharedivs` is from the `Class` vocabulary and we do not need one extra metaclass each time we want to share a variable. This design is similar to the one of `Pharo` where a class has a `classVariable` instance variable holding variable shared in all the subclasses of the class defining it.

1.20 Conclusion

We presented a really small kernel composed of two classes `Object` root of the inheritance tree and `Class` the first metaclass root of the instantiation tree. We revisited all the key points related to method lookup, object and class creation and initialisation. In the subsequent chapter we propose you to implement such kernel.

Further readings

The kernel presented in this chapter is a kernel with explicit metaclasses and as such it is not a panacea. Indeed it raised metaclass composition problems as explained in Bouraqadi et al. excellent article or .

Building a minimal reflective class-based kernel

The objectives of this chapter is to help you to implement step by step the ObjVlisp model explained in the previous chapter. ObjVlisp was designed by P. Cointe who got inspired by the kernel of Smalltalk-78. It has explicit metaclasses and it is composed of two classes `Object` and `Class`.

2.1 Objectives

During the previous chapter, you saw the main points of the ObjVlisp model, now you will implement it. The goals of this implementation are to give a concrete understanding of the concepts presented previously. Here are some of the points you can deeply understand while writing the implementation:

- What is a possible object structure?
- What is object allocation and initialization?
- What is class initialization?
- What the semantics of the method lookup?
- What is a reflective kernel?
- What are the roles of the classes `Class` and `Object`?
- What is the role of a metaclass?

2.2 Preparation

In this section we discuss the set up that you will use, the implementation choices and the conventions that we will follow during all this chapter.

Getting Pharo

You need to download and install Pharo from <http://www.pharo.org/>. You need a virtual machine, and the couple image and changes. You can use <http://get.pharo.org> to get a script to download Pharo.

The current version that you can use is Pharo 6.1.

```
[ wget -O- get.pharo.org/61+vm | bash
```

You can use the book Pharo by Example from <http://www.pharo.org/PharoByExample/> for an overview of the syntax and the system.

Getting infrastructure definitions

All the necessary definitions are provided as a Monticello package. It contains all the classes, the method categories and the method signatures of the methods that you have to implement. It provides additional functionality such as a dedicated inspector and some extra methods that will make your life easy and help you to concentrate on the essence of the model. It contains also all the tests of the functionality you have to implement.

To load the code, execute the following expression:

```
[ Gofer new
  url:
    'http://smalltalkhub.com/mc/StephaneDucasse/ObjVLispSkeleton/main';
  package: 'ObjVLispSkeleton';
  load.
```

Alternatively, to load the code open a monticello browser,

- Add a file repository to point to an ObjVLispSkeleton project under StephaneDucasse in the ObjVLispSkeleton project at <http://www.smalltalkhub.com>. Use the following expression in the smalltalkhub repository creation pop up.

```
[ MCSmalltalkhubRepository
  owner: 'StephaneDucasse'
  project: 'ObjVLispSkeleton'
  user: ''
  password: ''
```

- Select the latest file and load it.

Running tests

For each functionality you will have to run some tests.

For example to run a particular test named `testPrimitive`,

- evaluate the following expression (`ObjTest selector: #testPrimitiveStructure`) run or
- click on the icon of the method named `testPrimitiveStructure`.

Note that since you are developing the kernel, to test it we implemented manually some mocks of the classes and kernel. This is the setup method of the test classes that build this fake kernel. Now pay attention because the setups are often taking shortcuts, so do not copy them blindly.

2.3 Naming conventions

We use the following conventions: we name as *primitives* all the Pharo methods that participate in the building of ObjVLisp. These primitives are mainly implemented as methods of the class `Obj`. Note that in a Lisp implementation such primitives would be just lambda expressions, in a C implementation such primitives would be represented by C functions.

To help you to distinguish between classes in the implementation language (Pharo) and the ObjVLisp model, we prefix all the ObjVLisp classes by `Obj`. Finally, some of the crucial and confusing primitives (mainly the class structure ones) are all prefixed by `obj`. For example the primitive that given an *objInstance* returns its class identifier is named `objClassId`. We also talk about `objInstances`, `objObjects` and `objClasses` to refer to specific instances, objects or classes defined in ObjVLisp.

2.4 Inheriting from class Array

We do not want to implement a scanner, a parser and a compiler for ObjVLisp but concentrate on the essence of the language. That's why we chose to use as much as possible the implementation language, here Pharo. As Pharo does not support macro definition, we will use as much as possible the existing classes to avoid extra syntactic problems.

In our implementation, every object in the ObjVLisp world is instance of the class `Obj`. The class `Obj` is a subclass of `Array`.

Since `Obj` is a subclass of `Array`, `#(ObjPoint 10 15)` is an `objInstance` of the class `ObjPoint` which is also an array instance of the Pharo class `ObjClass`.

As we will see:

- `#(#ObjPoint 10 15)` represents an `objPoint (10,15)`. It is an `objInstance` of the class `ObjPoint`.
- `#(#ObjClass #ObjPoint #ObjObject #(class x y) #(:x :y) nil)` is the array that represents the `objclass ObjPoint`.

About representation choices

You can skip this discussion in a first reading. We could have implemented `ObjVLisp` functionality at the class level of a class named `Obj` inheriting simply from `Object`. However, to use the `ObjVLisp` primitive (a Pharo method) `objInstanceVariableValue: anObject for: anInstanceVariable` that returns the value of the instance variable in an `Object`, we would have been forced to write the following expression:

```
[Obj objInstanceVariableValue: 'x' for: aPoint
```

We chose to represent any `ObjVLisp` object by an array and to define the `ObjVLisp` functionality in the instance side of the class `Obj` (a subclass of `Array`). That way we can write in a more natural and readable way the previous functionality as:

```
[aPoint objInstanceVariableValue: 'x'.
```

2.5 Facilitating objclass class access

We need a way to store and access `ObjVLisp` classes. As a solution, on the class level of the Pharo class `Obj` we defined a dictionary holding the defined classes. This dictionary acts as the namespace for our language. We defined the following methods to store and access defined classes.

- `declareClass: anObjClass` stores the `objinstance anObjClass` given as argument in the class repository (here a dictionary whose keys are the class names and values the `ObjVLisp` classes themselves).
- `giveClassNamed: aSymbol` returns the `ObjVLisp` class named `aSymbol` if it exists. The class should have been declared previously.

With such methods we can write code like the following one that looks for the class of the class `ObjPoint`.

```
[Obj giveClassNamed: #ObjPoint
[>>> #( #ObjClass 'ObjPoint' #ObjObject #(class x y) #(:x :y) ... )
```

To make class access less heavy, we also implemented a shortcut: We trap messages not understood sent to `Obj` and look into the defined class dictionary. Since `ObjPoint` is an unknown message, this same code is then written as:

```
Obj ObjPoint
[>>> #(#ObjClass 'ObjPoint' #ObjObject #(class x y) #(:x :y) ... )
```

Now you are ready to start.

2.6 Structure and primitives

The first issue is how to represent objects. We have to agree on an initial representation. In this implementation we chose to represent the objinstances as arrays (instances of Obj a subclass of Array). In the following we use the terms array for talking about instances of the class Obj.

Your job.

Check that the class Obj exists and inherits from Array.

2.7 Structure of a class

The first object that we will create is the class ObjClass. Therefore we focus now on the minimal structure of the classes in our language.

An objinstance representing a class has the following structure: an identifier to its class, a name, an identifier to its superclass (we limit the model to single inheritance), a list of instance variables, a list of initialization keywords, and a method dictionary.

For example the class ObjPoint has then the following structure:

```
[#(#ObjClass #ObjPoint #ObjObject #(class x y) #(:x :y) nil)
```

It means that ObjPoint is an instance of ObjClass, is named #ObjPoint, inherits from a class named ObjObject, has three instance variables, two initialization keywords and an uninitialized method dictionary. To access this structure we define some primitives as shown in Figure 2-1.

Your job.

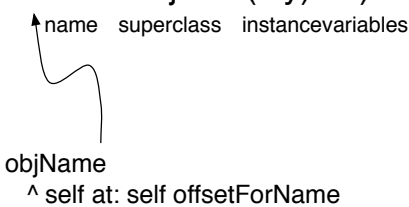
The test methods of the class RawObjTest that are in the categories 'step1-tests-structure of objects' and 'step2-tests-structure of classes' give some examples of structure accesses.

```
RawObjTest >> testPrimitiveStructureObjClassId
  "(self selector: #testPrimitiveStructureObjClassId) run"
  self assert: (pointClass objClassId = #ObjClass).
```

```
#(
  #ObjClass      offsetForClass (1)
  #ObjPoint      offsetForName (2)
  #ObjObject     offsetForSuperclass (3)
  #(class x y)   offsetForIVs (4)
  #(:x :y)       offsetForKeywords (5)
  nil            offsetForMethodDict (6)
)
```

Figure 2-1 Class structure representation.

```
#(Class 'Point' 'Object' '(x y) ....)
```



↑ name superclass instancevariables

objName
^ self at: self offsetForName

Figure 2-2 Using offset to access information.

```
RawObjTest >> testPrimitiveStructureObjIVs
"(self selector: #testPrimitiveStructureObjIVs) run"

self assert: ((pointClass objIVs) = #(class #x #y)).
```

Figure 2-2 shows how offsets are used to access in a controlled manner the raw objclass information.

Implement the primitives that are missing to run the following tests `testPrimitiveStructureObjClassId`, `testPrimitiveStructureObjIVs`, `testPrimitiveStructureObjKeywords`, `testPrimitiveStructureObjMethodDict`, `testPrimitiveStructureObjName`, and `testPrimitiveStructureObjSuperclassId`.

You can execute them by selecting the following expression (`RawObjTest selector: #testPrimitiveStructureObjClassId`) `run`. Note that arrays start at 1 in Pharo. Below is the list of the primitives that you should implement.

Implement in protocol 'object structure primitives' the primitives that manage:

- the class of the instance represented as a symbol. `objClassId`, `obj-`

`ClassId`: aSymbol. The receiver is an objObject. This means that this primitive can be applied on any objInstances to get its class identifier.

Implement in protocol 'class structure primitives' the primitives that manage:

- the class name: `objName`, `objName`: aSymbol. The receiver is an objClass.
- the superclass: `objSuperclassId`, `objSuperclassId`: aSymbol. The receiver is an objClass.
- the instance variables: `objIVs`, `objIVs`: anOrderedCollection. The receiver is an objClass.
- the keyword list: `objKeywords`, `objKeywords`: anOrderedCollection. The receiver is an objClass.
- the method dictionary: `objMethodDict`, `objMethodDict`: anIdentityDictionary. The receiver is an objClass.

2.8 Finding the class of an object

Every object keeps the identifier of its class (its name). For example an instance of `ObjPoint` has then the following structure: `(#ObjPoint 10 15)` where `#ObjPoint` is a symbol identifying the class `ObjPoint`.

Your job.

Using the primitive `giveClassNamed`: aSymbol defined at the class level of `Obj`, define the primitive `objClass` in the protocol 'object-structure primitive' that returns the objInstance that represents its class (Classes are objects too in `ObjVLisp`).

Make sure that you execute the test method: `testClassAccess`

```
RawObjTest >> testClassAccess
  "(self selector: #testClassAccess) run"

  self assert: (aPoint objClass = pointClass)
```

Now we will be ready to manipulate objInstances via proper API. We will now use the class `ObjTest` for more elaborated tests.

```

#(
  #ObjClass
  #ObjPoint
  #ObjObject
  #(class x y) offsetFromClassOfInstanceVariable: #x
  #(x :y)
  nil
) >>> 2

```




Figure 2-3 Instance variable offset asked to the class.

2.9 Accessing object instance variable values

A first simple method.

The following test illustrate the behavior of the message `offsetFromClassOfInstanceVariable:`

```

ObjTest >> testIVOffset
  "(self selector: #testIVOffset) run"

  self assert: ((pointClass offsetFromClassOfInstanceVariable: #x)
    = 2).
  self assert: ((pointClass offsetFromClassOfInstanceVariable:
    #lulu) = 0)

```

Your job.

In the protocol 'iv management' define a method called `offsetFromClassOfInstanceVariable: aSymbol` that returns the offset of the instance variable represented by the symbol given in parameter. It returns 0 if the variable is not defined. Look at the tests `#testIVOffset` of the class `ObjTest`.

Hints: Use the Pharo method `indexOf:.` Pay attention that such a primitive is applied to an `objClass` as shown in the test.

Make sure that you execute the test method: `testIVOffset`

A second simple method.

The following test illustrates the expected behavior

```

ObjTest >> testIVOffsetAndValue
  "(self selector: #testIVOffsetAndValue) run"

  self assert: ((aPoint offsetFromObjectOfInstanceVariable: #x) =
    2).

```

```

#( #ObjClass #ObjPoint #ObjObject #(class
x y) #(x :y) nil )
      ^
      |
#(Point 100 200)
  offsetFromObjectOfInstanceVariable: #x
      >>> 2

```

Figure 2-4 Instance variable offset asked to the instance itself.

```

self assert: ((aPoint valueOfInstanceVariable: #x) = 10)

```

Your job.

Using the preceding method, define in the protocol 'iv management'

1. the method `offsetFromObjectOfInstanceVariable: aSymbol` that returns the offset of the instance variable. Note that this time the method is applied to an `objInstance` presenting an instance and not a class (as shown in Figure 2-4).
2. the method `valueOfInstanceVariable: aSymbol` that returns the value of this instance variable in the given object as shown in the test below.

Note that for the method `offsetFromObjectOfInstanceVariable:` you can check that the instance variable exists in the class of the object and else raise an error using the Pharo method `error:.`

Make sure that you execute the test method: `testIVOffsetAndValue` and it passes.

2.10 Object allocation and initialization

The creation of an object is the composition of two elementary operations: its *'allocation'* and its *'initialization'*.

We now define all the primitives that allow us to allocate and initialize an object. Remember that

1. the allocation is a class method that returns a nearly empty structure, nearly empty because the instance represented by the structure should at least know its class and

2. the initialization of an instance is an instance method that given a newly allocated instance and a list of initialization arguments fill the instance.

Instance allocation

As shown in the class `ObjTest`, if the class `ObjPoint` has two instance variables: `ObjPoint allocateAnInstance` returns `##ObjPoint nil nil`.

```
ObjTest >> testAllocate
"(self selector: #testAllocate) run"
| newInstance |
newInstance := pointClass allocateAnInstance.
self assert: (newInstance at: 1) = #ObjPoint.
self assert: (newInstance size) = 3.
self assert: (newInstance at: 2) isNil.
self assert: (newInstance at: 3) isNil.
self assert: (newInstance objClass = pointClass)
```

Your job.

In the protocol 'instance allocation' implement the primitive called `allocateAnInstance` that sent to an `objClass` returns a new instance whose instance variable values are `nil` and whose `objClassId` represents the `objClass`.

Make sure that you execute the test method: `testAllocate`

2.11 Keywords primitives

The original implementation of `ObjVLisp` uses the facility offered by the Lisp keywords to ease the specification of the instance variable values during instance creation. It also provides an uniform and unique way to create objects. We have to implement some functionality to support keywords. However as this is not really interesting that you lose time we give you all the necessary primitives.

Your job.

All the functionality for managing the keywords are defined into the protocol 'keyword management'. Read the code and the associated test called `testKeywords` in the class `ObjTest`.

```
ObjTest >> testKeywords
"(self selector: #testKeywords) run"

| dummyObject |
dummyObject := Obj new.
self assert:
```

```

    ((dummyObject generateKeywords: (#titi #toto #lulu))
     = (#titi: #toto: #lulu:)).
self assert:
    ((dummyObject keywordValue: #x
     getFrom: (#toto 33 #x 23)
     ifAbsent: 2) = 23).
self assert:
    ((dummyObject keywordValue: #x
     getFrom: (#toto 23)
     ifAbsent: 2) = 2).
self assert:
    ((dummyObject returnValuesFrom: (#x 22 #y 35)
     followingSchema: (#y #yy #x #y))
     = #(35 nil 22 35))

```

Make sure that you execute the test method: `testKeywords` and that it passes.

2.12 Object initialization

Once an object is allocated, it may be initialized by the programmer by specifying a list of initialization values. We can represent such list by an array containing alternatively a keyword and a value like `(#toto 33 #x 23)` where 33 is associated with `#toto` and 23 with `#x`.

Your job.

Read in the protocol 'instance initialization' the primitive `initializeUsing: anArray` that sent an object with an initialization list returns an initialized object.

```

ObjTest >> testInitialize
"(self selector: #testInitialize) run"

| newInstance |
newInstance := pointClass allocateAnInstance.
newInstance initializeUsing: (#y: 2 #z: 3 #t: 55 #x: 1).
self assert: (newInstance at: 1) equals: #ObjPoint.
self assert: (newInstance at: 2) equals: 1.
self assert: (newInstance at: 3) equals: 2.

```

2.13 Static inheritance of instance variables

Instance variables are statically inherited at the class creation time. The simplest form of instance variable inheritance is to define the complete set of instance variables as the ordered fusion between the inherited instance variables and the locally defined instance variables. For simplicity reason and as

most of the languages, we chose to forbid duplicated instance variables in the inheritance chain.

Your job.

In the protocol 'iv inheritance', read and understand the primitive `computeNewIVFrom: superIVOrdCol with: localIVOrdCol`.

The primitive takes two ordered collections of symbols and returns an ordered collection containing the union of the two ordered collections but with the extra constraint that the order of elements of the first ordered collection is kept. Look at the test method `testInstanceVariableInheritance` below for examples.

Make sure that you execute the test method: `testInstanceVariableInheritance` and that it passes.

```
ObjTest >> testInstanceVariableInheritance
  "(self selector: #testInstanceVariableInheritance) run"

  "a better choice would be to throw an exception if there are
  duplicates"
  self assert:
    ((Obj new computeNewIVFrom: #(#a #b #c #d) asOrderedCollection
      with: #(#a #z #b #t) asOrderedCollection)
     = #(#a #b #c #d #z #t) asOrderedCollection).
  self assert:
    ((Obj new computeNewIVFrom: #() asOrderedCollection
      with: #(#a #z #b #t) asOrderedCollection)
     = #(#a #z #b #t) asOrderedCollection)
```

Side remark

You could think that keeping the same order of the instance variables between a superclass and its subclass is not an issue. This is partly true in this simple implementation because the instance variable accessors compute each time the corresponding offset to access an instance variable using the primitive `offsetFromClassOfInstanceVariable:.` However, the structure (instance variable order) of a class is hardcoded by the primitives. That's why your implementation of the primitive `computeNewIVFrom:with:` should take care of that aspect.

2.14 Method management

A class stores the behavior (expressed by methods) shared by all its instances into a method dictionary. In our implementation, we represent methods by associating a symbol to a Pharo *block* a kind of anonymous method. The block is then stored in the method dictionary of an `objClass`.

In this implementation we do not offer the ability to access directly instance variables of the class in which the method is defined. This could be done by sharing a common environment among all the methods. The programmer has to use accessors or the `setIV` and `getIV` objMethods defined on `ObjObject` to access the instance variables. You can find the definition of such methods in the bootstrap method on the class side of `Obj`.

In our `ObjVlisp` implementation, we do not have a syntax for message passing. Instead we call the primitives using the Pharo syntax for message passing (using the message `send:withArguments:`).

The following expression `objself getIV: x` is expressed in `ObjVlisp` as `objself send: #getIV withArguments: #(x)`.

The following code describes the definition of the accessor method `x` defined on the objClass `ObjPoint` that invokes a field access using the message `getIV`.

```
ObjPoint
  addUnaryMethod: #accessInstanceVariableX
  withBody: 'objself send: #getIV withArguments: #(x)'
```

As a first approximation this code will create the following block that will get stored into the class method dictionary. `[:objself | objself send: #getIV withArguments: #(x)]`. As you may notice, in our implementation, the receiver is always an explicit argument of the method. Here we named it `objself`.

Defining a method and sending a message

As we want to keep this implementation as simple as possible, we define only one primitive for sending a message: it is `send:withArguments:.` To see the mapping between Pharo and `ObjVlisp` ways of expressing message sent, look at the comparison below:

```
Pharo Unary: self odd
ObjVlisp: objself send: #odd withArguments: #()

Pharo Binary: a + 4
ObjVlisp: a send: #+ withArguments: #(4)

Pharo Keyword: a max: 4
ObjVlisp: a send: #max: withArguments: #(4)
```

While in Pharo you would write the following method definition:

```
bar: x
  self foo: x
```

In our implementation of `ObjVlisp` you write:

```

anObjClass
  addMethod: #bar:
  args: 'x'
  withBody: 'objself send: #foo: withArguments: #x'.

```

Your job.

We provide all the primitives that handle with method definition. In the protocol 'method management' look at the methods `addMethod: aSelector args: aString withBody: aStringBlock`, `removeMethod: aSelector` and `doesUnderstand: aSelector`. Implement `bodyOfMethod: aSelector`.

Make sure that you execute the test method: `testMethodManagement`

```

ObjTest >> testMethodManagement
  "(self selector: #testMethodManagement) run"
  self assert: (pointClass doesUnderstand: #x).
  self assert: (pointClass doesUnderstand: #xx) not.

  pointClass
    addMethod: #xx
    args: ''
    withBody: 'objself valueOfInstanceVariable: #x '.
  self assert: (((pointClass bodyOfMethod: #xx) value: aPoint) =
    10).
  self assert: (pointClass doesUnderstand: #xx).
  pointClass removeMethod: #xx.
  self assert: (pointClass doesUnderstand: #xx) not.
  self assert: (((pointClass bodyOfMethod: #x) value: aPoint) = 10)

```

2.15 Message passing and dynamic lookup

Sending a message is the result of the composition of *method lookup* and *execution*. The following `basicSend:withArguments:from:` primitive just implements it. First it looks up the method into the class or superclass of the receiver then if a method has been found it execute it, else lookup: returned nil and we raise a Pharo error.

```

Obj >> basicSend: selector withArguments: arguments from: aClass
  "Execute the method found starting from aClass and whose name is
  selector.
  The core of the sending a message, reused for both a normal send
  or a super one."
  | methodOrNil |
  methodOrNil := aClass lookup: selector.
  ^ methodOrNil
    ifNotNil: [ methodOrNil valueWithArguments: (Array with: self)
      , arguments ]

```



```
ifNil: [ Error signal: 'Obj message' , selector asString, '
not understood' ]
```

Based on this primitive we can express `send:withArguments:` and `super:withArguments:` as follows:

```
Obj >> send: selector withArguments: arguments
"send the message whose selector is <selector> to the receiver.
The arguments of the messages are an array <arguments>. The
method is looked up in the class of the receiver. self is an
objObject or a objClass."

^ self basicSend: selector withArguments: arguments from: self
objClass
```

2.16 Method lookup

The primitive `lookup: selector` applied to an `objClass` should return the method associated to the selector if it found it, else `nil` to indicate that it failed.

Your job.

Implement the primitive `lookup: selector` that sent to an `objClass` with a method selector, a symbol and the initial receiver of the message, returns the method-body of the method associated with the selector in the `objClass` or its superclasses. Moreover if the method is not found, `nil` is returned.

Make sure that you execute the test methods: `testNilWhenErrorInLookup` and `testRaisesErrorSendWhenErrorInLookup` whose code is given below:

```
ObjTest >> testNilWhenErrorInLookup
"(self selector: #testNilWhenErrorInLookup) run"

self assert: (pointClass lookup: #zork) isNil.
"The method zork is NOT implement on pointClass"

ObjTest >> testRaisesErrorSendWhenErrorInLookup
"(self selector: #testRaisesErrorSendWhenErrorInLookup) run"

self should: [ pointClass send: #zork withArguments: { aPoint }
] raise: Error.
"Open a Transcript to see the message trace"
```

2.17 Managing super

To invoke a superclass hidden method, in Java and Pharo you use `super`, which means that the lookup up will start above the class defining the method

containing the super expression. In fact we can consider that in Java or Pharo, super is a syntactic sugar to refer to the receiver but changing where the method lookup starts. This is what we see in our implementation where we do not have syntactic support.

Let us see how we will express the following situation.

```
bar: x
    super foo: x
```

In our implementation of ObjVlisp we do not have a syntactic construct to express super, you have to use the super:withArguments: Pharo message as follows.

```
anObjClass
  addMethod: #bar:
    args: 'x'
    withBody: 'objself super: #foo: withArguments: #(#x) from:
              superClassOfClassDefiningTheMethod'.
```

Note that superClassOfClassDefiningTheMethod is a variable that is bound to the superclass of anObjClass i.e., the class defining the method bar (see later).

```
Pharo Unary: super odd
ObjVlisp: objself super: #odd withArguments: #() from:
          superClassOfClassDefiningTheMethod

Pharo Binary: super + 4
ObjVlisp: objself super: #+ withArguments: #(4) from:
          superClassOfClassDefiningTheMethod

Pharo Keyword: super max: 4
ObjVlisp: objself super: #max: withArguments: #(4) from:
          superClassOfClassDefiningTheMethod
```

2.18 Representing super

We would like to explain you where the superClassOfClassDefiningTheMethod variable comes from. When we compare the primitive send:withArguments:, for super sends we added a third parameter to the primitive and we called it super:withArguments:from:.

This extra parameter corresponds to the superclass of class in which the method is defined. This argument should always have the same name, i.e., superClassOfClassDefiningTheMethod. This variable will be bound when the method is added in the method dictionary of an objClass.

If you want to understand how we bind the variable, here is the explanation: In fact, a method is not only a block but it needs to know the class that de-

finds it or its superclass. We added such information using currrification. (a currrification is the transformation of a function with n arguments into function with less argument but an environment capture: $f(x,y) = (+ x y)$ is transformed into a function $f(x)=f(y)(+ x y)$ that returns a function of a single argument y and where x is bound to a value and obtain a function generator). For example, $f(2,y)$ returns a function $f(y)=(+ 2 y)$ that adds its parameter to 2. A currrification acts as a generator of function where one of the argument of the original function is fixed.

In Pharo we wrap the block representing the method around another block with a single parameter and we bind this parameter with the superclass of the class defining the method. When the method is added to the method dictionary, we evaluate the first block with the superclass as parameter as illustrated as follows:

```
method := [ :superClassOfClassDefiningTheMethod |
  [ :objself :otherArgs |
    ... method code ...
  ]]
method value: (Obj giveClassNameed: self objSuperclassId)
```

So now you know where the `superClassOfClassDefiningTheMethod` variable comes from. Make sure that you execute the test method: `testMethod-Lookup` and that is passes.

Your job.

Now you should be implement `super: selector withArguments: arguments from: aSuperclass` using the primitive `basicSend:withArguments:from:`.

2.19 Handling not understood messages

Now we can revisit error handling. Instead of raising a Pharo error, we want to send an `ObjVlisp` message to the receiver of the message to give him a chance to trap the error.

Compare the two following versions of `basicSend: selector withArguments: arguments from: aClass` and propose an implementation of `sendError: selector withArgs: arguments`.

```
Obj >> basicSend: selector withArguments: arguments from: aClass
"Execute the method found starting from aClass and whose name is
selector."
"The core of the sending a message, reused for both a normal send
or a super one."
| methodOrNil |
methodOrNil := (aClass lookup: selector).
^ methodOrNil
```

```

        ifNotNil: [ methodOrNil valueWithArguments: (Array with: self)
, arguments ]
        ifNil: [ Error signal: 'Obj message' , selector asString, '
not understood' ]
Obj >> basicSend: selector withArguments: arguments from: aClass
"Execute the method found starting from aClass and whose name is
selector."
"The core of the sending a message, reused for both a normal send
or a super one."
| methodOrNil |
methodOrNil := (aClass lookup: selector).
^ methodOrNil
    ifNotNil: [ methodOrNil valueWithArguments: (Array with: self)
, arguments ]
    ifNil: [ self sendError: selector withArgs: arguments ]

```

It should be noted that the `objVlisp` method is defined as follows in the `ObjObject` class (see the `bootstrap` method on the class side of `Obj`). The `obj` error method expects a single parameter: an array of arguments whose first element is the selector of the not understood message.

```

objObject
    addMethod: #error
    args: 'arrayOfArguments'
    withBody: 'Transcript show: ''error '' , arrayOfArguments first.
''error '' , arrayOfArguments first'.
Obj >> sendError: selector withArgs: arguments
"send error wrapping arguments into an array with the selector as
first argument. Instead of an array we should create a message
object."

^ self send: #error withArguments: {(arguments copyWithFirst:
selector)}

```

Make sure that you read and execute the test method: `testSendErrorRaisesErrorSendWhenErrorInLookup`. Have a look at the implementation of the `#error` method defined in `ObjObject` and in the `assembleObjectClass` of the `ObjTest` class.

2.20 Bootstrapping the system

Now you have implemented all the behavior we need, you are ready to bootstrap the system: this means creating the kernel consisting of `ObjObject` and `ObjClass` classes from themselves. The idea of a smart bootstrap is to be as lazy as possible and to use the system to create itself by creating fast a fake but working first class with which we will build the rest.

Three steps compose the `ObjVlisp` bootstrap,

1. we create by hand the minimal part of the objClass ObjClass and then
2. we use it to create normally ObjObject objClass and then
3. we recreate normally and completely ObjClass.

These three steps are described by the following bootstrap method of Obj class. Note the bootstrap is defined as class methods of the class Obj.

```
Obj class >> bootstrap
  "self bootstrap"

  self initialize.
  self manuallyCreateObjClass.
  self createObjObject.
  self createObjClass.
```

To help you to implement the functionality of the objClasses ObjClass and ObjObject, we defined another set of tests in the class ObjTestBootstrap. Read them.

2.21 Manually creating ObjClass

The first step is to create manually the class ObjClass. By manually we mean create an array (because we chose an array to represent instances and classes in particular) that represents the objClass ObjClass, then define its methods. You will implement/read this in the primitive manuallyCreateObjClass as shown below:

```
Obj class >> manuallyCreateObjClass
  "self manuallyCreateObjClass"

  | class |
  class := self manualObjClassStructure.
  Obj declareClass: class.
  self defineManualInitializeMethodIn: class.
  self defineAllocateMethodIn: class.
  self defineNewMethodIn: class.
  ^ class
```

For this purpose, you have to implement/read all the primitives that compose it.

Your job.

At the class level in the protocol 'bootstrap objClass manual' read or implement: the primitive manualObjClassStructure that returns an objObject that represents the class ObjClass.

Make sure that you execute the test method: testManuallyCreateObjClassStructure

- As the initialize of this first phase of the bootstrap is not easy we give you its code. Note that the definition of the objMethod initialize is done in the primitive method defineManualInitializeMethodIn:.

```
Obj class >> defineManualInitializeMethodIn: class

class
  addMethod: #initialize
  args: 'initArray'
  withBody:
    '| objsuperclass |
     objself initializeUsing: initArray. "Initialize a class as an
     object. In the bootstrapped system will be done via super"
     objsuperclass := Obj giveClassNamed: objself objSuperclassId
  ifAbsent: [nil].
     objsuperclass isNil
     ifFalse:
       [ objself
         objIVs: (objself computeNewIVFrom: objsuperclass objIVs
         with: objself objIVs)]
     ifTrue:
       [ objself objIVs: (objself computeNewIVFrom: #(#class)
         with: objself objIVs)].
     objself
     objKeywords: (objself generateKeywords: (objself objIVs
     copyWithout: #class)).
     objself objMethodDict: (IdentityDictionary new: 3).
     Obj declareClass: objself.
     objself'
```

Note that this method works without inheritance since the class ObjObject does not exist yet.

The primitive defineAllocateMethodIn: anObjClass defines in anObjClass passed as argument the objMethod allocate. allocate takes only one argument: the class for which a new instance is created as shown below:

```
defineAllocateMethodIn: class

class
  addUnaryMethod: #allocate
  withBody: 'objself allocateAnInstance'
```

Following the same principle, define the primitive defineNewMethodIn: anObjClass that defines in anObjClass passed as argument the objMethod new. new takes two arguments: a class and an initargs-list. It should invoke the objMethod allocate and initialize.

Your job.

Make sure that you read and execute the test method: `testManuallyCreateObjClassAllocate`

Remarks

Read carefully the following remarks below and the code.

- In the objMethod `manualObjClassStructure`, the instance variable inheritance is simulated. Indeed the instance variable array contains `#class` that should normally be inherited from `ObjObject` as we will see in the third phase of the bootstrap.
- Note that the class is declared into the class repository using the method `declareClass:.`
- Note the method `#initialize` is method of the metaclass `ObjClass`: when you create a class the initialize method is invoked on a class! The `initialize` objMethod defines on `ObjClass` has two aspects: the first one dealing with the initialization of the class like any other instance (first line). This behavior is normally done using a super call to invoke the `initialize` method defined in `ObjObject`. The final version of the `initialize` method will do it using `perform`. The second one dealing with the initialization of classes: performing the instance variable inheritance, then computing the keywords of the newly created class. Note in this final step that the keyword array does not contain the `#class: keyword` because we do not want to let the user modify the class of an object.

2.22 Creation of ObjObject

Now you are in the situation where you can create the first real and normal class of the system: the class `ObjObject`. To do that you send the message `new` to class `ObjClass` specifying that the class you are creating is named `#ObjObject` and only have one instance variable called `class`. Then you will add the methods defining the behavior shared by all the objects.

Your job: objObjectStructure

Implement/read the following primitive `objObjectStructure` that creates the `ObjObject` by invoking the `new` message to the class `ObjClass`:

```
Obj class >> objObjectStructure
  ^ (self giveClassName: #ObjClass)
    send: #new
    withArguments: ((#(#name: #ObjObject #iv: (#class)))
```

The class `ObjObject` is named `ObjObject`, has only one instance variable `cClass` and does not have a superclass because it is the inheritance graph root.

Your job: createObjObject

Now implement the primitive `createObjObject` that calls `objObjectStructure` to obtain the `objObject` representing `objObject` class and define methods in it. To help you we give here the beginning of such a method

```
Obj class >> createObjObject
| objObject |
objObject := self objObjectStructure.
objObject addUnaryMethod: #class withBody: 'objself objClass'.
objObject addUnaryMethod: #isClass withBody: 'false'.
objObject addUnaryMethod: #isMetaClass withBody: 'false'.
...
...
^ objObject
```

Implement the following methods in `ObjObject`

- the `objMethod class` that given an `objInstance` returns its class (the `objInstance` that represents the class).
- the `objMethod isClass` that returns `false`.
- the `objMethod isMetaClass` that returns `false`.
- the `objMethod error` that takes two arguments the receiver and the selector of the original invocation and raises an error.
- the `objMethod getIV` that takes the receiver and an attribute name, `aSymbol`, and returns its value for the receiver.
- the `objMethod setIV` that takes the receiver, an attribute name and a value and sets the value of the given attribute to the given value.
- the `objMethod initialize` that takes the receiver and an `initargs-list` and initializes the receiver according to the specification given by the `initargs-list`. Note that here the `initialize` method only fill the instance according to the specification given by the `initargs-list`. Compare with the `initialize` method defined on `ObjClass`.

Make sure that you read and execute the test method: `testCreateObjObjectStructure`

In particular notice that this class does not implement the class method `new` because it is not a metaclass but does implement the instance method `initialize` because any object should be initialized.

Your job: run the tests

- Make sure that you read and execute the test method: `testCreateObjObjectMessage`
- Make sure that you read and execute the test method: `testCreateObjObjectInstanceMessage`

2.23 Creation of ObjClass

Following the same approach, you can now recreate completely the class `ObjClass`. The primitive `createObjClass` is responsible to create the final class `ObjClass`. So you will implement it and define all the primitive it needs. Now we only define what is specific to classes, the rest is inherited from the superclass of the class `ObjClass`, the class `ObjObject`.

```
Obj class >> createObjClass
  "self bootstrap"

  | objClass |
  objClass := self objClassStructure.
  self defineAllocateMethodIn: objClass.
  self defineNewMethodIn: objClass.
  self defineInitializeMethodIn: objClass.
  objClass
    addUnaryMethod: #isMetaClass
      withBody: 'objself objIVs includes: #superclass'.
  "an object is a class if is class is a metaClass. cool"

  objClass
    addUnaryMethod: #isClass
      withBody: 'objself objClass send: #isMetaClass
withArguments:#()'.

  ^ objClass
```

To make the method `createObjClass` working we should implement the method it calls. Implement then:

- the primitive `objClassStructure` that creates the `ObjClass` class by invoking the `new` message to the class `ObjClass`. Note that during this method the `ObjClass` symbol refers to two different entities because the new class that is created using the old one is declared in the class dictionary with the same name.

Your job.

Make sure that you read and execute the test method: `testCreateObjClassStructure`. Now implement the primitive `createObjClass` that starts

as follow:

```
Obj class >> createObjClass

| objClass |
objClass := self objClassStructure.
self defineAllocateMethodIn: objClass.
self defineNewMethodIn: objClass.
self defineInitializeMethodIn: objClass.
...
^ objClass
```

Also define the following methods:

- the objMethod isClass that returns true.
- the objMethod isMetaclass that returns true.

```
objClass
addUnaryMethod: #isMetaclass
withBody: 'objself objIVs includes: #superclass'.

"an object is a class if is class is a metaclass. cool"
```

```
objClass
addUnaryMethod: #isClass
withBody: 'objself objClass send: #isMetaclass withArguments: #()'
```

- the primitive defineInitializeMethodIn: anObjClass that adds the objMethod initialize to the objClass passed as argument. The objMethod initialize takes the receiver (an objClass) and an initargs-list and initializes the receiver according to the specification given by the initargs-list. In particular, it should be initialized as any other object, then it should compute its instance variable (i.e., inherited instance variables are computed), the keywords are also computed, the method dictionary should be defined and the class is then declared as an existing one. We provide the following template to help you.

```
Obj class>>defineInitializeMethodIn: objClass

objClass
addMethod: #initialize
args: 'initArray'
withBody:
  'objself super: #initialize withArguments: {initArray} from:
  superClassOfClassDefiningTheMethod.
  objself objIVs: (objself
    computeNewIVFrom:
      (Obj giveClassNamed: objself
objSuperclassId) objIVs
    with: objself objIVs).
  objself computeAndSetKeywords.'
```

```

    objself objMethodDict: IdentityDictionary new.
    Obj declareClass: objself.
    objself'

Obj class >> defineInitializeMethodIn: objClass

objClass
  addMethod: #initialize
  args: 'initArray'
  withBody:
    'objself super: #initialize withArguments: {initArray}
    from: superClassOfClassDefiningTheMethod.
    objself objIVs: (objself
      computeNewIVFrom: (Obj giveClassName: objself
        objSuperclassId) objIVs
      with: objself objIVs).
    objself computeAndSetKeywords.
    objself objMethodDict: IdentityDictionary new.
    Obj declareClass: objself.
    objself'
```

Your job.

Make sure that you execute the test method: `testCreateObjClassMessage`.

Note the following points:

- The locally specified instance variables now are just the instance variables that describe a class. The instance variable `class` is inherited from `ObjObject`.
- The `initialize` method now does a `super` send to invoke the initialization performed by `ObjObject`.

2.24 First User Classes: ObjPoint

Now that `ObjVLisp` is created and we can start to program some classes. Implement the class `ObjPoint` and `ObjColoredPoint`. Here is a possible implementation.

You can choose to implement it at the class level of the class `Obj` or even better in class named `ObjPointTest`.

Pay attention that your scenario covers the following aspects:

- First just create the class `ObjPoint`.
- Create an instance of the class `ObjPoint`.
- Send some messages defined in `ObjObject` to this instance.

Define the class `ObjPoint` so that we can create points as below (create a `Pharo` method to define it).

```
[ ObjClass send: #new
  withArguments: #((#name: #ObjPoint #iv: #(#x y) #superclass:
    #ObjObject)).

aPoint := pointClass send: #new withArguments: #((#x: 24 #y: 6)).
aPoint send: #getIV withArguments: #(#x).
aPoint send: #setIV withArguments: #(#x 25).
aPoint send: #getIV withArguments: #(#x).
```

Then add some functionality to the class `ObjPoint` like the methods `x`, `x:`, `display` which prints the receiver.

```
[ Obj ObjPoint
  addUnaryMethod: #givex
  withBody: 'objself valueOfInstanceVariable: #x '.
Obj ObjPoint
  addUnaryMethod: #display
  withBody:
    'Transcript cr;
     show: ''aPoint with x = ''.
    Transcript show: (objself send: #givex withArguments: #())
    printString;
    cr'.
```

Then test these new functionality.

```
[ aPoint send: #x withArguments: #().
aPoint send: #x: withArguments: #(33).
aPoint send: #display withArguments: #().
```

2.25 First User Classes: `ObjColoredPoint`

Following the same idea, define the class `ObjColored`.

Create an instance and send it some basic messages.

```
[ aColoredPoint := coloredPointClass
  send: #new
  withArguments: #((#x: 24 #y: 6 #color: #blue)).

aColoredPoint send: #getIV withArguments: #(#x).
aColoredPoint send: #setIV withArguments: #(#x 25).
aColoredPoint send: #getIV withArguments: #(#x).
aColoredPoint send: #getIV withArguments: #(#color).
```

Your job.

Define some functionality and invoke them: the method `color`, implement the method `display` so that it invokes the superclass and adds some information related to the color. Here is an example:

```
coloredPointClass addUnaryMethod: #display
  withBody:
    'objself super: #display withArguments: #() from:
    superClassOfClassDefiningTheMethod.
    Transcript cr;
    show: '' with Color = ''.
    Transcript show: (objself send: #giveColor withArguments: #())
    printString; cr'.

aColoredPoint send: #x withArguments: #().
aColoredPoint send: #color withArguments: #().
aColoredPoint send: #display withArguments: #()
```

2.26 A First User Metaclass: ObjAbstract

Now implement the metaclass `ObjAbstract` that defines instances (classes) that are abstract i.e., that cannot create instances. This class should raise an error when it executes the new message.

Then the following shows you a possible use of this metaclass.

```
ObjAbstractClass
  send: #new
  withArguments: #(#(#name: #ObjAbstractPoint
    #iv: #()
    #superclass: #ObjPoint)).

ObjAbstractPoint send: #new
  withArguments: #(#(#x: 24 #y: 6)) "should raise an error"
```

You should redefine the new method. Note that the `ObjAbstractClass` is an instance of `ObjClass` because this is a class and inherits from it because this is a metaclass.

2.27 New features that you could implement

You can implement some simple features:

- define a metaclass that automatically defines accessors for the specified instances variables.
- avoid that we can change the selector and the arguments when calling a super send.

Shared Variables

Note that contrary to the proposition made in the 6th postulate of the original ObjVLisp model, class instance variables are not equivalent of shared variables. According to the 6th postulate, a shared variable will be stored into the instance representing the class and not in an instance variable of the class representing the shared variables. For example if a workstation has a shared variable named `domain`. But `domain` should not be an extra instance variable of the class of `Workstation`. Indeed `domain` has nothing to do with class description.

The correct solution is that `domain` is a value hold into the list of the shared variable of the class `Workstation`. This means that a *class* has an extra information to describe it: an instance variable `sharedVariable` holding pairs. So we should be able to write

```
Obj Workstation getIV: #sharedVariable
or
Obj Workstation sharedVariableValue: #domain

and get
#((domain 'inria.fr'))
```

introduce shared variables: add a new instance variable in the class `ObjClass` to hold a dictionary of shared variable bindings (a symbol and a value) that can be queried using specific methods: `sharedVariableValue:`, `sharedVariableValue:put:`.

Selected definitions

Smith was the first to introduce reflection in a programming language with 3Lisp . He defines reflection as:

- An entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates on and deals with its primary subject matter.

In the context of meta-object protocols, Bobrow refines the definition as follows:

- Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: *introspection* and *intercession* (...) Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*.

Maes proposed some definitions for reflexive programming :

- A *computational system* is something that *reasons* about and *acts* upon some part of the world, called the *domain* of the system.
- A computational system may also be *causally connected* to its domain. This means that the system and its domain are linked in such a way that if one of the two changes, this leads to an effect upon the other.
- A *meta-system* is a computational system that has as its domain another computational system, called its *object-system*. (...) A meta-system has a representation of its object-system in its data. Its program specifies *meta-computation* about the object-system and is therefore called a *meta-program*.
- *Reflection* is the process of reasoning about and/or acting upon oneself.

- A *reflective system* is a causally connected meta-system that has as object-system itself. The data of a reflective system contain, besides the representation of some part of the external world, also a causally connected representation of itself, called *self-representation* of the system. [...] When a system is reasoning or acting upon itself, we speak of *reflective computation*.
- A language with a *reflective architecture* is a language in which all systems have access to a causally connected representation of themselves.
- A programming environment has a *meta-level architecture* if it has an architecture which supports meta-computation, without supporting reflective computation.
- The *meta-object* of an object X represents the explicit information about X (e.g. about its behavior and its implementation). The object X itself groups the information about the entity of domain it represents.