Chapter 1

Building ObjVlisp a Minimal, Uniform and Reflective Object-Oriented Language Kernel

This tutorial written by Stéphane Ducasse will step by step guide you to build the kernel of the ObjVLisp model. ObjVlisp was designed by P. Cointe as got inspired by the kernel of Smalltalk 78. It has explicit metaclasses and it is composed of two classes `Object` and `Class`.

1.1 Objectives

During the lecture 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 in the lecture. Here are some of the points you can deeply understand while doing the exercise.

- 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?

1.2 Before Starting

In this section we discuss the files that you will use, the implementation choices and the conventions that we will follow during all this tutorial.

Provided Files

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

All the necessary files 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. 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 to 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.

To load the code open a monticello browser, add a file repository to point to the ObjVLispSkeleton project under StephaneDucasse in ht eObjVLispSkeleton project at http://www.smalltalkhub.com and select and load the package.

To do this, 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.

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.

Inheriting from 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.

Every object in the ObjVLisp world is instance of Obj in our implementation world (Pharo). In Pharo Obj is a subclass of Array.

Since Obj is a subclass of Array, #(#ObjPoint 10 15) is an objInstance of the class ObjPoint. ObjPoint is the name of an objClass. #(#ObjClass #ObjPoint #ObjObject #(class x y) #(:x :y) nil) is the array that represents the objclass ObjPoint.

About representation choices

We could have implemented ObjVLisp functionality at the class level of a class named Obj inheriting only from Object. However, to use the ObjVlisp primitive (a Pharo method) objInstanceVariableValue: anObject for: anInstanceVariable that returns the value of the instance variable in anObject, 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 a subclass of Array named Obj.

That way we can write in a more natural and readable way the previous functionality as:

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

Facilitating ObjVLisp class access

We need a way to declare, 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 instance of ObjClass given as arguments in the class repository (here a dictionary whose keys are the names of the classes and values the ObjVLisp classes themselves).

- giveClassNamed: aSymbol returns if it exists the ObjVLisp

class named aSymbol. 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
```

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
```

Now you are ready to start.

1.3 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 used the terms array for talking about instances of the class Obj.

Note that we could extend the model so that the metaclasses support possible instance structure changes but in the current implementation we will simply hardcode the class structure.

Your Job.

Check that the class Obj exists and inherits from Array.

Structure of a Class

As one of the first objects that we will create is the class ObjClass we focus now on the minimal structure of the classes in our language. Given an array 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.

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).
```

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

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

Implement the primitives that are missing to run the following tests testPrimitiveStructureObjClassId, testPrimitiveStructureObjIVs, testPrimitiveStructureObjKeywords, testPrimitiveStructureObjMethodDict, testPrimitiveStructureObjName, testPrimitiveStructureObjIVs 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 category 'object structure primitives' the primitives that manage:

- the class of the instance represented as a symbol. objClassId, objClassId: aSymbol. The receiver is an objObject. This means that this primitive can be applied on any objInstances to get its class identifier.

Implement in category '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.

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.

Implement the following:

Using the primitive giveClassNamed: aSymbol defined at the class level of Obj, define the primitive objClass in the category '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.

In the category '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 primitive is applied to an objClass as shown in the test.

```
ObjText >> testIVOffset
  "(self  selector: #testIVOffset) run"

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

Make sure that you execute the test method: testIVOffset

Using the preceding method, define in the category '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.

2. the method valueOfInstanceVariable: aSymbol that returns the value of this instance variable in the given object as shown in the test below.

The following test illustrates the expected behavior

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

  self assert: ((aPoint offsetFromObjectOfInstanceVariable: #x ) = 2).
  self assert: ((aPoint valueOfInstanceVariable: #x ) = 10)
```

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.

1.4 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 knows 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

Your Job.

In the category '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.

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)
```

Make sure that you execute the test method: testAllocate

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 category '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.

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 category '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.
```

1.5 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 category '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 is 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 computes 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.

1.6 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 an 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 them in the bootstrap method on the class side of Obj.

In our ObjVLisp implementation, we do not have a syntax for message passing. Instead of we call the primitives using the Pharo syntax for message passing (using the message send:withArguments:) The following expression objself getIV: x is expressed as 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 category '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: #testMethodManagment) 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)
```

1.7 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 lookep up in the
      class of the receiver. self is an objObject or a objClass."

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

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"
```

1.8 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
```

Representing super

We would like to explain you where the superClassOfClassDefiningTheMethod variable comes from. When we compare the primitive send:withAguements:, 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 defines it or its superclass. We added such information using currification. (a currification 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 currification 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 giveClassNamed: self objSuperclassId)
```

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

Your Job.

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

1.9 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.

1.10 Bootstrapping the system

Now you have implemented all the behavior we need and 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. Three steps compose the 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.

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 category '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.

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

Your Job

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.

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

Implement/read the following methods:

- the primitive objObjectStructure that creates the ObjObject by invoking the new message

to the class ObjClass:

```
Obj class >> objObjectStructure

  ^ (self giveClassNamed: #ObjClass)
    send: #new
    withArguments: #(#(#name: #ObjObject #iv: #(#class)))
```

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

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 method 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.

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

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

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.

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
```

- 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 initializesthe 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 giveClassNamed: objself objSuperclassId) objIVs
        with: objself objIVs).
      objself computeAndSetKeywords.
      objself objMethodDict: IdentityDictionary new.
      Obj declareClass: objself.
      objself'
```

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.

1.11 First User Classes: ObjPoint and ColoredObjPoint

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

ObjPoint

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: #().
```

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).
```

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: #()
```

1.12 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.

1.13 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 pair. 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:.