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

Stéphane Ducasse
(Alexandre Bergel and Simon Denier)
Language and Software Evolution
INRIA - Lille Nord Europe, CNRS UMR 8022 - LIFL-USTL
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr

December 17, 2013

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?

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.

2.1 Provided Files

You need to download and install Pharo from http://www.pharo-project.org/. You need a virtual machine, and the couple image and changes. You can use the http://get.pharo.org to get a script to download Pharo. You can use the book Pharo by Example from http://www.pharo.project.org/PharoByExample/ for an overview of the syntax and the system. You can check some old videos available at http://stephane.ducasse.free.fr/Videos SqueakOriginalMov/.

All the necessary files are provided as 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 you have to evaluate the following expression (ObjTest selector: #testPrimitiveStructure) run or to click on button run once you selected the method named testPrimitiveStructure.

Note that since you are developing the kernel, to test it we implemented manually a mock of the kernel. This is the setup method of the test classes that build this kernel.

To load the code open a monticello browser, add a file repository to point to the ObjVLispSkeleton project under StephaneDucasse 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 ObjVSkeleton-StephaneDucasse.ducasse.11.mcz

2.2 Conventions

We use the following conventions: we name as *primitives* all the Smalltalk 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 are just lambda expressions, in a C implementation such primitives will be represented by functions.

In the same way to help you to distinguish between classes in the implementation language (Smalltalk) and the ObjVLisp model, we prefix 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 is named objClassId.

We also talk about objInstances, objObjects and objClasses to refer to specific instances, objects or classes defined in ObjVLisp. For example, #(#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 an objClass.

2.3 Implementation Choices

Every object in the Object-Lisp world is instance of Obj in our implementation world (Smalltalk). In Smalltalk Obj is a subclass of Array.

2.3.1 Implementation Inheritance.

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 Smalltalk. As Smalltalk does not contain an easy way to define macroes, we will use as much as possible the existing classes to avoid extra syntaxic problems.

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 Smalltalk 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 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'.

2.3.2 Facilitating ObjVLisp class access.

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

- declareClass: anObjClass stores an ObjClass 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 whose name is 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) objClass
```

Now you are ready to start.

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 objects using arrays, in fact instances of Obj a subclass of Array. 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.

3.1 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 (in the following we used the terms array for talking about instances of the class Obj) 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 ObjObject, has three instance variables, two initialization keywords and an uninitialized method dictionary. To access this structure we define some primitives.

To help you to implement ObjVLisp, we provide you an inspector dedicated to the inspection of ObjVLisp objects. You can invoke this inpector sending the message debug to an objInstance or sending the message openOn: to ObjClassInspector with the objInstance as parameter.

```
| pointClass |
pointClass := Obj giveClassNamed: #ObjPoint.
pointClass debug.
```

```
| pointClass |
pointClass := Obj ObjPoint.
pointClass debug.
```

```
| pointClass |
pointClass := Obj ObjPoint.
ObjClassInspector openOn: pointClass
```

```
| aPt |
aPt := Obj new: 3.
aPt at: 1 put: #ObjPoint3.
aPt debug
```

Your job: The test methods of the class ObjTest that are in the categories 'structure of objects' and 'structure of classes give some examples of structure accesses. 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 (ObjTest selector: #testPrimitiveStructureObjClassId) run. Note that arrays start at 1 in Smalltalk. 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.

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.

3.2 Finding the class of an object

Every object keeps the identifier of its class (its name). We do not keep directly its class to avoid endless recursion.

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 primitives:

- 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).

  Evaluate: (ObjTest selector: #testClassAccess) run.

- In the category 'iv management' define a method called offsetFromClassOfInstanceVariable: aSymbol that returns the offset of the instance variable represented by the symbol. It returns 0 if the variable is not defined. Look at the tests #testIVOffset. (Hints: Use the Smalltalk method indexOf:).

  Evaluate: (ObjTest selector: #testIVOffset) run.

- Using the preceeding method define in the category 'iv management' (a) the method offsetFromObjectOfInstanceVariable: aSymbol that returns the offset of the instance variable and (b) the method valueOfInstanceVariable: aSymbol that returns the value of this instance variable in the given object. Look at the tests #testIVOffsetAndValue. 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 method error:.

  Evaluate: (ObjTest selector: #testIVOffsetAndValue) run.

4 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. Remind that (a) 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 (b) the initialization of an instance is an instance method that given a newly allocated instance and a list of initialization arguments fill the instance.

4.1 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 classId represents the objClass.

As shown in the class ObjTest, if the class ObjPoint has two instance variables: ObjPoint allocateAnInstance returns #(#ObjPoint nil nil).
Evaluate: (ObjTest selector: #testAllocate) run.

4.2 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 then providing an uniform and unique way to create object. 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'. So look at the code and the associated test called testKeywords in the class ObjTest.
Evaluate: (ObjTest selector: #testKeywords) run.

4.3 Initialization

Once an object is allocated, it may be initialized by the programmer by specifying a list of initialization values, called initargs-list. We can represent an initargs-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 initargs-list returns an initialized object.

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 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 for examples. Evaluate: (ObjTest selector: #testInstanceVariableInheritance) run.

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 Smalltalk *block i.e.,* 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.

The following code describes the definition of the method x defined on the objClass ObjPoint that invokes a field access

```
ObjPoint
    addMethod: #bar
    args:
    withBody: objself binarySend: #getIV with: #x.
```

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

In the ObjVLisp world, we do not have a syntax for message passing. Instead of we call the primitives using the Smalltalk syntax.

Defining a method and sending a message. While in Smalltalk 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 binarySend: #foo: with: #x'.
```

Note the the block is not part of the syntax of ObjectLisp since we need to attach extra information the block that will be created.

Invoking Super. To invoke a superclasses' hidden method, in Java and Smalltalk 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 Smalltalk, 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.

```
bar: x

  super foo: x
```

In our implementation of ObjVlisp we do not have a syntactic construct to express super, you have to write:

```
anObjClass
    addMethod: #bar
    args: 'x'
    withBody:    'objself binarySuper: #foo: with: #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).

As we want to keep this implementation as simple as possible and that Smalltalk does not support the concept of argument representing a list of values like the dot notation in C or Lisp. We will ask you to define 6 primitives (which could be only two in fact) to send messages to an object corresponding to {unary binary or keyword} cross {super or self} send. For a clearer view take a look at the Table 1

Smalltalk Syntax	ObjectLisp equivalent
Temporary: | a |	| a |
Assignment: a := 3	a := 3
Sends	
Unary: self odd	objself unarySend: #odd
Binary: a + 4	a binarySend: #+ with: #(4)
Keyword: a max: 4	s send: #max: withArguments: #(4)
Super Sends	
Unary: super odd	objself unarySuper: #odd from: superClassOfClassDefiningTheMethod
Binary: super + 4	objself binarySuper: #+ with: #(4) from: superClassOfClassDefiningTheMethod
Keyword: super max: 4	objself super: #max: withArguments: #(4) from: superClassOfClassDefiningTheMethod

Table 1: ObjectLisp Syntax

Your job: We provide you all the primitives that deals 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. Evaluate: (ObjTest selector: #testMethodManagement) run.

7 Message Passing and Dynamic Lookup

Sending a message is the result of the composition of method lookup and execution. The following unarySend: aSelector primitive just implements it. First it looks up the method into the class or superclass of the receiver then binds the method (returned block) parameters to the only argument of the message, here the receiver object (self).

```
Obj>>unarySend: selector
    | ans |
    ans := (self objClass lookup: selector for: self) value: self.
    ^ ans
```

7.1 Method Lookup

Your job: Implement the primitive lookup: selector for: anObjObject 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, the message #error is sent to an objInstance with aString representing the error. Note here that error should be sent to the receiver. Evaluate: (ObjTest selector: #testMethodLookup) run.

7.2 Send Methods

Your job: Implement the other primitives for message passing: one for binary messages binarySend: selector with: argument and one for n-ary messages send: selector withArguments: arguments. Evaluate: (ObjTest selector: #testMethodSelfSend) run.

7.3 Representing super

We would like to explain you where the superClassOfClassDefiningTheMethod variable comes from. For super sends we add a parameter to the primitive. This parameter corresponds to the super class where 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.

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 $f(x) = f(y)(+\ x\ y)$ where we bind x to a value and obtain a function generator).

In Smalltalk we wrapped the block representing the method around another block with a single parameter and binds 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 core ...
      ]]
method value: Obj giveClassNamed: self objSuperclassId
```

So now you know where the superClassOfClassDefiningTheMethod variable comes from. Evaluate: (ObjTest selector: #testMethodLookup) run.

7.4 Implementing Super Sends

Your job: Implement three different primitives for super message passing invocation: one for unary messages unarySuper: selector from: anObjClass, one for binary messages binarySuper: selector with: argument from: aSuperClass: and one for n-ary messages super: selector withArguments: arguments from: aSuperclass.

You can get inspired by the methods you should have written earlier.

```
Obj>>binarySend: selector with: argument
   | ans |
   ans := (self objClass lookup: selector for: self) value: self value: argument.
   ˆ ans

Obj>>send: selector withArguments: arguments
   | ans |
   ans := (self objClass lookup: selector for: self)
                  valueWithArguments: (Array with: self) , arguments.
   ˆ ans
```

8 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 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.

8.1 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.

  Evaluate: (ObjTestBootstrap selector: #testManuallyCreateObjClassStructure) run.

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

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

- the primitive defineNewMethodIn: anObjClass that defines in anObjClass (the class passed as argument) the objMethod new. new takes two arguments: a class and an initargs-list.
- the primitive defineAllocateMethodIn: anObjClass that defines in anObjClass (the class passed as argument) the objMethod allocate. allocate takes only one argument: the class for which a new instance is created.

  Evaluate: (ObjTestBootstrap selector: #testManuallyCreateObjClassAllocate) run.

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 list 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 automatically declared into the class repository using the method declareClass:.
- Note the method #initialize is method of the metaclass Class: 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 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 list does not contain the #class: keyword because we do not want to let the user modify the class of an object.

8.2 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 primitive objObjectStructure that creates the ObjObject by invoking the new message to the class ObjClass

  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 primitive

```
Obj class>>createObjObject
    | objObject |
    objObject := self objObjectStructure.
    objObject addMethod: #class withBody: [:object | object objClass].
    objObject addMethod: #isClass withBody: [:object | 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.

Evaluate: (ObjTestBootstrap selector: #testCreateObjObjectStructure) run.

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.
Evaluate: (ObjTestBootstrap selector: #testCreateObjObjectMessage) run.
Evaluate: (ObjTestBootstrap selector: #testCreateObjObjectInstanceMessage) run.

8.3 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.

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.

  Evaluate: (ObjTestBootstrap selector: #testCreateObjClassStructure) run.

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.

Note that we could have an alternate implementation for isClass and isMetaclass as shown hereafter.

```
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 unarySend: #isMetaclass'.
```

- 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 binarySuper: #initialize with: 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'
```

Evaluate: (ObjTestBootstrap selector: #testCreateObjClassMessage) run.
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.

9 First User Classes: ObjPoint and ColoredObjPoint

Now ObjVLisp is created and we can start to program some classes. Implement the class ObjPoint and ObjColoredPoint as follow.

9.1 ObjPoint

You can choose to implement it at the class level of the class Obj.

- 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 Point so that we can create points as follows:

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

Then add some functionality to the class ObjPoint like x, x:, display which prints the receiver.
Then test these new functionality.

```
aPoint unarySend: #x.
aPoint binarySend: #x: with: #(33).
aPoint unarySend: #display
```

9.2 ObjColoredPoint

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 binarySend: #getIV with: #x.
aColoredPoint send: #setIV withArguments: #(#x 25).
aColoredPoint binarySend: #getIV with: #x.
aColoredPoint binarySend: #getIV with: #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.

```
aColoredPoint unarySend: #x.
aColoredPoint unarySend: #color.
aColoredPoint unarySend: #display
```

10 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"
```

Note that the ObjAbstractClass is an instance of ObjClass because this is a class and inherits from of because this is a metaclass.

11 New features that you could implement

You can:

- 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.

- 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 'iam.unibe.ch'))
  ```

  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:.

12 About super Implementation

Sending a message is the result of the composition of the method lookup and the method application. Now we discuss an alternative way of implementing super. The idea is to keep a stack of classes.

The following unarySend: aSelector primitive just implements it. First it looks up the method into the class or superclass of the receiver then binds the method (returned block) parameters to the only argument of the message, here the receiver object (self).

```
Obj>>unarySend: selector
   | ans |
   ans := (self objClass lookup: selector for: self) value: self.
   ClassesImplementingLookupMethod removeLast.
   ^ ans
```

The variable ClassesImplementingLookupMethod represents a stack of classes.

The Problem. Whenever a message is sent to an object, the class where the lookup has to start is given by the receiver object. The lookup starts in its class. The same is true when a message is sent to self or this pseudo-variables. But things are slightly more complicated when the receiver is super. The question is: From which class the lookup has to start whenever the receiver of a message is super? The answer is the superclass of the class defining the method (the one which sends a message to super). The problem is this class need to be referenced, and that is what ClassesImplementingLookupMethod is doing.

In the implementation suggested before as well as in Java and Smalltalk there is no such a problem because during the compilation of a method, whenever super is encountered, the compiler knows what the super class is: it corresponds to the superclass of the class being compiled, and the compiler marks this information within the compiled method itself. Note that this superclass has nothing to do with the receiver!

In the case of this alternate implementation, when executed, a method has a reference to the receiver, but has not a reference to the class which defines this method. And it is necessary to know it when super is used. This information has to be computed at run-time, while in Java or Smalltalk it is computed as compile-time. The trick for always keeping the class defining the actual method executed is to maintain a stack of classes implementing the method currently executed. As there is a stack related to methods execution, a stack for classes is required. *Can you explain why such a solution is not adequate conceptually even if it works in practice?*

Maintaining a Stack. The method Obj class>>initializeStack initializes the shared variable ClassesImplementingLookupMethod to an empty ordered collection. This variable represents the stack. During the lookup process, when a method is found, the class which defines the method found is added at the end of this collection. And the last element of this stack is removed when a method is found, just before returning to its caller. This is illustrated in the given method unarySend: selector

Method Lookup. While implementing the method lookup, you should pay attention of adding a class when a method is found.

Modify the primitive lookup: selector for: anObjObject 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, the message #error is sent to an objInstance with aString representing the error. Note here that error should be sent to the receiver. Implement also the primitive classToLookForSuperSend that returns the objClass where the lookup should start in case of super send.